Ba10

Data Structure

01/04/15 Rs=90/-

Data structure :-

① Any kind of inform^n is called data. Structure means representation.

② D.S is a representation of information in primary or disk storage area

③ D.S is a kind of relationship b/w logical related data elements.

④ In D.S's any kind of operations must be required to perform b/w logical oriented elements.

Real Time applications of D.S

ⓐ Compiler designing. (stack memory allocation can be organized in compiler designing)

ⓑ Operating system Design. like memory mgt (Linked list + Hash-map)

ⓒ Database mgt system (B Tree)

ⓓ File system representation of O.S. (Trees)

ⓔ statistical analysis package (Data mining algorithm

ⓕ Network Data model (Graph)

ⓖ Electronic circuit and simulations (Graphs)

In programming languages data structures are classified into two types.

① Linear D.S ② Non-Linear D.S.

① **Linear Data Structure :-**

When we are working with this data structures then all elements required to organized in sequencial manner.

sequencial manner means it is not required to follow always continuous memory location.

② When we are working with linear D.S then sequencial relations are formatted b/w elements.

Ex: Array, stack, queue and LinkList.

② **Non-Linear Data Structure :-**

In this D.S elements are arranged in hierarchical format.

*) when we are working with non-linear D.S then elements are not arranged in sequencial manner.

Ex: Trees, Graphs, Tables

02/01/2015.

Memory management in 'C'.

① In 'C' P.L. we having two types of memory mgt

ⓐ static memory allocation ⓑ Dynamic memory allocation.

(a) Static Memory allocation:-

*) When we are creating the memory at the time of compilation then it is called static memory or compile time memory mgt.

*) When we are working with compile time memory mgt hard coding we require to manage which is not adjustable at run time.

*) When we are working with static memory allocation it is not possible to handle memory acc. to the requirement.

*) When we are working with static memory allocation always memory required to create in continuous memory location only. i.e if memory is not available then we cannot load the program.

*) Insertion and deletion is not possible to perform when we are working with array. i.e static M.A

*) If we required to utilized the memory more effeciently then recommended to go for dynamic memory allocation.

Dynamic Memory Allocation:-

*) It is procedure allocating or dealocating the memory at run time i.e Dynamically.

*) By using DMA we can utilize the memory effeciently acc. to the requirement.

*) When we are working with DMA we require to use - malloc(), realloc(), calloc() functions.

*) Dynamically created memory if we require to release then go for free() fun.

① malloc() :-

*) By using this predefined function we can create the memory dynamically at initial stage.

*) malloc() funn required one argum. of type size_type i.e data type size.

*) malloc() funn. creates the memory in bytes formate and initial value is garbage.

*) In implementation when we are creating element memory then it recommended to go for malloc() funn.

SYNTAX:- void * malloc (size_type);

for duplicate values:

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

int main()
 {
 int* arr, *dump;
 int size, i, j, dumpIndex = 0, flag;
 int currentvalue, count;
 system("CLS");
 printf("Enter array size");
 scanf("%d", &size);
 arr = (int*)malloc(sizeof(int)*size);
 dump = (int*)malloc(sizeof(int)*size);

 printf("Enter %d values:", size);
 for(i=0; i<size; i++)
 scanf("%d", &arr[i]);

 for(i=0; i<size; i++)
 {
  currentvalue = arr[i];
  count=0;
  flag=0;
  for(j=0; j<dumpindex; j++)
  {
   if(currentvalue == dump[j])
   {
    flag=1;
    break;
```

```
}
}
if (flag != 1)
{
for (j = i+1 ; j < size ; j++)
{
 if (currentvalue == arr[j])
   ++count;
 }
 if (count > 0)
 {
 printf("\n%d --> %d", currentvalue, count+1);
 dump[dumpindex++] = currentvalue;
 }
}
}
free(arr);
free(dump);
return EXIT_SUCCESS;
}
```

calloc():—

*) By using this predefined funt we can create the memory dynamically at initial stage.

*) calloc () funt requires two arguments count, size_type.

*) count is indicates no. of elements and size_type is datatype size.

*) calloc() funⁿ will create the memory in blocks format and initial value is '0'.

SYNTAX:- void* calloc (count, ~~sizeof~~type);

Ex:- int* arr;

arr = (int*) calloc (10, sizeof(int)); //20B

size of array.

free(arr);

realloc():- By using this predefined funⁿ we can realloc the memory i.e extending.

) realloc() can takes two arguments i.e void, size_type.

) void indicates previous block address, size_type is data type size.

SYNTAX:- void* realloc (void*, size_type);

Ex:- int* arr;

arr = (int*) calloc (5, sizeof(int)); //10B

arr = (int*) realloc (arr, sizeof(int)*10); //20B

free (arr);

03/04/2015 **LinkedList:**

*) In computers memory a linked list is a linear data structure where all elements are stores in sequence manner.

*) In linked list every record or element holds a pointer which maintain next or previous records inform automatically.

*) Each record of the linked list is called an element or node.

*) Every node contains inform. and one additional field which maintain previous or next record inform.

*) In complete linked list first element is called Head and last node is called Tail.

*) Head always maintain first node inform and Tail will maintain terminal node inform.

*) In 'C' P.L. linked list are classified into 4 types - i.e.

(a) single Linked List
(b) Double Linked List
(c) single circular linked list
(d) double circular linked list

*) count is indicates no. of elements and size_type is datatype size.

*) calloc() fun^n will create the memory in blocks format and initial value is '0'.

SYNTAX:- void* calloc (count, ~~sizeof~~ type);

Ex: int* arr;

arr = (int*) calloc (10, sizeof(int)); // 20B

size of array.

free(arr);

realloc():- By using this predefined fun^n we can realloc the memory i.e extending.

) realloc() can takes two arguments i.e void, size_type.

) void indicates previous block address, size_type is data type size.

SYNTAX:- void* realloc (void*, size_type);

Ex: int* arr;

arr = (int*) calloc (5, sizeof(int)); // 10B

arr = (int*) realloc (arr, sizeof(int)*10); // 20B

free(arr);

03/01/2015

LinkedList.

*) In computers memory a linked list is a linear data structure. where all elements are stores in sequence manner.

*) In linked list every record or element holds a pointer which maintain next or previous records inform automatically.

*) Each record of the linked list is called an element or node.

*) Every node contains inform. and one additional field which maintain previous or next record inform.

*) In complete linked list first element is called Head. and last node is called Tail.

*) Head always maintain first node inform and Tail will maintain terminal node inform.

*) In 'C' P.L. linked list are classified into 4 types - i.e

(a) single Linked List
(b) Double linked List
(c) single circular linked list
(d) double circular linked list

Advantages of Linked lists:-

1. Linked list is a dynamic D.S so it can grow or shrink at run time.
2. On linked list insertion and deletion operatile can perform easily.
3. Effecient memory utilization i.e no any free allocation is required like array.
4. Linear data structure such as stacks and queues can be implemented using linked list
5. Faster access time without memory over-head on compiler in circular linked list only (double circular linked)

Drawbacks of Linked list:

1. Extra memory is required for storing next or previous node address.
2. It is not possible to acess elements randomly. we need to travel in sequence only.
3. When we are working with linked list every element doesnt creates the memory in continuous memory location.

Difference b/w Arrays and Linked List:

1. Arrays and Linked list are having four main difference.

(1) Element Access. (3) Insertion or Deletion
(2) memory structure (4) Memory allocation.

(1) When we are working with arrays elements can be access randomly with the help of index's becoz memory is created in continuous memory location but in linked list elements required to access in sequence becoz elements are stored in random memory location.

(2) When we are working with arrays it allocates the memory in stack and it having continuous memory locations. but in linked list it creates the memory in heap and it is random memory locations.

(3) Arrays doesnt allows insertion and deletion operation but linked list are allows insertion and deletion also.

(4) Generally when we are working with arrays we are using compile time memory mangt. but in linked list always dynamic memory mngt only.

Structure:- A structure is collection of different types of data elements in a single entity.

*) By using struc. we can create user-defined data type which is not available in programming lang.

*) The size of structure is sum of all memember variable size required to calculate.

SYNTAX:

```
struct tagname
{
Datatype1 mem1;
Datatype2 mem2;
Datatype3 mem3;
};
```

Ex:

```
struct emp
{
int id;
char name[36];
float sal;
};
sizeof(struct emp) --> 42 B (2+36+4)

struct abc
{
int id;
char ~~chae~~;
float f;
};
sizeof(struct abc) --> 7B.
```

① Self-referential structure:

ⓐ When we are placing structure type pointer as a member to same structure it is called self-referential structure.

ⓑ Any type of datastructure require to imple-ment using self-referential struct. only.

ex.1)

```
struct slink
{
int data;
struct slink *next;
};
```

ex-2)

```
struct dlink
{
struct dlink *prev;
int data;
struct dlink *next;
};
```

When we are working with structure we require to use following operators i.e

① struct to member (.)

② pointer to member (->)

*) struct to member is required if it is normal variable and pointer to member is required if it is pointer variable.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct emp
 {
 int id;
 char name[30];
 int sal;
 };
 int main()
  {
  struct emp e1;
  struct emp *ptr;
 ptr->id = 101  // e1.id = 101;
 strcpy(ptr->name, "Rajesh"); //strcpy(e1.name,"Rajesh"
 ptr->sal = 12500;  // e1.sal = 12800;
//printf("\nID"%d NAME : %s SAL : %d." e1.id , e1.name,
                                          e1.sal);
printf("\nID:%d NAME : %s SAL : %d." ptr->id, ptr->name
                                        ptr->sal);
  getch();
  return 0;
 }
```

Single Linked List: 5/01/2015

*) In single linked list every node having two field only i.e data and pointer to next node.

*) Data holds information of the ~~hold~~ node that can be premitive or user-defined data type.

*) pointer to next node is a address type which maintains next node address.

*) First node of the list is called and it is a dedicated element always points to first list node only.

*) When we are working with single linked list last node next pointer is always null.

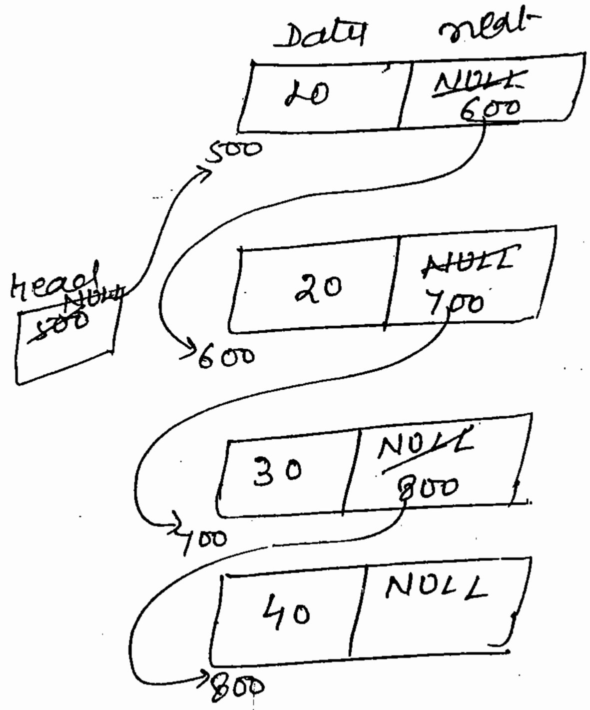
*) When we are working with S.L.L always sequential travelling only possible i.e no directe access.

Structure of single linked list:

Data	Pointer
information	Address of next node

single linked list

Logical Representation of S.L.L with integer datatype




```
struct slink
{
int data;
struct slink* next;
};
```

Implementation of single linked list:-

*) In single L.L we can perform following operations -

1) Adding the node.
2) Travelling in Linked List
3) Counting no. of nodes.
4) Inserting new node.
5) deletion of a node.
6) updation of a node.
7) Reversal of a list

procedure for Add node method.

①*) If head status is null then perform following task.

① create a new node and stores the address into head.

② Read the data and stored into head → data.

③ make it null at head → next location.

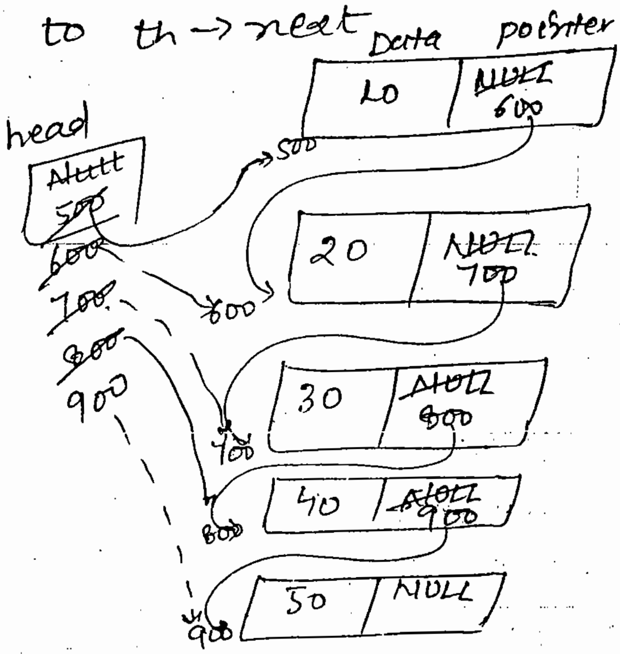
② If head value is other then null then perform following task.

① create a temporary head and assign current value to 'th'

② create a iteration statement and travelling upto tail position i.e. th → next become null

③ Create a new node and stores the address th → next th → next

④ shift the temporary head to newly constructed node then read data in th → data and assign null value to th → next



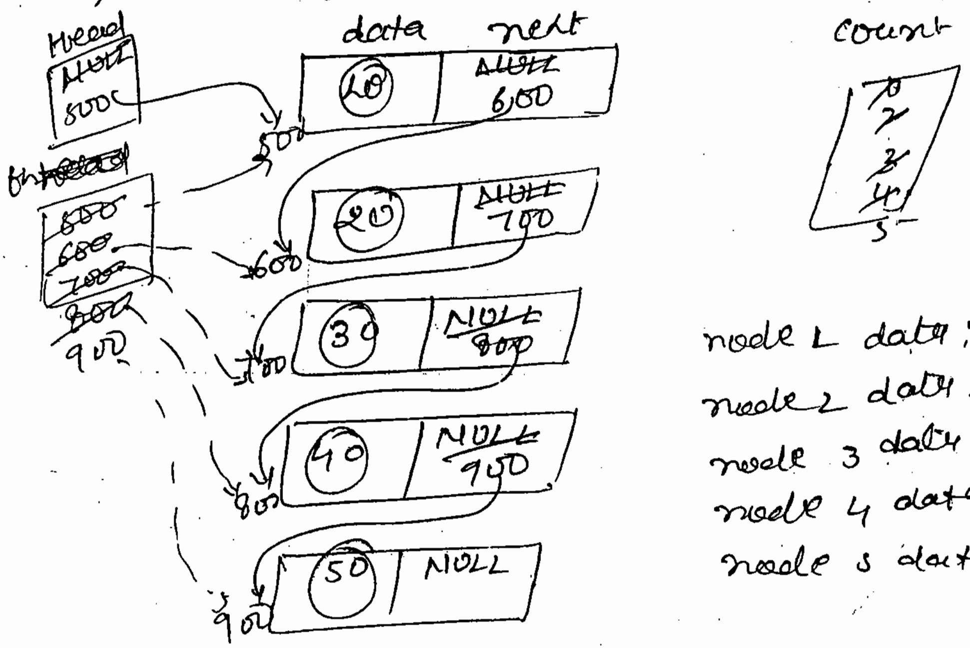
Procedure of Travelling in S.L.L:

1. If head status is null then it is called empty list
2. If the head status is other than null then perform following steps

(a) Take a temporary head and assign the ~~temporary~~ current value to th.

(b) Designing iterative statement until 'th' value becomes null.

(c) In order to travel for every iteration read the data from node and shift th→next.



node 1 data: 10
node 2 data: 20
node 3 data: 30
node 4 data: 40
node 5 data: 50

06/01/2015 Procedure of node counting in single L.L.

1. If head status is Null then return value is '0' i.e empty linked list
2. If head status is other than null then take counter and initialized with zero.
3. Design a iterative statement and assign current head value to temporary

head and travel until th value become null, for every repetation increment the count value by 1.

Procedure of Insert Node:

*) when we are performing insertion operation we require to handle 3 cases

Case-1:) If head status is null no need to perform insert operation.

Case-2) If head status is other than null and link position is 1. then perform following procedure.

Step-1) Create a new node and assign the address to th1.

2) Read the data into th1 and assign ~~th1->next to~~ th1->next value to current head. i.e

th1->next = head;

(3) current head Required to shift to newly constructed node.

Case-3) If user is selected any other position except 1 then perform following tasks.

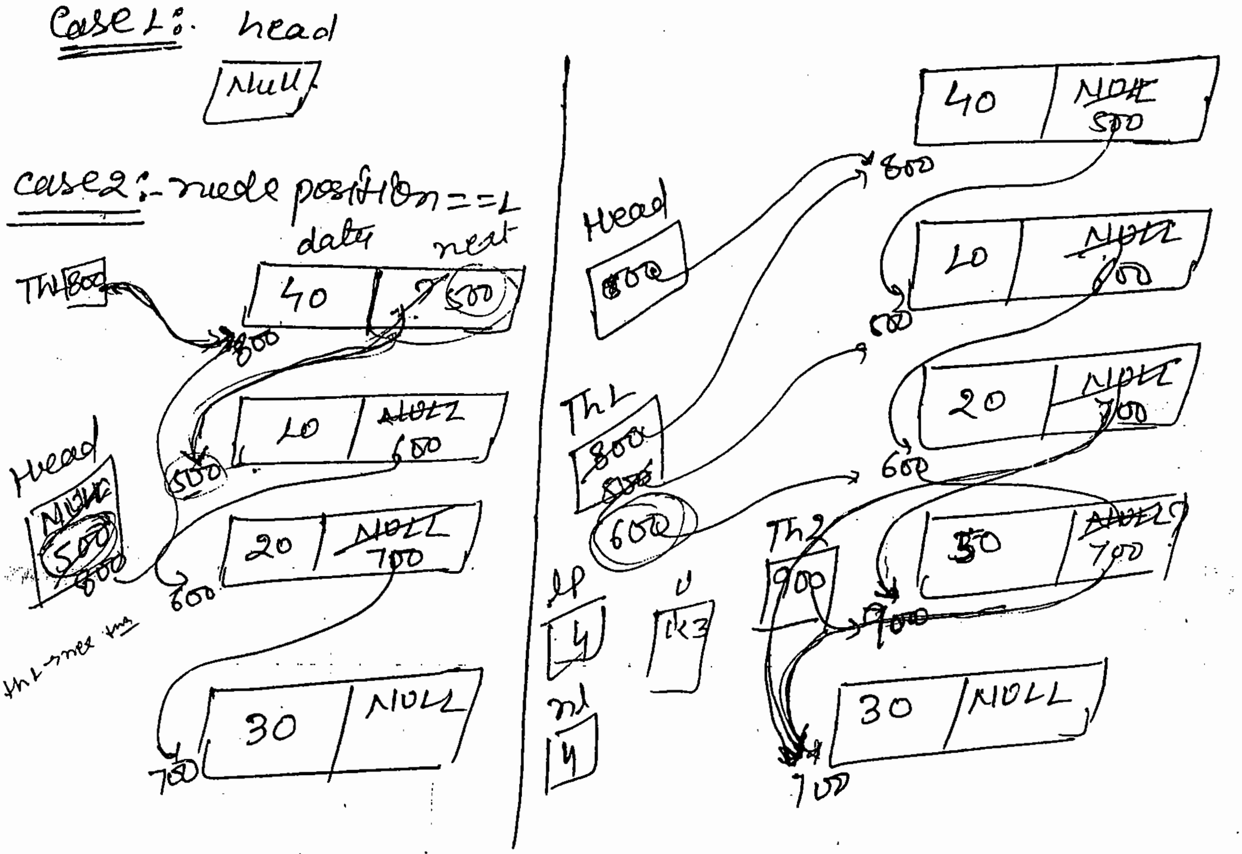
(1) Take a temporary head Th1 and assign current head value.

(2) Design a iterative statement and traverse upto linkposition-1 i.e just before actual position

*) Create a new node and assign the address 'th2'.

*) Read the data into 'th2' and assign th2->next value to th1->next (through this statement newly created node will points to next node in previous order)

*) Assign th1->next value to th2 (through this statement we are making the relation of newly constructed node to old order previous node.



Procedure to Delete:

*) When we require to perform deletion operation then we need to handle 4 cases. i.e

case1:) if head status is null list is empty.

case2:) if no. of nodes are 1 then delete head and reassign null value.

case3:) when we are deleting at position 1.

(a) Take a temporary head thr and assign current head value.

(b) Shift the current ~~node~~ head to next node i.e head = head->next;

(c) Assign thr to next value to null and delete thr.

case-4:- Deletion of any position

(a) Take a temporary head thr and assign the current head value.

(b) Design a iterative statement and stop at upto position -1 i.e just before deletion node.

(c) Take temporary head. ~~tie~~ thr2 and point to deletion node. i.e thr2 = thr->next;

(d) change previous node to next value to upcoming next node. i.e thr->next = thr2->next. (This statement will makes the relation b/w just before deletion node with next node after deletion.

(e) Make thr2 independent and delete.

case1: Head

| null |

case2:

Head

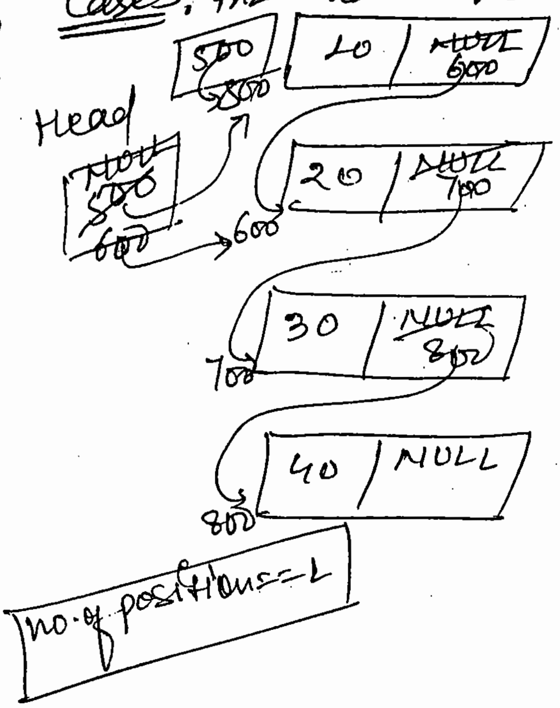
| 800 NULL |

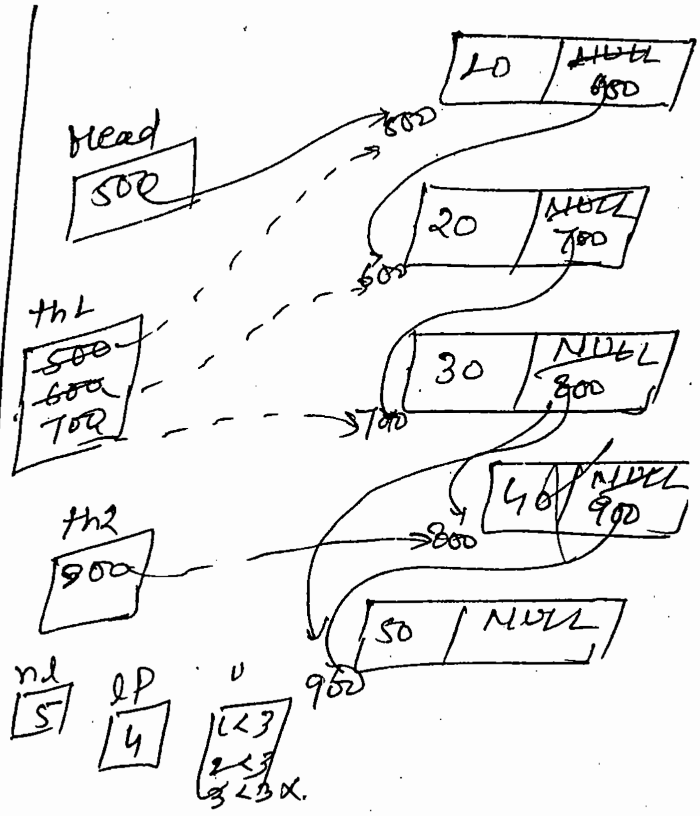
800

data	next
20	NULL

no. of node == 1

case3: th1 data new





07/01/15: procedure to update node data:

*) If the head status is null then it is called empty list.

*) If head status is other than null then perform following operations.

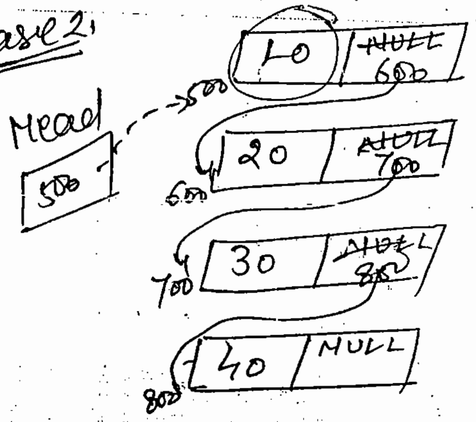
(a) If link position is one then update head data directly.

(b) If link position is other than ~~null~~ one then travel to corresponding link position then update.

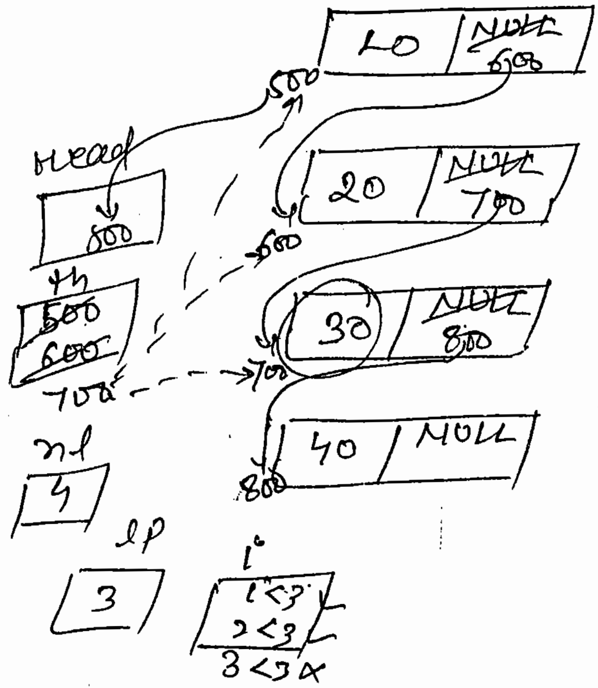
case-1: Head

NULL

case2:



Case ③



*) If Head status is null then no need to be perform any operation.

*) If Head status is other then null then perform following task.

ⓐ Create 3 ~~temporary~~ pointers ~~heads~~ with the name, prev, current, and next and assign the prev. value to null.

ⓑ Designing iterative statement until the current value become null and perform following task.

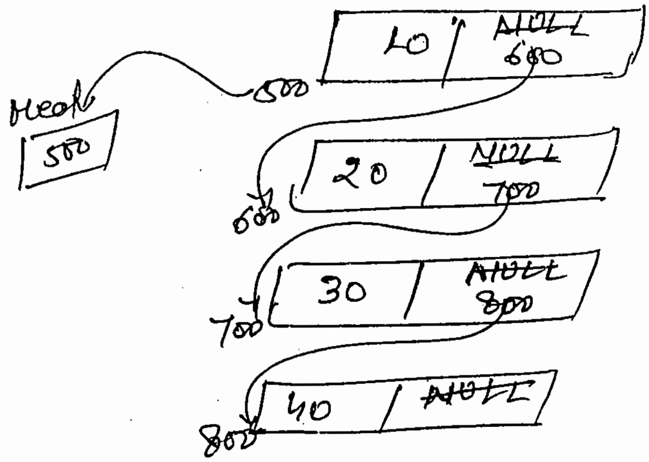
① ~~Current~~ Assign ~~the~~ current->next value to next pointer.

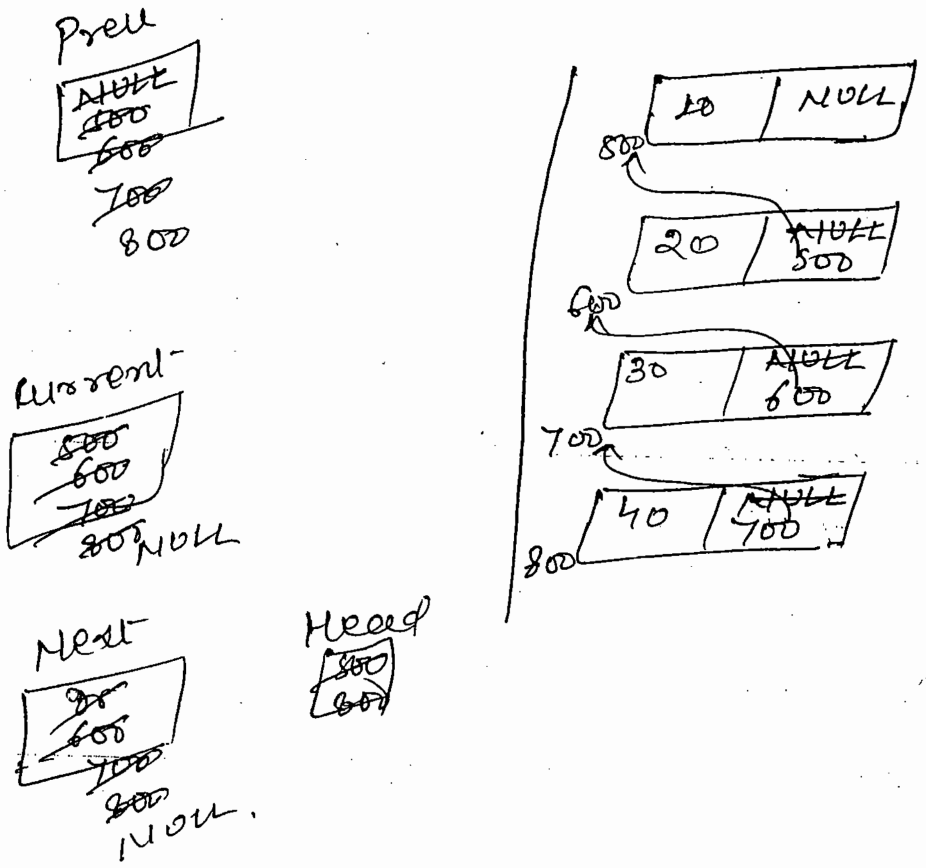
② Assign the previous value to current -> next.

③ Assign the current value to previous.

④ Assign the ~~next~~ current value to current

4) When the loop is terminated ~~then~~ assign the previous value to head.





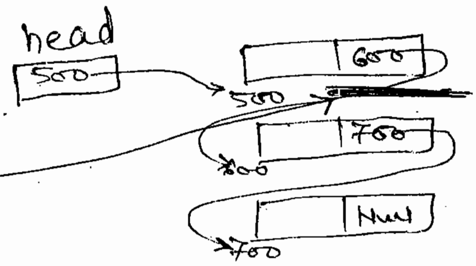
```
#include<stdio.h>
#include<stdlib.h>
#include <dos.h>
#include <malloc.h>

struct slink
{
 int data;
 struct slink *next;
};
struct slink*head=NULL;
void addnode( )
{
 int value;
 struct slink*th;
 if (head=NULL)
 {
 head = (struct slink *)malloc (sizeof(struct slink));
 printf ("\n Enter data:");
 scanf ("%d", &value);
  head->data = value;
  head->next = NULL;
  return;
 }
 else
 {
 th = head
   while (~~th->next!=NULL~~)   while(th->next!=NULL)
                                th = th->next;
```

for adding.

```
th->next = (struct slink*)malloc(sizeof(struct slink));
    th = th->next;   // next address
 printf ("\n Enter data: ");
 scanf ("%d", &value);
  th->data = value;
  th->next = NULL;
  return;
  }
  }
 void displaynode()
 {
  int count = 1;
  struct slink *th;
  if (head == NULL)
   {
    printf ("List is EMPTY");
    system ("PAUSE");
    return;
   }
   th = head;
   do
   {
   printf ("node %d data: %d \n", count, th->data);
   th = th->next;
   ++count;
   } while (th != NULL);
   system("PAUSE");
   return;
   }
```

```
int nodecount()
{
 int count = 0;
 struct slink *th;
 if (head == NULL)
   return count;
 th = head;
 do
 {
  ++count;
  th = th->next;
  } while(th != NULL);
 return count;
 }
void insertnode( )
{
 struct slink *th1, *th2;
 int nl, lp, i;
 if (head == NULL)
 {
  printf ("List is empty");
  system("PAUSE");
  return;
  }
  printf("Enter link position: ");
  scanf ("%d", lp);
 nl = node count();
 if (lp < 1 || lp < nl)
```

```
{
printf ("Invalid link position\n");
system ("PAUSE");
return;
}
if (lp==1)
{
th1 = (struct slink*)malloc(sizeof(struct slink));
printf ("Enter data : ");
scanf ("%d", &th1->data);
th1->next = head;
head = th1;
return;
}
th1 = head;
for (i=1; i<lp-1; i++)
th1 = th1->next;
th2 = (struct slink*)malloc (sizeof (struct slink));
printf ("\n Enter data ");
scanf ("%d", &th2->data);
th2->next = th1->next;
th1->next = th2;
return;
}
```



```
void deletenode()
{
struct slink *th1, *th2;
 int i, lp, nl;
 if (head==NULL)
 {
  printf("List is empty\n");
  system("PAUSE");
  return;
 }
printf("Enter link position: ");
scanf("%d",&lp);
 nl= nodecount();
 if (lp<1 || lp>nl)
 {
 printf("\n Invalid link position \n");
  system("PAUSE");
  return;
 }
 if (nl==1)
 {
          free(head);
          head =NULL;
          return;
        }
  if(lp==1)
  {
```

```
 th1 = head;
head = head->next; // head = th1 -> next;
 th1 -> next = NULL;
 free(th1);
 return;
}
th1 = head;
for (i=1; i<lp-1; i++)
th1 = th1->next;
th2 = th1->next;
th1->next = th2->next;
th2->next = null;
free(th2);
return;
}
void updatenode ()
{
 int nl, lp, i;
 struct slink *th;
 if (head = =NULL)
 {
  Printf ("List is empty:");
  system ("PAUSE");
  return;
 }
 printf (" Enter link position ");
 scanf ("%d", &lp);
 nl = nodecount();
```

```
if (lp < 1 || lp > nl)
 {
 printf ("Invalid line position\n");
 system ("PAUSE");
 return;
 }
if ( lp == 1)
 {
 printf ("\nEnter data: ");
 scanf ("%d", &head-> data);
 return;
 }
 th = head;
for ( i=1; i< lp; i++)
 th = th->next;
 printf ("\nEnter new data: ");
 scanf ("%d", & th->data);
 return;
 }
void reversenode()
{
struct slink * prev = null;
struct slink * current = head;
struct slink * next ;
 while (current != NULL)
 if (head == NULL)
  {
```

```
printf ("List is empty\n");
system ("PAUSE");
return;
}
while(current != NULL)
{
next = current -> next;
current->next = prev;
prev = current;
current = next;
}
head = prev;
}
int main()
{
int option;
while (1)
{
~~clrscr();~~ system ("CLS");
printf ("\n1 FOR ADD NODE! ");
printf ("\n2 FOR DISPLAY : ");
printf ("\n3 FOR NODE COUNT : ");
printf ("\n4 FOR INSERT COUNT NODE : ");
printf ("\n5 FOR DELETE NODE: ");
printf ("\n6 FOR UPDATE NODE: ");
printf ("\n7 FOR Reverse node: ");
printf ("\n8 FOR Exit : ");
```

```
scanf("%d", &option);
{
switch (option)
{
case 1: addnode();
        break;
case 2: displaynode();
        break;
case 3: printf("NODE count=%d\n",
        nodecount());
        system("PAUSE");
        break;
case 4: insertnode();
        break;
case 5: deletenode();
        break;
case 6: updatenode();
        break;
case 7: reversenode();
        break;
case 8; ~~exit(0);~~ free(head);
  return exit_success;
} // switch
} // while
} // main
```

Double Linked List :-

*) When we are working with DLL every node having 3 fields. i.e two fields are address type and another field is data.

*) First address field maintain previous node inform. if it is exist. and data maintain inform^n and next address field maintain next-node inform^n in linked list.

*) when we are working with D.L.L both end travelling is possible i.e first to last and last to first (Travelling last to first is time waste process).

*) In D.L.L also we can't access elements directly i.e sequenenal travelling only.

*) when we are working with D.L.L also linear relationship will forms. but it is bi-direction.

STRUCTURE OF DOUBLE Linked LIST :-

```
struct dlink
 {
 struct dlink *prev.
 int data;
 struct dlink * next;
 };
```

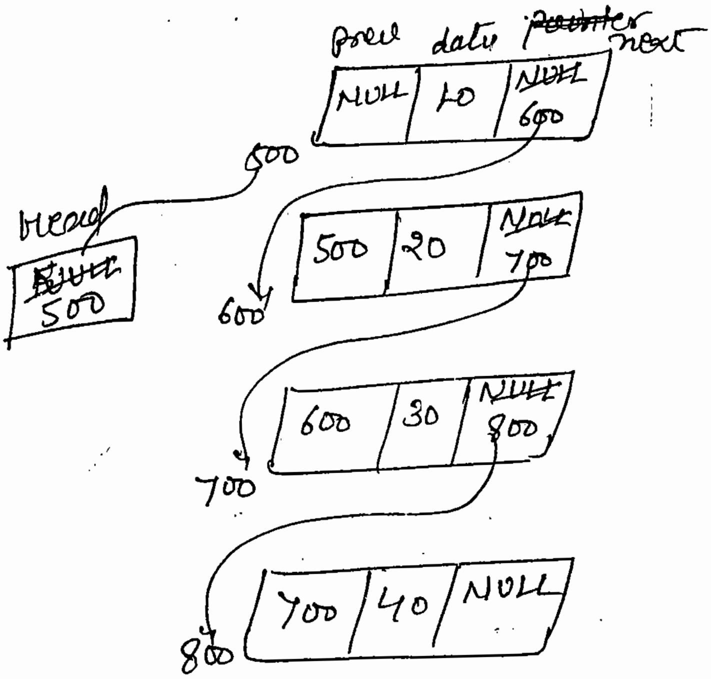
~~pointer~~ previous data ~~pointer~~ next

previous node address	information	next node address

Logical Representation of D.L.L:-

08/01/15 -

*)



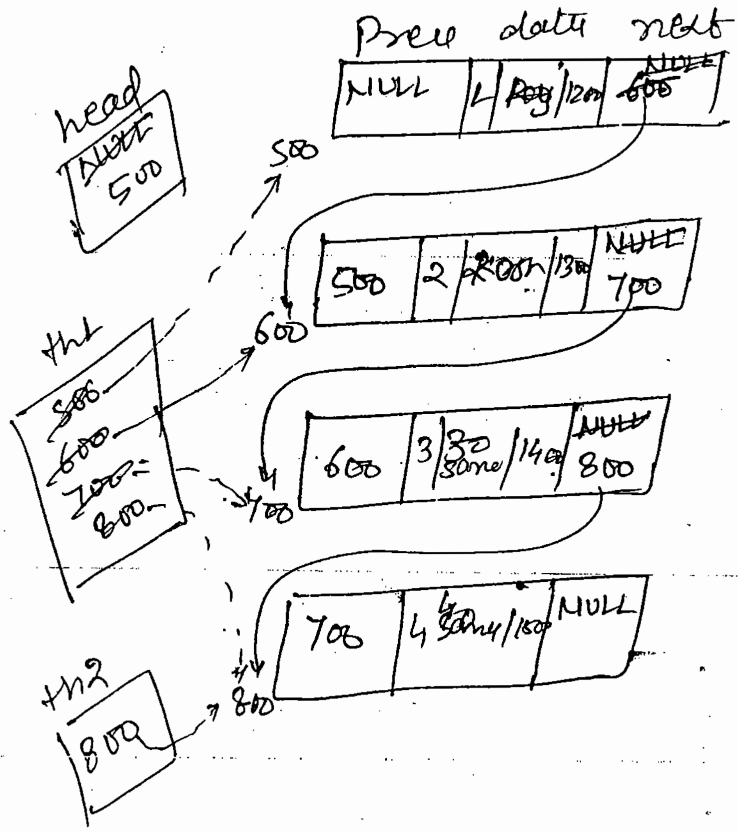
Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <malloc.h>
typedef struct
{
int id;
char name[36];
float sal;
} EMP; // just alias name
EMP getdata();
{
EMP te;
printf("\nEnter ID:");
```

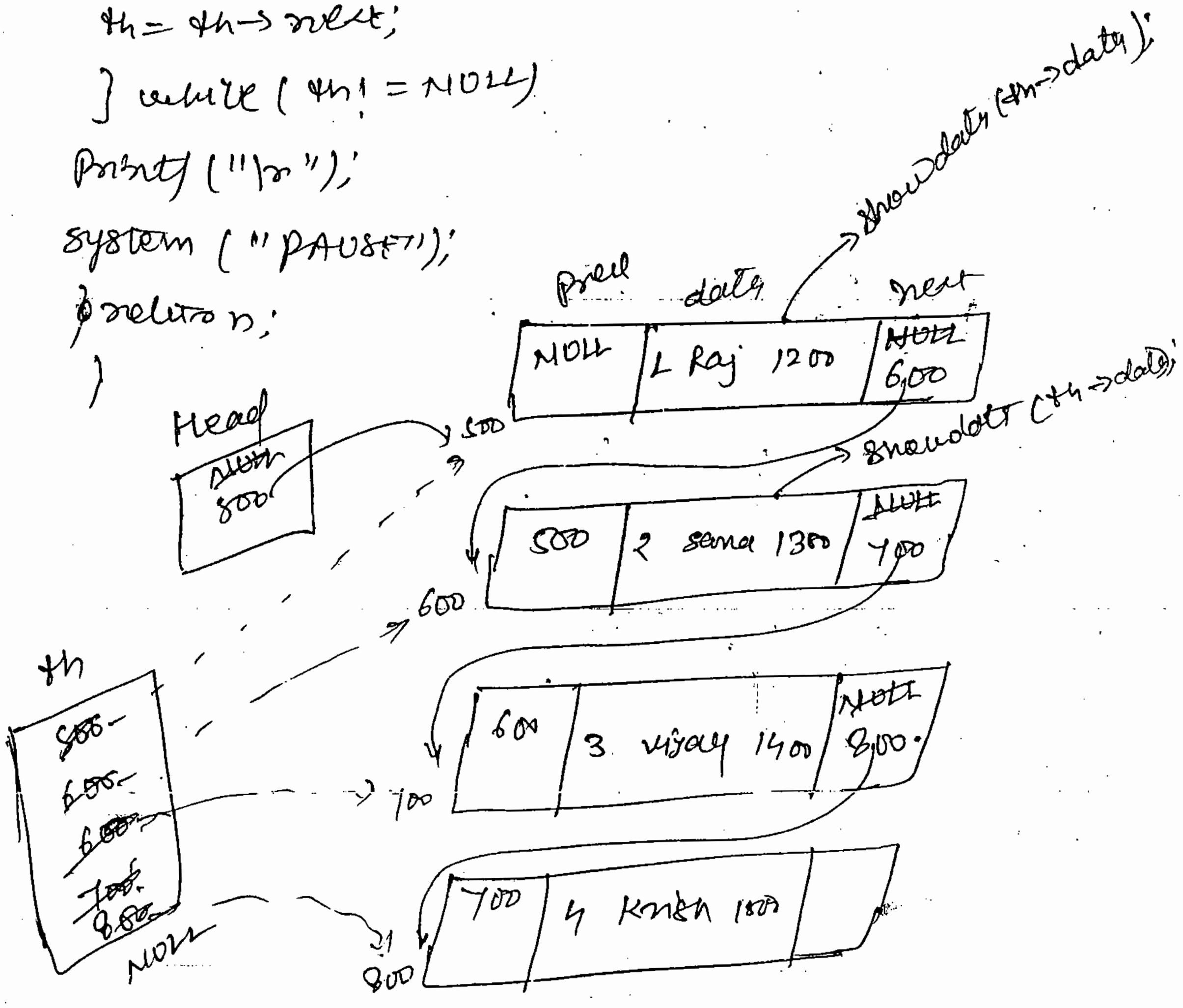
```
scanf ("%d", & te.id);
printf ("\n Enter name : ");
fflush(stdin);
Scanf ("%d
    gets (te.name);
printf ("\n Enter salaray: ");
 scanf ("%d", &sal);
 return te;
 }
 Emp  updatedata ()
 {
 Emp te;
printf ("\n Enter new ID: ");
scanf ("%d", &te.id);
printf ("\n Enter new name: ");
fflush(stdin);
gets (te.name);
printf ("\n Enter new salary: ");
scanf ("%d", & te.sal);
return te;
}
void  showdata (Emp te)
 {
 printf ("\n ID: %d Name: 5%s sal : 3%d",
                 te.id, te.name, te.sal);
  }
```

```
struct dlink
{
struct dlink* prev.
EMP data;
struct dlink* next;
};
typedef struct dlink LINK; //Just alias name.
LINK *head = NULL;
void addnode()
{
char ch;
LINK *th1, *th2;
if(head ==NULL)
{
head =(LINK*)malloc(sizeof(LINK));
head->prev = NULL;
head->data = getdata();
head->next = NULL;
printf("\n Do you want to continue y?");
fflush(stdin);
ch= getchar();
if (ch!='Y' || ch!='y')
return;
}
th1 = head;
while(th1->next != NULL)
th1= th1->next;
do
```

```
th2 = (LINK*)malloc(LINK));
 th2->prev = th1;
 th2->data = getdata();
 th2->next = NULL;
 th1->next = th2;
   th1 = th2;
printf("\n Do you want to continue Y?");
 fflush(stdin);
 ch = getchar();
}while(ch=='Y' || ch=='y');
 th1 = th2 = NULL;
 return;
 }
```



```
void displaynode()
{
LINK *th
if (head == NULL)
{
printf ("LIST IS EMPTY!");
system ("PAUSE");
return;
}
th = head;
do
{
showdata( th->data);
th = th->next;
} while ( th != NULL)
printf ("\n");
system ("PAUSE");
return;
}
```

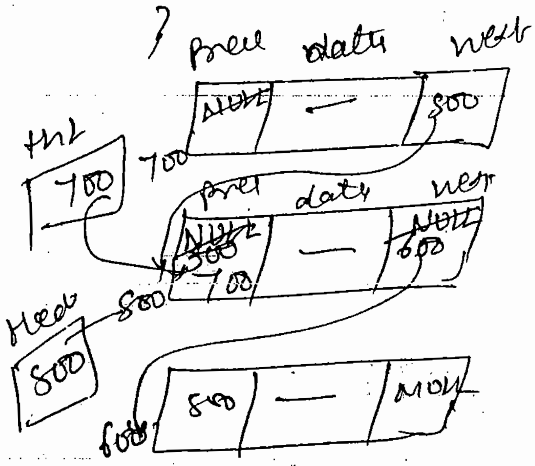


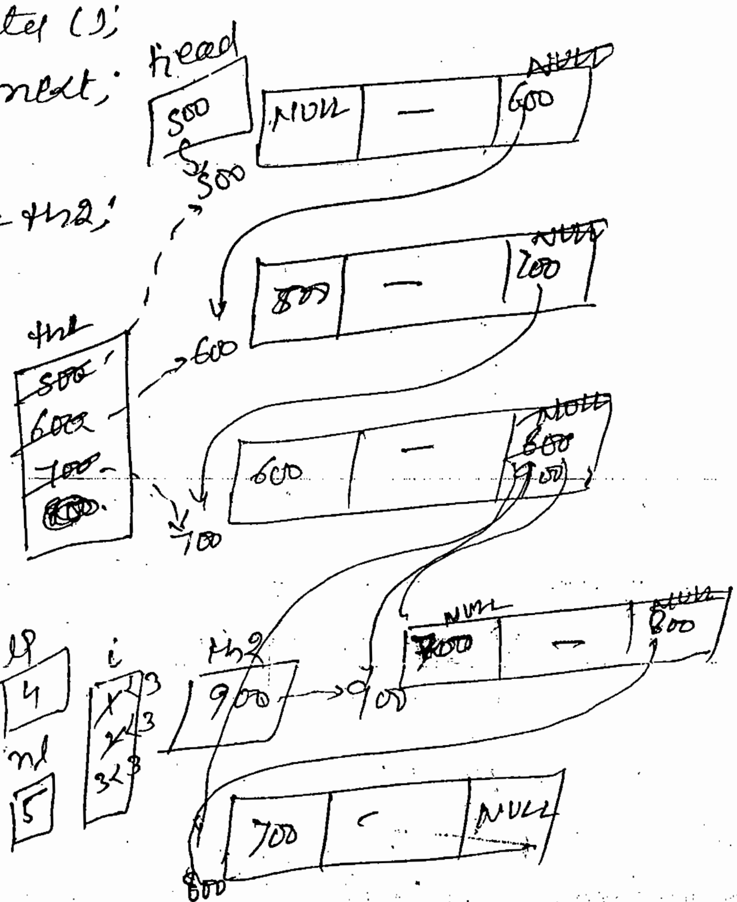
```
int nodecount()
{
 int count=0,
    LINK*th;
 if (head==NULL)
  return count; // default value =0
  do
  {
  ++count;
  th= th->next;
  } while(th!=NULL);
  return count;
 }
```

09/01/15:

```
 void insertnode()
  {
   int lp, nl, i;
   LINK *th1, *th2;
   if (head==NULL)
    {
    printf ("list is empty");
    system ("PAUSE");
     return;
    }
   printf ("Enter the link position");
   scanf ("%d", &lp);
   nl= nodecount();
   if(lp<1 || lp>nl)
    {
    printf (" invalid link position");
     system ("PAUSE");
    return;
```

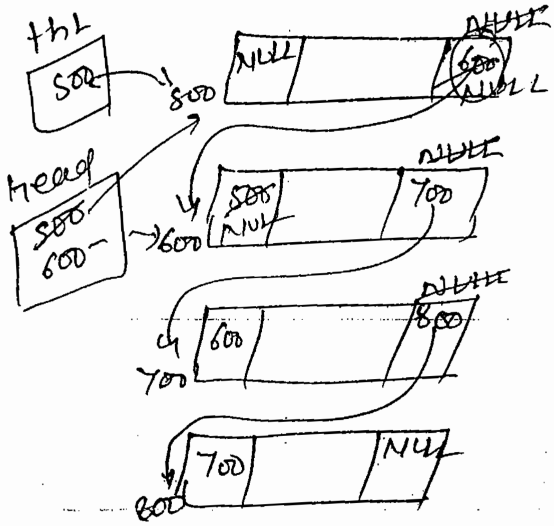
```
if (~~he~~lp==1)
{
th1 = (LINK*)malloc(sizeof(LINK));
th1->prev = NULL;
th1->data = getdata();
th1->next = head;
head->prev = th1;
head = th1;
return;
}
th1 = head;
for(i=1; i<lp-1; i++)
th1 = th1->next;
th2 = (LINK*)malloc(sizeof(LINK));
th2->prev = th1;
th2->data = getdata();
th2->next = th1->next;
th1->next = th2;
if(nl!=lp)
th2->next->prev = th2;
th1=th2=NULL;
return 0;
}
```

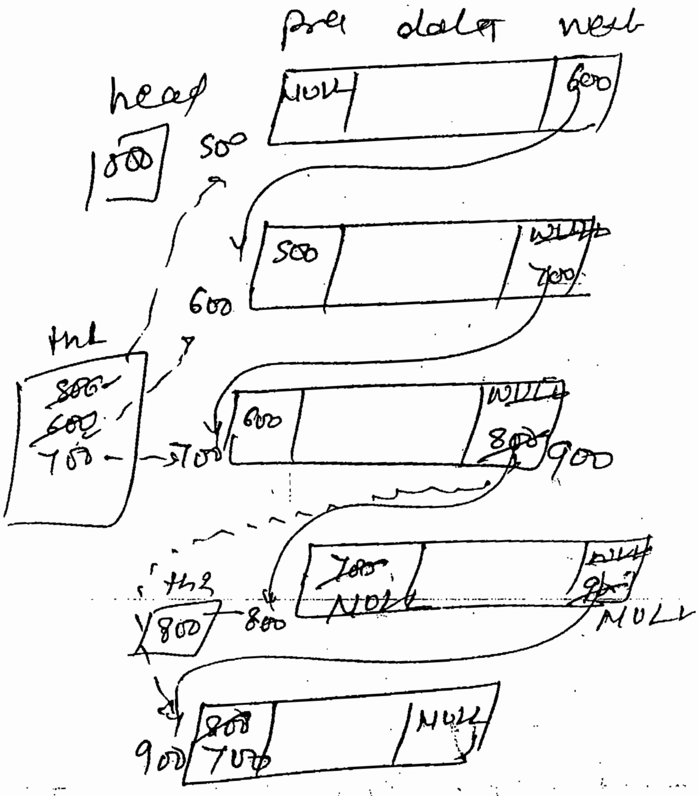




```
void deletenode()
{
LINK*th1,*th2;
int i, lp, nl;
if(head==NULL)
{
printf("List is empty");
system("PAUSE");
return;
}
printf("Enter the link position");
scanf("%d",&lp);
nl=nodecount();
if(lp<1|| lp>nl)
{
printf("Invalid link");
system("PAUSE");
return;
}
if(nl==1)
{
free(head);
head=NULL;
return; system("PAUSE");
}
if(lp==1)
{
th1=head;
head=head->next;
head->prev=NULL;
```

```
free (th1);
th1 = NULL;
return;
}
th1 = head;
for (i=1 ; i<lp-1 ; i++)
th1 = th1->next

th2 = th1->next
th1->next = th2->next
if ( nl != lp)
th2->next->prev = th1; // th2->next->prev = th2->prev
th2->prev = NULL;
th2->next = NULL;
free(th2);
th1 = th2 = NULL;
return;
}
```

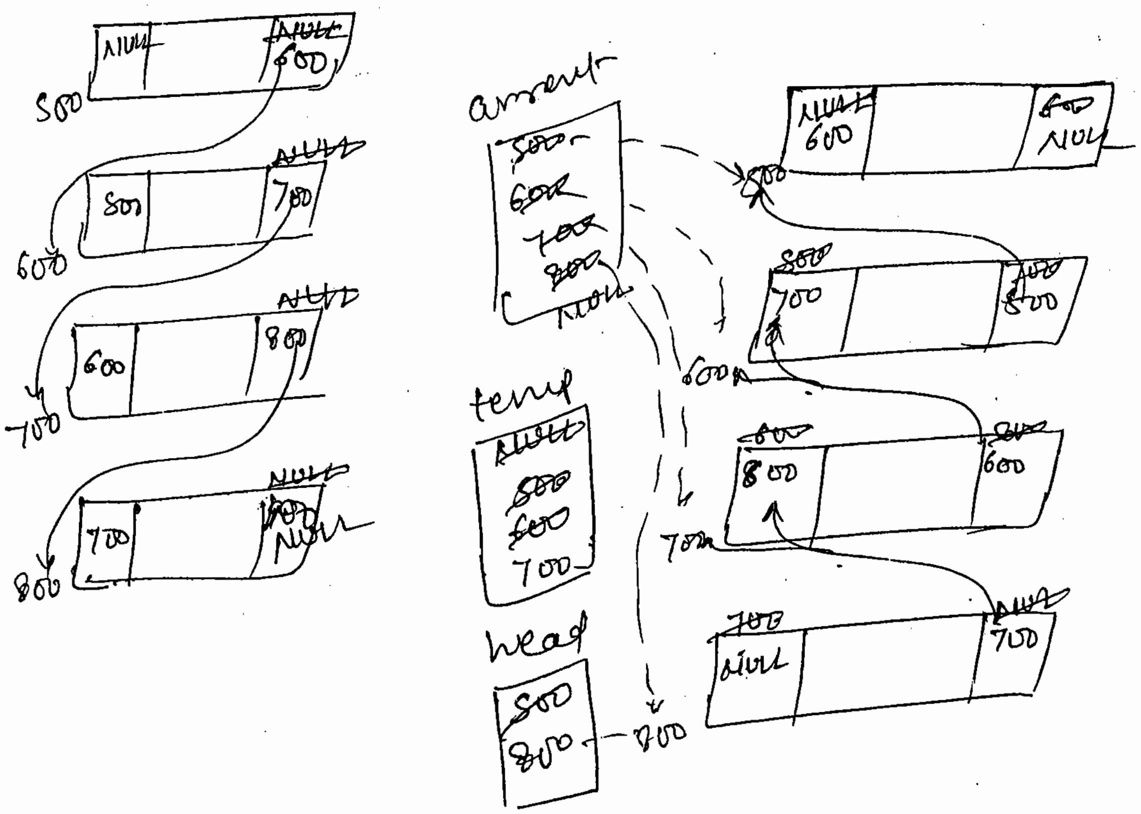




10/01/2015

```
void updatenode()
{
LINK *thl;
int nl, lp, i;
if (head==NULL)
{
printf ("list is empty");
system ("PAUSE");
return;
}
printf ("Enter the update position");
scanf ("%d", & lp)
n = nodecount();
if (lp<1 || lp > nl)
{
printf ("Invalid link position");
system ("PAUSE")
return;
}
if (lp==1)
{
head => data = updatedata();
printf (" node is updated");
system ("PAUSE");
return;
}
thl = head;
for (i=1; ~~lp~~ i < lp; i++)
thl => thl -> next
th->data = updatedata();
```

```
printf (" node is update \n");
system ("PAUSE");
return;
 }
}

void reversenode()
{

 LINK *temp = NULL;
 LINK *current = head;
 int nl;
if (head == NULL)
{
 printf (" List is empty");
 system ("PAUSE");
 return;
 }
 ~~printf~~ nl = nodecount();
  if (nl == 1)
 {
 printf (" reverse can't be possible");
 system ("PAUSE");
 return;
 }
 while (current != NULL)
  {
  temp = current->prev;
  current->prev = current->next;
  current->next = temp;
  current = current->prev;
  }
  head = temp->prev;
  return;
  }
```



int main
{
int option;
while (1)
{ system ("CLS");
Printf ("1 For ~~insert~~ ADD node");
Printf ("\n2 for ~~delete~~ display node");
Printf ("\n3 for count node");
Printf ("\n4 for insert node");
Printf ("\n5 for delete node");
Printf ("\n6 for update node");
Printf ("\n7 for reverse node");
Printf ("\n8 for exit node");

```
scanf ("%d", &option);
Switch (option)
{
case 1: Addnode ();
        break;
case 2: displaynode();
        break;
case 3: printf ("node count = %d", nodecount());
        System ("PAUSE");
        break;
case 4: ~~printf~~ Insertnode();
        break;
case 5: deletenode();
        break;
case 6: updatenode();
        break;
case 7: reversenode();
        break;
case 8: free(head)
        return EXIT_SUCCESS;
default: printf ("invalid option")
         system ("PAUSE");
         break;
}}
```

Single circular linked list:-

*) when we are working S.C.L.L every node contain two fields i.e data and pointer to next node

*) data is maintain inform of the node and pointer to next is next node ~~inform~~ address.

*) when we are working with S.C.L.L. every node having linear relationship only. i.e forward.

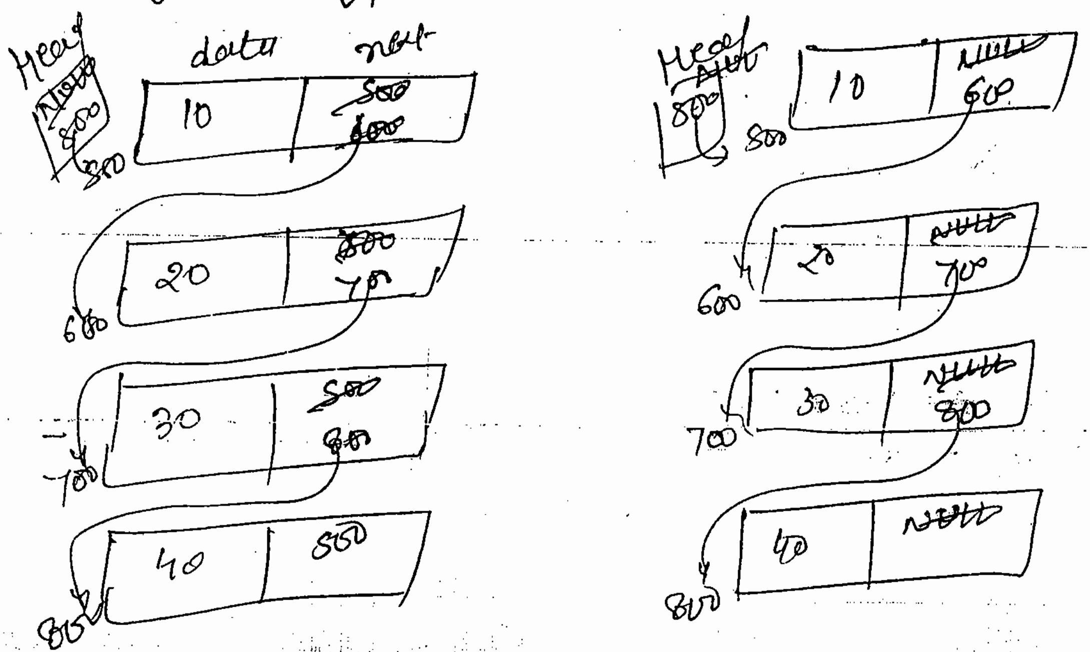
*) In S.C.L.L also linear travelling process only possible because bidirectional relation is not maintain.

*) The basic diff. b/w S.L.L and S.C.L.L is —

→ In S.L.L tail->next value is null. becoz it is terminal node.

→ In S.C.L.L tail->next is maintain head inform

⇒ Logical Representation of S.C.L.L



```c
#include<stdio.h>
#include<stdlib.h>
#include<dos.h>
#include<malloc.h>

struct sclink
{
 int data;
 struct sclink *next;
 };
typedef struct sclink LINK;
LINK*head =NULL;
void addnode()
 {
 LINK*th;
 char ch;
 if (head =NULL)
 {
  head = (LINK*) malloc(sizeof(LINK));
  printf ("enter the data");
  scanf ("%d" &head ->data);
  head -> next = head;
  printf ("Do you want to continue Y?");
  fflush (stdin);
  ch= getchar ();
  if(ch!='Y'&& ch!='y')
  return;
  }
```

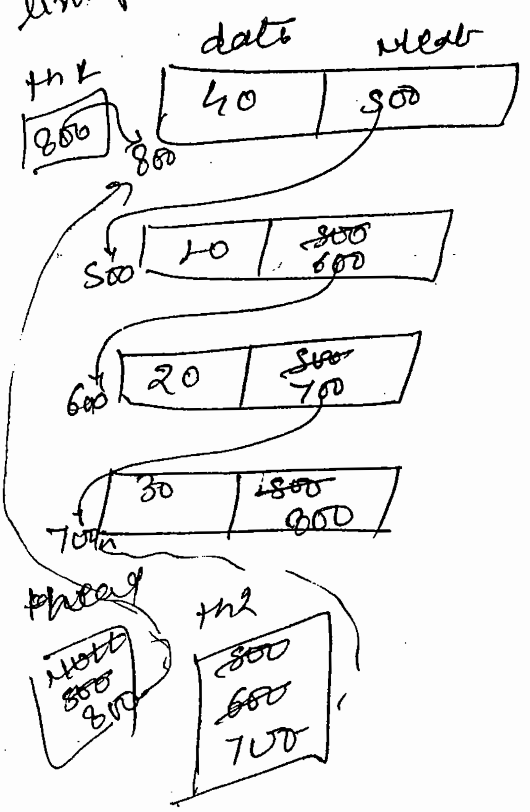
```
th = head;
while ( th ! -> next ! = head)
th = th->next;
do
{
th->next = (LINK*) malloc(sizeof(LINK));
~~th~~ = th -> next;
printf ("~~%d~~ Enter the data);
scanf ("%d", & th -> data);
th-> next = data;
printf (" Do you want to continue ? ");
fflush(stdin);
ch = getchar();
} while ( ch = = 'Y' || ch = = 'y');

}
void displaynode ()
{
LINK *th;
if ( head = = NULL)
{
printf ("List is empty ");
system ("PAUSE");
return;
}
th = head;
~~while ( th->next ! = head)~~
do
{
printf (" node data : %d\n", th -> data);
th= th->next;
```

```
} while (th != head);
system ("PAUSE");
return;
}
int nodecount()
{
 int count = 0;
 LINK *th;
 if (head == NULL)
  return count;
 th = head;
 do
 {
  ++count;
  th = th->next;
 } while (th != head);
 return count;
}
void insertnode()
{
 LINK *th, *thR;
 int lp, nl, i;
 if (head == NULL)
 {
 printf ("list is empty\n");
  system("PAUSE");
  return;
 }
```

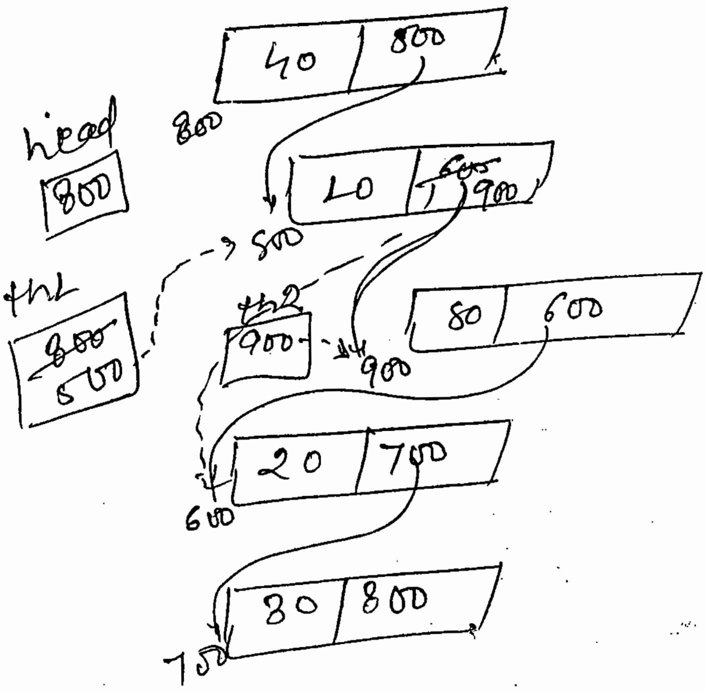
```
printf ("Enter the link position");
scanf ("%d", &lp);
nl = nodecount ();
if (lp<1 || lp >=nl)
{
printf ("Invalid link position\n");
system ("PAUSE");
return;
}
if (lp==1)
{
thl = (LINK*) malloc (sizeof(LINK));
printf ("Enter data");
scanf ("%d", &thl->data);
thl->next = head;
th2 =head;
while (th2->next != head)
th2 = th2->next
head =thl;
th2->next =head;
return;
}
thl =head;
for (i=1; i< lp-1; i++)
thl =th->next;
th2 = (LINK*) malloc (sizeof (LINK));
printf("Enter the data");
scanf ("%d", & th2->data);
```

```
th2 -> next = th1 -> next
th1 -> next = th2;
return;
}
```

link position=1



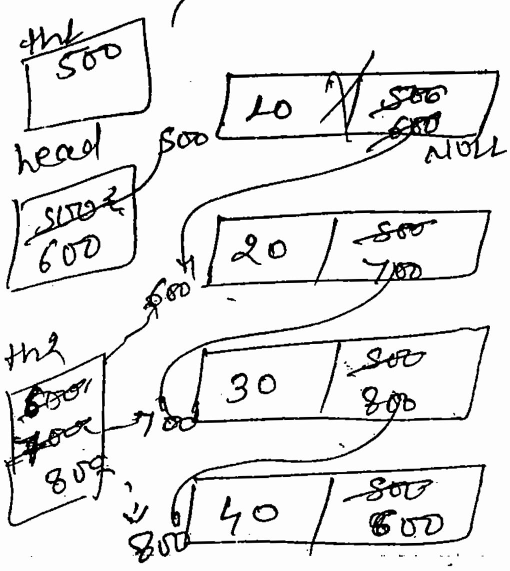
Any other position.

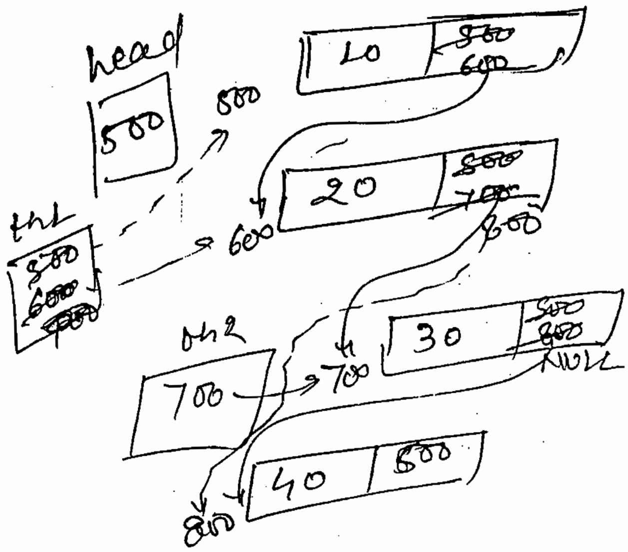



```
void deletenode()
{
LINK *th1, *th2;
int i, lp, nl;
if (head==NULL)
{
printf("\n list is empty\n");
system("PAUSE");
return;
}
```

```
Printf ("Enter the link position)
scanf ("%d", &lp);
nl = nodecount ();
if (lp<1 || lp>nl)
 {
Printf ("invalied position:\n");
system ("PAUSE");
return;
}
if (nl==1)
{
free(head)
head=NULL;
return;
}
if (lp==1)
{
tn1 = head
head = head->next; // head = tn1->next;
  tn1->next = NULL;
  tn2 = head;
  while (tn2->next != tn1)
    tn2 = tn2->next;
    tn2->next = head;
    free(tn1);
    tn1 = tn2 = NULL;
    return;
     }
```

```
tn1 = head
for(i=1 ; i< lp -1; i++)
tn1 = tn1->next;
tn2 = tn1 -> next;
tn1->next = tn2->next;
tn2->next = NULL;
free(tn2);
return;
}
```

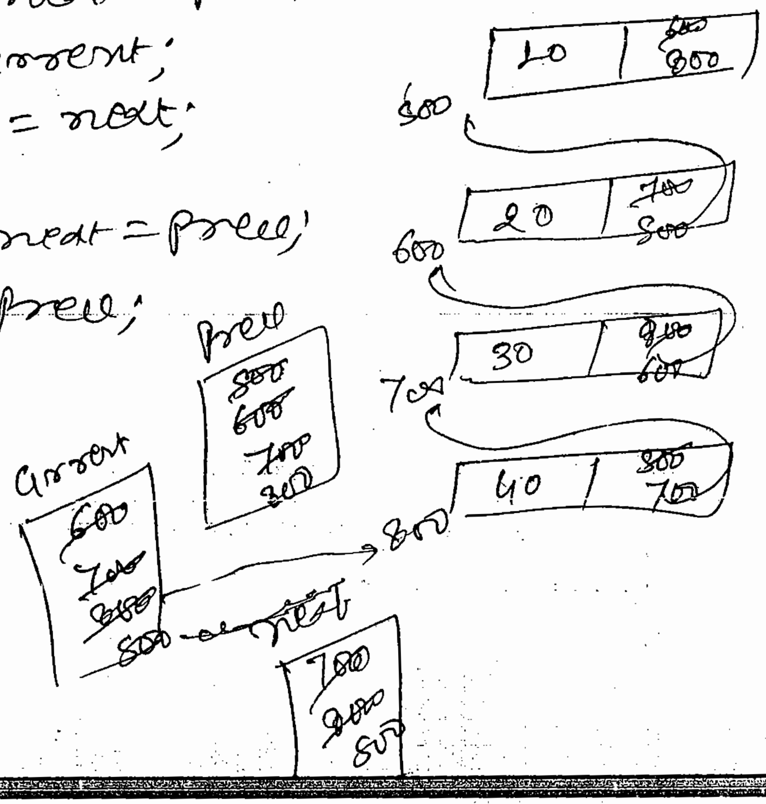




```
void updatenode()
{
LINK* tn1, *tn2;
int i, lp, nl;
if (head = NULL)
{
printf ("List is empty !");
system ("PAUSE");
return;
}
```

```
printf ("Enter the position: ");
scanf ("%d", & lp);
nl = nodecount ();
if(lp < 1 || lp > nl)
  {
  printf ("invalid position\n");
  system ("PAUSE");
  return;
  }
if( lp == 1)
  {
  printf ("Enter the data)
  scanf ("%d", &head -> data)
  return;
  }
th1= head;
for (i=1 ; i< lp ; i++)
 th1= th->next;
printf ("Enter the data");
 scanf ("%d", & th1 -> data);
 return;
 }
void reversenode ()
 {
 LINK* prev= head;
 LINK* current = head -> next;
 LINK * next;
 int nl;
```

```
if (head == null)
{
Printf ("List is empty");
system ("PAUSE");
return;
}
~~Printf ("Enb~~
nl = nodecount ();
if (nl == 1)
{
Printf ("Can't be reverse!\n");
system ("PAUSE");
return;
}
while (current != head)
{
next = current->next;
current->next = prev;
prev = current;
current = next;
}
head->next = prev;
head = prev;
}
```



4/04/15 Double Circular Linked List:

*) When we are working D.C.L.L then every node contains three fields i.e pointer to previous node, data and pointer to next node.

*) Pointer to prev node holds previous node inform and data holds inform and pointer to next field hols next node inform.

*) When we are working with D.C.L.L every node must be required to maintain prev and next node inform.

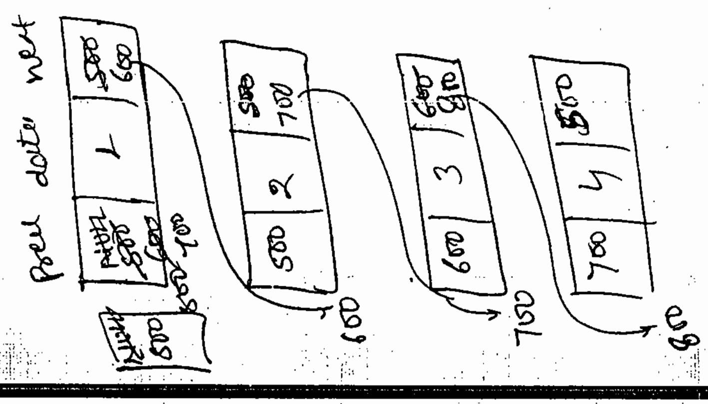
*) In D.C.L.L only bi-directional travelling is possible i.e head to tail and tail to head also.

NOTE:- The basic diff. b/w double l.l. and D.C.L.L is —

a) In D.L.L head to previous and tail to next both are null values.

b) In Double and circular linked list head→prev must maintain tail inform and tail to next must maintain head inform in circular format.

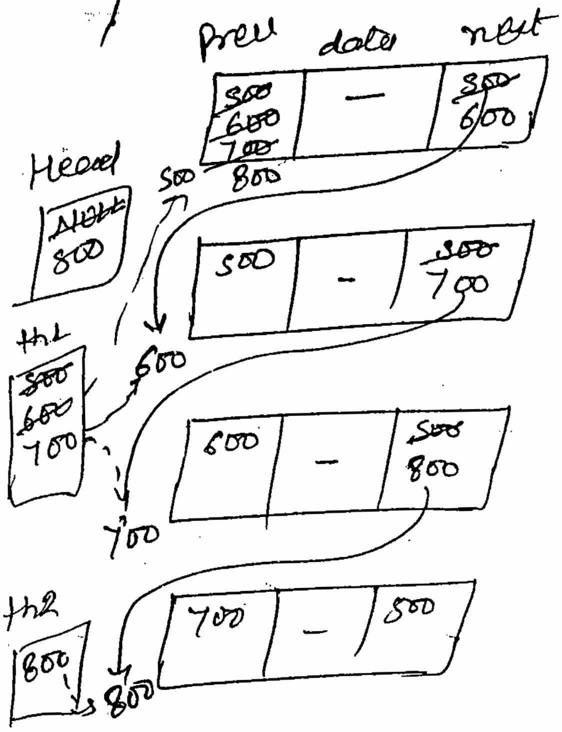
Logical Representation of D.C.L.L:



```
#include<stdlib.h>
#include< dos.h>
#include< stdlib.h>
#include<malloc.h>
typedef struct
{
int id;
char name[36];
int sal;
}Emp;
Emp getdata()
{
Emp te;
return te;
}
Emp updatedata()
{


}
Emp showdata(Emp te)
{


}
struct dclink
{
struct dclink *prev;
Emp data;
struct dclink *next;
};
```

```
typedef struct dellink LINK;
LINK*head=NULL;
void addnode()
 {
 LINK* th1, *th2;
 int ch;
 if(head==NULL)
 {
 head = (LINK*)malloc(sizeof(LINK));
 head->prev = head;
 head->data = getdata();
 head->next = head;
 printf("Do you want to continue y?:");
  fflush(stdin);
 ch= getchar();
 if(ch!='Y' && ch!='y')
 return;
 }
 th1 = head;
 while(th1->next!= head)
  th1 = th1->next;
do
{ th2=(LINK*)malloc(sizeof(LINK));
  th2->prev = th1;
  th2->data = getdata();
  th2->next = head;
  th1->next = th2;
  head->prev = th2;
  th1 = th2;
```

```
printf("Do you want to continue y/y");
fflush(stdin);
ch= getchar();
} while(ch=='Y' || ch=='y')
th1 = th2 = NULL;
return;
}
```

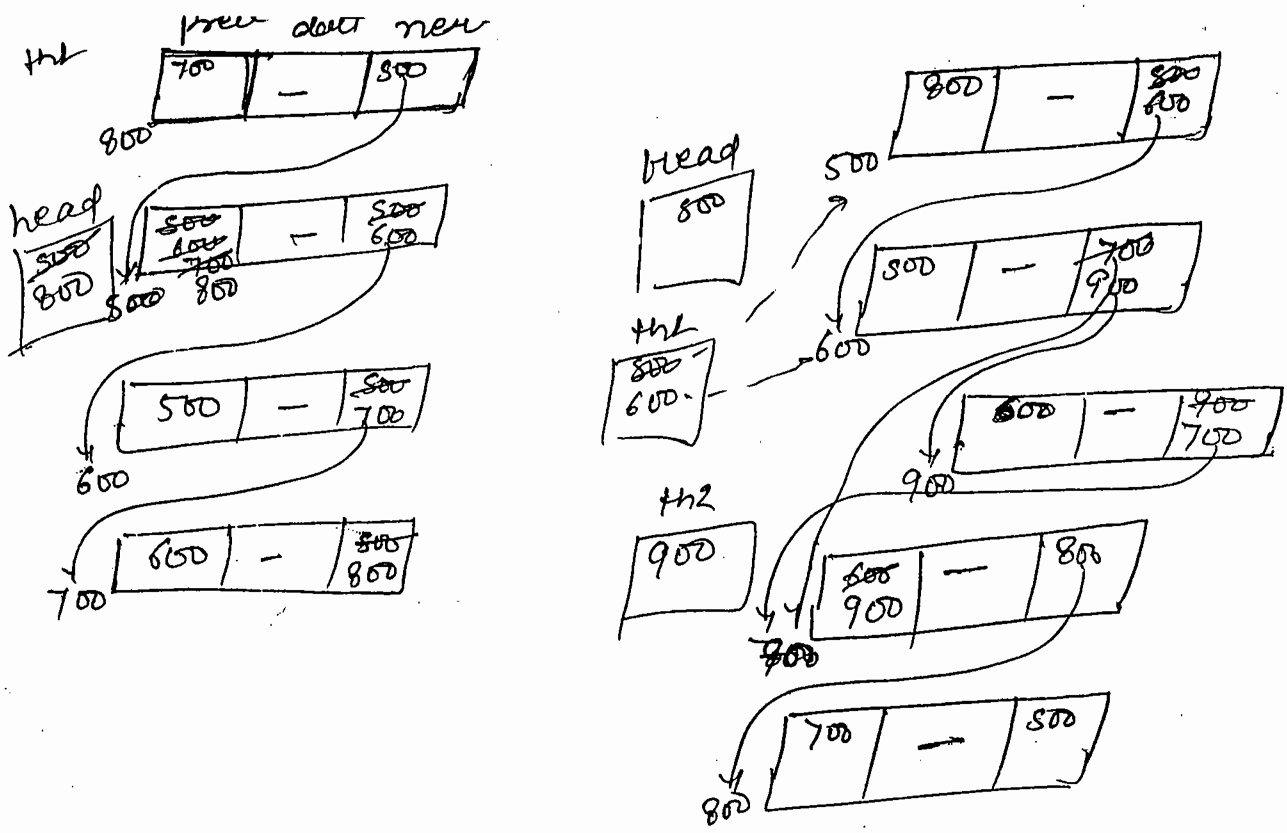



```
void displaynode();
{
LINK *th1;
if (head == NULL)
{
printf("List is empty");
system("PAUSE");
return;
}
th1 = head;
do
{
```

```
showdata ( thl->data);
thl = thl -> next;
} while ( thl != head);
Printf ("\n");
system ("PAUSE");
return;
}
void displaynodeLtoF()
{
LINK *th;
~~int~~ if (head ==NULL)
{
printf (" List is empty ");
system ("PAUSE");
return;
}
th = head ->prev;
do
{
showdata (th -> data);
th = th->prev;
}while (th! = head -> prev);
Printf ("\n");
system ("PAUSE");
return;
}
```

```
void nodecount ()
{
int count =0;
LINK *th;
if (head =NULL)
{ return count;
th = head;
~~while~~ do
{
++count
th = th-> next;
} while (th! = head);
return count;
}
void insertnode ()
{
LINK *th1, *th2;
int i, lp, nl;
if (head = =NULL)
{
printf ("list is empty:");
system ("PAUSE");
return;
}
printf ("Enter the insert position:");
scanf ("%d" &lp);
nl = nodecount ();
if (lp <1 || lp > nl)
```

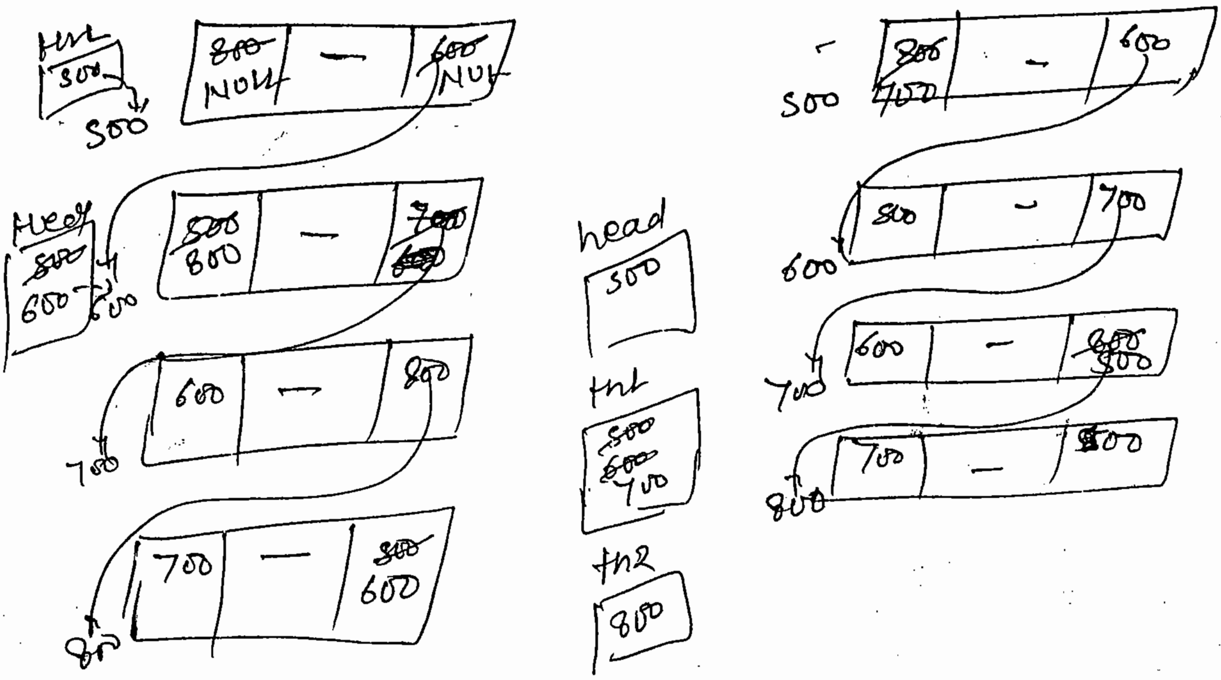
```
}
printf ("Invalid position:");
system ("PAUSE");
return;
}
if (lp == 1)
{
th1 = (LINK*) malloc (sizeof(LINK));
th1 -> prev = head -> prev;
th1 -> data = getdata();
th1 -> next = head;
head -> prev -> next = th1;
head -> prev = th1;
head = th1;
return;
}
th1 = head;
for (i=1; i< lp-1; i++)
th1 = th1 -> next;
th2 = (LINK*) malloc (sizeof (LINK));
th2 -> prev = th1;
th2 -> data = getdata ();
th2 -> next = th1 -> next;
th1 -> next = th2;
th2 -> next -> prev = th2;
th1 = th2 = NULL;
return;
}
```



```
void deletenode()
{
LINK *th1, *th2;
int i, lp, nl;
if(head==NULL)
 {
 printf("List is empty!");
 system("PAUSE");
 return;
 }
 printf("Enter the delete position:");
 scanf("%d",&lp);
 nl=nodecount();
 if(lp<1|| lp>nl)
 {
```

```
printf ("Invalid delete position\n");
system ("PAUSE");
return;
 }
if (nl == 1)
 {
 free (head);
 head = NULL;
 printf (" node is deleted\n");
 system ("PAUSE");
 return;
 }
 if (lp == 1)
 {
 tn1 = head;
 head = head->next;
 head->prev = tn1->prev;
 head->prev->next = head;
 tn1->next = tn1->prev = NULL;
 free (tn1);
 tn1 = NULL;
 return;
  }
  tn1 = head;
 for (i = 1; i < lp-1; i++)
 tn1 = tn1->next;
 tn2 = tn1->next;
 tn1->next = tn2->next;
```

th2->next->prev = th1;
th2->prev = th2->next = NULL;
free (th2);
th1 = th2 = NULL;
return;
}



19/01/2015:- **STACK'S:**

*) Stack is a linear data structure which can allows any kind of operations on only one end called 'TOP'.

*) When we are working with stack insertion or deletion required to perform on top of the stack only.

*) When we are working with stack it allows three specific operations only. i.e

1) Push. 2) POP 3) Peek.

*) Push is a procedure of inserting an element on Top of the stack.

*) POP is a procedure of deleting an element from TOP of the stack.

*) peek is a procedure of retrieving all elements from the stack without deletion.

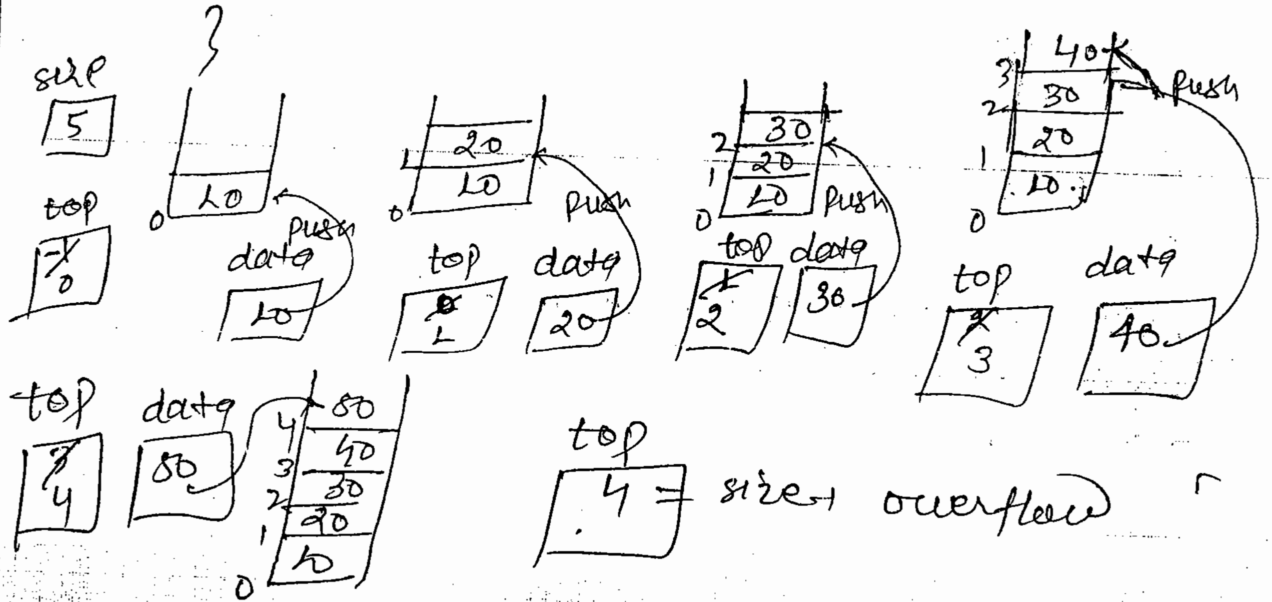
When we are working with stack always it follow LIFO Approach. i.e LAST IN FIRST OUT behaviour

Implementation of stack using Array's:-

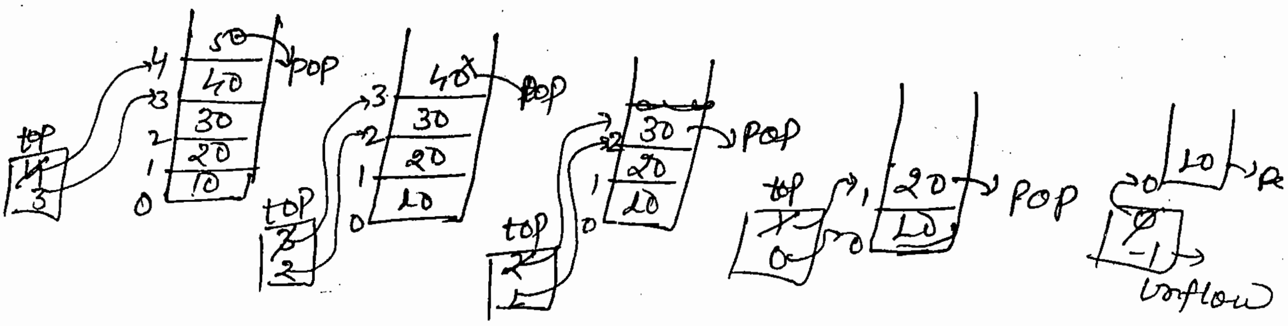
```
#include <stdio.h>
# include < stdlib.h>
# include < dos.h>

# define SIZE 5

int STACK [SIZE];
int top = -1;

int main()
{
 int option;
 void push (void);
 void pop (void);
 void peek( void);
 while (1).
 {
  system ("CLS");
  printf ("\n push press 1 --");
  printf ("\n POP Press 2 --");
  printf ("\n peek pree 3 --");
  printf ("\n EXIT  4 --");

  scanf ("%d", &option)
  switch (option):
  {
   case 1: push ();
           break;
   case 2: pop ();
           break;
   case 3: peek();
           break;
```

```
case 4: return EXIT_SUCCESS;
default : Printf ("Invalid option");
          system ("PAUSE");
    }
   }
  }
void push ( )
{
int data;
if ( top = = size -1 )
{
Printf (" stack is overflow");
 system ("PAUSE");
 return;
}
Printf (" Enter the data");
Scanf ("%d", &data);
++top;
STACK[TOP] = data;
} ~~++top;~~
void pop ( )
{
if ( top = = -1 )
{
Printf (" stack is underflow");
 system ("PAUSE");
return;
}
```

```c
printf ("POP element is :%d\n", STACK[top]);
 --top;
 system ("PAUSE");
}
void peek()
{
int i;
if (top == -1)
{
 printf ("STACK is empty --\n");
 system ("PAUSE");
 return;
}
 printf ("STACK ELEMENT is:");
 for (i = top; i>=0; i--)
 printf ("%d", STACK[i]);
 printf ("\n");
 system ("PAUSE");
 return;
}
```



top 4 = size-1 overflow



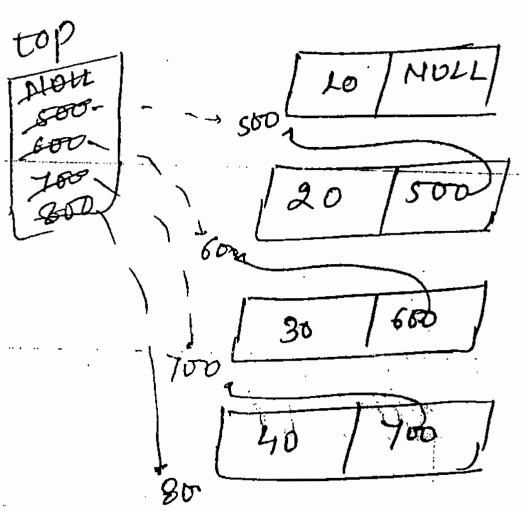
STACK Implementation using ~~single~~ Linked List:-

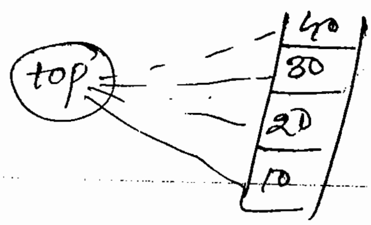
1) when we are implementing the stack using linked list every element required two fields i.e data and pointer to next element.

2) when we are working with stack using L.L. then dedicated pointer is called top. but in general it is equivalent to head node in L.L

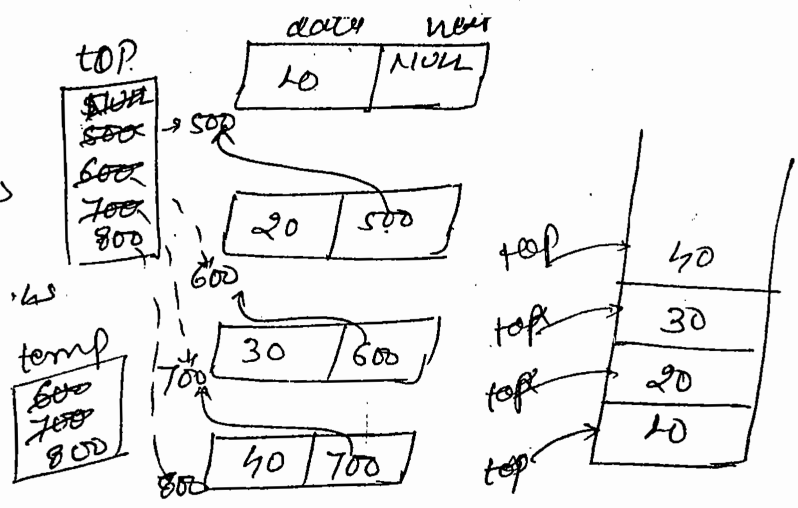
3) when we are working with stack any kind of operation require to perform on top end only but in linked list randomly it is possible.

Logical Relation of Stack:-



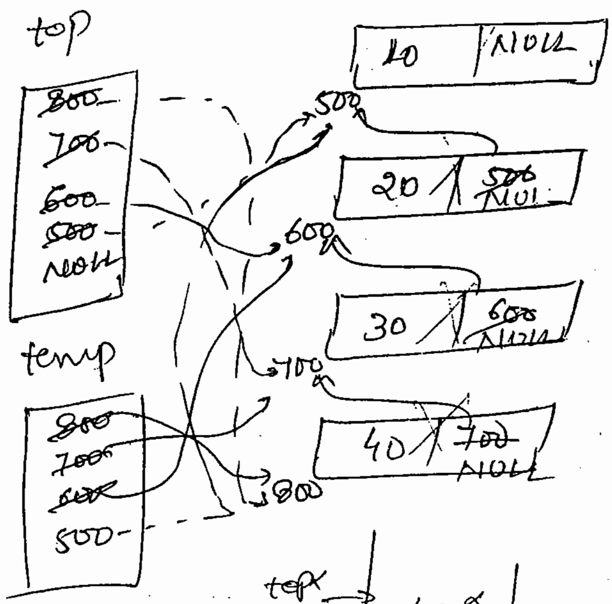


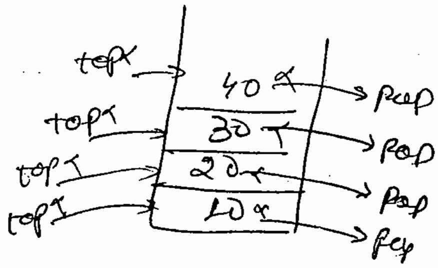
20/01/2015:




```
#include<stdio.h>
#include<stdlib.h>
#include<dos.h>
#include <malloc.h>
struct stack
{
int data.
struct stack *next;
};
typedef struct stack *STACK;
STACK *top=NULL;
void push()
{
STACK *temp;
if(top==NULL)
{
top = (STACK*) malloc (sizeof(STACK));
printf("\nEnter the data:");
scanf("%d",&top->data);
top->next = NULL;
return;
}
temp = (STACK*)malloc (sizeof(STACK));
printf(" Enter the data");
scanf("%d", & temp->data);
temp->next = top;
top = temp;
return; } }
```

```
void pop()
{
STACK* temp;
if (top==NULL)
{
printf("\nstack is underflow");
return;
}
printf("\n pop element is: %d", top->data);
temp= top;
top = top->next;
temp->next =NULL;
free(temp);
return;
}
```



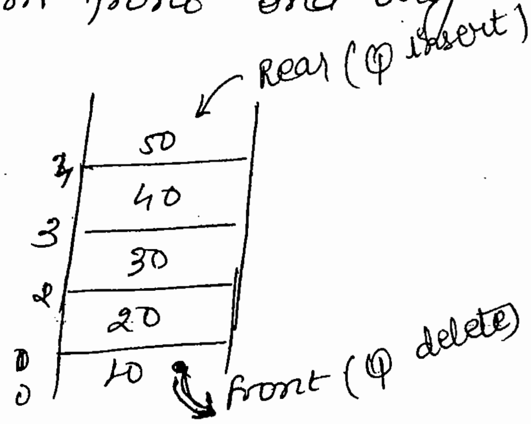



```
void peek()
{
STACK* temp;
if(top==NULL)
{
printf("\nstack is empty:");
system("PAUSE");
return;
}
printf("stack elements is :");
~~top~~ temp=top;

do
{
printf("%d", temp->data);
temp= temp->next;
}while(temp!=NULL);
```

```
system ("PAUSE");
return;
}
int main (void)
{
int ~~main~~ option;
while (1)
{
system ("CLS");
printf ("Press push-1 ");
printf ("Press pop-2 ");
printf ("Press peek - 3");
printf ("Exit--");
scanf (" %d", & option);
~~while~~ switch (option)
    {
    case 1: push();
            break;
    case 2: pop();
            break;
    case 3: peek();
            break;
    case 4: free (top);
            return EXIT_SUCCESS
    ~~default~~: printf ("Invalid option");
            return EXIT_SUCCESS;
    }
    }}
```

Queue:

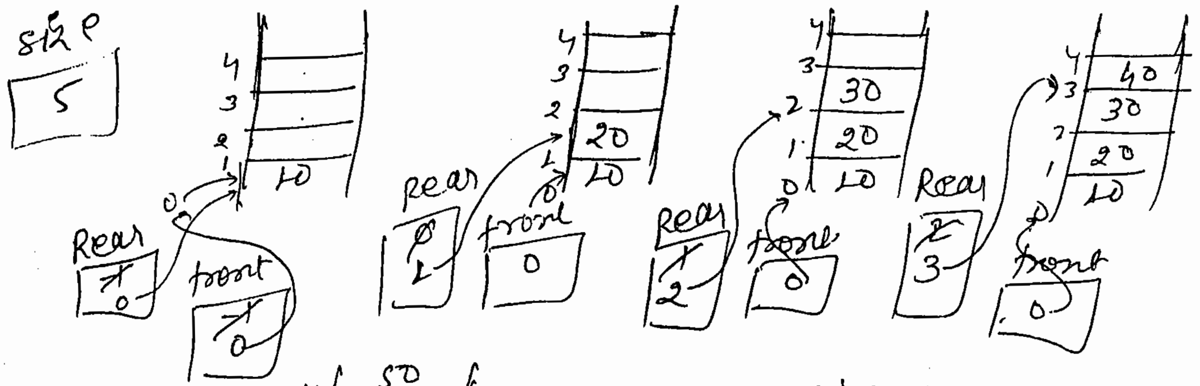
1. Queue is a linear data structure which can allows operations only one end ~~front~~ or Rear.
2. When we are working with stack top end is ~~opened~~ but for queue both ends are open.
3. When we are working with queue operations can allow ~~for~~ both end but specific operation allowed.
4. When we are adding the element then it is possible from queue rear end only.
5. When we are deleting the element then it is possible from front end only.

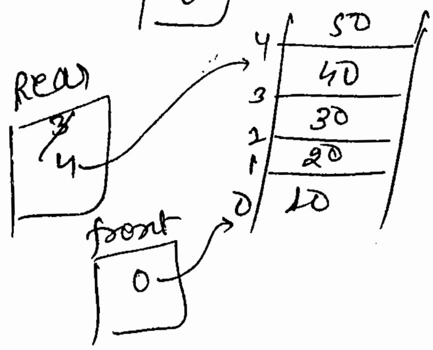


6. When we are working with Queue it allows 3 types of operations i.e
   1) Q insert
   2) Q delete
   3) Q display.

Q insert is a procedure of adding an element into ~~the~~ Q.
Q delete is a procedure of deleting an element from Q.
By using Q display we require to retrieve the data without deletion.

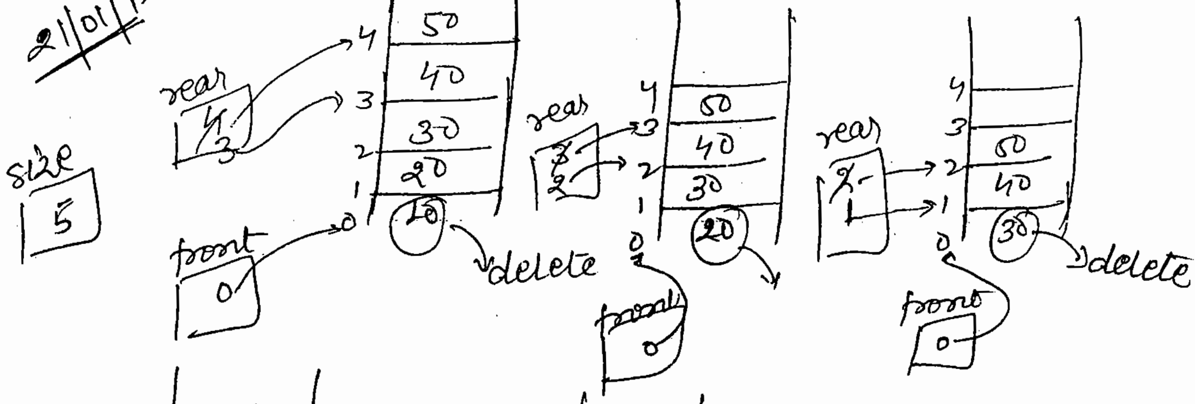
* when we are working with Queue it must be require to satisfy FIFO or FCFS system.

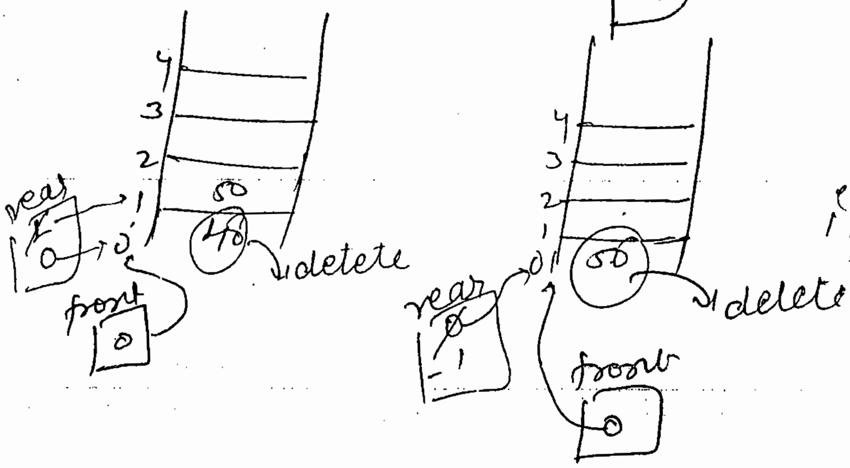




$$\underbrace{rear = size-1}_{full}$$

21/01/15





if (rear == -1)
front = -1

```
#include <stdio.h>
#include < dos.h>
#include < stdlib.h>
# define  SIZE 5
int QUEUE[SIZE];
int front = -1;
int Rear = -1;
void qinsert()
{
int data;
if( rear = size -1)
{
printf (" Queue is overflow \n");
system ("PAUSE");
return;
}
++rear;
if(rear ==0)
  front = 0;
printf (" Enter the data : ");
scanf ("%d",&data);
QUEUE[rear] = data;
 return;
}
void qdelete ()
{
int data;
if (rear == -1)
{
printf (" Queue is underflow \n");
```

```
system ("PAUSE");
return;
}
printf ("delete element is : %d", QUEUE[front]);
for (i=0; i< ~~front~~ rear ; i++)
QUEUE[i] = QUEUE[i+1];
--rear;
if (rear = = -1)
front = -1;
system ("PAUSE");
return;
}
void qdisplay()
{
int i;
if ( rear = = -1)
{
printf ("Queue is empty");
system ("PAUSE");
return;
}
printf (" Queue element :");
for (i=0; i<= rear ; i++)
printf ("%d", Queue[i]);
printf("\n");
system ("PAUSE");
return;
}
```

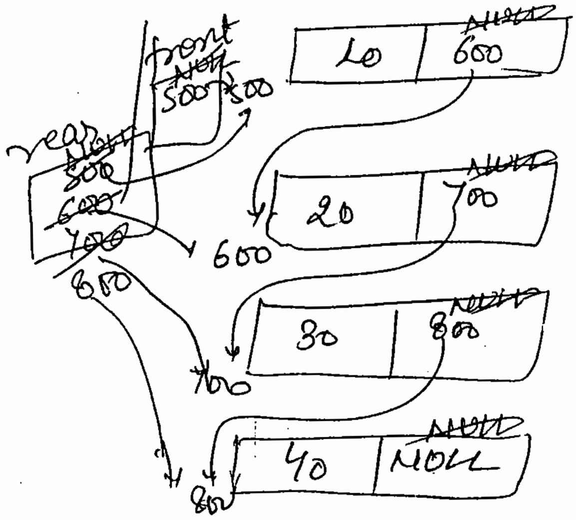
```
void main()
{
int option;
while(1)
{
System ("CLS");
Printf (INSERT QUEUE:");
Printf ("\n2 Delete queue :");
Printf ("\n3 display queue:");
Printf ("\n4 Exit - ——:");
Scanf ("%d", & option)
switch (option)
{
case 1: qinsert ();
        break;
case 2: qdelete ();
        break;
case 3: qdisplay ();
        break;
case 4: ~~delete~~ EXIT_SUCCESS_
default : Printf (" Invalid option");
        system ("PAUSE");
        return EXIT_SUCCESS;
}
}
}
```

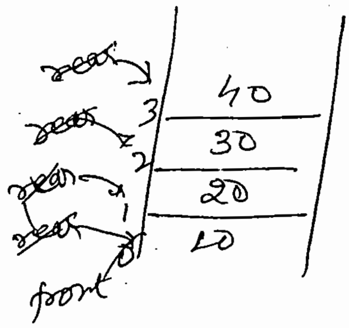
07409705758

Queue Implementation using Linked List (Linear queue)

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

struct queue
{
int data;
struct queue *next;
};
typedef struct queue QUEUE;
QUEUE* rear = NULL;
QUEUE *front = NULL;
void qinsert node( )
{
if (rear == NULL)
{
printf (Queue.
rear = (QUEUE*) malloc (sizeof (QUEUE));
printf ("Enter the data");
scanf ("%d",&rear->data);
rear->next = NULL;
front = rear;
     return;
}
```

```
rear->next = (QUEUE*)malloc(sizeof(QUEUE));
rear = rear->next;
printf("Enter the data:");
scanf("%d", &rear->data);
rear->next = NULL;
return;
}
```

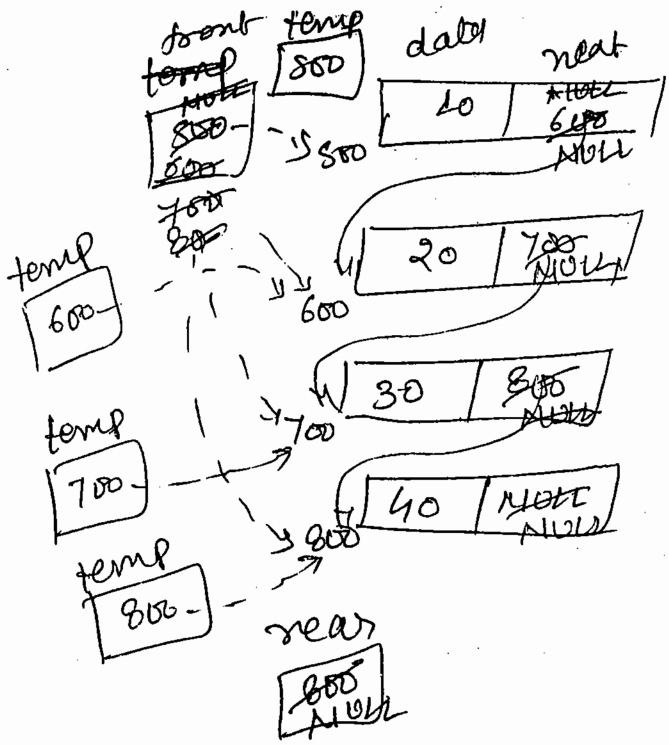





```
void Qdelete()
{
QUEUE*temp;
if (front == NULL)
 {
printf("Queue is underflow:");
 system("PAUSE");
 return;
 }
 temp = front;
 front = front->next;
 temp->next = NULL;
printf("deleleted element is:%d", temp->data);
```

```
free (temp);
system ("PAUSE");
if (front==NULL)
rear = NULL;
return;
}
void qdisplay()
{
QUEUE* temp;
if (front==NULL)
{
printf ("Queue is empty");
system ("PAUSE");
return;
}
temp = front;
~~fronttemp~~ printf (" Queue element is : ");
    do
    {
    printf("%d", temp->data);
    temp= temp->next;
    } while (temp!= NULL);
    printf ("\n");
    system ("PAUSE");
    return;
    }
    }
    }
```

int main()
{
int option;
while (1)
{
system ("CLS");



22/01/15 Types of Queue.

1) In 'C' P.L, we having 4 types of Queues i.e
a) Linear Queue.
b) Priority queue.
c) Circular Queue
d) De-queue.

a) Linear Queue:-

1) When we are working with linear queue all elements are arranged in sequencial manner. i.e How they are inserted.

b) Priority queue:-

1) When we are working with this type of queue every element contains a key value called priority and acc. to the priority elements are process.
2) In priority queue which elements contains highest priority it will process first, which contains least priority it will process at last.
3) when the equal priority is occur in priority queue then it performs FIFO approach.

(c) Circular Queues:-

(1) when we are working with circular queue all elements are arranged in sequential manner like linear queue. but after last element first element is occur.

(d) De-Queues:-

(k) It is also called double ended queue.

*) when we are working with de-queue operations can be perform on both ends.

*) de queues are classified into two types.

1) Input restricted de-queue.

(2) o/p restricted de-queue.

*) In i/p restricted de-queue insertion can be take place, in one end but deletions are allowed in both ends.

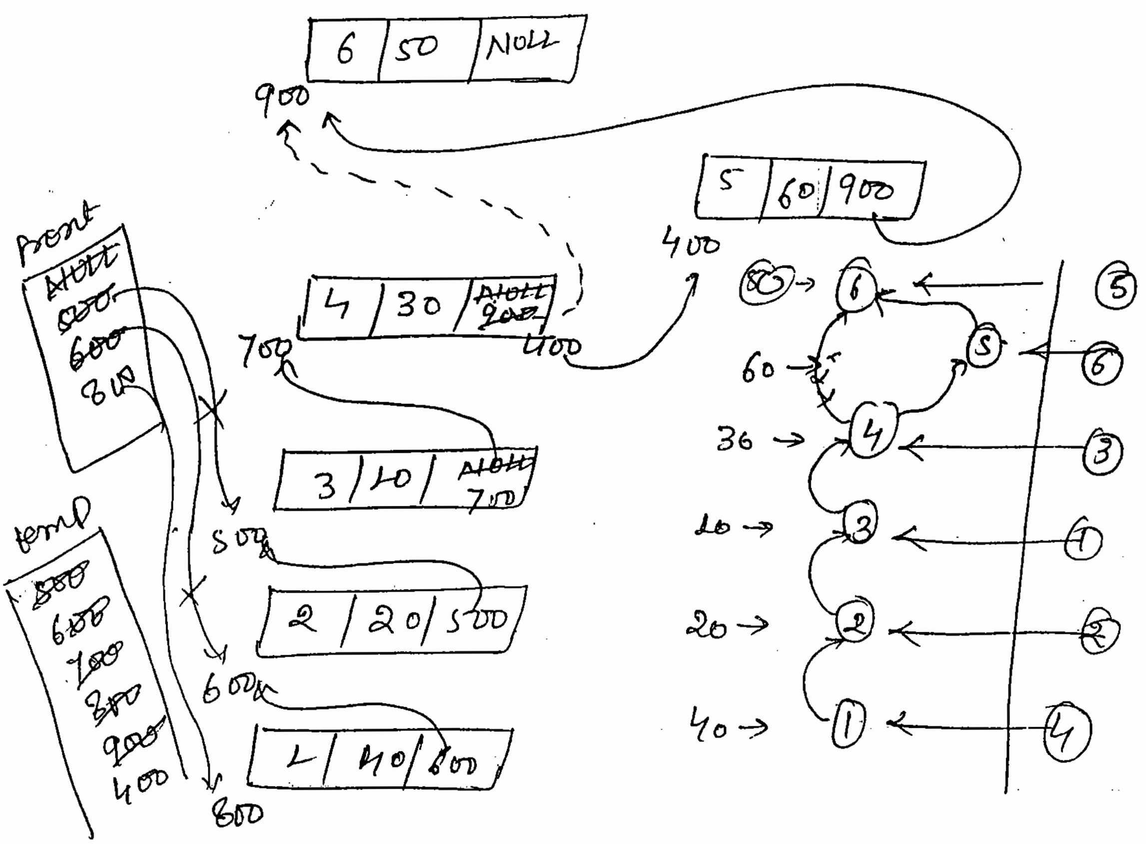
*) In o/p restricted de-queue insertion can be perform in both ends but deletion is only one end.

Implementation of priority queue using L.L.

```
#include<stdio.h>
#include<stdlib.h>
#include< malloc.h>
struct pqueue
{
int key;
int data;
struct pqueue* next;
};
typedef struct pqueue QUEUE;
QUEUE *front = NULL;
void insert()
{
QUEUE* temp,*flag;
temp = (QUEUE*) malloc (sizeof(QUEUE));
printf ("Enter the key");
scanf ("%d", &temp->key);
printf ("Enter the data:");
scanf ("%d", &temp->data);
if (front = =NULL || temp->key < front->key)
  {
  temp->next = front;
  front = temp;
  return;
  }
  flag = front;
while(flag->next!=NULL && flag->next->key < temp->key)
```

```
flag = flag->next;
temp->next = flag->next;
flag->next = temp;
return;
}
void qdelete ()
{
QUEUE *temp;
if (front == NULL)
{
printf ("Queue is underflow\n");
system ("PAUSE");
return;
}
temp = front;
front = front->next;
temp->next = NULL;
printf ("Deleted data is %d", temp->data);
free(temp);
system("PAUSE");
return;
void qdisplay ()
{
int data;
if (front == NULL)
{
printf ("Queue is empty");
system ("PAUSE");
return;
}
temp = front;
~~while (temp->next) = NULL)~~
printf ("Queue element is :");
do
{
printf ("\nKey : %d data : %d", temp->key, temp->data
temp = temp->next;
} while (temp != NULL);
```

printf ("\n");
system ("PAUSE");
return;
}



Circular Queue Implementation using Array

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
int rear = -1;
int front = -1;
int cqueue[MAX];
void qinsert()
{
int data;
if((front == 0 && rear == MAX-1) || (front == rear+1))
{
 printf("\n Queue is overflow(full)");
 getch();
 return;
}
if(front == -1)
 {
 front = 0;
 rear = 0;
 }
else if(rear == MAX-1)
  rear = 0;
else
   rear = rear+1;
printf("Enter the data");
scanf("%d", &data);
cqueue[rear] = data;
}
```

```
void qdelete ()
 {
  if (front == -1)
  {
 printf ("\n Queue is underflow");
 getch();
  }
printf ("deleted data : %d", cqueue [~~front~~]);
  getch();
  if (front == rear)
  {
   front = -1;
   rear = -1;
   }
  else if (front = MAX-1)
    front = 0;
  else
  front = front+1;
}
void qdisplay()
{
int fposition = front, reposition = rear;
 if (front == -1)
  {
   printf ("\n Cqueue is empty");
   getch();
   return;
  }
```

```
printf ("queue elements is :");
if (fposition <= reposition)
{
 while (fposition <= reposition)
 {
 printf ("%d", cqueue [fposition]);
  fposition ++;
  }
 }
else
 {
while ( fposition <= MAX -1 )
 {
 printf ("%d", cqueue[fposition]);
  fposition ++;
 }
 fposition = 0;
 while(fposition <= reposition)
  {
  printf ("%d", cqueue [fposition]);
   }
  }
  getch();
  }
  int main()
  {
  int option;
  while (1)
  {
  printf ("\n1for qinsert :");
  printf ("\n 2 for qdelete:");
```

```
printf ("\n3 for qdisplay");
printf ("\n4 for Exit");
scanf ("%d", &option);
switch(option)
{
case 1: qinsert();
        break;
case 2: qdelete();
        break;
case 3: qdisplay();
        break;
case 4: return QATT_SUCCESS;
default: printf("Invalid option");
        getch();
  }
  }
  }
```

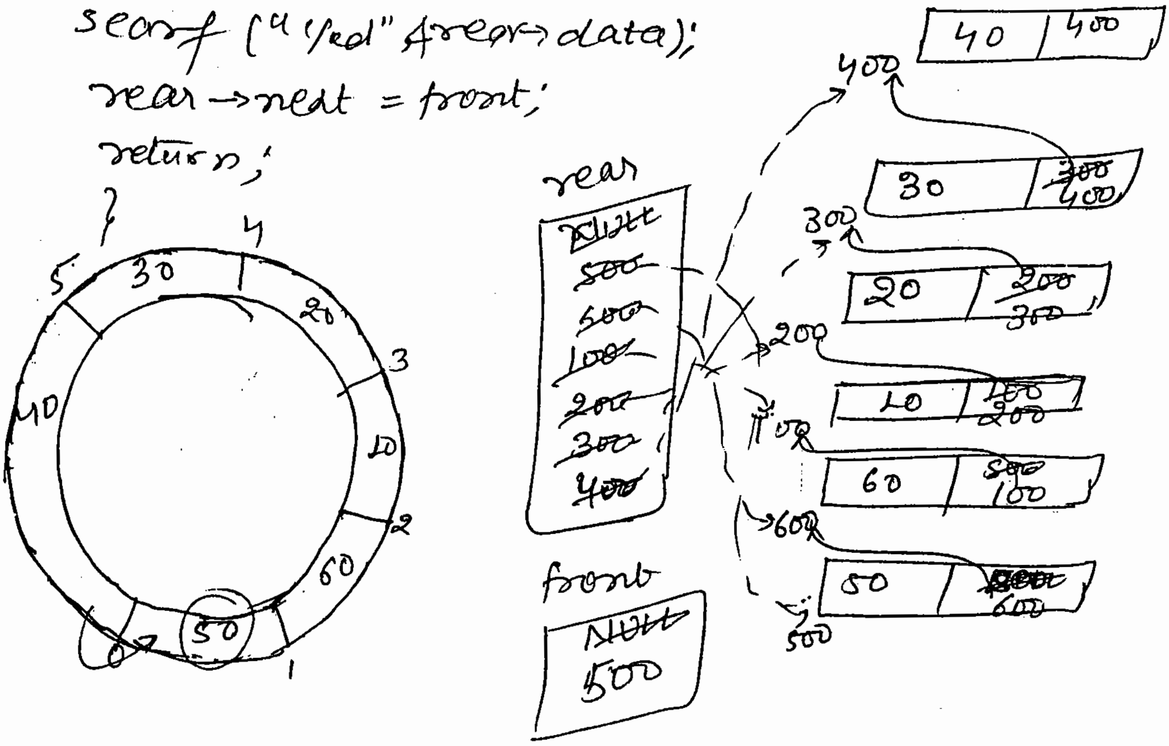
23/01/15: Circular Queue Implementation using Linked List

(1) When we are working with circular queue all elements require to arrange in the form of circle. i.e after last element first element should be occur.

(2)

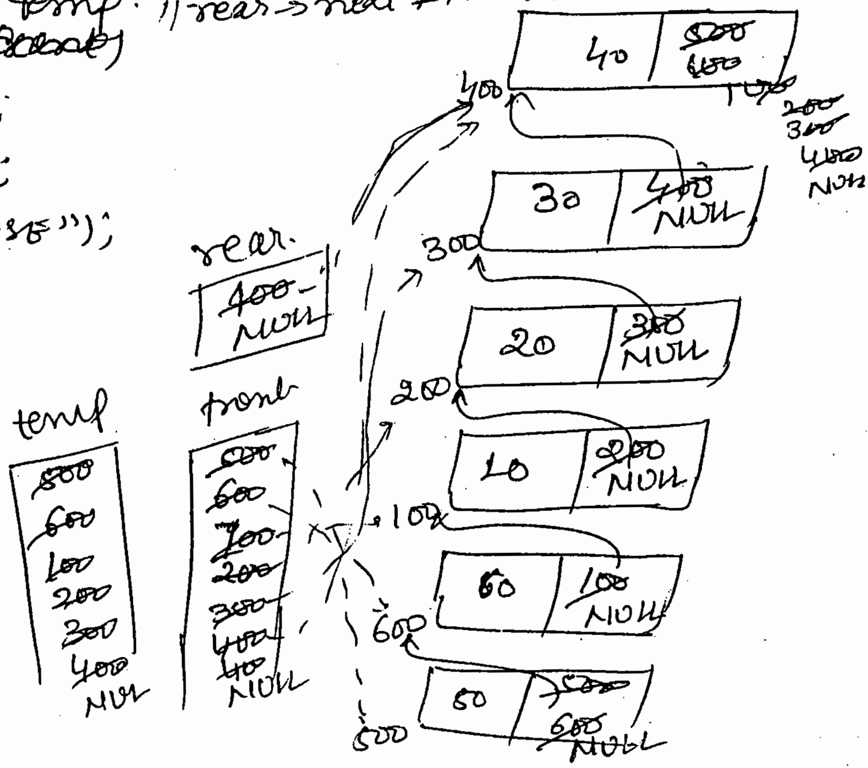
```
#include<stdio.h>
#include<malloc.h>
#include<dos.h>
struct cqueue
{
 int data;
 struct queue *next;
};
typedef struct queue Queue;
Queue *front = NULL;
Queue *rear = NULL;

void qinsert ()
{
 if (rear == NULL)
 {
  rear = (Queue*) malloc (sizeof(Queue));
  printf (" Enter the data:");
  scanf ("%d", &rear->data);
  rear->next = rear;
  front = rear;
  return;
 }
```

```
rear->next = (Queue*) malloc (sizeof(Queue));
rear = rear->next;
printf ("Enter the data: \n");
scanf ("%d" &rear->data);
rear->next = front;
return;
}
```



```
void dqueue()
{
Queue *temp;
if (rear == NULL)
{
printf ("Queue is underflow");
system ("PAUSE");
return;
}
temp = front;
front = front->next;
rear->next = front;
temp->next = NULL;
```

```
printf("Deleted element is : %d", temp->data);
 free(temp);
 if(rear == temp)  // rear->next != NULL;
  front = NULL;
  rear = NULL;
  system("PAUSE");
  return;
  }
void qdisplay()
 {
 Queue* temp;
if(front == NULL)
 {
printf("Queue is empty:");
 system("PAUSE");
 return;
 }
 temp = front;
printf("Queue element is :");
 do
 {
printf("%d", temp->data);
  temp = temp->next;
 } while(temp != front);
  system("PAUSE");
  return;
  }
```



```
int main(void)
 {
 int option;
 while (1)
  {
  system ("CLS");
  printf("\n1 for insert:");
  printf("\n2 for delete:");
  printf("\n3 for display:");
  printf ("\n4 for Exit:");
  scanf("%d", &option);
  switch(option)
   {
   case 1: qinsert();
           break;
   case 2; qdelete();
           break;
   case 3: qdisplay();
           break;
   case 4; free (temp);
           return Exit_success.
   default: printf ("Invalid option");
           system ("PAUSE");
           return;
         }
        }
       }
```

Implementation of De-Queue using Array:

```
#include < stdio.h>
#include < process.h>
#include< conio.h>
# define MAX 5
int dequeue[MAX];
int left =-1;
int right =-1;
void insert_right();
void insert_left();
void delete_right();
void delete_left();
void display_queue();
void input_que()
{
int choice;
while(1)
{
printf("\n1 Insert at right :");
printf("\n2 delete from left :");
printf("\n3 delete from right :");
printf("\n4 display :");
printf("\n5 return to main");
printf("\n6 Quit :");
scanf("%d", &choice);
switch (choice);
{
```

```
}
case 1: insert_right();
        break;
case 2: delete_left();
        break;
case 3: delete_right();
        break;
case 4: display_queue();
        break;
case 5: return;
case 6: exit(1);
default: printf("\nInvalid choice");
}
}
}
void output_que()
{
int choice;
while (1)
{
printf("\n1 Insert_right:");
printf("\n2 Insert_left:");
printf("\n3 delete_left:");
printf("\n4 display_que:");
printf("\n5 return to main:");
printf("\n6 exit:");
scanf("%d", &choice);
```

```
switch (choice)
{
 case 1: insert_right();
        break;
 case 2: insert_left();
        break;
 case 3: delete_left();
        break;
 case 4: display_que();
 case 5: return;
 case 6:  exit(1);
 default : printf("wrong choice\n");
  }
 }
}
int main()
 {
  int option;
 while(1);
 {
 clrscr();
 printf("\n1: input restricted deque:");
 printf("\n2: output restricted dequeue:");
 printf("\n3: Exit --");
 printf("\n Enter your choice:");
 scanf("%d", &option);
  switch(option)
  {
  case 1: input_que();
          break;
```

```
case 2: output_ques;
        break;
case 3: return 0;
default: printf("\n wrong choice");
  }
 }
}
void insert_right()
{
int data;
if (left == 0 && right == MAX-1) || (left == right+1))
{
 printf("Queue is overflow:");
 getch();
 return;
}
if (left == -1)
{
 left = 0;
 right = 0;
}
else if (right == MAX-1)
  return;
else
  right = right+1;
printf("\n Enter data:");
scanf("%d", &data);
dequeue[right] = data;
}
```

```
void insert_left()
{
int data;
if(left==0 && right == MAX-1) || (left == right+1))
{
printf("\nQueue is overflow:");
getch();
return;
}
if(left == -1)
{
left = 0;
right = 0;
}
else if (left ==0)
left = MAX-1;
else
left = left -1;
printf("\n enter data :");
scanf("%d", &data);
dqueue[left] = data;
}
void delete_left()
{
if (left == -1)
{
printf("\n Queue is underflow:");
getch();
return;
}
```

```
printf("\nDeleted data is : %d", dqueue[left]);
 if (left==right)
 {
  left = -1;
  right = -1;
 }
 else if (left==MAX-1)
      left = 0;
 else
 left = left+1;
 }
 void delete_right()
 {
  if (left == -1)
  {
  printf("Queue is underflow:");
  getch();
  return;
  }
printf("\n Deleted data is : %d", dqueue[right]);
 if (left == right)
 {
 left = -1;
 right = -1;
 }
 else if (right==0)
     right = MAX-1;
  else
     right = right+1;
     }
```

```
void display_que()
 {
 int fposition = left, rposition = right;
 if(left==-1)
  {
  printf("Queue is empty:");
   getch();
   return;
   }
  printf("\n Queue elements");
  if(fposition<= rposition)
   {
   while(fposition <= rposition)
    {
    printf("%d",dqueue[fposition]);
     fposition++;
    }
   }
 else
  {
  while(fposition <= MAX-1)
  {
  printf("%d",dqueue[fposition]);
   fposition++;
  }
  fposition = 0;
  while(fposition <= rposition)
  {
  printf("%d", dqueue[fposition]);
   fposition++; } } }
```

24/01/15: ## Application of Stack:

(1) Balancing of symbols:- i.e infix to postfix/prefix conversion.

(2) Redo/Undo features at many places likes editors or IDE.

(3) Forward and backward feature in web browser

(4) Used in many algorithm like tower of Hanoi

(5) Other application can be backtracking, knight tour problem.

(2) Applications of Queues:

(a) Queue is used when things don't have to processed immediately, but have to be processed in FIFO like BFS.

(b) When a resource is shared among multiple consumers like printer, CPU scheduling and disk shedule

(c) When the data is transferred asynchronously b/w two processes.

Notations:-

(1) When we are evaluating any kind of expression then those expressions are evaluted acc. to notations only

(2) Notation means evaluting the expression acc. to the position of the operator.

(8) Notations are classified into 3 types.

i.e a) Infix Notation.

b) Prefix. Notation.

c) Postfix Notation.

a) Infix Notation :- when the operator is present in b/w operand. then it is called infix notation.

*) Normal regular expression are all are infix notations only.

*) when we are working with infix operator are presents b/w operands.

ex 1) a+b

2) a+b*c

b) Prefix Notation :- When we are working with prefix notations operators are place before the operands.

*) If any operators are present middle of the operand then syntactically it is invalid expression.

ex +ab

1) +*bca

c) Postfix Notation :- when we are working with postfix notations all operators are place after the operands only.

ex ab+

abc*+

Infix to postfix Notations:-

*) when we are converting infix to postfix then follow following algorithm.

step-1) Read given infix expression in string called infix.

step-2) Read a single character from infix string and perform following task.

a) If the read the character is an operand then add the operand into another string called postfix.

b) If the readed character is operator then
→ If stack is empty
→ If stack is already having operator then compare current operator with existing operator then.
- If existing operator having highest priority then current operator then pop existing operator and push current operator into stack.
→ If existing operator having least priority or equal priority then push current operator into stack.

c) Repeat step-2 until all characters are expected from infix string

d) If stack is not empty then pop all operators from stack push into postfix string.

e) Repeat step 4 until all elements are pop.
f) display the resultant string on console by using postfix string

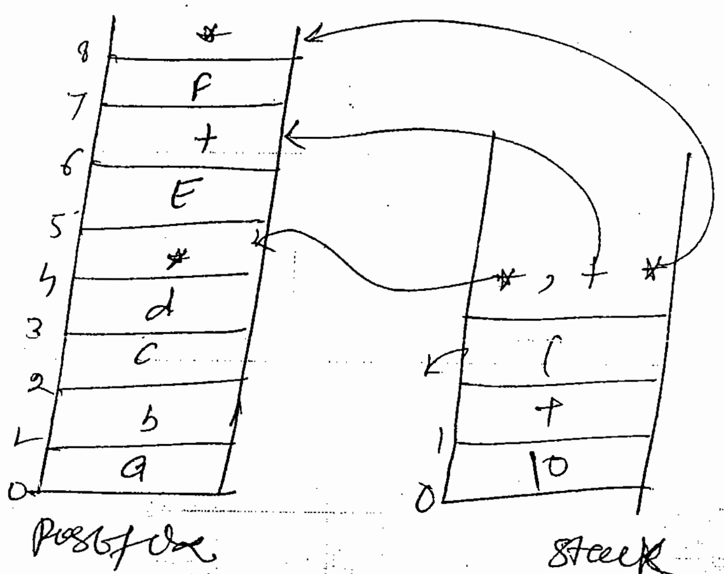
```
#include<stdio.h>
#include<dos.h>
#include<conio.h>
#include<string.h>
#define operand(x) (x>='a' && x<='z' ||
 (x>='A' && x<='Z') || x>='0' && x<='9')

char infix [30];
char postfix [30];
char top, i=0;
/* Function To initialize the stack */

void init()
 {
  top=~~-1~~;
 }
void push (char x)
 {
  stack [++top]=x;
 }
char pop()
{
 return (stack [top--]);
}
int isp(char x)
 {
   int y;
y.
```

```
y=(x=='('?0:x=='^'?4:x=='*'?2:x=='/'?
    2:x=='+'?1:x=='-'?1:x==')'?6:-1);
    return y;
}
int isp(char x)
{
    int y;
y=(x=='('?4:x=='^'?4:x=='*'?2:x=='/'
    ?2:x=='+'?1:x=='-'?1:x==')'?6:-1);
    return y;
}
void infixtopostfix()
{
    int i,l=0;
    char x,y;
    stack[++top]='\0';
for(i=0;(x=infix[i++])!='\0';i--)
  {
    if(operand(x))
     postfix[l++]=x;
    else
    {if(x==')')
     {
    while((y=pop())!='(')
      postfix[l++]=y;
     }
     else
     {
```

```
while ( isp ( stack [top] ) >= icp(x))
  postfix [l++]= pop ( );
   push (x);
   }} }//for
 while (top>=0)
 postfix [l++]= pop ( );
 }
int main()
{
 init ();
printf ("Enter infix expression : ");
 scanf ("%s", infix);
 infixtopostfix ();
printf ("The resulting postfix expression %s",
                                 postfix);
 return 0;
}
```

a+b(c*d+E)*f

a	+	b	(	c	*	d	+	E	)	*	f	\0



abcd*E+f*

26/01/15

Infix to Prefix Notation:

⇒ when we required to convert infix notation to prefix then we need to follow following algorithm

step-1) Read a string in infix expression and store into infix string

step-2) Reverse infix string and read 1 character at a time and perform following operations

a) If the readed character is operand then push into a string called prefix.

b) If the readed character is a operator then check if the stack is not empty and existing operator having highest priority then current operator then pop existing operator and push current operator.

c) If the existing operator having least priority then current operator then push directly

step-3) Repeat step-② until all characters of readed.

step-4) If stack is not empty then pop all the operator from stack and push into prefix string

step-5 - Repeat step-4 until stack become empty

step-6 - Reverse the prefix string and display result on console.

D

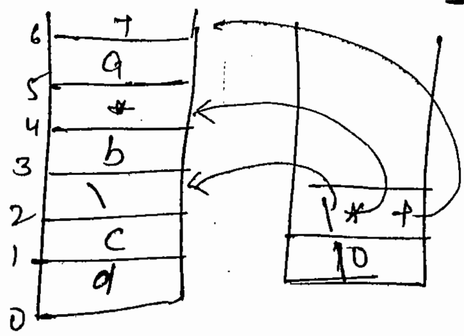
Implementation:-

```
#include <stdlib.h>
#include <dos.h>
# include < string.h>
# include < stdio.h> -
#define operand (x) (x>='a' && x<='z' || x>='A' && x<='Z'
                                 || x>='0' && x<='9')
char infix[30];
char prefix[30];
char stack[30];
int top = 0;
void init()
 {
 top = -1;
 }
void push (char x)
 {
  stack [++top] = x;
 }
char pop()
 {
 return (stack [top --]);
 }
int isp (char x)
 {
 int y;
y= (x=='(' ? 6 : x=='^' ? 4 : x=='*' ? : x=='/' ? 2:
    x=='+' ? 1 : x=='-' ? 1 : x==')' ? 0 : -1);
 }
int icp (char x)
 {
 int y;
y= (
```

```
void infix to prefix()
{
int i, l=0;
char x, y;
stack [++top]='\0';
for (i= strlen(infix)-1; i>=0; i--)
{
x = infix[i];
if operand (x))
prefix [l++]=x;
else
{
if (x = = '(')
while ((y = pop())! = ')')
prefix [l++] = y;
else
{
while (isp (stack [top]) > = icp (x))
prefix [l++] = pop();
push (x);
}
}
}
int main ()
{
system ("CLS");
init();
printf (" enter an infix :");
scanf ("%s", infix);
infix toprefix();
strrev (prefix);
printf ("The resulting prefix is %s", prefix);
return exit_success;
}
```

$a + b * c / d$

0	1	2	3	4	5	6	7
a	+	b	*	c	/	d	\0



=

d	c	/	b	*	a	+	\0

$+a*b/cd$

27/01/15: **Searching & Sorting:-**

Searching: it is a procedure of finding an element in a list of values.

*) When we are working with searching if searching element is available then it returns position of the elements if it is not available then it is called element is not found.

*) In 'C' - Programming language searching are classified into two types :- i.e

a) Linear search.

b) Binary search.

a) Linear Search: when we are searching the element in a sequence then it is called linear searching procedure.

*) Linear searching mechanism can be applied for sorted or unsorted array elements also

b) Binary Search:- In this search elements are started to search from middle position of the array.

*) If the searching element is less then of middle value then searching is process in left hand side, or if the searching element is greater then of the middle

full.

27-01-15

Searching -

searching is a procedure of finding an element an list of values.

→ ~~the If we~~ when we are working with searching if searching element is available then it returns position of the element, if it is not available then it is called element is not found.

→ In c programming language searching are classified into two types i.e

1. linear search
2. Binary search

(1) linear search -

when we are searching the element in a sequence then it is called linear searching position procedure

Page No.
Date / /

→ linear searching mechanism can be applied for sorted or unsorted array element also.

→ In binary search element are started to search from middle position of the array.

→ If the searching element is less than of middle values then searching is process in left hand side, if the seaching element is greater than of middle value then searching is ~~with~~ process in R.H.S
→ When array element are sorted order then recommended to go for binary search if it is unsorted manner then recommended to go for linear search.

Implementation of linear search -

```
#include <stdio.h>
#include < conio.h>
#include <stdlib.h>
 #define size 10
 int main()
 {
   int arr[size];
   int data, i, position, flag=0;
   clrscr();
   printf ("Enter %d values: ", size);
   for (i=0; i<size; i++)
   scanf("%d" &arr[i]);
```

```
printf("Enter searching data : ");
scanf("%d", &data);
for (i=0; i<size; i++)
{
    if (arr[i] == data)
    {
        flag=1;
        position = i+1;
        break;
    }
}
if (flag==1)
{
    printf("\n %d position is %d", data, position);
}
else
    printf("\n %d is not found", data);
return 0;
}
```

O/p - Enter 10 values: 50 60 40 70 30 80 20 90 10 45

0	1	2	3	4	5	6	7	8	9
50	60	40	70	30	80	20	90	10	45

size: $\boxed{10}$

i: ~~0~~ ~~1~~ ~~2~~ 3

data: $\boxed{70}$

position: $\boxed{= i+1 = 4}$

flag: ~~0~~ 1

Implementation of binary search.

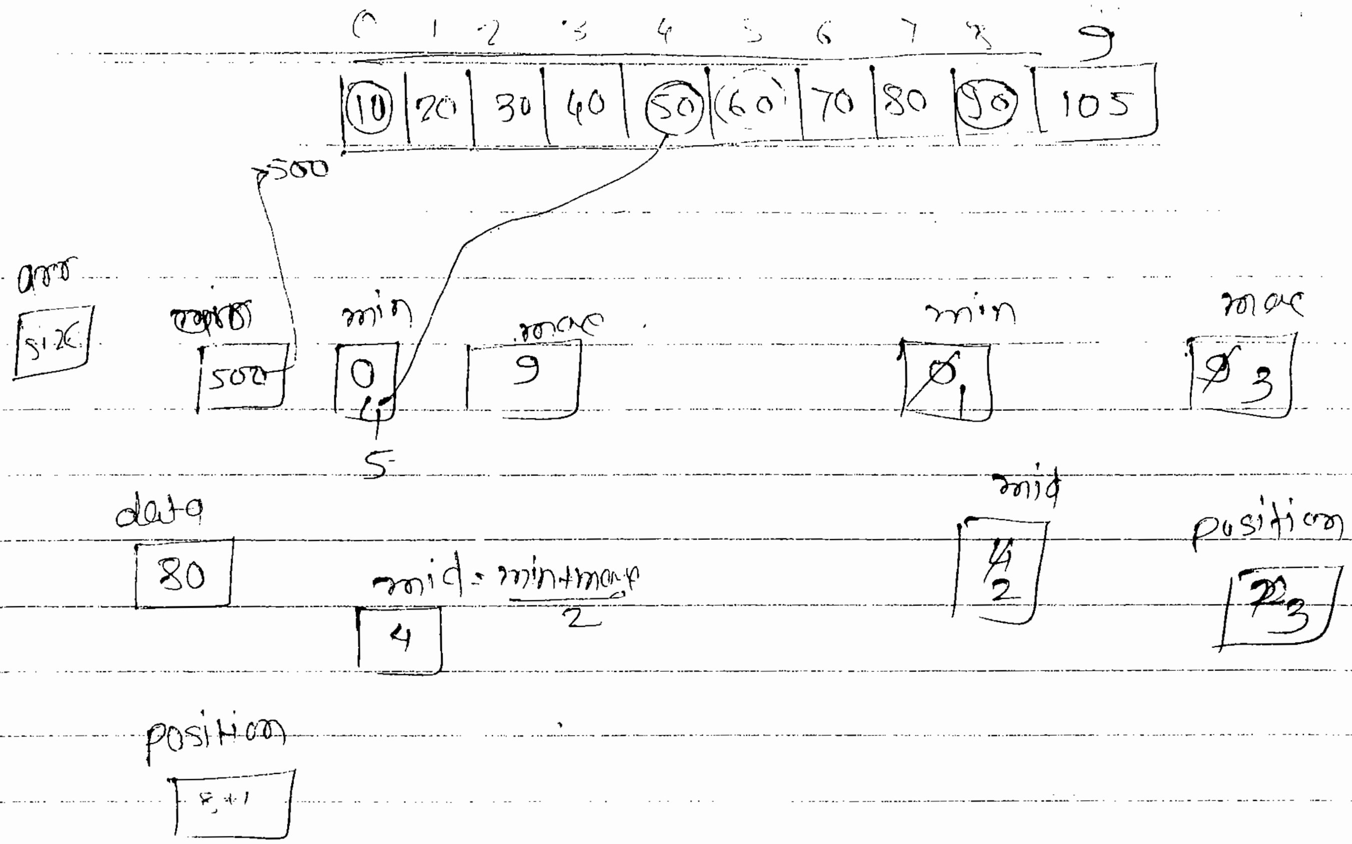
```
# include <stdio.h>
#include <stdlib.h>
# include <malloc.h>
#include <conio.h>
int main()
 {
    int *arr, size, data;
    int min, max, mid, position, i, flag=0;
    clrscr();
    printf("\n Enter array size:");
    scanf("%d", &size);
    arr = (int *) calloc (size, sizeof(int));
    printf("\n Enter %d values:", size);
    for (i=0; i<size; i++)
     scanf ("%d", &arr[i]);
    printf("\n Enter data to be search:");
    scanf("%d", &data);
  max =size -1;
  for (min=0; min<=max; min++)
   {
      if (arr[min]==data)          ①
      {
        position =min+1;
        flag=1;
        break;
      }
  mid =(min+max)/2;                ②
```

(1) main for (; min < max ; min++) then if (arr[min] == data) — 10 20 30 50 10
common mid
mid = min + max / 2 ; → (Imp)

↑data

(1) ==
(2) <=
~~(3) > =~~

```
if (arr[mid] <= data)      (3)
{
    if (arr[mid] == data)
    {
        position = mid+1;
        flag 1;
        break;
    }
    min = mid;              (4)
}
else
{
    max = mid-1;            (5)
    min = 0;
}
}
if (flag == 1)
printf("%d is at %d position", data, position);
else
printf(" DATA is not FOUND);
free (arr);
arr = NULL;
getch();
return 0;
}
```



Sorting -

it is a procedure of arranging the data in a particular order i.e asscending or decending order.

→ In CPL we having 8 types of sorting.

1. Bubble sort
2. selection sort
3. Insertion sort
4. Merge sort
5. peak sort
6. heap sort
7. Bucket or Radix sort
8. shell sort.

1. Bubble sort -

When we are working with bubble sort adjcent elements are compare until last value is fixed i.e max value of list.

→ when we are working with bubble sort if n number of unsorted element present then n-1 comparision will take place.

→ when we are working with bubble sort element are arranged in desending order but final result is assending order only.

Implementation of Bubble Sort -

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main ()
{
    int *arr;
    int size, i, j, t;
    clrscr ();
    printf ("Enter array size: ");
    scanf ("%d", &size);
    arr = (int *) calloc (size, sizeof(int));
    printf ("Enter %d values :", size);
    for ( i=0; i<size; i++)
    scanf ("%d", &arr[i]);
```

Fix from starting.
(i) i=0 [illegible] first value fixed, j.e [illegible] value.
(2) [50|60|40|70|30|80|20|90|10|45]   0<9 0<8
1<9 1<8
2<9 2<8

```
for (i=1; i<size; i++)
{
    for(j=0; j<size-i; j++)
    {
        if (arr[j] > arr[j+1])
        {
            t=arr[j] ;
            arr[j] = arr[j+1];
            arr[j+1] = t;
        }
    }
}
printf("\n sorted data are : ");
for (i=0; i<size; i++)
    printf("%d", arr[i]);
}
```

without temp: if (arr[j] > arr[j+1])
{
arr[j] = arr[j] + arr[j+1];
arr[j+1] = arr[j] - arr[j+1];
arr[j] = arr[j] - arr[j+1];
}

O/p -

Entre 10 values - 50 60 40 70 30 80 20 90 10 45

size

10

0	1	2	3	4	5	6	7	8	9
50	~~60~~	~~40~~	~~70~~	~~30~~	~~80~~	~~20~~	~~90~~	~~10~~	~~45~~
	40	60	30	70	20	80	10	~~90~~ 45	90

500 ... 518

arr
500

j < size-i

i				
~~1~~	0<9	0<8	0<7	0<6
~~2~~	1<9	1<8	1<7	1<6
~~3~~	2<9	2<8	2<7	2<6
~~4~~	3<9	3<8	3<7	3<6
~~5~~	4<9	4<8	4<7	4<6
~~6~~	5<9	5<8	5<7	5<6
~~7~~	6<9	6<8	6<7	6<6 X
~~8~~	7<9	7<8	7<7 X	
9	8<9	8<8 X		
	9<9 X			

29/01/15.

2. Selection sort:-

When we are working with selection sort always comparision will take place in sequence until first element is fixed i.e minimum value of the list.

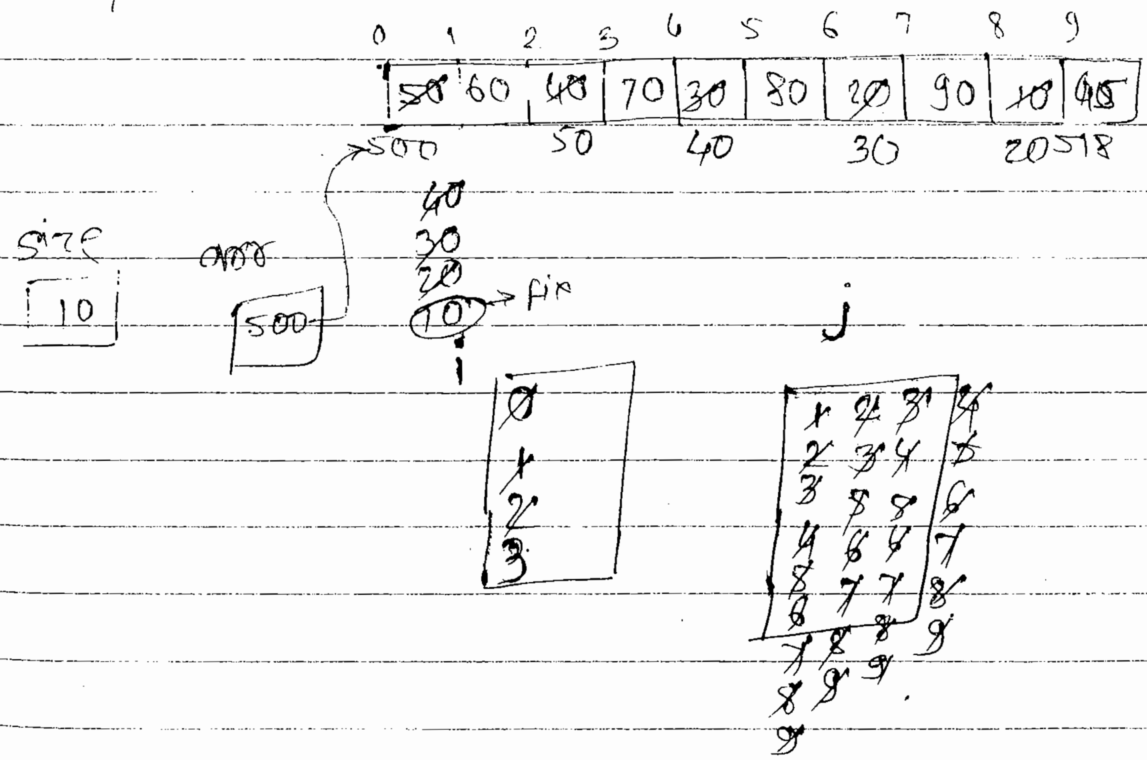
→ When we are working with selection sort if n number of unsorted elements are present n-1 comparision will takesplace.

→ When we are working with selection sort elements are arranged in asscending order & final result also assending order only.

Implementation of Selection Sort:-

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main ()
{
    int *arr, size;
    int i, t, j;
    clrscr();
    printf ("Enter array size:");
    scanf ("%d", &size);
    arr = (int *) calloc (size, sizeof(int));
    printf (" Enter %d values", size);
    for (i=0; i<size; i++)
    scanf("%d", &arr[i]);

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

```
for (j=i+1; j<size; j++)
{
    if (arr[j] < arr[i])
    {
        t = arr[i];
        arr[i] = arr[j];
        arr[j] = arr[i];
    }
}
}
printf ("sorted elements are: ");
for (i=0; i<size; i++)
printf ("%d ", arr[i]);
return 0;
}
```



3. Insertion sort

When we are working with insertion sort it will compare the element in sequentional order but whenever a element require to place in a particular position then remaining all elements will be shifted to next position.

→ In insertion sort it doesn't compare with all element's because one element position will search in proper way and to insert that element remaining all element automatically need to be arranged in next position.

→ when we are working with insertion sort to arrange the data (n-1) iteration will happen

Implementation of insertion sort -

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
int main()
{
    int *arr;
    int size, i, j, k, n;
    clrscr();
    printf(" Enter array size :");
    scanf("%d", &size);
```

```
arr = (int *) calloc (size, sizeof(int));
printf ("Enter %d value: ", size);
scanf ("          <- struck out
for (i=0; i<size; i++)
scanf ("%d", &arr[i]);

for( j=1; j<size; j++)
{
    k = arr[j];
    for( i=j-1; i>=0&& k<arr[i]; i--)
        arr[i+1] = arr[i];

        arr[i+1] = k;
}
printf ("After sorting: ");
for (i=0; i<size; i++)
printf ("%d ", arr[i]);
free (arr);
arr = NULL;
getch();
return 0;
}
```

i=8
① check 45 < arr[8] then
② ~~i=7~~ arr[i+1] = arr[i]

0	1	2	3	4	5	6	7	8	9
50	60	40	70	30	80	20	90	10	45

arr → 500

10 20 30 40 50 60 ~~70~~ ~~80~~ ~~90~~ 45

⑤ condn fail put data 50 60 70 80 90 ← ④

i (cond) ① ② ③ ④ last

0 1 2 3 8
-1 0 1 2 7
(-1+1) -1 0 . 6
-1 0 5
(-1+1) -1 0 4
~~(-1+1) -1~~

j (cond) ① ② ③ ④ ⑤ last

1 2 3 4 5 6 9

k

~~60~~ ~~80~~
~~40~~ ~~20~~
~~70~~ ~~90~~
~~30~~ ~~10~~
45 last

Page No.
Date / /

4. Merge sort -
This method we required to used when we required to combine two sorted array list

→ this sorting procedure we can apply for sorted & unsorted array also.

→ when we are apply for unsorted array list then first refer to sort every individual array then we need to go for merge sort

Implementation of merge sort :-

```
#include <stdio.h>
#include <malloc.h>
#include < conio.h>
~~int main()~~
{
 void bsort (} int arr[], int size)
 {
   int i, j, t ;
   for (i=0; i<size; i++)
   {
     for ( j=0 ; j<size -i -1 ; j ++)
     {
       if (arr[j] > arr[ j+1] )
       {
         t = arr[ j];
```

```
arr[j] = arr[j+1];
arr[j+1] = t;
}
}
}
}
int main()
{
int *arr1, *arr2, *arr3;
int s1,s2,s3;
int i,j,k;
clrscr();
printf("\n Enter size of array 1: ");
scanf("%d",&s1);
arr1 = (int *) calloc(s1, sizeof(int));
printf(" Enter %d value", s1);
for (i=0; i<s1; i++)
scanf("%d", & arr[i]); bsort(arr1, s1);
printf("\n Enter size of array 2: ");
scanf("%d",&s2);
arr2 = (int *) calloc(s2, sizeof(int));
printf(" Enter %d value ", s2);
for (j=0; j<s2; j++)
scanf("%d", & arr[j]);
bsort(arr2, s2);          // sorting array2.

printf("\n sorted arr1: ");
for(i=0; i<s1; i++)
printf("%d ", arr[i]);
```

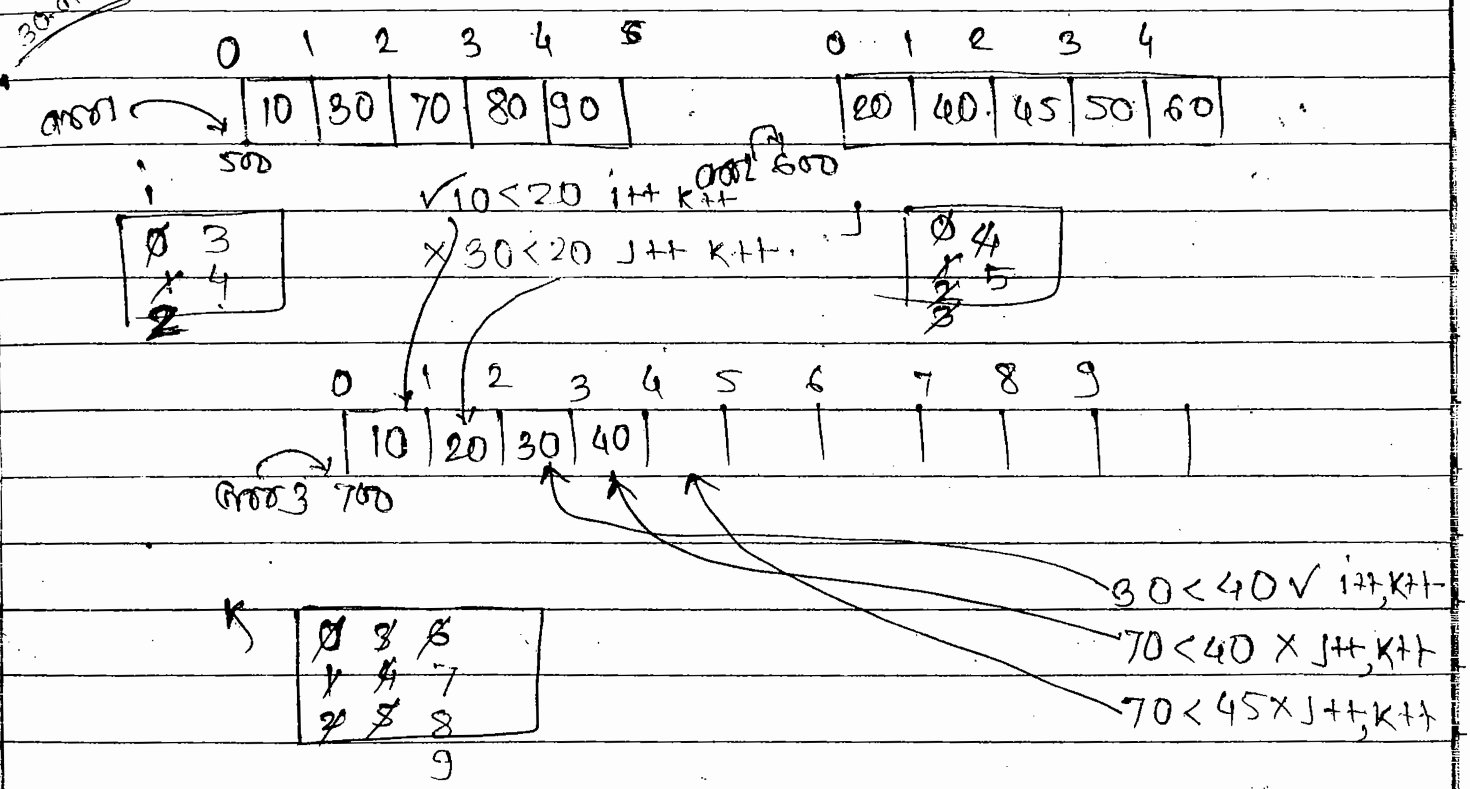
Page No.

Date / /

```
printf ("\n sorted array 2:");
for ( j=0; j<s2; j++)
printf ( "%d", arr[j]);

s3 = s1+s2;
arr3 = (int*) calloc (s3, sizeof (int));
k=i=j=0;

while ( k<s3)
{
   if (arr1[i] < arr2 [j])
     arr3 [k++] = arr1 [i++];
     else
     arr3 [k++] = arr2[j++];

   if (i==s1 ||j==s2)
     break;
}
while (i<s1)
{
 arr3 [k++] = arr1 [i++];
}
while (j<s2)
 {
     arr3 [k++] = arr2 [j++];
 }
printf ("\n sorted arr3: ");
for ( k=0; k<s3; k++)
printf ("%d", arr3[k]);
```

```c
free (arr1);
free (arr2);
free (arr3);
arr1 = arr2 = arr3 = NULL;
getch();
 return 0;
}
```

30-01-15



5. Implementation of Quick sort -

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

```
enum bool {FALSE, TRUE};

void display(int arr[], int low, int up)
{
  int i;
  for (i=low; i<=up; i++)
    printf("%d", arr[i]);
}

void Quick(int arr[], int low, int up)
 {
    int piv, temp, left, right;
    enum bool pivot_placed = FALSE;
    left = low;
    right = up;
    piv = low;

   if (low >= up)
     return;
  while (pivot_placed == FALSE)
  {
   while (arr[piv] <= arr[right] && piv != right)
    right = right - 1;

    if (piv == right)
    pivot_placed = TRUE;
   if (arr[piv] > arr[right])
   {
        temp = arr[piv];
```

```
        arr [piv] = arr [right] ;
        arr [right] = temp;
        piv = right;
    }
    while (arr [piv] >= arr [left] && left != piv)
        left = left + 1;

    if (piv == left)
    pivot_placed = TRUE ; // data is sorted

    if (arr [piv] < arr [left])
      {
        temp = arr [piv];
        arr [piv] = arr [left];
        arr [left] = temp;
        piv = left;
      }
    } // end of while

    quick ( arr , low, piv-1) ;
    quick ( arr , piv+1, up);

} // end of quick;

int main()
{
    int array [MAX], n, i;
    printf (" Enter the number of elements: ");
    scanf("%d", &n);
```

Page No.
Date / /

```
printf ("\n Enter %d values : ",n);
for (i=0; i<n ; i++)
scanf( "%d", &array [i]);

printf (" Unsorted list is : ");
display (array, 0, n-1);        // calling for printing

printf ("\n");
quick (array, 0, n-1) ; // (calling for sorting)

printf(" sorted list is: ");
display (array, 0, n-1);        calling for printing
printf("\n");
getch ();
return 0 ;
}
```

0	1	2	3	4	5	6	7	8	9
~~50~~ 45	60	40	70	30	80	20	90	10	~~45~~ 50

arr → 500

n: 10

(1) tie 50

pp: False or 0

piv: ~~0~~ ~~9~~ 8, 9, 1, 8, 3, 6, 5

L: ~~0~~ 1, 2, 3, 4, 5

R: 9, 8, 7, 6, 5

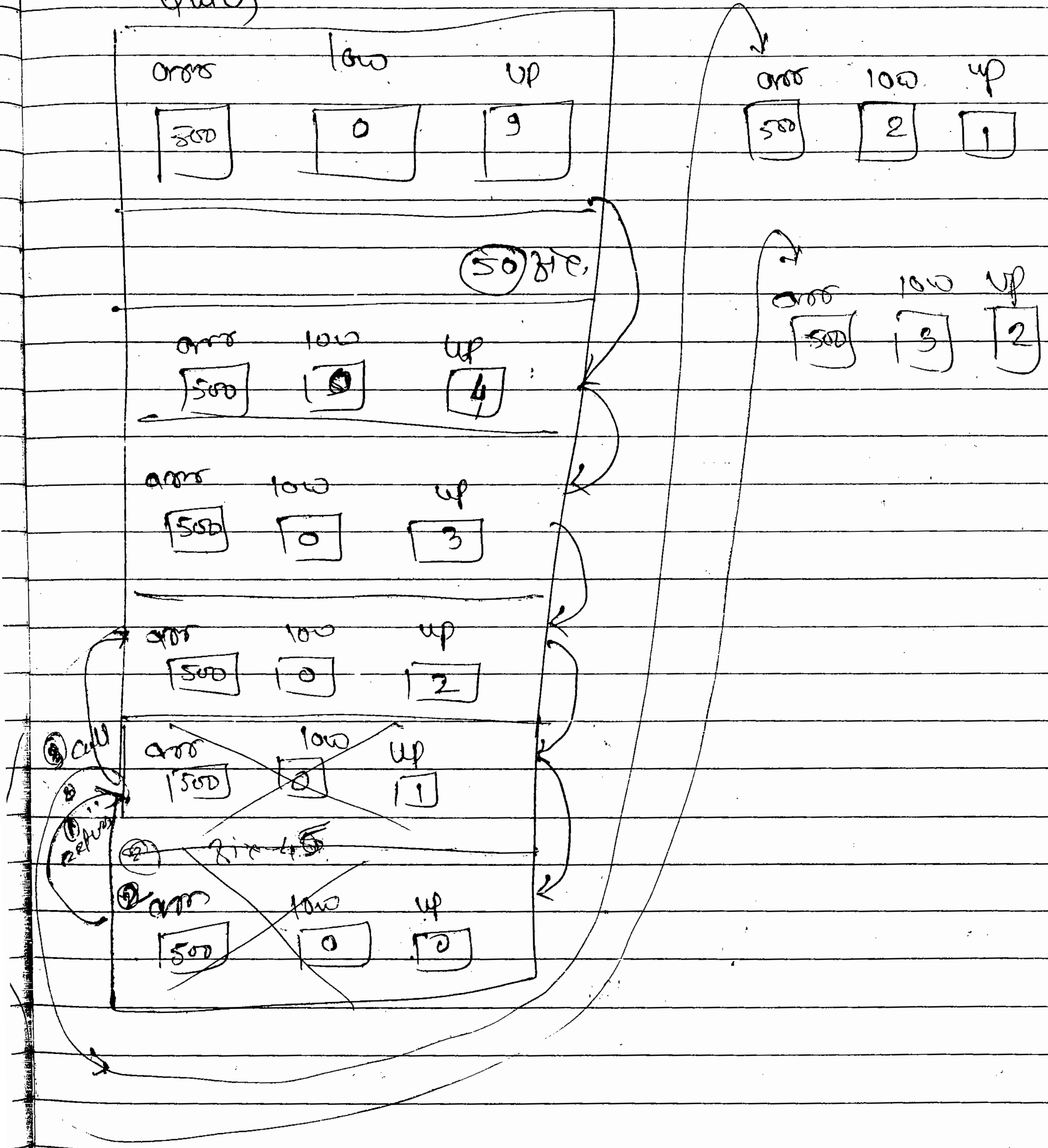
t: 0 50

low 6 up 5

Page No.

Date / /

Quick

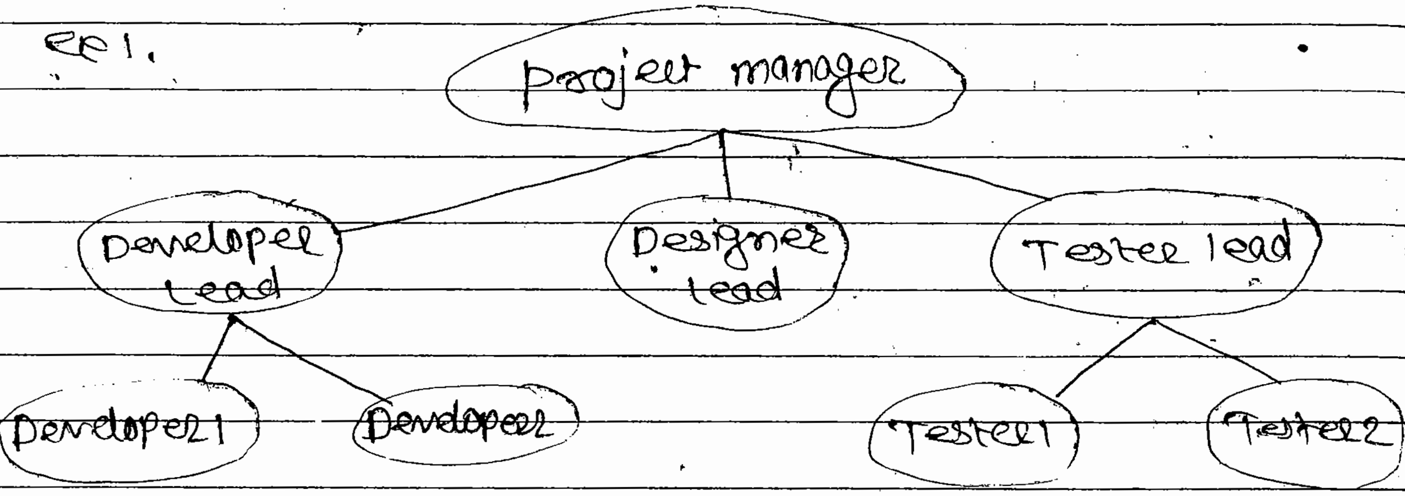


Trees -

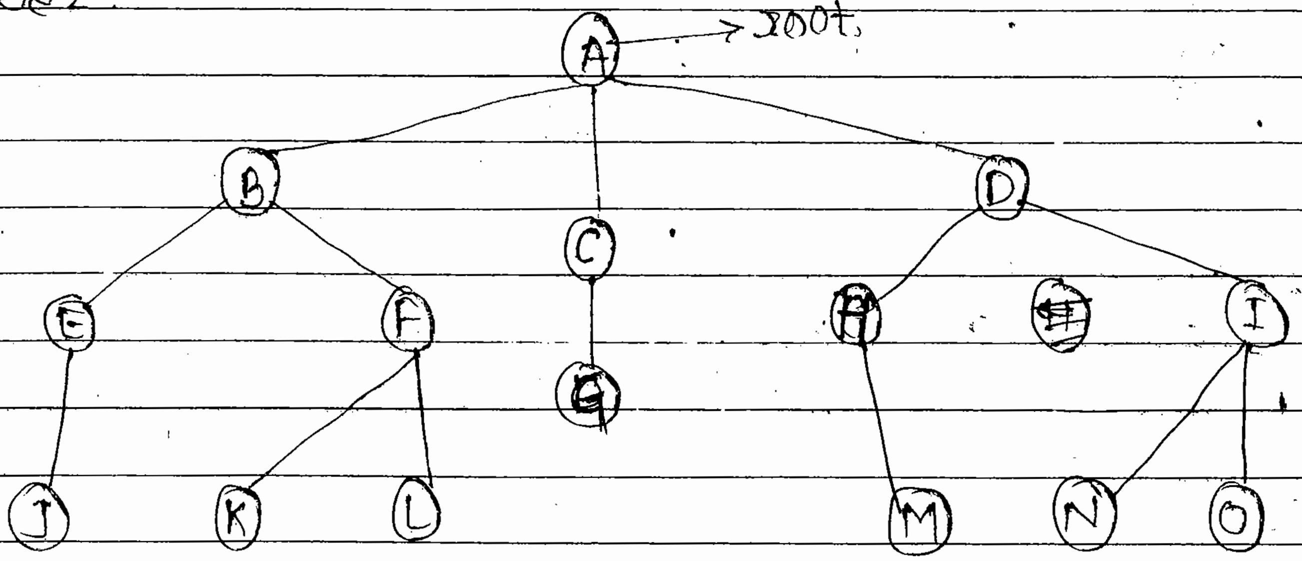
A tree is a non-linear data structure in which item are arranged in sorted sequence.

→ Tree data structure used to represente hierachical ~~re~~ relationship betn existing serial items.

ex 1.



ex 2



Page No.
Date / /

* Tree terminology -

Root - it is a specially designed data item in a tree. it is the first in the hierarchical arrangement of data items.
i.e A is root

Node - Each data item in a tree is called ~~in~~ a Node.
→ A Node is the basic structure in a tree.
→ Always node specifies the data inform & links to other data items. There are 14 nodes in the above tree.

Degree of a Node -
Number of sub tree's of a node is called degree of a node. in the above tree.
→ The degree of node A is 3.

→ Total no. of child nodes of a node is called degree of a node.
→ The degree of node A is 3
The degree C is 1
B is 2
M is 0.

Degree of tree -
Maximum degree of a nodes in a given tree is called degree of tree.

* In the ~~root~~ above ~~th~~ tree the node A has degree 3.

Terminal node -

A node with degree zero is called a terminal node ~~of~~ or leaf.

→ In the above tree there are seven terminal nodes are available, they are J, K, L, G, M, N & O.

Non Terminal node

Any node (except the root node) whose degree is not zero ~~to~~ is called non-terminal node. i.e B, C, D, E, F, H & I.

Siblings -

The children node of a given parent ~~of~~ node are called siblings, They are also called brother i.e E & F

Level -

The entire tree structure is level in such a way that root node is always at level 0. then its immediate children are at level 1 and there immediate children are at level 2 and so on upto the terminal node In above tree level 3.

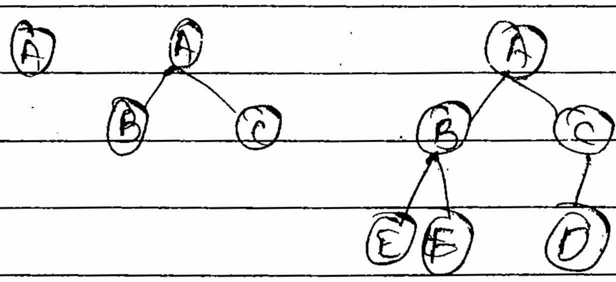
Edge
it is a connecting line of ~~betn~~ two nodes that is the line ~~is~~ drawn from one node to another node is called edge

Path -
it is a sequence of consecutive edges from source node to the destination node.
→ In the above tree, the path between A & J is given by the node pairs (A,B), (B,E), & (E,J).

~~Dept~~ Depth -
Maximum level of any node in a given tree i.e height of tree.

Forest -
it is a set of disjoint tree. in above tree if we remove ROOT then it become forest.

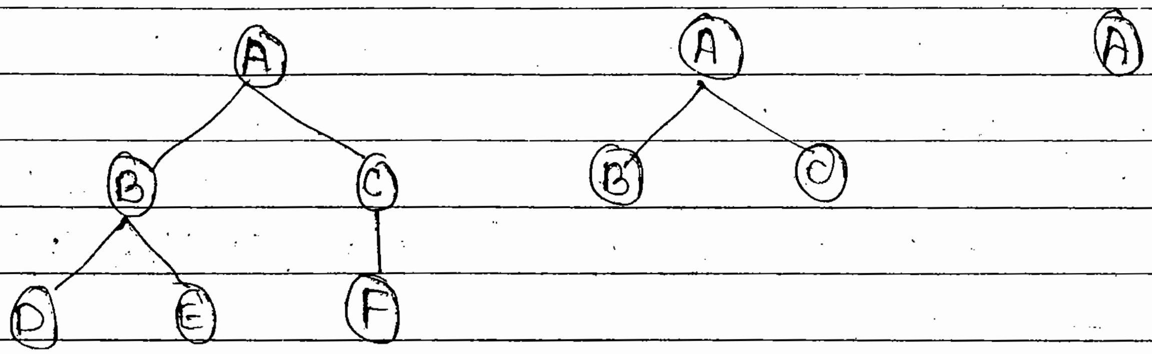
* Binary tree -
cond(n)
level 0 - 1 node
level 1 - 2 sub tree
level 3 - may be two or one



A binary tree is a finite set of data items which is either empty or consist of a single item called the root and two disjoint binary tree called the left subtree & right subtree.

In a binary tree the max degree of any node is at most two that means there may be a zero degree node or a one degree node & two degree node.

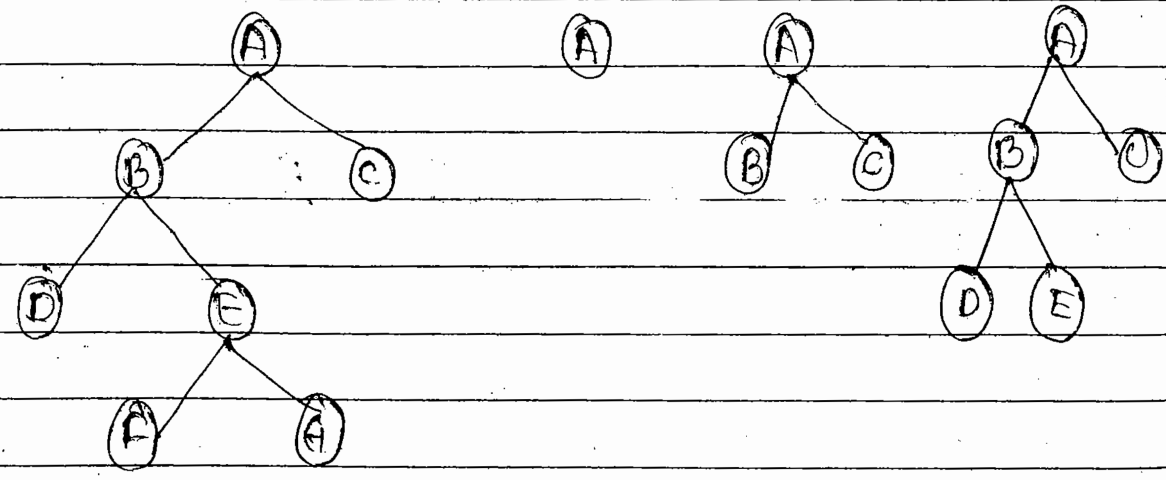
ee.



strictly binary tree -

if every non-terminal node in a binary tree consist of ~~of~~ a non-empty left subtree & right subtree, then such a tree is called strictly binary tree.

ee.

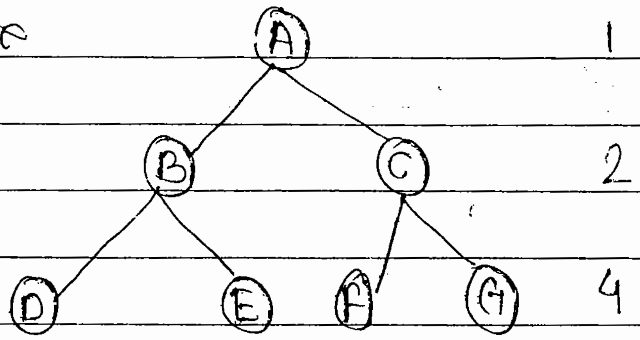


compleate binary tree-

if there are 'm' nodes at level l then a binary tree contains at most '2m' nodes at level l+1 and so on it is called compleate binary tree

→ In compleate binary tree there is exactly one node at level 0, two nodes at level 1 & four nodes at level 2 and so on.

Ex



Implementation of binary tree:-

when we are working with binary tree dedicated node is called root & it always holds first node data only.

→ when we are working with binary tree every node required to maintain 3 fields i.e left, data, right.

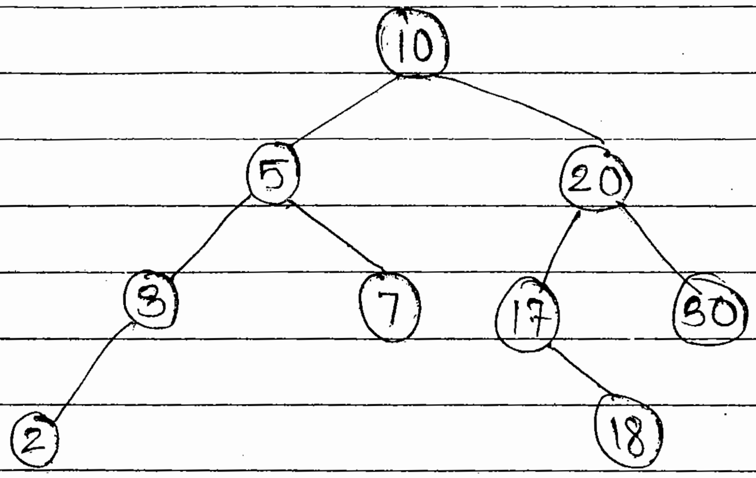
→ left ~~is~~ member is a pointer which always holds left sub-tree information, right member is a ptr which always holds right subtree information.

→ when we are organizing the b.T then need to follow following steps.

① 1st entered element always required to store in root position

② Newly added element value is less than of root element then it should be organised in left subtree.

③ If Newly entered element is greater than of root value then it should be organised in right sub-tree.



Tree traversal -

In fully developed binary tree there are three type of traversal is posible i.e

① preorder traversal.
② Inorder traversal.
③ postorder traversal.

① Preorder traversal -

when we are working with preorder we need to perform following task.

① visit the root node first
② travel the left sub tree in preorder
③ travel the right sub tree in preorder.

10 5 3 2 7 20 17 18 30

Page No.
Date : / /

② Inorder traversal -

When we are travelling inorder procedure then

① first visit left sub-tree in inorder
② visit the root noode
③ travel in ~~sut~~ right sub-tree in inorder.

Ans- 2 3 5 6 7 10 17 18 20 30

post-order -

When we are travelling in post order then
① travel left sub tree in post-order.
② travel right sub tree in post order.
③ visit root node atlast.

2 3 6 7 5 18 17 30 20 10

* logical Implementation

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

struct btree
{
    struct btree *left;
    int data;
    struct btree *right;
};
typedef struct btree node
```

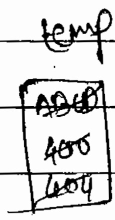
Page No.
Date / /

```
void insert (node **, int);
void inorder (node *);
void preorder ( node *);
void postorder ( node *);

int main ()
{
   node *root = NULL;
   int choice, num;
   clrscr();
   while (1)
   {
    clrscr();
    printf("\n1 for insert a node in the BT:");
    printf ("\n2 for Display (Preorder) the BT:");
    printf ("\n3 for Display (inorder) the BT:");
    printf("\n4 for Display (postorder) the BT:");
    printf ("\n 5 for Exit ......:");
    scanf("%d", &choice);
    switch ( choich)
    {
     case1:
        printf(" Enter the data:");
        scanf ("%d ", &num);
        insert (&root, num);
        break;
     case 2:
        printf ("\n pre order traversal :");
        preorder (root);
```

Page No.
Date: / /

```
getch ();
 break;

case 3:
    printf("\n In-order Traversal : ");
    inorder (root);
    getch ();
    break.
case 4;
    printf("\n post-order traversal :");
    postorder (root);
    getch ();
    break;
case 5; free (root);
        return 0;
    default : printf("\n Invalid option : ");
  }
 }
}
void insert (node **temp, int num)
{
    if (* temp == NULL)
    {
     *temp = malloc (sizeof (node));
     (*temp) -> left = NULL;
     (*temp) -> ~~right~~ data = num;
     (*temp) -> right = NULL;
      return;
    }
    else
```

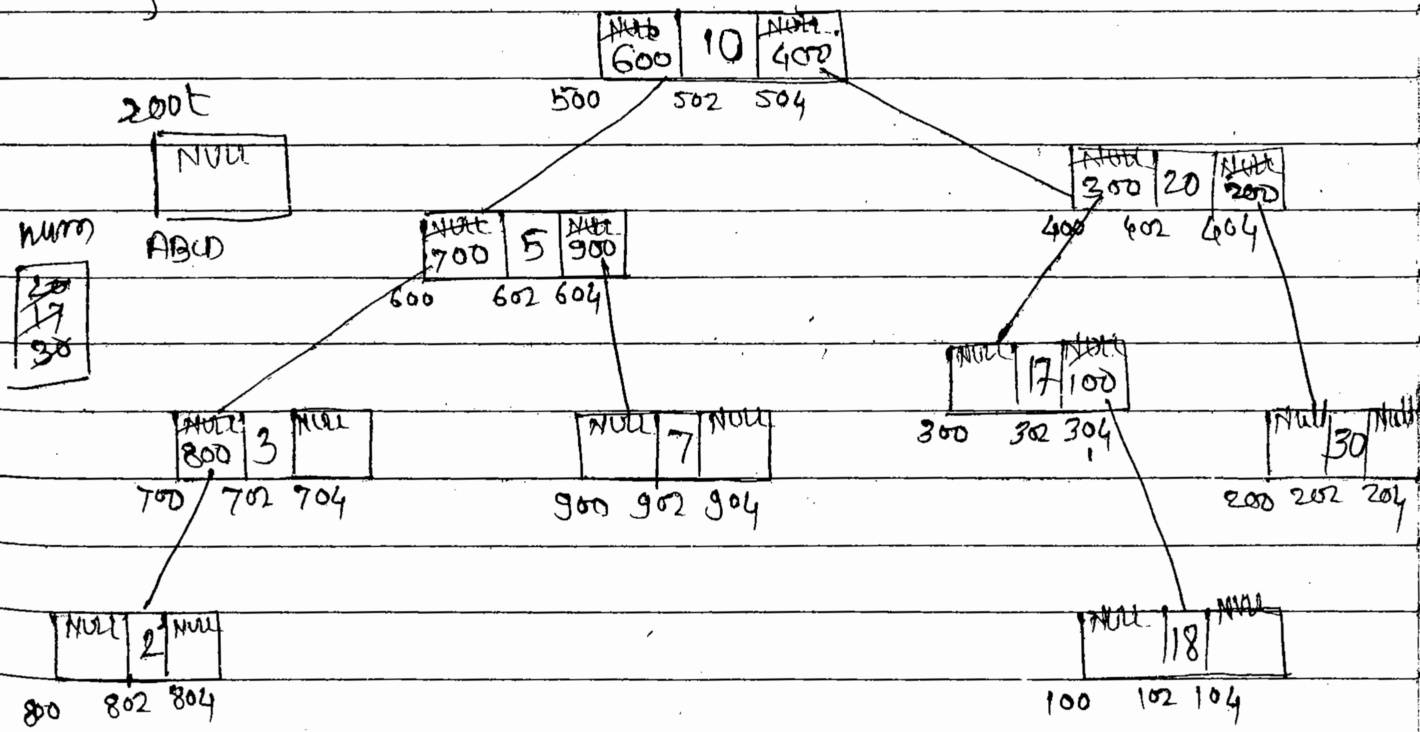
```
{
  if (num < (*temp)->data)
   insert (&((*temp)->left), num);
  else
   insert (&((*temp)->right), num);
 }
 return;
}

void inorder (node *temp)
 {
   if (temp != NULL)
   {
     inorder (temp->left);
     printf("%d", temp->data);
     inorder (temp->right);
   }
   else
   return;
 }

void preorder (node *temp)
 {
   if (temp != NULL)
   {
     printf("%d", temp->data);
     preorder (temp->left);
     preorder (temp->right);
   }
```



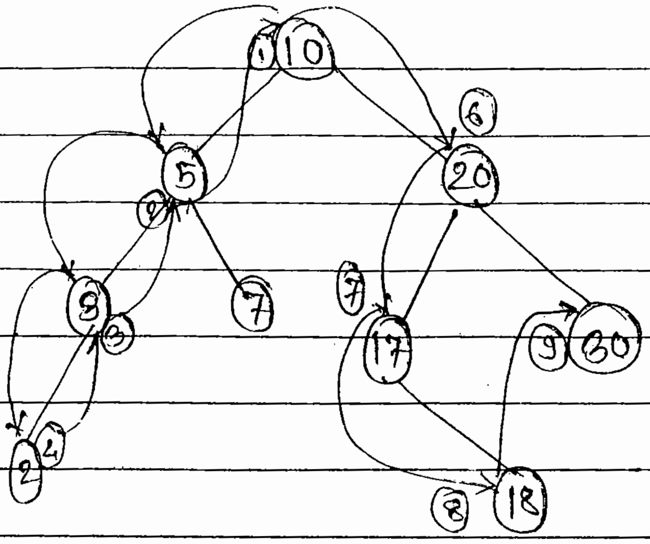
Page No.

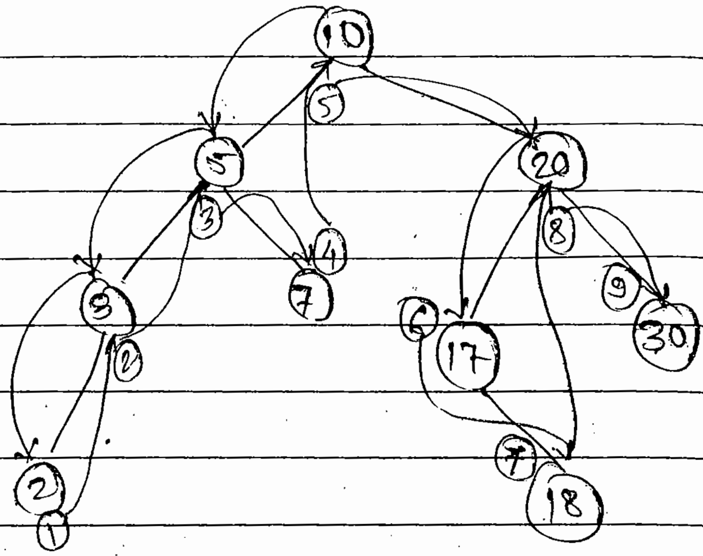
Date: / /

```
else
 return;
}

void postorder (node *temp)
{
   if (temp != NULL)
   {
      postorder (temp->left);
      postorder (temp->right);
      printf ("%d", temp->data);
   }
   else
   return;
}
```



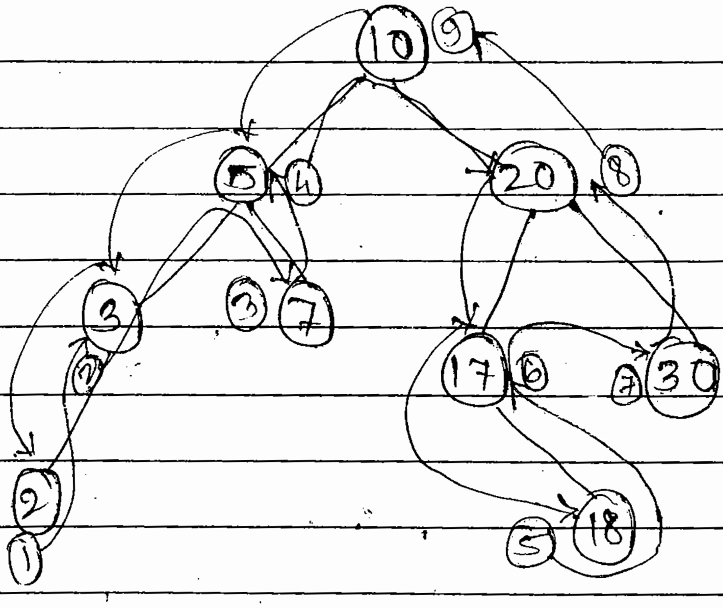
Page No.
Date / /





preorder -
10, 5, 3, 2, 7, 20, 17, 18 30

inorder - 2, 3, 5, 7, 10, 17, 18
20, 30.

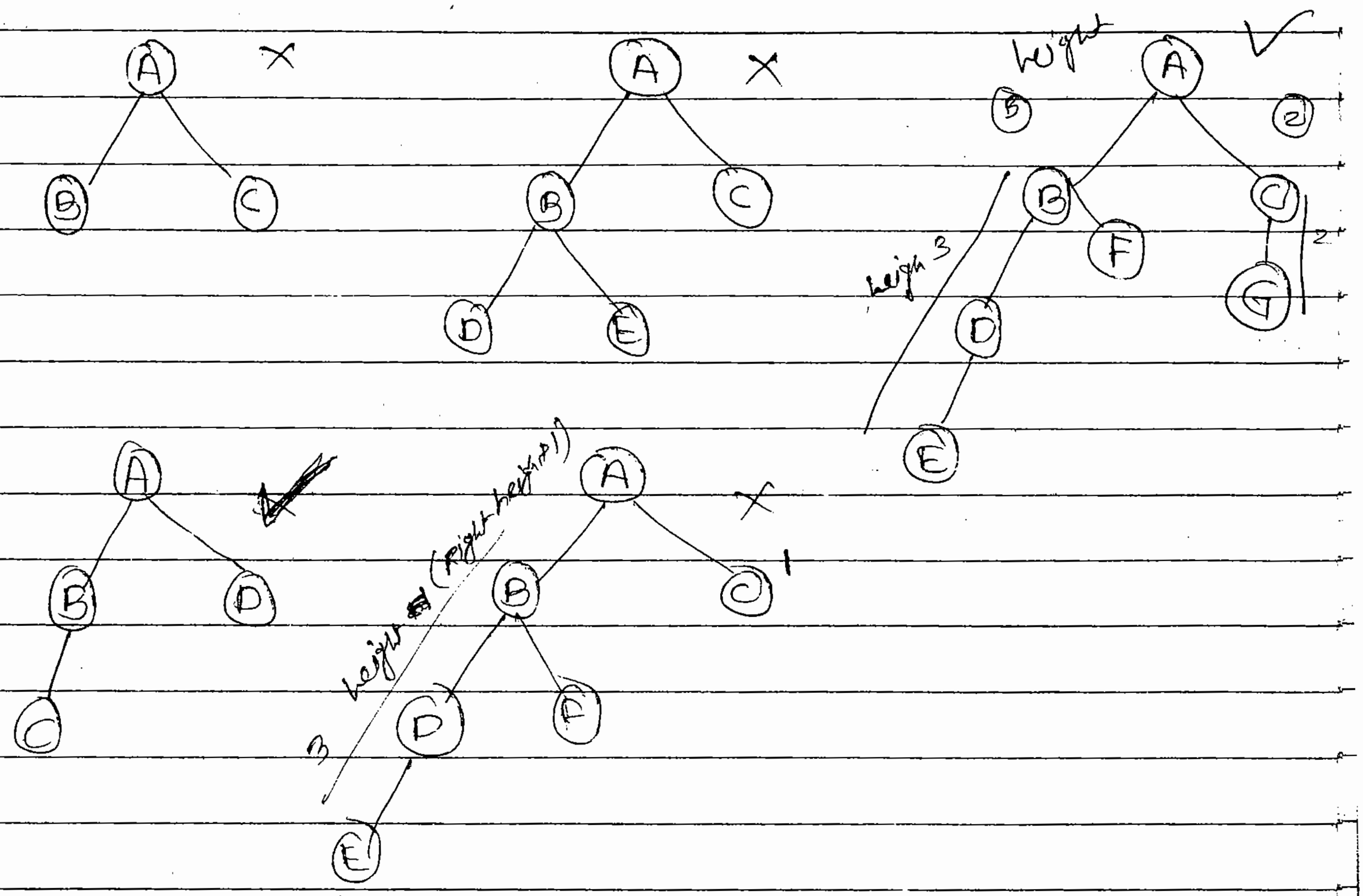


Postorder - 2, 3, 7, 5, 18, 17, 30, 20, 10

3-02-15

AVL Tree (Adelson - Velskii & Landis) -
it is a self balenced - binary search tree
(2) when we are working with AVL tree left sub-tree height should be equal to right sub tree height plus 1 should be always.

③ if above condition is not satisfied then it is called binary search tree only.



④ when we are deleting an element from AVL tree then automatically we required to rebalance it depends on availability of elements.

Implementation of AVL tree with ~~all~~ traversing process

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
typedef struct node
{
    struct node *left;
```

Page No.
Date / /

```
int data;
struct node *right;
int ht;
}node;

node *Insert (node*,int) // return address of a node
node *Delete (node *,int); // return address of a node
~~void preorder~~
void preorder (node *); // root address is parameter
void inorder (node*) ;         — // —
int height (node *);
node *rotateright (node *); // auto balence in right subtree
node *rotateleft (node *); // auto balence in left subtree.
node *RR(node *); // right to right
node * LL(node *); // left to left.
node * LR (node *); // Left to right
node * RL (node *); // right to left.
int BF(node *) ; // balence factor.
int main ()
{
  node * root = NULL;
  int x,n,i,OP;
  do
   {
      printf ("\n1 create:");
      printf ("\n2 Insert:");
      printf ("\n3 Delete:");
      printf ("\n4 print :");
      printf ("\n5 Quit: ");
      printf("\n Enter your choice:");
```

Page No.
Date: / /

```
scanf("%d", &op);
switch (op)
{
case1: printf("\n Enter no.of elements : ");
        scanf("%d",&n);
        printf("\n Enter tree data:");
        root=NULL;
        for(i=0 ; i<n; i++)
        {
         scanf("%d",&x);
         root=insert (root, x);
        }
        break;
case2: printf("\n Enter a data :");
        scanf("%d",&x);
        root =Insert (root, x);
        break;
case3 : printf("\n Enter a data :");
        scanf("%d",&x);
        root= Delete (root, x);
        break;
case4: printf("\n preorder sequence : \n");
        preorder (root);
        inorder(root);
        printf("\n");
        break;
        }
}
} while (op! =5);
```

```
return EXIT_SUCCESS;
}

node *Insert (node *temp, int x)
{
    if (temp == NULL)
    {
        temp = (node *) malloc (sizeof (node));
        temp -> data = x;
        temp -> left = NULL;
        temp -> right = NULL;
    }
    else
        if (x > temp -> data) // insert in right subtree
        {
            temp -> right = Insert (temp -> right, x);
            if (BF (temp) == -2)
                if (x > temp -> right -> data)
                    temp = RR (temp);
                else
                    temp = RL (temp);
        }
        else
            if (x < temp -> data)
            {
                temp -> left = Insert (temp -> left, x);
                if (BF (temp) == 2)
                    if (x < temp -> left -> data)
                        temp = LL (temp);
```

Page No.
Date / /

```
        else
          temp = LR( temp);
        }
        temp->ht = height (temp);
        return (temp);
}

node * Delete (node *temp, int x)
{
    node *p;
    if ( temp ==NULL)
    {
      return NULL;
    }
    else
     if (x > temp->data)  // insert in right subtree.
    {
      temp -> right = Delet (temp->right, x);
      if (BF (temp) ==2)
        if(/BF( temp->left) >=0)
         temp  = LL(temp);
         else
         T = LR (temp);
    }
    else
     if (x<temp -> data)
     {
       temp->left = Delete (temp->left, x);
       if (BF (temp)== -2)// rebalence during
```

Page No.
Date / /

```
    if (BF(temp -> right) <= 0)
       temp = RR(temp);
       else
       temp = RL(temp);
  }
  else
  {
     // data to be deleted found.
     if (temp -> right != NULL)
     {
       / deleted its inorder successor
         p = temp -> right;
         while (p -> left != NULL)
           p = p -> left;
         temp -> data = p -> data;
         temp -> right = Delete (temp -> right, p -> data);
         if (BF(temp -> left) >= 0)
           temp = LL(temp);
           else
           temp = LR(temp);
     }
     else
     return (temp -> left);
   }
       temp -> ht = height(T);
       return (temp);
}
```

```
int height (node *temp)
{
    int lh,rh;
    if (temp==NULL)
    return 0;
    if (temp->left ==NULL)
    lh=0;
    else
    lh = 1 + temp->left->ht;
    if (temp->right ==NULL)
    rh=0;
    else
    rh =1+temp->right->ht;
    if (lh>rh)
     return (lh);
    else
     return (rh);
}
node * rotateright (node *temp)
{
    node *y;
    y=x->left;
    temp->left =y->right;
    y->right = temp;
    temp->ht = height (temp);
    y->ht=height (y);
    return y;
}
```

```
node * rotateleft (node * temp)
{
    node *y;
    y = temp -> right;
    temp -> right = y -> left;
    y -> left = temp;
    temp -> ht = height (temp);
    y -> ht = height (y);
    return y;
}
node * RR(node * temp)
{
    temp = rotateleft (temp);
    return (temp);
}
node *LL (node * temp)
{
    temp = rotateright (temp);
    return (temp);
}
node *LR (node * temp)
{
    temp -> left = rotateleft (temp -> left);
    temp = rotateright (temp);
    return (temp);
}
node *RL (node * temp)
{
    temp -> right = rotateright (temp -> right);
```

```
temp = rotateleft (temp);
return (temp);
}

int BF (node *Temp)
{
    int lh, rh;
    if (temp == NULL)
    return 0;
    if (temp -> left == NULL)
    lh = 0;
    else
    lh = 1 + temp -> left -> ht;
    if (temp -> right == NULL)
    rh = 0;
    else
    rh = 1 + temp -> right -> ht;
    return (lh - rh);
}

void preorder (node *temp)
{
    if (temp != NULL)
    {
        printf ("%d (BF=%d)", temp -> data, BF (temp));
        preorder (temp -> left);
        preorder (temp -> right);
    }
}
```

```
void inorder (node *temp)
{
    if (temp! = NULL)
    {
        inorder (temp -> left);
        printf("%d (BF= %d)", temp -> data, BF(temp));
        inorder (temp -> right);
    }
}
```

04-02-15

Graphs -

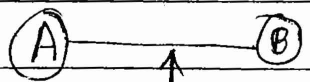
A graph 'G' Consist of a set 'V' of vertices (nodes) & a set 'E' of Edge.

→ 'V' is a finite & non-empty set of vertices

→ 'E' is a set of pairs of vertices, their are called edge.

→ V(G) read as V of G, is a set of vertices.

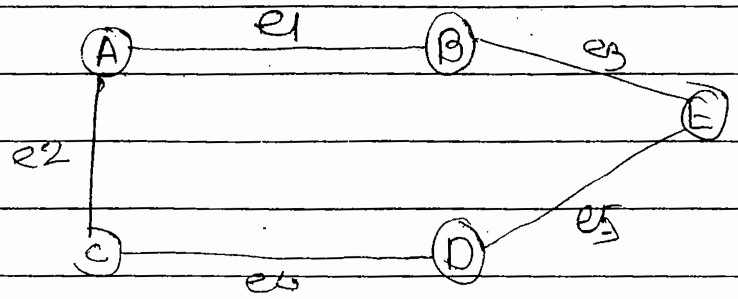
→ E(G) read as E of G, is a set of edges.



node or vertice   edge (relation bet A & B or Link)

V (G) = { V1. V2, V3, V4, V5}

E(G) = { e1, e2, e3, e4, e5}

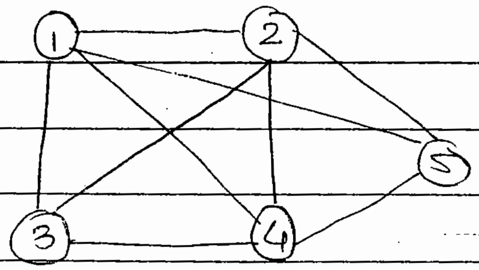


there are five vertices & five edges in the graph.

- Graph can be represent by using number also we have numbered the nodes as 1,2,3,4 & 5 so

$V(G) = \{1, 2, 3, 4, 5\}$

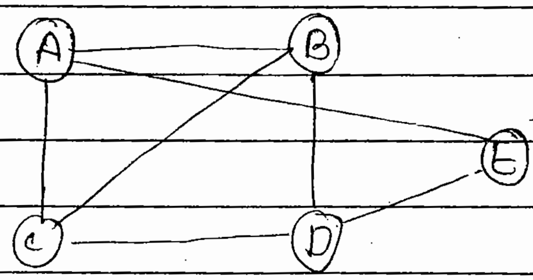
$E(G) = \{(1,2), (2,4), (2,3), (1,4), (1,5), (4,5), (3,4)\}$



Graph terminology -

1) Adjcent vertices -

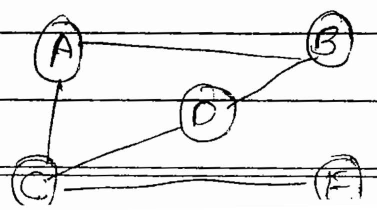
vertex V1 is said to be adjcent to a vertex V2 if there is an edge (V1, V2) or (V2, V1).



Adjcent vertex - (A,C) (A,B) (A,E) (C,D) (B,D) (D,E) (B,C)

Path -

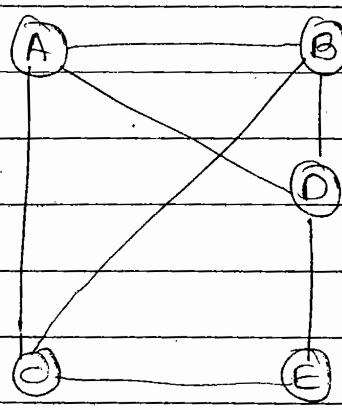
A path from vertex w is a sequence of vertices, each adjcent to the next.



Page No.
Date / /

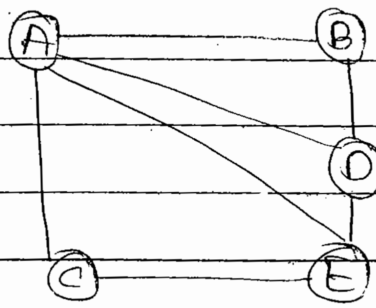
$W(A, E) = \{ (A,B)(B,D), (D,C), (C,E) \}$
$W(A, E) = \{ (A,C), (C,E) \}$

Cycle -
A cycle is the path in which first & last vertices are the same.

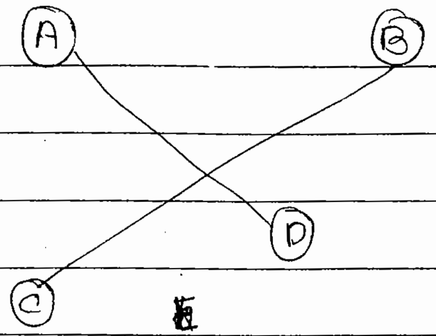


(reverse dirn is possible)

~~Undir~~ connected graph -
A graph is called if there exists a path from any vertex to any other vertex.

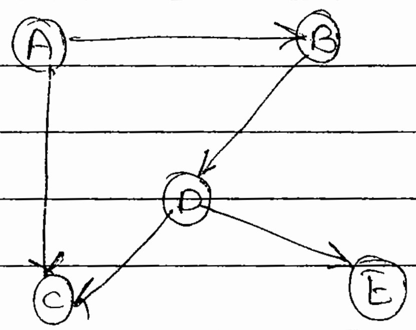


unconnected graph -
A graph is called unconnected if there no path is exists between any one of the vertex



Directed graph -

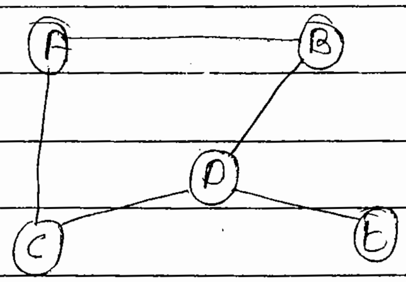
A graph is called directed if there exists a path direction from any vertex to any other vertex.



(it doesn't support reverse dirn)

undirected graph

if there is no path direction is exist bet the vertex.

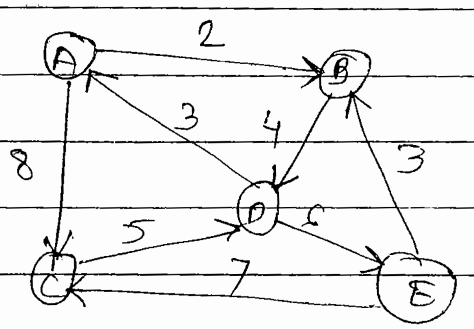


Degree -

The number of edges incident is called degree.

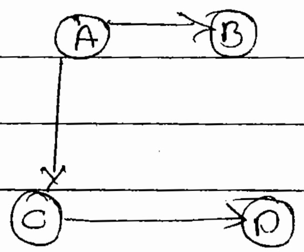
Weighted graph -

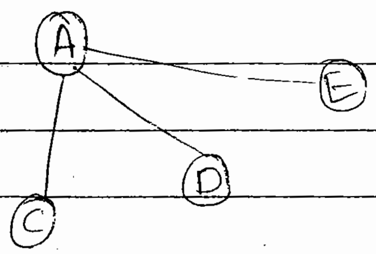
A graph is said to be weighted graph if every edge in the graph is assigned some weight or value.

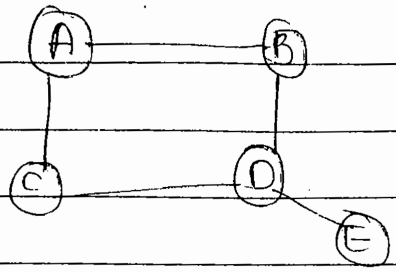


Tree -

A graph is said to be a tree if it is connected & there is no cycles in the graph.



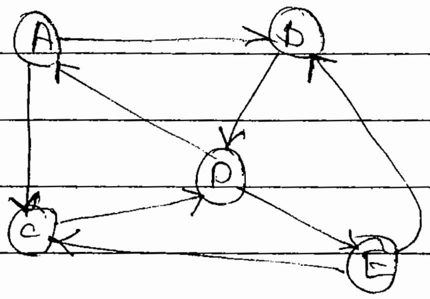




A degree → 2 ((A,B)(A,C))
B degree → 2 ((B,A),(B,D))
C degree → 2 ((C,A),(C,D))
d degree → 3 ((D,C)(D,B)(D,E))
E degree → 1 (E,D)

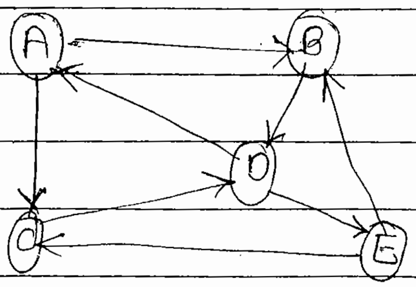
Indegree -
Incomming edges of vertex is called indegree

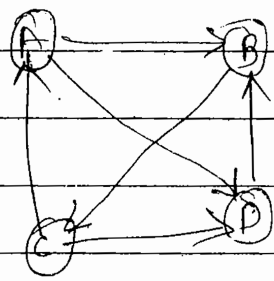
outdegree -
outgoing edges of vertex is called outdegree.



ID(A) - 1	OD(A) - 2
ID(B) - 2	OD(B) - 1
ID(C) - 2	OD(C) - 1
ID(D) - 2	OD(D) - 2
ID(E) - 1	OD(E) - 2

complete graph -
A graph G is said to complete or fully connected if there a path from every vertex to every other vertex.





/* Program of sorting using shell sort */

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

int main()
{
     int arr[MAX],i,j,k,n,incr;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }
    printf("Unsorted list is :\n");
   for(i=0;i<n;i++)
     printf("%d ",arr[i]);
 printf("\nEnter maximum increment : ");
 scanf("%d",&incr);
 while(incr>=1)
 {
   for(j=incr;j<n;j++)
   {
      k=arr[j];
      for(i = j-incr; i >= 0 && k < arr[i]; i- =incr)
            arr[i+incr]=arr[i];
      arr[i+incr]=k;
   }
   printf("Increment=%d \n",incr);
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   incr=incr-2;
}
printf("Sorted list is :\n");
for (i = 0; i < n; i++)
   printf("%d ", arr[i]);
printf("\n");
return 0;
}
```

/* Program of sorting through heapsort*/

```
# include <stdio.h>
int arr[20],n;
int main()
{
  int i;
  printf("Enter number of elements : ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
   printf("Enter element %d : ",i+1);
   scanf("%d",&arr[i]);
  }
  printf("Entered list is :\n");
  display();
  create_heap();
  printf("Heap is :\n");
  display();
  heap_sort();
  printf("Sorted list is :\n");
  display();
}

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

void create_heap()
{
 int i;
 for(i=0;i<n;i++)
  insert(arr[i],i);
}

void insert(int num,int loc)
{
 int par;
 while(loc>0)
 {
  par=(loc-1)/2;
  if(num<=arr[par])
  {
   arr[loc]=num;
   return;
  }
  arr[loc]=arr[par];
  loc=par;
 }
arr[0]=num;
}

void heap_sort()
{
 int last;
 for(last=n-1;last>0;last--)
  del_root(last);
}

void del_root(int last)
{
 int left,right,i,temp;
 i=0;
 temp=arr[i];
 arr[i]=arr[last];
 arr[last]=temp;
 left=2*i+1;
```

```
 right=2*i+2;
 while(right<last)
 {
  if(arr[i]>=arr[left]&&arr[i]>=arr[right])
   return;
  if( arr[right]<=arr[left] )
  {
   temp=arr[i];
   arr[i]=arr[left];
   arr[left]=temp;
   i=left;
  }
  else
  {
    temp=arr[i];
    arr[i]=arr[right];
    arr[right]=temp;
    i=right;
  }
  left=2*i+1;
  right=2*i+2;
 }
 if(left==last-1&&arr[i]<arr[left])
 {
   temp=arr[i];
   arr[i]=arr[left];
   arr[left]=temp;
 }
}
```

/*Program of sorting using radix sort*/

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

struct node
{
 int info ;
 struct node *link;
}*start=NULL;

int main()
{
 struct node *tmp,*q;
 int i,n,item;
 printf("Enter the number of elements in the list :
");
 scanf("%d", &n);
 for(i=0;i<n;i++)
 {
  printf("Enter element %d : ",i+1);
  scanf("%d",&item);
  tmp=malloc(sizeof(struct node));
  tmp->info=item;
  tmp->link=NULL;
  if(start==NULL)
    start=tmp;
  else
  {
   q=start;
   while(q->link!=NULL)
   q=q->link;
   q->link=tmp;
  }
 }
 printf("Unsorted list is :\n");
 display();
 radix_sort();
 printf("Sorted list is :\n");
 display ();
 return 0;
}

void display()
{
 struct node *p=start;
 while( p !=NULL)
 {
  printf("%d ", p->info);
  p=p->link;
 }
 printf("\n");
}

void radix_sort()
{
int i,k,dig,maxdig,mindig,least_sig,most_sig;
 struct node *p, *rear[10], *front[10];
 least_sig=1;
 most_sig=large_dig(start);
 for(k=least_sig;k<=most_sig;k++)
 {
  printf("PASS %d : Examining %dth digit from
right   ",k,k);
  for(i=0;i<=9;i++)
  {
   rear[i]=NULL;
   front[i] = NULL ;
  }
  maxdig=0;
  mindig=9;
  p=start ;
  while(p!=NULL)
  {
   dig=digit(p->info,k);
   if(dig>maxdig)
     maxdig=dig;
   if(dig<mindig)
     mindig=dig;
   if(front[dig]==NULL)
    front[dig]=p;
   else
    rear[dig]->link=p ;
  rear[dig] = p ;
```

```
 p=p->link;
 }
 printf("mindig=%d ",mindig);
 printf("maxdig=%d\n",maxdig);
 start=front[mindig];
 for(i=mindig;i<maxdig;i++)
 {
   if(rear[i+1]!=NULL)
    rear[i]->link=front[i+1];
   else
    rear[i+1]=rear[i];
 }
 rear[maxdig]->link=NULL;
 printf("New list : ");
 display();
 }
}
int large_dig()
{
 struct node *p=start ;
 int large = 0,ndig = 0 ;
 while(p != NULL)
 {
  if(p->info>large)
    large=p->info;
   p=p->link ;
 }
 printf("Largest Element is %d , ",large);
 while(large!=0)
 {
  ndig++;
  large=large/10 ;
 }
 printf("Number of digits in it are %d\n",ndig);
 return(ndig);
}
int digit(int number, int k)
{
 int digit, i ;
 for(i=1;i<=k;i++)
 {
  digit=number%10 ;
  number = number /10 ;
 }
 return(digit);
}
```

```
/* Program for BFS (Breadth-First -Search)*/
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

#define TRUE 1
#define FALSE 0
#define MAX 8

struct node
{
 int data ;
 struct node *next ;
};

int visited[MAX] ;
int q[8] ;
int front, rear ;

void bfs(int,struct node**);
struct node*getnode_write(int);
void addqueue(int);
int deletequeue();
int isempty() ;
void del(struct node*);

void main( )
{
 struct node*arr[MAX];
 struct node*v1,*v2,*v3,*v4;
 int i ;
 clrscr() ;
 v1=getnode_write(2);
 arr[0]=v1 ;
 v1->next=v2=getnode_write(3);
 v2->next=NULL ;
 v1=getnode_write(1);
 arr[1]=v1 ;
 v1->next=v2=getnode_write(4);
 v2->next=v3=getnode_write(5);
 v3->next=NULL;

 v1=getnode_write(1);
 arr[2]=v1 ;
 v1->next=v2=getnode_write(6);
 v2->next=v3=getnode_write(7);
 v3->next=NULL ;

 v1=getnode_write(2);
 arr[3]=v1 ;
 v1->next=v2=getnode_write(8);
 v2->next=NULL ;

 v1=getnode_write(2);
 arr[4]=v1;
 v1->next=v2=getnode_write(8);
 v2->next=NULL ;
```

```
 v1=getnode_write(3);
 arr[5]=v1 ;
 v1->next=v2=getnode_write(8);
 v2->next=NULL;

 v1=getnode_write(3);
 arr[6]=v1 ;
 v1->next=v2=getnode_write(8);
 v2->next=NULL ;

 v1=getnode_write(4);
 arr[7]=v1;
 v1->next=v2=getnode_write(5);
 v2->next=v3=getnode_write(6);
 v3->next=v4=getnode_write(7);
 v4->next=NULL;

 front=rear=-1;
 bfs(1,arr);

 for(i=0;i<MAX;i++)
  del(arr[i]);
 getch() ;
}
void bfs(int v,struct node**p )
{
 struct node *u ;
 visited[v - 1]=TRUE ;
 printf( "%d\t",v);
 addqueue(v);
 while(isempty()==FALSE )
 {
  v=deletequeue();
  u=*(p+v-1);
  while(u!=NULL)
  {
   if(visited[u->data-1]==FALSE )
   {
    addqueue(u->data);
    visited[u->data-1]=TRUE;
    printf("%d ",u->data);
   }
   u=u->next;
  }
 }
}

struct node*getnode_write(int val)
{
 struct node*newnode;
 newnode=(struct node*)malloc(sizeof(struct
node));
 newnode->data=val;
 return newnode ;
}
void addqueue(int vertex)
{
 if(rear==MAX-1)
 {
  printf("\nQueue Overflow.");
  return;
 }
 rear++ ;
 q[rear]=vertex;
 if(front==-1 )
  front = 0 ;
}
int deletequeue( )
{
 int data ;
 if(front==-1 )
 {
  printf("\nQueue Underflow.");
  return;
 }
 data=q[front];
 if(front==rear)
  front=rear=-1;
 else
  front++ ;
 return data ;
}
int isempty( )
{
 if(front==-1)
  return TRUE ;
 else
 return FALSE ;
}
void del ( struct node *n )
{
 struct node*temp ;
 while(n!=NULL )
 {
  temp=n->next ;
  free(n);
  n=temp;
 }
}
```

```
/* Program for DFS (Depth-First-Search*/
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

#define TRUE 1
#define FALSE 0
#define MAX 8

struct node
{
 int data ;
 struct node *next ;
};

int visited[MAX] ;

void dfs(int,struct node**);
struct node*getnode_write(int);
void del(struct node*);

void main( )
{
    struct node*arr[MAX] ;
    struct node *v1,*v2,*v3,*v4;
    int i ;
    clrscr() ;
    v1=getnode_write(2);
    arr[0]=v1;
    v1->next=v2=getnode_write(3);
    v2->next=NULL;

    v1=getnode_write(1);
    arr[1]=v1 ;
    v1->next=v2=getnode_write(4);
    v2->next=v3=getnode_write(5);
    v3->next=NULL ;

    v1=getnode_write(1);
    arr[2]=v1;
    v1->next=v2=getnode_write(6);
    v2->next=v3=getnode_write(7);
    v3->next=NULL ;

    v1=getnode_write(2);
    arr[3]=v1;
    v1->next=v2=getnode_write(8);
    v2->next=NULL;

    v1=getnode_write(2);
    arr[4]=v1;
    v1->next=v2=getnode_write(8);
    v2->next=NULL;

    v1=getnode_write(3);
    arr[5]=v1;
    v1->next=v2=getnode_write(8);
    v2->next=NULL ;

    v1=getnode_write(3);
    arr[6]=v1 ;
    v1->next=v2=getnode_write(8);
    v2->next=NULL;

    v1=getnode_write(4) ;
    arr[7]=v1;
    v1->next=v2=getnode_write(5);
    v2->next=v3=getnode_write(6);
    v3->next=v4=getnode_write(7);
    v4->next=NULL ;

    dfs(1,arr);
    for(i=0;i<MAX;i++)
     del(arr[i]);
    getch();
}

void dfs(int v,struct node**p)
{
  struct node*q;
  visited[v-1]=TRUE ;
  printf("%d ",v);

  q=*(p+v-1);

  while(q!=NULL )
  {
   if(visited[q->data-1]==FALSE)
   dfs(q->data,p);
   else
   q=q->next;
  }
}

struct node*getnode_write(int val)
{
 struct node*newnode ;
 newnode=(struct node*)malloc(sizeof(struct
node));
 newnode->data=val ;
 return newnode ;
}

void del ( struct node *n )
{
 struct node*temp ;
 while (n!=NULL)
 {
  temp=n->next;
  free(n);
  n=temp;
 }
}
```

Type	Best Case	Average Case	Worst Case
Insertion sort	n	n^2	n^2
Shell sort	n	$n\log^2 n$	$O(n\log^2 n)$
Selection sort	n^2	n^2	n^2
Heapsort	$n \log n$	$n \log n$	$n \log n$
Bubble sort	n	n^2	n^2
Merge sort	$n \log n$	$n \log n$	$n \log n$
Quicksort	$n \log n$	$n \log n$	n^2

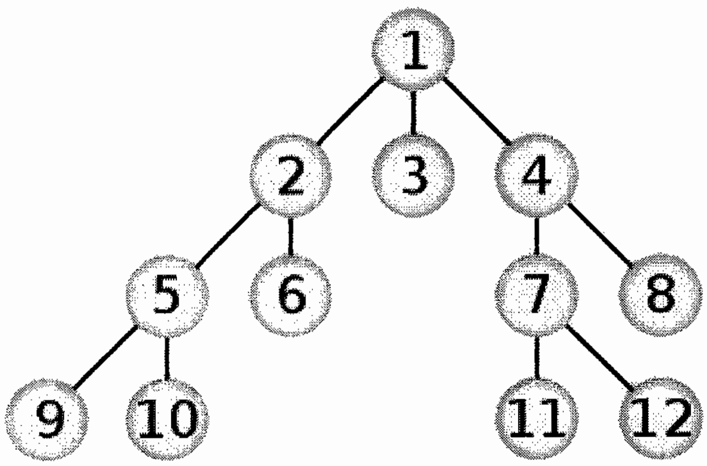
What's the difference between DFS and BFS?

DFS (Depth First Search) and BFS (Breadth First Search) are search algorithms used for graphs and trees.

In a breadth first search, you start at the root node, and then scan each node in the first level starting from the leftmost node, moving towards the right. Then you continue scanning the second level (starting from the left) and the third level, and so on until you've scanned all the nodes, or until you find the actual node that you were searching for. In a BFS, when traversing one level, we need some way of knowing which nodes to traverse once we get to the next level. The way this is done is by storing the pointers to a level's child nodes while searching that level. The pointers are stored in FIFO (First-In-First-Out) queue. This, in turn, means that BFS uses a large amount of memory because we have to store the pointers.

An example of BFS

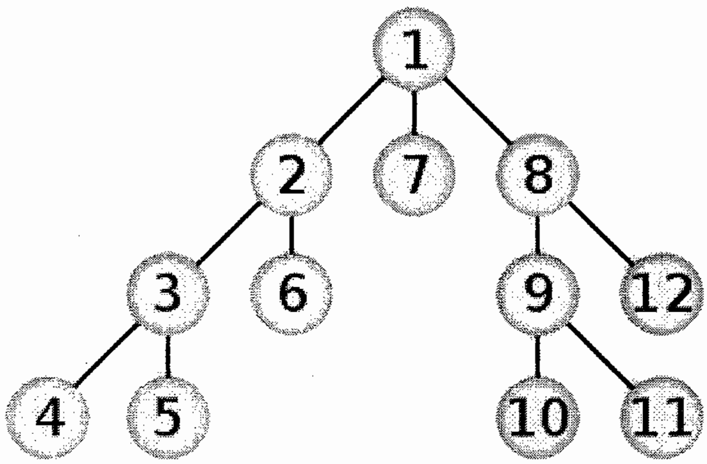
Here's an example of what a BFS would look like. The numbers represent the order in which the nodes are accessed in a BFS:



In a depth first search, you start at the root, and follow one of the branches of the tree as far as possible until either the node you are looking for is found or you hit a leaf node (a node with no children). If you hit a leaf node, then you continue the search at the nearest ancestor with unexplored children.

An example of DFS

Here's an example of what a DFS would look like. The numbers represent the order in which the nodes are accessed in a DFS:



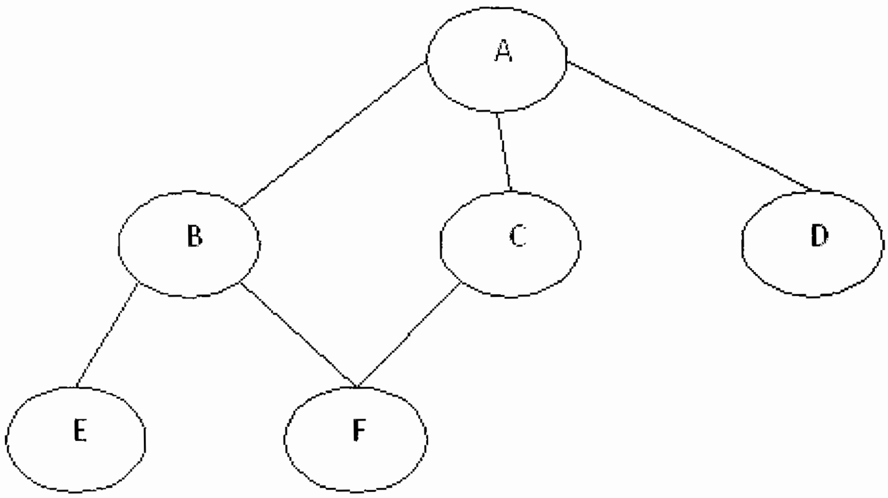
Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it's not necessary to store all of the child pointers at each level.

Sometimes BFS is better than DFS.....Explain it with an example

Depending on the data and what you are looking for, either DFS or BFS could be advantageous. For example, given a family tree if one were looking for someone on the tree who's still alive, then it would be safe to assume that person would be on the bottom of the tree. This means that a BFS would take a very long time to reach that last level. A DFS, however, would find the goal faster. But, if one were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. Then, a BFS would usually be faster than a DFS. So, the advantages of either vary depending on the data and what you're looking for.

Graph Traversal

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.



As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

1. **Step 1**: Push the root node in the Stack.
2. **Step 2**: Loop until stack is empty.
3. **Step 3**: Peek the node of the stack.
4. **Step 4**: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
5. **Step 5**: If the node does not have any unvisited child nodes, pop the node from the stack.

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:

If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step 1**: Push the root node in the Queue.
2. **Step 2**: Loop until the queue is empty.
3. **Step 3**: Remove the node from the Queue.
4. **Step 4**: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Hashing

Hashing is the process to find the index/location in the array to insert/retrieve the data. You take a data item(s) and pass it as a key(s) to a hash function and you would get the index/location where to insert/retrieve the data.

Hashing is the process of mapping large amount of data item to a smaller table with the help of a **hashing function**. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection

Hash Table is the result of storing the hash data structure in a smaller table which incorporates the hash function within itself. The Hash Function primarily is responsible to map between the original data item and the smaller table itself. Here the mapping takes place with the help of an output integer in a consistent range produced when a given data item (any data type) is provided for storage and this output integer range determines the location in the smaller table for the data item. In terms of implementation, the hash table is constructed with the help of an array and the indices of this array are associated to the output integer range.

A Hash Table is nothing but an array (single or multi-dimensional) to store values

Hash Table Example

```
#include<stdio.h>
#include<conio.h>
void main() {
 int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int n, value;
```

```
int temp, hash;
clrscr();
printf("\nEnter the value of n(table size):");
scanf("%d", &n);
do {
 printf("\nEnter the hash value");
 scanf("%d", &value);
 hash = value % n;
 if (a[hash] == 0) {
  a[hash] = value;
  printf("\na[%d]the value %d is stored", hash, value);
 } else {
  for (hash++; hash < n; hash++) {
   if (a[hash] == 0) {
    printf("Space is allocated give other value");
    a[hash] = value;
    printf("\n a[%d]the value %d is stored", hash, value);
    goto menu;
   }
  }

  for (hash = 0; hash < n; hash++) {
   if (a[hash] == 0) {
    printf("Space is allocated give other value");
    a[hash] = value;
    printf("\n a[%d]the value %d is stored", hash, value);
    goto menu;
   }
  }
  printf("\n\nERROR\n");
  printf("\nEnter '0' and press 'Enter key' twice to exit");
 }
 menu:

 printf("\n Do u want enter more");

 scanf("%d", &temp);

}

while (temp == 1);

getch();
}
```