**IMPLEMENTING A WEIGHTED GRAPH:**

This implementation of weighted graph can be used to

1. Create a weighted Directed/Undirected graph.
2. Display the adjacency list of the graph.
3. Display the adjacency matrix of the weighted graph.
4. Perform BFS and DFS algorithms.
5. Display the elements in the graph.

**PROGRAM:**

/\*\*

 \* @file graph\_construct.cpp

 \* @author rajayogesh

 \* @brief a simple implementation of the Graph DS

 \* @version 0.1

 \* @date 2020-11-18

 \*

 \* @copyright Copyright (c) 2020

 \*

 \*/

#include <bits/stdc++.h>

using namespace std;

//structure to represent each vertices

struct Graph

{

    int data, weight;

    vector<pair<struct Graph \*,int>> edge; //list to sore the edges of the vertex

    Graph(int d) : data{d}, weight{0} {};

};

typedef struct Graph vertex;

map<int,vertex \*> monitor;       //map to store the adjacency graph

map<int,vector<pair<int,int>>> graph\_impl; //map to store the adjacency list

vertex \*start = NULL;

void printAdjMat()

{

    int n = graph\_impl.size();

    int adj\_mat[n + 1][n + 1];

    memset(adj\_mat, 0, (n + 1) \* (n + 1) \* sizeof(int));

    int i = 1;

    map<int, int> mp;

    for (auto x : graph\_impl)

    {

        adj\_mat[0][i] = adj\_mat[i][0] = x.first;

        mp[x.first] = i;

        i++;

    }

    i = 1;

    for (auto x : graph\_impl)

    {

        int j = 1;

        for (auto y : x.second)

        {

            adj\_mat[i][mp[y.first]] = y.second;

            j++;

        }

        i++;

    }

    cout << "----------------------------------  ADJACENCY MATRIX  ----------------------------------" << endl;

    for (int i = 0; i <= n; i++)

    {

        for (int j = 0; j <= n; j++)

        {

            cout << adj\_mat[i][j] << " ";

        }

        cout << endl;

    }

}

/\*\*

 \* @brief a program that uses the principal of BFS on graphs to print all the elements in the graph that are accessible directly

 \* or indirectly from the given vertex.

 \* @param stVer indicates the start vertex of the graph

 \*/

void displayGraph(vertex \*stVer)

{

    cout << "----------------------------------  ADJACENCY LIST  ----------------------------------\n"

         << endl;

    stack<vertex \*> st;

    st.push(stVer);

    vector<vertex \*> vec;

    while (!st.empty())

    {

        vertex \*temp = st.top();

        st.pop();

        if (find(vec.begin(), vec.end(), temp) == vec.end())

        {

            cout << temp->data << " ";

            vec.push\_back(temp);

        }

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

                st.push(i.first);

        }

    }

    cout << endl;

}

/\*\*

 \* @brief

 \* a function to print the adjacency list.

 \* @param graph stores the map containing the adjacency list

 \*/

void displayAdjList(map<int, vector<pair<int,int>>> graph)

{

    cout << "----------------------------------  ADJACENCY LIST  ----------------------------------\n"

         << endl;

    for (auto i = graph.begin(); i != graph.end(); i++)

    {

        cout << i->first << " ";

        for (auto j : i->second)

        {

            cout << " --> " << j.first;

        }

        cout << endl;

    }

}

/\*\*

 \* @brief

 \* a function to print the adjacency list.

 \* @param graph stores the map containing the graph itself

 \*/

void displayAdjList(map<int, vertex \*> graph)

{

    cout << "----------------------------------  ADJACENCY MAP  ----------------------------------\n"

         << endl;

    for (auto i = graph.begin(); i != graph.end(); i++)

    {

        cout << i->first << " ";

        for (auto j : i->second->edge)

        {

            cout << " --> " << j.first->data;

        }

        cout << endl;

    }

}

/\*\*

 \* @brief function to insert the vertices to the graph

 \*

 \* @param d

 \* @param s

 \*/

void insert(int d, int s,int weight)

{

    vertex \*temp = NULL;

    if (start == NULL)

    {

        start = new Graph(d);

        monitor[d] = start;

    }

    else if (monitor.find(d) == monitor.end())

        monitor[d] = new Graph(d);

    temp = monitor[d];

    pair<vertex \*, int> p;

    if (monitor.find(s) == monitor.end())

    {

        monitor[s] = new Graph(s);

    }

    p.first = monitor[s];

    p.second = weight;

    if (find(temp->edge.begin(), temp->edge.end(), p) == temp->edge.end())

    {

        temp->edge.push\_back(p);

    }

}

int main()

{

    int n; //number of vertices

    cin >> n;

    int i = n;

    while (i--)

    {

        int v;

        cin >> v; //vertex

        string s;

        getchar();

        getline(cin, s); //edges

        istringstream iss(s);

        for (string str; iss >> str;)

        {

            int n = stoi(str);

            iss >> str;

            int weight = stoi(str);

            insert(v,n,weight);

            pair<int, int> p;

            p.first = n;

            p.second = weight;

            graph\_impl[v].push\_back(p);

        }

    }

    cout << "\nDisplaying the data stored in the Graph\n";

    displayGraph(start);

    cout << "\nDisplaying the data stored in the adjacency list\n";

    displayAdjList(graph\_impl);

    cout << "\nDisplaying the data stored in the Graph Map\n";

    displayAdjList(monitor);

    cout << "\nDisplaying the Adjacency Matrix of the Graph\n";

    printAdjMat();

}

/\*\*

5

1

2 1 3 2

2

3 5 4 6

3

4 2 5 3

4

5 2 1 4

5

1 2 2 3

 \*/

**PROVIDED USER-DEFINED HEADER:**

#include <bits/stdc++.h>

using namespace std;

template <typename t>

vector<t> graphDFS(t stVer)

{

    cout << "----------------------------------  DFS ON GRAPH  ----------------------------------\n"

         << endl;

    stack<t> st;

    st.push(stVer);

    vector<t> vec;

    while (!st.empty())

    {

        t temp = st.top();

        st.pop();

        if (find(vec.begin(), vec.end(), temp) == vec.end())

        {

            vec.push\_back(temp);

        }

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

                st.push(i.first);

        }

    }

    return vec;

}

template <typename t>

vector<t> graphBFS(t stVer)

{

    cout << "----------------------------------  BFS ON GRAPH  ----------------------------------\n"

         << endl;

    queue <t> q;

    q.push(stVer);

    vector<t> vec;

    while (!q.empty())

    {

        t temp = q.front();

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

            {

                q.push(i.first);

                vec.push\_back(i.first);

            }

        }

        q.pop();

    }

    return vec;

}

**INPUT AND OUTPUT:**

C:\Users\rajay\OneDrive\Desktop\DATA STRUCTURES\practice\_test\GRAPHS>.\"weighted\_graph\_construct.exe"

5

1

2 1 3 2

2

3 5 4 6

3

4 2 5 3

4

5 2 1 4

5

1 2 2 3

Displaying the data stored in the Graph

----------------------------------  ADJACENCY LIST  ----------------------------------

1 3 5 2 4

Displaying the data stored in the adjacency list

----------------------------------  ADJACENCY LIST  ----------------------------------

1  --> 2 --> 3

2  --> 3 --> 4

3  --> 4 --> 5

4  --> 5 --> 1

5  --> 1 --> 2

Displaying the data stored in the Graph Map

----------------------------------  ADJACENCY MAP  ----------------------------------

1  --> 2 --> 3

2  --> 3 --> 4

3  --> 4 --> 5

4  --> 5 --> 1

5  --> 1 --> 2

Displaying the Adjacency Matrix of the Graph

----------------------------------  ADJACENCY MATRIX  ----------------------------------

0 1 2 3 4 5

1 0 1 2 0 0

2 0 0 5 6 0

3 0 0 0 2 3

4 4 0 0 0 2

5 2 3 0 0 0

**HEADER FILES FOR BFS AND DFS ON GRAPH:**

#include <bits/stdc++.h>

using namespace std;

template <typename t>

vector<t> graphDFS(t stVer)

{

    cout << "----------------------------------  DFS ON GRAPH  ----------------------------------\n"

         << endl;

    stack<t> st;

    st.push(stVer);

    vector<t> vec;

    while (!st.empty())

    {

        t temp = st.top();

        st.pop();

        if (find(vec.begin(), vec.end(), temp) == vec.end())

        {

            vec.push\_back(temp);

        }

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

                st.push(i.first);

        }

    }

    return vec;

}

template <typename t>

vector<t> graphBFS(t stVer)

{

    cout << "----------------------------------  BFS ON GRAPH  ----------------------------------\n"

         << endl;

    queue <t> q;

    q.push(stVer);

    vector<t> vec;

    while (!q.empty())

    {

        t temp = q.front();

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

            {

                q.push(i.first);

                vec.push\_back(i.first);

            }

        }

        q.pop();

    }

    return vec;

}

**HEADER FOR A GRAPH STRUCTURE :**

/\*\*

 \* @file graph\_implementation.cpp

 \* @author your name (you@domain.com)

 \* @brief

 \* @version 0.1

 \* @date 2020-11-27

 \*

 \* @copyright Copyright (c) 2020

 \*

 \*/

#include <bits/stdc++.h>

#include "graph\_traversals.h"

using namespace std;

//structure to represent each vertices

struct Graph

{

    int data, weight;

    vector<pair<struct Graph \*, int>> edge; //list to sore the edges of the vertex

    Graph(int d) : data{d}, weight{INT\_MAX} {};

};

typedef struct Graph vertex;

map<int, vertex \*> monitor;                  //map to store the adjacency graph

map<int, vector<pair<int, int>>> graph\_impl; //map to store the adjacency list

map<int, vector<pair<int, int>>> adj\_impl;   //map to store the generated adjacency list

vertex \*start = NULL;

void printAdjMat(map<int, vector<pair<int, int>>> gp\_impl)

{

    int n = gp\_impl.size();

    int adj\_mat[n + 1][n + 1];

    memset(adj\_mat, 0, (n + 1) \* (n + 1) \* sizeof(int));

    int i = 1;

    map<int, int> mp;

    for (auto x : gp\_impl)

    {

        adj\_mat[0][i] = adj\_mat[i][0] = x.first;

        mp[x.first] = i;

        i++;

    }

    i = 1;

    for (auto x : gp\_impl)

    {

        int j = 1;

        for (auto y : x.second)

        {

            adj\_mat[i][mp[y.first]] = y.second;

            j++;

        }

        i++;

    }

    cout << "----------------------------------  ADJACENCY MATRIX  ----------------------------------" << endl;

    for (int i = 0; i <= n; i++)

    {

        for (int j = 0; j <= n; j++)

        {

            cout << adj\_mat[i][j] << " ";

        }

        cout << endl;

    }

}

/\*\*

 \* @brief a program that uses the principal of BFS on graphs to print all the elements in the graph that are accessible directly

 \* or indirectly from the given vertex.

 \* @param stVer indicates the start vertex of the graph

 \*/

void displayGraph(vertex \*stVer)

{

    cout << "----------------------------------  ADJACENCY LIST  ----------------------------------\n"

         << endl;

    stack<vertex \*> st;

    st.push(stVer);

    vector<vertex \*> vec;

    while (!st.empty())

    {

        vertex \*temp = st.top();

        st.pop();

        if (find(vec.begin(), vec.end(), temp) == vec.end())

        {

            cout << temp->data << " ";

            vec.push\_back(temp);

        }

        for (auto i : temp->edge)

        {

            if (find(vec.begin(), vec.end(), i.first) == vec.end())

                st.push(i.first);

        }

    }

    cout << endl;

}

/\*\*

 \* @brief

 \* a function to print the adjacency list.

 \* @param graph stores the map containing the adjacency list

 \*/

void displayAdjList(map<int, vector<pair<int, int>>> graph)

{

    cout << "----------------------------------  ADJACENCY LIST  ----------------------------------\n"

         << endl;

    for (auto i = graph.begin(); i != graph.end(); i++)

    {

        cout << i->first << " ";

        for (auto j : i->second)

        {

            cout << " --> " << j.first;

        }

        cout << endl;

    }

}

/\*\*

 \* @brief

 \* a function to print the adjacency list.

 \* @param graph stores the map containing the graph itself

 \*/

void displayAdjList(map<int, vertex \*> graph)

{

    cout << "----------------------------------  ADJACENCY MAP  ----------------------------------\n"

         << endl;

    for (auto i = graph.begin(); i != graph.end(); i++)

    {

        vector<pair<int, int>> vec;

        cout << i->first << " ";

        for (auto j : i->second->edge)

        {

            pair<int, int> p;

            p.first = j.first->data;

            p.second = j.second;

            cout << " --> " << j.first->data;

            vec.push\_back(p);

        }

        adj\_impl[i->first] = vec;

        cout << endl;

    }

}

/\*\*

 \* @brief function to create a directed graph

 \*

 \* @param d from vertex

 \* @param s to vertex

 \* @param weight cost of traversal

 \*/

void directedInsert(int d, int s, int weight)

{

    vertex \*temp = NULL;

    if (start == NULL)

    {

        start = new Graph(d);

        //start->weight = 0;

        monitor[d] = start;

    }

    else if (monitor.find(d) == monitor.end())

        monitor[d] = new Graph(d);

    temp = monitor[d];

    pair<vertex \*, int> p;

    if (monitor.find(s) == monitor.end())

    {

        monitor[s] = new Graph(s);

    }

    p.first = monitor[s];

    p.second = weight;

    if (find(temp->edge.begin(), temp->edge.end(), p) == temp->edge.end())

    {

        temp->edge.push\_back(p);

    }

}

/\*\*

 \* @brief to create an undirected graph

 \*

 \* @param d from vertex

 \* @param s  to vertex

 \* @param weight cost of traversal

 \*/

void undirectedInsert(int d, int s, int weight)

{

    vertex \*temp = NULL;

    if (start == NULL)

    {

        start = new Graph(d);

        //start->weight = 0;

        monitor[d] = start;

    }

    else if (monitor.find(d) == monitor.end())

        monitor[d] = new Graph(d);

    temp = monitor[d];

    pair<vertex \*, int> p;

    if (monitor.find(s) == monitor.end())

    {

        monitor[s] = new Graph(s);

    }

    p.first = monitor[s];

    p.second = weight;

    if (find(temp->edge.begin(), temp->edge.end(), p) == temp->edge.end())

    {

        temp->edge.push\_back(p);

    }

    pair<vertex \*, int> e;

    e.first = monitor[d];

    e.second = weight;

    if (find(monitor[s]->edge.begin(), monitor[s]->edge.end(), e) == monitor[s]->edge.end())

    {

        monitor[s]->edge.push\_back(e);

    }

}

int getChoice()

{

    cout << "Please select your Graph\n1.)Directed Graph\n2.)Undirected Graph\n";

    int ch;

    cin >> ch;

    return (ch == 1 || ch == 2) ? ch : getChoice();

}

**IMPKEMENTATION OF KRUSKAL’S ALGO:**

#include<bits/stdc++.h>

#include"Graph\_includes/Graph.h"

using namespace std;

map<vertex \*,vertex \*> kruskalsMST(map<int, vertex \*> gp)

{

    map<vertex \*, vertex \*> mp;

    for (auto i = gp.begin(); i != gp.end();i++)

    {

        for (auto j = i->second->edge.begin(); j != i->second->edge.end();j++)

        {

            if(j->first->weight>(j->second))

            {

                j->first->weight =  j->second;

                mp[j->first]=i->second;

            }

        }

    }

    return mp;

}

int main()

{

    int ch = getChoice();

    int n; //number of vertices

    cin >> n;

    int i = n;

    while (i--)

    {

        int v;

        cin >> v; //vertex

        string s;

        getchar();

        getline(cin, s); //edges

        istringstream iss(s);

        for (string str; iss >> str;)

        {

            int n = stoi(str);

            iss >> str;

            int weight = stoi(str);

            if (ch == 1)

                directedInsert(v, n, weight);

            else

                undirectedInsert(v, n, weight);

            pair<int, int> p;

            p.first = n;

            p.second = weight;

            graph\_impl[v].push\_back(p);

        }

    }

    cout << "\nDisplaying the data stored in the Graph\n";

    displayGraph(start);

    cout << "\nDisplaying the data stored in the adjacency list\n";

    displayAdjList(graph\_impl);

    cout << "\nDisplaying the data stored in the Graph Map\n";

    displayAdjList(monitor);

    cout << "\nDisplaying the Adjacency Matrix of the Graph\n";

    printAdjMat(adj\_impl);

    for (auto i : graphDFS<vertex \*>(start))

        cout << i->data << " ";

    cout << endl;

    for (auto i : graphBFS<vertex \*>(start))

        cout << i->data << " ";

    cout << endl;

    for(auto i:kruskalsMST(monitor))

    {

        cout << i.second->data << " ---> " << i.first->data << endl;

    }

}

/\*\*

5

1

2 1 3 2

2

3 5 4 6

3

4 2 5 3

4

5 2 1 4

5

1 2 2 3

5

3

1 4 5 3

1

5 2 2 5 4 6

5

4 1 6 4

4

2 2 6 2

2

6 3

\*/