1) Write a program to implement stack using arrays

```
#include <iostream>
#define MAX_SIZE 100
Using namespace std;
class Stack {
private:
    int top;
    int stackArray[MAX_SIZE];

public:
    Stack() {
        top = -1;
    }

    void push(int value) {
        if (top == MAX_SIZE - 1) {
            cout << "Stack overflow! Cannot push element " << value <<endl;
            return;
        }
        stackArray[++top] = value;
        cout << "Pushed element " << value << " into the stack." <<endl;
    }

    void pop() {
        if (top == -1) {
           cout << "Stack underflow! Cannot pop element from an empty stack." <<endl;
            return;
        }
        int poppedValue = stackArray[top--];
        cout << "Popped element: " << poppedValue <<endl;
```

```
    }

    int peek() {
        if (top == -1) {
            cout << "Stack is empty!" << endl;
            return -1;
        }
        return stackArray[top];
    }

    bool isEmpty() {
        return (top == -1);
    }
};

int main() {
    Stack myStack;

    myStack.push(5);
    myStack.push(10);
    myStack.push(15);

    cout << "Top element of the stack: " << myStack.peek() <<endl;

    myStack.pop();
    myStack.pop();

    cout << "Is the stack empty? " << (myStack.isEmpty() ? "Yes" : "No") << endl;

    return 0;
```

```
}
```

2) Write a program to evaluate a given a given postfix expression using stacks

```
#include <iostream>
#include <stack>
#include <string>
Using namespace std;

int evaluatePostfixExpression(const string& postfixExpression) {
    stack<int> operandStack;

    for (char ch : postfixExpression) {
        if (isdigit(ch)) {
            operandStack.push(ch - '0');
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            int operand2 = operandStack.top();
            operandStack.pop();
            int operand1 = operandStack.top();
            operandStack.pop();

            int result;
            switch (ch) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
```

```
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
            }
            operandStack.push(result);
        }
    }

    return operandStack.top();
}

int main() {
    string postfixExpression = "82+4*";
    int result = evaluatePostfixExpression(postfixExpression);
    cout << "Result: " << result << std::endl;

    return 0;
}
```

3) Write a program to convert a given infix expression to postfix using form using stacks

```
#include <iostream>
#include <stack>
#include <string>
Using namespace std;
int getPrecedence(char ch) {
    if (ch == '^') {
        return 3;
    }
    else if (ch == '*' || ch == '/') {
        return 2;
    }
    else if (ch == '+' || ch == '-') {
        return 1;
    }
    return 0;
}

string infixToPostfix(const string& infixExpression) {
string postfixExpression;
stack<char> operatorStack;

    for (char ch : infixExpression) {
        if (isalnum(ch)) {
            postfixExpression += ch;
```

```
        }
        else if (ch == '(') {
            operatorStack.push(ch);
        }
        else if (ch == ')') {
            while (!operatorStack.empty() && operatorStack.top() != '(') {
                postfixExpression += operatorStack.top();
                operatorStack.pop();
            }
            operatorStack.pop();
        }
        else {
            while (!operatorStack.empty() && getPrecedence(ch) <= getPrecedence(operatorStack.top())) {
                postfixExpression += operatorStack.top();
                operatorStack.pop();
            }
            operatorStack.push(ch);
        }
    }

    while (!operatorStack.empty()) {
        postfixExpression += operatorStack.top();
        operatorStack.pop();
    }

    return postfixExpression;
}

int main() {
    string infixExpression = "5+3*(2-1)";
```

```
    string postfixExpression = infixToPostfix(infixExpression);
    cout << "Infix Expression: " << infixExpression << std::endl;
    cout << "Postfix Expression: " << postfixExpression << std::endl;

    return 0;
}
```

4) Write a program to implement circular queue using arrays

```
#include <iostream>
#define MAX_SIZE 5
Using namespace std;
class CircularQueue {
private:
    int* queueArray;
    int front;
    int rear;
    int currentSize;
    int maxSize;

public:
    CircularQueue() {
        queueArray = new int[MAX_SIZE];
        front = -1;
        rear = -1;
        currentSize = 0;
        maxSize = MAX_SIZE;
    }

    ~CircularQueue() {
```

```
        delete[] queueArray;
    }

    bool isEmpty() {
        return currentSize == 0;
    }

    bool isFull() {
        return currentSize == maxSize;
    }

    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue overflow! Cannot enqueue element " << value << endl;
            return;
        }
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % maxSize;
        queueArray[rear] = value;
        currentSize++;
        cout << "Enqueued element " << value << endl;
    }

    void dequeue() {
        if (isEmpty()) {
           cout << "Queue underflow! Cannot dequeue element from an empty queue." << endl;
            return;
        }
```

```
        int dequeuedValue = queueArray[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        }
        else {
            front = (front + 1) % maxSize;
        }
        currentSize--;
        cout << "Dequeued element: " << dequeuedValue << endl;
    }

    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        return queueArray[front];
    }
};

int main() {
    CircularQueue myQueue;

    myQueue.enqueue(5);
    myQueue.enqueue(10);
    myQueue.enqueue(15);

    cout << "Front element of the queue: " << myQueue.peek() << endl;
```

```
    myQueue.dequeue();
    myQueue.dequeue();

    cout << "Is the queue empty? " << (myQueue.isEmpty() ? "Yes" : "No") << endl;

    return 0;
}
```

5) Write a program to implement double ended queue (de queue) using arrays.

```
#include <iostream>
#define MAX_SIZE 5
Using namespace std;
class Deque {
private:
    int* dequeArray;
    int front;
    int rear;
    int currentSize;
    int maxSize;

public:
    Deque() {
        dequeArray = new int[MAX_SIZE];
        front = -1;
        rear = -1;
        currentSize = 0;
        maxSize = MAX_SIZE;
    }
```

```
    ~Deque() {
        delete[] dequeArray;
    }

    bool isEmpty() {
        return currentSize == 0;
    }

    bool isFull() {
        return currentSize == maxSize;
    }

    void enqueueFront(int value) {
        if (isFull()) {
            cout << "Deque overflow! Cannot enqueue element " << value << " at the front." << endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
            rear = 0;
        }
        else {
            front = (front - 1 + maxSize) % maxSize;
        }
        dequeArray[front] = value;
        currentSize++;
        cout << "Enqueued element " << value << " at the front." << endl;
    }

    void enqueueRear(int value) {
```

```
        if (isFull()) {
            std::cout << "Deque overflow! Cannot enqueue element " << value << " at the rear." <<
std::endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
            rear = 0;
        }
        else {
            rear = (rear + 1) % maxSize;
        }
        dequeArray[rear] = value;
        currentSize++;
        cout << "Enqueued element " << value << " at the rear." << endl;
    }

    void dequeueFront() {
        if (isEmpty()) {
            cout << "Deque underflow! Cannot dequeue element from the front of an empty deque." << endl;
            return;
        }
        int dequeuedValue = dequeArray[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        }
        else {
            front = (front + 1) % maxSize;
        }
```

```
        currentSize--;
        cout << "Dequeued element from the front: " << dequeuedValue << endl;
    }

    void dequeueRear() {
        if (isEmpty()) {
            cout << "Deque underflow! Cannot dequeue element from the rear of an empty deque." << endl;
            return;
        }
        int dequeuedValue = dequeArray[rear];
        if (front == rear) {
            front = -1;
            rear = -1;
        }
        else {
            rear = (rear - 1 + maxSize) % maxSize;
        }
        currentSize--;
        cout << "Dequeued element from the rear: " << dequeuedValue << endl;
    }

    int getFront() {
        if (isEmpty()) {
            cout << "Deque is empty!" << endl;
            return -1;
        }
        return dequeArray[front];
    }

    int getRear() {
```

```
        if (isEmpty()) {
            cout << "Deque is empty!" << endl;
            return -1;
        }
        return dequeArray[rear];
    }
};

int main() {
    Deque myDeque;

    myDeque.enqueueRear(5);
    myDeque.enqueueRear(10);
    myDeque.enqueueFront(3);

    cout << "Front element of the deque: " << myDeque.getFront() << endl;
    cout << "Rear element of the deque: " << myDeque.getRear() << endl;

    myDeque.dequeueFront();
    myDeque.dequeueRear();

    cout << "Is the deque empty? " << (myDeque.isEmpty() ? "Yes" : "No") <<endl;

    return 0;
}
```

6) Write a program to implement a stack using two queues such that push operation runs in constant time and pop operation runs in linear time.

```
#include <iostream>
```

```cpp
#include <queue>

class Stack {
private:
    queue<int> q1;
    queue<int> q2;

public:
    void push(int value) {
        q1.push(value);
        cout << "Pushed element: " << value << endl;
    }

    void pop() {
        if (q1.empty()) {
            cout << "Stack underflow! Cannot perform pop operation on an empty stack." << endl;
            return;
        }

        // Move elements from q1 to q2 except the last element
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }

        // Pop the last element from q1
        int poppedValue = q1.front();
        q1.pop();

        // Swap the queues
```

```
        queue<int> temp = q1;
        q1 = q2;
        q2 = temp;

        cout << "Popped element: " << poppedValue << std::endl;
    }

    int top() {
        if (q1.empty()) {
            cout << "Stack is empty!" << endl;
            return -1;
        }

        // Move elements from q1 to q2 except the last element
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }

        // Get the last element from q1
        int topValue = q1.front();

        // Move the element to q2
        q2.push(topValue);

        // Swap the queues
        queue<int> temp = q1;
        q1 = q2;
        q2 = temp;
```

```
        return topValue;
    }

    bool isEmpty() {
        return q1.empty();
    }
};

int main() {
    Stack myStack;

    myStack.push(5);
    myStack.push(10);
    myStack.push(15);

    cout << "Top element of the stack: " << myStack.top() << endl;

    myStack.pop();
    myStack.pop();

    cout << "Is the stack empty? " << (myStack.isEmpty() ? "Yes" : "No") << endl;

    return 0;
}
```

7) Write a program to implement stack using two queues such that the push operation runs in linear time and the pop operation runs in constant time.

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

```
Using namespace std;
class Stack {
private:
    queue<int> q1, q2;

public:
    // Push operation
    void push(int data) {
        // Enqueue the new element into q1
        q1.push(data);
    }

    // Pop operation
    int pop() {
        if (q1.empty()) {
            cout << "Stack is empty." << endl;
            return -1; // Assuming -1 represents an empty stack
        }

        // Transfer (n-1) elements from q1 to q2
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }

        // Dequeue and return the last element in q1 (top of the stack)
        int poppedElement = q1.front();
        q1.pop();

        // Swap the names of q1 and q2
```

```
        swap(q1, q2);

        return poppedElement;
    }

    // Check if the stack is empty
    bool isEmpty() {
        return q1.empty();
    }
};

int main() {
    Stack stack;

    stack.push(5);
    stack.push(10);
    stack.push(15);

    cout << stack.pop() << endl; // Output: 15
    cout << stack.pop() << endl; // Output: 10
    cout << stack.pop() << endl; // Output: 5

    cout << boolalpha << stack.isEmpty() << endl; // Output: true

    return 0;
}
```

8) Write a program to implement a queue using two stacks such that the enqueue operation runs in linear time and the pop operation runs in a constant time.

```
#include <iostream>
#include <stack>
Using namespace std;
class Queue {
private:
    stack<int> s1, s2;

public:
    // Enqueue operation
    void enqueue(int data) {
        // Push the new element onto s1
        s1.push(data);
    }

    // Pop operation
    int dequeue() {
        if (s1.empty() && s2.empty()) {
            cout << "Queue is empty." << endl;
            return -1; // Assuming -1 represents an empty queue
        }

        if (s2.empty()) {
            // Transfer elements from s1 to s2 in reverse order
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }

        // Pop the top element from s2 (front of the queue)
```

```
        int poppedElement = s2.top();
        s2.pop();

        return poppedElement;
    }

    // Check if the queue is empty
    bool isEmpty() {
        return s1.empty() && s2.empty();
    }
};

int main() {
    Queue queue;

    queue.enqueue(5);
    queue.enqueue(10);
    queue.enqueue(15);

    cout << queue.dequeue() << endl; // Output: 5
    cout << queue.dequeue() << endl; // Output: 10
    cout << queue.dequeue() << endl; // Output: 15

    cout << boolalpha << queue.isEmpty() << endl; // Output: true

    return 0;
}
```

9) Write a program to implement a queue using two stacks such that the enqueue operation runs in linear time and the dequeue operation runs in constant time.

10) Write a program to implement the following operation
-Single linked list
-Double linked list

```
#include <iostream>
Using namespace std;
// Node structure for single linked list
struct Node {
    int data;
    Node* next;
};

// Class for single linked list
class SingleLinkedList {
private:
    Node* head;

public:
    SingleLinkedList() {
        head = NULL;
    }

    // Insert a new node at the beginning of the list
    void insert(int data) {
        Node* newNode = new Node();
        newNode->data = data;
        newNode->next = head;
        head = newNode;
    }
```

```
    // Display the contents of the list
    void display() {
        Node* current = head;
        while (current != NULL) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

// Node structure for double linked list
struct DNode {
    int data;
    DNode* prev;
    DNode* next;
};

// Class for double linked list
class DoubleLinkedList {
private:
    DNode* head;

public:
    DoubleLinkedList() {
        head = NULL;
    }

    // Insert a new node at the beginning of the list
    void insert(int data) {
```

```
        DNode* newNode = new DNode();
        newNode->data = data;
        newNode->prev = NULL;
        newNode->next = head;

        if (head != NULL) {
            head->prev = newNode;
        }

        head = newNode;
    }

    // Display the contents of the list
    void display() {
        DNode* current = head;
        while (current != NULL) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    // Single linked list
    SingleLinkedList sList;
    sList.insert(3);
    sList.insert(7);
    sList.insert(9);
    sList.display();  // Output: 9 7 3
```

```
    // Double linked list
    DoubleLinkedList dList;
    dList.insert(5);
    dList.insert(2);
    dList.insert(8);
    dList.display();  // Output: 8 2 5

    return 0;
}
```

11) Write a program to implement a stack using linked list operation such as push and pop operations

```
#include <iostream>
Using namespace std;
// Node structure for stack
struct Node {
    int data;
    Node* next;
};

// Class for stack
class Stack {
private:
    Node* top;

public:
    Stack() {
        top = NULL;
    }
```

```
// Push an element onto the stack
void push(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = top;
    top = newNode;
}

// Pop an element from the stack
void pop() {
    if (isEmpty()) {
        cout << "Stack is empty. Cannot pop." << endl;
    } else {
        Node* temp = top;
        top = top->next;
        delete temp;
    }
}

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

// Display the top element of the stack
void displayTop() {
    if (isEmpty()) {
        cout << "Stack is empty." << endl;
    } else {
```

```
            cout << "Top element: " << top->data << endl;
        }
    }
};

int main() {
    Stack stack;

    stack.push(5);
    stack.push(10);
    stack.push(15);
    stack.displayTop();  // Output: Top element: 15

    stack.pop();
    stack.displayTop();  // Output: Top element: 10

    stack.pop();
    stack.displayTop();  // Output: Top element: 5

    stack.pop();
    stack.displayTop();  // Output: Stack is empty.

    return 0;
}
```

12) Write a program to implement a queue using a linked list such that enqueue and dequeue operations

```
#include <iostream>
Using namespace std;
// Node structure for queue
```

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

// Class for queue
class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() {
        front = NULL;
        rear = NULL;
    }

    // Enqueue an element into the queue
    void enqueue(int data) {
        Node* newNode = new Node();
        newNode->data = data;
        newNode->next = NULL;

        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
```

```
    }

    // Dequeue an element from the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue." << endl;
        } else {
            Node* temp = front;
            front = front->next;

            if (front == NULL) {
                rear = NULL;
            }

            delete temp;
        }
    }

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

    // Display the front element of the queue
    void displayFront() {
        if (isEmpty()) {
            cout << "Queue is empty." << endl;
        } else {
            cout << "Front element: " << front->data << endl;
        }
```

```
    }
};

int main() {
    Queue queue;

    queue.enqueue(5);
    queue.enqueue(10);
    queue.enqueue(15);
    queue.displayFront();  // Output: Front element: 5

    queue.dequeue();
    queue.displayFront();  // Output: Front element: 10

    queue.dequeue();
    queue.displayFront();  // Output: Front element: 15

    queue.dequeue();
    queue.displayFront();  // Output: Queue is empty.

    return 0;
}
```

13) Write a program to create a binary search tree(BST) by considering the keys in given order and perform the following operations on it. (a) Minimum key (b) Maximum key (c) Search for a given key (d) Find predecessor of anode (e) Find successor of anode (f) delete a node with given key

```
#include <iostream>

// Node structure for BST
```

```
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int value) {
        key = value;
        left = NULL;
        right = NULL;
    }
};

// Class for BST
class BinarySearchTree {
private:
    Node* root;

    // Helper function to insert a node recursively
    Node* insertRecursive(Node* root, int key) {
        if (root == NULL) {
            return new Node(key);
        }

        if (key < root->key) {
            root->left = insertRecursive(root->left, key);
        } else if (key > root->key) {
            root->right = insertRecursive(root->right, key);
        }

        return root;
```

```
}

// Helper function to find the minimum key recursively
Node* findMinRecursive(Node* root) {
    if (root == NULL || root->left == NULL) {
        return root;
    }

    return findMinRecursive(root->left);
}

// Helper function to find the maximum key recursively
Node* findMaxRecursive(Node* root) {
    if (root == NULL || root->right == NULL) {
        return root;
    }

    return findMaxRecursive(root->right);
}

// Helper function to search for a given key recursively
Node* searchRecursive(Node* root, int key) {
    if (root == NULL || root->key == key) {
        return root;
    }

    if (key < root->key) {
        return searchRecursive(root->left, key);
    }
```

```
    return searchRecursive(root->right, key);
}

// Helper function to find the predecessor of a node recursively
Node* findPredecessorRecursive(Node* root, int key) {
    Node* current = searchRecursive(root, key);

    if (current == NULL) {
        return NULL;
    }

    if (current->left != NULL) {
        return findMaxRecursive(current->left);
    } else {
        Node* predecessor = NULL;
        Node* ancestor = root;

        while (ancestor != current) {
            if (current->key > ancestor->key) {
                predecessor = ancestor;
                ancestor = ancestor->right;
            } else {
                ancestor = ancestor->left;
            }
        }

        return predecessor;
    }
}
```

```
// Helper function to find the successor of a node recursively
Node* findSuccessorRecursive(Node* root, int key) {
    Node* current = searchRecursive(root, key);

    if (current == NULL) {
        return NULL;
    }

    if (current->right != NULL) {
        return findMinRecursive(current->right);
    } else {
        Node* successor = NULL;
        Node* ancestor = root;

        while (ancestor != current) {
            if (current->key < ancestor->key) {
                successor = ancestor;
                ancestor = ancestor->left;
            } else {
                ancestor = ancestor->right;
            }
        }

        return successor;
    }
}

// Helper function to delete a node with a given key recursively
Node* deleteNodeRecursive(Node* root, int key) {
    if (root == NULL) {
```

```
            return NULL;
        }

        if (key < root->key) {
            root->left = deleteNodeRecursive(root->left, key);
        } else if (key > root->key) {
            root->right = deleteNodeRecursive(root->right, key);
        } else {
            if (root->left == NULL) {
                Node* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == NULL) {
                Node* temp = root->left;
                delete root;
                return temp;
            }

            Node* successor = findMinRecursive(root->right);
            root->key = successor->key;
            root->right = deleteNodeRecursive(root->right, successor->key);
        }

        return root;
    }

public:
    BinarySearchTree() {
        root = nullptr;
    }
```

```
// Insert a key into the BST
void insert(int key) {
    root = insertRecursive(root, key);
}

// Find the minimum key in the BST
int findMin() {
    Node* minNode = findMinRecursive(root);
    if (minNode != NULL) {
        return minNode->key;
    }
    return -1; // Assuming -1 represents an empty tree
}

// Find the maximum key in the BST
int findMax() {
    Node* maxNode = findMaxRecursive(root);
    if (maxNode != NULL) {
        return maxNode->key;
    }
    return -1; // Assuming -1 represents an empty tree
}

// Search for a given key in the BST
bool search(int key) {
    Node* foundNode = searchRecursive(root, key);
    return foundNode != NULL;
}
```

```
    // Find the predecessor of a node with a given key
    int findPredecessor(int key) {
        Node* predecessor = findPredecessorRecursive(root, key);
        if (predecessor != NULL) {
            return predecessor->key;
        }
        return -1; // Assuming -1 represents no predecessor
    }

    // Find the successor of a node with a given key
    int findSuccessor(int key) {
        Node* successor = findSuccessorRecursive(root, key);
        if (successor != NULL) {
            return successor->key;
        }
        return -1; // Assuming -1 represents no successor
    }

    // Delete a node with a given key from the BST
    void deleteNode(int key) {
        root = deleteNodeRecursive(root, key);
    }
};

int main() {
    BinarySearchTree bst;

    // Create the BST by considering the keys in a given order
    bst.insert(50);
    bst.insert(30);
```

```
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    cout << "Minimum key: " << bst.findMin() << endl;  // Output: Minimum key: 20
    cout << "Maximum key: " << bst.findMax() << endl;  // Output: Maximum key: 80

    int searchKey = 60;
    if (bst.search(searchKey)) {
        cout << searchKey << " found in the BST." << endl;  // Output: 60 found in the BST.
    } else {
        cout << searchKey << " not found in the BST." << endl;
    }

    int predecessorKey = 50;
    int predecessor = bst.findPredecessor(predecessorKey);
    if (predecessor != -1) {
        cout << "Predecessor of " << predecessorKey << ": " << predecessor << endl;  // Output:
Predecessor of 50: 40
    } else {
        cout << "No predecessor found for " << predecessorKey << endl;
    }

    int successorKey = 30;
    int successor = bst.findSuccessor(successorKey);
    if (successor != -1) {
        cout << "Successor of " << successorKey << ": " << successor << endl;  // Output: Successor of
30: 40
```

```
    } else {
        cout << "No successor found for " << successorKey << endl;
    }

    int deleteKey = 30;
    bst.deleteNode(deleteKey);
    cout << "Deleted node with key " << deleteKey << endl;

    return 0;
}
```

14) Write a program to construct an AVL tree for the given set of keys. Also write a function for deleting a key from a given AVL tree.

```
#include <iostream>

// Node structure for AVL tree
struct Node {
    int key;
    int height;
    Node* left;
    Node* right;

    Node(int value) {
        key = value;
        height = 1;
        left = nullptr;
        right = nullptr;
    }
};
```

```
// Function to get the height of a node
int getHeight(Node* node) {
    if (node == nullptr) {
        return 0;
    }
    return node->height;
}

// Function to get the balance factor of a node
int getBalanceFactor(Node* node) {
    if (node == nullptr) {
        return 0;
    }
    return getHeight(node->left) - getHeight(node->right);
}

// Function to update the height of a node
void updateHeight(Node* node) {
    if (node == nullptr) {
        return;
    }
    node->height = max(getHeight(node->left), getHeight(node->right)) + 1;
}

// Function to perform a right rotation
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
```

```
    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    updateHeight(y);
    updateHeight(x);

    return x;
}

// Function to perform a left rotation
Node* rotateLeft(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    updateHeight(x);
    updateHeight(y);

    return y;
}

// Function to insert a key into the AVL tree
Node* insert(Node* root, int key) {
    // Perform standard BST insertion
```

```
    if (root == nullptr) {
        return new Node(key);
    }

    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    } else {
        // Duplicate keys are not allowed in AVL tree
        return root;
    }

    // Update the height of the current node
    updateHeight(root);

    // Check the balance factor and perform rotations if necessary
    int balanceFactor = getBalanceFactor(root);

    // Left Left case
    if (balanceFactor > 1 && key < root->left->key) {
        return rotateRight(root);
    }

    // Right Right case
    if (balanceFactor < -1 && key > root->right->key) {
        return rotateLeft(root);
    }

    // Left Right case
```

```
    if (balanceFactor > 1 && key > root->left->key) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }

    // Right Left case
    if (balanceFactor < -1 && key < root->right->key) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }

    return root;
}

// Function to find the node with the minimum key value in an AVL tree
Node* findMinNode(Node* root) {
    if (root == nullptr || root->left == nullptr) {
        return root;
    }
    return findMinNode(root->left);
}

// Function to delete a key from the AVL tree
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 {
        // Node to be deleted found

        // Node with only one child or no child
        if (root->left == nullptr || root->right == nullptr) {
            Node* temp = root->left ? root->left : root->right;

            // No child case
            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else { // One child case
                *root = *temp; // Copy the contents of the non-empty child
            }

            delete temp;
        } else {
            // Node with two children
            Node* temp = findMinNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node, return
    if (root == nullptr) {
        return root;
    }
```

```
    // Update the height of the current node
    updateHeight(root);

    // Check the balance factor and perform rotations if necessary
    int balanceFactor = getBalanceFactor(root);

    // Left Left case
    if (balanceFactor > 1 && getBalanceFactor(root->left) >= 0) {
        return rotateRight(root);
    }

    // Left Right case
    if (balanceFactor > 1 && getBalanceFactor(root->left) < 0) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }

    // Right Right case
    if (balanceFactor < -1 && getBalanceFactor(root->right) <= 0) {
        return rotateLeft(root);
    }

    // Right Left case
    if (balanceFactor < -1 && getBalanceFactor(root->right) > 0) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }

    return root;
```

```
}

// Function to print the AVL tree in in-order traversal
void printInOrder(Node* root) {
    if (root != nullptr) {
        printInOrder(root->left);
        std::cout << root->key << " ";
        printInOrder(root->right);
    }
}

int main() {
    Node* root = nullptr;

    // Construct the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    std::cout << "AVL tree (in-order traversal): ";
    printInOrder(root);
    std::cout << std::endl;

    // Delete a key from the AVL tree
    root = deleteNode(root, 30);

    std::cout << "AVL tree after deleting key 30 (in-order traversal): ";
```

```
    printInOrder(root);
    std::cout << std::endl;

    return 0;
}
```

15) Write a program to implement hashing with a) Separate chaining and b) Open addressing methods.

```
#include <iostream>
#include <list>
#include <vector>

// Separate Chaining Hash Table
class SeparateChainingHash {
private:
    int tableSize;
    vector<list<int>> table;

    // Hash function to map a key to an index
    int hashFunction(int key) {
        return key % tableSize;
    }

public:
    SeparateChainingHash(int size) {
        tableSize = size;
        table.resize(tableSize);
    }

    // Insert a key into the hash table
```

```
    void insert(int key) {
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    // Search for a key in the hash table
    bool search(int key) {
        int index = hashFunction(key);
        for (int k : table[index]) {
            if (k == key) {
                return true;
            }
        }
        return false;
    }

    // Delete a key from the hash table
    void remove(int key) {
        int index = hashFunction(key);
        table[index].remove(key);
    }
};

// Open Addressing Hash Table
class OpenAddressingHash {
private:
    int tableSize;
    vector<int> table;

    // Hash function to map a key to an index
```

```
    int hashFunction(int key) {
        return key % tableSize;
    }

public:
    OpenAddressingHash(int size) {
        tableSize = size;
        table.resize(tableSize, -1);
    }

    // Insert a key into the hash table
    void insert(int key) {
        int index = hashFunction(key);

        while (table[index] != -1) {
            index = (index + 1) % tableSize;
        }

        table[index] = key;
    }

    // Search for a key in the hash table
    bool search(int key) {
        int index = hashFunction(key);

        int count = 0;
        while (table[index] != -1 && count < tableSize) {
            if (table[index] == key) {
                return true;
            }
```

```
            index = (index + 1) % tableSize;
            count++;
        }

        return false;
    }

    // Delete a key from the hash table
    void remove(int key) {
        int index = hashFunction(key);

        int count = 0;
        while (table[index] != -1 && count < tableSize) {
            if (table[index] == key) {
                table[index] = -1;
                return;
            }
            index = (index + 1) % tableSize;
            count++;
        }
    }
};

int main() {
    // Separate Chaining Hashing
    SeparateChainingHash separateChainingHash(10);

    separateChainingHash.insert(5);
    separateChainingHash.insert(15);
    separateChainingHash.insert(25);
```

```
    cout << "Separate Chaining Hashing:" << endl;
    cout << "5 is " << (separateChainingHash.search(5) ? "found" : "not found") << endl;
    cout << "10 is " << (separateChainingHash.search(10) ? "found" : "not found") << endl;

    separateChainingHash.remove(15);

    cout << "After removing 15, 15 is " << (separateChainingHash.search(15) ? "found" : "not found") <<
endl;

  cout << endl;

    // Open Addressing Hashing
    OpenAddressingHash openAddressingHash(10);

    openAddressingHash.insert(5);
    openAddressingHash.insert(15);
    openAddressingHash.insert(25);

    cout << "Open Addressing Hashing:" << std::endl;
    cout << "5 is " << (openAddressingHash.search(5) ? "found" : "not found") << endl;
    cout << "10 is " << (openAddressingHash.search(10) ? "found" : "not found") << endl;

    openAddressingHash.remove(15);

    cout << "After removing 15, 15 is " << (openAddressingHash.search(15) ? "found" : "not found") <<
endl;

    return 0;
}
```

Implementation of linked lists

```
#include <iostream>

// Node structure for the linked list
struct Node {
    int data;
    Node* next;

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

// Linked list class
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }

    // Insert a new node at the beginning of the linked list
    void insert(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
```

```
        head = newNode;
    }

    // Delete a node with a given value from the linked list
    void remove(int value) {
        if (head == nullptr) {
            return;
        }

        // Check if the value to be removed is at the head of the list
        if (head->data == value) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        Node* curr = head;
        Node* prev = nullptr;

        // Traverse the list to find the node to be deleted
        while (curr != nullptr && curr->data != value) {
            prev = curr;
            curr = curr->next;
        }

        // If the value is found, remove the node
        if (curr != nullptr) {
            prev->next = curr->next;
            delete curr;
```

```
        }
    }

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

int main() {
    LinkedList myList;

    // Insert nodes into the linked list
    myList.insert(10);
    myList.insert(20);
    myList.insert(30);
    myList.insert(40);

    // Print the linked list
    std::cout << "Linked List: ";
    myList.print();

    // Delete a node from the linked list
    myList.remove(20);
```

```
    // Print the updated linked list
    std::cout << "Updated Linked List: ";
    myList.print();

    return 0;
}
```

Implementation of stack

```
#include <iostream>

#define MAX_SIZE 100

// Stack class
class Stack {
private:
    int top;
    int stackArray[MAX_SIZE];

public:
    Stack() {
        top = -1;
    }

    // Push an element onto the stack
    void push(int value) {
        if (top == MAX_SIZE - 1) {
            std::cout << "Stack overflow! Cannot push element onto the stack." << std::endl;
            return;
        }
        stackArray[++top] = value;
```

```
    }

    // Pop an element from the stack
    void pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow! Cannot pop element from the stack." << std::endl;
            return;
        }
        --top;
    }

    // Get the top element of the stack
    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty! Cannot retrieve top element." << std::endl;
            return -1;
        }
        return stackArray[top];
    }

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

    // Print the stack
    void print() {
        if (isEmpty()) {
            std::cout << "Stack is empty!" << std::endl;
            return;
```

```
        }
        std::cout << "Stack: ";
        for (int i = top; i >= 0; --i) {
            std::cout << stackArray[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    Stack myStack;

    // Push elements onto the stack
    myStack.push(10);
    myStack.push(20);
    myStack.push(30);
    myStack.push(40);

    // Print the stack
    myStack.print();

    // Pop an element from the stack
    myStack.pop();

    // Print the updated stack
    myStack.print();

    // Get the top element of the stack
    int topElement = myStack.peek();
    std::cout << "Top element of the stack: " << topElement << std::endl;
```

```
    return 0;
}
```