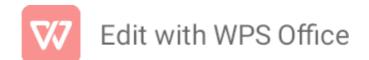
## **ASSIGNMENT NO. 1**

```
#include <iostream>
#include <string>
using namespace std;
class Student {
public:
  int rollNo;
  string name;
  float sgpa;
};
void displayList(Student arr[], int n) {
  cout << "Roll No\tName\tSGPA\n";
  for (int i = 0; i < n; i++) {
     cout << arr[i].rollNo << "\t" << arr[i].name << "\t" << arr[i].sgpa << "\n";
  }
  cout << endl;
}
void swapStudents(Student &a, Student &b) {
  Student temp = a;
  a = b;
  b = temp;
}
// Bubble Sort
void sortByRollNo(Student arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
     for (int j = 0; j < n - i - 1; j++) {
```



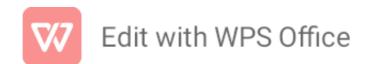
```
if (arr[j].rollNo > arr[j + 1].rollNo) {
          swapStudents(arr[j], arr[j + 1]);
       }
    }
  }
}
// Insertion Sort
void sortAlphabetically(Student arr[], int n) {
  for (int i = 1; i < n; i++) {
     Student key = arr[i];
     int j = i - 1;
     while (j \ge 0 \&\& arr[j].name > key.name) {
       arr[j + 1] = arr[j];
       j = j - 1;
     }
     arr[j + 1] = key;
  }
}
// Quick Sort
int partition(Student arr[], int low, int high) {
  float pivot = arr[high].sgpa;
  int i = low - 1;
  for (int j = low; j <= high - 1; j++) {
     if (arr[j].sgpa >= pivot) {
       i++;
       swapStudents(arr[i], arr[j]);
    }
  }
  swapStudents(arr[i + 1], arr[high]);
```



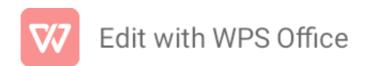
```
return i + 1;
}
void quickSort(Student arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
// Binary Search
int binarySearchByName(Student arr[], int low, int high, string name) {
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid].name == name) {
       return mid;
    }
    if (arr[mid].name < name) {
       low = mid + 1;
    } else {
       high = mid - 1;
    }
  }
  return -1;
}
int main() {
  const int n = 10; // Assuming there are at least 10 records in the database
  Student studentList[n] = {
    {101, "John", 8.5},
```



```
{102, "Alice", 9.2},
  {103, "Bob", 8.7},
  // Add more records as needed
};
// a) Sort by Roll No (Bubble Sort)
sortByRollNo(studentList, n);
cout << "Sorted by Roll No:\n";
displayList(studentList, n);
// b) Sort alphabetically (Insertion Sort)
sortAlphabetically(studentList, n);
cout << "Sorted alphabetically:\n";
displayList(studentList, n);
// c) Sort by SGPA (Quick Sort) to find top 10 toppers
quickSort(studentList, 0, n - 1);
cout << "Top 10 Toppers:\n";
displayList(studentList, min(10, n));
// d) Search students by SGPA
float searchSGPA;
cout << "Enter SGPA to search: ";
cin >> searchSGPA;
cout << "Students with SGPA " << searchSGPA << ":\n";
for (int i = 0; i < n; i++) {
  if (studentList[i].sgpa == searchSGPA) {
    cout << studentList[i].rollNo << "\t" << studentList[i].name << "\n";
  }
}
```



```
// e) Search student by name (Binary Search)
  string searchName;
  cout << "Enter name to search: ";
  cin >> searchName;
  int result = binarySearchByName(studentList, 0, n - 1, searchName);
  if (result != -1) {
    cout << "Student found at index " << result << ":\n";
    cout << studentList[result].rollNo << "\t" << studentList[result].name << "\t" <<
studentList[result].sgpa << "\n";
  } else {
    cout << "Student not found.\n";
  }
  return 0;
}
                                    ASSIGNMENT NO. 2
#include <iostream>
using namespace std;
// Node structure to store student information
struct Node {
  int regNo;
  string name;
  Node* next;
};
// Linked list class
class ClubList {
```



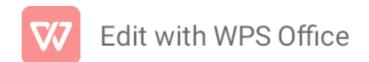
```
private:
  Node* head; // Pointer to the head of the linked list
public:
  // Constructor to initialize an empty list
  ClubList(): head(nullptr) {}
  // Function to add a new member to the club
  void addMember(int regNo, const string& name) {
    Node* newNode = new Node{regNo, name, nullptr};
    if (!head) {
      head = newNode;
    } else {
      newNode->next = head;
      head = newNode;
    }
  }
  // Function to delete a member from the club
  void deleteMember(int regNo) {
    Node* current = head;
    Node* prev = nullptr;
    while (current && current->regNo != regNo) {
      prev = current;
      current = current->next;
    }
    if (!current) {
      cout << "Member with Registration No. " << regNo << " not found." << endl;
      return;
```



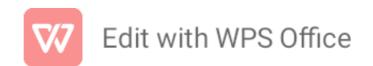
```
}
  if (!prev) {
    head = current->next;
  } else {
    prev->next = current->next;
  }
  delete current;
  cout << "Member with Registration No. " << regNo << " deleted." << endl;
}
// Function to compute the total number of members in the club
int getTotalMembers() {
  int count = 0;
  Node* current = head;
  while (current) {
    count++;
    current = current->next;
  }
  return count;
}
// Function to display all members
void displayMembers() {
  Node* current = head;
  while (current) {
    cout << "Reg No: " << current->regNo << ", Name: " << current->name << endl;
```



```
current = current->next;
    }
  }
  // Function to display list in reverse order using recursion
  void displayReverseOrder(Node* current) {
    if (!current) {
      return;
    }
    displayReverseOrder(current->next);
    cout << "Reg No: " << current->regNo << ", Name: " << current->name << endl;
  }
  // Function to concatenate two linked lists
  void concatenateLists(ClubList& otherList) {
    if (!head) {
      head = otherList.head;
    } else {
       Node* current = head;
      while (current->next) {
         current = current->next;
      }
      current->next = otherList.head;
    }
  }
};
int main() {
  ClubList cometDivision1;
  ClubList cometDivision2;
```



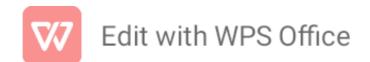
```
// Adding members to Division 1
cometDivision1.addMember(1, "President1");
cometDivision1.addMember(2, "Member1");
cometDivision1.addMember(3, "Member2");
cometDivision1.addMember(4, "Secretary1");
// Adding members to Division 2
cometDivision2.addMember(101, "President2");
cometDivision2.addMember(102, "Member3");
cometDivision2.addMember(103, "Member4");
cometDivision2.addMember(104, "Secretary2");
// Displaying members of Division 1
cout << "Division 1 Members:" << endl;
cometDivision1.displayMembers();
cout << "Total Members: " << cometDivision1.getTotalMembers() << endl;</pre>
// Displaying members of Division 2
cout << "\nDivision 2 Members:" << endl;
cometDivision2.displayMembers();
cout << "Total Members: " << cometDivision2.getTotalMembers() << endl;</pre>
// Concatenating the two divisions
cometDivision1.concatenateLists(cometDivision2);
// Displaying concatenated list
cout << "\nConcatenated List:" << endl;
cometDivision1.displayMembers();
cout << "Total Members: " << cometDivision1.getTotalMembers() << endl;</pre>
```



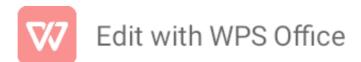
```
// Deleting a member from the concatenated list
  cometDivision1.deleteMember(102);
  // Displaying updated concatenated list
  cout << "\nUpdated Concatenated List:" << endl;</pre>
  cometDivision1.displayMembers();
  cout << "Total Members: " << cometDivision1.getTotalMembers() << endl;</pre>
  // Displaying concatenated list in reverse order using recursion
  cout << "\nConcatenated List in Reverse Order:" << endl;</pre>
  cometDivision1.displayReverseOrder(cometDivision1.head);
  return 0;
}
                                   ASSIGNMENT NO.3/1
#include <iostream>
#include <stack>
#include <string>
#include <sstream>
using namespace std;
// Node class for the linked list
class Node {
public:
  int data;
  Node* next;
  Node(int val) : data(val), next(nullptr) {}
```



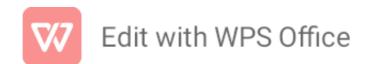
```
};
// Stack class using a linked list
class LinkedListStack {
private:
  Node* top;
public:
  LinkedListStack(): top(nullptr) {}
  void push(int val) {
     Node* newNode = new Node(val);
     newNode->next = top;
     top = newNode;
  }
  int pop() {
     if (isEmpty()) {
       cerr << "Stack is empty. Cannot pop.\n";
       return -1; // Assuming -1 as an error value
    }
     int poppedValue = top->data;
     Node* temp = top;
     top = top->next;
     delete temp;
     return poppedValue;
  }
  int peek() {
```



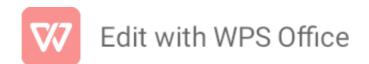
```
if (isEmpty()) {
      cerr << "Stack is empty. Cannot peek.\n";
      return -1; // Assuming -1 as an error value
    }
                                  ASSSIGNMENT NO. 4/2
#include <iostream>
#include <queue>
using namespace std;
// Function to add a job to the queue
void addJob(queue<string>& jobQueue, const string& job) {
  jobQueue.push(job);
  cout << "Job "" << job << "" added to the queue." << endl;
}
// Function to delete a job from the queue
void deleteJob(queue<string>& jobQueue) {
  if (jobQueue.empty()) {
    cout << "Queue is empty. No job to delete." << endl;
  } else {
    string deletedJob = jobQueue.front();
    jobQueue.pop();
    cout << "Job "" << deletedJob << "" deleted from the queue." << endl;
  }
}
// Function to display the current jobs in the queue
void displayJobs(const queue<string>& jobQueue) {
  if (jobQueue.empty()) {
    cout << "Queue is empty. No jobs to display." << endl;
```



```
} else {
    cout << "Current Jobs in the Queue:" << endl;
    queue<string> tempQueue = jobQueue;
    while (!tempQueue.empty()) {
      cout << tempQueue.front() << " ";
      tempQueue.pop();
    }
    cout << endl;
  }
}
int main() {
  // Creating a queue to simulate the job queue
  queue<string> jobQueue;
  // Adding jobs to the queue
  addJob(jobQueue, "Job1");
  addJob(jobQueue, "Job2");
  addJob(jobQueue, "Job3");
  // Displaying current jobs
  displayJobs(jobQueue);
  // Deleting a job from the queue
  deleteJob(jobQueue);
  // Displaying updated jobs
  displayJobs(jobQueue);
  // Deleting a job from an empty queue
  deleteJob(jobQueue);
```



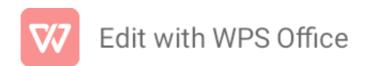
```
return 0;
}
                                   ASSIGNMENT NO. 5/3
#include <iostream>
using namespace std;
class Deque {
private:
  int* arr;
             // Array to store deque elements
  int capacity; // Maximum capacity of the deque
  int front;
            // Front index of the deque
  int rear;
             // Rear index of the deque
  int size; // Current size of the deque
public:
  // Constructor to initialize the deque with a given capacity
  Deque(int cap): capacity(cap), front(-1), rear(-1), size(0) {
    arr = new int[capacity];
  }
  // Destructor to free the allocated memory
  ~Deque() {
    delete[] arr;
  }
  // Function to check if the deque is empty
  bool isEmpty() {
    return size == 0;
  }
```



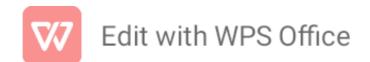
```
// Function to check if the deque is full
bool isFull() {
  return size == capacity;
}
// Function to add an element to the front of the deque
void addFront(int value) {
  if (isFull()) {
    cout << "Deque is full. Cannot add more elements." << endl;
    return;
  }
  if (isEmpty()) {
    front = rear = 0;
  } else {
    front = (front - 1 + capacity) % capacity;
  }
  arr[front] = value;
  size++;
  cout << "Added " << value << " to the front of the deque." << endl;
}
// Function to add an element to the rear of the deque
void addRear(int value) {
  if (isFull()) {
    cout << "Deque is full. Cannot add more elements." << endl;
    return;
  }
```



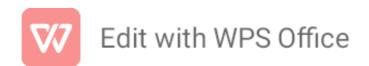
```
if (isEmpty()) {
    front = rear = 0;
  } else {
    rear = (rear + 1) % capacity;
  }
  arr[rear] = value;
  size++;
  cout << "Added " << value << " to the rear of the deque." << endl;
}
// Function to delete an element from the front of the deque
void deleteFront() {
  if (isEmpty()) {
    cout << "Deque is empty. Cannot delete from an empty deque." << endl;
    return;
  }
  if (size == 1) {
    front = rear = -1;
  } else {
    front = (front + 1) % capacity;
  }
  size--;
  cout << "Deleted element from the front of the deque." << endl;
}
```



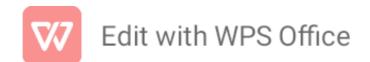
```
// Function to delete an element from the rear of the deque
void deleteRear() {
  if (isEmpty()) {
    cout << "Deque is empty. Cannot delete from an empty deque." << endl;
    return:
  }
  if (size == 1) {
    front = rear = -1;
  } else {
     rear = (rear - 1 + capacity) % capacity;
  }
  size--;
  cout << "Deleted element from the rear of the deque." << endl;
}
// Function to display the elements of the deque
void display() {
  if (isEmpty()) {
    cout << "Deque is empty." << endl;
    return;
  }
  cout << "Deque elements: ";</pre>
  int i = front;
  for (int count = 0; count < size; count++) {
    cout << arr[i] << " ";
    i = (i + 1) \% capacity;
  }
```



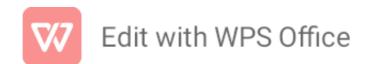
```
cout << endl;
 }
};
int main() {
  // Creating a deque with a capacity of 5
  Deque myDeque(5);
  // Adding elements to the front and rear of the deque
  myDeque.addFront(1);
  myDeque.addRear(2);
  myDeque.addFront(3);
  myDeque.addRear(4);
  // Displaying the deque
  myDeque.display();
  // Deleting elements from the front and rear of the deque
  myDeque.deleteFront();
  myDeque.deleteRear();
  // Displaying the updated deque
  myDeque.display();
  return 0;
}
                                 ASSIGNMENT NO. 6/4
#include <iostream>
#include <vector>
```



```
using namespace std;
class Subsection {
public:
  string name;
  Subsection(const string& n): name(n) {}
};
class Section {
public:
  string name;
  vector<Subsection> subsections;
  Section(const string& n): name(n) {}
};
class Chapter {
public:
  string name;
  vector<Section> sections;
  Chapter(const string& n): name(n) {}
};
class Book {
public:
  string name;
  vector<Chapter> chapters;
  Book(const string& n) : name(n) {}
```



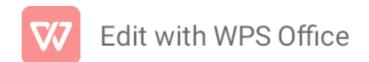
```
};
void printBook(const Book& book) {
  cout << "Book: " << book.name << endl;
  for (const auto& chapter : book.chapters) {
    cout << " Chapter: " << chapter.name << endl;
    for (const auto& section : chapter.sections) {
       cout << " Section: " << section.name << endl;</pre>
      for (const auto& subsection : section.subsections) {
         cout << " Subsection: " << subsection.name << endl;
      }
    }
  }
}
int main() {
  // Constructing a sample book with chapters, sections, and subsections
  Book myBook("My Book");
  Chapter chapter1("Introduction");
  chapter1.sections.push_back(Section("Overview"));
  chapter1.sections.push_back(Section("Objectives"));
  chapter1.sections[0].subsections.push_back(Subsection("History"));
  chapter1.sections[1].subsections.push_back(Subsection("Scope"));
  Chapter chapter2("Main Content");
  chapter2.sections.push_back(Section("Chapter A"));
  chapter2.sections.push_back(Section("Chapter B"));
```



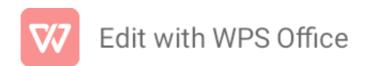
```
chapter2.sections[0].subsections.push_back(Subsection("Topic 1"));
  chapter2.sections[1].subsections.push_back(Subsection("Topic 2"));
  myBook.chapters.push_back(chapter1);
  myBook.chapters.push_back(chapter2);
  // Print the book hierarchy
  printBook(myBook);
  return 0;
}
                                   ASSIGNMENT NO. 7/5
#include <iostream>
using namespace std;
// Node structure for the binary tree
struct Node {
  int data;
  Node* left;
  Node* right;
  Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
// Binary tree class
class BinaryTree {
private:
  Node* root;
  // Private helper functions for recursive traversals
```



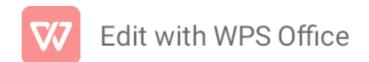
```
void inOrderTraversalRecursive(Node* node) {
    if (node) {
      inOrderTraversalRecursive(node->left);
      cout << node->data << " ";
      inOrderTraversalRecursive(node->right);
    }
  }
  void preOrderTraversalRecursive(Node* node) {
    if (node) {
      cout << node->data << " ";
      preOrderTraversalRecursive(node->left);
      preOrderTraversalRecursive(node->right);
    }
  }
  void postOrderTraversalRecursive(Node* node) {
    if (node) {
      postOrderTraversalRecursive(node->left);
      postOrderTraversalRecursive(node->right);
      cout << node->data << " ";
    }
  }
public:
  // Constructor to initialize an empty binary tree
  BinaryTree() : root(nullptr) {}
  // Function to insert a new node into the binary tree
  void insert(int value) {
    root = insertRecursive(root, value);
```



```
}
// Private helper function for recursive insertion
Node* insertRecursive(Node* node, int value) {
  if (!node) {
    return new Node(value);
  }
  if (value < node->data) {
    node->left = insertRecursive(node->left, value);
  } else if (value > node->data) {
    node->right = insertRecursive(node->right, value);
  }
  return node;
}
// Function to perform in-order traversal
void inOrderTraversal() {
  cout << "In-order Traversal: ";
  inOrderTraversalRecursive(root);
  cout << endl;
}
// Function to perform pre-order traversal
void preOrderTraversal() {
  cout << "Pre-order Traversal: ";
  preOrderTraversalRecursive(root);
  cout << endl;
}
```

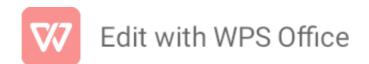


```
// Function to perform post-order traversal
  void postOrderTraversal() {
    cout << "Post-order Traversal: ";
    postOrderTraversalRecursive(root);
    cout << endl;
  }
};
int main() {
  // Creating a binary tree
  BinaryTree myTree;
  // Inserting elements into the binary tree
  myTree.insert(50);
  myTree.insert(30);
  myTree.insert(70);
  myTree.insert(20);
  myTree.insert(40);
  myTree.insert(60);
  myTree.insert(80);
  // Performing recursive traversals
  myTree.inOrderTraversal();
  myTree.preOrderTraversal();
  myTree.postOrderTraversal();
  return 0;
}
```



## ASSIGNMENT NO. 8/6

```
#include <iostream>
#include <climits>
using namespace std;
// Node structure for the binary search tree
struct Node {
  int data:
  Node* left:
  Node* right;
  Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
// Binary Search Tree class
class BinarySearchTree {
private:
  Node* root;
  // Private helper function for inserting a new node
  Node* insertRecursive(Node* node, int value) {
    if (!node) {
       return new Node(value);
    }
    if (value < node->data) {
       node->left = insertRecursive(node->left, value);
    } else if (value > node->data) {
       node->right = insertRecursive(node->right, value);
    }
```



```
return node;
}
// Private helper function to find the number of nodes in the longest path
int findLongestPathLengthRecursive(Node* node) {
  if (!node) {
    return 0;
  }
  int leftPath = findLongestPathLengthRecursive(node->left);
  int rightPath = findLongestPathLengthRecursive(node->right);
  return 1 + max(leftPath, rightPath);
}
// Private helper function to find the minimum data value in the tree
int findMinValueRecursive(Node* node) {
  if (!node) {
    // Return a large value for an empty tree
    return INT_MAX;
  }
  int leftMin = findMinValueRecursive(node->left);
  int rightMin = findMinValueRecursive(node->right);
  return min(node->data, min(leftMin, rightMin));
}
// Private helper function to swap left and right pointers at every node
Node* swapPointersRecursive(Node* node) {
  if (!node) {
```



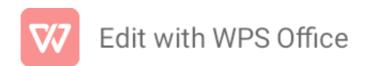
```
return nullptr;
    }
    // Swap left and right pointers
    Node* temp = node->left;
    node->left = swapPointersRecursive(node->right);
    node->right = swapPointersRecursive(temp);
    return node;
  }
  // Private helper function to search for a value in the tree
  bool searchValueRecursive(Node* node, int value) {
    if (!node) {
      return false;
    }
    if (node->data == value) {
       return true;
    } else if (value < node->data) {
       return searchValueRecursive(node->left, value);
    } else {
      return searchValueRecursive(node->right, value);
    }
  }
public:
  // Constructor to initialize an empty binary search tree
  BinarySearchTree() : root(nullptr) {}
  // Function to insert a new node into the binary search tree
```



```
void insert(int value) {
    root = insertRecursive(root, value);
  }
  // Function to find the number of nodes in the longest path
  int findLongestPathLength() {
    return findLongestPathLengthRecursive(root);
  }
  // Function to find the minimum data value in the tree
  int findMinValue() {
    return findMinValueRecursive(root);
  }
  // Function to change the tree so that the roles of the left and right pointers are swapped
  void swapPointers() {
    root = swapPointersRecursive(root);
  }
  // Function to search for a value in the tree
  bool searchValue(int value) {
    return searchValueRecursive(root, value);
  }
};
int main() {
  // Create a binary search tree
  BinarySearchTree bst;
  // Insert values into the binary search tree
  bst.insert(50);
```



```
bst.insert(30);
  bst.insert(70);
  bst.insert(20);
  bst.insert(40);
  bst.insert(60);
  bst.insert(80);
  // Find and display the number of nodes in the longest path
  cout << "Number of nodes in the longest path: " << bst.findLongestPathLength() << endl;
  // Find and display the minimum data value in the tree
  cout << "Minimum data value in the tree: " << bst.findMinValue() << endl;
  // Swap the left and right pointers at every node
  bst.swapPointers();
  // Search for a value in the tree
  int searchValue = 40;
  if (bst.searchValue(searchValue)) {
    cout << "Value " << searchValue << " found in the tree." << endl;
  } else {
    cout << "Value " << searchValue << " not found in the tree." << endl;
  }
  return 0;
                                   ASSIGNMENT NO. 9/7
// C++ implementation of the approach
#include <bits/stdc++.h>
```



}

```
using namespace std;
class Graph {
// Number of vertex
int v;
// Number of edges
int e;
// Adjacency matrix
int** adj;
public:
// To create the initial adjacency matrix
Graph(int v, int e);
// Function to insert a new edge
void addEdge(int start, int e);
// Function to display the BFS traversal
void BFS(int start);
};
// Function to fill the empty adjacency matrix
Graph::Graph(int v, int e)
{
this->v = v;
this->e = e;
adj = new int*[v];
for (int row = 0; row < v; row++) {
adj[row] = new int[v];
for (int column = 0; column < v; column++) {
adj[row][column] = 0;
}
}
}
// Function to add an edge to the graph
void Graph::addEdge(int start, int e)
```



```
{
// Considering a bidirectional edge
adj[start][e] = 1;
adi[e][start] = 1;
}
// Function to perform BFS on the graph
void Graph::BFS(int start)
{
// Visited vector to so that
// a vertex is not visited more than once
// Initializing the vector to false as no
// vertex is visited at the beginning
vector<br/>vector<br/>visited(v, false);
vector<int> q;
q.push_back(start);
// Set source as visited
visited[start] = true;
int vis;
while (!q.empty()) {
vis = q[0];
// Print the current node
cout << vis << " ";
q.erase(q.begin());
// For every adjacent vertex to the current vertex
for (int i = 0; i < v; i++) {
if (adj[vis][i] == 1 && (!visited[i])) {
// Push the adjacent node to the queue
q.push_back(i);
// Set
visited[i] = true;
}
```



```
}
}
}
// Driver code
int main()
{
int v = 5, e = 4;
// Create the graph
Graph G(v, e);
G.addEdge(0, 1);
G.addEdge(0, 2);
G.addEdge(1, 3);
G.BFS(0);
}
                                   ASSIGNMENT NO. 10/8
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class Edge {
public:
  int src, dest, weight;
  Edge(int s, int d, int w): src(s), dest(d), weight(w) {}
};
class Graph {
public:
```



```
int vertices;
  vector<Edge> edges;
  Graph(int v) : vertices(v) {}
  void addEdge(int src, int dest, int weight) {
    edges.emplace_back(src, dest, weight);
  }
  // Kruskal's Algorithm
  void minimumSpanningTree() {
    // Sort edges by weight
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
      return a.weight < b.weight;
    });
    vector<int> parent(vertices, -1);
    cout << "Minimum Spanning Tree:\n";
    for (const Edge& edge: edges) {
      int srcParent = find(parent, edge.src);
      int destParent = find(parent, edge.dest);
      if (srcParent != destParent) {
         cout << "Edge " << edge.src << " - " << edge.dest << " (Weight: " << edge.weight <<
")\n";
         unionSets(parent, srcParent, destParent);
      }
    }
  }
```



```
int find(const vector<int>& parent, int i) {
    if (parent[i] == -1)
       return i;
    return find(parent, parent[i]);
  }
  void unionSets(vector<int>& parent, int x, int y) {
    int xRoot = find(parent, x);
    int yRoot = find(parent, y);
    parent[xRoot] = yRoot;
  }
};
int main() {
  // Create a graph with 4 vertices (offices)
  Graph graph(4);
  // Add edges with their weights
  graph.addEdge(0, 1, 2);
  graph.addEdge(0, 2, 4);
  graph.addEdge(1, 2, 1);
  graph.addEdge(1, 3, 3);
  graph.addEdge(2, 3, 5);
  // Find and print the minimum spanning tree
  graph.minimumSpanningTree();
  return 0;
}
```

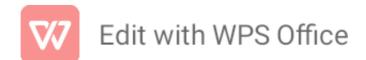
## ASSIGNMENT NO. 11/10

```
#include <iostream>
#include <vector>
using namespace std;
// Function to print the chessboard with queens
void printChessboard(const vector<int>& queens) {
  int n = queens.size();
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
       if (queens[i] == j) {
         cout << "Q ";
       } else {
         cout << ". ";
       }
    }
    cout << endl;
  }
  cout << endl;
}
// Function to check if placing a queen at a certain position is safe
bool isSafe(const vector<int>& queens, int row, int col) {
  int n = queens.size();
  // Check for queens in the same column
  for (int i = 0; i < row; ++i) {
    if (queens[i] == col) {
```



```
return false;
    }
  }
  // Check for queens in the left diagonal
  for (int i = row, j = col; i \ge 0 \&\& j \ge 0; -i, -j) {
     if (queens[i] == j) {
       return false;
    }
  }
  // Check for queens in the right diagonal
  for (int i = row, j = col; i >= 0 \&\& j < n; -i, ++j) {
     if (queens[i] == j) {
       return false;
    }
  }
  return true;
// Function to solve the N-Queens problem using backtracking
void solveNQueens(vector<int>& queens, int row, int n) {
  if (row == n) {
     // All queens are placed successfully, print the configuration
     printChessboard(queens);
     return;
  }
  for (int col = 0; col < n; ++col) {
     if (isSafe(queens, row, col)) {
```

}



```
queens[row] = col;
solveNQueens(queens, row + 1, n);
// Backtrack
queens[row] = -1;
}

int main() {
  int n = 4; // Number of queens
  vector<int> queens(n, -1); // Vector to store the column position of queens in each row
  // Solve the N-Queens problem
  solveNQueens(queens, 0, n);

return 0;
}
```