Course Code: CSM - 6215

Course Name: Data Structure using C++

MASTER OF COMPUTER APPLICATIONS (MCA)

CONTENT

Data Structure using C++ & Lab -4

BLOCK I: INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES

UNIT – 1: ANALYSIS OF ALGORITHMS

Structure

1.0 INTRODUCTION

In the realm of computer science, the efficiency and effectiveness of algorithms are paramount to solving complex problems and handling large datasets. Understanding and analyzing algorithms is essential for developing optimal solutions that perform well under various conditions. This unit delves into the fundamental aspects of algorithm analysis, providing a comprehensive overview of the mathematical tools and techniques necessary for evaluating the performance and resource requirements of algorithms.

We will begin by exploring the mathematical background needed for algorithm analysis, including key concepts such as Big O, Big Theta, and Big Omega notations, logarithms, exponential functions, and summation formulas. These foundational elements are crucial for accurately describing and comparing the efficiency of different algorithms.

Next, we will examine the process of analyzing algorithms, which involves understanding the problem statement, writing pseudocode, identifying basic operations, and establishing input sizes. This systematic approach ensures that algorithms are evaluated consistently and accurately. Additionally, we will cover the calculation of storage complexity and run time complexity, providing detailed methods for assessing an algorithm's space and time requirements. By the end of this unit, you will have a solid understanding of how to analyze and optimize algorithms for practical applications.

1.1 OBJECTIVES

After completing this unit, you will be able to understand,

- Comprehend the significance and application of Big O, Big Theta, and Big Omega notations.
- Identify the basic operations that dictate the performance of an algorithm.
- Calculate the storage complexity for various types of data structures and algorithms, including simple and recursive algorithms, as well as dynamic data structures.
- Analyze the run time complexity of algorithms, using mathematical tools to determine their efficiency.
- Recognize the trade-offs between time and space complexity in algorithm design.

1.2 MATHEMATICAL BACKGROUND

Understanding the mathematical foundations is essential for analyzing the efficiency of algorithms. This involves mastering concepts such as Big O, Big Theta, and Big Omega notations, which are used to describe the upper, exact, and lower bounds of an algorithm's complexity, respectively. These notations provide a standardized way to express the growth rates of functions, helping to compare the performance of different algorithms. Additionally, logarithms and exponential functions are crucial for understanding the behavior of algorithms that deal with exponentially growing data sets, such as those involving tree structures or divide-and-conquer strategies.

Summation formulas play a vital role in evaluating the total cost of an algorithm, particularly when dealing with loops and iterative processes. For example, understanding arithmetic and geometric series can simplify the calculation of the total number of operations in nested loops or recursive calls. Recurrence relations, on the other hand, are mathematical equations that define sequences based on previous terms, and solving these relations is key to analyzing recursive algorithms. Methods like the Master Theorem provide powerful tools to directly solve these recurrences, offering insights into the time complexity of algorithms like Merge Sort and Quick Sort.

Finally, proof techniques such as induction, contradiction, and direct proofs are indispensable for validating algorithm correctness and analyzing their behavior rigorously. Induction, for example, is often used to prove that an algorithm works for all possible input sizes, while contradiction helps in disproving incorrect assumptions about an algorithm's performance. Direct proofs and counterexamples further aid in establishing or refuting claims about the properties and efficiencies of algorithms. Together, these mathematical tools form the backbone of algorithm analysis, enabling a deeper and more precise understanding of how algorithms perform and scale.

Basic Mathematics for Algorithm Analysis

Big O, Big Theta, and Big Omega Notations: Big O, Big Theta, and Big Omega notations are mathematical tools used to describe the time and space complexity of algorithms. Big O notation (O) provides an upper bound on the growth rate of an algorithm, signifying the worst-case scenario. It helps in understanding the maximum amount of time or space an algorithm may require as the input size grows. For example, an algorithm with a time complexity of O(n^2) will have its execution time increase quadratically with the input size. Big Theta (Θ) notation, on the other hand, gives a tight bound, describing the exact asymptotic behavior of an algorithm, representing both the upper and lower bounds. Big Omega (Ω) notation provides a lower bound, representing the best-case scenario or the minimum amount of time or space required.

Example:

Big O (O) Notation Example: Consider the function f(n) = 3n^2 + 2n + 1. To find the Big O notation, we focus on the term with the highest growth rate as n increases. Here, it's n^2. Thus, f(n) is O(n^2).

Big Theta (Θ) Notation Example: If an algorithm's running time is given by f(n) = 5n log n + 4n, the dominant term is 5n log n. Therefore, the algorithm's time complexity is Θ (n log n).

Big Omega (Ω) Notation Example: For the function g(n) = 2n + 1, in the best case, the term 2n dominates. Thus, g(n) is Ω(n).

Logarithms and Exponential Functions

Logarithms and exponential functions are fundamental in analyzing the efficiency of algorithms, especially those that divide problems into smaller subproblems. Logarithmic functions, such as log(n), are prevalent in algorithms that halve their input size at each step, such as binary search. These functions grow slowly compared to polynomial or exponential functions, indicating highly efficient algorithms. Exponential functions, like 2^n, are associated with algorithms that exhibit rapid growth rates, often found in brute-force approaches or recursive algorithms solving combinatorial problems. Understanding these functions is crucial for evaluating the scalability and performance of different algorithmic approaches.

Example:

Logarithmic Function Example: The binary search algorithm repeatedly divides the search interval in half. Its time complexity is O (log n), meaning the number of comparisons grows logarithmically with the input size.

Exponential Function Example: The recursive algorithm for solving the Tower of Hanoi problem has a time complexity of $O(2^n)$. As the number of disks increases, the number of moves required grows exponentially.

Summation Formulas

Summation formulas are used to calculate the total cost of algorithms that involve iterative or repetitive operations. For

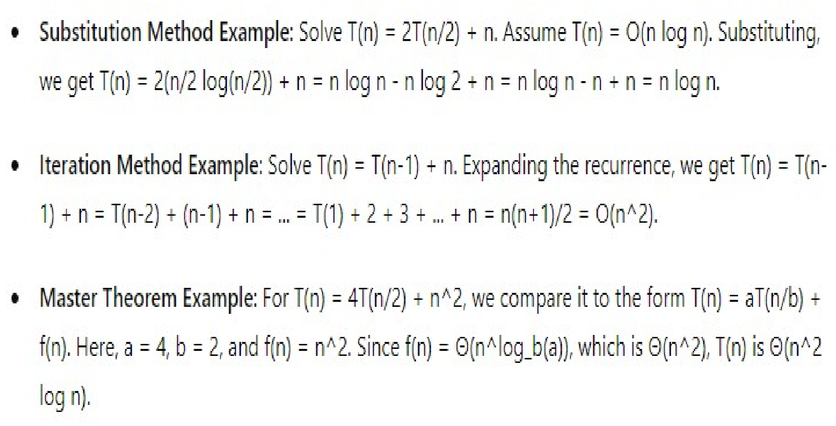
example, the sum of the first n natural numbers, given by (n(n + 1))/2, helps in analyzing loops that run linearly. Geometric series and other summation formulas are also useful in evaluating the cost of algorithms with nested loops or recursive calls. These formulas simplify the process of determining the total number of operations, providing a clear picture of an algorithm's complexity.
Example:

Sum of First n Natural Numbers: The formula for the sum of the first n natural numbers is (n (n + 1))/2. For example, if n = 10, the sum is (10 * 11)/2 = 55.

Geometric Series Example: Consider the geometric series 1 + r + r^2 + ... + r^(n-1). The sum of this series is (1 - r^n) / (1 - r) for r ≠ 1. If r = 2 and n = 4, the sum is (1 - 2^4) / (1 - 2) = 15.

Recurrence Relations and Their Solutions
Recurrence relations are equations that define sequences based on previous terms, commonly used to describe the time complexity of recursive algorithms. Solving these relations is key to understanding the behavior of algorithms like Merge Sort and Quick Sort. Techniques such as the substitution method, iteration method, and the Master Theorem are employed to solve recurrence relations. The Master Theorem, in particular, provides a straightforward way to determine the time complexity of divide-and-conquer algorithms, offering insights into their efficiency and scalability. Understanding recurrence relations and their solutions is essential for analyzing and optimizing recursive algorithms.

Example:



- **Substitution Method Example:** Solve T(n) = 2T(n/2) + n. Assume T(n) = O(n log n). Substituting, we get T(n) = 2(n/2 log(n/2)) + n = n log n - n log 2 + n = n log n - n + n = n log n.
- **Iteration Method Example:** Solve T(n) = T(n-1) + n. Expanding the recurrence, we get T(n) = T(n-1) + n = T(n-2) + (n-1) + n = ... = T(1) + 2 + 3 + ... + n = n(n+1)/2 = O(n^2).
- **Master Theorem Example:** For T(n) = 4T(n/2) + n^2, we compare it to the form T(n) = aT(n/b) + f(n). Here, a = 4, b = 2, and f(n) = n^2. Since f(n) = Θ(n^log_b(a)), which is Θ(n^2), T(n) is Θ(n^2 log n).

Proof Techniques

Induction

Mathematical Induction is a method of mathematical proof typically used to establish a given statement for all natural numbers. It consists of two steps: the base case and the inductive step.

Base Case: Prove that the statement holds for the initial value (usually n = 1).
Inductive Step: Assume the statement holds for some arbitrary natural number k, and then prove it holds for k + 1.

Example: Prove that the sum of the first n natural numbers is (n (n + 1))/2.
Base Case: For n = 1, the left side is 1 and the right side is (1(1 + 1))/2 = 1. Thus, the statement holds for n = 1.

Inductive Step: Assume the statement holds for n = k, i.e., 1 + 2 + ... + k = k(k+1)/2. We need to prove it holds for n = k + 1.

$$1 + 2 + \ldots + k + (k+1) = \frac{k(k+1)}{2} + (k+1)$$
$$= \frac{k(k+1)+2(k+1)}{2}$$
$$= \frac{(k+1)(k+2)}{2}$$

Thus, the statement holds for k + 1, completing the induction proof.

Contradiction

Proof by Contradiction involves assuming the negation of the statement to be proved and showing that this assumption leads to a contradiction, thereby proving the original statement to be true.

Example: Prove that √2 is irrational.

Assume the Opposite: Suppose √2 is rational. Then it can be expressed as a/b where a and b are integers with no common factors and $b \neq 0$.

Square Both Sides: $2 = \frac{a^2}{b^2} \Rightarrow a^2 = 2b^2$.

Parity Argument: This implies a^2 is even, so a must be even. Let a = 2k for some integer k. Substituting in, we get $(2k)^2 = 2b^2$, thus $4k^2 = 2b^2$, and $b^2 = 2k^2$. Hence, b^2 is even, and b must also be even.

Contradiction: This implies that both a and b are even, contradicting the initial assumption that a and b have no common factors. Thus, √2 is irrational.

Direct Proofs and Counterexamples

Direct Proofs involve straightforwardly showing that a statement is true using definitions, theorems, and logical deductions.

Example: Prove that the sum of two even numbers is even.

Let a and b be two even numbers. By definition of even numbers, there exist integers mmm and n such that a = 2m and b = 2.

Sum: a + b = 2m + 2n =2 (m + n).

Conclusion: Since m + n is an integer, a + b is even.

Counterexamples are used to disprove a statement by providing a specific example that shows the statement is false.

Example: Disprove the statement "All prime numbers are odd."

Counterexample: The number 2 is prime and even.

Conclusion: Therefore, the statement is false.

These proof techniques form the backbone of mathematical reasoning, providing systematic approaches to validating theorems and propositions in algorithm analysis and other areas of mathematics.

1.3 PROCESS OF ANALYSIS

The process of analyzing algorithms is a critical aspect of computer science and involves a systematic approach to understanding the efficiency and effectiveness of algorithms. This process typically includes several key steps: defining the problem, determining the computational model, designing the algorithm, and analyzing its performance.

Defining the Problem: The first step in algorithm analysis is to clearly define the problem that the algorithm aims to solve. This involves specifying the input, the desired output, and any constraints or requirements. Understanding the problem space helps in selecting or designing an appropriate algorithm and sets the stage for further analysis.

Determining the Computational Model: Next, it is essential to choose a computational model that best represents the environment in which the algorithm will run. Common models include the Random Access Machine (RAM) model, which assumes a sequential execution of instructions with uniform cost, and the Turing machine model, which is more theoretical and abstract. The choice of model affects how the algorithm's performance is measured and understood.

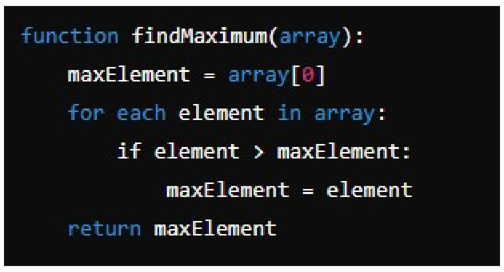
Designing the Algorithm: Once the problem and model are defined, the next step is to design the algorithm. This involves creating a step-by-step procedure to solve the problem. The design process may include selecting appropriate data structures, breaking down the problem into smaller sub-problems, and determining the logical flow of operations.

Analyzing Performance: The final and most crucial step is analyzing the performance of the algorithm. This typically involves two main aspects: time complexity and space complexity. Time complexity measures the amount of time an algorithm takes to complete as a function of the size of its input, often expressed using Big O notation. Space complexity, on the other hand, measures the amount of memory the algorithm uses. Both aspects are critical for understanding the feasibility and efficiency of the algorithm, especially for large input sizes. Additionally, average-case, best-case, and worst-case scenarios are considered to provide a comprehensive performance profile.

Steps in Analyzing an Algorithm with Examples

Understanding the Problem Statement: Consider the problem of finding the maximum element in an array of integers. The problem statement can be defined as follows: Given an array of nnn integers, find the largest integer in the array. The input is the array of integers, and the output is the maximum integer within that array.

Writing Pseudocode: Pseudocode for finding the maximum element in an array might look like this:



```
function findMaximum(array):
    maxElement = array[0]
    for each element in array:
        if element > maxElement:
            maxElement = element
    return maxElement
```

This pseudocode describes the algorithm in a clear, step-by-step manner, making it easier to understand and analyze.

Identifying Basic Operations: In this example, the basic operations include:

Initialization of maxElement with the first element of the array.

Comparison of each element in the array with maxElement.

Assignment of a new value to maxElement if a larger element is found.

These operations are fundamental to the algorithm's logic and are performed repeatedly as the algorithm processes the input array.

Establishing Input Size: The input size, n, in this problem is the number of elements in the array. If the array has 10 elements, n is 10. This input size will help us understand how the algorithm's performance scales with larger inputs.

Time Complexity Analysis: To analyze the time complexity, we count the number of basic operations performed. In the worst-case scenario, the algorithm will compare each element in the array to maxElement, resulting in $n - 1$ comparisons and n−1n-1n−1 potential assignments.

For nnn elements:

The initialization of maxElement takes O (1) time.

The for-loop iterates n times, performing a comparison and possibly an assignment each iteration, which takes O (n) time.

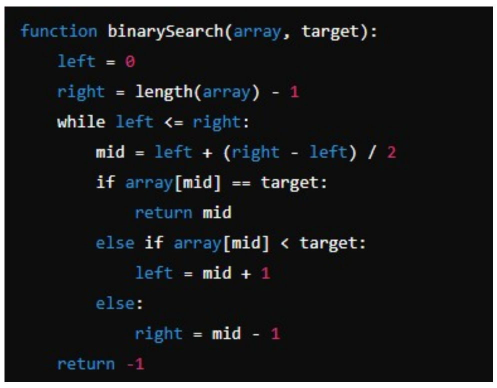
Thus, the total time complexity is O (n).

Space Complexity Analysis: The space complexity of this algorithm is O (1) because it uses a constant amount of extra space, regardless of the input size n.

Example 2: Binary Search Algorithm

Understanding the Problem Statement: Consider the problem of searching for a specific integer in a sorted array of integers using binary search. The input is a sorted array of n integers and the integer to search for, and the output is the index of the integer in the array or -1 if it is not found.

Writing Pseudocode: Pseudocode for binary search:



```
function binarySearch(array, target):
    left = 0
    right = length(array) - 1
    while left <= right:
        mid = left + (right - left) / 2
        if array[mid] == target:
            return mid
        else if array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Identifying Basic Operations: The basic operations include:

Initialization of left and right pointers.

Calculation of the middle index mid.

Comparison of the target value with the middle element of the array.

Adjusting the left or right pointers based on the comparison result.

Establishing Input Size: The input size, n, is the number of elements in the array.

Time Complexity Analysis: Binary search reduces the search space by half each iteration. The number of iterations required to search an array of size n is $\log_2$ (n). Therefore, the time complexity is O (log n).

Space Complexity Analysis: The space complexity of binary search is O(1) because it uses only a constant amount of extra space for the pointers and variables, regardless of the input size n.

These examples illustrate the steps involved in analyzing algorithms, from understanding the problem statement to determining time and space complexity, using clear and structured pseudocode.

Types of Analysis

Worst-case Analysis: This type of analysis focuses on the maximum time or space that an algorithm can take for any input of size nnn. It provides an upper bound on the running time and is particularly useful for guaranteeing performance in real-time systems or critical applications. For instance, in the case of quicksort, the worst-case occurs when the pivot selection is poor, leading to $O(n^2)$ time complexity. Knowing the worst-case performance helps in understanding the algorithm's efficiency under the least favorable conditions.

Average-case Analysis: Average-case analysis calculates the expected time or space an algorithm will take, considering all possible inputs. This type of analysis is more realistic than worst-case analysis because it provides an average performance measure, which can be more representative of typical use cases. For example, in quicksort, the average-case time complexity is O (n log n), assuming that the pivots are chosen randomly. This analysis often involves probabilistic reasoning and is useful for understanding the algorithm's performance on average inputs.

Best-case Analysis: Best-case analysis evaluates the minimum time or space an algorithm can take. It provides a lower bound on the running time and is useful for understanding the most efficient scenario. However, it is less practical for assessing an algorithm's performance in general. For instance, in insertion sort, the best-case occurs when the array is already sorted, resulting in O(n) time complexity. This analysis shows how well the algorithm performs with the most favorable input but doesn't account for average or worst-case scenarios.

Examples

Worst-case Analysis Example: Consider the insertion sort algorithm. In the worst-case scenario, the input array is in reverse order. Here, each insertion operation will have to shift all the previously sorted elements, leading to a time complexity of $O(n^2)$.

Average-case Analysis Example: For binary search, if the target element is equally likely to be at any position in a sorted array, the average-case time complexity remains O (log n). This is because each step reduces the problem size by half, and the expected number of comparisons averages out over all possible positions.

Best-case Analysis Example: For linear search, if the target element is at the first position of the array, the algorithm will only require one comparison, resulting in a best-case time complexity of O (1).

Algorithm Design Techniques

Divide and Conquer: The divide and conquer technique involves breaking a problem into smaller subproblems, solving each subproblem independently, and then combining their solutions to solve the original problem. This approach is highly effective for problems that can be divided into similar smaller problems. Classic examples include merge sort and quicksort. In merge sort, the array is recursively divided into halves until the base case of a single-element array is reached. These small arrays are then merged in a sorted manner, resulting in a sorted array. The time complexity of merge sort is O (n log n), making it efficient for large datasets.

Greedy Algorithms: Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or is locally optimal. This approach is used when a problem can be solved by making a series of choices, each of which looks the best at the moment. However, greedy algorithms do not always guarantee a globally optimal solution. A well-known example is the Kruskal's algorithm for finding the minimum spanning tree in a graph. At each step, it selects the smallest edge that does not form a cycle, ensuring that the spanning tree is built efficiently. The time complexity depends on the graph representation but is generally O (E log E), where E is the number of edges.

Dynamic Programming: Dynamic programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems that can be solved independently. DP stores the results of subproblems to avoid redundant computations. This technique is used in problems like the Fibonacci sequence, where the value of each element is the sum of the two preceding ones. Instead of recalculating Fibonacci numbers, DP stores intermediate results, reducing the time complexity from exponential to O (n). Another example is the Knapsack problem, where DP is used to find the maximum value that can be obtained without exceeding the weight limit.

Backtracking: Backtracking is an algorithmic technique for solving problems incrementally, one piece at a time, and removing those solutions that fail to satisfy the problem's constraints at any point of time. It is often used for constraint satisfaction problems, such as puzzles, crosswords, and combinatorial problems. The classic example is the N-Queens problem, where the goal is to place N queens on an N × N chessboard such that no two queens threaten each other. The algorithm tries to place a queen in a row and then recursively attempts to place queens in subsequent rows, backtracking whenever it encounters a conflict. While the worst-case time complexity is exponential, O (N!), backtracking can be very efficient with appropriate pruning.

1.4 CALCULATION OF STORAGE COMPLEXITY

Calculation of storage complexity, also known as space complexity, is a fundamental aspect of algorithm analysis that evaluates how much memory or storage space an algorithm requires to execute based on the input size. It is crucial for determining the efficiency and scalability of algorithms, particularly in scenarios where memory resources are limited or costly.

Understanding Storage Complexity

Definition: Storage complexity measures the amount of memory space required by an algorithm to solve a problem as a function of the input size n. It includes all types of memory used during execution, such as variables, data structures (arrays, lists, trees), and auxiliary space required by recursion stacks or temporary variables.

Types of Space Complexity:

Constant Space (O (1)): Algorithms that use a constant amount of memory regardless of the input size. Examples include algorithms that operate on a fixed number of variables or use a fixed-size data structure.

Linear Space (O(n)): Algorithms where the space requirement grows linearly with the size of the input. Typically, this occurs when the algorithm uses data structures whose size scales directly with n, such as arrays or linked lists.

Logarithmic Space (O (log n): Algorithms that reduce the space usage logarithmically as the input size increases. This is common in divide and conquer algorithms or algorithms that use balanced data structures like binary search trees.

Polynomial Space (O(n^k)): Algorithms where space complexity grows polynomially with the input size. These algorithms are less efficient in terms of space and can become impractical for large inputs.

Techniques for Calculating Storage Complexity

Auxiliary Space: Identify all additional space requirements beyond the input size n. This includes variables, data structures, and recursive function call stacks.

Input Size Impact: Determine how storage requirements change relative to different input sizes. Analyze worst-case, average-case, and best-case scenarios to understand the full spectrum of memory usage.

Analytical Tools: Use mathematical analysis, such as asymptotic notation (Big O notation), to express and compare the growth rate of space complexity concerning input size.

Implementation-Specific Considerations: Consider implementation details like system-specific memory allocation and overheads, especially in lower-level programming languages.

Practical Example

Consider the space complexity of a merge sort algorithm. Merge sort typically operates with a space complexity of O (n) due to its requirement to temporarily store input elements in auxiliary arrays during the merging phase. This linear space usage makes merge sort efficient in terms of memory compared to other sorting algorithms like quicksort, which may require O (log n) additional space due to recursive call stacks.

Memory Usage in Algorithms

Memory usage in algorithms revolves around managing various data types effectively to optimize space utilization. This involves understanding both primitive and composite data types, which are crucial for storing and manipulating data efficiently during algorithm execution.

Primitive Data Types

Primitive data types are fundamental building blocks in programming languages that represent basic values. These include:

Integer: Represents whole numbers (e.g., int in C++, Java).

Floating Point: Represents decimal numbers with fractional parts (e.g., float, double).

Boolean: Represents true/false values (e.g., bool).

Character: Represents single characters (e.g., char).

These data types typically have fixed sizes depending on the programming language and system architecture. For instance, an int might be 4 bytes in size in many programming languages.

Composite Data Types

Composite data types combine primitive data types to create more complex structures for storing and organizing data. Key examples include:

Arrays: A collection of elements stored in contiguous memory locations, accessed by indexing.

Lists: Linear data structures where elements are linked by pointers or references.

Trees: Hierarchical structures composed of nodes, with each node having references to child nodes.

Graphs: Non-linear data structures with nodes (vertices) and edges connecting these nodes.

Memory Usage Considerations

Arrays: Use contiguous memory, making them efficient for direct access via indexing but limiting in dynamic resizing.

Lists: Linked lists dynamically allocate memory per element, allowing flexibility in size but incurring overhead due to pointers.

Trees: Memory usage varies based on the type (e.g., binary trees, AVL trees). Trees balance between efficient storage and retrieval operations.

Graphs: Storage varies based on the representation (e.g., adjacency list, adjacency matrix). Each representation offers trade-offs between space and operations efficiency.

Efficient Memory Management

Efficient memory management in algorithms involves:

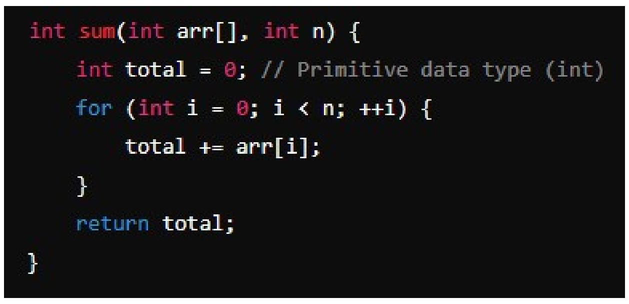
Optimal Data Structures: Choosing the right data structure based on the operations required and memory constraints.

Memory Allocation: Using appropriate allocation techniques (e.g., static vs. dynamic allocation) to minimize wastage and fragmentation.

Garbage Collection: In languages with automatic memory management, ensuring timely release of unused memory.

Example Scenario

Consider an algorithm that computes the sum of elements in an array:



```cpp
int sum(int arr[], int n) {
    int total = 0; // Primitive data type (int)
    for (int i = 0; i < n; ++i) {
        total += arr[i];
    }
    return total;
}
```

In this example:

Primitive: int total is used to accumulate the sum.

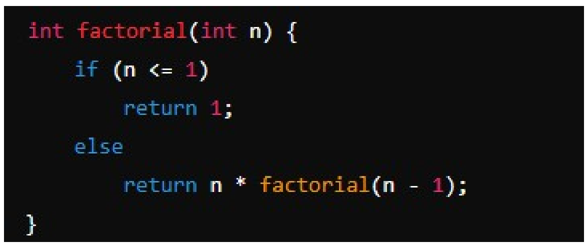
Composite: int arr[] represents an array storing multiple integers.

Examples of Storage Complexity Calculation

Calculating storage complexity involves understanding how much memory an algorithm or data structure requires based on its operations and data handling. Here are examples of how storage complexity is calculated for different scenarios:

Calculating Storage for Simple Algorithms

Consider a simple algorithm that computes the factorial of a number n:



```cpp
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Storage Calculation:

Primitive Data Types: The function uses int for the parameter and the return value.

Space for int n: Typically 4 bytes (assuming a 32-bit integer).

Space for the return value (int): 4 bytes.

Recursive Call Stack: Each recursive call adds to the stack memory.

For factorial(n), there will be n recursive calls.

Assuming each call uses 8 bytes for function call overhead and local variables (on a typical 64-bit system).

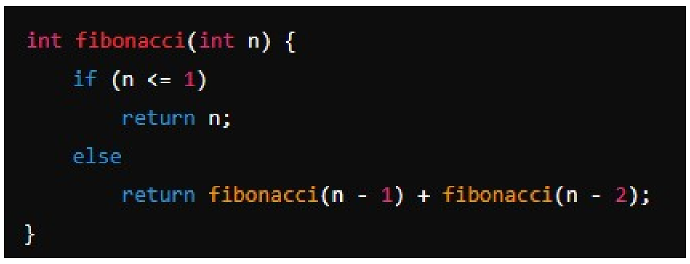
Total Storage:

Fixed Memory: Around 8 bytes for n and the return value.

Stack Memory: Approximately 8 * n bytes for the recursive calls.

Calculating Storage for Recursive Algorithms

Consider the Fibonacci sequence computed recursively:



```cpp
int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Storage Calculation:

Primitive Data Types: Uses int for the parameter and the return value.

Space for int n: 4 bytes.

Space for the return value (int): 4 bytes.

Recursive Call Stack: Similar to factorial, Fibonacci also has n recursive calls.

Each call uses 8 bytes for function call overhead and local variables.

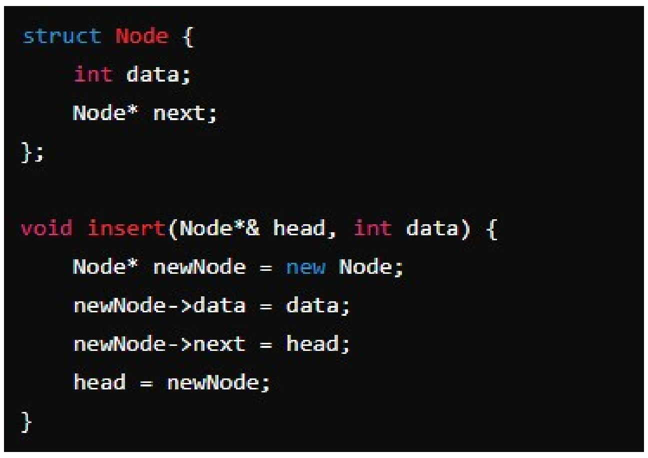
Total Storage:

Fixed Memory: Around 8 bytes for n and the return value.

Stack Memory: Approximately 8 * n bytes for the recursive calls.

Storage Complexity in Dynamic Data Structures

Consider a dynamic data structure like a linked list with n nodes:



```
struct Node {
    int data;
    Node* next;
};

void insert(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}
```

Storage Calculation:

Node Structure: Each Node structure contains an int and a pointer (Node*).

Size of int data: 4 bytes.

Size of Node* next: 8 bytes (assuming a 64-bit system).

Heap Memory: Each new Node allocates memory dynamically.

Total memory depends on the number of nodes (n).

Total Storage:

Fixed Memory: Minimal fixed memory for variables like head and function parameters.

Heap Memory: Approximately (4 + 8) * n bytes for data and next pointers across n nodes.

1.5 CALCULATION OF RUN TIME COMPLEXITY

Calculating the runtime complexity of an algorithm involves analyzing how its execution time increases with respect to the input size. This analysis is crucial for understanding the efficiency of algorithms and making informed decisions about their applicability in different scenarios. Here's how the calculation of runtime complexity is typically approached:

Steps in Calculating Runtime Complexity

Identify Basic Operations:

Determine the fundamental operations that contribute most significantly to the execution time. For example, in sorting algorithms, comparisons and swaps are often primary operations.

Establish Input Size:

Define the parameter that represents the size of the input data. For sorting algorithms, this could be the number of elements n.

Count Operations:

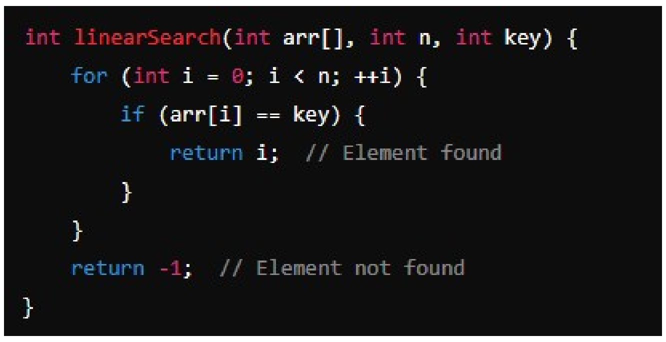
Analyze how many times the identified basic operations are executed as a function of the input size n. This step often involves considering different cases: best-case, average-case, and worst-case scenarios.

Express Complexity:

Use Big O notation to express the asymptotic upper bound of the algorithm's runtime complexity in terms of n. This notation provides a concise way to describe how the algorithm's performance scales with input size.

Examples of Calculating Runtime Complexity

Example 1: Linear Search



```cpp
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == key) {
            return i;  // Element found
        }
    }
    return -1;  // Element not found
}
```

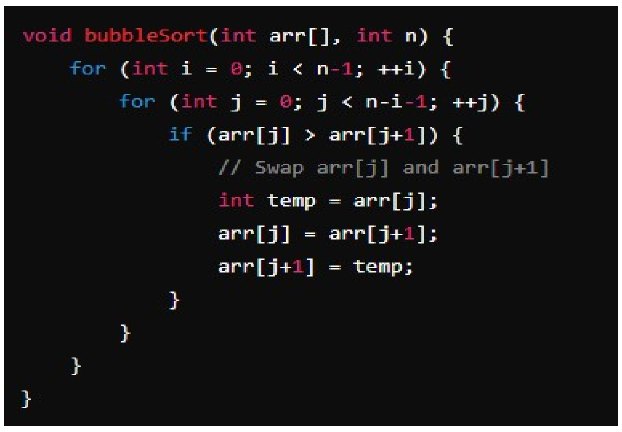
Basic Operation: Comparison (arr[i] == key).

Input Size: n, where arr is an array of size n.

Operations Count: In the worst case, the loop executes n times.

Runtime Complexity: O(n), as the algorithm checks each element in the array once in the worst case.

Example 2: Bubble Sort



```cpp
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < n-i-1; ++j) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Basic Operations: Comparisons (arr[j] > arr[j+1]) and Swaps.
Input Size: n, where arr is an array of size n.

Operations Count: In the worst case, bubble sort performs n-1 passes over the array, with n-i-1 comparisons in the i-th pass.

Runtime Complexity: O(n^2), as the algorithm performs quadratic time operations in the worst case due to nested loops.

Importance of Runtime Complexity Calculation
Understanding runtime complexity helps in:

Algorithm Selection: Choosing the most efficient algorithm for a given problem size.

Performance Prediction: Estimating how an algorithm will perform as the input size grows.

Optimization: Identifying opportunities for improving algorithm efficiency through algorithmic design or data structure selection.

1.6 CONCLUSION

In this unit, we delved into the foundational aspects of algorithm analysis, emphasizing the importance of understanding mathematical concepts such as Big O, Big Theta, and Big Omega notations. These notations are critical tools for describing the efficiency of algorithms and predicting their behavior as input sizes grow. We also explored logarithms, exponential functions, summation formulas, and recurrence relations, which are essential for analyzing and solving problems related to algorithm performance.

We examined the process of analyzing algorithms, starting with a clear understanding of the problem statement and progressing through writing pseudocode, identifying basic operations, and establishing input sizes. This systematic approach helps in accurately assessing an algorithm's efficiency and potential bottlenecks. Furthermore, we discussed the different types of analysis—worst-case, average-case, and best-case scenarios—highlighting their significance in practical applications.

Lastly, we covered the calculation of storage and run time complexity, which are crucial for evaluating an algorithm's resource requirements. By understanding these concepts, we can make informed decisions about algorithm design, balancing the trade-offs between time and space efficiency. This unit has equipped you with the essential tools and knowledge to analyze algorithms methodically, optimize their performance, and apply these principles to solve real-world problems effectively.

1.7 QUESTIONS AND ANSWERS

1. What is the significance of Big O notation in algorithm analysis?

Answer: Big O notation is crucial in algorithm analysis because it provides a high-level understanding of an algorithm's efficiency in terms of time and space complexity. It describes the upper bound of an algorithm's running time, helping to predict its performance and scalability as the input size increases. This notation allows for the comparison of different algorithms' efficiencies, facilitating the selection of the most suitable algorithm for a given problem.

2. Explain the difference between worst-case, average-case, and best-case analyses.

Answer: Worst-case analysis evaluates an algorithm's performance under the most unfavorable conditions, providing an upper bound on its running time. Average-case analysis considers the algorithm's performance across all possible inputs, giving a more realistic expectation of its efficiency. Best-case analysis examines the algorithm's performance under the most favorable conditions, offering a lower bound on its running time. Each type of analysis provides different insights into the algorithm's behavior and helps in understanding its efficiency comprehensively.

3. What are recurrence relations and why are they important in algorithm analysis?

Answer: Recurrence relations are equations that define a sequence of values based on previous terms. They are essential in algorithm analysis for expressing the running time of recursive algorithms. By solving these relations, we can determine the time complexity of the algorithm. This is particularly useful for divide-and-conquer algorithms, where the problem is broken down into smaller subproblems, and the running time depends on the solutions of these subproblems.

4. How do you calculate the storage complexity of an algorithm?

Answer: The storage complexity of an algorithm is calculated by analyzing the amount of memory it requires during execution. This involves considering the memory used by variables, data structures, and any additional space needed for recursion or dynamic allocation. For simple algorithms, this can be straightforward, but for more complex algorithms involving dynamic data structures or recursion, a detailed breakdown of memory usage is necessary to determine the total storage complexity.

5. Why is it important to establish the input size when analyzing an algorithm?

Answer: Establishing the input size is crucial because it directly influences the algorithm's running time and space requirements. The efficiency of an algorithm is often expressed as a function of the input size, allowing us to understand how the algorithm scales with larger inputs. Accurate input size estimation ensures that the analysis reflects real-world performance and helps in identifying potential inefficiencies and bottlenecks.

6. What are the basic steps in the process of analyzing an algorithm?

Answer: The basic steps in analyzing an algorithm include:

Understanding the Problem Statement: Clearly define the problem and its requirements.

Writing Pseudocode: Develop a high-level representation of the algorithm to understand its flow and logic.

Identifying Basic Operations: Determine the fundamental operations that significantly impact the running time.

Establishing Input Size: Define the variable representing the size of the input, which will be used in the complexity analysis.

Analyzing Complexity: Calculate the time and space complexity based on the identified operations and input size.

1.8 REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in C++* (4th ed.). Addison-Wesley.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2006). *Algorithms*. McGraw-Hill Education.

UNIT – 2: ARRAYS AND POINTERS IN C++

Structure

2.0 INTRODUCTION

In the realm of computer science and programming, understanding fundamental data structures like arrays and pointers forms the bedrock of efficient algorithm design and application development. These concepts not only facilitate storage and manipulation of data but also play crucial roles in optimizing memory usage and enhancing computational efficiency. This unit delves into these foundational concepts, exploring their definitions, operations, representations, and practical applications in various domains.

Arrays, as a cornerstone of data structures, provide a systematic way to store homogeneous elements in contiguous memory locations. They offer quick access to elements using indices and

support a wide range of operations, making them versatile for applications ranging from simple list storage to complex numerical computations. Pointers, on the other hand, enhance the flexibility of memory management by allowing dynamic memory allocation and manipulation of addresses, enabling efficient data structures like linked lists and trees.
Sparse matrices and polynomials extend the concept of arrays into specialized domains. Sparse matrices, characterized by a majority of zero elements, employ efficient representation techniques such as triplet and compressed formats to conserve memory and accelerate operations like addition and multiplication. Polynomials, represented using arrays or linked lists, demonstrate how basic data structures can be adapted for mathematical computations, showcasing operations like addition and multiplication that are pivotal in scientific computing and engineering applications.

Throughout this unit, we explore not only the theoretical underpinnings of these data structures but also their real-world applications. Understanding their representations in memory and their computational advantages and disadvantages equips us with the knowledge to leverage arrays, pointers, sparse matrices, and polynomials effectively in solving practical problems across diverse fields.

2.1 OBJECTIVES

After completing this unit, you will be able to understand,
Learn how to declare, initialize, and access elements in arrays.
Understand the concept of multidimensional arrays and their practical uses.
Grasp the basics of pointers, including declaration, initialization, and dereferencing.

Discover how arrays and pointers are used in fundamental data structures like linked lists, stacks, and queues.
Develop problem-solving abilities by applying arrays and pointers to solve programming challenges.

2.2 ARRAYS

An array in C++ is a structured data type that stores a fixed-size sequential collection of elements of the same type. It provides a contiguous memory location to store multiple values under a single name, allowing efficient access to each element using an index. Arrays are declared by specifying the data type of the elements they will hold and the number of elements, which must be known at compile time. Elements in an array are accessed using zero-based indexing, where the first element is at index 0 and the last element is at index size - 1. Arrays facilitate efficient storage and retrieval of data, making them essential for tasks that involve managing and manipulating collections of homogeneous data elements in C++ programs.

Declaration of Arrays:
In C++, an array is declared by specifying the data type of its elements followed by the array name and the size of the array enclosed in square brackets ([]). The syntax for declaring an array is:

datatype arrayName[arraySize];
Here, datatype specifies the type of elements the array will hold (e.g., int, double, char), arrayName is the identifier used to refer to the array, and arraySize is the number of elements in the array. The size of the array must be a constant expression or a literal value known at compile time.

For example, to declare an array of integers named myArray with 5 elements:

int myArray[5];

This declaration reserves space in memory to store 5 integers contiguously.

Initialization of Arrays:

Arrays in C++ can be initialized at the time of declaration or later during the program execution. Initialization assigns initial values to the elements of the array. There are several ways to initialize arrays:

Initialization at Declaration:

int myArray[5] = {1, 2, 3, 4, 5};

This initializes an array myArray with 5 elements {1, 2, 3, 4, 5}.

Partial Initialization:

int myArray[5] = {1, 2, 3};

This initializes the first three elements of myArray as {1, 2, 3} and sets the remaining elements to zero (0 for numeric types).

Empty Initialization:

int myArray[5] = {};

This initializes all elements of myArray to zero (0).

Initialization using Iteration:

int myArray[5];

for (int i = 0; i < 5; ++i)

{myArray[i] = i + 1; }

This initializes myArray with values {1, 2, 3, 4, 5} using a loop.

Accessing Elements of Arrays:

In C++, elements of an array are accessed using zero-based indexing. Once an array is declared and initialized, you can access individual elements by specifying the index within square brackets ([]). The syntax is:

arrayName[index]

Here, arrayName is the name of the array, and index is the position of the element you want to access. Indexing starts from 0 for the first element and goes up to arraySize - 1 for the last element.

For example, consider an array of integers myArray:

int myArray[5] = {10, 20, 30, 40, 50};

To access elements of myArray:

int firstElement = myArray[0]; // Accessing the first element (10)

int thirdElement = myArray[2]; // Accessing the third element (30)

You can also modify array elements using the same indexing syntax:

myArray[1] = 25; // Changing the value of the second element to 25

Multidimensional Arrays:

A multidimensional array in C++ is an array that contains more than one dimension, allowing data to be stored in a tabular form. The most common multidimensional array is the 2D array, but C++ supports arrays with more dimensions as well.

Declaration of Multidimensional Arrays:

To declare a multidimensional array, you specify the data type of its elements followed by the array name and the sizes of each dimension enclosed in square brackets ([]). The syntax for a 2D array is:

datatype arrayName[rowSize][colSize];

Here, rowSize specifies the number of rows, and colSize specifies the number of columns.

For example, a 2D array matrix with 3 rows and 4 columns of integers can be declared as:

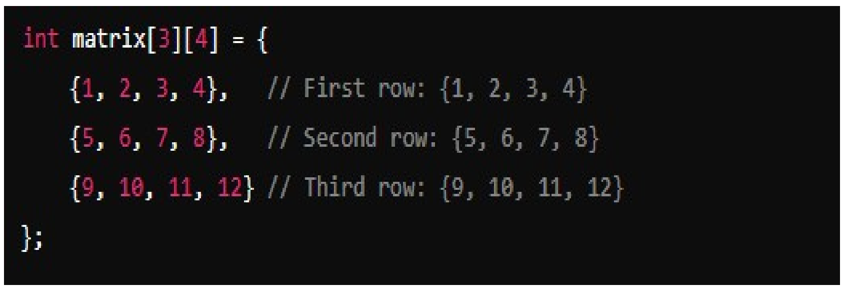
cpp

Copy code

int matrix[3][4];

Initialization of Multidimensional Arrays:

Multidimensional arrays can be initialized similarly to 1D arrays, using nested braces {} to enclose the elements:



```
int matrix[3][4] = {
    {1, 2, 3, 4},   // First row: {1, 2, 3, 4}
    {5, 6, 7, 8},   // Second row: {5, 6, 7, 8}
    {9, 10, 11, 12} // Third row: {9, 10, 11, 12}
};
```

Accessing Elements of Multidimensional Arrays:

Elements in a 2D array are accessed using two indices: one for the row and one for the column:

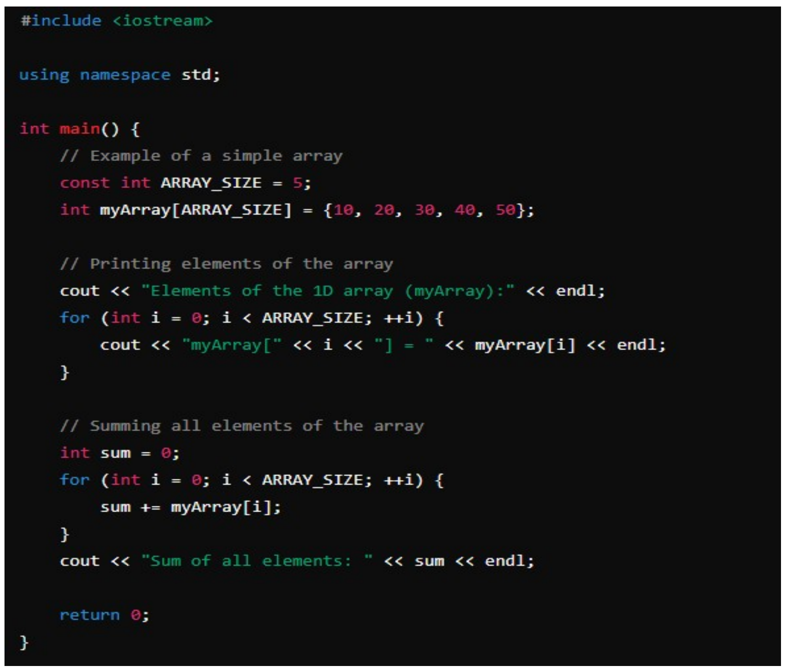
int element = matrix[rowIndex][colIndex];

Here, rowIndex specifies the row number (starting from 0), and colIndex specifies the column number (also starting from 0).

For example, to access the element at the second row and third column of matrix:

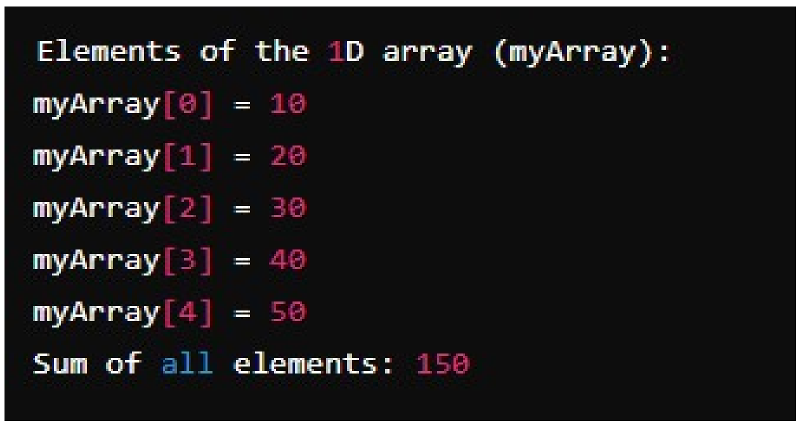
int value = matrix [1][2]; // Accessing element at second row, third column (value 7)

Example 1: Simple Array Operations



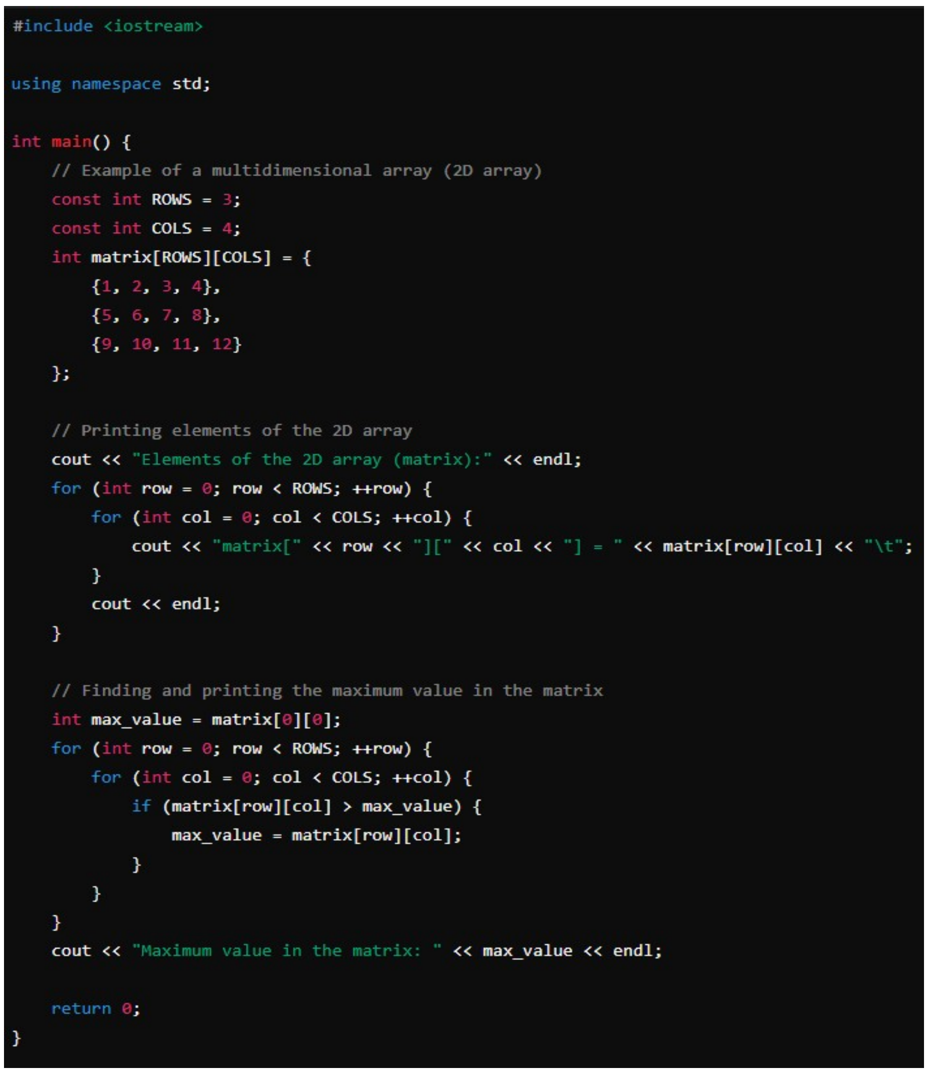
```
#include <iostream>

using namespace std;

int main() {
    // Example of a simple array
    const int ARRAY_SIZE = 5;
    int myArray[ARRAY_SIZE] = {10, 20, 30, 40, 50};

    // Printing elements of the array
    cout << "Elements of the 1D array (myArray):" << endl;
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        cout << "myArray[" << i << "] = " << myArray[i] << endl;
    }

    // Summing all elements of the array
    int sum = 0;
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        sum += myArray[i];
    }
    cout << "Sum of all elements: " << sum << endl;

    return 0;
}
```

Output:



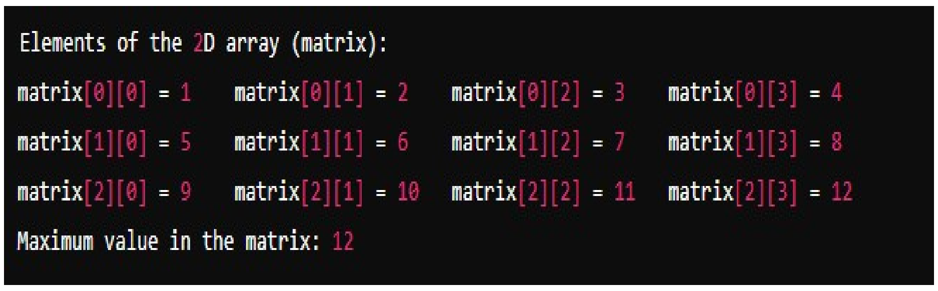
```
Elements of the 1D array (myArray):
myArray[0] = 10
myArray[1] = 20
myArray[2] = 30
myArray[3] = 40
myArray[4] = 50
Sum of all elements: 150
```

Example 2: Multidimensional Array Operations



```cpp
#include <iostream>

using namespace std;

int main() {
    // Example of a multidimensional array (2D array)
    const int ROWS = 3;
    const int COLS = 4;
    int matrix[ROWS][COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Printing elements of the 2D array
    cout << "Elements of the 2D array (matrix):" << endl;
    for (int row = 0; row < ROWS; ++row) {
        for (int col = 0; col < COLS; ++col) {
            cout << "matrix[" << row << "][" << col << "] = " << matrix[row][col] << "\t";
        }
        cout << endl;
    }

    // Finding and printing the maximum value in the matrix
    int max_value = matrix[0][0];
    for (int row = 0; row < ROWS; ++row) {
        for (int col = 0; col < COLS; ++col) {
            if (matrix[row][col] > max_value) {
                max_value = matrix[row][col];
            }
        }
    }
    cout << "Maximum value in the matrix: " << max_value << endl;

    return 0;
}
```

Output:



```
Elements of the 2D array (matrix):
matrix[0][0] = 1    matrix[0][1] = 2    matrix[0][2] = 3    matrix[0][3] = 4
matrix[1][0] = 5    matrix[1][1] = 6    matrix[1][2] = 7    matrix[1][3] = 8
matrix[2][0] = 9    matrix[2][1] = 10   matrix[2][2] = 11   matrix[2][3] = 12
Maximum value in the matrix: 12
```

2.3 POINTERS

Pointers in C++ are variables that store memory addresses rather than values directly. They provide a way to directly access and manipulate memory locations, enabling efficient dynamic memory allocation and management.

In C++, every variable is stored in a specific memory location with a unique address. Pointers allow us to store and manipulate these addresses as values. They are declared using the asterisk (*) symbol before the variable name, indicating that the variable is a pointer. For example, int* ptr; declares a pointer ptr that can hold the address of an integer variable.

One of the fundamental operations with pointers is dereferencing, which is done using the asterisk (*) operator. Dereferencing a pointer retrieves the value stored at the memory address it points to. For instance, if ptr points to an integer variable num, *ptr accesses the value of num. This capability makes pointers powerful for indirect access to data, especially useful in data structures and dynamic memory allocation scenarios where memory addresses are manipulated directly.

Pointers are extensively used in C++ for tasks like passing parameters to functions by reference, dynamic memory allocation with new and delete operators, and implementing complex data structures such as linked lists and trees. While powerful, pointers require careful handling to avoid common pitfalls like dereferencing null pointers or accessing memory out of bounds, which can lead to runtime errors like segmentation faults. Mastery of pointers is essential for C++ programmers to fully utilize the language's capabilities for memory management and efficient data manipulation.

Declaration of Pointers:

In C++, pointers are declared using the asterisk (*) symbol before the pointer variable name. The syntax for declaring a pointer to a specific data type is:

datatype *pointerName;

Here, datatype specifies the type of data that the pointer will point to (e.g., int, double, char), and pointerName is the name of the pointer variable.

For example, to declare a pointer to an integer (int):

int *ptr; // Declares a pointer to an integer

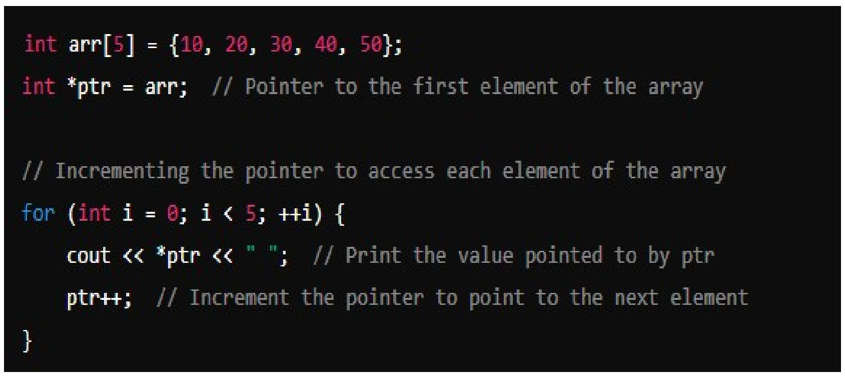
The pointer ptr can now hold the memory address of an integer variable.

Pointer Arithmetic Operations:

Pointer arithmetic allows you to perform arithmetic operations on pointers to manipulate memory addresses. This is particularly useful when iterating through arrays or dynamically allocated memory blocks.

Incrementing Pointers:

Incrementing a pointer moves it to point to the next memory location of its data type. The increment operation depends on the size of the data type the pointer is pointing to.

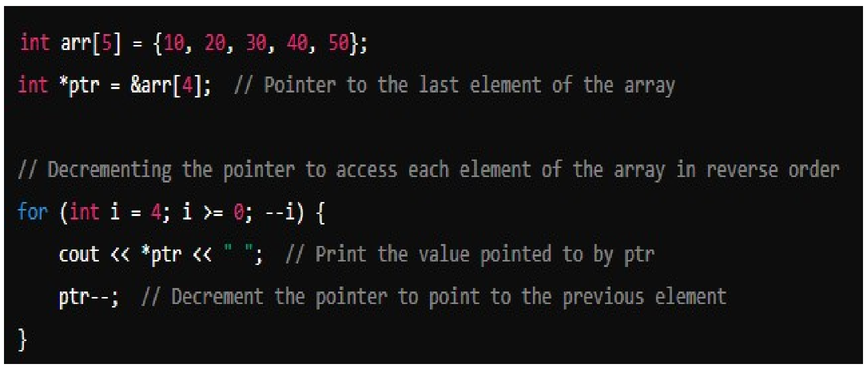


```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;  // Pointer to the first element of the array

// Incrementing the pointer to access each element of the array
for (int i = 0; i < 5; ++i) {
    cout << *ptr << " ";  // Print the value pointed to by ptr
    ptr++;  // Increment the pointer to point to the next element
}
```

In this example, ptr++ increments the pointer ptr to point to the next integer in the array arr.

Decrementing Pointers:

Decrementing a pointer moves it to point to the previous memory location of its data type.



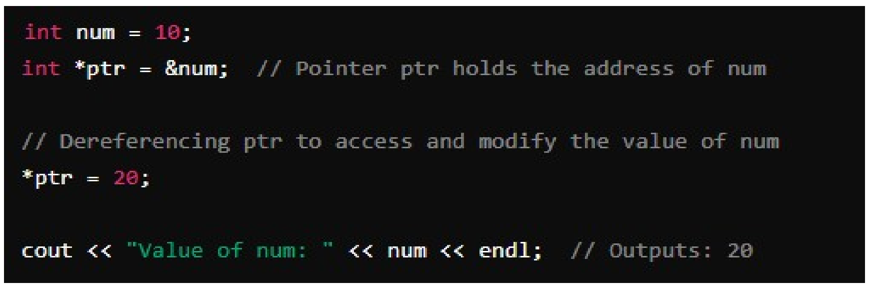
```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = &arr[4];  // Pointer to the last element of the array

// Decrementing the pointer to access each element of the array in reverse order
for (int i = 4; i >= 0; --i) {
    cout << *ptr << " ";  // Print the value pointed to by ptr
    ptr--;  // Decrement the pointer to point to the previous element
}
```

Here, ptr-- decrements the pointer ptr to point to the previous integer in the array arr.

Pointer arithmetic also allows addition and subtraction of integers to/from pointers, which moves the pointer by a certain number of elements, scaled by the size of the data type it points to. Care must be taken with pointer arithmetic to ensure that pointers remain within valid memory bounds to avoid undefined behavior.

Pointer Indirection (Dereferencing):

Pointer indirection, also known as dereferencing, refers to the process of accessing the value stored at the memory address held by a pointer. It is denoted by the asterisk (*) operator placed before the pointer variable name. Dereferencing a pointer allows you to manipulate the data stored in the memory location pointed to by the pointer.



```
int num = 10;
int *ptr = &num;  // Pointer ptr holds the address of num

// Dereferencing ptr to access and modify the value of num
*ptr = 20;

cout << "Value of num: " << num << endl;  // Outputs: 20
```

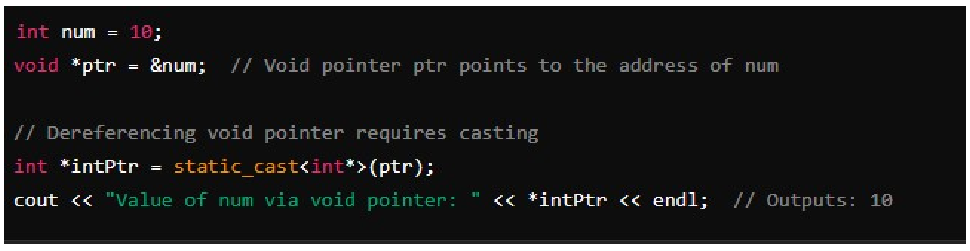
In this example, *ptr = 20; assigns the value 20 to the memory location pointed to by ptr, effectively updating the value of num.

Null Pointers and Void Pointers:

Null Pointers: A null pointer is a pointer that does not point to any memory location. It is initialized explicitly to a null value (nullptr) or implicitly when not initialized at all. Null pointers are often used to indicate that a pointer does not currently point to a valid object or memory location.

int *ptr = nullptr; // Initializing ptr as a null pointer

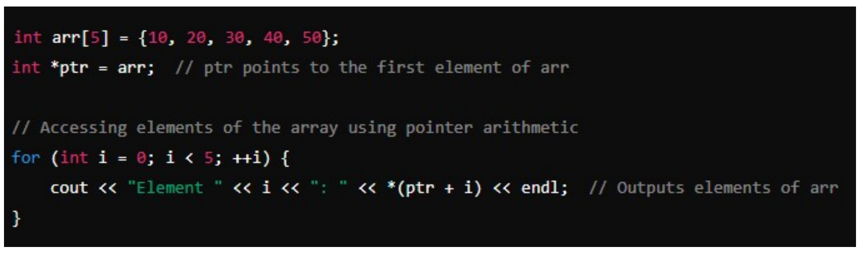
Void Pointers: A void pointer (or void*) is a special type of pointer that can point to objects of any data type. It is used when the specific type of data pointed to is not known at compile time or when dealing with functions that accept pointers to any type. However, you cannot directly dereference a void pointer without first casting it to a specific pointer type.



```
int num = 10;
void *ptr = &num;  // Void pointer ptr points to the address of num

// Dereferencing void pointer requires casting
int *intPtr = static_cast<int*>(ptr);
cout << "Value of num via void pointer: " << *intPtr << endl;  // Outputs: 10
```

Pointers and Arrays (Relationship between Pointers and Arrays):

In C++, arrays and pointers are closely related concepts due to the way arrays are implemented. An array name can be used as a pointer to its first element. When an array name is used in an expression, it is automatically converted to a pointer to the first element of the array.



```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;  // ptr points to the first element of arr

// Accessing elements of the array using pointer arithmetic
for (int i = 0; i < 5; ++i) {
    cout << "Element " << i << ": " << *(ptr + i) << endl;  // Outputs elements of arr
}
```

In this example, ptr is initialized to point to arr[0], the first element of the array arr. Using pointer arithmetic (ptr + i), you can access successive elements of the array. Thus, arrays and pointers are interchangeable in many contexts, making pointers an essential tool for efficiently manipulating arrays in C++.

2.4 SPARSE MATRICES

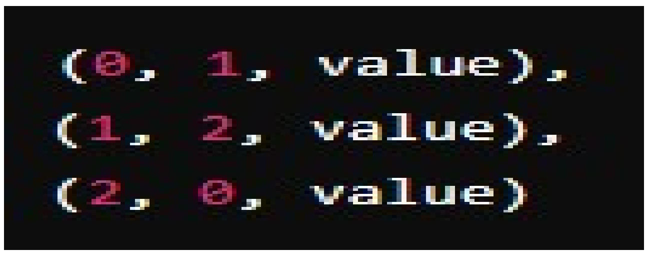
Sparse matrices are matrices where the majority of elements are zero. In contrast, dense matrices have mostly non-zero elements. The sparsity of a matrix refers to the proportion of zero elements to the total number of elements. Sparse matrices are commonly encountered in various fields, including scientific computing, data mining, and machine learning, where they help optimize storage and computation.

Representation Techniques:

Triplet Representation (COO - Coordinate Format):

In this representation, each non-zero element is stored with its row and column indices and its value.

Example: If a matrix M has non-zero elements at (0, 1), (1, 2), and (2, 0), it would be represented as:



```
(0, 1, value),
(1, 2, value),
(2, 0, value)
```

Pros: Simple and easy to understand. Suitable for matrices with irregular non-zero patterns.

Cons: Requires additional space for storing row and column indices.

Compressed Sparse Row (CSR) Format:

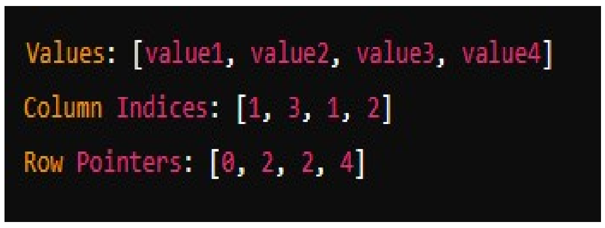
In CSR format, the matrix is represented using three arrays:

Values array: Contains non-zero elements of the matrix in row-major order.

Column indices array: Stores the column indices corresponding to each non-zero element in the values array.

Row pointers array: Indicates the start index in the values array for each row.

Example: For a matrix with rows [0, 0, 2, 3] and column indices [1, 3, 1, 2], CSR format would be:



```
Values: [value1, value2, value3, value4]
Column Indices: [1, 3, 1, 2]
Row Pointers: [0, 2, 2, 4]
```

Pros: Efficient for row-wise operations like addition and multiplication.

Cons: More complex to construct and maintain compared to COO format.

Operations on Sparse Matrices:

Addition:

Add two sparse matrices by adding corresponding non-zero elements.

Example: Adding two sparse matrices A and B involves adding elements at corresponding positions where both matrices have non-zero elements.

Multiplication:

Multiply two sparse matrices using appropriate algorithms such as the traditional algorithm or the Strassen algorithm.

Example: Multiplying two sparse matrices A and B involves multiplying rows of A with columns of B, taking into account zero elements to optimize computation.

2.5 POLYNOMIALS

Polynomials are mathematical expressions consisting of variables and coefficients raised to non-negative integer powers. They can be represented using arrays or linked lists, with each element storing a coefficient and an exponent.

Array Representation:

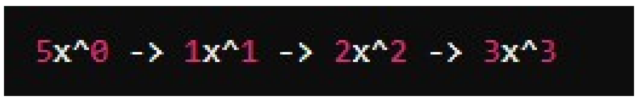
In this representation, an array stores coefficient where each index corresponds to the exponent of the variable.

Example: The polynomial $3x^3 + 2x^2 + x + 5$ can be represented as an array [5, 1, 2, 3], where index 0 corresponds to the constant term, index 1 to the linear term, and so on.

Linked List Representation:

Using a linked list, each node contains a coefficient and an exponent.

Example: The polynomial $3x^3 + 2x^2 + x + 5$ can be represented as a linked list:



```
5x^0 -> 1x^1 -> 2x^2 -> 3x^3
```

Operations on Polynomials:

Addition:

Add two polynomials by combining like terms (terms with the same exponent).

Example: Adding (3x2+2x+1)(3x^2 + 2x + 1)(3x2+2x+1) and (4x2+3x−2)(4x^2 + 3x - 2)(4x2+3x−2) results in 7x2+5x−17x^2 + 5x - 17x2+5x−1.

Multiplication:

Multiply two polynomials using distributive property and combining like terms.

Example: Multiplying (3x+2)(3x + 2)(3x+2) and (4x−1)(4x - 1)(4x−1) results in 12x2+5x−212x^2 + 5x - 212x2+5x−2.

Applications of Polynomials:

Polynomials find applications in various computational problems, including:

Curve Fitting and Interpolation: Polynomials are used to approximate and fit curves to data points, facilitating trend analysis and predictive modeling in fields like statistics and engineering.

Signal Processing: In digital signal processing, polynomials are used to model and manipulate signals for filtering, noise reduction, and compression.

Numerical Methods: Polynomial interpolation and approximation are fundamental in numerical analysis for solving differential equations, optimization problems, and root finding algorithms.

Computer Graphics: Polynomials are used extensively in computer graphics to represent curves and surfaces, enabling realistic rendering and animation in applications such as gaming and simulation.

Error Detection and Correction: Error-correcting codes and algorithms in communication systems rely on polynomials for encoding and decoding information, ensuring reliable data transmission.

2.6 REPRESENTATION OF ARRAYS

Arrays are fundamental data structures that store elements of the same data type in contiguous memory locations. How elements are stored in memory can significantly impact access patterns and performance, especially in large datasets.

Row-Major Representation:

Definition: In row-major representation, elements of a multidimensional array are stored row by row in memory.

Memory Layout: If you have a 2D array A[m][n], the elements are stored sequentially such that all elements of row 0 are followed by all elements of row 1, and so forth.

Access Pattern: Accessing elements is optimized for row-wise traversal. For example, accessing A[i][j] is efficient because the next element A[i][j+1] is adjacent in memory.

Column-Major Representation:

Definition: In column-major representation, elements of a multidimensional array are stored column by column in memory.

Memory Layout: Similar to row-major but stored column-wise. Elements of column 0 are followed by elements of column 1, and so on.

Access Pattern: Accessing elements is optimized for column-wise traversal. For example, accessing A[i][j] is efficient because the next element A[i+1][j] is adjacent in memory.

Differences between Row-Major and Column-Major Order:

Memory Storage Order:

Row-Major: Elements of each row are stored contiguously in memory.

Column-Major: Elements of each column are stored contiguously in memory.

Traversal Efficiency:

Row-Major: Optimized for row-wise traversal due to contiguous memory access.

Column-Major: Optimized for column-wise traversal for the same reason.

Access Patterns:

Row-Major: Accessing adjacent elements within the same row is efficient.

Column-Major: Accessing adjacent elements within the same column is efficient.

Advantages and Disadvantages:

Row-Major:

Advantages:

Efficient for row-wise operations such as matrix addition, subtraction, and multiplication.

Suitable for applications where row-oriented access patterns dominate, such as image processing and linear algebra operations.

Disadvantages:

Less efficient for column-wise operations, which may result in cache misses and reduced performance.

Not optimal for applications requiring frequent column-oriented data access.

Column-Major:

Advantages:

Efficient for column-oriented operations like transposition and certain types of matrix manipulations.

Suitable for applications where column-wise access patterns are prevalent, such as database queries and statistical analysis.

Disadvantages:

May lead to inefficiencies in row-wise access, especially in algorithms that heavily depend on sequential row access.

Limited utility in applications that primarily utilize row-wise data manipulation.

2.7 APPLICATIONS OF ARRAYS AND POINTERS

Arrays and pointers are fundamental concepts in C++ programming with diverse applications across various domains. Here are some common applications where arrays and pointers play a crucial role:

Data Structures: Arrays are the building blocks for implementing fundamental data structures such as lists, stacks, queues, and hash tables. For instance, dynamic arrays (using pointers) allow resizing based on runtime needs, making them versatile for data storage and manipulation.

Dynamic Memory Allocation: Pointers are essential for dynamic memory allocation using operators like new and delete. This capability is crucial when the size of data is not known at compile time or when memory needs to be managed dynamically during program execution.

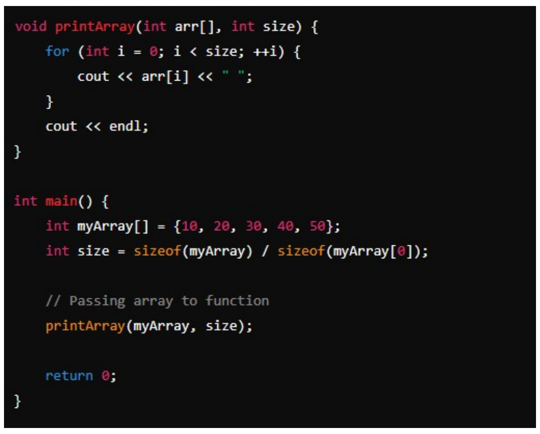
String Manipulation: In C++, strings are often represented as arrays of characters (char[]). Pointers to characters (char*) are extensively used to manipulate and access individual characters within strings, allowing for efficient string operations like concatenation, comparison, and parsing.

Function Parameters: Pointers are commonly used to pass parameters by reference to functions. This allows functions to modify variables outside their scope directly, facilitating efficient parameter passing and avoiding unnecessary copying of large data structures.

Multidimensional Arrays: Arrays of pointers or pointers to arrays enable the creation and manipulation of multidimensional data structures. This flexibility is crucial for representing matrices, images, and other complex data sets where data is organized in multiple dimensions.

Iterating and Accessing Data: Pointers provide a mechanism for efficient iteration over arrays and other sequential data structures. Using pointer arithmetic, programmers can traverse arrays, access elements, and perform operations without explicitly calculating indices, thereby improving performance in data-intensive applications.

Passing Arrays to Functions (Arrays as Function Arguments): Arrays can be passed to functions in C++ either directly or using pointers. When passed directly, the size of the array must be specified. However, using pointers allows passing arrays of varying sizes and enables the function to modify the original array.

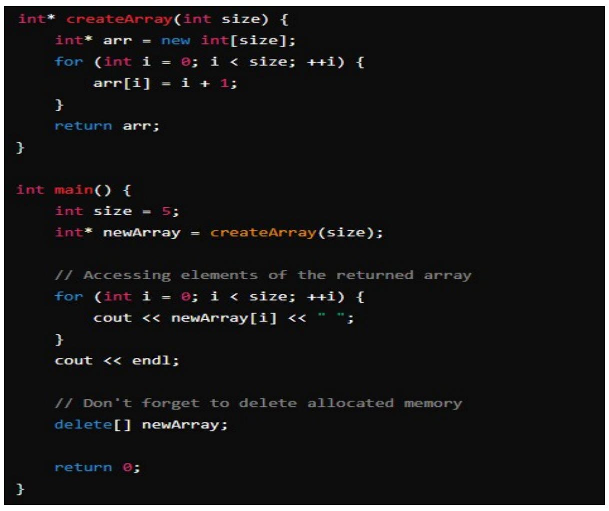


```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int myArray[] = {10, 20, 30, 40, 50};
    int size = sizeof(myArray) / sizeof(myArray[0]);

    // Passing array to function
    printArray(myArray, size);

    return 0;
}
```

In this example, printArray accepts an array arr and its size as arguments. The main function passes myArray and its size to printArray, which then prints each element of the array.

Returning Arrays from Functions:

C++ does not allow directly returning an entire array from a function. Instead, you can return a pointer to the first element of the array or use dynamic memory allocation to return arrays of variable size.

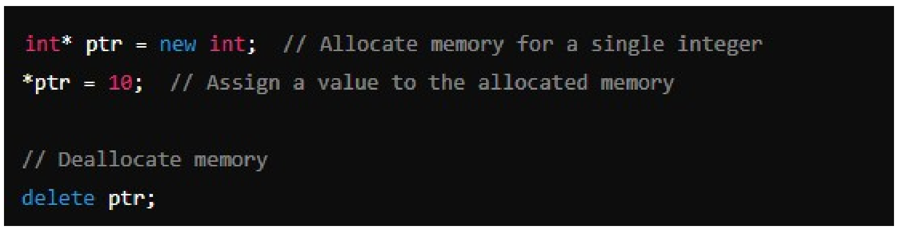


```cpp
int* createArray(int size) {
    int* arr = new int[size];
    for (int i = 0; i < size; ++i) {
        arr[i] = i + 1;
    }
    return arr;
}

int main() {
    int size = 5;
    int* newArray = createArray(size);

    // Accessing elements of the returned array
    for (int i = 0; i < size; ++i) {
        cout << newArray[i] << " ";
    }
    cout << endl;

    // Don't forget to delete allocated memory
    delete[] newArray;

    return 0;
}
```

Here, createArray dynamically allocates an array of integers of size size, initializes it, and returns a pointer to the first element. In main, newArray receives the returned pointer, allowing access to the elements of the dynamically allocated array.

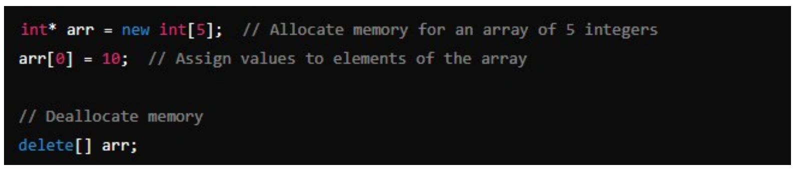
Dynamic Memory Allocation (Using new and delete):

Dynamic memory allocation in C++ is achieved using new and delete operators. new allocates memory dynamically, while delete deallocates the memory allocated by new.



```cpp
 int* ptr = new int;  // Allocate memory for a single integer
*ptr = 10;  // Assign a value to the allocated memory

// Deallocate memory
delete ptr;
```

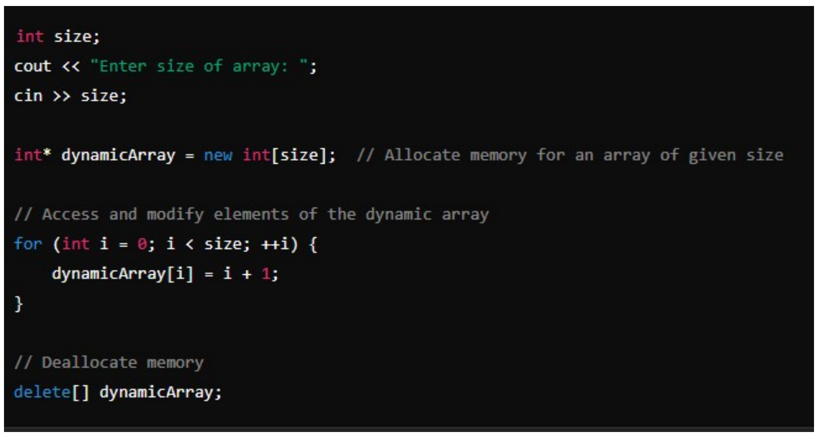
Arrays can also be allocated dynamically:



```
int* arr = new int[5];  // Allocate memory for an array of 5 integers
arr[0] = 10;  // Assign values to elements of the array

// Deallocate memory
delete[] arr;
```

Dynamic Arrays (Arrays Allocated on the Heap):

Dynamic arrays in C++ are arrays whose size is determined at runtime using dynamic memory allocation. They are allocated on the heap, allowing flexibility in size and lifetime compared to static arrays allocated on the stack.



```
int size;
cout << "Enter size of array: ";
cin >> size;

int* dynamicArray = new int[size];  // Allocate memory for an array of given size

// Access and modify elements of the dynamic array
for (int i = 0; i < size; ++i) {
    dynamicArray[i] = i + 1;
}

// Deallocate memory
delete[] dynamicArray;
```

Here, dynamicArray is allocated dynamically based on user input for size. It allows for efficient memory usage and flexibility compared to fixed-size arrays.

2.8 CONCLUSION

In conclusion, arrays and pointers form integral components of C++ programming, offering powerful capabilities in data management and memory manipulation. Arrays provide a structured way to store and access data elements sequentially, while pointers enable dynamic memory allocation and efficient memory management. Understanding these concepts is essential for developing efficient algorithms, implementing data structures, and optimizing program performance.

Throughout this exploration, we've highlighted how arrays allow for organized data storage and manipulation, supporting various operations such as iteration and sorting. Pointers, on the other hand, offer flexibility by facilitating direct memory access and dynamic memory allocation, crucial for handling large datasets and implementing complex data structures.

Moreover, the synergy between arrays and pointers extends to enhancing string manipulation, supporting function parameter passing, and enabling advanced programming techniques. Mastery of these concepts equips programmers with the tools needed to build scalable and robust software solutions in C++, ensuring efficient memory usage and effective data handling. In summary, arrays and pointers are foundational elements in C++ programming, empowering developers to tackle diverse programming challenges with precision and efficiency. Continued practice and exploration of these concepts will further strengthen programming skills and expand capabilities in software development contexts.

2.9 QUESTIONS AND ANSWERS

1. What is an array? How does it differ from a linked list?

Answer: An array is a contiguous block of memory elements where each element is of the same data type and accessed using an index. It offers constant-time access to elements but has a fixed size. In contrast, a linked list is a data structure where each element (node) contains a data field and a reference (pointer) to the next node. It allows dynamic size and efficient insertion/deletion at any position but requires linear-time access.

2. Explain the concept of pointers in C/C++ and their significance in memory management.

Answer: Pointers in C/C++ are variables that store memory addresses of other variables. They enable direct access to memory locations, facilitating dynamic memory allocation and manipulation of data structures like arrays and linked lists. They are crucial for efficient memory management and are used extensively for tasks like passing arguments to functions by reference and implementing data structures.

3. What are sparse matrices, and why are they used? Provide an example of their application.

Answer: Sparse matrices are matrices with a large number of elements that are zero. They are represented efficiently using techniques like triplet representation (COO format) or compressed sparse row/column (CSR/CSC formats). They are used to save memory and optimize operations in applications where most matrix elements are zero, such as in finite element analysis, graph algorithms, and image processing.

4. Compare the representation of arrays in row-major and column-major order. What are their advantages and disadvantages?

Answer: In row-major order, elements of a 2D array are stored row-wise in memory, while in column-major order, they are stored column-wise. Row-major order provides faster traversal of rows but slower traversal of columns, whereas column-major order is efficient for column-wise operations but slower for rows. The choice depends on the access pattern of the application and the underlying hardware architecture.

5. How are polynomials represented using arrays or linked lists? Describe an efficient way to perform polynomial addition.

Answer: Polynomials can be represented using arrays (coefficient array where index represents the exponent) or linked lists (nodes containing coefficient and exponent fields). Polynomial addition involves iterating through both polynomials and adding corresponding coefficients for each exponent. Efficient addition can be achieved by iterating through the arrays/lists simultaneously, combining terms with the same exponent, and appending remaining terms.

2.10 REFERENCES

Stroustrup, Bjarne. "The C++ Programming Language." Addison-Wesley Professional, 2013.

Eckel, Bruce. "Thinking in C++." Prentice Hall, 2000.

Deitel, Paul, and Harvey Deitel. "C++ How to Program." Pearson, 2017.

"C++ Reference - cppreference.com." Available online at: https://en.cppreference.com/w/

"C++ Programming Tutorials - GeeksforGeeks." Available online at: https://www.geeksforgeeks.org/c-plus-plus/

UNIT – 3: LISTS

Structure

3.0 INTRODUCTION

In the realm of computer science and software engineering, understanding and effectively utilizing data structures are fundamental to building efficient and scalable applications. Among these structures, lists play a pivotal role by offering a flexible means to store and manipulate collections of data elements in a linear sequence. This chapter explores various facets of lists, ranging from their theoretical underpinnings to practical implementations using different data structures.

Lists are versatile and can be implemented in multiple ways, each method offering unique advantages and addressing specific operational needs. This chapter delves into the Abstract Data Type (ADT) of lists, which provides a conceptual framework defining operations like insertion, deletion, and traversal. We explore how

lists can be implemented using arrays, linked lists—including singly linked, doubly linked, and circularly linked variations—and delve into more advanced structures like skip lists.
Understanding these implementations is crucial for developers seeking to optimize data management strategies, balance performance with memory efficiency, and adapt to diverse application requirements. By the end of this chapter, readers will gain a comprehensive understanding of lists as a foundational data structure and how different implementations cater to various computational challenges.

3.1 OBJECTIVES

After completing this unit, you will be able to understand,
Understand the Concept of Lists: Define what lists are in the context of data structures, emphasizing their linear sequence and operations.

Explore Abstract Data Type (ADT) of Lists: Introduce the ADT of lists, specifying its operations and abstracting away implementation details.

Compare and Contrast Implementations: Compare different implementations of lists, including array-based lists and various forms of linked lists (singly linked, doubly linked, circularly linked), highlighting their advantages and disadvantages.

Discuss Efficiency Considerations: Analyze the efficiency of list operations such as insertion, deletion, and search in different implementations, considering time complexity and memory usage.

Introduce Skip Lists: Introduce skip lists as a probabilistic data structure alternative to balanced trees, explaining their structure, operations, and advantages.

3.2 LISTS

A list is a linear data structure in C++ that illustrates an ordered group of elements. Every element in the list has a unique location that determines whether it may be viewed, added, or deleted. There are several ways to implement lists, but the most popular ones are linked lists and array-based lists. Contiguous memory regions are used by array-based lists, which enables quick indexed access but necessitates resizing when the capacity is reached. In contrast, linked lists are efficient for insertion and deletion operations at any location since they are made up of nodes that each contain data and a pointer to the next node. However, accessing items of linked lists is slower than with array-based lists. C++ lists are flexible and capable of managing homogeneous data types. They can also dynamically modify their size to fit different data sizes. They are essential in many applications, such as algorithm implementation and data management.

Characteristics of Lists

Ordered Collection: The elements in a list are ordered, meaning each element has a specific position (index) within the list.

Indexed Access: Elements can be accessed, inserted, or deleted based on their index.

Dynamic Size: Lists can grow and shrink in size dynamically, allowing elements to be added or removed.

Homogeneous or Heterogeneous: Depending on the implementation and language, lists can contain elements of the same type (homogeneous) or elements of different types (heterogeneous).

Operations on Lists (insertion, deletion, traversal)

Insertion: Adding an element to the list at a specific position.

Deletion: Removing an element from the list based on its position or value.

Traversal: Accessing each element of the list, typically using loops or recursion.

Searching: Finding an element in the list based on its value.

Updating: Changing the value of an element at a specific position.

Insertion:

Algorithm for Insertion:

At the end:

Check if the array is full. If yes, resize the array.

Add the new element to the end of the array.

Increment the size of the array.

At a specific position:

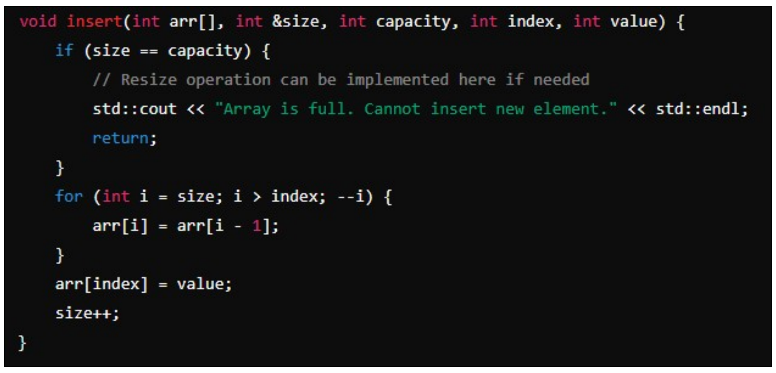
Check if the array is full. If yes, resize the array.

Shift elements from the specified position to the right.

Insert the new element at the specified position.

Increment the size of the array.

Code Example:



```cpp
void insert(int arr[], int &size, int capacity, int index, int value) {
    if (size == capacity) {
        // Resize operation can be implemented here if needed
        std::cout << "Array is full. Cannot insert new element." << std::endl;
        return;
    }
    for (int i = size; i > index; --i) {
        arr[i] = arr[i - 1];
    }
    arr[index] = value;
    size++;
}
```

Deletion

Algorithm for Deletion:

At the end:

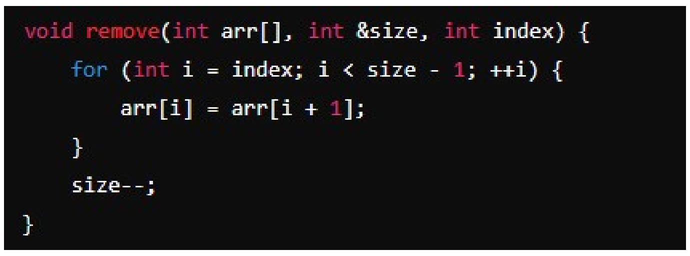
Simply decrement the size of the array.

At a specific position:

Shift elements from the specified position to the left.

Decrement the size of the array.

Code Example:



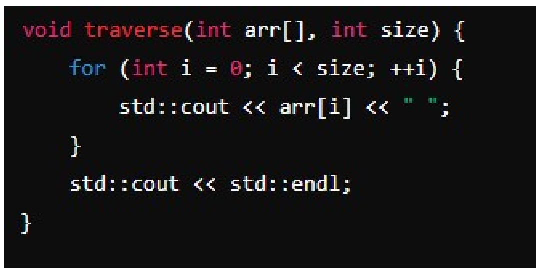
```cpp
void remove(int arr[], int &size, int index) {
    for (int i = index; i < size - 1; ++i) {
        arr[i] = arr[i + 1];
    }
    size--;
}
```

Traversal

Algorithm for Traversal:

Iterate over each element in the array and perform the desired operation.

Code Example:



```cpp
void traverse(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

Types of Lists (array-based, linked lists, skip lists)

Array-based Lists

Array-based lists, often implemented using arrays or vectors, store elements in contiguous memory locations. This structure allows for O(1) time complexity for accessing elements by their index, making it ideal for applications requiring frequent random access. However, insertions and deletions, especially in the middle or beginning of the list, are less efficient, typically O(n) due to the need to shift elements. Array-based lists are suitable for use cases where the list size does not change frequently or can be resized dynamically, such as managing a list of fixed-size records or a collection of items that is primarily read-only.

Singly Linked Lists

A singly linked list consists of nodes, each containing data and a pointer to the next node. This structure provides dynamic sizing and allows for efficient O(1) insertions and deletions at the beginning of the list. However, accessing elements requires O(n) time as it involves traversing the list sequentially from the head node. Singly linked lists are advantageous in scenarios where frequent insertions and deletions are required, such as implementing stacks, queues, or managing a dynamic collection of elements where the order of elements needs to be maintained without frequent random access.

Doubly Linked Lists

Doubly linked lists enhance singly linked lists by having nodes that contain pointers to both the next and previous nodes, enabling bidirectional traversal. This feature allows for efficient insertions and deletions at both ends and anywhere within the list with O(1) complexity, provided the node to be inserted or deleted is known.

However, they require additional memory for the extra pointer in each node. Doubly linked lists are useful in applications such as navigation systems where backward and forward traversal is needed, or in implementing complex data structures like deques and certain types of caches.

Circular Linked Lists

Circular linked lists are a variation of linked lists where the last node points back to the first node, forming a circle. This allows for continuous traversal of the list and can be implemented as either singly or doubly linked. Circular linked lists are particularly useful in scenarios requiring cyclic iteration, such as in round-robin scheduling or implementing a circular buffer. They provide the same benefits as their singly or doubly linked counterparts, with the added advantage of naturally supporting circular traversal without additional checks.

Skip Lists

Skip lists are an advanced data structure that enhances linked lists with multiple levels of links, allowing for efficient O(log n) search, insertion, and deletion operations. By using randomization, skip lists maintain a balanced structure probabilistically, providing performance similar to balanced trees but with simpler algorithms. Each element in the skip list is part of multiple linked lists at different levels, with higher levels skipping over multiple elements, thus speeding up the search process. Skip lists are ideal for applications requiring fast search times, such as databases, in-memory data structures, and distributed systems.

3.3 ABSTRACT DATA TYPE - LIST

An Abstract Data Type (ADT) for a list is a conceptual model that defines a collection of elements organized in a linear sequence. It provides a clear interface specifying operations that can be performed on the list, without specifying how these operations are implemented. Here's an overview of the Abstract Data Type - List:

A list is an ordered collection of elements where each element has a specific position or index. Elements can be of any data type, and the list can dynamically grow or shrink in size. Elements in a list are arranged in a linear sequence, where each element (except possibly the first and last) has a unique predecessor and successor.

Operations Defined for List ADT

Insertion: Adds an element at a specified position in the list.

Deletion: Removes an element from a specified position in the list.

Access: Retrieves the element at a specified position in the list.

Traversal: Iterates through all elements in the list, typically from the beginning to the end.

Search: Finds the position of a specified element in the list, if it exists.

Size Management: Provides operations to determine the number of elements currently in the list.

Concatenation: Combines two lists into a single list.

Sorting: Arranges elements in a specified order, such as ascending or descending.

Implementation Considerations

Array-based Implementation: Uses a contiguous block of memory to store elements, allowing direct access by index but requiring resizing operations for dynamic lists.

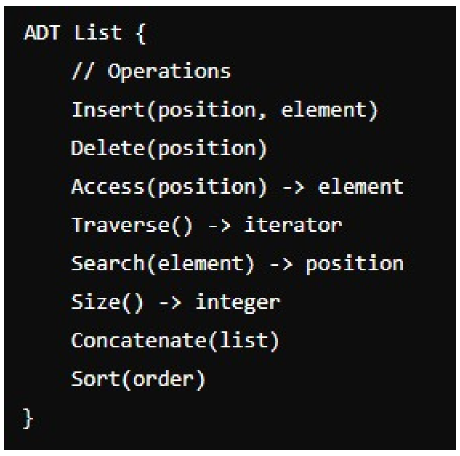
Linked List Implementation: Utilizes nodes with pointers/references to connect elements, providing flexibility in size and efficient insertion/deletion operations.

Usage and Applications

Data Structures: Lists are fundamental in various data structures like stacks, queues, and priority queues.

Applications: Used in applications requiring dynamic data management, such as databases, text processing, and simulations.

Example of List ADT Interface (Pseudocode)



```
ADT List {
    // Operations
    Insert(position, element)
    Delete(position)
    Access(position) -> element
    Traverse() -> iterator
    Search(element) -> position
    Size() -> integer
    Concatenate(list)
    Sort(order)
}
```

ADT operations and their specifications

list of operations commonly associated with Abstract Data Types (ADTs) and their typical specifications. These operations provide a standardized interface for interacting with data structures, ensuring consistency in behavior while abstracting away implementation details:

1. Insertion (Insert): Adds an element to the data structure at a specified position or according to specific rules.

Parameters:

position: Position where the element should be inserted.

element: The element to be inserted.

Returns: true if insertion is successful, false otherwise (e.g., if position is out of bounds).

2. Deletion (Delete)

Description: Removes an element from the data structure at a specified position.

Parameters: position: Position of the element to be deleted.

Returns: true if deletion is successful, false otherwise (e.g., if position is out of bounds).

3. Access (Get): Retrieves the element from the data structure at a specified position without modifying the data structure.

Parameters: position: Position of the element to retrieve.

Returns: The element at the specified position, or a specified default value or error indicator if position is out of bounds.

4. Search (Find)

Description: Searches for a specified element within the data structure.

Parameters: element: Element to search for.

Returns: Position/index of the element if found, or a specified indicator (e.g., -1 or nullptr) if not found.

Advantages and disadvantages of using ADT List

Using an Abstract Data Type (ADT) List offers several advantages and disadvantages, depending on the specific requirements and context of the application. Here's a breakdown of the key advantages and disadvantages:

Advantages:

Flexibility: ADT List provides a flexible structure for storing and manipulating elements in a linear sequence. It supports various operations such as insertion, deletion, access, and traversal, making it versatile for different application needs.

Modularity: ADT List abstracts away the implementation details, allowing programmers to focus on the interface and functionality of the data structure rather than low-level operations. This promotes modular programming and enhances code reusability.
Ease of Use: The defined operations (insertion, deletion, etc.) provide a clear and standardized way to interact with the data structure. This makes it easier for developers to understand and maintain the code.

Performance: Depending on the implementation (e.g., array-based or linked list-based), ADT List can offer efficient performance characteristics for specific operations. For example, arrays provide O(1) access time, while linked lists offer O(1) insertion/deletion time at the head/tail.

Scalability: ADT List implementations can scale well with the size of the data. Dynamic resizing (in array-based lists) or node allocation (in linked lists) allows the list to grow or shrink as needed, accommodating varying data sizes efficiently.

Disadvantages:

Memory Overhead: Some implementations of ADT List, especially linked lists, can incur memory overhead due to storing additional pointers or references for linking elements. This overhead may affect memory usage efficiency, particularly for large datasets.

Access Time Complexity: Depending on the implementation, certain operations such as random access (e.g., accessing elements by index in linked lists) may have higher time complexity (e.g., O(n) for linked lists vs. O(1) for arrays). This can impact performance in applications requiring frequent random access.

Complexity of Operations: While ADT List abstracts implementation details, certain operations like insertion or deletion in specific positions (e.g., middle of the list) can be complex and may require careful handling of pointers/references (in linked lists) or resizing operations (in arrays).

Lack of Specificity: ADT List provides a general-purpose interface for lists but may not optimize performance for specific use cases. Specialized data structures (e.g., queues, stacks, priority queues) may offer more tailored solutions for particular application requirements.

Dependency on Implementation: The efficiency and characteristics of ADT List heavily depend on the chosen implementation (e.g., array-based vs. linked list-based). Selecting the appropriate implementation is crucial for achieving desired performance and memory usage goals.

3.4 ARRAY IMPLEMENTATION OF LISTS

Array implementation of lists involves using a contiguous block of memory to store the elements of the list. In this structure, each element is stored in an indexed position, allowing for O (1) time complexity for access by index, which makes it efficient for random access operations. However, array-based lists require resizing when the capacity is exceeded, which involves creating a new larger array and copying the elements from the old array to the new one, an operation with O(n) time complexity. Additionally, insertions and deletions, especially at the beginning or in the middle of the list, are less efficient because they require shifting elements to maintain order, also with O(n) time complexity. Despite these limitations, array-based lists are widely used due to their straightforward implementation and efficient access times, making them suitable for applications where frequent random access is needed and the list size doesn't change dramatically. Common examples include dynamic arrays such as C++'s std::vector and Java's ArrayList.

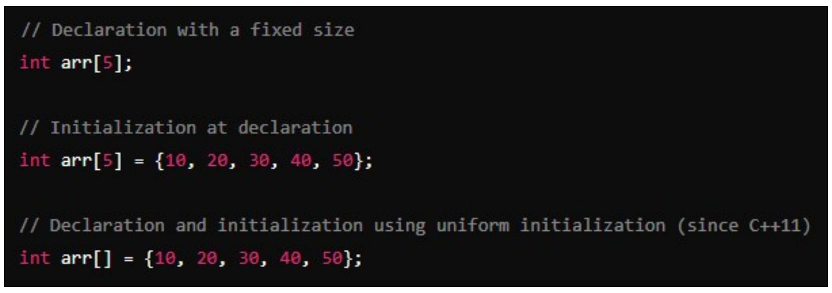
Basics of array data structure

An array is a linear data structure consisting of a collection of elements (values or variables), each identified by at least one index or key. Elements are typically stored in contiguous memory locations, allowing for efficient access to individual elements using their index.

Syntax

Declaration and Initialization

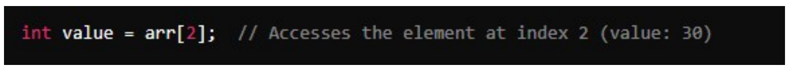
Arrays in C++ are declared using a fixed size and can be initialized with specific values at the time of declaration or later.



```
// Declaration with a fixed size
int arr[5];

// Initialization at declaration
int arr[5] = {10, 20, 30, 40, 50};

// Declaration and initialization using uniform initialization (since C++11)
int arr[] = {10, 20, 30, 40, 50};
```

Accessing Elements

Elements in an array are accessed using zero-based indexing.



```
int value = arr[2];  // Accesses the element at index 2 (value: 30)
```

Updating Elements

Individual elements of an array can be updated by assigning a new value to the corresponding index.

```
arr[3] = 45;  // Updates the element at index 3 to 45
```

Iterating Through an Array

Arrays are typically iterated using loops like for or while.

```
for (int i = 0; i < 5; ++i) {
    std::cout << arr[i] << " ";
}
std::cout << std::endl;
```

Characteristics

Fixed Size: Arrays have a fixed size defined at the time of declaration, which determines the maximum number of elements they can store.

Homogeneous Elements: Arrays usually store elements of the same data type (e.g., integers, characters).

Index-based Access: Elements in an array are accessed using numeric indices starting from 0 up to size-1, providing O(1) time complexity for accessing an element by its index.

Contiguous Memory Allocation: Elements in an array are stored next to each other in memory, which facilitates efficient traversal and sequential access.

Operations

Access: Retrieve the value of an element at a specific index.

Insertion: Add an element at a specified position within the array.

Deletion: Remove an element from a specified position, often requiring elements to be shifted.

Update: Modify the value of an existing element at a specific index.

Traversal: Iterate through all elements of the array sequentially.

Usage

Arrays are widely used in programming for various purposes:

Data Storage: Storing collections of data elements that need to be accessed efficiently.

Implementing Other Data Structures: Serving as the underlying structure for more complex data structures like stacks, queues, and hash tables.

Matrix Operations: Representing and manipulating matrices in mathematical computations and algorithms.

Sorting and Searching: Arrays are essential for implementing sorting algorithms (e.g., bubble sort, quicksort) and searching algorithms (e.g., binary search).

Buffering: Handling input/output operations and buffering data in applications.

Implementing a list using arrays

Implementing a list (or a dynamic array-based list) using arrays involves creating a data structure that can dynamically resize itself as elements are added or removed. Here's a basic implementation of a list using arrays in C++:

```
#include <iostream>
class ArrayList {
private:
    int capacity;   // Maximum capacity of the list
    int size;       // Current number of elements in the list
    int* arr;       // Pointer to the array storing elements
public:
    // Constructor to initialize an empty list
    ArrayList(int capacity) {
```

```
    this->capacity = capacity;
    this->size = 0;
    this->arr = new int[capacity];
}
// Destructor to free memory allocated to the array
~ArrayList() {
    delete[] arr;
}
// Function to insert an element at the end of the list
void insert(int value) {
    if (size < capacity) {
        arr[size++] = value;
    } else {
        std::cout << "List is full. Cannot insert." << std::endl;
    }
}
// Function to remove an element from the list at a specific index
void remove(int index) {
    if (index < 0 || index >= size) {
        std::cout << "Invalid index. Cannot remove." << std::endl;
    } else {
        for (int i = index; i < size - 1; ++i) {
            arr[i] = arr[i + 1];
        }
        size--;
    }
}
// Function to get the size of the list (number of elements)
int getSize() {
    return size;
}
// Function to print all elements in the list
```

```
    void print() {
        std::cout << "List elements:";
        for (int i = 0; i < size; ++i) {
            std::cout << " " << arr[i];
        }
        std::cout << std::endl;
    }
};
// Example usage of the ArrayList class
int main() {
    // Create an ArrayList with initial capacity of 5
    ArrayList list(5);
    // Insert elements into the list
    list.insert(10);
    list.insert(20);
    list.insert(30);
    // Print current elements in the list
    list.print();  // Output: List elements: 10 20 30
    // Remove an element from the list
    list.remove(1);  // Removes element at index 1 (20)
    // Print updated list
    list.print();  // Output: List elements: 10 30
    return 0;
}
```

Efficiency considerations (time and space complexity)

When implementing a list using arrays, efficiency considerations revolve around several key aspects that impact the performance and usability of the data structure:

1. Dynamic Resizing

Arrays have a fixed size once allocated, which necessitates careful handling when the number of elements exceeds the initial capacity. Dynamic resizing strategies involve:

Doubling the Array Size: When the array reaches capacity, allocate a new array of double the current size, copy existing elements, and deallocate the old array. This strategy amortizes the cost of resizing, typically resulting in O(1) average time complexity for insertions.

Shrinking the Array: When the number of elements decreases significantly, consider resizing the array to save memory, though this operation may be less frequent.

2. Insertion and Deletion

Efficient insertion and deletion operations are critical for list implementations using arrays:

Insertion:

End of List: O(1) average time complexity if space is available.

Middle of List: O(n) time complexity due to shifting elements after the insertion point.

Deletion:

End of List: O(1) time complexity for removing the last element.

Middle of List: O(n) time complexity due to shifting elements after the deletion point.

3. Access and Search

Arrays offer O(1) time complexity for accessing elements by index, which is advantageous for random access:

Ensure indices are within bounds to prevent out-of-bound errors, which can lead to runtime issues.

4. Memory Management

Efficient memory management practices include:

Allocating Memory: Allocate sufficient memory initially based on expected usage to minimize frequent resizing.

Deallocating Memory: Properly deallocate memory when elements are removed or when the list is destroyed to prevent memory leaks.

5. Trade-offs with Other Data Structures

Consider trade-offs between array-based lists and other data structures like linked lists:

Arrays vs. Linked Lists: Arrays offer efficient random access but can be inefficient for frequent insertions/deletions in the middle. Linked lists excel in dynamic resizing and efficient insertions/deletions but may consume more memory due to node overhead.

6. Amortized Analysis

Use amortized analysis to evaluate the average time complexity of operations over a series of operations rather than individual ones, especially for resizing operations in dynamic arrays.

Example Considerations

In the context of the previously discussed ArrayList implementation:

Insertions: Efficient at the end (O(1)), less efficient in the middle (O(n)).

Deletions: Efficient at the end (O(1)), less efficient in the middle (O(n)).

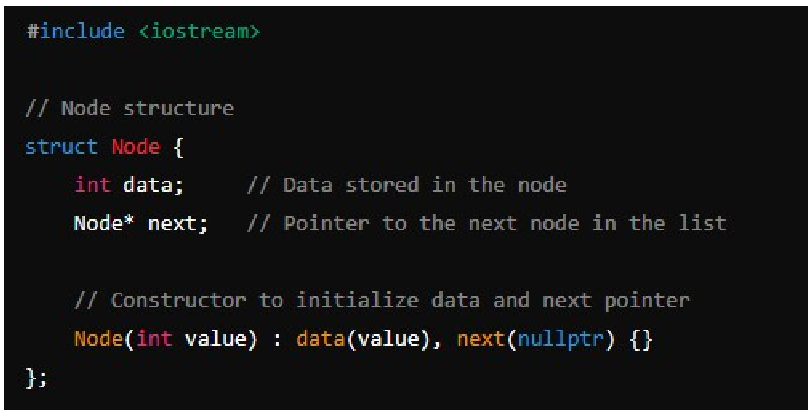
Resizing: Occurs infrequently due to doubling strategy, amortizing the cost of resizing over multiple operations.

3.5 LINKED LISTS - IMPLEMENTATION

Implementing linked lists involves defining the structure of nodes and operations to manipulate these nodes. Linked lists are composed of nodes where each node contains data and a pointer/reference to the next node in the sequence. Here's a basic outline of how linked lists can be implemented in C++:

Node Structure

First, define a structure for the nodes of the linked list:



```cpp
#include <iostream>

// Node structure
struct Node {
    int data;      // Data stored in the node
    Node* next;    // Pointer to the next node in the list

    // Constructor to initialize data and next pointer
    Node(int value) : data(value), next(nullptr) {}
};
```

Types of linked lists (singly linked, doubly linked, circularly linked)

Linked lists are versatile data structures that come in several types, each offering unique advantages for different applications. Here's an overview of the types of linked lists—singly linked, doubly linked, and circularly linked—and their operations with algorithms:

1. Singly Linked List

In a singly linked list, each node contains data and a pointer/reference to the next node in the sequence. It only allows traversal in one direction—from the head to the last node.

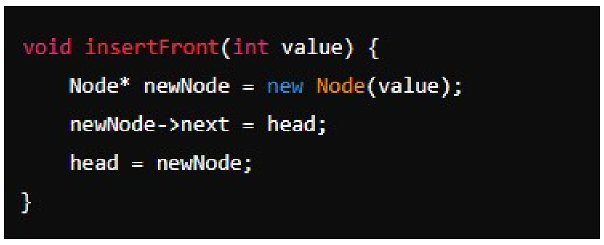
Operations:

Insertion at the Beginning (insertFront):

Create a new node with the given data.

Point the new node's next to the current head.

Update head to point to the new node.



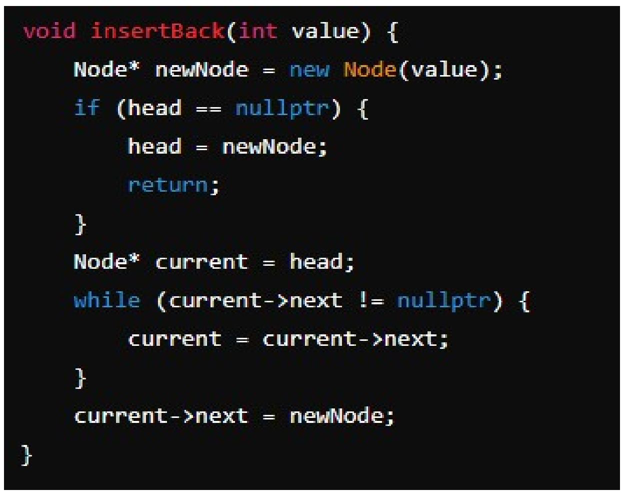
```
void insertFront(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}
```

Insertion at the End (insertBack):

Traverse the list to find the last node.

Create **a new node with the given data and set its next to nullptr.**

Point **the last node's next to the new node.**



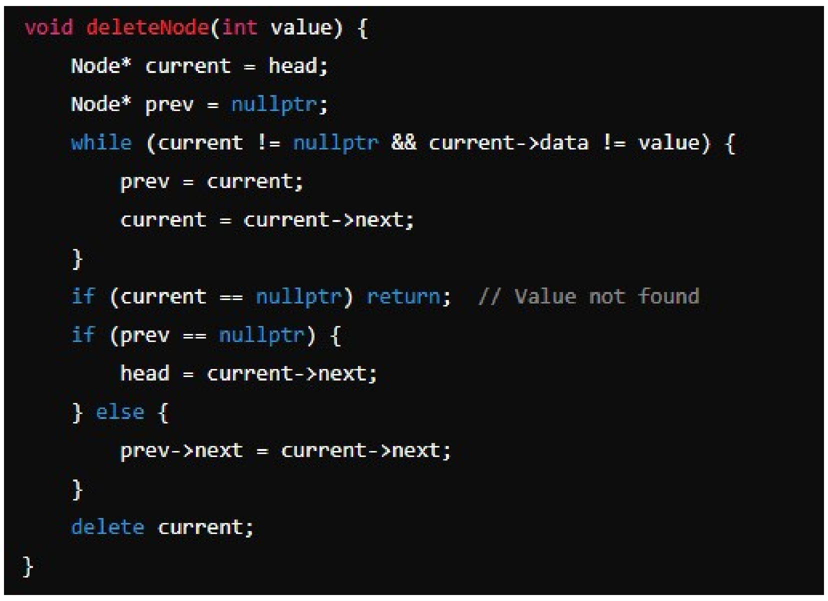
```
void insertBack(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
}
```

Deletion by Value (deleteNode):

Traverse the list to find the node with the given value and its predecessor.

Update the predecessor node's next to skip the node to be deleted.

Delete the node and free memory.



```
void deleteNode(int value) {
    Node* current = head;
    Node* prev = nullptr;
    while (current != nullptr && current->data != value) {
        prev = current;
        current = current->next;
    }
    if (current == nullptr) return;  // Value not found
    if (prev == nullptr) {
        head = current->next;
    } else {
        prev->next = current->next;
    }
    delete current;
}
```

Comparisons with array-based lists

	Array-based Lists	Linked Lists
Memory Allocation	Allocate contiguous memory block, typically resizing when capacity is exceeded.	Memory Allocation: Nodes dynamically allocated as needed, supporting efficient memory usage.
Memory Usage	May allocate more memory than needed due to pre-allocation or resizing strategies.	Overhead due to storing pointers/references for linking nodes.
Insertions and Deletions	Insertions: Efficient at the end with amortized constant time complexity (O(1)), but inefficient in the middle due to shifting elements (O(n)).	Insertions: Efficient at both ends (O(1) for head/tail), and efficient in the middle with direct node manipulation (O(1) given node reference). Deletions: Efficient with direct node access (O(1)

	Deletions: Similar to insertions, O(n) in worst-case for deletions in the middle.	given node reference), but O(n) for searching node to delete.
Random Access	O(1) time complexity for accessing elements by index, due to contiguous memory allocation.	O(n) time complexity for accessing elements by index, requiring traversal from the head to the desired index.
Space Efficiency	Efficient in terms of space utilization when the list is nearly full due to contiguous allocation.	May consume more memory due to node overhead (next/prev pointers), especially for small data sizes.
Implementation Complexity	Simple to implement and understand, with direct indexing and straightforward operations.	More complex due to pointer manipulation, requiring careful management of node connections and potential for memory leaks.

3.6 DOUBLY LINKED LISTS - IMPLEMENTATION

A doubly linked list extends the singly linked list by each node containing an additional pointer/reference to the previous node, allowing bidirectional traversal.

Operations:

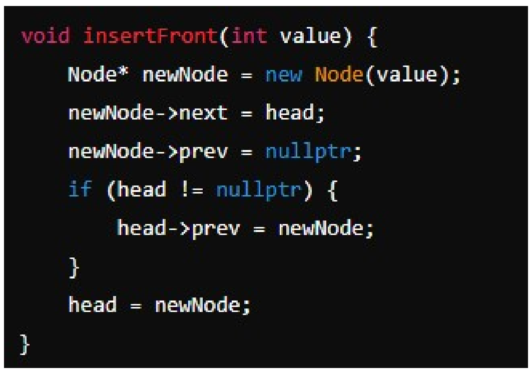
Insertion at the Beginning (insertFront):

Create a new node with the given data.

Set its next to the current head and its prev to nullptr.

Update the prev of the current head to point to the new node.

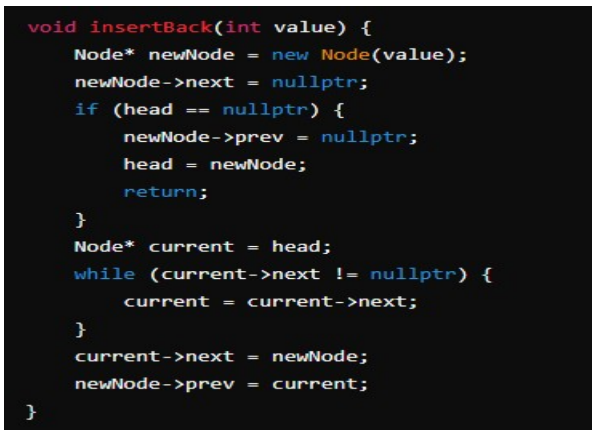
Update head to point to the new node.



```cpp
void insertFront(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    newNode->prev = nullptr;
    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}
```

Insertion at the End (insertBack):

Similar to singly linked list, but also update the prev of the new node to point to the current last node.



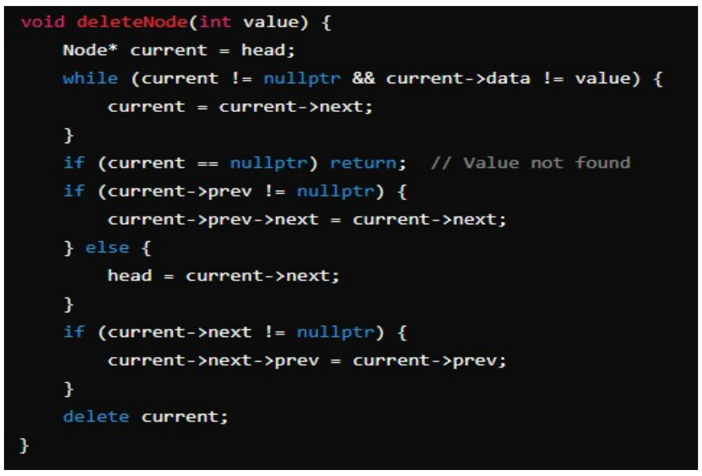
```cpp
void insertBack(int value) {
    Node* newNode = new Node(value);
    newNode->next = nullptr;
    if (head == nullptr) {
        newNode->prev = nullptr;
        head = newNode;
        return;
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
    newNode->prev = current;
}
```

Deletion by Value (deleteNode):

Traverse the list to find the node with the given value.

Update the next of the predecessor node and the prev of the successor node to skip the node to be deleted.

Delete the node and free memory.



```
void deleteNode(int value) {
    Node* current = head;
    while (current != nullptr && current->data != value) {
        current = current->next;
    }
    if (current == nullptr) return;  // Value not found
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    } else {
        head = current->next;
    }
    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }
    delete current;
}
```

Advantages over singly linked lists

Doubly linked lists offer several advantages over singly linked lists, primarily due to their ability to support bidirectional traversal and more flexible node manipulation. Here are the key advantages of doubly linked lists over singly linked lists:

1. Bidirectional Traversal

In a doubly linked list, each node maintains pointers to both its previous and next nodes. This bidirectional linkage allows traversal in both directions—from head to tail and from tail to head. This feature enables efficient operations that require accessing nodes in reverse order, which is not possible or efficient with singly linked lists.

2. Easy Deletion of Nodes

Deleting a node in a doubly linked list is more straightforward compared to a singly linked list:

Singly Linked List: To delete a node, you typically need to traverse the list to find the node and modify its previous node's next pointer to skip over the node to be deleted. This requires knowing the previous node, which may involve an additional traversal.

Doubly Linked List: In contrast, a doubly linked list allows direct access to both the previous and next nodes of any given node. Thus, deleting a node involves simply adjusting the next and prev pointers of its adjacent nodes, without needing to traverse the list again to find the previous node.

3. Insertions and Deletions at Both Ends

Doubly linked lists support efficient insertions and deletions at both the head and tail of the list:

Insertion at the Head: In a doubly linked list, inserting a node at the head involves updating the next pointer of the new node to point to the current head, updating the prev pointer of the current head (if it exists), and updating the head pointer to the new node. This operation is O(1) constant time complexity.

Insertion at the Tail: Similarly, inserting a node at the tail of a doubly linked list is efficient. It involves updating the next pointer of the current last node to point to the new node, updating the prev pointer of the new node to point to the current last node, and updating the tail pointer (if maintained) to the new node. This operation is also O(1) constant time complexity.

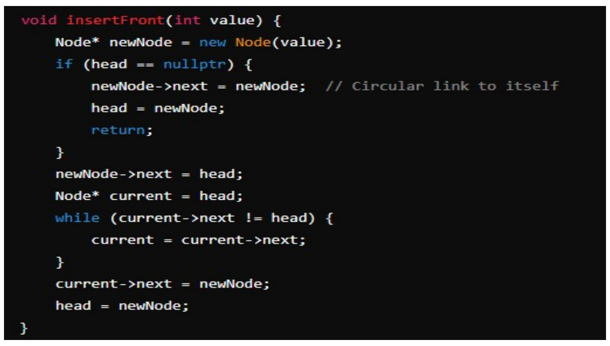
3.7 CIRCULARLY LINKED LIST SIMPLEMENTATION

In a circularly linked list, the last node points back to the first node, forming a circular loop. This structure allows for continuous traversal.

Operations:

Insertion at the Beginning (insertFront):

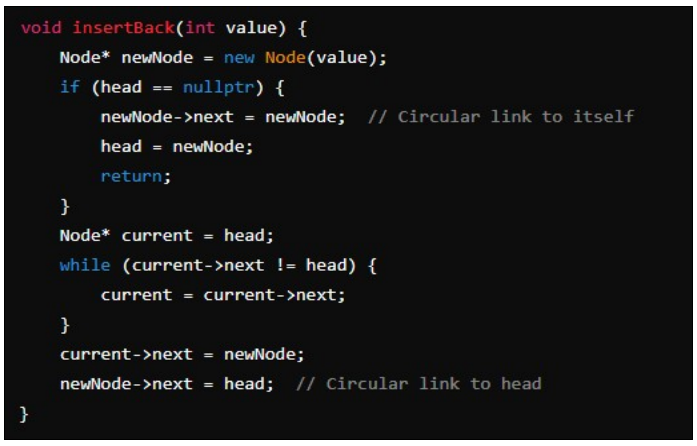
Similar to singly linked list insertion at the beginning, but handle the circular link by pointing the last node's next to the new node.



```
void insertFront(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        newNode->next = newNode;  // Circular link to itself
        head = newNode;
        return;
    }
    newNode->next = head;
    Node* current = head;
    while (current->next != head) {
        current = current->next;
    }
    current->next = newNode;
    head = newNode;
}
```

Insertion at the End (insertBack):

Traverse to find the last node and update its next to point to the new node.



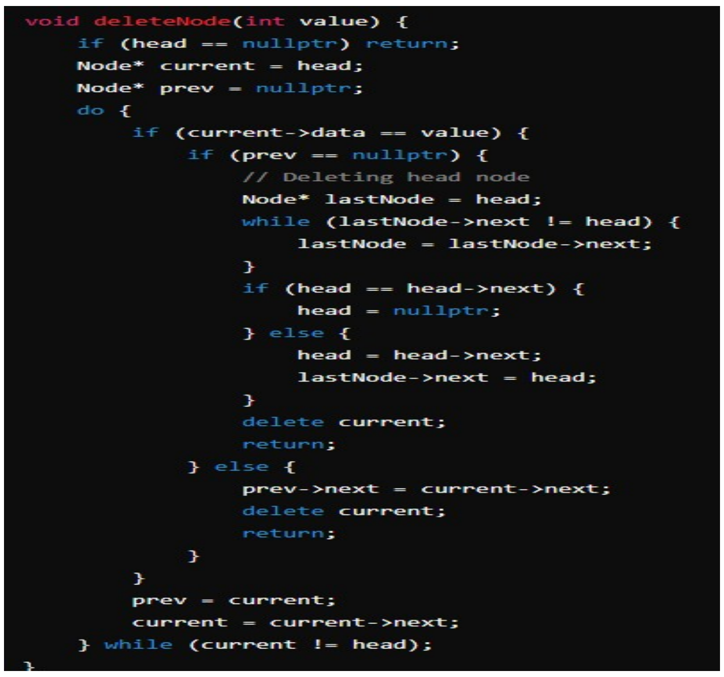
```
void insertBack(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        newNode->next = newNode;  // Circular link to itself
        head = newNode;
        return;
    }
    Node* current = head;
    while (current->next != head) {
        current = current->next;
    }
    current->next = newNode;
    newNode->next = head;  // Circular link to head
}
```

Deletion by Value (deleteNode):

Traverse the list to find the node with the given value and its predecessor.

Update the predecessor node's next to skip the node to be deleted.

Handle circular links to maintain integrity.



```
void deleteNode(int value) {
    if (head == nullptr) return;
    Node* current = head;
    Node* prev = nullptr;
    do {
        if (current->data == value) {
            if (prev == nullptr) {
                // Deleting head node
                Node* lastNode = head;
                while (lastNode->next != head) {
                    lastNode = lastNode->next;
                }
                if (head == head->next) {
                    head = nullptr;
                } else {
                    head = head->next;
                    lastNode->next = head;
                }
                delete current;
                return;
            } else {
                prev->next = current->next;
                delete current;
                return;
            }
        }
        prev = current;
        current = current->next;
    } while (current != head);
}
```

Applications where circular lists are useful

ircular lists, also known as circularly linked lists, find applications in various scenarios where cyclic or continuous access patterns are advantageous. Here are some notable applications where circular lists are useful:

1. Circular Buffers or Ring Buffers

Circular lists are commonly used to implement circular buffers, also known as ring buffers or cyclic buffers. These buffers are fixed-size arrays managed as circular lists, where elements wrap around upon reaching the end of the buffer. Key applications include:

Data Streaming: In real-time data processing or streaming applications, circular buffers efficiently manage continuous data flow, such as audio or video streams, without needing to resize or shift data.

Embedded Systems: Circular buffers are extensively used in embedded systems for managing data between different parts of a system, where efficient memory management and predictable behavior are crucial.

2. Round-Robin Scheduling

In operating systems and task scheduling algorithms, circular lists facilitate round-robin scheduling, where tasks are scheduled in a circular sequence. Each task gets a predefined time slice before the scheduler moves to the next task in the sequence. This approach ensures fair allocation of CPU time among multiple tasks.

CPU Scheduling: In multitasking environments, round-robin scheduling using circular lists ensures that all processes receive an equal share of CPU time, promoting fairness and preventing starvation.

3. Managing Circular Lists of Objects

Circular lists are also useful in managing cyclic relationships or sequences of objects that naturally form a loop:

Game Development: In game development, circular lists can manage objects or entities that move in a continuous loop, such as a game world where characters or objects wrap around the screen.

Data Structures: Circular lists are employed in implementing data structures like circular queues, which efficiently manage data in applications such as event handling or task processing where data needs to be processed in a continuous loop.

4. Navigation and Routing Algorithms

In geographical applications and routing algorithms, circular lists can represent circular paths or continuous routes:

Navigation Systems: Circular lists are used to represent circular routes or paths in navigation systems, where routes wrap around to the starting point.

Network Routing: In network protocols and algorithms, circular lists can represent circular paths in data packet routing, ensuring packets are forwarded in a loop until reaching their destination or timing out.

5. Resource Management and Allocation

Circular lists are also utilized in resource management and allocation scenarios:

Memory Management: Circular lists can manage memory allocation in memory pools or memory caches, where memory blocks are reused in a continuous loop to optimize memory usage and access.

Resource Allocation: In resource allocation algorithms, circular lists can manage the allocation and deallocation of resources, ensuring efficient utilization and recycling of resources in a cyclic manner.

3.8 SKIP LIST

Skip lists are a data structure that combines the advantages of linked lists with probabilistic balancing, allowing for fast search, insertion, and deletion operations. They are particularly useful in scenarios where balanced trees (like AVL trees or red-black trees) might be too complex or where dynamic data structures with efficient average-case performance are required. Here’s an overview of skip lists, their structure, operations, and applications:

Structure of Skip Lists

Layers: Skip lists are composed of multiple layers (or levels), where each level is essentially a linked list. The bottom level (level 0) contains all elements in sorted order.

Skip Pointers: Nodes at each level have pointers that may skip over several elements in the list. Higher levels have fewer nodes, with each node skipping more elements, effectively speeding up search operations.

Header and Sentinel Nodes: Skip lists typically include header nodes at each level to simplify boundary conditions and sentinel nodes (often nullptr or an infinite node) at the end of each list.

Operations on Skip Lists

Search: Skip lists support efficient search operations, similar to binary search trees but without requiring strict balance conditions. Starting from the top level, skip pointers are used to quickly narrow down the search range.

Insertion: To insert an element, determine the insertion point using search. Randomly decide the level of the new node (higher levels are less probable), and update skip pointers at each level accordingly to maintain the list's structure.

Deletion: Deleting an element involves updating skip pointers to bypass the node to be deleted at each level. This operation requires careful adjustment to maintain the skip list's properties.

Advantages of Skip Lists

Average-Case Performance: Skip lists offer average-case O(log n) time complexity for search, insertion, and deletion operations, similar to balanced binary search trees but with simpler maintenance requirements.

Simplicity: Compared to balanced trees, skip lists are easier to implement and manage. They do not require rebalancing operations, making them more suitable for dynamic datasets with frequent updates.

Versatility: Skip lists can be adapted for various applications where efficient search and insertion operations are critical, such as database indexing, priority queues, and probabilistic data structures.

Applications of Skip Lists

Database Indexing: Skip lists are used in databases to speed up search operations, providing efficient indexing structures for large datasets.

Concurrency Control: In concurrent programming, skip lists can be adapted for lock-free data structures, enabling efficient and scalable access to shared resources.

Priority Queues: Skip lists can serve as the basis for priority queues, where elements are dynamically prioritized based on their keys or values.

3.9 CONCLUSION

In conclusion, the study of lists as fundamental data structures reveals their indispensable role in computer science and software engineering. Lists, characterized by their linear arrangement of elements, provide a versatile framework for organizing and manipulating data in a sequential manner. Throughout this chapter, we have explored various implementations and aspects of lists, starting with their conceptual underpinnings as the Abstract Data Type (ADT) of lists. This foundational understanding paved the way for delving into practical implementations such as array-based lists, which offer direct access but require careful management of memory and resizing, and linked lists, including singly linked, doubly linked, and circularly linked variations, each suited to different operational needs and efficiency considerations.

Additionally, skip lists emerged as a notable alternative, leveraging probabilistic techniques to provide efficient search, insertion, and deletion operations. By comparing these implementations, we have underscored how different design choices impact performance metrics like time complexity and memory usage, crucial for optimizing data-intensive applications. Practical examples across databases, scheduling algorithms, and more illustrate the versatility and real-world applicability of lists. Mastery of these structures equips developers with the tools to design efficient, scalable solutions tailored to diverse computational challenges, ensuring robust performance and adaptability in software systems.

In essence, lists remain pivotal in both theoretical foundations and practical applications within the realm of data structures. Their continual evolution and adaptation underscore their enduring relevance in modern computing, promising continued exploration and innovation in leveraging lists for optimal data management and computational efficiency.

3.10 QUESTIONS AND ANSWERS

Q1: What is the main difference between an array-based list and a linked list?
Answer: The main difference lies in how they store and access elements.
Array-based lists store elements in contiguous memory locations, allowing for fast random access using indices. However, resizing an array can be costly, especially if it exceeds its allocated capacity.

Linked lists, on the other hand, use nodes with pointers to link elements, which allows for efficient insertion and deletion operations but does not support direct indexing. Each type has its advantages based on the specific application needs for access and modification operations.

Q2: Why would you choose a doubly linked list over a singly linked list?

Answer: Doubly linked lists offer bidirectional traversal capabilities compared to singly linked lists, which only support forward traversal. This bidirectional feature allows for efficient backward traversal and easier node deletion operations as each node maintains references to both its previous and next nodes. However, doubly linked lists require more memory due to storing an additional pointer for each node, and they are more complex to implement and maintain than singly linked lists.

Q3: What are skip lists, and what advantages do they offer over traditional balanced trees?

Answer: Skip lists are probabilistic data structures that provide efficient search, insertion, and deletion operations similar to balanced trees (e.g., AVL trees, red-black trees) but with simpler implementation and maintenance requirements. They achieve this by linking elements across multiple levels, where each level represents a progressively sparser subset of the elements. Skip lists offer average-case O(log n) time complexity for search, insertion, and deletion operations, making them suitable for applications where maintaining balanced trees would be overly complex or unnecessary.

Q4: In what scenarios would you prefer using a circularly linked list?

Answer: Circularly linked lists are particularly useful in scenarios where data elements need to be processed in a continuous loop or cycle. Examples include:

Round-robin scheduling: Managing tasks or processes in a cyclic manner, ensuring fair allocation of resources over time.

Buffer management: Implementing circular buffers or queues where elements wrap around once the end of the buffer is reached, useful in data streaming and real-time processing applications.

Q5: How can lists be used in database management systems?

Answer: Lists play a crucial role in database management systems for storing and managing collections of records or entries:

Indexing: Lists can serve as index structures, facilitating fast access to records based on indexed keys.

Sorting and querying: Lists enable efficient sorting and querying operations, essential for optimizing database queries and data retrieval processes.

Transaction management: Lists can be used to manage transaction logs or sequences of operations, ensuring data consistency and reliability in transactional processing.

3.11 REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*. MIT Press.

Goodrich, M. T., & Tamassia, R. (2012). *Data Structures and Algorithms in Java (6th ed.)*. Wiley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th ed.)*. Addison-Wesley.

Brass, P. (2008). *Advanced Data Structures*. Cambridge University Press.

Mehta, D., & Sahni, S. (2007). *Data Structures, Algorithms, & Applications in C++ (2nd ed.)*. Silicon Press.

Lafore, R. (2002). *Data Structures and Algorithms in Java (2nd ed.)*. Sams Publishing.

BLOCK II: STACKS, QUEUES AND TREES

UNIT – 4: STACKS

Structure

4.0 INTRODUCTION

In the realm of computer science and software engineering, stacks represent a pivotal concept deeply ingrained in the fabric of efficient data management and algorithm design. A stack operates on the principle of Last In, First Out (LIFO), where elements are added and removed from one end, known as the top. This characteristic makes stacks particularly suited for scenarios where strict ordering of operations is essential, such as function call management, expression evaluation, and backtracking algorithms. By adhering to the LIFO principle, stacks ensure that the most recent operation or data element processed is the first one to be reversed or retrieved, facilitating streamlined and predictable control flow in software systems.

The core operations on a stack—push, pop, and peek—form the cornerstone of its functionality. Pushing adds an element to the top of the stack, pop removes and returns the top element, and peek retrieves the top element without removing it. These operations are typically executed in constant time, O(1), regardless of the size of the stack, ensuring efficiency in both time and space. This efficiency is crucial in applications where rapid access to and manipulation of data is paramount, such as in real-time systems, interactive applications, and embedded computing environments.

Implementing stacks can be achieved using various underlying data structures, most commonly arrays and linked lists. Each approach offers distinct advantages: array-based stacks provide direct access to elements but are limited by fixed sizes, while linked list-based stacks offer dynamic memory management but may incur overhead due to pointer operations. Understanding these implementations and their trade-offs is essential for choosing the most suitable approach based on specific application requirements and constraints. Overall, stacks embody a foundational concept in computer science, driving innovation and efficiency across diverse fields by enabling structured and efficient data handling in software systems.

4.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand Stack Fundamentals: Gain a comprehensive understanding of the stack data structure, including its characteristics, operations, and the Last In, First Out (LIFO) principle.

Explore Stack Operations: Learn how to perform fundamental stack operations such as push, pop, and peek. Understand their functionalities, complexities, and applications in real-world scenarios.

Compare Implementation Methods: Compare and contrast different implementations of stacks using arrays and linked lists. Evaluate the advantages, disadvantages, and optimal use cases for each implementation approach.

Implement Multiple Stacks: Explore advanced stack concepts by learning how to implement and manage multiple stacks within a single array or memory block. Understand the benefits and challenges of dynamic stack management.
Analyze Efficiency Considerations: Evaluate the efficiency of stack operations and implementations, considering factors such as time complexity, space complexity, and practical considerations in software development.

4.2 INTRODUCTION TO STACKS

A stack is a fundamental data structure in computer science, known for its simplicity and versatility in managing data. It operates on the principle of Last In, First Out (LIFO), meaning that the last element added to the stack is the first one to be removed. This characteristic makes stacks ideal for scenarios where the order of operations must be strictly controlled.

Last In, First Out (LIFO) principle:
The Last In, First Out (LIFO) principle is a core concept in the stack data structure, which dictates the order in which elements are accessed and removed. According to this principle, the most

recently added element is the first one to be removed. This behavior is analogous to a stack of plates where you can only add or remove the top plate.

How LIFO Works

Push Operation: When an element is added to the stack, it is "pushed" onto the top of the stack. This element becomes the most recent addition and the first candidate for removal.

Pop Operation: When an element needs to be removed from the stack, it is "popped" from the top of the stack. Since only the top element can be removed, this ensures that the most recent addition is the first to be removed.

Illustrative Example

Consider a stack of books:

Initially, the stack is empty.

You place Book A on the stack (push operation). Now, Book A is at the top.

You then place Book B on the stack (push operation). Book B is now at the top, with Book A underneath it.

Next, you place Book C on the stack (push operation). Book C is at the top, with Book B and Book A below it in that order.

If you now remove a book from the stack (pop operation), Book C, the last one added, will be removed first. The stack now has Book B at the top. If you perform another pop operation, Book B will be removed next, leaving Book A as the topmost element.

Characteristics of Stacks

LIFO Principle: The most recent addition is the first to be removed, akin to a stack of plates where you can only take the top plate off.

Dynamic Size: Depending on the implementation, the size of the stack can grow or shrink dynamically as elements are added or removed.

Restricted Access: Elements can only be added (pushed) or removed (popped) from one end of the structure, referred to as the top of the stack.

Basic Operations

Push: Adds an element to the top of the stack.

Pop: Removes the element from the top of the stack.

Peek/Top: Returns the element at the top of the stack without removing it.

IsEmpty: Checks if the stack is empty.

IsFull: Checks if the stack has reached its capacity (relevant for array-based implementations).

Real-world Analogies

The stack data structure mirrors many real-world scenarios:

Plate Dispenser: Imagine a spring-loaded plate dispenser in a cafeteria. Plates are added on top, and the last plate added is the first to be taken off.

Browser History: When navigating web pages, the browser stores the visited pages in a stack. The back button removes (pops) the last visited page from the stack and displays it.

Applications of Stacks

Stacks are widely used in various applications across computer science and programming:

Expression Evaluation and Syntax Parsing: Stacks are used to evaluate arithmetic expressions, convert infix expressions to postfix, and check for balanced parentheses in expressions.

Function Call Management: In programming languages, the call stack keeps track of function calls, enabling proper return from functions and managing recursive calls.

Undo Mechanism: Applications like text editors use stacks to keep track of changes, allowing users to undo recent actions.

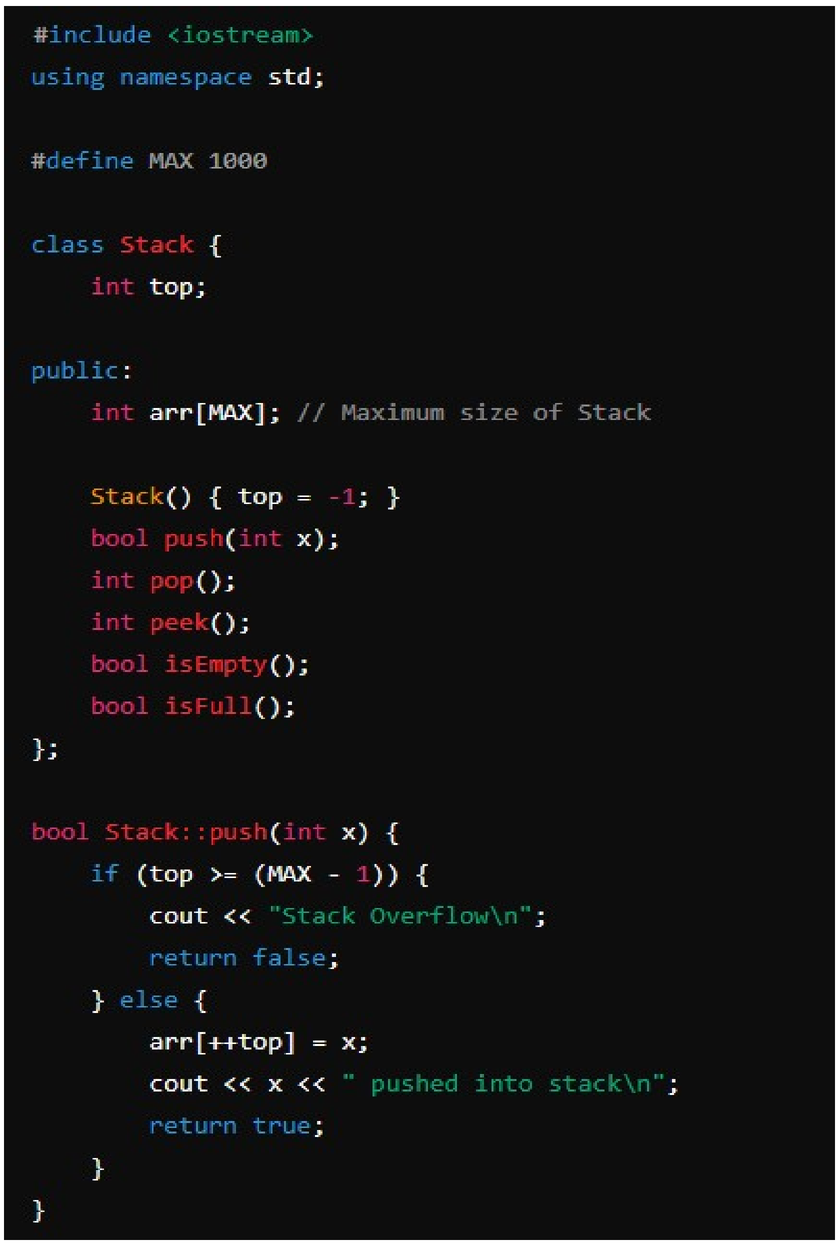
4.3 OPERATIONS ON STACKS (PUSH, POP, PEEK)

Push Operation

The push operation adds an element to the top of the stack.

Algorithm:

Check if the stack is full. If full, print an overflow message and exit.
If not full, increment the top index.
Add the element at the new top index.



```cpp
#include <iostream>
using namespace std;

#define MAX 1000

class Stack {
    int top;

public:
    int arr[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    bool isFull();
};

bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow\n";
        return false;
    } else {
        arr[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
```

Pop Operation

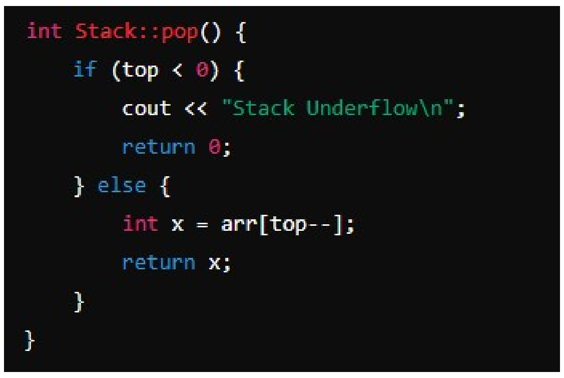
The pop operation removes the element from the top of the stack.

Algorithm:

Check if the stack is empty. If empty, print an underflow message and exit.

If not empty, return the element at the top index and decrement the top index.

C++ Implementation:



```cpp
int Stack::pop() {
    if (top < 0) {
        cout << "Stack Underflow\n";
        return 0;
    } else {
        int x = arr[top--];
        return x;
    }
}
```

Peek Operation

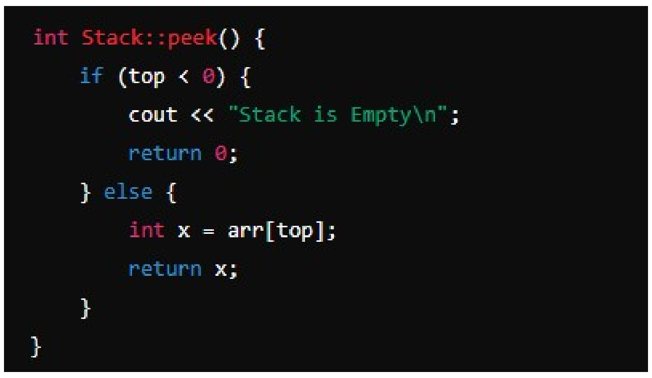
The peek operation returns the top element of the stack without removing it.

Algorithm:

Check if the stack is empty. If empty, print an empty stack message and exit.

If not empty, return the element at the top index.

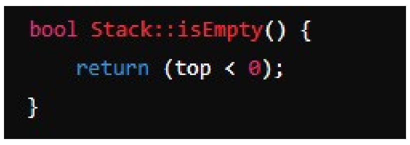
C++ Implementation:



```cpp
int Stack::peek() {
    if (top < 0) {
        cout << "Stack is Empty\n";
        return 0;
    } else {
        int x = arr[top];
        return x;
    }
}
```

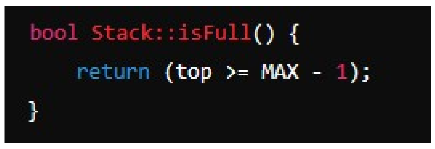
Utility Functions

isEmpty Function



```cpp
bool Stack::isEmpty() {
    return (top < 0);
}
```

Is Full Function



```
bool Stack::isFull() {
    return (top >= MAX - 1);
}
```

Complete Implementation

Here's the complete implementation combining all the above methods:

```
#include <iostream>
using namespace std;
#define MAX 1000
class Stack {
    int top;
public:
    int arr[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    bool isFull();
};
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow\n";
        return false;
    } else {
        arr[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
int Stack::pop() {
    if (top < 0) {
        cout << "Stack Underflow\n";
        return 0;
    } else {
        int x = arr[top--];
        return x;
    }
}
int Stack::peek() {
    if (top < 0) {
        cout << "Stack is Empty\n";
        return 0;
    } else {
        int x = arr[top];
        return x;
    }
}
bool Stack::isEmpty() {
    return (top < 0);
}
bool Stack::isFull() {
    return (top >= MAX - 1);
}
// Driver program to test above functions
int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";
    cout << "Top element is " << s.peek() << endl;
```

```
   cout << "Stack is empty: " << (s.isEmpty() ? "True" : "False")
<< endl;
   return 0;}
```

4.4 IMPLEMENTATION OF STACK USING ARRAYS

An array-based stack structure uses a fixed-size array to store stack elements. This implementation is straightforward and efficient for managing stack operations, which include pushing elements onto the stack, popping elements from the stack, and peeking at the top element.

Key Components of Array-based Stack

Array: A fixed-size array to store stack elements.

Top: An integer to keep track of the index of the top element in the stack. It is initialized to -1 to indicate that the stack is initially empty.

Capacity: A constant defining the maximum size of the stack.

Basic Operations

Push: Adds an element to the top of the stack.

Pop: Removes and returns the element from the top of the stack.

Peek: Returns the top element without removing it.

isEmpty: Checks if the stack is empty.

isFull: Checks if the stack is full.

C++ Implementation

Here is the complete implementation of an array-based stack structure in C++:

```
#include <iostream>
using namespace std;
#define MAX 1000 // Define the maximum size of the stack
```

```
class Stack {
    int top;
public:
    int arr[MAX]; // Array to store stack elements
    Stack() { top = -1; } // Constructor to initialize the stack
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    bool isFull();
};
// Function to add an element to the stack
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow\n";
        return false;
    } else {
        arr[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
// Function to remove an element from the stack
int Stack::pop() {
    if (top < 0) {
        cout << "Stack Underflow\n";
        return 0;
    } else {
        int x = arr[top--];
        return x;
    }
}
```

```
// Function to get the top element of the stack without removing it
int Stack::peek() {
  if (top < 0) {
    cout << "Stack is Empty\n";
    return 0; // or return an error code or throw an exception
  } else {
    int x = arr[top];
    return x;
  }
}
// Function to check if the stack is empty
bool Stack::isEmpty() {
  return (top < 0);
}
// Function to check if the stack is full
bool Stack::isFull() {
  return (top >= MAX - 1);
}
// Driver program to test above functions
int main() {
  Stack s;
  s.push(10);
  s.push(20);
  s.push(30);
  cout << s.pop() << " Popped from stack\n";
  cout << "Top element is " << s.peek() << endl;
  cout << "Stack is empty: " << (s.isEmpty() ? "True" : "False")
<< endl;
  cout << "Stack is full: " << (s.isFull() ? "True" : "False") <<
endl;
  return 0;
}
```

Advantages and Disadvantages of Array-based Stack

Advantages:

Simplicity: Easy to implement and understand.

Constant Time Operations: Push, pop, and peek operations have O(1) time complexity.

Memory Contiguity: Array-based stacks are stored in contiguous memory locations, which can lead to better cache performance.

Disadvantages:

Fixed Size: The size of the stack is fixed at compile time, limiting flexibility. If the stack is full, no more elements can be added without resizing.

Wasted Space: If the maximum size is much larger than the actual number of elements, memory may be wasted.

Stack Overflow: If too many elements are pushed onto the stack, it can cause stack overflow, which can crash the program.

Handling dynamic resizing

Handling dynamic resizing of a stack implemented using arrays allows the stack to grow or shrink as needed, avoiding the limitations of fixed-size arrays. Below is an enhanced implementation of a stack in C++ that supports dynamic resizing.

Dynamic Resizing Stack Implementation

Key Enhancements

Dynamic Array: Instead of a fixed-size array, use a dynamic array (pointer) that can be resized.

Resize Function: A function to resize the array when the stack is full or when it is sparsely populated to optimize memory usage.

Capacity Management: Maintain the current capacity of the array and resize it as necessary.

4.5 IMPLEMENTATION OF STACK USING LINKED LISTS

A stack can be implemented using a linked list to provide a dynamic, flexible stack structure that can grow and shrink as needed without the limitations of a fixed-size array.

Key Components of Linked List-based Stack

Node: A structure representing each element in the stack, containing the data and a pointer to the next node.

Top: A pointer to the top node of the stack.

Basic Operations

Push: Adds an element to the top of the stack.

Pop: Removes and returns the element from the top of the stack.

Peek: Returns the top element without removing it.

isEmpty: Checks if the stack is empty.

C++ Implementation

Here is the complete implementation of a stack using linked lists in C++:

```
#include <iostream>
using namespace std;
// Define the structure of a node
struct Node {
    int data;
    Node* next;
};
class Stack {
    Node* top; // Pointer to the top node
```

```
public:
    Stack() { top = nullptr; } // Constructor to initialize the stack
   void push(int x);
    int pop();
    int peek();
    bool isEmpty();
    void display(); // Utility function to display the stack elements
};
// Function to add an element to the stack
void Stack::push(int x) {
    Node* newNode = new Node(); // Create a new node
    if (!newNode) {
        cout << "Heap Overflow\n";
        return;
    }
    newNode->data = x;
    newNode->next = top;
    top = newNode;
    cout << x << " pushed into stack\n";
}
// Function to remove an element from the stack
int Stack::pop() {
    if (isEmpty()) {
        cout << "Stack Underflow\n";
        return 0; // or return an error code or throw an exception
    } else {
        Node* temp = top;
        top = top->next;
        int popped = temp->data;
        delete temp;
        return popped;
    }
```

```
}
// Function to get the top element of the stack without removing it
int Stack::peek() {
    if (!isEmpty()) {
        return top->data;
    } else {
        cout << "Stack is Empty\n";
        return 0; // or return an error code or throw an exception
    }
}
// Function to check if the stack is empty
bool Stack::isEmpty() {
    return top == nullptr;
}
// Utility function to display the stack elements
void Stack::display() {
    if (isEmpty()) {
        cout << "Stack is Empty\n";
    } else {
        Node* temp = top;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
}
// Driver program to test above functions
int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.display();
    cout << s.pop() << " Popped from stack\n";
    s.display();
    cout << "Top element is " << s.peek() << endl;
    cout << "Stack is empty: " << (s.isEmpty() ? "True" : "False")
<< endl;
    return 0;
}
```

Advantages of Linked List-based Stack

Dynamic Size: The stack can grow and shrink as needed, limited only by available memory.

No Wasted Space: Memory is allocated only when needed, avoiding the wasted space issue of fixed-size arrays.

No Overflow: Unlike array-based stacks, a linked list-based stack does not overflow unless the system runs out of memory.

Disadvantages

Memory Overhead: Each element requires additional memory for the pointer, which can be significant if the stack contains many elements.

Non-contiguous Memory: Elements are not stored in contiguous memory locations, which can lead to cache inefficiencies compared to array-based stacks.

4.6 ALGORITHMIC IMPLEMENTATION OF MULTIPLE STACKS

In computer science and software engineering, the stack is a fundamental data structure that follows the Last In, First Out (LIFO) principle. This means that the most recently added element is the first one to be removed. Stacks are widely used in various applications, including function call management, expression evaluation, and backtracking algorithms. However, in certain scenarios, a single stack is not sufficient to handle multiple sets of data independently. This is where the concept of multiple stacks comes into play.

Multiple stacks involve managing several stack data structures within a single array or memory block. This approach can optimize memory usage and improve the efficiency of algorithms that require simultaneous and independent stack operations. By leveraging multiple stacks, one can avoid the overhead of maintaining separate arrays for each stack, leading to more compact and manageable code.

There are two primary strategies for implementing multiple stacks:
Fixed Division: The array is divided into fixed-size sections, each allocated to a specific stack. This method is straightforward but lacks flexibility, as it cannot adjust the size of individual stacks dynamically.
Dynamic Division: The boundaries between stacks are adjusted dynamically based on the current usage of each stack. This method is more complex but offers greater flexibility and efficient utilization of memory.

In practical applications, multiple stacks are particularly useful in scenarios such as:

Memory Management: Managing multiple stack frames for different threads or processes in a concurrent computing environment.

Resource Allocation: Keeping track of resource usage and availability in systems that need to handle multiple independent tasks.

Algorithm Optimization: Implementing complex algorithms that require simultaneous traversal or manipulation of multiple data sets.

Applications and scenarios where multiple stacks are useful

Multiple stacks are highly beneficial in various applications and scenarios where independent management of multiple sets of data or operations is required. Here are some key areas where multiple stacks find significant utility:

Applications and Scenarios

Expression Evaluation and Parsing: In compilers and interpreters, multiple stacks are used to handle nested expressions, function calls, and operator precedence. Each stack can manage operands, operators, and function call contexts independently, ensuring correct evaluation and parsing of complex expressions.

Function Call Management: In programming languages and runtime environments, multiple stacks are employed to manage function calls and local variables. Each stack corresponds to a different function or subroutine, ensuring proper execution flow and efficient memory allocation for local variables.

Backtracking Algorithms: Algorithms like depth-first search (DFS) and recursive backtracking often require multiple stacks to manage state transitions and backtracking paths independently. Each stack maintains a different path or state sequence, facilitating efficient exploration of solution spaces.

Memory Management in Operating Systems: Operating systems use multiple stacks to manage execution contexts, interrupts, and system calls for different processes or threads. Each stack provides isolated memory space and execution flow, ensuring security and efficient resource utilization.

Undo/Redo Mechanisms in Applications: Applications with undo/redo functionalities often employ multiple stacks to store previous states or actions. Each stack represents a history of user actions or modifications, enabling seamless navigation and recovery of application states.

Simulation and Modelling: Simulation software and modelling tools utilize multiple stacks to manage different simulation scenarios or model configurations. Each stack stores parameters, states, or simulation steps independently, facilitating parallel or sequential simulation runs.

Resource Allocation and Management: Systems managing resources such as memory, network connections, or database transactions benefit from multiple stacks to allocate and track resource usage efficiently. Each stack handles resource requests or transactions independently, ensuring optimal resource utilization and performance.

Algorithmic Optimization: Complex algorithms, including graph traversal, dynamic programming, and state machines, often utilize multiple stacks to manage different states, paths, or data structures. Each stack supports efficient traversal or manipulation of algorithmic data structures, enhancing algorithm performance and scalability.

Advantages of Multiple Stacks

Independence: Each stack operates independently, allowing separate handling of data sets or operations without interference.

Efficiency: Multiple stacks optimize memory usage and improve algorithm performance by isolating and managing distinct data sets or operations efficiently.

Flexibility: Dynamic adjustment of stack boundaries (in dynamic division) provides flexibility in managing varying sizes and requirements of individual stacks.

Simplicity: While providing more robust and efficient data management than a single stack, multiple stacks can be managed with the same ease of a single stack

4.7 CONCLUSION

In conclusion, the study of stacks reveals their foundational role in computer science and software engineering. Through the exploration of their Last In, First Out (LIFO) principle and essential operations like push, pop, and peek, we have seen how stacks efficiently manage data with predictable ordering. The implementations using arrays and linked lists underscore their versatility in accommodating different needs—from fixed-size memory management to dynamic and flexible data structures.

Moreover, the concept of multiple stacks within a single array expands the utility of stacks, demonstrating their adaptability in handling complex scenarios where independent management of multiple data sets is required. Efficiency considerations, such as time complexity for operations and space management, highlight the trade-offs between array-based and linked list-based implementations, crucial for optimizing performance in diverse applications.

Looking ahead, the practical applications of stacks—from parsing expressions in compilers to managing function calls in programming languages—underscore their indispensable role in modern computing. By mastering stack operations and understanding their implementations, developers can leverage stacks effectively in algorithm design, system programming, and various software applications, ensuring robust and efficient data management.

In conclusion, stacks remain a cornerstone of computational efficiency and structured data handling, continuing to inspire innovation and optimal solutions across a wide range of technological domains.

4.8 QUESTIONS AND ANSWERS

Q1: What is the LIFO principle, and why is it important in stacks?

Answer: The LIFO (Last In, First Out) principle states that the last element inserted into a stack is the first one to be removed. It ensures that operations are processed in reverse order of their insertion, making stacks ideal for scenarios requiring strict ordering and efficient data management.

Q2: What are the main operations performed on a stack, and how do they work?

Answer: The main operations on a stack are:

Push: Adds an element to the top of the stack.

Pop: Removes and returns the top element from the stack.

Peek (or Top): Returns the top element without removing it. These operations are typically executed in constant time, O(1), making stacks efficient for managing data with predictable access patterns.

Q3: What are the advantages of using an array-based implementation of stacks over a linked list-based implementation?

Answer: Array-based stacks offer direct access to elements using indices, which can be faster in scenarios where random access is important. They also use contiguous memory, which may result in better cache performance. However, they are limited by a fixed size and require resizing if the stack grows beyond its initial capacity.

Q4: How can multiple stacks be implemented using a single array, and what are the benefits of this approach?
Answer: Multiple stacks can be managed within a single array by partitioning the array into sections allocated to each stack. This approach optimizes memory usage by allowing stacks to dynamically expand and contract within the same memory block, enhancing flexibility and reducing memory fragmentation.

Q5: What are some practical applications of stacks in software development and computer science?
Answer: Stacks are widely used in expression evaluation, function call management, backtracking algorithms, memory management in operating systems, and parsing techniques in compilers. They play a critical role in managing program execution flow and optimizing memory utilization in various computational tasks.

4.9 REFERENCES

Academic Journals: Search databases like Google Scholar, IEEE Xplore, or ACM Digital Library for peer-reviewed articles on stack data structures, algorithms, and their applications.

Textbooks: Explore textbooks on data structures and algorithms, such as "Introduction to Algorithms" by Thomas H. Cormen et al., "Data Structures and Algorithms in Java" by Robert Lafore, or "Data Structures Using C++" by D.S. Malik.

Online Learning Platforms: Websites like Coursera, edX, or Khan Academy offer courses on data structures that cover stacks, along with recommended readings and references.

Official Documentation: Refer to official documentation from programming language providers (e.g., C++, Java) or system documentation (e.g., Linux kernel) for implementation details and best practices.

Research Papers: Look for relevant research papers presented at conferences like ACM SIGMOD, IEEE INFOCOM, or USENIX Symposium on Operating Systems Design and Implementation (OSDI).

UNIT – 5: QUEUES

Structure

5.0 INTRODUCTION

In computer science and software engineering, queues are fundamental data structures that facilitate the orderly processing of data based on the First-In-First-Out (FIFO) principle. They play a crucial role in various applications where data needs to be managed and processed sequentially. Queues ensure that the first element added to the queue is the first one to be removed, making them essential in scenarios ranging from operating system task scheduling to network packet management and beyond.

This section explores the foundational concepts, operations, and implementations of queues, covering their diverse forms such as linear queues using arrays, linked lists, and circular structures. Additionally, it delves into specialized variants like priority queues, which prioritize elements based on specific criteria, and double-ended queues (deques), offering flexibility with operations at both ends. Understanding these structures equips developers with powerful tools to optimize data handling and application performance.

The subsequent sections will delve into each aspect of queues, detailing their operations, implementations in both arrays and linked lists, specialized forms like circular queues, priority queues for managing prioritized tasks, and versatile double-ended queues. By examining these topics comprehensively, this exploration aims to provide a thorough understanding of how queues function, their practical applications, and their role in efficient data management strategies.

5.1 OBJECTIVES

After completing this unit, you will be able to understand,
Understand Queue Basics: Define queues and grasp the FIFO principle essential for orderly data processing.

Master Queue Operations: Explore insertion, deletion, and traversal operations crucial for queue management.

Implement Using Arrays and Linked Lists: Implement queues using both arrays and linked lists, understanding their advantages and limitations.

Explore Circular Queue Mechanics: Study circular queues, including their implementation and advantages in managing continuous data streams.

Examine Specialized Queue Types: Investigate priority queues and double-ended queues (deques), exploring their unique applications and operational efficiency.

5.2 QUEUE

Queues are fundamental data structures in computer science that adhere to the First In, First Out (FIFO) principle. Similar to real-life queues, such as waiting in line at a ticket counter, queues manage elements in the order they are added. The structure ensures that the oldest element, added first, is the first to be removed or processed. This characteristic makes queues ideal for scenarios where tasks must be handled in the order they arrive.

In programming, queues are crucial for managing tasks that require sequential processing, such as job scheduling, printer spooling, and asynchronous data transfer. Operations on queues typically include adding an element to the rear (enqueue), removing an element from the front (dequeue), and peeking at the front element without removing it. These operations enable efficient data handling and ensure that processes are executed in a fair and orderly manner.

Queues can be implemented using various underlying data structures, including arrays and linked lists, each offering distinct advantages based on specific application requirements. Understanding queues and their implementations is foundational for designing efficient algorithms and systems that rely on orderly task execution and data management.

FIFO (First In, First Out) principle

The FIFO principle dictates that the oldest elements in a queue are processed or removed first, maintaining the sequential order of arrival. This principle is fundamental to how queues operate and distinguishes them from other data structures like stacks, which follow the Last In, First Out (LIFO) principle. FIFO ensures fairness in task scheduling and data processing by handling tasks in the order they are queued.

Queue Syntax in C++

Include Header:

#include <queue>

Declare Queue: To declare a queue of integers:

std::queue<int> myQueue;

Operations:

Push (Enqueue): Adds an element to the back of the queue.

myQueue.push(value);

Pop (Dequeue): Removes the element at the front of the queue.

myQueue.pop();

Front: Accesses the element at the front of the queue.

int frontElement = myQueue.front();

Back: Accesses the element at the back of the queue.

int backElement = myQueue.back();

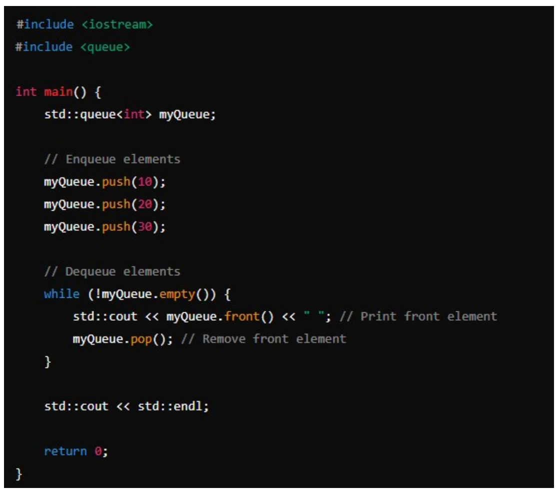
Size: Returns the number of elements in the queue.

int size = myQueue.size();

Empty: Checks if the queue is empty.

bool isEmpty = myQueue.empty();

Example:



```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> myQueue;

    // Enqueue elements
    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);

    // Dequeue elements
    while (!myQueue.empty()) {
        std::cout << myQueue.front() << " "; // Print front element
        myQueue.pop(); // Remove front element
    }

    std::cout << std::endl;

    return 0;
}
```

Explanation:

std::queue<int>: Declares a queue of integers.

myQueue.push(value);: Adds value to the back of the queue.

myQueue.pop();: Removes the front element from the queue.

myQueue.front(); and myQueue.back();: Accesses the front and back elements of the queue, respectively.

myQueue.size();: Returns the number of elements in the queue.

myQueue.empty();: Checks if the queue is empty.

Real-World Analogies and Examples

Queues have numerous real-world analogies and applications, reflecting their ubiquitous nature in everyday scenarios:

Waiting Lines: Queues resemble physical lines at ticket counters, checkout lanes in supermarkets, or queues at amusement parks, where individuals wait in order to be served or processed.

Print Spooling: Printers use queues to manage print jobs, ensuring that documents are printed in the order they were sent to the printer.

Job Scheduling: Operating systems manage processes using queues to prioritize tasks and allocate resources based on their arrival and priority.

Network Data Packet Processing: Network routers and switches use queues to buffer and forward data packets based on their arrival time and network conditions.

Real-world analogies and examples (e.g., waiting lines)

Ticket Counter at a Movie Theater: At a movie theater, patrons line up to purchase tickets. The ticket counter operates as a queue where customers are served in the order they arrived. This ensures efficient ticket sales and customer satisfaction.

Call Center Customer Service: Call centers manage customer queries and support requests using queues. Calls are queued based on their arrival, and customer service representatives handle them sequentially. This approach ensures that all customer inquiries are addressed promptly and fairly.

Print Spooling: Printers use queues to manage print jobs submitted by multiple users. Each print job is queued based on its submission time, and the printer processes them in the order they were received. This ensures orderly printing and prevents job conflicts.

Traffic at Intersection Signals: Traffic signals manage vehicle movement at intersections using queues. Vehicles waiting at a red-light form queues in each lane, and when the light turns green, vehicles are allowed to proceed based on their position in the queue. This systematic approach helps in managing traffic flow efficiently.

Job Scheduling in Operating Systems: Operating systems use queues to manage processes and tasks. Jobs are queued based on their priority and resource requirements, and the operating system schedules them for execution accordingly. This ensures efficient utilization of system resources and optimal performance.

Buffering in Data Communication: Network devices use queues to buffer data packets during transmission. Data packets are queued based on network conditions and bandwidth availability, ensuring smooth and efficient data transfer without packet loss.

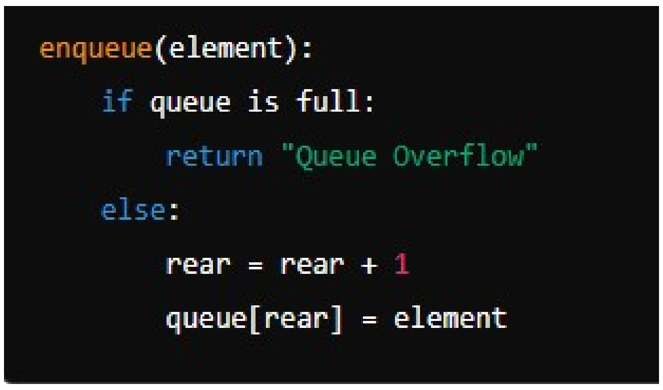
5.3 OPERATIONS ON QUEUES

Operations on queues are fundamental to their functionality and efficient management of data. The main operations typically supported by a queue data structure include:

Operations on Queues

Enqueue (Insertion): Adds an element to the rear (end) of the queue. It Increases the queue's size and stores new data for processing.

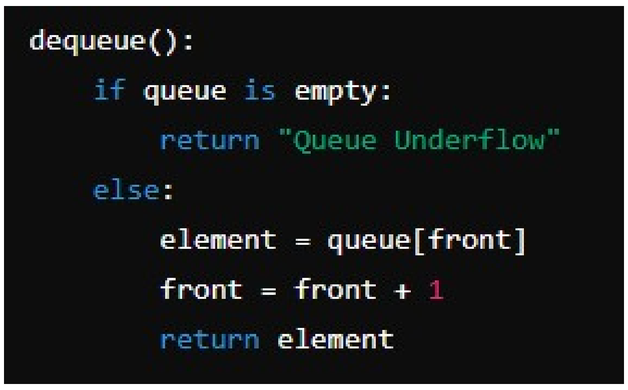
Algorithm:



```
enqueue(element):
    if queue is full:
        return "Queue Overflow"
    else:
        rear = rear + 1
        queue[rear] = element
```

Dequeue (Deletion): Removes an element from the front (beginning) of the queue. It Retrieves and processes the oldest data entered into the queue.

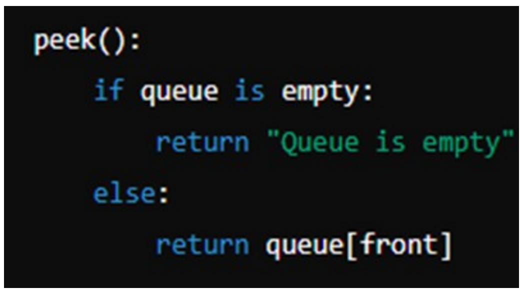
Algorithm:



```
dequeue():
    if queue is empty:
        return "Queue Underflow"
    else:
        element = queue[front]
        front = front + 1
        return element
```

Peek (Front): Retrieves the element at the front of the queue without removing it. It Allows inspection of the next element to be dequeued.

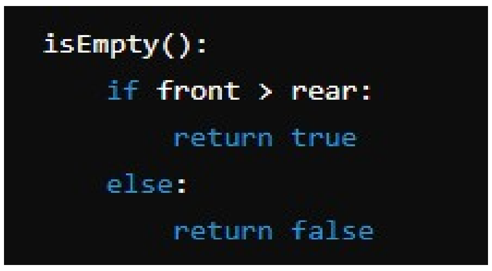
Algorithm:



```
peek():
    if queue is empty:
        return "Queue is empty"
    else:
        return queue[front]
```

isEmpty: It Checks if the queue is empty. It Determines whether there are elements present in the queue for processing.

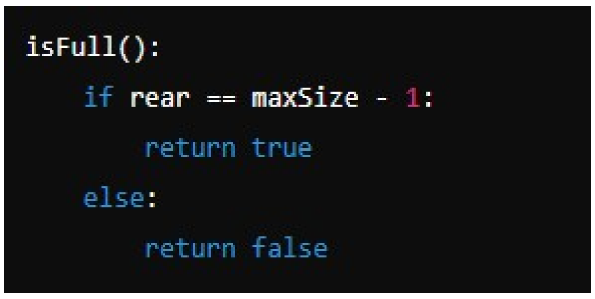
Algorithm:



```
isEmpty():
    if front > rear:
        return true
    else:
        return false
```

isFull: Checks if the queue is full (not always applicable for dynamic-sized implementations). Determines if additional elements can be added to the queue without causing overflow.

Algorithm:



```
isFull():
    if rear == maxSize - 1:
        return true
    else:
        return false
```

5.4 IMPLEMENTATION OF QUEUE USING ARRAYS

An array-based representation of a queue involves using a fixed-size array to store elements and maintaining pointers (or indices) to track the front and rear of the queue. Here's a detailed explanation and implementation of an array-based queue in C++:

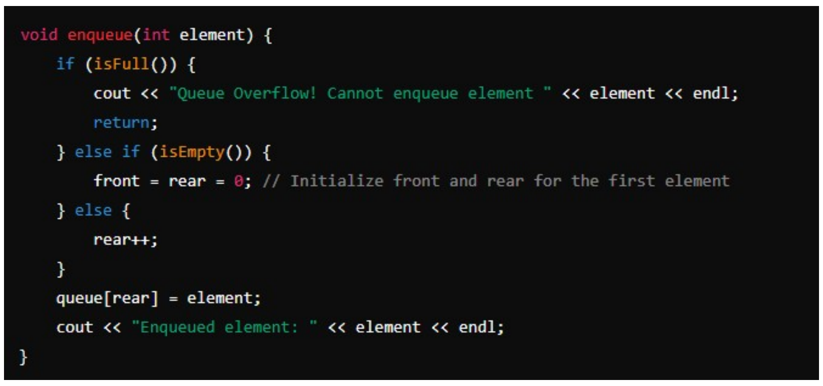
Array-Based Representation of a Queue

In this implementation, we'll define a queue class that uses an array to store elements. We'll include operations for enqueue (adding elements), dequeue (removing elements), peek (viewing the front element), and utility functions to check if the queue is empty or full.

Array-based representation of a queue

1. Enqueue (Insertion): Adds an element to the rear (end) of the queue.

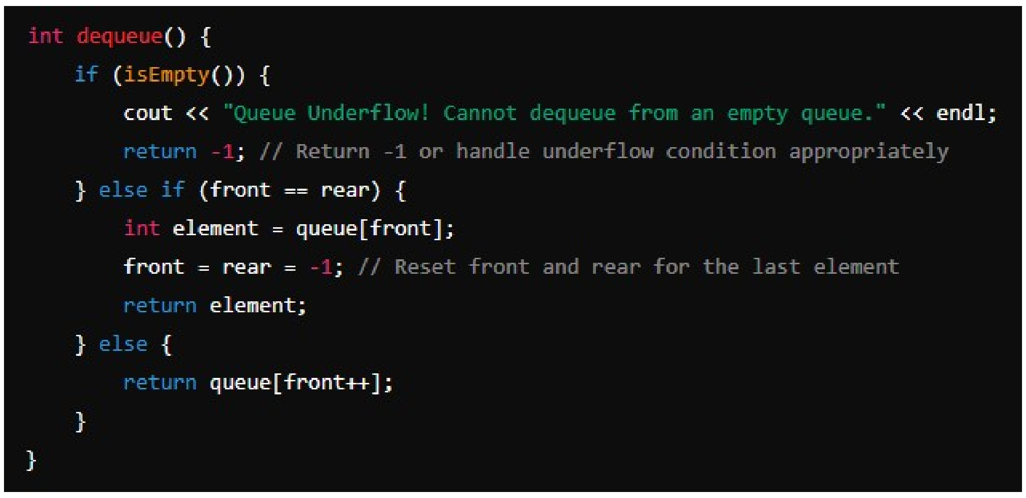
Algorithm:



```
void enqueue(int element) {
    if (isFull()) {
        cout << "Queue Overflow! Cannot enqueue element " << element << endl;
        return;
    } else if (isEmpty()) {
        front = rear = 0; // Initialize front and rear for the first element
    } else {
        rear++;
    }
    queue[rear] = element;
    cout << "Enqueued element: " << element << endl;
}
```

2. Dequeue (Deletion): Removes an element from the front (beginning) of the queue.

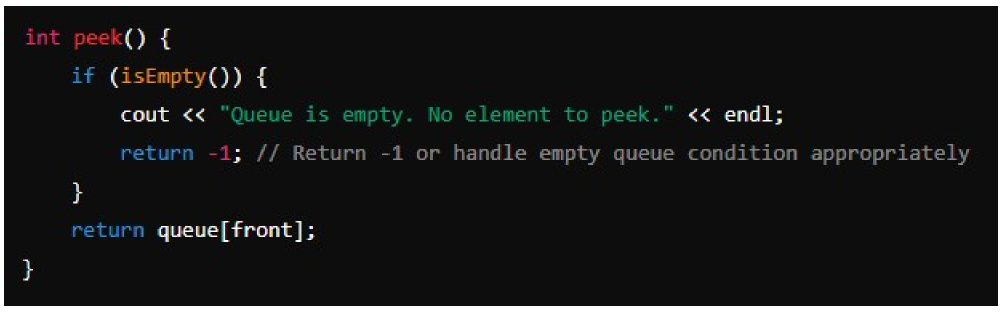
Algorithm:



```cpp
int dequeue() {
    if (isEmpty()) {
        cout << "Queue Underflow! Cannot dequeue from an empty queue." << endl;
        return -1; // Return -1 or handle underflow condition appropriately
    } else if (front == rear) {
        int element = queue[front];
        front = rear = -1; // Reset front and rear for the last element
        return element;
    } else {
        return queue[front++];
    }
}
```

3. Peek (Front): Retrieves the element at the front of the queue without removing it.

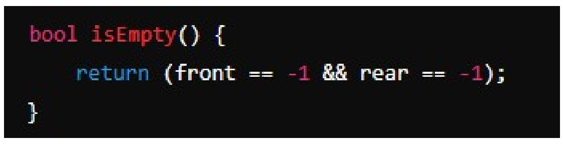
Algorithm:



```cpp
int peek() {
    if (isEmpty()) {
        cout << "Queue is empty. No element to peek." << endl;
        return -1; // Return -1 or handle empty queue condition appropriately
    }
    return queue[front];
}
```

4. IsEmpty: Checks if the queue is empty.

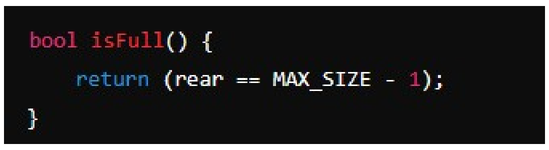
Algorithm:



```cpp
bool isEmpty() {
    return (front == -1 && rear == -1);
}
```

5. IsFull: Checks if the queue is full (only applicable for array implementation).

Algorithm:



```cpp
bool isFull() {
    return (rear == MAX_SIZE - 1);
}
```

Full Implementation Example in C++

Combining the above operations, here's a complete example for both array and linked list implementations of a queue:

Array-Based Queue Implementation:

```
#include <iostream>
using namespace std;
#define MAX_SIZE 100
class Queue {
private:
   int queue[MAX_SIZE];
   int front, rear;
public:
   Queue() {
      front = -1;
      rear = -1;
   }
   bool isEmpty() {
      return (front == -1 && rear == -1);
   }

   bool isFull() {
      return (rear == MAX_SIZE - 1);
   }
   void enqueue(int element) {
      if (isFull()) {
         cout << "Queue Overflow! Cannot enqueue element " <<
element << endl;
         return;
      } else if (isEmpty()) {
         front = rear = 0;
      } else {
            rear++;
        }
        queue[rear] = element;
        cout << "Enqueued element: " << element << endl;
    }
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! Cannot dequeue from an
empty queue." << endl;
            return -1;
        } else if (front == rear) {
            int element = queue[front];
            front = rear = -1;
            return element;
        } else {
            return queue[front++];
        }
    }
    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty. No element to peek." << endl;
            return -1;
        }
        return queue[front];
    }
};
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    cout << "Front element: " << q.peek() << endl;
    cout << "Dequeued element: " << q.dequeue() << endl;
    cout << "Front element after dequeue: " << q.peek() << endl;
    return 0;
}
```

Pros and cons of array-based implementation

Pros:

Simple and Easy to Implement: Array-based queues are straightforward and easy to understand, making them suitable for beginners and educational purposes.

Constant Time Complexity: Enqueue and dequeue operations have O(1) time complexity, ensuring fast execution for queue operations.

Cache-Friendly: Arrays provide contiguous memory allocation, which is more cache-friendly and can lead to better performance in terms of memory access speed.

Predictable Memory Usage: The memory usage is fixed and known in advance, which can be an advantage in systems with limited or predictable memory resources.

Direct Access: Elements in an array can be accessed directly via indices, which can be beneficial for certain operations or optimizations.

Cons:

Fixed Size: The size of the array must be defined at the time of queue creation. This fixed size can lead to inefficiency if the queue size is either too small (leading to overflow) or too large (leading to wasted memory).

No Dynamic Resizing: Without additional logic for dynamic resizing, the array cannot grow or shrink based on the actual number of elements, which can be limiting in scenarios with varying data sizes.

Overflow and Underflow: Array-based queues are prone to overflow if the queue is full and an additional element is enqueued. Similarly, if all elements are dequeued, the queue becomes empty and underflow conditions must be handled.
Circular Queue Complexity: To efficiently use space in an array-based queue, a circular queue implementation is often used. This adds complexity to the implementation, especially in managing the wrap-around of front and rear indices.

Wasted Space: If the queue is not used to its full capacity, there can be wasted space in the array, leading to inefficient memory usage.

Reallocation Overhead: In implementations that handle dynamic resizing, the reallocation process (copying elements to a new, larger array) can be time-consuming and add overhead.

5.5 CONCLUSION

In summary, queues represent a fundamental data structure in computer science, crucial for managing data in a First-In-First-Out (FIFO) manner. Throughout this section, we have explored the foundational principles of queues, emphasizing their role in maintaining order and facilitating efficient data processing. Operations such as insertion, deletion, and traversal have been examined, illustrating how queues enable systematic handling of data items based on their arrival sequence.

Implementation-wise, we have investigated queue implementations using arrays and linked lists. Arrays offer direct access and simplicity but require careful management of dynamic resizing. Linked lists provide flexibility in memory management and support dynamic operations, making them suitable for scenarios requiring frequent insertions and deletions.

Additionally, specialized forms like circular queues, priority queues, and double-ended queues (deques) have been explored. Circular queues optimize memory usage and support continuous data processing, while priority queues prioritize elements based on predefined criteria. Deques provide flexibility with operations at both ends, catering to applications that demand versatile data handling capabilities.

In conclusion, mastering the concepts and implementations of queues equips developers with essential tools for designing efficient algorithms and robust software systems. By understanding the principles behind queues and their practical applications, developers can leverage these structures effectively to enhance system performance, manage data workflows, and ensure reliable data processing in various computational environments.

5.6 QUESTIONS AND ANSWERS

1. What is the primary principle that queues follow?

Answer: Queues follow the First-In-First-Out (FIFO) principle, where the first element inserted into the queue is the first one to be removed.

2. How is a queue implemented using arrays different from using linked lists?

Answer:

Arrays: Queues implemented using arrays offer direct access to elements and require resizing when the array becomes full, which can be inefficient for large queues.

Linked Lists: Queues implemented using linked lists offer flexibility in dynamic memory allocation and efficient insertion and deletion operations but may have higher memory overhead due to storing pointers.

3. What is the purpose of implementing a circular queue?

Answer: Circular queues are implemented to efficiently manage continuous data streams or buffer scenarios. They utilize a circular array structure where elements wrap around when the end of the array is reached, optimizing memory usage and enabling constant-time operations for both insertion and deletion.

4. How does a priority queue differ from a regular queue?

Answer:

Regular Queue: A regular queue follows the FIFO principle, where elements are processed in the order they are added.

Priority Queue: A priority queue allows elements to be inserted with a priority and processed in order of priority rather than the order of insertion. Higher priority elements are processed before lower priority ones, ensuring that the most urgent tasks or elements are handled first.

5. What are the advantages of using a deque (double-ended queue) over a regular queue?

Answer:

Flexibility: Deques allow elements to be added or removed from both the front and the back, offering greater flexibility in data management compared to queues, which only support operations at one end.

Efficiency: Operations such as insertion and deletion at both ends of a deque are typically efficient, often with constant time complexity, making deques suitable for scenarios requiring dynamic data processing from both directions.

5.7 REFERENCES

Books:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

"Data Structures and Algorithms in Java" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser.

"Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss.

Documentation:

C++ Standard Library documentation for std::queue, std::deque, and related containers.

UNIT – 6: LINKED LIST IMPLEMENTATION

Structure

6.0 INTRODUCTION

Queues are pivotal data structures in computer science, designed to manage elements in a First-In-First-Out (FIFO) manner. This section delves into various aspects of queues, exploring their implementations and specialized variants to cater to diverse application needs. From basic linear structures to advanced forms like circular queues and priority queues, understanding these concepts equips developers with powerful tools for efficient data management and algorithm design.

We begin by examining the implementation of queues using linked lists, highlighting the flexibility they offer in dynamic memory allocation and operations. Moving forward, we explore circular queues, which optimize space utilization and streamline continuous data processing scenarios. Priority queues come next, providing mechanisms to prioritize elements based on predefined criteria, essential for real-time systems and task scheduling. Lastly, double-ended queues (deques) are investigated, showcasing their versatility in supporting operations from both ends, enabling sophisticated data handling capabilities.

This exploration aims to provide a comprehensive understanding of queue data structures, their implementations, and practical applications. By delving into each variant's operational nuances and performance considerations, this section aims to empower developers with the knowledge needed to leverage queues effectively in software development and system optimization.

6.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand Queue Fundamentals: Define queues and their fundamental characteristics, focusing on the FIFO (First-In-First-Out) principle.

Explore Linked List Implementation: Implement queues using linked lists, emphasizing dynamic memory allocation and efficient insertion/deletion operations.

Study Circular Queue Mechanics: Investigate circular queues, including their implementation details and advantages in managing continuous data streams.

Examine Priority Queue Operations: Understand priority queues and their operations, prioritizing elements based on specified criteria for optimal task management.

Master Double-ended Queue Functionality: Explore double-ended queues (deques), their implementations, and operations allowing insertion and deletion from both ends for flexible data handling.

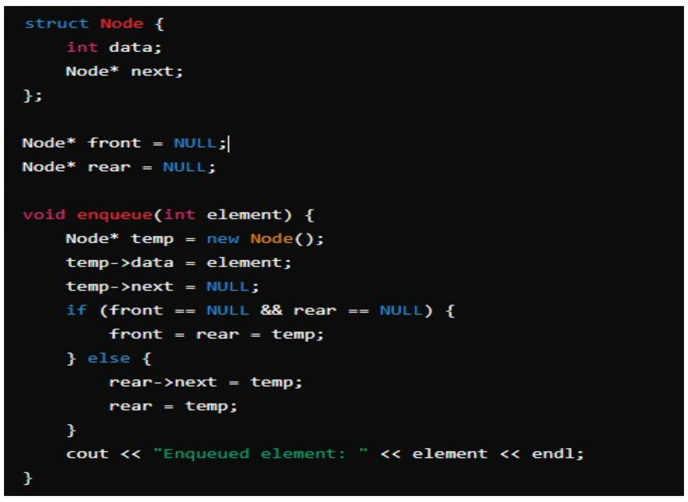
6.2 IMPLEMENTATION OF QUEUE USING LINKED LISTS

Implementing a queue using linked lists involves dynamically creating nodes to store data and linking them together in a sequence. Each node contains a data part and a pointer (or reference) to the next node in the sequence. The queue maintains two pointers: front and rear. The front pointer points to the first node in the queue, and the rear pointer points to the last node. When an element is enqueued (added) to the queue, a new node is created and linked to the end of the list, and the rear pointer is updated to point to this new node. When an element is dequeued (removed) from the queue, the node at the front is removed, and the front pointer is updated to the next node in the list. If the queue becomes empty, both front and rear pointers are set to null. This implementation allows the queue to dynamically adjust its size, avoiding the fixed size limitations of array-based implementations and efficiently handling memory usage.

1. Enqueue (Insertion)

Description: Adds an element to the rear (end) of the queue.

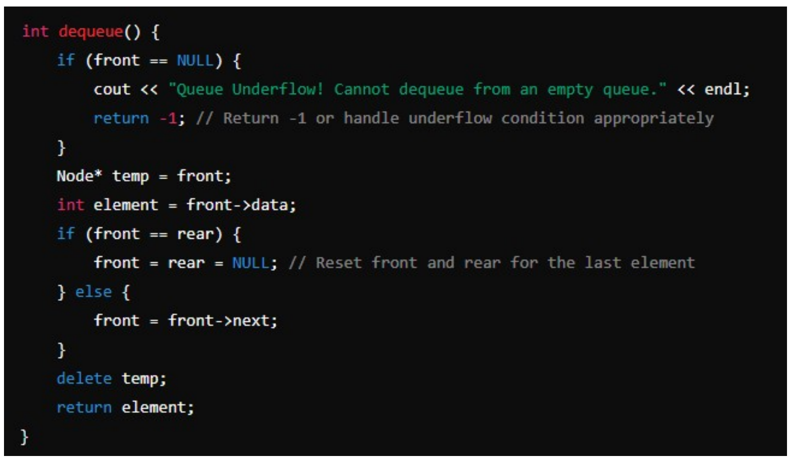
Algorithm:


```
struct Node {
    int data;
    Node* next;
};

Node* front = NULL;
Node* rear = NULL;

void enqueue(int element) {
    Node* temp = new Node();
    temp->data = element;
    temp->next = NULL;
    if (front == NULL && rear == NULL) {
        front = rear = temp;
    } else {
        rear->next = temp;
        rear = temp;
    }
    cout << "Enqueued element: " << element << endl;
}
```

2. Dequeue (Deletion)

Description: Removes an element from the front (beginning) of the queue.

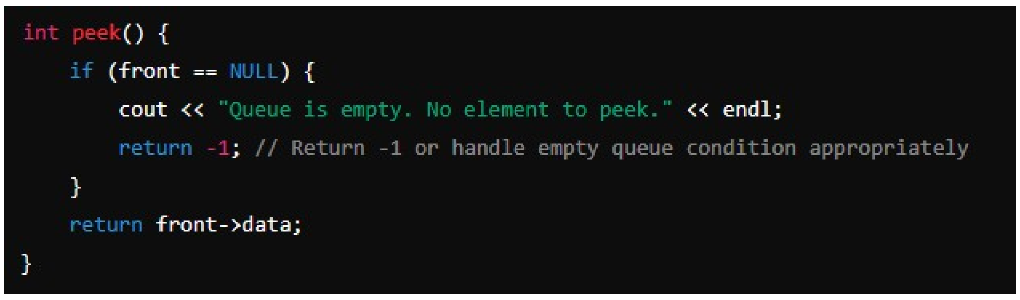
Algorithm:


```
int dequeue() {
    if (front == NULL) {
        cout << "Queue Underflow! Cannot dequeue from an empty queue." << endl;
        return -1; // Return -1 or handle underflow condition appropriately
    }
    Node* temp = front;
    int element = front->data;
    if (front == rear) {
        front = rear = NULL; // Reset front and rear for the last element
    } else {
        front = front->next;
    }
    delete temp;
    return element;
}
```

3. Peek (Front)

Description: Retrieves the element at the front of the queue without removing it.

Algorithm:

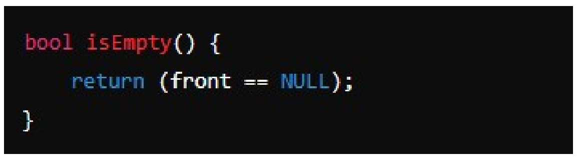


```
int peek() {
    if (front == NULL) {
        cout << "Queue is empty. No element to peek." << endl;
        return -1; // Return -1 or handle empty queue condition appropriately
    }
    return front->data;
}
```

4. IsEmpty

Description: Checks if the queue is empty.

Algorithm:



```
bool isEmpty() {
    return (front == NULL);
}
```

Linked List Based Queue Implementation

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class Queue {
private:
    Node* front;
    Node* rear;
public:
    Queue() {
        front = NULL;
        rear = NULL;
    }
    bool isEmpty() {
        return (front == NULL);
    }
    void enqueue(int element) {
        Node* temp = new Node();
        temp->data = element;
        temp->next = NULL;
        if (front == NULL && rear == NULL) {
            front = rear = temp;
        } else {
            rear->next = temp;
            rear = temp;
        }
        cout << "Enqueued element: " << element << endl;
    }
    int dequeue() {
        if (front == NULL) {
            cout << "Queue Underflow! Cannot dequeue from an
empty queue." << endl;
            return -1;
        }
        Node* temp = front;
        int element = front->data;
        if (front == rear) {
            front = rear = NULL;
        } else {
            front = front->next;
        }
        delete temp;
        return element;
    }
    int peek() {
        if (front == NULL) {
            cout << "Queue is empty. No element to peek." << endl;
            return -1;
        }
        return front->data;
    }
};
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    cout << "Front element: " << q.peek() << endl;
    cout << "Dequeued element: " << q.dequeue() << endl;
    cout << "Front element after dequeue: " << q.peek() << endl;
    return 0;
}
```

Advantages of linked list implementation

Dynamic Size: Linked lists provide a dynamic size, meaning the queue can grow or shrink as needed without predefining a maximum size. This is particularly useful when the maximum number of elements is unknown or varies significantly.

Efficient Memory Utilization: Memory is allocated only as needed. Unlike array-based implementations that may have unused space, linked lists do not waste memory.

No Need for Resizing: Since linked lists grow dynamically, there is no need for complex resizing operations or handling the overhead associated with array reallocation.

No Overflow: As long as there is available memory, a linked list-based queue will not overflow. This is a significant advantage over fixed-size array queues where overflow can occur when the capacity is exceeded.

Ease of Insertions and Deletions: Inserting (enqueue) and deleting (dequeue) elements are straightforward operations in a linked list, with both operations being O(1). There is no need to shift elements as in array-based implementations.

6.3 CIRCULAR QUEUE IMPLEMENTATION

A circular queue is a linear data structure that overcomes the limitations of a standard linear queue by treating the queue as a circular buffer. Unlike a linear queue where the end of the queue is fixed and adding more elements requires shifting or resizing, a circular queue allows for efficient reuse of space by connecting the end of the queue back to the front, forming a circular structure. This means that once the end of the queue is reached, the next element is inserted at the beginning of the array, provided there is free space (i.e., positions that have been dequeued).

Key Features

Circular Nature: The queue's end connects back to the beginning, enabling efficient use of space.

Fixed Size: Like an array-based queue, the size of the circular queue is fixed, but it utilizes the available space more efficiently.

Two Pointers: Maintains two pointers:

Front: Points to the first element in the queue.

Rear: Points to the last element in the queue.

Full and Empty Conditions: Specific conditions determine whether the queue is full or empty:

Empty Queue: When front and rear are both -1 or when front is equal to rear + 1.

Full Queue: When the position next to rear is the front (i.e., (rear + 1) % size == front).

Benefits

Efficient Space Utilization: Eliminates the problem of unused space in a standard array-based queue.

Fixed Size Management: Useful in scenarios where a fixed buffer size is required, such as in circular buffers for streaming data.

Implementation using arrays (circular array)

implementation of a circular queue using arrays in C++. This implementation covers different operations (enqueue, dequeue, peek, and display) with various algorithms to ensure the circular nature of the queue is properly handled.

Circular Queue Implementation Using Arrays

```
#include <iostream>
using namespace std;
class CircularQueue {
private:
   int *queue;
   int front, rear, size;
public:
    CircularQueue(int s) {
        size = s;
        queue = new int[size];
        front = rear = -1;
    }
    // Function to check if the queue is full
    bool isFull() {
        return (front == 0 && rear == size - 1) || (rear == front - 1);
    }
    // Function to check if the queue is empty
    bool isEmpty() {
        return front == -1;
    }
    // Function to add an element to the queue (enqueue operation)
    void enqueue(int element) {
        if (isFull()) {
            cout << "Queue Overflow! Cannot enqueue element " <<
element << endl;
            return;
        }
        if (front == -1) {
            front = rear = 0;
        } else if (rear == size - 1 && front != 0) {
            rear = 0;
        } else {
            rear = (rear + 1) % size;
        }
        queue[rear] = element;
        cout << "Enqueued element: " << element << endl;
    }
    // Function to remove and return an element from the queue
(dequeue operation)
```

```
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! Cannot dequeue from an
empty queue." << endl;
            return -1;
        }
        int element = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else if (front == size - 1) {
            front = 0;
        } else {
            front = (front + 1) % size;
        }
        return element;
    }
    // Function to return the front element of the queue without
removing it (peek operation)
    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty. No element to peek." << endl;
            return -1;
        }
        return queue[front];
    }
    // Function to display all elements of the queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty." << endl;
            return;
        }
        cout << "Queue elements: ";
        if (rear >= front) {
            for (int i = front; i <= rear; i++) {
                cout << queue[i] << " ";
            }
        } else {
            for (int i = front; i < size; i++) {
                cout << queue[i] << " ";
            }
            for (int i = 0; i <= rear; i++) {
                cout << queue[i] << " ";
            }
        }
        cout << endl;
    }
};
int main() {
    CircularQueue q(5);
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);
    q.display();
    cout << "Dequeued element: " << q.dequeue() << endl;
    q.enqueue(60);
    cout << "Peeked element: " << q.peek() << endl;
    q.display();
    return 0;
}
```

Applications and advantages of circular queues

Applications of Circular Queues:

Buffering Data in I/O Systems: Circular queues are commonly used in I/O operations where data is continuously received or transmitted. They provide a fixed-size buffer that can wrap around, allowing seamless data processing without the need for resizing or complex memory management.

Resource Management: They are useful in managing resources with a fixed capacity that need to be accessed in a circular manner. For example, in operating systems, circular queues can manage resources like CPU scheduling queues or device driver queues.

Implementation of Task Schedulers: Circular queues are employed in task scheduling algorithms where tasks are scheduled in a round-robin manner. Each task gets a turn to execute for a specified time quantum before being preempted, which is facilitated efficiently using a circular queue.

Network Traffic Management: In networking applications, circular queues can be used to manage network packets. They allow packets to be stored temporarily before processing or transmission, ensuring efficient handling of network traffic.

Memory Management: Circular queues can be utilized in memory management algorithms to implement caching mechanisms or in systems where memory allocation and deallocation need to be handled efficiently.

Advantages of Circular Queues:

Efficient Use of Memory: Circular queues use a fixed-size buffer, which makes efficient use of memory compared to dynamic data structures that may require resizing operations.

Constant Time Complexity: Operations such as enqueue and dequeue in circular queues typically have a constant time complexity O (1), assuming the circular nature is properly managed. This makes them highly efficient for real-time and embedded systems.

Simplified Implementation: Implementing circular queues is straightforward compared to other data structures like linked lists. They involve simple arithmetic operations (modular arithmetic) to manage the circular behavior.

Optimal for Streaming Applications: Due to their circular nature, these queues are ideal for streaming applications where data is continuously flowing. They ensure that the oldest data is overwritten when new data is added, maintaining a consistent flow.

Avoids Fragmentation: Unlike dynamic data structures, circular queues do not suffer from memory fragmentation issues because they use a contiguous block of memory. This makes them reliable for long-running applications.

6.4 PRIORITY QUEUES

A priority queue is an abstract data type (ADT) similar to a regular queue or stack but with added functionality that allows elements to be stored with a priority. Unlike a regular queue where the first element added is the first to be removed (FIFO - First In, First Out), a priority queue retrieves elements based on their priority. Here's an overview of priority queues:

A **priority queue** is a collection of elements where each element has a priority assigned to it. Elements with higher priorities are dequeued before elements with lower priorities. Priority queues do not follow the FIFO principle of regular queues; instead, they provide operations that allow elements to be added and removed based on their priority level.

Operations on Priority Queues:

Insertion (Enqueue): Adds an element to the priority queue based on its priority. Higher priority elements are placed at the front of the queue.

Algorithm:

Add the new element at the end of the heap (array representation).

Adjust the heap upwards (up-heap or bubble-up) to maintain the heap property (min-heap or max-heap).

Deletion (Dequeue): Removes and returns the highest priority element from the priority queue. If multiple elements have the same highest priority, they are typically removed in a FIFO order.

Algorithm:

Remove the root element (highest priority element in a max-heap or lowest priority in a min-heap).

Move the last element of the heap to the root position.

Adjust the heap downwards (down-heap or bubble-down) to restore the heap property.

Peek: Retrieves the highest priority element without removing it from the queue.

Algorithm:

Return the root element of the heap without removing it. This operation retrieves the highest priority element.

Implementation:

Priority queues can be implemented using various data structures, such as:

Binary Heap: A binary heap is a complete binary tree where each node satisfies the heap property (either min-heap or max-heap). This structure allows efficient insertion and deletion operations.

Binary Search Tree (BST): A BST can be used to implement a priority queue where elements are ordered based on their priority, allowing logarithmic time complexity for insertion and deletion operations.

Applications:

Priority queues are useful in scenarios where:

Task Scheduling: Operating systems use priority queues to manage tasks with different levels of priority. Higher priority tasks are executed sooner.

Dijkstra's Algorithm: In graph theory, priority queues are essential for implementing algorithms like Dijkstra's shortest path algorithm, where nodes are processed based on their shortest known path distance.

Event-driven Simulations: Systems that simulate real-world events (e.g., discrete event simulations) often use priority queues to manage events scheduled for future processing.

Advantages:

Efficient Operations: Priority queues allow O (log n) time complexity for insertion and deletion operations (depending on the implementation), making them suitable for real-time applications.

Flexible Data Structure: They provide flexibility in managing data with varying priorities, allowing dynamic adjustments based on application needs.

Optimized for Specific Applications: Priority queues are tailored to handle specific scenarios where prioritization of elements is critical, such as in scheduling and optimization problems.

6.5 Double-ended Queues (Deque)

A double-ended queue, often abbreviated as deque (pronounced "deck"), is a versatile data structure that allows insertion and deletion of elements from both ends. Unlike queues and stacks, which support insertion and deletion from only one end, deques support operations from both the front and the back. Here's an overview of double-ended queues:

A **deque** is a linear collection of elements that supports efficient insertion and deletion operations at both ends. It allows elements to be added or removed from the front or the back, making it suitable for scenarios requiring flexibility in data access patterns.

Operations on Deques:

Insertion at Front and Back:

push_front(value): Adds an element to the front of the deque.

push_back(value): Adds an element to the back of the deque.

Deletion from Front and Back:

pop_front(): Removes and returns the element at the front of the deque.

pop_back(): Removes and returns the element at the back of the deque.

Accessing Elements:

front(): Returns (but does not remove) the element at the front of the deque.

back(): Returns (but does not remove) the element at the back of the deque.

Size and Empty Check:

size(): Returns the number of elements currently stored in the deque.

empty(): Checks if the deque is empty and returns true if no elements are present.

Implementation Considerations:

Array-based Implementation: Deques can be implemented using dynamic arrays or circular arrays to allow efficient insertion and deletion operations at both ends.

Linked List Implementation: Using a doubly linked list allows constant time complexity for insertion and deletion operations at both ends, but it requires more memory overhead due to storing pointers for each element.

Applications:

Double-ended Queues: Used in applications where elements need to be accessed and modified efficiently from both ends, such as implementing deque-based data structures like deque-based algorithms.

Simulation Systems: Deques are suitable for implementing simulation systems where events can be added or removed from the front or back based on their priority or timestamp.

Memory Management: Used in memory management systems where elements need to be dynamically allocated or deallocated from both ends.

Advantages:

Flexibility: Provides flexibility in accessing and manipulating elements from both ends, allowing various data processing and algorithmic operations.

Efficiency: Supports efficient insertion and deletion operations with constant time complexity when implemented using arrays or linked lists.

Versatility: Offers a versatile approach to handling data structures that require dynamic management of elements based on their position and priority.

Operations: insert front, insert rear, delete front, delete rear

1. Insert Front (push_front)

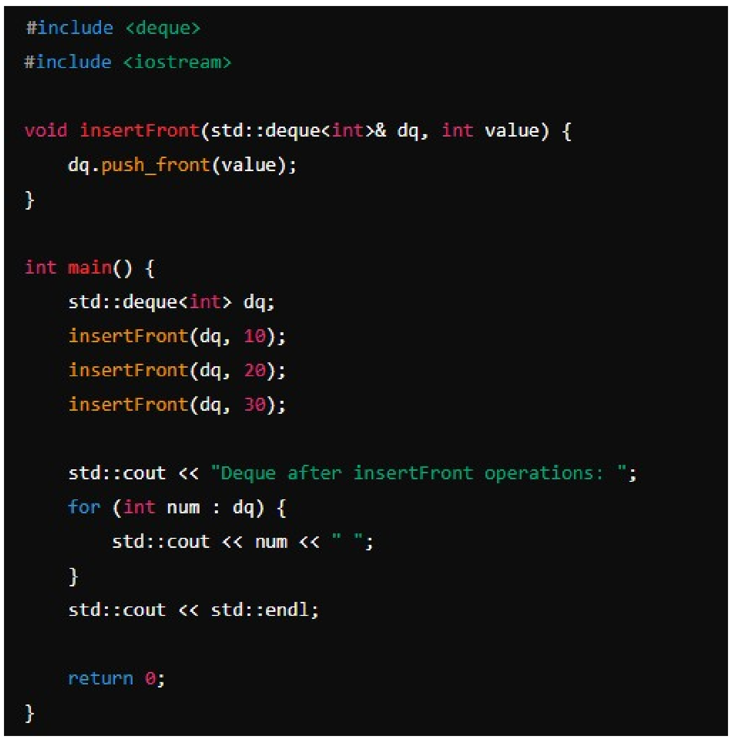
Description: Adds an element to the front of the deque.

Algorithm:

If using a dynamic array or a vector, shift all existing elements to the right to make space for the new element at the front.

If using a doubly linked list, create a new node and adjust pointers to insert it at the beginning of the list.

C++ Implementation (using std::deque):



```cpp
#include <deque>
#include <iostream>

void insertFront(std::deque<int>& dq, int value) {
    dq.push_front(value);
}

int main() {
    std::deque<int> dq;
    insertFront(dq, 10);
    insertFront(dq, 20);
    insertFront(dq, 30);

    std::cout << "Deque after insertFront operations: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

2. Insert Rear (push_back)

Description: Adds an element to the back of the deque.

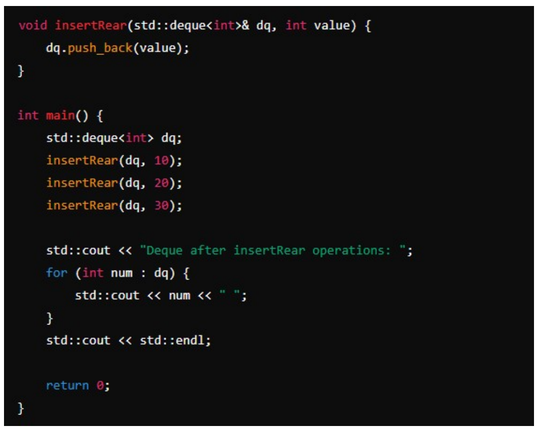
Algorithm:

Append the new element at the end of the deque.

For a dynamic array, this typically involves appending the element to the vector.

For a doubly linked list, simply adjust pointers to insert the new node at the end.

C++ Implementation (using std::deque):



```cpp
void insertRear(std::deque<int>& dq, int value) {
    dq.push_back(value);
}

int main() {
    std::deque<int> dq;
    insertRear(dq, 10);
    insertRear(dq, 20);
    insertRear(dq, 30);

    std::cout << "Deque after insertRear operations: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

3. Delete Front (pop_front)

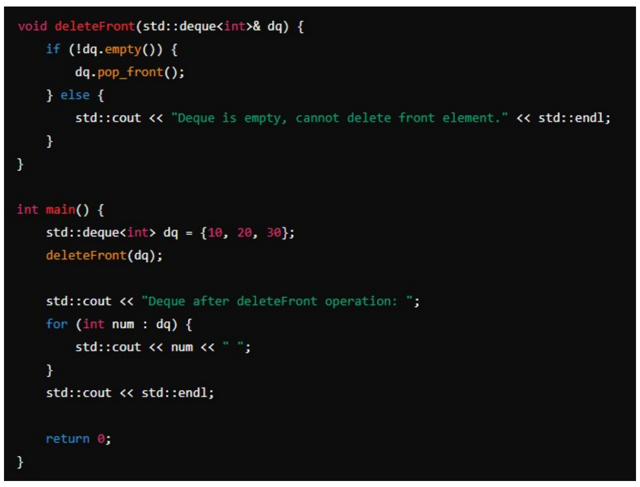
Description: Removes and returns the element at the front of the deque.

Algorithm:

For a dynamic array, remove the first element and shift all other elements to the left.

For a doubly linked list, adjust pointers to remove the first node.

C++ Implementation (using std::deque):



```cpp
void deleteFront(std::deque<int>& dq) {
    if (!dq.empty()) {
        dq.pop_front();
    } else {
        std::cout << "Deque is empty, cannot delete front element." << std::endl;
    }
}

int main() {
    std::deque<int> dq = {10, 20, 30};
    deleteFront(dq);

    std::cout << "Deque after deleteFront operation: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4. Delete Rear (pop_back)

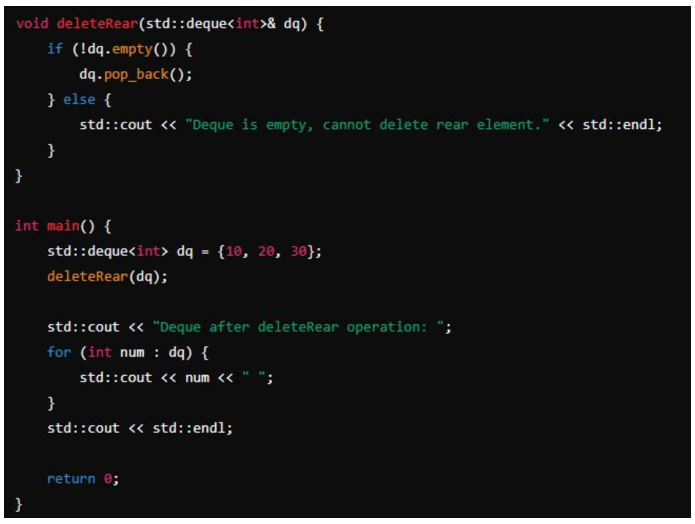
Description: Removes and returns the element at the back of the deque.

Algorithm:

For a dynamic array, remove the last element.

For a doubly linked list, adjust pointers to remove the last node.

C++ Implementation (using std::deque):



```
void deleteRear(std::deque<int>& dq) {
    if (!dq.empty()) {
        dq.pop_back();
    } else {
        std::cout << "Deque is empty, cannot delete rear element." << std::endl;
    }
}

int main() {
    std::deque<int> dq = {10, 20, 30};
    deleteRear(dq);

    std::cout << "Deque after deleteRear operation: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

6.6 CONCLUSION

Throughout this section, we have delved into the fundamental role of queues as crucial data structures for managing data in a First-In-First-Out (FIFO) manner, pivotal across a wide array of computational applications. Our exploration began with an in-depth look at linked list implementations of queues, emphasizing their dynamic memory allocation and efficient handling of insertion and deletion operations. Linked lists provide adaptability to varying queue sizes, making them particularly suited for scenarios where data changes frequently and unpredictably.

Circular queues were also examined for their specialized ability to manage continuous data streams efficiently. By leveraging circular arrays or linked structures, circular queues minimize memory overhead and ensure seamless data circulation, ideal for applications requiring uninterrupted processing loops. This variant underscores the importance of efficient data management in optimizing computational workflows and system performance.

Additionally, our discussion encompassed priority queues and double-ended queues (deques), each tailored to specific operational needs. Priority queues prioritize elements based on predefined criteria, essential for time-sensitive tasks where urgency dictates processing order. Deques, offering operations from both ends of the queue, provide flexibility in data manipulation, catering to diverse data handling requirements.

In mastering the concepts and implementations covered here, developers gain essential tools for designing robust algorithms and efficient software systems. Understanding the operational mechanics and strategic applications of different queue structures empowers developers to make informed decisions, enhancing their ability to optimize data workflows, manage real-time tasks effectively, and ultimately improve overall system performance.

6.7 QUESTIONS AND ANSWERS

1. What is the primary principle that queues follow, and why is it important in data management?
Answer: Queues follow the First-In-First-Out (FIFO) principle, where the first element inserted into the queue is the first one to be removed. This principle ensures that data is processed in the order of arrival, essential for managing tasks or data items that need to be handled sequentially.

2. How does a linked list implementation of a queue differ from an array-based implementation?
Answer: Linked List: Implementing a queue using linked lists offers flexibility in dynamic memory allocation and efficient insertion and deletion operations. It allows for easy expansion and contraction of the queue size as elements are added or removed.

Array-based: Arrays provide direct access to elements but require resizing when the array becomes full, which can be less efficient for large queues or dynamic data sizes.

3. What are the advantages of using a circular queue over a linear queue?
Circular queues optimize memory usage by reusing space in a circular manner, preventing the need for shifting elements when the front of the queue becomes empty. This efficiency is crucial for continuous data processing scenarios where data elements need to be processed in a loop without interruption.
4. How are priority queues different from regular queues, and what are their typical applications?
Answer: Difference: Priority queues prioritize elements based on predefined criteria (such as numerical value or urgency) rather than the order of insertion. Higher priority elements are processed before lower priority ones, making them suitable for applications like task scheduling, job prioritization, and event handling in real-time systems.

5. What advantages do double-ended queues (deques) offer over standard queues, and in what scenarios are they beneficial?
Answer: Deques support operations at both ends (front and back), allowing for flexible data manipulation. This capability is advantageous in scenarios where elements need to be added or removed from either end dynamically, such as managing sliding windows in data processing or implementing advanced data structures like stacks and queues simultaneously.

6.8 REFERENCES

Books:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

"Data Structures and Algorithms in Java" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser.

"Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss.

Documentation:

C++ Standard Library documentation for std::queue, std::deque, and related containers.

UNIT – 7: TREES

Structure

7.0 INTRODUCTION

Binary trees are a fundamental data structure in computer science, serving as the foundation for various complex data structures and algorithms. Understanding binary trees is crucial for effectively managing and organizing hierarchical data. This chapter delves into the intricacies of binary trees, covering their abstract data types, implementation, and traversal techniques. We will explore the properties and types of binary trees, emphasizing their practical applications and operational methodologies.

Binary trees not only provide efficient means for data storage and retrieval but also enhance the performance of search and sort operations. They are employed in a myriad of applications, from database indexing and syntax parsing in compilers to network routing algorithms and artificial intelligence.

This chapter aims to equip you with a comprehensive understanding of binary trees, including their conceptual framework, implementation strategies, and common operations. By the end of this chapter, you will have a solid grasp of how to construct, manipulate, and traverse binary trees, preparing you for more advanced topics in data structures and algorithms.

7.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand Binary Trees: Define the structure and characteristics of binary trees.

Implement Binary Trees: Implement binary trees using linked representation (nodes and pointers).

Perform Tree Traversals: Implement and apply inorder, preorder, and postorder traversal methods.

Conduct Operations on Binary Trees: Perform insertion, deletion, and search operations on binary trees.

Apply Binary Trees in Problem Solving: Recognize and apply binary trees in solving real-world problems.

7.2 ABSTRACT DATA TYPE

An Abstract Data Type (ADT) refers to a theoretical model that defines a set of operations and the semantics of those operations on a data structure, without specifying how the data structure should be implemented. It provides a high-level description of data and operations, allowing for flexibility in implementation while ensuring consistency in behavior.

Characteristics of ADTs:

Encapsulation: ADTs encapsulate data and operations within a cohesive unit, shielding internal details from external access. This promotes information hiding and ensures that operations are performed through well-defined interfaces.

Operations: ADTs define a set of operations that can be performed on the data structure. These operations include creating the structure, inserting or deleting elements, accessing elements, and other manipulations specific to the type of data structure.

Data Abstraction: ADTs abstract away the underlying details of data representation and storage. Users interact with the ADT through a predefined set of operations, focusing on what operations can be performed rather than how they are implemented.

Implementation Flexibility: ADTs can be implemented using various programming paradigms and data structures. For example, a queue ADT can be implemented using arrays, linked lists, or other structures, as long as it adheres to the specified operations and behavior.

Example ADTs:

Stack: Supports operations like push (add element), pop (remove element), and peek (view top element), following Last-In-First-Out (LIFO) order.

Queue: Allows operations such as enqueue (add element), dequeue (remove element), and peek (view front element), adhering to First-In-First-Out (FIFO) order.

Tree: Defines operations for creating nodes, inserting or deleting nodes, and traversing the tree (e.g., inorder, preorder, postorder).

Graph: Defines vertices and edges, supporting operations like adding vertices, adding edges between vertices, and traversing through vertices and edges.

Benefits of ADTs:

Modularity: ADTs promote modular programming by separating data structure definitions from their implementations, facilitating code reuse and maintenance.

Abstraction: ADTs hide complex implementation details, allowing programmers to focus on solving problems at a higher level of abstraction.

Flexibility: ADTs provide flexibility in choosing implementation strategies based on performance requirements or specific application needs, without affecting the overall functionality.

7.3 TREE DATA STRUCTURE

A tree in computer science is a hierarchical data structure composed of nodes. Each node typically contains a value and a list of references to its child nodes. The structure starts from a root node and branches out into subtrees, where each subtree is also a tree in itself. Trees are used to represent hierarchical relationships, such as file systems, organizational charts, or abstract syntax trees in programming languages. Key concepts in trees include the root (topmost node), parent and child relationships, siblings (nodes sharing the same parent), and leaves (nodes without children). Common operations on trees include traversal (visiting each node in a specific order), insertion of nodes, deletion of nodes, and searching for nodes based on their values or properties. Trees can vary in complexity and types, such as binary trees (where each node has at most two children), balanced trees (maintaining a balanced structure for efficient operations), and more specialized structures like binary search trees (BSTs) for efficient searching and sorting operations. Understanding trees is fundamental for designing efficient algorithms and data structures in computer science.

The TreeNode struct defines each node in the tree, storing an integer value and pointers to its left and right child nodes. The BinaryTree class manages the tree operations, including insertion (insert method) and inorder traversal (inorderTraversal method). Insertion is handled recursively (insertRecursive), ensuring each value is placed correctly according to its relation with existing nodes. The inorderTraversal method recursively visits nodes in left subtree, root, and right subtree order, printing node values to display them in sorted order. The main function exemplifies the usage by creating a BinaryTree instance, inserting values (50, 30, 20, 40, 70, 60, 80) into the tree, and then performing an inorder traversal to output the values in ascending order. This example serves as a foundational implementation of a binary tree in C++, suitable for basic tree operations and traversal techniques.

Operations supported by the tree ADT, such as insertion, deletion, traversal, and searching.

The operations supported by the tree Abstract Data Type (ADT) include:

Insertion: Adding a new node to the tree. The node is typically inserted based on specific rules, such as maintaining a sorted order in a binary search tree.

Deletion: Removing a node from the tree while maintaining the tree's structural integrity. This operation may involve reorganizing the tree to ensure it remains a valid tree structure.

Traversal: Visiting all nodes in the tree in a specific order. Common traversal methods include:

Inorder: Visit left subtree, current node, right subtree.
Preorder: Visit current node, left subtree, right subtree.
Postorder: Visit left subtree, right subtree, current node. These traversals are useful for various applications, such as evaluating mathematical expressions (inorder) or copying a tree (preorder).

Searching: Finding a node with a specific value or property within the tree. Searching in a binary search tree, for example, can be efficient due to its ordered nature, allowing for logarithmic time complexity in balanced trees.

7.4 IMPLEMENTATION OF TREE

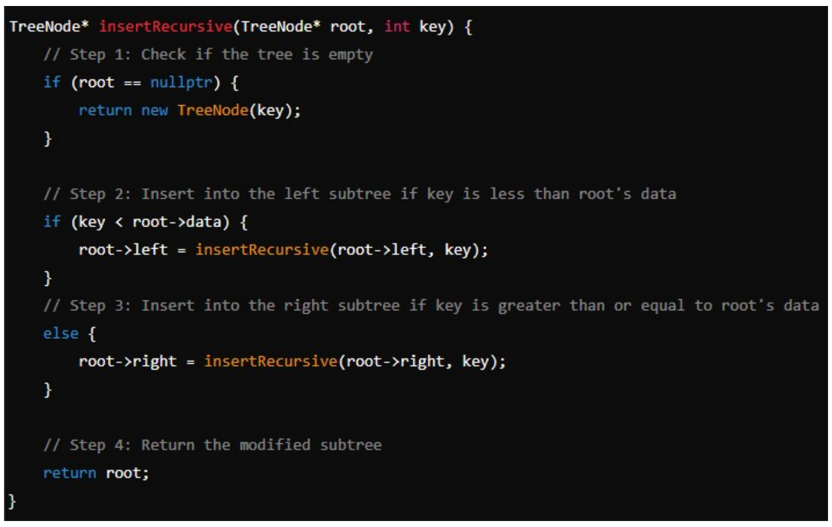
An algorithmic approach for inserting nodes into a binary search tree (BST) and performing an inorder traversal:

Algorithm: Insertion in Binary Search Tree (BST)

```
1. Start with the root of the BST.
2. If the tree is empty, insert the new node as the root.
3. Otherwise, compare the data value of the new node with the current node:
   a. If the new node's value is less than the current node's value,
      - Recursively insert the new node into the left subtree.
   b. If the new node's value is greater than or equal to the current node's value,
      - Recursively insert the new node into the right subtree.
4. Return the modified subtree with the new node.
```

Example:

Let's apply this algorithm to insert a value key into a BST:



```
TreeNode* insertRecursive(TreeNode* root, int key) {
    // Step 1: Check if the tree is empty
    if (root == nullptr) {
        return new TreeNode(key);
    }

    // Step 2: Insert into the left subtree if key is less than root's data
    if (key < root->data) {
        root->left = insertRecursive(root->left, key);
    }
    // Step 3: Insert into the right subtree if key is greater than or equal to root's data
    else {
        root->right = insertRecursive(root->right, key);
    }

    // Step 4: Return the modified subtree
    return root;
}
```

Considerations for dynamic memory management and efficient node operations.

When implementing tree data structures, especially in languages like C++ where manual memory management is common, considerations for dynamic memory management and efficient node operations are crucial for performance and memory usage optimization. Here are some key considerations:

Dynamic Memory Management:

Node Allocation: Each node in the tree should be dynamically allocated using new in C++ to manage memory efficiently. This ensures nodes are allocated on the heap and can be accessed globally throughout the tree.

Node Deallocation: When nodes are no longer needed (e.g., during deletion operations), they should be explicitly deallocated using delete to avoid memory leaks. This is particularly important for recursive operations like tree traversal and deletion.

Memory Efficiency: Consider using memory pooling or custom memory allocation strategies if managing a large number of nodes to reduce overhead from frequent allocations and deallocations.

Efficient Node Operations:

Insertion: Implement insertion operations (e.g., for BST) using recursive or iterative methods that maintain the BST properties efficiently. Ensure nodes are inserted in the correct position to maintain the ordering properties of the tree.

Deletion: Implement deletion operations carefully to preserve the structure and properties of the tree (e.g., BST deletion). Handle cases for nodes with zero, one, or two children, ensuring the tree remains balanced and valid after deletion.

Traversal: Use efficient traversal algorithms (e.g., inorder, preorder, postorder) to visit nodes in a specific order. Recursive implementations are straightforward but may consume stack space for deep trees; iterative implementations using stacks or queues can be more memory efficient.

Balancing (for balanced trees): Consider implementing self-balancing tree structures (e.g., AVL tree, Red-Black tree) for operations like insertion and deletion that maintain balance automatically, ensuring logarithmic time complexity for search operations.

Node Access and Modification: Design node structures with efficient access and modification methods (e.g., getters, setters) to manipulate node data and relationships without compromising tree integrity or performance.

7.5 TREE TRAVERSALS

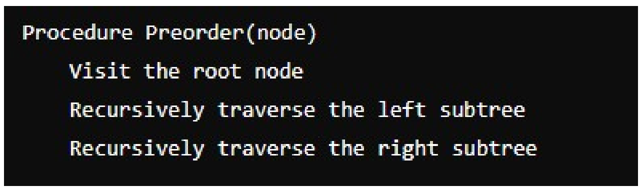
Tree traversals are essential techniques for accessing and processing nodes in a tree data structure. Here's an overview of the different traversal methods:

Depth-First Traversals

1. Preorder Traversal

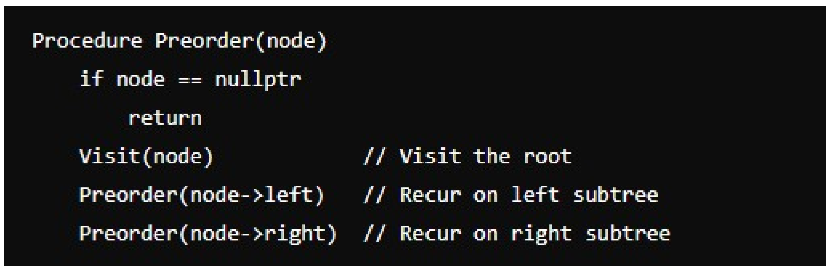
Definition: Visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a preorder traversal of the right subtree.

Algorithm:



```
Procedure Preorder(node)
    Visit the root node
    Recursively traverse the left subtree
    Recursively traverse the right subtree
```

Implementation:



```
Procedure Preorder(node)
    if node == nullptr
        return
    Visit(node)            // Visit the root
    Preorder(node->left)   // Recur on left subtree
    Preorder(node->right)  // Recur on right subtree
```

Usage: Useful for creating a copy of the tree or evaluating expressions.

2. Inorder Traversal

Definition: In an inorder traversal, nodes are visited in ascending order (for BSTs) by recursively visiting the left subtree, then the node itself, and finally the right subtree.

Algorithm:

```
Procedure Inorder(node)
    Recursively traverse the left subtree
    Visit the root node
    Recursively traverse the right subtree
```

Implementation:

```
Procedure Inorder(node)
    if node == nullptr
        return
    Inorder(node->left)    // Recur on left subtree
    Visit(node)            // Visit the root
    Inorder(node->right)   // Recur on right subtree
```

Usage: Useful for retrieving elements in sorted order from BSTs.

3. Postorder Traversal

Definition: Recursively do a postorder traversal of the left subtree, then recursively do a postorder traversal of the right subtree, and finally visit the root node.

Algorithm:

```
Procedure Postorder(node)
    Recursively traverse the left subtree
    Recursively traverse the right subtree
    Visit the root node
```

Implementation:

```
Procedure Postorder(node)
    if node == nullptr
        return
    Postorder(node->left)  // Recur on left subtree
    Postorder(node->right) // Recur on right subtree
    Visit(node)            // Visit the root
```

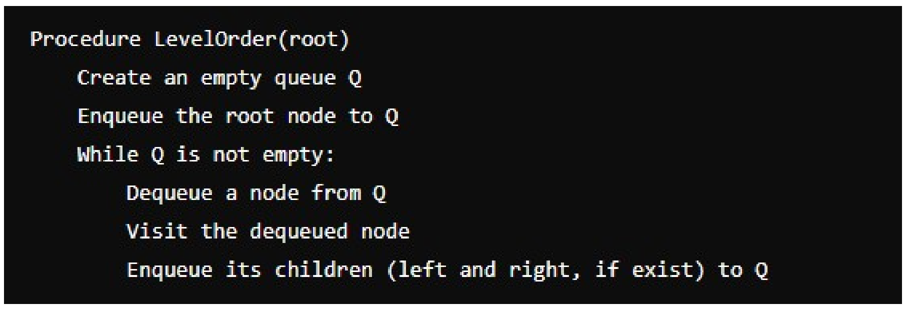
Usage: Useful for deleting nodes or evaluating expressions.

Breadth-First Traversal

4. Level Order Traversal (Breadth-First)

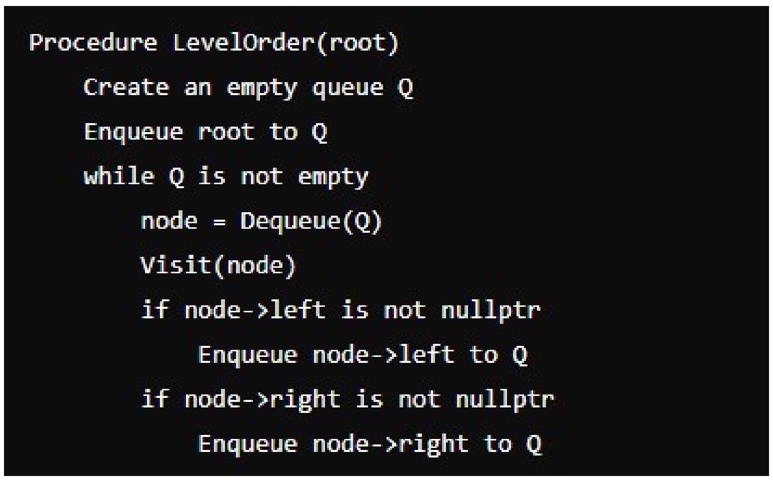
Definition: In a level order traversal, nodes are visited level by level from left to right.

Algorithm:



```
Procedure LevelOrder(root)
    Create an empty queue Q
    Enqueue the root node to Q
    While Q is not empty:
        Dequeue a node from Q
        Visit the dequeued node
        Enqueue its children (left and right, if exist) to Q
```

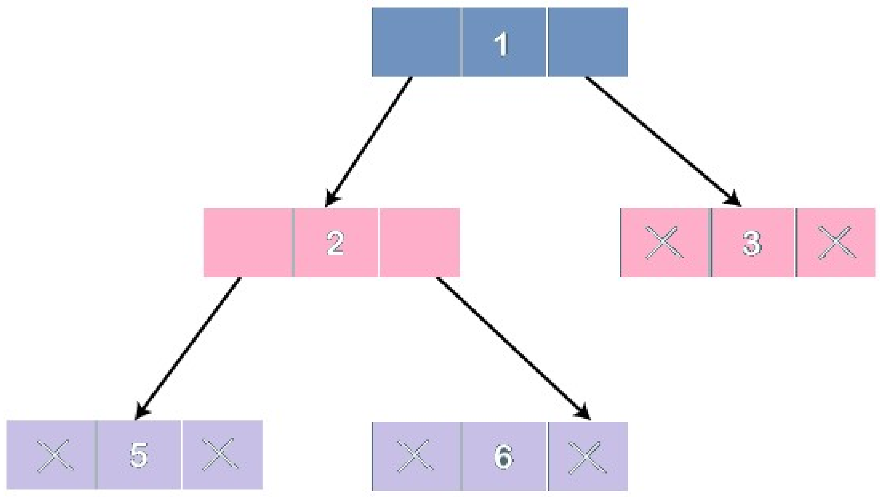
Implementation:



```
Procedure LevelOrder(root)
    Create an empty queue Q
    Enqueue root to Q
    while Q is not empty
        node = Dequeue(Q)
        Visit(node)
        if node->left is not nullptr
            Enqueue node->left to Q
        if node->right is not nullptr
            Enqueue node->right to Q
```

7.6 BINARY TREES

According to the binary tree, a node can only have a maximum of two children. Since the binary name in this case implies "two," each node may have zero, one, or two children.



Properties:

Height of Binary Tree: The length of the path from the root to the deepest node.

Number of Nodes: In a binary tree of height h, the maximum number of nodes is 2^{h+1} –

1. Depth of a Node: The length of the path from the root to that node.

Leaf Node: A node with no children.

Internal Node: A node with at least one child.

Binary Search Tree (BST): A binary tree in which for every node, the value of all the nodes in the left subtree is less, and the value of all the nodes in the right subtree is greater.

Types of Binary Tree

There are four types of Binary tree

Full/ proper/ strict Binary tree

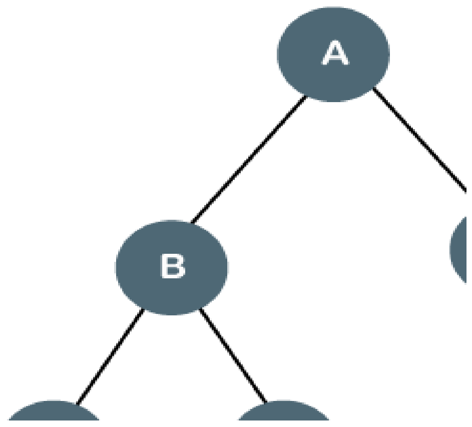
Complete Binary tree

Perfect Binary tree

Balanced Binary tree

Full Binary Tree:

Strict binary trees are another name for full binary trees. Only when every node has either 0 or 2 offspring can the tree be said to be the full binary tree. Another way to describe the full binary tree is as a tree where every node—aside from leaf nodes—must have two children.



The aforementioned tree is a Full Binary tree since every node can be shown to have either zero or two offspring.

Properties of Full Binary Tree

One more internal node is added to the total number of leaf nodes. Since there are five internal nodes in the example above, there are six leaf nodes overall.

The maximum number of nodes, or 2^{h+1} -1, is equal to the number of nodes in the binary tree.

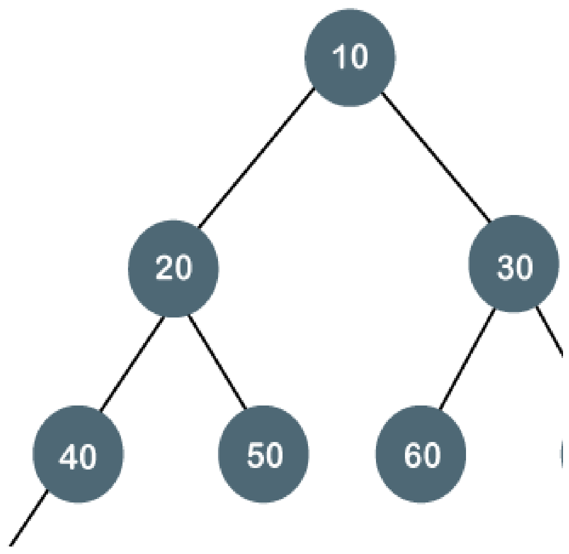
In the whole binary tree, there must be at least 2 * h - 1 nodes.

The whole binary tree has a minimum height of $\log_2 (n+1)$ - 1.

The formula for calculating the maximum height of the entire binary tree is n= 2*h - 1.

Complete Binary Tree

With the exception of the final level, every node in the complete binary tree is fully filled. Every node needs to be as far to the left as feasible in the final level. The nodes in a full binary tree have to be inserted from the left.



Because every node in the last level is inserted at the left first and every node is fully filled, the aforementioned tree is a complete binary tree.

Properties of Complete Binary Tree

A binary tree with all nodes can have a maximum of 2^{h+1} - 1.

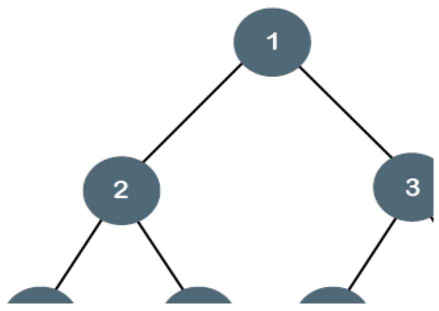
The minimum number of nodes in complete binary tree is 2h.

A full binary tree has a minimum height of log2(n+1) - 1.

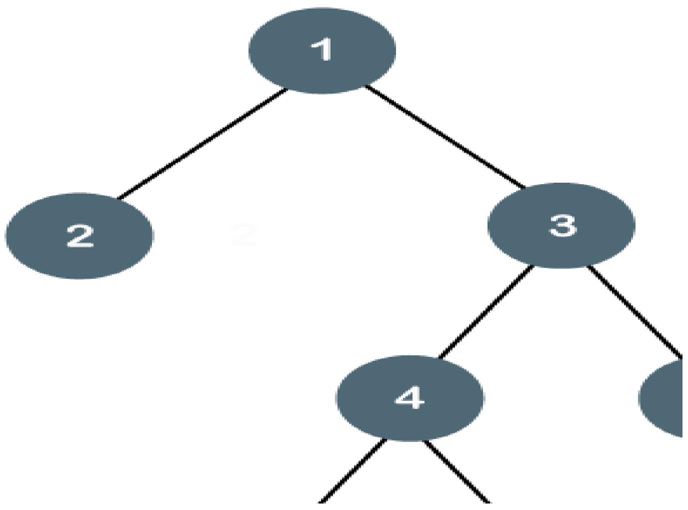
The highest point on an entire binary tree is

Perfect Binary Tree

If every internal node in a tree has two offspring and every leaf node is at the same level, the tree is a perfect binary tree.

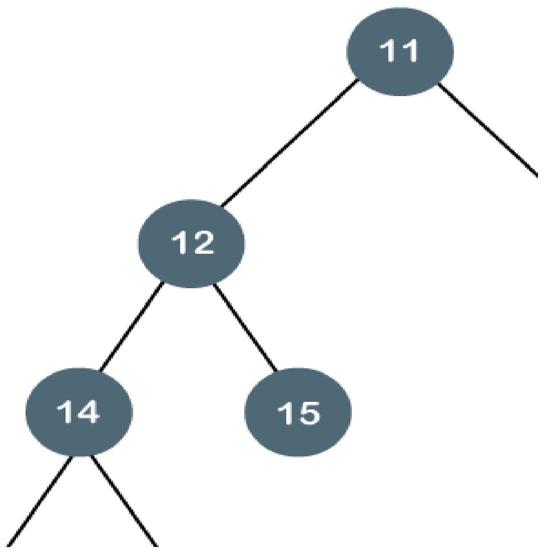


The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



Balanced Binary Tree

A balanced binary tree is one in which the difference between the left and right trees is no more than 1. Red-Black and AVL trees are two examples of balanced binary trees.

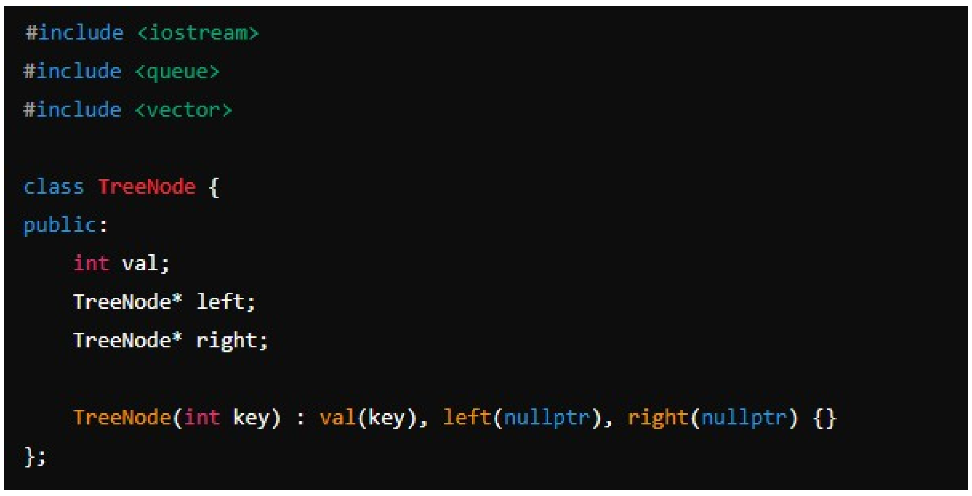


7.7 IMPLEMENTATION OF BINARY TREE

Here is a basic implementation of a binary tree in C++. The implementation includes a node class, the binary tree class with insertion and traversal methods.

Binary Tree Node Class

First, we define a class for the nodes of the binary tree:



```
#include <iostream>
#include <queue>
#include <vector>

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int key) : val(key), left(nullptr), right(nullptr) {}
};
```

Array representation of binary trees:

Array representation of binary trees is a way to store a binary tree using an array (or vector). This method is particularly useful for complete binary trees. Here's how it works:

Root: The root of the binary tree is stored at the first index of the array (index 0).

Parent-Child Relationship:

For a node at index i:

The left child is at index 2i + 1.

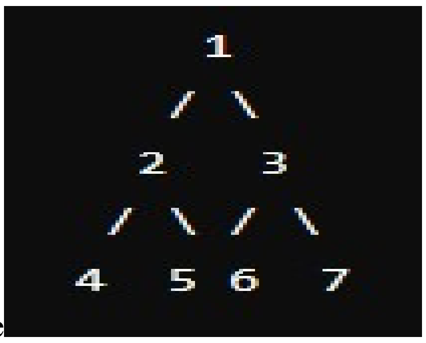
The right child is at index 2i + 2.

Conversely:

The parent of a node at index i is at index (i - 1) / 2.

Example

Consider the following binary tree:

e

```
      1
     / \
    2   3
   / \ / \
  4  5 6  7
```

This tree can be represented in an array as:

[1, 2, 3, 4, 5, 6, 7]

Implementation in C++

Here's a basic implementation of a binary tree using array representation in C++:

```
#include <iostream>
#include <vector>
class BinaryTree {
public:
    BinaryTree() {}
    void insert(int key) {
        arr.push_back(key);
    }
    void inorderTraversal(int index, std::vector<int>& result) {
        if (index < arr.size()) {
            inorderTraversal(2 * index + 1, result); // Visit left subtree
            result.push_back(arr[index]);            // Visit node
            inorderTraversal(2 * index + 2, result); // Visit right subtree
        }
    }
    void preorderTraversal(int index, std::vector<int>& result) {
        if (index < arr.size()) {
            result.push_back(arr[index]);            // Visit node
            preorderTraversal(2 * index + 1, result); // Visit left subtree
            preorderTraversal(2 * index + 2, result); // Visit right
subtree
        }
    }
    void postorderTraversal(int index, std::vector<int>& result) {
        if (index < arr.size()) {
            postorderTraversal(2 * index + 1, result); // Visit left
subtree
            postorderTraversal(2 * index + 2, result); // Visit right
subtree
            result.push_back(arr[index]);              // Visit node
        }
    }
    std::vector<int> inorder() {
        std::vector<int> result;
        inorderTraversal(0, result);
        return result;
    }
    std::vector<int> preorder() {
        std::vector<int> result;
        preorderTraversal(0, result);
        return result;
    }
    std::vector<int> postorder() {
        std::vector<int> result;
        postorderTraversal(0, result);
        return result;
    }
private:
    std::vector<int> arr;
```

```
};
int main() {
    BinaryTree tree;
    tree.insert(1);
    tree.insert(2);
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    tree.insert(6);
    tree.insert(7);
    std::vector<int> inorderResult = tree.inorder();
    std::vector<int> preorderResult = tree.preorder();
    std::vector<int> postorderResult = tree.postorder();
    std::cout << "Inorder traversal: ";
    for (int val : inorderResult) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    std::cout << "Preorder traversal: ";
    for (int val : preorderResult) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    std::cout << "Postorder traversal: ";
    for (int val : postorderResult) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Explanation

Insert: Adds a new element to the end of the array.

Traversal Methods:

Inorder Traversal: Recursively visits the left subtree, the node, and then the right subtree.

Preorder Traversal: Recursively visits the node, the left subtree, and then the right subtree.

Postorder Traversal: Recursively visits the left subtree, the right subtree, and then the node.

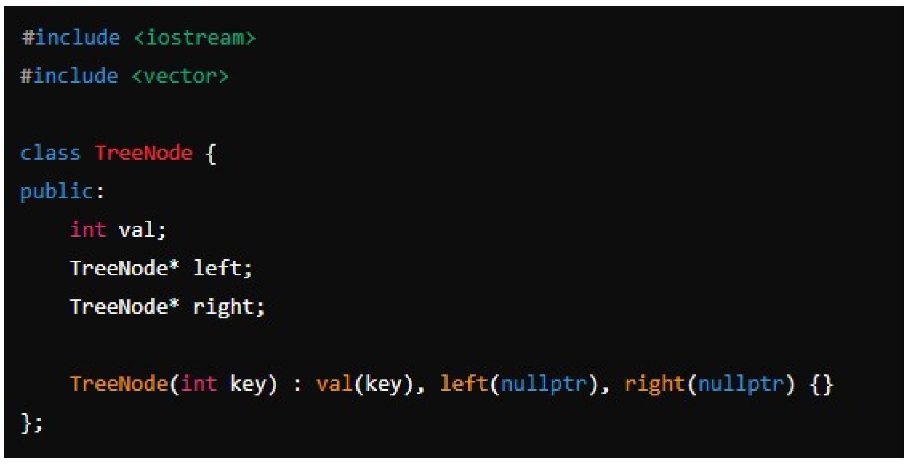
Main Function: Demonstrates the usage of the BinaryTree class by inserting elements and performing different traversals.

Linked representation (using nodes and pointers) of binary trees.

In the linked representation of binary trees, each node is represented by a structure (or class) that contains data and pointers to its left and right children. This is a more flexible way to represent binary trees compared to array representation, especially for trees that are not complete.

Node Structure

First, define a structure (or class) for the nodes of the binary tree:



```cpp
#include <iostream>
#include <vector>

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int key) : val(key), left(nullptr), right(nullptr) {}
};
```

Binary Tree Class

Next, define a class for the binary tree that includes methods to insert nodes and perform traversals:

```
class BinaryTree {
public:
  BinaryTree() : root(nullptr) {}
  void insert(int key) {
    if (root == nullptr) {
      root = new TreeNode(key);
    } else {
      insert(root, key);
    }
  }
  void inorderTraversal(TreeNode* node, std::vector<int>& result) {
    if (node != nullptr) {
      inorderTraversal(node->left, result);
      result.push_back(node->val);
      inorderTraversal(node->right, result);
    }
  }
  void preorderTraversal(TreeNode* node, std::vector<int>& result) {
    if (node != nullptr) {
      result.push_back(node->val);
      preorderTraversal(node->left, result);
      preorderTraversal(node->right, result);
    }
  }
  void postorderTraversal(TreeNode* node, std::vector<int>& result) {
    if (node != nullptr) {
            postorderTraversal(node->left, result);
            postorderTraversal(node->right, result);
            result.push_back(node->val);
        }
    }
    std::vector<int> inorder() {
        std::vector<int> result;
        inorderTraversal(root, result);
        return result;
    }
    std::vector<int> preorder() {
        std::vector<int> result;
        preorderTraversal(root, result);
        return result;
    }
    std::vector<int> postorder() {
        std::vector<int> result;
        postorderTraversal(root, result);
        return result;
    }
private:
    TreeNode* root;
    void insert(TreeNode* node, int key) {
        if (key < node->val) {
            if (node->left == nullptr) {
                node->left = new TreeNode(key);
            } else {
                insert(node->left, key);
            }
        } else {
            if (node->right == nullptr) {
                node->right = new TreeNode(key);
            } else {
                insert(node->right, key);
            }
        }
    }
};
```

7.8 OPERATIONS ON BINARY TREES

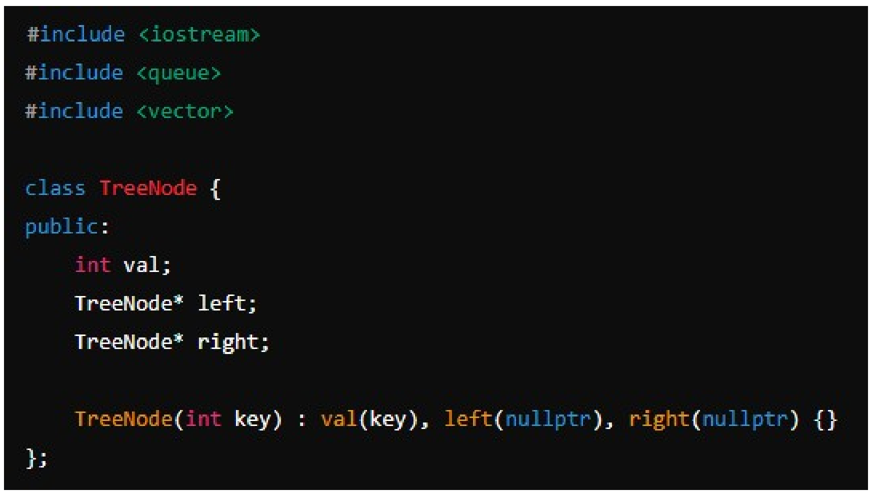
Here’s a comprehensive guide to various operations on binary trees using different algorithms in C++:

Operations on Binary Trees: Insertion with Algorithms

In a binary tree, insertion can be performed in different ways based on the type of binary tree (e.g., Binary Search Tree, Complete Binary Tree). Here, we'll explore insertion in both a Binary Search Tree (BST) and a Complete Binary Tree.

Node Structure

Define a structure for the nodes of the binary tree:



```cpp
#include <iostream>
#include <queue>
#include <vector>

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int key) : val(key), left(nullptr), right(nullptr) {}
};
```

Insertion in a Binary Search Tree (BST)

In a BST, nodes are inserted such that the left subtree of a node contains only nodes with keys less than the node’s key, and the right subtree only nodes with keys greater than the node’s key.

Insertion Algorithm

Start at the root node.

Compare the key to be inserted with the current node's key:

If the key is less, move to the left child.

If the key is greater, move to the right child.

Repeat step 2 until finding an appropriate null position.

Insert the new node at the found null position.

C++ Implementation

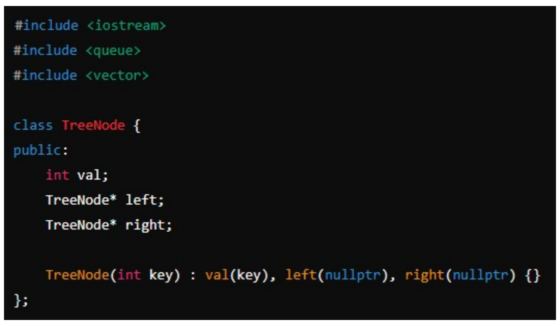
```
class BinarySearchTree {
public:
    BinarySearchTree() : root(nullptr) {}
    void insert(int key) {
        root = insert(root, key);
    }
    void inorder() {
        inorderTraversal(root);
    }
private:
    TreeNode* root;
    TreeNode* insert(TreeNode* node, int key) {
        if (node == nullptr) {
            return new TreeNode(key);
        }
        if (key < node->val) {
            node->left = insert(node->left, key);
        } else if (key > node->val) {
            node->right = insert(node->right, key);
        }
        return node;
    }
    void inorderTraversal(TreeNode* node) {
        if (node != nullptr) {
            inorderTraversal(node->left);
            std::cout << node->val << " ";
            inorderTraversal(node->right);
        }
    }
};
```

Deletion in Binary Trees

Deletion in binary trees can vary based on the type of binary tree (e.g., Binary Search Tree, Complete Binary Tree). Here, we'll explore deletion in both a Binary Search Tree (BST) and a Complete Binary Tree.

Node Structure

Define a structure for the nodes of the binary tree:



```
#include <iostream>
#include <queue>
#include <vector>

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int key) : val(key), left(nullptr), right(nullptr) {}
};
```

Deletion in a Binary Search Tree (BST)

In a BST, deletion involves three main cases:

The node to be deleted is a leaf node (no children).

The node to be deleted has one child.

The node to be deleted has two children.

Deletion Algorithm

Find the node to be deleted.

Handle the three cases for deletion:

No children: Simply remove the node.

One child: Replace the node with its child.

Two children: Find the in-order successor (smallest node in the right subtree), replace the node's value with the successor's value, and then delete the successor.

7.9 CONCLUSION

In conclusion, binary trees stand as foundational structures in computer science, pivotal for organizing hierarchical data efficiently and enabling optimized algorithms. Throughout this chapter, we have explored the fundamental concepts of binary trees, from their basic definition and implementation to the intricacies of traversal methods and essential operations.

Binary trees' versatility is evident in their application across various domains, including database management, expression parsing in compilers, and efficient routing in networks. Their ability to store and retrieve data in a hierarchical manner makes them indispensable in scenarios requiring structured data organization and rapid access.

Implementing and manipulating binary trees involves mastering operations such as insertion, deletion, and traversal. Each operation impacts the tree's structure and performance, necessitating careful consideration of algorithms to maintain balance and optimize efficiency. The choice of traversal method—whether inorder, preorder, or postorder—affects how nodes are accessed and processed, influencing the outcome of algorithms built upon binary tree structures.

Looking ahead, further exploration into balanced binary trees like AVL trees and Red-Black trees offers insights into maintaining optimal performance across operations, especially in scenarios involving large datasets and critical applications. Mastery of binary trees provides a solid foundation for tackling advanced data structures and algorithmic challenges, essential for aspiring computer scientists and engineers alike.

7.10 QUESTIONS AND ANSWERS

1. What is a binary tree?
Answer: A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child.

2. What are the main operations on binary trees?
Answer: The main operations on binary trees include insertion, deletion, and searching for nodes based on their key values. Additionally, traversal methods such as inorder, preorder, and postorder allow for accessing and processing nodes in different sequences.

3. What are the advantages of binary trees over other data structures?
Answer: Binary trees excel in scenarios where data needs to be organized hierarchically with efficient search, insertion, and

deletion operations. They are particularly useful in applications requiring sorted data and balanced access patterns.

4. How do you implement a binary tree in practice?
Answer: Binary trees can be implemented using linked structures (nodes with pointers to left and right children) or array-based representations (for complete binary trees). The implementation typically involves defining a node structure and methods to perform operations like insertion, deletion, and traversal.

5. What are some real-world applications of binary trees?
Answer: Binary trees find applications in various fields, including database indexing, hierarchical data storage, expression parsing in compilers, file system organization, and network routing algorithms.

7.11 REFERENCES

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd Edition)*. MIT Press.
- Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++ (4th Edition)*. Pearson Education.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in C++ (2nd Edition)*. John Wiley & Sons.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th Edition)*. Addison-Wesley.
- Sahni, S. (2006). *Data Structures, Algorithms, and Applications in C++*. McGraw-Hill Education.

UNIT – 8: BINARY

Structure

8.0 INTRODUCTION

Binary trees are fundamental data structures in computer science, characterized by nodes that have at most two children, commonly referred to as the left child and the right child. Traversing these structures involves systematically visiting each node, facilitating various operations and analyses crucial across numerous computational tasks. Binary tree traversals provide methods to explore and process nodes in specific sequences, each offering unique advantages in data manipulation and algorithmic applications.

Binary tree traversal algorithms, both recursive and iterative, are pivotal in understanding and manipulating hierarchical data efficiently. Recursive implementations, characterized by their straightforward approach using function calls, offer simplicity and clarity in algorithm design. Conversely, non-recursive approaches, employing explicit data structures like stacks or queues, provide control over memory usage and are often favored in environments sensitive to stack depth or performance.

This discussion explores the intricacies of binary tree traversals, delving into both their theoretical underpinnings and practical applications. We will examine recursive and non-recursive implementations of traversal algorithms, highlighting their respective strengths and use cases. Furthermore, we will explore diverse applications where binary tree traversals play a crucial role, from expression evaluation to graph algorithms and tree optimizations.

Understanding these traversal techniques equips us with essential tools for efficiently navigating and manipulating binary tree structures, underpinning foundational concepts in computer science and enabling sophisticated solutions across various domains.

8.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand Binary Tree Traversals: Explore the concepts of inorder, preorder, and postorder traversals, comprehending their definitions and traversal sequences within binary tree structures.

Compare Recursive and Iterative Implementations: Analyze the differences between recursive and iterative approaches to binary tree traversals, evaluating their performance characteristics and memory usage.

Implement Binary Tree Traversals: Develop proficiency in implementing recursive and non-recursive algorithms for inorder, preorder, and postorder traversals in practical scenarios.

Explore Applications: Investigate real-world applications of binary tree traversals across various domains, including data processing, algorithmic problem-solving, and data structure optimizations.

Gain Practical Skills: Acquire hands-on experience in utilizing binary tree traversals for tasks such as expression evaluation, tree manipulation, pathfinding, and graph algorithms.

8.2 BINARY TREE TRAVERSALS

Binary tree traversals involve systematically visiting each node in a binary tree according to a specified order. The three primary traversal methods are inorder, preorder, and postorder. In inorder traversal, nodes are visited in a left-root-right sequence, making it useful for accessing nodes in sorted order in a BST. Preorder traversal visits the root before its left and right children, making it suitable for creating a copy of a tree or prefix expression evaluation. Postorder traversal visits the left and right children before the root, often used for deleting a tree or evaluating postfix expressions. These traversal techniques are fundamental for accessing, modifying, or analyzing binary tree structures in various computational tasks and algorithms.

Inorder Traversal: Visit left subtree, then root, then right subtree. Inorder traversal is a method used to visit nodes in a binary tree where each node is recursively visited in the order: left subtree, root, right subtree. This traversal method is particularly useful for binary search trees (BSTs) as it visits nodes in ascending order of their keys.

Algorithm

The recursive algorithm for inorder traversal can be defined as follows:

Base Case: If the current node is null (empty tree), return.

Recursive Step:

Recursively traverse the left subtree.

Visit (print, process, or store) the current node's value.

Recursively traverse the right subtree.

This approach ensures that nodes are visited in the correct order according to the properties of inorder traversal.

Example

Let's illustrate the inorder traversal algorithm with a simple C++ implementation using a class TreeNode for the tree nodes:

```
#include <iostream>
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}
};
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // Traverse the left subtree
    inorderTraversal(root->left);
    // Visit the current node (print its value)
    std::cout << root->val << " ";
    // Traverse the right subtree
    inorderTraversal(root->right);
}
// Example usage
int main() {
    // Constructing a sample binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    std::cout << "Inorder traversal: ";
    inorderTraversal(root);
    std::cout << std::endl;
    return 0;
}
```

Explanation

In the above example:

We define the TreeNode class to represent nodes of the binary tree.

The inorderTraversal function is a recursive function that performs inorder traversal.

Starting from the root node (root), it recursively traverses the left subtree (root->left), then visits the current node (root->val), and finally recursively traverses the right subtree (root->right).

The traversal prints node values in ascending order due to the nature of inorder traversal, resulting in output: 4 2 5 1 3.

Preorder Traversal: Visit root, then left subtree, then right subtree.

Preorder traversal is a method used to visit nodes in a binary tree where each node is recursively visited in the order: root, left subtree, right subtree. This traversal method is useful for creating a copy of the tree, prefix expression evaluation, or constructing prefix notation from infix notation.

Algorithm

The recursive algorithm for preorder traversal can be defined as follows:

Base Case: If the current node is null (empty tree), return.

Recursive Step:

Visit (print, process, or store) the current node's value.

Recursively traverse the left subtree.

Recursively traverse the right subtree.

This approach ensures that the root node is visited before its left and right subtrees.

Example in Python

Let's illustrate the preorder traversal algorithm with an example implementation in C++:

```
#include <iostream>
// Definition of TreeNode
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}
};
// Function to perform preorder traversal
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // Visit the current node
    std::cout << root->val << " ";
    // Traverse the left subtree
    preorderTraversal(root->left);
    // Traverse the right subtree
    preorderTraversal(root->right);
}
// Main function for example usage
int main() {
    // Constructing a sample binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    // Perform preorder traversal
    std::cout << "Preorder traversal: ";
    preorderTraversal(root);
    std::cout << std::endl;
    return 0;
}
```

Explanation:

TreeNode Class: Defines a simple binary tree node with an integer value (val) and pointers to left (left) and right (right) children.

preorderTraversal Function:

Recursively traverses the binary tree in preorder.

Prints the value of the current node (root->val) before recursively calling preorderTraversal on its left and right children (root->left and root->right).

Main Function:
Constructs a sample binary tree with values 1, 2, 3, 4, and 5.
Calls preorderTraversal starting from the root node (TreeNode (1)).
Outputs the result of the preorder traversal, which in this case would be: 1 2 4 5 3.

Postorder Traversal: Visit left subtree, then right subtree, then root.
Postorder traversal is a method used to visit nodes in a binary tree where each node is recursively visited in the order: left subtree, right subtree, root. This traversal method is useful for deleting a tree, evaluating postfix expressions, or performing certain types of bottom-up processing.

Algorithm
The recursive algorithm for postorder traversal can be defined as follows:
Base Case: If the current node is null (empty tree), return.

Recursive Step:
Recursively traverse the left subtree.
Recursively traverse the right subtree.
Visit (print, process, or store) the current node's value.
This approach ensures that the root node is visited after its left and right subtrees have been fully explored.

Example in C++

Here's how you can implement postorder traversal of a binary tree in C++ using a class TreeNode for the nodes:

```
#include <iostream>
// Definition of TreeNode
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}
};
// Function to perform postorder traversal
void postorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // Traverse the left subtree
    postorderTraversal(root->left);
    // Traverse the right subtree
    postorderTraversal(root->right);
    // Visit the current node
    std::cout << root->val << " ";
}
// Main function for example usage
int main() {
    // Constructing a sample binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    // Perform postorder traversal
    std::cout << "Postorder traversal: ";
    postorderTraversal(root);
    std::cout << std::endl;
    return 0;
}
```

Level-order Traversal: Visit nodes level by level, left to right. Level-order traversal, also known as breadth-first traversal, is a method used to visit nodes in a binary tree where each level of the tree is visited before moving on to the next level. This traversal method explores nodes level by level, from left to right, making it suitable for tasks such as level-wise printing or searching in a tree structure.

Algorithm

Level-order traversal can be implemented using a queue data structure to keep track of nodes at each level:

Initialize: Start with a queue initialized with the root node.
Process Nodes: Dequeue a node from the front of the queue, visit (print, process, or store) its value.

Enqueue Children: Enqueue its left and right children (if they exist) into the queue.
Repeat: Continue this process until the queue is empty.
Example in C++

Here's how you can implement level-order traversal of a binary tree in C++ using a class TreeNode for the nodes and std::queue for managing the traversal:

```
#include <iostream>
#include <queue>
```

```
// Definition of TreeNode
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}
};
// Function to perform level-order traversal
void levelOrderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // Create a queue for level-order traversal
    std::queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        // Visit the current node
        std::cout << current->val << " ";
        // Enqueue left child
        if (current->left) {
            q.push(current->left);
        }
        // Enqueue right child
        if (current->right) {
            q.push(current->right);
        }
    }
}
// Main function for example usage
```

```
int main() {
    // Constructing a sample binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    // Perform level-order traversal
    std::cout << "Level-order traversal: ";
    levelOrderTraversal(root);
    std::cout << std::endl;
    return 0;
}
```

Explanation:

TreeNode Class: Defines a simple binary tree node with an integer value (val) and pointers to left (left) and right (right) children.

levelOrderTraversal Function:

Implements level-order traversal using a std::queue.

Starts with the root node (root) enqueued.

Dequeues each node (current) from the front of the queue, visits its value (std::cout << current->val << " "), and enqueues its children (if they exist) into the queue.

Continues this process until all nodes at every level have been visited.

Main Function:

Constructs a sample binary tree with values 1, 2, 3, 4, and 5.

Calls levelOrderTraversal starting from the root node (TreeNode(1)).

Outputs the result of the level-order traversal, which in this case would be: 1 2 3 4 5.

8.3 RECURSIVE IMPLEMENTATION OF BINARY TREE TRAVERSALS

Recursive implementation of binary tree traversal refers to the method of visiting each node in a binary tree using recursive function calls. There are three primary types of binary tree traversals: inorder, preorder, and postorder.

In each traversal method, the recursive function ensures that all nodes are visited in the prescribed order, leveraging the function call stack to manage the sequence of node visits. Recursive implementations are typically concise and intuitive for tree traversal, suitable for operations such as printing tree nodes in a specific order, evaluating expressions, or performing depth-first searches in binary tree structures.

The algorithms for recursive binary tree traversals in C++ format:
In **inorder traversal**, the nodes are visited in the sequence: left subtree, root, right subtree. The recursive algorithm starts by checking if the current node is null; if so, it returns. Otherwise, it recursively traverses the left subtree, then visits the current node (e.g., prints its value), and finally recursively traverses the right subtree.

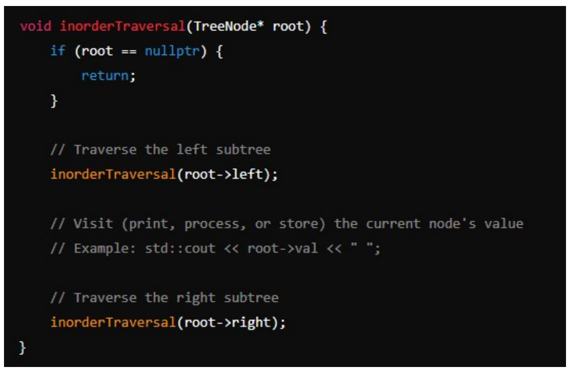
Time Complexity of Recursive Traversals

Each node is visited exactly once.

Time complexity: O(n), where n is the number of nodes in the binary tree.

This is because every node is processed once, and processing each node takes constant time.

Inorder Traversal Algorithm (C++ Format)



```
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    // Traverse the left subtree
    inorderTraversal(root->left);

    // Visit (print, process, or store) the current node's value
    // Example: std::cout << root->val << " ";

    // Traverse the right subtree
    inorderTraversal(root->right);
}
```

Preorder traversal visits nodes in the sequence: root, left subtree, right subtree. Similarly, the recursive function checks if the current node is null; if not, it visits the current node first, then recursively traverses the left subtree, followed by the right subtree.

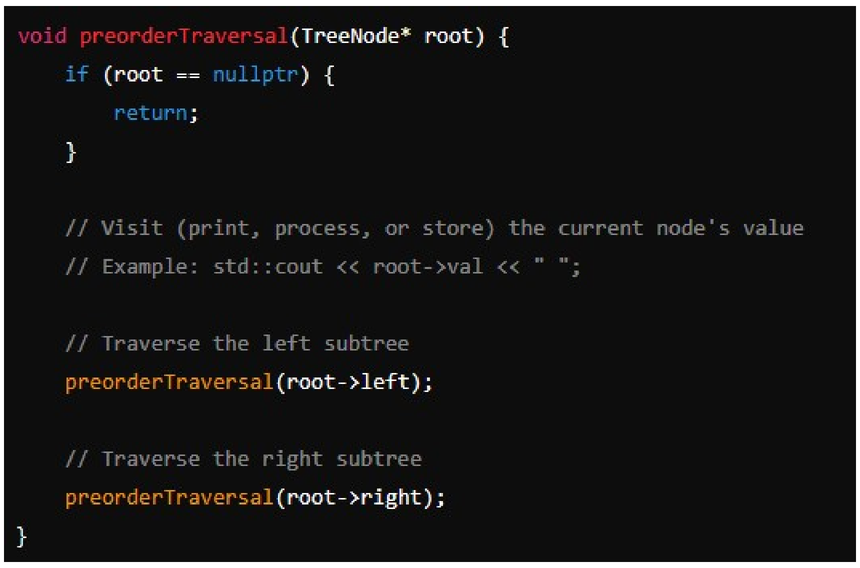
Time Complexity of Recursive Traversals

Each node is visited exactly once.

Time complexity: O(n), where n is the number of nodes in the binary tree.

Similar to inorder traversal, all nodes are processed once.

Preorder Traversal Algorithm (C++ Format)



```
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    // Visit (print, process, or store) the current node's value
    // Example: std::cout << root->val << " ";

    // Traverse the left subtree
    preorderTraversal(root->left);

    // Traverse the right subtree
    preorderTraversal(root->right);
}
```

Postorder traversal visits nodes in the sequence: left subtree, right subtree, root. The recursive approach begins by recursively traversing the left subtree, then the right subtree, and finally visiting the current node.

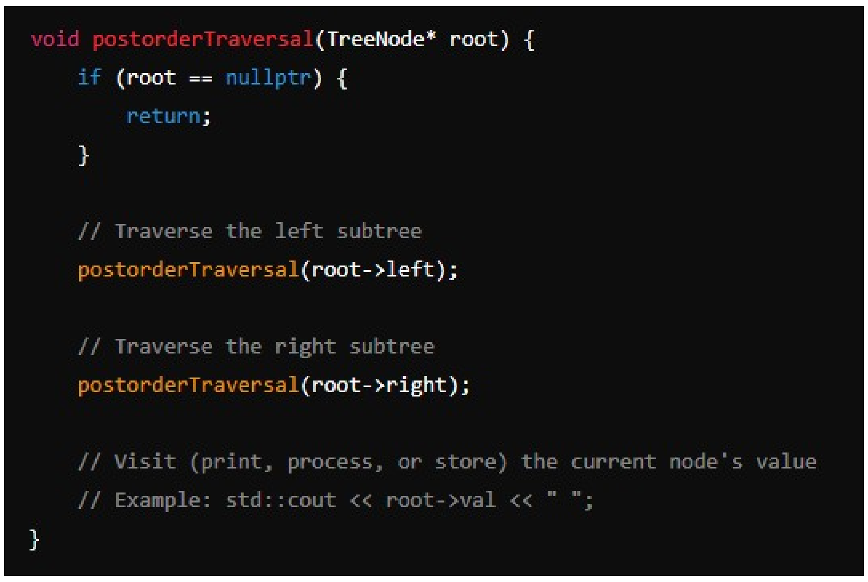
Time Complexity of Recursive Traversals

Each node is visited exactly once.

Time complexity: O (n), where nnn is the number of nodes in the binary tree.

Like the other traversals, all nodes are processed once.

Postorder Traversal Algorithm (C++ Format)



```
void postorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    // Traverse the left subtree
    postorderTraversal(root->left);

    // Traverse the right subtree
    postorderTraversal(root->right);

    // Visit (print, process, or store) the current node's value
    // Example: std::cout << root->val << " ";
}
```

Explanation

Each function (inorderTraversal, preorderTraversal, postorderTraversal) takes a TreeNode* parameter root, representing the root of the subtree to traverse.

Base Case: if (root == nullptr) checks if the current node (root) is null (empty tree). If true, the function returns immediately, halting further recursion.

Recursive Step: For each traversal:

Inorder: Recursively call inorderTraversal on the left subtree, visit the current node, then recursively call it on the right subtree.

Preorder: Visit the current node, recursively call preorderTraversal on the left subtree, then on the right subtree.

Postorder: Recursively call postorderTraversal on the left subtree, then on the right subtree, and finally visit the current node.

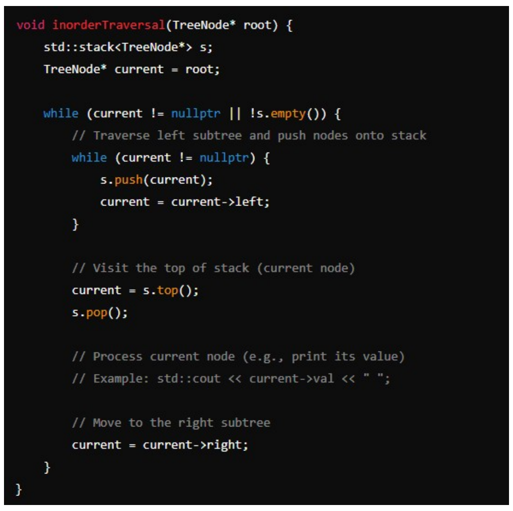
Visit Node: This part of the algorithm is where you would typically perform an action on the current node, such as printing its value (std::cout << root->val << " ";). This action can be adjusted based on the specific requirements of your application.

8.4 NON-RECURSIVE IMPLEMENTATIONS OF BINARY TREE TRAVERSALS

The algorithmic outlines for non-recursive implementations of binary tree traversals:

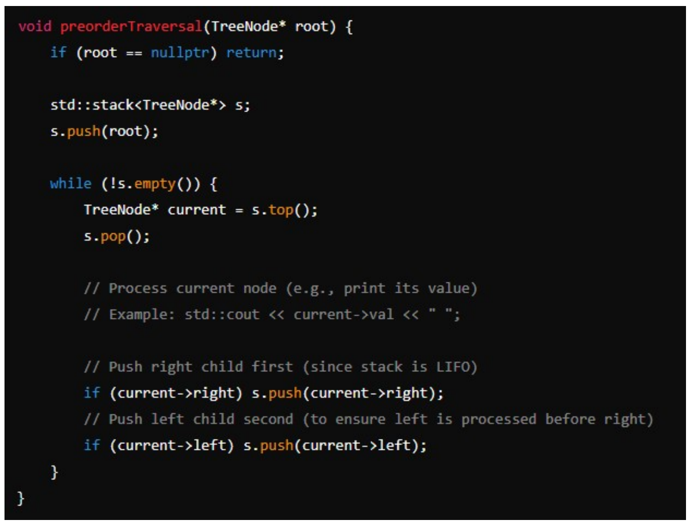
Non-Recursive Algorithms for Binary Tree Traversals

Inorder Traversal Algorithm (Non-Recursive)



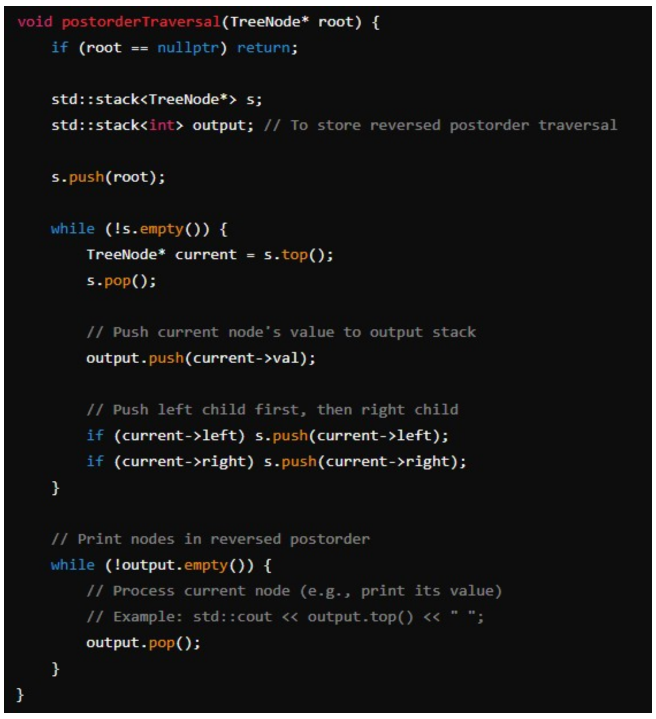
```
void inorderTraversal(TreeNode* root) {
    std::stack<TreeNode*> s;
    TreeNode* current = root;

    while (current != nullptr || !s.empty()) {
        // Traverse left subtree and push nodes onto stack
        while (current != nullptr) {
            s.push(current);
            current = current->left;
        }

        // Visit the top of stack (current node)
        current = s.top();
        s.pop();

        // Process current node (e.g., print its value)
        // Example: std::cout << current->val << " ";

        // Move to the right subtree
        current = current->right;
    }
}
```

Preorder Traversal Algorithm (Non-Recursive)



```
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;

    std::stack<TreeNode*> s;
    s.push(root);

    while (!s.empty()) {
        TreeNode* current = s.top();
        s.pop();

        // Process current node (e.g., print its value)
        // Example: std::cout << current->val << " ";

        // Push right child first (since stack is LIFO)
        if (current->right) s.push(current->right);
        // Push left child second (to ensure left is processed before right)
        if (current->left) s.push(current->left);
    }
}
```

Postorder Traversal Algorithm (Non-Recursive)



```
void postorderTraversal(TreeNode* root) {
    if (root == nullptr) return;

    std::stack<TreeNode*> s;
    std::stack<int> output; // To store reversed postorder traversal

    s.push(root);

    while (!s.empty()) {
        TreeNode* current = s.top();
        s.pop();

        // Push current node's value to output stack
        output.push(current->val);

        // Push left child first, then right child
        if (current->left) s.push(current->left);
        if (current->right) s.push(current->right);
    }

    // Print nodes in reversed postorder
    while (!output.empty()) {
        // Process current node (e.g., print its value)
        // Example: std::cout << output.top() << " ";
        output.pop();
    }
}
```

Explanation

Inorder Traversal: Uses a stack to simulate the call stack of recursive approach. It traverses left subtree first, processes the current node, and then moves to the right subtree.

Preorder Traversal: Starts from the root node, processes it, and pushes its right and left children onto the stack. This ensures nodes are processed in the correct preorder sequence.

Postorder Traversal: Uses two stacks: the main stack pushes nodes in root-right-left order, and the output stack reverses this order to achieve the postorder sequence.

Compare the recursive and iterative approaches in terms of performance and memory usage.

Recursive Approach

Performance:

Time Complexity: Recursive traversals (inorder, preorder, postorder) typically have a time complexity of O (n), where n is the number of nodes in the binary tree. Each node is visited exactly once.

Space Complexity: The space complexity depends on the maximum depth of the recursion stack, which is O (h) where h is the height of the binary tree. In the best-case scenario (balanced tree), this is O (log n); in the worst case (unbalanced tree), it can be O (n).

Memory Usage:

Recursive calls use memory on the call stack for function calls and local variables. Each recursive call adds a stack frame, which can potentially lead to stack overflow errors if the tree is deeply nested or unbalanced.

Despite potential drawbacks, recursive approaches are often simpler to implement and understand due to their natural recursive nature.

Iterative Approach

Performance:

Time Complexity: Iterative traversals also have a time complexity of O (n), similar to recursive traversals. Each node is processed exactly once.

Space Complexity: Iterative traversals typically use an explicit data structure such as a stack (or queue for level-order traversal). The space complexity is also O (h), where h is the height of the binary tree. This is because the stack or queue stores nodes as they are processed, similar to the depth of recursion in the recursive approach.

Memory Usage:

Iterative approaches often use additional memory for data structures like stacks or queues to manage the order of node processing.

They may be more memory-efficient in some cases compared to recursive approaches, especially in situations where tail-call optimization is not available (as in many programming languages).

8.5 APPLICATIONS OF BINARY TREE TRAVERSALS

Binary tree traversals, both recursive and iterative, are fundamental operations with numerous practical applications across various domains. Here are some key applications of binary tree traversals:

Binary Search Trees (BSTs): Inorder traversal of a BST results in a sorted sequence of elements. This property is utilized for tasks such as generating sorted outputs from data stored in a BST or validating the ordering of elements.

Expression Evaluation: Preorder or postorder traversals are used to evaluate arithmetic expressions stored in binary expression trees (expression trees). Each traversal method corresponds to a different evaluation strategy (prefix, postfix), making it efficient for computational tasks.

Path Finding and Reconstruction: Traversals are employed to reconstruct or find paths within binary trees. For example, determining the path from the root to a specific node or finding all root-to-leaf paths in the tree.

Binary Tree Operations: Traversals facilitate various operations such as cloning a tree (preorder), transforming a tree structure (inorder), or deleting nodes (postorder). These operations leverage the sequential access provided by traversals to manipulate tree data effectively.

Binary Tree Serialization and Deserialization: Preorder or postorder traversals are used to serialize binary trees into a linear data format (e.g., arrays or strings). This serialized format can be stored or transmitted across networks and later deserialized back into a binary tree.

Graph Algorithms: Binary tree traversals serve as a basis for several graph algorithms, such as depth-first search (DFS), which explores vertices in a similar manner to preorder traversal. Applications include finding connected components, cycle detection, and topological sorting in directed acyclic graphs (DAGs).

8.6 CONCLUSION

Binary tree traversals represent a cornerstone in the study of data structures and algorithms, offering essential tools for navigating and manipulating hierarchical data efficiently. Throughout this exploration, we have delved into the intricacies of inorder, preorder, and postorder traversals, each methodically visiting nodes in distinct sequences that serve various computational purposes.

Recursive implementations of these traversals provide a clear and intuitive approach, leveraging function calls to explore tree structures depth-first. They offer simplicity in algorithmic design but necessitate careful consideration of stack space in deeply nested trees. In contrast, non-recursive approaches utilize explicit data structures like stacks or queues to achieve iterative traversal, offering finer control over memory usage and stack depth.

From practical applications in expression evaluation and tree manipulation to supporting complex graph algorithms and optimizing data structures like binary search trees, binary tree traversals find wide-ranging utility. They empower efficient solutions across domains, enhancing computational efficiency and enabling sophisticated data processing tasks.

As we conclude, understanding the nuances of binary tree traversals equips us with essential skills for tackling algorithmic challenges, optimizing code performance, and developing robust software solutions. Mastery of these traversal techniques not only enriches our understanding of data structures but also fosters creativity in algorithm design, ensuring proficiency in navigating the complexities of binary tree structures.

8.7 QUESTIONS AND ANSWERS

1. What are the three main types of binary tree traversals?

Answer: The three main types of binary tree traversals are:

Inorder traversal: Visit left subtree, then current node, then right subtree.

Preorder traversal: Visit current node, then left subtree, then right subtree.

Postorder traversal: Visit left subtree, then right subtree, then current node.

2. How does inorder traversal of a binary search tree differ from preorder and postorder traversals?

Answer: In an inorder traversal of a binary search tree (BST), the nodes are visited in ascending order of their keys. This property makes inorder traversal useful for generating sorted lists from BSTs. In contrast, preorder and postorder traversals follow different sequences: preorder visits the root before its subtrees, while postorder visits the root after its subtrees.

3. What is the advantage of using iterative approaches for binary tree traversals over recursive methods?

Answer: Iterative approaches using stacks or queues offer more control over memory usage, especially in environments where stack depth is a concern (such as in deeply nested trees). They can also be more efficient in terms of space utilization and are suitable for iterative modifications or optimizations of tree structures.

4. Give an example where postorder traversal of a binary tree is particularly useful.

Answer: Postorder traversal is useful in scenarios where operations need to be performed on subtrees before processing the root node. For example, in deleting a binary tree, postorder traversal ensures that child nodes are deleted before their parent nodes, preventing memory leaks and ensuring proper cleanup.

5. How does the time complexity of binary tree traversals compare to each other?

Answer: All three types of binary tree traversals (inorder, preorder, postorder) have a time complexity of O (n), where nnn is the number of nodes in the binary tree. This is because each node is visited exactly once during traversal, making them equally efficient in terms of time complexity.

8.8 REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd Edition)*. MIT Press.

Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++ (4th Edition)*. Pearson Education.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in C++ (2nd Edition)*. John Wiley & Sons.

Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th Edition)*. Addison-Wesley.

Sahni, S. (2006). *Data Structures, Algorithms, and Applications in C++*. McGraw-Hill Education.

BLOCK III: GRAPH ALGORITHMS AND SEARCHING TECHNIQUES

UNIT – 9: ADVANCED TREES

Structure

9.0 INTRODUCTION

AVL trees are a type of self-balancing binary search tree named after their inventors, Georgy Adelson-Velsky and Evgenii Landis. They maintain a balanced structure to ensure that the height of the tree remains logarithmic with respect to the number of nodes, which allows for efficient operations such as insertion, deletion, and searching. The balance of an AVL tree is managed through rotations, which ensure that the tree remains balanced after modifications. This balancing mechanism makes AVL trees particularly suitable for applications requiring frequent updates and efficient data retrieval.

In this chapter, we will explore the fundamentals of AVL trees, including the different types of rotations used to maintain balance. We will delve into the implementation of various AVL tree operations, such as insertion and deletion, and discuss their algorithms in detail. Additionally, we will examine practical applications of AVL trees, highlighting their significance in various computational and real-world scenarios. Finally, we will provide a set of questions and answers to reinforce the concepts covered, along with references for further reading.

This comprehensive overview aims to equip you with a thorough understanding of AVL trees, their operations, and their applications, providing a solid foundation for further exploration and implementation of this essential data structure.

9.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand the basic concepts and properties of AVL trees and why they are called self-balancing binary search trees.

Learn and implement the different types of rotations (RR, LL, LR, RL) used to maintain the balance in AVL trees.

Implement insertion and deletion operations in AVL trees, ensuring the tree remains balanced through appropriate rotations.

Explore various real-world applications of AVL trees to understand their practical significance and efficiency.

Reinforce your understanding of AVL trees through questions and answers, and refer to additional resources for deeper insights into the topic.

9.2 AVL TREES

In 1962, GM Adelson-Velsky and EM Landis created the AVL Tree. To honor its creators, the tree is called AVL.

The height of each node in an AVL Tree is determined by subtracting the height of its left sub-tree from the height of its right sub-tree, creating a height balanced binary search tree.

If every node's balance factor falls between -1 and 1, the tree is said to be balanced; if not, it needs to be balanced.
Balance Factor (k) = height (left(k)) - height (right(k))

The left sub-tree is one level higher than the right sub-tree if the balancing factor of any node is 1.

Any node whose balancing factor is zero indicates that the heights of the left and right subtrees are equal.
The left sub-tree is one level lower than the right sub-tree if the balancing factor of any node is -1.

The AVL tree is shown in the image below. It is evident that every node has a balance factor that ranges from -1 to +1. It is an AVL tree example as a result.

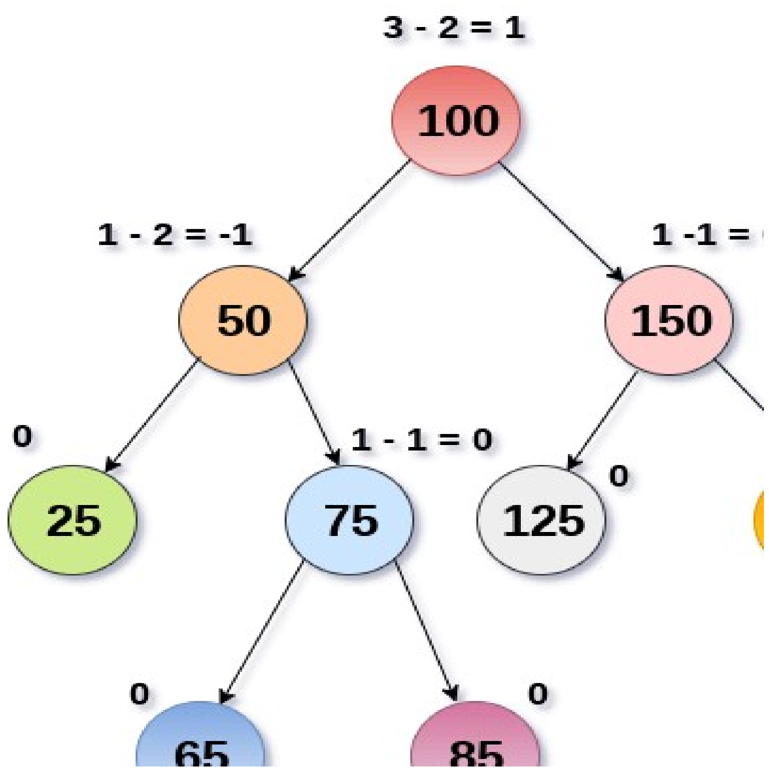


Image Source: Javat Point

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operation on AVL Tree

All operations are carried out in the same manner as they are carried out in a binary search tree as the AVL tree is likewise a binary search tree. There is no property violation of the AVL tree as a result of searching or traversing. Nevertheless, insertion and deletion are the operations that need to be reviewed because they have the potential to break this characteristic.

Insertion: The process of inserting data into an AVL tree is identical to that of inserting data into a binary search tree. It might, however, result in an AVL tree property violation, necessitating the balancing of the tree. Rotations can be used to balance the tree.

Deletion: The process of deletion can be carried out similarly to how it is in a binary search tree. Various rotations are performed to rebalance the tree because deletions can also throw it out of balance.

The AVL Tree: Why?

By preventing skewing, the AVL tree regulates the height of the binary search tree. In a binary search tree of height h, the total processing time is O(h). On the other hand, in the worst-case scenario, if the BST skews, it can be stretched to O(n). The AVL tree sets an upper limitation on each operation to be O(log n), where n is the number of nodes, by restricting this height to log n.

Rotations of AVL

Rotation in the AVL tree is only carried out when the Balance Factor is not equal to -1, 0 or 1. Rotations can be broadly classified into four categories, which are as follows:

L L rotation: Inserted node is in the left subtree of left subtree of A

R R rotation: Inserted node is in the right subtree of right subtree of A

L R rotation: Inserted node is in the right subtree of left subtree of A

R L rotation: Inserted node is in the left subtree of right subtree of A

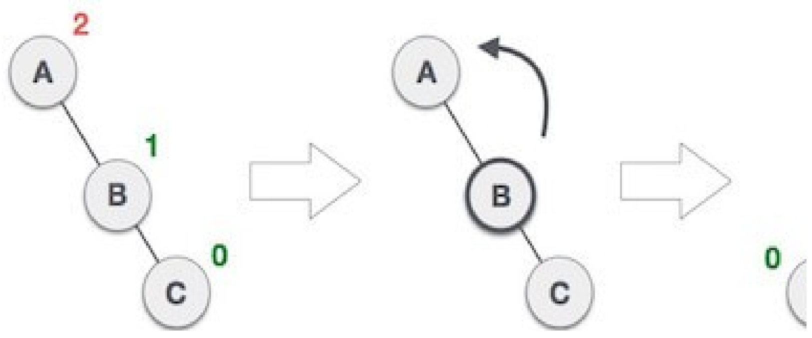
Assuming that node A is the node with a balancing factor that is not -1, 0, 1.

The initial two iterations: The next two rotations, LR and RL, are double rotations, whereas LL and RR are single rotations. A tree must have a minimum height of two in order to be considered imbalanced. Let's examine each revolution.

9.2.1 RR Rotation

We apply RR rotation, an anticlockwise rotation, on the edge beneath a node with a balance factor of -2 when BST becomes unbalanced as a result of a node being placed into the right subtree of the right subtree of A.

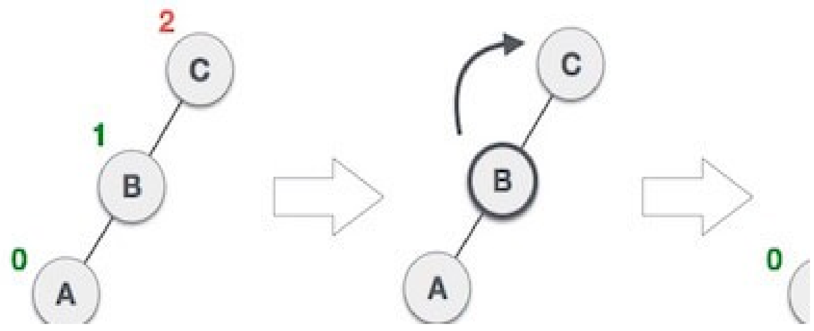
Because node C is added into the right subtree of node A, node A in the example above has a -2 balancing factor. We rotate the RR on the edge beneath A.



9.2.2 LL Rotation

We apply LL rotation, or clockwise rotation, on the edge beneath a node with balance factor 2 when BST becomes unbalanced as a result of a node being added into the left subtree of the left subtree of C.

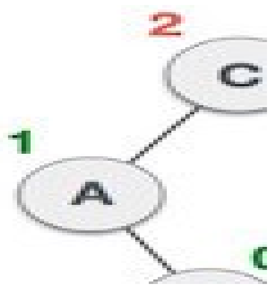
Because node A is inserted into the left subtree of the C left subtree, node C in the example above has a balance factor of 2. The LL rotation is applied to the edge beneath A.



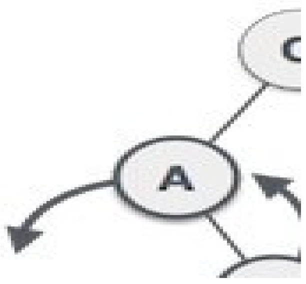
9.2.3 LR Rotation

As was already mentioned, single rotations are a little easier than double rotations. RR rotation is initially applied to the subtree, then LL rotation is applied to the complete tree (which is defined as the first node from the route of the inserted node whose balance factor is not -1, 0, or 1). This means that LR rotation is equal to RR rotation plus LL rotation.

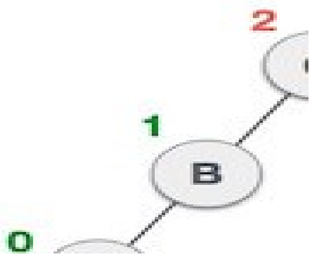
Step – I: Node B was inserted into both the left and right subtrees of C and A, resulting in C being an unbalanced node with a balance factor of 2. In this case of rotation from L to R, where: The inserted node can be found in the right subtree of C's left subtree.



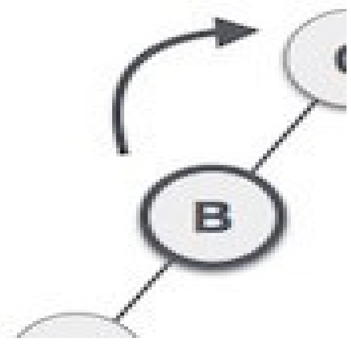
Step – II: Given that LR rotation is equal to RR plus LL rotation, RR (anticlockwise) on the subtree rooted at A is done first. Node A has become the left subtree of B through RR rotation.



Step – III: Because inserted node A is to the left of C, node C is still unbalanced, or has a balance factor of 2, even after RR rotation.



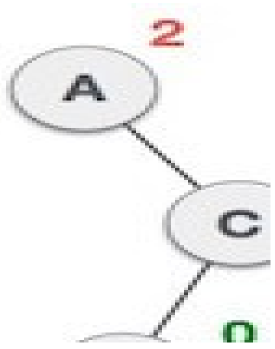
Step – IV: We now rotate the entire tree, or node C, in a clockwise direction using LL. Node C is now node B's right subtree, and node A is node B's left subtree.



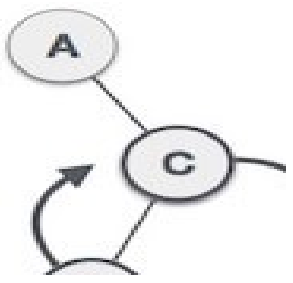
9.2.4 RL Rotation

Double rotations are a little more difficult than single rotations, as was previously mentioned and previously explained. The formula for R L rotation is equal to LL rotation plus RR rotation. This means that LL rotation is applied to the subtree first, then RR rotation is applied to the complete tree (which is defined as the first node from the route of the inserted node whose balancing factor is not equal to -1, 0, or 1).

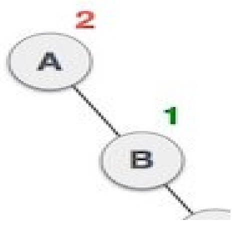
Step – I: Due to the insertion of node B into the right subtree of A and left subtree of C, A is now an unbalanced node with a balance factor of 2. In this RL rotation scenario, where: The node that was inserted is in the left subtree of A's right subtree.



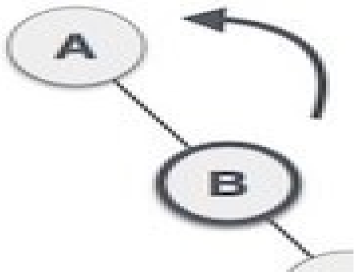
Step – II: Since LL rotation plus RR rotation equals RL rotation, LL (clockwise) rotation on the subtree rooted at C is done first. After performing RR rotation, node C is now B's correct subtree.



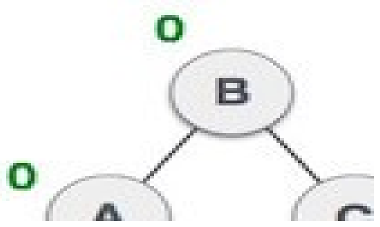
Step – III: Node A remains unbalanced, with a balance factor of -2, even after LL rotation has been performed. This is due to the fact that node A's right-subtree is also its right-subtree.



Step – IV: We now rotate the entire tree, or node A, in an anticlockwise direction, or RR rotation. Now, node A is the left subtree of node B, and node C is the right subtree of node B.



Step – V: Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.



9.3 IMPLEMENTATION OF AVL TREES OPERATIONS

Operations on AVL Trees

Insertion of a Node

Algorithm:

Perform a standard BST insertion.

Update the height of each node from the inserted node to the root.

Check the balance factor of each node.

If the balance factor of any node becomes greater than 1 or less than -1, perform rotations (LL, RR, LR, RL) to balance the tree.

Deletion of a Node

Algorithm:

Perform a standard BST deletion.

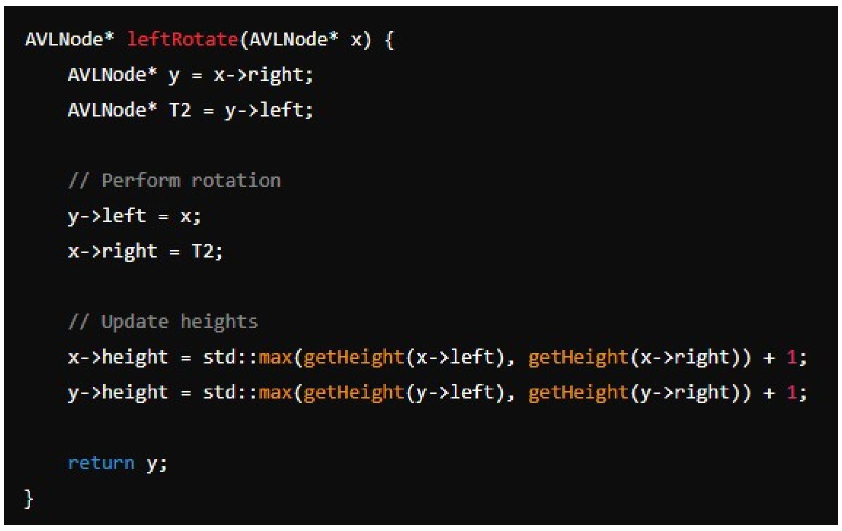
Update the height of each node from the deleted node to the root.

Check the balance factor of each node.

If the balance factor of any node becomes greater than 1 or less than -1, perform rotations to balance the tree.

Left Rotation

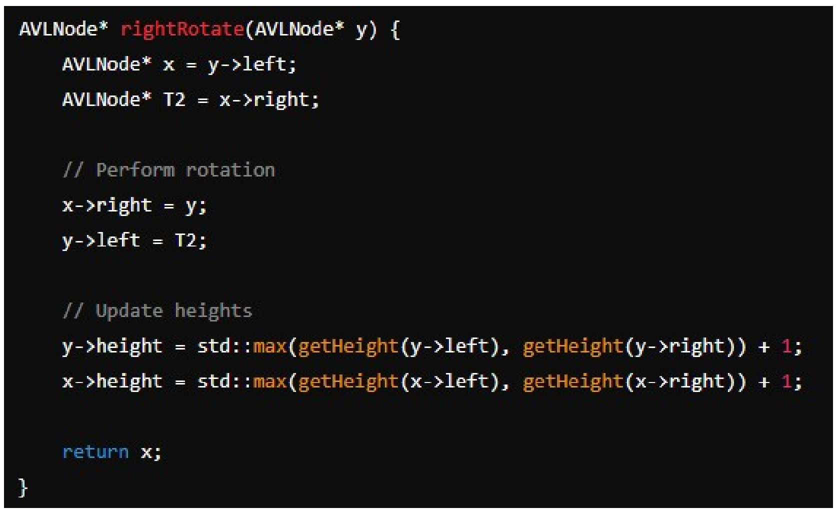
Algorithm:



```
AVLNode* leftRotate(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}
```

Right Rotation

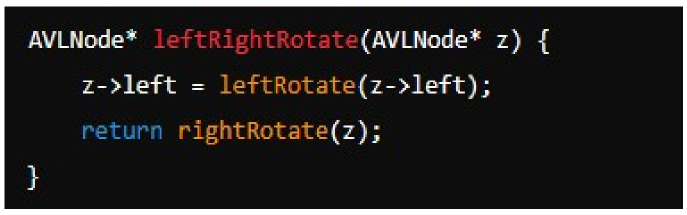
Algorithm:



```
AVLNode* rightRotate(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```

Double Rotation (Left-Right Rotation)

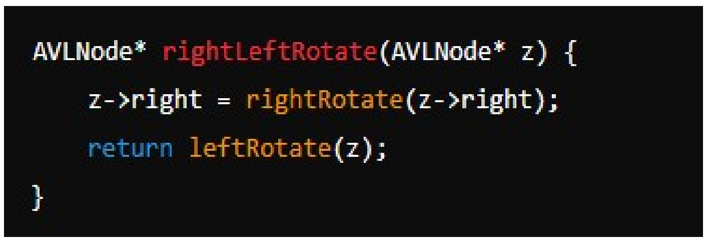
Algorithm:



```
AVLNode* leftRightRotate(AVLNode* z) {
    z->left = leftRotate(z->left);
    return rightRotate(z);
}
```

Double Rotation (Right-Left Rotation)

Algorithm:



```
AVLNode* rightLeftRotate(AVLNode* z) {
    z->right = rightRotate(z->right);
    return leftRotate(z);
}
```

Balancing and Maintenance

Check Balance Factor: Calculate the balance factor (height difference between left and right subtrees) of each node.

Rebalance Tree: After insertions or deletions, check and rebalance the tree using rotations if necessary to maintain AVL properties.

9.4 APPLICATIONS OF AVL TREES

Data Storage and Retrieval

Efficient Searching: AVL trees maintain a balanced structure, ensuring that the height of the tree is logarithmic in the number of nodes. This guarantees that search operations can be performed in

O(log n)O(\log n)O(logn) time, making them highly efficient for data retrieval tasks.

Dynamic Sets: AVL trees are useful in applications where dynamic data sets are frequently updated with insertions and deletions. The self-balancing property ensures that the tree remains balanced after each update, maintaining efficient access times.

Database Indexing

Balanced Index Structures: AVL trees are often used in database indexing to maintain sorted data. The balanced nature of AVL trees ensures that the depth of the index remains low, allowing for quick searches, insertions, and deletions.

Multilevel Indexes: In databases, AVL trees can be used to implement multilevel indexes, where each level of the index is a balanced tree, providing efficient access paths to the data.

Memory Management

Garbage Collection: AVL trees are employed in memory management systems, such as garbage collectors, to keep track of free memory blocks. The balanced structure allows for efficient allocation and deallocation of memory.

Buddy System: In the buddy memory allocation system, AVL trees can be used to manage the free memory blocks, ensuring that the system can quickly find the best-fit block for memory allocation requests.

File Systems

File Indexing: File systems use AVL trees to index files and directories. The balanced nature of AVL trees ensures that file operations such as searching, insertion, and deletion are performed efficiently.

Metadata Management: AVL trees are used to manage file metadata, enabling quick access and updates to file attributes such as permissions, timestamps, and sizes.

Network Routing

Routing Tables: AVL trees can be used in the implementation of routing tables in network routers. The balanced structure allows for efficient lookup, insertion, and deletion of routing entries, ensuring quick and accurate routing decisions.

IP Address Management: AVL trees are useful in managing IP address ranges and routing prefixes, enabling efficient searching and allocation of IP addresses in large networks.

Event Scheduling

Priority Queues: AVL trees can be used to implement priority queues for event scheduling. The balanced structure ensures that events are processed in the correct order of priority, with efficient insertion and extraction operations.

Task Scheduling: In operating systems, AVL trees are used to manage the scheduling of tasks and processes. The balanced nature of the tree ensures that tasks are scheduled and executed efficiently based on their priority and deadlines.

Computational Geometry

Range Searching: AVL trees are employed in computational geometry for range searching problems, where the goal is to efficiently find all points within a given range. The balanced structure allows for quick and efficient searches.

Intersection Detection: AVL trees are used to detect intersections of geometric objects such as lines and polygons. The efficient insertion and deletion operations facilitate the dynamic updating of the geometric structure.

9.5 CONCLUSION

AVL trees play a crucial role in ensuring efficient data management through their self-balancing properties. By maintaining a balanced structure, AVL trees guarantee logarithmic time complexity for insertion, deletion, and search operations, which is essential for applications requiring frequent updates and rapid data retrieval. The use of rotations, such as RR, LL, LR, and RL, is fundamental in preserving this balance after modifications, demonstrating the sophisticated nature of AVL trees compared to simple binary search trees.

Throughout this chapter, we have explored the intricacies of AVL trees, starting with the basic concepts and advancing to the implementation of various operations. We have examined how rotations help in maintaining the balance and efficiency of AVL trees. Furthermore, we have discussed the practical applications of AVL trees in fields like database indexing, memory management, and network routing, showcasing their versatility and importance in real-world scenarios.

By understanding and implementing AVL trees, you gain a valuable tool for optimizing data structures in your applications. This chapter has equipped you with the necessary knowledge and skills to apply AVL trees effectively, ensuring that your data operations are performed efficiently and reliably.

9.6 QUESTIONS AND ANSWERS

1. What is an AVL tree?

Answer: An AVL tree is a self-balancing binary search tree named after its inventors, Georgy Adelson-Velsky and Evgenii Landis. It maintains its balance by ensuring the height difference between the left and right subtrees of any node is no more than one.

2. Why are AVL trees considered self-balancing?

Answer: AVL trees are considered self-balancing because they automatically perform rotations to maintain a balanced structure after insertion and deletion operations, ensuring the height difference (balance factor) between the left and right subtrees of any node is -1, 0, or +1.

3. What is the balance factor in an AVL tree?

Answer: The balance factor of a node in an AVL tree is the difference between the height of its left subtree and the height of its right subtree. It helps in determining whether the tree needs rebalancing through rotations.

4. Explain the RR rotation in AVL trees.

Answer: RR (Right-Right) rotation is a single left rotation used to rebalance an AVL tree when a node's right subtree is heavier (i.e., its balance factor is -2) and the right child has a balance factor of -1 or 0. This rotation shifts the unbalanced subtree to the left.

5. Describe the LL rotation in AVL trees.

Answer: LL (Left-Left) rotation is a single right rotation used to rebalance an AVL tree when a node's left subtree is heavier (i.e., its balance factor is +2) and the left child has a balance factor of +1 or 0. This rotation shifts the unbalanced subtree to the right.

6. What is the difference between LR and RL rotations in AVL trees?

Answer: LR (Left-Right) rotation is a double rotation, first a left rotation on the left child and then a right rotation on the node, used when the balance factor of the node is +2 and the left child has a balance factor of -1. RL (Right-Left) rotation is also a double rotation, first a right rotation on the right child and then a left rotation on the node, used when the balance factor of the node is -2 and the right child has a balance factor of +1.

9.7 REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd Edition)*. MIT Press.

Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++ (4th Edition)*. Pearson Education.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in C++ (2nd Edition)*. John Wiley & Sons.

Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th Edition)*. Addison-Wesley.

Sahni, S. (2006). *Data Structures, Algorithms, and Applications in C++*. McGraw-Hill Education.

UNIT – 10: B-TREES

Sturcture

10.0 INTRODUCTION

In the realm of data structures, trees play a pivotal role in organizing and managing hierarchical data efficiently. Among the diverse types of trees, balanced trees stand out for their ability to maintain balanced structures that ensure optimal performance for various operations. This chapter explores several important balanced trees, including B-Trees, Splay Trees, Red-Black Trees, and AA-Trees, along with their properties, operations, applications, and the broader significance of balanced trees in computer science.

Balanced Trees are designed to keep the tree height proportional to the logarithm of the number of nodes, ensuring that operations such as search, insertions, and deletions remain efficient even as the dataset grows. These trees are essential in scenarios where maintaining balance is critical to performance, such as in databases, file systems, and compilers. Understanding the principles and applications of balanced trees equips us with powerful tools for optimizing data structures in real-world applications.

This chapter will delve into the intricacies of various balanced trees, exploring their structural properties, algorithms for balancing and rebalancing, and practical applications. By the end, you will gain a comprehensive understanding of how these trees contribute to efficient data management and algorithmic design, laying a foundation for advanced studies and applications in computer science.

10.1 OBJECTIVES

After completing this unit, you will be able to understand,

Explore B-Trees: Understand the structure and properties of B-Trees, including rules for balancing.

Study Splay Trees: Learn operations such as splaying, insertion, deletion, and search in Splay Trees.

Examine Red-Black Trees: Understand the properties and rules that define Red-Black Trees as balanced binary search trees.

Understand AA-Trees: Learn about AA-Trees, a variant of Red-Black Trees with simplified balancing rules.

Analyze Applications of Balanced Trees: Investigate practical uses of balanced trees in databases, file systems, and compilers.

10.2 B-TREE

A B-Tree is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. It is designed to work well on systems that read and write large blocks of data, such as databases and filesystems. B-Trees are characterized by their ability to manage large amounts of data by keeping all leaf nodes at the same depth, ensuring that the tree remains balanced. This balance ensures that the time complexity for insertion, deletion, and search operations remains logarithmic. In a B-Tree of order m, each node can have at most m children and must have at least ⌈m/2⌉ children, except for the root node which can have fewer children. The keys within each node are maintained in sorted order, and internal nodes act as guides to direct searches to the correct subtree. This structure allows B-Trees to efficiently handle large datasets and makes them particularly suitable for use in database indexing and filesystems, where quick access to large volumes of data is crucial.

10.2.1 Properties of B-Trees

Order of B-Tree (m): The order m of a B-Tree defines the maximum number of children a node can have. An internal node in a B-Tree of order m can have at most m children.

Key Range in Nodes:

Each node (except for the root and leaves) must have at least ⌈m/2⌉ children and ⌈m/2⌉ - 1 keys.

The root node must have at least 2 children if it is not a leaf node.

A non-leaf node with k children must contain k-1 keys.

Balance: B-Trees are balanced, meaning all leaf nodes are at the same depth, ensuring that the tree remains balanced and the operations (insertion, deletion, search) have logarithmic time complexity.

Height of the Tree: The height of a B-Tree with n keys and minimum degree t is at most log_t(n+1)/2.

Nodes and Keys:
Nodes in a B-Tree contain multiple keys and children pointers. Keys within each node are sorted in non-decreasing order.
Internal nodes store keys to guide the search operations by directing them to the appropriate child subtree.

Root Node: The root node of a B-Tree has at least one key and can have as few as two children or more, depending on the order of the tree.
Leaf Nodes: All leaf nodes appear at the same level and do not contain children. They only store keys.

Node Splitting: When a node becomes full (i.e., contains m-1 keys), it splits into two nodes. The median key is moved up to the parent node, ensuring that the properties of the B-Tree are maintained.

Node Merging: During deletion, if a node has fewer than ⌈m/2⌉ - 1 keys, it may borrow a key from its sibling or merge with a sibling to maintain the minimum number of keys required.

Efficiency: B-Trees are optimized for systems that read and write large blocks of data. They are widely used in database systems and filesystems to ensure efficient data access and management.

B-Trees are balanced search trees designed to work well on disks or other direct-access secondary storage devices.
Every node in a B-Tree contains several keys and children, and all leaves are at the same depth.

A B-Tree of order m is defined as:

Each node has at most m children.

Each internal node (except the root) has at least ⌈m/2⌉ children.

Each non-leaf node has at least ⌈m/2⌉ - 1 keys.

The root has at least two children if it is not a leaf node.

All leaves appear on the same level.

A non-leaf node with k children contains k-1 keys.

10.2.2 Operations on B-Trees

1. Insertion

Algorithm:

Start at the root node.
If the root is full, split it and make the new root its parent, then proceed with insertion.
Traverse down the tree to find the appropriate leaf node.
If the leaf node is full, split it into two nodes and move the middle key up to the parent.
Insert the new key into the appropriate position in the non-full node.

Insertion Algorithm in Detail:

Insert (k):

If the root is full, create a new root and split the old root, and set the new root as the parent of the old root.
Traverse the tree from the root to the appropriate leaf node.

Insert the key k into the non-full node.

Split (x, i):

Split the child x.child[i] of node x into two nodes.

Create a new node z that contains the second half of the keys and children from x.child[i].

Move the median key of x.child[i] up to x, making space in x for the new child pointer.

Example:

```
void BTreeInsert(BTreeNode *root, int k) {
    if (root->n == 2*t - 1) {
        BTreeNode *s = new BTreeNode(false);
        s->C[0] = root;
        BTreeSplitChild(s, 0, root);
        BTreeInsertNonFull(s, k);
        root = s;
    } else {
        BTreeInsertNonFull(root, k);
    }
}
void BTreeInsertNonFull(BTreeNode *x, int k) {
    int i = x->n - 1;
    if (x->leaf) {
        while (i >= 0 && x->keys[i] > k) {
            x->keys[i+1] = x->keys[i];
            i--;
        }
        x->keys[i+1] = k;
        x->n++;
    } else {
        while (i >= 0 && x->keys[i] > k) {
            i--;
        }
        i++;
        if (x->C[i]->n == 2*t - 1) {
            BTreeSplitChild(x, i, x->C[i]);
            if (x->keys[i] < k) {
                i++;
            }
        }
        BTreeInsertNonFull(x->C[i], k);
    }
}
void BTreeSplitChild(BTreeNode *x, int i, BTreeNode *y) {
    BTreeNode *z = new BTreeNode(y->leaf);
    z->n = t - 1;
    for (int j = 0; j < t - 1; j++) {
        z->keys[j] = y->keys[j+t];
    }
    if (!y->leaf) {
        for (int j = 0; j < t; j++) {
            z->C[j] = y->C[j+t];
        }
    }
    y->n = t - 1;
    for (int j = x->n; j >= i+1; j--) {
        x->C[j+1] = x->C[j];
    }
    x->C[i+1] = z;
    for (int j = x->n-1; j >= i; j--) {
        x->keys[j+1] = x->keys[j];
    }
    x->keys[i] = y->keys[t-1];
    x->n++;
}
```

Example:

Insert keys 10, 20, 5, 6, 12, 30, 7, 17 into a B-Tree of order 3.

2. Deletion

Algorithm:

Start at the root node and locate the key to be deleted.

If the key is in a leaf node, delete it directly.

If the key is in an internal node, replace it with its predecessor or successor and delete the key.

If the node has fewer than ⌈m/2⌉ - 1 keys after deletion, perform redistribution or merging:

If a sibling has more than ⌈m/2⌉ - 1 keys, redistribute keys.

If no sibling has extra keys, merge the node with a sibling.

Deletion Algorithm in Detail:

Delete (k):

If k is in the leaf node, remove k from the node.

If k is in the internal node:

If the predecessor child node has at least t keys, replace k with the predecessor key.

Otherwise, if the successor child node has at least t keys, replace k with the successor key.

Otherwise, merge k and its two children into a single node.

If the child has fewer than t keys, ensure that the child has at least t keys by borrowing from the sibling or merging.

Example:

Delete keys 6, 13 from the B-Tree obtained in the insertion example.

3. Search

Algorithm:

Start at the root node.

Compare the key with the keys in the current node.

If the key is found, return the key and the node.

If the key is not found and the node is a leaf, the key does not exist in the tree.

If the key is not found and the node is not a leaf, recursively search the appropriate child node.

Example:

Search for keys 6, 15, 30 in the B-Tree obtained after insertions.

10.2.3 Applications of B-Trees

B-Trees are widely used in scenarios that require efficient insertion, deletion, and searching operations on large amounts of data. Some of the key applications of B-Trees include:

File Systems: B-Trees are extensively used in file systems to manage large amounts of disk blocks efficiently. File systems like NTFS (New Technology File System) and HFS+ (Hierarchical File System Plus) use B-Trees to store file metadata such as file names, sizes, permissions, and pointers to data blocks. The balanced nature of B-Trees ensures that file system operations such as file creation, deletion, and searching are performed efficiently even as the file system grows.

Database Systems: B-Trees are a fundamental data structure in database indexing. They are used to index large datasets based on key values, allowing quick retrieval, insertion, and deletion of records. Database management systems (DBMS) like Oracle, PostgreSQL, and MySQL employ B-Trees to index primary keys, secondary keys, and other indexed columns. This indexing structure enables efficient querying and sorting operations, which are crucial for optimizing database performance.

Persistent Data Structures: B-Trees are suitable for persistent storage environments, such as databases and file systems, where data needs to be stored permanently even after power loss or system restarts. The structure of B-Trees ensures that data can be efficiently written to and read from disk storage, minimizing disk I/O operations and ensuring faster access times compared to other data structures like binary search trees.

Multilevel Indexing: B-Trees are used in multilevel indexing scenarios where the index itself is too large to fit entirely in memory. By organizing index entries into a hierarchical structure of nodes, B-Trees allow efficient traversal through multiple levels of index nodes to quickly locate data blocks or records. This hierarchical indexing scheme reduces the time complexity of search operations compared to linear search methods.

Concurrency Control in Databases: In database systems that support concurrent transactions, B-Trees are used to manage locks and ensure data integrity. B-Trees provide efficient mechanisms for locking individual nodes during concurrent data access and updates, allowing multiple transactions to read and write data simultaneously without causing data inconsistency or conflicts.

Routing Tables in Networks: B-Trees are utilized in computer networking for storing and managing routing tables. In network routers and switches, B-Trees are employed to maintain information about network addresses, routing paths, and next-hop destinations. The balanced structure of B-Trees ensures efficient routing table lookups and updates, enabling fast packet forwarding and routing decision making in large-scale networks.

Compiler Symbol Tables: B-Trees are used in compilers and interpreters to manage symbol tables that store information about variables, functions, and other program entities. Symbol tables implemented with B-Trees allow efficient lookup and manipulation of symbols during compilation and runtime, supporting tasks such as scope resolution, type checking, and code optimization.

10.3 SPLAY TREES

Splay Trees are a self-adjusting binary search tree data structure where every operation, whether it's search, insertion, or deletion, adjusts the tree to bring the accessed node to the root. This characteristic of splaying differentiates it from traditional balanced trees like AVL trees or Red-Black trees, which maintain balance through explicit rotations or color adjustments.

Amortized analysis of operations

Amortized analysis is a method used to determine the average time complexity of a sequence of operations on a data structure, even if some operations may be costlier than others in isolation. It provides a more accurate representation of the overall performance of data structures over time, considering both efficient and potentially costly operations that may occur intermittently.

Key Concepts in Amortized Analysis:

Aggregate Method:

In amortized analysis, the aggregate method considers the total cost of a sequence of operations and divides it by the number of operations to determine the average cost per operation.

This method assumes that some operations may be cheaper than their actual worst-case scenario due to previous operations potentially offsetting higher costs.

Potential Method:

The potential method compares each operation's actual cost to an average cost or potential function.

It calculates how much "potential" or credit is accumulated or spent by each operation, ensuring that the total potential across all operations remains non-negative.

This method is particularly useful for dynamic data structures where the cost of one operation affects future operations.

Amortized Analysis Examples:

Dynamic Arrays (Resizable Arrays):

Operation: Inserting an element into a dynamic array that needs resizing.

Cost: Normally, resizing involves copying elements to a larger array, which is O(n). However, this cost is amortized over multiple insertions.

Amortized Cost: Each insertion operation has an average cost of O(1), considering the occasional resizing operation.

Binary Counters:

Operation: Incrementing a binary counter represented as an array of bits.

Cost: Incrementing may cause a series of cascading flips from 0 to 1, potentially affecting multiple bits.

Amortized Cost: Despite occasional longer sequences of bit flips, the average cost of each increment operation remains O(1) due to the infrequency of longer sequences.

Splay Trees:

Operation: Splaying a node to the root during search, insertion, or deletion.

Cost: The cost of splaying involves rotations and restructuring, which can vary depending on the depth of the node.

Amortized Cost: Over a series of operations, the average cost of splaying is reduced by subsequent operations that benefit from the structure adjustments made during previous splay operations.

Benefits of Amortized Analysis:

Accurate Performance Prediction: It provides a more realistic assessment of the average time complexity of operations, accounting for worst-case scenarios that occur sporadically.

Useful for Dynamic Data Structures: Amortized analysis is particularly valuable for dynamic data structures where operations can vary in complexity depending on the structure's state.

Splaying Steps:

Access Operation:

When searching for a node in a splay tree, the tree undergoes a splaying process where the accessed node moves to the root.

This splaying operation involves a sequence of rotations and restructuring of nodes to promote the accessed node closer to the root.

Splaying Algorithm:

Upon accessing a node during search, the splaying algorithm performs rotations to move the accessed node upwards.

Depending on whether the node to be splayed is a left or right child, single or double rotations (zig-zig or zig-zag rotations) are applied to bring the node to the root.

Balancing:

Unlike balanced trees that maintain a specific height or balance factor, splay trees balance themselves dynamically during operations.

The splaying process ensures that frequently accessed nodes remain closer to the root, optimizing future access times for those nodes.

Insertion and Deletion:

Insertion and deletion in splay trees also involve a splaying process where the inserted or deleted node is splayed to the root.

This self-adjustment ensures that subsequent operations benefit from the recent structural changes, potentially improving overall performance.

Example of Splaying Steps:

Consider a splay tree where we perform a search operation to access a node with key value k. Here are simplified steps for splaying:

Start from the root of the tree.

Traverse down the tree to find the node with key k.

As you traverse, perform rotations and restructuring to move the accessed node towards the root.

After accessing the node with key k, ensure it becomes the root or is placed close to the root through appropriate rotations (zig-zig or zig-zag).

Applications of Splay Trees:

Caching Mechanisms: Splay trees are used in caching scenarios where frequently accessed items are kept in memory for quick retrieval. The self-adjusting nature of splay trees ensures that the most recently accessed cache items remain quickly accessible, optimizing cache hit rates.

Data Compression Algorithms: Splay trees have been used in data compression algorithms where frequent patterns or symbols are dynamically adjusted to the root, enhancing compression efficiency by reducing access times for common patterns.

Adaptive Data Structures: In scenarios where data access patterns are unpredictable or dynamic, splay trees adapt efficiently by adjusting their structure based on recent access history. This adaptability makes them suitable for real-time applications where data access patterns evolve over time.

Implementations in Libraries: Although less common in standard libraries compared to AVL trees or Red-Black trees, splay trees find specialized applications in certain libraries and systems requiring dynamic and adaptive data structures.

10.3.1 Splaying Algorithm

The splaying algorithm is the core procedure used in splay trees to bring a specified node closer to the root, thereby optimizing future access times for that node. Here is a structured outline of the splaying algorithm:

Splaying Algorithm Outline:

Search for the Node:

Start the splaying algorithm by searching for the node with the specified key or value in the splay tree.

Traverse the tree starting from the root and move towards the node that needs to be splayed.

Splay Operation:

Once the node is found or accessed (either through search, insertion, or deletion), begin the splay operation to bring this node closer to the root.

Rotation and Restructuring:

During the splay operation, perform rotations and restructuring of the tree to move the accessed node (X) towards the root.

Rotations are based on the relationship between X, its parent (P), and potentially its grandparent (G) in the tree structure.

Zig-Zig Rotation:

If X and P are both left children or both right children, perform a double rotation (zig-zig rotation) to bring X directly under the root.

Rotate P around G and then X around P.

Zig-Zag Rotation:

If X and P are opposite children (one is a left child and the other is a right child), perform a double rotation (zig-zag rotation) to bring X closer to the root.

Rotate X around P and then rotate X's new parent around G.

Continue Splaying Upwards:

Repeat the rotation and restructuring steps until the accessed node X becomes the root of the splay tree or is positioned close to the root.

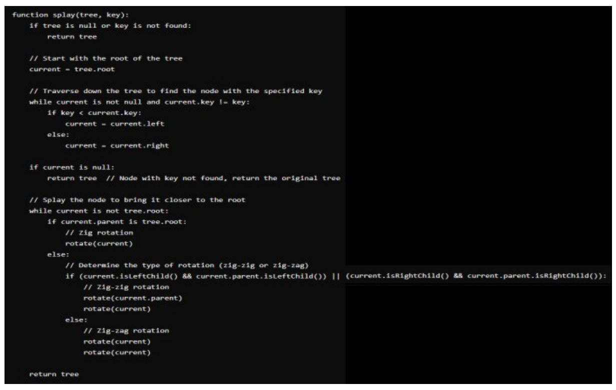
Each rotation aims to move X towards the root, adjusting the tree structure dynamically based on recent access patterns.

Return the Splayed Tree:

After completing the splaying operation, the splay tree structure is updated with the accessed node (X) at or near the root, optimizing future accesses to this node.

Example Splaying Algorithm:

Here’s a simplified pseudocode outline of the splaying algorithm:



```
function splay(tree, key):
    if tree is null or key is not found:
        return tree

    // Start with the root of the tree
    current = tree.root

    // Traverse down the tree to find the node with the specified key
    while current is not null and current.key != key:
        if key < current.key:
            current = current.left
        else:
            current = current.right

    if current is null:
        return tree  // Node with key not found, return the original tree

    // Splay the node to bring it closer to the root
    while current is not tree.root:
        if current.parent is tree.root:
            // Zig rotation
            rotate(current)
        else:
            // Determine the type of rotation (zig-zig or zig-zag)
            if (current.isLeftChild() && current.parent.isLeftChild()) || (current.isRightChild() && current.parent.isRightChild()):
                // Zig-zig rotation
                rotate(current.parent)
                rotate(current)
            else:
                // Zig-zag rotation
                rotate(current)
                rotate(current)

    return tree
```

In this pseudocode:

The rotate(node) function performs the necessary rotations to move node closer to the root based on its relationship with its parent and grandparent.

The splaying algorithm ensures that after accessing or manipulating a node in the tree, it is splayed towards the root, optimizing future access operations.

10.3.2 Operations on Splay Trees

Operations on splay trees include fundamental operations like search, insert, and delete, each of which involves the splaying process to optimize the tree structure based on recent access patterns. Here’s a breakdown of these operations in splay trees:

1. Search Operation:

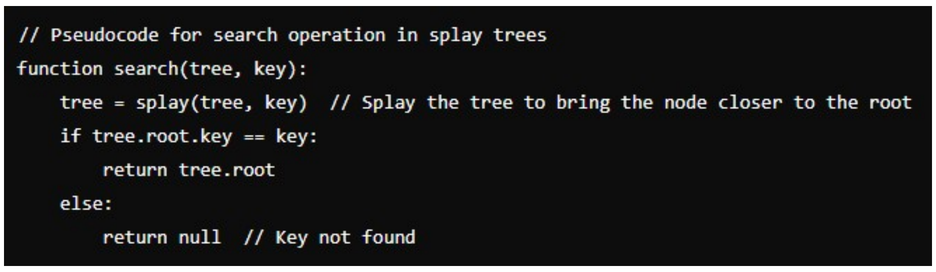
Algorithm:

Start from the root and traverse the tree to find the node with the specified key.

During traversal, perform splaying to bring the accessed node closer to the root.

If the node is found, splay it to the root.

If the node is not found, splay the last accessed node to the root.



```
// Pseudocode for search operation in splay trees
function search(tree, key):
    tree = splay(tree, key)  // Splay the tree to bring the node closer to the root
    if tree.root.key == key:
        return tree.root
    else:
        return null  // Key not found
```

2. Insertion Operation:

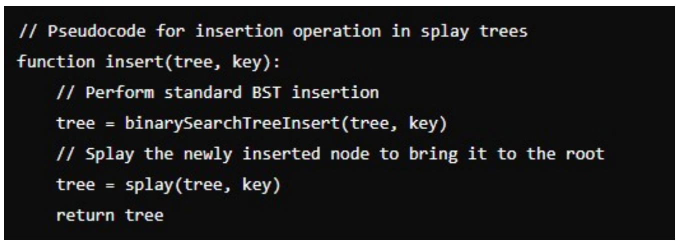
Algorithm:

Perform a standard binary search tree insertion to place the new node in its appropriate position.

After insertion, splay the newly inserted node to bring it to the root.

This step ensures that the most recently inserted node becomes the root, optimizing future accesses.

Example:



```
// Pseudocode for insertion operation in splay trees
function insert(tree, key):
    // Perform standard BST insertion
    tree = binarySearchTreeInsert(tree, key)
    // Splay the newly inserted node to bring it to the root
    tree = splay(tree, key)
    return tree
```

3. Deletion Operation:

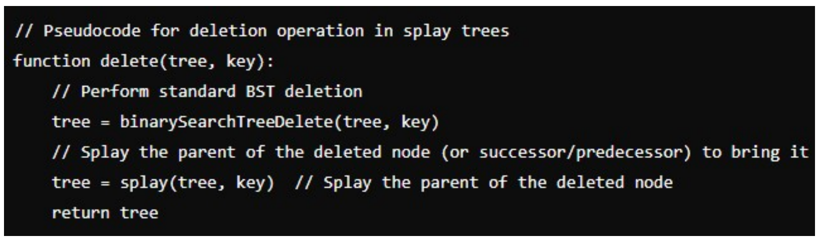
Algorithm:

Perform a standard binary search tree deletion to remove the node with the specified key.

After deletion, splay the parent of the deleted node (or the successor/predecessor node) to bring it to the root.

This step maintains the splay tree properties and optimizes the structure after deletion.

Example:



```
// Pseudocode for deletion operation in splay trees
function delete(tree, key):
    // Perform standard BST deletion
    tree = binarySearchTreeDelete(tree, key)
    // Splay the parent of the deleted node (or successor/predecessor) to bring it
    tree = splay(tree, key)  // Splay the parent of the deleted node
    return tree
```

Key Points:

Splaying Mechanism:

Each operation (search, insert, delete) in a splay tree involves splaying the accessed or manipulated node towards the root. Splaying optimizes the tree structure dynamically based on recent access patterns, ensuring that frequently accessed nodes are closer to the root.

Efficiency: While individual splaying operations can have a worst-case time complexity of O (n) in skewed trees, the amortized time complexity of operations tends to be efficient due to the self-adjusting nature of splay trees.

Adaptability: Splay trees adapt their structure to optimize access times for recently accessed nodes, making them suitable for applications where access patterns are dynamic and unpredictable.

10.4 RED-BLACK TREES

Red-Black trees are self-balancing binary search trees that ensure balanced operations like search, insert, and delete, with a worst-case time complexity of O(log n). They maintain balance using color properties and rotation operations, making them efficient for dynamic data storage and retrieval. Here are the key properties and operations of Red-Black trees:

Properties of Red-Black Trees:

Node Coloring:

Each node in a Red-Black tree is colored either red or black.

The root is always black.

Every red node must have two black children (no consecutive red nodes).

Every path from a node to its descendant null nodes must have the same number of black nodes (black height).

Balanced Height:

Red-Black trees maintain balanced height by ensuring that the longest path from the root to any leaf is no more than twice the shortest path.

This property guarantees O(log n) time complexity for search, insert, and delete operations.

Operations on Red-Black Trees:

Search Operation:

Similar to standard binary search trees, search operations in Red-Black trees follow the properties of binary search, utilizing node colors to maintain balance.

Insertion Operation:

Insertions in Red-Black trees start with a standard BST insertion.

After insertion, the tree may violate Red-Black properties, necessitating restructuring (rotations) and recoloring to restore balance.

The tree is adjusted to maintain Red-Black properties while ensuring the balanced height.

Deletion Operation:

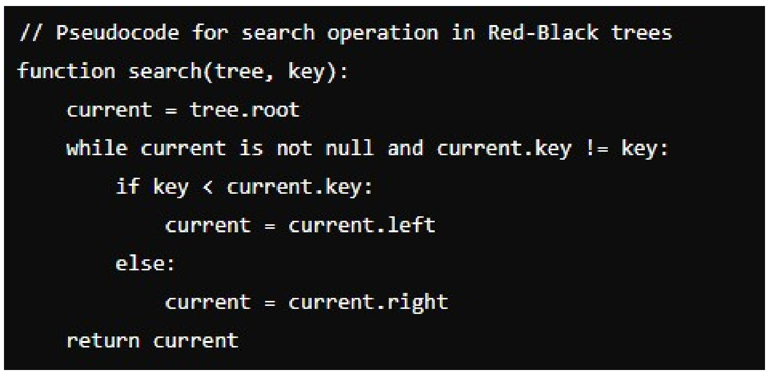
Deletions in Red-Black trees begin with a standard BST deletion.

After deletion, the tree may temporarily violate Red-Black properties.

To restore balance, perform rotations and recoloring operations as necessary to maintain Red-Black properties and balanced height.

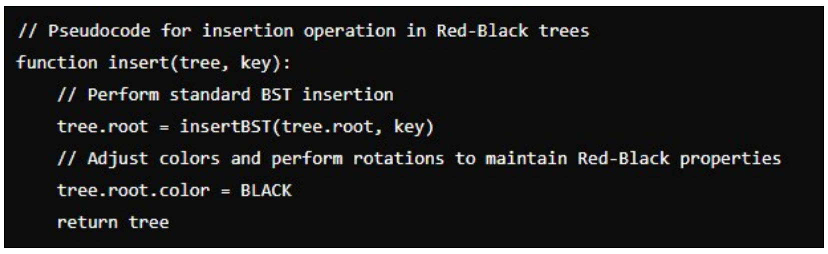
Example of Red-Black Tree Operations:

Search Operation:



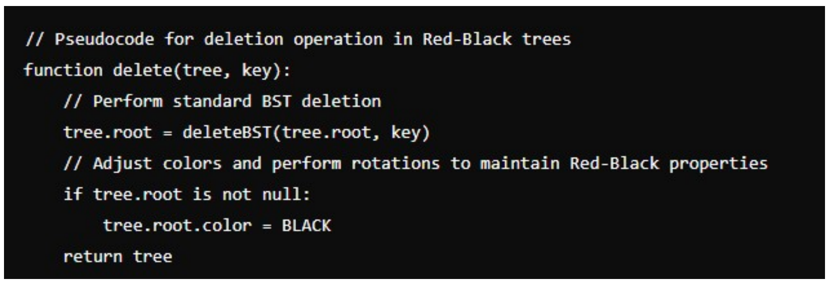
```
// Pseudocode for search operation in Red-Black trees
function search(tree, key):
    current = tree.root
    while current is not null and current.key != key:
        if key < current.key:
            current = current.left
        else:
            current = current.right
    return current
```

Insertion Operation:



```
// Pseudocode for insertion operation in Red-Black trees
function insert(tree, key):
    // Perform standard BST insertion
    tree.root = insertBST(tree.root, key)
    // Adjust colors and perform rotations to maintain Red-Black properties
    tree.root.color = BLACK
    return tree
```

Deletion Operation:



```
// Pseudocode for deletion operation in Red-Black trees
function delete(tree, key):
    // Perform standard BST deletion
    tree.root = deleteBST(tree.root, key)
    // Adjust colors and perform rotations to maintain Red-Black properties
    if tree.root is not null:
        tree.root.color = BLACK
    return tree
```

Advantages of Red-Black Trees:

Balanced Operations: Ensure O(log n) time complexity for search, insert, and delete operations.

Predictable Performance: Provide predictable and efficient performance in dynamic environments.

Widely Used: Commonly used in libraries and applications where efficient data insertion, deletion, and retrieval are crucial.

10.5 AA-TREES

An AA-Tree is a type of self-balancing binary search tree that maintains balance using only a single type of rotation, known as skew and split operations. It ensures that the tree remains balanced by enforcing specific level and structural properties rather than complex color rules or multiple rotation types like Red-Black trees. Here's an overview of AA-Trees, including their properties and operations:

Properties of AA-Trees:

Level Properties:

Every leaf node (null node) is at level 1.

For any node with a left child, the left child must have a level equal to or one less than the node's level.

Nodes without a left child have the same level as their right child.

Skew Operation:

A skew operation is applied to correct consecutive right links (right-right situation).

It rotates the node to the left to balance the tree structure.

After skew operation, the level properties are adjusted to maintain balance.

Split Operation:

A split operation is applied to correct double left links (left-left situation).

It rotates the node to the right and increases its level to balance the tree structure.

After split operation, the level properties are adjusted to maintain balance.

Operations on AA-Trees:

Search Operation:

Similar to standard binary search trees, search operations in AA-Trees follow the properties of binary search, utilizing level properties to maintain balance.

Insertion Operation:

Insertions in AA-Trees start with a standard BST insertion.

After insertion, the tree may violate AA-Tree properties, necessitating skew and split operations to restore balance.

Adjustments are made to ensure that level properties are maintained after each operation.

Deletion Operation:

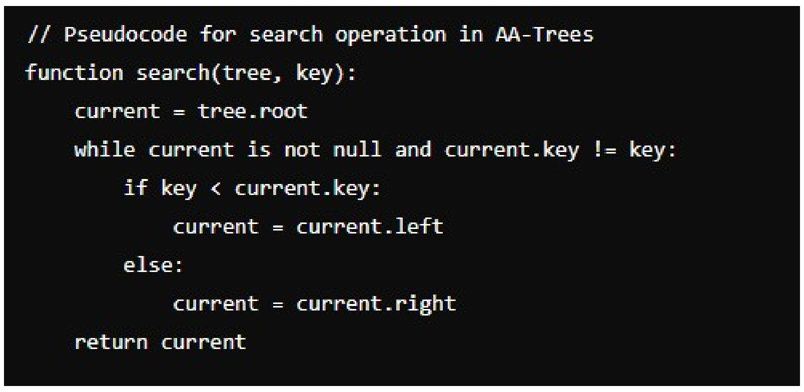
Deletions in AA-Trees begin with a standard BST deletion.

After deletion, the tree may temporarily violate AA-Tree properties.

Skew and split operations are applied as necessary to restore balance and maintain level properties.

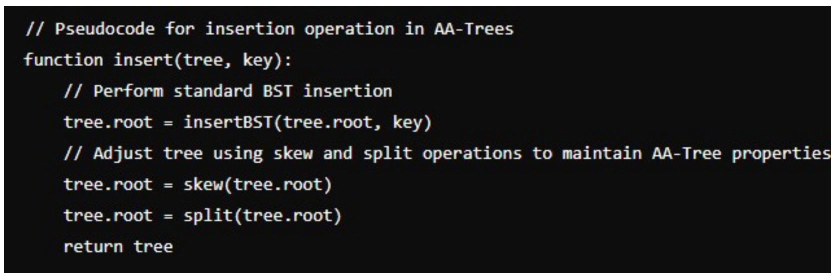
Example of AA-Tree Operations:

Search Operation:



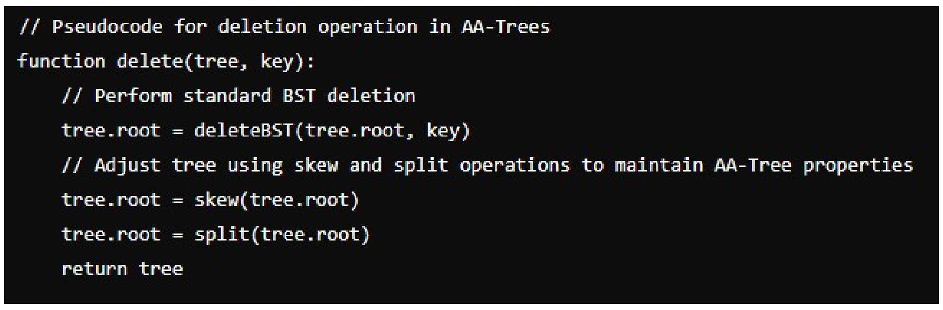
```
// Pseudocode for search operation in AA-Trees
function search(tree, key):
    current = tree.root
    while current is not null and current.key != key:
        if key < current.key:
            current = current.left
        else:
            current = current.right
    return current
```

Insertion Operation:



```
// Pseudocode for insertion operation in AA-Trees
function insert(tree, key):
    // Perform standard BST insertion
    tree.root = insertBST(tree.root, key)
    // Adjust tree using skew and split operations to maintain AA-Tree properties
    tree.root = skew(tree.root)
    tree.root = split(tree.root)
    return tree
```

Deletion Operation:



```
// Pseudocode for deletion operation in AA-Trees
function delete(tree, key):
    // Perform standard BST deletion
    tree.root = deleteBST(tree.root, key)
    // Adjust tree using skew and split operations to maintain AA-Tree properties
    tree.root = skew(tree.root)
    tree.root = split(tree.root)
    return tree
```

Advantages of AA-Trees:

Simplified Balancing: Use only skew and split operations for balancing, which simplifies implementation compared to Red-Black trees.

Efficient Operations: Maintain O(log n) time complexity for search, insert, and delete operations.

Less Overhead: Avoids complex color rules and multiple rotation types, reducing implementation complexity and potential overhead.

10.6 APPLICATIONS OF BALANCED TREES

Balanced trees, including Red-Black trees, AVL trees, B-trees, and AA-trees, find applications in various domains where efficient data storage and retrieval are critical. Here are some common applications of balanced trees:

Databases: B-trees and AVL trees are widely used in database systems for indexing. They provide efficient retrieval of data records based on keys, ensuring that operations like search, insert, and delete are performed in O (log n) time complexity.

File Systems: B-trees are commonly used in file systems to manage large amounts of data efficiently. They ensure that data blocks are organized and accessible in a balanced manner, optimizing disk access and storage.

Compiler Design: Symbol tables in compilers often use balanced trees to store identifiers and their associated attributes. This allows for quick lookup and modification of symbols during compilation.

Networking: Routing tables in computer networks employ balanced trees to store and manage routing information efficiently. This facilitates fast routing decisions and network packet forwarding.

Concurrency Control: In concurrent programming and transaction processing systems, **B-trees** and **Red-Black trees** are used to implement data structures like **transactional maps**. These ensure that data operations are thread-safe and efficient.

Caches and Memory Management: AA-trees and **AVL trees** are used in memory management systems and caches to maintain efficient data retrieval and replacement strategies. They help in managing limited memory resources effectively.

Geospatial and GIS Systems: R-trees, a variant of balanced trees, are used in geospatial databases and Geographic Information Systems (GIS) for indexing and querying spatial data efficiently.

Data Compression: Balanced trees are used in **Huffman coding**, a popular data compression technique. They help in constructing optimal prefix codes for encoding data, where frequently used symbols have shorter codes.

Database Query Optimization: Query planners and optimizers in relational databases use balanced trees to represent query execution plans and optimize data retrieval strategies, ensuring efficient execution of complex queries.

10.7 CONCLUSION

In this chapter, we explored a variety of balanced tree structures that are essential in computer science for maintaining efficient data organization and retrieval. Balanced trees such as B-Trees, Splay Trees, Red-Black Trees, and AA-Trees each offer unique advantages and applications.

B-Trees are widely used in databases and file systems due to their ability to efficiently manage large datasets with a balanced structure that supports fast operations like insertion, deletion, and search.

Splay Trees dynamically adjust their structure through the splaying algorithm, optimizing access times for frequently accessed elements. This property makes them valuable in applications requiring dynamic data management and caching.

Red-Black Trees ensure balanced operations with logarithmic time complexity for insertion, deletion, and search. They find extensive use in memory management, language implementations, and persistent data structures where efficient data retrieval is crucial.

Each of these tree structures plays a critical role in optimizing performance across various computational domains, from database systems to memory management and beyond. By mastering the principles and applications of balanced trees, one gains essential tools for designing efficient algorithms and systems in modern computing environments.

10.8 QUESTIONS AND ANSWERS

1. What are B-Trees and why are they used in databases?
Answer: B-Trees are balanced tree structures designed to handle large amounts of data and frequent operations like insertion, deletion, and search efficiently. They are used in databases because they can maintain balance and optimal access times even with large datasets, ensuring fast retrieval and modification operations.

2. How does the splaying algorithm work in Splay Trees?
Answer: The splaying algorithm in Splay Trees reorganizes the tree by bringing the most recently accessed node to the root position through a series of rotations. This optimization ensures that

frequently accessed elements are closer to the root, improving future access times.

3. What properties define Red-Black Trees as balanced binary search trees?

Answer: Red-Black Trees maintain balance by adhering to specific rules: each node is either red or black, the root is black, and no two red nodes can be adjacent. These properties ensure that the tree remains balanced, with operations like insertion and deletion maintaining logarithmic time complexity.

4. **How do AA-Trees differ from Red-Black Trees?**

Answer: AA-Trees are a variation of Red-Black Trees that simplify the balancing rules. They use only two types of nodes (horizontal and vertical) and employ skew and split operations instead of color changes and rotations. AA-Trees provide efficient performance for dynamic sets and are used in applications requiring balanced tree structures.

5. What are some practical applications of balanced trees?

Answer: Balanced trees are used extensively in databases for indexing and efficient data retrieval, in file systems for managing file directories, in compilers for symbol table management, and in memory management systems for efficient allocation and deallocation of memory blocks.

10.9 REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*. MIT Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th ed.)*. Addison-Wesley.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2015). *Data Structures and Algorithms in Java (6th ed.)*. Wiley.

Mehlhorn, K., & Sanders, P. (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.

Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in Java (3rd ed.)*. Pearson Education.

UNIT – 11: GRAPH DATA STRUCTURE

Structure

11.0 INTRODUCTION

A basic data structure in computer science, graphs are used to represent the connections and interactions between items. They are made up of edges that join pairs of vertices and vertices, also known as nodes. Based on their characteristics, graphs can be classified as directed or undirected, weighted or unweighted, among other varieties. Because of these qualities, graphs are quite flexible and can be used to illustrate a variety of real-world situations, such as social networks and transportation networks.

Graph representation is essential for effective manipulation and storage. The adjacency matrix and the adjacency list are two popular techniques, both with pros and cons related to time and space complexity. The efficiency of graph algorithms can be greatly impacted by selecting the right representation, particularly for big and complicated datasets. Implementing and optimizing graph-related operations requires an understanding of various representations.

For examining and assessing graph structures, graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) are crucial resources. These algorithms serve as the foundation for more complex graph algorithms, such as those that build minimal spanning trees, identify cycles, and locate the shortest pathways. This course explores both simple and complex graph algorithms, emphasizing how they can be used to solve issues in real life and how important they are in a variety of domains, including artificial intelligence, network analysis, and computer graphics.

11.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understand Graph Basics: Define what constitutes a graph, including vertices (nodes) and edges, and distinguish between directed and undirected graphs.

Graph Representation Techniques: Explore various methods for representing graphs, such as adjacency matrices and adjacency lists, and understand the trade-offs between these representations in terms of space and time complexity.

Graph Traversal Algorithms: Learn about fundamental graph traversal algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS), including their applications in solving problems such as finding connected components and detecting cycles.

Advanced Graph Algorithms: Delve into more complex algorithms like Dijkstra's algorithm for finding shortest paths, Prim's and Kruskal's algorithms for Minimum Spanning Trees (MSTs), and algorithms for topological sorting and cycle detection.

Real-World Applications: Explore practical applications of graphs in various domains, such as social networks, transportation networks, and recommendation systems, to understand how graph algorithms solve real-world problems effectively.

11.2 GRAPH

A graph is a type of data structure made up of edges connecting a finite number of vertices, also known as nodes. Graphs are employed to represent pairwise relationships among entities. G = (V, E), where V is a collection of vertices and E is a set of edges linking the vertices, is the definition of a graph.

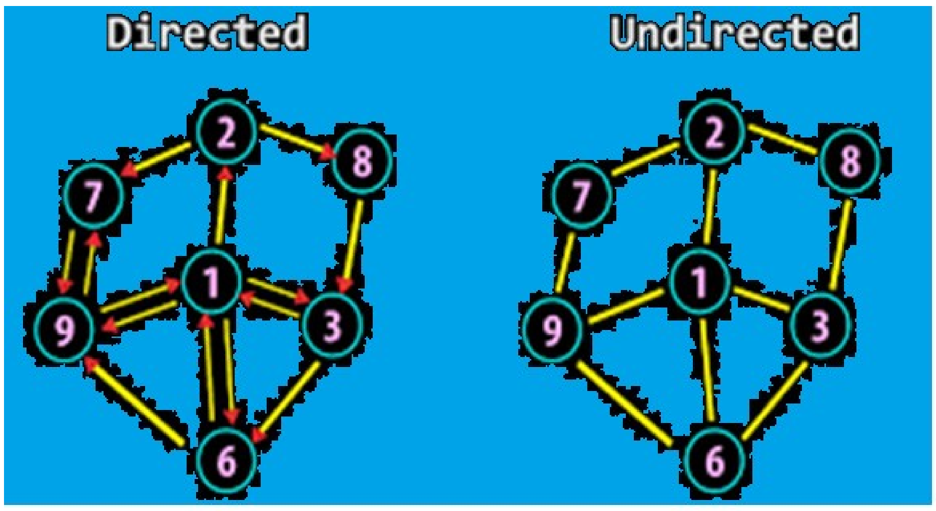
Several real-world structures, including networks, interactions, and paths, can be represented by graphs. People can be shown as vertices in a social network, for example, and friendships as edges.

Graphs: Directed versus Undirected

Graphs fall into two categories according on the orientation of their edges:

Directed Graphs (Digraphs): Every edge in a directed graph indicates a one-way relationship between two vertices. As an ordered pair of vertices, this is represented. An edge that is oriented from vertex u to vertex v, for instance, is represented as (u, v).

Undirected Graphs: An undirected graph has no direction assigned to any of its edges. Because of the bidirectional nature of the link between vertices, an edge between u and v can be traveled in both directions. A pair {u, v} that is unordered is used to express this.



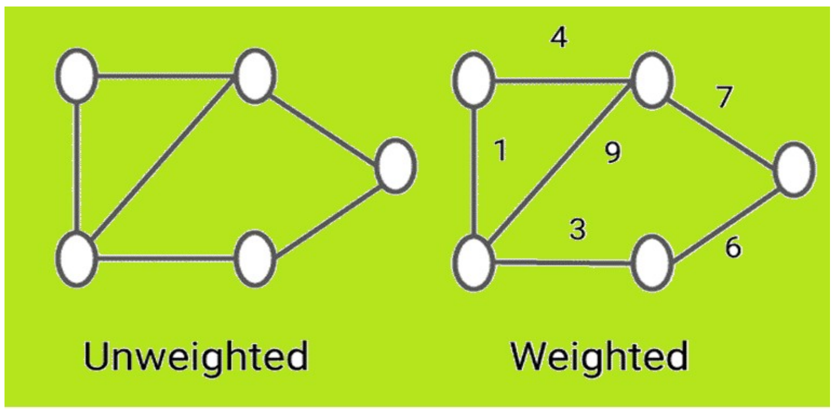
Source: Simple Snippets

Weighted vs. Unweighted Graphs

Another way to categorize graphs is by the weights assigned to their edges:

Weighted Graphs: The weight of a weighted graph is a numerical number that is assigned to each edge. Weights can be used to represent expenses, distances, or any other quantitative metric. The weights can be used, for instance, to represent the distance between two points in a road network.

Unweighted Graphs: An unweighted graph has no weights assigned to its edges. Every edge is regarded as equal. For instance, in a social network, the edges might, in the lack of further information, indicate whether a friendship exists.



Key Terms:

Vertices, or Nodes: They are the basic building blocks of a graph; they stand for entities.

Edges: The links that connect vertices are called edges.

Degree: A vertex's degree is the total number of edges that connect to it. The number of incoming edges in a directed graph is called the in-degree, while the number of exiting edges is called the out-degree.

Path: A series of vertices connected by a series of edges.

Cycle: A path that, aside from the start/end vertex, repeats neither edges nor vertices but instead begins and terminates at the same vertex.

Connected Components: In an undirected graph, a connected component is a subgraph that has no further connections to any other vertices in the supergraph and any two vertices connected to each other by pathways.

11.3 REPRESENTATION OF GRAPHS

Adjacency Matrix Representation

A 2D array of size V × V, where V is the number of vertices in the graph, is called an adjacency matrix. The matrix's cells, each represented by the notation adj [i] [j], each indicate whether or not vertex i and vertex j have an edge.

In case of an unweighted graph:

Vertex i and vertex j have an edge, adj [i] [j] = 1.

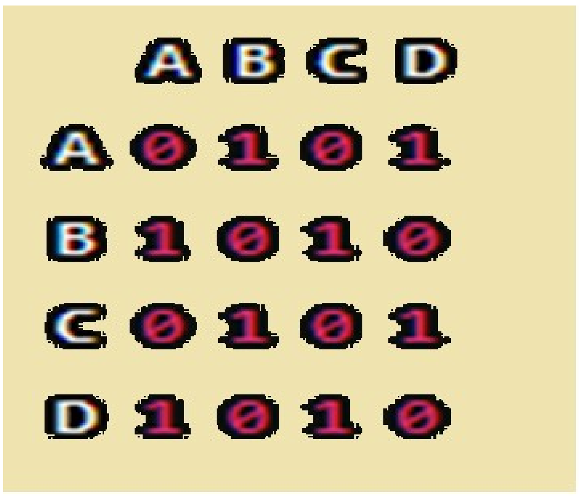
If there isn't an edge connecting vertex i and vertex j, adj [i] [j] = 0.

For a graph with weights:

The weight of the edge between vertex i and vertex j is contained in adj [i] [j].

If there is no edge, adj [i] [j] = 0.

For instance: if a graph has the vertices A, B, C, and D:



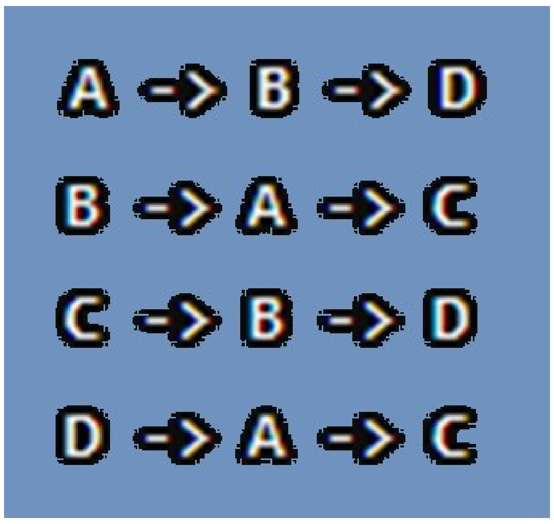
	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

An undirected graph is represented by this matrix, in which A is connected to B and D, B to A and C, and so on.

Adjacency List Representation

An array of lists is called an adjacency list. The number of vertices in the array determines its size. Every entry in the array is a list that has every vertex connected to the vertex the array index represents in it.

For instance, if a graph has the vertices A, B, C, and D:



Vertex A is related to vertices B and D, vertex B is connected to vertices A and C, and so on, as this list demonstrates.

Adjacency Matrix and Adjacency List Comparison

Complexity of Space:

Adjacency Matrix: V is the number of vertices, and O(V^2) space is used. When there are fewer edges in a sparse graph, this is less effective.

(V + E) space is used by the adjacency list, where E is the number of edges. For sparse graphs, this is a more space-efficient method.

Time Complexity:

The Adjacency Matrix

Verifying the existence of an edge: O (1).

Going through every edge once: O (V2).

Adjacencies List:

Verifying if an edge exists: in the worst situation, O(V).

Going around all edges in turn: O (V + E).

Use Cases for Each Representation

The Adjacency Matrix

Ideal for thick graphs with an edge count that approaches V2.

helpful when it's necessary to quickly check whether edges exist.

Adjacencies List:

Ideal for graphs that are sparse, meaning they have a lot less edges than V2.

Faster and more space-efficient for iterating across all edges.

favored in situations when the graph is not tightly connected, such as social networks, road networks, or other applications.

11.4 GRAPH TRAVERSAL ALGORITHMS

Graph traversal refers to the process of visiting all nodes (vertices) in a graph in a systematic way. It involves systematically exploring each vertex and its connected edges to ensure that all nodes are visited exactly once. Two common algorithms for graph traversal are breadth-first search (BFS) and depth-first search (DFS).

Breadth-First Search (BFS): Explores all nodes at the present "depth" prior to moving on to nodes at the next level of depth.

Depth-First Search (DFS): Traverses by exploring as far as possible down a branch before backtracking.

These algorithms are essential for tasks like finding the shortest path, connectivity analysis, and spanning tree construction

First-Breadth Search (BFS)

The graph traversal technique known as Breadth-First Search (BFS) investigates a graph's vertices level by level. BFS begins with a source vertex, visits each of its neighbors, and then advances to the next level of neighbors. It is especially helpful for level-order traversal of trees and for determining the shortest path in unweighted graphs.

Steps of Algorithm:

Initialize a queue and enqueue the starting vertex.

Mark the starting vertex as visited.

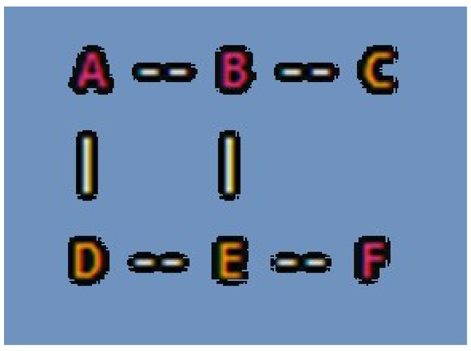
While the queue is not empty:

Dequeue a vertex from the queue.

Process the dequeued vertex.

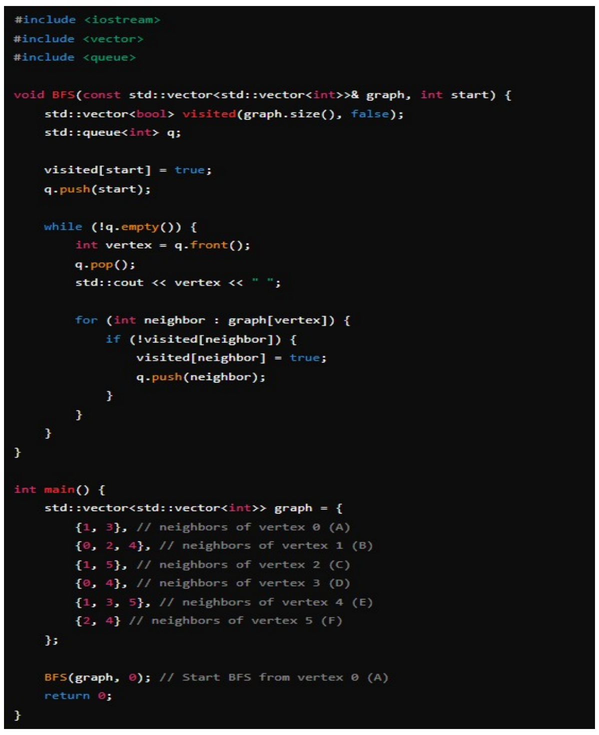
Enqueue all unvisited neighboring vertices and mark them as visited.

Example: Consider the following graph:



BFS would visit the vertices in the following order, beginning at vertex A: A, B, D, C, E, and F.

Implementation in C++:



```cpp
#include <iostream>
#include <vector>
#include <queue>

void BFS(const std::vector<std::vector<int>>& graph, int start) {
    std::vector<bool> visited(graph.size(), false);
    std::queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int vertex = q.front();
        q.pop();
        std::cout << vertex << " ";

        for (int neighbor : graph[vertex]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    std::vector<std::vector<int>> graph = {
        {1, 3}, // neighbors of vertex 0 (A)
        {0, 2, 4}, // neighbors of vertex 1 (B)
        {1, 5}, // neighbors of vertex 2 (C)
        {0, 4}, // neighbors of vertex 3 (D)
        {1, 3, 5}, // neighbors of vertex 4 (E)
        {2, 4} // neighbors of vertex 5 (F)
    };

    BFS(graph, 0); // Start BFS from vertex 0 (A)
    return 0;
}
```

DFS, or Depth-First Search

Concept and Use Cases: The graph traversal technique known as Depth-First Search (DFS) searches as far as feasible down each branch before turning around. Either directly or implicitly through recursion, it makes use of a stack data structure. DFS is used to solve puzzles like mazes and find cycles in topological sorting.

Steps of Algorithm:

Initialize a stack and push the starting vertex.

Mark the starting vertex as visited.

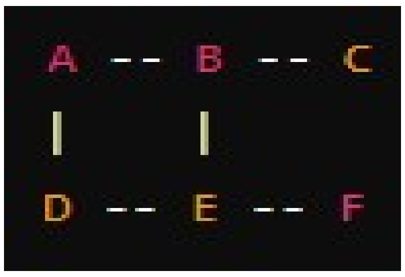
While the stack is not empty:

Pop a vertex from the stack.

Process the popped vertex.

Push all unvisited neighboring vertices onto the stack and mark them as visited.

Example: Consider the following graph:



DFS could visit the vertices in the following order, starting with vertex A: A, B, E, F, D, and C (this is only one possible order among many).

Implementation in C++:

```
#include <iostream>
#include <vector>
void DFSUtil(const std::vector<std::vector<int>>& graph, int vertex, std::vector<bool>& visited) {
    visited[vertex] = true;
    std::cout << vertex << " ";
    for (int neighbor : graph[vertex]) {
        if (!visited[neighbor]) {
            DFSUtil(graph, neighbor, visited);
        }
    }
}
void DFS(const std::vector<std::vector<int>>& graph, int start) {
    std::vector<bool> visited(graph.size(), false);
    DFSUtil(graph, start, visited);
}
int main() {
    std::vector<std::vector<int>> graph = {
        {1, 3}, // neighbors of vertex 0 (A)
        {0, 2, 4}, // neighbors of vertex 1 (B)
        {1, 5}, // neighbors of vertex 2 (C)
        {0, 4}, // neighbors of vertex 3 (D)
        {1, 3, 5}, // neighbors of vertex 4 (E)
        {2, 4} // neighbors of vertex 5 (F)
    };
    DFS(graph, 0); // Start DFS from vertex 0 (A)
    return 0;
}
```

A comparison between DFS and BFS

BFS:

Improved for determining the shortest path in graphs without weights.

extra memory is used (queue).

traversal at level-order.

DFS:

Ideal for activities that necessitate delving into the most profound area of the graph, including resolving puzzles.

reduces memory usage (recursion/stack).

Process before children in a preorder traversal; however, this can be modified for other traversals as well.

11.5 ADVANCED GRAPH ALGORITHMS

Minimum Spanning Trees (MST) Algorithm

The subset of edges in a connected, undirected graph that joins all of the vertices together without creating any cycles and with the least amount of edge weight overall is found using Minimum Spanning Tree (MST) techniques. The following are the main ideas and methods pertaining to minimum spanning trees:

MST Algorithm Concepts:

Minimum Spanning Tree (MST): An edge subset that joins all of the vertices in a graph without creating any cycles is known as a spanning tree. A spanning tree with a minimum sum of edge weights is known as a minimum spanning tree.

Properties:

An MST consisting of N vertices has precisely N-1 edges.

It is acyclic—it lacks cycles.

It joins every vertex with the least amount of edge weight overall.

Applications:

Network design: It is the process of connecting all nodes, or cities, with the fewest possible total edge weights, or roads, cables, etc.

Cluster Analysis: It Put related items in groups with the least amount of overall dissimilarity is known as cluster analysis.

Algorithms for Approximation: Used in a variety of approximation techniques to address optimization issues.

Prim's Algorithm: A weighted undirected graph's Minimum Spanning Tree (MST) can be found using Prim's algorithm. Beginning with an arbitrary vertex, it adds the shortest edge possible between each vertex in the expanding tree and any vertex that is not yet in the tree, growing the MST one edge at a time.

Steps of an Algorithm:

Set the MST's representation to an empty set at startup.

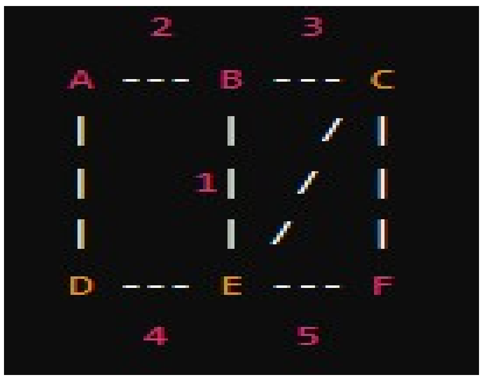
Add an arbitrary vertex to the MST to begin with.

Even so, not every vertex is included in the MST:

To connect a vertex inside the MST to a vertex outside the MST, choose the edge with the least weight.

To the MST, add the chosen edge and vertex.

As an illustration, look at the weighted graph below:



Prim's algorithm would add edges (A-B, B-E, E-D, B-C, and E-F) starting at vertex A to create the MST, which would have a total weight of 2 + 1 + 4 + 3 + 5 = 15.

Kruskal's Algorithm:

Another technique for determining the MST of a weighted undirected graph is the Kruskal's algorithm. It adds edges to the MST while making sure no cycle forms by sorting all of the edges in non-decreasing order of their weights.

Steps of Algorithm:

Arrange each edge according to its weight in a non-decreasing sequence.

Set the MST's representation to an empty set at startup.

Apply edges to the MST using a Union-Find data structure in ascending weight order, making sure that no cycles arise.

Example: Using the previously given example graph, Kruskal's algorithm would create an MST with a total weight of 15 by adding edges (A-B, B-E, E-D, B-C, and E-F).

Shortest Path Algorithms

The shortest path between a source vertex and every other vertex in a weighted graph with non-negative weights is found using Dijkstra's algorithm. The shortest known path is always expanded through the usage of a priority queue.

Steps of Algorithm:

Set the distances between the source vertex and itself to 0 and to all other vertices to infinity.

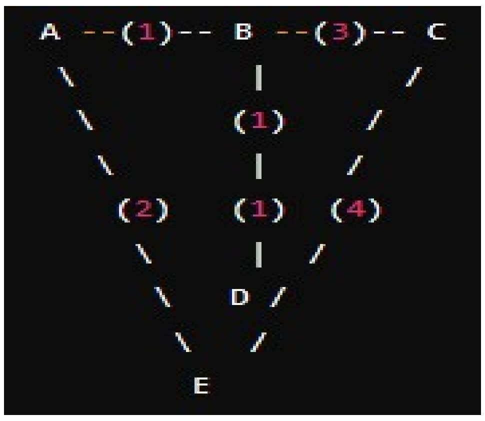
Store the vertices to be processed in a priority queue, beginning with the source vertex.

Although there are items in the priority queue:

Take out of the priority queue the vertex that is the closest to the center.

If a shorter path is discovered, update the distances to the vertices that are nearby.

As an illustration, look at the weighted graph below:



Dijkstra's method would calculate the shortest paths to each of the vertices (A, B, C, D, and E) with their corresponding distances (1, 4, 2).

Bellman-Ford Algorithm: In a weighted graph with negative weight edges, the Bellman-Ford algorithm finds the shortest pathways between a single source vertex and every other vertex.

All edges are relaxed repeatedly by a number equal to the vertices minus one.

Steps of Algorithm:

Set the distances between the source vertex and itself to 0 and to all other vertices to infinity.

Let |V| be the number of vertices, and relax all edges |V| - 1 times.

By repeatedly iterating over all edges and updating distances, look for cycles with negative weights.

Example: The Bellman-Ford algorithm can accurately compute shortest paths for a graph with edges that have negative weights, even when the graph has cycles and negative weights.

Maximum Flow Algorithms

The **Ford-Fulkerson Algorithm** is a directed graph with a capacity for each edge. It calculates the maximum flow in a flow network. It finds augmenting paths by applying the notion of residual capacity, and it then raises flow along these paths until no more augmenting paths are found.

Steps of Algorithm:

Set the flow's initial value to 0.

As the path from source to sink is augmentable:

Use DFS or BFS to determine the augmenting path.

Determine the path's residual capacity.

Enhance the flow as it travels.

Example: The Ford-Fulkerson algorithm determines the maximum flow from source to sink in a flow network.

Edmonds-Karp Algorithm: This algorithm use BFS to determine the augmenting pathways and is an application of the Ford-

Fulkerson technique. It guarantees that the algorithm's temporal complexity is O(VE^2), where V and E are the numbers of vertices and edges, respectively.

Uses:

MST Algorithms: In networks where linking all vertices at the lowest possible cost is critical, the Prim and Kruskal algorithms are indispensable for determining the minimal spanning tree.

Shortest way Algorithms: Based on flight paths or road networks, navigation systems employ Dijkstra's and Bellman-Ford algorithms to determine the shortest way between two points.

Maximum Flow Algorithms: In network flow problems, such transportation and communication networks, the Ford-Fulkerson and Edmonds-Karp algorithms are used to maximize the flow from source to sink while taking into account each edge's capacity restrictions.

11.6 APPLICATIONS OF GRAPHS

Uses for Graphs

Graphs are widely used in practical applications.

Applications in the real world include recommendation systems (user-item relationships), maps (routing and navigation), and social networks (modeling user connections).

Algorithmic problems include network flow optimization (maximizing flow in a network) and the traveling salesman problem (finding the shortest path to visit each vertex).

Three types of graph-based data structures are available: spanning trees, which are a subset of a graph that connects all of its vertices, trees, and connectedness, which examines connected components.

11.7 CONCLUSION

Graphs and the algorithms that go along with them are an essential component of computer science because they offer strong tools for problem modeling and addressing intricate issues. We have studied the definitions, important characteristics, and different kinds of graph data structures throughout this unit. Gaining an understanding of these fundamentals is necessary in order to apply graphs to real-world situations and to efficiently solve problems involving connections and relationships.

Additionally, we looked at other graph representation techniques, including adjacency lists and adjacency matrices, and talked about the benefits and drawbacks of each. This information is essential for choosing the best representation depending on an application's unique requirements, including the graph's size and the kinds of operations that must be carried out. Furthermore, graph traversal algorithms such as BFS and DFS offer fundamental methods for graph exploration and serve as building blocks for more complex algorithms.

Complex graph algorithms show the breadth of applications and depth of issues that graph theory can solve. Examples include finding minimal spanning trees, shortest pathways, and maximum flows. These algorithms are essential in many fields, including data analysis, resource management, network design, and optimization. Gaining proficiency in these ideas and methods will enable one to effectively use graphs to solve challenging, real-world problems.

11.8 QUESTIONS AND ANSWERS

1. What is a Minimum Spanning Tree (MST)

Answer: In a connected, undirected graph, an MST is a subset of edges that joins all vertices with the fewest feasible total edge weights and without any cycles.

2. Describe the algorithm used by Dijkstra.

Answer: In a weighted graph with non-negative weights, Dijkstra's algorithm uses a priority queue to explore vertices and determines the shortest path between each source vertex and all other vertices.

3. What are some practical uses for graphs?

Answer: In social networks, maps are utilized for routing and navigation; in recommendation systems, graphs are employed to depict user-item relationships.

4. The Traveling Salesman Problem (TSP): What is it?

Answer: In order to discover the shortest path that visits each vertex once and returns to the origin vertex, a salesman must solve the algorithmic problem known as TSP.

5. What are the differences between the MST algorithms found by Prim and Kruskal?

Answer: In response, Kruskal's algorithm adds the shortest edge to the MST until all vertices are connected, guaranteeing no cycles emerge. Prim's approach grows the MST from an arbitrary beginning vertex by adding the cheapest edge to the tree.

6. What role do graph traversal algorithms play?

Answer: In order to perform activities like pathfinding, connectivity checking, and cycle detection, graph traversal algorithms like BFS and DFS are essential for network exploration and analysis.

7. What are the adjacency matrix and adjacency list space and temporal complexities?
Answer: The adjacency matrix allows for O(1) time complexity for edge look-up and O(V2) space complexity. The traversal difficulty of an adjacency list is O (V + E) in both space and time.

11.9 REFERENCES

- Bjarne Stroustrup, "The C++ Programming Language"
- Herb Sutter, "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions"
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, "C++ Primer"
- Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Nicolai M. Josuttis, "C++ Standard Library: A Tutorial and Reference"

UNIT – 12: MINIMUM COST SPANNING TREES

Structure

12.0 INTRODUCTION

Graphs are fundamental data structures in computer science, representing a network of interconnected nodes or vertices and the edges connecting them. They are versatile tools used in a wide range of applications, from social networks and web page ranking to network routing and scheduling problems. Understanding how to efficiently traverse and manipulate graphs is crucial for solving many complex computational problems. This unit delves into several key graph algorithms, each with unique properties and applications.

We'll begin by exploring Minimum Cost Spanning Trees, which are essential for optimizing the connections within a network. Two prominent algorithms for constructing these trees, Kruskal's and Prim's algorithms, will be examined in detail. Following this, we'll look at Breadth-First Search (BFS) and Depth-First Search (DFS), foundational algorithms for traversing graphs that form the basis for more advanced graph operations. These traversal techniques are vital for exploring and understanding the structure of a graph.

Finally, we will cover the concept of Strongly Connected Components (SCCs) in directed graphs. Identifying SCCs helps in understanding the underlying structure and connectivity of complex networks, leading to more efficient designs and analyses. Through this unit, you'll gain a comprehensive understanding of key graph algorithms and their applications, equipping you with the knowledge to tackle a wide array of problems in computer science and beyond.

12.1 OBJECTIVES

After completing this unit, you will be able to understand,
Understand Graph Fundamentals: Gain a solid understanding of graph theory, including the basic definitions, properties, and representations of graphs.

Explore Minimum Cost Spanning Trees: Learn about Minimum Cost Spanning Trees and their importance in optimizing network connections. Study Kruskal’s and Prim’s algorithms for constructing these trees.

Master Graph Traversal Algorithms: Develop proficiency in Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms, and understand their applications in graph traversal and problem-solving.

Analyze Strongly Connected Components (SCCs): Understand the concept of SCCs in directed graphs and learn methods to identify them, which is crucial for analyzing and designing complex networks.

Apply Graph Algorithms: Explore practical applications of graph algorithms in various domains such as network design, data analysis, machine learning, and logistics.

Evaluate Algorithm Efficiency: Analyze the time and space complexities of different graph algorithms to understand their performance and scalability.

12.2 MINIMUM COST SPANNING TREES

Minimum Cost Spanning Trees (MSTs) are crucial concepts in graph theory, representing the subset of edges that connect all vertices of a graph with the lowest possible total edge weight. An MST of a graph ensures that all vertices are connected while minimizing the sum of the edge weights, without forming any cycles. This structure finds wide application in various fields such as network design, telecommunications, and computer science algorithms.

The primary goal of finding an MST is to establish a spanning tree that spans all vertices with the least total weight, making it an optimal solution for connecting nodes in networks or organizing data points in clustering algorithms. Two well-known algorithms for finding MSTs include Kruskal’s and Prim’s algorithms. Kruskal’s algorithm sorts all edges by weight and adds them to the MST if they do not form cycles, using a union-find data structure for efficiency. On the other hand, Prim’s algorithm starts from an arbitrary vertex and grows the MST by always adding the shortest edge connecting the current MST to an adjacent vertex until all vertices are included.

Applications of MSTs extend to optimizing routes in transportation networks, minimizing costs in manufacturing processes, and organizing hierarchical data structures efficiently. The ability to compute MSTs efficiently ensures optimal solutions to a variety of real-world problems where minimizing connectivity costs is essential.

Algorithms:

Kruskal’s Algorithm: Kruskal’s algorithm constructs an MST by iteratively adding the smallest edge that doesn’t form a cycle until all vertices are connected. It uses a union-find data structure to efficiently manage and merge subsets of vertices.

Prim’s Algorithm: Prim’s algorithm starts from an arbitrary vertex and grows the MST one vertex at a time, always choosing the shortest edge that connects a vertex in the MST to a vertex outside of it. It typically uses a priority queue to manage candidate edges efficiently.

Applications:

Network Design: MSTs are used to minimize the cost of connecting cities in a telecommunications network or computers in a LAN.

Clustering: In data science, MSTs can be used to identify clusters by treating each vertex as a data point and edges as distances between points.

Optimization Problems: MSTs are essential in optimization problems like finding the minimum cost of connecting components in a manufacturing process or the shortest route in transportation networks.

12.3 KRUSKAL'S ALGORITHM

Kruskal's algorithm is a popular method used to find the Minimum Spanning Tree (MST) of a connected, weighted graph. The algorithm operates by sorting all the edges in the graph by their weights and then iteratively adding the smallest edge to the growing MST, provided that adding the edge does not form a cycle. This process continues until all vertices are included in the MST.

Here's a step-by-step outline of Kruskal's algorithm:

Initialization: Start with a graph containing V vertices and E edges.

Sort Edges: Sort all edges in the graph in non-decreasing order of their weights.

Union-Find Data Structure: Initialize a union-find data structure (or disjoint-set data structure) to keep track of which vertices are in which components and to efficiently check whether adding an edge would form a cycle.

Iterate Through Edges: Iterate through the sorted edges and for each edge:

Check if adding the edge to the MST would not create a cycle using the union-find structure.

If it does not create a cycle, add the edge to the MST.

Update the union-find structure to merge the components of the vertices connected by the edge.

Termination: Stop when V − 1 edges have been added to the MST, where V is the number of vertices in the graph.

Kruskal’s algorithm is efficient with a time complexity of O (E log E) due to the sorting step, where E is the number of edges in the graph. This makes it suitable for graphs with a large number of edges, especially sparse graphs where E is much smaller than V^2. Applications of Kruskal’s algorithm include network design, circuit design, and clustering algorithms where finding the MST helps minimize costs or optimize connections between nodes. Its simplicity and efficiency make it a valuable tool in various computational and practical settings.

Example:

Let's consider the following graph with 4 vertices (A, B, C, D) and the following weighted edges:

AB: 1

AC: 4

AD: 3

BC: 2

BD: 5

CD: 6

Step-by-Step Execution:

Sort Edges: Sort edges by weight:

AB: 1

BC: 2

AD: 3

AC: 4

BD: 5

CD: 6

Initialize Union-Find: Initialize each vertex as its own component.

Process Edges:

Edge AB (Weight 1): Include AB in MST (A-B).

Edge BC (Weight 2): Include BC in MST (B-C).

Edge AD (Weight 3): Include AD in MST (A-D).

Edge AC (Weight 4): Include AC in MST (A-C).

Edge BD (Weight 5): Include BD in MST (B-D).

Union Operations:

Union(A, B)

Union(B, C)

Union(A, D)

Union(A, C)

Union(B, D)

Resulting MST: The MST includes edges AB, BC, AD, AC. The total weight of the MST is 1+2+3+4=101 + 2 + 3 + 4 = 101+2+3+4=10.

Explanation:

Kruskal's algorithm selects edges based on their weights in ascending order and ensures that no cycles are formed by checking if the endpoints of each edge belong to the same connected component using the union-find data structure.

It's efficient for sparse graphs and can handle graphs with different edge weights, making it versatile for various applications such as network design, circuit layout, and clustering algorithms.

Implementation in C++:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Structure to represent an edge in the graph
struct Edge {
    int u, v, weight;
    Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}
};

// Union-Find data structure with path compression and union by
rank
class UnionFind {
private:
    vector<int> parent, rank;
public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i)
            parent[i] = i;
    }
    int find(int u) {
        if (parent[u] != u)
            parent[u] = find(parent[u]); // Path compression
        return parent[u];
    }
    void union_set(int u, int v) {
        int root_u = find(u);
        int root_v = find(v);
        if (root_u != root_v) {
            // Union by rank
            if (rank[root_u] > rank[root_v])
                parent[root_v] = root_u;
            else if (rank[root_u] < rank[root_v])
                parent[root_u] = root_v;
            else {
                parent[root_v] = root_u;
                rank[root_u]++;
            }
        }
    }
};
// Comparator function to sort edges by weight
bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}
// Function to find Minimum Spanning Tree using Kruskal's
algorithm
vector<Edge> kruskalMST(vector<Edge>& edges, int V) {
    // Sort edges by weight
    sort(edges.begin(), edges.end(), compareEdges);
    UnionFind uf(V);
    vector<Edge> result;
    for (Edge& edge : edges) {
        int u = edge.u;
        int v = edge.v;
        if (uf.find(u) != uf.find(v)) {
            uf.union_set(u, v);
            result.push_back(edge);
        }
        // Stop when MST is found (V-1 edges)
        if (result.size() == V - 1)
        break;
    }
    return result;
}
```

12.4 PRIM'S ALGORITHM

Prim's algorithm is another popular greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, weighted graph. Similar to Kruskal's algorithm, Prim's algorithm builds the MST incrementally, starting from an arbitrary vertex and adding the shortest edge that connects a vertex in the MST to a vertex outside the MST. Here’s how Prim's algorithm works, explained with an example:

Prim's Algorithm Steps:

Initialization: Start with an arbitrary vertex as the initial MST, or a single vertex as the starting point.

Priority Queue: Use a priority queue (min-heap) to keep track of the minimum-weight edge that connects the MST to vertices outside the MST.

Process Edges: Repeat the following steps until all vertices are included in the MST:

Add the vertex with the smallest edge weight that connects the current MST to a vertex not yet in the MST.

Update the priority queue with new edges that connect the current MST to vertices outside the MST.

Termination: Stop when all vertices are included in the MST, forming V−1V-1V−1 edges, where VVV is the number of vertices in the graph.

Example:

Consider the following graph with 4 vertices (A, B, C, D) and the following weighted edges:

AB: 1

AC: 4

AD: 3

BC: 2

BD: 5

CD: 6

Step-by-Step Execution:

Start with Vertex A: Assume we start with vertex A.

Priority Queue Contents:

Initially, vertex A is in the MST.

Edges: AB (1), AC (4), AD (3).

Process:

Step 1: Add edge AB to the MST (A-B). Priority queue now has AC (4), AD (3).

Step 2: Add edge AD to the MST (A-D). Priority queue now has AC (4), BD (5).

Step 3: Add edge AC to the MST (A-C). Priority queue now has BC (2), BD (5).

Step 4: Add edge BC to the MST (B-C). Priority queue now has BD (5), CD (6).

Resulting MST: The MST includes edges AB, AD, AC, BC. The total weight of the MST is 1+3+4+2=101 + 3 + 4 + 2 = 101+3+4+2=10.

Explanation:

Prim's algorithm starts from an initial vertex and grows the MST one vertex at a time by adding the shortest edge that connects the current MST to a vertex outside the MST.

It uses a priority queue to efficiently retrieve the next minimum-weight edge to process, ensuring that the algorithm runs efficiently even for large graphs.
Prim's algorithm is particularly useful for dense graphs or when a specific starting vertex is known, as it guarantees that the MST grows incrementally with minimal edge weights.

Implementation in C++:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
#define V 5 // Number of vertices in the graph
// Function to find the vertex with the minimum key value,
// from the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[]) {
  int min = INT_MAX, min_index;
  for (int v = 0; v < V; v++) {
    if (mstSet[v] == false && key[v] < min) {
      min = key[v];
      min_index = v;
    }
  }
  return min_index;
}
// Function to print the MST stored in parent array
void printMST(int parent[], vector<vector<int>>& graph) {
  cout << "Edge \tWeight\n";
  for (int i = 1; i < V; i++) {
    cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]]
<< "\n";
```

```
    }
}
// Function to construct and print MST using Prim's algorithm
void primMST(vector<vector<int>>& graph) {
    int parent[V]; // Array to store constructed MST
    int key[V];     // Key values used to pick minimum weight edge
in cut
    bool mstSet[V]; // To represent set of vertices included in MST
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    // Always include first vertex in MST
    key[0] = 0;      // Make key 0 so that this vertex is picked as first
vertex
    parent[0] = -1; // First node is always root of MST
    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not
yet included in MST
        int u = minKey(key, mstSet);
        // Add the picked vertex to the MST set
        mstSet[u] = true;
        // Update key value and parent index of the adjacent vertices
of the picked vertex
        // Consider only those vertices which are not yet included in
MST
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non-zero only for adjacent vertices of u
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    // Print the constructed MST
    printMST(parent, graph);
}
// Driver program to test above functions
int main() {
    vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    // Print the MST using Prim's algorithm
    primMST(graph);
    return 0;
}
```

12.5 APPLICATIONS OF MINIMUM COST SPANNING TREES

Use cases in network design (telecommunications, computer networks).

In network design, Minimum Cost Spanning Trees (MCST) find numerous applications across various domains. Here are some specific use cases:

Telecommunications Networks: MCST algorithms are extensively used in telecommunications to design efficient networks of communication channels, such as fiber optic cables or wireless links between cities, towns, or network nodes. The goal is to minimize the total cost of establishing and maintaining these connections while ensuring reliable and high-speed communication.

Computer Networks: In computer networks, MCST algorithms help in designing network topologies that connect all devices (computers, routers, switches) with minimal total cable length or transmission cost. This ensures efficient data transmission, reduces latency, and enhances network reliability.

Wireless Sensor Networks: MCST algorithms are applied in designing wireless sensor networks (WSNs) where sensors need to communicate with each other and with a central node (sink) using minimal energy consumption. The algorithm helps in forming a tree structure that optimizes energy usage and ensures data from sensors is efficiently routed to the sink.

Satellite Communication Networks: For satellite communication systems, MCST algorithms are used to establish communication links between satellites and ground stations or between different satellites in a constellation. The objective is to minimize signal propagation delay and maximize bandwidth utilization while keeping operational costs low.

Internet of Things (IoT) Networks: In IoT applications, where numerous devices (sensors, actuators, smart appliances) are interconnected, MCST algorithms play a role in optimizing the network topology for efficient data exchange and resource management. This ensures that IoT devices can communicate seamlessly while conserving energy and reducing communication overhead.

Clustering applications in data analysis and machine learning

In data analysis and machine learning, clustering refers to the process of grouping data points into clusters based on their similarity or distance metrics. Minimum Cost Spanning Trees (MCST) and related algorithms have applications in clustering contexts, particularly in graph-based clustering methods. Here's how MCST and clustering intersect:

Graph-based Clustering:

Minimum Spanning Tree Clustering: In this approach, MCST algorithms like Kruskal's or Prim's are used to construct a minimum spanning tree of a graph where nodes represent data points and edges represent distances or similarities between them. Once the tree is constructed, clusters can be derived by cutting edges based on a threshold distance or similarity measure. The resulting clusters are connected subgraphs with minimal total edge weights, ensuring compact and cohesive clusters.

Hierarchical Clustering:

Agglomerative Clustering: MCST algorithms can be adapted for hierarchical clustering methods. Starting with each data point as a separate cluster, the algorithm progressively merges clusters based on proximity until all points belong to a single cluster. The merging process can be guided by the edges of the MCST, ensuring that clusters are merged in a way that minimizes the total inter-cluster similarity or distance.

Community Detection:

Graph Partitioning: MCST algorithms are also used in community detection tasks where the goal is to identify densely connected subgroups of nodes in a network (graph). By constructing an MCST or other graph-based structures, community detection algorithms can efficiently identify these subgroups, which often correspond to clusters of similar data points in applications like social network analysis or recommendation systems.

Optimization and Representation Learning:

MCST-based clustering methods can help in optimizing representation learning tasks by constructing a graph representation of data points and then applying MCST algorithms to discover meaningful structures or patterns. This approach can enhance the efficiency of clustering tasks in large datasets or high-dimensional spaces where traditional clustering algorithms may struggle.

Optimization problems in logistics and transportation.

In logistics and transportation, optimization problems are pervasive, involving efficient resource allocation, route planning, and network management. Minimum Cost Spanning Trees (MCST) and related algorithms play crucial roles in solving these optimization challenges:

Network Design and Maintenance:

Infrastructure Planning: MCST algorithms like Prim's and Kruskal's are used to design efficient transportation networks such as roadways, railways, and telecommunications grids. By constructing minimum spanning trees, these algorithms help minimize construction costs while ensuring connectivity and accessibility across the network.

Vehicle Routing and Scheduling:

Optimal Route Planning: In transportation logistics, MCST algorithms aid in determining the most cost-effective routes for vehicles, considering factors like distance, traffic conditions, and fuel costs. By constructing minimal spanning trees or related structures, these algorithms optimize delivery routes, reduce transportation times, and lower operational costs.

Supply Chain Management:

Inventory and Distribution Networks: MCST algorithms optimize supply chain networks by identifying the most efficient distribution routes between warehouses, suppliers, and retail locations. This ensures timely delivery of goods while minimizing transportation costs and maintaining inventory levels.

Facility Location and Service Coverage:

Service Area Design: MCST algorithms assist in locating facilities (such as warehouses or distribution centers) strategically to maximize service coverage while minimizing transportation distances and costs. These algorithms ensure that service areas are efficiently defined and maintained.

Resource Allocation and Management:

Energy and Resource Networks: In energy distribution and resource management, MCST algorithms optimize the layout of power grids or resource networks. By minimizing the total network cost (including construction and maintenance), these algorithms improve resource allocation efficiency and reliability.

12.6 BREADTH-FIRST SEARCH (BFS)

Breadth-First Search (BFS) is a fundamental graph traversal algorithm used to explore nodes level by level. It starts at a specified node (often called the "source" node) and explores all its neighbors at the present depth level before moving on to nodes at the next depth level.

Key Characteristics and Steps:

Initialization:

BFS begins by selecting a starting node and marking it as visited.
It uses a queue data structure to manage the order of exploration. The starting node is enqueued.

Exploration Process:

Dequeue a node from the front of the queue.
Visit all adjacent nodes (neighbors) of the dequeued node that have not been visited yet.
Mark each visited node to prevent re-processing and enqueue it into the queue.

Level-wise Exploration:

BFS ensures that all nodes at a certain depth (distance from the source) are visited before moving on to nodes at the next depth level.
This ensures that BFS explores the shortest path first in an unweighted graph.

Termination:

The process continues until the queue is empty, meaning all reachable nodes have been visited.

Applications:

Shortest Path and Minimum Spanning Tree: BFS can be used to find the shortest path in an unweighted graph and to construct the minimum spanning tree in conjunction with other algorithms.

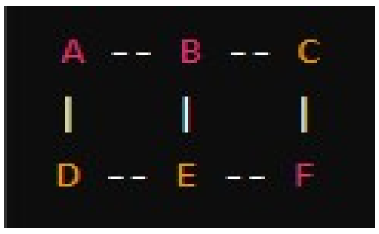
Web Crawling and Social Networking: BFS is used by search engines to crawl the web and by social networking sites to find friends or connections within a limited number of hops.

Puzzle Solving: BFS is employed in solving puzzles like the 8-puzzle or maze traversal, where finding the shortest path or reaching a target configuration is essential.

Example:

Consider a simple graph with nodes connected in a way that resembles a tree structure. Starting from node A, BFS would explore each level of nodes before moving to the next level. For instance, from A, it would explore B and C, then from B, it would explore D and E, and so on.

Consider a graph represented as follows:



```
A -- B -- C
|    |    |
D -- E -- F
```

Starting BFS from node A:

Initialization:

Begin at node A and mark it as visited.

Enqueue A into the queue.

Exploration Process:

Dequeue A, visit its neighbors B and D, and enqueue them (B before D).

Dequeue B, visit its neighbors A, C, and E. Enqueue C and E (E before D).

Dequeue C, visit its neighbors B and F. Enqueue F.

Dequeue D, visit its neighbors A and E (skip A as it's already visited).

Dequeue E, visit its neighbors B, D, and F (skip B and D as they're visited).

Dequeue F, visit its neighbor E (skip as it's visited).

Result:

The BFS traversal order from node A would be: A, B, D, C, E, F.

In this example:

BFS explores all nodes at the current depth level before moving on to nodes at the next depth level.

It ensures that the shortest path (in terms of number of edges) from the starting node A to any other reachable node is found first.

Complexity:

Time Complexity: O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph.

Space Complexity: O(V), due to the storage required for the queue and the visited list.

12.7 DEPTH-FIRST SEARCH (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

It traverses a graph depthwise, exploring vertices and edges to reach the deepest nodes before backtracking to explore other paths. The main properties of DFS include its recursive nature, which utilizes a stack to keep track of vertices, and its ability to uncover all vertices in a connected component.

Definition and Properties of DFS:

DFS starts from an initial vertex, visits all its neighbors recursively, and marks visited vertices to avoid revisiting. It follows these properties:

Recursive Nature: DFS uses recursion or an explicit stack to manage traversal.

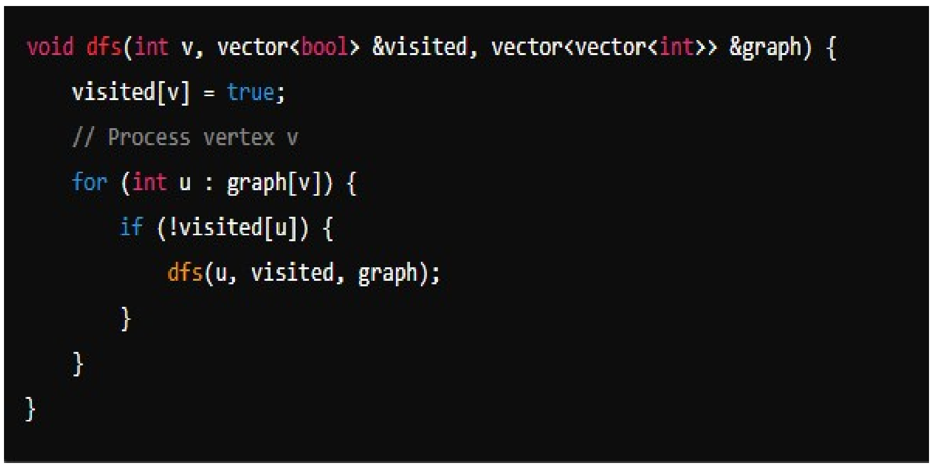
Backtracking: It explores all paths from the current vertex before moving to the next vertex.

Visited Marking: Ensures each vertex is visited once to avoid infinite loops in cyclic graphs.

Implementation Details:

Recursive Implementation:

In a recursive approach, DFS uses function calls to traverse the graph:

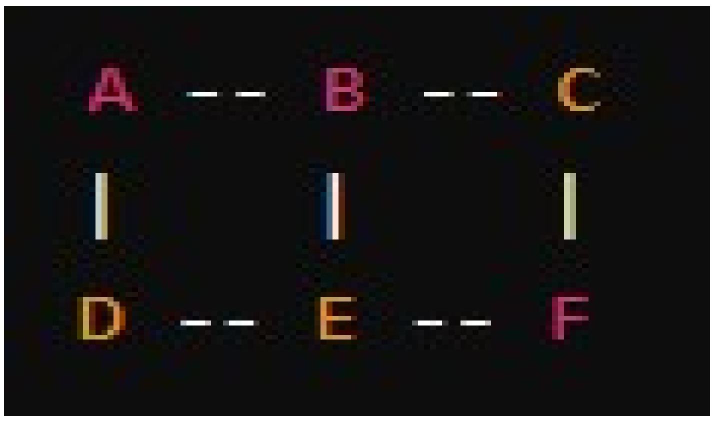


```cpp
void dfs(int v, vector<bool> &visited, vector<vector<int>> &graph) {
    visited[v] = true;
    // Process vertex v
    for (int u : graph[v]) {
        if (!visited[u]) {
            dfs(u, visited, graph);
        }
    }
}
```

Example:

Depth-First Search (DFS) is another fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. Here's an example of how DFS works on a simple graph:

Consider a graph represented as follows:



Starting DFS from node A:

Initialization:

Begin at node A and mark it as visited.

Exploration Process:

Visit A's neighbors recursively: B, D, E, F.

From B, visit its unvisited neighbor E (since B to A is visited).

From E, visit its unvisited neighbors F (since E to A is visited).

Result:

The DFS traversal order from node A would be: A, B, E, F, C, D.

In this example:

DFS explores as far as possible along each branch before backtracking.

It uses a stack (implicitly through recursion or explicitly) to keep track of the path and visited nodes.

DFS is used for tasks like finding connected components, detecting cycles, and topological sorting in directed graphs.

Applications of DFS:
DFS finds applications in various graph-related problems:

Cycle Detection: Detects cycles in directed and undirected graphs by checking for back edges during traversal.

Topological Sorting: Orders vertices such that for every directed edge u -> v, u comes before v in the ordering.
Maze Solving: Used to find paths through mazes or grids by exploring all possible paths until the exit is found.

12.8 FINDING STRONGLY CONNECTED COMPONENTS (SCCS)

Finding Strongly Connected Components (SCCs) in a directed graph is a fundamental graph algorithm that identifies subsets of vertices where each vertex is reachable from any other vertex within the same subset. Formally, an SCC in a directed graph is a maximal subgraph such that for every pair of vertices uuu and vvv in the SCC, there exists a path from u to v and a path from v to u.

Steps to Find Strongly Connected Components (Kosaraju's Algorithm):
First Pass (DFS on Original Graph):
Perform a Depth-First Search (DFS) on the original graph, tracking the finishing times of vertices. This step helps identify the order in which vertices finish processing.
Store vertices based on their finishing times in a stack.

Transpose Graph:

Create a transpose or reverse graph where all the edges of the original graph are reversed. Essentially, if there is an edge from u to v in the original graph, there is an edge from vvv to uuu in the transpose graph.

Second Pass (DFS on Transposed Graph):

Pop vertices from the stack (ordered by finishing times from the first pass).

Perform DFS on the transpose graph starting from each popped vertex to explore all vertices in the same SCC.

Each DFS call from an unvisited vertex in the stack identifies a new SCC.

Example:

Consider a directed graph with vertices V = {1, 2, 3, 4, 5, 6} and edges {(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4)}.

First Pass (Original Graph):

Perform DFS on the original graph.

Track finishing times: finishing time (1) > finishing time (2) > finishing time (3) > finishing time (4) > finishing time (5) >finishing time (6).

Vertices in order of decreasing finishing times: [6, 5, 4, 3, 2, 1].

Transpose Graph:

Reverse all edges: {(2, 1), (3, 2), (1, 3), (4, 3), (5, 4), (6, 5), (4, 6)}.

Second Pass (DFS on Transposed Graph):

Start DFS from vertex 6 (top of the stack).

Explore all vertices reachable from 666: {6, 5, 4} forms an SCC.

Continue with other unvisited vertices in the stack until all SCCs are identified.

Applications:

Compiler Design: Used in optimizing code by identifying code blocks that can be executed independently.

Network Analysis: Identifying clusters of nodes that can communicate effectively.

Component-based Systems: Finding modules or components with interdependencies that must be analyzed together.

Kosaraju's algorithm efficiently finds all SCCs in O (V + E) time, making it suitable for large graphs encountered in real-world applications like social networks, transportation networks, and data flow analysis.

12.9 CONCLUSION

In this unit, we have explored a comprehensive range of topics centered around graph theory and its applications. We began with the fundamental concepts of graphs, delving into various ways they can be represented and manipulated. This foundational knowledge set the stage for understanding more complex algorithms and their practical uses.

We examined Minimum Cost Spanning Trees and studied Kruskal’s and Prim’s algorithms in detail. These algorithms are crucial for solving optimization problems in network design and other fields. We then moved on to essential graph traversal techniques, such as Breadth-First Search (BFS) and Depth-First Search (DFS), highlighting their implementation and diverse applications, from pathfinding to topological sorting.

Finally, we explored the identification of Strongly Connected Components (SCCs) in directed graphs, an important concept for analyzing the structure of complex networks. Throughout this unit, the emphasis has been on both theoretical understanding and practical implementation, equipping you with the skills to apply these powerful graph algorithms to real-world problems.

12.10 QUESTIONS AND ANSWERS

Q1: What is a graph in the context of data structures?
Answer: A graph is a data structure that consists of a set of nodes (vertices) connected by edges. Graphs can be directed or undirected, and they are used to represent relationships between entities in various applications such as social networks, transportation systems, and network topology.

Q2: What is the primary difference between Kruskal's Algorithm and Prim's Algorithm for finding Minimum Cost Spanning Trees?
Answer: Kruskal's Algorithm builds the Minimum Cost Spanning Tree (MST) by adding edges in order of increasing weight, ensuring no cycles are formed. Prim's Algorithm, on the other hand, starts from an arbitrary node and grows the MST by adding the smallest edge that connects a vertex in the tree to a vertex outside the tree.

Q3: How does the Breadth-First Search (BFS) algorithm work?
Answer: BFS is a graph traversal algorithm that starts from a given node and explores all its neighbors at the present depth before moving on to nodes at the next depth level. It uses a queue to keep track of nodes to be explored, ensuring a level-order traversal.

Q4: What are the typical applications of Depth-First Search (DFS)?

Answer: DFS is used in various applications, including cycle detection in graphs, topological sorting, solving maze puzzles, and finding connected components in a graph. It is characterized by its use of a stack or recursion to explore as far as possible along each branch before backtracking.

Q5: Explain the concept of Strongly Connected Components (SCCs) in a graph.

Answer: Strongly Connected Components (SCCs) are subgraphs in a directed graph where every node is reachable from every other node within the same subgraph. Identifying SCCs is crucial for understanding the structure of complex networks, such as identifying clusters or modules within a larger system.

Q6: What are the key properties of a Minimum Cost Spanning Tree (MCST)?

Answer: A Minimum Cost Spanning Tree (MCST) connects all vertices in a graph with the minimum possible total edge weight, without forming any cycles. It ensures that the spanning tree is as light as possible, which is essential for optimizing network design and other applications.

12.11 REFERENCES

- **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- **Sedgewick, R., & Wayne, K.** (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
- **Dasgupta, S., Papadimitriou, C., & Vazirani, U.** (2008). *Algorithms*. McGraw-Hill Education.

- **Kleinberg, J., & Tardos, É.** (2005). *Algorithm Design*. Pearson.
- **Tarjan, R. E.** (1972). "Depth-First Search and Linear Graph Algorithms". *SIAM Journal on Computing*, 1(2), 146-160.
- **Prim, R. C.** (1957). "Shortest Connection Networks and Some Generalizations". *Bell System Technical Journal*, 36(6), 1389-1401.
- **Kruskal, J. B.** (1956). "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". *Proceedings of the American Mathematical Society*, 7(1), 48-50.

UNIT – 13: SEARCHING AND SORTING ALGORITHMS

13.0 INTRODUCTION

In computer science and software engineering, sorting and searching algorithms are essential for efficiently managing and manipulating data. These algorithms are essential resources that facilitate the effective arrangement and retrieval of data for a wide range of applications. We thoroughly examine sorting and searching algorithms in this topic, including their foundations, applications, effectiveness, and practical applications.

Sorting algorithms are methods for putting data in a specific order, like lexicographical or numerical order. Because of their differences in efficiency and complexity, some are more suited for particular jobs than others. Conversely, searching algorithms make it easier to retrieve data from structured data structures. They are essential for quickly and efficiently finding elements in sorted arrays or other data repositories.

The first part of this subject looks at different sorting strategies, such as Quick Sort, Insertion Sort, Selection Sort, and Merge Sort. Every technique is examined in detail to comprehend its working principles, computational intricacies in the best, average, and worst-case situations, and useful applications. Additionally, we investigate and compare the optimal use cases and efficiency of several searching algorithms, including Binary Search and Linear Search. Our goal is to offer a thorough grasp of how these algorithms support effective data management and retrieval in theoretical and real-world settings by the conclusion.

13.1 OBJECTIVES

After completing this unit, you will be able to understand,
Understanding Sorting Algorithms: To comprehend the fundamental principles behind various sorting algorithms such as Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. This includes exploring their respective implementation strategies, advantages, and disadvantages.

Analyzing Time and Space Complexity: To conduct a detailed analysis of the time complexity (best, average, and worst-case scenarios) and space complexity of each sorting algorithm. This analysis helps in understanding their efficiency and suitability for different data sizes and structures.

Exploring Searching Algorithms: To investigate essential searching algorithms, particularly Linear Search and Binary Search. This involves understanding their operational mechanisms, efficiency in terms of time complexity, and practical applications in data retrieval scenarios.

Comparing Efficiency: To compare the efficiency of sorting and searching algorithms based on their time and space complexities. This comparison aids in selecting the most appropriate algorithm for specific tasks, considering factors like data size, structure, and order.

Real-world Applications: To examine practical applications of sorting and searching algorithms across various domains, including database management, information retrieval, and computational problem-solving.

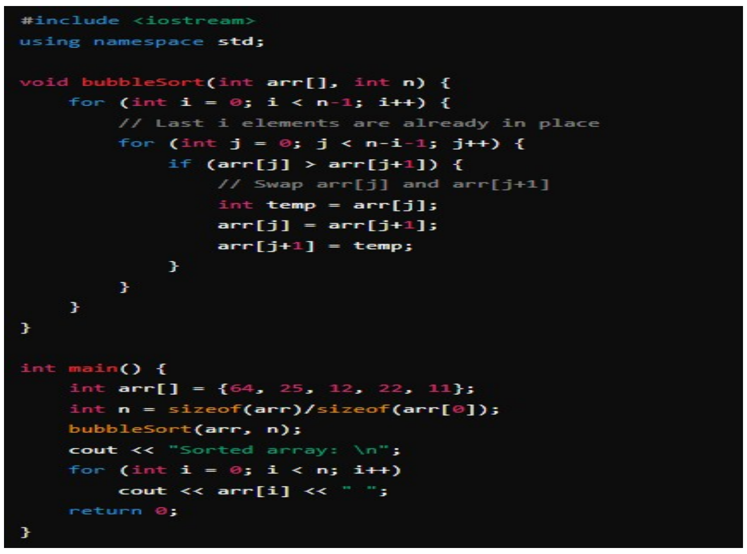
13.2 SORTING ALGORITHMS

Sorting algorithms are methods for putting items in a list or array in a specific order, usually lexicographically or numerically, ascending or descending. A basic operation in computer science, sorting is frequently employed as an initial step in a variety of algorithms and applications. Sorting algorithms can be assessed for efficiency using two metrics: space complexity (the amount of additional memory needed) and time complexity (the rate at which the algorithm's execution time grows with the size of the input).

Sorting Techniques:

Bubble Sort: This straightforward sorting algorithm analyzes each pair of adjacent items and swaps them if they are out of order as it iteratively goes through the list to be sorted. Until the list is sorted, the trip through the list is repeated. Smaller elements "bubble" to the top of the list (the beginning of the array) with each iteration of the algorithm, hence its name.

Methodical Application in C++



```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Analysis of Time Complexity

Best Case: O(n): When the array has previously been sorted, this happens. In this case, the algorithm merely performs one trip around the array to verify that it is sorted; swaps are not required.

Average Case: O(n^2): This happens when there is an average random element order. The method does n passes, comparing and sometimes swapping neighboring pairings on each iteration.

Worst-Case Scenario: The worst-case scenario, which happens when the array is sorted in reverse order, is O(n2). The method must perform n passes, requiring n-1 swaps and comparisons for each pass.

Analysis of Space Complexity

Space Complexity: O (1): Bubble Sort requires a constant amount of additional memory space because it is an in-place sorting method. The temporary variable that holds elements during swaps is the only purpose for the additional space.

Detailed Description with Illustration

Considering an array arr[] = {64, 25, 12, 22, 11}, let us perform the Bubble Sort steps:

First Pass,

When comparing 64 and 25, swap them out because 64 > 25 → {25, 64, 12, 22, 11}

As 64 > 12 → {25, 12, 64, 22, 11}, compare 64 and 12: Swap

Examine 64 and 22: Exchange, as 64 > 22 → {25, 12, 22, 64, 11}

Examine 64 and 11: Exchange, as 64 > 11 → {25, 12, 22, 11, 64}

Second Pass,

In contrast, swap 25 and 12 because 25 > 12 => {12, 25, 22, 11, 64}

In order to swap 25 and 22, remember that 25 > 22 → {12, 22, 25, 11, 64}.

In order to swap 25 and 11, note that 25 > 11 → {12, 22, 11, 25, 64}.

64 is already positioned correctly.

Third Pass

Examine 12 and 22: There is no exchange because 12 < 22 → {12, 22, 11, 25, 64}

In contrast, 22 > 11 → {12, 11, 22, 25, 64}. Compare 22 with 11.

The numbers 25 and 64 are already in the right places.

Fourth Pass:

As 12 > 11 → {11, 12, 22, 25, 64}, compare 12 with 11.

The numbers 22, 25, and 64 are already in the right place.

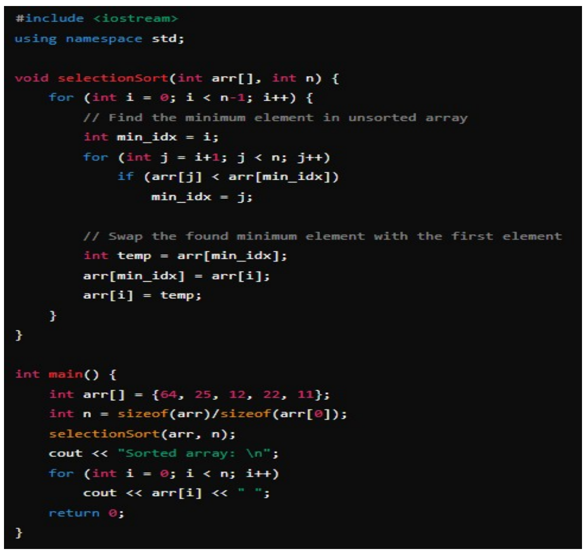
The array is sorted in the following order: {11, 12, 22, 25, 64}.

13.3 SELECTION SORT

Selection Sort is a basic sorting algorithm that relies on comparisons. The input list is split into two sections by the algorithm: a sublist of the remaining unsorted items and a sorted sublist of items that are accumulated from left to right at the front (left) of the list.

The input list as a whole is the unsorted sublist at first, and the sorted sublist is empty. The method then finds the smallest (or largest, depending on the order of sorting) element in the unsorted sublist, moves the sublist borders one element to the right, and exchanges it with the leftmost unsorted element to put it in sorted order.

Step-by-Step Implementation in C++



```
#include <iostream>
using namespace std;

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Analysis of Time Complexity

Best Case: O(n^2): Because the technique does not check to see if the list is already sorted, the best-case scenario still requires n passes through the list and n comparisons each pass.

Average Case: O(n^2): In this scenario, the algorithm runs through n passes, averaging n/2 comparisons each pass.

Worst Case: O(n^2): In this scenario, there are n comparisons made for each run through the list and n passes overall.

Analysis of Space Complexity

Because Selection Sort is an in-place sorting algorithm, it has a constant memory space need (space complexity: O (1)). The temporary variable that holds elements during swaps is the only purpose for the additional space.

Use Cases and Realistic Implementations

Small Data Sets: Selection Sort is helpful when dealing with small data sets since its simplicity and convenience of usage outweigh its drawbacks.

Partially Sorted Arrays: Selection Sort occasionally works better if you are aware that the array has previously been partially sorted.

Educational Purposes: Because of its simplicity, it is frequently used in school contexts to teach the principles of sorting algorithms.

Memory-Constrained Environments: Because it is an in-place sort, it can be used in settings with constrained memory.

Detailed Description with Illustration

Considering an array arr[] = {64, 25, 12, 22, 11}, let us perform the Selection Sort steps:

First Pass:

Determine the least element—11—among 64, 25, 12, 22, and 11.

Replace element 64 with 11, resulting in {11, 25, 12, 22, 64}.

Second Pass:

Determine the least element—12—among 25, 12, 22, and 64.

Replace element 12 with element 25 to get {11, 12, 25, 22, 64}.

Third Pass

Determine the least element—22—among 25, 22, 64.

Replace element 22 with the third one (25) to get {11, 12, 22, 25, 64}.

Fourth Pass:

Determine the least element—25—among 25, 64.

Since 25 is already in the right place, there is no need to swap.

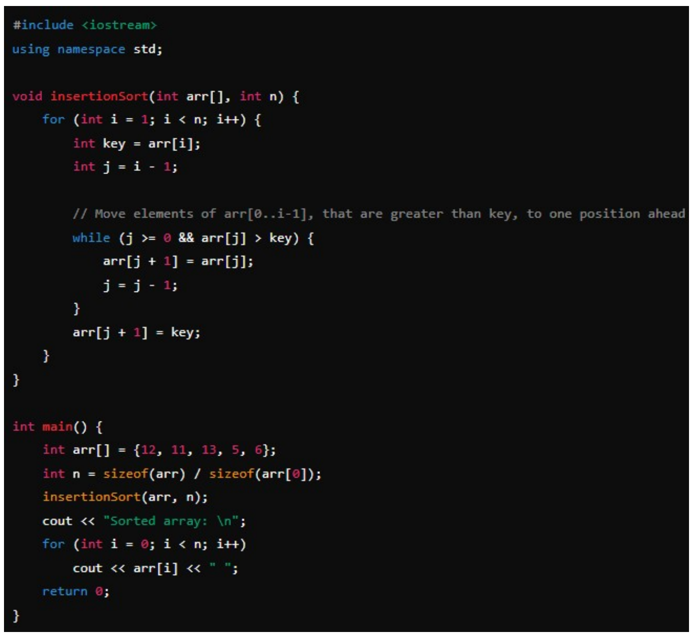
The array is sorted in the following order: {11, 12, 22, 25, 64}.

13.4 INSERTION SORT

Insertion Sort is a basic and intuitive comparison-based sorting algorithm. It builds the final sorted array (or list) one item at a time. It is substantially less efficient on huge lists than more complex algorithms such as quicksort, heapsort, or merge sort. However, it has the virtue of being simple to implement and efficient for small data sets.

The list is split into sorted and unsorted regions in order for the algorithm to function. Initially, the sorted region comprises only the first element, and the rest of the list is unsorted. The method proceeds by taking the next element from the unsorted section and inserting it into the correct position in the sorted region. This process continues until the full list is sorted.

Step-by-Step Implementation in C++



```cpp
#include <iostream>
using namespace std;

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Time Complexity Analysis

Best Case: O(n): The best-case scenario happens when the array is already sorted. The algorithm just needs to pass through the list once, making n-1 comparisons.

Average Case: O(n^2): On average, each element in the array is compared half of the elements preceding it. This leads in a quadratic amount of comparisons and shifts.

Worst Case: O(n^2): The worst-case scenario happens when the array is sorted in reverse order. The method needs to bring each element to the front of the sorted zone, resulting in the maximum amount of comparisons and shifts.

Analysis of Space Complexity

Space Complexity: O(1): Insertion Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional memory space. The only extra space used is for a temporary variable to hold components during shifts.

Use Cases and Realistic Implementations

Small Data Sets: Insertion Sort works well for small data sets since it is straightforward to use and has a low implementation overhead.

Nearly Sorted Arrays: Insertion Sort works well when you know the array is already almost sorted because it requires less shifts.

Online Sorting: Insertion Sort is appropriate for online sorting where data is received one piece at a time since it can sort a list as it gets items.

For educational purposes: Because of its simplicity, it is frequently used in school contexts to teach the principles of sorting algorithms.

Detailed Description with Illustration

Considering an array arr[] = {12, 11, 13, 5, 6}, let us proceed with the Insertion Sort steps:

First Pass:

Key = 11, contrast with 12.

Move 12 to the right since 11 < 12.

Put 11 where it belongs → {11, 12, 13, 5, 6}.

Second Pass:

Key = 13, contrast with 12.

There is no need for shifts because 13 > 12.

Place 13 where it belongs → {11, 12, 13, 5, 6}.

Third Pass:

Key: 5, with relation to 13, 12, and 11.

Due to the fact that 5 < 13, 12, and 11, move them right.

Put 5 in the proper place: {5, 11, 12, 13, 6}.

Fourth Pass:

Key: 6, in relation to 13, 12, and 11.

As 6 is less than 13, 12, and 11, move them to the right.

Put 6 in the proper place: {5, 6, 11, 12, 13}.

The array is sorted in the following order: {5, 6, 11, 12, 13}.

13.5 MERGE SORT

A divide-and-conquer method called merge sort splits the input array in half, sorts each half recursively, and then combines the sorted halves to create a sorted array. O (n log n) time complexity is guaranteed in all scenarios (best, average, and worst-case), making it a stable sorting algorithm.

Step-by-Step Implementation in C++

The basic instance of the C++ implementation of merge sort is splitting the array in half recursively until each sub-array has one entry. To create the final sorted array, it then combines the sorted sub-arrays once more.

```
#include <iostream>
using namespace std;
// Function to merge two halves sorted arrays
void merge(int arr[], int left, int mid, int right) {
  int n1 = mid - left + 1;
  int n2 = right - mid;
    // Create temporary arrays
  int L[n1], R[n2];
    // Copy data to temporary arrays L[] and R[]
  for (int i = 0; i < n1; i++)
    L[i] = arr[left + i];
  for (int j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];
    // Merge the temporary arrays back into arr[left..right]
  int i = 0; // Initial index of first subarray
  int j = 0; // Initial index of second subarray
  int k = left; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
        // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Function to perform merge sort on array arr[left..right]
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
                // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
```

```
}
// Main function to test merge sort
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
        cout << "Given array is \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
        mergeSort(arr, 0, n - 1);
        cout << "Sorted array is \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
        return 0;
}
```

Analysis of Time Complexity

Best Case: O (n log n): When the array is sorted or almost sorted, this is the best-case situation. The array is split in half by Merge Sort until each sub-array contains one element, at which point it merges the two halves back together.

Merge Sort: Merge Sort on average, splits the array in half and then combines them back together in an O (n log n) time. Its time complexity is expressed as $T(n) = 2T(n/2) + O(n)$ in the recurrence relation.

Worst Case: O (n log n): When the array is unsorted, this is also the worst-case situation. Recursively splitting the array in half and merging them together, Merge Sort preserves O (n log n) time complexity.

Analysis of Space Complexity

Space Complexity: O(n): The temporary arrays utilized during the merge operation necessitate additional memory space for Merge Sort. O(n) is the space complexity because auxiliary arrays are used.

Use Cases and Realistic Implementations

Sorting Big Data Sets: Because of its O(n log n) time complexity, merge sort is effective for sorting huge data sets.

External Sorting: Merge sort is used in external sorting when data needs to be stored on external storage devices because it cannot fit in the main memory.

Parallel Processing: Merge Sort is easily adaptable to parallel processing, which allows various parts of the data to be sorted simultaneously by several processors or cores.

13.6 QUICK SORT

Quick Sort is a sorting algorithm that uses comparison and the divide-and-conquer tactic. To operate, one 'pivot' element is chosen from the array, and the remaining elements are divided into two sub-arrays according to whether they are bigger or less than the pivot. After that, the sub-arrays are sorted recursively.

Step-by-Step Implementation in C++

In order to implement Quick Sort in C++, one must first choose a pivot element, divide the array around it, and then recursively sort the sub-arrays. The main steps involved are as follows:

Select Pivot: The pivot should be one of the array's elements. There are several ways to determine which element will serve as the pivot: you can choose to use the first, last, or random element.

Partitioning: Slide the array back and forth until all elements larger than the pivot are on the right side and all elements less than the pivot are on the left. The pivot is in its final position following partitioning.

Recursive Sort: Sort the sub-arrays created by partitioning by applying Quick Sort recursively until the full array is sorted.

```
#include <iostream>
#include <vector>
using namespace std;
// Function to partition the array and return the index of the pivot
element
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
    int i = low - 1; // Index of smaller element
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
// Function to implement Quick Sort
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);
        // Recursively sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Utility function to print an array
void printArray(const vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
// Main function
int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    int n = arr.size();
    cout << "Original array: ";
    printArray(arr);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
   printArray(arr);
    return 0;
}
```

Time Complexity Analysis

Best Case: O(n log n): The array is divided into two roughly equal halves by the pivot in the best-case scenario. The array is divided into two pieces by each partitioning step, resulting in O(log n) divisions and O(n) comparisons for each division.

Average Case: O(n log n): Quick Sort's effective partitioning technique allows it to perform well on average. Because each partitioning step splits the array into two sub-arrays proportionate to the pivot, the temporal complexity is O(n log n).

Worst Case: O(n^2): Unbalanced partitions result when the pivot is either the smallest or largest element in the array. This is the worst-case scenario. O(n^2) time complexity results from this situation, which is uncommon but can be avoided by deliberately selecting the pivot.

Analysis of Space Complexity

Space Complexity: O (log n) to O(n): The recursive call stack for Quick Sort normally takes up O(log n) of space. If more arrays are used in the implementation, in the worst scenario, O(n) auxiliary space might be needed for partitioning.

Use Cases and Realistic Implementations

General-Purpose Sorting: Because of its effective average-case performance, Quick Sort is frequently used for general-purpose sorting.

In-Place Sorting: Quick Sort can be done in-place for the recursive call stack using O(log n) auxiliary space.

Optimized Libraries: Because Quick Sort and its variants are reliable and efficient sorting algorithms, many computer languages and libraries utilize them by default.

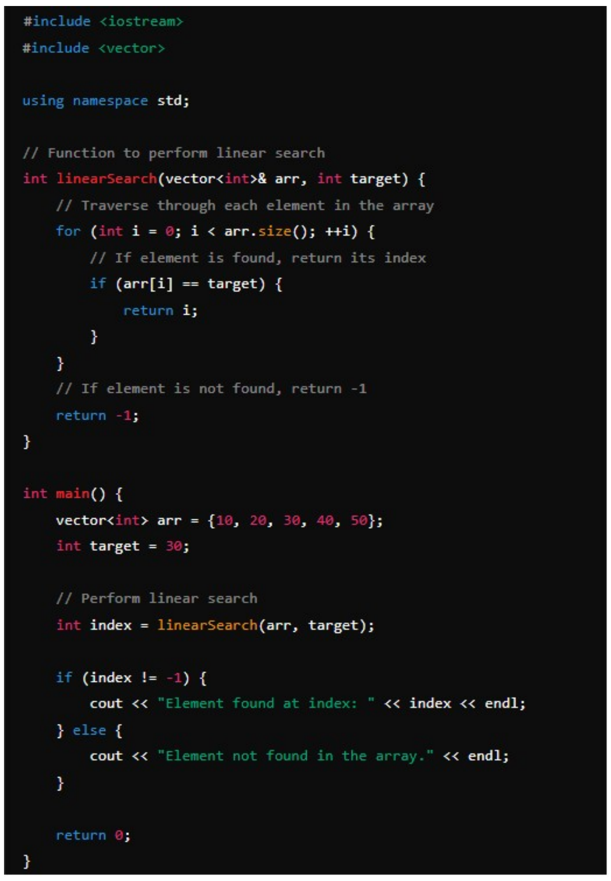
13.7 SEARCHING ALGORITHMS

Algorithms for searching are techniques for locating particular elements in a set of data, like trees, arrays, or lists. Finding the location of a specific element within the data structure and, if it does, retrieving it are the objectives. The efficiency of various searching algorithms vary, and they are frequently assessed in terms of space complexity, time complexity, and overall performance depending on the quantity and quality of the data.

The following are a few popular search algorithms:

- **Linear Search:** another name for linear search, is a simple searching technique that goes over each element in a data structure one after the other until the target element is located or all the items have been examined. When data is randomly arranged or unsorted and each element is compared one after the other consecutively, it performs admirably.

Step-by-step Implementation in C++:



```
#include <iostream>
#include <vector>

using namespace std;

// Function to perform linear search
int linearSearch(vector<int>& arr, int target) {
    // Traverse through each element in the array
    for (int i = 0; i < arr.size(); ++i) {
        // If element is found, return its index
        if (arr[i] == target) {
            return i;
        }
    }
    // If element is not found, return -1
    return -1;
}

int main() {
    vector<int> arr = {10, 20, 30, 40, 50};
    int target = 30;

    // Perform linear search
    int index = linearSearch(arr, target);

    if (index != -1) {
        cout << "Element found at index: " << index << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```

Analysis of Time Complexity:

Best Case: O(1): This happens when the element of interest is located at the initial position.

Average Case: O(n): - The average case requires the algorithm to scan through half of the array on average, since the target element can be anywhere in the array.

Worst Case: O(n): - This situation necessitates a complete traversal of the array and happens when the target element is at the last position or absent.

Analysis of Space Complexity:

Because it only needs a fixed amount of additional memory to store variables like the loop counter and target element, Linear Search has a space complexity of O(1).

Uses and Real-World Implementations:

Searching Unsorted Arrays: Because linear search examines each element one after the other, it is frequently employed when data is not sorted.

Simple and Easy to Implement: It is helpful in circumstances where efficiency is not as crucial as simplicity and ease of implementation.

Tiny Datasets: Appropriate for tiny datasets or situations in which sorting using more sophisticated algorithms, such as Binary Search, would not be cost-effective.

Binary Search: Working with sorted arrays or lists, Binary Search is a very effective searching technique. Until the target element is located or the interval is empty, it operates by periodically halving the search interval in half. Depending on whether the target value is higher or less than the array's middle element, it compares it to that element before determining whether to carry on looking in the left or right subarray.

Step-by-step Implementation in C++:

```
#include <iostream>
#include <vector>
using namespace std;
// Function to perform binary search
int binarySearch(vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid;
        }
        // If target is greater, ignore left half
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        // If target is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }
    // If target is not found in the array
    return -1;
}
int main() {
    vector<int> arr = {10, 20, 30, 40, 50, 60};
    int target = 40;
    // Perform binary search
    int index = binarySearch(arr, target);
    if (index != -1) {
        cout << "Element found at index: " << index << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }
    return 0;
}
```

Analysis of Time Complexity:

In the best scenario, the target element is located in the middle of the array (O(1)).

Average Case: O(log n) - Binary search is very effective for huge datasets since it splits the search interval in half with each comparison.

The worst situation, which is similar to the average scenario, happens when the target element is at either extreme of the array. It is expressed as O(log n).

Analysis of Space Complexity:

Because it only needs a fixed amount of additional memory to store variables like the left, right, and mid indices, Binary Search has a space complexity of O(1).

Uses and Real-World Implementations:

Sorted Arrays and Lists: Binary search works well for searching in sorted arrays and lists that allow for random access.

Effective Searching: Because of its logarithmic time complexity, it performs substantially quicker than linear search on huge datasets.

Algorithmic Foundations: In computer science, binary search is a basic algorithm that forms the basis of more intricate algorithms and data structures.

14.8 COMPARING THE EFFICIENCY OF SORTING AND SEARCHING ALGORITHMS

Comparing Time Complexities of Sorting Algorithms:

Bubble Sort:

Best Case: O(n) - Occurs when the array is already sorted.

Average Case: O(n^2)

Worst Case: O(n^2) - Occurs when the array is sorted in reverse order.

Selection Sort:

Best Case: O(n^2)

Average Case: O(n^2)

Worst Case: O(n^2)

Insertion Sort:

Best Case: O(n) - Occurs when the array is already sorted.

Average Case: O(n^2)

Worst Case: O(n^2)

Merge Sort:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Quick Sort:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) - Occurs when the pivot is consistently the smallest or largest element.

Comparing Space Complexities of Sorting Algorithms:

Bubble Sort: O(1) - In-place algorithm.

Selection Sort: O(1) - In-place algorithm.

Insertion Sort: O(1) - In-place algorithm.

Merge Sort: O(n) - Requires additional space for merging.

Quick Sort: O(log n) - Space complexity is dominated by the call stack due to recursion.

Best Scenarios to Use Each Sorting Algorithm:

Bubble Sort, Selection Sort, Insertion Sort: These algorithms are simple and efficient for small datasets or nearly sorted arrays due to their O(n) best case scenarios.

Merge Sort: Suitable for sorting large datasets or when stable sorting is required (maintaining the relative order of equal elements).

Quick Sort: Preferred for average and best-case scenarios due to its average O(n log n) time complexity and in-place partitioning.

13.9 CONCLUSION

In conclusion, learning about sorting and searching algorithms is essential to comprehending the basic principles of computer science that underlie effective data organizing and retrieval. This lesson covered a variety of sorting algorithms, such as Quick Sort, Insertion Sort, Selection Sort, and Merge Sort. Based on their time and space complexity, each sorting algorithm offers a distinct method for sorting data with differing degrees of efficiency. Comparably, we examined fundamental searching algorithms including Binary Search and Linear Search, emphasizing their efficiency metrics and working principles in various contexts.

Furthermore, contrasting these algorithms revealed information about their respective advantages and disadvantages, which increased our understanding of the significance of algorithmic efficiency in practical applications. In addition to being fundamental to software development, sorting and searching algorithms are also vital in domains where effective data handling is required, such as data analysis, database administration, and computational research.

Basically, knowing these algorithms gives practitioners useful abilities to improve system performance, optimize data processing jobs, and efficiently handle challenging computational problems.

13.10 QUESTIONS AND ANSWERS

1. Why does sorting and searching depend on an algorithm's time complexity?
Answer: The answer is that temporal complexity quantifies how an algorithm's runtime grows as the amount of input data grows. Lower time complexity algorithms are chosen for sorting and searching because they operate more quickly, particularly for huge datasets.

2. Contrast Binary and Linear Search. Which would you prefer to use, and when?
Answer: The answer is that Linear Search works well with unsorted lists because it iteratively examines each element in the list until the target element is located. Contrarily, Binary Search is quicker for huge datasets since it only needs a sorted list and effectively reduces the search space at each step.

3. Describe the idea of sorting algorithms' stability. Why does it matter?
Answer: The answer is that elements with equal keys appear in the sorted output in the same order as they do in the original input because sorting algorithms are stable. When sorting data using multiple keys (e.g., sorting employees by department and then by name), this feature is essential.

4. How is the O(n log n) average-case time complexity of Quick Sort achieved?
Answer: In response, Quick Sort divides the array recursively into smaller subarrays depending on a pivot element and sorts each subarray separately to reach O(n log n) average-case complexity. By using a divide and conquer approach, the workload is balanced and the number of comparisons required is decreased.

5. Give an instance in which Merge Sort would not be preferred over Insertion Sort.
Answer: Because of its ease of use and effective performance on tiny datasets, insertion sort is the method of choice for sorting small arrays or almost sorted arrays. Merge Sort, on the other hand, works well with large datasets because of its O(n log n) complexity.

6. What is the temporal complexity of Selection Sort and how does it operate?
Answer: To answer your question, selection sort operates by repeatedly identifying the minimum element in the array's unsorted section and replacing it with the element that was initially unsorted. For large datasets, it is less efficient than algorithms like Quick Sort or Merge Sort due to its O(n^2) time complexity.

7. Talk about the trade-offs sorting algorithms have between time and space complexity.
Answer: Because they may sort data in place, altering the input array, sorting algorithms with higher time complexity frequently require less additional memory (lower space complexity). Higher space complexity algorithms can offer faster runtime, but they may require more data structures to help in sorting.

13.11 REFERENCES

- Bjarne Stroustrup, "The C++ Programming Language"
- Herb Sutter, "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions"
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, "C++ Primer"
- Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Nicolai M. Josuttis, "C++ Standard Library: A Tutorial and Reference"

BLOCK IV: FILE STRUCTURES AND ADVANCED DATA STRUCTURES

UNIT – 14: HASHING

Structure

14.0 INTRODUCTION

Effective system performance in the fields of computer science and data management frequently rests on efficient data retrieval. Hashing is a key idea in this field that provides a reliable way to quickly arrange and retrieve data. Fundamentally, hashing maps data pieces to distinct index values inside a data structure called a hash table using hash algorithms. This method directly accesses the stored value linked to a computed index, enabling quick data retrieval. Hashing is therefore essential for maximizing the effectiveness of a variety of processes, including algorithmic calculations and database administration.

Gaining an understanding of hashing requires exploring its theoretical underpinnings as well as its real-world applications. The foundation of hashing approaches is the idea of hash functions, which transform data of any size into a fixed-size result. The purpose of hash functions is to generate hash codes with characteristics such as collision resistance, predictable calculation, and uniform distribution. These characteristics guarantee that hash tables can effectively manage big datasets while preserving data integrity and cutting down on retrieval times. This unit delves further into these concepts, offering insights into the implementation and optimization of hash functions to meet a range of computing requirements.

Furthermore, collision resolution methods are essential to hash tables' dependability and efficiency. When two distinct keys hash to the same index, a collision occurs, requiring a resolution strategy. Effective collision mitigation requires the use of strategies like open addressing, which includes searching for empty slots until one is discovered, and chaining, which stores several keys that hash to the same index in linked lists within the same table slot. Learners can comprehend the subtleties of hash table management and recognize the vital part these approaches play in contemporary computing infrastructures by grasping these strategies.

14.1 OBJECTIVES

After completing this unit, you will be able to understand,

Understanding of Hashing: Gain a clear understanding of hashing as a fundamental technique for efficient data retrieval. This includes grasping the concept of hash functions, their properties, and how they are applied to map data elements to unique indices.

Collision Resolution Techniques: Explore various collision resolution techniques used in hash tables. Learn how methods like chaining and open addressing handle collisions and maintain the integrity and efficiency of hash-based data structures.

Importance of Collision Resolution: Recognize the significance of collision resolution in hash tables. Understand how effective collision resolution techniques contribute to improving the performance and reliability of data retrieval operations.

Applications of Hashing: Explore real-world applications where hashing is instrumental in data storage and retrieval. This includes database indexing, caching mechanisms, symbol tables in compilers, and data deduplication strategies.

Practical Knowledge: Acquire practical knowledge through examples and implementations that illustrate the concepts of hashing, collision resolution, and their applications. Develop skills in designing and optimizing hash-based data structures for various computational tasks.

14.2 INTRODUCTION TO HASHING

In computer science and data structures, hashing is a basic method for quickly and effectively storing and retrieving data. In order to map data of any size to fixed-size values, usually integers, known as hash codes or hash values, a hash function is applied. By reducing the temporal complexity of accessing items, hashing aims to achieve efficient data retrieval and storage processes.

To be more specific, hashing is the process of taking an input (or key), applying a hash function to get an index (or hash code), and then storing or retrieving the appropriate data (or value) from a data structure (like a hash table). Usually, the hash function processes the key via some sort of calculation and outputs an index that points directly to the position of the data in the underlying storage. This makes it useful for situations where fast access is essential, such database indexing, caching techniques, and various algorithmic applications. It also provides for speedy insertion, deletion, and retrieval of data pieces.

Since hashing typically yields average-case constant-time operations for search, insert, and delete operations—assuming that a suitable collision resolution approach and a strong hash function are put into place—it is a widely utilized technique. It serves as the foundation for a number of data structures, including dictionaries, hash tables, and hash maps, all of which are crucial in contemporary computer science for handling massive datasets and maximizing efficiency.

Qualities of an Effective Hash Function

Even Distribution: The hash codes should be dispersed equally throughout the hash table or array by a strong hash algorithm. In other words, the function should reduce the amount of collisions—that is, the instances in which two distinct keys map to the same index or bucket. By ensuring that every bucket in the hash table has an equal chance of being reached, uniform distribution maximizes operational efficiency.

Deterministic: For a given input key, the hash function should always produce the same hash code. Because determinism guarantees predictability and dependability, it makes it possible to update and retrieve stored data consistently. Stated otherwise, you should consistently obtain the same hash code if you hash the same key many times.

Quick Computation: A strong hash function should be computationally efficient since hashing relies heavily on efficiency. All hash codes should be generated rapidly, irrespective of the input key's size. This guarantees that in applications requiring frequent data access or manipulation, the hashing process itself does not constitute a bottleneck.

Common Hash Function Examples

Division Method: This is one of the most basic hash functions; it calculates the hash code by dividing the hash table size by the remainder of the key (a modulo operation). As an example, consider hash(key) = key % table_size. Even though it is straightforward, if the table size is not prime, improper selection may result in clustering.

Multiplication Method: In this method, the fractional part of the product is multiplied by the table size after the key is multiplied by a constant, usually a fraction of a power of two. As an illustration, consider the formula hash(key) = floor (table_size * (key * A % 1)), where A is a constant selected for acceptable distribution through actual research.

Universal Hashing: Using a random selection process, a family of hash functions is employed to choose which particular hash function is used. Because of its inherent randomness, it can be used in situations where security or resilience against enemies are crucial. Selecting the hash function according to the size of the hash table and the characteristics of the data being hashed is a widely used method in universal hashing.

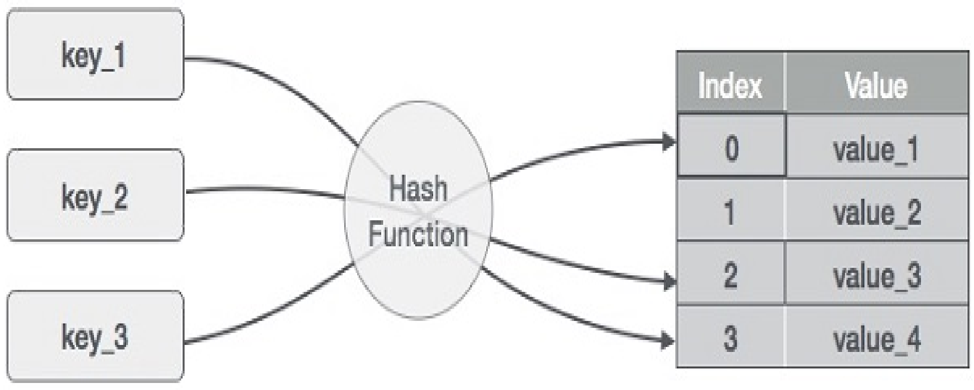


Image Source: TutorialsPoints

14.3 COLLISION RESOLUTION TECHNIQUES

When two distinct keys hash to the same index or location in a hash table, this is known as a collision in hashing. This indicates that numerous keys are assigned to the same slot by the hash function, which maps keys to locations in the hash table.

When Collisions Occurs?

When two distinct keys hash to the same index in a hash table, this is known as a hash collision. There are a number of possible causes for this, and knowing them is crucial to putting successful collision resolution techniques into practice. The following list of frequent collision causes is illustrated with examples:

Limitations of Hash Functions: Generally speaking, hash functions convert an infinite number of keys into a finite number of hash values, or indices. Collisions are unavoidable because of this mapping constraint, particularly when the number of unique keys (domain of input) surpasses the total number of potential hash values (range of output).

As an illustration, let's say we have a basic hash function that calculates the index by taking the table size and the modulo of the key. The hash function for keys 15 (hashValue = 15% 10) and 25 (hashValue = 25% 10) yields 5, for example, if the table size is 10 (TABLE_SIZE = 10).

Cause: If keys are not evenly distributed, the hash function's simplicity and lack of complexity may lead to multiple keys mapping to the same index.

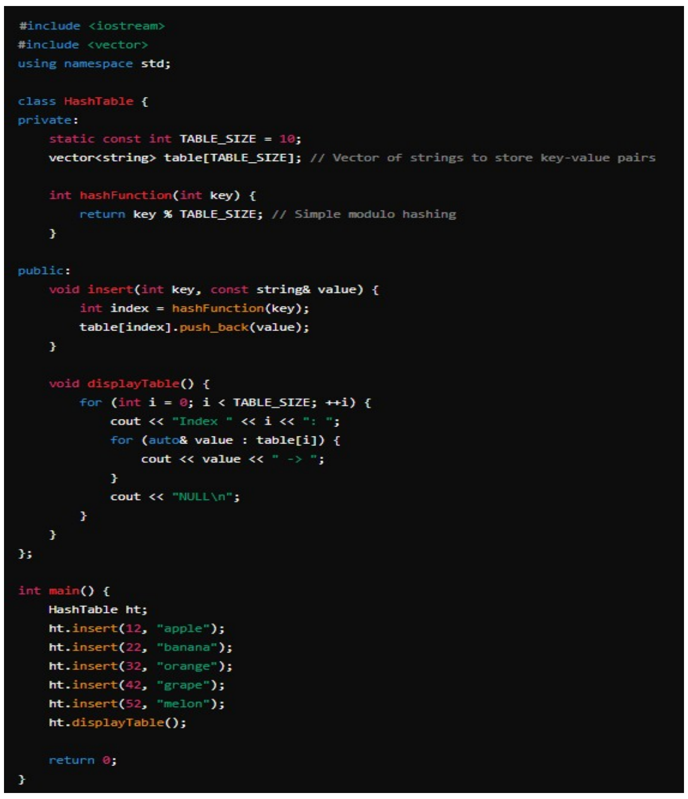
Limited Hash Table Size: When hashing a large number of keys, collisions become more likely if the hash table has a restricted number of buckets (or slots).

As an illustration, let's look at a hash table with the size TABLE_SIZE = 5. All of these keys will compute to index 2 (hashValue = key % 5), if we hash them 12, 22, 32, 42, and 52.

Cause: Collisions become more common when there are substantially less potential hash values (depending on table size) than there are keys being hashed.

Example Scenario

Examine a hash table that employs a basic modulo hashing method:



```cpp
#include <iostream>
#include <vector>
using namespace std;

class HashTable {
private:
    static const int TABLE_SIZE = 10;
    vector<string> table[TABLE_SIZE]; // Vector of strings to store key-value pairs

    int hashFunction(int key) {
        return key % TABLE_SIZE; // Simple modulo hashing
    }

public:
    void insert(int key, const string& value) {
        int index = hashFunction(key);
        table[index].push_back(value);
    }

    void displayTable() {
        for (int i = 0; i < TABLE_SIZE; ++i) {
            cout << "Index " << i << ": ";
            for (auto& value : table[i]) {
                cout << value << " -> ";
            }
            cout << "NULL\n";
        }
    }
};

int main() {
    HashTable ht;
    ht.insert(12, "apple");
    ht.insert(22, "banana");
    ht.insert(32, "orange");
    ht.insert(42, "grape");
    ht.insert(52, "melon");
    ht.displayTable();

    return 0;
}
```

Handling Collisions:

There are various approaches to successfully manage collisions:

Chaining: When using chaining, every hash table slot keeps track of every key that hashes to the same index in a linked list or other data structure. The new key is added to the linked list at the appropriate slot in the event of a collision. Every bucket in the hash table is a linked list when using distinct chaining. The colliding elements (key-value pairs) are added to the linked list that corresponds to their hash index in order to handle collisions.

For illustration, let's say we have a hash table with ten buckets. The keys "apple" and "banana" hash to the same index, which is, for example, index 3. Rather than replacing "apple" with "banana," "banana" is added to index 3 of the linked list. This produces a structure similar to:

```
Index 3: -> (apple, value1) -> (banana, value2) -> NULL
```

Benefits: Easy to implement, effectively manages several collisions.
Drawbacks: If many keys hash to the same index, performance suffers and there is additional memory expense because of linked lists.

Open Addressing: When a collision occurs, it can be avoided by probing or looking through the hash table's alternate slots until an empty one is located. Common methods of probing include double hashing (calculating the next slot using a secondary hash function), quadratic probing (using a quadratic function to identify the next slot), and linear probing (examining successive slots).

Idea: In open addressing, a different place is found (by probing) inside the hash table to resolve clashes.

Example: If a collision happens at a certain index using linear probing, the algorithm successively tries the next index until it locates an empty slot. In case "apple" hashes to index 3 and it's filled, the algorithm proceeds to check index 4, then 5 and so on until it finds an empty space.

Benefits: Less need for extra data structures, faster cache operation than chaining.
Drawbacks: It includes the possibility of performance loss with high load factors and increased clustering.

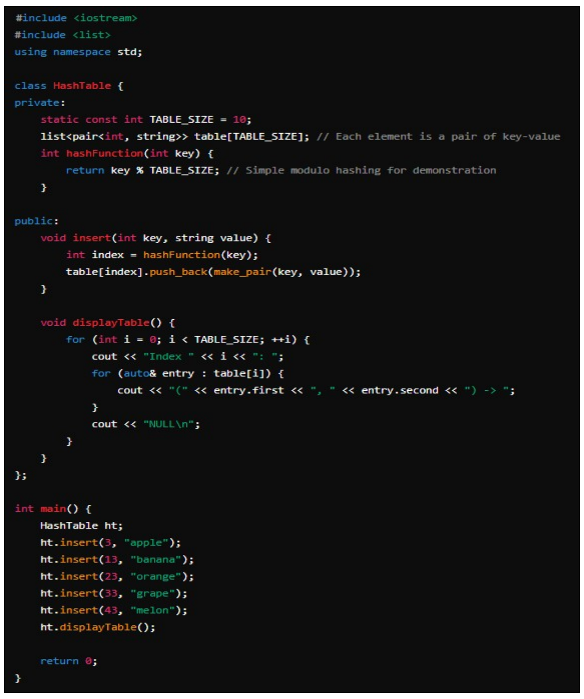
Double Hashing: This technique determines the time between probes by using a second hash function to handle collisions.

As an illustration, a secondary hash function decides the step size for probing if a collision happens at index 3. For instance, the algorithm would investigate index 5 (3 + 2) if the secondary hash function yielded a result of 2 for "apple".

Benefits: Effective for a variety of keys, helps prevent major clustering problems.
Cons: To guarantee uniform distribution, the secondary hash function must be implemented carefully.

Example Scenario

Think about a hash table that uses distinct chaining:



```cpp
#include <iostream>
#include <list>
using namespace std;

class HashTable {
private:
    static const int TABLE_SIZE = 10;
    list<pair<int, string>> table[TABLE_SIZE]; // Each element is a pair of key-value
    int hashFunction(int key) {
        return key % TABLE_SIZE; // Simple modulo hashing for demonstration
    }

public:
    void insert(int key, string value) {
        int index = hashFunction(key);
        table[index].push_back(make_pair(key, value));
    }

    void displayTable() {
        for (int i = 0; i < TABLE_SIZE; ++i) {
            cout << "Index " << i << ": ";
            for (auto& entry : table[i]) {
                cout << "(" << entry.first << ", " << entry.second << ") -> ";
            }
            cout << "NULL\n";
        }
    }
};

int main() {
    HashTable ht;
    ht.insert(3, "apple");
    ht.insert(13, "banana");
    ht.insert(23, "orange");
    ht.insert(33, "grape");
    ht.insert(43, "melon");
    ht.displayTable();

    return 0;
}
```

14.4 IMPORTANCE OF COLLISION RESOLUTION

For hash tables to continue operating effectively and performing as intended, efficient collision resolution is essential. Negatively handled collisions can make a hash-based data structure less effective overall by lengthening search times and decreasing operational efficiency.

For instance:

Consider a hash table with the numbers 0 through 9 as the slots. A collision happens when two keys—let's say "apple" and "banana"—hash to the same index—let's say index 3. The hash table would either store both keys in a linked list at index 3 (chaining) or select an alternate slot (open addressing) for one of the keys to avoid overlap, depending on the collision resolution approach used.

Different kinds of collision-resolution methods:

In a hash table, collision resolution strategies are ways to deal with the case where several keys hash to the same index. The following are some typical methods for resolving collisions:

Chaining: When using chaining, every hash table slot keeps track of all the keys that hash to the same index in a linked list or other data structure. The new key-value pair is added to the linked list at the appropriate place in the event of a collision.

Benefits: Easy to set up and doesn't need any more room than what the hash table itself requires.

Cons: If linked lists get too big, there may be an increase in memory overhead and a possible decline in speed.

Open Addressing: In this method, collisions are avoided by locating a different slot inside the hash table.

Probing Techniques:

Linear Probing: Slots are examined one after the other until an empty slot or a slot holding a deleted item is discovered.

Quadratic Probing: A quadratic function is used to identify the next slot to probe, as opposed to inspecting each one one after the other.

Double Hashing: To improve dispersion and lessen clustering, this technique uses a second hash function to determine the next slot to explore.

Benefits: Due to proximity of reference, cache performance may be improved; also, memory overhead may be reduced in comparison to chaining.

Drawbacks: Can be more difficult to execute than chaining and necessitates careful selection of probing techniques to prevent clustering.

Robin Hood Hashing: This method seeks to lessen the possibility of chaining-related volatility in chain length (linked lists). If the new object is closer to the start of its chain than the old item when it collides with it, it may "steal" a position from the existing item.

Benefits: May improve average search times and contribute to the maintenance of a more balanced hash table.

Cons: It could take more calculation to decide whether or not the elements in the hash table need to be rearranged.

Selecting a Method for Resolving Collisions:

Hash Table Size: When memory is an issue or the hash table is relatively tiny, open addressing approaches may be more effective.

Expected Number of Collisions: Chaining is appropriate when collisions are anticipated to occur frequently since it can smoothly tolerate a higher number of collisions.

Performance Requirements: The technique to use may depend on the application and the trade-offs between memory utilization, retrieval speed, and insertion speed.

14.5 APPLICATIONS OF HASHING IN DATA STORAGE AND RETRIEVAL

Because hashing allows for quick access and efficiently manages big datasets, it is essential for many applications that require efficient data storage and retrieval. The following are some important uses for hashing:

Indexing in Databases: Hashing is a common technique used in databases to index records. Data may be quickly retrieved based on a key thanks to hash functions, which map keys to particular places in a hash table. This greatly enhances query performance when compared to linear search techniques.

Use of Hash Functions:

Mapping Keys to Addresses: In a hash table, keys—typically primary keys or unique identifiers—are mapped to particular addresses using hash functions. Direct access to the records connected to those keys is made possible by this mapping, eliminating the need to search through the whole dataset.

Effective Recovery:

Quick Access: Hashing offers a constant-time complexity of O(1) for average-case lookups in place of a linear search through the database, which can be laborious, particularly for large datasets. This effectiveness is attained by calculating a key's hash value and using it as an index to get the associated record quickly.

Database Systems Examples and Use Cases

Primary Key Lookup: To enable quick access to certain rows in tables, primary keys in relational databases are frequently hashed. For example, if a database table contains a primary key on a unique identifier column called "user_id," the location of each user's data in the database can be found by applying a hash function to the "user_id" values.

Hash-based Indexing Structures: Hash tables and hash indexes are examples of hash-based indexing structures that database management systems (DBMS) implement. Key-value pairs are stored in these structures; the keys are hashed, and the values are pointers to the real data records or locations. Use of hash indexes in NoSQL databases such as MongoDB for fast document retrieval based on unique identifiers is one example.

Enhancement of Performance: The efficiency of database operations like searching, inserting, and removing records is greatly enhanced by hash-based indexing. Databases can handle massive volumes of data more effectively, guaranteeing quick query response times, by lowering the average time complexity of data access operations to O(1).

Handling Collisions: Although hash algorithms make every effort to provide each key a unique address, collisions—two keys hashing to the same index—can nevertheless happen. In hash-based indexing, strategies like chaining or open addressing are used to control collisions and guarantee reliable data retrieval mechanisms.

Caching Management: In order to improve access speeds, data is temporarily cached via caching methods, where hashing is essential. It makes retrieval processes more efficient by storing cached objects in hash tables, which enables speedy lookup of recently used items.

Using Hash Tables to Put Caches in Place

LRU Cache (Least Recently Used):

Concept: When an LRU cache fills up, it starts with the least recently used items and removes them one at a time.

Implementation with Hash Tables:

Hash Map: The cache makes use of a hash map, in which values hold the content that has been cached (such as a web page or file contents) and keys represent the identifier of the cached item (such as a URL or file name).

Doubly Linked List: In addition, a doubly linked list makes sure that the items that have been accessed the most recently are at the head of the list by maintaining the order of access.

Instances and Advantages for Performance:

Online Browser Caching: To save online pages, pictures, and scripts, contemporary web browsers employ caches. The browser uses a hash table lookup to swiftly obtain content from its cache when a user returns to a page. This lowers bandwidth use and loading times.

Operating System Caches: To speed up disk access times, operating systems use caches for frequently accessed file system data. Cache data blocks are indexed using hash tables, facilitating quick lookup and retrieval.

Database Caching: Caches are used by database systems to hold the answers to frequently asked queries. Hash tables speed up query response times by effectively managing query IDs and stored results.

Performance Advantages

Quick Access: Hash tables offer insertion, deletion, and lookup operations with average-case constant-time O (1) complexity. This guarantees that, irrespective of the cache size, data access from the cache is efficient and consistent.

Space Efficiency: By compactly storing key-value pairs, hash tables maximize memory use, enabling caches to hold vast amounts of data with fast access times.

Scalability: Because hash tables can split data evenly among buckets and manage collisions well, they continue to operate at a steady pace even as the number of cached items increases.

Compiler Design: Symbol tables are essential data structures in compiler design that are used to store and organize information about symbols (such as variables, functions, and identifiers) that are encountered during compilation. Because hash tables may quickly look up values based on mapped symbols to attributes or information, they are essential for effectively implementing and using symbol tables. An outline of how hash tables are used in compiler symbol tables is provided below:

Hash tables' function in symbol tables

Effective Search and Insertion:

Hashing Mechanism: To find their storage location in the hash table, symbols are hashed using a hash function.

Quick Access: During compilation stages such as parsing, semantic analysis, and code generation, hash tables guarantee fast access to symbol information by offering average-case constant-time O (1) complexity for lookups and insertions.

Handling Declarations and Scope:

Scope Management: By classifying symbols according to their lexical scope (local, global, function-level), hash tables make scope management easier.

Collision Handling: Even with enormous symbol tables, collisions can be effectively handled using methods like chaining (using linked lists) or open addressing (probing), which ensure minimum influence on lookup performance.

Illustrations and Significance

Lexical Analysis: The compiler uses hash tables to identify tokens (such as keywords, identifiers, and literals) and stores them in the symbol table. This makes it possible for later compiler stages to quickly access and validate symbols.

Semantic Analysis: Hash tables help to ensure proper usage throughout the program by confirming symbol definitions and types. For example, comparing function prototypes and variable declarations to definitions kept in the symbol table.

Improving Compiler Performance: Compiler efficiency is increased when symbol tables are managed effectively using hash tables. Compilers are able to efficiently handle big codebases by decreasing lookup times and memory cost, which enhances compilation responsiveness and speed.

Relevance to the Design of Compilers

Error Detection: By helping to identify mistakes like undeclared variables or conflicting declarations, hash tables improve the compiler's capacity to give developers precise diagnostics and error messages.

Code Optimization: Symbol tables play a crucial role in code optimization stages, when compilers examine and modify code to increase efficiency. Hash tables guarantee that all symbol dependencies and references are appropriately taken into account during efficient code generation.
In compiler symbol tables, hashing is essential because it allows variable names or identifiers to be promptly resolved in terms of their characteristics or memory locations throughout the compilation and execution stages.

Data Deduplication: By taking advantage of hash functions' special characteristics, duplicate data saved across many systems can be found and removed. This is a summary of the use of hash functions in data deduplication, including practical applications and efficiency considerations:

Data Deduplication Using Hash Functions

Hashing chunks of data:

Hash Function Selection: The process of selecting an appropriate hash function involves taking into account several aspects, such as uniformity of distribution and collision resistance. Cryptographic hashes like MD5, SHA-1, and SHA-256 are popular options, as are non-cryptographic hashes like MurmurHash.

Chunking Data: Chunking data is the process of dividing large files or information into smaller bits or chunks.

Computing Hashes: A unique hash value, which is typically expressed as a fixed-length hexadecimal string, is produced by hashing each piece of data using the chosen hash algorithm.

Recognizing Duplicates

Comparing generated hash values allows for the identification of duplicate chunks. Hash values are identical when data chunks are identical.

Effective Lookup: To store and retrieve hash values of previously processed data chunks fast, hash tables or hash-based data structures (such as hash maps) are used.

Reducing Redundancy

Keeping Unique Data: The storage system only keeps unique chunks (chunks with unique hash values) permanently. We locate and remove duplicate pieces.

Storage Optimization: Deduplication saves a lot of storage space by storing only one copy of each unique chunk. This is especially useful in contexts where there is a lot of data redundancy.

Examples and Thoughts on Efficiency

Cloud Storage: Data deduplication employing hash functions in cloud storage systems optimizes storage use amongst several users and organizations sharing storage resources. By identifying duplicate files or chunks at the data center level, redundant data storage is reduced.

Backup Systems: To cut down on backup times and storage expenses, backup systems make use of hash-based deduplication techniques. Backup systems are able to handle massive amounts of data backups with efficiency by recognizing and preserving distinct data blocks.

Efficiency: The effectiveness of the deduplication algorithm and the collision resistance of the selected hash function determine the efficiency of data deduplication. Accurate detection of duplicate data is ensured by hash algorithms with low collision probabilities, and quick lookup and comparison operations are provided by effective hash table implementations.

Storage of Passwords: One essential cybersecurity practice is to store and secure passwords using hashing algorithms to prevent unauthorized access to user credentials. Here are some examples of secure hash algorithms and their uses, as well as a summary of how hashing is used to save passwords:

Techniques for Hashing Passwords

Password hashing:

Hash Function Selection: Secure hash functions are selected based on their cryptographic characteristics, such as resistance to collisions and preimages. A few examples include bcrypt, Argon2, SHA-3, SHA-256 (Secure Hash Algorithm 256-bit), and PBKDF2 (Password-Based Key Derivation Function 2).

Salting: Before hashing a password, a unique random value known as a salt is applied to protect against assaults such as rainbow table attacks. By using salting, two users with the same password will nonetheless have distinct hashed results.

Keeping Passwords Hashed:

Storage of Hashed Passwords: The database only contains the hashed password and, if applicable, the salt that goes with it, rather than the plaintext passwords.

Verification: The user-entered password is hashed with the salt that has been stored during authentication, and the resulting hash is compared to the hash that has been stored. If they line up, the password is regarded as legitimate.

Applications and Examples of Secure Hash Algorithms

SHA-256: This popular cryptographic hash algorithm generates a hash value of 256 bits, or 32 bytes. It belongs to the SHA-2 family and is regarded as safe for use in hashing passwords, among other purposes.

bcrypt: The Blowfish cipher is the foundation of the bcrypt password hashing algorithm. It is immune to brute-force attacks because it includes a cost element (work factor) that establishes the computational complexity of the hashing operation.

Argon2: The Password Hashing Competition (PHC) winner, Argon2 is built to fend off side-channel and GPU-accelerated attacks alike. By offering customizable options to modify memory consumption and processing duration, it makes brute-force attacks more challenging.

Uses

Web authentication: Hash functions are used by websites and web apps to safely store user passwords. Hashed passwords shield user accounts against intrusion even in the event that the database is compromised.

Database security: By hashing passwords prior to database storage, confidential data is shielded against security lapses and unwanted access by bad actors.

Compliance Requirements: In order to safeguard user data and guarantee privacy compliance, a number of cybersecurity standards and laws (such as GDPR and HIPAA) require the adoption of secure password storage methods like hashing.

Security and cryptography: In many applications, hash functions are essential for preserving data integrity and guaranteeing the legitimacy of data. This study examines the role that hashing methods, such as the Secure Hash Algorithm (SHA), have in maintaining data integrity:

Verification of Data Integrity

Hash Functions as Digital Fingerprints: From any size of input data, hash functions like SHA-256 produce fixed-size hash values, or digests. These hash values function as distinct checksums or digital fingerprints of the original material.

Identifying Data Alterations: Hashing values can be calculated at the sender and recipient ends when data is sent over networks or kept in databases. It confirms that no changes have been made to the data during transmission if the hash value received and the hash value computed at the sender's end match.

Use Cases: Applications where data integrity is crucial, including the following, heavily utilize hashing.

File Integrity Checking: By comparing hash values, hashing verifies that files are received exactly as sent before sending them over the internet.

Digital Signatures: Hashing functions produce a message digest, which is then signed using a private key in digital signature technology. By using the public key of the signer, the recipient can confirm the message's integrity.

Password Storage: **Hashing** secures passwords before storing them in databases by transforming them into hash values. The password entered by the user is hashed and compared with the hash value that has been stored during authentication.

Hash Function Examples

The Secure Hash Algorithm (SHA) family of hash functions, which includes SHA-1, SHA-256, SHA-384, and SHA-512, is extensively utilized. They are intended to be collision-resistant, which means it is computationally impossible to find two separate

inputs that result in the same hash value. They produce hash values of specified lengths (256 bits for SHA-256, for example).
Message Digest Algorithm 5, or MD5, was once extensively used for digital signatures and integrity verification even though it was less secure than SHA-256. A 128-bit hash value is generated.

Maintaining Data Integrity
Checksums and Validation: **Hash** values are used as checksums to verify the integrity of the data. It is simple to identify changes since even a small change in the input data produces a drastically different hash value.

Cryptographic Strength: SHA-256 and other contemporary hash functions are resistant to a variety of assaults, including collision attacks, in which two different inputs result in the same hash value.

14.6 CONCLUSION

In summary, research on hashing and collision resolution methods shows how important a role they play in contemporary computing and data management. Rapid data retrieval and storage procedures are made possible by hashing, which offers an effective way to map data items to unique identifiers. Potential conflicts inside hash tables are efficiently controlled by means of collision resolution techniques like chaining and open addressing, guaranteeing the integrity and performance of data structures even in the face of heavy loads.

Furthermore, hashing finds use in a wide range of fields, including security protocols, compiler design, database indexing, and caching techniques. Every application makes use of hashing algorithms' speed and dependability to improve system performance and optimize data access. Recognizing these uses emphasizes how crucial it is to understand hashing and related methods in computer science and other fields.

Robust data storage and retrieval solutions are still essential as long as technology keeps developing. Hashing is the foundation of many contemporary data structures and algorithms, especially when combined with efficient collision resolution techniques. Professionals and students alike can help solve increasingly difficult computational problems and create more effective systems by grasping these ideas.

14.7 QUESTIONS AND ANSWERS

1. How does hashing function in data storage and what does it entail?
Answer: The process of mapping arbitrary-sized data to fixed-size values—usually integers—known as hash codes is called hashing. To create the hash code, which is used as an index to quickly store or retrieve data in a hash table, the data must first be subjected to a hash function.

2. What are collision resolution methods, and what makes hashing require them?
Answer: In response to a question like this, collision resolution techniques are ways to deal with scenarios in which two or more different data pieces produce the same hash code. Among the methods are open addressing (identifying different locations within the hash table) and chaining (using linked lists or other structures at the same hash index). Because they guarantee that all data can be saved and retrieved correctly, they are essential to preserving the efficiency and integrity of hash tables.

3. Chaining and open addressing for hash table collision resolution are compared and contrasted.

Answer: The process of chaining entails building linked lists or other structures to hold numerous data components with the same hash code at each index of the hash table. Although it is easy to implement, there may be more memory overhead. Open addressing, on the other hand, looks for different places to put colliding components directly within the hash table. It is more memory-efficient but can cause clustering and calls for cautious probing techniques like linear or quadratic probing.

4. What are the primary uses of hashing for storing and retrieving data?

Answer: The answer is that hashing is widely employed in symbol tables in compilers to effectively handle identifiers, caching systems to store frequently accessed data, and database indexing for speedy data retrieval. It is also essential to cryptographic algorithms that verify data integrity and secure passwords.

5. Describe the idea of the quality of a hash function. How should a hash function be designed?

Answer: To reduce collisions, a good hash function should distribute hash codes evenly throughout the hash table. It should be resistant to hash collisions from identical inputs (avalanche effect), computationally efficient, and deterministic (same input creates same output). Division, multiplication, and cryptographic hash functions like SHA (Secure Hash Algorithm) are a few examples.

14.8 REFERENCES

- Bjarne Stroustrup, "The C++ Programming Language"
- Herb Sutter, "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions"
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, "C++ Primer"
- Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Nicolai M. Josuttis, "C++ Standard Library: A Tutorial and Reference"

UNIT – 15: ADVANCED DATA STRUCTURES

Structure

15.0 INTRODUCTION

In the ever-evolving field of computer science, advanced data structures play a pivotal role in optimizing performance and solving complex problems. This unit delves into several sophisticated data structures, each with unique characteristics and applications. We will explore Scapegoat Trees, a self-balancing binary search tree that offers an efficient alternative to other balanced trees. Additionally, we will cover Tries and their variants, including Binary Tries, X-Fast Tries, and Y-Fast Tries, which are crucial for efficient information retrieval and prefix matching.

Understanding these advanced data structures is essential for enhancing the efficiency of algorithms, particularly in scenarios that require fast data access and manipulation. Scapegoat Trees provide a robust method for maintaining balanced trees without the need for frequent rotations, while Tries and their derivatives excel in tasks such as dictionary implementation, autocomplete features, and IP routing.

By the end of this unit, you will gain a comprehensive understanding of these advanced data structures, their implementation, and practical applications. This knowledge will enable you to make informed decisions about selecting the appropriate data structure for specific computational problems, ultimately improving the performance and scalability of your software solutions.

15.1 OBJECTIVES

After completing this unit, you will be able to understand,
Understand the concept and properties of Scapegoat Trees.
Explore the structure and properties of Tries.
Understand the structure and properties specific to Binary Tries.
Explore the structure and properties unique to X-Fast Tries.
Discuss the theoretical and practical applications of Y-Fast Tries in computational problems.

15.2 SCAPEGOAT TREES

Scapegoat trees are a type of self-balancing binary search tree designed to maintain an efficient average-case performance with logarithmic depth. Unlike other balanced trees such as AVL or Red-Black trees, scapegoat trees do not perform rebalancing after every insertion or deletion. Instead, they monitor the tree's balance and only rebalance when the tree becomes significantly unbalanced. The key concept is to identify a scapegoat node, whose subtree is then rebuilt to restore balance. This approach simplifies the implementation while ensuring that the tree remains reasonably balanced over time, providing efficient average-case time complexity for insertion, deletion, and search operations.

The insertion process in a scapegoat tree involves standard BST insertion followed by a check for imbalance, which may trigger the identification and rebalancing of the scapegoat subtree if necessary. Deletion similarly follows standard BST procedures but includes periodic rebalancing to maintain overall tree balance. The tree maintains a balance factor, typically a constant between 0 and 1, which helps determine when rebalancing is needed. This method ensures that most operations are performed in O (log n) time on average, with occasional O (n) operations when rebalancing is required, making scapegoat trees suitable for applications like database indexing and memory management where dynamic data and efficient performance are crucial.

Characteristics of scapegoat trees

Scapegoat trees possess several distinctive characteristics that differentiate them from other self-balancing binary search trees:

Amortized Rebalancing: Scapegoat trees do not rebalance after every insertion or deletion. Instead, they perform rebalancing operations in an amortized manner, meaning that the cost of rebalancing is spread out over multiple operations. This helps maintain efficient performance over a sequence of operations without the overhead of constant rebalancing.

Balance Factor: A scapegoat tree maintains a balance factor, denoted by α, which is typically a constant between 0 and 1 (commonly set to 2/3). This balance factor is used to determine whether a node in the tree is unbalanced. If the size of a subtree exceeds a certain threshold relative to α, the subtree is identified for rebalancing.

Scapegoat Identification: When the tree detects an imbalance, it identifies a scapegoat node. A scapegoat node is an ancestor of the recently inserted or deleted node whose subtree size violates the balance factor. The subtree rooted at the scapegoat node is then rebuilt to restore balance. This approach ensures that the tree does not become excessively unbalanced.

Efficient Average-Case Performance: Scapegoat trees are designed to provide efficient average-case time complexity for insertion, deletion, and search operations. While individual rebalancing operations can be costly, occurring in O (n) time, the amortized cost remains O (log n) over a series of operations. This ensures that the tree performs well in practical applications.

Simple Implementation: Compared to other balanced trees like AVL or Red-Black trees, scapegoat trees have a simpler implementation. They avoid the need for complex rotations and color properties, making them easier to implement and understand while still maintaining balanced tree properties.

Operations on Scapegoat Trees

Scapegoat trees involve several operations such as insertion, deletion, and searching. Below are the key operations along with their algorithms in C++.

1. Insertion

The insertion operation involves adding a new node to the tree and checking if the tree remains balanced. If the balance condition is violated, a scapegoat node is identified and the subtree is rebuilt.

Algorithm:

Insert the new node as in a standard binary search tree.

Check if the tree is unbalanced.

If unbalanced, identify the scapegoat node.

Rebuild the subtree rooted at the scapegoat node.

C++ Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
template<typename T>
class ScapegoatTree {
public:
    struct Node {
        T key;
        Node *left, *right;
        Node(T k) : key(k), left(nullptr), right(nullptr) {}
    };
    ScapegoatTree(double alpha) : root(nullptr), alpha(alpha),
maxSize(0) {}
    void insert(T key) {
        root = insert(root, key);
        if (size(root) > maxSize) maxSize = size(root);
    }
private:
    Node* root;
    double alpha;
    int maxSize;
    Node* insert(Node* node, T key) {
        if (!node) return new Node(key);
        if (key < node->key) node->left = insert(node->left, key);
        else node->right = insert(node->right, key);
        if (!isBalanced(node)) {
            node = rebuild(node);
        }
        return node;
    }
    bool isBalanced(Node* node) {
        return size(node) <= alpha * size(parent(node));
    }
    Node* parent(Node* node) {
        // Function to find the parent of a node.
        // This function's implementation depends on the context and
additional bookkeeping.
        return nullptr;
    }
    Node* rebuild(Node* node) {
        std::vector<Node*> nodes;
        flatten(node, nodes);
        return buildTree(nodes, 0, nodes.size());
    }
    void flatten(Node* node, std::vector<Node*>& nodes) {
        if (!node) return;
        flatten(node->left, nodes);
        nodes.push_back(node);
        flatten(node->right, nodes);
    }
    Node* buildTree(std::vector<Node*>& nodes, int start, int end)
{
        if (start >= end) return nullptr;
        int mid = (start + end) / 2;
        Node* node = nodes[mid];
        node->left = buildTree(nodes, start, mid);
        node->right = buildTree(nodes, mid + 1, end);
        return node;
    }

    int size(Node* node) {
        if (!node) return 0;
        return 1 + size(node->left) + size(node->right);
    }
};
```

2. Deletion

The deletion operation involves removing a node and checking the balance of the tree. If the tree becomes unbalanced, it is restructured to maintain balance.

Algorithm:

```
Delete the node as in a standard binary search tree.
Check if the tree is unbalanced.
If unbalanced, rebuild the entire tree if the current size is less than half of the maximum size.
```

C++ Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
template<typename T>
class ScapegoatTree {
public:
    struct Node {
        T key;
        Node *left, *right;
        Node(T k) : key(k), left(nullptr), right(nullptr) {}
    };
    ScapegoatTree(double alpha) : root(nullptr), alpha(alpha),
maxSize(0) {}
    void insert(T key) {
        root = insert(root, key);
        if (size(root) > maxSize) maxSize = size(root);
    }
    void remove(T key) {
        root = remove(root, key);
        if (size(root) < maxSize / 2) {
            root = rebuild(root);
            maxSize = size(root);
        }
    }
private:
    Node* root;
    double alpha;
    int maxSize;
    Node* insert(Node* node, T key) {
        if (!node) return new Node(key);
        if (key < node->key) node->left = insert(node->left, key);
        else node->right = insert(node->right, key);

        if (!isBalanced(node)) {
            node = rebuild(node);
        }
        return node;
    }
    Node* remove(Node* node, T key) {
        if (!node) return nullptr;
        if (key < node->key) node->left = remove(node->left, key);
        else if (key > node->key) node->right = remove(node->right,
key);
```

```
    else {
        if (!node->left) {
            Node* rightChild = node->right;
            delete node;
            return rightChild;
        } else if (!node->right) {
            Node* leftChild = node->left;
            delete node;
            return leftChild;
        } else {
            Node* minNode = findMin(node->right);
            node->key = minNode->key;
            node->right = remove(node->right, minNode->key);
        }
    }
    if (!isBalanced(node)) {
        node = rebuild(node);
    }
    return node;
}
Node* findMin(Node* node) {
    while (node && node->left) {
        node = node->left;
    }
    return node;
}
bool isBalanced(Node* node) {
    return size(node) <= alpha * size(parent(node));
}
Node* parent(Node* node) {
    // Function to find the parent of a node.
```

```
        // This function's implementation depends on the context and
additional bookkeeping.
        return nullptr;
    }
    Node* rebuild(Node* node) {
        std::vector<Node*> nodes;
        flatten(node, nodes);
        return buildTree(nodes, 0, nodes.size());
    }
    void flatten(Node* node, std::vector<Node*>& nodes) {
        if (!node) return;
        flatten(node->left, nodes);
        nodes.push_back(node);
        flatten(node->right, nodes);
    }
    Node* buildTree(std::vector<Node*>& nodes, int start, int end)
{
        if (start >= end) return nullptr;
        int mid = (start + end) / 2;
        Node* node = nodes[mid];
        node->left = buildTree(nodes, start, mid);
        node->right = buildTree(nodes, mid + 1, end);
        return node;
    }
    int size(Node* node) {
        if (!node) return 0;
        return 1 + size(node->left) + size(node->right);
    }
};
```

3. Searching

The search operation is similar to that in a standard binary search tree, where we traverse the tree based on the comparison of the search key with the node keys.

C++ Implementation:

```
template<typename T>
class ScapegoatTree {
public:
    struct Node {
        T key;
        Node *left, *right;
        Node(T k) : key(k), left(nullptr), right(nullptr) {}
    };
    ScapegoatTree(double alpha) : root(nullptr), alpha(alpha),
maxSize(0) {}
    void insert(T key) {
        root = insert(root, key);
        if (size(root) > maxSize) maxSize = size(root);
    }
    bool search(T key) {
        return search(root, key);
    }
private:
    Node* root;
    double alpha;
    int maxSize;
    Node* insert(Node* node, T key) {
        if (!node) return new Node(key);
        if (key < node->key) node->left = insert(node->left, key);
        else node->right = insert(node->right, key);
        if (!isBalanced(node)) {
            node = rebuild(node);
        }
        return node;
    }
    bool search(Node* node, T key) {
        if (!node) return false;
        if (node->key == key) return true;
        if (key < node->key) return search(node->left, key);
        return search(node->right, key);
    }
    bool isBalanced(Node* node) {
        return size(node) <= alpha * size(parent(node));
    }
    Node* parent(Node* node) {
        // Function to find the parent of a node.
        // This function's implementation depends on the context and
additional bookkeeping.
        return nullptr;
    }
    Node* rebuild(Node* node) {
        std::vector<Node*> nodes;
        flatten(node, nodes);
        return buildTree(nodes, 0, nodes.size());
    }
    void flatten(Node* node, std::vector<Node*>& nodes) {
        if (!node) return;
        flatten(node->left, nodes);
        nodes.push_back(node);
        flatten(node->right, nodes);
    }
    Node* buildTree(std::vector<Node*>& nodes, int start, int end)
{
        if (start >= end) return nullptr;
        int mid = (start + end) / 2;
        Node* node = nodes[mid];
        node->left = buildTree(nodes, start, mid);
        node->right = buildTree(nodes, mid + 1, end);
        return node;
    }
    int size(Node* node) {
        if (!node) return 0;
        return 1 + size(node->left) + size(node->right);
    }
};
```

Balancing and Restructuring

Balancing and restructuring are essential aspects of maintaining the efficiency of scapegoat trees. These processes ensure that the tree remains balanced, providing efficient access times for insertion, deletion, and search operations. Below are the details about the conditions for imbalance and techniques for rebalancing.

Conditions for Imbalance

A scapegoat tree becomes imbalanced when a node's subtree size exceeds a certain threshold compared to its parent. Specifically, the imbalance condition is determined by a parameter α\alphaα, where $0 < \alpha < 1$. This parameter is used to maintain a balance between the left and right subtrees of any node. The imbalance condition can be defined as:

Imbalance Condition: A node v in a scapegoat tree is considered unbalanced if the size of any of its child subtrees exceeds α\alphaα times the size of v's subtree.

Techniques for Rebalancing

When an imbalance is detected, the subtree rooted at the scapegoat node is rebuilt to restore balance. The techniques for rebalancing involve the following steps:

Identify the Scapegoat Node:

Traverse up from the newly inserted or deleted node to find the first ancestor node that violates the balance condition.

Rebuild the Subtree:

Flatten the subtree rooted at the scapegoat node into a sorted array.

Rebuild the balanced subtree from the sorted array.

Algorithm for Rebalancing:

Identify the Scapegoat Node:

Start from the node where the imbalance is detected.

Move up the tree to find the first node that violates the balance condition.

Flatten the Subtree:

Perform an in-order traversal of the subtree rooted at the scapegoat node to create a sorted array of nodes.

Rebuild the Subtree:

Use the sorted array to construct a balanced subtree.

Recursively split the array to ensure the tree remains balanced.

C++ Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
template<typename T>
```

```
class ScapegoatTree {
public:
    struct Node {
        T key;
        Node *left, *right;
        Node(T k) : key(k), left(nullptr), right(nullptr) {}
    };
    ScapegoatTree(double alpha) : root(nullptr), alpha(alpha),
maxSize(0) {}
    void insert(T key) {
        root = insert(root, key);
        if (size(root) > maxSize) maxSize = size(root);
    }
    void remove(T key) {
        root = remove(root, key);
        if (size(root) < maxSize / 2) {
            root = rebuild(root);
            maxSize = size(root);
        }
    }
private:
    Node* root;
    double alpha;
    int maxSize;
    Node* insert(Node* node, T key) {
        if (!node) return new Node(key);
        if (key < node->key) node->left = insert(node->left, key);
        else node->right = insert(node->right, key);
        if (!isBalanced(node)) {
            node = rebuild(node);
        }
        return node;
```

```
  }
Node* remove(Node* node, T key) {
if (!node) return nullptr;
 if (key < node->key) node->left = remove(node->left, key);
else if (key > node->key) node->right = remove(node->right, key);
else {
if (!node->left) {
Node* rightChild = node->right;
delete node;
return rightChild;
} else if (!node->right) {
Node* leftChild = node->left;
delete node;
return leftChild;
} else {
Node* minNode = findMin(node->right);
node->key = minNode->key;
node->right = remove(node->right, minNode->key);
}
}
if (!isBalanced(node)) {
node = rebuild(node);
}
return node;
}
Node* findMin(Node* node) {
while (node && node->left) {
node = node->left;
}
return node;
}
bool isBalanced(Node* node) {
```

```
return size(node) <= alpha * size(parent(node));
    }
    Node* parent(Node* node) {
        // Function to find the parent of a node.
        // This function's implementation depends on the context and
additional bookkeeping.
        return nullptr;
    }
    Node* rebuild(Node* node) {
        std::vector<Node*> nodes;
        flatten(node, nodes);
        return buildTree(nodes, 0, nodes.size());
    }
    void flatten(Node* node, std::vector<Node*>& nodes) {
        if (!node) return;
        flatten(node->left, nodes);
        nodes.push_back(node);
        flatten(node->right, nodes);
    }
    Node* buildTree(std::vector<Node*>& nodes, int start, int end)
{
        if (start >= end) return nullptr;
        int mid = (start + end) / 2;
        Node* node = nodes[mid];
        node->left = buildTree(nodes, start, mid);
        node->right = buildTree(nodes, mid + 1, end);
        return node;
    }
    int size(Node* node) {
        if (!node) return 0;
        return 1 + size(node->left) + size(node->right);
    }
```

```
};
```

Time Complexity Analysis

Here's a breakdown of the time complexity analysis for the operations in a Scapegoat Tree:

Insertion (insert function):

Average Case: O(log n)

Worst Case (Rebuilding): O(n log n) due to the need to rebuild the tree when a rebalance condition is violated.

Deletion (remove function):

Average Case: O(log n)

Worst Case (Rebuilding): O(n log n) due to potential tree rebuilds.

Search (assuming balanced tree):

Average Case: O(log n)

Worst Case: O(log n)

Rebuilding (rebuild function):

Time Complexity: O(n)

Size Calculation (size function):

Time Complexity: O(n)

Explanation:

Insertion and Deletion: In the average case, insertion and deletion operations perform in O(log n) time due to the binary search tree properties of the Scapegoat Tree. However, when the tree needs rebalancing (when the size condition is violated), rebuilding the tree takes O(n log n) time as it involves flattening the subtree and reconstructing it. This worst-case scenario occurs when the tree becomes highly unbalanced.

Search: Searching in a balanced Scapegoat Tree also operates in O(log n) time in both average and worst cases, similar to standard binary search trees.

Rebuilding: The rebuild function is triggered when an imbalance is detected and requires flattening the subtree and reconstructing it in sorted order, resulting in a time complexity of O(n), where n is the number of nodes in the subtree.

Size Calculation: The size function computes the number of nodes in a subtree, requiring traversal of all nodes, leading to a time complexity of O(n).

Applications

Scapegoat Trees find applications in scenarios where a balance between efficient search, insertion, and deletion operations is crucial, and where the tree structure needs to adapt dynamically to changing data sizes. Some common applications include:

Database Indexing: Scapegoat Trees are used in database systems for indexing large datasets efficiently. They provide logarithmic time complexity for search operations, making them suitable for fast retrieval of indexed data.

Dynamic Data Structures: Due to their self-adjusting nature, Scapegoat Trees are employed in applications where the dataset size varies over time. This includes real-time systems, web servers, and applications handling streams of data.

File Systems: They are used in file systems for managing directory structures efficiently. Scapegoat Trees allow quick lookup and modification of file paths, ensuring efficient file management operations.

Networking: In networking applications, Scapegoat Trees can be used for routing tables, where fast lookup and updates are essential for handling network traffic and routing decisions.

Compiler Design: Scapegoat Trees are utilized in compilers for symbol table management. They facilitate quick insertion and retrieval of identifiers and variables during the compilation process.

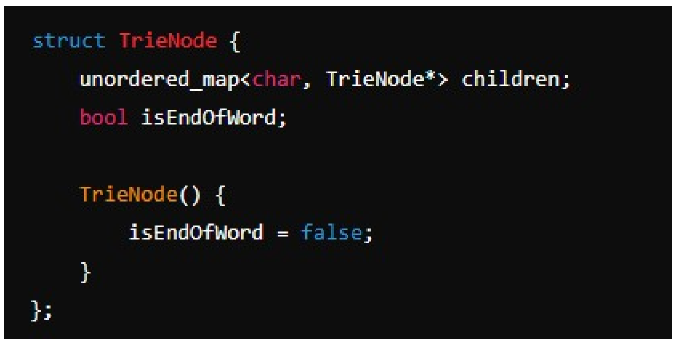
Key-Value Stores: In systems like distributed databases and key-value stores, Scapegoat Trees offer an efficient data structure for storing and retrieving key-value pairs with logarithmic time complexity.

15.3 TRIES

Tries, also known as prefix trees or digital trees, are tree-based data structures used for efficient storage and retrieval of strings or keys. Each node in a trie represents a character, and paths from the root to the leaf nodes correspond to sequences of characters (strings). This structure allows for rapid prefix-based operations such as search, insert, and delete. Tries are particularly useful in scenarios where fast autocomplete functionalities or efficient dictionary lookups are required.

Operations on tries involve traversing the tree from the root based on the characters of the key being processed. Insertion involves creating new nodes as necessary to build the path for a new key. Searching in a trie involves following the path corresponding to the characters of the search key; if the path exists to a leaf node, the key is present. Deletion can be more complex as it might involve pruning nodes that are no longer part of any stored key's path. Tries are space-efficient when keys share common prefixes but can consume more memory compared to other data structures when dealing with large alphabets or sparse data. They find applications in areas such as autocomplete systems, spell-checking, IP routing tables, and data compression algorithms like Huffman coding.

Trie Node Structure



```
struct TrieNode {
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
    }
};
```

Key properties of tries include:

Prefix-based Storage: Tries excel in storing keys with common prefixes efficiently. Each node along a path in the trie corresponds to a character in the key, allowing for rapid prefix-based operations.

Search Complexity: Searching in a trie is efficient, typically in O(m) time complexity, where m is the length of the key being searched. This efficiency arises because the search operation involves following a path from the root to a leaf or until no further nodes can be traversed.

Insertion and Deletion: Inserting a key into a trie involves creating nodes as necessary to form the path for the key. Deletion can be more complex, potentially requiring the removal of nodes if they no longer correspond to any other keys' prefixes.

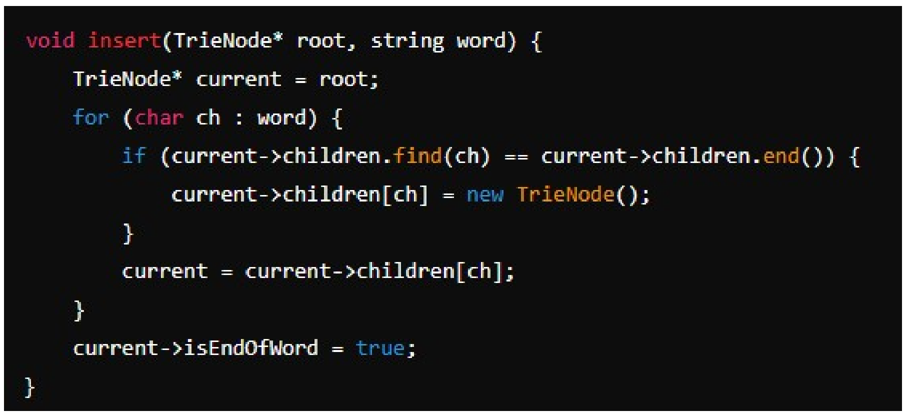
Space Efficiency: Tries can be memory-intensive, especially for large alphabets or sparse data, due to the potentially large number of nodes. However, they are efficient in scenarios where keys share common prefixes, thereby reducing redundant storage of prefixes.

Applications: Tries find applications in various domains such as autocomplete systems, spell-checking, IP routing tables, and database indexing. They are particularly useful in scenarios requiring fast prefix matching or predictive text functionality.

Operations on Tries

Here's a brief overview of operations on tries:

Insertion: Inserting a new key into a trie involves traversing the trie based on the characters of the key. Starting from the root, each character in the key determines the path through the trie. If a path corresponding to the key doesn't exist, new nodes are created. At the end of the key, a flag or marker is typically set to indicate that the key exists in the trie.

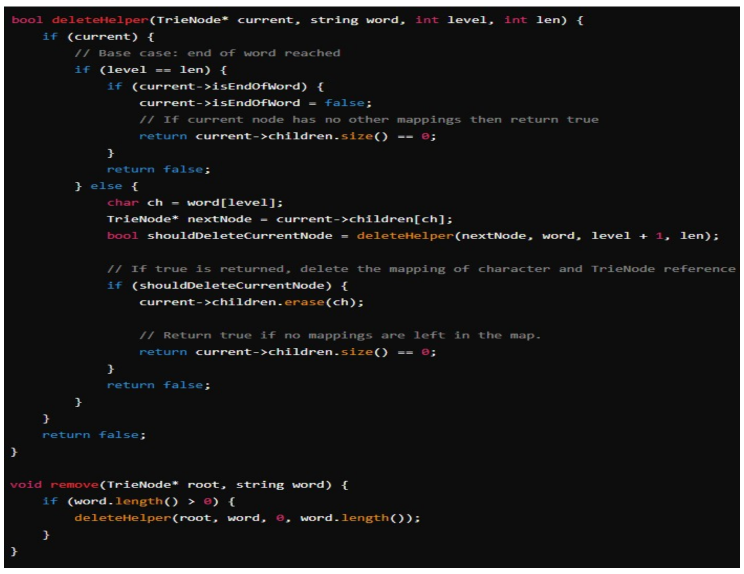


```
void insert(TrieNode* root, string word) {
    TrieNode* current = root;
    for (char ch : word) {
        if (current->children.find(ch) == current->children.end()) {
            current->children[ch] = new TrieNode();
        }
        current = current->children[ch];
    }
    current->isEndOfWord = true;
}
```

Insertion: The insert function iterates through each character of the word. If the character doesn't exist in the current node's children map, it creates a new node. At the end of the word, it marks the isEndOfWord flag as true.

Deletion: Deleting a key from a trie requires traversing the trie similarly to insertion, but instead of adding nodes, nodes corresponding to the key may be removed. This operation can be more complex than insertion because nodes might need to be pruned to maintain trie properties. Removal typically involves checking if the key exists, then removing nodes upwards if they are no longer needed.

Deletion: The remove function calls a helper function deleteHelper, which recursively traverses the Trie until the end of the word. If the word is found, it marks isEndOfWord as false. If a node has no children after deletion, its mapping in the parent's children map is erased recursively.



```
bool deleteHelper(TrieNode* current, string word, int level, int len) {
    if (current) {
        // Base case: end of word reached
        if (level == len) {
            if (current->isEndOfWord) {
                current->isEndOfWord = false;
                // If current node has no other mappings then return true
                return current->children.size() == 0;
            }
            return false;
        } else {
            char ch = word[level];
            TrieNode* nextNode = current->children[ch];
            bool shouldDeleteCurrentNode = deleteHelper(nextNode, word, level + 1, len);

            // If true is returned, delete the mapping of character and TrieNode reference
            if (shouldDeleteCurrentNode) {
                current->children.erase(ch);

                // Return true if no mappings are left in the map.
                return current->children.size() == 0;
            }
            return false;
        }
    }
    return false;
}

void remove(TrieNode* root, string word) {
    if (word.length() > 0) {
        deleteHelper(root, word, 0, word.length());
    }
}
```

Searching: Searching in a trie involves traversing nodes based on the characters of the key. Starting from the root, each character determines the next node to visit. If the key exists in the trie, the search will successfully find the key by following the path corresponding to its characters. If any character path is missing during traversal, the search concludes that the key is not present in the trie.

Searching: The search function traverses the Trie starting from the root. It checks if each character of the word exists in the children map of the current node. If it reaches the end of the word and isEndOfWord is true, it returns true; otherwise, false.

```
bool search(TrieNode* root, string word) {
    TrieNode* current = root;
    for (char ch : word) {
        if (current->children.find(ch) == current->children.end()) {
            return false;
        }
        current = current->children[ch];
    }
    return current != nullptr && current->isEndOfWord;
}
```

Types of Tries

Standard Tries (Prefix Trees): Standard Tries, also known as Prefix Trees, are fundamental data structures where each node represents a single character of the stored keys. They efficiently support operations like insertion, deletion, and searching based on prefixes. Standard Tries are versatile and used in scenarios where efficient prefix-based lookups are required, such as autocomplete systems and dictionary implementations.

Compressed Tries: Compressed Tries, also called Radix Trees or Compact Prefix Trees, optimize space by compressing nodes that have a single child into one. This compression reduces memory usage compared to Standard Tries while maintaining fast prefix search capabilities. Compressed Tries are useful in applications where storage efficiency is critical, such as in memory-constrained environments or when storing large sets of keys with common prefixes.

Suffix Tries: Suffix Tries store all suffixes of a given text string. Each node represents a suffix rather than a prefix. They are particularly useful in string processing tasks like pattern matching, substring search, and text indexing. Suffix Tries facilitate fast searches for specific patterns within a text corpus and are integral to algorithms like the construction of suffix arrays and suffix trees.

15.4 BINARY TRIES

Binary Tries, also known as Radix Tries or Patricia Tries, are specialized data structures that store keys in a binary format rather than a character-by-character manner like Standard Tries. In Binary Tries, each node represents a bit in the binary representation of the key. This approach allows for efficient storage and retrieval of keys that are typically represented in binary form, such as IP addresses or binary-encoded data.

The nodes in Binary Tries can have up to two children, representing '0' and '1' branches corresponding to the binary digits. This binary representation ensures that searches, insertions, and deletions in Binary Tries operate efficiently, often in logarithmic time relative to the length of the keys. Binary Tries are particularly useful in applications where keys are binary data or where efficient bitwise operations are required, such as in network routing tables or database indexing systems that handle binary-coded data formats. Their structure lends itself well to scenarios where memory efficiency and quick lookup times are crucial.

Structure and properties of binary tries

Binary Tries, also known as Radix Tries or Patricia Tries, are structured similarly to standard tries but are optimized for storing keys represented in binary format. Here's an outline of their structure and properties:

Node Structure: Each node in a Binary Trie contains:
Children Pointers: Typically two pointers, representing '0' and '1', corresponding to the binary digits.

Key: Optionally, a node may store part or all of the key associated with the node.

Properties:
Binary Representation: Keys are stored in a compressed binary format, where each node represents a bit in the key.
Efficient Storage: Compared to standard tries, binary tries can save space by combining nodes along paths that share prefixes.
Prefix Matching: Like standard tries, binary tries support prefix matching efficiently, making them suitable for applications requiring fast lookups based on binary data.

Operations: Binary tries support operations such as insertion, deletion, and searching, typically in logarithmic time relative to the length of the keys.

Operations on Binary Tries

Binary Tries, also known as Patricia Tries, are a type of trie data structure optimized for storing keys that can be represented as sequences of bits. Here’s how insertion, deletion, and searching operations are typically implemented in Binary Tries:

Insertion

Algorithm: Insert (Binary Trie T, Key k)

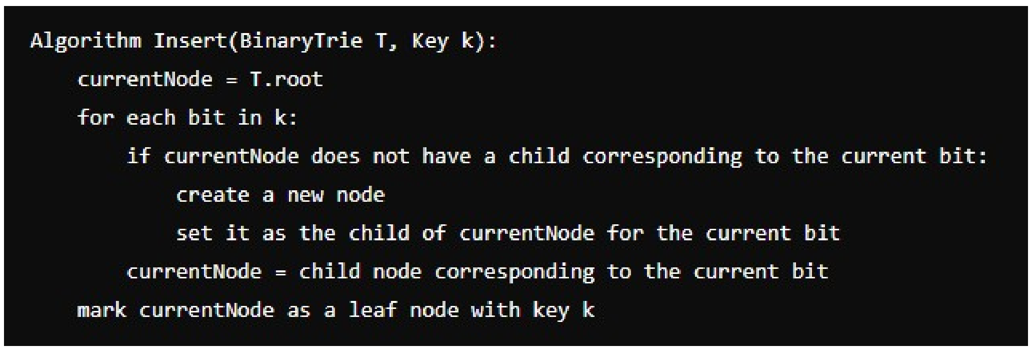
Initialization: Start from the root of the trie.

Traversal: For each bit in the key k:

If the current node does not have a child corresponding to the current bit of k, create a new node and attach it as a child.

Move to the child node corresponding to the current bit.

Leaf Node Marking: Once all bits of k are processed, mark the current node as a leaf node and store k in it.



```
Algorithm Insert(BinaryTrie T, Key k):
    currentNode = T.root
    for each bit in k:
        if currentNode does not have a child corresponding to the current bit:
            create a new node
            set it as the child of currentNode for the current bit
        currentNode = child node corresponding to the current bit
    mark currentNode as a leaf node with key k
```

Insertion: Inserts a key k into the trie by following the bits of k and creating nodes as necessary. At the end of the insertion, a leaf node is marked with k.

Deletion

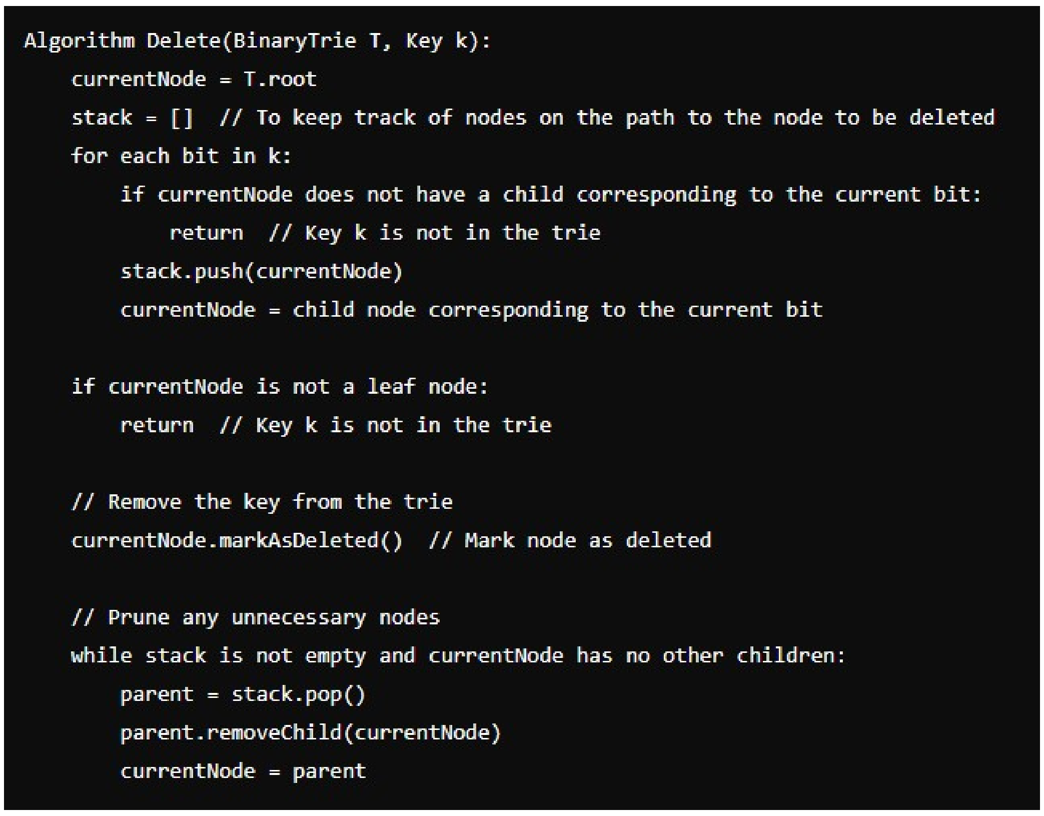
Algorithm: Delete (Binary Trie T, Key k)

Traversal: Start from the root and traverse the trie following the bits of k.

Marking for Deletion: If k exists in the trie (i.e., you reach a leaf node marked with k):

Mark the leaf node as deleted or remove the key from the leaf node.

Pruning: Check if any parent node of the leaf node can be pruned (i.e., it has no other children). If so, continue pruning up to the root as long as it does not violate trie properties.



```
Algorithm Delete(BinaryTrie T, Key k):
    currentNode = T.root
    stack = []  // To keep track of nodes on the path to the node to be deleted
    for each bit in k:
        if currentNode does not have a child corresponding to the current bit:
            return  // Key k is not in the trie
        stack.push(currentNode)
        currentNode = child node corresponding to the current bit

    if currentNode is not a leaf node:
        return  // Key k is not in the trie

    // Remove the key from the trie
    currentNode.markAsDeleted()  // Mark node as deleted

    // Prune any unnecessary nodes
    while stack is not empty and currentNode has no other children:
        parent = stack.pop()
        parent.removeChild(currentNode)
        currentNode = parent
```

Deletion: Deletes a key k from the trie by finding the leaf node marked with k and removing it. It then prunes unnecessary nodes from the trie to maintain efficiency.

Searching

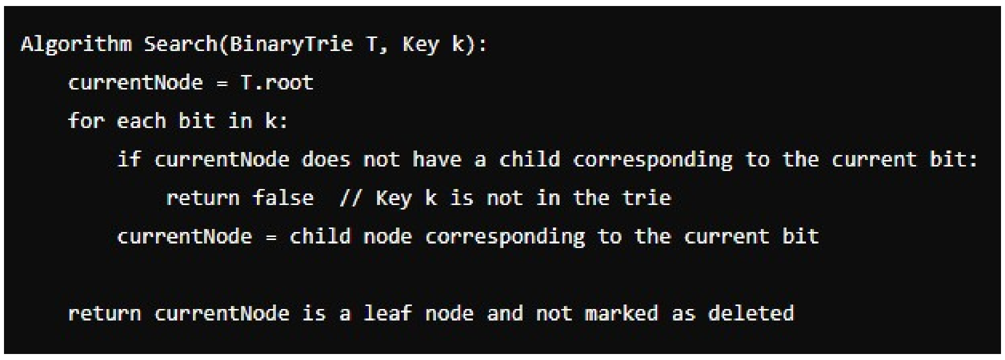
Algorithm: Search (Binary Trie T, Key k)

Traversal: Start from the root and follow the bits of k.

Existence Check: If all bits of k are found in the trie:

Check if the node corresponding to the last bit of k is a leaf node and not marked as deleted.

Return true if k is found; otherwise, return false.



```
Algorithm Search(BinaryTrie T, Key k):
    currentNode = T.root
    for each bit in k:
        if currentNode does not have a child corresponding to the current bit:
            return false  // Key k is not in the trie
        currentNode = child node corresponding to the current bit

    return currentNode is a leaf node and not marked as deleted
```

Searching: Searches for a key k in the trie by following the bits of k. It checks if k exists by ensuring that the path ends at a leaf node marked with k and not deleted.

15.5 X-FAST TRIES

X-Fast Tries are a type of data structure that extends the concept of binary tries (or Patricia Tries) to efficiently support dynamic sets of keys drawn from a universe of size U. They are designed to provide fast operations for searching, insertion, deletion, and predecessor/successor queries, all in O (log log U) time complexity.

Structure and Properties

X-Fast Tries are structured as a hierarchical set of binary search trees where each level iii corresponds to a trie storing keys of length iii. At the lowest level, the leaves store the actual keys, and each internal node at level iii maintains pointers to predecessor and successor nodes in the trie. This structure allows for rapid traversal and query operations.

Operations on X-Fast Tries

X-Fast Tries are advanced data structures that support efficient operations on dynamic sets of keys. Here's how each operation is performed:

Insertion

Search for Insertion Point:

Begin at the root of the X-Fast Trie and traverse down through levels according to the bits of the key being inserted.

Determine the path in the trie that matches the key's bits until you reach the leaf level.

Update Predecessor and Successor Pointers:

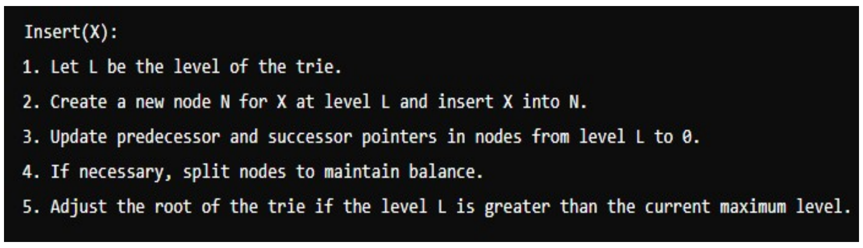
Once the correct leaf node is found (or created if the key doesn't exist), update the predecessor and successor pointers at each level of the trie.

Ensure that these pointers correctly reflect the position of the new key relative to existing keys in the trie.

Balance and Maintenance:

Adjust the trie structure as necessary to maintain the O (log log U) time complexity for predecessor/successor queries.

This may involve splitting nodes or merging nodes to balance the trie.



```
Insert(X):
1. Let L be the level of the trie.
2. Create a new node N for X at level L and insert X into N.
3. Update predecessor and successor pointers in nodes from level L to 0.
4. If necessary, split nodes to maintain balance.
5. Adjust the root of the trie if the level L is greater than the current maximum level.
```

Deletion

Search for the Key:

Start at the root and traverse the trie to locate the node that contains the key to be deleted.

Adjust Predecessor and Successor Pointers:

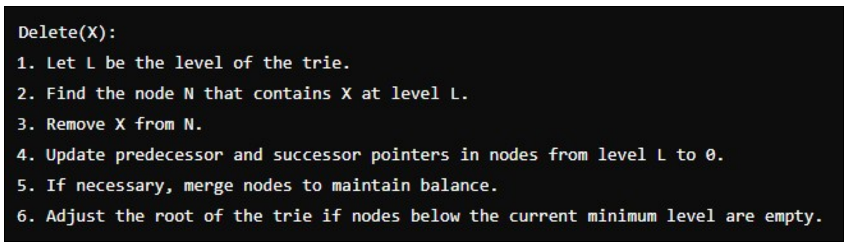
Update the predecessor and successor pointers at each level to reflect the removal of the key.

Ensure that the trie remains balanced and maintains the desired time complexity for queries.

Remove the Key:

Once the correct node is found, remove the key from the trie structure.

Adjust the trie as needed to maintain its integrity and balance.



```
Delete(X):
1. Let L be the level of the trie.
2. Find the node N that contains X at level L.
3. Remove X from N.
4. Update predecessor and successor pointers in nodes from level L to 0.
5. If necessary, merge nodes to maintain balance.
6. Adjust the root of the trie if nodes below the current minimum level are empty.
```

Searching

Search Operation:

Begin at the root of the trie and traverse down through levels according to the bits of the search key.

Follow the path in the trie that matches the bits of the key until you reach the leaf level.

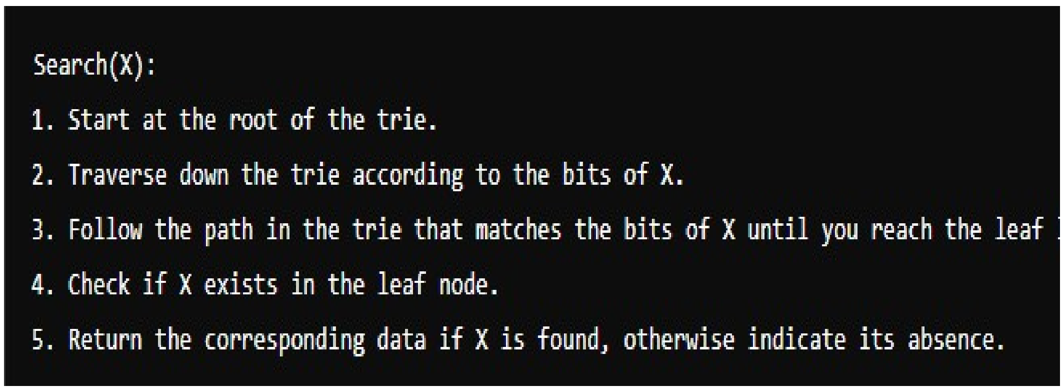
Validation:

Check if the key exists in the leaf node reached by the traversal.

If the key is found, return the corresponding data or indicate its presence.

Handling Non-existent Keys:

If the key is not found during the traversal, handle the search operation accordingly (e.g., returning a null value or indicating absence).



```
Search(X):
1. Start at the root of the trie.
2. Traverse down the trie according to the bits of X.
3. Follow the path in the trie that matches the bits of X until you reach the leaf
4. Check if X exists in the leaf node.
5. Return the corresponding data if X is found, otherwise indicate its absence.
```

15.6 CONCLUSION

Since the introduction of advanced data structures like Scapegoat Trees, Tries, Binary Tries, X-Fast Tries, and Y-Fast Tries, we have explored their unique characteristics and operations. Each structure offers distinct advantages in terms of efficiency and applicability across different problem domains.

Scapegoat Trees provide a balanced tree structure with efficient insertion, deletion, and search operations, leveraging a dynamic resizing mechanism to maintain balance. Tries, known for their suitability in string-related applications, offer fast prefix search capabilities and are used extensively in dictionary implementations and autocomplete features. Binary Tries extend this concept to binary structures, accommodating more diverse data types and enhancing search efficiency.

The introduction of X-Fast Tries introduces a hierarchical approach to searching, leveraging hash tables at multiple levels for rapid data retrieval. Similarly, Y-Fast Tries further optimize this structure by addressing the limitations of X-Fast Tries, particularly in terms of space complexity and query performance.

In conclusion, these advanced data structures represent significant advancements in data management and algorithm design, catering to modern computational needs across various industries. Understanding their intricacies and applications equips developers with powerful tools to tackle complex data organization and retrieval challenges effectively.

15.7 QUESTIONS AND ANSWERS

1. What are the main characteristics of Scapegoat Trees?
Answer: Scapegoat Trees are self-balancing binary search trees that maintain balance through periodic rebuilding. They ensure that no node's height exceeds a certain threshold, balancing the tree by performing partial or complete rebuilds when necessary. This results in efficient insertion, deletion, and search operations with guaranteed logarithmic time complexity.

2. How do Tries differ from traditional binary search trees?
Answer: Tries are specialized tree-like data structures used for storing associative arrays, typically strings. Unlike binary search trees that compare entire keys, Tries store characters of keys at each node, enabling efficient prefix-based searching. This makes Tries especially useful for applications like autocomplete and spell checkers.

3. What are Binary Tries and how are they utilized?
Answer: Binary Tries are a variation of Tries optimized for binary data. Each node in a Binary Trie represents a bit in the key, leading to a maximum tree height equal to the bit-length of the keys. They are commonly used in networking for IP routing and prefix matching due to their efficient handling of binary data.

4. Explain the concept of X-Fast Tries.

Answer: X-Fast Tries are advanced data structures that use a combination of Trie structures and hash tables to achieve efficient lookups, insertions, and deletions. They reduce the height of the Trie by storing nodes at various levels in hash tables, allowing for faster access times compared to traditional Tries.

5. What advantages do Y-Fast Tries offer over X-Fast Tries?

Answer: Y-Fast Tries improve upon X-Fast Tries by further optimizing space and query efficiency. They use a hierarchical structure where elements are grouped into buckets, each managed by a balanced binary search tree. This allows for efficient space usage while maintaining fast query performance.

6. In what scenarios would you prefer using a Scapegoat Tree over other balanced trees like AVL or Red-Black Trees?

Answer: Scapegoat Trees are particularly useful when insertions and deletions are more frequent and when predictable performance is essential. Their periodic rebalancing mechanism can be advantageous in environments where maintaining strict balance at all times (like in AVL or Red-Black Trees) might introduce overhead.

7. Can you give an example of a real-world application of Tries?

Answer: One common real-world application of Tries is in implementing autocomplete features in search engines. As users type in search queries, the Trie structure allows for efficient prefix matching, quickly suggesting possible completions based on the input provided so far.

15.8 REFERENCES

- Bjarne Stroustrup, "The C++ Programming Language"
- Herb Sutter, "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions"
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, "C++ Primer"
- Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Nicolai M. Josuttis, "C++ Standard Library: A Tutorial and Reference"

UNIT – 16: FILE STRUCTURES

Structure

16.0 INTRODUCTION

In the realm of data management, efficient storage, organization, and retrieval of information are paramount. File structures play a critical role in achieving these objectives by providing systematic methods to manage data within files. The selection of an appropriate file structure can greatly impact the performance, accessibility, and overall efficiency of data handling operations.

This unit delves into various file structures, including sequential, direct (random), and indexed sequential file organizations. Each of these structures offers distinct advantages and is suitable for different types of applications, depending on the specific requirements of data access patterns and storage needs. Understanding these structures is essential for optimizing data storage and retrieval processes in various computing environments.

Moreover, we will explore fundamental file operations that are crucial for manipulating and managing files effectively. This includes file creation, opening, closing, reading, writing, and deletion. Additionally, we will examine practical applications of these file structures in real-world scenarios, highlighting their significance in database management systems, information retrieval systems, and file management systems. This comprehensive overview will provide a solid foundation for understanding the importance of file structures and their practical applications in data management.

16.1 OBJECTIVES

After completing this unit, you will be able to underst and,

Understand Different File Structures: Gain a comprehensive understanding of various file structures, including sequential, direct (random), and indexed sequential file organizations, and their respective advantages and disadvantages.

Learn File Operations: Explore the fundamental file operations such as creation, opening, closing, reading, writing, and deletion, and understand their implementation and usage in different file structures.

Analyze File Organization Types: Examine the characteristics, benefits, and limitations of different file organization types, and understand how they impact data storage, retrieval efficiency, and overall system performance.

Apply Knowledge to Real-World Scenarios: Investigate practical applications of different file structures in various domains such as database management systems, information retrieval systems, and file management systems, and understand how to choose the appropriate file structure for specific use cases.

Develop Skills in Implementing Algorithms: Gain hands-on experience in implementing algorithms for file operations and file structures, particularly in the C++ programming language, to solidify theoretical knowledge through practical application.

16.2 FILE STRUCTURES

File structures in data structures encompass various methodologies for organizing and managing data within computer systems. At its core, file structures refer to how data is stored, accessed, and manipulated in files, which are logical collections of records.
File structures refer to the organization and layout of data in computer files, essential for efficient data storage, retrieval, and management. The structure of a file determines how data is stored within it, the methods used to access and modify that data, and the overall efficiency of operations performed on the file.

The primary components of file structures include:
Record Format: Defines how data is organized within each record, specifying the type and order of fields (data elements) stored in the file. Records can be of fixed length or variable length, depending on the application's requirements.

A **record format** defines how data is structured and organized within each record in a file. It specifies the layout, type, and order of data elements (fields) stored in the record. The format chosen depends on the nature of the data being stored and the requirements of the application accessing the file.

Key aspects of a record format include:

Field Definition: Each field represents a data item of a specific type (integer, floating-point number, string, etc.). Fields are typically defined with a fixed length or a maximum length for variable-length fields.

Field Order: Specifies the sequence in which fields are arranged within the record. This order is crucial for correctly interpreting and accessing the data during read and write operations.

Field Attributes: Attributes such as field names, data types (e.g., integer, character, date), and constraints (e.g., maximum length, allowed values) are defined to ensure data integrity and facilitate efficient querying and manipulation.

Delimiter or Separator: In some formats, especially text-based ones, fields may be separated by delimiters (e.g., commas, tabs, spaces) or have fixed positions within the record structure.

Padding: Padding refers to the addition of extra characters or bytes to ensure that each field occupies its allotted space within the record. This helps maintain alignment and facilitates efficient storage and retrieval operations.

File Organization: File organization refers to the way data is stored and structured within files on a computer's storage system. It encompasses various methods and techniques designed to optimize data access, retrieval, and management. The choice of file organization depends on factors such as the type of data, access patterns, and efficiency considerations. It describes how records are physically arranged within the file. Common file organizations include sequential, indexed sequential, direct (or hashed), and more complex structures like B-trees for large-scale databases. Each organization method offers different trade-offs in terms of access speed, storage efficiency, and ease of modification.

Access Methods: Determine how data can be retrieved from or stored into the file. Sequential access reads data in order from start to end, making it suitable for batch processing. Direct access allows quick retrieval of records based on their storage location, beneficial for random access operations. Indexed access combines the benefits of both, using a separate index structure to facilitate fast lookups based on keys.
Access methods in the context of file organization refer to the techniques and algorithms used to retrieve and manipulate data stored within files. These methods are crucial for efficient data access and retrieval in computer systems. Here's an overview of common access methods:

Types of Access Methods:
Sequential Access:
Description: In sequential access, data is accessed in a linear or sequential manner, starting from the beginning of the file and proceeding sequentially to the end.

Usage: Suitable for applications that process data in a batch mode or require full file scans, such as processing logs or sequential data streams.

Advantages: Simple to implement and efficient for reading large amounts of data sequentially.

Disadvantages: Not efficient for random access or frequent updates, as accessing data out of sequence requires scanning through all preceding records.

Direct Access:

Description: Direct access allows data to be accessed randomly by specifying a key or address associated with each record. This method enables quick retrieval and modification of specific records without needing to traverse others.

Usage: Ideal for applications that require frequent random access to data, such as databases and real-time systems.

Advantages: Enables fast retrieval and modification of records using direct addressing based on keys or addresses.

Disadvantages: More complex to implement compared to sequential access; may lead to fragmentation of data and increased storage overhead.

Indexed Access:

Description: Indexed access combines the benefits of sequential and direct access methods. It involves maintaining an index structure alongside the main data file, which maps keys to physical addresses or offsets of records.

Usage: Suitable for applications that require both sequential and random access patterns, balancing efficient access with flexibility.

Advantages: Provides efficient retrieval and modification operations using indexed lookup, supports both sequential and random access patterns.

Disadvantages: Requires additional storage for maintaining index structures; insertion and deletion operations may be slower due to index maintenance.

Hashing:

Description: Hashing involves mapping keys directly to addresses using a hash function, which calculates the storage location based on the key's value. It enables rapid access to data by reducing search time to constant time complexity.

Usage: Commonly used in hash tables and hash-based data structures for fast data retrieval and storage.

Advantages: Provides constant-time average access for retrieval and insertion operations, efficient for large datasets.

Disadvantages: Collision handling (when two keys hash to the same address) requires additional processing; not suitable for range queries or ordered data retrieval.

16.3 SEQUENTIAL FILE ORGANIZATION

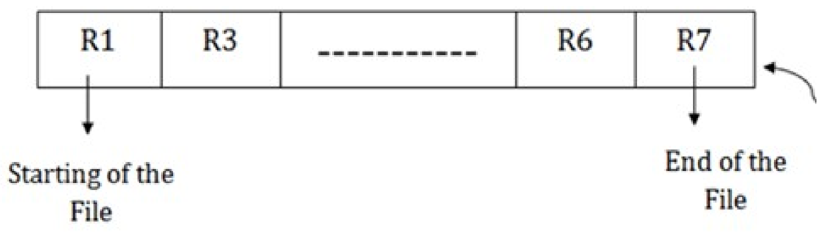
Sequential file organization is a method of storing and accessing data records in a sequential order, typically based on their physical placement in the file. Here's a detailed explanation of sequential file organization:

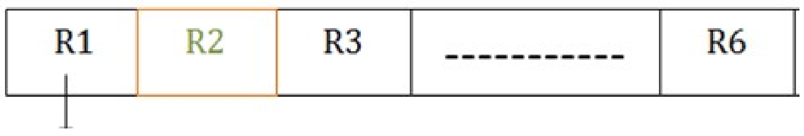
Overview and Characteristics:

Sequential file organization arranges data records consecutively in the order they are added to the file. Each record is stored immediately after the previous one, forming a continuous sequence. Key characteristics include:

Storage Structure: Data records are stored one after another in a linear fashion within the file. This layout simplifies appending new records but complicates insertion and deletion operations, as they may require shifting subsequent records.
Access Method: Data access is performed sequentially, starting from the beginning of the file and continuing to the end. This means to access a specific record, all preceding records must be read sequentially.

Applications: Sequential files are suitable for applications where data is processed in batches or where data is primarily read sequentially, such as log files, transaction processing systems with archival needs, and batch processing applications.





(Image Source: Javat Point)
Operations and Usage:
Sequential files support basic operations tailored to their access pattern:

Reading: Data is read sequentially from the beginning of the file to the end. This operation is efficient for processing large volumes of data sequentially without requiring random access.

Writing: New records are typically appended to the end of the file, which simplifies insertion. However, modifying or deleting existing records may require rewriting the entire file after the modification point.

Searching: Sequential searching involves scanning the file from the start until the desired record is found. This process can be slow for large files or when the record is located towards the end of the file.

Advantages and Disadvantages:

Advantages:

Simple and easy to implement.

Efficient for applications that primarily read data sequentially.

Requires less overhead compared to indexed or direct access methods.

Disadvantages:

Inefficient for random access operations, as accessing records out of sequence requires scanning through all preceding records.

Insertions and deletions may be slow and costly, especially in large files.

Not suitable for applications requiring frequent updates or random access patterns.

16.4 DIRECT (RANDOM) FILE ORGANIZATION

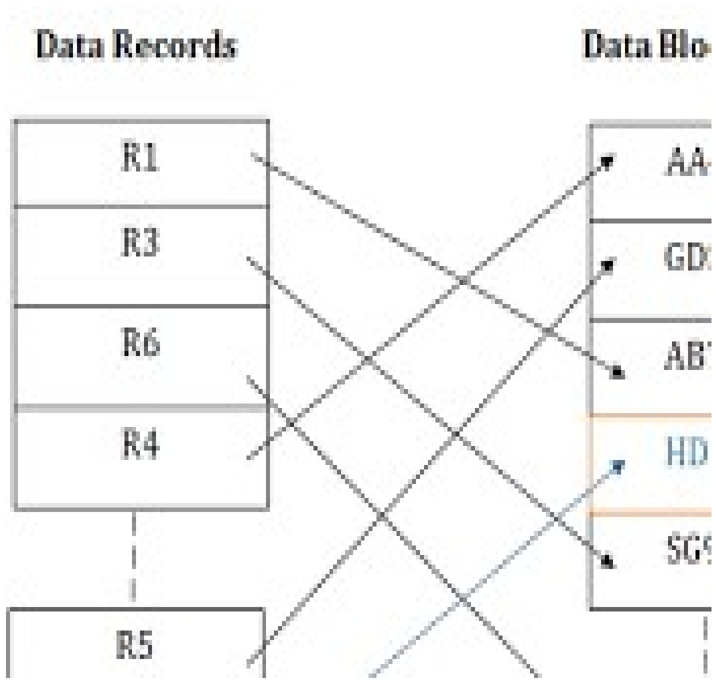
Direct file organization, also known as random file organization, is a method of organizing data in a file that allows for direct access to any record based on its unique identifier or key. Unlike sequential file organization where records are stored in a linear sequence, direct file organization uses indexing or hashing techniques to facilitate rapid access to specific records. Here’s a detailed explanation of direct file organization:

Overview and Characteristics:
Direct file organization employs indexing or hashing to map record keys to specific locations within the file. Key characteristics include:

Indexing: Each record in the file is assigned a unique key, which serves as an index. This index is used to directly locate the record within the file, bypassing the need to sequentially read through preceding records.
Hashing: Alternatively, records may be stored in the file using a hashing function that computes a location based on the record's key. This allows for rapid calculation of the record's storage location and retrieval.

Access Method: Accessing records in a direct file involves using the record's key to determine its location in the file. This method enables efficient random access, where any record can be retrieved directly without scanning through other records.

Applications: Direct file organization is suitable for applications requiring frequent and rapid access to specific data records, such as database systems, file systems, and data retrieval systems.



(Image Source: JavatPoint)

Operations and Usage:

Direct file organization supports operations tailored to random access patterns:

Reading: Records can be retrieved directly using their unique keys, making retrieval operations efficient even for large files.

Writing: New records can be added directly into the file at their designated locations based on their keys. This operation requires updating the index or hash table accordingly.

Updating: Existing records can be modified or deleted efficiently by directly accessing and modifying their locations in the file.

Advantages and Disadvantages:

Advantages:

Enables rapid access to specific records without scanning through other records.

Efficient for applications requiring frequent data retrieval based on specific criteria.

Supports direct insertion, deletion, and modification operations.

Disadvantages:

Requires additional overhead for maintaining and updating indexes or hash tables.

Complexities may arise in handling collisions in hashing-based implementations.

Initial setup and maintenance of indexes or hash tables can introduce additional complexity.

16.5 INDEXED SEQUENTIAL FILE ORGANIZATION

Indexed Sequential File Organization combines the benefits of both sequential and direct (random) file organization methods. It is designed to optimize data retrieval and storage efficiency by using indexing for fast access and maintaining sequential order to support range queries and efficient sequential processing. Here's a detailed explanation of Indexed Sequential File Organization:

Overview and Characteristics:

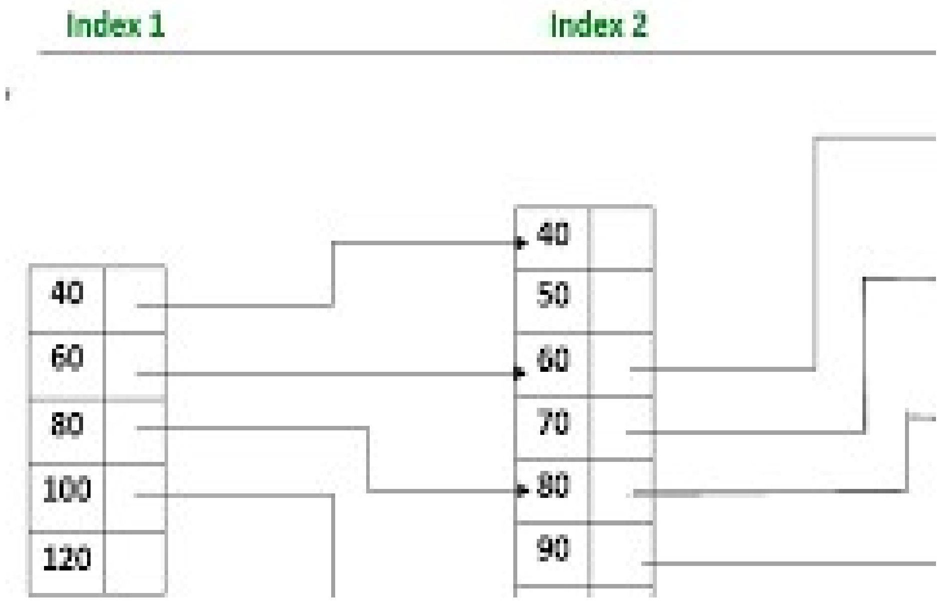
Indexed Sequential File Organization organizes records in a sequential manner on disk while maintaining an index structure that allows for direct access to individual records based on keys. Key characteristics include:

Sequential Storage: Records are stored sequentially on disk, which facilitates efficient sequential processing of data.

Indexing: Each record has a unique key, and an index is maintained separately to map these keys to their physical locations in the file. This index enables fast direct access to specific records.

Access Method: Records can be accessed directly using their keys through the index, allowing for rapid retrieval operations similar to direct file organization.

Hybrid Approach: Combines the benefits of sequential organization (efficient sequential processing) with direct organization (rapid access to individual records).



(Image Source: Geeeksforgeeks)

Operations and Usage:

Indexed Sequential File Organization supports various operations tailored to both random and sequential access patterns:

Indexing Structure: Typically, a B-tree or a multi-level index structure is used to maintain efficient access to records. This structure optimizes searches, insertions, and deletions.

Reading: Records can be retrieved directly using their keys, leveraging the index structure for rapid access.

Writing: New records are appended to the end of the file sequentially, while the index structure is updated to reflect the new record's location.

Updating: Existing records can be modified or deleted, with updates managed both in the sequential file and the index structure.

Advantages and Disadvantages:

Advantages:

Supports fast access to individual records based on keys through indexing.

Facilitates efficient range queries by maintaining sequential order.

Suitable for applications requiring both random and sequential access patterns.

Disadvantages:

Requires additional overhead for maintaining and updating index structures.

Complexities may arise in managing and balancing index structures, especially in distributed or large-scale systems.

Initial setup and maintenance of indexes can be resource-intensive.

16.6 FILE OPERATIONS

File operations encompass a range of activities involved in managing files within a computer system, typically handled by operating systems or file management libraries. These operations are fundamental for reading, writing, and manipulating data stored in files. Here's an overview of key file operations:

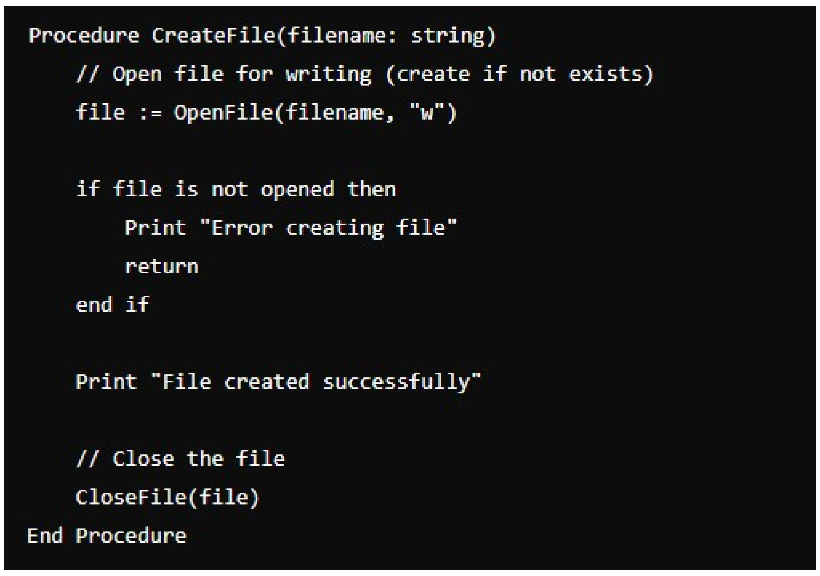
File Operations:

File Creation:

Definition: Creating a new file involves allocating space in the file system and establishing metadata structures to manage the file.

Process: Typically, the operating system or application creates a file by specifying a name, location, and sometimes initial content or attributes.

Algorithm:



```
Procedure CreateFile(filename: string)
    // Open file for writing (create if not exists)
    file := OpenFile(filename, "w")

    if file is not opened then
        Print "Error creating file"
        return
    end if

    Print "File created successfully"

    // Close the file
    CloseFile(file)
End Procedure
```

Explanation:

CreateFile:

Opens the file specified by filename for writing ("w" mode), which creates the file if it does not already exist.

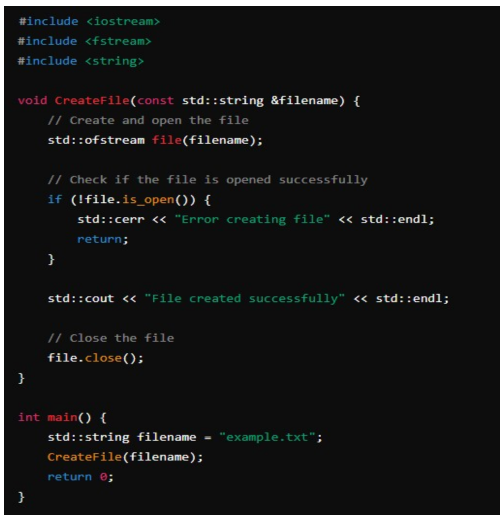
Checks if the file was opened successfully.

Prints a success message if the file was created.

Closes the file after creation.

Here's how this algorithm can be implemented in C++:

Implementation in C++

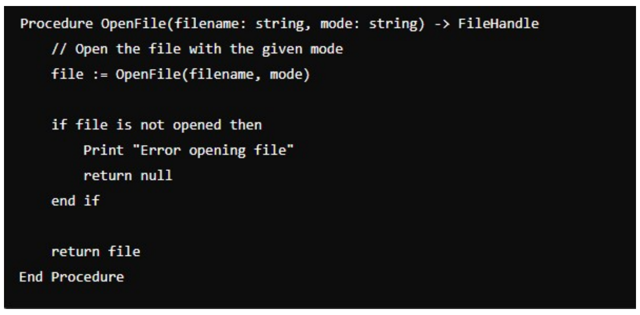


```cpp
#include <iostream>
#include <fstream>
#include <string>

void CreateFile(const std::string &filename) {
    // Create and open the file
    std::ofstream file(filename);

    // Check if the file is opened successfully
    if (!file.is_open()) {
        std::cerr << "Error creating file" << std::endl;
        return;
    }

    std::cout << "File created successfully" << std::endl;

    // Close the file
    file.close();
}

int main() {
    std::string filename = "example.txt";
    CreateFile(filename);
    return 0;
}
```

Opening and Closing Files:

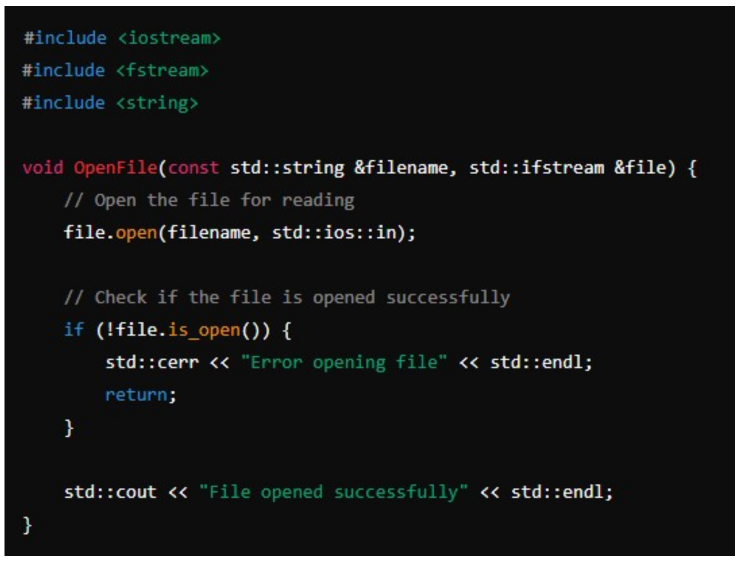
Opening: Accessing a file to perform read or write operations.

Process: Applications request file access by providing a file path or identifier, which the operating system verifies and grants if permissions allow.



```
Procedure OpenFile(filename: string, mode: string) -> FileHandle
    // Open the file with the given mode
    file := OpenFile(filename, mode)

    if file is not opened then
        Print "Error opening file"
        return null
    end if

    return file
End Procedure
```

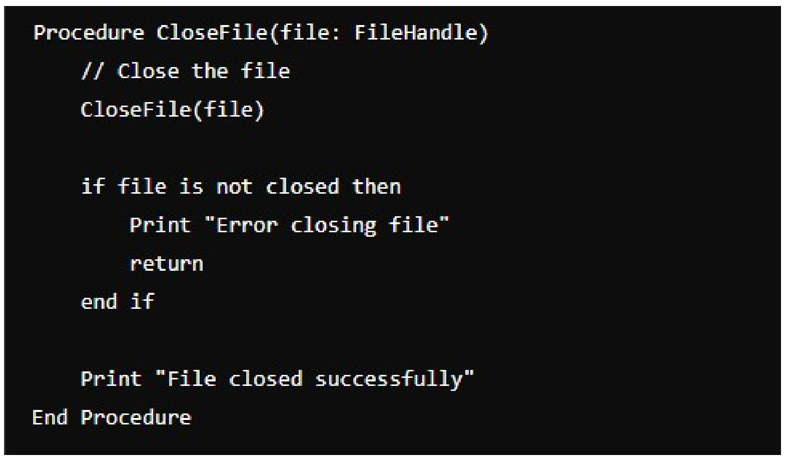
Implementation:



```cpp
#include <iostream>
#include <fstream>
#include <string>

void OpenFile(const std::string &filename, std::ifstream &file) {
    // Open the file for reading
    file.open(filename, std::ios::in);

    // Check if the file is opened successfully
    if (!file.is_open()) {
        std::cerr << "Error opening file" << std::endl;
        return;
    }

    std::cout << "File opened successfully" << std::endl;
}
```

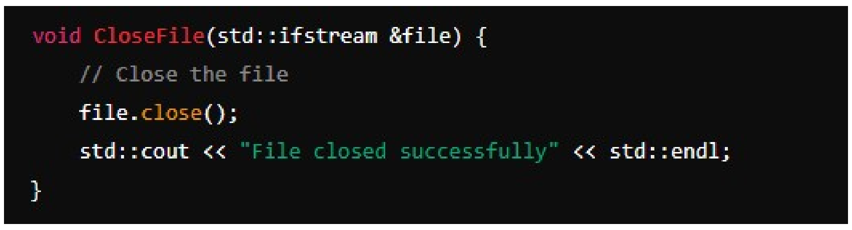
Closing: Releasing resources associated with the file after operations are complete.

Process: Ensures that all data buffers are flushed and file locks, if any, are released.



```
Procedure CloseFile(file: FileHandle)
    // Close the file
    CloseFile(file)

    if file is not closed then
        Print "Error closing file"
        return
    end if

    Print "File closed successfully"
End Procedure
```

Implementation:



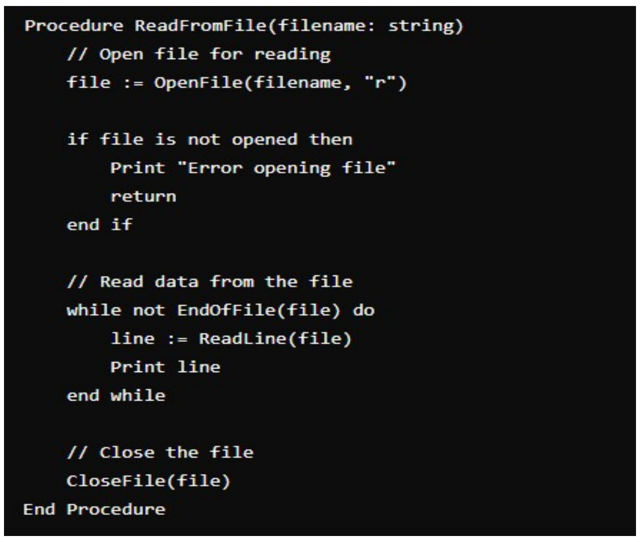
```cpp
void CloseFile(std::ifstream &file) {
    // Close the file
    file.close();
    std::cout << "File closed successfully" << std::endl;
}
```

Reading from Files:

Definition: Retrieving data stored in a file for processing or display.

Process: Applications specify read operations, which involve positioning the file pointer to the desired location within the file and transferring data to memory buffers.

Algorithm:



```
Procedure ReadFromFile(filename: string)
    // Open file for reading
    file := OpenFile(filename, "r")

    if file is not opened then
        Print "Error opening file"
        return
    end if

    // Read data from the file
    while not EndOfFile(file) do
        line := ReadLine(file)
        Print line
    end while

    // Close the file
    CloseFile(file)
End Procedure
```

ReadFromFile:

Opens the file specified by filename for reading ("r" mode).

Checks if the file was opened successfully.

Reads each line from the file until the end of the file (EndOfFile).

Prints each line to the console.

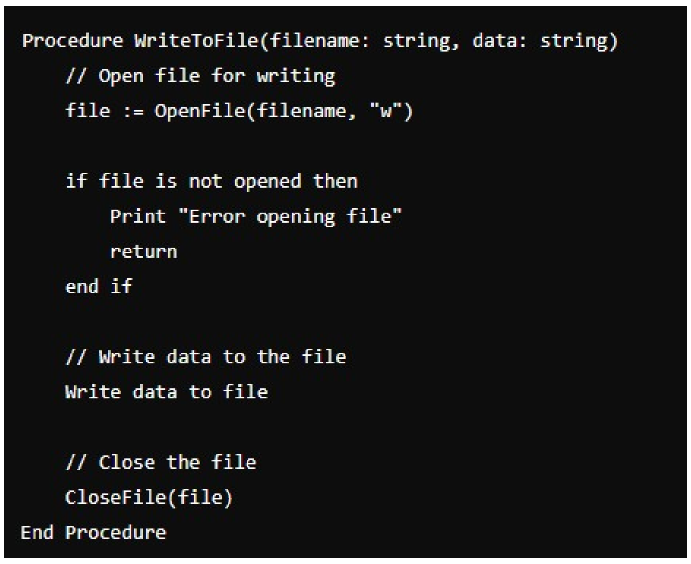
Closes the file after reading.

Writing to Files:

Definition: Storing new data or modifying existing data in a file.

Process: Applications specify write operations, which involve positioning the file pointer and transferring data from memory buffers to the specified location in the file.

Algorithm:



```
Procedure WriteToFile(filename: string, data: string)
    // Open file for writing
    file := OpenFile(filename, "w")

    if file is not opened then
        Print "Error opening file"
        return
    end if

    // Write data to the file
    Write data to file

    // Close the file
    CloseFile(file)
End Procedure
```

WriteToFile:

Opens the file specified by filename for writing ("w" mode).

Checks if the file was opened successfully.

Writes the data string to the file.

Closes the file after writing.

Updating and Modifying Files:

Definition: Changing or appending data within a file.

Process: Combines reading and writing operations, where data is read, modified in memory, and then written back to the file.

16.7 APPLICATIONS

Database Management Systems (DBMS)

File structures are integral to the efficiency and effectiveness of Database Management Systems. Here's how different file structures are applied:

Sequential Files:

Backup and Archival: Sequential files are ideal for creating backups and archives of data because they allow for easy and efficient sequential reading and writing.

Batch Processing: Used in situations where data processing can occur in batches, such as end-of-day processing in banking systems.

Indexed Sequential Files:

Transaction Processing Systems: Often used in transaction processing systems where quick access to records is required, but the records are processed in a sequential manner. The index allows for quick lookup, while the sequential nature aids in efficient data management.

Customer Relationship Management (CRM): Enables fast access to customer records while maintaining an ordered structure for efficient bulk operations.

Direct (Random) Files:

High-Performance Applications: Used in applications requiring rapid access to individual records, such as real-time systems in finance and stock trading.

Database Indexing: Implements hash-based indexing where direct access to records is essential.

Information Retrieval Systems

In information retrieval systems, efficient data storage and quick access are crucial:

Sequential Files:

Log Files: Ideal for maintaining log files where entries are recorded in sequence over time.

Historical Data Archives: Useful for archiving historical data that is not frequently accessed but needs to be stored efficiently.

Indexed Sequential Files:
Search Engines: Used in search engines to store indexed data, allowing quick search operations while maintaining an ordered index for efficient retrieval.
Library Systems: In library management systems, indexed sequential files help in organizing and retrieving book records efficiently.
Direct (Random) Files:

Document Management Systems: Used for storing and retrieving documents where each document needs to be accessed directly without scanning through other records.
User Profile Management: In systems managing large user bases, direct files facilitate quick access to individual user profiles.

File Management Systems
File management systems rely heavily on the underlying file structures to ensure efficient file storage, access, and management:
Sequential Files:

Tape Storage Systems: Commonly used in tape storage systems where data is stored sequentially for backup and archival purposes.
Simple Log Files: Used for simple log file management in operating systems and applications.

Indexed Sequential Files:

File Indexing: Helps in creating indexes of large file directories, allowing for quick searches and organized storage.

Metadata Management: Used in systems that need to maintain and retrieve file metadata efficiently.

Direct (Random) Files:

Operating System File Systems: Employed in operating systems to manage files and directories where rapid access to files is necessary.

Database Index Files: Used for managing database index files where direct access to index entries is critical for performance.

16.8 CONCLUSION

In this unit, we delved into the essential concepts of file structures and their organization within the context of data management systems. We explored the various types of file organizations, including sequential, direct (random), and indexed sequential file organizations, highlighting their unique characteristics and use cases. Understanding these structures is crucial for optimizing data access and storage efficiency, which are foundational elements in the design and implementation of robust data systems.

We also covered the fundamental file operations that underpin these file structures, such as creation, opening, closing, reading, writing, and deletion. Mastery of these operations is essential for effective data management, ensuring that data is accurately and efficiently manipulated within different organizational contexts. By examining the implementation details, particularly in the C++

programming language, we bridged the gap between theoretical knowledge and practical application, providing a comprehensive view of how these concepts are realized in real-world systems.

Finally, we discussed the applications of various file structures in different domains, including database management systems, information retrieval systems, and file management systems. These applications underscore the importance of choosing the right file organization method to meet specific needs and performance requirements. The insights gained from this unit equip us with the knowledge to design and implement efficient file management strategies, ensuring optimal data handling and retrieval in diverse computational environments.

16.9 QUESTIONS AND ANSWERS

1. What are the main types of file organizations covered in this unit?

Answer: The main types of file organizations covered in this unit are:

Sequential File Organization: Data is stored in a linear sequence, making it simple and efficient for reading large blocks of sequential data.

Direct (Random) File Organization: Data is accessed directly using a key or address, providing quick retrieval but requiring more complex management.

Indexed Sequential File Organization: Combines the advantages of both sequential and direct access by maintaining an index to allow for fast searches and sequential data processing.

2. What are the key advantages of sequential file organization?

Answer: Sequential file organization offers several advantages:

Simplicity in implementation and management.

Efficient for operations that process large volumes of data sequentially.

Minimal overhead for file management, as no indexing or hashing is required.

3. How does direct file organization improve data retrieval times?

Answer: Direct file organization improves data retrieval times by using a key or address to directly access the desired data record. This eliminates the need to search through data sequentially, significantly reducing the time required to locate and retrieve specific records.

4. What is an indexed sequential file organization, and how does it work?

Answer: Indexed sequential file organization is a hybrid approach that combines sequential and direct access methods. It maintains an index that allows for quick searches and random access to data records while still enabling efficient sequential data processing. The index maps keys to their corresponding storage locations, providing the benefits of both quick searches and organized sequential data management.

5. What are some common file operations discussed in this unit?

Answer: Common file operations discussed include:

File Creation: Establishing a new file in the storage system.

Opening and Closing Files: Preparing a file for reading or writing and properly closing it after operations are complete.

Reading and Writing: Accessing data from a file and modifying or adding data to a file.

Deletion: Removing a file or specific data records from the storage system.

6. Why is it important to understand different file structures and their applications?

Answer: Understanding different file structures and their applications is crucial because it allows for the selection of the most appropriate file organization method based on the specific needs and performance requirements of an application. This ensures optimal data handling, efficient storage, and quick retrieval, which are vital for the overall performance and reliability of data management systems.

16.10 REFERENCES

- Bjarne Stroustrup, "The C++ Programming Language"
- Herb Sutter, "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions"
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, "C++ Primer"
- Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Nicolai M. Josuttis, "C++ Standard Library: A Tutorial and Reference"