// A C Program to demonstrate **adjacency list** representation of graph

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

Struct AdjListNode {
   Int dest;
 Struct AdjListNode* next;
};
Struct AdjList {
 Struct AdjListNode *head;
};
Struct Graph {
   Int V;
   Struct AdjList* array;
};
Struct AdjListNode* newAdjListNode(int dest) {
   Struct AdjListNode* newNode = (struct AdjListNode*) malloc(
         Sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->next = NULL;
   return newNode;
}

Struct Graph* createGraph(int V) {
   Struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
   Graph->V = V;
   Graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));
   Int I;
   For (I = 0; I < V; ++i)
      Graph->array[i].head = NULL;
   Return graph;
}
Void addEdge(struct Graph* graph, int src, int dest) {
     Struct AdjListNode* newNode = newAdjListNode(dest);
   newNode->next = graph->array[src].head;
   graph->array[src].head = newNode;
   newNode = newAdjListNode(src);
   newNode->next = graph->array[dest].head;
   graph->array[dest].head = newNode;
}
Void printGraph(struct Graph* graph) {
   Int v;
   For (v = 0; v < graph->V; ++v) {
      Struct AdjListNode* pCrawl = graph->array[v].head;
      Printf("\n Adjacency list of vertex %d\n head ", v);
      While (pCrawl) {
         Printf("-> %d", pCrawl->dest);
         pCrawl = pCrawl->next;
```

```
      }
      Printf("\n");
   }
}
Int main() {
   Int V = 5;
   Struct Graph* graph = createGraph(V);
   addEdge(graph, 0, 1);
   addEdge(graph, 0, 4);
   addEdge(graph, 1, 2);
   addEdge(graph, 1, 3);
   addEdge(graph, 1, 4);
   addEdge(graph, 2, 3);
   addEdge(graph, 3, 4);   printGraph(graph);
    return 0;
}
```

2.Aim: Arrange the list of numbers in ascending order using **Heap Sort**.

```
#include<stdio.h>
#include<conio.h>
void Heapsort(int[],int);
int Parent(int);
int Left(int);
int Right(int);
void Heapify(int[],int,int);
void Buildheap(int[],int);
void main()
{
int x[20],i,n;
clrscr();
printf("\n Enter the no of element to be sorted:");
scanf("%d",&n);
printf("\n Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
Heapsort(x,n);
printf("\n The sorted array is:\n");
for(i=0;i<n;i++)
printf("%4d",x[i]);
getch();
}
int Parent(int i)
{
return(i/2);
```

```
}
int Left(int i)
{
return(2*i+1);
}
int Right(int i)
{
return(2*i+2);
}
void Heapify(int a[],int i,int n)
{
int l,r,large,temp ;
l=Left(i);
r=Right(i);
if((l<=n-1)&&(a[l]>a[i]))
large=l;
else
large=i;
if((r<=n-1)&&(a[r]>a[large]))
large=r;
if(large!=i)
{
temp=a[i];
a[i]=a[large];
a[large]=temp;
Heapify(a,large,n);
}
}
void Buildheap(int a[],int n)
{
int i;
for(i=(n-1)/2;i>=0;i--)
Heapify(a,i,n);
}
void Heapsort(int a[],int n)
{
 int i,m,temp;
Buildheap(a,n);
m=n;
for(i=n-1;i>=1;i--)
{
temp=a[0];
a[0]=a[i];
a[i]=temp;
```

```
m=m-1;
Heapify(a,0,m);
}
}
```

3.* C Program to Check whether **two binary trees** are **similar** or not */

```
#include<stdio.h>
#include<stdlib.h>
struct node
{ struct node *lchild;
      int info;
      struct node *rchild;
};
struct node *insert(struct node *ptr, int ikey);
void display(struct node *ptr,int level);
int isSimilar(struct node *p1, struct node *p2);
int main()
{
      struct node *root=NULL,*root1=NULL,*ptr;
      int choice,k,item;
  while(1)
      {
            printf("\n");
            printf("1.Insert Tree 1\n");
            printf("2.Insert Tree 2\n");
            printf("3.Display Tree 1\n");
            printf("4.Display Tree 2\n");
            printf("5.Check for Similar\n");
            printf("6.Quit\n");
            printf("\nEnter your choice : ");
            scanf("%d",&choice);
           switch(choice)
            {

            case 1:
                  printf("\nEnter the key to be inserted : ");
                  scanf("%d",&k);
                  root = insert(root, k);
                  break;

      case 2:
                  printf("\nEnter the key to be inserted : ");
                  scanf("%d",&k);
                  root1 = insert(root1, k);
```

```
                break;
        case 3:
           printf("\n");
           display(root,0);
           printf("\n");
           break;

        case 4:
           printf("\n");
           display(root1,0);
           printf("\n");
           break;

        case 5:
           printf("\n");
           if(isSimilar(root,root1))
              printf("Tree 1 and 2 are Similar\n");
           else
              printf("Tree 1 and 2 are Not Similar\n");
           printf("\n");
           break;

        case 6:
             exit(1);
            default:
                 printf("\nWrong choice\n");
               }     }
        return 0;
}
struct node *insert(struct node *ptr, int ikey )
{
        if(ptr==NULL)
        {
              ptr = (struct node *) malloc(sizeof(struct node));
              ptr->info = ikey;
              ptr->lchild = NULL;
              ptr->rchild = NULL;
        }
        else if(ikey < ptr->info) /*Insertion in left subtree*/
              ptr->lchild = insert(ptr->lchild, ikey);
        else if(ikey > ptr->info) /*Insertion in right subtree */
              ptr->rchild = insert(ptr->rchild, ikey);
        else
              printf("Duplicate key\n");
```

```
    return(ptr);
}/*End of insert( )*/

void display(struct node *ptr,int level)
{
    int i;
    if(ptr == NULL )/*Base Case*/
        return;
    else
  {
        display(ptr->rchild, level+1);
        printf("\n");
        for (i=0; i<level; i++)
            printf("    ");
        printf("%d", ptr->info);
        display(ptr->lchild, level+1);
    }
}/*End of display()*/

int isSimilar(struct node *p1, struct node *p2)
{
    if(p1==NULL && p2==NULL)
        return 1;
    if(p1!=NULL && p2!=NULL)
        if(isSimilar(p1->lchild, p2->lchild) && isSimilar(p1->rchild, p2->rchild))
            return 1;
    return 0;
}
```

4 . Program to Count Number of **Nodes** at **each level** in **Binary Tree** */

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *lchild;
    int info;
    struct node *rchild;
};

struct node *insert(struct node *ptr, int ikey);
void display(struct node *ptr,int level);
int NodesAtLevel(struct node *ptr, int level) ;
int main()
```

```
{
      struct node *root=NULL,*root1=NULL,*ptr;
     int choice,k,item,level;
      while(1)
      {
            printf("\n");
            printf("1.Insert Tree \n");
            printf("2.Display Tree \n");
            printf("3.Number of Nodes \n");
            printf("4.Quit\n");
            printf("\nEnter your choice : ");
            scanf("%d",&choice);
            switch(choice)
            {
            case 1:
                  printf("\nEnter the key to be inserted : ");
                  scanf("%d",&k);
                  root = insert(root, k);
                  break;
      case 2:
         printf("\n");
         display(root,0);
         printf("\n");
         break;
      case 3:
         printf("\n");
         printf("Enter any level :: ");
         scanf("%d",&level);
         printf("\nNumber of nodes at [ %d ] Level :: %d\n",level,NodesAtLevel(root,level));
        break;
      case 4:
                  exit(1);
           default:
                  printf("\nWrong choice\n");
            }/*End of switch */
      }/*End of while */
      return 0;
}/*End of main( )*/
struct node *insert(struct node *ptr, int ikey )
{
      if(ptr==NULL)
      {
            ptr = (struct node *) malloc(sizeof(struct node));
            ptr->info = ikey;
```

```
            ptr->lchild = NULL;
            ptr->rchild = NULL;
      }
      else if(ikey < ptr->info) /*Insertion in left subtree*/
            ptr->lchild = insert(ptr->lchild, ikey);
      else if(ikey > ptr->info) /*Insertion in right subtree */
            ptr->rchild = insert(ptr->rchild, ikey);
      else
            printf("\nDuplicate key\n");
      return(ptr);
}/*End of insert( )*/

void display(struct node *ptr,int level)
{
      int i;
      if(ptr == NULL )/*Base Case*/
            return;
      else
   {
            display(ptr->rchild, level+1);
            printf("\n");
            for (i=0; i<level; i++)
                  printf("    ");
            printf("%d", ptr->info);
            display(ptr->lchild, level+1);
      }
}/*End of display()*/
int NodesAtLevel(struct node *ptr, int level)
{
      if(ptr==NULL)
            return 0;
      if(level==0)
            return 1;
      return NodesAtLevel(ptr->lchild,level-1) + NodesAtLevel(ptr->rchild,level-1);
}
```

5. // **Kruskal's algorithm** in C

```
#include <stdio.h>
#define MAX 30
typedef struct edge {
  int u, v, w;
} edge;
typedef struct edge_list {
  edge data[MAX];
```

```
  int n;
} edge_list;
edge_list elist;
int Graph[MAX][MAX], n;
edge_list spanlist;
void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;
  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }
  sort();
  for (i = 0; i < n; i++)
    belongs[i] = i;
  spanlist.n = 0;
  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);
    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}
int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}
void applyUnion(int belongs[], int c1, int c2) {
  int i;
  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
```

```
    belongs[i] = c1;
}
void sort() {
  int i, j;
  edge temp;
  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}
void print() {
  int i, cost = 0;
  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }
  printf("\nSpanning tree cost: %d", cost);
}
int main() {
  int i, j, total_cost;
  n = 6;
  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 0;
  Graph[1][6] = 0;

  Graph[2][0] = 4;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 3;
```

```
  Graph[2][4] = 4;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 0;
  Graph[3][1] = 0;
  Graph[3][2] = 3;
  Graph[3][3] = 0;
  Graph[3][4] = 3;
  Graph[3][5] = 0;
  Graph[3][6] = 0;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 4;
  Graph[4][3] = 3;
  Graph[4][4] = 0;
  Graph[4][5] = 0;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 0;
  Graph[5][2] = 2;
  Graph[5][3] = 0;
  Graph[5][4] = 3;
  Graph[5][5] = 0;
  Graph[5][6] = 0;
  kruskalAlgo();
  print();
}
```

6..**Topology sort**

```
#include<stdio.h>
#include<stdlib.h>
int s[100], j, res[100]; /*GLOBAL VARIABLES */
void AdjacencyMatrix(int a[][100], int n) { //To generate adjacency matrix for given nodes

   int i, j;
   for (i = 0; i < n; i++) {
      for (j = 0; j <= n; j++) {
         a[i][j] = 0;
      }
```

```
    }
    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            a[i][j] = rand() % 2;
            a[j][i] = 0;
        }
    }
}

void dfs(int u, int n, int a[][100]) { /* DFS */

    int v;
    s[u] = 1;
    for (v = 0; v < n - 1; v++) {
        if (a[u][v] == 1 && s[v] == 0) {
            dfs(v, n, a);
        }
    }
    j += 1;
    res[j] = u;
}

void topological_order(int n, int a[][100]) { /* TO FIND TOPOLOGICAL ORDER*/

    int i, u;
    for (i = 0; i < n; i++) {
        s[i] = 0;
    }
    j = 0;
    for (u = 0; u < n; u++) {
        if (s[u] == 0) {
            dfs(u, n, a);
        }
    }
    return;
}
int main() {
    int a[100][100], n, i, j;

    printf("Enter number of vertices\n"); /* READ NUMBER OF VERTICES */
    scanf("%d", &n);

    AdjacencyMatrix(a, n); /*GENERATE ADJACENCY MATRIX */
```

```
   printf("\t\tAdjacency Matrix of the graph\n"); /* PRINT ADJACENCY MATRIX */
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         printf("\t%d", a[i][j]);
      }
      printf("\n");
   }
   printf("\nTopological order:\n");

   topological_order(n, a);

   for (i = n; i >= 1; i--) {
      printf("-->%d", res[i]);  }
   return 0;
}
```

7.**Adjancy matrix**

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v) {
        int i;
        reach[v]=1;
        for (i=1;i<=n;i++)
          if(a[v][i] && !reach[i]) {
                printf("\n %d->%d",v,i);
                dfs(i);
        }
}
void main() {
        int i,j,count=0;
        clrscr();
        printf("\n Enter number of vertices:");
        scanf("%d",&n);
        for (i=1;i<=n;i++) {
                reach[i]=0;
                for (j=1;j<=n;j++)
                  a[i][j]=0;
        }
        printf("\n Enter the adjacency matrix:\n");
        for (i=1;i<=n;i++)
          for (j=1;j<=n;j++)
           scanf("%d",&a[i][j]);
```

```
        dfs(1);
        printf("\n");
        for (i=1;i<=n;i++) {
                if(reach[i])
                  count++;
        }
        if(count==n)
          printf("\n Graph is connected"); else
          printf("\n Graph is not connected");
        getch();
```

```
/*
 * C Program to find the shortest path between two vertices in a graph
 * using the Floyd-Warshall algorithm
 */

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

void floydWarshall(int **graph, int n)
{
   int i, j, k;
   for (k = 0; k < n; k++)
   {
      for (i = 0; i < n; i++)
      {
         for (j = 0; j < n; j++)
         {
            if (graph[i][j] > graph[i][k] + graph[k][j])
               graph[i][j] = graph[i][k] + graph[k][j];
         }
      }
   }
}

int main(void)
{
   int n, i, j;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   int **graph = (int **)malloc((long unsigned) n * sizeof(int *));
   for (i = 0; i < n; i++)
   {
```

```
        graph[i] = (int *)malloc((long unsigned) n * sizeof(int));
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (i == j)
                graph[i][j] = 0;
            else
                graph[i][j] = 100;
        }
    }
    printf("Enter the edges: \n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("[%d][%d]: ", i, j);
            scanf("%d", &graph[i][j]);
        }
    }
    printf("The original graph is:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    floydWarshall(graph, n);
    printf("The shortest path matrix is:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```