

100  
100

20/20

```
1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //problem-0.cpp
4:
5: /*Let V be the number of vertices and E be the number of Edges*/
6:
7: #include <iostream>
8: #include <stdio.h>
9: #include <queue>
10: using namespace std;
11:
12: class Graph{
13:
14: public:
15:
16:     //Constructor
17:     Graph(int num){
18:
19:         //num is assigned to size
20:         this->Size = num;
21:
22:         //Adjacency matrix is assigned to be all 0's
23:         AdjMat = new int* [Size];
24:
25:         //Boolean Visited array for depth first search is initialised to be all false
26:         visited = new bool [Size];
27:
28:         for (int i = 0; i < Size; i++){
29:
30:             AdjMat[i] = new int[Size];
31:
32:             for (int j = 0; j < Size; j++)
33:                 AdjMat[i][j] = 0;
34:         }
35:
36:         for (int i = 0; i < Size; i++)
37:             visited[i] = false;
38:     }
39:
40:     //inserts an edge in the graph
41:     //Time Complexity: O(1) since there are a constant amount of instructions
42:     void insertEdge(int Vertex1, int Vertex2){
43:
44:         //The case where the vertices entered are greater than or equal to the Size
45:         if (Vertex1 >= Size || Vertex2 >= Size){
46:
47:             cout << "You cannot go above the capacity!\n";
48:             return;
49:         }
50:
51:         //The case where the edge already exists
52:         if (AdjMat[Vertex1][Vertex2] == 1 && AdjMat[Vertex2][Vertex1] == 1){
53:
54:             cout << Vertex1 << " and " << Vertex2 << " are already connected!\n";
55:             return;
```

```

56:     }
57:
58:     //Assigns an edge to the both vertices
59:     AdjMat[Vertex1][Vertex2] = 1;
60:     AdjMat[Vertex2][Vertex1] = 1;
61:
62:     printf("(%d) --> (%d)\n", Vertex1, Vertex2);
63: }
64:
65: //Removes an edges from the graph
66: //Time Complexity: O(1) Since there is a constant number of instructions
67: void removeEdge(int Vertex1, int Vertex2){
68:
69:     //The case where the vertices entered are greater than or equal to the Size
70:     if (Vertex1 > Size || Vertex2 > Size){
71:
72:         cout << "No such vertex exists!\n";
73:         return;
74:     }
75:
76:     //The case where the edge already does not exist
77:     if (AdjMat[Vertex1][Vertex2] == 0 && AdjMat[Vertex2][Vertex1] == 0){
78:
79:         cout << Vertex1 << " and " << Vertex2 << " were never connected!\n";
80:         return;
81:     }
82:
83:     //Removes the edge by assigning zero to the vertices
84:     AdjMat[Vertex1][Vertex2] = 0;
85:     AdjMat[Vertex2][Vertex1] = 0;
86:
87:     printf("(%d) -/-> (%d)\n", Vertex1, Vertex2);
88: }
89:
90: //Prints the vertex and all its connected vertices
91: //Time Complexity: O(n^2) since there is a nested loop
92: void printGraph(){
93:
94:     //Goes through the matrix and prints all the vertices connected by the vertex i
95:     for (int i = 0; i < Size; i++){
96:
97:         cout << "(" << i << ") --> ";
98:         for (int j = 0; j < Size; j++)
99:             if (AdjMat[i][j] == 1){
100:
101:                 cout << j << " ";
102:             }
103:
104:         cout << endl;
105:     }
106: }
107:
108: //Prints the graph as a depth first search tree
109: //Time Complexity: O(V + E) Since it goes through all the vertices and edges
110: void DFS(int start){

```

```

111:
112:     //The starting vertex is marked as visited and is printed
113:     visited[start] = true;
114:     cout << start << " ";
115:
116:     //Goes through the rest of the vertices
117:     for (int i = 0; i < Size; i++)
118:
119:         //If the current vertex is not 0 and is not marked as visited,
120:         //then the current vertex is treated as the starting vertex and the function is c
121:         if (AdjMat[start][i] == 1 && visited[i] == false)
122:             DFS(i);
123:
124:     }
125:
126:     //Prints the graph as a breadth first search tree
127:     //Time Complexity: O(V + E) Since it goes through all the vertices and edges
128:     void BFS(int start){
129:
130:         //Boolean visited array to keep track on which vertices have been visited.
131:         //It is initialised to false
132:         bool isVisited[Size];
133:         for (int i = 0; i < Size; i++)
134:             isVisited[i] = false;
135:
136:         //Queue to keep the loop going and to print the vertices as a BFS traversal
137:         queue<int> Q;
138:
139:         //The starting vertex is pushed and is marked as visited
140:         Q.push(start);
141:         isVisited[start] = true;
142:
143:         //Goes through the rest of the graph and prints it as a breadth first search tree
144:         while (!Q.empty()){
145:
146:             //Goes through the graph and checks if the queue front's neighbors are not zero a
147:             for (int i = 0; i < Size; i++)
148:
149:                 //The case where the queue front's neighbors are not zero and have not been v
150:                 if (AdjMat[Q.front()][i] == 1 && isVisited[i] == false){
151:
152:                     //The current vertex is pushed into the queue and is marked as visited
153:                     Q.push(i);
154:                     isVisited[i] = true;
155:                 }
156:
157:                 //The current vertex is popped
158:                 for (int i = 0; i < Size; i++)
159:                     cout << Q.front() << " ";
160:                 Q.pop();
161:             }
162:         }
163:     private:
164:
165:         int **AdjMat;

```

```
166:     bool *visited;
167:     int Size;
168: };
169:
170:
171:
172: int main(){
173:
174:     Graph g(6);
175:
176:     g.insertEdge(0,1);
177:     g.insertEdge(0,2);
178:     g.insertEdge(0,3);
179:     g.insertEdge(1,5);
180:     g.insertEdge(2,3);
181:     g.insertEdge(2,4);
182:
183:     cout << "Printing graph:\n";
184:     g.printGraph();
185:
186:     cout << "printing dfs: ";
187:     g.DFS(0); cout << "\n\n";
188:
189:     cout << "printing bfs: ";
190:     g.BFS(0);
191:
192:     return 0;
193:
194: }
```



(29) 128

```

1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //problem-1.cpp
4:
5: /*Let n be the size of the matrix row or column*/
6:
7: #include <iostream>
8: #include <list>
9: using namespace std;
10: //Size of the matrix
11: #define S 7
12:
13: //Solves a given maze. Returns "NO SOLUTION" if there isn't any
14: //Time Complexity:  $O(n^2)$  Because the worst case scenario is a matrix with all 1's
15: void MazeSolver(int Matrix[S][S], int Size){
16:
17:     //Matrix that keeps track of which squares have been visited
18:     bool isVisited[Size][Size] = {false};
19:
20:     //Lists that portray stacks. One stores the row index, and the other stores the column
21:     //Lists are used for their iterative and popping from the back features
22:     list<int> StackRow, StackCol;
23:     int i = 0, j = 0; //i and j keep track of where the maze pointer is positioned
24:     int cost = 0; //Keeps track of the number of steps taken
25:
26:     //StackRow and StackCol have 0 pushed as the starting place
27:     StackRow.push_back(i);
28:     StackCol.push_back(j);
29:
30:     //(0,0) is marked as visited
31:     isVisited[i][j] = true;
32:
33:
34:     //The maze pointer begins to move. It's main goal is to find a path to the end
35:     //If there is no path to the end or the number of steps take less than  $2*Size + 3$ 
36:     //Then the function returns "NO SOLUTION" and exits
37:     while (true){
38:
39:         //Checks if the two stacks are empty. If they are, then there is no solution.
40:         if (StackCol.empty() && StackRow.empty()){
41:
42:             cout << "NO SOLUTION!\n";
43:             return;
44:         }
45:
46:         //Checks if the last element entered in both stacks are at the end
47:         if (StackRow.back() == Size - 1 && StackCol.back() == Size - 1){
48:
49:             //If the number of steps is greater than  $2*Size + 3$ , then the function has succeeded
50:             if (cost >= (2 * Size) + 4 || cost <= (2 * Size) + 10)
51:                 break;
52:
53:             //If the cost is less than  $2*Size + 3$ , then the function returns "NO SOLUTION" and exits
54:             else{
55:

```

```

56:         cout << "NO SOLUTION!\n";
57:         return;
58:     }
59: }
60:
61: //Checks if the right of the current position is not zero
62: //and if moving one unit to the right does not exit the maze boundry
63: //and if the position to the right not marked as visited
64: //If so, then the maze pointer moves one unit to the right
65: if (Matrix[i][j + 1] != 0 && j + 1 < Size && !isVisited[i][j + 1]){
66:
67:     j++; //j is incremented by 1
68:     isVisited[i][j] = true; //Current position is marked as visited
69:
70:     //The row index and and the column index have been pushed into their respective s
71:     StackRow.push_back(i);
72:     StackCol.push_back(j);
73:     cost++; //The number of steps is incremented
74: }
75:
76: //Checks if the Left of the current position is not zero
77: //and if moving one unit to the left does not exit the maze boundry
78: //and if the position to the Left not marked as visited
79: //If so, then the maze pointer moves one unit to the Left
80: else if (Matrix[i][j - 1] != 0 && j - 1 >= 0 && !isVisited[i][j - 1]){
81:
82:     j--; //j is decremented by 1
83:     isVisited[i][j] = true; //Current position is marked as visited
84:
85:     //The row index and and the column index have been pushed into their respective s
86:     StackRow.push_back(i);
87:     StackCol.push_back(j);
88:     cost++; //The number of steps is incremented
89: }
90:
91: //Checks if the position above the maze pointer is not zero
92: //and if moving one unit up does not exit the maze boundry
93: //and if the position above the current position not marked as visited
94: //If so, then the maze pointer moves one unit up
95: else if (Matrix[i - 1][j] != 0 && i - 1 >= 0 && !isVisited[i - 1][j]){
96:
97:     i--; //i is decremented by 1
98:     isVisited[i][j] = true; //Current position is marked as visited
99:
100:    //The row index and and the column index have been pushed into their respective s
101:    StackRow.push_back(i);
102:    StackCol.push_back(j);
103:    cost++; //The number of steps is incremented
104: }
105:
106: //down
107: //Checks if the position under the maze pointer is not zero
108: //and if moving one unit down does not exit the maze boundry
109: //and if the position under the current position not marked as visited
110: //If so, then the maze pointer moves one unit up

```

```

111:         else if (Matrix[i + 1][j] != 0 && i + 1 < Size && !isVisited[i + 1][j]){
112:
113:             i++; //i is incremented by 1
114:             isVisited[i][j] = true; //Current position is marked as visited
115:
116:             //The row index and the column index have been pushed into their respective s
117:             StackRow.push_back(i);
118:             StackCol.push_back(j);
119:             cost++; //The number of steps is incremented
120:         }
121:
122:         //If the maze pointer reached a dead end, then it back-tracks one position at a time
123:         else{
124:
125:             //Both stacks are popped
126:             StackCol.pop_back();
127:             StackRow.pop_back();
128:
129:             //i and j are assigned to the current top, which is the previous position
130:             i = StackRow.back();
131:             j = StackCol.back();
132:
133:             cost--; //The number of steps is decremented by 1
134:         }
135:     }
136:
137:     //iterators for StackRow and StackCol
138:     list<int>::iterator itRow = StackRow.begin();
139:     list<int>::iterator itCol = StackCol.begin();
140:
141:     //Prints the starting position then increments the iterators
142:     printf("(%d,%d)", *itRow, *itCol); itRow++; itCol++;
143:
144:     //Goes through the rest of both Lists and prints the positions
145:     while (itRow != StackRow.end() && itCol != StackCol.end()){
146:
147:         printf(" -> (%d,%d)", *itRow, *itCol);
148:
149:         //Both iterators are incremented
150:         itRow++;
151:         itCol++;
152:     }
153:
154:     cout << "\n\nThe number of steps: " << cost << endl;
155: }
156:
157: //Prints the matrix
158: //Time Complexity:  $O(n^2)$  since there is a nested loop
159: void//If there is no path to the end or the number of steps take less than  $2*Size + 3$ 
160:
161:     for (int i = 0; i < size; i++){
162:         for (int j = 0; j < size; j++){
163:             cout << arr[i][j] << " ";
164:
165:             cout << endl;

```

```
166:     }
167:     cout << endl;
168: }
169:
170: //Beginning of program
171: int main(){
172:
173:     int Matrix[S][S] = {{1,1,1,1,0,1,0},
174:                          {0,0,0,1,1,1,1},
175:                          {1,1,1,0,0,0,1},
176:                          {1,0,1,1,1,1,1},
177:                          {1,0,0,0,0,0,0},
178:                          {1,0,0,1,1,1,0},
179:                          {1,1,1,1,0,1,1}
180:     };
181:
182:     PrintMatrix(Matrix, S);
183:
184:     MazeSolver(Matrix, S);
185:
186:     return 0;
187: }
188: //End of Program
```





10/10

```


1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //Problem-2.cpp
4:
5: /*Let n be the number of rows or columns in the matrix*/
6:
7: #include <iostream>
8: #include <stdio.h>
9: using namespace std;
10:
11: class Graph{
12:
13: public:
14:
15:     //Constructor
16:     Graph(int num): Size(num)/*num is assigned to size*/
17:     {
18:
19:         //The adjacency matrix is initialised to be all zeros
20:         AdjMat = new int* [Size];
21:
22:         for (int i = 0; i < Size; i++){
23:
24:             AdjMat[i] = new int[Size];
25:
26:             for (int j = 0; j < Size; j++)
27:                 AdjMat[i][j] = 0;
28:         }
29:     }
30:
31:     //inserts an edge in the graph
32:     //Time Complexity: O(1) since there are a constant amount of instructions
33:     void insertEdge(int Vertex1, int Vertex2){
34:
35:         //The case where the vertices entered are greater than or equal to the Size
36:         if (Vertex1 >= Size || Vertex2 >= Size){
37:
38:             cout << "You cannot go above the capacity!\n";
39:             return;
40:         }
41:
42:         //The case where the edge already exists
43:         if (AdjMat[Vertex1][Vertex2] == 1 && AdjMat[Vertex2][Vertex1] == 1){
44:
45:             cout << Vertex1 << " and " << Vertex2 << " are already connected!\n";
46:             return;
47:         }
48:
49:         //Assigns an edge to the both vertices
50:         AdjMat[Vertex1][Vertex2] = 1;
51:         AdjMat[Vertex2][Vertex1] = 1;
52:
53:         printf("(%d) --> (%d)\n", Vertex1, Vertex2);
54:     }
55:

```

```

56: //Removes an edges from the graph
57: //Time Complexity: O(1) Since there is a constant number of instructions
58: void removeEdge(int Vertex1, int Vertex2){
59:
60:     //The case where the vertices entered are greater than or equal to the Size
61:     if (Vertex1 >= Size || Vertex2 >= Size){
62:
63:         cout << "No such vertex exists!\n";
64:         return;
65:     }
66:
67:     //The case where the edge already does not exist
68:     if (AdjMat[Vertex1][Vertex2] == 0 && AdjMat[Vertex2][Vertex1] == 0){
69:
70:         cout << Vertex1 << " and " << Vertex2 << " were never connected!\n";
71:         return;
72:     }
73:
74:     //Removes the edge by assigning zero to the vertices
75:     AdjMat[Vertex1][Vertex2] = 0;
76:     AdjMat[Vertex2][Vertex1] = 0;
77:
78:     printf("(%d) -/-> (%d)\n", Vertex1, Vertex2);
79: }
80:
81: //Prints the vertex and all its connected vertices
82: //Time Complexity: O(n^2) since there is a nested loop
83: void printGraph(){
84:
85:     //Goes through the matrix and prints all the vertices connected by the vertex i
86:     for (int i = 0; i < Size; i++){
87:
88:         cout << "(" << i << ") --> ";
89:
90:         for (int j = 0; j < Size; j++)
91:
92:             if (AdjMat[i][j] == 1){
93:
94:                 cout << j << " ; ";
95:             }
96:
97:         cout << endl;
98:     }
99: }
100:
101: //Checks if the graph is complete. Returns true if it is complete. Otherwise, returns
102: //Time Complexity: O(n^2) since there is a nested loop
103: bool isComplete(){
104:
105:     for (int i = 0; i < Size - 1; i++)
106:         for (int j = i + 1; j < Size; j++)
107:             if (AdjMat[i][j] == 0)
108:                 return false;
109:
110:     return true;

```



```

111:
112:     }
113:
114: private:
115:
116:     int **AdjMat;
117:     int Size;
118: };
119:
120: //Beginning of program
121: int main(){
122:
123:     Graph g(5);
124:
125:     g.insertEdge(0,1);
126:     g.insertEdge(0,2);
127:     g.insertEdge(0,3);
128:     g.insertEdge(0,4);
129:     g.insertEdge(1,2);
130:     g.insertEdge(1,3);
131:     g.insertEdge(1,4);
132:     g.insertEdge(2,3);
133:     g.insertEdge(2,4);
134:     g.insertEdge(3,4);
135:
136:     cout << "\nprinting graph: \n";
137:     g.printGraph();
138:
139:     cout << "\n\nIs the graph complete? The answer is: ";
140:     if (g.isComplete())
141:         cout << "True!\n";
142:     else
143:         cout << "False!\n";
144:
145:     return 0;
146: }
147: //End of program

```

```

1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //problem-3.cpp
4:
5: #include <iostream>
6: #include <stdio.h>
7: #include <queue>
8: using namespace std;
9:
10: class Graph{
11:
12: public:
13:
14:     //Constructor
15:     Graph(int num): Size(num) /*num is assigned to size*/
16:     {
17:
18:         //The adjacency matrix is initialised to be all zeros
19:         AdjMat = new int* [Size];
20:
21:         for (int i = 0; i < Size; i++){
22:
23:             AdjMat[i] = new int [Size];
24:             for (int j = 0; j < Size; j++)
25:                 AdjMat[i][j] = 0;
26:         }
27:     }
28:
29:     //inserts an edge in the graph
30:     //Time Complexity: O(1) since there are a constant amount of instructions
31:     void insertEdge(int V1, int V2, int W){
32:
33:         //The case where the vertices entered are greater than or equal to the Size
34:         if (V1 >= Size || V2 >= Size){
35:
36:             cout << "Cannot exceed capacity!\n";
37:             return;
38:         }
39:
40:         //The case where the edge already exists
41:         if ( AdjMat[V1][V2] != 0 && AdjMat[V2][V1] != 0){
42:
43:             cout << "This edge already exists!\n";
44:             return;
45:         }
46:
47:         //Assigns an edge to the both vertices
48:         AdjMat[V1][V2] = W;
49:         AdjMat[V2][V1] = W;
50:     }
51:
52:     //Prints the vertex and all its connected vertices
53:     //Time Complexity: O(n^2) since there is a nested loop
54:     void printGraph(){
55:

```

```

56:         //Goes through the matrix and prints all the vertices connected by the vertex i
57:         for (int i = 0; i < Size; i++){
58:
59:             cout << "(" << i << ")" --> ";
60:
61:             for (int j = 0; j < Size; j++)
62:
63:                 if (AdjMat[i][j] != 0){
64:
65:                     cout << j << " ";
66:                 }
67:
68:             cout << endl;
69:         }
70:     }
71:
72:     //Removes an edges from the graph
73:     //Time Complexity: O(1) Since there is a constant number of instructions
74:     void removeEdge(int V1, int V2){
75:
76:         //The case where the vertices entered are greater than or equal to the Size
77:         if (V1 >= Size || V2 >= Size){
78:
79:             cout << "Cannot exceed capacity!\n";
80:             return;
81:         }
82:
83:         //The case where the edge already does not exist
84:         if ( AdjMat[V1][V2] == 0 && AdjMat[V2][V1] == 0){
85:
86:             cout << "This edge already doesn't exists!\n";
87:             return;
88:         }
89:
90:         //Removes the edge by assigning zero to the vertices
91:         AdjMat[V1][V2] = 0;
92:         AdjMat[V2][V1] = 0;
93:     }
94:
95:     //Finds the smallest edge while marking it as visited . If there is none, then it ret
96:     //Time Complexity: O(n) since the row length is variable
97:     int findSmallestEdge(bool arr[], int Row){
98:
99:         //Min is to find the minimum and chosen_index is to find the index with the minimum v
100:         int Min = 999, Chosen_Index = -1;
101:
102:         //Goes through the matrix row to find the minimum unmarked value
103:         for (int i = 0; i < Size; i++)
104:
105:             if (AdjMat[Row][i] != 0 && !arr[i] && AdjMat[Row][i] < Min){
106:
107:                 Min = AdjMat[Row][i];
108:                 Chosen_Index = i;
109:             }
110:

```

```

111:         //Returns the minimum value.
112:         return Chosen_Index;
113:     }
114:
115:     //Finds the shortest path from start to all nodes
116:     //Time Complexity:  $O(n^2)$  since the loop will iterate  $n$  times and findSmallestEdge wi
117:     void ShortestPaths(int start){
118:
119:         /* Parent array is to keep track of the parents
120:          Dist array is to keep track of the minimum distance
121:          Visited array is to keep track of which vertices have been visited
122:          Queue is to push the unvisited and pop the visited vertices and also to ke
123:         */
124:
125:         int Parent[Size];
126:         int Dist[Size];
127:         bool Visited[Size];
128:         queue<int> Q;
129:
130:         //Initialising all the arrays
131:         for (int i = 0; i < Size; i++){
132:
133:             Parent[i] = -1;
134:             Dist[i] = 9999;
135:             Visited[i] = false;
136:         }
137:
138:         //pushing the start vertex into the queue and the Length of the path from the start t
139:         Dist[start] = 0;
140:         Q.push(start);
141:
142:         //Goes through all the vertices and finds the shortest path to each
143:         while (!Q.empty()){
144:
145:             //Boolean array to keep track of all the visited vertices on the current row
146:             bool subVisited[Size] = {false};
147:
148:             //gets the minimum unmarked value index
149:             int MinValIndex = 0;
150:
151:
152:             //Finds the edges in ascending order and picks the shortest path while updating t
153:             //It stops when MinValIndex is -1
154:             do{
155:
156:                 //Finds the smallest unmarked edge in the current row
157:                 MinValIndex = findSmallestEdge(subVisited, Q.front());
158:
159:                 //If not all edges are unmarked, then continue
160:                 if (MinValIndex != -1){
161:
162:                     //current row index is marked as true
163:                     subVisited[MinValIndex] = true;
164:
165:                     //If the minimum edge from the current row is less than or equal to its d

```

```

166:         //then the new distance to the minimum vertex is updated to be the distan
167:         //the minimum edge from the current row
168:         //Its parent will be updated to be the current row and it will be pushed
169:         if (AdjMat[Q.front()][MinValIndex] + Dist[Q.front()] <= Dist[MinValIndex]
170:
171:             Dist[MinValIndex] = AdjMat[Q.front()][MinValIndex] + Dist[Q.front()];
172:             Parent[MinValIndex] = Q.front();
173:             Q.push(MinValIndex);
174:         }
175:     }
176:
177:     }while (MinValIndex != -1);
178:
179:     //Marks the current row as visited and pops it from the queue
180:     Visited[Q.front()] = true;
181:     Q.pop();
182: }
183:
184: //prints the path from a vertex to the starting vertex
185: for (int i = Size - 1; i >= 0; i--){
186:
187:     int stopping_Cond = i;
188:
189:     cout << "The shortest path from " << i << " to " << start << " is: " << i;
190:     while (true){
191:
192:         if (Parent[stopping_Cond] == -1)
193:             break;
194:
195:         else{
196:
197:             cout << " --> " << Parent[stopping_Cond];
198:             stopping_Cond = Parent[stopping_Cond];
199:         }
200:     }
201:
202:     cout << endl;
203: }
204:
205: }
206:
207: private:
208:
209:     int** AdjMat;
210:     int Size;
211: };
212:
213: //Beginning of main
214: int main(){
215:
216:     Graph g(8);
217:
218:     g.insertEdge(0,1,3);
219:     g.insertEdge(1,2,7);
220:     g.insertEdge(1,3,2);

```

```
221:     g.insertEdge(2,4,1);
222:     g.insertEdge(2,7,6);
223:     g.insertEdge(2,3,2);
224:     g.insertEdge(3,5,4);
225:     g.insertEdge(3,6,3);
226:
227:     g.printGraph();
228:
229:     cout << "\n\nShortest path: \n\n";
230:
231:     g.ShortestPaths(0);
232:
233:     return 0;
234: }
235: //End of main
```



20/20

```

1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //Problem-4
4:
5: /*Let n be the size of the adjacency list, and m be the size of the linked lists*/
6:
7: #include <iostream>
8: #include <stdio.h>
9: #include <list>
10: #include <queue>
11: using namespace std;
12:
13: //Searches for an element in a given linked list. Returns true if it is found, otherw
14: //Time Complexity: O(m) since the size of the linked list is variable
15: bool search(list<int> List, int data){
16:
17:     //If the linked list is empty, it returns false
18:     if (List.empty())
19:         return false;
20:
21:     //Iterator that goes through the list
22:     list<int>::iterator it;
23:
24:     //Goes through the list and checks if the element exists. Returns true if it does
25:     for (it = List.begin(); it != List.end(); it++)
26:         if (*it == data)
27:             return true;
28:
29:     //If the loop stops, then the element does not exist. In that case, returns false
30:     return false;
31: }
32:
33: //Removes an element from a given list
34: //Time Complexity: O(m) since the size of the linked list is variable
35: void Remove(list<int>& List, int data){
36:
37:     //If the list is empty, then the function does nothing and exists.
38:     if (List.empty())
39:         return;
40:
41:     //Iterator that goes through the list
42:     list<int>::iterator it;
43:
44:     //Goes through the list and checks if the element exists. If it does, then it removes
45:     for (it = List.begin(); it != List.end(); it++)
46:         if (*it == data){
47:
48:             List.erase(it);
49:             return;
50:         }
51: }
52:
53: class Graph{
54:
55: public:

```

```

56:
57: //Constructor
58: Graph(int num){
59:
60:     //num is assigned to Size
61:     this->Size = num;
62:
63:     //Adjacency List is defined
64:     AdjList = new list<int> [Size];
65:
66:     //Weight matrix initialised to 0's
67:     Weight = new int* [Size];
68:     for (int i = 0; i < Size; i++){
69:
70:         Weight[i] = new int [Size];
71:
72:         for (int j = 0; j < Size; j++){
73:             Weight[i][j] = 0;
74:         }
75:     }
76:
77:     //Inserts an edge to the graph
78:     //Time Complexity: O(1) since there is a constant number of instructions
79:     void InsertEdge(int V1, int V2, int W){
80:
81:         //The case where the vertices entered are greater than or equal to the Size
82:         if (V1 >= Size || V2 >= Size){
83:
84:             cout << "Cannot go beyond the capacity!\n";
85:             return;
86:         }
87:
88:         //The case where the edge already exists
89:         if (search(AdjList[V1],V2) && search(AdjList[V2],V1)){
90:
91:             cout << "The two vertices are already connected!\n";
92:             return;
93:         }
94:
95:         //Both vertices are pushed
96:         AdjList[V1].push_back(V2);
97:         AdjList[V2].push_back(V1);
98:
99:         //Both edges are assigned their weights
100:         Weight[V1][V2] = W; Weight[V2][V1] = W;
101:
102:         printf("(%d) --> (%d)\n", V1, V2);
103:     }
104:
105:     //Removes an edge from the graph
106:     //Time Complexity: O(1) Since there is a constant amount of instructions
107:     void RemoveEdge(int V1, int V2){
108:
109:         //The case where the vertices entered are greater than or equal to the Size
110:         if (V1 >= Size || V2 >= Size){

```

```

111:
112:         cout << "Cannot go beyond the capacity!\n";
113:         return;
114:     }
115:     //The case where the edge already does not exist
116:     if (!search(AdjList[V1],V2) && !search(AdjList[V2],V1)){
117:
118:         cout << "The two vertexes were never connected!\n";
119:         return;
120:     }
121:
122:     //Removes the edge
123:     Remove(AdjList[V1], V2);
124:     Remove(AdjList[V2], V1);
125:
126:     //The edge's weight is assigned to zero
127:     Weight[V1][V2] = 0; Weight[V2][V1] = 0;
128:
129:     printf("(%d) -/-> (%d)\n", V1, V2);
130: }
131:
132: //Prints the vertex and all its connected vertices
133: //Time Complexity: O(n*m) since it's going through the array while going through the
134: void printGraph(){
135:
136:     for (int i = 0; i < Size; i++){
137:
138:         printf("(%d) --> ", i);
139:
140:         list<int>::iterator it;
141:         for(it = AdjList[i].begin(); it != AdjList[i].end(); it++)
142:             cout << *it << " ";
143:
144:         cout << endl;
145:     }
146: }
147: //Finds the minimum spanning tree and prints its cost
148: //Time Complexity: O(n^2) since it goes through the list of elements while going thro
149: //and goes through the matrix while printing the graph
150: void MinimumSpanningTree(int start){
151:
152:     //Adjacency List for the resultant tree (only used for printing)
153:     list<int> List[Size];
154:
155:     //Boolean array to keep track of vertices that have been visited. It is initialised t
156:     bool isVisited[Size];
157:     for (int i = 0; i < Size; i++)
158:         isVisited[i] = false;
159:
160:     //A list to keep track of all the vertices with the lightest weights
161:     list<int> Verticies;
162:
163:     //Queue to make the loop continue and to print all the vertices
164:     queue<int> Q;
165:

```

```

166:      /*Min is to find out which vertex has the minimum weight
167:      Chosen_Vertix is used to store the vertex with the lowest weight
168:      Current_Row is the row where Chosen_Vertix lies
169:      Cost is the cost of the minimum spanning tree
170:      */
171:      int Min, Chosen_Vertix, Cost = 0, Current_Row;
172:
173:      //The starting vertex is pushed into the queue and the list. It is also marked as visited
174:      Vertices.push_back(start);
175:      Q.push(start);
176:      isVisited[start] = true;
177:
178:      //Goes through the graph to find and determine the minimum spanning tree and its cost
179:      while (!Q.empty()){
180:
181:          //Min and Chosen_Vertix are both initialised at the start of each loop
182:          Min = 999;
183:          Chosen_Vertix = -1;
184:
185:          /*
186:          The iterator it goes through all the elements and their neighbors to
187:          which edge has the lowest weight.
188:          It also has to be from a vertex that has been unvisited
189:          */
190:          list<int>::iterator it = Vertices.begin();
191:          for (; it != Vertices.end(); it++){
192:              list<int>::iterator it2 = AdjList[*it].begin();
193:              for (; it2 != AdjList[*it].end(); it2++){
194:
195:                  //The case where the lowest weight is found
196:                  if (Weight[*it][*it2] != 0 && Weight[*it][*it2] <= Min && !isVisited[*it2])
197:                      //Min is changed to not affect the if statement
198:                      Min = Weight[*it][*it2];
199:
200:                      //it and it2 are assigned to Current_Row and Chosen_Vertix
201:                      Chosen_Vertix = *it2;
202:                      Current_Row = *it;
203:
204:                  }
205:              }
206:          }
207:
208:          //If Chosen_Vertix is changed (meaning a new lowest weight was found) then mark Chosen_Vertix as visited
209:          //Push it into both the queue and the vertices list
210:          //and add the weight of the current edge to the cost
211:          if (Chosen_Vertix != -1){
212:              isVisited[Chosen_Vertix] = true;
213:              Q.push(Chosen_Vertix);
214:              Vertices.push_back(Chosen_Vertix);
215:              Cost += Weight[Current_Row][Chosen_Vertix];
216:
217:              //Setting the current row and column to 1
218:              List[Current_Row].push_back(Chosen_Vertix);
219:
220:

```

```

221:         List[Chosen_Vertix].push_back(Current_Row);
222:     }
223:
224:     //The queue is popped
225:     Q.pop();
226: }
227:
228:
229: //Prints the minimum spanning tree
230: for (int i = 0; i < Size; i++){
231:
232:     list<int>::iterator it = List[i].begin();
233:
234:     printf("(%d) --> ", i);
235:     for (; it != List[i].end(); it++)
236:         cout << *it << " ";
237:
238:     cout << endl;
239: }
240:
241: //Prints the cost
242: cout << "\nThe cost is : " << Cost << endl;
243:
244: }
245:
246: private:
247:
248:     list<int>* AdjList;
249:     int **Weight;
250:     int Size;
251:
252:
253: };
254:
255: int main(){
256:
257:     Graph g(7);
258:
259:     g.InsertEdge(0,1,3);
260:     g.InsertEdge(0,2,4);
261:     g.InsertEdge(0,6,7);
262:     g.InsertEdge(1,5,2);
263:     g.InsertEdge(2,3,5);
264:     g.InsertEdge(2,4,6);
265:     g.InsertEdge(2,5,1);
266:     g.InsertEdge(4,5,4);
267:
268:     cout << "\nPrinting graph: " << endl;
269:
270:     g.printGraph();
271:
272:     cout << "\nMinimum spanning tree: " << endl;
273:     g.MinimumSpanningTree(0);
274: }

```

