**history -c && history -w**

**clear**

**sudo bash**

**sudo chmod 777 /media/**

**Enable**

**chmod 444 /media/**

**Disable**

**G++**

**gedit filename.cpp**

**g++ filename.cpp**

**./a.out**

**1.** Consider a student database of SEIT class (at least 15 records). The database contains different fields of every student like Roll No, Name, and SGPA (array of structure).  
a) Design a roll call list. Arrange the list of students according to roll numbers in ascending order (Use Bubble Sort).  
b) Search for a particular student according to their name using binary search without recursion. (All the student records having the presence of the search key should be displayed).

#include <iostream>

#include <cstring>

using namespace std;

struct Student {

int rollNo;

char name[50];

float sgpa;

};

// Function to sort students by roll number using Bubble Sort

void bubbleSort(Student students[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (students[j].rollNo > students[j + 1].rollNo) {

// Swap students[j] and students[j + 1]

Student temp = students[j];

students[j] = students[j + 1];

students[j + 1] = temp;

}

}

}

}

// Function to perform binary search by name

void binarySearchByName(Student students[], int n, const char key[]) {

int left = 0, right = n - 1;

bool found = false;

while (left <= right) {

int mid = (left + right) / 2;

if (strcmp(students[mid].name, key) == 0) {

// Display the student record

cout << "\nStudent Found:\n";

cout << "Roll No: " << students[mid].rollNo << ", Name: " << students[mid].name << ", SGPA: " << students[mid].sgpa << endl;

found = true;

break;

} else if (strcmp(students[mid].name, key) < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

if (!found) {

cout << "\nStudent with name " << key << " not found!\n";

}

}

int main() {

const int n = 15;

Student students[n];

// Input student details from the user

cout << "Enter details for " << n << " students:\n";

for (int i = 0; i < n; i++) {

cout << "\nEnter Roll No: ";

cin >> students[i].rollNo;

cout << "Enter Name: ";

cin >> students[i].name; // Using cin directly for name input

cout << "Enter SGPA: ";

cin >> students[i].sgpa;

}

// Sort the students by roll number

bubbleSort(students, n);

// Display the sorted list

cout << "\nRoll Call List (Sorted by Roll Number):\n";

for (int i = 0; i < n; i++) {

cout << "Roll No: " << students[i].rollNo << ", Name: " << students[i].name << ", SGPA: " << students[i].sgpa << endl;

}

// Search for a student by name

char searchKey[50];

cout << "\nEnter the name of the student to search: ";

cin >> searchKey; // Using cin directly for name input

binarySearchByName(students, n, searchKey);

return 0;

}

**2.** Consider a student database of SEIT class (at least 15 records). The database contains different fields of every student like Roll No, Name, and SGPA (array of structure).  
a) Arrange the list of students alphabetically (Use Insertion Sort).  
b) Search for students according to SGPA. If more than one student has the same SGPA, then print the list of all students having the same SGPA (Use Linear Search).

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**struct Student {**

**int rollNo;**

**char name[50];**

**float sgpa;**

**};**

**// Function to sort students alphabetically by name using Insertion Sort**

**void insertionSortByName(Student students[], int n) {**

**for (int i = 1; i < n; i++) {**

**Student key = students[i];**

**int j = i - 1;**

**// Compare strings using strcmp**

**while (j >= 0 && strcmp(students[j].name, key.name) > 0) {**

**students[j + 1] = students[j];**

**j--;**

**}**

**students[j + 1] = key;**

**}**

**}**

**// Function to search for students by SGPA using Linear Search**

**void searchBySGPA(Student students[], int n, float targetSGPA) {**

**bool found = false;**

**cout << "\nStudents with SGPA " << targetSGPA << ":\n";**

**for (int i = 0; i < n; i++) {**

**if (students[i].sgpa == targetSGPA) {**

**found = true;**

**cout << "Roll No: " << students[i].rollNo << ", Name: " << students[i].name << ", SGPA: " << students[i].sgpa << endl;**

**}**

**}**

**if (!found) {**

**cout << "No students found with SGPA " << targetSGPA << ".\n";**

**}**

**}**

**int main() {**

**const int n = 15; // Number of students**

**Student students[n];**

**// Input student details from the user**

**cout << "Enter details for " << n << " students:\n";**

**for (int i = 0; i < n; i++) {**

**cout << "\nEnter Roll No: ";**

**cin >> students[i].rollNo;**

**cout << "Enter Name (without spaces): ";**

**cin >> students[i].name; // Using cin directly for name input**

**cout << "Enter SGPA: ";**

**cin >> students[i].sgpa;**

**}**

**// Sort students alphabetically by name**

**insertionSortByName(students, n);**

**// Display the sorted list**

**cout << "\nList of Students Sorted Alphabetically by Name:\n";**

**for (int i = 0; i < n; i++) {**

**cout << "Roll No: " << students[i].rollNo << ", Name: " << students[i].name << ", SGPA: " << students[i].sgpa << endl;**

**}**

**// Search for students by SGPA**

**float targetSGPA;**

**cout << "\nEnter SGPA to search: ";**

**cin >> targetSGPA;**

**searchBySGPA(students, n, targetSGPA);**

**return 0;**

**}**

**3.** Implement In-order Threaded Binary Tree and traverse it in In-order and Pre-order.

**#include <iostream>**

**using namespace std;**

**class Node {**

**public:**

**int data;**

**Node\* left;**

**Node\* right;**

**bool isThreaded;**

**Node(int val) : data(val), left(nullptr), right(nullptr), isThreaded(false) {}**

**};**

**class ThreadedBinaryTree {**

**public:**

**Node\* root;**

**void insert(Node\*& node, int data) {**

**if (!node) node = new Node(data);**

**else if (data < node->data) insert(node->left, data);**

**else insert(node->right, data);**

**}**

**public:**

**ThreadedBinaryTree() : root(nullptr) {}**

**void insert(int data) { insert(root, data); }**

**void createThreads(Node\* node, Node\*& prev) {**

**if (!node) return;**

**createThreads(node->left, prev);**

**if (prev && !prev->right) {**

**prev->right = node;**

**prev->isThreaded = true;**

**}**

**prev = node;**

**createThreads(node->right, prev);**

**}**

**void inOrder() {**

**Node\* current = root;**

**while (current->left) current = current->left;**

**while (current) {**

**cout << current->data << " ";**

**if (current->isThreaded) current = current->right;**

**else {**

**current = current->right;**

**while (current && current->left) current = current->left;**

**}**

**}**

**}**

**void preOrder(Node\* node) {**

**if (!node) return;**

**cout << node->data << " ";**

**preOrder(node->left);**

**if (!node->isThreaded) preOrder(node->right);**

**}**

**void preOrder() { preOrder(root); }**

**};**

**int main() {**

**ThreadedBinaryTree tbt;**

**tbt.insert(10); tbt.insert(5); tbt.insert(20); tbt.insert(3); tbt.insert(7);**

**Node\* prev = nullptr;**

**tbt.createThreads(tbt.root, prev);**

**cout << "In-order Traversal: ";**

**tbt.inOrder();**

**cout << "\nPre-order Traversal: ";**

**tbt.preOrder();**

**return 0;**

**}**

**4.** Implement a stack as an abstract data type using a singly linked list and use this ADT for:  
a) Conversion of infix expression to postfix and prefix.  
b) Evaluation of postfix and prefix expressions.

**#include <iostream>**

**#include <stack>**

**#include <cctype>**

**#include <algorithm> // For reverse()**

**using namespace std;**

**// Node for Stack**

**struct Node {**

**char data;**

**Node\* next;**

**};**

**// Stack Class using Linked List**

**class Stack {**

**Node\* top;**

**public:**

**Stack() { top = nullptr; }**

**void push(char val) {**

**Node\* newNode = new Node{val, top};**

**top = newNode;**

**}**

**char pop() {**

**if (top == nullptr) return '\0';**

**char val = top->data;**

**Node\* temp = top;**

**top = top->next;**

**delete temp;**

**return val;**

**}**

**char peek() {**

**return top ? top->data : '\0';**

**}**

**bool isEmpty() {**

**return top == nullptr;**

**}**

**};**

**// Precedence function for operators**

**int precedence(char op) {**

**if (op == '+' || op == '-') return 1;**

**if (op == '\*' || op == '/') return 2;**

**return 0;**

**}**

**// Convert Infix to Postfix**

**string infixToPostfix(string exp) {**

**Stack s;**

**string result;**

**for (char ch : exp) {**

**if (isalnum(ch)) result += ch;**

**else if (ch == '(') s.push(ch);**

**else if (ch == ')') {**

**while (!s.isEmpty() && s.peek() != '(') result += s.pop();**

**s.pop();**

**} else {**

**while (!s.isEmpty() && precedence(s.peek()) >= precedence(ch)) result += s.pop();**

**s.push(ch);**

**}**

**}**

**while (!s.isEmpty()) result += s.pop();**

**return result;**

**}**

**// Convert Infix to Prefix (reverse method)**

**string infixToPrefix(string exp) {**

**reverse(exp.begin(), exp.end());**

**for (char &ch : exp) {**

**if (ch == '(') ch = ')';**

**else if (ch == ')') ch = '(';**

**}**

**string result = infixToPostfix(exp);**

**reverse(result.begin(), result.end());**

**return result;**

**}**

**// Evaluate Postfix Expression**

**int evaluatePostfix(string exp) {**

**Stack s;**

**for (char ch : exp) {**

**if (isdigit(ch)) s.push(ch - '0');**

**else {**

**int val2 = s.pop(), val1 = s.pop();**

**switch (ch) {**

**case '+': s.push(val1 + val2); break;**

**case '-': s.push(val1 - val2); break;**

**case '\*': s.push(val1 \* val2); break;**

**case '/': s.push(val1 / val2); break;**

**}**

**}**

**}**

**return s.pop();**

**}**

**// Evaluate Prefix Expression**

**int evaluatePrefix(string exp) {**

**reverse(exp.begin(), exp.end());**

**Stack s;**

**for (char ch : exp) {**

**if (isdigit(ch)) s.push(ch - '0');**

**else {**

**int val1 = s.pop(), val2 = s.pop();**

**switch (ch) {**

**case '+': s.push(val1 + val2); break;**

**case '-': s.push(val1 - val2); break;**

**case '\*': s.push(val1 \* val2); break;**

**case '/': s.push(val1 / val2); break;**

**}**

**}**

**}**

**return s.pop();**

**}**

**int main() {**

**string infix;**

**cout << "Enter Infix Expression: ";**

**cin >> infix;**

**string postfix = infixToPostfix(infix);**

**string prefix = infixToPrefix(infix);**

**cout << "Postfix: " << postfix << endl;**

**cout << "Prefix: " << prefix << endl;**

**cout << "Postfix Evaluation: " << evaluatePostfix(postfix) << endl;**

**cout << "Prefix Evaluation: " << evaluatePrefix(prefix) << endl;**

**return 0;**

**}**

**5.** Implement a Circular Queue using an Array. Perform the following operations:  
a) Insertion (Enqueue).  
b) Deletion (Dequeue).  
c) Display.  
(Note: Handle queue full condition by considering a fixed size of the queue).

**#include <iostream>**

**using namespace std;**

**#define SIZE 5 // Fixed size of the queue**

**class CircularQueue {**

**int queue[SIZE]; // Array to store queue elements**

**int front, rear; // Front and rear pointers**

**public:**

**// Constructor to initialize the queue**

**CircularQueue() {**

**front = -1; // Queue is empty initially**

**rear = -1;**

**}**

**// Enqueue function to insert an element into the queue**

**void enqueue(int value) {**

**if ((rear + 1) % SIZE == front) {**

**cout << "Queue is full! Cannot enqueue " << value << endl;**

**return; // Queue is full**

**}**

**if (front == -1) { // If the queue is empty**

**front = 0;**

**}**

**rear = (rear + 1) % SIZE; // Circular increment**

**queue[rear] = value;**

**cout << "Enqueued " << value << endl;**

**}**

**// Dequeue function to remove an element from the queue**

**void dequeue() {**

**if (front == -1) {**

**cout << "Queue is empty! Cannot dequeue" << endl;**

**return; // Queue is empty**

**}**

**cout << "Dequeued " << queue[front] << endl;**

**if (front == rear) { // Only one element was in the queue**

**front = rear = -1; // Reset the queue**

**} else {**

**front = (front + 1) % SIZE; // Circular increment**

**}**

**}**

**// Function to display the queue**

**void display() {**

**if (front == -1) {**

**cout << "Queue is empty!" << endl;**

**return;**

**}**

**cout << "Queue elements: ";**

**int i = front;**

**while (i != rear) {**

**cout << queue[i] << " ";**

**i = (i + 1) % SIZE;**

**}**

**cout << queue[rear] << endl; // Print last element**

**}**

**};**

**int main() {**

**CircularQueue cq;**

**int choice, value;**

**while (true) {**

**cout << "\nCircular Queue Operations:" << endl;**

**cout << "1. Enqueue" << endl;**

**cout << "2. Dequeue" << endl;**

**cout << "3. Display Queue" << endl;**

**cout << "4. Exit" << endl;**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**cout << "Enter value to enqueue: ";**

**cin >> value;**

**cq.enqueue(value);**

**break;**

**case 2:**

**cq.dequeue();**

**break;**

**case 3:**

**cq.display();**

**break;**

**case 4:**

**cout << "Exiting program!" << endl;**

**return 0;**

**default:**

**cout << "Invalid choice! Please enter a valid option." << endl;**

**}**

**}**

**return 0;**

**}**

**6.** Implement a Binary Search Tree (BST) and perform the following operations:  
a) Insert (Handle insertion of duplicate entries).  
b) Search.  
c) Display the tree (Traversal).  
d) Create a copy of the tree.

**#include <iostream>**

**using namespace std;**

**// Node structure**

**struct Node {**

**int data;**

**Node\* left;**

**Node\* right;**

**// Constructor to create a new node**

**Node(int value) {**

**data = value;**

**left = right = nullptr;**

**}**

**};**

**// Function to insert a new node in the BST**

**Node\* insert(Node\* root, int value) {**

**// If the tree is empty, create a new node**

**if (root == nullptr) {**

**return new Node(value);**

**}**

**// Otherwise, recur down the tree**

**if (value < root->data) {**

**root->left = insert(root->left, value);**

**} else if (value > root->data) {**

**root->right = insert(root->right, value);**

**}**

**// Return the unchanged node pointer**

**return root;**

**}**

**// Function to search for a value in the BST**

**bool search(Node\* root, int value) {**

**if (root == nullptr) {**

**return false;**

**}**

**if (root->data == value) {**

**return true;**

**} else if (value < root->data) {**

**return search(root->left, value);**

**} else {**

**return search(root->right, value);**

**}**

**}**

**// Function to perform inorder traversal and display the tree**

**void inorder(Node\* root) {**

**if (root != nullptr) {**

**inorder(root->left);**

**cout << root->data << " ";**

**inorder(root->right);**

**}**

**}**

**// Function to create a copy of the tree (deep copy)**

**Node\* copyTree(Node\* root) {**

**if (root == nullptr) {**

**return nullptr;**

**}**

**Node\* newNode = new Node(root->data);**

**newNode->left = copyTree(root->left);**

**newNode->right = copyTree(root->right);**

**return newNode;**

**}**

**// Function to delete the tree (free memory)**

**void deleteTree(Node\* root) {**

**if (root == nullptr) {**

**return;**

**}**

**deleteTree(root->left);**

**deleteTree(root->right);**

**delete root;**

**}**

**int main() {**

**Node\* root = nullptr;**

**int choice, value;**

**while (true) {**

**// Display menu**

**cout << "\nMenu:\n";**

**cout << "1. Insert value into the tree\n";**

**cout << "2. Search for a value\n";**

**cout << "3. Display tree (Inorder Traversal)\n";**

**cout << "4. Copy the tree\n";**

**cout << "5. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**// Insert value into the tree**

**cout << "Enter value to insert: ";**

**cin >> value;**

**root = insert(root, value);**

**cout << value << " has been inserted.\n";**

**break;**

**case 2:**

**// Search for a value**

**cout << "Enter value to search: ";**

**cin >> value;**

**if (search(root, value)) {**

**cout << value << " is found in the tree.\n";**

**} else {**

**cout << value << " is not found in the tree.\n";**

**}**

**break;**

**case 3:**

**// Display the tree (Inorder Traversal)**

**cout << "Inorder Traversal of the tree: ";**

**inorder(root);**

**cout << endl;**

**break;**

**case 4:**

**// Copy the tree**

**{**

**Node\* copiedTree = copyTree(root);**

**cout << "The tree has been copied successfully.\n";**

**cout << "Inorder Traversal of the copied tree: ";**

**inorder(copiedTree);**

**cout << endl;**

**// Don't forget to delete the copied tree when done**

**deleteTree(copiedTree);**

**}**

**break;**

**case 5:**

**// Exit the program**

**cout << "Exiting the program.\n";**

**deleteTree(root); // Free memory**

**return 0;**

**default:**

**cout << "Invalid choice! Please try again.\n";**

**}**

**}**

**return 0;**

**}**

**7.** Implement a Binary Search Tree (BST) and perform the following operations:  
• Insert (Handle insertion of duplicate entries).  
• Search.  
• Display the tree (Traversal).  
• Display the tree level-wise.

**#include <iostream>**

**#include <queue> // For level-wise display**

**using namespace std;**

**// Node structure**

**struct Node {**

**int data;**

**Node\* left;**

**Node\* right;**

**// Constructor**

**Node(int val) {**

**data = val;**

**left = right = nullptr;**

**}**

**};**

**// Insert function to add a node**

**Node\* insert(Node\* root, int val) {**

**if (!root) return new Node(val); // If tree is empty, create a new node**

**if (val < root->data) {**

**root->left = insert(root->left, val); // Insert to the left**

**} else if (val > root->data) {**

**root->right = insert(root->right, val); // Insert to the right**

**}**

**return root;**

**}**

**// Search function to find a node**

**bool search(Node\* root, int val) {**

**if (!root) return false; // If tree is empty, return false**

**if (root->data == val) return true; // Found the node**

**else if (val < root->data) return search(root->left, val); // Search left**

**else return search(root->right, val); // Search right**

**}**

**// Inorder traversal to display the tree**

**void inorder(Node\* root) {**

**if (root) {**

**inorder(root->left); // Visit left**

**cout << root->data << " "; // Visit root**

**inorder(root->right); // Visit right**

**}**

**}**

**// Level-wise display using Queue (Breadth-First Search)**

**void levelOrder(Node\* root) {**

**if (!root) return;**

**queue<Node\*> q;**

**q.push(root);**

**while (!q.empty()) {**

**Node\* current = q.front();**

**q.pop();**

**cout << current->data << " "; // Print current node's data**

**if (current->left) q.push(current->left); // Add left child to queue**

**if (current->right) q.push(current->right); // Add right child to queue**

**}**

**}**

**// Main function**

**int main() {**

**Node\* root = nullptr;**

**int choice, value;**

**while (true) {**

**// Menu**

**cout << "\n1. Insert a value\n";**

**cout << "2. Search a value\n";**

**cout << "3. Display tree (Inorder Traversal)\n";**

**cout << "4. Display tree Level-wise\n";**

**cout << "5. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1: // Insert value**

**cout << "Enter value to insert: ";**

**cin >> value;**

**root = insert(root, value);**

**cout << value << " has been inserted.\n";**

**break;**

**case 2: // Search value**

**cout << "Enter value to search: ";**

**cin >> value;**

**if (search(root, value)) {**

**cout << value << " is found in the tree.\n";**

**} else {**

**cout << value << " is not found in the tree.\n";**

**}**

**break;**

**case 3: // Inorder Traversal**

**cout << "Inorder Traversal: ";**

**inorder(root);**

**cout << endl;**

**break;**

**case 4: // Level-order Traversal**

**cout << "Level-order Traversal: ";**

**levelOrder(root);**

**cout << endl;**

**break;**

**case 5: // Exit**

**cout << "Exiting the program.\n";**

**return 0;**

**default:**

**cout << "Invalid choice. Please try again.\n";**

**}**

**}**

**return 0;**

**}**

**8.** Implement a Binary Search Tree (BST) and perform the following operations:  
• Insert (Handle insertion of duplicate entries).  
• Search.  
• Display the tree (Traversal).  
• Display the depth of the tree.

**#include <iostream>**

**#include <algorithm> // For max function**

**using namespace std;**

**// Node structure to represent each node in the tree**

**struct Node {**

**int data;**

**Node\* left;**

**Node\* right;**

**Node(int val) {**

**data = val;**

**left = right = nullptr;**

**}**

**};**

**// Function to insert a new node in the BST**

**Node\* insert(Node\* root, int val) {**

**if (root == nullptr) {**

**return new Node(val); // If the tree is empty, create a new node**

**}**

**if (val < root->data) {**

**root->left = insert(root->left, val); // Insert in the left subtree**

**} else if (val > root->data) {**

**root->right = insert(root->right, val); // Insert in the right subtree**

**}**

**return root; // Return the unchanged root**

**}**

**// Function to search a value in the BST**

**bool search(Node\* root, int val) {**

**if (root == nullptr) return false; // If the tree is empty, return false**

**if (root->data == val) return true; // If the node is found**

**else if (val < root->data) return search(root->left, val); // Search in left subtree**

**else return search(root->right, val); // Search in right subtree**

**}**

**// Function for Inorder traversal to display the tree**

**void inorder(Node\* root) {**

**if (root != nullptr) {**

**inorder(root->left); // Visit left child**

**cout << root->data << " "; // Visit root**

**inorder(root->right); // Visit right child**

**}**

**}**

**// Function to calculate the depth (height) of the tree**

**int depth(Node\* root) {**

**if (root == nullptr) return 0; // If tree is empty, depth is 0**

**int leftDepth = depth(root->left); // Get depth of left subtree**

**int rightDepth = depth(root->right); // Get depth of right subtree**

**return max(leftDepth, rightDepth) + 1; // Depth is the max of left and right subtrees plus 1**

**}**

**// Main function to interact with the user**

**int main() {**

**Node\* root = nullptr; // Initialize an empty tree**

**int choice, value;**

**while (true) {**

**// Display menu**

**cout << "\n1. Insert a value\n";**

**cout << "2. Search a value\n";**

**cout << "3. Display tree (Inorder Traversal)\n";**

**cout << "4. Display tree depth\n";**

**cout << "5. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1: // Insert value**

**cout << "Enter value to insert: ";**

**cin >> value;**

**root = insert(root, value);**

**cout << value << " has been inserted.\n";**

**break;**

**case 2: // Search value**

**cout << "Enter value to search: ";**

**cin >> value;**

**if (search(root, value)) {**

**cout << value << " is found in the tree.\n";**

**} else {**

**cout << value << " is not found in the tree.\n";**

**}**

**break;**

**case 3: // Inorder Traversal**

**cout << "Inorder Traversal: ";**

**inorder(root);**

**cout << endl;**

**break;**

**case 4: // Display tree depth**

**cout << "Depth of the tree: " << depth(root) << endl;**

**break;**

**case 5: // Exit**

**cout << "Exiting the program.\n";**

**return 0;**

**default:**

**cout << "Invalid choice. Please try again.\n";**

**}**

**}**

**return 0;**

**}**

**9.** Implement a Binary Search Tree (BST) and perform the following operations:  
• Insert (Handle insertion of duplicate entries).  
• Delete.  
• Display the tree (Traversal).  
• Display the mirror image of the tree.

**#include <iostream>**

**#include <algorithm>**

**using namespace std;**

**// Define the Node structure**

**struct Node {**

**int data;**

**Node\* left;**

**Node\* right;**

**Node(int value) {**

**data = value;**

**left = right = nullptr;**

**}**

**};**

**// Insert a value into the BST (ignoring duplicates)**

**Node\* insert(Node\* root, int value) {**

**if (root == nullptr) {**

**return new Node(value);**

**}**

**if (value < root->data) {**

**root->left = insert(root->left, value);**

**} else if (value > root->data) {**

**root->right = insert(root->right, value);**

**}**

**return root;**

**}**

**// Delete a node from the BST**

**Node\* deleteNode(Node\* root, int value) {**

**if (root == nullptr) return root; // If tree is empty, nothing to delete**

**// If value is smaller than root, go to the left subtree**

**if (value < root->data) {**

**root->left = deleteNode(root->left, value);**

**}**

**// If value is larger than root, go to the right subtree**

**else if (value > root->data) {**

**root->right = deleteNode(root->right, value);**

**}**

**// If value is equal to root, it's the node to be deleted**

**else {**

**// Case 1: Node with only one child or no child**

**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 2: Node with two children: Get the inorder successor (smallest in the right subtree)**

**Node\* temp = root->right;**

**while (temp && temp->left != nullptr) {**

**temp = temp->left;**

**}**

**root->data = temp->data; // Copy the inorder successor's value to this node**

**root->right = deleteNode(root->right, temp->data); // Delete the inorder successor**

**}**

**return root;**

**}**

**// Inorder Traversal (Display the tree)**

**void inorder(Node\* root) {**

**if (root == nullptr) return;**

**inorder(root->left);**

**cout << root->data << " ";**

**inorder(root->right);**

**}**

**// Mirror Image of the Tree**

**void mirror(Node\* root) {**

**if (root == nullptr) return;**

**// Swap the left and right children**

**swap(root->left, root->right);**

**// Recursively do this for left and right subtrees**

**mirror(root->left);**

**mirror(root->right);**

**}**

**// Main function to interact with the user**

**int main() {**

**Node\* root = nullptr; // Start with an empty tree**

**int choice, value;**

**while (true) {**

**// Display menu**

**cout << "\n1. Insert a value\n";**

**cout << "2. Delete a value\n";**

**cout << "3. Display tree (Inorder Traversal)\n";**

**cout << "4. Display mirror image of tree\n";**

**cout << "5. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1: // Insert value**

**cout << "Enter value to insert: ";**

**cin >> value;**

**root = insert(root, value);**

**cout << value << " has been inserted.\n";**

**break;**

**case 2: // Delete value**

**cout << "Enter value to delete: ";**

**cin >> value;**

**root = deleteNode(root, value);**

**cout << value << " has been deleted.\n";**

**break;**

**case 3: // Inorder Traversal**

**cout << "Inorder Traversal: ";**

**inorder(root);**

**cout << endl;**

**break;**

**case 4: // Mirror Image**

**cout << "Mirror Image of the tree: ";**

**mirror(root);**

**inorder(root); // Print the tree after mirroring it**

**cout << endl;**

**break;**

**case 5: // Exit**

**cout << "Exiting the program.\n";**

**return 0;**

**default:**

**cout << "Invalid choice. Please try again.\n";**

**}**

**}**

**return 0;**

**}**

**10.** Construct an Expression Tree from postfix and prefix expressions. Perform:  
• Recursive and non-recursive In-order traversal.  
• Pre-order traversal.  
• Post-order traversal.

#include <iostream>

#include <stack>

#include <string>

using namespace std;

// Node structure for the expression tree

struct Node {

char data;

Node\* left, \* right;

Node(char value) {

data = value;

left = right = nullptr;

}

};

// Check if a character is an operator

bool isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

// Construct Expression Tree from Postfix

Node\* constructPostfixTree(const string& postfix) {

stack<Node\*> s;

for (char c : postfix) {

if (isOperator(c)) {

Node\* node = new Node(c);

node->right = s.top(); s.pop(); // Right child

node->left = s.top(); s.pop(); // Left child

s.push(node);

} else {

s.push(new Node(c));

}

}

return s.top();

}

// Construct Expression Tree from Prefix

Node\* constructPrefixTree(const string& prefix) {

stack<Node\*> s;

for (int i = prefix.length() - 1; i >= 0; i--) {

char c = prefix[i];

if (isOperator(c)) {

Node\* node = new Node(c);

node->left = s.top(); s.pop(); // Left child

node->right = s.top(); s.pop(); // Right child

s.push(node);

} else {

s.push(new Node(c));

}

}

return s.top();

}

// Recursive In-order Traversal (Left, Root, Right)

void inorderRecursive(Node\* root) {

if (!root) return;

inorderRecursive(root->left);

cout << root->data << " ";

inorderRecursive(root->right);

}

// Non-recursive In-order Traversal

void inorderNonRecursive(Node\* root) {

stack<Node\*> s;

Node\* current = root;

while (current != nullptr || !s.empty()) {

while (current != nullptr) {

s.push(current);

current = current->left;

}

current = s.top(); s.pop();

cout << current->data << " ";

current = current->right;

}

}

// Pre-order Traversal (Root, Left, Right)

void preorder(Node\* root) {

if (root) {

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

}

// Post-order Traversal (Left, Right, Root)

void postorder(Node\* root) {

if (root) {

postorder(root->left);

postorder(root->right);

cout << root->data << " ";

}

}

int main() {

string postfix, prefix;

// Input Postfix and Prefix expressions

cout << "Enter Postfix expression: ";

cin >> postfix;

cout << "Enter Prefix expression: ";

cin >> prefix;

// Construct the trees from expressions

Node\* postfixTree = constructPostfixTree(postfix);

Node\* prefixTree = constructPrefixTree(prefix);

// Display the traversals

cout << "\nPostfix Expression Tree Traversals:\n";

cout << "In-order (Recursive): "; inorderRecursive(postfixTree); cout << endl;

cout << "In-order (Non-Recursive): "; inorderNonRecursive(postfixTree); cout << endl;

cout << "Pre-order: "; preorder(postfixTree); cout << endl;

cout << "Post-order: "; postorder(postfixTree); cout << endl;

cout << "\nPrefix Expression Tree Traversals:\n";

cout << "In-order (Recursive): "; inorderRecursive(prefixTree); cout << endl;

cout << "In-order (Non-Recursive): "; inorderNonRecursive(prefixTree); cout << endl;

cout << "Pre-order: "; preorder(prefixTree); cout << endl;

cout << "Post-order: "; postorder(prefixTree); cout << endl;

return 0;

}