Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

It works as follows:

Create an ascending order of all the edges

1. create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
2. create a set *S* containing all the edges in the graph
3. while *S* is nonempty
   a. remove an edge with minimum weight from *S*
   b. if that edge connects two different trees, and connecting it does not form a loop, then add it to the forest, combining two trees into a single tree
   c. otherwise discard that edge

Program Using linked List for Kruskal algorithm. It uses the following structure as data strucrure for each edge. Let u, v be the starting and eding edge of an edge, whose weight is given by wt. The representation is as follows:

```
struct edges
{
        int u, v, wt;
        struct edges *next;
}

/* n no of vertices, f is a pointer to the first list of edges given in
asceding order*/

*/parent[i]=I for all I 0-n /* parent is made itself for all nodes
/* rank[i]=0 for all I 0 to n;/* rank is made 0 for all nodes

        edges* kruskal(edges *f, int n)

        edges *t=NULL, *e;

        int edgenos=0;


while(edgenos<n-1)&&(f!=NULL)
{
        e=f;
        f=f->next;
        p1=find(e->u);
        p2=find(e->v);
        if(p1!=p2)
        {
                unionRank(p1,p2);
                edgenos++;
                If(T==NULL)
                {
                        T=e;
                        r=T
                        r->next=NULL;
                }

                else
                {
                        r->next=e;
                        r=r->next;
                        r->next=NULL;
                }
        }
}
return(T);
}
```

The find(n) and rank are used together to find out the if adding a new edge will cause any cycle in the graph. Let array *parent[]* be used to store the parent of each node, i.e; if parent[i]=j.then the parent of the node 'i' is j. if k is the root then the parent of k is k itself. Rank[] is used to store the rank of each node.. meaning no. of subtrees it has.. which is 0 first…

```
Int find(int i)
{
        Int t;
        t=I;
        while(parent[t]!=t)
        {
                t=parent[t];
return t;
}
```

The find(i) is used to find the root of vertes i, ie, to which set 'I' belongs, it searches till root of I is I itself…

```
void unionRank(int i, int j)
{       if (rank[i] < rank[j])
        {
                parent[i] = j;    // j is made the root of the  vertex i
                ++rank[j];     // j is the root of the new tree; its rank
                                                        // is increased by 1
        }
        else
        {       parent[j] = i; // i is made the root of the  vertex j
                ++rank[i];     // i is the root of the new tree; its rank
                                                        // is increased by 1
        }
}
```

```
#include<iostream.h>
#define MAX 100

       struct edge_info
         {
              int u, v, weight;
         } edge[MAX];

                int tree[MAX][2], set[MAX];
                int n;
                int readedges();
                void makeset();
                int find(int);
                void join(int, int);
                void arrange_edges(int);
                int spanningtree(int);
                void display(int);
};

int readedges()
{
       int i, j, k, cost;
       k = 1;
       cout << "\nEnter the number of Vertices in the Graph : ";
       cin  >> n;
       cout << endl;
       for (i = 1; i <= n; i++)
              for (j = 1; j < i; j++)
              {
                     cout << "weight[" << i << "][" << j << "] : ";
                     cin  >> cost;
                     if (cost != 999)
                     {
                            edge[k].u = i;
                            edge[k].v = j;
                            edge[k++].weight = cost;
                     }
              }
       return (k - 1);
}

void makeset()
{
       int i;
       for (i = 1; i <= n; i++)
              set[i] = i;
}

int find(int vertex)
{
       return (set[vertex]);
}
```

```
void join(int v1, int v2)
{
        int i, j;
        if (v1 < v2)
                set[v2] = v1;
        else
                set[v1] = v2;
}

void arrange_edges(int k)
{
        int i, j;
        struct edge_info temp;
        for (i = 1; i < k; i++)
                for (j = 1; j <= k - i; j++)
                        if (edge[j].weight > edge[j + 1].weight)
                        {
                                temp = edge[j];
                                edge[j] = edge[j + 1];
                                edge[j + 1] = temp;
                        }
}


int spanningtree(int k)
{
        int i, t, sum;
        arrange_edges(k);
        t = 1;
        sum = 0;
        for (i=1;i<=k;i++)
        cout<<edge[i].u<<edge[i].v<<" "<<edge[i].weight<<endl;
        getch();
        for (i = 1; i <= k; i++)
                if (find (edge[i].u) != find (edge[i].v))
                {
                        tree[t][1] = edge[i].u;
                        tree[t][2] = edge[i].v;
                        sum += edge[i].weight;
                        join (edge[t].u, edge[t].v);
                        t++;
                }
        return sum;
}

void display(int cost)
{
        int  i;
        cout << "\nThe Edges of the Minimum Spanning Tree are\n\n";
        for (i = 1; i < n; i++)
                cout << tree[i][1] << " - " << tree[i][2] << endl;
```

```
        cout << "\nThe Cost of the Minimum Spanning Tree is : " <<
cost;
}

int main()
{
        int ecount, totalcost;
        kruskal k;
        ecount = k.readedges();
        k.makeset();
        totalcost = k.spanningtree(ecount);
        k.display(totalcost);
        return 0;
}
```