ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ARRAYS, LINKED LISTS (SINGLY, DOUBLY, CIRCULAR)

ARRAYS

An **array** is a fundamental data structure in computer science that stores a fixed-size sequence of elements of the same type. Arrays allow for fast access to elements using indices, making them an essential structure for managing collections of data in many types of applications, from databases to games. Arrays are often used when you know in advance how many elements you will need to store, as they provide constant-time access to any element by index.

In C++, an array is declared by specifying the data type of the elements followed by square brackets containing the size of the array. Once declared, an array occupies a contiguous block of memory, and each element is accessible via an index, starting from zero. The size of an array is fixed after it is declared, meaning that the number of elements cannot be altered dynamically. However, arrays are efficient in terms of memory usage and data access.

Example: Basic Array Usage in C++

```
#include <iostream>

using namespace std;
```

```
int main() {
   int arr[5] = {1, 2, 3, 4, 5};  // Declaring and initializing an array

   for(int i = 0; i < 5; i++) {
      cout << "Element " << i + 1 << ": " << arr[i] << endl;  // Accessing elements by index
   }

   return 0;
}
```

Explanation: In this example, an array arr of size 5 is initialized with values from 1 to 5. We use a for loop to access each element by its index and print it to the console. The arr[i] notation is used to access each element in the array.

KEY POINTS ABOUT ARRAYS

1. **Fixed Size**: The size of an array must be specified at the time of creation and cannot be changed during runtime.
2. **Indexed Access**: Arrays allow efficient access to elements using an index, which makes them suitable for situations where fast access to elements is required.
3. **Memory Contiguity**: Arrays are stored in contiguous memory locations, which allows for efficient access but also means they can't dynamically grow or shrink.

LINKED LISTS

A **linked list** is another fundamental data structure where elements are stored in nodes. Each node contains two parts: the data and a pointer (or reference) to the next node in the sequence. Linked lists are particularly useful for dynamic data storage where the number of elements is not known in advance, or when frequent insertions and deletions are required. Unlike arrays, linked lists are not stored in contiguous memory locations, and elements are linked via pointers.

SINGLY LINKED LISTS

A **singly linked list** is the simplest form of a linked list. In a singly linked list, each node points to the next node in the sequence, and the last node points to nullptr to indicate the end of the list. Singly linked lists provide flexibility in terms of size since nodes can be added or removed dynamically without needing contiguous memory.

Example: Singly Linked List in C++

```
#include <iostream>

using namespace std;


// Node definition

class Node {

public:

  int data;

  Node* next;
```

```
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Insert at the end of the list
    void insert(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
        } else {
```

```
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Print the linked list
void printList() {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
};

int main() {
    LinkedList list;
```

```
    list.insert(10);

    list.insert(20);

    list.insert(30);


    list.printList();  // Output: 10 20 30


    return 0;
}
```

Explanation: In this code, the Node class defines a single node in the list, which holds the data and a pointer to the next node. The LinkedList class contains a pointer to the first node (head) and methods to insert nodes at the end and print the list. The insert() method traverses the list until the last node and appends the new node.

KEY POINTS ABOUT SINGLY LINKED LISTS

1. **Dynamic Size**: Unlike arrays, the size of a linked list can grow or shrink dynamically.
2. **Non-Contiguous Memory**: Nodes are stored in non-contiguous memory locations, and each node has a pointer to the next one.
3. **Efficient Insertions and Deletions**: Insertions and deletions in a singly linked list can be done in constant time, especially at the beginning or the end of the list.

DOUBLY LINKED LISTS

A **doubly linked list** is similar to a singly linked list but with an added complexity: each node has two pointers. One points to the next node, and the other points to the previous node. This allows for easier traversal in both directions (forward and backward), which makes certain operations more efficient.

Example: Doubly Linked List in C++

```
#include <iostream>

using namespace std;


class Node {

public:

  int data;

  Node* next;

  Node* prev;


  Node(int value) {

    data = value;

    next = nullptr;

    prev = nullptr;

  }

};
```

```
class DoublyLinkedList {
public:
  Node* head;

  DoublyLinkedList() {
    head = nullptr;
  }

  // Insert at the end of the doubly linked list
  void insert(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
      head = newNode;
    } else {
      Node* temp = head;
      while (temp->next != nullptr) {
        temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
```

```
    }
}

// Print the doubly linked list
void printList() {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Print the list in reverse
void printReverse() {
    Node* temp = head;
    while (temp != nullptr && temp->next != nullptr) {
        temp = temp->next;
    }
    while (temp != nullptr) {
        cout << temp->data << " ";
```

```
        temp = temp->prev;
    }
    cout << endl;
  }
};

int main() {
  DoublyLinkedList list;
  list.insert(10);
  list.insert(20);
  list.insert(30);

  list.printList();     // Output: 10 20 30
  list.printReverse();  // Output: 30 20 10

  return 0;
}
```

Explanation: The DoublyLinkedList class has a similar structure to the singly linked list but with an additional pointer (prev) in the Node class, which points to the previous node. The printReverse() method demonstrates how to traverse the list backward by following the prev pointers.

KEY POINTS ABOUT DOUBLY LINKED LISTS

1. **Bidirectional Traversal**: Doubly linked lists allow traversal in both directions.
2. **More Memory Usage**: Each node in a doubly linked list uses extra memory for the prev pointer.
3. **Efficient Deletions**: Deletions from the middle of the list are more efficient compared to singly linked lists because you have direct access to the previous node.

CIRCULAR LINKED LISTS

A **circular linked list** is a variation where the last node points back to the first node, forming a loop. This structure can be useful for problems where the last node needs to link back to the first node to create a circular flow, like in round-robin scheduling.

Example: Circular Linked List in C++

```
#include <iostream>

using namespace std;


class Node {

public:

  int data;

  Node* next;
```

```
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }

    // Insert at the end of the circular linked list
    void insert(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;  // Point to itself
        } else {
```

```
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Point to head to make it circular
    }
}

// Print the circular linked list
void printList() {
    if (head == nullptr) return;
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
};
```

```
int main() {
    CircularLinkedList list;
    list.insert(10);
    list.insert(20);
    list.insert(30);

    list.printList();  // Output: 10 20 30

    return 0;
}
```

Explanation: The CircularLinkedList class has an insert() method that ensures the last node points to the first node to form a circular structure. The printList() method traverses the list until it loops back to the head.

Key Points About Circular Linked Lists

1. **Circular Traversal**: The list can be traversed indefinitely by following the next pointers.
2. **Efficient Memory**: Circular linked lists avoid the need for nullptr checks when looping back to the head.

CONCLUSION

In this study material, we've covered **arrays** and **linked lists** (singly, doubly, and circular), which are essential data structures used to

store collections of data. Arrays provide efficient indexing but have fixed sizes, while linked lists are dynamic and allow for efficient insertions and deletions. The different types of linked lists (singly, doubly, and circular) provide varying levels of flexibility for managing data and traversing through elements. These concepts are fundamental in programming, and understanding them is essential for developing efficient and scalable systems.

EXERCISES

1. **Arrays Exercise**:
    - Write a program to perform basic operations on an array, such as searching for an element, finding the maximum value, and sorting the array.
2. **Linked List Exercise**:
    - Implement a function to reverse a singly linked list and print the reversed list.
3. **Circular Linked List Exercise**:
    - Write a program to detect a loop in a circular linked list.

STACKS & QUEUES: IMPLEMENTATION & APPLICATIONS

STACKS

A **stack** is a linear data structure that follows the **Last In, First Out** (LIFO) principle. In a stack, the last element added to the stack is the first one to be removed. Stacks are often used in situations where you need to keep track of function calls, undo operations in software applications, and perform operations in a specific order, such as evaluating mathematical expressions.

A stack can be visualized as a collection of elements, with two primary operations:

1. **Push**: Adds an element to the top of the stack.
2. **Pop**: Removes the element from the top of the stack.

Additionally, stacks typically support the following operations:

- **Peek** or **Top**: Returns the top element without removing it.
- **IsEmpty**: Checks whether the stack is empty.
- **Size**: Returns the number of elements in the stack.

STACK IMPLEMENTATION IN C++

In C++, a stack can be implemented using either an array or a linked list. However, it is commonly implemented using the std::stack container from the Standard Template Library (STL). For illustration, here's how to implement a stack using an array:

```
#include <iostream>
```

```
using namespace std;


class Stack {
private:
   int *arr;
   int top;
   int capacity;

public:
   Stack(int size) {
      capacity = size;
      arr = new int[size];
      top = -1;
   }

   // Push an element onto the stack
   void push(int x) {
      if (top == capacity - 1) {
         cout << "Overflow: Stack is full" << endl;
         return;
      }
```

```
    arr[++top] = x;
    cout << x << " pushed to stack" << endl;
}

// Pop an element from the stack
int pop() {
    if (top == -1) {
        cout << "Underflow: Stack is empty" << endl;
        return -1;
    }
    return arr[top--];
}

// Peek the top element
int peek() {
    if (top == -1) {
        cout << "Stack is empty" << endl;
        return -1;
    }
    return arr[top];
}
```

```
    // Check if the stack is empty
    bool isEmpty() {
        return top == -1;
    }

    // Return the size of the stack
    int size() {
        return top + 1;
    }
};

int main() {
    Stack stack(5);

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.push(40);
    stack.push(50);
```

```
    cout << stack.pop() << " popped from stack" << endl;


    cout << "Top element is " << stack.peek() << endl;
    cout << "Stack size is " << stack.size() << endl;


    return 0;
}
```

Explanation:

- The Stack class uses an array (arr) to store elements, and top is used to track the index of the top element.
- The push() function adds elements to the stack, while the pop() function removes and returns the top element.
- The peek() function returns the top element without removing it.
- isEmpty() checks if the stack is empty, and size() returns the current number of elements in the stack.

Applications of Stacks

1. **Expression Evaluation**: Stacks are widely used to evaluate mathematical expressions such as infix, prefix, and postfix notations.
2. **Function Call Stack**: Every time a function is called, its information is pushed onto the call stack, and it is popped when the function returns.

3. **Undo Mechanism**: Stacks can be used in applications (like text editors) to keep track of changes so that the user can undo operations in reverse order.

QUEUES

A **queue** is another linear data structure, but it follows the **First In, First Out** (FIFO) principle. In a queue, the first element added is the first one to be removed. This is similar to a line of people waiting for a service, where the first person to arrive is the first one to be served.

A queue supports two main operations:

1. **Enqueue**: Adds an element to the back of the queue.
2. **Dequeue**: Removes the element from the front of the queue.

Additional operations include:

- **Front**: Returns the front element without removing it.
- **IsEmpty**: Checks whether the queue is empty.
- **Size**: Returns the number of elements in the queue.

QUEUE IMPLEMENTATION IN C++

Like stacks, queues can be implemented using arrays or linked lists. Below is a simple queue implementation using an array in C++:

```
#include <iostream>

using namespace std;


class Queue {
```

```
private:
    int *arr;
    int front;
    int rear;
    int capacity;

public:
    Queue(int size) {
        capacity = size;
        arr = new int[size];
        front = 0;
        rear = -1;
    }

    // Enqueue an element to the queue
    void enqueue(int x) {
        if (rear == capacity - 1) {
            cout << "Overflow: Queue is full" << endl;
            return;
        }
        arr[++rear] = x;
```

```
    cout << x << " enqueued to queue" << endl;
}

// Dequeue an element from the queue
int dequeue() {
    if (front > rear) {
        cout << "Underflow: Queue is empty" << endl;
        return -1;
    }
    return arr[front++];
}

// Get the front element
int frontElement() {
    if (front > rear) {
        cout << "Queue is empty" << endl;
        return -1;
    }
    return arr[front];
}
```

```
    // Check if the queue is empty
    bool isEmpty() {
        return front > rear;
    }

    // Return the size of the queue
    int size() {
        return rear - front + 1;
    }
};

int main() {
    Queue queue(5);

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.enqueue(40);
    queue.enqueue(50);

    cout << queue.dequeue() << " dequeued from queue" << endl;
```

```
    cout << "Front element is " << queue.frontElement() << endl;

    cout << "Queue size is " << queue.size() << endl;


    return 0;
}
```

Explanation:

- The Queue class uses an array (arr) to store elements, with front and rear pointers to track the positions of the front and rear elements.
- The enqueue() method adds an element to the back of the queue, while the dequeue() method removes and returns the front element.
- The frontElement() method returns the front element without removing it, and isEmpty() checks whether the queue is empty.

APPLICATIONS OF QUEUES

1. **Scheduling Tasks**: Queues are used in scheduling tasks in operating systems, where tasks are processed in the order they arrive.
2. **Breadth-First Search (BFS)**: Queues are essential in graph traversal algorithms like BFS, where nodes are processed level by level.

3. **Traffic Management**: In traffic control systems, queues are used to manage vehicles waiting at signals, ensuring they move in a fair order.

CIRCULAR QUEUES

A **circular queue** is a type of queue where the last element is connected to the first element, forming a circle. This is useful when you want to efficiently manage memory in cases where the queue is used in a fixed-size buffer, such as in computer networking or resource management.

In a circular queue, when the rear pointer reaches the end of the queue, it is reset to the beginning, thus using memory efficiently.

Circular Queue Implementation

```
#include <iostream>

using namespace std;


class CircularQueue {

private:

  int *arr;

  int front, rear, capacity;


public:

  CircularQueue(int size) {
```

```
    capacity = size;
    arr = new int[size];
    front = -1;
    rear = -1;
}

// Enqueue an element to the queue
void enqueue(int x) {
    if ((rear + 1) % capacity == front) {
        cout << "Overflow: Queue is full" << endl;
        return;
    }
    if (front == -1) front = 0;
    rear = (rear + 1) % capacity;
    arr[rear] = x;
    cout << x << " enqueued to queue" << endl;
}

// Dequeue an element from the queue
int dequeue() {
    if (front == -1) {
```

```
        cout << "Underflow: Queue is empty" << endl;
        return -1;
    }
    int value = arr[front];
    if (front == rear) {
        front = rear = -1;  // Reset the queue when it's empty
    } else {
        front = (front + 1) % capacity;
    }
    return value;
}

// Front element
int frontElement() {
    if (front == -1) {
        cout << "Queue is empty" << endl;
        return -1;
    }
    return arr[front];
}
```

```
    // Check if the queue is empty
    bool isEmpty() {
        return front == -1;
    }

    // Size of the queue
    int size() {
        if (front == -1) return 0;
        return (rear - front + 1 + capacity) % capacity;
    }
};

int main() {
    CircularQueue queue(5);

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.enqueue(40);
    queue.enqueue(50);
```

```
    cout << queue.dequeue() << " dequeued from queue" << endl;

    cout << "Front element is " << queue.frontElement() << endl;


    return 0;
}
```

EXPLANATION:

- In the circular queue, when rear + 1 equals front, the queue is full.
- When an element is dequeued, the front pointer is updated circularly, ensuring that elements wrap around.
- The size() method calculates the current size of the queue by considering the circular nature.

Applications of Circular Queues

1. **Fixed-Size Buffers**: Circular queues are used in buffer management in operating systems, network routers, and data streaming, where the data continually moves in a loop.
2. **Round-Robin Scheduling**: In CPU scheduling, circular queues are used for implementing round-robin scheduling algorithms, where processes are given time slices in a circular manner.

CONCLUSION

In this study material, we have explored the concepts of **stacks** and **queues**, two fundamental data structures with various applications in computer science and software development. Stacks follow a

LIFO approach, while queues adhere to the FIFO principle. Both structures are vital for managing tasks that require specific processing orders, like function calls, task scheduling, and expression evaluation. Additionally, circular queues offer an efficient way to manage memory in fixed-size buffers.

Exercises

1. **Stacks Exercise**:
    - Write a program to evaluate a postfix expression using a stack.
2. **Queues Exercise**:
    - Implement a program that simulates the ticketing system using queues, where tickets are served in FIFO order.
3. **Circular Queue Exercise**:
    - Modify the circular queue implementation to handle dynamic resizing when the queue is full.

TREES: BINARY TREES, BINARY SEARCH TREES (BST)

CHAPTER 1: INTRODUCTION TO TREES

Trees are one of the most fundamental and widely used data structures in computer science, designed to efficiently store and manage hierarchical data. A tree is a non-linear data structure composed of nodes, where each node contains a value and a reference to its child nodes. Unlike linear data structures such as arrays, linked lists, stacks, and queues, trees allow for efficient searching, insertion, and deletion operations. Trees are used in various applications, including databases, file systems, artificial intelligence, and network routing.

Each tree has a root node, from which other nodes branch out in a hierarchical manner. Nodes with no children are called leaf nodes, while nodes with at least one child are known as internal nodes. Trees can have various types, including general trees, binary trees, binary search trees, AVL trees, and more. Among these, binary trees and binary search trees (BST) are widely studied and implemented in computer science due to their efficiency in managing and retrieving data.

Key Characteristics of Trees

- **Hierarchy-Based Structure:** Trees follow a parent-child relationship.
- **Recursive Nature:** A tree can be defined recursively, where each subtree is itself a tree.
- **No Cycles:** Trees do not contain cycles, making them a special form of graph known as an acyclic connected graph.

- **Levels and Height:** The level of a node is its distance from the root, while the height of a tree is the longest path from the root to a leaf node.

CHAPTER 2: BINARY TREES

A binary tree is a type of tree data structure in which each node has at most two children, known as the left and right child. The binary tree is extensively used in applications such as expression evaluation, sorting, searching, and network topology.

Properties of a Binary Tree

1. **Maximum Nodes at Level 'L':** The maximum number of nodes at any level 'L' of a binary tree is given by 2L2^L.
2. **Maximum Nodes in a Binary Tree of Height 'h':** The maximum number of nodes possible in a binary tree of height 'h' is 2(h+1)−12^{(h+1)} - 1.
3. **Minimum Height of a Binary Tree:** The minimum height for a binary tree with 'N' nodes is log⁡2(N+1)−1\log_2(N+1) - 1.
4. **Number of Leaf Nodes:** In a binary tree with 'n' total nodes, at least one node is a leaf node.

Types of Binary Trees

- **Full Binary Tree:** Every node in the tree either has zero or two children.
- **Complete Binary Tree:** All levels are fully filled except possibly the last level, which is filled from left to right.
- **Perfect Binary Tree:** All internal nodes have exactly two children, and all leaf nodes are at the same level.

- **Skewed Binary Tree:** A tree in which all nodes have only one child (either left or right), forming a linked-list-like structure.
- **Balanced Binary Tree:** A tree is balanced if the height difference between the left and right subtrees of any node does not exceed one.

Example of a Binary Tree

In this example, node 'A' is the root, nodes 'B' and 'C' are its children, and the tree continues to branch out further.

CHAPTER 3: BINARY SEARCH TREES (BST)

A Binary Search Tree (BST) is a specialized form of a binary tree that follows the property that for each node:

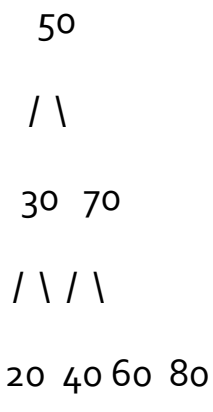
- The left subtree contains nodes with values less than the node's key.
- The right subtree contains nodes with values greater than the node's key.
- The left and right subtrees are also binary search trees.

Properties of BST

1. **Ordered Structure:** The left subtree holds smaller values, and the right subtree holds larger values.

2. **Efficient Searching:** Searching in a BST has an average time complexity of O(log n).
3. **Dynamic Data Storage:** Elements can be inserted and deleted dynamically while maintaining order.
4. **No Duplicates (in most implementations):** Each element appears only once in a standard BST.

Example of a BST



- The left subtree of 50 contains nodes with values less than 50.
- The right subtree of 50 contains nodes with values greater than 50.
- This property holds recursively for all subtrees.

CHAPTER 4: OPERATIONS ON BINARY SEARCH TREES

1. Insertion in BST

Insertion in BST follows a recursive approach where:

- If the tree is empty, the node becomes the root.
- If the value is smaller than the root, it is inserted in the left subtree.

- If the value is larger than the root, it is inserted in the right subtree.

2. Deletion in BST

- **Case 1:** If the node has no children, simply remove it.
- **Case 2:** If the node has one child, replace it with the child.
- **Case 3:** If the node has two children, replace it with its inorder successor or predecessor.

3. Searching in BST

Searching follows the property of BST by comparing the key with the root:

- If the key matches the root, return the node.
- If the key is smaller, search in the left subtree.
- If the key is larger, search in the right subtree.

CHAPTER 5: CASE STUDY – USING BST IN A LIBRARY MANAGEMENT SYSTEM

Consider a library that wants to manage books based on their ISBN numbers efficiently. Using a BST:

- When a new book is added, its ISBN is inserted into the tree.
- Searching for a book becomes efficient as the BST allows O(log n) search time.
- When a book is removed, the BST properties are maintained while deleting the node.

This case study highlights how BSTs are used in real-world applications for fast searching, insertion, and deletion, making them ideal for database management.

EXERCISE

1. Construct a Binary Tree with the following values: 15, 10, 20, 8, 12, 17, 25. Draw its structure.
2. Convert the above Binary Tree into a Binary Search Tree.
3. Perform an inorder, preorder, and postorder traversal on the BST.
4. Write a function in Python to search for a value in a BST.
5. Explain how a balanced binary tree differs from a skewed binary tree with examples.

GRAPHS: REPRESENTATION & TRAVERSAL ALGORITHMS (BFS, DFS)

GRAPHS

A **graph** is a non-linear data structure consisting of a set of vertices (nodes) connected by edges (links). Graphs are used to represent relationships or connections between objects. They are widely used in computer science for a variety of applications such as social networks, computer networks, recommendation systems, and route finding algorithms.

A graph can be classified based on two main characteristics:

1. **Directed vs. Undirected**:
    - **Directed Graph**: The edges have a direction, meaning they go from one vertex to another (e.g., a one-way street).
    - **Undirected Graph**: The edges do not have a direction, meaning the relationship is bidirectional (e.g., a two-way street).
2. **Weighted vs. Unweighted**:
    - **Weighted Graph**: Each edge has a weight or cost associated with it (e.g., distance between two locations).
    - **Unweighted Graph**: All edges are assumed to have equal weight.

Graphs can be represented in two primary ways:

1. **Adjacency Matrix**

2. **Adjacency List**

GRAPH REPRESENTATION

1. **Adjacency Matrix**: This is a 2D array where each cell [i][j] represents the presence of an edge between vertex i and vertex j. The matrix is symmetric for undirected graphs, and for directed graphs, the edge from i to j is represented by matrix[i][j] = 1 (or the weight of the edge).

Example: A graph with 3 vertices and edges between vertex 0 to 1 and 1 to 2 would have the following adjacency matrix:

0 1 0

0 0 1

0 0 0

2. **Adjacency List**: This representation uses an array of lists or vectors. The array's index represents a vertex, and the list at each index contains all the vertices connected to that vertex. This representation is more space-efficient, especially for sparse graphs.

Example: For the same graph as above, the adjacency list representation would be:

0 -> 1

1 -> 2

2 -> (empty)

Graph Implementation in C++

Here's an example of graph representation using an adjacency list in C++:

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

class Graph {
private:
  vector<vector<int>> adjList;

public:
  Graph(int vertices) {
    adjList.resize(vertices);
  }

  // Add an edge to the graph
  void addEdge(int u, int v) {
    adjList[u].push_back(v);
  }

  // Display the adjacency list
  void display() {
    for (int i = 0; i < adjList.size(); i++) {
```

```
            cout << "Vertex " << i << ": ";
            for (int j : adjList[i]) {
                cout << j << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    Graph g(3);  // Create a graph with 3 vertices

    g.addEdge(0, 1);  // Add an edge from vertex 0 to 1
    g.addEdge(1, 2);  // Add an edge from vertex 1 to 2

    g.display();  // Display the graph

    return 0;
}
```

In this code, we create a graph with 3 vertices, add edges, and display the adjacency list.

Traversal Algorithms: BFS and DFS

Graph traversal refers to the process of visiting all the vertices in a graph. There are two primary ways to traverse a graph:

1. **Breadth-First Search (BFS)**
2. **Depth-First Search (DFS)**

Both of these algorithms are used to explore all vertices and edges of a graph, but they differ in their approach.

Breadth-First Search (BFS)

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at a selected node (often referred to as the "root" in trees) and explores the neighbor nodes at the present depth level before moving on to nodes at the next depth level. BFS uses a **queue** data structure to maintain the order of traversal.

BFS Algorithm:

1. Start at the root (or any arbitrary node in the graph).
2. Visit all the neighbors of the current node.
3. Mark the node as visited.
4. Enqueue all unvisited neighbors into a queue.
5. Dequeue a node from the front of the queue and repeat the process for each node in the queue.

BFS Implementation in C++

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Graph {
private:
  vector<vector<int>> adjList;

public:
  Graph(int vertices) {
    adjList.resize(vertices);
  }

  void addEdge(int u, int v) {
    adjList[u].push_back(v);
  }

  void bfs(int start) {
    vector<bool> visited(adjList.size(), false);
    queue<int> q;
```

```
        visited[start] = true;
        q.push(start);

        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << "Visited " << node << endl;

            for (int neighbor : adjList[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }
};

int main() {
    Graph g(4);
```

```
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);

    cout << "BFS starting from vertex 0:" << endl;
    g.bfs(0);

    return 0;
}
```

Explanation:

- BFS starts from vertex 0 and explores all its neighbors (vertices 1 and 2). Then, it moves to vertex 1, explores its neighbor (vertex 3), and continues the process.

APPLICATIONS OF BFS:

1. **Shortest Path**: BFS can be used to find the shortest path in an unweighted graph.
2. **Level Order Traversal**: BFS is used in tree traversal algorithms, such as printing nodes in level order.
3. **Connected Components**: BFS can help identify all nodes in a connected component of a graph.

DEPTH-FIRST SEARCH (DFS)

DFS is an algorithm for traversing or searching tree or graph data structures, starting from the root (or an arbitrary node in the case of a graph). It explores as far down a branch as possible before backtracking. DFS uses a **stack** data structure (either explicit or via recursion) to manage the order of traversal.

DFS Algorithm:

1. Start at the root (or any arbitrary node in the graph).
2. Visit the current node and mark it as visited.
3. Recursively visit all the unvisited neighbors of the current node.
4. Backtrack when no unvisited neighbors are left.

DFS Implementation in C++

```
#include <iostream>

#include <vector>

#include <stack>

using namespace std;


class Graph {

private:

  vector<vector<int>> adjList;


public:

  Graph(int vertices) {
```

```
    adjList.resize(vertices);
}

void addEdge(int u, int v) {
    adjList[u].push_back(v);
}

void dfs(int start) {
    vector<bool> visited(adjList.size(), false);
    stack<int> s;

    visited[start] = true;
    s.push(start);

    while (!s.empty()) {
        int node = s.top();
        s.pop();
        cout << "Visited " << node << endl;

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
```

```
                visited[neighbor] = true;
                s.push(neighbor);
            }
        }
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);

    cout << "DFS starting from vertex 0:" << endl;
    g.dfs(0);

    return 0;
}
```

Explanation:

- DFS starts at vertex 0, explores as far as possible down each branch, and backtracks when necessary.

Applications of DFS:

1. **Topological Sorting**: DFS is used in topological sorting of Directed Acyclic Graphs (DAGs).
2. **Cycle Detection**: DFS helps in detecting cycles in a graph.
3. **Pathfinding**: DFS can be used to find paths between two nodes, although it is not guaranteed to find the shortest path.

Comparison Between BFS and DFS

Feature	BFS	DFS
Traversal Strategy	Level-wise (breadth-first)	Depth-first (deepest node first)
Data Structure Used	Queue	Stack (or recursion)
Space Complexity	O(V + E)	O(V) (with recursion, O(V) call stack)
Time Complexity	O(V + E)	O(V + E)
Applications	Shortest path in unweighted graphs, Level order traversal	Topological sort, Cycle detection, Pathfinding

---

CONCLUSION

In this study material, we've explored the fundamental concepts of **graphs**, their **representation**, and two key **traversal algorithms**: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. BFS is ideal for finding the shortest path in an unweighted graph, while DFS is more suited for tasks like cycle detection and pathfinding. Both algorithms have distinct applications and are crucial in various domains, including network analysis, scheduling, and solving puzzles.

EXERCISES

1. **Graph Representation**:
    - Implement a graph using an adjacency matrix and perform BFS and DFS on the graph.
2. **Shortest Path**:
    - Modify the BFS implementation to return the shortest path from a start vertex to a target vertex in an unweighted graph.
3. **DFS with Recursion**:
    - Implement DFS recursively and compare the iterative and recursive versions in terms of performance.

SORTING & SEARCHING ALGORITHMS

CHAPTER 1: INTRODUCTION TO SORTING & SEARCHING ALGORITHMS

Sorting and searching algorithms are fundamental in computer science, enabling efficient organization and retrieval of data. Sorting arranges data in a specific order (ascending or descending), making it easier to search, analyze, and process. Searching algorithms, on the other hand, help locate specific elements in a data set, ensuring quick access to required information. These algorithms are widely used in database management, data analytics, and real-time applications such as navigation systems, e-commerce platforms, and artificial intelligence.

Sorting algorithms are broadly categorized into comparison-based and non-comparison-based methods. Common sorting techniques include Bubble Sort, Selection Sort, Merge Sort, Quick Sort, and Radix Sort. Searching algorithms, meanwhile, are classified into linear and binary search, each suitable for different types of datasets.

The efficiency of these algorithms is measured in terms of time complexity (execution speed) and space complexity (memory usage). The best sorting or searching technique depends on the dataset size, initial order of data, and required execution speed.

CHAPTER 2: SORTING ALGORITHMS

Sorting algorithms play a crucial role in data organization. Depending on their approach, they can be classified as:

- **Comparison-Based Sorting:** These algorithms compare elements to determine their order. Examples include Bubble Sort, Merge Sort, and Quick Sort.
- **Non-Comparison-Based Sorting:** These use alternative techniques like counting, bucketing, or digit-based sorting. Examples include Radix Sort and Counting Sort.

1. Bubble Sort

Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the entire list is sorted.

Algorithm Steps:

1. Compare adjacent elements in the list.
2. Swap them if necessary to maintain order.
3. Repeat the process for all elements until no swaps are needed.

Example:
Consider the list [5, 3, 8, 4, 2]:
1st Pass: [3, 5, 4, 2, 8]
2nd Pass: [3, 4, 2, 5, 8]
3rd Pass: [3, 2, 4, 5, 8]
4th Pass: [2, 3, 4, 5, 8] (Sorted)

Time Complexity:

- Best Case: O(n)
- Worst Case: $O(n^2)$

2. Merge Sort

Merge Sort is a divide-and-conquer algorithm that splits an array into smaller subarrays, sorts them, and merges them back into a sorted sequence.

Algorithm Steps:

1. Divide the list into two halves.
2. Recursively apply Merge Sort to both halves.
3. Merge the sorted halves.

Example:
Given [6, 3, 7, 1], the steps are:

1. Divide into [6, 3] and [7, 1]
2. Recursively sort [3, 6] and [1, 7]
3. Merge to form [1, 3, 6, 7]

Time Complexity:

- Best, Worst, Average: O(n log n)

3. Quick Sort

Quick Sort selects a pivot, partitions the array around it, and recursively sorts the partitions.

Algorithm Steps:

1. Select a pivot element.
2. Partition the array into elements smaller and greater than the pivot.
3. Recursively apply Quick Sort to both partitions.

Example:
For [8, 3, 1, 7] (pivot = 3):

1. Partition into [1] | 3 | [8, 7]
2. Sort [8, 7] to [7, 8]
3. Merge: [1, 3, 7, 8]

Time Complexity:

- Best/Average: O(n log n)
- Worst: O(n^2)

CHAPTER 3: SEARCHING ALGORITHMS

Searching algorithms find elements in a dataset. The two most common are Linear Search and Binary Search.

1. Linear Search

Linear Search checks each element sequentially until the target value is found.

Algorithm Steps:

1. Start at index 0.
2. Compare each element with the target.
3. If found, return the index; otherwise, continue until the end.

Example:
Searching 7 in [2, 4, 7, 9]:

- Compare 2 (not a match).

- Compare 4 (not a match).
- Compare 7 (match found at index 2).

Time Complexity:

- Best: O(1)
- Worst: O(n)

2. Binary Search

Binary Search works only on sorted lists and divides the search space in half to quickly locate the target.

Algorithm Steps:

1. Find the middle element.
2. If it matches the target, return it.
3. If the target is smaller, repeat search in the left half; otherwise, search in the right half.

Example:
Searching 15 in [5, 10, 15, 20, 25]:

1. Middle = 15 ⟶ found!

Time Complexity:

- Best: O(1)
- Worst: O(log n)

CHAPTER 4: CASE STUDY – SORTING & SEARCHING IN E-COMMERCE

Consider an e-commerce website handling a database of thousands of products. Efficient searching and sorting are essential for:

- **Sorting products by price (low to high).** Quick Sort is effective here.
- **Searching for a product by ID.** Binary Search, combined with indexing, enables instant retrieval.

By using Merge Sort for stability and Binary Search for efficiency, e-commerce platforms can enhance user experience with fast product recommendations and search results.

EXERCISE

1. Implement Bubble Sort, Merge Sort, and Quick Sort in Python.
2. Given an unsorted array, sort it using Merge Sort and show step-by-step execution.
3. Apply Binary Search on [5, 12, 19, 30, 45] and find 19. Show steps.
4. Compare Quick Sort and Merge Sort for large datasets.
5. Analyze how Binary Search Tree can be used to improve search efficiency.

Assignment Solution: Implement a Binary Search Tree (BST) with Insert, Delete, and Search Operations

This step-by-step guide provides a complete solution for implementing a **Binary Search Tree (BST)** in **C++** with **insert, delete, and search operations**.

Step 1: Understanding Binary Search Tree (BST)

A **Binary Search Tree (BST)** is a hierarchical data structure that maintains order properties for efficient searching, insertion, and deletion.

BST Properties:

1. **Each node has at most two children.**
2. **Left subtree** contains values **less than the node's key**.
3. **Right subtree** contains values **greater than the node's key**.
4. **Each subtree is also a BST**.

BST Operations & Their Time Complexity:

- **Insertion:** O(log n) in balanced BST, O(n) in worst case (skewed tree).
- **Deletion:** O(log n) in balanced BST, O(n) in worst case.
- **Search:** O(log n) in balanced BST, O(n) in worst case.

Step 2: Define the BST Node Structure

Each node in the BST contains:

- An **integer value**
- A **pointer to the left child**
- A **pointer to the right child**

BST Node Implementation

```
#include <iostream>

using namespace std;


// Node structure for BST

class Node {

public:

  int key;

  Node* left;

  Node* right;


  Node(int value) {

    key = value;

    left = right = nullptr;

  }

};
```

STEP 3: IMPLEMENT BST INSERT OPERATION

The **insert operation** ensures that:

- If the tree is **empty,** the new node becomes the root.
- If the value is **less than the current node,** insert it into the **left subtree**.
- If the value is **greater than the current node,** insert it into the **right subtree**.

Insert Function

```
class BST {

public:

  Node* root;


  BST() { root = nullptr; }


  // Recursive function to insert a key

  Node* insert(Node* node, int key) {

    if (node == nullptr) {

      return new Node(key);

    }


    if (key < node->key)
```

```
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
  }

  // Public function to call insert
  void insert(int key) {
    root = insert(root, key);
  }
};
```

Step 4: Implement BST Search Operation

The **search operation** follows:

- If the node is **null or matches the key**, return the node.
- If the key is **smaller than the node's key**, search in the **left subtree**.
- If the key is **larger than the node's key**, search in the **right subtree**.

Search Function

```
Node* search(Node* node, int key) {
```

```
    if (node == nullptr || node->key == key)
        return node;

    if (key < node->key)
        return search(node->left, key);

    return search(node->right, key);
}

// Public function to call search
bool search(int key) {
    return search(root, key) != nullptr;
}
```

Step 5: Implement BST Delete Operation

The **delete operation** follows three cases:

1. **Node has no children:** Delete the node.
2. **Node has one child:** Replace the node with its child.
3. **Node has two children:** Replace the node with its **in-order successor** (smallest value in the right subtree).

Delete Function

```
Node* findMin(Node* node) {
  while (node->left != nullptr)
    node = node->left;
  return node;
}

Node* deleteNode(Node* root, int key) {
  if (root == nullptr)
    return root;

  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // Case 1: Node with no children
    if (root->left == nullptr && root->right == nullptr) {
      delete root;
      return nullptr;
    }
    // Case 2: Node with one child
```

```
        else if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Case 3: Node with two children
        Node* temp = findMin(root->right);  // Find in-order successor
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Public function to call deleteNode
void remove(int key) {
    root = deleteNode(root, key);
```

```
}
```

Step 6: Implement Traversal Methods

To verify correctness, we implement **in-order traversal**, which prints nodes in ascending order.

In-Order Traversal Function

```
void inOrder(Node* node) {
   if (node != nullptr) {
      inOrder(node->left);
      cout << node->key << " ";
      inOrder(node->right);
   }
}

// Public function to call in-order traversal
void display() {
   inOrder(root);
   cout << endl;
}
```

Step 7: Testing the BST

We test **insert, search, and delete operations**.

Main Function

```
int main() {
   BST tree;

   // Insert nodes
   tree.insert(50);
   tree.insert(30);
   tree.insert(70);
   tree.insert(20);
   tree.insert(40);
   tree.insert(60);
   tree.insert(80);

   cout << "BST In-Order Traversal: ";
   tree.display(); // Output: 20 30 40 50 60 70 80

   // Search for elements
   cout << "Search 40: " << (tree.search(40) ? "Found" : "Not Found")
<< endl;
   cout << "Search 100: " << (tree.search(100) ? "Found" : "Not
Found") << endl;
```

```
    // Delete node
    tree.remove(50);
    cout << "BST after deleting 50: ";
    tree.display(); // Output: 20 30 40 60 70 80

    return 0;
}
```

Step 8: Expected Output

BST In-Order Traversal: 20 30 40 50 60 70 80

Search 40: Found

Search 100: Not Found

BST after deleting 50: 20 30 40 60 70 80

- **Insertion works** by maintaining BST properties.
- **Search verifies if an element is present.**
- **Deletion removes an element while preserving BST order.**

Step 9: Additional Exercises

1. Implement **pre-order and post-order traversals**.
2. Extend the BST to support **duplicate values**.

3. Optimize the BST using **self-balancing trees** like **AVL** or **Red-Black trees**.
4. Implement an **iterative approach** for search and insert.

Assignment Solution: Develop a Stack-Based Expression Evaluator

This step-by-step guide provides a complete solution for developing a **stack-based expression evaluator** in **C++**. The evaluator will handle **infix, postfix, and prefix expressions** and support *operators such as +, -, , / while following **operator precedence and associativity rules**.

Step 1: Understanding Expression Evaluation

Expression evaluation is a crucial aspect of compilers and calculators. Expressions exist in three formats:

1. **Infix Notation**: Operands are between operators.
    - Example: 3 + 5 * 2
2. **Postfix Notation (Reverse Polish Notation - RPN)**: Operators follow operands.
    - Example: 3 5 2 * +
3. **Prefix Notation (Polish Notation)**: Operators precede operands.
    - Example: + 3 * 5 2

Why Use Stacks?
Stacks efficiently handle operator precedence, associativity, and evaluation order, making them ideal for expression evaluation.

Step 2: Define Operator Precedence and Associativity

Operators have different precedence levels that affect their execution order.

Precedence Table:

Operator	Associativity	Precedence Level
* /	Left	2
+ -	Left	1

Step 3: Convert Infix to Postfix using Stack

We use the **Shunting Yard Algorithm** by Dijkstra to convert infix expressions to postfix.

Algorithm Steps:

1. If the token is an **operand**, add it to the output.
2. If the token is an **operator**:
    - Pop operators from the stack if they have **higher or equal precedence**.
    - Push the current operator onto the stack.
3. If the token is an **opening parenthesis (**, push it onto the stack.
4. If the token is a **closing parenthesis)**, pop from the stack until (is found.
5. After reading all tokens, pop remaining operators from the stack.

Implementation:

```
#include <iostream>
```

```
#include <stack>
#include <cctype>
using namespace std;

// Function to return precedence of operators
int precedence(char op) {
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}

// Convert infix to postfix expression
string infixToPostfix(string infix) {
    stack<char> opStack;
    string postfix = "";

    for (char ch : infix) {
        if (isdigit(ch)) {
            postfix += ch; // Add operand to output
        } else if (ch == '(') {
            opStack.push(ch);
```

```
    } else if (ch == ')') {
      while (!opStack.empty() && opStack.top() != '(') {
        postfix += opStack.top();
        opStack.pop();
      }
      opStack.pop(); // Remove '('
    } else { // Operator encountered
      while (!opStack.empty() && precedence(opStack.top()) >=
precedence(ch)) {
        postfix += opStack.top();
        opStack.pop();
      }
      opStack.push(ch);
    }
  }

  // Pop remaining operators
  while (!opStack.empty()) {
    postfix += opStack.top();
    opStack.pop();
  }
```

```
    return postfix;
}
```

Step 4: Evaluate a Postfix Expression using Stack

Algorithm Steps:

1. **Push operands** onto the stack.
2. When encountering an **operator**, pop two operands, perform the operation, and push the result back onto the stack.
3. The final result is the **last value** in the stack.

Implementation:

```
int evaluatePostfix(string postfix) {
    stack<int> evalStack;

    for (char ch : postfix) {
        if (isdigit(ch)) {
            evalStack.push(ch - '0'); // Convert char to int
        } else { // Operator encountered
            int val2 = evalStack.top(); evalStack.pop();
            int val1 = evalStack.top(); evalStack.pop();
```

```
        switch (ch) {
          case '+': evalStack.push(val1 + val2); break;
          case '-': evalStack.push(val1 - val2); break;
          case '*': evalStack.push(val1 * val2); break;
          case '/': evalStack.push(val1 / val2); break;
        }
      }
    }

  return evalStack.top(); // Final result
}
```

Step 5: Convert Prefix to Postfix

Algorithm:

1. **Reverse** the prefix expression.
2. Convert it to **postfix** using the same stack approach.
3. Reverse the output to get the correct postfix expression.

Implementation:

```
#include <algorithm>

string prefixToPostfix(string prefix) {
```

```
    reverse(prefix.begin(), prefix.end());
    stack<string> stack;

    for (char ch : prefix) {
        if (isdigit(ch)) {
            stack.push(string(1, ch));
        } else {
            string op1 = stack.top(); stack.pop();
            string op2 = stack.top(); stack.pop();
            string newExpr = op1 + op2 + ch;
            stack.push(newExpr);
        }
    }

    return stack.top();
}
```

STEP 6: IMPLEMENT THE MAIN FUNCTION

Main Function to Test Expression Evaluator

```
int main() {
    string infix = "3+5*2";
```

```
    string postfix = infixToPostfix(infix);
    cout << "Infix: " << infix << endl;
    cout << "Postfix: " << postfix << endl;
    cout << "Evaluated Result: " << evaluatePostfix(postfix) << endl;

    string prefix = "+3*52";
    string convertedPostfix = prefixToPostfix(prefix);
    cout << "Prefix to Postfix: " << convertedPostfix << endl;

    return 0;
}
```

STEP 7: EXPECTED OUTPUT

Infix: 3+5*2

Postfix: 352*+

Evaluated Result: 13

Prefix to Postfix: 35*2+

How It Works:

- **Infix 3+5*2 is converted to 352*+ (Postfix).**
- **Postfix 352*+ is evaluated to 13.**
- **Prefix +3*52 is converted to 35*2+.**

Step 8: Additional Enhancements

1. **Support Floating-Point Numbers:** Modify evaluatePostfix to handle decimal numbers.
2. **Add More Operators:** Extend to support ^ (exponentiation) and mod.
3. **Handle Parentheses in Prefix Expressions.**
4. **Implement an Interactive Calculator using Stack.**

Step 9: Additional Exercises

1. Convert the following infix expressions to postfix and evaluate them:
   - (8 + 2 * 5) / (1 + 3 * 2 - 4)
   - 5 * (6 + 2) - 12 / 4
2. Modify the evaluator to handle unary operators (like -5).
3. Implement a function to convert **postfix to infix**.

Conclusion

This guide provides a **complete stack-based expression evaluator,** including: ✅ **Infix to Postfix Conversion**
✅ **Postfix Evaluation**
✅ **Prefix to Postfix Conversion**
✅ **Interactive Testing & Exercises**