DATA STRUCTURES

Data Structures:

The logical or mathematical model of a particular organization of data is called data structures. Data structures is the study of logical relationship existing between individual data elements, the way the data is organized in the memory and the efficient way of storing, accessing and manipulating the data elements.

Choice of a particular data model depends on two considerations: it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

Data Structures can be classified as:
- Primitive data structures
- Non-Primitive data structures.

Primitive data structures are the basic data structures that can be directly manipulated/operated by machine instructions. Some of these are character, integer, real, pointers etc.

Non-primitive data structures are derived from primitive data structures, they cannot be directly manipulated/operated by machine instructions, and these are group of homogeneous or heterogeneous data items. Some of these are Arrays, stacks, queues, trees, graphs etc.

Data structures are also classified as
- Linear data structures
- Non-Linear data structures.

In the Linear data structures processing of data items is possible in linear fashion, i.e., data can be processed one by one sequentially.
Example of such data structures are:
- Array
- Linked list
- Stacks
- Queues

A data structure in which insertion and deletion is not possible in a linear fashion is called as non linear data structure. i.e., which does not show the relationship of logical adjacency between the elements is called as non-linear data structure. Such as trees, graphs and files.

Data structure operations:

The particular data structures that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following operations play major role in the processing of data.
- i) Traversing.
- ii) Searching.
- iii) Inserting.
- iv) Deleting.
- v) Sorting.
- vi) Merging

STACKS:

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at the same end, called the TOP of the stack. A stack is a non-primitive linear data structure. As all the insertion and deletion are done from the same end, the first element inserted into the stack is the last element deleted from the stack and the last element inserted into the stack is the first element to be deleted. Therefore, the stack is called Last-In First-Out (LIFO) data structure.

QUEUES:

A queue is a non-primitive linear data structure. Where the operation on the queue is based on First-In-First-Out FIFO process — the first element in the queue will be the first one out. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be removed.

For inserting elements into the queue are done from the rear end and deletion is done from the front end, we use external pointers called as rear and front to keep track of the status of the queue. During insertion, Queue Overflow condition has to be checked. Likewise during deletion, Queue Underflow condition is checked.

LINKED LIST:

Disadvantages of static/sequential allocation technique:

1) If an item has to be deleted then all the following items will have to be moved by one allocation. Wastage of time.
2) Inefficient memory utilization.
3) If no consecutive memory (free) is available, execution is not possible.

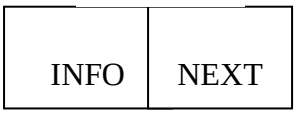
Linear Linked Lists
Types of Linked lists:
1) Single Linked lists
2) Circular Single Linked Lists
3) Double Linked Lists
4) Circular Double Linked Lists.

NODE:

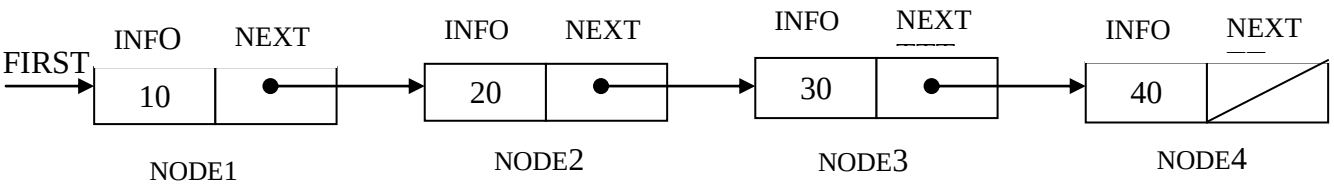
Each node consists of two fields. Information (info) field and next address(next) field. The info field consists of actual information/data/item that has to be stored in a list. The second field next/link contains the address of the next node. Since next field contains the address, It is of type pointer.Here the nodes in the list are logically adjacent to each other. Nodes that are physically adjacent need not be logically adjacent in the list.

The entire linked list is accessed from an external pointer FIRST that points to (contains the address of) the first node in the list. (By an "external" pointer, we mean, one that is not included within a node. Rather its value can be accessed directly by referencing a variable).

NODE



The list containing 4 items/data 10, 20, 30 and 40 is shown below.



The nodes in the list can be accessed using a pointer variable. In the above fig. FIRST is the pointer having the address of the first node of the list, initially before creating the list, as list is empty. The FIRST will always be initialized to NULL in the beginning. Once the list is created, FIRST contains the address of the first node of the list.
As each node is having only one link/next, the list is called single linked list and all the nodes are linked in one direction.

Each node can be accessed by the pointer pointing (holding the address) to that node, Say P is pointer to a particular node, then the information field of that node can be accessed using info(P) and the next field can be accessed using next(P).
The arrows coming out of the next field in the fig. indicates that the address of the succeeding node is stored in that field.
The link field of last node contains a special value known as NULL which is shown using a diagonal line pictorially. This NULL pointer is used to signal the end of a list.

The basic operations of linked lists are Insertion, Deletion and Display. A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and deleted(removed). The dynamic nature of list may be contrasted with the static nature of an array, whose size remains constant. When an item has to inserted, we will have to create a node, which has to be got from the available free memory of the computer system, So we shall use a mechanism to find an unused node which makes it available to us. For this purpose we shall use the getnode operation (getnode() function).

The C language provides the built-in functions like malloc(), calloc(), realloc() and free(), which are stored in alloc.h or stdlib.h header files. To dynamically allocate and release the memory locations from/to the computer system.

TREES:

Definition: A data structure which is accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom. A tree can also be defined as a connected, acyclic di-graph.

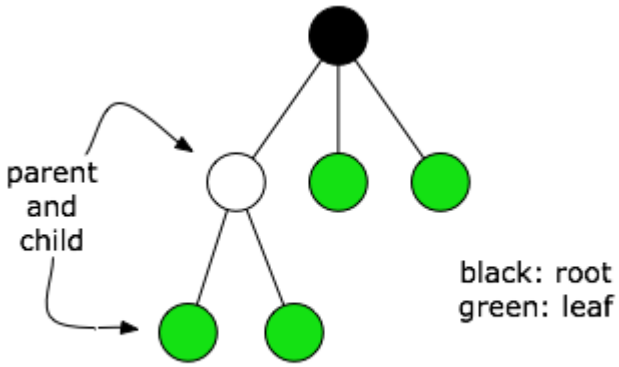


Figure: tree data structure

Binary tree: A tree with utmost two children for each node.
Complete binary tree: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.
Binary search tree: A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.
Tree traversal is a technique for processing the nodesof a tree in some order.
The different tree traversal techniques are Pre-order, In-order and Post-order traversal.
In Pre-order traversal, the tree node is visited first and the left subtree is traversed recursively and later right sub-tree is traversed recursively.

Graph:

Definition: A Graph G consists of two sets, V and E. V is a finite set of vertices. E is a set of edges or pair of vertices.

Two types of graphs,

Undirected Graph
Directed Graph

Undirected Graph: In undirected graph the pair of vertices representing any edge is unordered. Thus the pairs (u,v)and (v,u) represent the same edge.

Directed graph: In a directed graph each edge is represented by a directed pair <u,v>, u is the tail and v is the head of the edge. Therefore <v,u>and <u,v> represent two different edges.

Graph representation can be done using adjacency matrix.

Adjacency matrix: Adjacency matrix is a two dimensional n x n array a, with the property that a[i][j]=1 iff the edge (i,j) is in E(G). a[i][j]=0 if there is no such edge in G.

Connectivity of the graph: A graph is said to be connected iff for every pair of distinct vertices u and v , there is a path from u to v.

Path: A path from vertex u to v in a graph is a sequence of vertices u, i_1 , i_2 …… i_k , v such that (u, i_1), (i_1 , i_2) …… (i_k , v) are the edges in G.

Graph traversal can be done in two ways: depth first search(dfs) and breadth first search (bfs).

Hashing: Hashing is a process of generating key or keys from a string of text using a mathematical function called hash function. Hashing is a key-to-address mapping process.

Hash Table: In hashing the dictionary pairs are stored in a table called hash table. Hash tables are partitioned into buckets, buckets consists of s slots , each slot is capable of holding one dictionary pair.

Hash function: A hash function maps a key into a bucket in the hash table.
Most commonly used hash function is,

$$h (k) = k \% D$$

where,
k is the key
D is Max size of hash table.

Collision Resolution: When we hash a new key to an address, collision may be created. There are several methods to handle collision,
Open addressing, linked lists and buckets.
In open addressing, several ways are listed,
Linear probing, quadratic probe, pseudorandom, and key offset.

Linear Probing: In linear probing, when data cannot be stored at the home address, collision is resolved by adding 1 to the current address.

1. **Design, Develop and Implement a menu driven program in c for the following Array operations,**
   a. **Creating an Array of N integer elements.**
   b. **Display of Array elements with suitable headings.**
   c. **Inserting an element (ele) at a given valid position(pos).**
   d. **Deleting an element at a given valid position (pos).**
   e. **Exit.**

   Support the program with functions for each of the above operations.

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

int *create(int);
void display(int *,int);
void insert(int *,int, int, int);
void delete( int *, int, int );

main()
{
        int *array,size,choice,pos,n=0,ele;
        while(1)
        {
                printf("\n\n Program to illustrate Array operations\n");
                printf("\n\t 1=> Create an ARRAY of required SIZE\n\t 2=> Display ARRAY
elements\n\t 3=> Insert an ELEMENT to ARRAY at any valid POSITION\n\t 4=> Delete an
element from ARRAY at given valid POSITION\n\t 5=> Exit\n\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("enter the SIZE of the array:\n");
                                scanf("%d",&size);
                                array=create(size);
                                break;
                        case 2: display(array,n);
                                break;
                        case 3: if(array)
                                {
                                        if(n==size)
                                        {
                                                printf("\nArray is FULL\n");
                                                break;
                                        }
                                        printf("Enter the valid position to insert an element\n");
                                        scanf("%d",&pos);
                                        if(pos>n+1)
                                        {
                                                printf("\n\tPlease enter the valid position\n");
                                        }
                                        else
                                        {
                                                printf("Enter the element to be inserted\n");
                                                scanf("%d",&ele);
                                                insert(array,ele,pos,n);
                                                n++;
                                        }
```

```
                        }
                        else
                        {
                                printf("\nArray has to be created first before inserting
element\n");
                        }
                        break;
                case 4: if(array)
                 {
                        printf("Enter the valid position to delete an element\n");
                        scanf("%d",&pos);
                        if(pos>=n)
                        {
                                printf("\n\tPlease enter the valid position\n");
                        }
                        else
                        {
                                delete(array,pos,n);
                                        n--;
                        }
                 }
                 else
                 {
                        printf("\nArray has to be created first before deleting an
element\n");
                 }
                 break;
                case 5: return;
            }
        }
}

int * create(int size)
{
        return ((int *)malloc(sizeof(int)*size));
}

void display( int *array, int n)
{
       int i;
       if(n==0)
              printf("\nArray is empty\n");
       else
       {
              printf("\nArray elements are: ");
              for(i=0;i<n;i++)
                     printf("%d   ",array[i]);
       }
}

void insert( int * array,int ele,int pos,int n)
{
        int i;
        for(i=n-1;i>=pos-1;i--)
                array[i+1]=array[i];
        array[pos-1]=ele;
}
```

```
void delete(int *array, int pos,int n)
{
        int i;
        printf("\nThe element deleted is %d",array[pos-1]);
        if(n==1)
                return;
        else
                for(i=pos-1;i<n;i++)
                        array[i]=array[i+1];
}
```

2. Design, Develop and Implement a program in c for the following operations on strings

a. Read a main string (str), a pattern string (pat) and a replace string (rep).

b. Perform pattern matching operation: Find and replace all occurrences of pat in str with rep if pat exits in str. Report with suitable messages in case pat does not exists in str.

Support the program with functions for each of the above operations. Don't use built in functions.

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

void read(char *,char *, char *);
void find_replace(char *, char *, char *);
main()
{
        int choice;
        char str[100],pat[25],rep[25];
        while(1)
        {
                printf("\n\n Program to find and replace all occurences of PATTERN in the main
STRING with a REPLACE_STRING\n");
                printf("\n\t 1=> Read main STRING, PATTERN string and REPLACE string\n \t
2=> Perform FIND and REPLACE operation\n \t 3=> Exit\n\n");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1: read(str,pat,rep);
                                break;
                        case 2: find_replace(str,pat,rep);
                                break;
                        case 3: return;
                }
        }
}

void read(char *str, char *pat, char *rep)
{

        printf("Enter the main string: ");
        scanf("%s",str);
        printf("\n\nEnter the Pattern string: ");
        scanf("%s",pat);
        printf("\n\nEnter the Replace String: ");
        scanf("%s",rep);
        return;
}

void find_replace(char *str, char *pat, char *rep)
{
        int i=0,j=0,k=0,l=0,m=0,step=0;
        step=strlen(rep)-strlen(pat);

        for(i=0,j=0;str[i]!='\0';i++)
        {
                if(str[i]==pat[j])
```

```
		{
			j++;
			if(pat[j]=='\0')
			{
				if(str[i+1]=='\0')
				{
					i=i-strlen(pat)+1;
					do
					{
						str[i++]=rep[m++];
					}while(rep[m]!='\0');
					str[i]='\0';i--;
				}
				else
				{
					if(step>0)
					{
						for(k=i;str[k]!='\0';k++);
					str[k+step]='\0';k--;
						do
						{
							str[k+step]=str[k];
							k--;
						}while(k!=i);
					}
					else
					{
						l=i+1;
						for(k=l+step;str[l]!='\0';k++,l++)
							str[k]=str[l];
						str[k]='\0';
					}
					i=i-strlen(pat)+1;
					while(rep[m]!='\0')
					{
						str[i++]=rep[m++];
					}
					m=0;
				i--;j=0;
				}
			}
		}

		else
			j=0;
	}
	printf("\n\n Main STRING after FIND and REPLACE\n\t");
	printf("\n\n%s",str);
}
```

3. Design, Develop and Implement a menu driven program in c for the following operations on STACK of integers(Array implementation of stack with maximum size MAX)

a. **Push an element onto the stack.**
b. **Pop an element from the stack.**
c. **Demonstrate how stack can be used to check palindrome.**
d. **Demonstrate overflow and underflow situations on stack.**
e. **Display the status of the stack.**
f. **Exit**

Support the program with appropriate functions for each of the above operations.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define max 20

int s[max],stop;
int ele,st[max],sp,ch;
void push(int ele,int s[],int *top);
int pop(int s[],int *top);
void palindrome(int ele,int st[]);
void display(int s[],int *top);

void main()
{
        stop=-1;
        sp=-1;
        while(1)
        {
                printf("Enter the choice\n");
                printf("Enter 1 to insert an element in to stack\n");
                printf("Enter 2 to delete an element from the stack\n");
                printf("Enter 3 to check an element is pailndrome or not\n");
                printf("Ebter 4 to check status of the stack\n");
                printf("Enter 5 to exit\n");
                scanf("%d",&ch);

                switch(ch)
                {
                        case 1: printf("Enter the element to be inserted in to stack\n");
                                scanf("%d",&ele);
                                push(ele,s,&stop);
                                break;
                        case 2:         ele = pop(s,&stop);
                                printf("Element poped is %d\n",ele);
                                break;
                        case 3:         printf("Enter the element to check whether it is
palindrome or not\n");
                                scanf("%d",&ele);
                                palindrome(ele,st);
                                break;
                        case 4:         printf("The status of the stack is \n");
                                display(s,&stop);
                                break;
                        case 5:         exit(0);
                }
```

```
        }
}

void push(int ele,int s[],int *stop)
{
        if(*stop>max-1)
                printf("stack overflow\n");
        else
        {
                s[++*stop]=ele;
        }
}

int pop(int s[],int *top)
{
        if(*top == -1)
                printf("stack overflow\n");
        else
                return(s[(*top)--]);
}

void palindrome(int ele,int st[])
{       int rem,rev=0,temp=ele,i=0;

        while(temp!=0)
        {
                rem=temp%10;
                push(rem,st,&sp);
                temp=temp/10;
        }
        while(sp!=-1)
        {

        rev=rev+(pop(st,&sp)*pow(10,i++));
        }
        if(ele==rev)
                printf("Palindrome\n");
        else
                printf("Not palindrome\n");
}

void display(int s[],int *stop)
{
        int i;
        if(*stop==-1)
                printf("stack empty\n");
        else
                for(i=*stop;i>-1;i--)
                        printf("%d\n",s[i]);
}
```

4. Design, develop, and implement a program in C for converting an infix expression to postfix expression. Program should support for both parenthesized and free parenthesized expressions with the operators: +, - , *,/,% (Remainder),^(power) and alphanumeric operands.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define SIZE 20
#define Operator (-10)
#define Operand (-20)
#define Leftparen (-30)
#define Rightparen (-40)
int top=-1;
char stack[SIZE],infix[SIZE],postfix[SIZE];
void push(char symbol)
{
   if(top==(SIZE-1))
   printf("\nSTACK OVERFLOW...!!");
 else
   stack[++top]=symbol;
}
char pop(void)
{
  char pop_ele;
    if(top==-1)
      printf("\nSTACK UNDERFLOW...!!");
    else
    {
      pop_ele=stack[top--];
      return pop_ele;
    }
}

void infix_to_postfix(void)
{
  int i,p,len,type,precedence;
  char next;
  i=p=0;
  len=strlen(infix);
  while(i<len)
  {
    if(!white_space(infix[i]))
    {
      type=get_type(infix[i]);
      switch(type)
      {
          case Leftparen:
                                push(infix[i]);
                        break;
          case Rightparen:
                                while((next=pop())!='(')    postfix[p++]=next;
                        break;
          case Operand:
                                postfix[p++]=infix[i];
                        break;
          case Operator:
```

```
                                precedence=get_prec(infix[i]);
                                while(top>-1 && precedence<=get_prec(stack[top]))
                                postfix[p++]=pop();
                                push(infix[i]);
                        break;
     }
    }
    i++;
  }while(top>-1)
        postfix[p++]=pop();
  postfix[p]='\0';
}
int get_type(char symbol)
{
  switch(symbol)
  {
    case '(':                return(Leftparen);
    case ')':                return(Rightparen);
    case '+':
    case '-':
    case '*':
    case '%':
    case '/':                return(Operator);
    default:                return(Operand);
  }}
int get_prec(char symbol)
{
  switch(symbol)
  {
    case '+':
    case '-':        return 1;
    case '*':
    case '/':
    case '%':        return 2;
    case '(':      return 0;
    default:       return 999;
  }
}

int white_space(char symbol)
{
   return(symbol=='\0' || symbol==' ' || symbol=='\t');
}


void main()
{
  int q=0,choice;
   do
   {
     top=-1;
     clrscr();
     printf("\n\n\n\n\t\t\t******MAIN MENU******");
     printf("\n\n\n\n\tENTER 1 --> TO CONVERT INFIX TO POSTFIX");
     printf("\n\tENTER 2 --> TO QUIT");
     printf("\n\nEnter your choice : ");
     scanf("%d",&choice);
```

```
    switch(choice)
    {
        case 1:
                printf("\nEnter an infix expression : ");
                fflush(stdin);
                gets(infix);
                infix_to_postfix();
                printf("\nInfix expression : %s",infix);
                printf("\nPostfix expression : %s",postfix);
                getch();
                clrscr();
        break;
        case 2:
                q=1;
                printf("\nTHANK YOU.... HAVE A NICE DAY..!!!");
                getch();
    }
  }while(!q);
}
```

5. Design, develop, and implement a program in C for the following stack applications

a. Evaluation of suffix expression with single digit operands and operators: +, - , *, /, % , ^

b. Solving Tower of Hanoi problem with n disks.

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

#include<math.h>
int count=0, top=-1;
int operate(char symb, int op1, int op2)
{
	switch(symb)
	{
		case '+':return op1+op2;
		case '-':return op1-op2;
		case '/':return op1/op2;
		case '*':return op1*op2;
		case '%':return op1%op2;
		case '^':return pow(op1,op2);
	}
}
void push(int stack[],int d)
{
	stack[++top]=d;
}
int  pop(int stack[])
{
	return(stack[top--]);
}
void tower( int n,char src, char intr, char des)
{
	if(n)
	{
		tower(n-1,src,des,intr);
		printf("disk %d  moved from %c to %c\n",n,src,des);
		count++;
		tower(n-1,intr,src,des);
	}
}
void main()
{
	int n, choice,i,op1,op2,ans,stack[50];
	char expr[20],symb;
	while(1)
	{
		printf("\nprogram to perform evaluation of suffix expression and tower of
hanoi problem\n");
		printf("\n1. evaluate suffix expression\n 2.Tower of hanoi\n 3.Exit\n ");
		printf("\nenter the choice\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1: printf("enter the suffix expression : ");
				scanf("%s",expr);
				for(i=0;expr[i]!= '\0';i++)
```

```
                        {
                                symb=expr[i];
                                if(symb>='0' && symb<='9')
                                push(stack, symb-'0');
                                else
                                {
                                        op2=pop(stack);
                                        op1=pop(stack);
printf("%d %d %c",op2,op1,symb);

                                        ans=operate(symb,op1,op2);
                                        push(stack,ans);
                                }
                        }
                        ans=pop(stack);
                        printf("The result of the suffix expression is %d",ans);
                break;

                case 2:         printf("enter the number of disks\n");
                        scanf("%d",&n);
                        tower(n,'a','b','c');
                        printf("number of moves taken to move disks from source to
destination %d",count);
                break;
                case 3: return;
            }
        }
}
```

6. Design, Develop and Implement a menu driven Program in C for the following operations on Circular QUEUE of Characters (Array Implementation of Queue with maximum size MAX)

a. Insert an Element on to Circular QUEUE
b. Delete an Element from Circular QUEUE
c. Demonstrate *Overflow* and *Underflow* situations on Circular QUEUE
d. Display the status of Circular QUEUE
e. Exit

Support the program with appropriate functions for each of the above operations.

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

int front=-1, rear=-1, n=0;
char cqueue[MAX],element;

void insert();
void delete();
void display();

main()
{
        int choice;
        while(1)
        {
                printf("\n\n Program to illustrate operations on CIRCULAR QUEUE of charecters
\n");
                printf("\n\t 1=> Insert an element on to CIRCULAR QUEUE \n\t 2=> Delete an
element from CIRCULAR QUEUE\n\t 3=> Display the status of CIRCULAR QUEUE\n\t 4=>
Exit\n\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: insert();
                                break;
                        case 2: delete();
                                break;
                        case 3: display();
                                break;
                        case 4: return;
                }
        }
}

void insert()
{
        rear=(rear+1)%MAX;
        if( front == rear && n == MAX )
        {
                printf("CIRCULAR QUEUE is FULL, Element cannot be inserted\n");
                return;
        }
        printf("\nEnter the element to be inserted into CIRCULAR QUEUE");
        scanf("%c",&element);
```

```
        cqueue[rear]=element;
        n++;
}

void delete()
{
        if( front == rear && n == MAX )
        {
                printf("CIRCULAR QUEUE is EMPTY, no ELEMENTS to delete\n");
                return;
        }
        front=(front+1)%MAX;
        printf("The ELEMENT deleted is %c\n",cqueue[front]);
        n--;
}

void display()
{
        int i;
        if( front == rear && n == 0 )
        {
                printf("CIRCULAR QUEUE is EMPTY, no ELEMENTS to display\n");
                return;
        }
        printf("CIRCULAR QUEUE contents are:  ");
        for(i=front;i<=rear;i++)
        {
                printf("%c   ",cqueue[i]);
        }
}
```

7. Design, Develop and Implement a menu driven Program in C for the following operations on
Singly Linked List (SLL) of Student Data with the fields: *USN, Name, Branch, Sem, PhNo*
a. Create a SLL of N Students Data by using *front insertion*.
b. Display the status of SLL and count the number of nodes in it
c. Perform Insertion and Deletion at End of SLL
d. Perform Insertion and Deletion at Front of SLL
e. Demonstrate how this SLL can be used as STACK and QUEUE
f. Exit

```
#include<stdio.h>
#include<conio.h>
struct list
{
        char usn[10],name[10];
        char branch[5];
        int sem;
        int phno;
        struct list *next;
};
typedef struct list node;




node* insertfront(node *first)
{

        node *temp;
        temp=(struct list *)malloc(sizeof(node));
        if(temp!=NULL)
        printf("memory allocated\n");


                printf("enter the usn,name,branch,sem,phno\n");
                scanf("%s%s%s%d%d",&temp->usn,&temp->name,&temp->branch,&temp-
>sem,&temp->phno);
                temp->next=NULL;
                if(!first)
                return(temp);
                temp->next=first;
                first=temp;
                return first;
}
node* delfront(node *first)
{
        int i;

        node *temp=first;
        if(first==NULL)
        {
                printf("list is empty..can't delete\n");
                getch();
                return first;
        }
```

```
        else
        {
                printf("the student record deleted is %s",temp->name);
                first=temp->next;
                free(temp);

        }
return first;
}
void insertend(node *first)
{
        node *ptr,*temp;
        temp=(struct list *)malloc(sizeof(node));
        printf("enter the usn,name,branch,sem,phno\n");
        scanf("%s%s%s%d%d",&temp->usn,&temp->name,&temp->branch,&temp-
>sem,&temp->phno);
        temp->next=NULL;
        ptr=first;
        while(ptr->next!=NULL)
        ptr=ptr->next;
        ptr->next=temp;



}


node* delend(node *first)
{
        node *last,*ptr;
        ptr=last=first;
        if(first->next==NULL)
        {
        printf("\nthe record deleted is %s",first->name);
        free(first);
        return NULL;
        }
        while(last->next)
        {
          ptr=last;
          last=last->next;
        }
        printf("\nthe record deleted is %s",last->name);
        ptr->next=NULL;
        free(last);
return first;
}
void display(node *first)
{

        int count=0;
        node *temp=first;

        if(first==NULL)
        {
        printf("no records to display\n");
        return;
```

```
        }
        while(temp!=NULL)
        {
                count++;
                printf("\n%s\t%s\t%s\t%d\t%d\n",temp->usn,temp->name,temp-
>branch,temp->sem,temp->phno);
                temp=temp->next;
        }
        printf("Total number of records is %d",count);
}
node* stackoper(node *first)
{

        int ch;


        printf("\n1.push\t2.pop\n");
        printf("enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:first=insertfront(first);
                        break;
                case 2:first=delfront(first);
                        break;
        }

return first;
}
node* queoper(node *first)
{

        int ch;


        printf("\n1.enque\t2.deque\n");
        printf("enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:insertend(first);
                break;
        case 2:first=delfront(first);
                break;
        }
return first;
}

void main()
{
   int i,n,choice;
   node *first=NULL;
   clrscr();
   printf("Single linked list\n");
   while(1)
   {
```

```
   printf("\n1.create list 2.insert front 3.insert end 4.delete front 5.delete at end ");
   printf("6.display 7.To demonstrate SLL as stack 8.To demonstrate SLL as queue
9.exit\n");


        printf("\nenter your choice\n");
        scanf("%d",&choice);
        switch(choice)
        {
                case 1:printf("\nenter the number of students\n");
                       scanf("%d",&n);
                       for(i=1;i<=n;i++)
                       first=insertfront(first);
                       break;
                case 2:first=insertfront(first);
                         break;
                case 3:insertend(first);
                         break;
                case 4:first=delfront(first);
                         break;
                case 5:first=delend(first);
                         getch();
                         break;
                case 6:display(first);
                         break;
                case 7:first=stackoper(first);
                         getch();
                         break;
                case 8:first=queoper(first);
                         getch();
                         break;
                case 9:exit();
        }

   }

}
```

8. Design, Develop and Implement a menu driven Program in C for the following operations on
Doubly Linked List (DLL) of Employee Data with the fields: *SSN, Name, Dept, Designation,*
Sal, PhNo
a. Create a DLL of N Employees Data by using *end insertion*.
b. Display the status of DLL and count the number of nodes in it
c. Perform Insertion and Deletion at End of DLL
d. Perform Insertion and Deletion at Front of DLL
e. Demonstrate how this DLL can be used as Double Ended Queue
f. Exit

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

struct employee
{
        char ssn[10];
        char name[25];
        char dept[25];
        char desig[10];
        double sal;
        char ph[12];
        struct employee *prev;
        struct employee *next;
};


struct employee * create(struct employee *, int);
void display(struct employee *);
struct employee * insert_front(struct employee *);
struct employee * delete_front(struct employee *);
struct employee * insert_end(struct employee *);
struct employee * delete_end(struct employee *);
void dequeue();

void main()
{
        int choice,n;
        struct employee *head=NULL;

        printf("\nProgram to illustrate Doubly Linked List operations\n");
        while(1)
        {
                printf("\nEnter 1=> Create DLL of N employees by using end insertion\n\t
2=> Display the status of DLL and count\n\t 3=> Perform Insertion at End\n\t 4=> Perform
Deletion at End\n\t 5=> Perform Insertion at front\n\t 6=> Perform Deletion at Front\n\t
7=> Demonstration of Dequeue\n\t 8=> Exit\n\t");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1:         head=NULL;
                                printf("Enter the number of Employees\n");
                                scanf("%d",&n);
                                head=create(head,n);
                                break;
```

```
                        case 2:         display(head);
                                break;

                        case 3:         head=insert_end(head);
                                break;

                        case 4:         head=delete_end(head);
                                break;

                        case 5: head=insert_front(head);
                                break;

                        case 6: head=delete_front(head);
                                break;

                        case 7: dequeue();
                                break;

                        case 8: return;
                }
        }
}

struct employee * create(struct employee *head, int n)
{
        int i;
        struct employee *temp,*ptr;

        for(i=0;i<n;i++)
        {
                temp=(struct employee*)malloc(sizeof(struct employee));
                printf("\nEnter the details of Employee %d\n\n",i+1);
                scanf("%s",temp->ssn);
                scanf("%s",temp->name);
                scanf("%s",temp->dept);
                scanf("%s",temp->desig);
                scanf("%lf",&temp->sal);
                scanf("%s",temp->ph);

                temp->prev=temp->next=NULL;

                if(!head)
                {
                        head=temp;
                        continue;
                }

                ptr=head;
                while(ptr->next) ptr=ptr->next;

                temp->prev=ptr;
                ptr->next=temp;
        }
        return head;
}
```

```
void display(struct employee *head)
{
	struct employee *ptr;
	int count=0;

	for(ptr=head;ptr;ptr=ptr->next)
	{
		count++;
		printf("\nEmployee %d\n\tSSN: %s\n\tNAME: %s\n\tDEPT:
%s\n\tDESIGNATION: %s\n\tSALARY: %lf\n\tph: %s\n\n",count,ptr->ssn,ptr->name,ptr-
>dept,ptr->desig,ptr->sal,ptr->ph);
	}
	printf("The total number of Employees are %d",count);
}


struct employee * insert_front(struct employee *head)
{
	struct employee *temp;

	temp=(struct employee*)malloc(sizeof(struct employee));
	printf("\nEnter the details of Employee\n\n");
	scanf("%s%s%s%s%lf%s",temp->ssn,temp->name,temp->dept,temp-
>desig,&temp->sal,temp->ph);
	temp->prev=temp->next=NULL;

	if(!head) return temp;

	head->prev=temp;
	temp->next=head;
	return temp;
}


struct employee * delete_front(struct employee *head)
{
	if(!head)
	{
		printf("No records to delete\n");
		return NULL;
	}
	else if(!head->next)
	{
		return NULL;
	}
	else
		return head->next;
}


struct employee * insert_end(struct employee *head)
{
	struct employee *temp,*ptr;

	temp=(struct employee*)malloc(sizeof(struct employee));
	printf("\nEnter the details of Employee \n\n");
```

```
        scanf("%s%s%s%s%lf%s",temp->ssn,temp->name,temp->dept,temp->desig,&temp->sal,temp->ph);
        temp->prev=temp->next=NULL;

        if(!head) return temp;

        ptr=head;
        while(ptr->next) ptr=ptr->next;

        temp->prev=ptr;
        ptr->next=temp;

        return head;
}


struct employee * delete_end(struct employee *head)
{
        struct employee *ptr;

        if(!head)
        {
                printf("No records to delete\n");
                return head;
        }

        if(!head->next) return NULL;

        ptr=head;
        while(ptr->next->next) ptr=ptr->next;

        free(ptr->next);
        ptr->next=NULL;

        return head;
}

void dequeue()
{
        struct employee *head=NULL;
        int choice;

        printf("\nDemonstration of Double-Ended_Queue operations\n");
        while(1)
        {
                printf("\nEnter 1=> Insert an element to front end\n\t 2=> Delete an element from front end\n\t 3=> Insert an element to rear end\n\t 4=> Delete an element from rear end\n\t 5=> Display Deque status\n\t 6=> Exit\n\t");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1:         head=insert_front(head);
                                break;

                        case 2:         head=delete_front(head);
                                break;
```

```
                    case 3:     head=insert_end(head);
                                break;

                    case 4:     head=delete_end(head);
                                break;

                    case 5:     display(head);
                                break;

                    case 6: return;
                }
        }
}
```

9. Design, Develop and Implement a Program in C for the following operations on Singly Circular
Linked List (SCLL) with header nodes
a. Represent and Evaluate a Polynomial P(x,y,z) = 6x2y2z-4yz5+3x3yz+2xy5z-2xyz3
b. Find the sum of two polynomials POLY1(x,y,z) and POLY2(x,y,z) and store the result in
POLYSUM(x,y,z)
Support the program with appropriate functions for each of the above operations

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

#define COMPARE(x,y) ((x>y)?1:((x<y)?-1:0))

struct clist
{
  int coef;
  int expo;
  struct clist *next;
};

addtolist(int,int,struct clist *);
display(struct clist *);
void addpoly(struct clist*,struct clist*, struct clist *);

void main()
{
     int element,choice,terms,i,coef,expo;
     struct clist *first,*second,*res;
     first=(struct clist*)malloc(sizeof(*first));
     first->next=first;
     second=(struct clist*)malloc(sizeof(*first));
     second->next=second;
     res=(struct clist*)malloc(sizeof(*first));
     res->next=res;
     clrscr();
     while(1)
     {
         printf("\n\n1 insert polynomials\n 2 add polynomials\n");
         printf("enter your choice");
         scanf("%d",&choice);
         switch(choice)
         {
             case 1:printf("enter the no of terms of poly A\n");
                    scanf("%d",&terms);
                    for(i=0;i<terms;i++)
                    {
                            printf("enter the element to the list\n");
                            scanf("%d%d",&coef,&expo);
                            addtolist(coef,expo,first);
                    }
                    printf("enter the no of terms of poly B\n");
                    scanf("%d",&terms);
                    for(i=0;i<terms;i++)
                    {
```

```
                        printf("enter the element to the list\n");
                        scanf("%d%d",&coef,&expo);
                        addtolist(coef,expo,second);
                }
                display(first);
                printf("\n");
                display(second);
                break;

        case 2: addpoly(first,second,res);
                display(res);
                getch();
                exit(0);
        }
    }
}

display(struct clist *first)
{
    struct clist *cur;
    cur=first->next;
    while(cur!=first)
    {
      printf("+ %dX^%d ",cur->coef,cur->expo);
      cur=cur->next;
    }
    return 0;
}

addtolist(int coef, int expo, struct clist *first)
{
    struct clist * temp,* current;
    temp=(struct clist*)malloc(sizeof(*temp));
    temp->coef=coef;
    temp->expo=expo;
    temp->next=first;
    current =first;
    while(current->next!=first)
    {
      current=current->next;
    }
      current->next=temp;
     return 0;
}

void addpoly(struct clist* first,struct clist* second, struct clist *res)
{
   struct clist *temp1,*temp2;
   temp1=first->next;
   temp2=second->next;

   while ((temp1!=first) && (temp2!=second))
        switch(COMPARE(temp1->expo,temp2->expo))
        {
                case 0: addtolist((temp1->coef+temp2->coef),temp1->expo,res);
                        temp1=temp1->next;
                        temp2=temp2->next;
```

```
                        break;

                case 1: addtolist(temp1->coef,temp1->expo,res);
                        temp1=temp1->next;
                        break;

                case -1:addtolist(temp2->coef,temp2->expo,res);
                      temp2=temp2->next;
                      break;
            }
    while (temp1!=first)
    {
            addtolist(temp1->coef,temp1->expo,res);
            temp1=temp1->next;
    }

    while (temp2!=second)
    {
            addtolist(temp2->coef,temp2->expo,res);
            temp2=temp2->next;
    }
}
```

10. Design, Develop and Implement a menu driven Program in C for the following operations on
Binary Search Tree (BST) of Integers
a. Create a BST of N Integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2
b. Traverse the BST in Inorder, Preorder and Post Order
c. Search the BST for a given element (KEY) and report the appropriate message
d. Delete an element(ELEM) from BST
e. Exit

```
//Binary Search Tree

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

struct tree
{
        int data;
        struct tree *left;
        struct tree *right;
};

struct tree * create(int);
void traverse(struct tree*);
void inorder(struct tree *);
void preorder(struct tree *);
void postorder(struct tree *);
struct tree * search(struct tree*,int);
struct tree * delete(struct tree*,int);

void main()
{
        int choice,n,key,element;
        struct tree *bst=NULL,*temp;

        printf("\nProgram to illustrate Binary Search Tree operations\n");
        while(1)
        {
                printf("\nEnter \t 1=> Create BST of N integers\n\t 2=> Traverse the BST in
Inorder, Preorder, and Postorder\n\t 3=> Search the BST for a Key element\n\t 4=> Delete
an Element from BST\n\t 5=> Exit\n\n\t");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1:         printf("Enter the number of elements of BST\n");
                                scanf("%d",&n);
                                bst=create(n);
                                break;

                        case 2:         traverse(bst);
                                break;

                        case 3:         printf("\nEnter the Key element to be searched in
BST\n");
                                scanf("%d",&key);
                                temp=search(bst,key);
                                if(temp)
```

```
                                printf("The Key element is found\n");
                        else
                                printf("The key element is not found\n");
                        break;

                case 4:         printf("\nEnter the Element to be deleted from BST\n");
                        scanf("%d",&element);
                        bst=delete(bst,element);
                        break;

                case 5: return;
                }
        }
}

struct tree *create(int n)
{
        struct tree *head=NULL,*ptr,*ptr1,*temp;
        int i,ele;

        printf("Enter the Elements\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&ele);
                temp=(struct tree *)malloc(sizeof(struct tree));
                temp->data=ele;
                temp->left=temp->right=NULL;

                if(!head)
                {
                        head=temp;
                        continue;
                }

                ptr=ptr1=head;

                do
                {
                        ptr=ptr1;
                        if(ptr->data>ele)
                                ptr1=ptr->left;
                        else if(ptr->data<ele)
                                ptr1=ptr->right;
                        else
                        {
                                printf("Duplicate element");
                                i--;
                                continue;
                        }
                }while(ptr1);

                if(ptr->data>ele)
                        ptr->left=temp;
                else
                        ptr->right=temp;
        }
        return head;
```

```
}

void traverse(struct tree *bst)
{
        printf("\nThe Inorder Traversal:\t");
        inorder(bst);
        printf("\nThe Preorder Traversal:\t");
        preorder(bst);
        printf("\nThe Postorder Traversal:\t");
        postorder(bst);
        printf("\n\n");
}

void inorder(struct tree *bst)
{
        struct tree *temp=bst;
        if(temp)
        {
                inorder(temp->left);
                printf("%d\t",temp->data);
                inorder(temp->right);
        }
}

void preorder(struct tree *bst)
{
        struct tree *temp=bst;
        if(temp)
        {
                printf("%d\t",temp->data);
                preorder(temp->left);
                preorder(temp->right);
        }
}

void postorder(struct tree *bst)
{
        struct tree *temp=bst;
        if(temp)
        {
                postorder(temp->left);
                postorder(temp->right);
                printf("%d\t",temp->data);
        }
}

struct tree* search(struct tree *bst,int key)
{
        struct tree* temp=bst;
        while(temp)
        {
                if(temp->data==key)
                        return temp;
                else if(temp->data>key)
                        temp=temp->left;
                else
                        temp=temp->right;
```

```
        }
        return temp;
}

struct tree *delete(struct tree *bst,int ele)
{
        struct tree *temp=bst,*temp1=bst,*ptr,*ptr1;

        if(bst->data==ele && !bst->left && !bst->right)
        {
                free(bst);
                return NULL;
        }
        if(bst->data==ele && !bst->left)
        {
                temp=bst->right;
                free(bst);
                return temp;
        }
        if(bst->data==ele && !bst->right)
        {
                temp=bst->left;
                free(bst);
                return temp;
        }
        if(bst->data==ele && bst->left && bst->right)
        {
                ptr=ptr1=bst;
                while(ptr1->left)
                {
                        ptr=ptr1;
                        ptr1=ptr->left;
                }
                ptr->left=NULL;
                ptr1->left=bst->left;
                ptr1->right=bst->right;
                free(bst);
                return ptr1;
        }

        while(temp1)
        {
                if(temp1->data==ele)
                        break;

                temp=temp1;
                if(temp->data>ele)
                        temp1=temp->left;
                else
                        temp1=temp->right;
        }


        if(temp1 && temp1->data==ele)
        {
                if(!temp1->left && !temp1->right)
                {
```

```
                if(temp->left==temp1)
                        temp->left=NULL;
                else
                        temp->right=NULL;
        }
        else if(!temp1->left)
        {
                if(temp->left==temp1)
                        temp->left=temp1->right;
                else
                        temp->right=temp1->right;
        }
        else if(!temp->right)
        {
                if(temp->left==temp1)
                        temp->left=temp1->left;
                else
                        temp->right=temp1->left;
        }
        else
        {
                ptr=ptr1=temp1->right;
                while(ptr1->left)
                {
                        ptr=ptr1;
                        ptr1=ptr->left;
                }
                ptr->left=NULL;

                ptr1->left=temp1->left;
                ptr1->right=temp1->right;

                if(temp->left==temp1)
                        temp->left=ptr;
                else
                        temp->right=ptr;
        }
        free(temp1);
    }
    else
        printf("The element not found\n");
    if(bst->data!=ele)
        return bst;
    else
        return ptr1;
}
```

11. Design, Develop and Implement a Program in C for the following operations on Graph(G) of
Cities
a. Create a Graph of N cities using Adjacency Matrix.
b. Print all the nodes reachable from a given starting node in a digraph using BFS method
c. Check whether a given graph is connected or not using DFS method.

```
#include<stdio.h>

void dfs(int src, int adj[10][10],int visited[10],int n)
{
	int k;
	visited[src]=1;

	for(k=0;k<n;k++)
	{
		if(adj[src][k]==1 && visited[k]==0)
		{

			dfs(k,adj,visited,n);
		}
	}
}
void bfs(int src, int adj[10][10],int n)
{
	int q[20], front=0,rear=-1,v,u,visited[10]={0};
	q[++rear]=src;
	visited[src]=1;
	printf("%d",src);
	while(front<=rear)
	{
		u=q[front++];
		for(v=0;v<n;v++)
		{
			if(adj[u][v]==1 && visited[v]==0)
			{
				q[++rear]=v;
				printf("%d",v);
				visited[v]=1;
			}
		}
	}


}

void main()
{
	int choice, i,j,src,flag=0;
	int adj[10][10],visited[10]={0},n;
	clrscr();

	while(1)
	{
		printf("\n\n\n program to perform graph operations\n\n");
```

```
            printf("\n\t 1=> Create a graph of N cities\n\t 2=> To print reachable nodes
from source node using BFS\n3.To check graph connected or not using DFS\n\n");
            printf("\nenter the choice\n");
            scanf("%d",&choice);



            switch(choice)
            {
                case 1:         printf("\nenter the number of cities\n");
                        scanf("%d",&n);
                        printf("\nenter the adjacency matrix\n");
                        for(i=0;i<n;i++)
                        for(j=0;j<n;j++)
                        scanf("%d",&adj[i][j]);
                        break;
                case 2: printf("Enter the source vertex to start traversal\n");
                        scanf("%d",&src);
                        printf("vertices visited are\n");
                        bfs(src,adj,n);
                        break;
                case 3:printf("Enter the source vertex to start traversal\n");
                      scanf("%d",&src);
                      for(i=0;i<n;i++)
                      visited[i]=0;
                      dfs(src,adj,visited,n);
                      for(i=0;i<n;i++)
                         if(visited[i]==0)
                                flag=1;
                        if(flag==1)
                           printf("the graph is not connected\n");

                         else
                           printf("the graph is connected\n");
                      break;
                case 4: exit();
            }
        }
}
```

12. Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table(HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are Integers. Design and develop a Program in C that uses Hash function H: K ®L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing.

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

typedef struct
{
        char name[30];
        char id[10];
        char address[50];
        char mail_id[20];
        char mobile[13];
}employee_list;

const int max_size=100;

void build_htable(employee_list *,int *);
void hash_search(employee_list *,int);
int hash_key( char *, int);
int collision(int,int);

void main()
{
        employee_list erecords[max_size];
        int last;

        last=max_size-1;

        build_htable(erecords, &last);
        hash_search(erecords,last);
}

void build_htable(employee_list *erecord,int * last)
{
        FILE *fp;
        employee_list list;
        int loc, cntcoli, end,i;

        fp=fopen("emp.txt","r");

        if(!fp)
        {
                printf("unable to open the file emp.txt\n");
                exit(0);
        }
        for(i=0;i<=*last;i++)
```

```
        {
                erecord[i].id[0]='\0';
        }
        while(!feof(fp))
        {
                fscanf(fp,"%[^;]%*c %[^;]%*c %[^;]%*c %[^;]%*c
%[^;]%*c",list.name,list.id,list.address,list.mail_id,list.mobile);
                loc=hashkey(list.id,*last);
                if(erecord[loc].id[0]!='\0')
                {
                        end=*last;
                        cntcoli=0;
                        while(erecord[loc].id[0]!='\0' && cntcoli++<=*last)
                        {
                                loc=collision(*last,loc);
                        }
                        if(erecord[loc].id[0]!='\0')
                        {
                                printf("List is full\n");
                                return;
                        }
                }
                erecord[loc]=list;
        }
        return;
}

int hashkey(char *key, int last)
{
        int add, klen,i;
        klen = strlen(key);
        add =0;
        for(i=0;i<klen;i++)
        {
                if(key[i]!=' ')
                        add=add+key[i];
        }
        return (add%last+1);
}

int collision(int last,int loc)
{
        return loc<last ? ++loc:0;
}

void hash_search(employee_list *erecord, int last)
{
        char id[30];
        char more;
        int loc, max_search, cntcoli,i;

        do
        {
                printf("enter the id\n");
                scanf("%s",id);
                loc=hashkey(id,last);
                if(strcmp(id,erecord[loc].id)!=0)
```

```
                {
                        max_search=last;
                        cntcoli=0;
                        while(strcmp(id,erecord[loc].id)!=0 && cntcoli++<=max_search)
                                loc=collision(last,loc);
                }
                if(strcmp(id,erecord[loc].id)==0)

        printf("%s%s%s%s%s",erecord[loc].name,erecord[loc].id,erecord[loc].address,ereco
rd[loc].mail_id,erecord[loc].mobile);
                else
                        printf("%s not found",id);

                printf("lookup for another id (y|n)");
                scanf("%c",&more);
        }while(more!='n');
}
```