**1.** Implement Breadth-first search

Adjacency \*BFS(int v);

where Adjacency is a structure to store list of number.

#include <iostream>

#include <list>

using namespace std;

class Adjacency

{

private:

list<int> adjList;

int size;

public:

Adjacency() {}

Adjacency(int V) {}

void push(int data)

{

adjList.push\_back(data);

size++;

}

void print()

{

for (auto const &i : adjList)

cout << " -> " << i;

}

void printArray()

{

for (auto const &i : adjList)

cout << i << " ";

}

int getSize() { return adjList.size(); }

int getElement(int idx)

{

auto it = adjList.begin();

advance(it, idx);

return \*it;

}

};

And Graph is a structure to store a graph (see in your answer box)  
**For example:**

| **Test** | **Result** |
| --- | --- |
| int V = 6;  int visited = 0;  Graph g(V);  Adjacency\* arr = new Adjacency(V);  int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};    for(int i = 0; i < 8; i++)  {  g.addEdge(edge[i][0], edge[i][1]);  }    arr = g.BFS(visited);  arr->printArray();  delete arr; | 0 1 2 3 4 5 |
| int V = 6;  int visited = 2;  Graph g(V);  Adjacency\* arr = new Adjacency(V);  int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};    for(int i = 0; i < 8; i++)  {  g.addEdge(edge[i][0], edge[i][1]);  }    arr = g.BFS(visited);  arr->printArray();  delete arr; | 2 0 4 1 3 5 |

|  |
| --- |
| class Graph {  private:  int V;  Adjacency \*adj;  public:  Graph (int V) {  this -> V = V;  adj = new Adjacency[V];  }    void addEdge (int v, int w) {  adj[v].push(w);  adj[w].push(v);  }    void printGraph() {  for (int v = 0; v < V; ++v) {  cout << "\nAdjacency list of vertex " << v << "\nhead ";  adj[v].print();  }  }    Adjacency \*BFS (int v) {  // v is a vertex we start DFS  int numVer = this -> adj -> getSize();  Adjacency \*traversedList = new Adjacency;  bool \*visited = new bool[numVer];  for (int i = 0; i < numVer; ++i) {  visited[i] = false;  }  list<int> queue;  visited[v] = true;  queue.push\_back(v);  while (!queue.empty()) {  int curVer = queue.front();  traversedList -> push (curVer);  queue.pop\_front();  int curSize = this -> adj[curVer].getSize();  for (int i = 0; i < curSize; ++i) {  int adjVer = this -> adj[curVer].getElement(i);  if (!visited[adjVer]) {  visited[adjVer] = true;  queue.push\_back (adjVer);  }  }  }  delete[] visited;  return traversedList;  }  }; |

**2.** Implement Depth-first search

Adjacency \*DFS(int v);

where Adjacency is a structure to store list of number.

#include <iostream>

#include <list>

using namespace std;

class Adjacency

{

private:

list<int> adjList;

int size;

public:

Adjacency() {}

Adjacency(int V) {}

void push(int data)

{

adjList.push\_back(data);

size++;

}

void print()

{

for (auto const &i : adjList)

cout << " -> " << i;

}

void printArray()

{

for (auto const &i : adjList)

cout << i << " ";

}

int getSize() { return adjList.size(); }

int getElement(int idx)

{

auto it = adjList.begin();

advance(it, idx);

return \*it;

}

};

And Graph is a structure to store a graph (see in your answer box)

**For example:**

| **Test** | **Result** |
| --- | --- |
| int V = 8, visited = 0;  Graph g(V);  Adjacency \*arr;  int edge[][2] = {{0,1}, {0,2}, {0,3}, {0,4}, {1,2}, {2,5}, {2,6}, {4,6}, {6,7}};  for(int i = 0; i < 9; i++)  {  g.addEdge(edge[i][0], edge[i][1]);  }  // g.printGraph();  // cout << endl;  arr = g.DFS(visited);  arr->printArray();  delete arr; | 0 1 2 5 6 4 7 3 |

|  |
| --- |
| class Graph {  private:  int V;  Adjacency \*adj;  public:  Graph (int V) {  this -> V = V;  adj = new Adjacency[V];  }    void addEdge (int v, int w) {  adj[v].push(w);  adj[w].push(v);  }    void printGraph() {  for (int v = 0; v < V; ++v) {  cout << "\nAdjacency list of vertex " << v << "\nhead ";  adj[v].print();  }  }    Adjacency \*DFS (int v) {  // v is a vertex we start DFS  int numVer = this -> adj -> getSize();  Adjacency \*traversedList = new Adjacency;  bool \*visited = new bool[numVer];  for (int i = 0; i < numVer; ++i) {  visited [i] = false;  }  this -> DFSRec (v, visited, traversedList);  delete[] visited;  return traversedList;  }  void DFSRec (int v, bool \*visited, Adjacency \*&traversedList) {  visited [v] = true;  traversedList -> push (v);  int curSize = this -> adj[v].getSize();  for (int i = 0; i < curSize; ++i) {  int adjVer = this -> adj[v].getElement(i);  if (!visited[adjVer])  this -> DFSRec(adjVer, visited, traversedList);  }  }  }; |

**3.** Given a graph and a source vertex in the graph, find shortest paths from source to destination vertice in the given graph using Dijsktra's algorithm.

**For example:**

| **Test** | **Result** |
| --- | --- |
| int n = 6;  int init[6][6] = {  {0, 10, 20, 0, 0, 0},  {10, 0, 0, 50, 10, 0},  {20, 0, 0, 20, 33, 0},  {0, 50, 20, 0, 20, 2},  {0, 10, 33, 20, 0, 1},  {0, 0, 0, 2, 1, 0} };  int\*\* graph = new int\*[n];  for (int i = 0; i < n; ++i) {  graph[i] = init[i];  }  cout << Dijkstra(graph, 0, 0); | 0 |
| int n = 6;  int init[6][6] = {  {0, 10, 20, 0, 0, 0},  {10, 0, 0, 50, 10, 0},  {20, 0, 0, 20, 33, 0},  {0, 50, 20, 0, 20, 2},  {0, 10, 33, 20, 0, 1},  {0, 0, 0, 2, 1, 0} };  int\*\* graph = new int\*[n];  for (int i = 0; i < n; ++i) {  graph[i] = init[i];  }  cout << Dijkstra(graph, 0, 1); | 10 |

|  |
| --- |
| // Some helping functions  int minDistance (int dist[], bool sptSet[], int V) {  int min = 99999, min\_index = -1;  for (int v = 0; v < V; v++)  if (sptSet [v] == false && dist [v] <= min)  min = dist [v], min\_index = v;  return min\_index;  }  int Dijkstra (int \*\*graph, int src, int dst) {  // TODO: return the length of shortest path from src to dst.  int V = 6;  int dist [V];  bool sptSet [V];  for (int i = 0; i < V; i++) {  dist [i] = 99999;  sptSet [i] = false;  }  dist [src] = 0;  for (int count = 0; count < V - 1; count++) {  int u = minDistance (dist, sptSet, V);  sptSet [u] = true;  for (int v = 0; v < V; v++)  if (!sptSet[v] && graph[u][v] && dist[u] != 99999 && dist[u] + graph[u][v] < dist[v])  dist[v] = dist[u] + graph[u][v];  }  return dist [dst];  } |

**4.** Implement function to detect a cyclic in Graph

bool isCyclic();

Graph structure in this lab is slightly different from previous labs.

#include<iostream>

#include <list>

using namespace std;

class DirectedGraph

{

int V;

list<int> \*adj;

bool isCyclicUtil(int v, bool visited[], bool \*rs);

public:

DirectedGraph(){

V = 0;

adj = NULL;

}

DirectedGraph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void addEdge(int v, int w)

{

adj[v].push\_back(w);

}

bool isCyclic();

};

**For example:**

| **Test** | **Result** |
| --- | --- |
| DirectedGraph g(8);  int edege[][2] = {{0,6}, {1,2}, {1,4}, {1,6}, {3,0}, {3,4}, {5,1}, {7,0}, {7,1}};  for(int i = 0; i < 9; i++)  g.addEdge(edege[i][0], edege[i][1]);  if(g.isCyclic())  cout << "Graph contains cycle";  else  cout << "Graph doesn't contain cycle"; | Graph doesn't contain cycle |

|  |
| --- |
| #include <iostream>  #include <list>  using namespace std;  class DirectedGraph {  int V;  list<int> \*adj;  bool isCyclicUtil (int v, bool visited[], bool \*rs) {  if (!visited[v]) {  visited [v] = true;  rs[v] = true;  list<int>::iterator it;  for (it = adj[v].begin(); it != adj[v].end(); ++it) {  if (!visited[\*it] && isCyclicUtil (\*it, visited, rs))  return true;  else if (rs [\*it])  return true;  }  }  rs [v] = false;  return false;  }  public:  DirectedGraph() {  V = 0;  adj = NULL;  }  DirectedGraph (int V) {  this -> V = V;  adj = new list<int> [V];  }  void addEdge (int v, int w) {  adj[v].push\_back(w);  }  bool isCyclic() {  bool \*visited = new bool [V];  bool \*rs = new bool [V];  for (int i = 0; i < V; i++) {  visited [i] = false;  rs [i] = false;  }  for (int i = 0; i < V; i++)  if (isCyclicUtil (i, visited, rs))  return true;  return false;  }  }; |

**5.** Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Kruskal’s algorithm.

Below are the steps for finding MST using Kruskal’s algorithm:

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <algorithm>  
#include <iostream>  
#include <utility>  
#include <map>  
#include <vector>  
#include <set>  
using namespace std;

struct Graph  
{  
    int V, E;  
    vector< pair<int, pair<int, int>> > edges;  
    // Constructor   
    Graph(int V, int E)  
    {  
        this->V = V;  
        this->E = E;  
    }

    void addEdge(int u, int v, int w)  
    {  
        edges.push\_back({ w, {u, v} });  
    }

//YOUR CODE HERE

};

**For example:**

| **Test** | **Result** |
| --- | --- |
| int V = 2, E = 1;  Graph g(V, E);  g.addEdge(0, 1, 2);  cout << g.kruskalMST(); | 2 |
| int V = 3, E = 3;  Graph g(V, E);  g.addEdge(0, 1, 2);  g.addEdge(1, 2, 2);  g.addEdge(0, 2, 3);  cout << g.kruskalMST(); | 4 |

|  |
| --- |
| // Some helping functions  class subset {  public:  int parent;  int rank;  };  int find (subset subsets[], int i) {  if (subsets[i].parent != i) {  subsets[i].parent = find (subsets, subsets[i].parent);  }  return subsets[i].parent;  }  void Union (subset subsets[], int x, int y) {  int xroot = find (subsets, x);  int yroot = find (subsets, y);  if (subsets[xroot].rank < subsets[yroot].rank)  subsets[xroot].parent = yroot;  else if (subsets[xroot].rank > subsets[yroot].rank)  subsets[yroot].parent = xroot;  else {  subsets[yroot].parent = xroot;  subsets[xroot].rank++;  }  }  int kruskalMST() {  // TODO: return weight of the minimum spanning tree.  vector <pair <int, pair <int, int> > > holder;  bool visit [edges.size()];  for (unsigned int i = 0; i < edges.size(); i++) {  visit [i]=0;  }  int min = -1;  int z = -1;  while (holder.size() < edges.size()) {  for (unsigned int i = 0; i < edges.size(); i++) {  if ((min > edges[i].first || min == -1) && visit [i] == 0) {  z = i;  min = edges[i].first;  }  }  holder.push\_back (edges [z]);  visit [z] = 1;  min = -1;  z = -1;  }  subset \*subsets = new subset [(V \* sizeof(subset))];  for (int v = 0; v < V; ++v) {  subsets[v].parent = v;  subsets[v].rank = 0;  }  int ans = 0;  int e = 0;  int i = 0;  vector <pair <int, pair <int, int> > > spanningTree;    while (e < V - 1 && i < E) {  pair <int, pair <int, int> > next\_edge;  next\_edge = holder [i++];  int x = find (subsets, next\_edge.second.first);  int y = find (subsets, next\_edge.second.second);  if (x != y) {  spanningTree.push\_back (next\_edge); e++;  Union (subsets, x, y);  }  }  for (unsigned i = 0; i < spanningTree.size(); ++i) {  ans = ans + spanningTree[i].first;  }  return ans;  } |

**6.** Implement **topologicalSort** function on a graph. (Ref [here](https://www.geeksforgeeks.org/topological-sorting/))

void topologicalSort();

where Adjacency is a structure to store list of number. Note that, the vertex index starts from 0. **To match the given answer, please always traverse from 0 when performing the sorting.**

#include <iostream>

#include <list>

using namespace std;

class Adjacency

{

private:

list<int> adjList;

int size;

public:

Adjacency() {}

Adjacency(int V) {}

void push(int data)

{

adjList.push\_back(data);

size++;

}

void print()

{

for (auto const &i : adjList)

cout << " -> " << i;

}

void printArray()

{

for (auto const &i : adjList)

cout << i << " ";

}

int getSize() { return adjList.size(); }

int getElement(int idx)

{

auto it = adjList.begin();

advance(it, idx);

return \*it;

}

};

And Graph is a structure to store a graph (see in your answer box). You could write one or more helping functions.  
**For example:**

| **Test** | **Result** |
| --- | --- |
| Graph g(6);  g.addEdge(5, 2);  g.addEdge(5, 0);  g.addEdge(4, 0);  g.addEdge(4, 1);  g.addEdge(2, 3);  g.addEdge(3, 1);  g.topologicalSort(); | 5 4 2 3 1 0 |

|  |
| --- |
| class Graph {  int V;  Adjacency \*adj;  public:  Graph (int V) {  this -> V = V;  adj = new Adjacency [V];  }  void addEdge (int v, int w) {  adj[v].push(w);  }  //Heling functions  void topologicalSortUtil (int v, bool visited[], list <int> &Stack) {  visited [v] = true;  for (int i = 0; i != adj[v].getSize(); ++i) {  if (!visited[adj[v].getElement(i)]) {  topologicalSortUtil (adj[v].getElement(i), visited, Stack);  }  }  Stack.push\_back(v);  }  void topologicalSort() {  //TODO  list <int> Stack;  bool \*visited = new bool [V];  for (int i = 0; i < V; i++) {  visited [i] = false;  }  for (int i = 0; i < V; i++) {  if (visited [i] == false) {  topologicalSortUtil (i, visited, Stack);  }  }  while (Stack.empty() == false) {  cout << Stack.back() << " ";  Stack.pop\_back();  }  }  }; |