$\frac{100}{100}$

20 | 20

```
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-0.cpp

/*Let V be the number of vertices and E be the number of Edges*/

#include <iostream>
#include <stdio.h>
#include <queue>
using namespace std;

class Graph{

public:

    //Contructor
    Graph(int num){

        //num is assigned to size
        this->Size = num;

        //Adjacency matrix is assigned to be all 0's
        AdjMat = new int* [Size];

        //Boolean Visited array for depth first search is initialised to be all false
        visited = new bool [Size];

        for (int i = 0; i < Size; i++){

            AdjMat[i] = new int[Size];

            for (int j = 0; j < Size; j++)
                AdjMat[i][j] = 0;
        }

        for (int i = 0; i < Size; i++)
            visited[i] = false;
    }

    //inserts an edge in the graph
    //Time Complexity: O(1) since there are a constant amount of instructions
    void insertEdge(int Vertex1, int Vertex2){

        //The case where the vertices entered are greater than or equal to the Size
        if (Vertex1 >= Size || Vertex2 >= Size){

            cout << "You cannot go above the capacity!\n";
            return;
        }

        //The case where the edge already exists
        if (AdjMat[Vertex1][Vertex2] == 1 && AdjMat[Vertex2][Vertex1] == 1){

            cout << Vertex1 << " and " << Vertex2 << " are already connected!\n";
            return;
```

```
        }

        //Assigns an edge to the both vertices
        AdjMat[Vertex1][Vertex2] = 1;
        AdjMat[Vertex2][Vertex1] = 1;

        printf("(%d) --> (%d)\n", Vertex1, Vertex2);
    }

    //Removes an edges from the graph
    //Time Complexity: O(1) Since there is a constant number of instructions
    void removeEdge(int Vertex1, int Vertex2){

        //The case where the vertices entered are greater than or equal to the Size
        if (Vertex1 > Size || Vertex2 > Size){

            cout << "No such vertex exists!\n";
            return;
        }

        //The case where the edge already does not exist
        if (AdjMat[Vertex1][Vertex2] == 0 && AdjMat[Vertex2][Vertex1] == 0){

            cout << Vertex1 << " and " << Vertex2 << " were never connected!\n";
            return;
        }

        //Removes the edge by assigning zero to the vertices
        AdjMat[Vertex1][Vertex2] = 0;
        AdjMat[Vertex2][Vertex1] = 0;

        printf("(%d) -/-> (%d)\n", Vertex1, Vertex2);
    }

    //Prints the vertix and all its connected vertices
    //Time Complexity: O(n^2) since there is a nested loop
    void printGraph(){

        //Goes through the matrix and prints all the vertices connected by the vertix i
        for (int i = 0; i < Size; i++){

            cout << "(" << i << ") --> ";
            for (int j = 0; j < Size; j++)
                if (AdjMat[i][j] == 1){

                    cout << j << " ; ";
                }

            cout << endl;
        }
    }

    //Prints the graph as a depth first search tree
    //Time Complexity: O(V + E) Since it goes through all the vertices and edges
    void DFS(int start){
```

```cpp

        //The starting vertex is marked as visited and is printed
        visited[start] = true;
        cout << start << " ";

        //Goes through the rest of the vertices
        for (int i = 0; i < Size; i++)

            //If the current vertex is not 0 and is not marked as visited,
            //then the current vertex is treated as the starting vertex and the function is c
            if (AdjMat[start][i] == 1 && visited[i] == false)
                DFS(i);

    }

    //Prints the graph as a breadth first search tree
    //Time Complexity: O(V + E) Since it goes through all the vertices and edges
    void BFS(int start){

        //Boolean visited array to keep track on which vertices have been visited.
        //It is initialised to false
        bool isVisited[Size];
        for (int i = 0; i < Size; i++)
            isVisited[i] = false;

        //Queue to keep the loop going and to print the vertices as a BFS traversal
        queue<int> Q;

        //The starting vertex is pushed and is marked as visited
        Q.push(start);
        isVisited[start] = true;

        //Goes through the rest of the graph and prints it as a breadth first search tree
        while (!Q.empty()){

            //Goes through the graph and checks if the queue front's neighbors are not zero a
            for (int i = 0; i < Size; i++)

                //The case where the queue front's neighbors are not zero and have not been v
                if (AdjMat[Q.front()][i] == 1 && isVisited[i] == false){

                    //The current vertex is pushed into the queue and is marked as visited
                    Q.push(i);
                    isVisited[i] = true;
                }

        for (int i = 0; i < Size; i++)ed then popped
            cout << Q.front() << " ";
            Q.pop();
        }
    }

private:

    int **AdjMat;
```

```
    bool *visited;
    int Size;
};



int main(){

    Graph g(6);

    g.insertEdge(0,1);
    g.insertEdge(0,2);
    g.insertEdge(0,3);
    g.insertEdge(1,5);
    g.insertEdge(2,3);
    g.insertEdge(2,4);

    cout << "Printing graph:\n";
    g.printGraph();

    cout << "printing dfs: ";
    g.DFS(0); cout << "\n\n";

    cout << "printing bfs: ";
    g.BFS(0);

    return 0;

}
```

25 |25

```
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-1.cpp

/*Let n be the size of the matrix row or column*/

#include <iostream>
#include <list>
using namespace std;
//Size of the matrix
#define S 7

//Solves a given maze. Returns "NO SOLUTION" if there isn't any
//Time Complexity: O(n^2) Because the worst case scenario is a matrix with all 1's
void MazeSolver(int Matrix[][S], int Size){

    //Matrix that keeps track of which squares have been visited
    bool isVisited[Size][Size] = {false};

    //Lists that portray stacks. One stores the row index, and the other stores the colum
    //Lists are used for their iterative and popping from the back features
    list<int> StackRow, StackCol;
    int i = 0, j = 0; //i and j keep track of where the maze pointer is positioned
    int cost = 0; //Keeps track of the number of steps taken

    //StackRow and StackCol have 0 pushed as the starting place
    StackRow.push_back(i);
    StackCol.push_back(j);

    //(0,0) is marked as visited
    isVisited[i][j] = true;


    //The maze pointer begins to move. It's main goal is to find a path to the end
    //If there is no path to the end or the number of steps take less than 2*Size + 3
    //Then the function returns "NO SOLUTION" and exits
    while (true){

        //Checks if the two stacks are empty. If they are, then there is no solution.
        if (StackCol.empty() && StackRow.empty()){

            cout << "NO SOUTION!\n";
            return;
        }

        //Checks if the last element entered in both stacks are at the end
        if (StackRow.back() == Size - 1 && StackCol.back() == Size - 1){

            //If the number of steps is greater than 2*Size + 3, then the function has succes
            if (cost >= (2 * Size) + 4 || cost <= (2 * Size) + 10)
                break;

            //If the cost is less than 2*Size + 3, then the function returns "NO SOLUTION" an
            else{

```

```
                cout << "NO SOUTION!\n";
                return;
            }
        }

        //Checks if the right of the current position is not zero
        //and if moving one unit to the right does not exit the maze boundry
        //and if the position to the right not marked as visited
        //If so, then the maze pointer moves one unit to the right
        if (Matrix[i][j + 1] != 0 && j + 1 < Size && !isVisited[i][j + 1]){

            j++; //j is incremented by 1
            isVisited[i][j] = true; //Current position is marked as visited

            //The row index and and the column index have been pushed into their respective s
            StackRow.push_back(i);
            StackCol.push_back(j);
            cost++; //The number of steps is incremented
        }

        //Checks if the left of the current position is not zero
        //and if moving one unit to the left does not exit the maze boundry
        //and if the position to the left not marked as visited
        //If so, then the maze pointer moves one unit to the left
        else if (Matrix[i][j - 1] != 0 && j - 1 >= 0 && !isVisited[i][j - 1]){

            j--; //j is decremented by 1
            isVisited[i][j] = true; //Current position is marked as visited

            //The row index and and the column index have been pushed into their respective s
            StackRow.push_back(i);
            StackCol.push_back(j);
            cost++;//The number of steps is incremented
        }

        //Checks if the position above the maze pointer is not zero
        //and if moving one unit up does not exit the maze boundry
        //and if the position above the current position not marked as visited
        //If so, then the maze pointer moves one unit up
        else if (Matrix[i - 1][j] != 0 && i - 1 >= 0 && !isVisited[i - 1][j]){

            i--; //i is decremented by 1
            isVisited[i][j] = true; //Current position is marked as visited

            //The row index and and the column index have been pushed into their respective s
            StackRow.push_back(i);
            StackCol.push_back(j);
            cost++;//The number of steps is incremented
        }

        //down
        //Checks if the position under the maze pointer is not zero
        //and if moving one unit down does not exit the maze boundry
        //and if the position under the current position not marked as visited
        //If so, then the maze pointer moves one unit up
```

```
        else if (Matrix[i + 1][j] != 0 && i + 1 < Size && !isVisited[i + 1][j]){

            i++; //i is incremented by 1
            isVisited[i][j] = true; //Current position is marked as visited

            //The row index and and the column index have been pushed into their respective s
            StackRow.push_back(i);
            StackCol.push_back(j);
            cost++;//The number of steps is incremented
        }

        //If the maze pointer reached a dead end, then it back-tracks one position at a time
        else{

            //Both stacks are popped
            StackCol.pop_back();
            StackRow.pop_back();

            //i and j are assigned to the current top, which is the previous position
            i = StackRow.back();
            j = StackCol.back();

            cost--; //The number of steps is decremented by 1
        }
    }

    //iterators for StackRow and StackCol
    list<int>::iterator itRow = StackRow.begin();
    list<int>::iterator itCol = StackCol.begin();

    //Prints the starting position then increments the iterators
    printf("(%d,%d)", *itRow, *itCol); itRow++; itCol++;

    //Goes through the rest of both lists and prints the positions
    while (itRow != StackRow.end() && itCol != StackCol.end()){

        printf(" -> (%d,%d)", *itRow, *itCol);

        //Both iterators are incremented
        itRow++;
        itCol++;
    }

    cout << "\n\nThe number of steps: " << cost << endl;
}

//Prints the matrix
//Time Complexity: O(n^2) since there is a nested loop
void//If there is no path to the end or the number of steps take less than 2*Size + 3

    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++)
            cout << arr[i][j] << " ";

        cout << endl;
```

```
    }
    cout << endl;
}

//Beginning of program
int main(){

    int Matrix[S][S] = {{1,1,1,1,0,1,0},
                        {0,0,0,1,1,1,1},
                        {1,1,1,0,0,0,1},
                        {1,0,1,1,1,1,1},
                        {1,0,0,0,0,0,0},
                        {1,0,0,1,1,1,0},
                        {1,1,1,1,0,1,1}
                       };

    PrintMatrix(Matrix, S);

    MazeSolver(Matrix, S);

    return 0;
}
//End of Program
```

10/10

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//Problem-2.cpp

/*Let n be the number of rows or columns in the matrix*/

#include <iostream>
#include <stdio.h>
using namespace std;

class Graph{

public:

    //Constructor
    Graph(int num): Size(num)/*num is assigned to size*/
    {

        //The adjacency matrix is initialised to be all zeros
        AdjMat = new int* [Size];

        for (int i = 0; i < Size; i++){

            AdjMat[i] = new int[Size];

            for (int j = 0; j < Size; j++)
                AdjMat[i][j] = 0;
        }
    }

    //inserts an edge in the graph
    //Time Complexity: O(1) since there are a constant amount of instructions
    void insertEdge(int Vertex1, int Vertex2){

        //The case where the vertices entered are greater than or equal to the Size
        if (Vertex1 >= Size || Vertex2 >= Size){

            cout << "You cannot go above the capacity!\n";
            return;
        }

        //The case where the edge already exists
        if (AdjMat[Vertex1][Vertex2] == 1 && AdjMat[Vertex2][Vertex1] == 1){

            cout << Vertex1 << " and " << Vertex2 << " are already connected!\n";
            return;
        }

        //Assigns an edge to the both vertices
        AdjMat[Vertex1][Vertex2] = 1;
        AdjMat[Vertex2][Vertex1] = 1;

        printf("(%d) --> (%d)\n", Vertex1, Vertex2);
    }

```

```
//Removes an edges from the graph
//Time Complexity: O(1) Since there is a constant number of instructions
void removeEdge(int Vertex1, int Vertex2){

    //The case where the vertices entered are greater than or equal to the Size
    if (Vertex1 >= Size || Vertex2 >= Size){

        cout << "No such vertex exists!\n";
        return;
    }

    //The case where the edge already does not exist
    if (AdjMat[Vertex1][Vertex2] == 0 && AdjMat[Vertex2][Vertex1] == 0){

        cout << Vertex1 << " and " << Vertex2 << " were never connected!\n";
        return;
    }

    //Removes the edge by assigning zero to the vertices
    AdjMat[Vertex1][Vertex2] = 0;
    AdjMat[Vertex2][Vertex1] = 0;

    printf("(%d) -/-> (%d)\n", Vertex1, Vertex2);
}

//Prints the vertix and all its connected vertices
//Time Complexity: O(n^2) since there is a nested loop
void printGraph(){

    //Goes through the matrix and prints all the vertices connected by the vertix i
    for (int i = 0; i < Size; i++){

        cout << "(" << i << ") --> ";

        for (int j = 0; j < Size; j++)

            if (AdjMat[i][j] == 1){

                cout << j << " ; ";
            }

        cout << endl;
    }
}

//Checks if the graph is complete. Returns true if it is complete. Otherwise, returns
//Time Complexity: O(n^2) since there is a nested loop
bool isComplete(){

    for (int i = 0; i < Size - 1; i++)
        for (int j = i + 1; j < Size; j++)
            if (AdjMat[i][j] == 0)
                return false;

    return true;
```

```

    }

private:

    int **AdjMat;
    int Size;
};

//Beginning of program
int main(){

    Graph g(5);

    g.insertEdge(0,1);
    g.insertEdge(0,2);
    g.insertEdge(0,3);
    g.insertEdge(0,4);
    g.insertEdge(1,2);
    g.insertEdge(1,3);
    g.insertEdge(1,4);
    g.insertEdge(2,3);
    g.insertEdge(2,4);
    g.insertEdge(3,4);

    cout << "\nprinting graph: \n";
    g.printGraph();

    cout << "\n\nIs the graph complete? The answer is: ";
    if (g.isComplete())
        cout << "True!\n";
    else
        cout << "False!\n";

    return 0;
}
//End of program
```

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-3.cpp

#include <iostream>
#include <stdio.h>
#include <queue>
using namespace std;

class Graph{

public:

    //Constructor
    Graph(int num): Size(num) /*num is assigned to size*/
    {

        //The adjacency matrix is initialised to be all zeros
        AdjMat = new int* [Size];

        for (int i = 0; i < Size; i++){

            AdjMat[i] = new int [Size];
            for (int j = 0; j < Size; j++)
                AdjMat[i][j] = 0;
        }
    }

    //inserts an edge in the graph
    //Time Complexity: O(1) since there are a constant amount of instructions
    void insertEdge(int V1, int V2, int W){

        //The case where the vertices entered are greater than or equal to the Size
        if (V1 >= Size || V2 >= Size){

            cout << "Cannot exceed capacity!\n";
            return;
        }

        //The case where the edge already exists
        if ( AdjMat[V1][V2] != 0 && AdjMat[V2][V1] != 0){

            cout << "This edge already exists!\n";
            return;
        }

        //Assigns an edge to the both vertices
        AdjMat[V1][V2] = W;
        AdjMat[V2][V1] = W;
    }

    //Prints the vertix and all its connected vertices
    //Time Complexity: O(n^2) since there is a nested loop
    void printGraph(){

```

```
        //Goes through the matrix and prints all the vertices connected by the vertix i
        for (int i = 0; i < Size; i++){

            cout << "(" << i << ") --> ";

            for (int j = 0; j < Size; j++)

                if (AdjMat[i][j] != 0){

                    cout << j << " ; ";
                }

            cout << endl;
        }
    }

    //Removes an edges from the graph
    //Time Complexity: O(1) Since there is a constant number of instructions
    void removeEdge(int V1, int V2){

        //The case where the vertices entered are greater than or equal to the Size
        if (V1 >= Size || V2 >= Size){

            cout << "Cannot exceed capacity!\n";
            return;
        }

        //The case where the edge already does not exist
        if ( AdjMat[V1][V2] == 0 && AdjMat[V2][V1] == 0){

            cout << "This edge already doesn't exists!\n";
            return;
        }

        //Removes the edge by assigning zero to the vertices
        AdjMat[V1][V2] = 0;
        AdjMat[V2][V1] = 0;
    }

    //Finds the smallest edge while marking it as visited . If there is none, then it ret
    //Time Complexity: O(n) since the row length is variable
    int findSmallestEdge(bool arr[], int Row){

        //Min is to find the minimum and chosen_index is to find the index with the minimum v
        int Min = 999, Chosen_Index = -1;

        //Goes through the matrix row to find the minimum unmarked value
        for (int i = 0; i < Size; i++)

            if (AdjMat[Row][i] != 0 && !arr[i] && AdjMat[Row][i] < Min){

                Min = AdjMat[Row][i];
                Chosen_Index = i;
            }

```

```
        //Returns the minimum value.
        return Chosen_Index;
    }

    //Finds the shortest path from start to all nodes
    //Time Complexity: O(n^2) since the loop will iterate n times and findSmallestEdge wi
    void ShortestPaths(int start){

        /* Parent array is to keep track of the parents
           Dist array is to keep track of the minimum distance
           Visited array is to keep track of which vertices have been visited
           Queue is to push the unvisited and pop the visited vertices and also to ke
        */

        int Parent[Size];
        int Dist[Size];
        bool Visited[Size];
        queue<int> Q;

        //Initialising all the arrays
        for (int i = 0; i < Size; i++){

            Parent[i] = -1;
            Dist[i] = 9999;
            Visited[i] = false;
        }

        //pushing the start vertex into the queue and the length of the path from the start t
        Dist[start] = 0;
        Q.push(start);

        //Goes through all the vertices and finds the shortest path to each
        while (!Q.empty()){

            //Boolean array to keep track of all the visited vertices on the current row
            bool subVisited[Size] = {false};

            //gets the minimum unmarked value index
            int MinValIndex = 0;


            //Finds the edges in ascending order and picks the shortest path while updating t
            //It stops when MinValIndex is -1
            do{

                //Finds the smallest unmarked edge in the current row
                MinValIndex = findSmallestEdge(subVisited, Q.front());

                //If not all edges are unmarked, then continue
                if (MinValIndex != -1){

                    //current row index is marked as true
                    subVisited[MinValIndex] = true;

                    //If the minimum edge from the current row is less than or equal to its d
```

```
                    //then the new distance to the minimum vertex is updated to be the distan
                    //the minimum edge from the current row
                    //Its parent will be updated to be the current row and it will be pushed
                    if (AdjMat[Q.front()][MinValIndex] + Dist[Q.front()] <= Dist[MinValIndex]

                        Dist[MinValIndex] = AdjMat[Q.front()][MinValIndex] + Dist[Q.front()];
                        Parent[MinValIndex] = Q.front();
                        Q.push(MinValIndex);
                    }
                }

            }while (MinValIndex != -1);

            //Marks the current row as visited and pops it from the queue
            Visited[Q.front()] = true;
            Q.pop();
        }

        //prints the path from a vertex to the starting vertex
        for (int i = Size - 1; i >= 0; i--){

            int stopping_Cond = i;

            cout << "The shortest path from " << i << " to " << start << " is: " << i;
            while (true){

                if (Parent[stopping_Cond] == -1)
                    break;

                else{

                    cout << " --> " << Parent[stopping_Cond];
                    stopping_Cond = Parent[stopping_Cond];
                }
            }

            cout << endl;
        }

    }

private:

    int** AdjMat;
    int Size;
};

//Beginning of main
int main(){

    Graph g(8);

    g.insertEdge(0,1,3);
    g.insertEdge(1,2,7);
    g.insertEdge(1,3,2);
```

```
    g.insertEdge(2,4,1);
    g.insertEdge(2,7,6);
    g.insertEdge(2,3,2);
    g.insertEdge(3,5,4);
    g.insertEdge(3,6,3);

    g.printGraph();

    cout << "\n\nShortest path: \n\n";

    g.ShortestPaths(0);

    return 0;
}
//End of main
```

20/20

```
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//Problem-4

/*Let n be the size of the adjacency list, and m be the size of the linked lists*/

#include <iostream>
#include <stdio.h>
#include <list>
#include <queue>
using namespace std;

//Searches for an element in a given linked list. Returns true if it is found, otherw
//Time Complexity: O(m) since the size of the linked list is variable
bool search(list<int> List, int data){

    //If the linked list is empty, it returns false
    if (List.empty())
        return false;

    //Iterator that goes through the list
    list<int>::iterator it;

    //Goes through the list and checks if the element exists. Returns true if it does
    for (it = List.begin(); it != List.end(); it++)
        if (*it == data)
            return true;

    //If the loop stops, then the element does not exist. In that case, returns false
    return false;
}

//Removes an element from a given list
//Time Complexity: O(m) since the size of the linked list is variable
void Remove(list<int>& List, int data){

    //If the list is empty, then the function does nothing and exists.
    if (List.empty())
        return;

    //Iterator that goes through the list
    list<int>::iterator it;

    //Goes through the list and checks if the element exists. If it does, then it removes
    for (it = List.begin(); it != List.end(); it++)
        if (*it == data){

            List.erase(it);
            return;
        }
}

class Graph{

public:
```

```
    //Constructor
    Graph(int num){

        //num is assigned to Size
        this->Size = num;

        //Adjacency list is defined
        AdjList = new list<int> [Size];

        //Weight matrix initialised to 0's
        Weight = new int* [Size];
        for (int i = 0; i < Size; i++){

            Weight[i] = new int [Size];

            for (int j = 0; j < Size; j++)
                Weight[i][j] = 0;
        }
    }

    //Inserts an edge to the graph
    //Time Complexity: O(1) since there is a constant number of instructions
    void InsertEdge(int V1, int V2, int W){

        //The case where the vertices entered are greater than or equal to the Size
        if (V1 >= Size || V2 >= Size){

            cout << "Cannot go beyond the capacity!\n";
            return;
        }

        //The case where the edge already exists
        if (search(AdjList[V1],V2) && search(AdjList[V2],V1)){

            cout << "The two verteces are already connected!\n";
            return;
        }

        //Both vertices are pushed
        AdjList[V1].push_back(V2);
        AdjList[V2].push_back(V1);

        //Both edges are assigned their weights
        Weight[V1][V2] = W; Weight[V2][V1] = W;

        printf("(%d) --> (%d)\n", V1, V2);
    }

    //Removes an edge from the graph
    //Time Complexity: O(1) Since there is a constant amount of instructions
    void RemoveEdge(int V1, int V2){

        //The case where the vertices entered are greater than or equal to the Size
        if (V1 >= Size || V2 >= Size){
```

```
            cout << "Cannot go beyond the capacity!\n";
            return;
        }
        //The case where the edge already does not exist
        if (!search(AdjList[V1],V2) && !search(AdjList[V2],V1)){

            cout << "The two verteces were never connected!\n";
            return;
        }

        //Removes the edge
        Remove(AdjList[V1], V2);
        Remove(AdjList[V2], V1);

        //The edge's weight is assigned to zero
        Weight[V1][V2] = 0; Weight[V2][V1] = 0;

        printf("(%d) -/-> (%d)\n", V1, V2);
    }

    //Prints the vertix and all its connected vertices
    //Time Complexity: O(n*m) since it's going through the array while going through the
    void printGraph(){

        for (int i = 0; i < Size; i++){

            printf("(%d) --> ", i);

            list<int>::iterator it;
            for(it = AdjList[i].begin(); it != AdjList[i].end(); it++)
                cout << *it << " ; ";

            cout << endl;
        }
    }
    //Finds the minimum spanning tree and prints its cost
    //Time Complexity: O(n^2) since it goes through the list of elements while going thro
                            //and goes through the matrix while printing the graph
    void MinimumSpanningTree(int start){

        //Adjacency list for the resultant tree (only used for printing)
        list<int> List[Size];

        //Boolean array to keep track of vertices that have been visited. It is initialised t
        bool isVisited[Size];
        for (int i = 0; i < Size; i++)
            isVisited[i] = false;

        //A list to keep track of all the vertices with the lightest weights
        list<int> Verticies;

        //Queue to make the loop continue and to print all the vertices
        queue<int> Q;
```

```
        /*Min is to find out which vertex has the minimum weight
          Chosen_Vertix is used to store the vertex with the lowest weight
          Current_Row is the row where Chosen_Vertix lies
          Cost is the cost of the minimum spanning tree
        */
        int Min, Chosen_Vertix, Cost = 0, Current_Row;

        //The starting vertex is pushed into the queue and the list. It is also marked as vis
        Verticies.push_back(start);
        Q.push(start);
        isVisited[start] = true;

        //Goes through the graph to find and determine the minimum spanning tree and its cost
        while (!Q.empty()){

            //Min and Chose_Verix are both initialised at the start of each loop
            Min = 999;
            Chosen_Vertix = -1;


            /*
                The iterator it goes through all the elements and their neighbors to
                which edge has the lowest weight.
                It also has to be from a vertex that has been unvisited
            */
            list<int>::iterator it = Verticies.begin();
            for (; it != Verticies.end(); it++){
                list<int>::iterator it2 = AdjList[*it].begin();
                for (; it2 != AdjList[*it].end(); it2++){

                    //The case where the lowest weight is found
                    if (Weight[*it][*it2] != 0 && Weight[*it][*it2] <= Min && !isVisited[*it2

                        //Min is changed to not affect the if statement
                        Min = Weight[*it][*it2];

                        //it and it2 are assigned to Current_Row and Chosen_Verix
                        Chosen_Vertix = *it2;
                        Current_Row = *it;
                    }
                }
            }

            //If Chosen_Verix is changed (meaning a new lowest weight was found) then mark Ch
            //Push it into both the queue and the vericies list
            //and add the weight of the current edge to the cost
            if (Chosen_Vertix != -1){

                isVisited[Chosen_Vertix] = true;
                Q.push(Chosen_Vertix);
                Verticies.push_back(Chosen_Vertix);
                Cost += Weight[Current_Row][Chosen_Vertix];

                //Setting the current row and column to 1
                List[Current_Row].push_back(Chosen_Vertix);
```

```
                List[Chosen_Vertix].push_back(Current_Row);
            }

            //The queue is popped
            Q.pop();
        }


        //Prints the minimum spanning tree
        for (int i = 0; i < Size; i++){

            list<int>::iterator it = List[i].begin();

            printf("(%d) --> ", i);
            for (; it != List[i].end(); it++)
                cout << *it << " ; ";

            cout << endl;
        }

        //Prints the cost
        cout << "\nThe cost is : " << Cost << endl;

    }

private:

    list<int>* AdjList;
    int **Weight;
    int Size;


};

int main(){

    Graph g(7);

    g.InsertEdge(0,1,3);
    g.InsertEdge(0,2,4);
    g.InsertEdge(0,6,7);
    g.InsertEdge(1,5,2);
    g.InsertEdge(2,3,5);
    g.InsertEdge(2,4,6);
    g.InsertEdge(2,5,1);
    g.InsertEdge(4,5,4);

    cout << "\nPrinting graph: " << endl;

    g.printGraph();

    cout << "\nMinimum spanning tree: " << endl;
    g.MinimumSpanningTree(0);
}
```