1. **Write a C program which uses Binary search tree library and implements following function with recursion: T copy(T) – create another BST which is exact copy of BST which is passed as parameter. int compare(T1, T2) – compares two binary search trees and returns 1 if they are equal and 0 otherwise.**

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
struct node {
   int data;
   struct node *left;
   struct node *right;
};
struct node *getNewNode(int data)
{

   struct node *newNode = (struct node *)malloc(sizeof(struct node));

   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}
struct node *generateBTree()
{

   struct node *root = getNewNode(1);

   root->left = getNewNode(2);
   root->right = getNewNode(3);

   root->left->left = getNewNode(4);
   root->left->right = getNewNode(5);
   root->right->left = getNewNode(6);
   root->right->right = getNewNode(7);

   root->left->left->left = getNewNode(8);
   return root;
}

int isIdentical(struct node *first, struct node *second) {
   if (first == NULL && second == NULL)
      return TRUE;

   if (first == NULL || second == NULL)
      return FALSE;

   if (isIdentical(first->left, second->left) && isIdentical(first->right, second->right) && first->data == second->data)
{
      return TRUE;
   }
   Else {
      return FALSE;
   }
}
```

```
int main() {
   struct node *root1 = generateBTree();
   struct node *root2 = generateBTree();
   if (isIdentical(root1, root2)) {
      printf("Both trees are identical.\n");
   }
   else {
      printf("Both trees are not identical.\n");
   }
   root2->left->data = 10;
   if (isIdentical(root1, root2)) {
      printf("Both trees are identical.\n");
   }
   Else {
      printf("Both trees are not identical.\n");
   }
   return 0;
}
```

2. Implement a Binary search tree (BST) library (btree.h) with operations – create, search, insert, inorder, preorder and postorder. Write a menu driven program that performs the above operations.

```
#include <stdio.h>
#include <stdlib.h>
struct BST
{
	int data;
	struct BST *left;
	struct BST *right;
};
typedef struct BST NODE;
NODE *node;
NODE* createtree(NODE *node, int data)
{
	if (node == NULL)
	{
		NODE *temp;
		temp= (NODE*)malloc(sizeof(NODE));
		temp->data = data;
		temp->left = temp->right = NULL;
		return temp;
	}
	if (data < (node->data))
	{
		node->left = createtree(node->left, data);
	}
	else if (data > node->data)
	{
		node -> right = createtree(node->right, data);
	}
	return node;
}
NODE* search(NODE *node, int data)
{
```

```
	if(node == NULL)
		printf("\nElement not found");
	else if(data < node->data)
	{
		node->left=search(node->left, data);
	}
	else if(data > node->data)
	{
		node->right=search(node->right, data);
	}
	else
		printf("\nElement found is: %d", node->data);
	return node;
}
void inorder(NODE *node)
{
	if(node != NULL)
	{
		inorder(node->left);
		printf("%d\t", node->data);
		inorder(node->right);
	}
}
void preorder(NODE *node)
{
	if(node != NULL)
	{
		printf("%d\t", node->data);
		preorder(node->left);
		preorder(node->right);
	}
}
void postorder(NODE *node)
{
	if(node != NULL)
	{
		postorder(node->left);
		postorder(node->right);
		printf("%d\t", node->data);
	}
}
NODE* findMin(NODE *node)
{
	if(node==NULL)
	{
		return NULL;
	}
	if(node->left)
		return findMin(node->left);
	else
		return node;
}
NODE* del(NODE *node, int data)
{
	NODE *temp;
	if(node == NULL)
```

```
	{
		printf("\nElement not found");
	}
	else if(data < node->data)
	{
		node->left = del(node->left, data);
	}
	else if(data > node->data)
	{
		node->right = del(node->right, data);
	}
	else
	{
		if(node->right && node->left)
		{
			temp = findMin(node->right);
			node -> data = temp->data;

			node -> right = del(node->right,temp->data);
		}
		else
		{

			temp = node;
			if(node->left == NULL)
				node = node->right;
			else if(node->right == NULL)
				node = node->left;
			free(temp); /* temp is longer required */
		}
	}
	return node;
}
int main()
{
	int data, ch, i, n;
	NODE *root=NULL;
	while (1)
	{
		printf("\n1.Insertion in Binary Search Tree");
		printf("\n2.Search Element in Binary Search Tree");
		printf("\n3.Delete Element in Binary Search Tree");
		printf("\n4.Inorder\n5.Preorder\n6.Postorder\n7.Exit");
		printf("\nEnter your choice: ");
		scanf("%d", &ch);
		switch (ch)
		{
			case 1: printf("\nEnter N value: " );
				scanf("%d", &n);
				printf("\nEnter the values to create BST like(6,9,5,2,8,15,24,14,7,8,5,2)\n");
				for(i=0; i<n; i++)
				{
					scanf("%d", &data);
					root=createtree(root, data);
				}
				break;
```

```
                case 2: printf("\nEnter the element to search: ");
                        scanf("%d", &data);
                        root=search(root, data);
                        break;
                case 3: printf("\nEnter the element to delete: ");
                        scanf("%d", &data);
                        root=del(root, data);
                        break;
                case 4: printf("\nInorder Traversal: \n");
                        inorder(root);
                        break;
                case 5: printf("\nPreorder Traversal: \n");
                        preorder(root);
                        break;
                case 6: printf("\nPostorder Traversal: \n");
                        postorder(root);
                        break;
                case 7: exit(0);
                default:printf("\nWrong option");
                        break;
            }
        }
        return 0;
}
```

3. **Write a program which uses binary search tree library and counts the total nodes and total leaf nodes in the tree. int count(T) – returns the total number of nodes from BST int countLeaf(T) – returns the total number of leaf nodes from BST.**

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *left;
   struct node *right;
};

struct node* getNewNode(int data) {

  struct node* newNode = ((struct node*)malloc(sizeof(struct node)));


  newNode->data = data;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}


struct node* generateBTree(){
   // Root Node
   struct node* root =  getNewNode(1);
   // Level 2 nodes
   root->left = getNewNode(2);
```

```
    root->right = getNewNode(3);
    // Level 3 nodes
    root->left->left = getNewNode(4);
    root->left->right = getNewNode(5);
    root->right->left = getNewNode(6);
    root->right->right = getNewNode(7);
    // Level 4 nodes
    root->left->left->left = getNewNode(8);

    return root;

}



int countLeafNode(struct node *root){

    if(root == NULL)
        return 0;

    if(root->left == NULL && root->right == NULL)
        return 1;


    return countLeafNode(root->left) + countLeafNode(root->right);
}

int main() {
 struct node *root = generateBTree();


    printf("Number of leaf Node : %d", countLeafNode(root));

    getchar();
    return 0;
}
```

4. Write a C program which uses Binary search tree library and displays nodes at each level, count of node at each level and total levels in the 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. Write a C program for the implementation of Topological sorting.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int n;          /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
void create_graph();
int queue[MAX], front = -1, rear = -1;
void insert_queue(int v);
int delete_queue();
int isEmpty_queue();
int indegree(int v);
int main()
{
    int i, v, count, topo_order[MAX], indeg[MAX];
    create_graph();
    /*Find the indegree of each vertex*/
    for (i = 0; i < n; i++)
    {
        indeg[i] = indegree(i);
        if (indeg[i] == 0)
```

```
            insert_queue(i);
    }
    count = 0;
    while (!isEmpty_queue() && count < n)
    {
        v = delete_queue();
        topo_order[++count] = v; /*Add vertex v to topo_order array*/
        /*Delete all edges going fron vertex v */
        for (i = 0; i < n; i++)
        {
            if (adj[v][i] == 1)
            {
                adj[v][i] = 0;
                indeg[i] = indeg[i] - 1;
                if (indeg[i] == 0)
                    insert_queue(i);
            }
        }
    }
    if (count < n)
    {
        printf("\nNo topological ordering possible, graph contains cycle\n");
        exit(1);
    }
    printf("\nVertices in topological order are :\n");
    for (i = 1; i <= count; i++)
        printf("%d ", topo_order[i]);
    printf("\n");
    return 0;
} /*End of main()*/
void insert_queue(int vertex)
{
    if (rear == MAX - 1)
        printf("\nQueue Overflow\n");
    else
    {
        if (front == -1) /*If queue is initially empty */
            front = 0;
        rear = rear + 1;
        queue[rear] = vertex;
    }
} /*End of insert_queue()*/
int isEmpty_queue()
{
    if (front == -1 || front > rear)
        return 1;
    else
        return 0;
} /*End of isEmpty_queue()*/
int delete_queue()
{
    int del_item;
    if (front == -1 || front > rear)
    {
        printf("\nQueue Underflow\n");
        exit(1);
```

```
    }
    else
    {
        del_item = queue[front];
        front = front + 1;
        return del_item;
    }
} /*End of delete_queue() */
int indegree(int v)
{
    int i, in_deg = 0;
    for (i = 0; i < n; i++)
        if (adj[i][v] == 1)
            in_deg++;
    return in_deg;
} /*End of indegree() */
void create_graph()
{
    int i, max_edges, origin, destin;
    printf("\nEnter number of vertices : ");
    scanf("%d", &n);
    max_edges = n * (n - 1);
    for (i = 1; i <= max_edges; i++)
    {
        printf("\nEnter edge %d(-1 -1 to quit): ", i);
        scanf("%d %d", &origin, &destin);
        if ((origin == -1) && (destin == -1))
            break;
        if (origin >= n || destin >= n || origin < 0 || destin < 0)
        {
            printf("\nInvalid edge!\n");
            i--;
        }
        else
            adj[origin][destin] = 1;
    }
}
```

6. Write a program to sort n randomly generated elements using Heapsort method.

```
#include <stdio.h>

void heapify(int arr[], int end, int start) {
    int largest = start;
    int left = 2 * start + 1;
    int right = 2 * start + 2;

    if(left < end && arr[left] > arr[largest]) {
        largest = left;
    }

    if(right < end && arr[right] > arr[largest]) {
        largest = right;
    }

    if(largest != start) {
        int temp = arr[start];
```

```
        arr[start] = arr[largest];
        arr[largest] = temp;

        heapify(arr,end,largest);
    }
}

void heapsort(int arr[], int n) {
    int i;

    for(i = n/2; i >= 0; i--) {
        heapify(arr,n,i);
    }

    for (i = n-1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr,i,0);
    }

}

void displayArr(int arr[], int n) {
    int i;
    for(i=0; i<n; i++) {
        printf("%d ",arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {48,10,23,43,28,26,1};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("\nBefore Sorting: ");
    displayArr(arr,n);

    heapsort(arr,n);
    printf("\nAfter Sorting: ");
    displayArr(arr,n);

    getchar();
    return 0;
}
```

7. Write a C program for the implementation of Dijkstra's shortest path algorithm for finding shortest path from a given source vertex using adjacency cost matrix.

```
#include<stdio.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;
	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];
	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;
		for(i=0;i<n;i++)
			if(distance[i]<mindistance&&!visited[i])
			{
				mindistance=distance[i];
				nextnode=i;
			}
		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++;
        }
        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);
                }
}
```

8. Write a C program for the Implementation of Kruskal's Minimum spanning tree algorithm.

```
// Kruskal Algorithm
#include <stdio.h>
#include <stdlib.h>
int i, j, k, a, b, u, v, n, ne = 1;
int min, mincost = 0, cost[9][9], parent[9];
int find(int);
int uni(int, int);
int main()
{
        printf("Kruskal's algorithm in C\n");
        printf("========================\n");
        printf("Enter the no. of vertices:\n");
        scanf("%d", &n);
        printf("\nEnter the cost adjacency matrix:\n");
        for (i = 1; i <= n; i++)
        {
                for (j = 1; j <= n; j++)
                {
                        scanf("%d", &cost[i][j]);
                        if (cost[i][j] == 0)
                                cost[i][j] = 999;
                }
        }
        printf("The edges of Minimum Cost Spanning Tree are\n");
        while (ne < n)
        {
                for (i = 1, min = 999; i <= n; i++)
                {
                        for (j = 1; j <= n; j++)
                        {
                                if (cost[i][j] < min)
                                {
                                        min = cost[i][j];
                                        a = u = i;
                                        b = v = j;
```

```
                    }
                }
            }
            u = find(u);
            v = find(v);
            if (uni(u, v))
            {
                printf("%d edge (%d,%d) =%d\n", ne++, a, b, min);
                mincost += min;
            }
            cost[a][b] = cost[b][a] = 999;
        }
        printf("\nMinimum cost = %d\n", mincost);
        return 0;
}
int find(int i)
{
        while (parent[i])
            i = parent[i];
        return i;
}
int uni(int i, int j)
{
        if (i != j)
        {
            parent[j] = i;
            return 1;
        }
        return 0;
}
```

9. Write a C program for the implementation of Floyd Warshall's algorithm for finding all pairs shortest path using adjacency cost matrix.

```
#include <stdio.h>
#define V 4
#define INF 99999

void printSolution(int dist[][V]);
void floydWarshall(int graph[][V])
{
        int dist[V][V], i, j, k;

        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];
        for (k = 0; k < V; k++) {

            for (i = 0; i < V; i++) {
                for (j = 0; j < V; j++) {
                    if (dist[i][j] > (dist[i][k] + dist[k][j])
                                && (dist[k][j] != INF
                                    && dist[i][k] != INF))
                        dist[i][j] = dist[i][k] + dist[k][j];
                }
```

```
			}
		}
		printSolution(dist);
}
void printSolution(int dist[][V])
{
int i,j;
		 printf( "The following matrix shows the shortest "
			"distances"
			" between every pair of vertices \n");
		for (i = 0; i < V; i++) {
			for (j = 0; j < V; j++) {
				if (dist[i][j] == INF)
					printf ( "INF");
				else
					printf("%d",dist[i][j]);
			}
			printf("\n");
		}
}

int main()
{
		int graph[V][V] = { { 0, 5, INF, 10 },
			{ INF, 0, 3, INF },
			{ INF, INF, 0, 1 },
			{ INF, INF, INF, 0 } };
		floydWarshall(graph);
		return 0;
}
```

10. Write a C program for the Implementation of Prim’s Minimum spanning tree algorithm.

```
// Prim's Algorithm

#include <stdio.h>
#include <limits.h>
#define vertices 5
int minimum_key(int k[], int mst[])
{
		int minimum = INT_MAX, min,i;

		for (i = 0; i < vertices; i++)
			if (mst[i] == 0 && k[i] < minimum )
				minimum = k[i], min = i;

		return min;
}

void prim(int g[vertices][vertices])
{
		int parent[vertices];
		int k[vertices];
		int mst[vertices];
```

```
        int i, count,u,v;

        for (i = 0; i < vertices; i++)  k[i] = INT_MAX, mst[i] = 0;

        k[0] = 0;
        parent[0] = -1;

        for (count = 0; count < vertices-1; count++)
        {

                u = minimum_key(k, mst);
                mst[u] = 1;

                for (v = 0; v < vertices; v++)

                        if (g[u][v] && mst[v] == 0 && g[u][v] < k[v])
                                parent[v] = u, k[v] = g[u][v];
        }

        for (i = 1; i < vertices; i++)
                printf("%d %d %d \n", parent[i], i, g[i][parent[i]]);
}
void main()
{
        int g[vertices][vertices] = {{3, 2, 1, 9, 0},
                {5, 1, 2, 10, 4},
                {0, 4, 1, 0, 9},
                {8, 10, 0, 2, 10},
                {1, 6, 8, 11, 0},
        };

        prim(g);
}
```

11. Write a C program that accepts the vertices and edges of a graph and stores it as an adjacency matrix. Display the adjacency matrix.

```
#include<stdio.h>
void main()
{
   int AdjMat[10][10], n;
   printf("\nEnter Number of Vertices\n");
   scanf("%d", &n);
   read_graph(AdjMat, n);
   printf("\n Given Adjacency Matrix is:\n");
   display(AdjMat, n);
}
void display(int AdjMat[20][20],int n)
{
   int i, j;
   for(i=1;i<=n;i++)
   {
      for(j=1;j<=n;j++)
      {
         printf("%d\t", AdjMat[i][j]);
      }
```

```
        printf("\n");
    }
}
void read_graph(int AdjMat[20][20],int n)
{
    int i, j;
    char reply;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(i==j)
                AdjMat[i][j]=0;
            else
            {
                printf("\n Vertices %d and %d are Adjacent?(Y||N)\n", i, j);
                fflush(stdin);
                scanf("%c", &reply);
                if(reply=='y'||reply=='Y')
                    AdjMat[i][j]=1;
                else
                    AdjMat[i][j]=0;
            }
        }
    }
}
```

12. Write a C program that accepts the vertices and edges of a graph. Create adjacency list and display the adjacency list.

```
#include<stdio.h>
#include<malloc.h>
struct node
{
    int vertex;
    struct node *next;
};
void main()
{
    int n;
    printf("\nEnter Number of Vertices\n");
    scanf("%d", &n);
    read_graph(n);
}
void read_graph(int n)
{
    int i, j;
    char reply;
    struct node *AdjList[10];
    struct node *newnode, *temp;
    for(i=1;i<=n;i++)
    {
        AdjList[i]=NULL;
    }
    for(i=1;i<=n;i++)
    {
```

```
        for(j=1;j<=n;j++)
        {
          if(i!=j)
          {
          printf("\n Vertices %d and %d are Adjacent?(Y||N)\n", i, j);
          fflush(stdin);
          scanf("%c", &reply);
          if(reply=='y'||reply=='Y')
          {
             newnode=(struct node*)malloc(sizeof(struct node));
             newnode->vertex=j;
             newnode->next=NULL;
             if(AdjList[i]==NULL)
             {
                AdjList[i]=newnode;
             }
             else
             {
                temp=AdjList[i];
                while(temp->next!=NULL)
                {
                   temp=temp->next;
                }
                temp->next=newnode;
             }

          }
          }
        }
    }
    display(AdjList, n);
}
void display(struct node *AdjList[10], int n)
{
    int i;
    struct node *temp;
    printf("Vertex\tList\n");
    for(i=1; i<=n;i++)
    {
        temp=AdjList[i];
        printf("%d\t", i);
        while(temp!=NULL)
        {
          printf("%d->",temp->vertex);
          temp=temp->next;
        }
        printf("NULL\n");
    }
}
```

13. Write a C program that accepts the vertices and edges of a graph and store it as an adjacency list. Implement function to traverse the graph using Depth First Search (BFS) traversal.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex
{
   char label;
   bool visited;
};
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;
struct Vertex *lstVertices[MAX];
int adjMatrix[MAX][MAX];
int vertexCount = 0;

void insert(int data)
{
   queue[++rear] = data;
   queueItemCount++;
}
int removeData()
{
   queueItemCount--;
   return queue[front++];
}
bool isQueueEmpty()
{
   return queueItemCount == 0;
}
void addVertex(char label)
{
   struct Vertex *vertex = (struct Vertex *)malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}
void addEdge(int start, int end)
{
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
void displayVertex(int vertexIndex)
{
   printf("%c ", lstVertices[vertexIndex]->label);
}
int getAdjUnvisitedVertex(int vertexIndex)
{
   int i;
   for (i = 0; i < vertexCount; i++)
   {
```

```
        if (adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
            return i;
    }
    return -1;
}
void breadthFirstSearch()
{
    int i;
    lstVertices[0]->visited = true;
    displayVertex(0);
    insert(0);
    int unvisitedVertex;
    while (!isQueueEmpty())
    {
        int tempVertex = removeData();
        while ((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1)
        {
            lstVertices[unvisitedVertex]->visited = true;
            displayVertex(unvisitedVertex);
            insert(unvisitedVertex);
        }
    }
    for (i = 0; i < vertexCount; i++)
    {
        lstVertices[i]->visited = false;
    }
}
int main()
{
    int i, j;
    for (i = 0; i < MAX; i++)
    {
        for (j = 0; j < MAX; j++)
            adjMatrix[i][j] = 0;
    }
    addVertex('S');
    addVertex('A');
    addVertex('B');
    addVertex('C');
    addVertex('D');
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(0, 3);
    addEdge(1, 4);
    addEdge(2, 4);
    addEdge(3, 4);
    printf("\nBreadth First Search: ");

    breadthFirstSearch();
    return 0;
}
```

14. Write a C program that accepts the vertices and edges of a graph and store it as an adjacency matrix. Implement function to traverse the graph using Depth First Search (DFS) traversal.

```
#include<stdio.h>
void DFS(int);
int G[10][10],visited[10],n;
void main()
{
 int i,j;
 printf("Enter number of vertices:");

scanf("%d",&n);
printf("\nEnter adjecency matrix of the graph:");

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
 for(i=0;i<n;i++)
 visited[i]=0;
 DFS(0);
}
void DFS(int i)
{
 int j;
printf("\n%d",i);
 visited[i]=1;
for(j=0;j<n;j++)
 if(!visited[j]&&G[i][j]==1)
 DFS(j);
}
```

15. Write a menu driven program to implement hash table using array (insert, search, delete, display). Use any of the above-mentioned hash functions. In case of collision apply quadratic probing.

```
#include<stdio.h>
#include<stdlib.h>
struct item
{
	int key;
	int value;
};
 struct hashtable_item {
	int flag;
	struct item *data;
};

struct hashtable_item *array;
int size = 0;
int max = 10;
int hashcode(int key) {
	return (key % max);
}
 void init_array() {
```

```
        int i;
        for (i = 0; i < max; i++) {
                array[i].flag = 0;
                array[i].data = NULL;
        }
}
 void insert(int key, int value) {
        int index = hashcode(key);
        int i = index;
        int h = 1;
        struct item *new_item = (struct item*) malloc(sizeof(struct item));
        new_item->key = key;
        new_item->value = value;
        while (array[i].flag == 1)  {

                if (array[i].data->key == key) {
                        printf("\n This key is already present in hash table, hence updating it's value \n");
                        array[i].data->value = value;
                        return;
                }
                i = (i + (h * h)) % max;
                h++;
                if (i == index) {
                        printf("\n Hash table is full, cannot add more elements \n");
                        return;
                }
        }
        array[i].flag = 1;
        array[i].data = new_item;
        printf("\n Key (%d) has been inserted\n", key);
        size++;
}

void remove_element(int key) {
        int index = hashcode(key);
        int i = index;
        int h = 1;
        while (array[i].flag != 0){
                if (array[i].flag == 1  &&  array[i].data->key == key) {
                        array[i].flag = 2;
                        array[i].data = NULL;
                        size--;
                        printf("\n Key (%d) has been removed \n", key);
                        return;
                }
                i = (i + (h * h)) % max;
                h++;
                if (i == index) {
                        break;
                }
        }
        printf("\n Key does not exist \n");
}
 void display() {

        int i;
```

```
	for(i = 0; i < max; i++) {
		if (array[i].flag != 1) {
			printf("\n Array[%d] has no elements \n", i);
		}
		else {
			printf("\n Array[%d] has elements \n  %d (key) and %d (value) \n", i, array[i].data->key, 
array[i].data->value);
		}
	}
}


int size_of_hashtable() {
	return size;
}

int main() {
	int choice, key, value, n, c;
	array = (struct hashtable_item*) malloc(max * sizeof(struct hashtable_item*));
	init_array();
	do {
		printf("Implementation of Hash Table in C with Quadratic Probing.\n\n");
		printf("MENU-: \n1.Inserting item in the Hash table"
		   "\n2.Removing item from the Hash table"
		   "\n3.Check the size of Hash table"
		   "\n4.Display Hash table"
		      "\n\n Please enter your choice-:");
		scanf("%d", &choice);
		switch(choice) {
		case 1:	printf("Inserting element in Hash table \n");
			printf("Enter key and value-:\t");
			scanf("%d %d", &key, &value);
			insert(key, value);
			break;
		case 2:	printf("Deleting in Hash table \n Enter the key to delete-:");
			scanf("%d", &key);
			remove_element(key);
			break;
		case 3:	n = size_of_hashtable();
			printf("Size of Hash table is-:%d\n", n);
			break;
		case 4:	display();
			break;
		default: printf("Wrong Input\n");
		}
		printf("\n Do you want to continue-:(press 1 for yes)\t");
		scanf("%d", &c);
	}while(c == 1);
	return 0;
}
```

16. Write a menu driven program to implement hash table using array (insert, search, delete, display). Use any of the above-mentioned hash functions. In case of collision apply linear probing.

Write a menu driven program to implement hash table using array (insert, search, delete, display). Use any of the above-mentioned hash functions. In case of collision apply linear probing.

```
#include<stdio.h>

#define size 7

int arr[size];

void init() {

    int i;

    for(i = 0; i < size; i++)

        arr[i] = -1;

}


void insert(int value) {

    int key = value % size;


    if(arr[key] == -1)

    {

        arr[key] = value;

        printf("%d inserted at arr[%d]\n", value,key);

    }

    else

    {

        printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);

        printf("Unable to insert %d\n",value);

    }

}


void del(int value)

{

    int key = value % size;

    if(arr[key] == value)

        arr[key] = -1;

    else

        printf("%d not present in the hash table\n",value);

}


void search(int value)
```

```
{
    int key = value % size;
    if(arr[key] == value)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
}

void print()
{
    int i;
    for(i = 0; i < size; i++)
        printf("arr[%d] = %d\n",i,arr[i]);
}

int main()
{
    init();
    int choice;
    printf("Eneter your choice");
    do{
    printf("\n1.Insert ");
    printf("\n2.Delete");
    printf("\n3.Searching");
    printf("\n4.Exit ");
    printf("\nEnter your choice");
    scanf("%d",&choice);
    switch(choice){
    case 1:

    insert(10); //key = 10 % 7 ==> 3
    insert(4);  //key = 4 % 7  ==> 4
    insert(2);  //key = 2 % 7  ==> 2
    insert(3);  //key = 3 % 7  ==> 3 (collision)
    printf("Hash table\n");
```

```
    print();
    printf("\n");
    break;
    case 2:
    printf("Deleting value 10..\n");
    del(10);
    printf("After the deletion hash table\n");
    print();
    printf("\n");
    printf("Deleting value 5..\n");
    del(5);
    printf("After the deletion hash table\n");
    print();
    printf("\n");
    break;
    case 3:
    printf("Searching value 4..\n");
    search(4);
    printf("Searching value 10..\n");
    search(10);
    break;
    case 4:
    exit(0);
    break;
    }
    }while(choice!=5);
    return 0;
}
```

17. Write a C program that accepts the vertices and edges of a graph and store it as an adjacency matrix. Implement functions to print indegree, outdegree and total degree of all vertices of graph.

```
#include<stdio.h>
void main()
{
    int indegree, outdegree, total, i, j;
    int AdjMat[10][10], n;
    printf("\nEnter Number of Vertices\n");
    scanf("%d", &n);
```

```
    read_graph(AdjMat, n);
    printf("\n Given Adjacency Matrix is:\n");
    display(AdjMat, n);
    printf("\nIndegree of All Vertices");
    InDegree(AdjMat, n);
    printf("\nOutdegree of All Vertices");
    OutDegree(AdjMat, n);
    printf("\nTotal degree of All Vertices");
    TotalDegree(AdjMat, n);
/*    printf("\n\nV\tInD\tOutD\tTotal\n");
    for(i=1;i<=n;i++)
    {
        indegree=0;
        outdegree=0;
        for(j=1;j<=n;j++)
        {
            if(AdjMat[i][j]!=0)
                indegree++;
        }
        for(j=1;j<=n;j++)
        {
            if(AdjMat[j][i]!=0)
                outdegree++;
        }
        printf("\n%d\t%d\t%d\t%d\t", i,indegree, outdegree,indegree+outdegree);
    }*/
}
void InDegree(int AdjMat[20][20],int n)
{
    int i, j, indeg=0;
    printf("\n\nV\tInD\n");
    for(i=1;i<=n;i++)
    {
        indeg=0;
        for(j=1;j<=n;j++)
        {
            if(AdjMat[j][i]!=0)
                indeg++;
        }
        printf("\n%d\t%d", i,indeg);
    }
}
void OutDegree(int AdjMat[20][20],int n)
{
    int i, j, outdeg=0;
    printf("\n\nV\tOutD\n");
    for(i=1;i<=n;i++)
    {
        outdeg=0;
        for(j=1;j<=n;j++)
        {
            if(AdjMat[i][j]!=0)
                outdeg++;
        }
        printf("\n%d\t%d", i,outdeg);
    }
```

```
}
void TotalDegree(int AdjMat[20][20],int n)
{
    int i, j, total=0;
    printf("\n\nV\tTotalD\n");
    for(i=1;i<=n;i++)
    {
        total=0;
        for(j=1;j<=n;j++)
        {
            if(AdjMat[j][i]!=0)
                total++;
        }
        for(j=1;j<=n;j++)
        {
            if(AdjMat[i][j]!=0)
                total++;
        }
        printf("\n%d\t%d\t", i,total);
    }
}
void display(int AdjMat[20][20],int n)
{
    int i, j;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("%d\t", AdjMat[i][j]);
        }
        printf("\n");
    }
}
void read_graph(int AdjMat[20][20],int n)
{
    int i, j;
    char reply;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(i==j)
                AdjMat[i][j]=0;
            else
            {
                printf("\n Vertices %d and %d are Adjacent?(Y||N)\n", i, j);
                fflush(stdin);
                scanf("%c", &reply);
                if(reply=='y'||reply=='Y')
                    AdjMat[i][j]=1;
                else
                    AdjMat[i][j]=0;
            }
        }
    }
}
```