```c
#include<stdio.h>
#include<stdlib.h>
#define size 40

struct node{
    int dest;
    struct node* next;
};
struct adj_list{
    struct node* head;

};
struct graph{
    int v;
    struct adj_list* array;
    int* visited;
    int* parent;
};

struct queue{
    int items[size];
    int front,rear;
};

struct node* cnode(int dest){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->dest=dest;
    temp->next=NULL;
    return(temp);
}

struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->array=(struct adj_list*)malloc(v*sizeof(struct adj_list));
    gr->visited=(int *)malloc(v*sizeof(int));
    gr->parent=(int*)malloc(v*sizeof(int));
    int i=0;
    for(i=0;i<v;++i){
        gr->array[i].head=NULL;
        gr->visited[i]=0;
        gr->parent[i]=-1;
    }
    return(gr);
}

void add_edge(struct graph* gr,int src,int dest){
    struct node* temp1=cnode(dest-1);
    temp1->next=gr->array[src-1].head;
    gr->array[src-1].head=temp1;
    struct node* temp2=cnode(src-1);
    temp2->next=gr->array[dest-1].head;
    gr->array[dest-1].head=temp2;
}
```

```
 55:
 56: struct queue* cqueue(){
 57:     struct queue* q=(struct queue*)malloc(sizeof(queue));
 58:     q->front=-1;
 59:     q->rear=-1;
 60:     return(q);
 61:
 62: }
 63:
 64: int isEmpty(struct queue* q) {
 65:     if(q->rear == -1)
 66:         return 1;
 67:      return 0;
 68: }
 69:
 70: void enqueue(struct queue* q, int value){
 71:     if(q->rear == size-1)
 72:         printf("\nQueue is Full!!");
 73:     else {
 74:         if(q->front == -1)
 75:             q->front = 0;
 76:         q->rear++;
 77:         q->items[q->rear] = value;
 78:     }
 79: }
 80:
 81: int dequeue(struct queue* q){
 82:     int item;
 83:     if(isEmpty(q)){
 84:         printf("Queue is empty");
 85:         item = -1;
 86:     }
 87:     else{
 88:         item = q->items[q->front];
 89:         q->front++;
 90:         if(q->front > q->rear){
 91:             q->front = q->rear = -1;
 92:         }
 93:     }
 94:     return item;
 95: }
 96:
 97: void bfs(struct graph* gr,int start){
 98:     struct queue*q=cqueue();
 99:     gr->visited[start]=1;
100:     enqueue(q,start);
101:     while(!isEmpty(q)){
102:         int currentVertex=dequeue(q);
103:         struct node* temp=gr->array[currentVertex].head;
104:         while(temp){
105:             int t=temp->dest;
106:             if(gr->visited[t]==0){
107:                 gr->parent[t]=currentVertex+1;
108:                 gr->visited[t]=1;
```

```
109:                 enqueue(q,t);
110:             }
111:             temp=temp->next;
112:         }
113:     }
114: }
115:
116: int main()
117: {    char ch;
118:     struct graph* gr = cgraph(10);
119:     add_edge(gr, 1,2);
120:     add_edge(gr, 1,3);
121:     add_edge(gr, 1,4);
122:     add_edge(gr, 2,1);
123:     add_edge(gr, 2,3);
124:     add_edge(gr, 3,1);
125:     add_edge(gr, 3,2);
126:     add_edge(gr, 4,1);
127:     add_edge(gr, 4,5);
128:     add_edge(gr, 4,8);
129:     add_edge(gr, 5,6);
130:     add_edge(gr, 5,7);
131:     add_edge(gr, 6,5);
132:     add_edge(gr, 6,8);
133:     add_edge(gr, 7,5);
134:     add_edge(gr, 7,6);
135:     add_edge(gr, 8,9);
136:     add_edge(gr, 9,10);
137:
138:     bfs(gr, 0);
139:     return 0;
140: }
```

```c
#include<stdio.h>
#include<stdlib.h>
static int m=0;
int pre[11];
int post[11];
struct node{
    int dest;
    struct node* next;
};
struct adj_list{
    struct node* head;

};
struct graph{
    int v;
    struct adj_list* array;
    int* visited;
    int* parent;
};
struct node* cnode(int dest){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->dest=dest;
    temp->next=NULL;
    return(temp);


}
struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->array=(struct adj_list*)malloc(v*sizeof(struct adj_list));
    gr->visited=(int *)malloc(v*sizeof(int));
    gr->parent=(int*)malloc(v*sizeof(int));
    int i=0;
    for(i=0;i<v;++i){
        gr->array[i].head=NULL;
        gr->visited[i]=0;
        gr->parent[i]=-1;
    }
    return(gr);
}
void add_edge(struct graph* gr,int src,int dest){
    struct node* temp1=cnode(dest);
    temp1->next=gr->array[src].head;
    gr->array[src].head=temp1;
    struct node* temp2=cnode(src);
    temp2->next=gr->array[dest].head;
    gr->array[dest].head=temp2;
}
void dfs(struct graph* gr,int s){
    struct node* temp=gr->array[s].head;
    gr->visited[s]=1;
    pre[s]=m++;
    while(temp!=NULL){
```

```
55:         int k=temp->dest;
56:         if(gr->visited[k]==0){
57:             dfs(gr,k);
58:         }
59:         temp=temp->next;
60:     }
61:     post[s]=m++;
62: }
63:
64: void cycle(){
65:
66: }
67: int main(){
68:
69: struct graph* gr = cgraph(11);
70:     add_edge(gr, 1,4);
71:     add_edge(gr, 1,3);
72:     add_edge(gr, 1,2);
73:    add_edge(gr, 2,1);
74:     add_edge(gr, 2,3);
75:     add_edge(gr, 3,1);
76:    add_edge(gr, 3,1);
77:     add_edge(gr, 4,1);
78:     add_edge(gr, 4,5);
79:     add_edge(gr, 4,8);
80:     add_edge(gr, 5,6);
81:     add_edge(gr, 5,7);
82:     add_edge(gr, 6,5);
83:     add_edge(gr, 6,8);
84:     add_edge(gr, 6,9);
85:    add_edge(gr, 7,5);
86:     add_edge(gr, 6,7);
87:     add_edge(gr, 8,9);
88:     add_edge(gr, 9,10);
89:     dfs(gr,4);
90:     for(int i=1;i<11;i++){
91:         printf(" %d --> [%d ,%d]\n",i,pre[i],post[i]);
92:     }
93:
94: return 0;
95: }
```

```c
#include<stdio.h>
#include<stdlib.h>
#define size 100

struct graph* gr=NULL;
struct queue* q=NULL;

struct queue{
    int items[size];
    int front,rear;
};

struct node{
    int dest;
    struct node* next;
};

struct adj_list{
    struct node* head;

};

struct graph{
    int v;
    struct adj_list* array;
};

struct node* cnode(int dest){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->dest=dest;
    temp->next=NULL;
    return(temp);
}

struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->array=(struct adj_list*)malloc(v*sizeof(struct adj_list));
    int i=0;
    for(i=0;i<v;++i){
        gr->array[i].head=NULL;
    }
    return(gr);
}
void add_edge(struct graph* gr,int src,int dest){
    struct node* temp=cnode(dest);
    temp->next=gr->array[src].head;
    gr->array[src].head=temp;
    temp=NULL;
    free(temp);
}

struct queue* cqueue(){
    struct queue* q=(struct queue*)malloc(sizeof(queue));
```

```
 55:     q->front=-1;
 56:     q->rear=-1;
 57:     return(q);
 58: 
 59: }
 60: 
 61: int isEmpty(struct queue* q) {
 62:     if(q->rear == -1)
 63:         return 1;
 64:     else
 65:         return 0;
 66: }
 67: 
 68: void enqueue(struct queue* q, int value){
 69:     if(q->rear == size-1)
 70:         printf("\nQueue is Full!!");
 71:     else {
 72:         if(q->front == -1)
 73:             q->front = 0;
 74:         q->rear++;
 75:         q->items[q->rear] = value;
 76:     }
 77: }
 78: 
 79: int dequeue(struct queue* q){
 80:     int item;
 81:     if(isEmpty(q)){
 82:         printf("Queue is empty");
 83:         item = -1;
 84:     }
 85:     else{
 86:         item = q->items[q->front];
 87:         q->front++;
 88:         if(q->front > q->rear){
 89:             q->front = q->rear = -1;
 90:         }
 91:     }
 92:     return item;
 93: }
 94: 
 95: void t node* cnode(int dest){
 96:     int indegree[gr->v];
 97: 
 98:      for(int i=0;i<gr->v;++i)
 99:         indegree[i]=0;
100: 
101:     for(int i=0;i<gr->v;++i){
102:         struct node* temp=gr->array[i].head;
103:         while(temp!=NULL){
104:             int k=temp->dest;
105:             indegree[k]++;
106:             temp=temp->next;
107:         }
108:     }
```

```
109:     for(int v=1;v<gr->v;v++){
110:         if(indegree[v]==0){
111:             indegree[v]=-1;
112:             enqueue(q,v);
113: 
114:         while(isEmpty(q)==false){
115:             int u=dequeue(q);
116:             indegree[u]=-1;
117:             printf("%d --->",u);
118:             struct node* temp=gr->array[u].head;
119:             while(temp!=NULL){
120:                 int k=temp->dest;
121:                 indegree[k]=indegree[k]-1;
122:                 if(indegree[k]==0)
123:                     enqueue(q,k);
124: 
125:                 temp=temp->next;
126:                 }
127:             }
128:         }
129:     }
130: }
131: 
132: int main(){
133:      gr = cgraph(9);
134:     add_edge(gr, 1,5);
135:     add_edge(gr, 1,4);
136:     add_edge(gr, 1,3);
137:    add_edge(gr, 2,3);
138:     add_edge(gr, 2,8);
139:     add_edge(gr, 3,6);
140:    add_edge(gr, 4,6);
141:     add_edge(gr, 4,8);
142:     add_edge(gr,5,8);
143:     add_edge(gr, 6,7);
144:     add_edge(gr, 7,8);
145:     printf("Topological order of given graph is:\n");
146:     q=cqueue();
147:     topl_order();
148: return 0;
149: }
```

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

struct graph* gr=NULL;
struct mh* mhp=NULL;
struct node{
    int data;
    int w;
    struct node* next;
};

struct list{
    struct node* head;
};

struct graph{
    int v;
    struct list* arr;
};

struct node* cnode(int data,int w){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->data=data;
    temp->w=w;
    temp->next=NULL;

     return temp;
}

struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->arr=(struct list*)malloc(v*sizeof(struct list));
    for(int i=0;i<v;++i)
     gr->arr[i].head=NULL;

     return gr;

}
void addedge(int u,int v,int w){
    struct node* temp=cnode(v,w);
    temp->next=gr->arr[u].head;
    gr->arr[u].head=temp;

    temp=cnode(u,w);
    temp->next=gr->arr[v].head;
    gr->arr[v].head=temp;

    temp=NULL;
    free(temp);

}

struct mhn{
```

```
 55:     int v,dist;
 56:
 57: };
 58:
 59: struct mh{
 60:     int size,capacity;
 61:     int* pos;
 62:     struct mhn** arr;
 63:
 64: };
 65:
 66: struct mhn* cmhn(int v,int dist){
 67:     struct mhn* temp=(struct mhn*)malloc(sizeof(struct mhn));
 68:     temp->dist=dist;
 69:     temp->v=v;
 70:     return temp;
 71:
 72: }
 73:
 74: struct mh* cmh(int c){
 75:     struct mh* temp=(struct mh*)malloc(sizeof(struct mh));
 76:     temp->size=0;
 77:     temp->capacity=c;
 78:     temp->pos=(int*)malloc(c*sizeof(int));
 79:     temp->arr=(struct mhn**)malloc(c*sizeof(struct mhn*));
 80:       return temp;
 81: }
 82:
 83: void swap(struct mhn**a,struct mhn**b){
 84:     struct mhn* temp=*a;
 85:     *a=*b;
 86:     *b=temp;
 87: }
 88:
 89: void heapify(int pos){
 90:     int s,l,r;
 91:     s=pos;
 92:     l=2*pos+1;
 93:     r=2*pos+2;
 94:
 95:  int data;hp->size&&mhp->arr[l]->dist<mhp->arr[
 96:       s=l;
 97:
 98:     if(r<mhp->size&&mhp->arr[r]->dist<mhp->arr[s]->dist)
 99:        s=r;
100:
101:     if(s!=pos){
102:     struct mhn* sm=mhp->arr[s];
103:     struct mhn* ipos=mhp->arr[pos];
104:
105:     mhp->pos[ipos->v]=s;
106:     mhp->pos[sm->v]=pos;
107:
108:     swap(&mhp->arr[s],&mhp->arr[pos]);
```

```
109:     heapify(s);
110: 
111:      }
112: }
113: 
114: bool isempty(){
115:     return mhp->size==0;
116: }
117: 
118: bool ispresent(int v){
119:     if(mhp->pos[v]<mhp->size)
120:      return true;
121:    return false;
122: }
123: 
124: struct mhn* extract_min(){
125:  int data;
126:      return NULL;
127: 
128:     struct mhn* root=mhp->arr[0];
129:     struct mhn* lnode=mhp->arr[mhp->size-1];
130:     mhp->pos[lnode->v]=0;
131:     mhp->pos[root->v]=mhp->size-1;
132:     mhp->size--;
133: 
134:     mhp->arr[0]=lnode;
135: 
136:     heapify(0);
137:     return root;
138: 
139: }
140: 
141: 
142: void modify(int v,int dist){
143: 
144:     int i=mhp->pos[v];
145:     mhp->arr[i]->dist=dist;
146:     while(i&&(mhp->arr[i]->dist<mhp->arr[(i-1)/2]->dist)){
147: 
148:         mhp->pos[mhp->arr[i]->v]=(i-1)/2;
149:         mhp->pos[mhp->arr[(i-1)/2]->v]=i;
150:         swap(&mhp->arr[i],&mhp->arr[(i-1)/2]);
151: 
152:         i=(i-1)/2;
153: 
154:     }
155: 
156: }
157: void printArr(int dist[], int n)
158: {
159:     printf("Vertex   Distance from Source\n");
160:     for (int i = 0; i < n; ++i)
161:         printf("%d \t\t %d\n", i, dist[i]);
162: }
```

```
163:
164: void dijstra(int src){
165:     int dist[gr->v];
166:     mhp=cmh(gr->v);
167:     for(int i=0;i<gr->v;++i){
168:         dist[i]=INT_MAX;
169:         mhp->arr[i]=cmhn(i,dist[i]);
170:         mhp->pos[i]=i;
171:     }
172:
173:     mhp->arr[src]=cmhn(src,dist[src]);
174:     mhp->pos[src]=src;
175:     dist[src]=0;
176:     modify(src,dist[src]);
177:
178:     mhp->size=gr->v;
179:
180:      /* dist[src]=0;
181:     mhp->arr[src]=cmhn(0,dist[0]);
182:     mhp->pos[0]=0;*/
183:
184:     mhp->size=gr->v;
185:     while(!isempty()){
186:         struct mhn* temp=extract_min();
187:         int u=temp->v;
188:         struct node* temp1=gr->arr[u].head;
189:         while(temp1){
190:
191:             int v=temp1->data;
192:             if(ispresent(v)&&dist[v]>dist[u]+temp1->w)
193:             {
194:                 dist[v]=dist[u]+temp1->w;
195:                 modify(v,dist[v]);
196:
197:             }
198:
199:             temp1=temp1->next;
200:         }
201:
202:     }
203:     printArr(dist,gr->v);
204: }
205:
206: int main(){
207:     int V = 9;
208:     gr= cgraph(V);
209:     addedge( 0, 1, 4);
210:     addedge( 0, 7, 8);
211:     addedge( 1, 2, 8);
212:     addedge( 1, 7, 11);
213:     addedge( 2, 3, 7);
214:     addedge( 2, 8, 2);
215:     addedge( 2, 5, 4);
216:     addedge( 3, 4, 9);
```

```
217:     addedge( 3, 5, 14);
218:     addedge( 4, 5, 10);
219:     addedge( 5, 6, 2);
220:     addedge( 6, 7, 1);
221:     addedge(6, 8, 6);
222:     addedge(7, 8, 7);
223:
224:     dijstra(0);
225: }
226:
```

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

int min(int a,int b){
    if(a<b)return a;

        return b;

}
struct graph* gr=NULL;
struct node{
    int dest;
    int weight;
    struct node* next;
};

struct list{
    struct node* head;

};

struct graph{
    int v;

    struct list* array;
};

struct node* cnode(int dest,int w){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->dest=dest;
    temp->weight=w;
    temp->next=NULL;

    return(temp);
}

struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->array=(struct list*)malloc(v*sizeof(struct list));
    for(int i=0;i<v;++i){
        gr->array[i].head=NULL;
    }
    return(gr);
}
void addedge(struct graph* gr,int src,int weight,int dest){
    struct node* temp=cnode(dest,weight);
    temp->next=gr->array[src].head;
    gr->array[src].head=temp;
    printf("ADDED EDGE:: %d---(%d)--->%d\n",src,weight,dest);
    temp=NULL;
    delete(temp);
}
```

```
 55:
 56: void bellman(int src){
 57:     int dist[gr->v];
 58:     for(int i=0;i<gr->v;++i)
 59:         dist[i]=INT_MAX;
 60:     dist[src]=0;
 61:     for(int j=0;j<gr->v;++j){
 62:         for(int i=0;i<gr->v;++i){
 63:               if(dist[i]!=INT_MAX){
 64:                 struct node* temp=gr->array[i].head;
 65:                 while(temp!=NULL){
 66:                 int j=temp->dest;
 67:                 int w=temp->weight;
 68:                 dist[j]=min(dist[j],dist[i]+w);
 69:
 70:                 temp=temp->next;
 71:                }
 72:            }
 73:
 74:        }
 75:    }
 76:    printf("::::CALCULATING SHORTEST PATH IN NEGATIVE WEIGHT EDGE
    GRAPH::::\n");
 77:    for(int i=0;i<gr->v;++i){
 78:     if(dist[i]==INT_MAX)
 79:         printf("\n|%d|---->---------|INFINITE|--->------------|%d|",src+1,
    i+1);
 80:     else
 81:         printf("\n|%d|---->---|%d|--->----|%d|",src+1,dist[i],i+1);
 82:     }
 83:     printf("\n-------------------------------------------------\n");
 84: }
 85: int main(){
 86:     gr=cgraph(8);
 87:     addedge(gr,0,10,1);
 88:     addedge(gr,0,8,7);
 89:     addedge(gr,1,2,5);
 90:     addedge(gr,2,1,1);
 91:     addedge(gr,2,1,3);
 92:     addedge(gr,3,3,4);
 93:     addedge(gr,4,-1,5);
 94:     addedge(gr,5,-2,2);
 95:     addedge(gr,6,-1,5);
 96:     addedge(gr,6,-4,1);
 97:     addedge(gr,7,1,6);
 98:     bellman(1);
 99:
100: }
```

```c
#include<stdio.h>
#define V 8
#define INF 999999

int min(int a,int b){
    if(a<b)return a;
    return b;
}
void printSolution(int dist[][V])
{
printf("\n:::::::::::::::::::::::::: ALL PAIR SHORTEST PATH ::::::::::::::::::::::::\n");

printf("_____________________________________________________________________________");

printf("\n___|_____1|_____2|_____3|_____4|_____5|_____6|_____7|_____8|\n");

    for (int i = 0; i < V; i++)
    {
    printf(" %d |  ",i+1);
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s|", "INF");
            else
                printf ("%7d|", dist[i][j]);
        }

        printf("\n");
    }

printf("_____________________________________________________________________________\n");
}
]
void floydWarshal(int graph[][V]){
    int dist[V][V];
    for(int i=0;i<V;++i){
      for(int j=0;j<V;++j){
        dist[i][j]=graph[i][j];
      }
    }

    for(int k=0;k<V;++k){
        for(int i=0;i<V;++i){
            for(int j=0;j<V;++j){
                dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]);
            }
        }
    }

    printSolution(dist);
}

int main(){
```

```
51:     int graph[V][V]={{0,10,INF,INF,INF,INF,INF,8},
52:                      {INF,0,INF,INF,INF,2,INF,INF},
53:                      {INF,1,INF,1,INF,INF,INF,INF},
54:                      {INF,INF,INF,0,3,INF,INF,INF},
55:                      {INF,INF,INF,INF,0,-1,INF,INF},
56:                      {INF,INF,-2,INF,INF,0,INF,INF},
57:                      {INF,-4,INF,INF,INF,-1,0,INF},
58:                      {INF,INF,INF,INF,INF,INF,1,0}};
59:
60:   floydWarshal(graph);
61:   return 0;
62: }
63:
```

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

struct graph* gr=NULL;
struct mh* mhp=NULL;
struct node{
    int data;
    int w;
    struct node* next;
};

struct list{
    struct node* head;
};

struct graph{
    int v;
    struct list* arr;
};

struct node* cnode(int data,int w){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->data=data;
    temp->w=w;
    temp->next=NULL;

     return temp;
}

struct graph* cgraph(int v){
    struct graph* gr=(struct graph*)malloc(sizeof(struct graph));
    gr->v=v;
    gr->arr=(struct list*)malloc(v*sizeof(struct list));
    for(int i=0;i<v;++i)
     gr->arr[i].head=NULL;

     return gr;

}
void addedge(int u,int v,int w){
    struct node* temp=cnode(v,w);
    temp->next=gr->arr[u].head;
    gr->arr[u].head=temp;

    temp=cnode(u,w);
    temp->next=gr->arr[v].head;
    gr->arr[v].head=temp;

    temp=NULL;
    free(temp);

}

struct mhn{
```

```
 55:     int v,dist;
 56:
 57: };
 58:
 59: struct mh{
 60:     int size,capacity;
 61:     int* pos;
 62:     struct mhn** arr;
 63:
 64: };
 65:
 66: struct mhn* cmhn(int v,int dist){
 67:     struct mhn* temp=(struct mhn*)malloc(sizeof(struct mhn));
 68:     temp->dist=dist;
 69:     temp->v=v;
 70:     return temp;
 71:
 72: }
 73:
 74: struct mh* cmh(int c){
 75:     struct mh* temp=(struct mh*)malloc(sizeof(struct mh));
 76:     temp->size=0;
 77:     temp->capacity=c;
 78:     temp->pos=(int*)malloc(c*sizeof(int));
 79:     temp->arr=(struct mhn**)malloc(c*sizeof(struct mhn*));
 80:       return temp;
 81: }
 82:
 83: void swap(struct mhn**a,struct mhn**b){
 84:     struct mhn* temp=*a;
 85:     *a=*b;
 86:     *b=temp;
 87: }
 88:
 89: void heapify(int pos){
 90:     int s,l,r;
 91:     s=pos;
 92:     l=2*pos+1;
 93:     r=2*pos+2;
 94:
 95:     if(l<mhp->size&&mhp->arr[l]->dist<mhp->arr[s]->dist)
 96:       s=l;
 97:
 98:     if(r<mhp->size&&mhp->arr[r]->dist<mhp->arr[s]->dist)
 99:        s=r;
100:
101: if(s!=pos){
102:     struct mhn* sm=mhp->arr[s];
103:     struct mhn* ipos=mhp->arr[pos];
104:
105:     mhp->pos[ipos->v]=s;
106:     mhp->pos[sm->v]=pos;
107:
108:     swap(&mhp->arr[s],&mhp->arr[pos]);
```

```
109:     heapify(s);
110: 
111: }
112: }
113: 
114: bool isempty(){
115:     return mhp->size==0;
116: 
117: }
118: 
119: bool ispresent(int v){
120:     if(mhp->pos[v]<mhp->size)
121:      return true;
122:    return false;
123: }
124: 
125: struct mhn* extract_min(){
126:     if(isempty())
127:      return NULL;
128: 
129:     struct mhn* root=mhp->arr[0];
130:     struct mhn* lnode=mhp->arr[mhp->size-1];
131:     mhp->pos[lnode->v]=0;
132:     mhp->pos[root->v]=mhp->size-1;
133:     mhp->size--;
134: 
135:     mhp->arr[0]=lnode;
136: 
137:     heapify(0);
138:     return root;
139: 
140: }
141: 
142: 
143: void modify(int v,int dist){
144: 
145:     int i=mhp->pos[v];
146:     mhp->arr[i]->dist=dist;
147:     while(i&&(mhp->arr[i]->dist<mhp->arr[(i-1)/2]->dist)){
148: 
149:         mhp->pos[mhp->arr[i]->v]=(i-1)/2;
150:         mhp->pos[mhp->arr[(i-1)/2]->v]=i;
151:         swap(&mhp->arr[i],&mhp->arr[(i-1)/2]);
152: 
153:         i=(i-1)/2;
154: 
155:     }
156: 
157: }
158: 
159: 
160: void printarr(int arr[], int n)
161: {
162:     for (int i = 1; i < n; ++i)
```

```
163:         printf("%d - %d\n", arr[i], i);
164: }
165:
166: void prims(int src){
167:
168:     int dist[gr->v];
169:     int parent[gr->v];
170:     mhp=cmh(gr->v);
171:     for(int i=0;i<gr->v;++i){
172:         parent[i]=-1;
173:         dist[i]=INT_MAX;
174:         mhp->arr[i]=cmhn(i,dist[i]);
175:         mhp->pos[i]=i;
176:     }
177:     dist[src]=0;
178:     mhp->arr[src]=cmhn(0,dist[0]);
179:     mhp->pos[0]=0;
180:
181:     mhp->size=gr->v;
182:     while(!isempty()){
183:         struct mhn* temp=extract_min();
184:         int u=temp->v;
185:         struct node* temp1=gr->arr[u].head;
186:         while(temp1){
187:              int v=temp1->data;
188:              if(ispresent(v)&&dist[v]>temp1->w)
189:              {
190:
191:                 dist[v]=temp1->w;
192:                 parent[v]=u;
193:                 modify(v,dist[v]);
194:              }
195:              temp1=temp1->next;
196:
197:         }
198:     }
199:
200:     printarr(parent,gr->v);
201: }
202: int main(){
203:     int V = 9;
204:     gr= cgraph(V);
205:     addedge( 0, 1, 4);
206:     addedge( 0, 7, 8);
207:     addedge( 1, 2, 8);
208:     addedge( 1, 7, 11);
209:     addedge( 2, 3, 7);
210:     addedge( 2, 8, 2);
211:     addedge( 2, 5, 4);
212:     addedge( 3, 4, 9);
213:     addedge( 3, 5, 14);
214:     addedge( 4, 5, 10);
215:     addedge( 5, 6, 2);
216:     addedge( 6, 7, 1);
```

```
217:     addedge(6, 8, 6);
218:     addedge(7, 8, 7);
219:
220:      prims(0);
221: }
222:
```

```
// :::::::UNION-FIND IMPLEMENTATION OF KRUSHKAL ALGOITHM---USING POINTER AND PATH COMPRESSION TECHNIQUE::::::::::::


#include<stdio.h>
#include<stdlib.h>
struct uf* ds=NULL;
struct data* dt=NULL;
static int p=0;

struct edge{
    int src,dest,w;
};

struct data{
    int v;
    int e;
    struct edge* ed;
};
struct node{
    int dest;
    struct node* next;
};

struct list{
    struct node* head;
};

struct uf{
    int *size;
    struct list* root;
    struct list* nod;
};

struct data* cdata(int v,int e){
    struct data* dt=(struct data*)malloc(sizeof(struct data));
    dt->v=v;
    dt->e=e;
    dt->ed=(struct edge*)malloc(e*sizeof(struct edge));
    return dt;
}

void addedge(struct data* dt,int src,int dest,int w){
    dt->ed[p].dest=dest;
     dt->ed[p].src=src;
      dt->ed[p++].w=w;

}

struct node* cnode(int d){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->dest=d;
    temp->next=temp;
    return(temp);
}

struct uf* cuf(int v){
    struct uf* ds=(struct uf*)malloc(sizeof(struct uf));
    ds->size=(int*)malloc(v*sizeof(int));
    ds->root=(struct list*)malloc(v*sizeof(struct list));
    ds->nod=(struct list*)malloc(v*sizeof(struct list));
    for(int i=0;i<v;++i){
        ds->size[i]=1;
        ds->root[i].head=ds->nod[i].head=cnode(i);
    }
    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:
```

```c
// Interval Scheduling

#include<stdio.h>
#include<stdlib.h>
static int k=0;
struct data* dt=NULL;
struct is{
    int start;
    int finish;
};

struct data{
    int v;
    struct is* array;
};

struct data* create_data(int v){

    struct data* temp=(struct data*)malloc(sizeof(struct data));
    temp->v=v;
    temp->array=(struct is*)malloc(v*sizeof(struct is));
    return temp;
}

void add_data(int start,int finish){
     dt->array[k].start=start;
     dt->array[k++].finish=finish;
}
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int partition ( int low, int high)
{
    int pivot = dt->array[high].finish;
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)
    {
        if (dt->array[j].finish <= pivot)
        {
            i++;
            swap(&dt->array[i].finish,&dt->array[j].finish);
             swap(&dt->array[i].start,&dt->array[j].start);
        }
    }
    swap(&dt->array[i+1].finish, &dt->array[high].finish);
    swap(&dt->array[i+1].start,&dt->array[high].start);

    return (i + 1);
}
```

```
55:
56: void quickSort(struct is* arr,int low, int high)
57: {
58:     if (low < high)
59:     {
60:         int pi = partition( low, high);
61:         quickSort(arr, low, pi - 1);
62:         quickSort(arr, pi + 1, high);
63:     }
64: }
65:
66: void interval_scheduling(){
67:
68:     quickSort(dt->array,0,dt->v);
69:     int i=0;
70:     printf("%d--%d",dt->array[i].start,dt->array[i].finish);
71:     for(int j=1;j<dt->v;++j){
72:         if(dt->array[j].start>=dt->array[i].finish)
73:         {  printf("\n%d--%d",dt->array[j].start,dt->array[j].finish);
74:           i=j;
75:         }
76:     }
77:
78: }
79:
80:
81:
82: int main(){
83:     dt=create_data(6);
84:     add_data(5,9);
85:     add_data(1,2);
86:     add_data(3,4);
87:     add_data(0,6);
88:     add_data(5,7);
89:     add_data(8,9);
90:     interval_scheduling();
91:     return 0;
92:
93: }
94:
95:
96:
```

```c
// Inversions

#include<stdio.h>
int merge(int arr[],int temp[],int left,int mid,int right){
    int i, j, k,count=0;

    i = left;
    j = mid;
    k = left;
    while ((i<=mid-1)&&(j<=right)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        }
        else {
            temp[k++] = arr[j++];
            count =count + (mid - i);
        }
    }
    while (i <= mid - 1)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];

    for(int i=left;i<=right;++i)
          arr[i]=temp[i];
  return count;
}
int merge_count(int arr[],int temp[],int left,int right){
    int mid;
    int count=0;
    if(right>left){
        mid=(left+right)/2;
        count=merge_count(arr,temp,left,mid);
        count+=merge_count(arr,temp,mid+1,right);
        count+=merge(arr,temp,left,mid+1,right);
    }
    return(count);
}



int main(){
    int arr[5]={2,4,3,1,5};
    int temp[5];
    int p=merge_count(arr,temp,0,4);
    printf("Ans: %d",p);
}
```

```c
#include<stdio.h>

int size(char A[]){
    int c=0,i=0;
    while(A[i]!='\0'){
        i=i+1;
        c=c+1;
    }
    return c;

}
int min(int a,int b,int c){
 if(a<b&&a<c)
   return a;
 if(b<c&&b<a)
  return b;
  return c;
}
void print(char A[]){
    int i=0;
    printf("      ");
    while(A[i]!='\0'){
        printf("|_%c_",A[i]);
        i=i+1;
    }
    printf("|_._|");
}
int ED(char A[],char B[]){
    int m=size(A);
    int n=size(B);
    int ED[m+1][n+1];
    for(int r=0;r<=m;++r)
      ED[r][n]=m-r;
    for(int c=0;c<=m;++c)
      ED[m][c]=n-c;
    for(int c=m-1;c>=0;--c){
        for(int r=n-1;r>=0;--r){
            if(A[r]==B[c])
              ED[r][c]=ED[r+1][c+1];
            else
              ED[r][c]=1+min(ED[r+1][c+1],ED[r+1][c],ED[r][c+1]);

        }

    }
     print(B);
    printf("\n\n");
    for(int i=0;i<=m;++i)
      {

        printf("|_%c_| ",A[i]);
      for(int j=0;j<=n;++j)
       printf("|_%d_",ED[i][j]);
    printf("|");
```

```
55:      printf("\n");
56: }
57:     int i=0,j=0;
58:     printf("\nTransforming %s to %s: ",B,A);
59:     while(i<=m&&j<=n){
60:         if(A[i]==B[j]){
61:             i++;j++;
62:         }
63:         else{
64:             int k=min(ED[i+1][j+1],ED[i+1][j],ED[i][j+1]);
65:                 if(k==ED[i+1][j])
66:                 {
67:                   printf("\n Deleting: %c at position: %d in %s--->%s ",A[i],
    i+1,A,A+i+1);
68:                   i++;
69:                 }
70:                 else
71:                 {
72:                   printf("\n Inserting: %c at position: %d ",B[j],j+1);
73:                   j++;
74:         }
75:         }
76:     }
77:      return ED[0][0];
78:
79: }
80: int main(){
81:     char B[]="secret";
82:     char A[]="bisect";
83:     int p=ED(A,B);
84:     printf("\n \nSo Number of minimum edit that are require: %d",p);
85: }
```

```c
//  Grid Path

#include<stdio.h>

int numberOfPaths(int m, int n)
{
    int count[++m][++n];
    for (int i = 0; i < m; i++)
        count[i][0] = 1;
    for (int j = 0; j < n; j++)
        count[0][j] = 1;
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
         if((i==2||i==4)&&j==4)
            // when there is a hole in the path
            count[i][j]=0;
          else
            //no hole in that path
            count[i][j] = count[i-1][j] + count[i][j-1];
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<n;++j){
            printf("| %d     ",count[i][j]);
        }
        printf("\n");
    }
    return count[m-1][n-1];
}

int main()
{
    printf("%d", numberOfPaths(5,10));

    return 0;
}
```

```c
// Longest Common Subword

#include<stdio.h>

int size(char A[]){
    int c=0,i=0;
    while(A[i]!='\0'){
        i=i+1;
        c=c+1;
    }
    return c+1;

    }

int LCW(char A[],char B[]){
    int LCW[size(A)][size(B)];
    for(int r=0;r<size(A);++r)
      LCW[r][size(B)-1]=0;
    for(int c=0;c<size(B);++c)
      LCW[size(A)-1][c]=0;

    int maxval=0;

    for(int c=size(B)-2;c>=0;--c){
        for(int r=size(A)-2;r>=0;--r){
            if(A[r]==B[c])
              LCW[r][c]=1+LCW[r+1][c+1];
            else
               LCW[r][c]=0;
            if(LCW[r][c]>maxval)
              maxval=LCW[r][c];
        }
    }
    for(int i=0;i<size(A);++i)
      {
      for(int j=0;j<size(B);++j)
       printf("|%d ",LCW[j][i]);
      printf("\n");
}
    return maxval;

}


int main(){
    char A[]="secret";
    char B[]="bisect";
    int p=LCW(A,B);
    printf("\n ANS: %d",p);
}
```

```c
//Longest common subsequence

#include<stdio.h>

int size(char A[]){
    int c=0,i=0;
    while(A[i]!='\0'){
        i=i+1;
        c=c+1;
    }
    return c+1;

    }
void print(char A[]){
    int i=0;
    printf("      ");
    while(A[i]!='\0'){
        printf("|_%c_",A[i]);
        i=i+1;
    }
    printf("|_._|");
}
int max(int a,int b){
    if(a>b)return a;
    return b;
}
int LCS(char A[],char B[]){
    int LCS[size(A)][size(B)];
    for(int r=0;r<size(B);++r)
      LCS[r][size(B)-1]=0;
    for(int c=0;c<size(A);++c)
      LCS[size(A)-1][c]=0;



    for(int c=size(B)-2;c>=0;--c){
        for(int r=size(A)-2;r>=0;--r){
            if(A[r]==B[c])
              LCS[r][c]=1+LCS[r+1][c+1];
            else
               LCS[r][c]=max(LCS[r+1][c],LCS[r][c+1]);

        }

    }  print(B);
    printf("\n\n");
    for(int i=0;i<size(A);++i)
      {

        printf("|_%c_| ",A[i]);
      for(int j=0;j<size(B);++j)
       printf("|_%d_",LCS[i][j]);
    printf("|");
     printf("\n");
```

```
}
    int i=0,j=0;
    printf("\nLongest common subsequence is: ");
    while(i<=size(A)&&j<=size(B)){
        if(A[i]==B[j]){
            printf("%c",A[i]);
            i++;j++;
        }
        else{
            int k=max(LCS[i+1][j],LCS[i][j+1]);
                if(k==LCS[i+1][j])
                  i++;
                else
                  j++;

        }
    }
    return LCS[0][0];

}

int main(){
    char A[]="sharma";
    char B[]="sourav";

    int p=LCS(A,B);
    printf("\nAnd it's length is %d",p);
}
```

```c
#include<stdio.h>

int max(int a, int b) { return (a > b)? a : b; }

int knapSack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n+1][W+1];
    for (i = 0; i <= n; i++){
       for (w = 0; w <= W; w++){
        if (i==0 || w==0)
            K[i][w] = 0;
        else if (wt[i-1] <= w)
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
        else
                K[i][w] = K[i-1][w];
        }
  }
return K[n][W];
}

int main()
{   printf(":::::::::::::::::: KNAPSACK PROBLEM::::::::::::::::::\n");
    int val[] = {60, 10, 12,28,11,30,15};
    int wt[] = {10, 20, 30,34,28,87,10};
        printf("\t _____________________\n");
    printf("\t|S.No.| Value  | Weight|\n");
    printf("\t|_____|________|_______|\n");
    for(int i=0;i<7;++i){

    printf("\t|%d    | %d     |  %d   |\n",i+1,val[i],wt[i]);
    printf("\t|_____|________|_______|\n");
}

    int W = 56;
    printf("\n\tALLOWED WEIGHT TO PICK : %d\n",W);

    int n = sizeof(val)/sizeof(val[0]);
    printf("\n THE MAXIMUM VALUE THAT WE CAN ACHIEVE IS :: %d", knapSack(W, wt, val, n));
    return 0;
}

```

```cpp
#include<iostream>
using namespace std;

int n=4;
int dist[10][10] = {
        {0,20,42,25},
        {20,0,30,34},
        {42,30,0,10},
        {25,34,10,0}
};

void print(){
    printf("GIVEN MATRIX IS: \n\n");
    for(int i=0;i<4;++i)
    {
        for(int j=0;j<4;++j)
        {
            printf(" %d ",dist[i][j]);
        }
        printf("\n");
    }
}

int VISITED_ALL = (1<<n) -1;
int dp[16][4];

int  tsp(int mask,int pos){
    if(mask==VISITED_ALL)
        return dist[pos][0];

    if(dp[mask][pos]!=-1)
       return dp[mask][pos];

    int ans = INT_MAX;

    for(int city=0;city<n;city++){
        if((mask&(1<<city))==0){
            int newAns = dist[pos][city] + tsp( mask|(1<<city), city);
            ans = min(ans, newAns);
        }
    }
    return dp[mask][pos] = ans;
}

int main(){
    for(int i=0;i<(1<<n);i++){
        for(int j=0;j<n;j++){
            dp[i][j] = -1;
        }
    }
    printf("::::::::::TRAVELLING SALESMAN PROBLEM ::::::::::::::::\n\n");
    print();
    cout<<"\n\nTravelling Saleman Distance is "<<tsp(1,0);

```

```
55: return 0;
56: }
57:
58:
```

```c
#include<stdio.h>
#include<stdlib.h>
#define queen 0
#define row 1
#define col 2
#define nwtose 3
#define swtone 4
static int count=0;

void intialize(int Board[5][100],int n){

   for(int i=0;i<n;++i){
    Board[queen][i]=-1;
    Board[row][i]=Board[col][i]=0;
   }
    for(int i=0;i<2*n-1;++i)
       Board[nwtose][i]=Board[swtone][i]=0;
}

bool free(int Board[5][100],int i,int j,int n){
    return(Board[row][i]==0 &&Board[col][j]==0 && Board[nwtose][j-i+n-1]==0&&Board[swtone][j+i]==0);
}

void addqueen(int Board[5][100],int i,int j,int n){
    Board[queen][i]=j;
    Board[row][i]=Board[col][j]=Board[nwtose][j-i+n-1]=Board[swtone][j+i]=1;

}

void undoqueen(int Board[5][100],int i,int j,int n){
   Board[queen][i]=-1;
   Board[row][i]=Board[col][j]=Board[nwtose][j-i+n-1]=Board[swtone][j+i]=0;
}

void printsol(int Board[5][100],int n){
    for(int i=0;i<n;++i){
      for(int j=0;j<n;++j)
        { if(Board[queen][i]==j)
           printf(" Q ");
          else
           printf(" - ");
        }
    printf("\n");
}
printf("\n_____\n\n");
}

bool placequeen(int Board[5][100],int i,int n){
      bool extendsoln=false,check=false;
         for(int j=0;j<n;++j){
            if(free(Board,i,j,n)){
             addqueen(Board,i,j,n);
             if(i==n-1)
```

```
53:                printsol(Board,n);
54:               else
55:               extendsoln=placequeen(Board,i+1,n);
56:               if(extendsoln){
57:                check=true;
58:                return true;
59:              }
60:              else
61:                undoqueen(Board,i,j,n);
62:           }
63:         }
64:
65:         if(check==false)
66:          return false;
67:
68:
69: }
70:
71:
72: int main(){
73:     int n;
74:     scanf("%d",&n);
75:     int Board[5][100];
76:     intialize(Board,n);
77:     if(placequeen(Board,0,n))
78:       printsol(Board,n);
79:
80:       return 0;
81: }
```

```c
#include<stdio.h>
#include<stdlib.h>
#define queen 0
#define row 1
#define col 2
#define nwtose 3
#define swtone 4

struct board* Board=NULL;

struct data{
    int* d;
};

struct board{
    struct data* head;
};

struct board* intialize(int n){
    struct board* temp=(struct board*)malloc(sizeof(struct board));
    temp->head=(struct data*)malloc(5*sizeof(struct data));
    for(int i=0;i<3;++i)
        temp->head[i].d=(int*)malloc(n*sizeof(int));
    for(int i=3;i<5;++i)
        temp->head[i].d=(int*)malloc((2*n-1)*sizeof(int));

   for(int i=0;i<n;++i){
    temp->head[queen].d[i]=-1;
   temp->head[row].d[i]=temp->head[col].d[i]=0;
   }

    for(int i=0;i<2*n-1;++i)
    temp->head[nwtose].d[i]=temp->head[swtone].d[i]=0;

  return temp;
}

bool free(int i,int j,int n){
    return(Board->head[1].d[i]==0&&Board->head[2].d[j]==0 && Board->head[3].d[j-i+n-1]==0&&Board->head[4].d[j+i]==0);
}

void addqueen(int i,int j,int n){
    Board->head[queen].d[i]=j;
    Board->head[row].d[i]=1;
    Board->head[col].d[j]=1;
    Board->head[nwtose].d[j-i+n-1]=1;
    Board->head[swtone].d[j+i]=1;
}

void undoqueen(int i,int j,int n){
   Board->head[queen].d[i]=-1;
    Board->head[row].d[i]=0;
    Board->head[col].d[j]=0;
```

```
    Board->head[nwtose].d[j-i+n-1]=0;
    Board->head[swtone].d[j+i]=0;
}

void printsol(int n){
    printf("_____\n");
 int* d;int i=

      for(int j=0;j<n;++j)
        {  printf("| ");
          if(Board->head[queen].d[i]==j)
           printf(" Q ");
          else
           printf("   ");
       }
       printf("|");
       printf("\n|____|____|____|____|____|____|____|____|");
    printf("\n");
}
printf("\n-------------------------------------------------\n");
}
bool placequeen(int i,int n){
      bool extendsoln=false,check=false;
         for(int j=0;j<n;++j){
            if(free(i,j,n)){
             addqueen(i,j,n);
             if(i==n-1)
              printsol(n);
             else
              extendsoln=placequeen(i+1,n);
             if(extendsoln){
             check=true;
              return true;
            }
            else
              undoqueen(i,j,n);
         }
       }
       if(check==false)return false;
}


int main(){
    int n;
    printf("Enter the number of queen:: ");
    sca(Board->head[1].d[i]==0&&Board->head[2].d[j]==0 && Board->head[3].d[j-i+n-1]==0
    if(n==2||n==3){
    printf("No solution exist");
   return 0;
}
    Board=intialize(n);
    if(placequeen(0,n))
      //printsol(n);

```

```
108:        return 0;
109: }
```

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

struct  node* t=NULL;

struct node{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;

};

struct node* cnode(int v){
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->data=v;
    temp->left=temp->right=NULL;
    temp->parent=NULL;
    return(temp);
}

/*int maxof(int a,int b){
    if(a>b)return a;
    return b;
}

void rotate_right(struct node* t){
    int x=t->data;
    int y=t->left->data;
    struct node* tll=t->left->left;
    struct node* tlr=t->left->right;
    struct node* tr=t->right;

    t->data=y;
    t->right=t->left;
    t->right->data=x;
    t->left=tll;
    t->right->left=tlr;
    t->right->right=tr;
}

void rotate_left(struct node* t){
    int y=t->data;
    int z=t->right->data;
    struct node* tll=t->left;
    struct node* tlrl=t->right->left;
    struct node* tlrr=t->right->right;

    t->data=z;
    t->left=t->right;
    t->left->data=y;
    t->left->left=tll;
    t->left->right=tlrl;
    t->right=tlrr;
```

```
 55: }
 56:
 57: int slope(struct node* t){
 58:     return(t->left->ht-t->right->ht);
 59: }
 60:
 61: void rebalance(struct node* t){
 62:     if(slope(t)==2){
 63:         if(slope(t->left)==-1)
 64:           rotate_left(t->left);
 65:     rotate_right(t);
 66:      }
 67:     if(slope(t)==-2){
 68:          if(slope(t->right)==1)
 69:             rotate_right(t->right);
 70:     rotate_left(t);
 71:     }
 72:     return;
 73: }*/
 74: struct node* min(struct node* t){
 75:     struct node* temp=t;
 76:     while(temp->left!=NULL){
 77:         temp=temp->left;
 78:     }
 79:     return(temp);
 80: }
 81: struct node* max(struct node* t){
 82:     struct node* temp=t;
 83:     while(temp->right!=NULL){
 84:         temp=temp->right;
 85:     }
 86:     return(temp);
 87: }
 88: struct node* find(struct node* t,int v){
 89:     if(v==t->data)
 90:       return t;
 91:     else if(v<t->data){
 92:         find(t->left,v);
 93:     }
 94:     else{
 95:         find(t->right,v);
 96:     }
 97: }
 98: struct node* succ(struct node* t,int v){
 99:     struct node* temp=find(t,v);
100:     if(temp->right!=NULL){
101:         return(min(temp->right));
102:     }
103:     struct node* temp2=temp->parent;
104:     while(temp2->parent!=NULL&&temp2->right==temp){
105:         temp=temp2;
106:         temp2=temp2->parent;
107:
108:     }
```

```
109:     if(temp2->parent==NULL&&temp2->right==temp){
110:         return temp2->parent;
111:     }
112:     return(temp2);
113:
114:
115: }
116: struct node* pred(struct node* t,int v){
117:     struct node* temp=find(t,v);
118:     if(temp->left!=NULL){
119:         return(max(temp->left));
120:     }
121:     struct node* temp2=temp->parent;
122:     while(temp2->parent!=NULL&&temp2->left==temp){
123:         temp=temp2;
124:         temp2=temp2->parent;
125:     }
126:     if(temp2->parent==NULL&&temp2->left==temp){
127:         return(temp2->parent);
128:     }
129:     return(temp2);
130: }
131:
132: struct node* insert(struct node* t,int v){
133:     if(t==NULL){
134:         t=cnode(v);
135:         return t;
136:     }
137:     else if(v<t->data){
138:
139:         t->left=insert(t->left,v);
140:         //rebalance(t->left);
141:     //  t->ht=1+maxof(t->left->ht,t->right->ht);
142:         t->left->parent=t;
143:     }
144:     else if(v>t->data){
145:         t->right=insert(t->right,v);
146:         //rebalance(t->right);
147:     //  t->ht=1+maxof(t->left->ht,t->right->ht);
148:         t->right->parent=t;
149:     }
150:     return t;
151: }
152:
153: struct node* del(struct node* t,int v){
154:     if(t==NULL)
155:        return t;
156:     if(v<t->data){
157:         t->left=del(t->left,v);
158:     }
159:     else if(v>t->data){
160:         t->right=del(t->right,v);
161:     }
162:     else{
```

```
163:         if(t->left==NULL){
164:             struct node* temp=t->right;
165:             free(t);
166:             return temp;
167:         }
168:         else if(t->right==NULL){
169:             struct node* temp=t->left;
170:             free(t);
171:             return temp;
172:         }
173:         else if(t->left!=NULL&&t->right!=NULL){
174:              struct node* p=min(t);
175:             t->data=p->data;
176:             t->right=del(t->right,t->data);
177:         }
178:
179:     }
180: }
181: void inorder(struct node* t){
182:     if(t!=NULL){
183:         inorder(t->left);
184:         printf("%d::\n",t->data);
185:         inorder(t->right);
186:     }
187: }
188: int main(){
189:     t=insert(t,5);
190:     t=insert(t,3);
191:     t=insert(t,10);
192:     t=insert(t,1);
193:     t=insert(t,2);
194:     t=insert(t,4);
195:     t=insert(t,11);
196:     inorder(t);
197:    int p;
198:     printf("\n");
199:     printf("For which value you want to know SUccessor: ");
200:     scanf("%d",&p);
201:     struct node* k=succ(t,p);
202:
203:     if(k!=NULL){
204:
205:         printf(" SUCCESSOR od %d IS: %d \n",p,k->data);
206:     }
207:     else{
208:         printf("NO SUCCESSOR EXIST for %d!",p);
209:     }
210:     k=NULL;
211:     printf("\n");
212:     printf("For which value you want to know predeccor: ");
213:     scanf("%d",&p);
214:     k=pred(t,p);
215:     if(k!=NULL){
216:
```

```
217:         printf("\npredeccsor of %d IS: %d \n",p,k->data);
218:     }
219:     else{
220:         printf("\nNO predesccor EXIST!");
221:     }
222: 
223:     printf("\n");
224: }
```