11

```c
#include<stdio.h>
#include <stdlib.h>
int main()
{
    int queue[100],ch=1,front=0,rear=0,i,j=1,x,n;
    printf("Queue using Array");
     printf("\n Enter the size of Queue:");
    scanf("%d",&n);
    x=n;
    printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
    while(ch)
    {
        printf("\nEnter the Choice:");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
            if(rear==x)
                printf("\n Queue is Full");
            else
            {
                printf("\n Enter no %d:",j++);
                scanf("%d",&queue[rear++]);
            }
            break;
        case 2:
            if(front==rear)
            {
                printf("\n Queue is empty");
            }
            else
            {
                printf("\n Deleted Element is %d",queue[front++]);
                x++;
            }
            break;
        case 3:
            printf("\nQueue Elements are:\n ");
            if(front==rear)
                printf("\n Queue is Empty");
            else
            {
```

```
        for(i=front; i<rear; i++)
        {
            printf("%d",queue[i]);
            printf("\n");
        }
```

12

```
#include<stdio.h>
int stack[10],choice,n,top,x,i; // Declaration of variables

void push();
void pop();
void display();

int main()
{
 top = -1;     // Initially there is no element in stack
 printf("\n Enter the size of STACK : ");
 scanf("%d",&n);
 printf("\nSTACK IMPLEMENTATION USING ARRAYS\n");
 do
 {
 printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n");
 printf("\nEnter the choice : ");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1:
 {
 push();
 break;
 }
 case 2:
 {
 pop();
 break;
 }
 case 3:
 {
 display();
 break;
 }
 case 4:
 {
```

```
 break;
 }
 default:
 {
 printf ("\nInvalid Choice\n");
 }}}
 while(choice!=4);
 return 0;
}

void push()
{
 if(top >= n - 1)
 {
 printf("\nSTACK OVERFLOW\n");

 }
 else
 {
 printf("Enter a value to be pushed : ");
 scanf("%d",&x);
 top++;              // TOP is incremented after an element is pushed
 stack[top] = x;   // The pushed element is made as TOP
 }}

void pop()
{
 if(top <= -1)
 {
 printf("\nSTACK UNDERFLOW\n");
 }
 else
 {
 printf("\nThe popped element is %d",stack[top]);
 top--;
 }}

void display()
{
 if(top >= 0)
 {
 printf("\nELEMENTS IN THE STACK\n\n");
 for(i = top ; i >= 0 ; i--)
 printf("%d\t",stack[i]);
```

```
}
else
{
printf("\nEMPTY STACK\n");
}}
```

13

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
int main()
{
int choice;
while (1)
{
printf("1.Insert element to queue n");
printf("2.Delete element from queue n");
printf("3.Display all elements of queue n");
printf("4.Quit n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(1);
default:
printf("Wrong choice n");
}
}
```

```
}
void insert()
{
int item;
if(rear == MAX - 1)
printf("Queue Overflow n");
else
{
if(front== - 1)
front = 0;
printf("Inset the element in queue : ");
scanf("%d", &item);
rear = rear + 1;
queue_array[rear] = item;
}
}
void delete()
{
if(front == - 1 || front > rear)
{
printf("Queue Underflow n");
return;
}
else
{
printf("Element deleted from queue is : %dn", queue_array[front]);
front = front + 1;
}
}
void display()
{
int i;
if(front == - 1)
printf("Queue is empty n");
else
{
printf("Queue is : n");
for(i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("n");
}
}
```

```c
// Tree traversal in C

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

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}
```

```
// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:exit(0);
        }
    }
}
```

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

#define MAX 100

#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);

int queue[MAX], front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

int main()
{
create_graph();
BF_Traversal();
return 0;
}

void BF_Traversal()
{
int v;
for(v=0; v<n; v++)
state[v] = initial;
printf("Enter Start Vertex for BFS: \n");
scanf("%d", &v);
BFS(v);
}

void BFS(int v)
{
int i;
insert_queue(v);
state[v] = waiting;
```

```
while(!isEmpty_queue())
{
v = delete_queue( );
printf("%d ",v);
state[v] = visited;
for(i=0; i<n; i++)
{
if(adj[v][i] == 1 && state[i] == initial)
{
insert_queue(i);
state[i] = waiting;
}
}
}
printf("\n");
}

void insert_queue(int vertex)
{
if(rear == MAX-1)
printf("Queue Overflow\n");
else
{
if(front == -1)
front = 0;
rear = rear+1;
queue[rear] = vertex ;
}
}

int isEmpty_queue()
{
if(front == -1 || front > rear)
return 1;
else
return 0;
}

int delete_queue()
{
int delete_item;
if(front == -1 || front > rear)
{
printf("Queue Underflow\n");
```

```
exit(1);
}
delete_item = queue[front];
front = front+1;
return delete_item;
}

void create_graph()
{
int count,max_edge,origin,destin;

printf("Enter number of vertices : ");
scanf("%d",&n);
max_edge = n*(n-1);

for(count=1; count<=max_edge; count++)
{
printf("Enter edge %d( -1 -1 to quit ) : ",count);
scanf("%d %d",&origin,&destin);

if((origin == -1) && (destin == -1))
break;

if(origin>=n || destin>=n || origin<0 || destin<0)
{
printf("Invalid edge!\n");
count--;
}
else
{
adj[origin][destin] = 1;
}
}
}
```

22.

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

typedef struct node
{
   struct node *next;
   int vertex;
}node;
```

```
node *G[20];
//heads of linked list
int visited[20];
int n;
void read_graph();
//create adjacency list
void insert(int,int);
//insert an edge (vi,vj) in te adjacency list
void DFS(int);
int main()
{
   int i;
   read_graph();
   //initialised visited to 0

for(i=0;i<n;i++)
       visited[i]=0;
   DFS(0);
}
void DFS(int i)
{
   node *p;

printf("\n%d",i);
   p=G[i];
   visited[i]=1;
   while(p!=NULL)
   {
      i=p->vertex;

  if(!visited[i])
          DFS(i);
      p=p->next;
   }
}
void read_graph()
{
   int i,vi,vj,no_of_edges;
   printf("Enter number of vertices:");

scanf("%d",&n);
   //initialise G[] with a null

for(i=0;i<n;i++)
```

```
    {
        G[i]=NULL;
        //read edges and insert them in G[]

printf("Enter number of edges:");
       scanf("%d",&no_of_edges);
       for(i=0;i<no_of_edges;i++)
        {
         printf("Enter an edge(u,v):");
scanf("%d%d",&vi,&vj);
insert(vi,vj);
        }
    }
}
void insert(int vi,int vj)
{
    node *p,*q;

//acquire memory for the new node
q=(node*)malloc(sizeof(node));
    q->vertex=vj;
    q->next=NULL;
    //insert the node in the linked list number vi
    if(G[vi]==NULL)
        G[vi]=q;
    else
    {
        //go to end of the linked list
        p=G[vi];

while(p->next!=NULL)
         p=p->next;
        p->next=q;
    }
}
```

23.

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);
```

```
int main()
{
int G[MAX][MAX],i,j,n,u;
printf("Enter no. of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
printf("\nEnter the starting node:");
scanf("%d",&u);
dijkstra(G,n,u);
return 0;
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{

int cost[MAX][MAX],distance[MAX],pred[MAX];
int visited[MAX],count,mindistance,nextnode,i,j;
//pred[] stores the predecessor of each node
//count gives the number of nodes seen so far
//create the cost matrix
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(G[i][j]==0)
cost[i][j]=INFINITY;
else
cost[i][j]=G[i][j];
//initialize pred[],distance[] and visited[]
for(i=0;i<n;i++)
{
distance[i]=cost[startnode][i];
pred[i]=startnode;
visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
mindistance=INFINITY;
//nextnode gives the node at minimum distance
for(i=0;i<n;i++)
```

```
if(distance[i]<mindistance&&!visited[i])
{
mindistance=distance[i];
nextnode=i;
}
//check if a better path exists through nextnode
visited[nextnode]=1;
for(i=0;i<n;i++)
if(!visited[i])
if(mindistance+cost[nextnode][i]<distance[i])
{
distance[i]=mindistance+cost[nextnode][i];
pred[i]=nextnode;
}
count++;
}

//print the path and distance of each node
for(i=0;i<n;i++)
if(i!=startnode)
{
printf("\nDistance of node%d=%d",i,distance[i]);
printf("\nPath=%d",i);
j=i;
do
{
j=pred[j];
printf("<-%d",j);
}while(j!=startnode);
}
}
```

24.

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 6

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
```

```
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
               graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
```

```
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// driver's code
int main()
{
    int graph[V][V] = { { 0, 3, 1, 0, 0, 0 },
                        { 3, 0, 0, 0, 3, 0 },
                        { 1, 0, 0, 0, 0, 4 },
                        { 0, 0, 0, 0, 0, 2 },
                        { 0, 3, 0, 0, 0, 0 },
                                                { 0, 0, 4, 2, 0, 0 }};

    //
    primMST(graph);
    return 0;
}
```

25.

```
// 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();

// Applying Krushkal Algo
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;
}

// Sorting algo
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;
      }
}

// Printing the result
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();
}
```