1. Write a program to find the shortest path in a graph using either Prim's Algorithm or Kruskal's Algorithm.

Code:

```
#include <iostream>
#define V 8
#define I 32767

using namespace std;

void PrintMST(int T[][V-2], int G[V][V]){
    cout << "\nMinimum Spanning Tree Edges (w/ cost)\n" << endl;
    int sum {0};
    for (int i {0}; i<V-2; i++){
        int c = G[T[0][i]][T[1][i]];
        cout << "[" << T[0][i] << "]---[" << T[1][i] << "] cost: " << c << endl;
        sum += c;
    }
    cout << endl;
    cout << "Total cost of MST: " << sum << endl;
}

void PrimsMST(int G[V][V], int n){
    int u;
    int v;
    int min {I};
    int track [V];
    int T[2][V-2] {0};

    // Initial: Find min cost edge
    for (int i {1}; i<V; i++){
        track[i] = I;  // Initialize track array with INFINITY
        for (int j {i}; j<V; j++){
            if (G[i][j] < min){
                min = G[i][j];
                u = i;
```

```cpp
                v = j;
            }
        }
    }
    T[0][0] = u;
    T[1][0] = v;
    track[u] = track[v] = 0;

    // Initialize track array to track min cost edges
    for (int i {1}; i<V; i++){
        if (track[i] != 0){
            if (G[i][u] < G[i][v]){
                track[i] = u;
            } else {
                track[i] = v;
            }
        }
    }

    // Repeat
    for (int i {1}; i<n-1; i++){
        int k;
        min = I;
        for (int j {1}; j<V; j++){
            if (track[j] != 0 && G[j][track[j]] < min){
                k = j;
                min = G[j][track[j]];
            }
        }
        T[0][i] = k;
        T[1][i] = track[k];
        track[k] = 0;

        // Update track array to track min cost edges
        for (int j {1}; j<V; j++){
            if (track[j] != 0 && G[j][k] < G[j][track[j]]){
                track[j] = k;
            }
        }
    }
    PrintMST(T, G);
}

int main() {
```

```
int cost [V][V] {
        {I, I, I, I, I, I, I, I},
        {I, I, 25, I, I, I, 5, I},
        {I, 25, I, 12, I, I, I, 10},
        {I, I, 12, I, 8, I, I, I},
        {I, I, I, 8, I, 16, I, 14},
        {I, I, I, I, 16, I, 20, 18},
        {I, 5, I, I, I, 20, I, I},
        {I, I, 10, I, 14, 18, I, I},
};

int n = sizeof(cost[0])/sizeof(cost[0][0]) - 1;

PrimsMST(cost, n);
```

Output:

```
Minimum Spanning Tree Edges (w/ cost)

[1]---[6] cost: 5
[5]---[6] cost: 20
[4]---[5] cost: 16
[3]---[4] cost: 8
[2]---[3] cost: 12
[7]---[2] cost: 10

Total cost of MST: 71
```

```
#include <iostream>
#define V 8
#define I 32767

using namespace std;

void PrintMST(int T[][V-2], int G[V][V]){
   cout << "\nMinimum Spanning Tree Edges (w/ cost)\n" << endl;
   int sum {0};
   for (int i {0}; i<V-2; i++){
      int c = G[T[0][i]][T[1][i]];
```

```
        cout << "[" << T[0][i] << "]---[" << T[1][i] << "] cost: " << c << endl;
        sum += c;
    }
    cout << endl;
    cout << "Total cost of MST: " << sum << endl;
}

void PrimsMST(int G[V][V], int n){
    int u;
    int v;
    int min {I};
    int track [V];
    int T[2][V-2] {0};

    // Initial: Find min cost edge
    for (int i {1}; i<V; i++){
        track[i] = I;  // Initialize track array with INFINITY
        for (int j {i}; j<V; j++){
            if (G[i][j] < min){
                min = G[i][j];
                u = i;
                v = j;
            }
        }
    }
    T[0][0] = u;
    T[1][0] = v;
    track[u] = track[v] = 0;

    // Initialize track array to track min cost edges
    for (int i {1}; i<V; i++){
        if (track[i] != 0){
            if (G[i][u] < G[i][v]){
                track[i] = u;
            } else {
```

```
                track[i] = v;
            }
        }
    }

    // Repeat
    for (int i {1}; i<n-1; i++){
        int k;
        min = I;
        for (int j {1}; j<V; j++){
            if (track[j] != 0 && G[j][track[j]] < min){
                k = j;
                min = G[j][track[j]];
            }
        }
        T[0][i] = k;
        T[1][i] = track[k];
        track[k] = 0;

        // Update track array to track min cost edges
        for (int j {1}; j<V; j++){
            if (track[j] != 0 && G[j][k] < G[j][track[j]]){
                track[j] = k;
            }
        }
    }
    PrintMST(T, G);
}

int main() {

    int cost [V][V] {
            {I, I, I, I, I, I, I, I},
            {I, I, 25, I, I, I, 5, I},
            {I, 25, I, 12, I, I, I, 10},
```

```
        {I, I, 12, I, 8, I, I, I},
        {I, I, I, 8, I, 16, I, 14},
        {I, I, I, I, 16, I, 20, 18},
        {I, 5, I, I, I, 20, I, I},
        {I, I, 10, I, 14, 18, I, I},
    };

    int n = sizeof(cost[0])/sizeof(cost[0][0]) - 1;

    PrimsMST(cost, n);

    return 0;
}
```

2. Write a program to create a binary search tree. Provide facilities to insert, delete, update and search nodes in the tree.

```
#include<stdlib.h>
#include<stdio.h>

struct bin_tree {
int data;
struct bin_tree * right, * left;
};
typedef struct bin_tree node;
void insert(node ** tree, int val)
{
    node *temp = NULL;
    if(!(*tree))
    {
        temp = (node *)malloc(sizeof(node));
        temp->left = temp->right = NULL;
        temp->data = val;
        *tree = temp;
        return;
```

```
    }
    if(val < (*tree)->data)
    {
        insert(&(*tree)->left, val);
    }
    else if(val > (*tree)->data)
    {
        insert(&(*tree)->right, val);
    }
}
void print_preorder(node * tree)
{
    if (tree)
    {
        printf("%d\n",tree->data);
        print_preorder(tree->left);
        print_preorder(tree->right);
    }
}
void print_inorder(node * tree)
{
    if (tree)
    {
        print_inorder(tree->left);
        printf("%d\n",tree->data);
        print_inorder(tree->right);
    }
}
void print_postorder(node * tree)
{
    if (tree)
    {
        print_postorder(tree->left);
        print_postorder(tree->right);
        printf("%d\n",tree->data);
```

```
    }
}
void deltree(node * tree)
{
    if (tree)
    {
        deltree(tree->left);
        deltree(tree->right);
        free(tree);
    }
}
node* search(node ** tree, int val)
{
    if(!(*tree))
    {
        return NULL;
    }
    if(val < (*tree)->data)
    {
        search(&((*tree)->left), val);
    }
    else if(val > (*tree)->data)
    {
        search(&((*tree)->right), val);
    }
    else if(val == (*tree)->data)
    {
        return *tree;
    }
}
void main()
{
    node *root;
    node *tmp;
    //int i;
```

```
    root = NULL;
    /* Inserting nodes into tree */
    insert(&root, 7);
    insert(&root, 4);
    insert(&root, 16);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 1);
    insert(&root, 9);

    /* Printing nodes of tree */
    printf("Pre Order Display\n");
    print_preorder(root);

    printf("In Order Display\n");
    print_inorder(root);

    printf("Post Order Display\n");
    print_postorder(root);

    /* Search node into tree */
    tmp = search(&root, 4);
    if (tmp)
    {
        printf("Searched node=%d\n", tmp->data);
    }
    else
    {
        printf("Data Not found in tree.\n");
    }
    /* Deleting all nodes of tree */
    deltree(root);
}
```

```
#include<stdlib.h>
#include<stdio.h>

struct bin_tree {
int data;
struct bin_tree * right, * left;
};
typedef struct bin_tree node;
void insert(node ** tree, int val)
{
    node *temp = NULL;
    if(!(*tree))
    {
        temp = (node *)malloc(sizeof(node));
        temp->left = temp->right = NULL;
        temp->data = val;
        *tree = temp;
        return;
    }
    if(val < (*tree)->data)
    {
        insert(&(*tree)->left, val);
    }
    else if(val > (*tree)->data)
    {
        insert(&(*tree)->right, val);
    }
}
void print_preorder(node * tree)
{
    if (tree)
    {
```

```c
        printf("%d\n",tree->data);
        print_preorder(tree->left);
        print_preorder(tree->right);
    }
}
void print_inorder(node * tree)
{
    if (tree)
    {
        print_inorder(tree->left);
        printf("%d\n",tree->data);
        print_inorder(tree->right);
    }
}
void print_postorder(node * tree)
{
    if (tree)
    {
        print_postorder(tree->left);
        print_postorder(tree->right);
        printf("%d\n",tree->data);
    }
}
void deltree(node * tree)
{
    if (tree)
    {
        deltree(tree->left);
        deltree(tree->right);
        free(tree);
```

```
    }
}
node* search(node ** tree, int val)
{
    if(!(*tree))
    {
        return NULL;
    }
    if(val < (*tree)->data)
    {
        search(&((*tree)->left), val);
    }
    else if(val > (*tree)->data)
    {
        search(&((*tree)->right), val);
    }
    else if(val == (*tree)->data)
    {
        return *tree;
    }
}
void main()
{
    node *root;
    node *tmp;
    //int i;
    root = NULL;
    /* Inserting nodes into tree */
    insert(&root, 7);
    insert(&root, 4);
```

```
    insert(&root, 16);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 1);
    insert(&root, 9);

    /* Printing nodes of tree */
    printf("Pre Order Display\n");
    print_preorder(root);

    printf("In Order Display\n");
    print_inorder(root);

    printf("Post Order Display\n");
    print_postorder(root);

    /* Search node into tree */
    tmp = search(&root, 4);
    if (tmp)
    {
        printf("Searched node=%d\n", tmp->data);
    }
    else
    {
        printf("Data Not found in tree.\n");
    }
    /* Deleting all nodes of tree */
    deltree(root);
}
```

Output

```
Pre Order Display
7
4
1
6
16
12
9
In Order Display
1
4
6
7
9
12
16
Post Order Display
1
6
4
9
12
16
7
Searched node=4

Process returned 1 (0x1)   execution time : 2.680 s
Press any key to continue.
```

3. Write algorithms to implement

A) Depth First Search Algorithm

B) Breadth First Search Algorithm

Code:

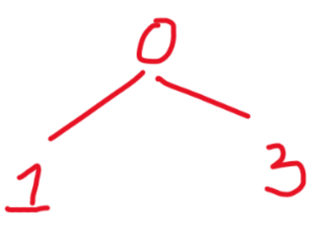
DFS for disconnected graph

```python
class Graph:
    def __init__(self,nVertices):
        self.nVertices = nVertices
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def removeEdge(self,v1,v2):  ## Before removing, check whether the
        if self.containsEdge(v1,v2) is False:
            return
        else:
            self.adjMatrix[v1][v2] = 0
            self.adjMatrix[v2][v1] = 0

```

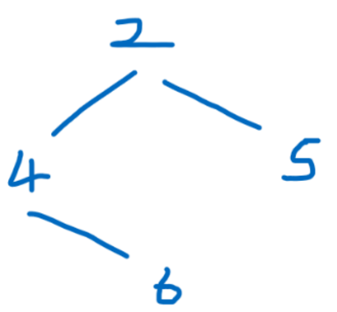
```
    def containsEdge(self,v1,v2):   ## if there is an edge,then it wil
        if self.adjMatrix[v1][v2]>0:
            return True
        else:
            return False

class DFS_diconnected(Graph):
    def __dfsHelper(self,sv,visited):  ## private class
        print(sv,end=' ')
        visited[sv] = True
        for i in range(self.nVertices):
            ## if there is an edge and that edge is not visited
            if (self.adjMatrix[sv][i]>0) and (visited[i] is False):
                self.__dfsHelper(i,visited)

    def dfs(self):
        cnt = 0  ## to maintain the count of number of disconnected gr
        visited = [False for i in range(self.nVertices)]  ## maintaini
        for i in range(self.nVertices):
            if visited[i] is False:  ## if that vertex is not at all v
                cnt+=1
                print("\nGraph - {}".format(cnt))
                self.__dfsHelper(i,visited)

    def __str__(self):
        return str(self.adjMatrix)

class Graph:
    def __init__(self,nVertices):
        self.nVertices = nVertices
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range
```

```python

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def removeEdge(self,v1,v2):  ## Before removing, check whether the
        if self.containsEdge(v1,v2) is False:
            return
        else:
            self.adjMatrix[v1][v2] = 0
            self.adjMatrix[v2][v1] = 0

    def containsEdge(self,v1,v2):   ## if there is an edge,then it wil
        if self.adjMatrix[v1][v2]>0:
            return True
        else:
            return False

    def __str__(self):
        return str(self.adjMatrix)


if __name__ == '__main__':
    obj1 = DFS_diconnected(7)
    obj1.addEdge(0,1)
    obj1.addEdge(0,3)

    obj1.addEdge(2,4)
    obj1.addEdge(2,5)
    obj1.addEdge(4,6)

    obj1.dfs()

    
```

Graph-1



Graph-2



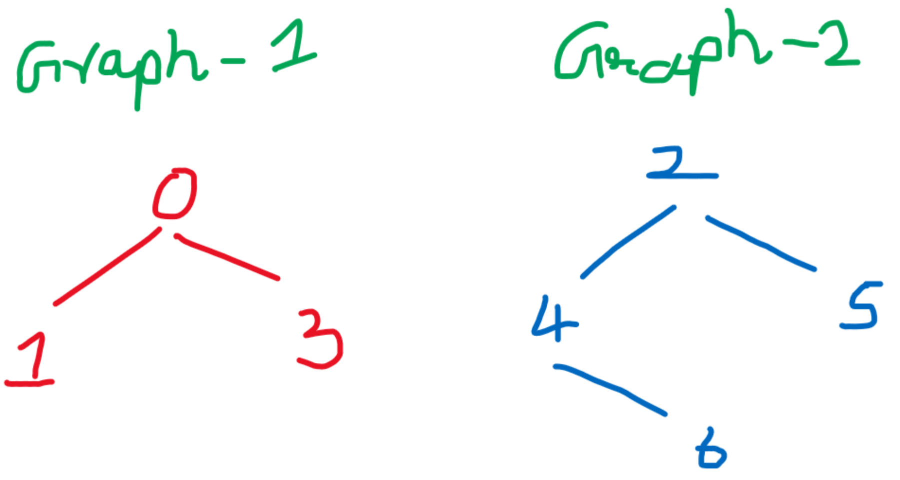
Output

```
Graph - 1
0 1 3
Graph - 2
2 4 6 5
***Repl Closed***
```

BFS for disconnected graph

```python
import queue

class Graph:
    def __init__(self,nVertices):
        self.nVertices = nVertices
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def removeEdge(self,v1,v2):  ## Before removing, check whether the
        if self.containsEdge(v1,v2) is False:
            return
        else:
            self.adjMatrix[v1][v2] = 0
            self.adjMatrix[v2][v1] = 0


    def containsEdge(self,v1,v2):   ## if there is an edge,then it wil
        if self.adjMatrix[v1][v2]>0:
            return True
        else:
            return False

    def __str__(self):
        return str(self.adjMatrix)

class BFS_disconnected(Graph):
    def __bfsHelper(self,sv,visited):
        q = queue.Queue()

        q.put(sv)   # intially pushing 0 into the queue
```

```python
        visited[sv] = True  # and 0 is visited

        while q.empty() is False:
            u = q.get() ## After Dequeue,start exploring all the verti
            print(u,end=' ')

            for v in range(self.nVertices):  ## if a vertex is there a
                if (self.adjMatrix[u][v]>0 and visited[v] is False):
                    q.put(v)
                    visited[v] = True

    def bfs(self):
        cnt = 0  ## to maintain the count of number of disconnected gr
        visited = [False for i in range(self.nVertices)]  ## maintaini
        for i in range(self.nVertices):
            if visited[i] is False:  ## if that vertex is not at all v
                cnt+=1
                print("\nGraph - {}".format(cnt))
                self.__bfsHelper(i,visited)

    def __str__(self):
        return str(self.adjMatrix)

if __name__ == '__main__':
    obj1 = BFS_disconnected(7)
    obj1.addEdge(0,1)
    obj1.addEdge(0,3)

    obj1.addEdge(2,4)
    obj1.addEdge(2,5)
    obj1.addEdge(4,6)

    obj1.bfs()
```



Output:

```
Graph - 1
0 1 3
Graph - 2
2 4 5 6
***Repl Closed***
```

For directed graph:

- **Approach:** Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.
- **Algorithm:**
    1. Create a recursive function that takes the index of node and a visited array.
    2. Mark the current node as visited and print the node.
    3. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

For undirected graph:

- **Approach:** This will happen by handling a corner case.
  The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex as in the case of a Disconnected graph. To do complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS.
  The recursive function remains the same.
- **Algorithm:**
    1. Create a recursive function that takes the index of node and a visited array.
    2. Mark the current node as visited and print the node.

3. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.
4. Run a loop from 0 to number of vertices and check if the node is unvisited in previous DFS then call the recursive function with current node.

B) Breadth First Search Algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

BFS pseudocode

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited) neighbours of u
```

4. Compare Ford-Fulkerson Method and Edmonds-Karp algorithm and explain them.

FORD-FULKERSON METHOD:

Ford-Fulkerson algorithm is a greedy approach for calculating the maximum possible flow in a network or a graph. A term, flow network, is used to describe a network of vertices and edges with a source (S) and a sink (T). Each vertex, except S and T, can receive and send an equal amount of stuff through it. S can only send and T can only receive stuff.

ALGORITHM:

1. Initialize the flow in all the edges to 0.
2. While there is an augmenting path between the source and the sink, add this path to the flow.
3. Update the residual graph.

EDMOND-KARP ALGORITHM:

The Edmonds-Karp Algorithm is a specific implementation of the Ford-Fulkerson algorithm. Like Ford-Fulkerson, Edmonds-Karp is also an algorithm that deals with the max-flow min-cut problem. Edmonds-Karp is identical to Ford-Fulkerson except for one very important trait. The search order of augmenting paths is well defined. As a refresher from the Ford-Fulkerson wiki, augmenting paths, along with residual graphs, are the two important concepts to understand when finding the max flow of a network.

Augmenting paths are simply any path from the source to the sink that can currently take more flow. Over the course of the algorithm, flow is monotonically increased. So, there are times when a path from the source to the sink can take on more flow, and that is an augmenting path.

ALGORITHM:

1. The flow is initially zero and the initial residual capacity array is all zeroes.
2. the outer loop executes until there are no more paths from the source to the sink in the residual graph.
3. Inside the loop breadth first search algorithm is implemented.
4. If there found a path with residual capacity m add it to current maximum flow.
5. Backtracking through the network.
6. Updates the residual flow matrix to reflect the newly found augmenting path capacity
7. loop is repeated until it reaches the source vertex.

COMPARISON:

1. Ford-Fulkerson is sometimes called a method because some parts of its protocol are left unspecified. Edmonds-Karp, on the other hand, provides a full specification. Most importantly, it specifies that breadth-first search should be used to find the shortest paths during the intermediate stages of the program.

2. Edmonds-Karp improves the runtime of Ford-Fulkerson.This improvement is important because it makes the runtime of Edmonds-Karp independent of the maximum flow of the network.

3. The complexity can be given independently of the maximal flow. The algorithm runs in O(VE2) time, even for irrational capacities. The intuition is, that every time we find an augmenting path one of the edges becomes saturated, and the distance from the edge to s will be longer, if it appears later again in an augmenting path. And the length of a simple paths is bounded by V.