```
/********* graph.h ********/


#ifndef MST
#define MST

#include<iostream>
#include<vector>
#include<array>
#include<tuple>

using namespace std;

#endif


class edge
{
    public:
        int vertex; // vertex endpoint of an edge
        int weight; // edge weight (distance)

        edge(int V = 1, int W = 0): vertex(V), weight(W) {}
};

    // encapsulates the info for a particular vertex
class vertexElem
{
    public:
        int vertexID;
        vector<edge> edgeList;

        vertexElem( int ID = 0, vector<edge> EL = vector<edge>() ):
                    vertexID(ID), edgeList(EL) {}
};

using vertElemItr = vector< vertexElem >::iterator;

using intTup = tuple<int, int, int>;
using vectOfTuples = vector< tuple<int, int, int> >;

class graph
{
    int size; // current graph size (amount of nodes/vertices)

        // graph implemented via adjacency list
    vector< vertexElem > vertices;

    public:
            // default constructor creates a graph of just one unconnected
            // vertex
        graph(int S = 1)
            { size = S; vertices = vector< vertexElem >(size); }

            // constuctor overloaded to create graph by specifying size,
            // edge density, and distance range:
            // S = size, density = edge density, maxd = distance range
        graph(int S, double density, int maxd);

        graph( vector<int> vrtx );

        graph(int vrtx, vectOfTuples EG);

        graph(vectOfTuples EG);

            // total vertex count
        int getVertexCount() { return size; }

            // total edge count
        int getEdges();
```

```
            // returns pointer to node if node "n" exists, returns nullptr if
            // node is nonexistent
        vector< vertexElem >::iterator nodeExist(int n);

        bool nodeFound(int x);

            // returns true if there's an edge from node x to y
        bool isAdjacent(int x, int y);

            // get all vertices connected to x
        vector<int> getNeighbors(int x);

            // adds edge between x & y, if one is currently nonexistent
            // returns true if edge was added, if not return false
        bool addEdge(int x, int y, int d = 1);

            // if there's an edge between x & y, delete it and return true
            // else return false
        bool deleteEdge(int x, int y);

            // get edge weight/distance from x to y,
            // if there is no edge return -1
        int getEdgeValue(int x, int y);

            // get the average path length of all the nodes connected to
            // node "n"
        double avePathLength(int n);
};


/********************** graph.cpp *****************/

#include "graph.h"



    // calculates probablity of edges between vertices
inline double prob()
    { return static_cast<double>( rand() ) / static_cast<double>(RAND_MAX); }

    // calculates distance between edges
inline int getDistance(int drange) { return rand() % drange + 1; }

    // constuctor overloaded to create graph by specifying size, edge density,
    // and distance range:
    // S = size, density = edge density, maxd = distance range
graph::graph(int S, double density, int maxd)
{
    srand( time(nullptr) );

    size = S;

    vertices = vector< vertexElem >(size);

    int vID = 1;

    for(auto& V: vertices)
    {
        V = vertexElem(vID);
        vID++;
    }

    int d; // edge distance
    int i = 0;

    for(auto& V: vertices)
    {
        for(int j = i; j < size; j++)
        {
```

```
            if (j != i)
            {
                if (prob() < density)
                {
                    d = getDistance(maxd);

                    addEdge(i+1, j+1, d);
                }
            }
        }

        i++;
    }
}

graph::graph( vector<int> vrtx )
{
    size = 0;

    vertices = vector< vertexElem >();

    for(auto& V: vrtx)
    {
        vertices.push_back(vertexElem(V));
        size++;
    }
}

graph::graph(int vrtx, vectOfTuples EG)
{
    size = 0;

    vertices = vector< vertexElem >();

    for(int i = 0; i < vrtx; ++i)
    {
        vertices.push_back( vertexElem(i) );
        ++size;
    }

    for(auto& e: EG)
        addEdge( get<0>(e), get<1>(e), get<2>(e) );
}

graph::graph(vectOfTuples EG)
{
    size = 0;

    vertices = vector< vertexElem >();

    for(auto& e: EG)
    {
        if ( !nodeFound( get<0>(e) ) )
        {
            vertices.push_back( vertexElem( get<0>(e) ) );
            ++size;
        }

        if ( !nodeFound( get<1>(e) ) )
        {
            vertices.push_back( vertexElem( get<1>(e) ) );
            ++size;
        }

        addEdge( get<0>(e), get<1>(e), get<2>(e) );
    }

    //cout << "size: " << size << endl;
}
```

```
    // total edge count
int graph::getEdges()
{
    int count = 0;

    for(int i = 0; i < size; i++)
    {
        for(auto& E: vertices[i].edgeList)
        {
            if(E.vertex > vertices[i].vertexID)
                count++;
        }
    }

    return count;
}

    // returns pointer to node if node "n" exists, returns nullptr if
    // node is nonexistent
vertElemItr graph::nodeExist(int n)
{
    for(auto itr = vertices.begin(); itr != vertices.end(); itr++)
    {
        if(itr->vertexID == n)
            return itr;
    }

    //return vertices.end();
    return static_cast< vertElemItr >(nullptr);
}

bool graph::nodeFound(int x)
{
    vertElemItr xptr = nodeExist(x);

    return xptr != static_cast< vertElemItr >(nullptr) ? true: false;
}

    // returns true if there's an edge from node x to y
bool graph::isAdjacent(int x, int y)
{
    vertElemItr xptr = nodeExist(x);
    vertElemItr yptr = nodeExist(y);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
    bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

    if ( xfound  && yfound )
    {
        for(auto& E: xptr->edgeList)
        {
            if (E.vertex == y)
                return true;
        }

        return false;
    }
    else
        return false;
}

    // get all vertices connected to x
vector<int> graph::getNeighbors(int x)
{
    vertElemItr xptr = nodeExist(x);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;

    vector<int> neighbors = vector<int>(0);
```

```
    if ( xfound )
    {
        for(auto& E: xptr->edgeList)
            neighbors.push_back(E.vertex);
    }

    return neighbors;
}

    // adds edge between x & y, if one is currently nonexistent
    // returns true if edge was added, if not return false
bool graph::addEdge(int x, int y, int d)
{
    auto xptr = nodeExist(x);
    auto yptr = nodeExist(y);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
    bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

    if ( xfound  && yfound )
    {
        if ( isAdjacent(x, y) && isAdjacent(y, x) )
            return false;
        else
        {
            xptr->edgeList.push_back( edge(y, d) );
            yptr->edgeList.push_back( edge(x, d) );

            return true;
        }
    }
    else
        return false;
}

    // if there's an edge between x & y, delete it and return true
    // else return false
bool graph::deleteEdge(int x, int y)
{
    auto xptr = nodeExist(x);
    auto yptr = nodeExist(y);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
    bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

    if ( xfound  && yfound )
    {
        if ( isAdjacent(x, y) && isAdjacent(y, x) )
        {
            for (auto itr = xptr->edgeList.begin(); itr != xptr->edgeList.end(); itr++)
            {
                if (itr->vertex == y)
                {
                    xptr->edgeList.erase(itr);
                    break;
                }
            }

            for (auto itr = yptr->edgeList.begin(); itr != yptr->edgeList.end(); itr++)
            {
                if (itr->vertex == x)
                {
                    yptr->edgeList.erase(itr);
                    break;
                }
            }

            return true;
        }
        else
```

```
            return false;
    }
    else
        return false;
}


    // get edge weight/distance from x to y, if there is no edge return -1
int graph::getEdgeValue(int x, int y)
{
    auto xptr = nodeExist(x);
    auto yptr = nodeExist(y);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
    bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

    if ( xfound  && yfound )
    {
        for(auto& V: xptr->edgeList)
        {
            if (V.vertex == y)
                return V.weight;
        }

        return -1;
    }
    else
        return -1;
}


    // get the average path length of all the nodes connected to node "n"
double graph::avePathLength(int n)
{
    double total = 0;

    int eSize = 0;

    for(auto& V: vertices)
    {
        if (V.vertexID == n)
        {
            eSize = V.edgeList.size();

            for(auto& E: V.edgeList)
                total += static_cast<double>(E.weight);

            break;
        }
    }

    return total / static_cast<double>( eSize );
}


/********** priorityQueue.h ***********/

#ifndef MST
#define MST

#include<iostream>
#include<vector>
#include<array>
#include<tuple>

using namespace std;

#endif
```

```
    // class to represent edges in spanning tree
class nodeEdge
{
    public:
        int node;
        array<int, 2> edge;
        int cost; // edge cost

        nodeEdge(int N = 0, array<int, 2> A = array<int, 2>{0, 0}, int C = 0):
                                                node(N), edge(A), cost(C) {}
};

class priorityQueue
{
    vector<nodeEdge> pqArray;
    int size;

        // make subtree with root at given index priority queue compliant
    void heapify(int);

        // get parent node index of node at index i
    int parent(int i) { return (i-1)/2; }

        // get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

        // get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    public:
        priorityQueue(int S = 0) { size = S; pqArray = vector<nodeEdge>(size); }

            // removes the top element of the queue
        nodeEdge getMin();

            // checks to see if the priority queue contains node <n>
            // if so return the index, if not return -1
        int contains(int EC);

            // insert queue element "QE" into queue
        void insert(nodeEdge QE);

            // Decreases value of key at index 'i' to new_val
        void decreaseKey(int i, int new_val);

            // returns the top element of the queue.
        nodeEdge top() { return pqArray[0]; }

            // returns the number of queue elements
        int getSize() { return size; }

        vector<nodeEdge> getQueContents();

        bool isEmpty() { return size == 0 ? true: false; }

        bool notEmpty() { return size > 0 ? true: false; }

            // emtpy out the Priority Queue
        void emptyPQ()
        {
            pqArray.clear();
            size = 0;
        }

            // modify a value in the Priority Queue
        void modVal(int idx, nodeEdge n) { pqArray[idx] = n; }
};


/*********************** priorityQueue.cpp ***************/
```

```cpp
#include "priorityQueue.h"

void PQswap(nodeEdge& x, nodeEdge& y)
{
    nodeEdge temp = x;
    x = y;
    y = temp;
}

    // make subtree with root at given index priority queue compliant
void priorityQueue::heapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;

    if (l < size && pqArray[l].cost < pqArray[i].cost)
        smallest = l;

    if (r < size && pqArray[r].cost < pqArray[smallest].cost)
        smallest = r;

    if (smallest != i)
    {
        PQswap(pqArray[i], pqArray[smallest]);
        heapify(smallest);
    }
}

    // removes the top element of the queue
nodeEdge priorityQueue::getMin()
{
    if (size <= 0)
        return nodeEdge(-1, array<int, 2>{-1, -1}, -1);

    nodeEdge root;

    if (size == 1)
    {
        size--;

        root = pqArray[0];
        pqArray.pop_back();

        return root;
    }

    root = pqArray[0];

    pqArray[0] = pqArray[size-1];

    size--;

    heapify(0);

    pqArray.pop_back();

    return root;
}

    // checks to see if the priority queue contains node <n>
    // if so return the index, if not return -1
int priorityQueue::contains(int n)
{
    for(int i = 0; i < size; i++)
    {
        if (pqArray[i].node == n)
            return i;
    }
```

```
    return -1;
}

    // insert queue element "QE" into queue
void priorityQueue::insert(nodeEdge QE)
{
    size++;
    int i = size - 1;

    pqArray.push_back(QE);

        // make priority queue compliant
    while (i != 0 && pqArray[parent(i)].cost > pqArray[i].cost)
    {
       PQswap(pqArray[i], pqArray[parent(i)]);
       i = parent(i);
    }
}

    // Decreases value of key at index 'i' to new_val
void priorityQueue::decreaseKey(int i, int new_val)
{
    pqArray[i].cost = new_val;

        // make priority queue compliant
    while (i != 0 && pqArray[parent(i)].cost > pqArray[i].cost)
    {
    swap(pqArray[i], pqArray[parent(i)]);
    i = parent(i);
    }
}

vector<nodeEdge> priorityQueue::getQueContents()
{
    vector<nodeEdge> elements;

    for(auto& E: pqArray)
        elements.push_back(E);

    return elements;
}


/************************ mst.h ****************/

#ifndef MST
#define MST

#include<iostream>
#include<vector>
#include<array>
#include<tuple>

using namespace std;

#endif

#include "graph.h"
#include "priorityQueue.h"



    // minimum spanning tree class
class mst
{
    priorityQueue openSet;
    graph mstGraph;
    vector<nodeEdge> closedSet;
```

```
        // returns true if given node is in the closed set
    bool nodeInClosedSet(int n);

    public:
            // constructor to appropriately initialize an instance of the
            // minimum spanning tree class
        mst( graph G = graph(),
            priorityQueue OS = priorityQueue(),
            vector<nodeEdge> CS = vector<nodeEdge>() ):
                                mstGraph(G), openSet(OS), closedSet(CS) {}

            // method to find minimum spanning tree taking a start node as
            // input, algorithm utilized is based on Jarnik-Prim
        vector<nodeEdge> getMinTree(int n = 0);
};


/********************** mst.cpp ***************/

#include "mst.h"


    // returns true if given node is in the closed set
bool mst::nodeInClosedSet(int n)
{
    for(auto& V: closedSet)
    {
        if (V.node == n)
            return true;
    }

    return false;
}

    // method to find minimum spanning tree, taking a start node as input
    // algorithm utilized is  based on Jarnik-Prim
vector<nodeEdge> mst::getMinTree(int n)
{
        // initialize open and closed sets, to empty
    openSet.emptyPQ();
    closedSet.clear();

    openSet.insert( nodeEdge(n) ); // place start node in open set

    nodeEdge ce; // current edge being examined

    vector<int> nodeNeighbors;

    int ev; // edge value between a node and a neighbor

    int idx; // index of a nodeEdge

    vector<nodeEdge> neV;

    while( openSet.notEmpty() )
    {
        ce = openSet.getMin();
        closedSet.push_back(ce);

        nodeNeighbors = mstGraph.getNeighbors( ce.node );

            // if a node has no neighbors then a minimum spanning tree cannot
            // be found
        if (nodeNeighbors.size() == 0)
            return vector<nodeEdge>{ nodeEdge(-1, array<int, 2>{-1, -1}, -1) };

        for(auto& N: nodeNeighbors)
        {
            if( nodeInClosedSet(N) )
                continue;
```

```
            ev = mstGraph.getEdgeValue(ce.node, N);

            idx = openSet.contains(N);

                // if edge is not already in open set, add it to open set, if
                // edge is in open set, update it if the newly found edge value
                // is lower
            if (idx < 0)
                openSet.insert( nodeEdge(N, array<int, 2>{ce.node, N}, ev) );
            else
            {
                neV = openSet.getQueContents();

                if (ev < neV[idx].cost)
                {
                    openSet.decreaseKey(idx, ev);

                        // need to find new index, after decreasing the PQ key
                        // the keys in question are the edge cost
                    idx = openSet.contains(N);

                    openSet.modVal(idx, nodeEdge(N, array<int, 2>{ce.node, N}, ev) );
                }
            }
        }
    }
        // checks to see if a minimum spanning tree could be found, a valid
        // minimum spanning tree is found if the node count in the closed set
        // matches the node count of the graph, so if a valid minimum spanning
        // tree is found return it, if not return some representation
        // indicating that one was not found
    if( mstGraph.getVertexCount() == closedSet.size() )
    {
        auto itr = closedSet.begin(); closedSet.erase(itr);
        return closedSet;
    }
    else
        return vector<nodeEdge>{ nodeEdge(-1, array<int, 2>{-1, -1}, -1) };
}


/*********************** main.cpp *******************/


#include<fstream>
#include<iomanip>

#include "mst.h"



int main(int argc, char** argv)
{
    ifstream fin("graph.dat"); // open file and connect it to a stream

    if ( fin.bad() )
    {
        cerr << "Error opening file!!" << endl;
        return 1;
    }

    int gsize; // node count

    fin >> gsize; // read in from the file the node count

    int v1, v2, w; // node pairs and their edge cost

    vectOfTuples edgeW; // vector to store the edges and their costs
```

```
        // read in from the file the edges and the cost associated with each
        // edge, and store that into a vector
    while( fin.good() )
    {
        fin >> v1 >> v2 >> w;

        edgeW.push_back( intTup{v1, v2, w} );
    }

    fin.close();

        // create an instance of the minimum spanning tree class,
        // with a graph instance as input, the graph instance has as it's input
        // the vector from above
    mst mstG{ graph(edgeW) };

    cout << "EDGE\t\tCOST" << endl;

    int total{0}; // used to calculate edge cost total

        // find the minimum spanning tree and display it
    for(auto& V: mstG.getMinTree() )
    {
        cout << "(" << V.edge[0] << ", " << V.edge[1] << ")";
        cout << setw(12) << setfill(' ') << right << V.cost  << '\n';
        total += V.cost;
    }

    cout << "\nTOTAL EDGE COST: " << total << endl;

    return 0;
}

********************************************************************************

PROGRAM OUTPUT:

EDGE            COST
(0, 2)           2
(2, 9)           1
(9, 8)           3
(8, 4)           1
(4, 7)           1
(4, 15)          2
(7, 10)          2
(15, 19)         2
(9, 12)          3
(12, 3)          1
(12, 11)         1
(12, 17)         1
(11, 14)         1
(17, 1)          1
(14, 18)         1
(1, 6)           1
(18, 5)          1
(5, 16)          2
(9, 13)          3

TOTAL EDGE COST: 30
```