

```

1: // ::::::::::UNION-FIND IMPLEMENTATION OF KRUSHKAL ALGOITHM---USING POINTER AND PATH COMPRESSION
   TECHNIQUE::::::::::::
2:
3:
4: #include<stdio.h>
5: #include<stdlib.h>
6: struct uf* ds=NULL;
7: struct data* dt=NULL;
8: static int p=0;
9:
10: struct edge{
11:     int src,dest,w;
12: };
13:
14: struct data{
15:     int v;
16:     int e;
17:     struct edge* ed;
18: };
19: struct node{
20:     int dest;
21:     struct node* next;
22: };
23:
24: struct list{
25:     struct node* head;
26: };
27:
28: struct uf{
29:     int *size;
30:     struct list* root;
31:     struct list* nod;
32: };
33:
34: struct data* cdata(int v,int e){
35:     struct data* dt=(struct data*)malloc(sizeof(struct data));
36:     dt->v=v;
37:     dt->e=e;
38:     dt->ed=(struct edge*)malloc(e*sizeof(struct edge));
39:     return dt;
40: }
41:
42: void addedge(struct data* dt,int src,int dest,int w){
43:     dt->ed[p].dest=dest;
44:     dt->ed[p].src=src;
45:     dt->ed[p++].w=w;
46:
47: }
48:
49: struct node* cnode(int d){
50:     struct node* temp=(struct node*)malloc(sizeof(struct node));
51:     temp->dest=d;
52:     temp->next=temp;
53:     return(temp);
54: }
55:
56: struct uf* cuf(int v){
57:     struct uf* ds=(struct uf*)malloc(sizeof(struct uf));
58:     ds->size=(int*)malloc(v*sizeof(int));
59:     ds->root=(struct list*)malloc(v*sizeof(struct list));
60:     ds->nod=(struct list*)malloc(v*sizeof(struct list));
61:     for(int i=0;i<v;++i){
62:         ds->size[i]=1;
63:         ds->root[i].head=ds->nod[i].head=cnode(i);
64:     }
65:     return(ds);

```

```

66: }
67:
68: void swap(int* a, int* b)
69: {
70:     int t = *a;
71:     *a = *b;
72:     *b = t;
73: }
74:
75: int partition ( int low, int high)
76: {
77:     int pivot = dt->ed[high].w;
78:     int i = (low - 1);
79:
80:     for (int j = low; j <= high- 1; j++)
81:     {
82:         if (dt->ed[j].w <= pivot)
83:         {
84:             i++;
85:             swap(&dt->ed[i].w, &dt->ed[j].w);
86:             swap(&dt->ed[i].src, &dt->ed[j].src);
87:
88:             swap(&dt->ed[i].dest, &dt->ed[j].dest);
89:         }
90:     }
91:     swap(&dt->ed[i+1].w, &dt->ed[high].w);
92:     swap(&dt->ed[i+1].src, &dt->ed[high].src);
93:
94:     swap(&dt->ed[i+1].dest, &dt->ed[high].dest);
95:     return (i + 1);
96: }
97:
98: void quickSort(struct edge* ar, int low, int high)
99: {
100:     if (low < high)
101:     {
102:         int pi = partition( low, high);
103:         quickSort(ar, low, pi - 1);
104:         quickSort(ar, pi + 1, high);
105:     }
106: }
107: int find(int k){
108:     struct node* temp=ds->nod[k].head;
109:     while(temp->next!=temp){
110:         temp=temp->next;
111:     }
112:     ds->nod[k].head->next=temp;
113:     return(temp->dest);
114: }
115:
116:
117: void Union(struct uf* ds,int u,int v){
118:     int x=find(u);
119:     int y=find(v);
120:
121:     if(ds->size[x]<ds->size[y]){
122:         printf(" %d ----> %d\n",u+1,v+1);
123:         struct node* temp=ds->nod[x].head;
124:         temp->next=ds->nod[y].head;
125:         ds->root[x].head=NULL;
126:         ds->size[y]+=ds->size[x];
127:         ds->size[x]=0;
128:         for(int i=0;i<7;++i){
129:             if(ds->root[i].head==NULL)
130:                 printf("| size: %d | |root:: NULL|         |node:: %d|
|parent::%d|\n",ds->size[i],ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);

```

```

131:
132:         else
133:             printf("| size: %d | |root::%d | |node:: %d| |parent::%d| \n",ds-
>size[i],ds->root[i].head->dest+1,ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
134:
135:
136:     }
137:
138:     dt->v=v;
139:
140:     else{
141:         printf(" %d ----> %d\n",u+1,v+1);
142:         struct node* temp=ds->nod[y].head;
143:         temp->next=ds->nod[x].head;
144:         ds->root[y].head=NULL;
145:         ds->size[x]+=ds->size[y];
146:         ds->size[y]=0;
147:         for(int i=0;i<7;++i){
148:             if(ds->root[i].head==NULL)
149:                 printf("| size: %d | |root:: NULL| |node:: %d|
|parent::%d|\n",ds->size[i],ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
150:
151:         else
152:             printf("| size: %d | |root::%d | |node:: %d| |parent::%d| \n",ds-
>size[i],ds->root[i].head->dest+1,ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
153:
154:
155:     }
156: }
157: }
158: void krushkal(){
159:     int v=dt->v;
160:     int i=0;
161:     int e=0;
162:     quickSort(dt->ed,0,dt->e-1);
163:     while(e<v-1){
164:
165:         struct edge dy=dt->ed[i++];
166:         printf("\ncall %d and %d \n",dy.src+1,dy.dest+1);
167:         int x=find(dy.src);
168:         int y=find(dy.dest);
169:         if(x!=y){
170:
171:             Union(ds,dy.src,dy.dest);
172:             e++;
173:         }
174:
175:     }
176:
177: }
178: }
179: int main(){
180:     ds=cuf(7)
181:     dt=cdata(7,8);
182:     addedge(dt,2,3,70);
183:     addedge(dt,4,6,10);
184:     addedge(dt,5,6,5);
185:     addedge(dt,1,2,6);
186:     addedge(dt,1,4,20);
187:     addedge(dt,0,1,10);
188:     addedge(dt,4,5,10);
189:     addedge(dt,0,2,18);
190:
191:
192:     printf("\n::::::::::::::::::::INITIAL SETUP::::::::::::::::::::\n");
193:     for(int i=0;i<7;++i){

```

```

194:         printf("| size: %d | |root::%d| |node:: %d| |parent::%d| \n",ds->size[i],ds-
>root[i].head->dest+1,ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
195:     }
196:
197:     printf("::::::::::::CALLING KRIUSHKAL'S::::::::::::\n");
198:
199:     krushkal();
200:
201:
202:     int o=find(6);
203:     int y=find(4);
204:     printf("\n \n:::FINAL ANSWER:::\n");
205:     for(int i=0;i<7;++i){
206:         if(ds->root[i].head==NULL)
207:             printf("| size: %d | |root:: NULL| |node:: %d|
|parent::%d|\n",ds->size[i],ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
208:
209:         else
210:             printf("| size: %d | |root::%d| |node:: %d| |parent::%d| \n",ds-
>size[i],ds->root[i].head->dest+1,ds->nod[i].head->dest+1,ds->nod[i].head->next->dest+1);
211:
212:
213:     }
214:
215:     return 0;
216:
217: }
218:
219:
220:
221:
222:

```