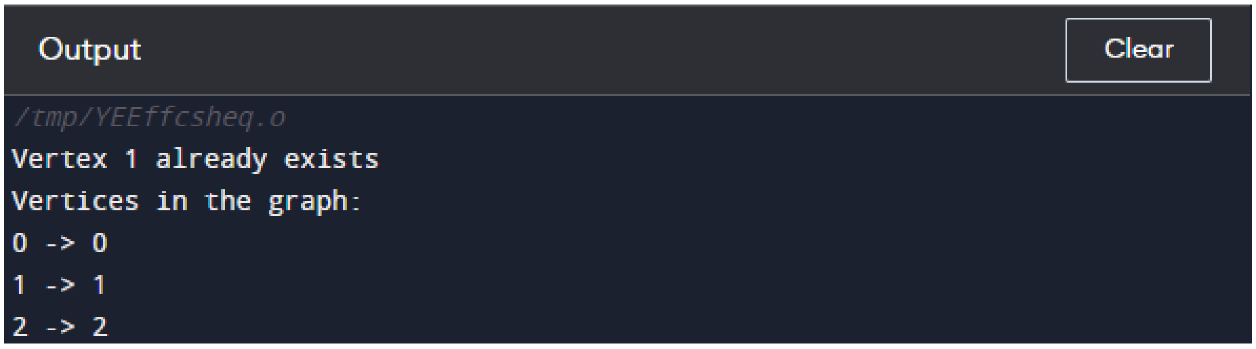
ASSIGNMENT 6

1. **Write a function void addVertex(int n) that adds a vertex with name n to the graph. If there is already a vertex with name n, then the function should do nothing. Otherwise the new vertex should be made the last vertex in the vertex list of the graph.**

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100 // Maximum number of vertices
// Node to represent each vertex in the graph
struct Node {
int vertex;
struct Node* next;
};
// Structure to represent the graph
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
// Initialize a graph
struct Graph* createGraph() {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = 0;
for (int i = 0; i < MAX_VERTICES; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
// Function to add a vertex to the graph
void addVertex(struct Graph* graph, int n) {
if (graph == NULL || n < 0 || n >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if the vertex already exists
if (graph->adjList[n] != NULL) {
printf("Vertex %d already exists\n", n);
return;
}
// Create a new node for the new vertex
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = n;
newNode->next = NULL;
// Add the new vertex to the end of the list
if (graph->numVertices < MAX_VERTICES) {
```

```
graph->adjList[n] = newNode;
graph->numVertices++;
} else {
printf("Exceeded maximum number of vertices\n");
free(newNode);
}
}
// Function to display the graph
void displayGraph(struct Graph* graph) {
if (graph == NULL) {
printf("Graph is NULL\n");
return;
}
printf("Vertices in the graph:\n");
for (int i = 0; i < MAX_VERTICES; ++i) {
if (graph->adjList[i] != NULL) {
printf("%d -> ", i);
struct Node* temp = graph->adjList[i];
while (temp != NULL) {
printf("%d ", temp->vertex);
temp = temp->next;
}
printf("\n");
}
}
}
// Example usage
int main() {
struct Graph* graph = createGraph();
addVertex(graph, 0);
addVertex(graph, 1);
addVertex(graph, 2);
addVertex(graph, 1); // Vertex 1 already exists
displayGraph(graph);
return 0;
}
```



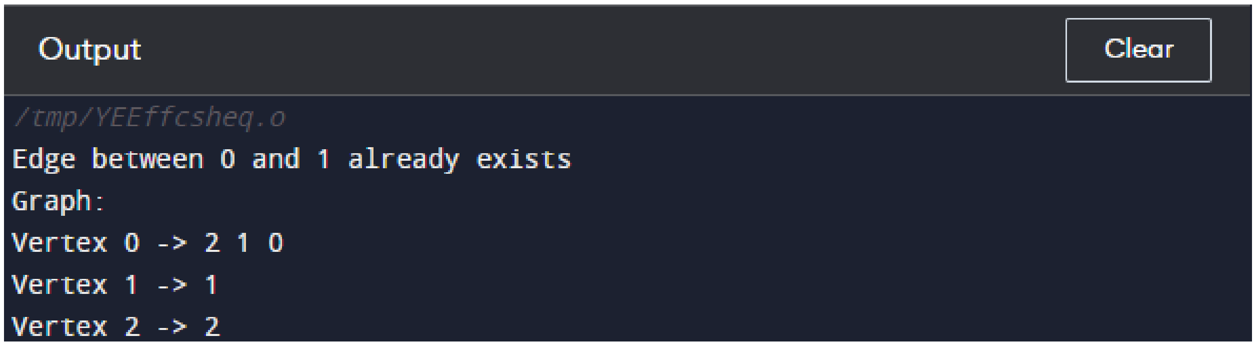
```
/tmp/YEEffcsheq.o
Vertex 1 already exists
Vertices in the graph:
0 -> 0
1 -> 1
2 -> 2
```

2. **Write a function void addEdge(int u, int v) that does the following. The function should add a new edge from the vertex with name u to vertex with name v to the graph. If there is no vertex named u or no vertex named v, then the function should do nothing. If there is already an edge between u and v, the function should not do anything.**

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100 // Maximum number of vertices
// Node to represent each vertex in the graph
struct Node {
int vertex;
struct Node* next;
};
// Structure to represent the graph
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
// Initialize a graph
struct Graph* createGraph() {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = 0;
for (int i = 0; i < MAX_VERTICES; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
// Function to add a vertex to the graph
void addVertex(struct Graph* graph, int n) {
if (graph == NULL || n < 0 || n >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if the vertex already exists
if (graph->adjList[n] != NULL) {
printf("Vertex %d already exists\n", n);
return;
}
// Create a new node for the new vertex
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = n;
newNode->next = NULL;
// Add the new vertex to the end of the list
if (graph->numVertices < MAX_VERTICES) {
```

```
graph->adjList[n] = newNode;
graph->numVertices++;
} else {
printf("Exceeded maximum number of vertices\n");
free(newNode);
}
}
// Function to add an edge between vertices u and v
void addEdge(struct Graph* graph, int u, int v) {
if (graph == NULL || u < 0 || u >= MAX_VERTICES || v < 0 || v >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if vertices u and v exist
if (graph->adjList[u] == NULL || graph->adjList[v] == NULL) {
printf("Vertices %d or %d do not exist\n", u, v);
return;
}
// Check if an edge between u and v already exists
struct Node* temp = graph->adjList[u];
while (temp != NULL) {
if (temp->vertex == v) {
printf("Edge between %d and %d already exists\n", u, v);
return;
}
temp = temp->next;
}
// Add edge from u to v
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
// Function to display the graph
void displayGraph(struct Graph* graph) {
if (graph == NULL) {
printf("Graph is NULL\n");
return;
}
printf("Graph:\n");
for (int i = 0; i < MAX_VERTICES; ++i) {
if (graph->adjList[i] != NULL) {
printf("Vertex %d -> ", i);
struct Node* temp = graph->adjList[i];
while (temp != NULL) {
printf("%d ", temp->vertex);
temp = temp->next;
}
printf("\n");
}
}
}
// Example usage
int main() {
struct Graph* graph = createGraph();
addVertex(graph, 0);
addVertex(graph, 1);
addVertex(graph, 2);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 0, 1); // Edge already exists
displayGraph(graph);
return 0;
}
```



```
/tmp/YEEffcsheq.o
Edge between 0 and 1 already exists
Graph:
Vertex 0 -> 2 1 0
Vertex 1 -> 1
Vertex 2 -> 2
```

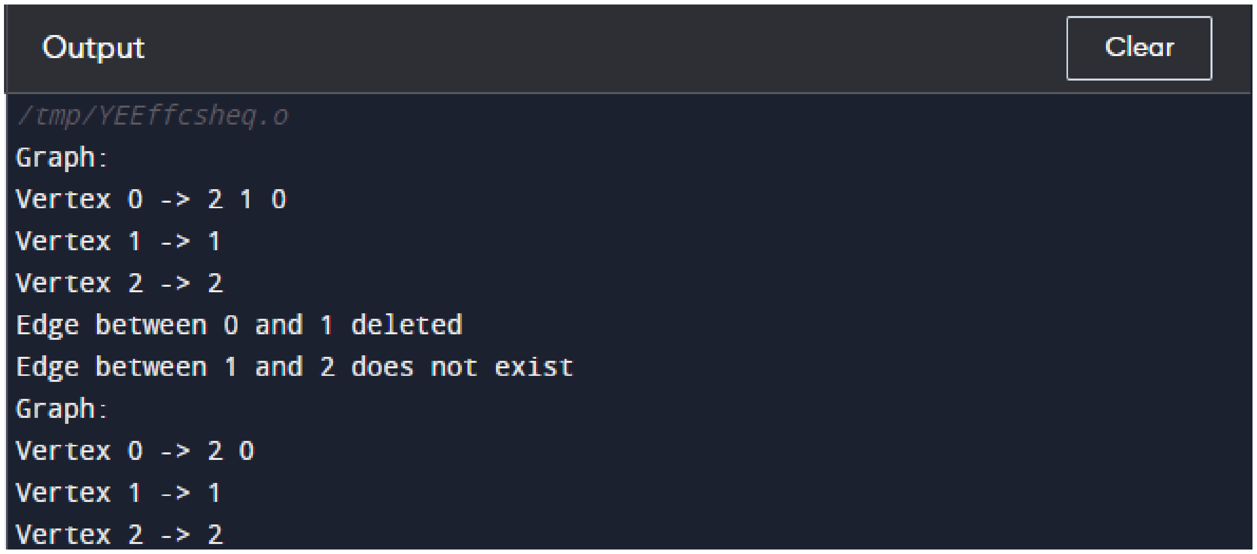
3. **Write a function void delEdge(int u, int v) that does the following. The function should remove the edge from vertex with name u to vertex with name v from the graph. If there is no such edge in the graph, then the function should do nothing.**

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100 // Maximum number of vertices
// Node to represent each vertex in the graph
struct Node {
int vertex;
struct Node* next;
};
// Structure to represent the graph
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
// Initialize a graph
struct Graph* createGraph() {
```

```
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = 0;
for (int i = 0; i < MAX_VERTICES; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
// Function to add a vertex to the graph
void addVertex(struct Graph* graph, int n) {
if (graph == NULL || n < 0 || n >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if the vertex already exists
if (graph->adjList[n] != NULL) {
printf("Vertex %d already exists\n", n);
return;
}
// Create a new node for the new vertex
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = n;
newNode->next = NULL;
// Add the new vertex to the end of the list
if (graph->numVertices < MAX_VERTICES) {
graph->adjList[n] = newNode;
graph->numVertices++;
} else {
printf("Exceeded maximum number of vertices\n");
free(newNode);
}
}
// Function to add an edge between vertices u and v
void addEdge(struct Graph* graph, int u, int v) {
if (graph == NULL || u < 0 || u >= MAX_VERTICES || v < 0 || v >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if vertices u and v exist
if (graph->adjList[u] == NULL || graph->adjList[v] == NULL) {
printf("Vertices %d or %d do not exist\n", u, v);
return;
}
// Check if an edge between u and v already exists
struct Node* temp = graph->adjList[u];
while (temp != NULL) {
if (temp->vertex == v) {
printf("Edge between %d and %d already exists\n", u, v);
return;
}
```

```
temp = temp->next;
}
// Add edge from u to v
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
// Function to display the graph
void displayGraph(struct Graph* graph) {
if (graph == NULL) {
printf("Graph is NULL\n");
return;
}
printf("Graph:\n");
for (int i = 0; i < MAX_VERTICES; ++i) {
if (graph->adjList[i] != NULL) {
printf("Vertex %d -> ", i);
struct Node* temp = graph->adjList[i];
while (temp != NULL) {
printf("%d ", temp->vertex);
temp = temp->next;
}
printf("\n");
}
}
}
// Function to delete an edge between vertices u and v
void delEdge(struct Graph* graph, int u, int v) {
if (graph == NULL || u < 0 || u >= MAX_VERTICES || v < 0 || v >= MAX_VERTICES) {
printf("Invalid graph or vertex value\n");
return;
}
// Check if vertices u and v exist
if (graph->adjList[u] == NULL || graph->adjList[v] == NULL) {
printf("Vertices %d or %d do not exist\n", u, v);
return;
}
// Check if the edge between u and v exists
struct Node* prev = NULL;
struct Node* curr = graph->adjList[u];
while (curr != NULL) {
if (curr->vertex == v) {
// Edge found, delete it
if (prev == NULL) {
// If the edge is the first node in the list
graph->adjList[u] = curr->next;
} else {
prev->next = curr->next;
}
free(curr);
printf("Edge between %d and %d deleted\n", u, v);
return;
}
prev = curr;
curr = curr->next;
}
printf("Edge between %d and %d does not exist\n", u, v);
}
// Example usage
int main() {
struct Graph* graph = createGraph();
addVertex(graph, 0);
addVertex(graph, 1);
addVertex(graph, 2);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
displayGraph(graph);
delEdge(graph, 0, 1);
delEdge(graph, 1, 2); // Edge doesn't exist
displayGraph(graph);
return 0;
}
```



```
/tmp/YEEffcsheq.o
Graph:
Vertex 0 -> 2 1 0
Vertex 1 -> 1
Vertex 2 -> 2
Edge between 0 and 1 deleted
Edge between 1 and 2 does not exist
Graph:
Vertex 0 -> 2 0
Vertex 1 -> 1
Vertex 2 -> 2
```

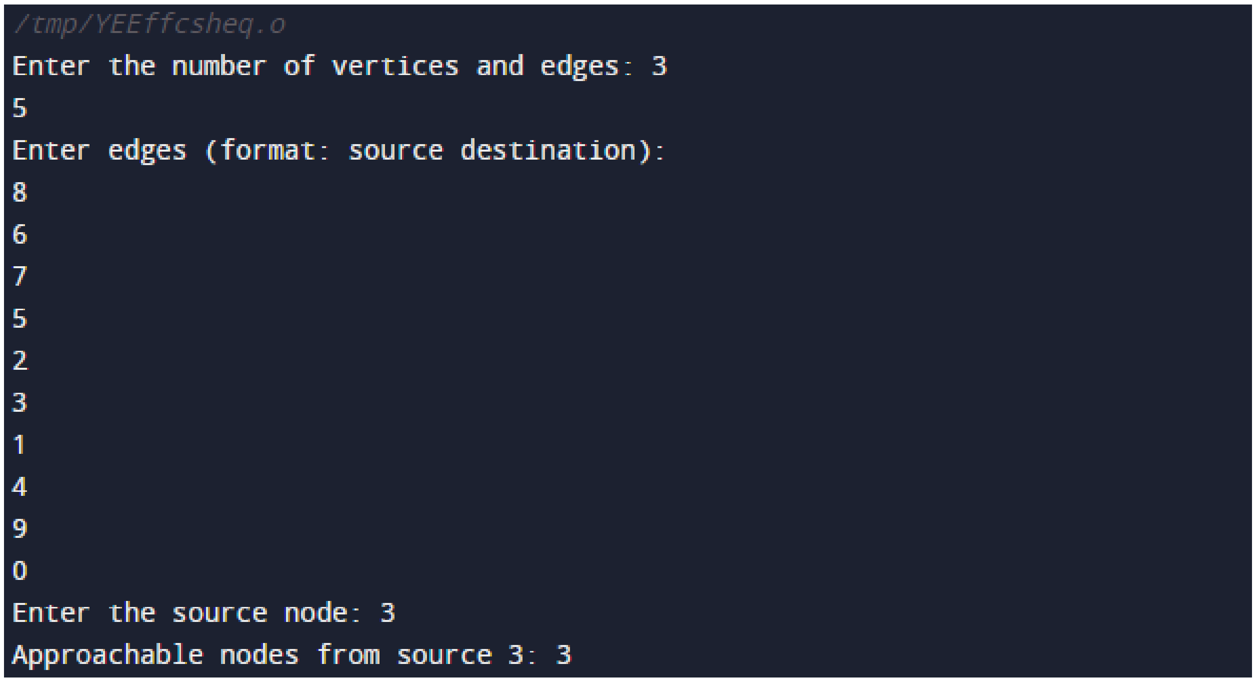
5. Write a program to find approachable nodes from a given source of a given graph using queue as an intermediate data structure (BFS).

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
// Node structure to represent each vertex in the graph
```

```
struct Node {
int vertex;
struct Node* next;
};
// Structure to represent the graph
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
// Function to create a new node
struct Node* createNode(int v) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->next = NULL;
return newNode;
}
// Function to add an edge between vertices u and v
void addEdge(struct Graph* graph, int u, int v) {
struct Node* newNode = createNode(v);
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
// Function to perform Breadth-First Search (BFS)
void BFS(struct Graph* graph, int source) {
bool visited[MAX_VERTICES] = {false};
int queue[MAX_VERTICES];
int front = -1, rear = -1;
visited[source] = true;
queue[++rear] = source;
printf("Approachable nodes from source %d: ", source);
while (front != rear) {
int current = queue[++front];
printf("%d ", current);
struct Node* temp = graph->adjList[current];
while (temp != NULL) {
int adjVertex = temp->vertex;
if (!visited[adjVertex]) {
visited[adjVertex] = true;
queue[++rear] = adjVertex;
}
temp = temp->next;
}
}
}
// Function to create a graph
struct Graph* createGraph(int vertices) {
```

```
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
for (int i = 0; i < vertices; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
// Main function
int main() {
int vertices, edges;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &vertices, &edges);
struct Graph* graph = createGraph(vertices);
printf("Enter edges (format: source destination):\n");
for (int i = 0; i < edges; ++i) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(graph, u, v);
}
int source;
printf("Enter the source node: ");
scanf("%d", &source);
BFS(graph, source);
printf("\n");
return 0;
}
```



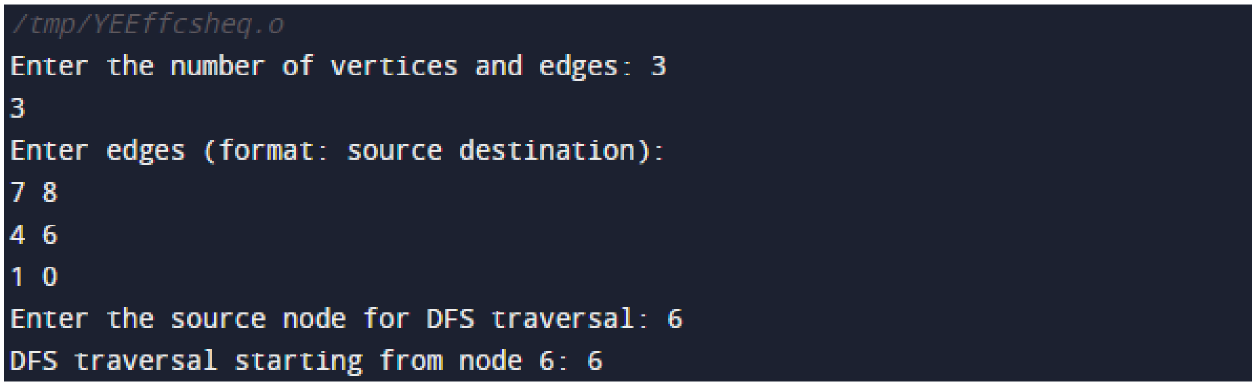
```
/tmp/YEEffcsheq.o
Enter the number of vertices and edges: 3
5
Enter edges (format: source destination):
8
6
7
5
2
3
1
4
9
0
Enter the source node: 3
Approachable nodes from source 3: 3
```

6. Write a program to traverse various nodes of a given graph using stack as an intermediate data structure (DFS).

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
// Node structure to represent each vertex in the graph
struct Node {
int vertex;
struct Node* next;
};
// Structure to represent the graph
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
// Function to create a new node
struct Node* createNode(int v) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->next = NULL;
return newNode;
}
// Function to add an edge between vertices u and v
void addEdge(struct Graph* graph, int u, int v) {
struct Node* newNode = createNode(v);
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
// Function for Depth-First Search (DFS)
void DFS(struct Graph* graph, int source, bool visited[]) {
visited[source] = true;
printf("%d ", source);
struct Node* temp = graph->adjList[source];
while (temp != NULL) {
int adjVertex = temp->vertex;
if (!visited[adjVertex]) {
DFS(graph, adjVertex, visited);
}
temp = temp->next;
}
}
// Function to perform DFS traversal
```

```
void DFSTraversal(struct Graph* graph, int source) {
bool visited[MAX_VERTICES] = {false};
printf("DFS traversal starting from node %d: ", source);
DFS(graph, source, visited);
printf("\n");
}
// Function to create a graph
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
for (int i = 0; i < vertices; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
int main() {
int vertices, edges;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &vertices, &edges);
struct Graph* graph = createGraph(vertices);
printf("Enter edges (format: source destination):\n");
for (int i = 0; i < edges; ++i) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(graph, u, v);
}
int source;
printf("Enter the source node for DFS traversal: ");
scanf("%d", &source);
DFSTraversal(graph, source);
return 0;
}
```



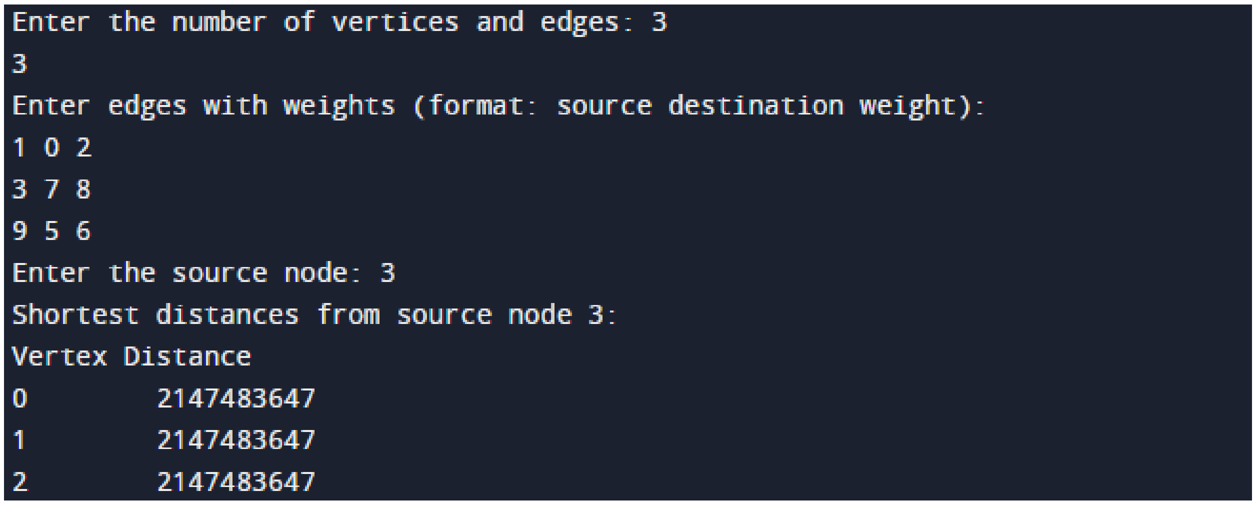
```
/tmp/YEEffcsheq.o
Enter the number of vertices and edges: 3
3
Enter edges (format: source destination):
7 8
4 6
1 0
Enter the source node for DFS traversal: 6
DFS traversal starting from node 6: 6
```

7. Write a program to find shortest path from a given source to all the approachable nodes (Single source shortest path Dijkstra's algorithm).

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
#define MAX_VERTICES 100
struct Node {
int vertex;
int weight;
struct Node* next;
};
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
struct Node* createNode(int v, int weight) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->weight = weight;
newNode->next = NULL;
return newNode;
}
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
for (int i = 0; i < vertices; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
void addEdge(struct Graph* graph, int u, int v, int weight) {
struct Node* newNode = createNode(v, weight);
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
void printShortestPath(int dist[], int V, int src) {
printf("Shortest distances from source node %d:\n", src);
printf("Vertex Distance\n");
for (int i = 0; i < V; ++i) {
printf("%d \t\t %d\n", i, dist[i]);
}
}
int minDistance(int dist[], bool sptSet[], int V) {
int min = INT_MAX, min_index;
for (int v = 0; v < V; ++v) {
if (sptSet[v] == false && dist[v] <= min) {
```

```
min = dist[v];
min_index = v;
}
}
return min_index;
}
void dijkstra(struct Graph* graph, int src) {
int V = graph->numVertices;
int dist[V];
bool sptSet[V];
for (int i = 0; i < V; ++i) {
dist[i] = INT_MAX;
sptSet[i] = false;
}
dist[src] = 0;
for (int count = 0; count < V - 1; ++count) {
int u = minDistance(dist, sptSet, V);
sptSet[u] = true;
struct Node* temp = graph->adjList[u];
while (temp != NULL) {
int v = temp->vertex;
if (!sptSet[v] && dist[u] != INT_MAX && dist[u] + temp->weight < dist[v]) {
dist[v] = dist[u] + temp->weight;
}
temp = temp->next;
}
}
printShortestPath(dist, V, src);
}
int main() {
int vertices, edges;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &vertices, &edges);
struct Graph* graph = createGraph(vertices);
printf("Enter edges with weights (format: source destination weight):\n");
for (int i = 0; i < edges; ++i) {
int u, v, weight;
scanf("%d %d %d", &u, &v, &weight);
addEdge(graph, u, v, weight);
}
int source;
printf("Enter the source node: ");
scanf("%d", &source);
dijkstra(graph, source);
return 0;
}
```



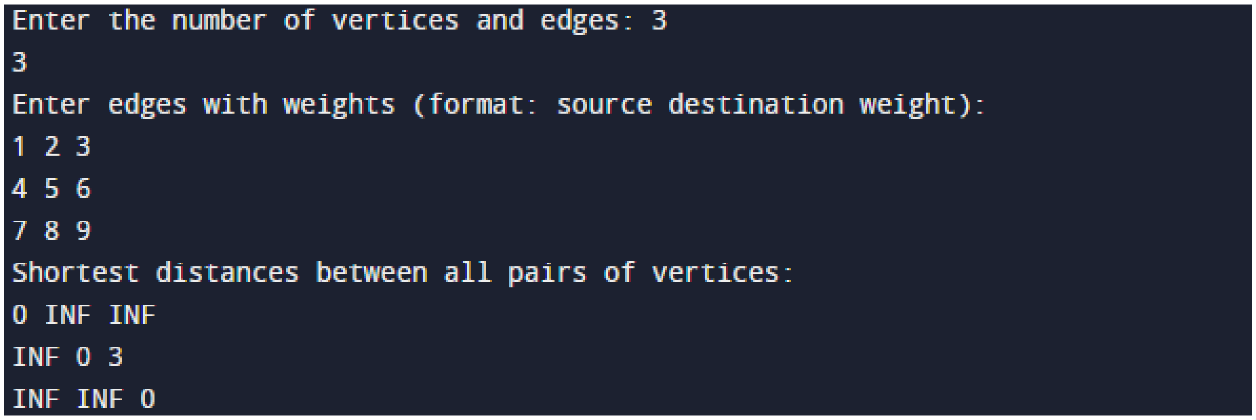
```
Enter the number of vertices and edges: 3
3
Enter edges with weights (format: source destination weight):
1 0 2
3 7 8
9 5 6
Enter the source node: 3
Shortest distances from source node 3:
Vertex Distance
0       2147483647
1       2147483647
2       2147483647
```

8. Write a program to find the shortest path between all the source destination pairs (All pairs shortest path Floyd's algorithm).

SOURCE CODE :

```
#include <stdio.h>
#include <limits.h>
#define MAX_VERTICES 100
// Function to find the minimum of two integers
int min(int a, int b) {
return (a < b) ? a : b;
}
// Function to perform Floyd Warshall algorithm to find all pairs shortest path
void floydWarshall(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
int dist[V][V];
// Initialize the distance matrix with the given graph
for (int i = 0; i < V; ++i) {
for (int j = 0; j < V; ++j) {
dist[i][j] = graph[i][j];
}
}
// Calculate shortest paths using all vertices as intermediate nodes
for (int k = 0; k < V; ++k) {
for (int i = 0; i < V; ++i) {
for (int j = 0; j < V; ++j) {
if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}
}
// Print the shortest path matrix
printf("Shortest distances between all pairs of vertices:\n");
for (int i = 0; i < V; ++i) {
for (int j = 0; j < V; ++j) {
```

```
if (dist[i][j] == INT_MAX) {
printf("INF ");
} else {
printf("%d ", dist[i][j]);
}
}
printf("\n");
}
}
// Main function
int main() {
int V, edges;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &V, &edges);
int graph[MAX_VERTICES][MAX_VERTICES];
// Initialize the graph matrix with maximum integer value as infinity
for (int i = 0; i < V; ++i) {
for (int j = 0; j < V; ++j) {
if (i == j) {
graph[i][j] = 0;
} else {
graph[i][j] = INT_MAX;
}
}
}
printf("Enter edges with weights (format: source destination weight):\n");
for (int i = 0; i < edges; ++i) {
int u, v, weight;
scanf("%d %d %d", &u, &v, &weight);
graph[u][v] = weight;
}
floydWarshall(graph, V);
return 0;
}
```



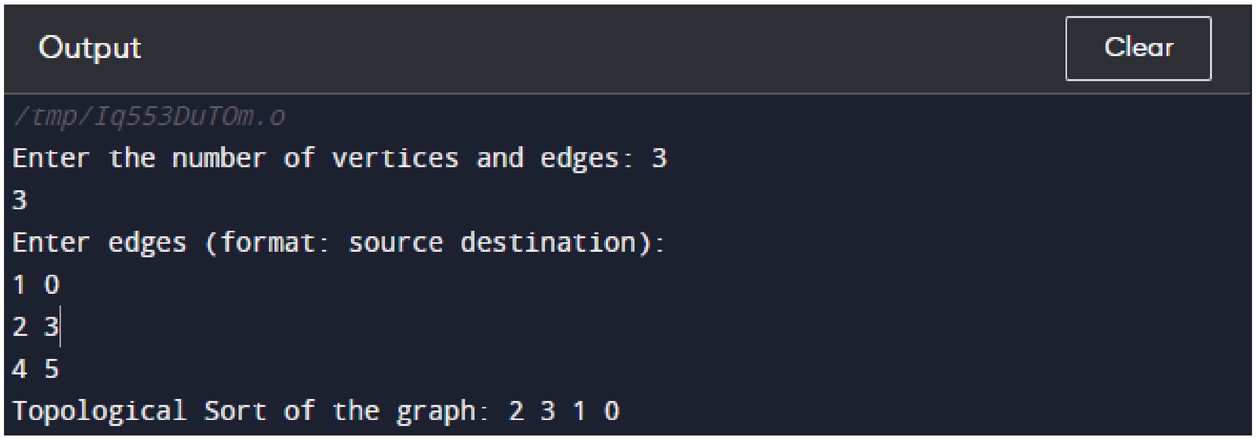
```
Enter the number of vertices and edges: 3
3
Enter edges with weights (format: source destination weight):
1 2 3
4 5 6
7 8 9
Shortest distances between all pairs of vertices:
0 INF INF
INF 0 3
INF INF 0
```

9. Write a program to arrange all the nodes of a given graph (Topological sort).

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
struct Node {
int vertex;
struct Node* next;
};
struct Graph {
int numVertices;
struct Node* adjList[MAX_VERTICES];
};
struct Node* createNode(int v) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;
newNode->next = NULL;
return newNode;
}
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
for (int i = 0; i < vertices; ++i) {
graph->adjList[i] = NULL;
}
return graph;
}
void addEdge(struct Graph* graph, int u, int v) {
struct Node* newNode = createNode(v);
newNode->next = graph->adjList[u];
graph->adjList[u] = newNode;
}
void topologicalSortUtil(int v, struct Graph* graph, bool visited[], int stack[], int* index) {
visited[v] = true;
struct Node* temp = graph->adjList[v];
while (temp != NULL) {
int adjVertex = temp->vertex;
if (!visited[adjVertex]) {
topologicalSortUtil(adjVertex, graph, visited, stack, index);
}
temp = temp->next;
}
stack[(*index)++] = v;
}
void topologicalSort(struct Graph* graph) {
```

```
int V = graph->numVertices;
bool visited[V];
for (int i = 0; i < V; ++i) {
visited[i] = false;
}
int stack[V];
int index = 0;
for (int i = 0; i < V; ++i) {
if (!visited[i]) {
topologicalSortUtil(i, graph, visited, stack, &index);
}
}
printf("Topological Sort of the graph: ");
while (index > 0) {
printf("%d ", stack[--index]);
}
printf("\n");
}
int main() {
int vertices, edges;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &vertices, &edges);
struct Graph* graph = createGraph(vertices);
printf("Enter edges (format: source destination):\n");
for (int i = 0; i < edges; ++i) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(graph, u, v);
}
topologicalSort(graph);
return 0;
}
```

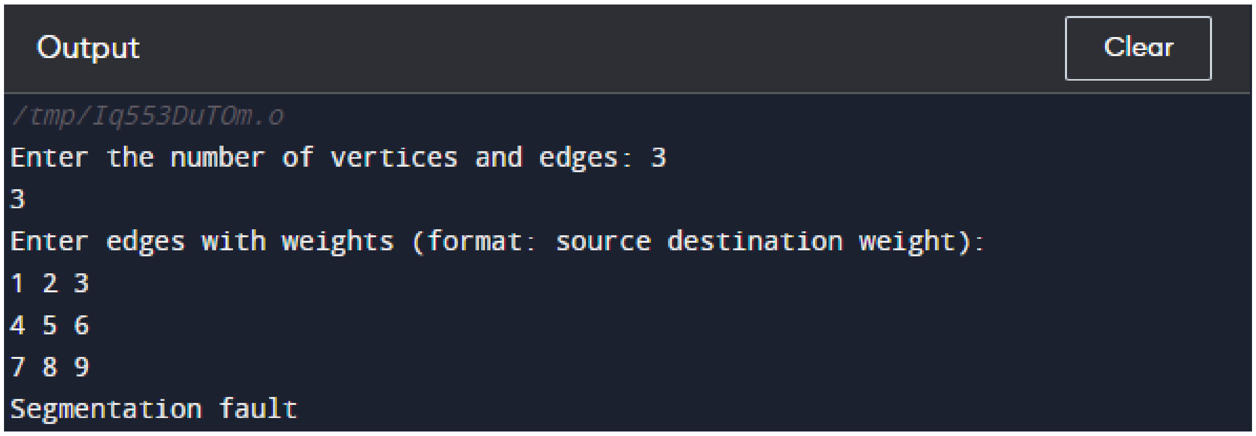


```
/tmp/Iq553DuTOm.o
Enter the number of vertices and edges: 3
3
Enter edges (format: source destination):
1 0
2 3
4 5
Topological Sort of the graph: 2 3 1 0
```

10. Write a program to find the Minimal spanning tree of a graph using Kruskal's algorithm.

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_EDGES 100
#define MAX_VERTICES 100
struct Edge {
int src, dest, weight;
};
struct Graph {
int numVertices, numEdges;
struct Edge* edges[MAX_EDGES];
};
struct Graph* createGraph(int V, int E) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = V;
graph->numEdges = E;
for (int i = 0; i < E; ++i) {
graph->edges[i] = (struct Edge*)malloc(sizeof(struct Edge));
}
return graph;
}
int find(int parent[], int i) {
if (parent[i] == i)
return i;
return find(parent, parent[i]);
}
void Union(int parent[], int rank[], int x, int y) {
int xRoot = find(parent, x);
int yRoot = find(parent, y);
if (rank[xRoot] < rank[yRoot])
parent[xRoot] = yRoot;
else if (rank[xRoot] > rank[yRoot])
parent[yRoot] = xRoot;
else {
parent[yRoot] = xRoot;
rank[xRoot]++;
}
}
int compare(const void* a, const void* b) {
struct Edge* aEdge = (struct Edge*)a;
struct Edge* bEdge = (struct Edge*)b;
return aEdge->weight - bEdge->weight;
}
void KruskalMST(struct Graph* graph) {
int V = graph->numVertices;
struct Edge resultMST[V];
int e = 0;
int i = 0;
qsort(graph->edges, graph->numEdges, sizeof(graph->edges[0]), compare);
int parent[V];
int rank[V];
for (int v = 0; v < V; ++v) {
parent[v] = v;
rank[v] = 0;
}
while (e < V - 1 && i < graph->numEdges) {
struct Edge nextEdge = *(graph->edges[i++]);
int x = find(parent, nextEdge.src);
int y = find(parent, nextEdge.dest);
if (x != y) {
resultMST[e++] = nextEdge;
Union(parent, rank, x, y);
}
}
printf("Edges in Minimum Spanning Tree (MST) using Kruskal's algorithm:\n");
for (i = 0; i < e; ++i) {
printf("(%d, %d) - weight: %d\n", resultMST[i].src, resultMST[i].dest, resultMST[i].weight);
}
}
int main() {
int V, E;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &V, &E);
struct Graph* graph = createGraph(V, E);
printf("Enter edges with weights (format: source destination weight):\n");
for (int i = 0; i < E; ++i) {
scanf("%d %d %d", &graph->edges[i]->src, &graph->edges[i]->dest, &graph->edges[i]->weight);
}
KruskalMST(graph);
return 0;
}
```

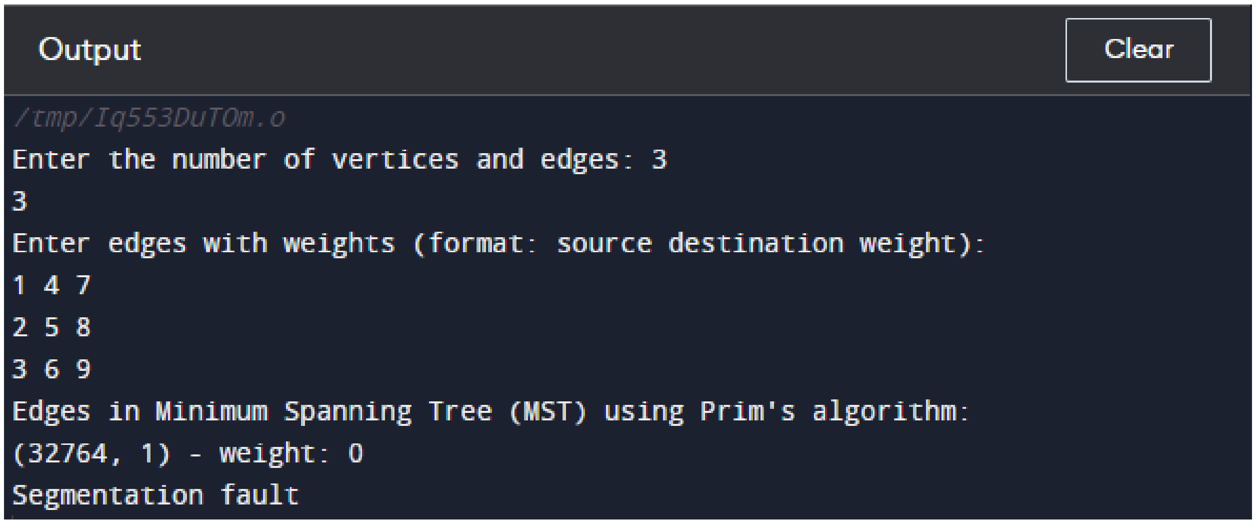


```
/tmp/Iq553DuTOm.o
Enter the number of vertices and edges: 3
3
Enter edges with weights (format: source destination weight):
1 2 3
4 5 6
7 8 9
Segmentation fault
```

11. Write a program to find Minimal spanning tree of a graph using Prim’s algorithm.

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 100
struct Graph {
int numVertices;
int graph[MAX_VERTICES][MAX_VERTICES];
};
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->numVertices = vertices;
for (int i = 0; i < vertices; ++i) {
for (int j = 0; j < vertices; ++j) {
graph->graph[i][j] = 0;
}
}
return graph;
}
int minKey(int key[], bool mstSet[], int V) {
int min = INT_MAX, min_index;
for (int v = 0; v < V; ++v) {
if (mstSet[v] == false && key[v] < min) {
min = key[v];
min_index = v;
}
}
return min_index;
}
void printMST(int parent[], struct Graph* graph, int V) {
printf("Edges in Minimum Spanning Tree (MST) using Prim's algorithm:\n");
for (int i = 1; i < V; ++i) {
printf("(%d, %d) - weight: %d\n", parent[i], i, graph->graph[i][parent[i]]);
}
}
void primMST(struct Graph* graph) {
int V = graph->numVertices;
int parent[V];
int key[V];
bool mstSet[V];
for (int i = 0; i < V; ++i) {
key[i] = INT_MAX;
mstSet[i] = false;
}
key[0] = 0;
parent[0] = -1;
for (int count = 0; count < V - 1; ++count) {
int u = minKey(key, mstSet, V);
mstSet[u] = true;
for (int v = 0; v < V; ++v) {
if (graph->graph[u][v] && mstSet[v] == false && graph->graph[u][v] < key[v]) {
parent[v] = u;
key[v] = graph->graph[u][v];
}
}
}
printMST(parent, graph, V);
}
int main() {
int V, E;
printf("Enter the number of vertices and edges: ");
scanf("%d %d", &V, &E);
struct Graph* graph = createGraph(V);
printf("Enter edges with weights (format: source destination weight):\n");
for (int i = 0; i < E; ++i) {
int u, v, weight;
scanf("%d %d %d", &u, &v, &weight);
graph->graph[u][v] = weight;
graph->graph[v][u] = weight; // For undirected graph
}
primMST(graph);
return 0;
}
```



```
/tmp/Iq553DuTOm.o
Enter the number of vertices and edges: 3
3
Enter edges with weights (format: source destination weight):
1 4 7
2 5 8
3 6 9
Edges in Minimum Spanning Tree (MST) using Prim's algorithm:
(32764, 1) - weight: 0
Segmentation fault
```