DATA STRUCTURE USING C

(KCS351)

LABORATORY MANUAL

B.TECH IInd YEAR – IIIrd SEMESTER

`

United College of Engineering & Research, Praygraj
Department of Computer Science & Engineering

Vision of the Department

- To enhance effective teaching and learning by strengthening high academic goals of the students and strong academic leadership among the faculty members. Besides, the department envisages generating an active professional and research environment through industry and R & D orientation at the national and international levels.

- To become a self-sustained unit with its expanded depth and breadth in the areas of Computer Science & Engineering focusing on innovative educational research, applied and interdisciplinary nature of Applied Computing, Consultancy and Training.

Mission of the Department

- To provide Quality education for latest technologies and involving them in live projects in order to achieve the highest standards in theoretical and practical aspects across the computer science discipline.

- To develop necessary skills in collaboration with industry and academia inculcating sincere learning and nobility in profession.

- To develop technical abilities and skills along with its practical implementation in youth to meet the need of profession and society.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1: To provide students the necessary fundamentals of mathematics, science and engineering to create, select and apply appropriate techniques, resources and modern IT tools including simulation and modeling to complex engineering application. Further to prepare them for R & D and consultancy enabling them formulate, solve and analyze engineering problems for the higher learning and professional outcomes.

PEO2: To provide students adequate exposure to skill enhancement, trainings and opportunities to work as teams on multidisciplinary projects with effective communication skills and leadership qualities enabling them equipped with the abilities to work logically, accurately, ethically and efficiently, to generate new knowledge, ideas or products, to implement these solutions in practice, and to develop an ability to analyze the requirements of the software, its design and its technical specifications yielding novel engineering solutions.

PEO3: To prepare students for a successful career and work with social and human values meeting the requirements of Indian and multinational companies. Further, to design, construct, implement and evaluate a computer based system, process, component or program to meet desired needs within realistic constraints such as economics, environmental, social, political, health and safety, manufacturability and sustainability and to promote student awareness on the life-long value-based professional learning.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO 1: The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, big data analytics, and networking for efficient design of computer-based systems of varying complexity.

PSO 2: The ability to understand the evolutionary changes in computing, apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success, real world problems and meet the challenges of the future.

PSO 3: The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, lifelong learning and a zest for higher studies and also to act as a good citizen by inculcating in them moral values & ethics.

PROGRAM OUTCOMES (POs)

PO -1	**Engineering Knowledge**: Apply knowledge of mathematics and science, with fundamentals of Computer Science & Engineering to be able to solve complex engineering problems related to Computer Science.
PO -2	**Problem Analysis**: Identify, Formulate, review research literature and analyze complex engineering problems related to CS and reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences
PO -3	**Design/Development of solutions**: Design solutions for complex engineering problems related to computer science and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety and the cultural societal and environmental considerations
PO -4	**Conduct Investigations of Complex problems**: Use research–based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO -5	**Modern Tool Usage**: Create, Select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to computer science related complex engineering activities with an understanding of the limitations
PO -6	**The Engineer and Society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the computer science professional engineering practice
PO -7	**Environment and Sustainability:** Understand the impact of the computer science professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of, and need for sustainable development
PO -8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
PO -9	**Individual and Team Work**: Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary Settings
PO -10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large such as able to comprehend and with write effective reports and design documentation, make effective presentations and give and receive clear instructions.
PO -11	**Project Management and Finance**: Demonstrate knowledge and understanding of the engineering management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi disciplinary environments
PO -12	**Life-Long Learning**: Recognize the need for and have the preparation and ability to engage in independent and life-long learning the broadest context of technological change

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.

2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.

3. Student should enter into the laboratory with:

   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.

   b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.

   c. Proper Dress code and Identity card.

4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.

5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.

6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.

7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.

8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.

9. Students must take the permission of the faculty in case of any urgency to go out ; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.

10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

OBJECTIVES AND OUTCOMES

OBJECTIVES:

1. To introduce the concept of basic data structures through ADT including List, Stack Queue and their implementations.
2. To understand the importance of data structures in context of writing efficient programs for real world problems.
3. To develop skills to apply appropriate data structures in problem solving and compare the complexity of various algorithms.

COURSE OUTCOMES:

Course Outcome (CO)	Bloom's Knowledge Level (KL)	
At the end of course , the student will be able to:		
CO 1	Understand how arrays, linked lists, stacks, queues, trees, and graphs are represented in memory and their applications in problem solving.	L1, L2
CO 2	Understand the working of stack and queue data structures and apply recursion to solve problems like tower of Hanoi.	L2, L3
CO 3	Implement appropriate sorting/searching technique for a given problem and discuss the computational efficiency.	L2, L3
CO 4	Apply non-linear data structure graph to solve real world problems like shortest distance and minimum spanning tree.	L2, L3
CO 5	Understand various types of tree data structure and be familiar with advanced data structures such as AVL Tree, B Tree & Binary Heaps.	L3, L4

RECOMMENDED SYSTEM / SOFTWARE REQUIREMENTS:

1. Intel based desktop PC of 166MHz or faster processor with at least 64 MB RAM and 100 MB free disk space.

2. Turbo C++ compiler or GCC compilers .

USEFUL TEXT BOOKS / REFERECES:

1. Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, “Data Structures Using C and C++”, PHI Learning Private Limited, Delhi India

2. Horowitz and Sahani, “Fundamentals of Data Structures”, Galgotia Publications Pvt Ltd Delhi India.

3. Lipschutz, “Data Structures” Schaum’s Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.

4. Thareja, “Data Structure Using C” Oxford Higher Education.

LIST OF PROGRAMS

Write C Programs to illustrate the concept of the following:

Topic	Program List
1. Sorting Algorithms-Non-Recursive.	Implementation of Selection Sort
	Implementation of Bubble Sort
	Implementation of Insertion Sort
2. Sorting Algorithms-Recursive.	Implementation of Quick Sort
	Implementation of Heap Sort
	Implementation of Merge Sort
3. Linked List	Implementation of Singly Linked List and operations performed on it
	Implementation of Doubly Linked List and operations performed on it
	Implementation of Circular Linked List and operations performed on it
4. Searching Algorithm.	Implementation of Linear Search
	Implementation of Binary Search
	Implementation of Index Sequential Search
5.STACK	Implementation of Stack using Array
	Implementation of Stack using Linked List
6. QUEUE	Implementation of Queue using Array
	Implementation of Queue using Linked List
	Implementation of Circular Queue using Array
	Implementation of Circular Queue using Linked List
7. TREE	Implementation of Tree Structures, Binary Tree
	Implementation of Binary Tree Traversal algorithms: In-order, Pre-order and Post-order
	Implementation of Binary Search Tree
	Insertion and Deletion in BST
8.GRAPH	Implementation of graph in memory
	Implementation of graph traversal algorithms: BFS, DFS
	Implementation of Minimum cost spanning tree: Kruskal‘s Algorithm
	Implementation of Minimum cost spanning tree: Prim’s Algorithm
	Implementation of shortest path algorithm: Dijkstra’s Algorithm
	Implementation of shortest path algorithm: Warshal’s Algorithm
9. Hashing	Implementation of Hashing Algorithms
	Implementation of Collision resolution techniques in Hashing

PROGRAM-1

AIM: Implementation of Selection Sort

ALGORITHM:

Step1:

Take first a list of unsorted values

Step2:

Consider the first element as minimum element store its index value in a variable

Step3:

Repeat the step 2 until last comparison takes place

Step4:

Compare the minimum with rest of all elements to find minimum value and interchange the minimum value with the first element

Step5: Repeat step 3 to 4 until the list is sorted

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int a[10],i,j,temp,n;
int min,loc;
clear();
printf("\n enter the max no.of elements u want to sort \n");
scanf("%d",&n);
printf("\n enter the elements u want to sort \n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(i=0;i<n-1;i++)
min=a[i];
loc=1;
```

```
for(j=i+1;j<=n;j++)
{
if(min>a[j])
{
min=a[j];
loc=j;
}
}
}
temp=a[i];
a[i]=a[loc];
a[loc]=temp;
}
for(i=0;i<n;i++)
{printf("%d\t",a[i]);
}}
Return 0; }
```

Result:

Enter the max no. of elements u want to sort

5

Enter the elements u want to sort

10 20 15 6 40

6 10 15 20 40

PROGRAM-2

AIM: Implementation of Bubble Sort

ALGORITHM:

Step1:

Take first two elements of a list and compare them

Step2:

If the first elements greater than second then interchange else keep the values as it

Step3:

Repeat the step 2 until last comparison takes place

Step4:

Repeat step 1 to 3 until the list is sorted

SOURCE CODE:

```
#include<stdio.h>
main()
{
int a[10],i,j,temp,n;
clear();
printf("\n enter the max no.of elements u want to sort \n");
scanf("%d",&n);
printf("\n enter the elements u want to sort \n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
```

```
a[j]=temp;
}
}
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
} getch();}
```

Result:

Enter the max no. of elements u want to sort

5

Enter the elements u want to sort

10 20 15 6 40

6 10 15 20 40

PROGRAM-3

AIM: Implementation of Insertion Sort

ALGORITHM:

Step1: take a list of values

Step2: compare the first two elements of a list if first element is greater than second interchange it else keep the list as it is.

Step3: now take three elements from the list and sort them as follows

Step4: repeat step 2 to 3 until the list is sorted

SOURCE CODE:

```
#include<stdio.h>
main()
{
int a[10],i,p,temp,n;
clear();
printf("\n enter the max no.of elements u want to sort \n");
scanf("%d",&n);
printf("\n enter the elements u want to sort \n");
for(i=1;i<=n;i++)
{
scanf("%d",&a[i]);
}
a[0]=100;
for(i=2;i<=n;i++)
temp=a[i];
p=i-1;
while(temp<a[p])
{
a[p+1]=a[p];
p=p-1;
}
```

```
a[p+1]=temp;
}
for(i=1;i<=n;i++)
{
printf("%d\t",a[i]);
} getch();}
```

Result:

Enter the max no.of elements u want to sort

5

Enter the elements u want to sort

10 20 15 6 40

6 10 15 20 40

PROGRAM-4

AIM: Implementation of Quick Sort

ALGORITHM:

Step1: take first a list of unsorted values

Step2: take first element as 'pivot'

Step3: keep the first element as 'pivot' and correct its position in the list

Step4: divide the list into two based on first element

Step5: combine the list

SOURCE CODE:

```
#include<stdio.h>
main()
{
int a[10],i,left,right,n;
int min,loc;
clear();
printf("\n enter the max no.of elements u want to sort \n");
scanf("%d",&n);
printf("\n enter the elements u want to sort \n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
left=0;
right=n-1;
quicksort(a,left,right);
display(a,n);
}
quicksort(int a[],int left,intright)
{
int temp,flag=1,i,j,p;
i=left;
```

```
j=right;
p=a[left];
if(right>left)
{
while(flag)
{
do
{
i++;
}
while(a[i]<p && i<=right);
while((a[i]>p) && j>left)
j--;
if(j<i)
flag=0;
else
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
temp=a[lest];
a[left]=a[j];
a[j]=temp;
quicksort[a,left,j-1];
quicksort[a,i,right];
}
}
display(int a[],int n)
{
int i;
for(i=0;i<n;i++)
```

```
{
printf("%d\t",a[i]);
}
getch();
}
```

Result:

enter the max no. of elements u want to sort

5

enter the elements u want to sort

10 20 15 6 40

6 10 15 20 40

PROGRAM-5

AIM: Implementation of Heap Sort

ALGORITHM:

Step1: arrange elements of a list in correct form of a binary tree

Step2: remove top most elements of the heap

Step3: re arrange the remaining elements from a heap this process is continued till we get sorted list

SOURCE CODE:

Program to implement Heap sort

```
#include<stdio.h>
main()
{
int a[10],i,j,n;
int min,loc;
clear();
printf("\n enter the max no.of elements u wanna sort \n");
scanf("%d",&n);
printf("\n enter the elements u want to sort \n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
heapsort(a,n);
display(a,n);
}
heapsort(inta[],int n)
{
int temp,i,key,q;
create heap(a,n);
for(q=n;q>2;q--)
{
```

```
temp=a[i];
a[i]=a[q];
a[q]=temp;
i=1;
key=a[1];
j=2;
if((j+1)<q)
if(a[j+1]>a[j])
j++;
while(j<=(q-1) && a[j]<key))
{
a[i]=a[j];
i=j;
j=2*i;
if((j+1)<q)
if(a[j+1]>a[j])
j++;
else
if(j>n)
j=n;
a[i]=key;
}
}}
```

Result:

enter the max no. of elements u wanna sort

5

enter the elements u want to sort

10 20 15 6 40

6 10 15 20 40

PROGRAM-6

AIM: Implementation of Merge Sort

ALGORITHM:

Merge Sort: Algorithm

```
Merge-Sort(A, p, r)
   if p < r then
      q←(p+r)/2
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
   Take the smallest of the two topmost elements of
sequences A[p..q] and A[q+1..r] and put into the
resulting sequence. Repeat this, until both sequences
are empty. Copy the resulting sequence into A[p..r].
```

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
int array[MAX];
void merge(int low, int mid, int high )
{
int temp[MAX];
int i = low;
int j = mid +1 ;
int k = low ;
while( (i <= mid) && (j <=high) )
{
if (array[i] <= array[ j])
```

```
temp[k++] = array[i++] ;
else
temp[k++] = array[ j++] ;
}
while( i <= mid )
temp[k++]=array[i++];
while( j <= high )
temp[k++]=array[j++];

for (i= low; i <= high ; i++)
array[i]=temp[i];
}
void merge_sort(int low, int high )
{
int mid;
if ( low != high )
{
mid = (low+high)/2;
merge_sort( low , mid );
merge_sort( mid+1, high );
merge(low, mid, high );
}
}
void main()
{
int i,n;
clrscr();
printf ("\nEnter the number of elements :");
scanf ("%d",&n);
for (i=0;i<n;i++)
{
```

```
printf ("\nEnter element %d :",i+1);
scanf ("%d",&array[i]);
}
printf ("\nUnsorted list is :\n");
for ( i = 0 ; i<n ; i++)
printf ("%d", array[i]);
merge_sort( 0, n-1);
printf ("\nSorted list is :\n");
for ( i = 0 ; i<n ; i++)
printf ("%d", array[i]);
getch();
}
```

PROGRAM-7

AIM: Write a program that uses functions to perform the following operations on Singly Linked List (i) Creation (ii) Insertion (iii) Deletion (iv) Traversal.

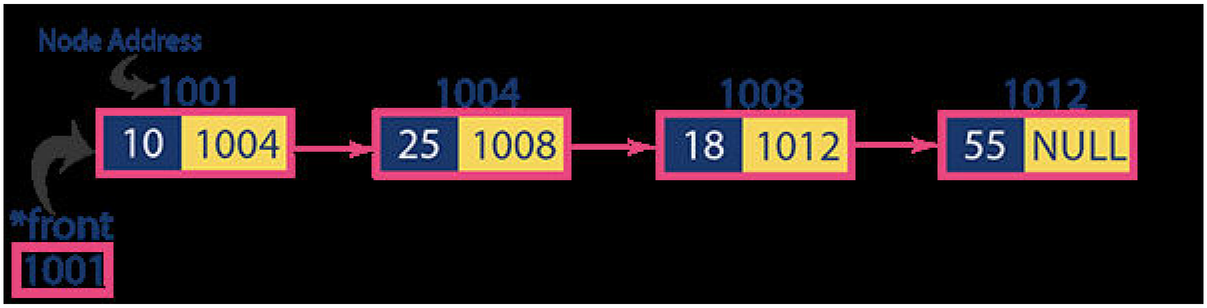
DESCRIPTION:

Linked List When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node". Linked List can be implemented as

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Single Linked List

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows... In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.



Operations on Single Linked List

The following operations are performed on a Single Linked List

1. Creation
2. Insertion
3. Deletion
4. Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

1. **Creation**

Step 1 - Define a Node structure with two members data and next
Step 2 - Define a Node pointer 'head' and set it to NULL.

2. **Insertion**

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

2.1 Inserting At Beginning of the list

2.2 Inserting At End of the list

2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.

Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL).

Step 3 - If it is Empty then, set head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Set temp → next = newNode.

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

3. Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...
3.1 Deleting from Beginning of the list
3.2 Deleting from End of the list
3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...
Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head. Step 4 - Check whether list is having only one node (temp → next == NULL)
Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions) Step 6 - If it is FALSE then set head = temp → next, and delete temp.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...
Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4 - Check whether list has only one Node (temp1 → next == NULL)
Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 →next == NULL)
Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list... Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).
Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

4. **Displaying a Single Linked List**

We can use the following steps to display the elements of a single linked list...
Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

SOURCE CODE:

```
#include<stdio.h>
#include<malloc.h>
void insertion_at_start();
void insertion_at_end();
void insertion_at_kth_position();
void deletion_at_start();
void deletion_at_end();
void deletion_at_kth_position();
void traversal();
struct node
{
	int info;
	struct node* next;
```

```
}*START=NULL,*Temp,*T1,*P;

int main()
{
	int choice,ch;
	do
	{
		printf("\n1.Insertion at beginning");
		printf("\n2.Insertion at the end");
		printf("\n3.Insertion at kth position");
		printf("\n4.Deletion at beginning");
		printf("\n5.Deletion at the end");
		printf("\n6.Deletion at kth position");
		printf("\n7.Traversing of Linked List");
		printf("\nEnter your choice:");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1: insertion_at_start(); break;
			case 2: insertion_at_end(); break;
			case 3: insertion_at_kth_position(); break;
			//case 4: deletion_at_start(); break;
			//case 5: deletion_at_end(); break;
			//case 6: deletion_at_kth_position(); break;
			case 7: traversal(); break;
			default: printf("\nEnter a valid choice");
		}
		printf("\nDo you want to continue(0/1):");
		scanf("%d",&ch);
	}while(ch);
return 0;
```

```
}

void insertion_at_start()
{
int data;
printf("\nEnter the data:");
scanf("%d",&data);
P=(struct node *)malloc(sizeof(struct node));
if(P==NULL)
{
printf("\nOVERFLOW");
return;
}
P->info=data;
P->next=NULL;
if(START==NULL)
{
	START=P;
	return;
}
else
{
	P->next=START;
	START=P;
}
}

void insertion_at_end()
{
	int data;
	printf("\nEnter the data:");
```

```
        scanf("%d",&data);
        P=(struct node *)malloc(sizeof(struct node));
        if(P==NULL)
        {
        printf("\nOVERFLOW");
        return;
        }
        P->info=data;
        P->next=NULL;
        if(START==NULL)
        {
                START=P;
                return;
        }
        if(START->next==NULL)
        {
                START->next=P;
                return;
        }
        T1=START;
        while(T1->next!=NULL)
        {
                T1=T1->next;
        }
        T1->next=P;
}

void insertion_at_kth_position()
{
        int data,LOC;
        printf("\nEnter the location:");
```

```
        scanf("%d",&LOC);
        printf("\nEnter the data:");
        scanf("%d",&data);
        P=(struct node*)malloc(sizeof(struct node));
        P->info=data;
        P->next=NULL;
        if(START==NULL)
        {
        START=P;
        return;
        }
        else if(LOC==1)
        {
                P->next=START;
                START=P;
        }
        else
        {
        Temp=START;
        while(LOC>2 && Temp->next!=NULL)
        {
                Temp=Temp->next;
                LOC=LOC-1;
        }
        P->next=Temp->next;
        Temp->next=P;
        }
}

void traversal()
{
```

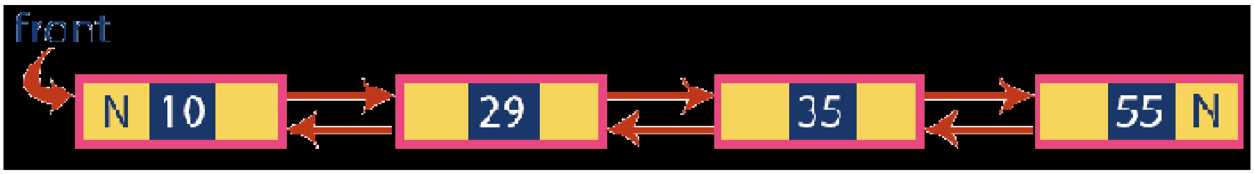
```
	Temp=START;
	printf("\nThe list is:\n");
	while(Temp!=NULL)
	{
		printf("%d\t",Temp->info);
		Temp=Temp->next;
	}
}
```

PROGRAM-8

AIM: Write a program that uses functions to perform the following operations on doubly linked List (i) Creation (ii) Insertion (iii) Deletion (iv) Traversal.

DESCRIPTION:

Double Linked List In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows... Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence. In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field.



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Display

1.Creation

Step 1 - Define a Node structure with two members data and next Step 2 - Define a Node pointer 'head' and set it to NULL.

2.Insertion

In a double linked list, the insertion operation can be performed in three ways as follows... 2.1 Inserting At Beginning of the list 2.2 Inserting At End of the list 2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list... Step 1 - Create a newNode with given value and newNode → previous as NULL. Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4 - If it is not Empty then, assign head to newNode → next and newNode to head.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty, then assign NULL to newNode → previous and newNode to head.

Step 4 - If it is not Empty, then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Assign newNode to temp → next and temp to newNode → previous

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.

Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.

Step 5 - Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step7- Assign temp1→next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → nextand newNode to temp2 → previous.

3. Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

3.1 Deleting from Beginning of the list

3.2 Deleting from End of the list

3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list is having only one node (temp → previous is equal to temp → next)
Step 5 - If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list has only one Node (temp → previous and temp → next both are NULL) Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7 - Assign NULL to temp → previous → next and delete temp.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5 - If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).

Step 8 - If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).

Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.

Step 10 - If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).

Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp(free(temp)).

Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp→next),temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp(free(temp)).

5. **Displaying**

We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4 - Display 'NULL <--- '.

Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node
Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

SOURCE CODE:

```
#include<stdio.h>
#include<malloc.h>
struct doubly
{
	int info;
	struct doubly *prev;
	struct doubly *next;
}*START=NULL,*LAST=NULL,*T1,*P,*Temp;
int main()
{
	int data,k,ch;
	do
	{
```

}

```
printf("\nEnter the locaation:");
scanf("%d",&k);
printf("\nEnter the data:");
scanf("%d",&data);
P=(struct doubly *)malloc(sizeof(struct doubly));
if(P==NULL)   //if the node successfully created or not?
{
        printf("\nOVERFLOW");
}
else
{
        P->info=data;
        P->next=NULL;
        P->prev=NULL;
}
if(START==NULL)
{
        START=P;
        LAST=P;
}
else
{
Temp=START;
while(k>2 && Temp->next!=NULL)
{
        Temp=Temp->next;
        k--;
}
if(Temp==LAST)
{
        P->prev=Temp;
        Temp->next=P;
        LAST=P;
}
else
{
        P->next=Temp->next;
        P->prev=Temp;
        Temp->next->prev=P;
        Temp->next=P;
}
```

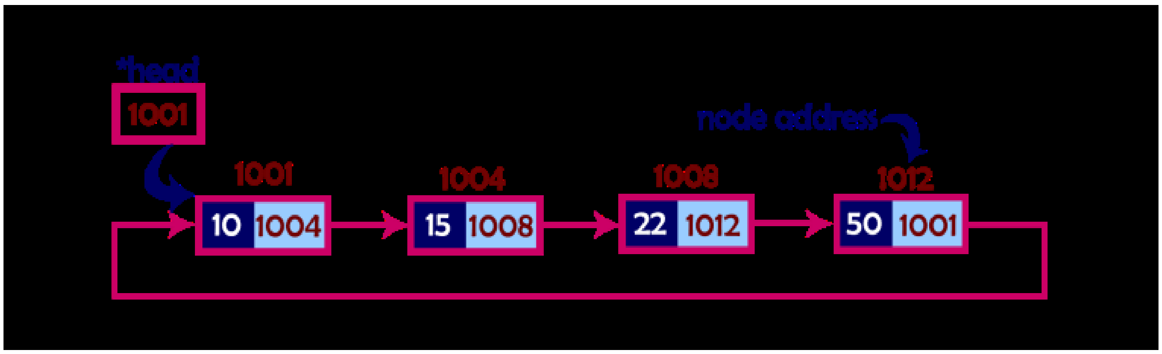
```
        }
        printf("\nDo you want to continue:");
        scanf("%d",&ch);
        }while(ch);
        Temp=START;
        printf("\nThe list is:\n");
        while(Temp!=NULL)
        {
                printf("%d\t",Temp->info);
                Temp=Temp->next;
                }
return 0;
}
```

PROGRAM-9

AIM: Write a program that uses functions to perform the following operations on circular linked List (i) Creation (ii) Insertion (iii) Deletion (iv) Traversal.

DESCRIPTION:

Circular Linked List In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list. A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element. That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.



Operations

In a circular linked list, we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Display

1. Creation

Step 1 - Define a Node structure with two members data and next Step 2 - Define a Node pointer 'head' and set it to NULL.

2.Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

2.1 Inserting At Beginning of the list
2.2 Inserting At End of the list
2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list... **Step 1 -** Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head == NULL**)
Step 3 - If it is **Empty** then, set **head = newNode** and **newNode→next = head** .
Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').
Step 6 - Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...
Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head == NULL**).
Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**).
Step 6 - Set **temp → next = newNode** and **newNode → next = head**.

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...
Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head == NULL**)
Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
Step 8 - If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.
Step 9 - If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

3. Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...
3.1 Deleting from Beginning of the list
3.2 Deleting from End of the list
3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...
Step 1 - Check whether list is **Empty** (**head == NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and

terminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
Step 4 - Check whether list is having only one node (**temp1 → next == head**)
Step 5 - If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)
Step 7 - Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...
Step 1 - Check whether list is **Empty** (**head == NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
Step 4 - Check whether list has only one Node (**temp1 → next == head**)
Step 5 - If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
Step 7 - Set **temp2 → next** = **head** and delete **temp1**.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...
Step 1 - Check whether list is **Empty** (**head == NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next== head**)
Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
Step 9 - If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
Step 1 1- If **temp1** is last node then set **temp2 → next = head** and delete **temp1** (**free(temp1)**).
Step 12 - If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).

4. Displaying a Circular Linked List
We can use the following steps to display the elements of a circular linked list...
Step 1 - Check whether list is **Empty** (**head == NULL**)
Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
Step 5 - Finally display **temp → data** with arrow pointing to **head → data**.

SOURCE CODE:

```
#include<iostream.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();
struct Node
{
int data;
struct Node *next;
}*head = NULL;
void main()
{
int choice1, choice2, value, location;
clrscr();
while(1)
{
cout<<"\n*********** MENU *************\n";
cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ";
cin>>choice1;
switch(choice1)
{
case 1: cout<<"Enter the value to be inserted: ";
cin>>value;
while(1)
{
```

```
cout<<"\nSelect from the following Inserting options\n";
cout<<"1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter
your choice: ";
cin>>choice2;
switch(choice2)
{
case 1: insertAtBeginning(value);
break;
case 2: insertAtEnd(value);
break;
case 3: cout<<"Enter the location after which you want to insert:";
cin>>location;
insertAfter(value,location);
break;
case 4: goto EndSwitch;
default: cout<<"\nPlease select correct Inserting option!!!\n";
}
}
case 2: while(1)
{
cout<<"\nSelect from the following Deleting options\n";
cout<<"1. At Beginning\n2. At End\n3. Specific Node\n4.
Cancel\nEnter your choice: ";
cin>>choice2;
switch(choice2)
{
case 1: deleteBeginning();
break;
case 2: deleteEnd();
break;
case 3: cout<<"Enter the Node value to be deleted: ";
cin>>location;
deleteSpecic(location);
break;
case 4: goto EndSwitch;
default: cout<<"\nPlease select correct Deleting option!!!\n";
}
}
EndSwitch: break;
case 3: display();
break;
case 4: exit(0);
default: cout<<"\nPlease select correct option!!!";
}
}
}
void insertAtBeginning(int value)
```

```
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
if(head == NULL)
{
head = newNode;
newNode -> next = head;
}
else
{
struct Node *temp = head;
while(temp -> next != head)
temp = temp -> next;
newNode -> next = head;
head = newNode;
temp -> next = head;
}
cout<<"\nInsertion success!!!";
}
void insertAtEnd(int value)
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
if(head == NULL)
{
head = newNode;
newNode -> next = head;
}
else
{
struct Node *temp = head;
while(temp -> next != head)
temp = temp -> next;
temp -> next = newNode;
newNode -> next = head;
}
cout<<"\nInsertion success!!!";
}
void insertAfter(int value, int location)
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
if(head == NULL)
{
```

```
head = newNode;
newNode -> next = head;
}
else
{
struct Node *temp = head;
while(temp -> data != location)
{
if(temp -> next == head)
{
cout<<"Given node is not found in the list!!!";
goto EndFunction;
}
else
{
temp = temp -> next;
}
}
newNode -> next = temp -> next;
temp -> next = newNode;
cout<<"\nInsertion success!!!";
}
EndFunction:
}
void deleteBeginning()
{
if(head == NULL)
cout<<"List is Empty!!! Deletion not possible!!!";
else
{
struct Node *temp = head;
if(temp -> next == head)
{
head = NULL;
free(temp);
}
else
{
head = head -> next;
free(temp);
}
cout<<"\nDeletion success!!!";
}
}
void deleteEnd()
{
if(head == NULL)
```

```
cout<<"List is Empty!!! Deletion not possible!!!";
else
{
struct Node *temp1 = head, temp2;
if(temp1 -> next == head)
{
head = NULL;
free(temp1);
}
else
{
while(temp1 -> next != head){
temp2 = temp1;
temp1 = temp1 -> next;
}
temp2 -> next = head;
free(temp1);
}
cout<<"\nDeletion success!!!";
}
}
void deleteSpecific(int delValue)
{
if(head == NULL)
cout<<"List is Empty!!! Deletion not possible!!!";
else
{
struct Node *temp1 = head, temp2;
while(temp1 -> data != delValue)
{
if(temp1 -> next == head)
{
cout<<"\nGiven node is not found in the list!!!";
goto FuctionEnd;
}
else
{
temp2 = temp1;
temp1 = temp1 -> next;
}
}
if(temp1 -> next == head)
{
head = NULL;
free(temp1);
}
else
```

```
{
if(temp1 == head)
{
temp2 = head;
while(temp2 -> next != head)
temp2 = temp2 -> next;
head = head -> next;
temp2 -> next = head;
free(temp1);
}
else
{
if(temp1 -> next == head)
{
temp2 -> next = head;
}
Else
{
temp2 -> next = temp1 -> next;
}
free(temp1);
}
}
cout<<"\nDeletion success!!!";
}
FuctionEnd:
}
void display()
{
if(head == NULL)
cout<<"\nList is Empty!!!";
else
{
struct Node *temp = head;
cout<<"\nList elements are: \n";
while(temp -> next != head)
{
cout<<temp -> data;
}
cout<< temp -> data, head -> data;
}
}
```

PROGRAM-10

AIM: Implementation of Linear Search

DESCRIPTION:

Linear Search begins by comparing the first element of the list with the target element. If it matches, the search ends. Otherwise, move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, conclude that target element is absent in the list.

Algorithm for Linear search

Linear_Search (A[ ], N, val , pos)
Step 1 : Set pos = -1 and k = 0
Step 2 : Repeat while k < N Begin
Step 3 : if A[k] = val
    Set pos = k
    print pos
    Goto step 5
    End while
Step 4 : print “Value is not present”
Step 5 : Exit

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char ch;
int arr[50],n,i,item;
clrscr();
printf("\nHow many elements you want to enter in the array:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter element %d:",i+1);
scanf("%d",&arr[i]);
}
printf("\n\nPress any key to continue.....");
getch();
do
{
clrscr();
printf("\nEnter the element to be searched:");
scanf("%d",&item);
for(i=0;i < n;i++)
{
```

```
if(item == arr[i])
{
printf("\n%d found at position %d\n",item,i+1);
break;
}
}
if (i == n)
printf("\nItem %d not found in array\n",item);
printf("\n\nPress (Y/y) to continue: ");
fflush(stdin);
scanf("%c",&ch);
}while(ch == 'Y'|| ch == 'y');
}
```

PROGRAM-11

AIM: Implementation of Binary Search

DESCRIPTION:

Before searching, the list of items should be sorted in ascending order. First compare the key value with the item in the mid position of the array.
If there is a match, we can return immediately the position. if the value is less than the element in middle location of the array, the required value is lie in the lower half of the array.
If the value is greater than the element in middle location of the array, the required value is lie in the upper half of the array. We repeat the above procedure on the lower half or upper half of the array.

ALGORITHM:

Binary_Search (A [ ], U_bound, VAL)
Step 1 : set BEG = 0 , END = U_bound , POS = -1
Step 2 : Repeat while (BEG <= END)
Step 3 :set MID = (BEG + END) / 2
POS = MID
print VAL " is available at ", POS
GoTo Step 6
End if
if A [MID] > VAL then
set END = MID – 1
Else
set BEG= MID + 1
End if
End while
Step 5 : if POS = -1 then
print VAL " is not present "
End if
Step 6 : EXIT

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char ch;
int arr[20],start,end,mid,n,i,data;
clrscr();
printf("\nHow many elements you want to enter in the array:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter element %d:",i+1);
```

```
scanf("%d",&arr[i]);
}
printf("\n\nPress any key to continue...");
getch();
do
{
clrscr();
printf("\nEnter the element to be searched:");
scanf("%d",&data);
start=0;
end=n-1;
mid=(start + end)/2;
while(data!=arr[mid] && start <=end)
{
if(data > arr[mid])
start=mid+1;
else
end=mid-1;
mid=(start+end)/2;
}
if(data==arr[mid])
printf("\n%d found at position %d\n",data,mid + 1);
if(start>end)
printf("\n%d not found in array\n",data);
printf("\n\n Press <Y or y> to continue:");
fflush(stdin);
scanf("%c",&ch);
}while(ch == 'Y' || ch == 'y');
}
```

PROGRAM-12

AIM: Implementation of Index Sequential Search

DESCRIPTION:

Index search is special search. This search method is used to search a record in a file. Searching a record refers to the searching of location **loc** in memory where the file is stored. Indexed search searches the record with a given key value relative to a primary key field. This search method is accomplished by the use of pointers.

Index helps to locate a particular record with less time. Indexed sequential files use the principal of index creation.

SOURCE CODE:

```
void indexedSequentialSearch(int arr[], int n, int k)
{
        int elements[20], indices[20], temp, i;
        int j = 0, ind = 0, start, end;
        for (i = 0; i < n; i += 3) {

                // Storing element
                elements[ind] = arr[i];

                // Storing the index
                indices[ind] = i;
                ind++;
        }
        if (k < elements[0]) {
                printf("Not found");
                exit(0);
        }
        else {
                for (i = 1; i <= ind; i++)
                        if (k < elements[i]) {
                                start = indices[i - 1];
                                end = indices[i];
                                break;
                        }
        }
        for (i = start; i <= end; i++) {
                if (k == arr[i]) {
                        j = 1;
                        break;
                }
        }
        if (j == 1)
                printf("Found at index %d", i);
        else
```

```
                    printf("Not found");
}

// Driver code
void main()
{

          int arr[] = { 6, 7, 8, 9, 10 };
          int n = sizeof(arr) / sizeof(arr[0]);

          // Element to search
          int k = 8;
          indexedSequentialSearch(arr, n, k);
}
```

PROGRAM-13

AIM: Implementation of Stack using Array

ALGORITHM:

1. push(s,top,x):

Step1: start

Step2:(check for stack overflow)

if(top>=max)

display "stack overflow"

return

Step3:[increment top pointer]

top++

Step4:[increment an element in thestack]

s[top] <- x

Step5:[finished]

return

2.pop(s,top)

Step1:(check for stack underflow)

if(top==0)

display() "stack underflow"

Step2:[decrement top operator]

top<- top-1

Step3:[delete an element from the stack]

return

(s[top+1])

SOURCE CODE:

Stack operations using arrays

```
#include<stdio.h>
#define max 10
void push();
void pop();
void display();
int s[max];
```

```
int top=0;
void main()
{
char ch;
int choice;
do
{
printf("enter choice of operation");
printf("1.push(),2.pop(),3.display()");
scanf("%d",&choice);
switch(choice)
{
case1:
push();
break;
case2:
pop();

break;
case3:
display();
break;
default:
printf("invalid option");
}
printf("do u wantto continue y/n");
fflush(stdin);
scanf("%c",&ch);
}
while(ch=='y'||ch=='y')
}
void push()
{
```

```
int item;
if(top>=max)
printf("stackisfull");
else
{
printf("enter any item");
scanf("%d",&item);
top++;
s[top]=item;
}
}
void pop()

{
int item;
if(top==0)
printf("stack is empty");

else
{
item=s[top];
printf("the related elemnt is %d",item);
top--;
}
}
void display()
{
int item;
int i;
if(top==0)
printf("\n  stack is empty no element isdisplayed");
else
{
```

```
printf("\n%d\n",s[i]);
printf("\n----\n");
}
}
```

PROGRAM-14

AIM: Implementation of Stack using Linked List

DESCRIPTION:

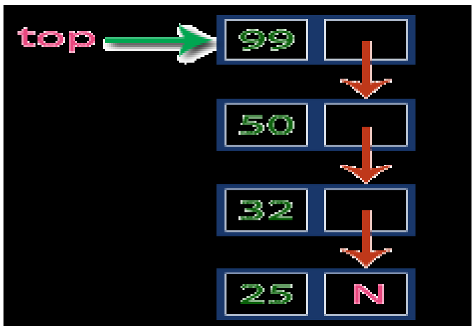
Stack Using Linked List

The major problem with the stack implemented using an arrays is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself.

Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure.

The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



1.Push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...
Step 1 - Create a **newNode** with given value.
Step 2 - Check whether stack is **Empty** (**top** == **NULL**)
Step 3 - If it is **Empty**, then set **newNode → next** = **NULL**.
Step 4 - If it is **Not Empty**, then set **newNode → next** = **top**.
Step 5 - Finally, set **top** = **newNode**.

2.Pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...
Step 1 - Check whether **stack** is **Empty** (**top == NULL**).
Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
Step 4 - Then set '**top = top → next**'.
Step 5 - Finally, delete '**temp**'. (**free(temp)**).

3.Display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...
Step 1 - Check whether stack is **Empty** (**top == NULL**).
Step 2 - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
Step 3 - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
Step 4 - Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next != NULL**).
Step 5 - Finally! Display '**temp → data ---> NULL**'.

SOURCE CODE:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
int info;
struct node *link;
};
struct node *top;
void main()
{
clrscr();
void create(),traverse (),push ( ),pop ();
create ();
printf("\nstack is:\n");
traverse ();
push ();
printf("\nAfter push the element the stack is:\n");
traverse ();
pop ();
printf("\nAfter pop the element the stack is:\n");
traverse ( );
```

```
getch ( );
}

void create ( )
{
struct node *ptr, *cpt;
char ch;
ptr = (struct node *) malloc (sizeof (struct node));
printf ("Input first info");
scanf("%d",&ptr ->info);
ptr->link = NULL;
do
{
cpt=(struct node *) malloc (sizeof (struct node));
printf("\nInput next information\n");
scanf ("%d", &cpt->info);
cpt->link= ptr;
ptr=cpt;
printf("Press <Y/N> for more information");
ch=getche();
}
while (ch=='Y');
top = ptr;
}
void traverse ( )
{
struct node *ptr;
printf ("Traversing of stack :\n");
ptr=top;
while (ptr !=NULL)
{
```

```
printf ("%d\n", ptr->info);
ptr=ptr->link;
}
}


void push ( )
{
struct node *ptr;
ptr = (struct node *) malloc (sizeof (struct node));
if (ptr==NULL)
{
printf("Overflow\n");
return;
}
printf ("Input New node information");
scanf ("%d", &ptr->info);
ptr->link=top;
top = ptr;
}
void pop ( )
{
struct node *ptr;
if (top==NULL)
{
printf ("Underflow \n");
return;
}
ptr=top;
top = ptr->link;
free (ptr);
```

```
}
```

Result:

enter choice of operation1.push(),2.pop(),3.display()1

enter any item3

do u wantto continue y/ny

enter choice of operation1.push(),2.pop(),3.display()1

enter any item4

do u wantto continue y/ny

enter choice of operation1.push(),2.pop(),3.display()3

15150

----

do u wantto continue y/n n

PROGRAM-15

AIM: Implementation of Queue using Array

DESCRIPTION:

Queue Using Arrays

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values.
The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1.
Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

We can Perform the following operations on Queue
1.enQueue()
2.deQueue()
3.Display()

1. **enQueue(value) - Inserting value into the queue**

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1 - Check whether **queue** is **FULL**. (**rear == SIZE-1**)
Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

2. **deQueue() - Deleting a value from the Queue**

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front = rear = -1**).

Display() - Displays the elements of a Queue
We can use the following steps to display the elements of a queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i = front+1**'.
Step 4 - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i <= rear**).

SOURCE CODE:

```
#include <stdio.h>
  #include <conio.h>
  #include <alloc.h>
  #include <stdlib.h>
  #define MAX 5

  int cqueue[MAX];
  int front = -1;
  int rear = -1;

  void main ( )
  {
  clrscr();
  void insert(),del(),display();
  int choice;
  while (1)
  {
  printf ("1.Insert\n");
  printf ("2.Delete\n");
  printf ("3.Display\n");
  printf ("4.Quit\n");
  printf ("Enter your choice :");
  scanf ("%d", &choice);
  switch(choice)
  {
  case 1 :
  insert( );
  break;
  case 2 :
  del( );
  break;
  case 3 :
  display( );
  break;
  case 4 :
```

```
exit(1);
default:
printf("Wrong choice\n");
}
}
}

void insert( )
{
int item;
if((front==0 && rear==MAX-1) || (front==rear+1))
{
printf("Queue is Overflow\n");
return;
}
if (front==-1)/*If queue is empty*/
{
front = 0;
rear = 0;
}
else
if (rear==MAX-1) /*rear is at last position of queue*/
rear = 0;
else
rear = rear + 1;
printf("Input the element for insertion :");
scanf("%d", &item);
cqueue[rear] = item;
}

void del( )
{
if (front == -1)
{
printf("Queue Underflow\n");
return;
}
printf ("Deleted element from queue is : %d\n", cqueue[front]);
if(front == rear)
/* queue has only one element */
{
front = -1;
rear = -1;
}
else
if(front==MAX-1)
front = 0;
```

```
else
front = front + 1;
}

void display( )
{
int front_pos = front, rear_pos = rear;
if(front == -1)
{
printf("Queue is empty\n");
return;
}
printf ("Queue elements are:\n");
if(front_pos <= rear_pos)
while(front_pos <= rear_pos)
{
printf(" %d\n", cqueue[front_pos]);
front_pos++;
}
else
{
while(front_pos <= MAX-1)
{
printf(" %d\n", cqueue[front_pos]);
front_pos++;
}
front_pos = 0;
while(front_pos <= rear_pos)
{
printf(" %d\n", cqueue[front_pos]);
front_pos++;
}
}
printf("\n");
}
```

PROGRAM-16

AIM: Implementation of Queue using Linked List

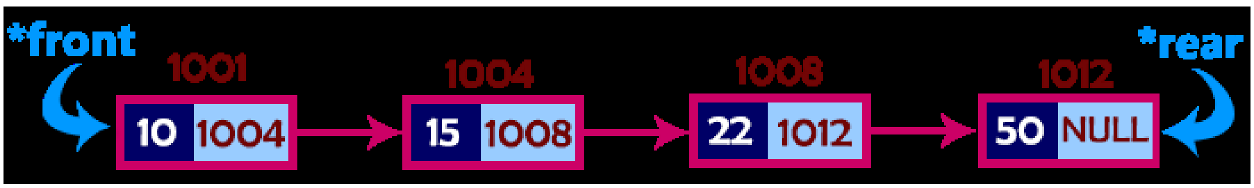
DESCRIPTION:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).

The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



Queue Operations using Array

We can Perform the following operations on Queue

1.enQueue()

2.deQueue()

3.Display()

To implement queue using linked list, we need to set the following things before implementing actual operations.

1. **enQueue(value) - Inserting an element into the Queue**

We can use the following steps to insert a new node into the queue...

Step 1 - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
Step 2 - Check whether queue is **Empty** (**rear == NULL**)
Step 3 - If it is **Empty** then, set **front = newNode** and **rear = newNode**.
Step 4 - If it is **Not Empty** then, set **rear → next = newNode** and **rear = newNode**.

2. **deQueue() - Deleting an Element from Queue**

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).
Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
Step 4 - Then set '**front = front → next**' and delete '**temp**' (**free(temp)**).

3. **Display() - Displaying the elements of Queue**

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty** (**front == NULL**).
Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
Step 4 - Display '**temp → data --->**' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next != NULL**).
Step 5 - Finally! Display '**temp → data ---> NULL**'.

Source Code:

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

struct node
{
int info;
struct node *link;
};
struct node *front, *rear;

void main ( )
{
        clrscr();
        void insert(),delet(),display();
        int ch;
        while (1)
        {
        printf ("1. Insert\n");
        printf ("2. Delete\n");
        printf ("3. Display\n");
        printf ("4. Exit\n");
        printf ("Enter your choice :");
        scanf ("%d", &ch);
        switch (ch)
        {
        case 1 :
        insert ( );
        break;
        case 2 :
        delet( );
        break;
        case 3 :
        display ( );
        break;
```

```
        case 4 :
        exit(0);
        default:
        printf ("Please enter correct choice \n");
        }
        }
        getch();
    }

void insert ( )
{
struct node *ptr;
ptr = (struct node*) malloc (sizeof (struct node));
int item;
printf ("Input the element for inserting :\n");
scanf ("%d", &item);
ptr-> info = item;
ptr->link = NULL;
if (front==NULL) /* queue is empty*/
front = ptr;
else
rear->link = ptr;
rear = ptr;
}

void delet( )
{
struct node *ptr;
if (front==NULL)
{
printf ("Queue is underflow \n");
return;
}
if (front==rear)
{
free (front);
rear = NULL;
}
else
{
ptr = front;
front = ptr->link;
free (ptr);
}
}

void display ( )
```

```
{
struct node *ptr;
ptr = front;
if (front==NULL)
printf("Queue is empty\n");
else
{
printf("\nElements in the Queue are:\n");
while(ptr!=NULL)
{
printf(" %d\n",ptr->info);
ptr=ptr->link;
}
printf("\n");
}
}
```

PROGRAM-17

AIM: Implementation of Tree Structures, Binary Search Tree and its operations.

DESCRIPTION:

Binary Search Tree:

So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged.

That means values at **left sub-tree < root node value < right sub-tree values**.

SOURCE CODE:

```
#include <conio.h>
#include <stdio.h>
#include <alloc.h>
typedef struct bst
{
int data;
struct bst *left, *right;
} node;

void insert(node *, node*);
void inorder(node *);
node *search(node *, int, node **);
void del(node *, int);

void main ( )
{
int ch;
char ans = 'N';
int key;
node *New, *root, *temp, *parent;
node *get_node( );
root=NULL;
clrscr ( );
printf("\n \t Program for Binary Search Tree");
do
{
printf("\n 1.Create \n 2.Search \n 3. Delete \n 4. Display");
printf("\n\n Enter your choice");
scanf("%d",&ch);
switch(ch)
{
case 1 : do
{
New=get_node( );
```

```
printf("\n Enter the Element");
scanf("%d",&New->data);
if(root==NULL)
root=New;
else
insert(root,New);
printf("\n Do u want to conitnue enter more elements(y/n)");
ans=getch( );
} while(ans=='N');
break;
case 2 :
printf("\n Enter the element which u want to search");
scanf("%d",&key);
temp=search(root,key,&parent);
printf("\n Parent of node %d is %d", temp->data,parent->data);
break;
case 3 : printf("\n Enter the Element u wish to delete");
scanf("%d",&key);
del(root,key);
break;
case 4 : if(root==NULL)
printf("Tree is not created");
else
{
printf("\n The tree is :");
inorder(root);
}
break;
}
} while (ch !=5);
}
node *get_node( )
{
node *temp;
temp=(node*)malloc(sizeof(node));
temp->left=NULL;
temp->right=NULL;
return temp;
}
void insert(node *root,node *New)
{
if(New->data<root->data)
{
if(root->left==NULL)
root->left=New;
else
insert(root->left,New);
```

```
}
if(New->data>root->data)
{
if(root->right==NULL)
root->right=New;
else
insert(root->right,New);
}
}
node *search(node *root,int key, node **parent)
{
node *temp;
temp=root;
while(temp !=NULL)
{
if(temp->data==key)
{
printf("\n The %d Element is present",temp->data);
return temp;
}
*parent=temp;
if(temp->data>key)
temp=temp->left;
else
temp=temp->right;
}
return NULL;
}
void del(node*root,int key)
{
node*temp,*parent,*temp_succ;
temp=search(root,key,&parent);
if(temp->left!=NULL&&temp->right!=NULL)
{
parent=temp;
temp_succ=temp->right;
while(temp_succ->left!=NULL)
{
parent=temp_succ;
temp_succ=temp_succ->left;
}
temp->data=temp_succ->data;
parent->right=NULL;
printf("\n Now deleted it!");
return;
}
if(temp->left!=NULL&&temp->right==NULL)
```

```
{
if(parent->left==temp)
parent->left=temp->left;
else
parent->right=temp->left;
temp=NULL;
free(temp);
printf("\n Now deleted it!");
return;
}
if(temp->left==NULL&&temp->right!=NULL)
{
if(parent->left==temp)
parent->left=temp->right;
else
parent->right=temp->right;
temp=NULL;
free(temp);
printf("\n Now deleted it!");
return;
}
if(temp->left==NULL &&temp->right==NULL)
{
if(parent->left==temp)
parent->right=NULL;
else
parent->right=NULL;
printf("\n Now Deleted it!");
return;
}
}
void inorder(node *temp)
{
if(temp !=NULL)
{
inorder(temp->left);
printf("%d", temp->data);
inorder(temp->right);
}
}
```

PROGRAM-18

AIM: Implementation of graph traversal algorithms: BFS, DFS

Algorithm-BFS

Step 1: SET STATUS = 1 (ready state)
for each node in G

Step 2: Enqueue the starting node A
and set its STATUS = 2
(waiting state)

Step 3: Repeat Steps 4 and 5 until
QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).

Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]

Step 6: EXIT

Algorithm-DFS

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT

SOURCE CODE: DFS

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

typedef enum boolean{false,true} bool;
int adj[MAX][MAX];
bool visited[MAX];
int n;
void main()
{
	int i,v,choice;
	clrscr();

	printf("\n\tTraversing of a graph through DFS\n");
	printf("\t*********************************\n\n");
	create_graph();
	while(1)
	{
		printf("\n");
		printf("1. Adjacency matrix\n");
		printf("2. Depth First Search using stack\n");
		printf("3. Depth First Search through recursion\n");
		printf("4. Number of components\n");
		printf("5. Exit\n");
		printf("Enter your choice : ");
		scanf("%d",&choice);

		switch(choice)
		{
		 case 1:
			printf("Adjacency Matrix is \n");
			display();
			break;

		 case 2:
			printf("Enter starting node:  ");
			scanf("%d",&v);
			for(i=1;i<=n;i++)
				visited[i]=false;
			dfs(v);
			break;

		 case 3:
			printf("Enter starting node:  ");
```

```
                scanf("%d",&v);
                for(i=1;i<=n;i++)
                    visited[i]=false;
                dfs_rec(v);
                break;

            case 4:
                 printf("Components are ");
                 components();
              break;

            case 5:
                exit(1);

            default:
                printf("Enter correct choice\n");
                break;
            }
    }
}

create_graph()
{
    int i,max_edges,origin,dest;

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

    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d( 0 0 to quit ) : ",i);
        scanf("%d %d",&origin,&dest);

        if((origin==0) && (dest==0))
            break;

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

```
return 0;
}

display()
{
	int i,j;
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
			printf("%4d",adj[i][j]);
		printf("\n");
	}
return 0;
}


dfs_rec(int v)
{
	int i;
	visited[v]=true;
	printf("%d ",v);
	for(i=1;i<=n;i++)
		if(adj[v][i]==1 && visited[i]==false)
			dfs_rec(i);
return 0;
}

dfs(int v)
{
	int i,stack[MAX],top=-1,pop_v,j,t;
	int  ch;

	top++;
	stack[top]=v;
	while (top>=0)
	{
		pop_v=stack[top];
		top--;			/*pop from stack*/
		if( visited[pop_v]==false)
		{
			printf("%d ",pop_v);
			visited[pop_v]=true;
		}
		else
			continue;

		for(i=n;i>=1;i--)
```

```
			{
				if( adj[pop_v][i]==1 && visited[i]==false)
				{
					top++;	/* push all unvisited neighbours of pop_v */
					stack[top]=i;
				}
			}
		}
return 0;
}

components()
{
	int i;
	for(i=1;i<=n;i++)
		visited[i]=false;
	for(i=1;i<=n;i++)
	{
		if(visited[i]==false)
			dfs_rec(i);
	}
	printf("\n");
return 0;
}
```

SOURCE CODE: BFS

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

typedef enum boolean{false,true} bool;
int adj[MAX][MAX];
bool visited[MAX];
int n;
void main()
{
	int i,v,choice;
	clrscr();

	printf("\n\tTraversing of a graph through BFS\n");
	printf("\t**********************************\n\n");
	create_graph();
	while(1)
	{
		printf("\n");
		printf("1. Adjacency matrix\n");
```

```
		printf("2. Breadth First Search\n");
		printf("3. Adjacent vertices\n");
		printf("4. Exit\n");
		printf("Enter your choice : ");
		scanf("%d",&choice);

		switch(choice)
		{
		 case 1:
			printf("Adjacency Matrix\n");
			display();
			break;

		 case 2:
			printf("Enter starting node for Breadth First Search : ");
			scanf("%d", &v);
			for(i=1;i<=n;i++)
				visited[i]=false;
			bfs(v);
			break;
		 case 3:
			printf("Enter node to find adjacent vertices : ");
			scanf("%d", &v);
			printf("Adjacent Vertices are : ");
			adj_nodes(v);
			break;
		 case 4:
			exit(1);
		 default:
			printf("Wrong choice\n");
			break;
		 }
	}
}

create_graph()
{
	int i,max_edges,origin,dest;

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

	for(i=1;i<=max_edges;i++)
	{
		printf("Enter edge %d( 0 0 to quit ) : ",i);
		scanf("%d %d",&origin,&dest);
```

```
		if((origin==0) && (dest==0))
			break;

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

display()
{
	int i,j;
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
			printf("%4d",adj[i][j]);
		printf("\n");
	}
return 0;
}


bfs(int v)
{
	int i,front,rear;
	int que[20];
	front=rear= -1;

	printf("%d ",v);
	visited[v]=true;
	rear++;
	front++;
	que[rear]=v;

	while(front<=rear)
	{
		v=que[front];	/* delete from queue */
		front++;
		for(i=1;i<=n;i++)
```

```
			{
				/* Check for adjacent unvisited nodes */
				if( adj[v][i]==1 && visited[i]==false)
				{
					printf("%d ",i);
					visited[i]=true;
					rear++;
					que[rear]=i;
				 }
			}
		}
return 0;
}

adj_nodes(int v)
{
	int i;
	for(i=1;i<=n;i++)
	if(adj[v][i]==1)
		printf("%d ",i);
	printf("\n");
return 0;
}
```

PROGRAM-19

AIM: Implementation of Minimum cost spanning tree: Kruskal's Algorithm

ALGORITHM:

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge

Step 6: END

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

struct edge
{
	int u;
	int v;
	int weight;
	struct edge *link;
}*front = NULL;

int parent[MAX]; /*Holds parent of each node */
struct edge tree[MAX]; /* Will contain the edges of spanning tree */
int n;   /*Denotes total number of nodes in the graph */
int wt=0; /*Weight of the spanning tree */
int count=0;    /* Denotes number of edges included in the tree */


void make_tree();
void insert_tree(int i,int j,int wt);
void insert_pque(int i,int j,int wt);
struct edge *del_pque();
```

```
void main()
{

	int i;
	clrscr();
	printf("\n\t MST from Kruskal's algorithm \n");
	printf("\t*************************\n\n");
	create_graph();
	make_tree();
	printf("\nEdges to be included in spanning tree are :\n");
	for(i=1;i<=count;i++)
	{
		printf("%d->",tree[i].u);
		printf("%d\n",tree[i].v);
	}
	printf("\n Weight of this MST is : %d\n", wt);
	getch();
}

create_graph()
{
	int i,wt,max_edges,origin,dest;

	printf("Enter number of nodes : ");
	scanf("%d",&n);
	max_edges=(n*(n-1))/2;
	for(i=1;i<=max_edges;i++)
	{
		printf("Enter edge %d(0 0 to quit): ",i);
		scanf("%d %d",&origin,&dest);
		if( (origin==0) && (dest==0) )
			break;
		printf("Enter weight for this edge : ");
		scanf("%d",&wt);
		if( origin > n || dest > n || origin<=0 || dest<=0)
		{
			printf("Invalid edge!\n");
			i--;
		}
		else
			insert_pque(origin,dest,wt);
	}
	if(i<n-1)
	{
		printf("Spanning tree is not possible\n");
		exit(1);
```

```
            }
return 0;
}

void make_tree()
{
        struct edge *tmp;
        int node1,node2,root_n1,root_n2;

        while( count < n-1) /*Loop till n-1 edges included in the tree*/
        {
                tmp=del_pque();
                node1=tmp->u;
                node2=tmp->v;

                printf("n1=%d  ",node1);
                printf("n2=%d  ",node2);

                while( node1 > 0)
                {
                        root_n1=node1;
                        node1=parent[node1];
                }
                while( node2 >0 )
                {
                        root_n2=node2;
                        node2=parent[node2];
                }
                printf("rootn1=%d  ",root_n1);
                printf("rootn2=%d\n",root_n2);

                if(root_n1!=root_n2)
                {
                    insert_tree(tmp->u,tmp->v,tmp->weight);
                    wt=wt+tmp->weight;
                    parent[root_n2]=root_n1;
                }
        }
}

void insert_tree(int i,int j,int wt)
{
        printf("This edge inserted in the spanning tree\n");
        count++;
        tree[count].u=i;
        tree[count].v=j;
        tree[count].weight=wt;
```

```
}


void insert_pque(int i,int j,int wt)
{
	struct edge *tmp,*q;

	tmp = (struct edge *)malloc(sizeof(struct edge));
	tmp->u=i;
	tmp->v=j;
	tmp->weight = wt;

	/*Queue is empty or edge to be added has weight less than first edge*/
	if( front == NULL || tmp->weight < front->weight )
	{
		tmp->link = front;
		front = tmp;
	}
	else
	{
		q = front;
		while( q->link != NULL && q->link->weight <= tmp->weight )
			q=q->link;
		tmp->link = q->link;
		q->link = tmp;
		if(q->link == NULL)
			tmp->link = NULL;
	}
}


struct edge *del_pque()
{
	struct edge *tmp;
	tmp = front;
	printf("\nEdge selected is %d->%d and weight is %d\n",tmp->u,tmp->v,tmp->weight);
	front = front->link;
	return tmp;
}
```

PROGRAM-20

AIM: Implementation of Minimum cost spanning tree: Prim’s Algorithm

ALGORITHM:

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the minimum spanning tree T [END OF LOOP]

Step 5: EXIT

SOURCE CODE:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
// Number of vertices in the graph
#define V 5

// 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 program to test above function
int main()
{
	/* Let us create the following graph
		2 3
	(0)--(1)--(2)
	| / \ |
	6| 8/ \5 |7
	| /	 \ |
	(3)-------(4)
			9			 */
	int graph[V][V] = { { 0, 2, 0, 6, 0 },
						{ 2, 0, 3, 8, 5 },
						{ 0, 3, 0, 0, 7 },
						{ 6, 8, 0, 0, 9 },
						{ 0, 5, 7, 9, 0 } };

	// Print the solution
	primMST(graph);

	return 0;
}
```

PROGRAM-21

AIM: Implementation of shortest path algorithm: Dijkstra's Algorithm

Dijkstra's Algorithm (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)

SOURCE CODE:

```
#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);
	}
}
```

PROGRAM-22

AIM: Implementation of shortest path algorithm: Warshal's Algorithm

ALGORITHM: FLOYD - WARSHALL (W)

1. n ← rows [W].
2. $D^0 \leftarrow W$
3. for k ← 1 to n
4. do for i ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. return $D^{(n)}$

SOURCE CODE:

```
#include <stdio.h>

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
```

```
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}
int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
             {2, 0, INF, 4},
             {INF, 1, 0, INF},
             {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

PROGRAM-23

AIM: Implementation of Hashing Algorithms

ALGORITHM:

HASH-INSERT (T, k)

1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] = NIL
4. then T [j] ← k
5. return j
6. else ← i= i +1
7. until i=m
8. error "hash table overflow"

HASH-SEARCH.T (k)

1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] =j
4. then return j
5. i ← i+1
6. until T [j] = NIL or i=m
7. return NIL

SOURCE CODE:

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

int a[MAX]={0};
void lp(int key);
void lpsr(int key);
void lpdel();
void display();

int main()
{
int i, key,ch ;

  do{
```

```
        printf("\n\n Program for insertion/searching keys with linear probing ");
        printf("\n*****************************************************************\n\n"
);
        printf("\n 1. Insert keys ");
        printf("\n 2. Search key ");
        printf("\n 3. Display keys ");
        printf("\n 4. Delete ");
        printf("\n 5. Exit ");
        printf("\n Select operation ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: do{
                        printf("\n Enter key value [type -1 for termination] ");
                        scanf("%d",&key);
                        if (key != -1)
                           lp(key);
                        }while(key!=-1);
                        display();
                        break;
        case 2: printf("\n Enter search key value ");
                        scanf("%d",&key);
                        lpsr(key);
                        break;
        case 3: display();
                break;
   case 4: lpdel();
        display();
      break;
        }
        }while(ch!=5);
return 0;
}

/* function lp find a location for key and insert it */
void lp(int key)
{
 int loc;
 loc = key % MAX;
 while (a[loc]>0)
            loc = ++loc % MAX;
 a[loc] = key;
}

/* function lpsr find a location for a key  */
void lpsr(int key)
{
```

```
 int loc;
 loc = key % MAX;
 while ((a[loc] != key) && (a[loc] !=0))
             loc=++loc%MAX;
 if (a[loc] != 0)
           printf("\n Search successful at index %d",loc);
 else
           printf("\n Search unsuccessful ");
}

void display()
{
int i;
printf("\n List of keys ('0' indicate that the location is empty): \n");
for (i=0;i<MAX;i++)
           printf(" %d ",a[i]);
}


void lpdel()
{
    int key,loc;
    printf("\nEnter the key to be deleted:");
    scanf("%d",&key);
    loc = key % MAX;
    while ((a[loc] != key) && (a[loc] !=0))
        loc=++loc%MAX;
    if (a[loc] != 0)
            {
        printf("\n %d is present at index %d",key,loc);
        a[loc]=-1;
            }

 else
           printf("\n %d is not present in the table ",key);
}
```

PROGRAM-24

AIM: Implementation of Collision resolution techniques in Hashing

ALGORITH:

Collision Resolution by Chaining:

In chaining, we place all the elements that hash to the same slot into the same linked list, As fig shows that Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

Open Addressing Techniques

Three techniques are commonly used to compute the probe sequence required for open addressing:

1. Linear Probing: $h(k, i) = (h'(k) + i) \bmod m$
2. Quadratic Probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
3. Double Hashing: $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

SOURCE CODE: Double Hashing

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int a[MAX];
void dh(int , int[]);
void dhsr(int key, int a[MAX]);
void display(int a[MAX]);
void main()
{
int i, key,ch ;
for(i=0;i<MAX;i++)
	a[i] = '\0';
do{
	printf("\n\n Program for insertion/searching keys with double hashing ");
	printf("\n*************************************************************\n\n"
);
	printf("\n 1. Insert keys ");
	printf("\n 2. Search key ");
	printf("\n 3. Display keys ");
	printf("\n 4. Exit ");
	printf("\n Select operation ");
	scanf("%d",&ch);
	switch(ch)
	{
```

```
        case 1: do{
                        printf("\n Enter key value [type -1 for termination] ");
                        scanf("%d",&key);
                        if (key != -1)

                                dh(key,a);
                        }while(key!=-1);
                        break;
        case 2: printf("\n Enter search key value ");
                        scanf("%d",&key);
                        dhsr(key,h);
                        break;
        case 3: display(a);

                    break;
        }
        }while(ch!=4);
}

/* Find the location for a key and insert it */
void dh(int key, int a[MAX])
{
 int loc, i=0;
 loc = key % MAX;
 while (a[loc] !='\0')
         loc = (loc + ++i*(key % (MAX -1))) % MAX;
 a[loc] = key;
 display(a);
}

/* find the location for a key */
void dhsr(int key, int a[MAX])
{
 int loc,i=0;
 loc = key % MAX;
 while ((a[loc] != key) && (a[loc] !='\0'))
         loc = (loc + ++i*(key % (MAX -1))) % MAX;
 if (a[loc] != '\0')
        printf("\n Search successful at index %d",loc);
 else
        printf("\n Search unsuccessful ");
}

void display(int a[MAX])
{
int i;
printf("\n List of keys, 0 indicate that the location is empty \n");
```

```
for (i=0;i<MAX;i++)
        printf(" %d ",a[i]);
}
```

SOURCE CODE: Quadratic Probing

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

int a[MAX];
void qp(int , int[]);
void qpsr(int key, int a[MAX]);
void display(int a[MAX]);

void main()
{
int i,key,ch ;
clrscr();
for(i=0;i<MAX;i++)
        a[i] = '\0';
  do{
        printf("\n\n Program for insertion/searching keys with quardratic probing ");
        printf("\n************************************************************\n\n"
);
        printf("\n 1. Insert keys ");
        printf("\n 2. Search key ");
        printf("\n 3. Display keys ");
        printf("\n 4. Exit ");
        printf("\n Select operation ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: do{
                        printf("\n Enter key value [type -1 for termination] ");
                        scanf("%d",&key);
                        if (key != -1)
                           qp(key,a);
                        }while(key!=-1);
                        display(a);
                        break;
        case 2: printf("\n Enter search key value ");
                        scanf("%d",&key);
                        qpsr(key,a);
                        break;
        case 3: display(a);
                break;
```

```
            }
            }while(ch!=4);
}

void qp(int key, int a[MAX])
{
 int loc, i=1;
 loc = key % MAX;
 while (a[loc] !='\0')
            {
             loc = (key % MAX + i*i) % MAX;
             i++;
             }
 a[loc] = key;
}

/* Find a location for a key  */
void qpsr(int key, int a[MAX])
{
 int loc;
 loc = key % MAX;
 while ((a[loc] != key) && (a[loc] !=-1))
             loc = ++loc % MAX;
 if (a[loc] != '\0')
            printf("\n Search successful at index %d",loc);
 else
            printf("\n Search unsuccessful ");
}

void display(int a[MAX])
{
int i;
printf("\n List of keys ('0' indicate that the location is empty): \n");
for (i=0;i<MAX;i++)
            printf(" %d ",a[i]);
}
```