**Task 1:**

#include <iostream>

#include <vector>

**using** **namespace** std;

// Structure of a graph

**struct** Edge {

**int** src, dest, weight;

};

// Structure to represent a graph

**struct** Graph {

**int** V, E;

vector<Edge> edges;

};

// Function to create a graph

Graph createGraph(**int** V, **int** E)

{

Graph graph;

graph.V = V;

graph.E = E;

graph.edges.resize(E);

**return** graph;

}

// Function to find the vertex with minimum distance value

// from the set of vertices not yet included in MST

**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;

}

// Function to print the constructed MST stored in parent[]

**void** printMST(**int** parent[], Graph graph)

{

cout << "Edge Weight\n";

**for** (**int** i = 1; i < graph.V; i++)

cout << parent[i] << " - " << i << " " << graph.edges[i].weight << endl;

}

// Function to construct and print MST for a graph represented

// using adjacency matrix representation

**void** primMST(Graph graph)

{

**int** V = graph.V;

**int** E = graph.E;

**int** parent[V];

**int** key[V];

**bool** mstSet[V];

// Initialize all keys as INFINITE

**for** (**int** i = 0; i < V; i++)

{

key[i] = INT\_MAX;

mstSet[i] = **false**;

}

// Choose 0th vertex and make its key 0

key[0] = 0;

parent[0] = -1;

// Looping to find the minimum spanning tree

**for** (**int** count = 0; count < V - 1; count++)

{

// Pick the minimum key vertex from the set of vertices

// not yet included in MST

**int** u = minKey(key, mstSet, V);

// Add the picked vertex to the MST Set

mstSet[u] = **true**;

// Update key value and parent index of the adjacent

// vertices of the picked vertex. Consider only those

// vertices which are not yet included in MST

**for** (**int** v = 0; v < V; v++)

{

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

**if** (graph.edges[u].weight && mstSet[v] == **false** && graph.edges[u].weight < key[v])

{

parent[v] = u;

key[v] = graph.edges[u].weight;

}

}

}

// Print the constructed MST

printMST(parent, graph);

}

**int** main()

{

**int** V = 9;

**int** E = 14;

Graph graph = createGraph(V, E);

graph.edges[0].src = 0;

graph.edges[0].dest = 1;

graph.edges[0].weight = 3;

graph.edges[1].src = 0;

graph.edges[1].dest = 7;

graph.edges[1].weight = 5;

graph.edges[2].src = 1;

graph.edges[2].dest = 2;

graph.edges[2].weight = 4;

graph.edges[3].src = 1;

graph.edges[3].dest = 7;

graph.edges[3].weight = 9;

graph.edges[4].src = 2;

graph.edges[4].dest = 3;

graph.edges[4].weight = 2;

graph.edges[5].src = 2;

graph.edges[5].dest = 8;

graph.edges[5].weight = 1;

graph.edges[6].src = 2;

graph.edges[6].dest = 5;

graph.edges[6].weight = 3;

graph.edges[7].src = 3;

graph.edges[7].dest = 4;

graph.edges[7].weight = 8;

graph.edges[8].src = 3;

graph.edges[8].dest = 5;

graph.edges[8].weight = 13;

graph.edges[9].src = 4;

graph.edges[9].dest = 5;

graph.edges[9].weight = 14;

graph.edges[10].src = 5;

graph.edges[10].dest = 6;

graph.edges[10].weight = 5;

graph.edges[11].src = 6;

graph.edges[11].dest = 7;

graph.edges[11].weight = 4;

graph.edges[12].src = 6;

graph.edges[12].dest = 8;

graph.edges[12].weight = 6;

graph.edges[13].src = 7;

graph.edges[13].dest = 8;

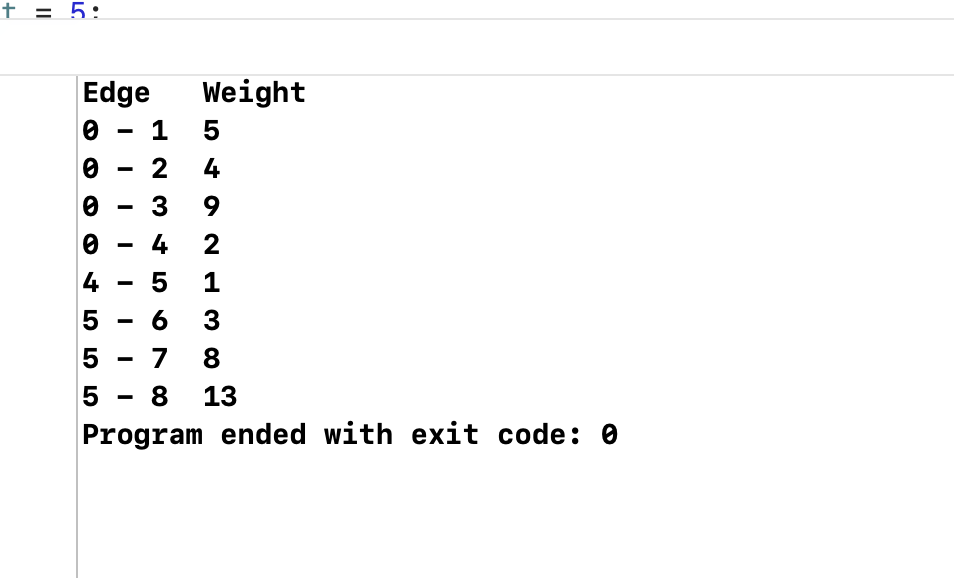
graph.edges[13].weight = 7;

primMST(graph);

**return** 0;

}

**Output:**

****

Task 2:

#include<iostream>

#include<vector>

**using** **namespace** std;

// a structure to represent an edge in graph

**struct** Edge

{

**int** src, dest, weight;

};

// a structure to represent a connected, undirected

// and weighted graph

**struct** Graph

{

// V-> Number of vertices, E-> Number of edges

**int** V, E;

// graph is represented as an array of edges

**struct** Edge\* edge;

};

// Creates a graph with V vertices and E edges

**struct** Graph\* createGraph(**int** V, **int** E)

{

**struct** Graph\* graph = **new** Graph;

graph->V = V;

graph->E = E;

graph->edge = **new** Edge[E];

**return** graph;

}

// A structure to represent a subset for union-find

**struct** subset

{

**int** parent;

**int** rank;

};

// A utility function to find set of an element i

// (uses path compression technique)

**int** find(**struct** subset subsets[], **int** i)

{

// find root and make root as parent of i

// (path compression)

**if** (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

**return** subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

**void** Union(**struct** subset subsets[], **int** x, **int** y)

{

**int** xroot = find(subsets, x);

**int** yroot = find(subsets, y);

// Attach smaller rank tree under root of high

// rank tree (Union by Rank)

**if** (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

**else** **if** (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and

// increment its rank by one

**else**

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// Compare two edges according to their weights.

// Used in qsort() for sorting an array of edges

**int** myComp(**const** **void**\* a, **const** **void**\* b)

{

**struct** Edge\* a1 = (**struct** Edge\*)a;

**struct** Edge\* b1 = (**struct** Edge\*)b;

**return** a1->weight > b1->weight;

}

// The main function to construct MST using Kruskal's algorithm

**void** KruskalMST(**struct** Graph\* graph)

{

**int** V = graph->V;

**struct** Edge result[V]; // Tnis will store the resultant MST

**int** e = 0; // An index variable, used for result[]

**int** i = 0; // An index variable, used for sorted edges

// Step 1: Sort all the edges in non-decreasing

// order of their weight. If we are not allowed to

// change the given graph, we can create a copy of

// array of edges

qsort(graph->edge, graph->E, **sizeof**(graph->edge[0]), myComp);

// Allocate memory for creating V ssubsets

**struct** subset \*subsets =

(**struct** subset\*) malloc( V \* **sizeof**(**struct** subset) );

// Create V subsets with single elements

**for** (**int** v = 0; v < V; ++v)

{

subsets[v].parent = v;

subsets[v].rank = 0;

}

// Number of edges to be taken is equal to V-1

**while** (e < V - 1)

{

// Step 2: Pick the smallest edge. And increment

// the index for next iteration

**struct** Edge next\_edge = graph->edge[i++];

**int** x = find(subsets, next\_edge.src);

**int** y = find(subsets, next\_edge.dest);

// If including this edge does't cause cycle,

// include it in result and increment the index

// of result for next edge

**if** (x != y)

{

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// print the contents of result[] to display

// the built MST

cout<<"Following are the edges in the constructed MST\n";

**for** (i = 0; i < e; ++i)

cout<<result[i].src<<" - "<<result[i].dest<<" = "<<result[i].weight<<endl;

**return**;

}

// Driver program to test above functions

**int** main()

{

**int** V = 4; // Number of vertices in graph

**int** E = 5; // Number of edges in graph

**struct** Graph\* graph = createGraph(V, E);

// add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 1;

// add edge 0-2

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 0-3

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 3;

// add edge 1-3

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 10;

// add edge 2-3

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

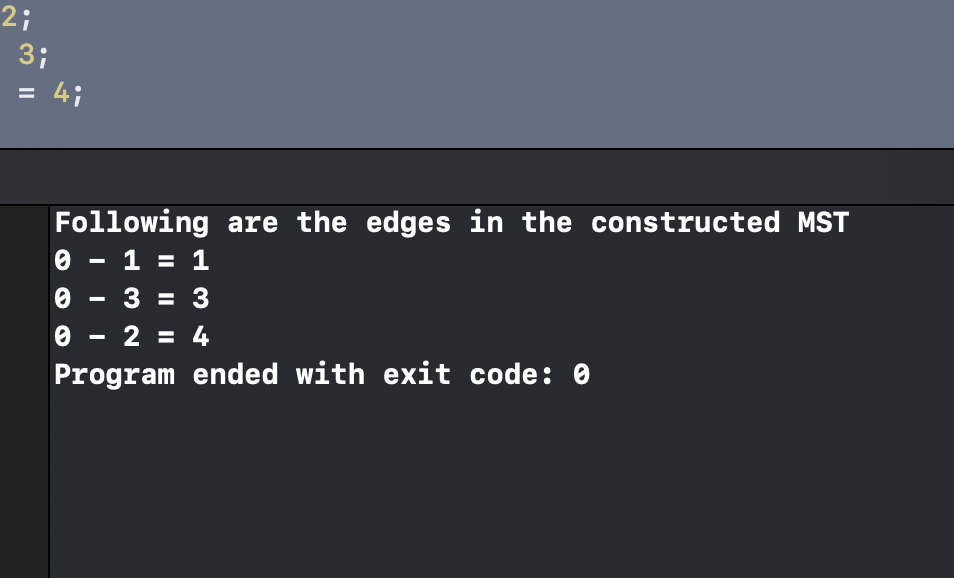
graph->edge[4].weight = 4;

KruskalMST(graph);

**return** 0;

}

Output:



Task 3:

A whiteboard with writing on it

Description automatically generated with medium confidenceText, letter

Description automatically generatedText, letter

Description automatically generatedText, letter

Description automatically generated