Lab 11

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 11: Minimum Spanning Tree

A spanning tree is a tree that connects all vertices of a undirected graph. A minimum spanning tree is a spanning tree with least possible weight. The tree weight is measured by adding up all graph edge weight.

Kruskal Algorithm

As minimum spanning tree can be seen as a matroid problem, we can apply the general greedy algorithm for solving matroid problems here. This algorithm is also called Kruskal algorithm particularly. It sorts the graph edges with weight. It maintain a set for each vertices. At first all vertices have their own set. In each iteration the algorithm takes the next least weight edge and see whether both the two vertices belong to the same set. If both vertices are in the same set, the algorithm skips that edge as adding that edge to the tree would make a cycle and it would not be a tree anymore. If the vertices are on different sets, the algorithm takes the edge to make the tree and finally merge the two sets of the two vertices into the same set so that all elements belong to the same set.

The Kruskal's algorithm is given below:

```
KRUSKAL(G)
        For each vertex v
                MakeSet(v)
        Sort the graph edges into non-decreasing order by their weight
        cost = 0
        For each edge (u,v,weight) taken after the sorting
                IF FindSet(u) is not equal to FindSet(v)
                        Mark/Print u, v as a minimum spanning tree edge
                        UnionSet(u,v)
                        cost += weight
```

Simple graph construction

As this algorithm does not require to find all adjacent nodes from a node, we do not need to implement any graph structure to maintain all neighbours by using adjacency list or adjacency matrix. Rather we can just put all the edges into an edge structure below:

```
typedef struct edge_struct
{
        int u1;
        int u2;
        int cost;
} Edge;
```

Edge sorting

Consider the undirectional graph below for 9 nodes (0,1,2,3...8) and 14 edges where {u,v,d} indicates that there is an edge between u and v with cost d.

{0, 1, 4}
{0, 5, 8}
{1, 2, 8}
{1, 5, 11}
{2, 6, 2}
{2, 3, 7}
{2, 8, 4}
{3, 4, 9}
{3, 8, 14}
{4, 8, 10}
{5, 6, 7}
{5, 7, 1}
{6, 7, 6}
{7, 8, 2}

The edges can be sorted by the built-in qsort method defined in stdlib.h. Here is a sample code to use qsort method:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct edge_struct
{
        int u1;
        int u2;
        int cost;
} Edge;

int compare(const void *a, const void *b)
{
        return ( ((Edge *)a)->cost - ((Edge *)b)->cost );
}

int main()
{
        int i;
        int numEdge = 14;

        Edge edgeList[14] = {
                                {0, 1, 4},
                                {0, 5, 8},
                                {1, 2, 8},
                                {1, 5, 11},
                                {2, 6, 2},
                                {2, 3, 7},
                                {2, 8, 4},
                                {3, 4, 9},
                                {3, 8, 14},
                                {4, 8, 10},
```

```
                                {5, 6, 7},
                                {5, 7, 1},
                                {6, 7, 6},
                                {7, 8, 2}
                        };

                qsort(edgeList, numEdge, sizeof(Edge), compare);

                for (i = 0; i < numEdge; i++)
                        printf("%d %d %d\n", edgeList[i].u1, edgeList[i].u2, edgeList[i].cost);

                return 0;
        }
```

Set Implementation

We will need to implement a few set operations such as UNION, FINDSET for this algorithm. The easiest way to implement set is by a parent[] array that holds the parent of an element. At the beginning all element will hold only themselves.

```
MAKE_SET
        For all n
                parent[n] = n
```

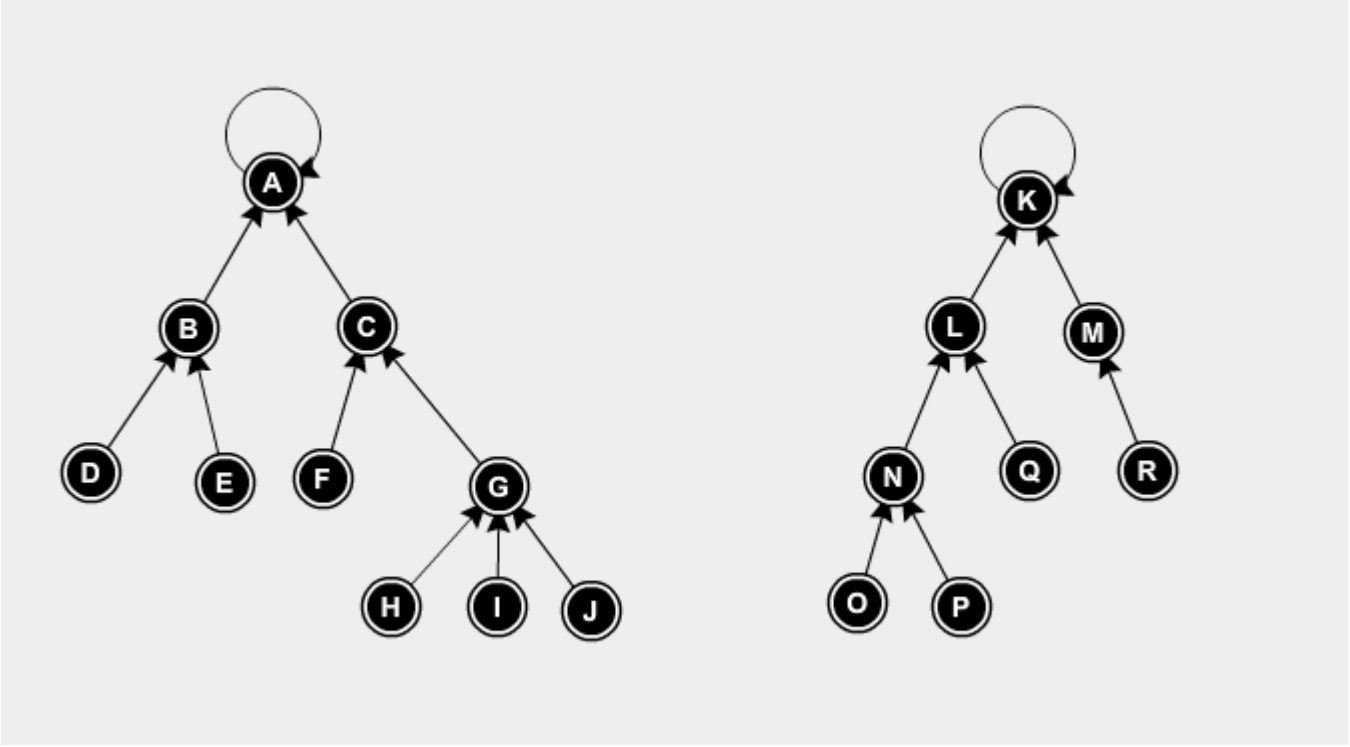
Every set will be represented by the root element. If the elements 1,3,5,6 belongs to the same set all of them would have the same root 1. We can implement the root the following way:

```
FIND_SET(n)
        While parent[n] is not equal to n
                n = parent[n]
```

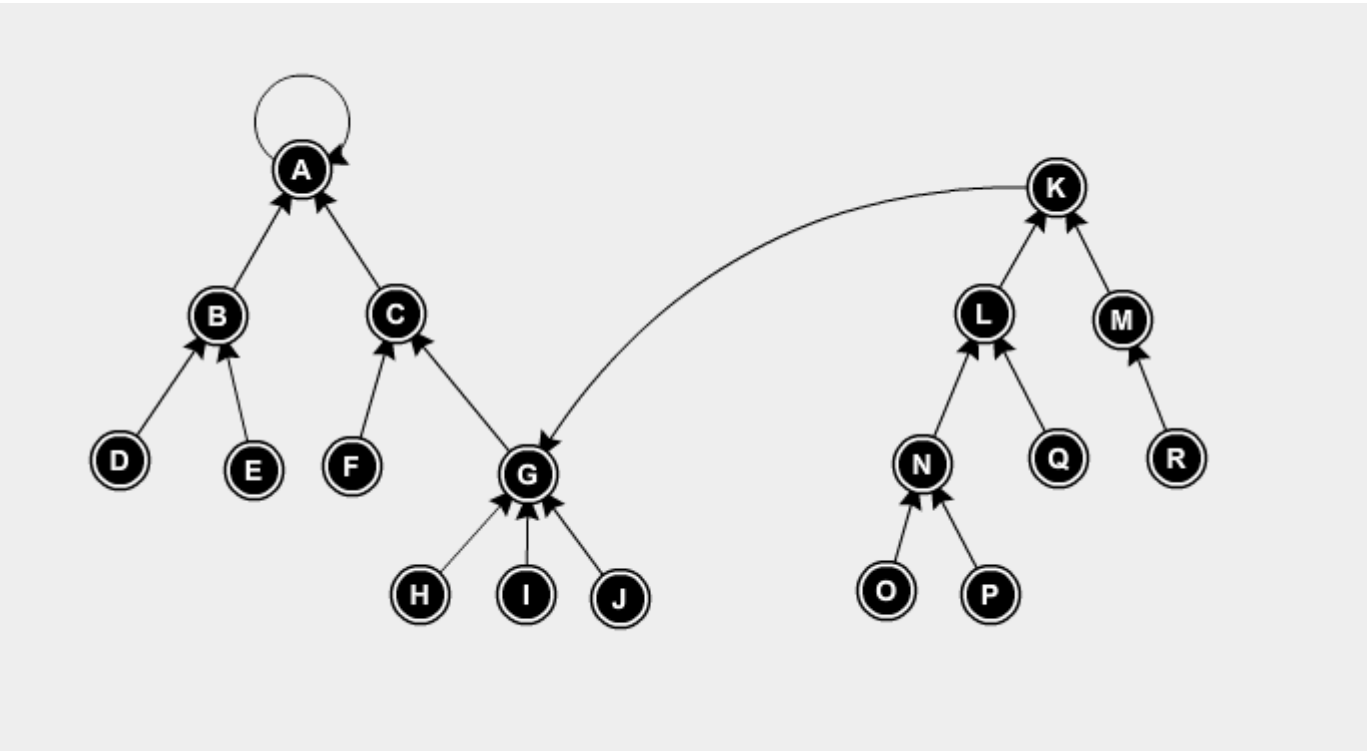
And finally we can do union operation by assigning the parent of the root of one element as the other element.

```
UNION(u, v)
        parent[FIND_SET(v)] = u
```

In the illustration below, the arrows points to the parent. All elements from A-J belong to the first set, they all have the same root element A. And all elements from K-R belong to the second set, they all have the same root element K.



If we want to do union operation on G and N, UNION(G,N) that will do the assignment parent[FIND_SET(N)] = G => parent[K] = G



After the union operation all elements A-R will have the same root element A, thus all will be in the same set.

Question

What is the worst case running time of the Kruskal Algorithm, using the set implementation above?

Bonus

We can improve the running time of our implementation by two technique called "**union by rank**" (always attach the shorter tree to the root of the higher tree) and "**path compression**" (after finding the root of a node, make it point to the rooot node directly). You can try to modify the code to implement these 2 techniques. And what is the worst case running time for Kruskal Algorithm now?