```cpp
#include <iostream>
#define MAX 5;
using namespace std;
struct Node
{
    int data;
    Node *next;
};
class linkedlistQueue
{
public:
    Node *front = nullptr;
    Node *rear = nullptr;
    void enqueue(int value)
    {
        Node *newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;
        if (front == nullptr)
        {
            front = rear = newNode;
        }
        else
        {
            rear->next = newNode;
            rear = newNode;
        }
    }
    void dequeue()
    {
        if (front == nullptr)
        {
            cout << "Queue is empty" << endl;
            return;
        }
        else if (front == rear)
        {
            Node *temp = front;
            front = rear = nullptr;
            delete temp;
```

```
        }
        else
        {
            Node *temp = front;
            front = front->next;
            cout << "Dequeued element is: " << temp->data <<
endl;
            delete temp;
        }
    }

    void peek()
    {
        if (front == nullptr)
        {
            cout << "Queue is empty" << endl;
            return;
        }
        cout << "Peek element is: " << front->data << endl;
    }

    bool isEmpty()
    {
        return (front == nullptr);
    }

    void display()
    {
        Node *current = front;
        while (current != nullptr)
        {
            cout << current->data;
            current = current->next;
        }
        cout << endl;
    }
};
class simpleQueue
{
public:
```

```
int queue[5];
int front = -1;
int rear = -1;
bool empty()
{
    return (front == -1) || front > rear;
}
bool full()
{
    return rear = 5 - 1 && front == 0;
}
void enqueue(int value)
{
    if (rear == 5 - 1 && front == 1)
    {
        cout << "Queue is full" << endl;
        return;
    }
    else
    {
        if (front == -1)
        {
            front = 0;
            queue[++rear] = value;
        }
        else
        {
            queue[++rear] = value;
        }
    }
}
void dequeue()
{
    if (front == -1 || front > rear)
    {
        cout << "Queue is empty" << endl;
        return;
    }
    else
    {
```

```
            front++;
        }
    }
};
class circularQueue
{
public:
    int queue[5];
    int front = -1;
    int rear = -1;
    bool isEmpty()
    {
        return front == -1;
    }
    bool isFull()
    {
        return (rear == 5 - 1 && front == 0) || (front == rear +
1);
    }
    void enqueue(int value)
    {
        if (isFull())
        {
            cout << "Queue is full" << endl;
            return;
        }
        else if (rear == 5 - 1)
        {
            rear = 0;
            queue[rear] = value;
        }
        else if (front == -1)
        {
            front = 0;
            rear = 0;
            queue[rear] = value;
        }
        else
        {
            queue[++rear] = value;
```

```
        }
    }
    void dequeue()
    {
        if (isEmpty())
        {
            cout << "Queue is empty" << endl;
            return;
        }
        else if (front == 5 - 1)
        {

            front = 0;
        }
        else
        {
            front++;
        }
    }
};
class linkedListStack
{
public:
    Node *head == nullptr;

    void push(int value)
    {
        Node *newNode = new Node();
        newNode->data = value;
        cout << "Created new node with value: " << value << endl;

        newNode->next = head;

        cout << "New node's next set to point to the current
head." << endl;

        head = newNode;
        cout << value << " pushed onto the stack. Head updated to
the new node." << endl;
    }
```

```
    void display()
    {
        Node *temp = head;
        cout << "Stack elements: ";
        while (temp != nullptr)
        {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
    void pop()
    {
        if (head == nullptr)
        {
            cout << "Stack is empty, nothing to pop." << endl;
            return;
        }

        Node *temp = head;

        head = head->next;
        cout << temp->data << " popped from the stack." << endl;

        delete temp;
    }
};
class Stack
{
private:
    int arr[MAX];
    int top;

public:
    Stack()
    {
        top = -1;
    }
    void push(int value)
```

```
    {
        if (top >= MAX - 1)
        {
            std::cout << "Stack Overflow! Cannot push " << value
<< ".\n";
        }
        else
        {
            top++;
            arr[top] = value;
            std::cout << value << " pushed to stack.\n";
        }
    }
    void pop()
    {
        if (top < 0)
        {
            std::cout << "Stack Underflow! Cannot pop.\n";
        }
        else
        {
            std::cout << arr[top] << " popped from stack.\n";
            top--;
        }
    }
    void displayStack()
    {
        if (top < 0)
        {
            std::cout << "Stack is empty.\n";
        }
        else
        {
            std::cout << "Stack elements are: ";
            for (int i = 0; i <= top; i++)
            {
                std::cout << arr[i] << " ";
            }
            std::cout << std::endl;
        }
```

```
    }
    int getTop()
    {
        if (top < 0)
        {
            std::cout << "Stack is empty.\n";
            return -1;
        }
        else
        {
            return arr[top];
        }
    }
    bool isEmpty()
    {
        return (top < 0);
    }
};
```

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

using namespace std;

struct Node
{
    int data;
    Node *right;
    Node *left;

    Node(int data) : data(data), right(nullptr), left(nullptr)
{};
};

class BST
{
public:
    Node *root;
    BST() : root(nullptr) {};
```

```
    bool search(int value);
    void insert(int value);
    bool isEmpty();
    Node *findMin(Node *root);
    int findHight(Node *root);
    void levelOrder(Node *root);
    void preOrder(Node *root);
    void inOrder(Node *root);
    void inOrder(Node *root, queue<int> &q);
    void postOrder(Node *root);
    Node *deleteNode(Node *root, int data);
};

bool BST::isEmpty()
{
    if (root == nullptr)
    {
        return true;
    }
    return false;
}

void BST::insert(int value)
{
    Node *newNode = new Node(value);
    if (isEmpty())
    {
        root = newNode;
    }
    else
    {
        Node *current = root;
        while (current)
        {
            if (value > current->data)
            {
                if (current->right == nullptr)
                {
                    current->right = newNode;
```

```
                    return;
                };
                current = current->right;
            }
            else if (value < current->data)
            {
                if (current->left == nullptr)
                {
                    current->left = newNode;
                    return;
                };
                current = current->left;
            }
        }
    }
}

bool BST::search(int value)
{

    Node *current = root;
    while (current)
    {
        if (value > current->data)
        {
            current = current->right;
        }
        else if (value < current->data)
        {
            current = current->left;
        }
        else if (value == current->data)
        {
            return true;
        }
    }
    return false;
};

Node *BST::findMin(Node *root)
```

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

int BST::findHight(Node *root)
{
    if (root == nullptr)
        return -1;

    return max(findHight(root->left), findHight(root->right)) +
1;
}

void BST::levelOrder(Node *root)
{
    queue<Node *> q;
    q.push(root);
    while (!q.empty())
    {
        Node *current = q.front();
        cout << current->data << " ";
        if (current->left)
            q.push(current->left);
        if (current->right)
            q.push(current->right);
        q.pop();
    }
}

void BST::preOrder(Node *root)
{
    if (root == nullptr)
        return;
    cout << root->data << " ";
    preOrder(root->left);
```

```
    preOrder(root->right);
}

void BST::inOrder(Node *root)
{
    if (root == nullptr)
        return;
    inOrder(root->left);
    cout << root->data << ' ';
    inOrder(root->right);
}

void BST::inOrder(Node *root, queue<int> &q)
{
    if (root == nullptr)
        return;
    inOrder(root->left, q);
    q.push(root->data);
    inOrder(root->right, q);
}

void BST::postOrder(Node *root)
{
    if (root == nullptr)
        return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->data << ' ';
}

Node *BST::deleteNode(Node *root, int data)
{
    if (root == nullptr)
        return root;
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else if (data < root->data)
        root->left = deleteNode(root->left, data);
    else
    {
```

```
        if (!root->left && !root->right)
        {

            delete root;
            root = nullptr;
        }
        else if (!root->left)
        {
            Node *temp = root;
            root = root->right;
            delete temp;
        }
        else if (!root->right)
        {
            Node *temp = root;
            root = root->left;
            delete temp;
        }
        else
        {
            Node *temp = findMin(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    };
    return root;
};

int main()
{
    BST tree;
    queue<int> q;
    tree.insert(50);
    tree.insert(40);
    tree.insert(60);
    tree.insert(35);
    tree.insert(45);
    tree.insert(55);
    tree.insert(65);
    tree.inOrder(tree.root, q);
```

```
    cout << q.size() << endl;
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }
}
```

```
#include <iostream>
#define MAX_VERTICES 100
using namespace std;

class Node
{
public:
    int vertex;
    Node *next;

    Node(int v) : vertex(v), next(nullptr) {};
};

class LinkedList
{
public:
    Node *head;
    LinkedList *down;
    LinkedList() : head(nullptr), down(nullptr) {};

    void addNode(int vertex)
    {
        Node *newNode = new Node(vertex);
        if (!head)
        {
            head = newNode;
        }
        else
        {
            Node *temp = head;
            while (temp->next)
            {
```

```
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }
    void addLinkedlist(int vertex)
    {
        if (!head)
        {
            head = new Node(vertex);
            down = new LinkedList();
            down->addNode(vertex);
        }
        else
        {
            LinkedList *linkedlist = new LinkedList();
            while (down)
            {
                down = down->down;
            }
            down = linkedlist;
            down->addNode(vertex);
        }
    }

    void printList()
    {
        Node *temp = head->next;
        while (temp)
        {
            std::cout << temp->vertex << " ";
            temp = temp->next;
        }
        std::cout << "\n";
    }
};

class Graph
{
public:
```

```
    int V;
    LinkedList *adjList[MAX_VERTICES];
    Graph(int V);
    void addEdge(int v, int w, bool directed);
    void printAdjList();
};

Graph::Graph(int V)
{
    this->V = V;
    for (int i = 0; i < V; i++)
    {
        adjList[i] = new LinkedList();
    }
}

void Graph::addEdge(int v, int w, bool directed)
{
    adjList[v]->addNode(w);
    if (!directed)
    {
        adjList[w]->addNode(v);
    }
}

void Graph::printAdjList()
{
    for (int i = 0; i < V; i++)
    {
        adjList[i]->printList();
    }
}

class MatrixGraph
{
public:
    int V;
    int adjMatrix[MAX_VERTICES][MAX_VERTICES];

    MatrixGraph(int V);
```

```
    void addEdgeDirected(int v, int w);
    void addEdgeUndirected(int v, int w);
    void printAdjMatrix();
    void BFS(int start);
    void DFS(int start);
};

MatrixGraph::MatrixGraph(int V)
{
    this->V = V;
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            adjMatrix[i][j] = 0;
        }
    }
}

void MatrixGraph::addEdgeDirected(int v, int w)
{
    adjMatrix[v][w] = 1;
}

void MatrixGraph::addEdgeUndirected(int v, int w)
{
    adjMatrix[v][w] = 1;
    adjMatrix[w][v] = 1;
}

void MatrixGraph::BFS(int start)
{
    bool visited[MAX_VERTICES] = {false};
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;
    visited[start] = true;
    queue[rear++] = start;
    int current;
    while (front < rear)
    {
```

```
        current = queue[front++];
        cout << current << " ";
        for (int i = 0; i < V; ++i)
        {
            if (adjMatrix[current][i] && !visited[i])
            {
                visited[i] = true;
                queue[rear++] = i;
            }
        }
    }
    cout << endl;
}
void MatrixGraph::DFS(int start)
{
    bool visited[MAX_VERTICES] = {false};
    int stack[MAX_VERTICES];
    int top = -1;
    stack[++top] = start;
    while (top >= 0)
    {
        int current = stack[top--];
        if (!visited[current])
        {
            cout << current << " ";
            visited[current] = true;
            for (int i = 0; i < V; ++i)
            {
                if (adjMatrix[current][i] && !visited[i])
                {
                    stack[++top] = i;
                }
            }
        }
    }
}

void MatrixGraph::printAdjMatrix()
{
    for (int i = 0; i < V; i++)
```

```
    {
        for (int j = 0; j < V; j++)
        {
            std::cout << adjMatrix[i][j] << " ";
        }
        std::cout << "\n";
    }
}
class linkedGraph
{
public:
    LinkedList *linkedlist;
    int V;
    linkedGraph(int V)
    {
        this->V = V;
        for (int i = 0; i < V; i++)
        {

            linkedlist->addLinkedlist(i);
        }
    };
    void addEdge(int v, int w, bool directed)
    {
        LinkedList *temp = linkedlist;

        for (int i = 0; i < v; i++)
        {
            temp = temp->down;
        }
        temp->addNode(w);
    };
    void print()
    {
        LinkedList *temp = linkedlist;
        while (temp)
        {
            std::cout << "List for vertex " << temp->head->vertex
<< ": ";
            temp->printList();
```

```
            temp = temp->down;
        }
    }
};

int main()
{
    linkedGraph l(2);
    l.addEdge(0, 1, false);
    l.print();

    return 0;
}
```