Q.2a. Discuss the procedure of writing a Recursive function. Also write C programs for the following with the explanation in support of your answer:

(i) To find the sum of n numbers by recursion.
(ii) To reverse a given number.

Sol:
Recursion is a special case of function call where a function calls itself. These are very useful in the situations where solution can be expressed in terms of successively applying same operation to the subsets of the problem.
Recursion is defined as a technique of defining a set or a process in terms of itself.
There are two important conditions that must be satisfied by any recursive procedure. They are: -
1. A smallest, **base case** that is processed without recursion and acts as decision criterion for stopping the process of computation and
2. A general method that makes a particular case to reach nearer in some sense to the base case.

For example:
A recursive function to calculate factorial of a number n is given below:

```
fact(int n)
{
int factorial;
if(n==1||n==0)
return(1);
else
factorial=n*fact(n-1);
return (factorial);
}
```

Assume n=4, we call fact(4)
Since n_1or 0, factorial=n*fact(n-1)
Factorial=4*fact(3) (again call fact function with n=3)
=4*3*fact(2) (again call fact function with n=2)
=4*3*2*fact(1) (again call fact function with n=1)
=4*3*2*1 (terminating condition)
=24.
We should always have a terminating condition with a recursive function call otherwise function will never return.

(i) **A program in c to find the addition of n numbers by recursion is as:**

```
#include<stdio.h>
int main()
{
```

```
 int n,sum;
printf("Enter the value of n: ");
scanf("%d",&n);
sum = getSum(n);
printf("Sum of n numbers: %d",sum);
return 0;
}
int getSum(n)
{
static int sum=0;
if(n>0)
{
sum = sum + n;
getSum(n-1);
}
return sum;
}
```

(ii) **A program in c to reverse a given number:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,r;
clrscr();
printf("enter an integer");
scanf("%d",&n);
rev(n);
getch();
}
rev (int n)
{
if (n>0)
{
printf ("%d",n%10);
rev(n/10);
}
}
```

b.Explain the memory Allocation in C and distinguish between compile time (static) and run time (dynamic) memory allocation.

Sol:

There are two types of memory allocations in C:
1. Static memory allocation or Compile time
2. Dynamic memory allocation or Run time

In static or compile time memory allocations, the required memory is allocated to the variables at the beginning of the program. Here the memory to be allocated is fixed and is determined by the compiler at the compile time itself. For example

int i, j; //Two bytes per (total 2) integer variables
float a[5], f; //Four bytes per (total 6) floating point variables

When the first statement is compiled, two bytes for both the variable '*i*' and '*j*' will be allocated. Second statement will allocate 20 bytes to the array A [5 elements of floating point type, *i.e.*, $5 \cdot 4$] and four bytes for the variable '*f* '. But static memory allocation has following drawbacks.

If we try to read 15 elements, of an array whose size is declared as 10, then first 10 values and other five consecutive unknown random memory values will be read. Again if we try to assign values to 15 elements of an array whose size is declared as 10, then first 10 elements can be assigned and the other 5 elements cannot be assigned or accessed.

The second problem with static memory allocation is that if we store less number of elements than the number of elements for which you have declared memory, and then the rest of the memory will be wasted. That is the unused memory cells are not made available to other applications (or process which is running parallel to the program) and its status is set as allocated and not free. This leads to the inefficient use of memory.

The dynamic or run time memory allocation helps us to overcome this problem. It makes efficient use of memory by allocating the required amount of memory whenever is needed. In most of the real time problems, we cannot predict the memory requirements.

Dynamic memory allocation does the job at run time.

C provides the following dynamic allocation and de-allocation functions:
(i) malloc( ) (ii) calloc( )
(iii) realloc( ) (iv) free( )

Allocating a block of memory

The malloc( ) function is used to allocate a block of memory in bytes. The malloc function returns a pointer of any specified data type after allocating a block of memory of specified size. It is of the form:

ptr = (int_type *) malloc (block_size)

'ptr' is a pointer of any type 'int_type' byte size is the allocated area of memory block.

For example

ptr = (int *) malloc (10 * sizeof (int));

On execution of this statement, 10 times memory space equivalent to size of an 'int' byte is allocated and the address of the first byte is assigned to the pointer variable 'ptr' of type 'int'. The malloc() function allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a NULL pointer. So it is always better to check whether the memory allocation is successful or not before we use the newly allocated memory pointer.

The following program is to find the sum of n elements using dynamic memory allocation:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
//Defining the NULL pointer as zero
#define NULL 0
void main()
{
int i,n,sum;
//Allocate memory space for two-integer pointer variable
int *ptr,*ele;
clrscr(); //Clear the screen
printf("\nEnter the number of the element(s) to be added = ");
scanf("%d",&n); //Enter the number of elements
//Allocating memory space for n integers of int type to *ptr
ptr=(int *)malloc(n*sizeof(int));
//Checking whether the memory is allocated successfully
if(ptr == NULL)
{
printf("\n\nMemory allocation is failed");
exit(0);
}
//Reading the elements to the pointer variable *ele
for(ele=ptr,i=1;ele<(ptr+n);ele++,i++)
{
printf("Enter the %d element = ",i);
scanf("%d",ele);
}
//Finding the sum of n elements
for(ele=ptr,sum=0;ele<(ptr+n);ele++)
sum=sum+(*ele);
printf("\n\nThe SUM of no(s) is = %d",sum);
getch();
}
```

Q.3a. How a pointer can be used to access the members of a structure? Explain by C programs as examples.

Sol:

Pointers can be accessed along with structures. A pointer variable of structure can be created as below:

```
struct name {
member1;
member2;
.
.
};
```

-------- Inside function -------

```
struct name *ptr;
```

Here, the pointer variable of type struct name is created. Structure's member through pointer can be used in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

The following program shows an example to access structure's member through pointer.

```
#include <stdio.h>
struct name{
int a;
float b;
};
int main(){
struct name *ptr,p;
ptr=&p; /* Referencing pointer to memory address of p */
printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

In the above example, the pointer variable of type struct name is referenced to the address of p. Then, only the structure member through pointer can accessed.

Structure pointer member can also be accessed using -> operator.

(*ptr).a is same as ptr->a

(*ptr).b is same as ptr->b

Accessing structure member through pointer using dynamic memory allocation:

To access structure member using pointers, memory can be allocated dynamically using

malloc() function defined under "stdlib.h" header file.

Syntax to use malloc():

ptr=(cast-type*)malloc(byte-size)

The following C program is an example to use structure's member through pointer using malloc() function:

```
#include <stdio.h>
#include<stdlib.h>
struct name {
int a;
float b;
char c[30];
};
int main(){
struct name *ptr;
int i,n;
printf("Enter n: ");
scanf("%d",&n);
ptr=(struct name*)malloc(n*sizeof(struct name));
/* Above statement allocates the memory for n structures with pointer ptr pointing
for(i=0;i<n;++i){
printf("Enter string, integer and floating number respectively:\n");
scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
}
printf("Displaying Infromation:\n");
for(i=0;i<n;++i)
printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);
return 0;
}
```

Output:

```
Enter n: 2
Enter string, integer and floating number respectively:
Programming
2
3.2
Enter string, integer and floating number respectively:
Structure
6
2.3
Displaying Information
Programming 2 3.20
Structure 6 2.30
```

b.Describe Major File operations with examples as C programs.

Major File Operations:

1. fopen() :-

FILE *fopen(const char *path, const char *mode);

The fopen() function is used to open a file and associates an I/O stream with it. This function takes two arguments. The first argument is a pointer to a string containing name of the file to be opened while the second argument is the mode in which the file is to be opened. The mode can be:

'r' : Open text file for reading. The stream is positioned at the beginning of the file.

'r+' : Open for reading and writing. The stream is positioned at the beginning of the file.

'w' : Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

'w+' : Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

'a' : Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

'a+' : Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.
The fopen() function returns a FILE stream pointer on success while it returns NULL in case of a failure.

2. fread() and fwrite() :-

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

The functions fread/fwrite are used for reading/writing data from/to the file opened by fopen function. These functions accept three arguments. The first argument is a pointer to buffer used for reading/writing the data. The data read/written is in the form of 'nmemb' elements each 'size' bytes long.
In case of success, fread/fwrite return the number of bytes actually read/written from/to the stream opened by fopen function. In case of failure, a lesser number of byes (then requested to read/write) is returned.

3. fseek() :-

int fseek(FILE *stream, long offset, int whence);

The fseek() function is used to set the file position indicator for the stream to a new position. This function accepts three arguments. The first argument is the FILE stream pointer returned by the fopen() function. The second argument 'offset' tells the amount of bytes to seek. The third argument 'whence' tells from where the seek of 'offset' number of bytes is to be done.
The available values for whence are SEEK_SET, SEEK_CUR, or SEEK_END. These three values (in order) depict the start of the file, the current position and the end of the file.
Upon success, this function returns 0, otherwise it returns -1.

4. fclose() :-

int fclose(FILE *fp);

The fclose() function first flushes the stream opened by fopen() and then closes the underlying descriptor. Upon successful completion this function returns 0 else end of file (eof) is returned. In case of failure, if the stream is accessed further then the behavior remains undefined.

The following C program code illustrates the use and description of above mentioned file operations and functions:

```
#include<stdio.h>
#include<string.h>
#define SIZE 1
#define NUMELEM 5
int main(void)
{
FILE* fd = NULL;
char buff[100];
memset(buff,0,sizeof(buff));
fd = fopen("test.txt","rw+");
if(NULL == fd)
{
printf("\n fopen() Error!!!\n");
return 1;
}
printf("\n File opened successfully through fopen()\n");
if(SIZE*NUMELEM != fread(buff,SIZE,NUMELEM,fd))
{
printf("\n fread() failed\n");
return 1;
```

```
}
printf("\n Some bytes successfully read through fread()\n");
printf("\n The bytes read are [%s]\n",buff);
if(0 != fseek(fd,11,SEEK_CUR))
{
printf("\n fseek() failed\n");
return 1;
}
printf("\n fseek() successful\n");
if(SIZE*NUMELEM != fwrite(buff,SIZE,strlen(buff),fd))
{
printf("\n fwrite() failed\n");
return 1;
}
printf("\n fwrite() successful, data written to text file\n");
fclose(fd);
printf("\n File stream closed through fclose()\n");
return 0;
}
```

Q.4a. Describe linear and binary search with their algorithms.

Sol:

Linear Search:

In linear search, each element of an array is read one by one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found.

Algorithm for linear search

Let A be an array of *n* elements, A[1],A[2],A[3], ... A[*n*]. "*data*" is the element to be searched. Then this algorithm will find the location "*loc*" of data in A. Set *loc* = – 1,if the search is unsuccessful.

1. Input an array A of n elements and "data" to be searched and initialise *loc* = – 1.
2. Initialise $i = 0$; and repeat through step 3 if ($i < n$) by incrementing *i* by one.
3. If (data = A[*i*])
(*a*) loc = *i*
(*b*) GOTO step 4
4. If (loc > 0)
(*a*) Display "data is found and searching is successful"
5. Else
(*a*) Display "data is not found and searching is unsuccessful"
6. Exit

Time Complexity

Time Complexity of the linear search is found by number of comparisons made in searching a record.
In the best case, the desired element is present in the first position of the array, i.e., only one comparison is made. So f (n) = O(1).
In the Average case, the desired element is found in the half position of the array, then f (n) = O[(n + 1)/2].
But in the worst case the desired element is present in the nth (or last) position of the array, so n comparisons are made. So f (n) = O(n + 1).

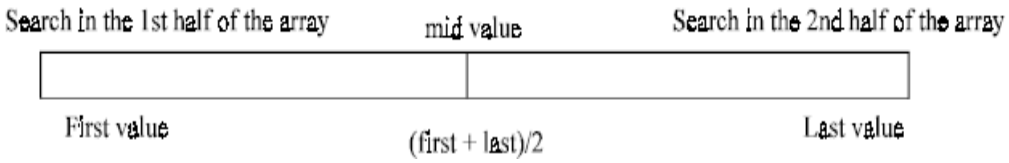
b.Write a C Program code for binary search using recursion.

Binary Search

Binary search is an extremely efficient algorithm when it is compared to linear search. Binary search technique searches "data" in minimum possible comparisons. Suppose the given array is a sorted one, otherwise first we have to sort the array elements.
Then apply the following conditions to search a "data".
1. Find the middle element of the array (i.e., n/2 is the middle element if the array or the sub-array contains n elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
(a) If it is a desired element, then search is successful.
(b) If it is less than desired data, then search only the first half of the array, i.e., the elements which come to the left side of the middle element.
(c) If it is greater than the desired data, then search only the second half of the array, i.e., the elements which come to the right side of the middle element.
Repeat the same steps until an element is found or exhaust the search area.

Algorithm for Binary Search

Let A be an array of n elements A[1],A[2],A[3],..... A[n]. "Data" is an element to be searched. "mid" denotes the middle location of a segment (or array or sub-array) of the element of A. LB and UB is the lower and upper bound of the array which is under consideration.



1. Input an array A of n elements and "data" to be sorted
2. LB = 0, UB = n; mid = int ((LB+UB)/2)
3. Repeat step 4 and 5 while (LB <= UB) and (A[mid] ! = data)

4. If (data < A[mid])
(a) UB = mid–1
5. Else
(a) LB = mid + 1
6. Mid = int ((LB + UB)/2)
7. If (A[mid]== data)
(a) Display “the data found”
8. Else
(a) Display “the data is not found”
9. Exit

The following program illustrates the Binary search using recursion:

```
#include <stdio.h>
binarysearch(int a[],int n,int low,int high)
{
int mid;
if (low > high)
return -1;
mid = (low + high)/2;
if(n == a[mid])
{
printf("The element is at position %d\n",mid+1);
return 0;
}
if(n < a[mid])
{
high = mid - 1;
binarysearch(a,n,low,high);
}
if(n > a[mid])
{
low = mid + 1;
binarysearch(a,n,low,high);
}
}
main()
{
int a[50];
int n,no,x,result;
printf("Enter the number of terms : ");
scanf("%d",&no);
printf("Enter the elements :\n");
for(x=0;x&ltno;x++)
scanf("%d",&a[x]);
```

```
printf("Enter the number to be searched : ");
scanf("%d",&n);
result = binarysearch(a,n,0,no-1);
if(result == -1)
printf("Element not found");
return 0;
}
```

Q.5a. Discuss the concept of Stack and Queues and write the C programs that demonstrate the following:

(i) Push and Pop operations in a stack

(ii) Insert and Delete operations in a Queue.

Sol:

Stack:

A stack is a data structure that organizes data similar to how one organizes a pile of coins. The new coin is always placed on the top and the oldest is on the bottom of the stack. When accessing coins, the last coin on the pile is the first coin removed from the stack. If we want the third coin, we must remove the first two coins from the top of the stack first so that the third coin becomes the top of the stack and we can remove it. There is no way to remove a coin from anywhere other than the top of the stack.

A stack is useful whenever we need to store and retrieve data in last in, first out order. For example, the computer processes instructions using a stack in which the next instruction to execute is at the top of the stack.

Queue:

Queue is a data structure that is used for the application of job scheduling. It is used in batch processing when the jobs are queued –up and executed one after the other they were received.

Queues ignore the existence of priority. There are two different ends called FRONT and REAR end. Deletions are made from the FRONT end. If job is submitted for execution it joins at the REAR end. Job at the front of queue is next to being executed. In queue the insertions are made at one end called rear and the deletion at one end called front. Queue follows the FIFO basis (**First in First Out).**

Deletions are made from the Front end and joins at the REAR end. Job at the FRONT of queue is next to be executed. Some operations performed on queues are:

CREATE Q (Q) → Creates an empty queue

ADD (i,Q) → Add i to rear of queue

DELETE (Q) → Removes the front element

FRONT (Q) → Returns the front element of queue.

ISEMTQ → returns true if queue is empty else false.

(i) PUSH & POP Operations in Stack

```
#include<stdio.h>
#include<conio.h>
struct {
int a[5],top;
}q;
void main()
{
int z,i,t;
clrscr();
for (i=0;i<=5;i++)
{
scanf("%d",&t);
push(t);
}
for(i=1;i<=5;i++)
{
z=pop();
printf("%d",z);
}
}
push(int t)
{
q.a[q.top]=t;
return(q.top++);
}
pop()
{
q.top--;
return(q.a[q.top]);
}
```

(ii) **INSERT and DELETE operations in a queue**

```
#include<stdio.h>
struct queue
{
int a[5],front,rear;
```

```
} q;
void main()
{
int z,i,t;
for(i=1;i<=5;i++)
{
scanf("%d",&t);
insert(t);
}
for(i=1;i<=5;i++)
{
z=delete();
printf("%d",z);
}
}
insert(int t)
{
q.a[q.front]=t;
q.front++;
}
delete()
{
int p;
p=q.a[q.rear];
q.rear++;
return(p);
}
```

b. Briefly describe the Circular Queue. Show the procedure of adding and deleting an element from a circular queue by a C programming function.

Circular Queue:

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Push and pop operation can be performed in circular as:

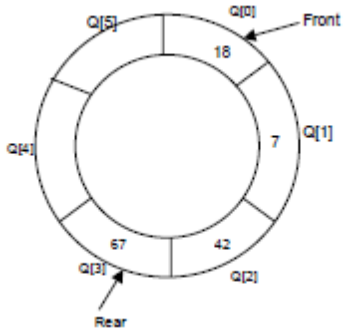


Fig. 1. A circular queue after inserting 18, 7, 42, 67

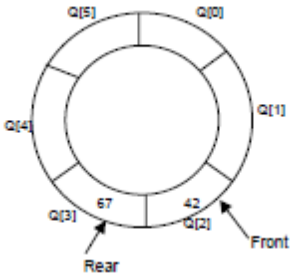


Fig. 2. A circular queue after popping 18, 7

The following is a C program function to add or insert an element to the circular queue:

```
void circular_queue::insert()
{
int added_item;
//Checking for overflow condition
if ((front == 0 && rear == MAX-1) || (front == rear +1))
{
cout<<"\nQueue Overflow \n";
getch();
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;
```

```
cout<<"\nInput the element for insertion in queue:";
cin>>added_item;
cqueue_arr[rear] = added_item;
}/*End of insert()*/
```

The following is a C program function to delete an element from the circular queue:

```
void circular_queue::del()
{
//Checking for queue underflow
if (front == –1)
{
cout<<"\nQueue Underflow\n";
return;
}
cout<<"\nElement deleted from queue is:"<<cqueue_arr[front]<<"\n";
if (front == rear) /* queue has only one element */
{
front = –1;
rear = –1;
}
else
if(front == MAX-1)
front = 0;
else
front = front + 1;
}/*End of del()*/
```

Q.6a.Define a singly Linked List. Write a C Program for appending a new node in the end as well as deleting the beginning or first node of the created linked list.

Sol:

Singly Linked List

Singly linked list is the most basic linked data structure. In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node contains the NULL reference. It has a dynamic size, which can be determined only at run time.

Performance:

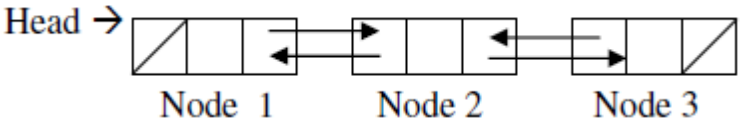
1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited

number of nodes in a fragmented memory environment.

2. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.

Representation of Linked list in Memory:-

Every node has an info part and a pointer to the next node also called as link. The number of pointers is two in case of doubly linked list. for example :



An external pointer points to the beginning of the list and last node in list has NULL. The space is allocated for nodes in the list using malloc or calloc functions. The nodes of the list are scattered in the memory with links to give linear order to the list.

The C Program for appending a new node in the end as well as deleting the beginning or first node of the created linked list is as follows:

```
//Create a single list
struct node
{
int data;
struct node *link
}
struct node *p,*q;
//Appending a node at the end
append(struct node **q,int num)
{
struct node *temp;
temp=*q;
if(*q==NULL)
{
*q=malloc(sizeof(struct node *));
temp=*q;
}
else
{
while(temp _ link!=NULL)
temp=temp_link;
temp_link = malloc(sizeof(struct node *));
temp=temp_link;
}
Temp_data=num;
```

```
temp _link=NULL;
}
//Delete the beginning or first node
delete(struct node *q,int num)
{
struct node *temp;
temp=*q;
*q=temp_link;
free(temp);
}
```

b. Give a C program module that shows the Insert, find, delete and print operations in a Singly Linked list.

Sol:

Insert, find, delete and print operations in a Singly Linked list:

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address in the next field of the new node as NULL.

2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list. Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key. Until next field of the pointer is equal to NULL,

check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

4. Print - function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

The following program module shows the Insert, find, delete and print operations in a Singly Linked list:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
int data;
struct Node *next;
}node;
void insert(node *pointer, int data)
{
/* Iterate through the list till we encounter the last node.*/
while(pointer->next!=NULL)
{
pointer = pointer -> next;
}
/* Allocate memory for the new node and put data in it.*/
pointer->next = (node *)malloc(sizeof(node));
pointer = pointer->next;
pointer->data = data; pointer->next = NULL;
}
int find(node *pointer, int key)
{
pointer = pointer -> next;    //First node is dummy node.
/* Iterate through the entire linked list and search for the key. */
while(pointer!=NULL)
{
if(pointer->data == key) //key is found.
{
return 1;
}
pointer = pointer -> next;//Search in the next node.
}
/*Key is not found */
return 0;
}
```

```
void delete(node *pointer, int data)
{
/* Go to the node for which the node next to it has to be deleted */
while(pointer->next!=NULL && (pointer->next)->data != data)
{
pointer = pointer -> next;
}
if(pointer->next= =NULL)
{
printf("Element %d is not present in the list\n",data);
return;
}
/* Now pointer points to a node and the node next to it has to be removed */
node *temp;
temp = pointer -> next;
/*temp points to the node which has to be removed*/
pointer->next = temp->next;
/*We removed the node which is next to the pointer (which is also temp) */
free(temp);
/* Beacuse we deleted the node, we no longer require the memory used for it .
free() will deallocate the memory. */
return;
}
void print(node *pointer)
{
if(pointer==NULL)
{
return;
}
printf("%d ",pointer->data);
print(pointer->next);
}
int main()
{
/* start always points to the first node of the linked list. temp is used to point to the last node
of the linked list.*/
node *start,*temp;
start = (node *)malloc(sizeof(node));
temp = start;
temp -> next = NULL;
/* Here in this code, we take the first node as a dummy node. The first node does not contain
data, but it used because to avoid handling special cases in insert and delete functions. */
```

```
printf("1. Insert\n");
printf("2. Delete\n");
printf("3. Print\n");
printf("4. Find\n");
while(1)
{
int query;
scanf("%d",&query);
if(query==1)
{
int data;
scanf("%d",&data);
insert(start,data);
}
else if(query==2)
{
int data;
scanf("%d",&data);
delete(start,data);
}
else if(query==3)
{
printf("The list is ");
print(start->next);
printf("\n");
}
else if(query==4)
{
int data;
scanf("%d",&data);
int status = find(start,data);
if(status)
{
printf("Element Found\n");
}
else
{
printf("Element Not Found\n");
}
}
}
```

Q.7 a. Write short notes on the following: (4+4)

(i) Circular linked lists
(ii) Doubly linked lists

Sol:

(i) Circular Linked List

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. A circular linked list is shown in following Fig.1:

.

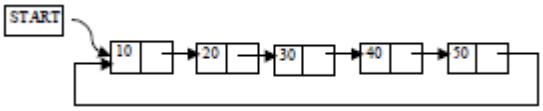


Fig. 1. Circular Linked list

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the following Fig.2.

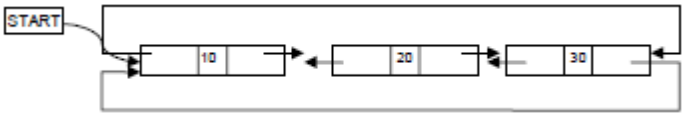


Fig. 2. Circular Doubly Linked list

Advantages:

1. If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node.
2. It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes.

Disadvantages:

1. It is not easy to reverse the linked list.
2. If proper care is not taken, then the problem of infinite loop can occur.
3. If we at a node and go back to the previous node, then we cannot do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node.

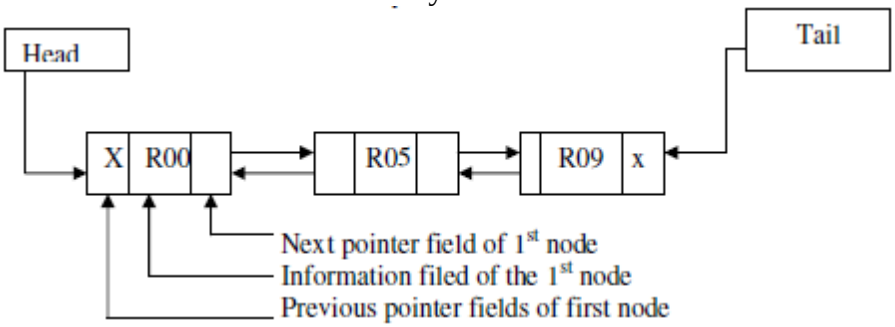
(ii) Doubly Linked List

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node.

The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

Fig.3. : A typical doubly linked list node

In a doubly linked list, also called as 2 way list, each node is divided into 3 parts. The first part is called previous pointer field. It contains the address of the preceding elements in the list. The second part contains the information of the element and the third part is called next pointer field. It contains the address of the succeeding elements in the list. In addition 2 pointer variable HEAD and TAIL are used that contains the address of the 1st element or the address of the last element of the list. Doubly linked list can be traversed in both directions.



Performance

1. The advantage of a doubly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.
2. A doubly linked list can be traversed in two directions; in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning of the list.
3. Given the location of a node 'N' in the list, one can have immediate access to both the next node and the preceding node in the list.
4. Given a pointer to a particular node 'N', in a doubly linked list, we can delete the Node 'N' without traversing any part of the list. Similarly, insertion can also be made before or after 'N' without traversing the list.
5. The disadvantage is that more pointers needs to be handled and more links need to updated.

b.Write a C program which demonstrates the merging of two circular lists.

Sol:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
typedef struct node
{
int data;
struct node *next;
}NODE;
NODE *create_list(NODE *last);
void display(NODE *last);
NODE *add_to_empty(NODE *last,int data );
NODE *add_at_end(NODE *last,int data);
NODE *merge(NODE *last1,NODE *last2);
int main( )
{
NODE *last1=NULL,*last2=NULL;
printf("\ncircular list-1\n===============");
last1=create_list(last1);
printf("\ncircular list-2\n===============");
last2=create_list(last2);
printf("First list is : ");
display(last1);
printf("Second list is : ");
display(last2);
last1=merge(last1, last2);
printf("Merged list is : ");
display(last1);
getch();
return 0;
}
NODE *merge( NODE *last1,NODE *last2)
{
NODE *ptr;
if(last1==NULL)
{
last1=last2;
return last1;
}
if(last2==NULL )
return last1;
ptr=last1->next;
last1->next=last2->next;
```

```
last2->next=ptr;
last1=last2;
return last1;
} NODE *
create_list(NODE *last)
{
int i,n;
int data;
printf("\nEnter the number of nodes : ");
scanf("%d",&n);
last=NULL;
if(n==0)
return last;
printf("\nEnter the element to be inserted : ");
scanf("%d",&data);
last=add_to_empty(last,data);
for(i=2;i<=n;i++)
{
printf("\nEnter the element to be inserted : ");
scanf("%d",&data);
last=add_at_end(last,data);
}
return last;
}
void display(NODE *last)
{
NODE *p;
if(last==NULL)
{
printf("\nList is empty\n");
return;
}
p=last->next; /*p points to first node*/
do
{
printf("%d ", p->data);
p=p->next;
}while(p!=last->next);
printf("\n");
}
NODE *add_to_empty(NODE *last,int data)
{
NODE *tmp;
tmp = (NODE *)malloc(sizeof(NODE));
```

```
tmp->data = data;
last = tmp;
last->next = last;
return last;
}
NODE *add_at_end(NODE *last,int data)
{
NODE *tmp;
tmp = (NODE *)malloc(sizeof(NODE));
tmp->data = data;
tmp->next = last->next;
last->next = tmp;
last = tmp;
return last;
}
```

Output:

Circular List – 1:

Enter the number of nodes : 3
Enter the element to be inserted : 23
Enter the element to be inserted : 25
Enter the element to be inserted : 27

Circular List – 2:

Enter the number of nodes : 2
Enter the element to be inserted : 56
Enter the element to be inserted : 78
First list is : 23 25 67
Second list is : 56 78
Merged list is: 23 25 27 56 78

Q.8 a. Express the non recursive algorithms for the Inorder and Preorder traversal of a binary tree.

Sol:

Non – recursive algorithm for the Inorder traversal of a binary tree:

Initially push NULL onto STACK and then set PTR = ROOT. Then repeat the following steps until NULL is popped from STACK.

i. Proceed down the left –most path rooted at PTR,

pushing each node N onto STACK and stopping when
a node N with no left child is pushed onto STACK.
ii. [Backtracking.] Pop and process the nodes on STACK.
If NULL is popped then
Exit.
If a node N with a right child R(N) is processed,
set PTR = R(N) (by assigning PTR = RIGHT[PTR] and
return to Step(a)).

Non – recursive algorithm for the Preorder traversal of a binary tree:
Initially push NULL onto stack and then set PTR=Root. Repeat the following steps until PTR= NULL.
1. Preorder down the left most path routed at PTR.
2. Processing each node N on the path and pushing each right child
If any onto the stack.[The traversing stops when (PTR)=NULL]
3. Backtracking: pop and assign to PTR the top element on stack.
If PTR not equal to NULL then return to step 1 otherwise exit.

b.Write the following algorithms:

(i) Write the algorithm for testing that a given binary tree is BST (Binary Search tree) or not.
(ii) Express the algorithm of inserting a node k in a BST (binary search tree) with a brief analysis.

Sol (i):
The algorithm for testing that a given Binary tree is BST (Binary Search tree) is as follows:

```
BSTbstree(*tree)
{
while((tree->left !=null)&& (tree->right !=null))
{
if(tree->left < tree->root)
BSTbstree(tree->left);
else
return(1);
if(tree->right > tree->root)
BSTbstree(tree->right);
else
return(1);
}
return(0);
}
```

Sol (ii):
The algorithm of inserting a node k in a BST (binary search tree) is as follows:-

```
/* Get a new node and make it a leaf*/
getnode (k)
left(k) = null
right(k) = null
info (k) = x
/* Initialize the traversal pointers */
p = root
trail = null
/* search for the insertion place */
while p <> null do
begin
trail = p
if info (p) > x
then p = left (p)
else
p = right (p)
end
/* To adjust the pointers */
If trail = null
Then root = k /* attach it as a root in the empty tree */
else
if info (trail) > x
then
left (trail) = k
else
right (trail) = k
```

Analysis

We notice that the shape of binary tree is determined by the order of insertion. If the values are sorted in ascending or descending order, the resulting tree will have maximum depth equal to number of input elements. The shape of the tree is important from the point of view of search efficiency. The depth of the tree determines the maximum number of comparisons. Therefore we can maximize search efficiency by minimizing the height of the tree.

Q.9 a. Distinguish between the Breadth first search (BFS) and Depth first search (DFS) traversal techniques of a graph in detail.

Sol:

Breadth First Search:

This traversal algorithm uses a queue to store the nodes of each level of the graph as and when they are visited. These nodes are then taken one by one and their adjacent nodes are visited and so on until all nodes have been visited. The algorithm terminates when the queue becomes empty.

Algorithm for Breadth First Traversal is as follows:

```
clearq (q)
```

```
visited (v) = TRUE
while not empty (q) do
for all vertices w adjacent to v do
if not visited then
{
insert (w , q)
visited (w) = TRUE
}
delete (v, q);
```

Here each node of the graph is entered in the queue only once. Thus the while loop is executed n times, where n is the order of the graph. If the graph is represented by adjacency list, then only those nodes that are adjacent to the node at the front of the queue are checked therefore, the for loop is executed a total of E times, where E is the number of edges in the graph. Therefore, breadth first algorithm is O (N*E) for linked expression. If the graph is represented by an adjacency matrix, the for loop is executed once for each other node in the graph because the entire row of the adjacency matrix must be checked. Therefore, breadth first algorithm is O (N2) for adjacency matrix representation.

Depth-First Search

The depth-first search traversal of a graph is like the depth-first traversal of a tree. Since a graph has no root, when we do a depth-first traversal, we must specify the vertex at which to begin. Depth-first traversal of a graph visits a vertex and then recursively visits all the vertices adjacent to that node. The catch is that the graph may contain cycles, but the traversal must visit every vertex at most once. The solution to the problem is to keep track of the nodes that have been visited, so that the traversal does not suffer the fate of infinite recursion.

An algorithm for depth first traversal is given below:

```
struct node
{
int data ;
struct node *next ;
} ; int visited[MAX] ;
void dfs ( int v, struct node **p )
{
struct node *q ;
visited[v - 1] = TRUE ;
printf ( "%d\t", v ) ;
q = * ( p + v - 1 ) ;
while ( q != NULL )
{
if ( visited[q -> data - 1] == FALSE )
dfs ( q -> data, p ) ;
else
```

```
q = q -> next ;}
}
```

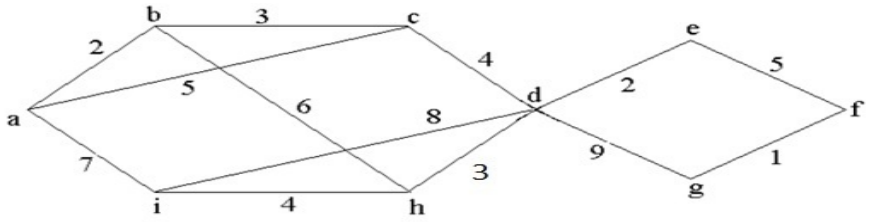
Difference between BFS and DFS

Depth-first search is different from Breadth-first search in the following ways:

1. A depth search traversal technique goes to the deepest level of the tree first and then works up while a breadth-first search looks at all possible paths at the same depth before it goes to a deeper level. When we come to a dead end in a depth first search, we back up as little as possible. We try another route from a recent vertex-the route on top of our stack.

2. In a breadth-first search, we want to back up as far as possible to find a route originating from the earliest vertices. So the stack is not an appropriate structure for finding an early route because it keeps track of things in the order opposite of their occurrence-the latest route is on top. To keep track of things in the order in which they happened, we use a FIFO queue. The route at the front of the queue is a route from an earlier vertex; the route at the back of the queue is from a later vertex.

b.Discuss the Kruskal's Algorithm with its analysis and apply it to find the minimum cost spanning tree for the following undirected graph: (4+4)



Sol:

Kruskal's Algorithm for minimum cost spanning tree

1. Make the tree T empty.
2. Repeat steps 3, 4 and 5 as long as T contains less than n-1 edges and E not empty; otherwise proceed to step 6.
3. Choose an edge (v,w) from E of lowest cost.
4. Delete (v,w) from E.
5. If (v,w) does not create a cycle in T then add(v,w) to T else dicard(v,w).
6. If T contains fewer than n-1 edges then print ('no spanning tree').

Analysis

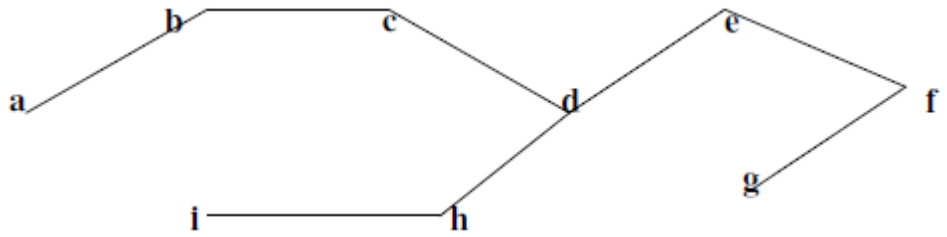
The complexity of this algorithm is determined by the complexity of sorting method applied;

for an efficient sorting it is O(|E|log|E|).It also depends on the complexity of the methods used for cycle detection which is of O(|V|) as only |V|-1 edges are added to tree which gives a total of O(E logV) time. So complexity of Kruskal's algorithm is is O(|V+E|log|V|). As V is asymptotically no larger than E, the running time can also be expressed as O(Elog V).

Application of kruskal's algorithm to find a minimum cost spanning tree on a given graph:

EDGES	COST	STATUS
(g, f)	1	Accept
(a, b)	2	Accept
(d, e)	2	Accept
(b, c)	3	Accept
(d, h)	3	Accept
(c, d)	4	Accept
(i, h)	4	Accept
(e, f)	5	Accept
(a, c)	5	Reject
(b, h)	6	Reject
(a, i)	7	Reject
(i, d)	8	Reject
(d, g)	9	Reject

So minimum cost spanning tree is:



And Minimum cost spanning cost is 2+3+4+2+5+1+3+4 = 24.

Textbook

1. **C & Data Structures, P.S. Deshpande and O.G. Kakde, Dreamtech Press, 2007**