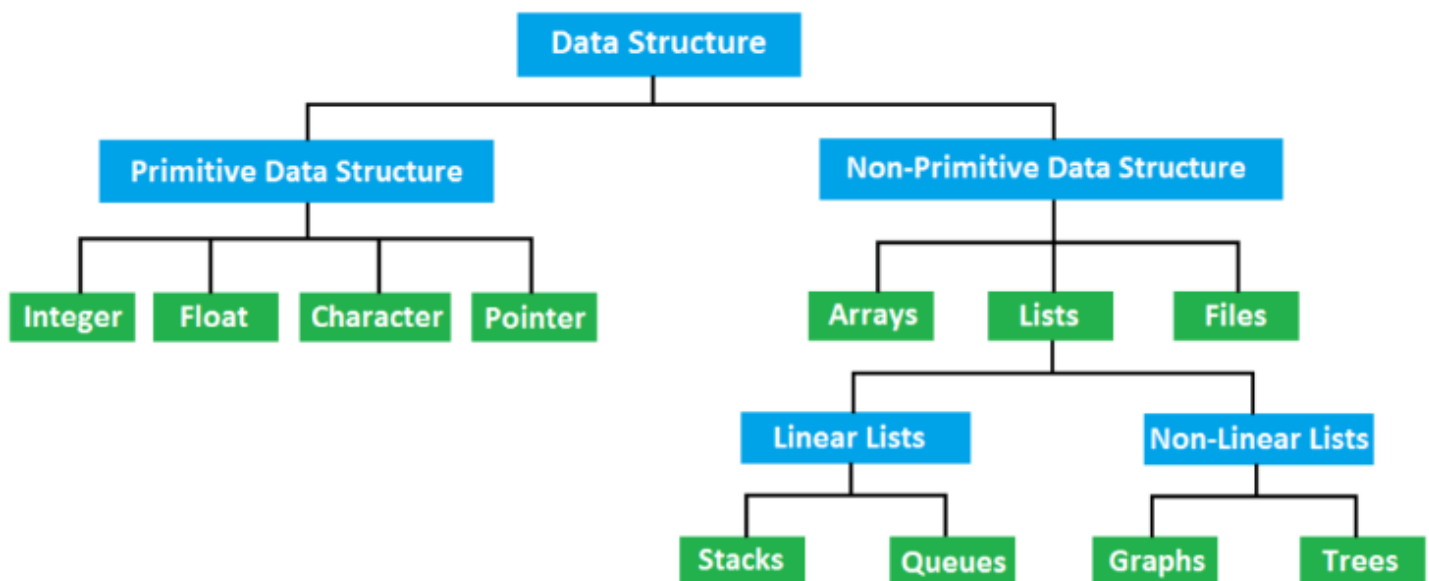


Data Structure Through “C”

Complete Notes

DEEPAK KUMAR SAHU



*This Notes is useful for B.sc
Computer science,
B.Tech,BCA,MCA,
M.Tech , NET,GATE ETC..*

Data Structure

04/04/15

Data structure :-

- ① Any kind of infor^m 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

- ① Compiler Designing. (stack memory allocation can be organized in compiler designing)
- ② Operating system Design. like memory mgt (linked list + Hash-map)
- ③ Database Mgt system (BTree)
- ④ File system representation of O.S. (Trees)
- ⑤ Statistical analysis package (Data mining algorithm)
- ⑥ Network Data model (Graph)
- ⑦ Electronic circuits and simulations (Queues)

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

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

Ex: Array, stack, queue and list.

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 sequential manner.

Ex: Trees, Graphs, Tables

02/01/2015

Memory management in 'C'.

- ① In 'C' p.l. we having two types of memory mgt
① static memory allocation ② Dynamic memory allocation.

② 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 required 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 efficiently then recommended to go for dynamic memory allocation.

Dynamic Memory Allocation:-

- *> It is procedure allocating or deallocating the memory at run time i.e. dynamically.
- *> By using DMA we can utilize the memory efficiently 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() funⁿ.

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 format and initial value is garbage.

*> In implementation when we are creating element memory then recommend 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("%d | n %d - -> %d", currentvalue, count);
        dump[dumpIndex++] = currentvalue;
    }
}
}
}
free(arr);
free(dump);
return EXIT_SUCCESS;
}

```

calloc() :-

*> By using this predefined funⁿ we can create the memory dynamically at initial stage.

*> calloc() funⁿ requires two arguments
count, size-type;

*) ~~count~~ 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, size of type);

Ex:

int* arr;

arr = (int*) calloc (10, ^{size of array.} sizeof(int)); // 20B

free(arr);

realloc() :- By using this predefined funⁿ we can reallocate the memory by 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)); // 20B

arr = (int*) realloc (arr, sizeof(int)*10); // 40B

free(arr);

03/04/2015

Linklists:

- *> In computer's memory a linked list is a linear data structure where all elements are stored in sequence manner.
- *> In linked list every record or element holds a pointer which maintains 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 maintains previous or next record inform.
- *> In complete linked list first element is called Head and last node is called Tail.
- *> Head always maintains first node inform and Tail will maintain terminal node inform.
- *> In 'C' P.L. linked list are classified into 4 types -
 - (a) Single linked list
 - (b) Double linked list
 - (c) single circular linked list
 - (d) double circular linked list

→ `count` indicates no. of elements and
`size_type` is datatype `size`.

*) caller's function will create the memory in blocks format and initial value is '0'.

SYNTAX: `void *calloc (count, size of type);`

```
Ex 3  
int *arr;  
arr = (int *) calloc (10, sizeof(int)); // 20B  
free(arr);
```

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

*> realloc() can takes two arguments
i.e void*, size-type.

* \rightarrow $word_size$ indicates previous block address.
size - type is data type size.

SYNTAX:- void *realloc (void *, size_t new);

```

Ex?
    int *arr;
    arr = (int *)calloc (5, sizeof(int)); // 17208
    - - -
    arr = (int *)realloc (arr, sizeof(int)*10); // 201
    - - -
    free (arr);

```

03/04/2015

Linklists:

- *> In computer's memory a linked list is a linear data structure where all elements are stored in sequence manner.
- *> In linked list every record or element holds a pointer which maintains 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 maintains previous or next record inform.
- *> In complete linked list first element is called Head and last node is called Tail.
- *> Head always maintains 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 list:-

- ① Linked list is a dynamic DS so it can grow or shrink at run time.
- ② On linked list insertion and deletion ^{operation} can perform easily.
- ③ Efficient 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 overhead 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 access elements randomly we need to travel in sequence only.
- ③ When we are working with linked list every element doesn't create 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 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 doesn't allow insertion and deletion operation but linked list are allow insertion and deletion also.

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

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

*> By using struct. we can create user-defined data type which is not available in preprogramming lang.

*> The size of structure is sum of all member 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) \rightarrow 44 B (2+36+4)

struct abc

```
{
    int id;
    char name;
    float f;
};
```

sizeof(struct abc) \rightarrow 73.

① Self-referential structure:

- ② When we are placing structure type pointer as a member to same structure it is called self-referential structure.
- ③ Any type of data structure require to implement using self-referential struct. only.

Ex: 17 struct link.

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

Q-27 struct clink

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

When we are working with structure we require to use following operators we

- 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 *p1;
```

```
p1->id = 101 // e1.id = 101;
```

```
p1->strcpy(p1->name, "Rajesh"); // strcpy(e1.name, "Rajesh")
```

```
p1->sal = 12500; // e1.sal = 12500;
```

```
printf("\nID: %d NAME: %s SAL: %d", e1.id, e1.name, e1.sal);
```

```
printf("\nID: %d NAME: %s SAL: %d", p1->id, p1->name, p1->sal);
```

```
getch();
```

```
return 0;
```

```
}
```


Single Linked List: 5/01/2015

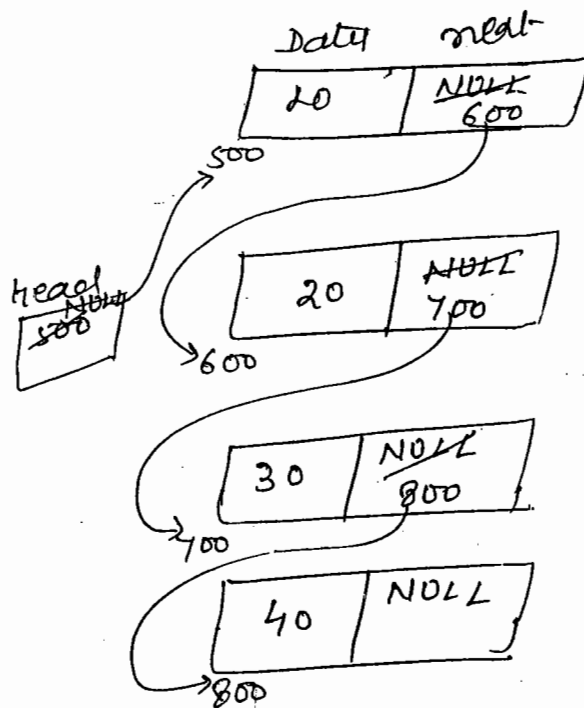
- * In single linked list every node having two field only i.e data and pointer to next node.
- * Data holds informⁿ of the node that can be primitive or user-defined data type.
- * pointer to next node is a address type which maintains next node address.
- * First node of the list is called and it is a dedicated element always points to first node only.
- * When we are working with single linked list last node next pointer is always null.
- * When we are working with SLL always sequential travelling only possible i.e no direct 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 link

{

int data;

struct link * next;

};

Implementation of single linked list:-

*> In single list we can perform following operations -

- ✓ Adding the node.
- ✓ Travelling in linked list
- ✓ Counting no. of nodes.
- ✓ Inserting new node.
- ✓ deletion of a node.
- ✓ updation of a node.
- ✓ 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.

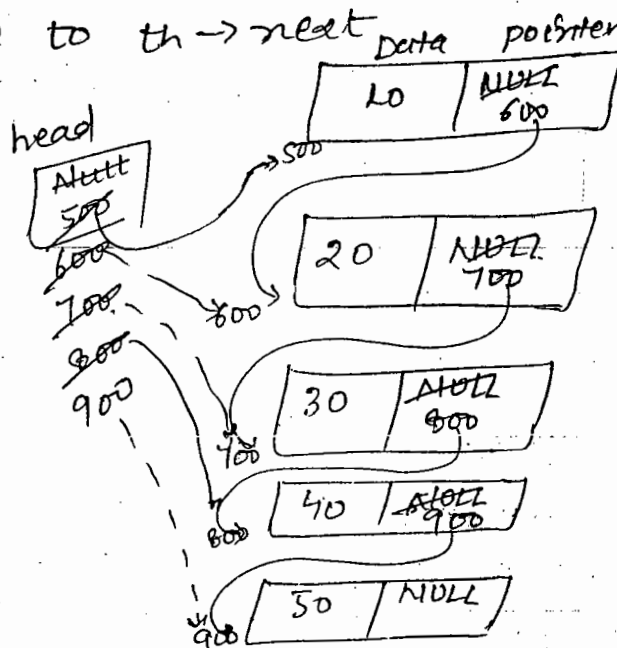
② If head value is other than null then perform following task.

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

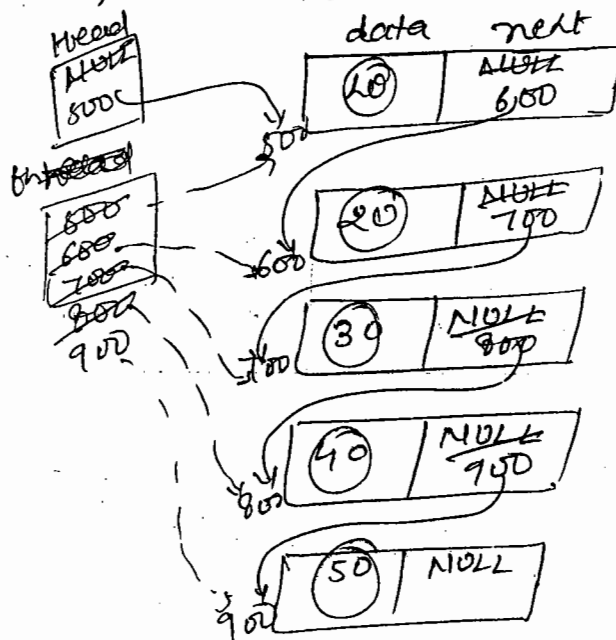
③ create a new node and store the address th → next
th → next

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



Procedure of Traversing linked list:

- 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 ^{current} temporary 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 \rightarrow next.



count

0
1
2
3
4
5

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 singly linked list:

- 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 the value become null, for every repetition increment the count value by 1.

procedure of Insert Node:

* when we are performing insert 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 it.

2) Read the data into it and assign ~~the next~~ $\text{it} \rightarrow \text{next}$ value to current head. i.e.
 $\text{it} \rightarrow \text{next} = \text{head};$

③ Current head Required to shift to newly constructed node.

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

① Take a temporary head TH and assign current head value.

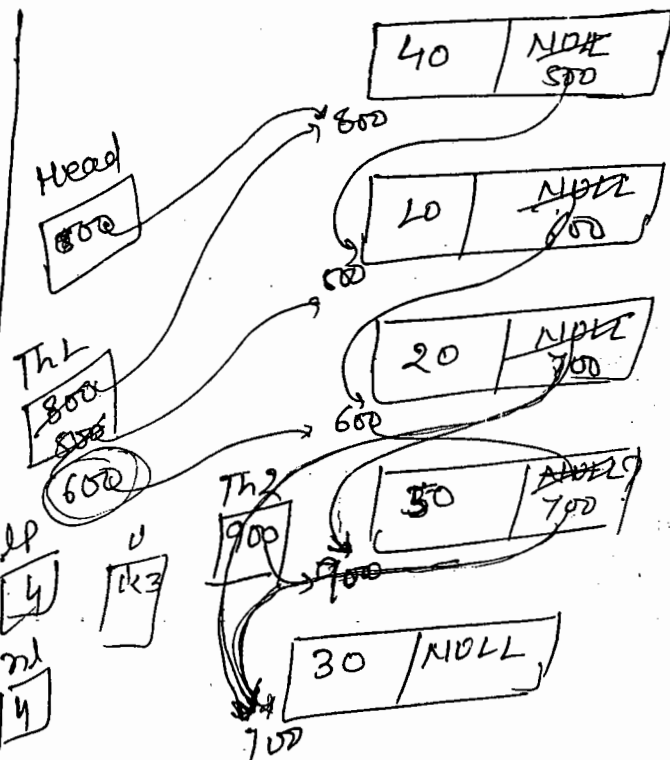
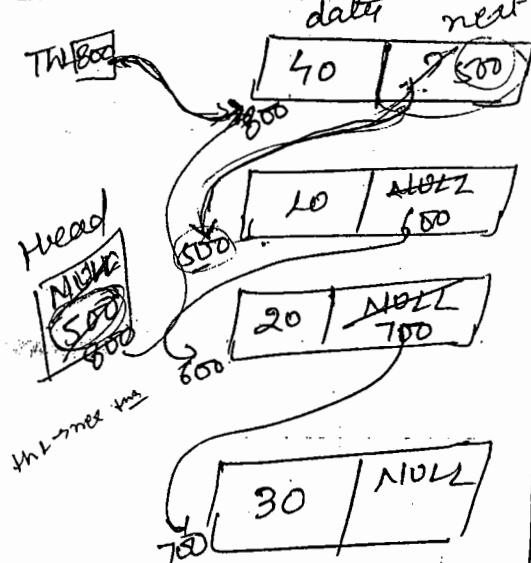
② Design a iterative statement and travel upto link position - 1 - i.e just before actual position

- *) 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).

Case 1: head

Null

Case 2: node position = 1



Procedure to Delete:

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

Case 1: If head status is null list is empty.

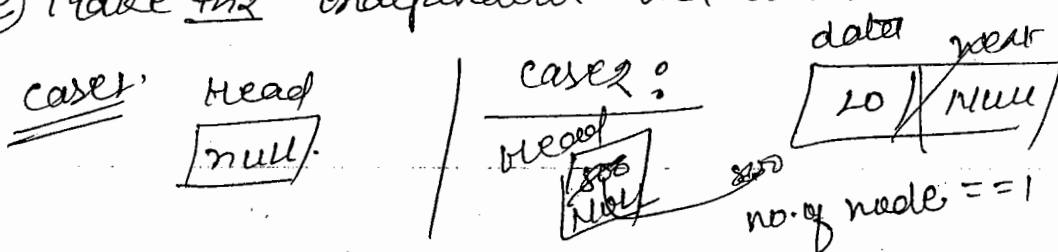
Case 2: If no. of nodes are 1 then delete head and reassign null value.

Case 3: When we are deleting at position 1.

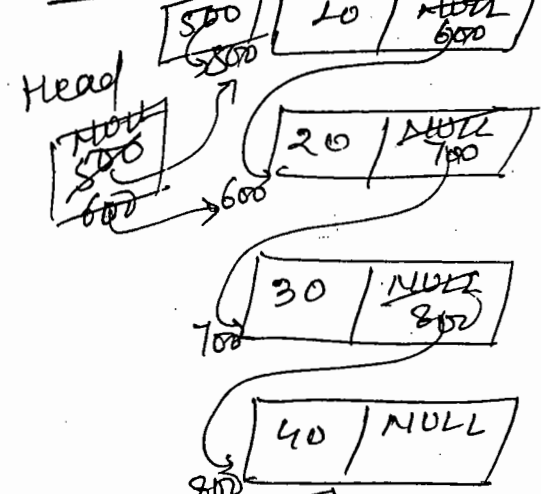
- Take a temporary head th1 and assign current head value.
- Shift one current node head to next node.
i.e. $head = head \rightarrow next;$
- Assign th1 to next value to null and delete th1.

case-4:- Deletion of any position

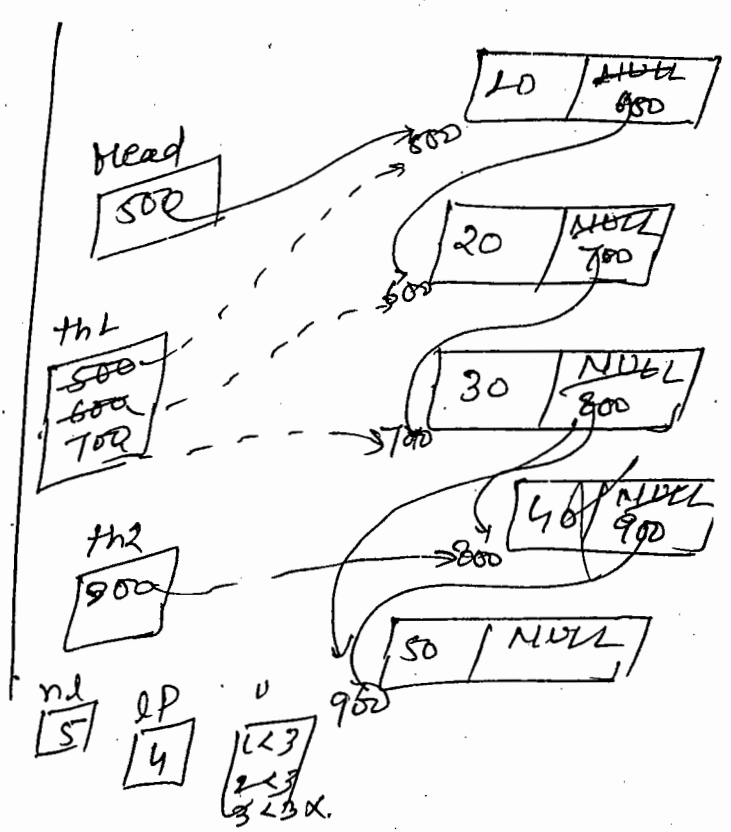
- Take a temporary head th1 and assign the current head value.
- Design a iterative statement and stop at the position - 1 i.e. just before deletion node.
- Take temporary head. i.e. th2 and point to deletion node. i.e. $th2 = th1 \rightarrow next;$
- change previous node to next value to upcoming next node. i.e. $th1 \rightarrow next = th2 \rightarrow next;$ (This statement will makes the relation b/w just before deletion node with next node after deletion.)
- Make th2 independent and delete.



Case 3: th1 data new



no. of positions = L

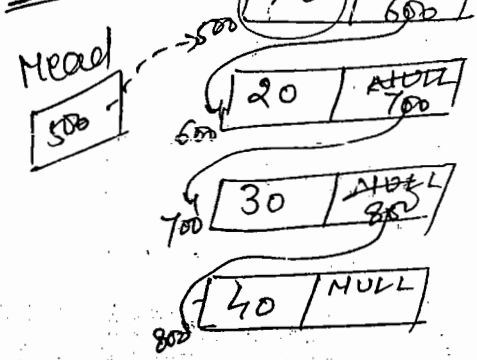


07/01/15: procedure to update node data:

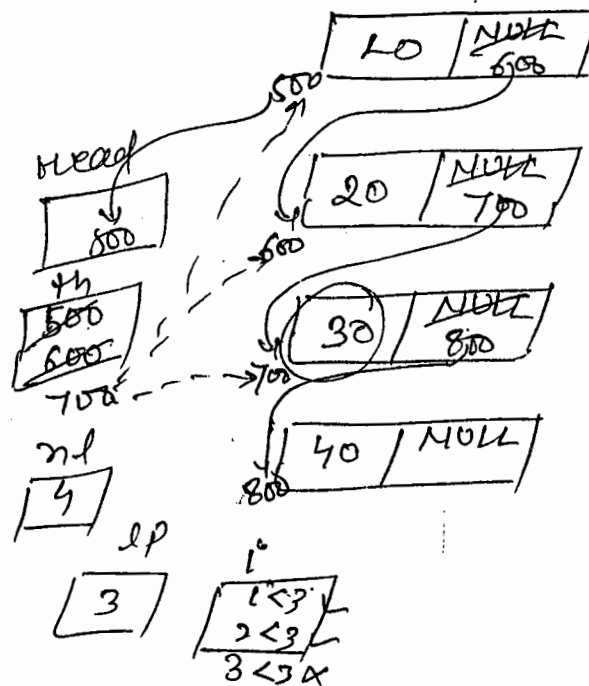
- 1) If the head status is null then it is called empty list.
- 2) If head status is other than null then perform following operations.
 - a) If link position is one then update head data directly
 - b) If link position is other than ~~one~~ one then travel to corresponding link position then update.

Case-1: Head [NULL]

Case-2:



Case 3



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

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

(a) Create 3 temporary ^{pointers} heads with the name, prev, current, and next and assign the prev value to null.

(b) Designing iterative statement until the current value become ^{null} and perform following task.

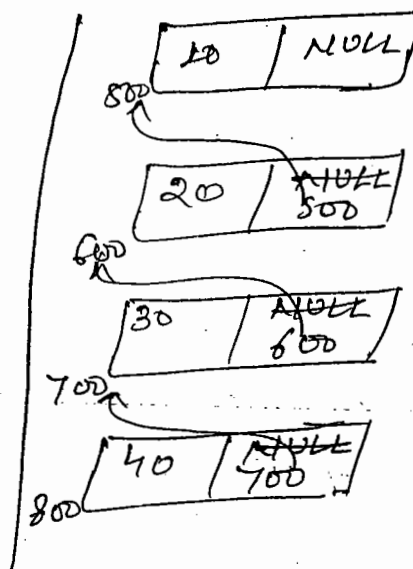
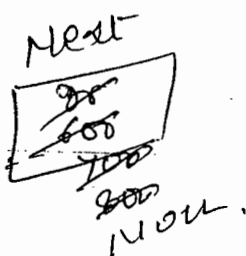
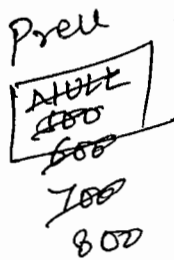
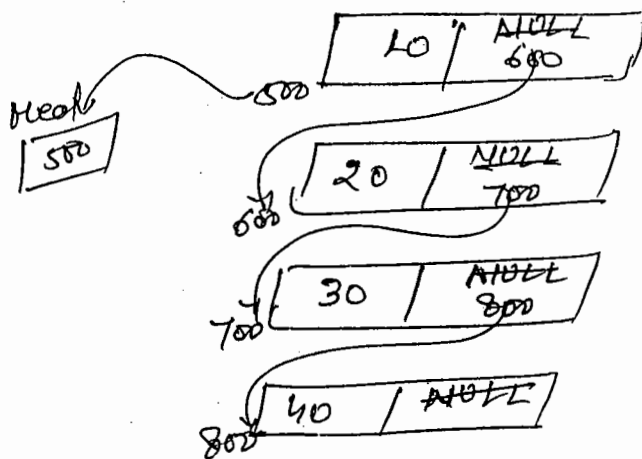
(1) ~~current~~ Assign the current → next value to next pointer.

(2) Assign the previous value to current → next.

(3) Assign the current value to previous.

(4) Assign the ~~next~~ ^{current} value to current.

*) 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 student
{
    int data;
    struct student *next;
};

struct student *head = NULL;

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

```

→ 808 adding.
th → next = (struct student) malloc (sizeof (struct student));

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 student *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 student *th;
```

```
    if (head == NULL)
```

```
    {
        return count;
```

```
    th = head;
```

```
    do
```

```
    {
```

```
        ++count;
```

```
        th = th->next;
```

```
    } while (th != NULL);
```

```
    return count;
```

```
}
```

```
void insertnode()
```

```
{
```

```
    struct student *th1, *th2;
```

```
    int nl, lp, i;
```

```
    if (head == NULL)
```

```
    {
```

```
        printf("List is empty");
```

```
        system("PAUSE");
```

```
        return;
```

```
    }
```

```
    printf("Enter link position: ");
```

```
    scanf("%d", &lp);
```

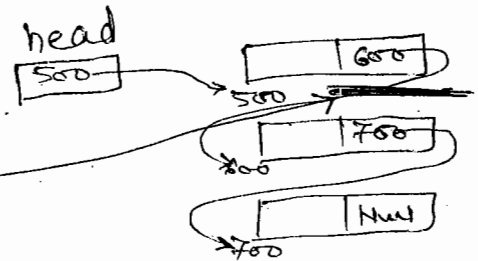
```
    nl = nodecount();
```

```
    if (lp < 1 || lp > nl)
```

```

    }
    printf ("Invalid link position\n");
    system ("PAUSE");
    return;
}
if (lp == 1)
{
    th1 = (struct student*) malloc (sizeof (struct student));
    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 student*) malloc (sizeof (struct student));
printf ("\n Enter data ");
scanf ("%d", &th2->data);
th2->next = th1->next;
th1->next = th2;
return;
}

```



void deletenode ()

{

struct tlink *t1, *t2;

int l, 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 ("Invalid link position");

system ("PAUSE");

return;

}

if (nl == 1)

{

~~tail~~ = head; 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 <= n; i++)
```

```
th1 = th1->next;
```

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

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

```
th2->next = NULL;
```

```
free(th2);
```

```
return;
```

```
}
```

```
void updateNode()
```

```
{
```

```
int n, lp, i;
```

```
struct tlink *th;
```

```
if (head == NULL)
```

```
{
```

```
printf("list is empty: ");
```

```
system("PAUSE");
```

```
return;
```

```
}
```

```
printf("Enter link position: ");
```

```
scanf("%d", &lp);
```

```
n = nodecount();
```



```
if (lp < 1 || lp > n)
```

```
{  
    printf ("Invalid link position |n");  
    system ("PAUSE");  
    return;  
}
```

```
if (lp == 1)
```

```
{  
    printf ("Enter data: ");  
    scanf ("%d", &head->data);  
    return;  
}
```

```
th = head;
```

```
for (i = 1; i < lp; i++)
```

```
th = th->next;
```

```
printf ("Enter new data: ");  
scanf ("%d", &th->data);  
return;  
}
```

```
void reverse() {
```

```
{
```

```
    struct stlink * prev = NULL;
```

```
    struct stlink * current = head;
```

```
    struct stlink * 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 ("1 for ADD NODE: ");
```

```
printf ("2 for DISPLAY: ");
```

```
printf ("3 for NODE COUNT: ");
```

```
printf ("4 for INSERT NODE COUNT: ");
```

```
printf ("5 for DELETE NODE: ");
```

```
printf ("6 for UPDATE NODE: ");
```

```
printf ("7 for Reverse linked list: ");
```

```
printf ("8 for Exit: ");
```

```

scanf ("%d", &option);
{
switch (option)
{
case 1: addnode ();
        break;
case 2: displaynode ();
        break;
case 3: printf ("MODE count=%d\n",
                modecount());
        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
{ // menu

```

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 and next address field maintain next - node inform in linked list.
- *> When we are working with DLL 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 we sequential travelling only.
- *> When we are working with D.L.L also linear relationship will form. but it is bi-direction.

STRUCTURE OF DOUBLE linked LIST:

struct dlink

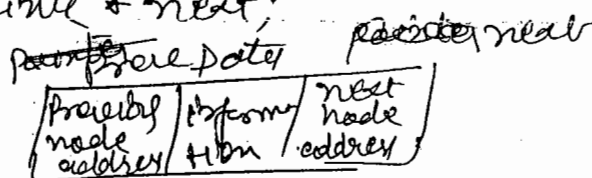
{

struct dlink *prev;

int data;

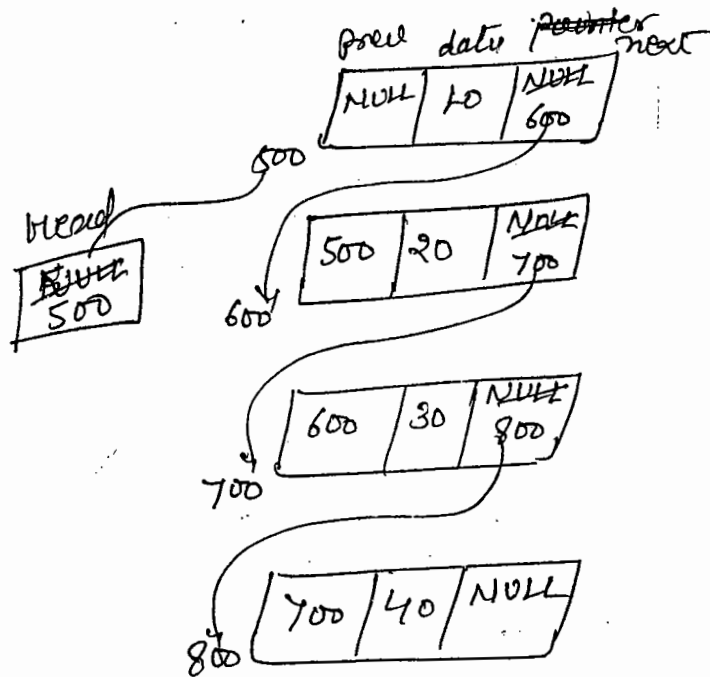
struct dlink *next;

};



Logical Representation of D.Links-

08/01/15 -



Source code:

```

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

typedef struct
{
    int id;
    char name[36];
    int sal;
} EMP; // Just alias name

EMP gebdata();
{
    EMP te;
    printf("Enter ID:");

```

```

scanf ("%d", &te.id);
printf ("|n Enter name: ");
scanf ("%d", &te.id);
flush(stdin);
gets (te.name);
printf ("|n Enter salary: ");
scanf ("%d", &te.sal);
return te;
}

```

```

Emp update data t;
{

```

```

    Emp te;
    printf ("|n Enter new ID: ");
    scanf ("%d", &te.id);
    printf ("|n Enter new name: ");
    flush(stdin);
    gets (te.name);
    printf ("|n Enter new salary: ");
    scanf ("%d", &te.sal);
    return te;
}

```

```

void show data (Emp te)
{

```

```

    printf ("|n ID: %d Name: %s sal: %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 ("Do you want to continue?");

flush (stdin);

ch = getch();

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

```
th1 = th2;
```

```
printf("\n Do you want to continue Y?");
```

```
flush(stdin);
```

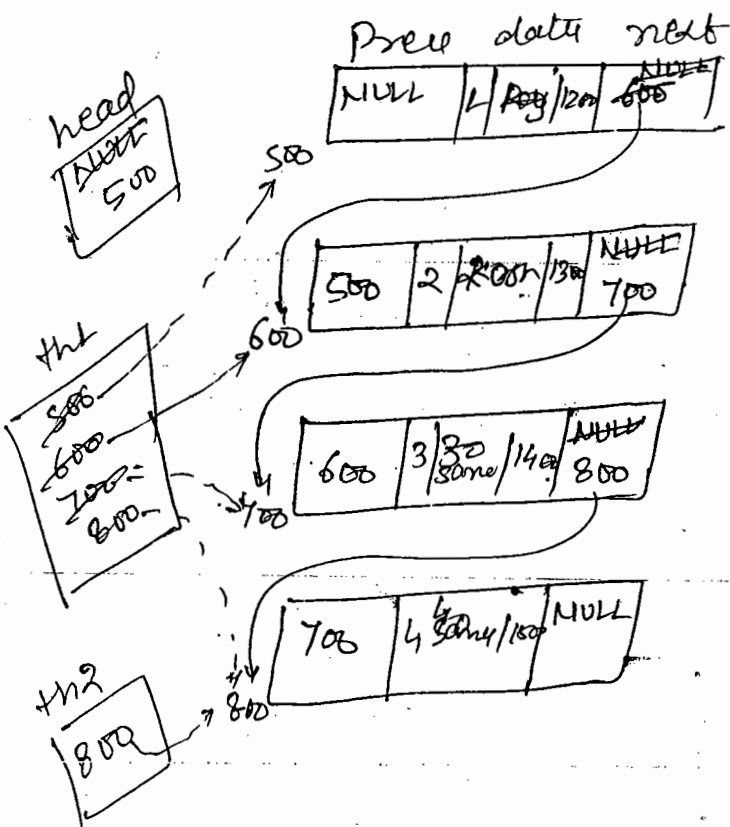
```
ch = getchar();
```

```
} while (ch != 'Y' || ch == '\0');
```

```
th1 = th2 = NULL;
```

```
return;
```

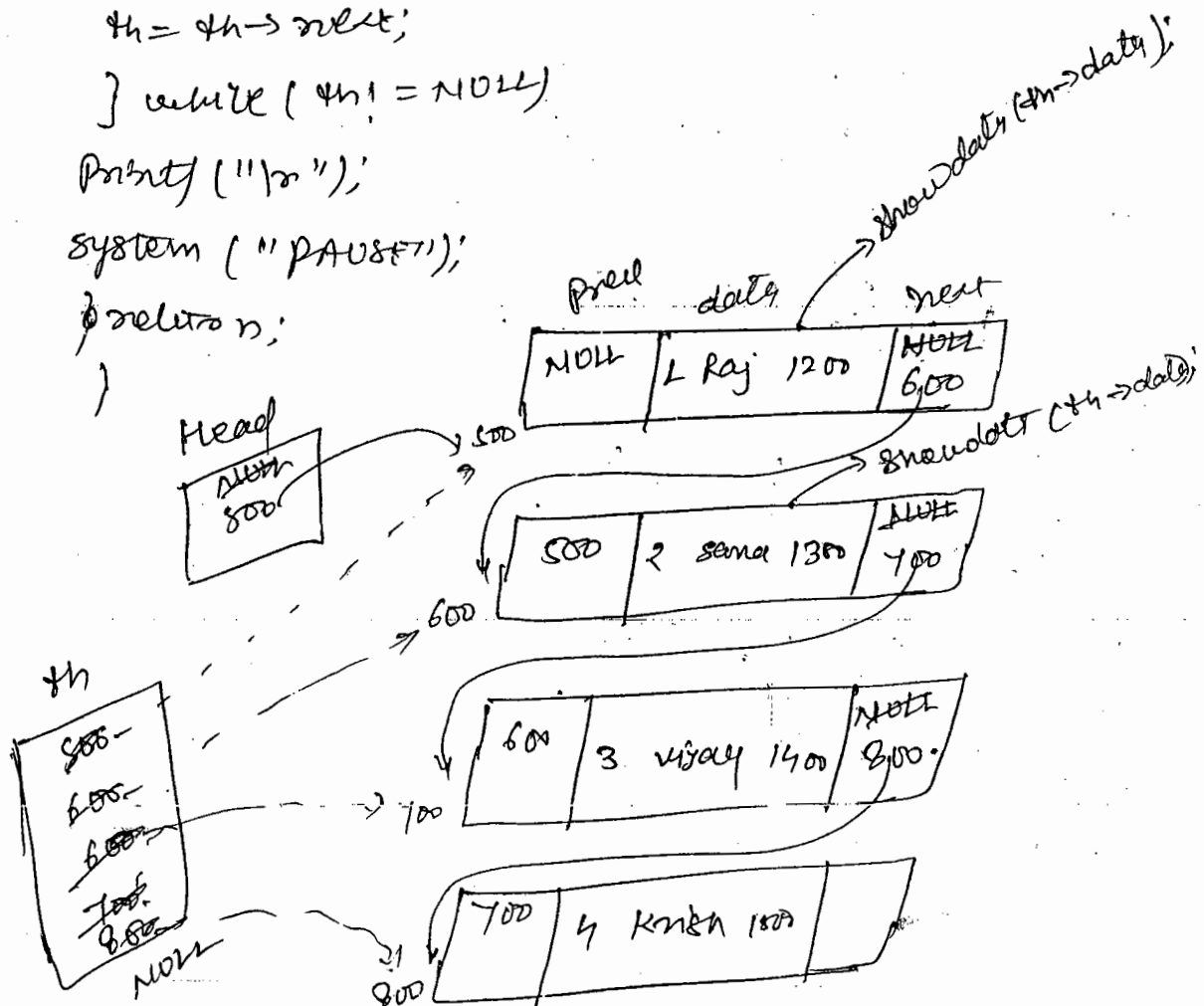
```
}
```




```

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 (help == 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;
```

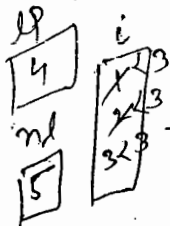
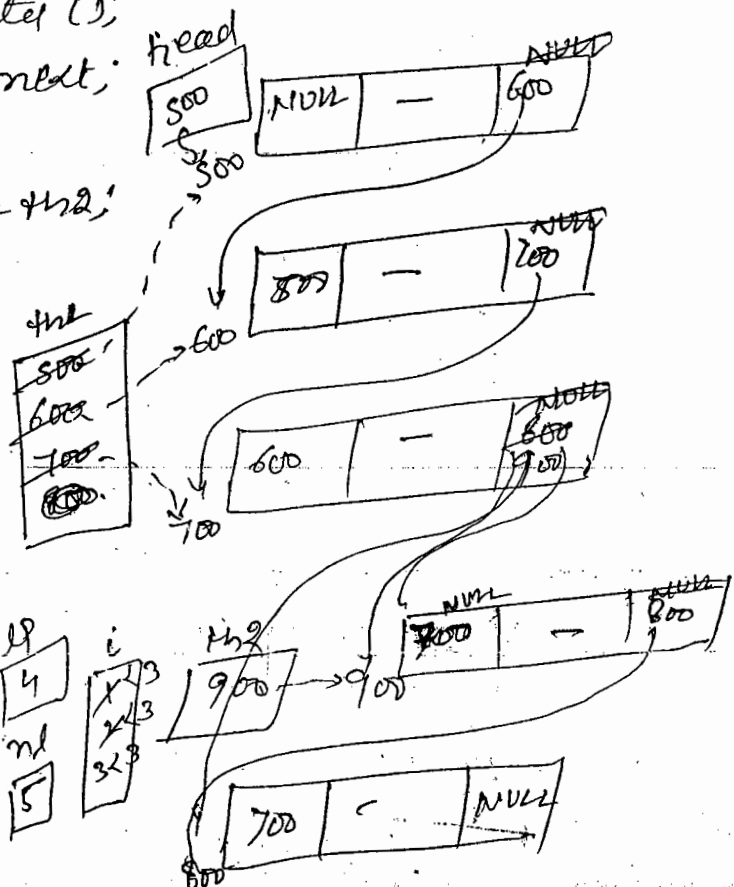
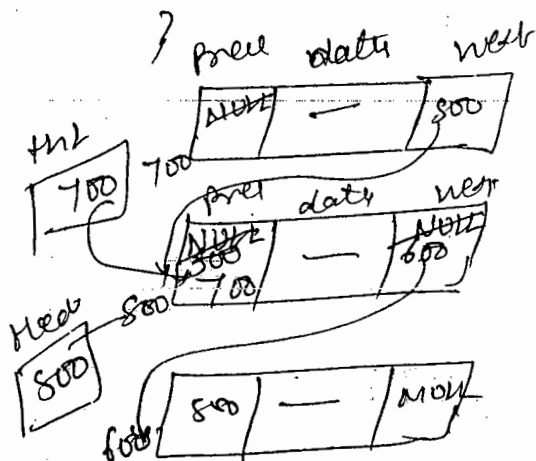
```
if (i != lp)
```

```
th2->next->prev = th2;
```

```
th1 = th2 = NULL;
```

```
return 0;
```

```
} prev data next
```



```

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 (th1->next th1 != lp)
```

```
th2->next->prev = th1; // th2->next->prev = th2->prev
```

```
th2->prev = NULL;
```

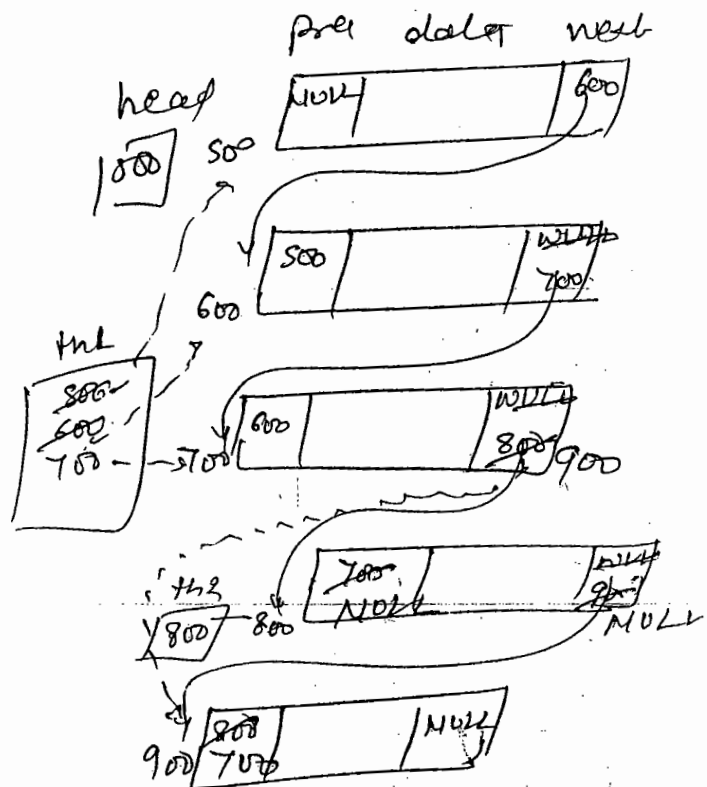
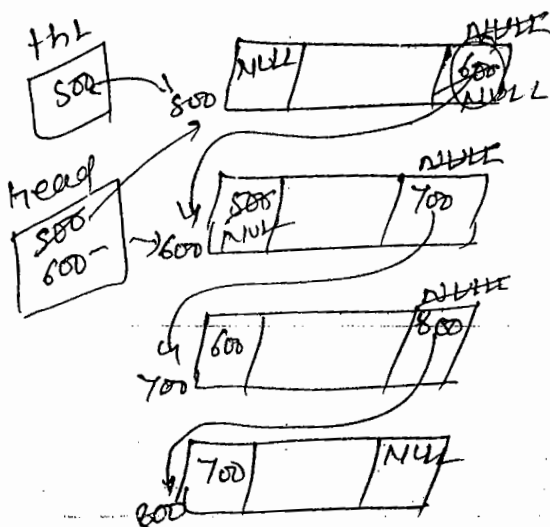
```
th2->next = NULL;
```

```
free (th2);
```

```
th2 = th2 = NULL;
```

```
return;
```

```
}
```



10/01/2015

```
void updateNode()
{
    LINK *thL;
    int nL, lP, i;
    if (head == NULL)
    {
        Print ("list is empty");
        system ("PAUSE");
        return;
    }
    Print ("Enter the update position");
    scanf ("%d", &lP);
    n = nodecount();
    if (lP < 1 || lP > nL)
    {
        Print ("Invalid link position");
        system ("PAUSE");
        return;
    }
    if (lP == 1)
    {
        head → data = updateData();
        Print ("node is updated");
        system ("PAUSE");
        return;
    }
    thL = head;
    for (i = 1; 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 = needcount();

```

```

    if (nl == 1)
    {

```

```

        Printf ("reverse cant be possible");

```

```

        system ("PAUSE");

```

```

        return;
    }

```

```

    while (current != NULL)
    {

```

```

        temp = current->prev;

```

```

        current->prev = current->next;

```

```

        current->next = temp;

```

```

        current = current->prev;

```

```

    }

```

```

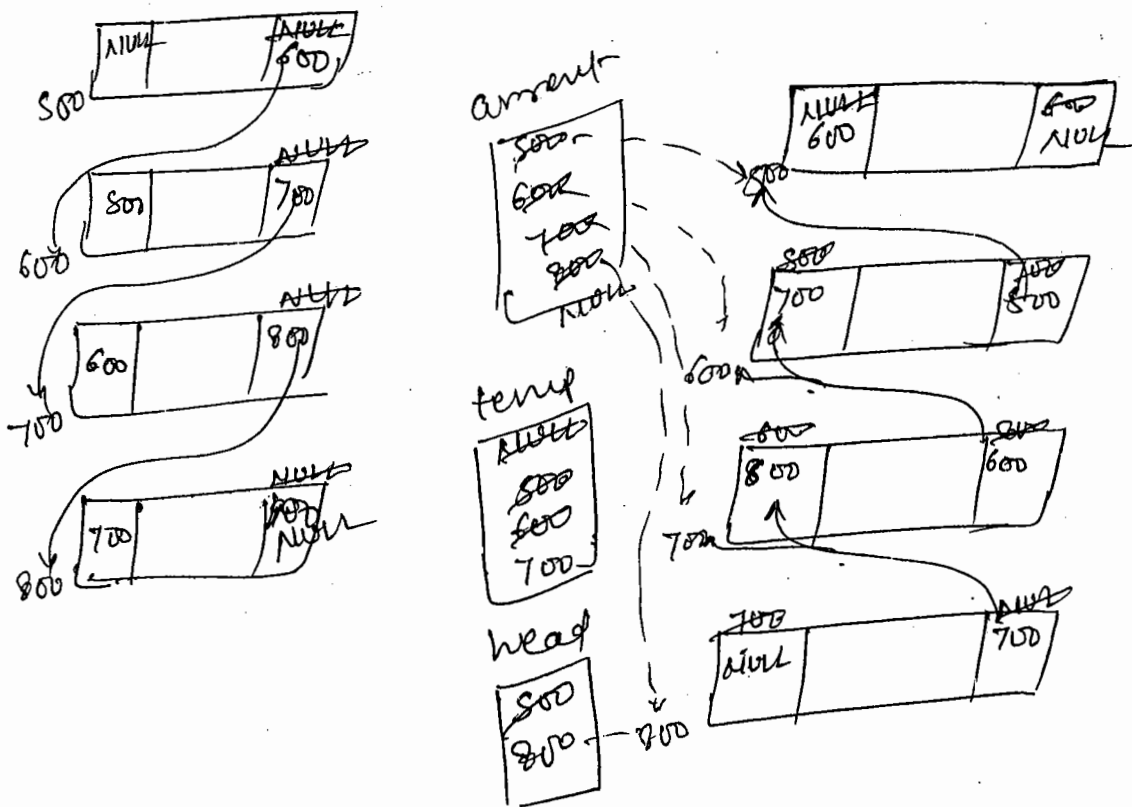
    head = temp->prev;

```

```

    return;
}

```



int main,

{

int option;

while (1)

{ system ("cls");
Print ("1 for ^{ADD} insert node");

Print ("2 for ^{display} delete node");

Print ("3 for current node");

Print ("4 for insert node");

Print ("5 for delete node");

Print ("6 for update node");

Print ("7 for reverse node");

Print ("8 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: delete node();
```

```
break;
```

```
case 6: update node();
```

```
break;
```

```
case 7: reverse node();
```

```
break;
```

```
case 8: free(head)
```

```
return EXIT_SUCCESS;
```

```
default: printf ("invalid option");
```

```
system ("PAUSE");
```

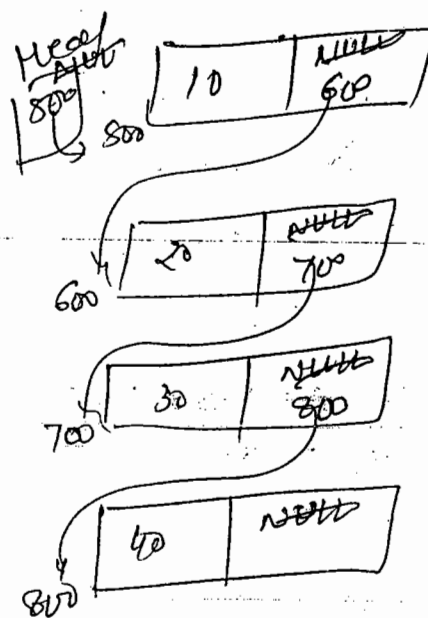
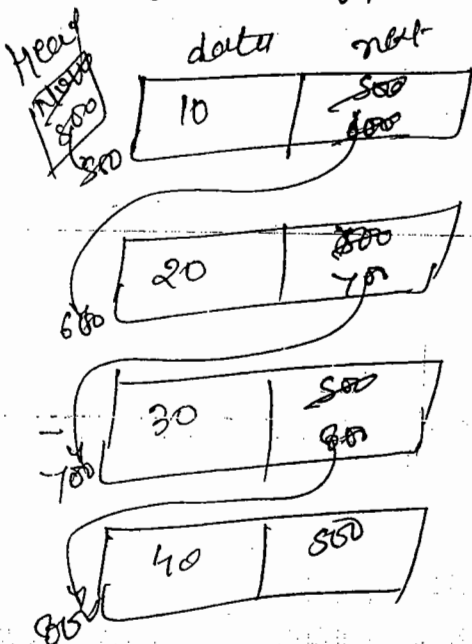
```
break;
```

```
} }
```

Single Circular Linked List:-

- *) when we are working S.C.L.L. every node contains two fields i.e. data and pointer to next node
- *) data is maintained in form of the node and pointer to next is next node ~~data~~ 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 traversal 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. bcz it is terminal node.
 - In S.C.L.L. tail → next is maintain head in form

⇒ Logical Representation of S.C.L.L.:



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

```
struct slink
{
```

```
    int data;
```

```
    struct slink *next;
```

```
};
```

```
typedef struct slink 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/n");
```

```
        fflush (stdin);
```

```
        ch = getchar();
```

```
        if (ch != 'y' && ch != 'Y')
```

```
            return;
```

```
    }
```

```

th = head;
while (th != NULL)
{
    th = th->next;
}
th = head;
do
{
    th->next = (LINK *) malloc(sizeof(LINK));
    th = th->next;
    printf ("Enter the data: ");
    scanf ("%d", &th->data);
    th->next = NULL;
    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 != NULL)
    {
        printf ("Node data: %d\n", th->data);
        th = th->next;
    }
}

```

```
{ while (th != head);
```

```
system("PAUSE");
```

```
return;
```

```
}  
int neelecount()
```

```
{
```

```
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 *th1, *th2;
```

```
int lp, nl, i;
```

```
if (head == NULL)
```

```
{
```

```
printf("list is empty\n");
```

```
system("PAUSE");
```

```
return;
```

```
}
```

```
printf ("Enter the linked position");
```

```
scanf ("%d", &lp);
```

```
nl = nodecount();
```

```
if (lp < 1 || lp >= nl)
```

```
{  
    printf ("Invalid linked position\n");
```

```
    system ("PAUSE");
```

```
    return;
```

```
}
```

```
if (lp == 1)
```

```
{
```

```
    th1 = (LINK*) malloc (sizeof(LINK));
```

```
    printf ("Enter data: ");
```

```
    scanf ("%d", &th1->data);
```

```
    th1->next = head;
```

```
    th2 = head;
```

```
    while (th2->next != head)
```

```
        th2 = th2->next;
```

```
    head = th1;
```

```
    th2->next = head;
```

```
    return;
```

```
}
```

```
th1 = head;
```

```
for (i = 1; i < lp - 1; i++)
```

```
    th1 = th1->next;
```

```
th2 = (LINK*) malloc (sizeof(LINK));
```

```
printf ("Enter the data: ");
```

```
scanf ("%d", &th2->data);
```

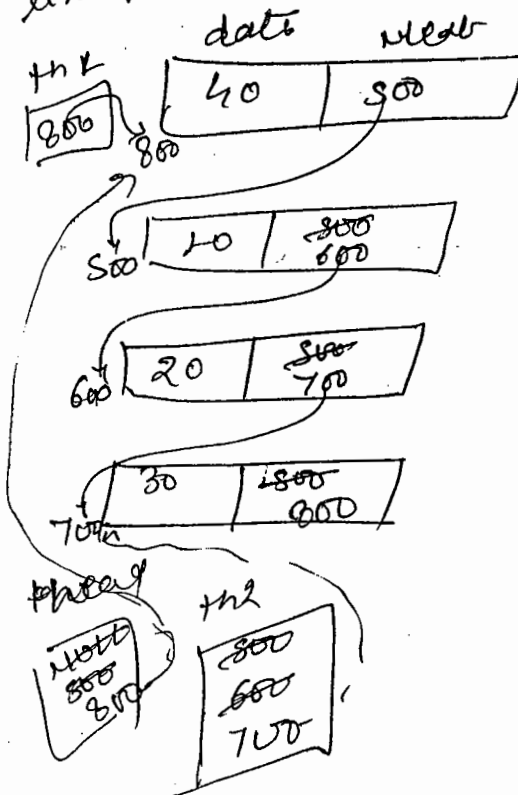
th2 → next = th1 → next

th1 → next = th2;

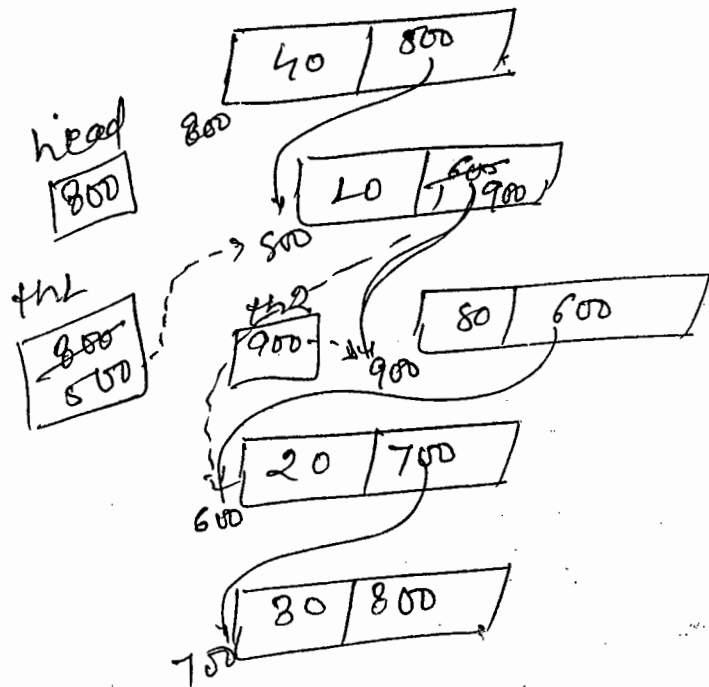
return;

}

link position = 1



Any other position.



void deletedelete()

{

LIST *th1, *th2;

int i, lp, ml;

If (head == NULL)

{

printf ("List is empty\n");

system ("PAUSE");

return;

}

Print ("Enter the link position")

scanf ("%d", &lp);

nl = nodelcount();

if (lp < 1 || lp > nl)

{

Print ("Invalid position:");

system ("PAUSE");

return;

}

if (nl == 1)

{

free (head)

head = NULL;

return;

}

if (lp == 1)

{

th1 = head

~~while~~ head = head->next; // head = th1->next;

th1->next = NULL;

th2 = head;

while (th2->next != th1)

th2 = th2->next;

th2->next = head;

free (th1);

th1 = th2 = NULL;

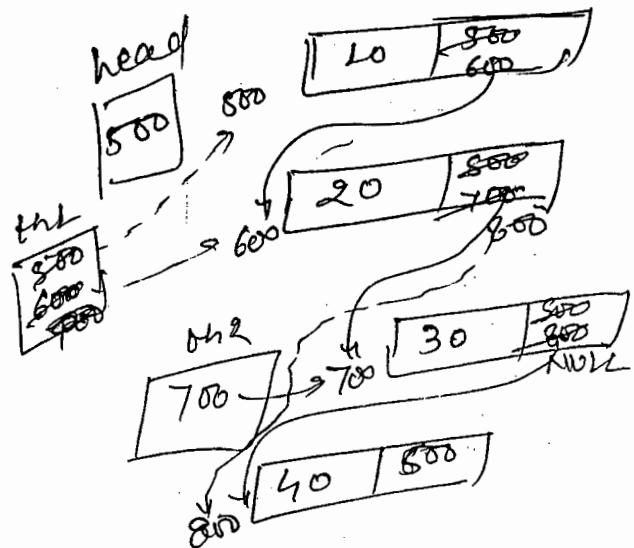
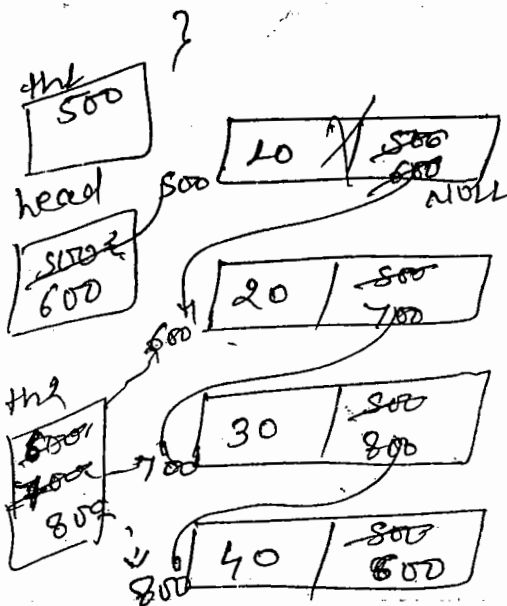
return;

}


```

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()
{
    LINK *th1, *th2;
    int i, lp, nl;
    if (head == NULL)
    {
        printf("List is empty!");
        system("PAUSE");
        return;
    }
}

```

```
Print ("Enter the position: ");
```

```
scanf ("%d", &lp);
```

```
nl = nodelcount();
```

```
If (lp < 1 || lp > nl)
```

```
{
```

```
Print ("Invalid position\n");
```

```
System ("PAUSE");
```

```
return;
```

```
}
```

```
If (lp == 1)
```

```
{
```

```
Print ("Enter the data:");
```

```
scanf ("%d", &head->data);
```

```
return;
```

```
}
```

```
th = head;
```

```
for (i = 1; i < lp; i++)
```

```
th = th->next;
```

```
Print ("Enter the data:");
```

```
scanf ("%d", &th->data);
```

```
return;
```

```
}
```

```
void reverseNode()
```

```
{
```

```
LINK* prev = head;
```

```
LINK* current = head->next;
```

```
LINK* next;
```

```
Ent nl;
```

```
if (head == null)
```

```
{
```

```
    printf ("List is empty");
```

```
    system ("PAUSE");
```

```
    return;
```

```
}
```

```
printf ("Enter
```

```
    nl = nodelcount();
```

```
    if (nl == 1)
```

```
    {
```

```
        printf ("Can't be reverse!");
```

```
        system ("PAUSE");
```

```
        return;
```

```
    }
```

```
    while (current != head)
```

```
    {
```

```
        next = current->next;
```

```
        current->next = prev;
```

```
        prev = current;
```

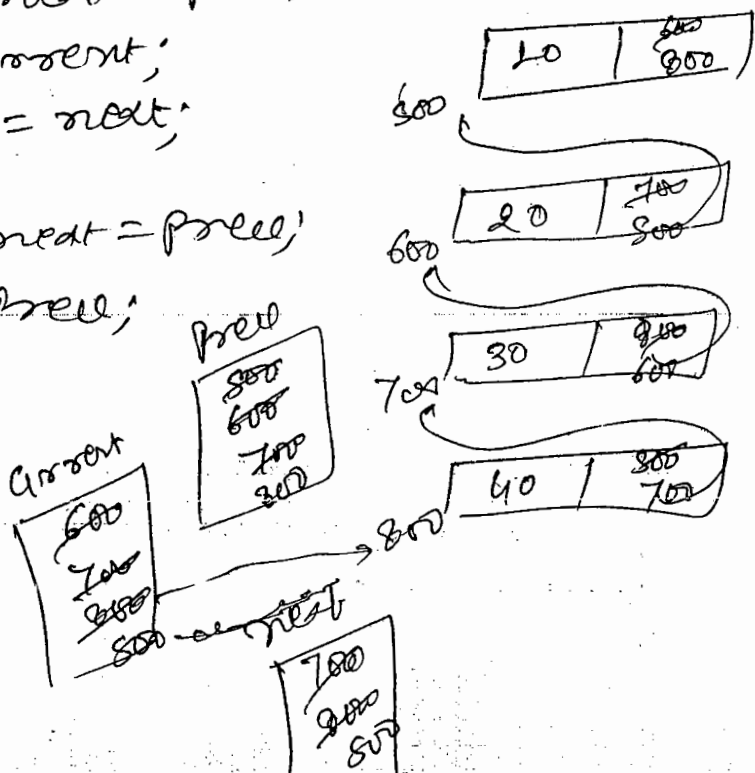
```
        current = next;
```

```
    }
```

```
    head->next = prev;
```

```
    head = prev;
```

```
}
```



14/04/15' 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 holds 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.S.L.L only by 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.S.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 \rightarrow prev maintains tail inform and tail to next maintains head inform in circular format.

Logical Representation of D.C.L.L:



```

#include <stdio.h>
#include <cs.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 delink LINK;
```

```
LINK *head = NULL;
```

```
void addnodes()
```

```
{
```

```
LINK *th1, *th2;
```

```
int ch;
```

```
if (head == NULL)
```

```
{
```

```
head = (LINK *) malloc (sizeof (LINK));
```

```
head->prev = head; head;
```

```
head->data = getdata();
```

```
head->next = head;
```

```
printf ("Do you want to continue? : ");
```

```
flush(stdin);
```

```
ch = getch();
```

```
if (ch != 'y' && ch != 'Y')
```

```
return;
```

```
}
```

```
th1 = head;
```

```
while (th1->next != head)
```

```
th1 = th1->next;
```

```
do
```

```
{ th2 = (LINK *) malloc (sizeof (LINK));
```

```
th2->prev = head; th1;
```

```
th2->data = getdata();
```

```
th2->next = head;
```

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

```
head->prev = th2;
```

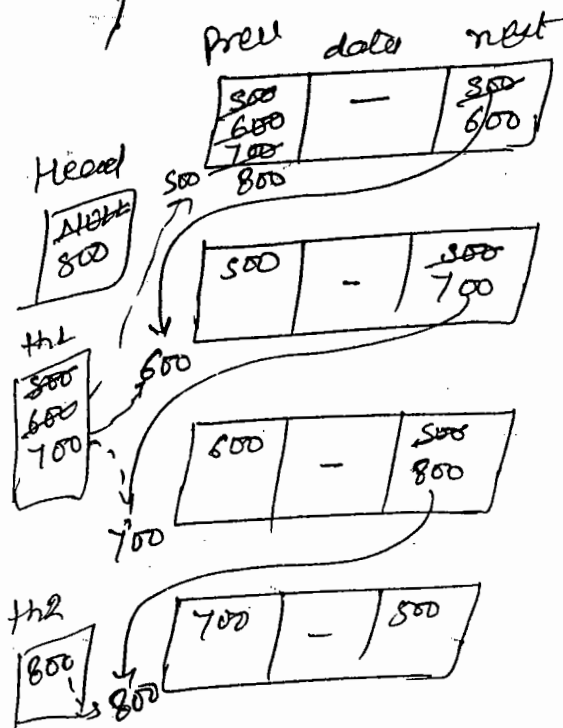
```
th1 = th2;
```

```

Printf ("Do you want to continue? ");
flush (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 (th->data);
th = th->next;
} while (th != head);
printf ("\n");
system ("PAUSE");
return;
}

```

void displaynodeL to FC

```

{
LENK *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;
    LENK *th;
    if (head == NULL)
    {
        return count;
    }
    th = head;
    while do
    {
        ++count;
        th = th->next;
    } while (th != head);
    return count;
}

```

```

void insertnode ()
{
    LENK *th1, *th2;
    int l, 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)

```

```
}
```

```
Print ("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;
```

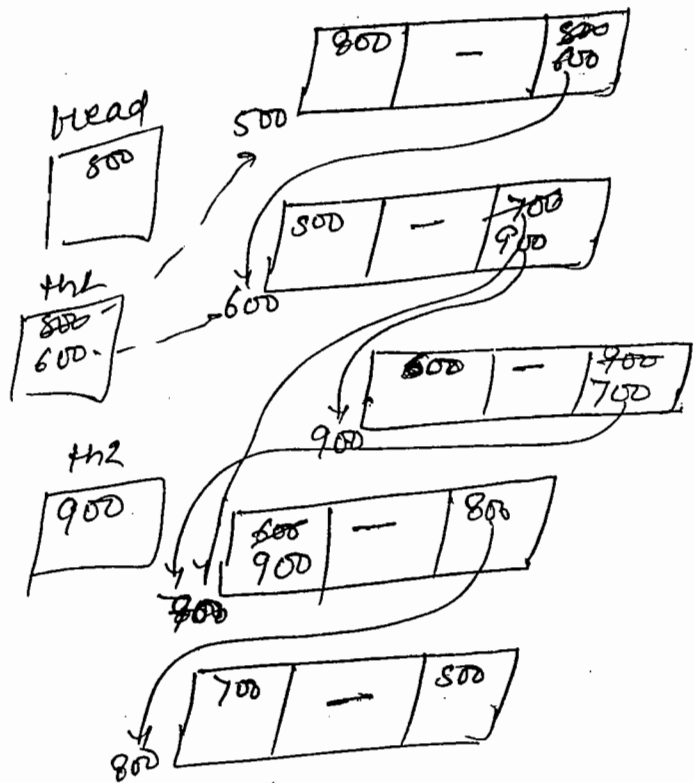
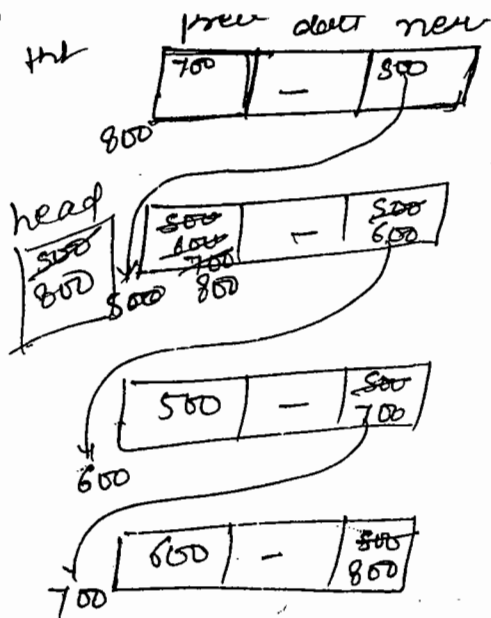
```
th1->next = th2;
```

```
th2->next->prev = th2;
```

```
th1 = th2 = NULL;
```

```
return;
```

```
}
```



void deletenode()

{

LENIK *th1, *th2;

int i, lp, nl;

if (head == NULL)

{

printf ("List is empty!");

system ("PAUSE");

return;

}

printf ("Enter one ^{delete} ~~long~~ position:");

scanf ("%d", &lp);

nl = nodecount();

if (lp < 1 || lp > nl)

{

```
Print ("Invalid delete position");
```

```
system ("PAUSE");
```

```
return;
```

```
}
```

```
if (n1 == 1)
```

```
{
```

```
free (head);
```

```
head = NULL;
```

```
Print ("node is deleted");
```

```
system ("PAUSE");
```

```
return;
```

```
}
```

```
if (lp == 1)
```

```
{
```

```
th1 = head;
```

```
head = head->next;
```

```
head->prev = th1->prev;
```

```
head->prev->next = head;
```

```
th1->next = th1->prev = NULL;
```

```
free (th1);
```

```
th1 = NULL;
```

```
return;
```

```
}
```

```
th1 = head;
```

```
for (i = 0; i < lp - 1; i++)
```

```
th1 = th1->next;
```

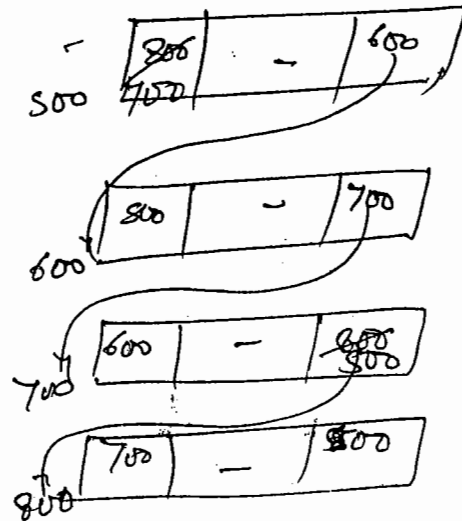
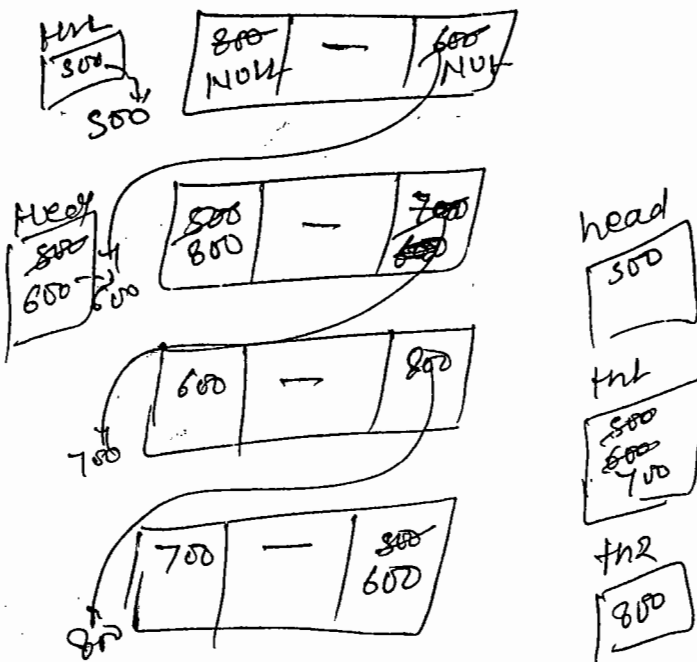
```
th2 = th1->next;
```

```
th1->next = th2->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) PECK.
- *> 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.
- *> PECK 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:-

```

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

#define SIZE 5

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

int main()
{
    int option;
    while (push (word));
    while (pop (word));
    while (peek (word));
    while (1)
    {
        system ("CLS");
        printf ("In Push Press 1--");
        printf ("In Pop Press 2--");
        printf ("In Peek Press 3--");
        printf ("In 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;
```

```
}
```



```
Printf ("Pop element is: %d\n", STACK[top]);
--top;
```

```
system ("PAUSE");
}
```

```
next peek()
```

```
{
```

```
int i;
```

```
if (top == -1)
```

```
{
```

```
Printf ("STACK is empty -- \n");
```

```
system ("PAUSE");
```

```
return;
```

```
}
```

```
for(Printf ("STACK ELEMENT is: "));
```

```
for (i = top; i >= 0; i--)
```

```
Printf ("%d", STACK[i]);
```

```
Printf ("\n");
```

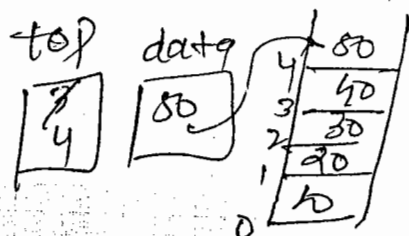
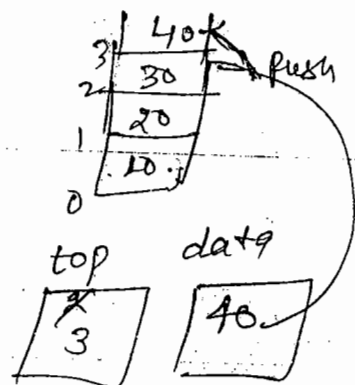
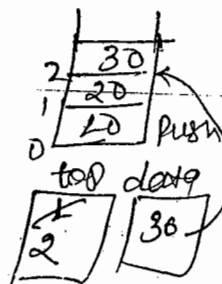
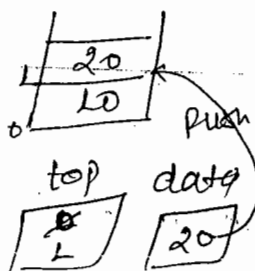
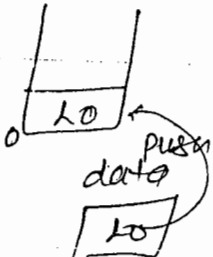
```
system ("PAUSE");
```

```
return;
```

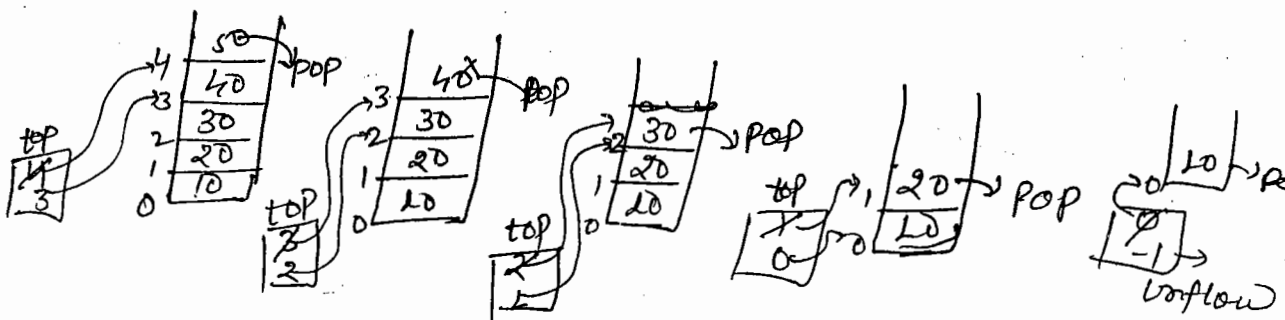
```
}
```

size
5

top
-1



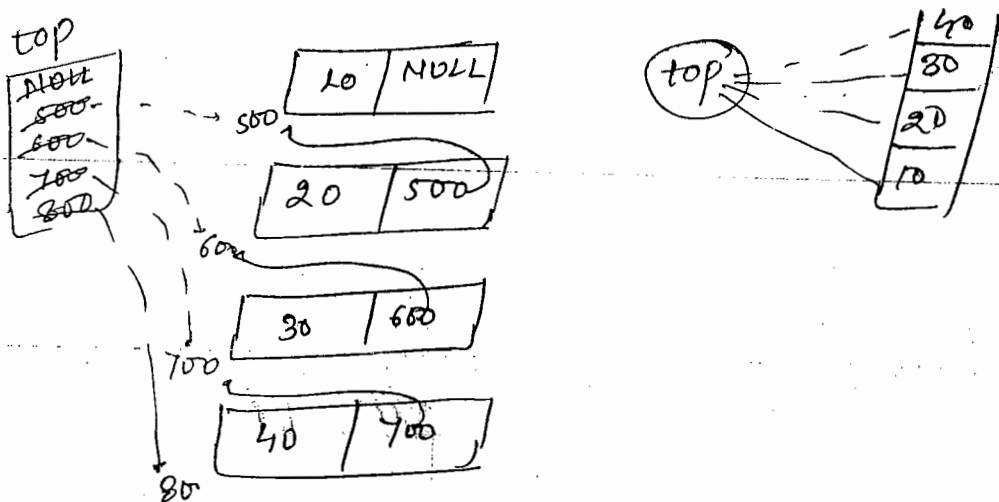
top
4 = size overflow



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 LL, then dedicated pointer is called top. but in general it is equivalent to head node in LL.
- 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 ("Enter the data: ");
```

```
scanf ("%d", &top->data);
```

```
top->next = NULL;
```

```
return;
```

```
}
```

```
temp = (STACK*) malloc (sizeof(STACK));
```

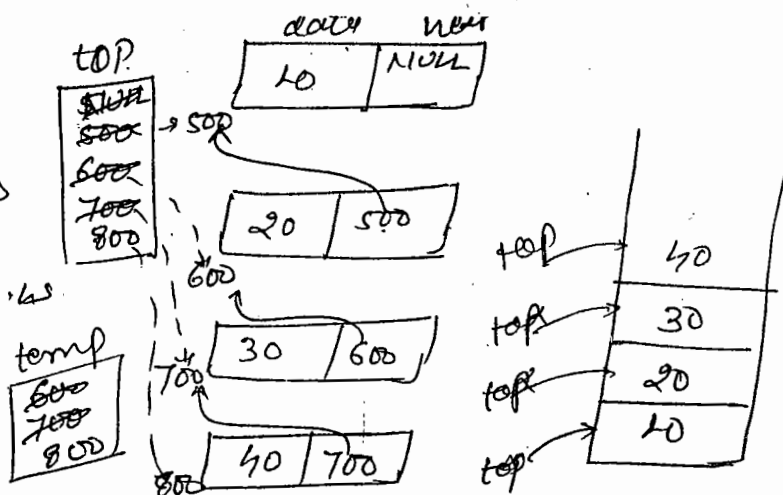
```
printf ("Enter the data: ");
```

```
scanf ("%d", &temp->data);
```

```
temp->next = top;
```

```
top = temp;
```

```
return; }
```



```
void pop()
```

```
{
```

```
stack *temp;
```

```
if (top == NULL)
```

```
{
```

```
printf("Stack is underflow");
```

```
return;
```

```
}
```

```
printf("%d\n", pop element is: %d", top->data);
```

```
temp = top;
```

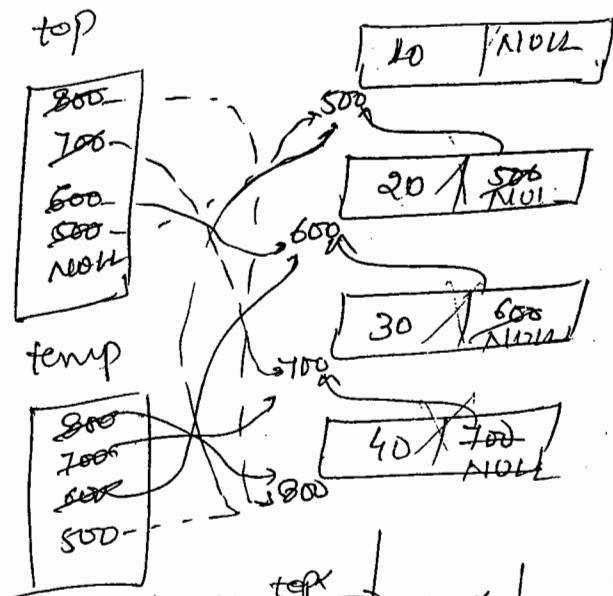
```
top = top->next;
```

```
temp->next = NULL;
```

```
free(temp);
```

```
return;
```

```
}
```



```
void peek()
```

```
{
```

```
STACK *temp;
```

```
if (top == NULL)
```

```
{
```

```
printf("Stack is empty:");
```

```
system("PAUSE");
```

```
return;
```

```
}
```

```
printf("Stack elements is:");
```

```
temp = top;
```

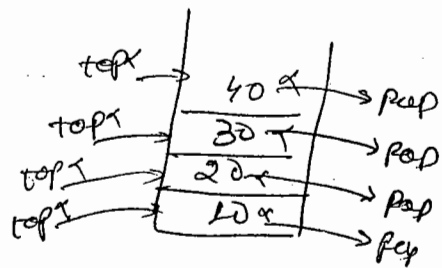
```
do
```

```
{
```

```
printf("%d", temp->data);
```

```
temp = temp->next;
```

```
} while (temp != NULL);
```



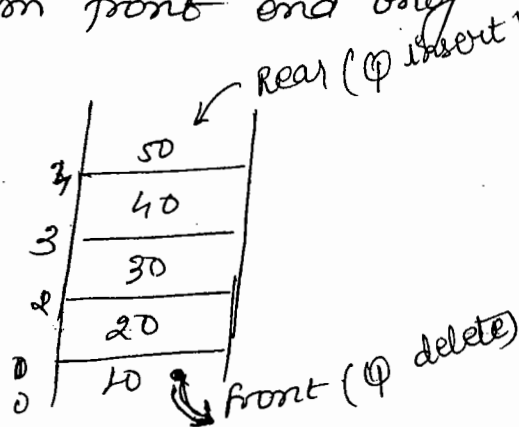
```

system("PAUSE");
return;
}
int main (void)
{
    int mainOption;
    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 allow operations only one end ~~front~~ or Rear.
- ② When we are working with stack top end is open ~~open~~ but in queue both ends are open.
- ③ When we are working with queue operations can allow in 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.



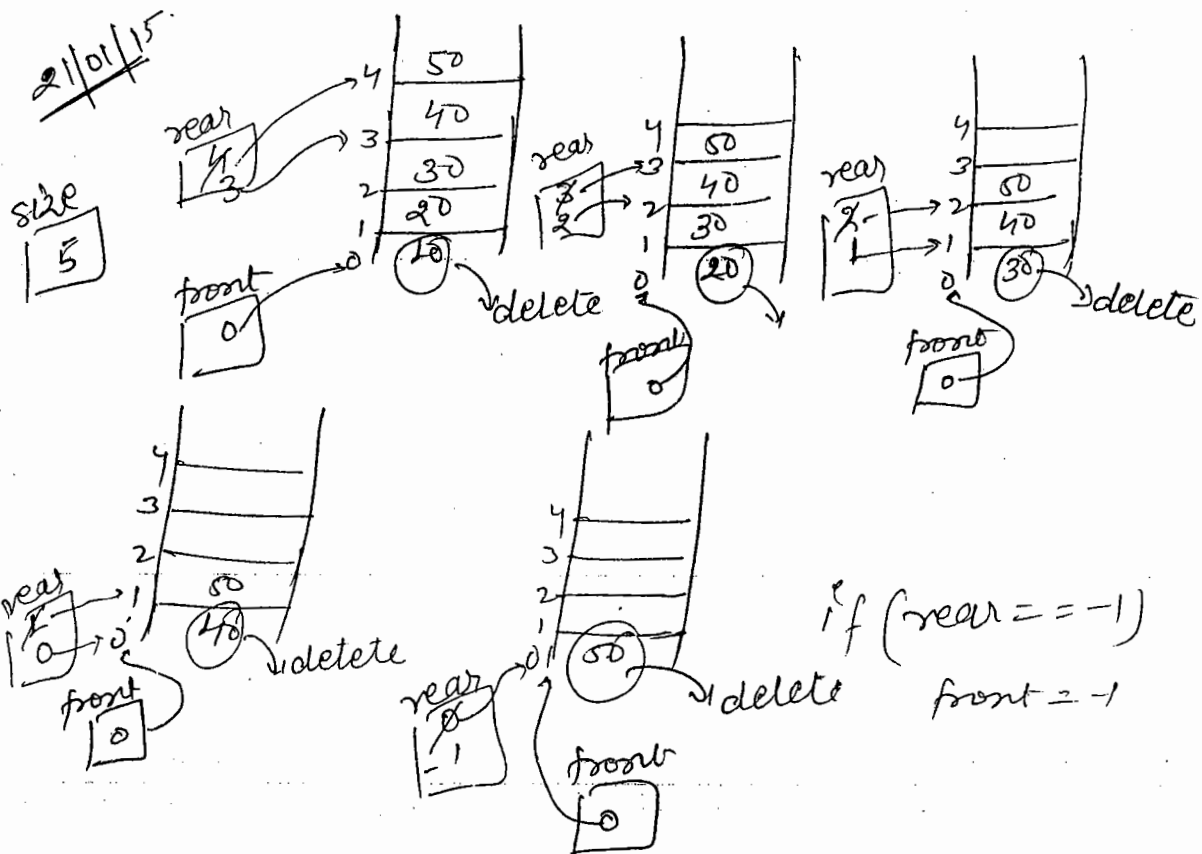
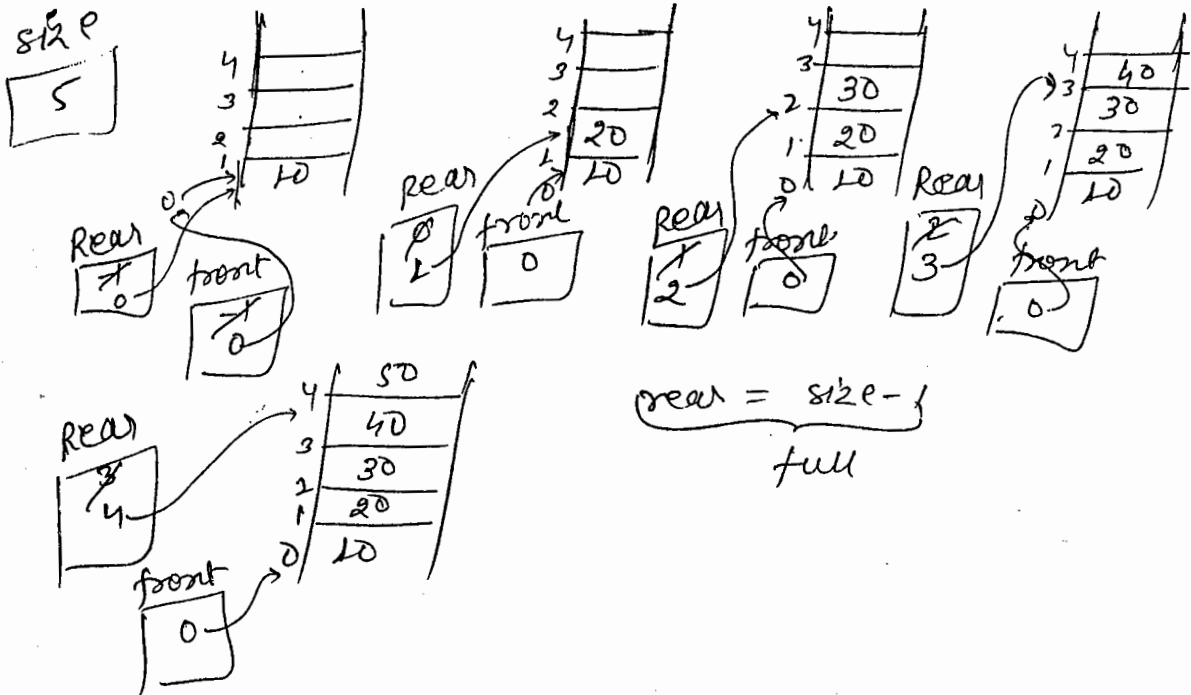
- ⑥ When we are working with queue it allows 3 types of operations. i.e.
 - 1) \varnothing insert
 - 2) \varnothing delete
 - 3) \varnothing display.

\varnothing insert is a procedure of adding an element into \varnothing .

\varnothing delete is a procedure of deleting an element from \varnothing .

By using \varnothing display we require to retrieve the data without deletion.

*) When we are working with queue it must be required to satisfy FIFO or FCFS system.



```

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

#define SIZE 5
int QUEUE[SIZE];
int front = -1;
int rear = -1;

void enqueue()
{
    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 dequeue()
{
    int data;
    if (rear == -1)
    {
        printf("Queue is underflow\n");
    }
}

```



```
system ("PAUSE");
```

```
return;
```

```
}
```

```
Printf ("Delete element is: %d", QUEUE[front]);
```

```
for (i=0; i<frontrear; i++)
```

```
    QUEUE[i] = QUEUE[i+1];
```

```
    --rear;
```

```
if (rear == -1)
```

```
    front = -1;
```

```
system ("PAUSE");
```

```
return;
```

```
}
```

```
void display()
```

```
{
```

```
    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");
        Print("INSERT QUEUE:");
        Print("\n 2 Delete queue:");
        Print("\n 3 display queue:");
        Print("\n 4 Exit - —:");
        scanf("%d", &option);
        switch(option)
        {
            case 1: qinsert();
                    break;
            case 2: qdelete();
                    break;
            case 3: qdisplay();
                    break;
            case 4: returnEXIT SUCCESS;
            default: Print("Invalid option");
                    system("PAUSE");
                    return EXIT_SUCCESS;
        }
    }
}

```

0740 9705758

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 insert_node( )
{
```

```
    if (rear == NULL)
```

```
    {
        printf ("Queue .
```

```
rear = (QUEUE *) malloc (sizeof (QUEUE));
```

```
printf ("Enter the data");
```

```
scanf ("%d", &rear->data);
```

```
rear->next = NULL;
```

```
rear & front = rear;
```

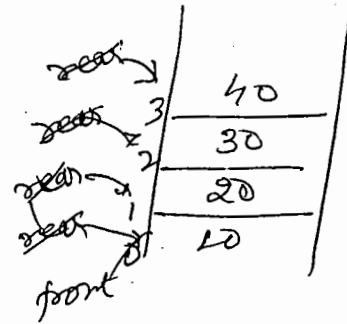
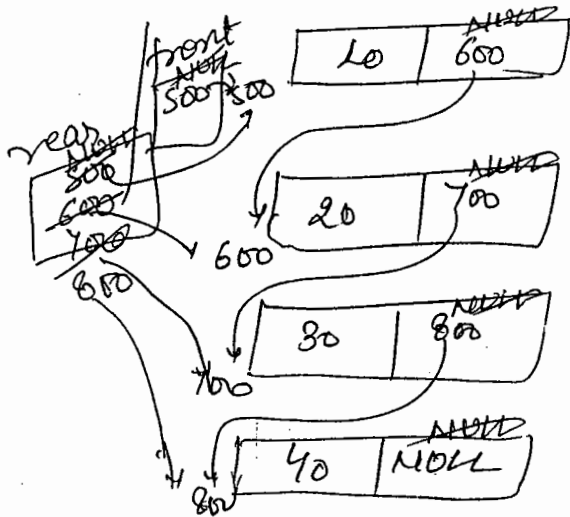
```
return;
```

```
}
```

```

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

```



```

void delete()
{
    QUEUE* temp;
    if (front == NULL)
    {
        printf ("Queue is underflow.");
        system ("PAUSE");
        return;
    }
    temp = front;
    front = front->next;
    temp->next = NULL;
    printf ("deleted 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;

```

```

    while(temp != NULL) printf("Queue element is: ");

```

```

    do
    {

```

```

        printf("%d", temp->data);

```

```

        temp = temp->next;

```

```

    } while (temp != NULL);

```

```

    printf("\n");

```

```

    system("PAUSE");

```

```

    return;
}

```

```

}

```

```

}

```

```

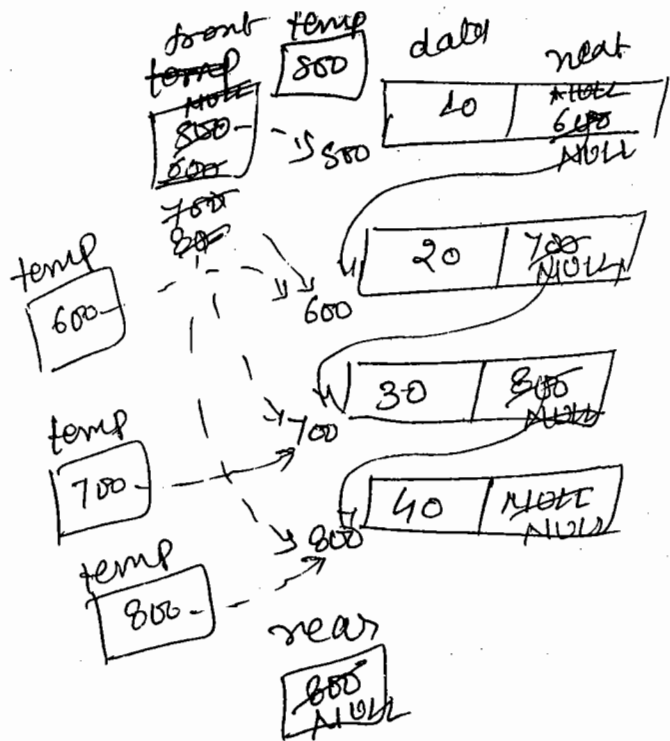
}

```

```

bit mark;
{
  bit 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 sequential manner. i.e. How they are inserted.

b) Priority Queue:-

② 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.

1) In priority queue which elements contains highest priority it will process first, whereas contains least priority it will process at last.

2) When the equal priority is occur in priority queue then it performs FIFO approach.

(c) Circular Queue:-

① When we are working with circular queue all elements are arranged in sequential manner like linear queue, but after last element first element occurs.

(d) De-Queue:-

① 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) Output restricted de-queue.

* In input restricted de-queue insertion can be take place in one end but deletions are allowed on both ends.

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

Implementation of priority queue using 2 lists


```

#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 Qinsert()
{
    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 delete ()
{

```

```

    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 display ()
{

```

```

    int data;
    if (front == NULL)
    {

```

```

        printf ("Queue is empty\n");
        system ("PAUSE");
        return;
    }

```

```

    temp = front;
    while (temp -> next != NULL)
    {

```

```

        printf ("Queue element %d\n", temp->data);
        temp = temp -> next;
    }

```

```

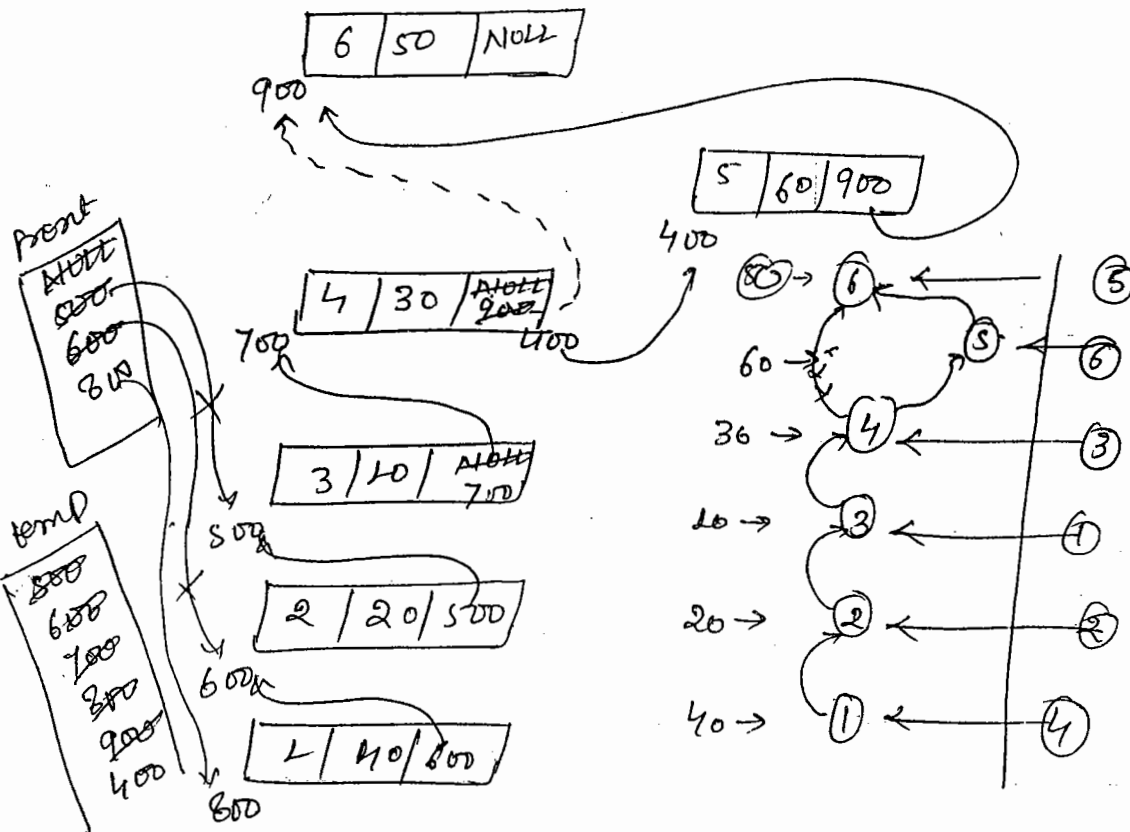
    printf ("key: %d data: %d", temp->key, temp->data);
    temp = temp -> next;
} while (temp != NULL);

```

```

Print ("4)");
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("\nEnter the data: ");
    scanf("%d", &data);
    cqueue[rear] = data;
}
```

```

void delete()
{
    if (front == -1)
    {
        printf ("Queue is underflow");
        getch();
    }
    printf ("deleted data : %d", queue[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, rposition = rear;
    if (front == -1)
    {
        printf ("Queue is empty");
        getch();
        return;
    }

```

```
Print ("Queue elements is: ");
```

```
if (fposition <= rposition)
```

```
{
```

```
while (fposition <= rposition)
```

```
{
```

```
Print ("%d", queue[fposition]);
```

```
fposition ++;
```

```
}
```

```
}
```

```
else
```

```
{
```

```
while (fposition <= MAX - 1)
```

```
{
```

```
Print ("%d", queue[fposition]);
```

```
fposition ++;
```

```
}
```

```
fposition = 0;
```

```
while (fposition <= rposition)
```

```
{
```

```
Print ("%d", queue[fposition]);
```

```
}
```

```
}
```

```
getch();
```

```
}
```

```
int main()
```

```
{
```

```
int option;
```

```
while (1)
```

```
{
```

```
Print ("1 for enqueue:");
```

```
Print ("2 for dequeue:");
```

```

Print ("In 3 for display");
Print ("In 4 for Exit");
scanf ("%d", &option);
switch (option)
{
case 1: insert();
        break;
case 2: delete();
        break;
case 3: display();
        break;
case 4: return EXIT_SUCCESS;
default: Print ("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 occur.

②

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

```
struct queue
{
    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;
```

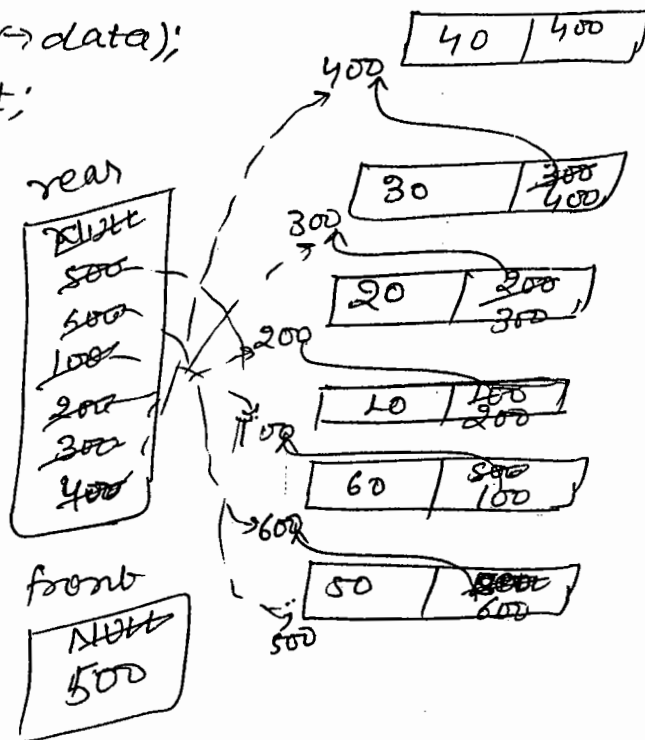
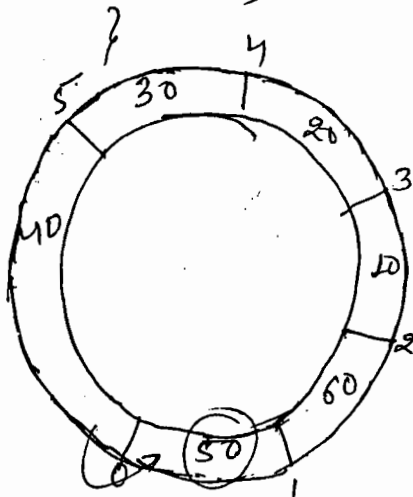
```
}
```



```

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

```



```

void dequeue()
{
    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 == front) // 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");

```

```

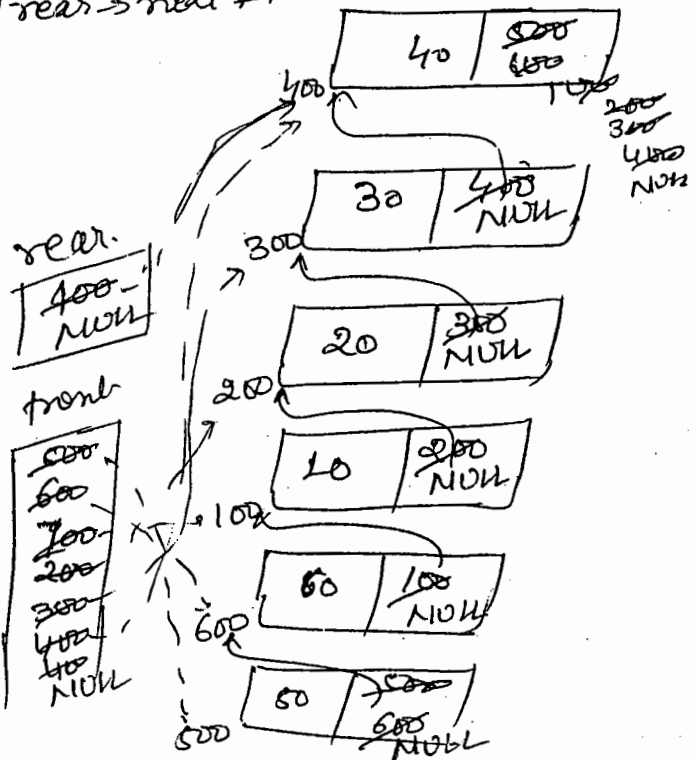
return;

```

```

}

```



```

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_queue()
```

```
{
```

```
int choice;
```

```
while(1)
```

```
{
```

```
printf("\n 1. Insert at right: ");
```

```
printf("\n 2. Delete from left: ");
```

```
printf("\n 3. Delete from right: ");
```

```
printf("\n 4. Display: ");
```

```
printf("\n 5. Return to main: ");
```

```
printf("\n 6. 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 ("Invalid choice");
}
}
}
void output_queue()
{
    int choice;
    while (1)
    {
        printf ("1. insert_right:");
        printf ("2. insert_left:");
        printf ("3. deleteleft_right:");
        printf ("4. display_queue:");
        printf ("5. return to menu:");
        printf ("6. exit:");
        scanf ("%d", &choice);
    }
}

```

```

switch (choice)
{
    case 1: insert-right();
            break;
    case 2: insert-left();
            break;
    case 3: delete-left();
            break;
    case 4: display-queue();
    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 deque:");
        printf("\n3: exit --");
        printf("\n4: enter your choice:");
        scanf("%d", &option);
        switch(option)
        {
            case 1: input-queue();
                    break;

```

```
case 2: output_queue;  
        break;
```

```
case 3: return 0;
```

```
default: printf("Wrong choice");  
        }  
        }  
        }
```

```
void enqueue() {
```

```
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("Enter data:");
```

```
    scanf("%d", &data);
```

```
    deque[right] = data;
```

```
}
```

void insert_left()

{

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 (left == 0)

left = MAX-1;

else

left = left + 2;

printf("Enter data:");

scanf("%d", &data);

dqueue[left] = data;

}

void delete_left()

{

if (left == -1)

{

printf("Queue is underflow:");

getch();

return;

}


```
Printf("\n Deleted data is : %d", dequeue[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", dequeue[right]);
```

```
if (left == right)
```

```
{
```

```
    left = -1;
```

```
    right = -1;
```

```
}
```

```
else if (right == 0)
```

```
    right = MAX-1;
```

```
else
```

```
    right = right + 1;
```

```
}
```

```
void display_queue
```

```
{
```

```
int fposition = left, rposition = right;
```

```
if (left == -1)
```

```
{
```

```
printf ("Queue is empty:");
```

```
getch();
```

```
return;
```

```
}
```

```
printf ("In Queue element ");
```

```
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 like 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:

- ① Queue is used when things don't have to be processed immediately, but have to be processed in FIFO like BFS.
- ② When a resource is shared among ^{multiple} consumers like printer, CPU scheduling and disk scheduler.
- ③ When one data is transferred asynchronously b/w two processes.

Notations:-

- ① When we are evaluating any kind of expression then those expressions are evaluated acc. to notations only.
- ② Notation means evaluating the expression acc. to the position of the operator.

● Notations are classified into 3 types.

● i.e a) Infix Notation.

● b) Prefix Notation.

● c) Postfix Notation.

● 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 operators are presents b/w operands.

● ex) $a + b$
 $a + b * c$

● 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 $+ab$
 $+*bca$

● Postfix Notation:- when we are working with postfix notations ^{all} operators are place after the operands only.

● ex $ab +$
 $abc * +$

Infix to postfix Notation:-

* 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 character is an operand then add the operand into another string called postfix.

b) If the read 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 extracted from infix string.

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

- Repeat step 4 until all elements are pop.
- display the resultant string on console by using prefix string

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

```
char prefix[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 pop(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 (j = 0; (x = infix[i++]) != '\0'; j++)

```

```

    {

```

```

        if (operand(x))

```

```

            postfix[l++] = x;

```

```

        else

```

```

        { if (x == ')')

```

```

            while ((y = pop()) != '(')

```

```

                postfix[l++] = y;

```

```

        }

```

```

        else

```

```

        {

```

```
while (isp(stack[top]) >= isp(x))
```

```
    postfix[ptr++] = pop();
```

```
    push(x);
```

```
}} //for
```

```
while (top >= 0)
```

```
    postfix[ptr] = pop();
```

```
}
```

```
int main()
```

```
{
```

```
    Exit();
```

```
    printf("Enter infix expression: ");
```

```
    scanf("%s", infix);
```

```
    infix2postfix();
```

```
    printf("The resulting postfix expression is",
```

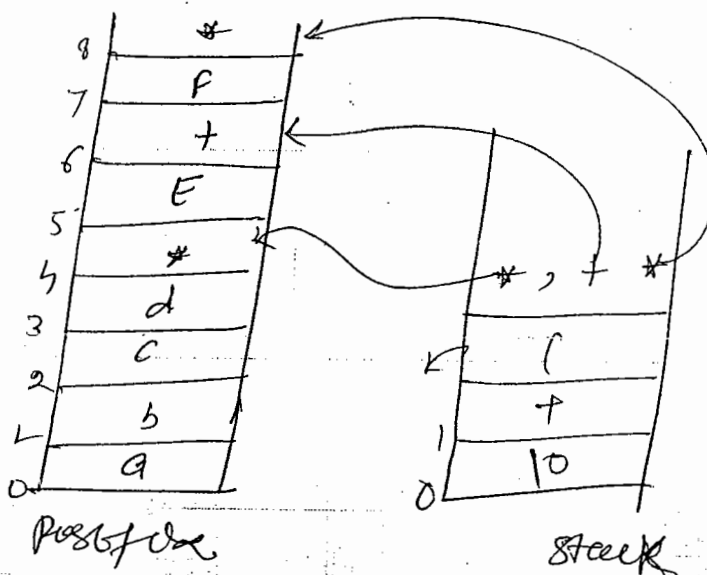
```
    postfix);
```

```
    return 0;
```

```
}
```

$a + b(c * d + e) * f$

a | + | b | (| c | * | d | + | e |) | * | f | \0



$a b c d * e + f *$

26/01/15

Infix to Prefix Notation:

When we are 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 operators and push current operators.

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

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

Step-4) If stack is not empty then pop all the operators 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 <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stack.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 isop(char x)
{
    int y;
    y = (x == '(' ? 6 : x == '^' ? 4 : x == '*' ? 3 : x == '/' ? 2 :
        x == '+' ? 1 : x == '-' ? 1 : x == ')' ? 0 : -1);
}

int icp(char x)
{
    int y;
    y = 1
```

```
void infixToPrefix()
```

```
{
```

```
    int j, l=0;
```

```
    char x, y;
```

```
    stack[++top]='\\0';
```

```
    for (j = strlen(infix) - 1; j >= 0; j--)
```

```
    {
```

```
        x = infix[j];
```

```
        if operand(x)
```

```
            prefix[l++] = x;
```

```
        else
```

```
        {
```

```
            if (x == '(')
```

```
                while ((y = pop()) != ')')
```

```
                    prefix[l++] = y;
```

```
            else
```

```
            {
```

```
                while (isp(stack[top]) >= dep(x))
```

```
                    prefix[l++] = pop();
```

```
                push(x);
```

```
            }
```

```
        }
```

```
    }
```

```
int main()
```

```
{
```

```
    system("CLS");
```

```
    clrscr();
```

```
    printf("Enter an infix: ");
```

```
    scanf("%s", infix);
```

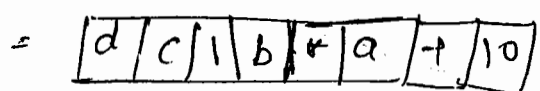
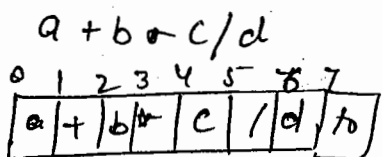
```
    infixToPrefix();
```

```
    store(prefix);
```

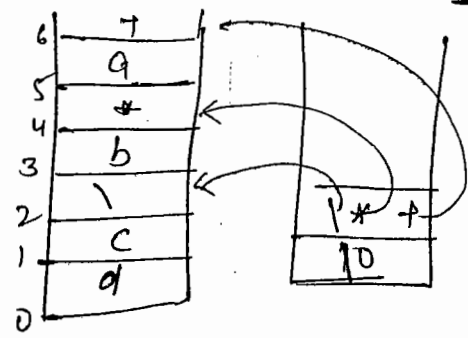
```
    printf("The resulting prefix is %s", prefix);
```

```
    return exit - success;
```

```
}
```



$+ a + b / cd$



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 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

- 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 than of middle value then searching is proceed in left hand side, or if the searching element is greater than of the middle

27-01-15

searching -

→ full.

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

→ ~~when~~ ~~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 a 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

→ 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 searching element is greater than of middle value then searching ~~is~~ 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("m^position is %d", data, position);
```

```
}
```

```
else
```

```
printf("m %d is not found", data);
```

```
} return 0;
```

Op - Enter 10 values: 50 60 60 70 30 80 20 90 10 45

0 1 2 3 4 5 6 7 8 9

50	60	60	70	30	80	20	90	10	45
----	----	----	----	----	----	----	----	----	----

size

10

i
0
1
2
3

data

70

position

i
i+1
= 4

flag

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 ("In Enter array size:");
    scanf ("%d", &size);
    arr = (int *) malloc (size, sizeof(int));
    printf ("In Enter %d values:", size);
    for (i = 0; i < size; i++)
        scanf ("%d", &arr[i]);
    printf ("In 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++) { if (arr[mid] == data) {
 mid = min + max / 2; } }

10 20 30 40 50 60 70 80 90 100
 ↑ 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");

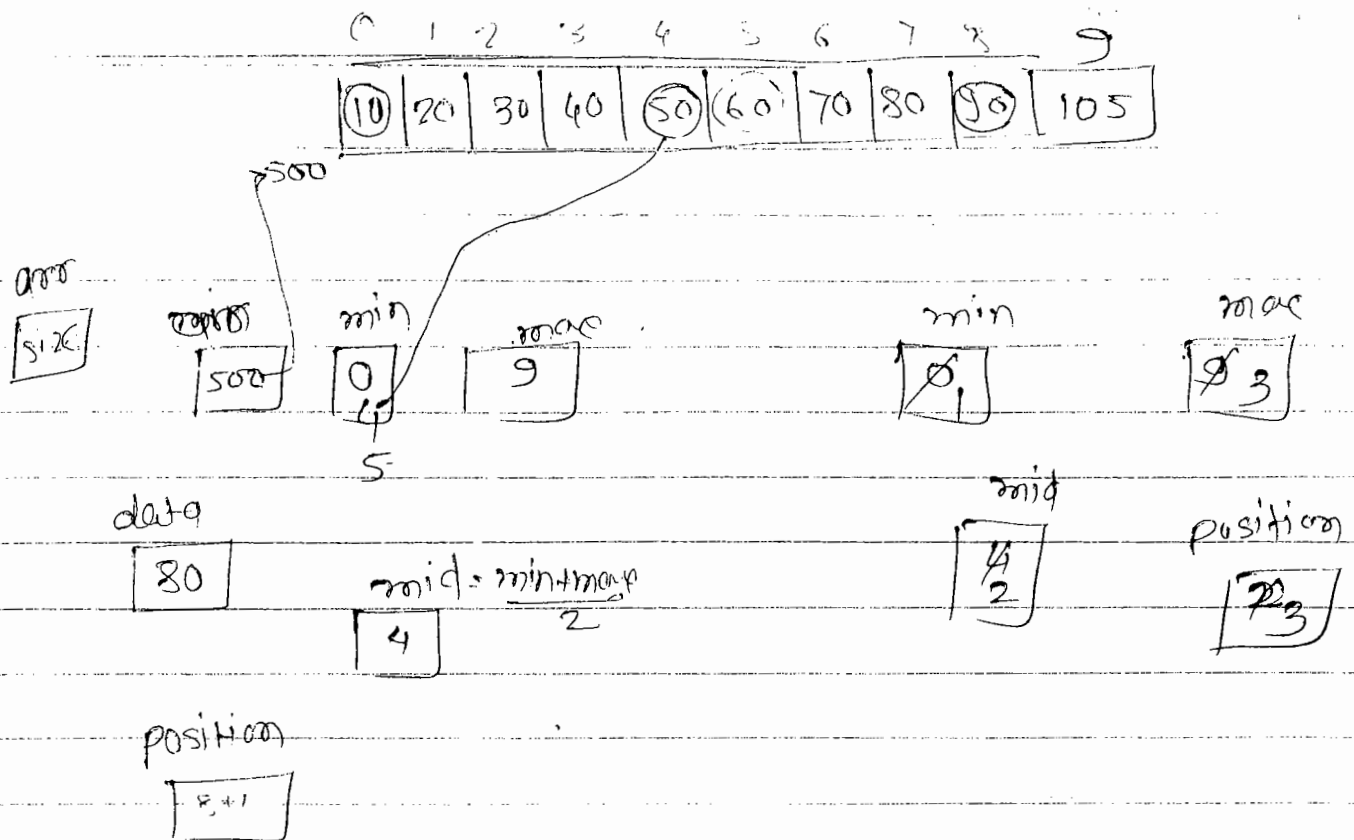
free(arr);

arr = NULL;

getch();

return 0;

}



Sorting -

it is a procedure of arranging the data in a particular order i.e. ascending or descending order.

→ In C++ we having 8 types of sorting.

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

1. Bubble sort -

When we are working with bubble sort adjacent elements are compared 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 $n-1$ comparison will take place.

→ When we are working with bubble sort element are arranged in descending order but final result is ascending 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]);
```

Answer starting.

(iii) Two values fixed, i.e. max or min value.

(iv)

[50 60 70 80 90]	0 < 9	0 < 8
[50 60 70 80 90]	1 < 9	0 < 8

$$\begin{array}{cc} 0 < 9 & 0 < 8 \\ 1 < 9 & 0 < 8 \\ 2 < 9 & 2 < 8 \end{array}$$

...

م

5

```
arr[j] = arr[j+1];
```

3

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

}

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

500

500-

8
6
7
9

0 < 9	0 < 8	0 < 7	0 < 6
1 < 9	1 < 8	1 < 7	1 < 6
2 < 9	2 < 8	2 < 7	2 < 6
3 < 9	3 < 8	3 < 7	3 < 6
4 < 9	4 < 8	4 < 7	4 < 6
5 < 9	5 < 8	5 < 7	5 < 6
6 < 9	6 < 8	6 < 7	6 < 6
7 < 9	7 < 8	7 < 7	
8 < 9	8 < 8		
9 < 9			

2. Selection sort -

When we are working with selection sort always comparison 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 $n-1$ comparison will take place.

→ When we are working with selection sort elements are arranged in ascending order & final result also ascending 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++)
```

```
    {
```

until 2021 is passed. for maintenance

```
for (j = i + 1; j < size; j++)
```

```
{
```

```
if (arr[j] < arr[i])
```

```
{
```

```
t = arr[i];
```

```
arr[i] = arr[j];
```

```
arr[j] = t;
```

```
arr[i]
```

```
}
```

```
}
```

```
}
```

```
printf ("sorted elements are: ");
```

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

```
printf ("%d ", arr[i]);
```

```
return 0;
```

```
}
```

0	1	2	3	4	5	6	7	8	9
50	60	40	70	30	80	20	90	10	45
500		50		40		30		20	18

size
10

arr
500

40
30
20
10
i

0
1
2
3

j

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8
6	7	8	9
7	8	9	
8	9		
9			

3. Insertion sort

When we are working with insertion sort it will compare the element in sequential order but whenever an element requires 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 elements because one element position will search in proper way and to insert that element remaining all elements 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 ("%d", &size);
```

```
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;
```

```
}
```

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

③ cond & arr[i] < arr[j] then
 ④ arr[i+1] = arr[i]

arr

i	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
-1	0	1	2	3	4	5	6	7	8	9
(-1)	-1	0	1	2	3	4	5	6	7	8
(-1)	-1	0	1	2	3	4	5	6	7	8
(-1)	-1	0	1	2	3	4	5	6	7	8

J

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

last

0	1	2	3	4	5	6	7	8	9
60	80	40	20	70	90	30	10	45	100

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 need 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("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", &arr1[i]);
```

```
bubbleSort(arr1, s1);
```

```
printf("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", &arr2[j]);
```

```
bubbleSort(arr2, s2); // sorting array 2.
```

```
printf("Sorted array 1: ");
```

```
for (i=0; i<s1; i++)
```

```
printf("%d ", arr1[i]);
```

```
printf ("1n 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;
}
```

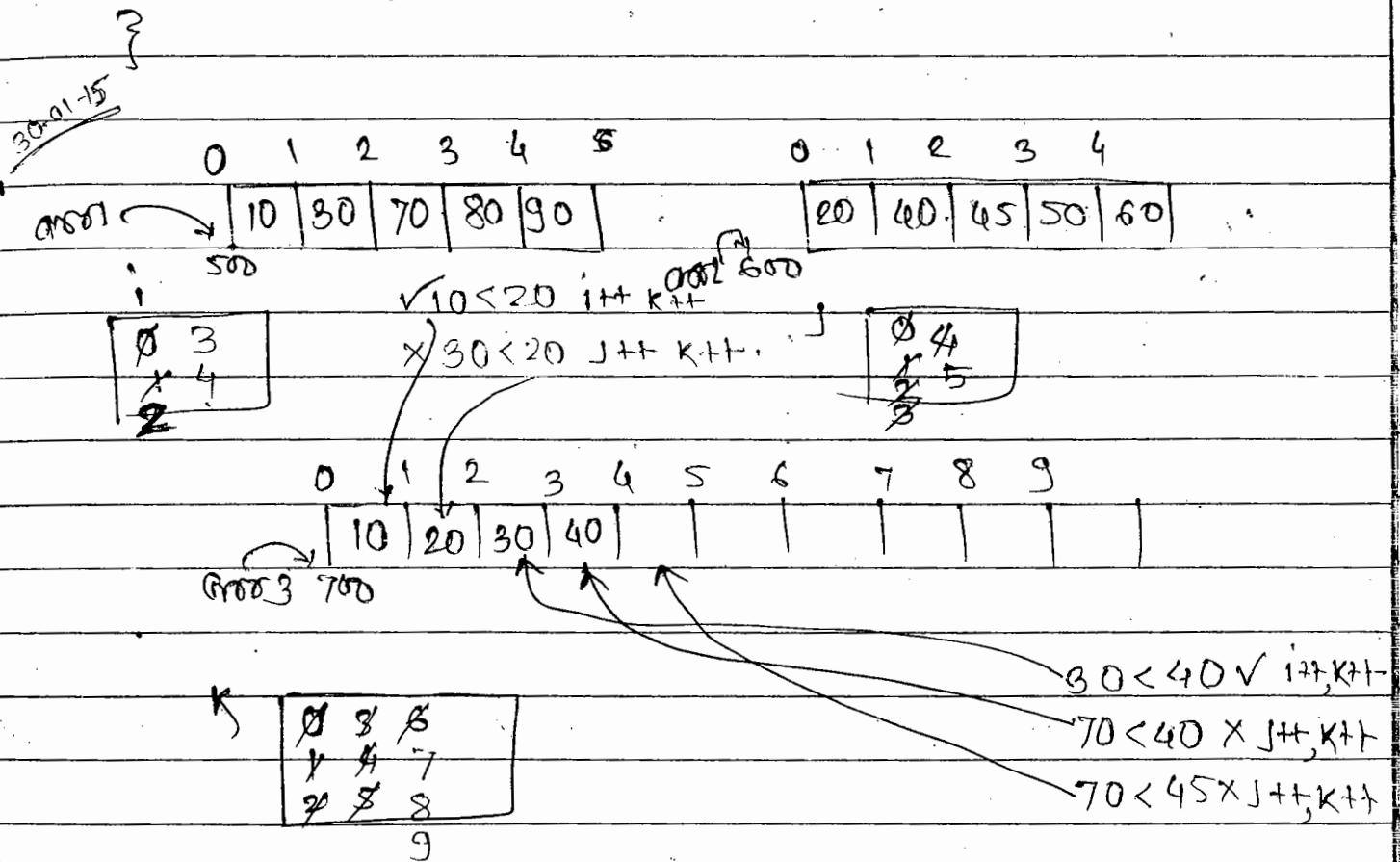
```
while (i < s1)
{
    arr3[k++] = arr1[i++];
}
while (j < s2)
{
    arr3[k++] = arr2[j++];
}
```

```
printf ("1n sorted array 3: ");
for ( k=0; k < s3; k++)
    printf ("%.d", arr3[k]);
```

```

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

```



5. Implementation of Quick sort -

```
#define n
```

```
#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);

```

```
printf("\nEnter %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	60	40	70	30	80	20	90	10	45
----	----	----	----	----	----	----	----	----	----

50 45 50

n

10

(1) fix so
pp

	Piv	L	R	t
False or 0	9 ^①	0 ^①	9	0 50 ^②
	9	1	8	
	1	2	7	
	8	3	6	
	3	4	5	
	6	5		
	5			

Quick

arr	low	up
500	0	9

(50) 8e.

arr	low	up
500	0	4

arr	low	up
500	0	3

arr	low	up
500	0	2

arr	low	up
500	0	1

arr	low	up
500	0	0

arr	low	up
500	2	1

arr	low	up
500	3	2

call
 return

2) 8e 45

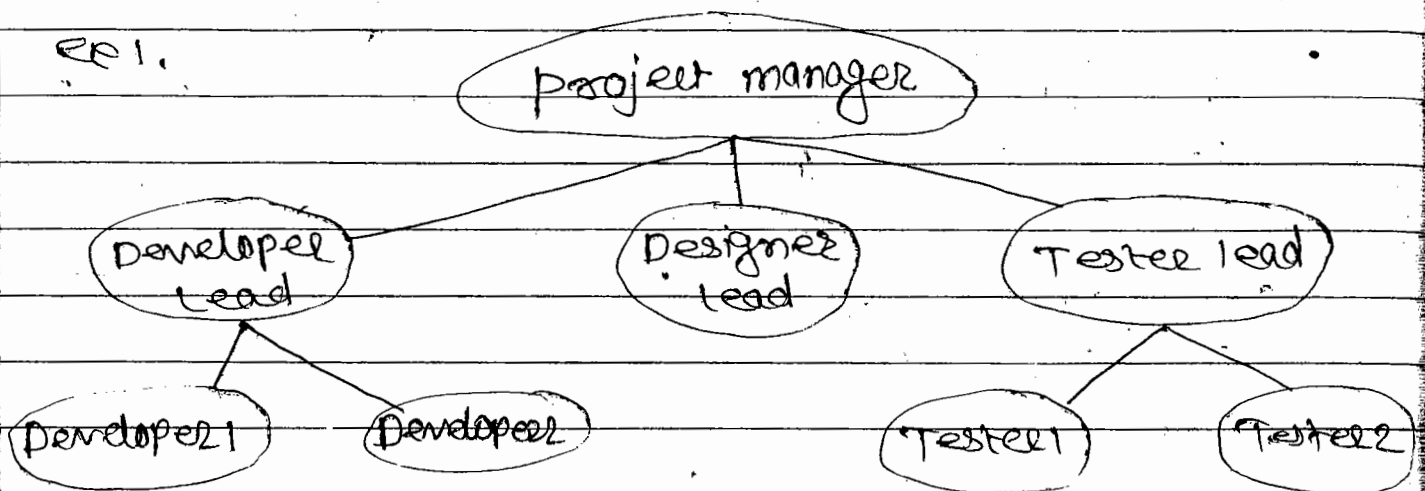
arr	low	up
500	0	0

Trees -

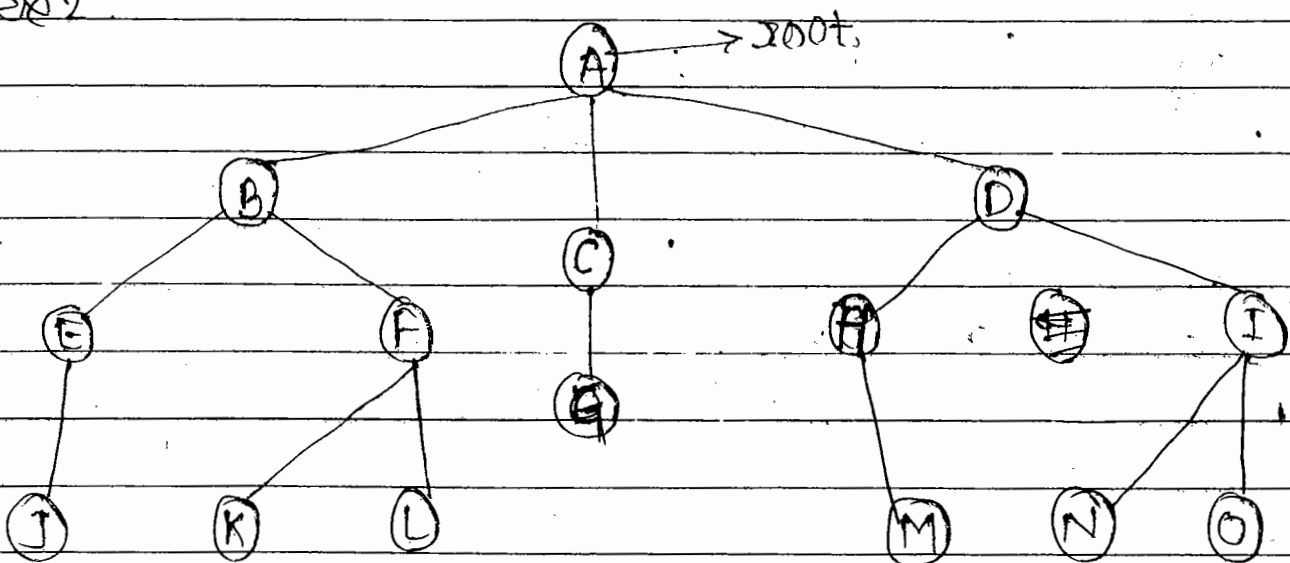
A tree is a non-linear data structure in which items are arranged in ^{sorted} sequence.

→ Tree data structure is used to represent hierarchical ~~se~~ relationship betⁿ existing serial items.

ex 1.



ex 2



* Tree terminology -

Root - it is a specially designed data item in a tree. it is the first int 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.
- Every node specifies the data info & 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 ~~errot~~ 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 ~~is~~ 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 ~~between~~ 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).

Depth -

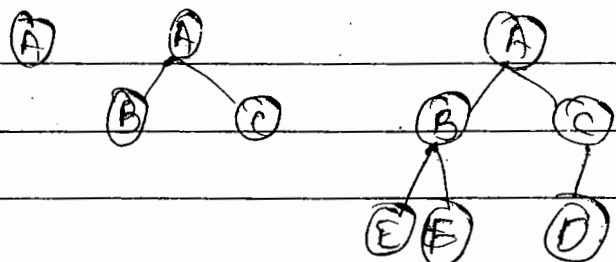
Maximum level of any node in a given tree i.e height of tree.

Forest -

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

* Binary tree -

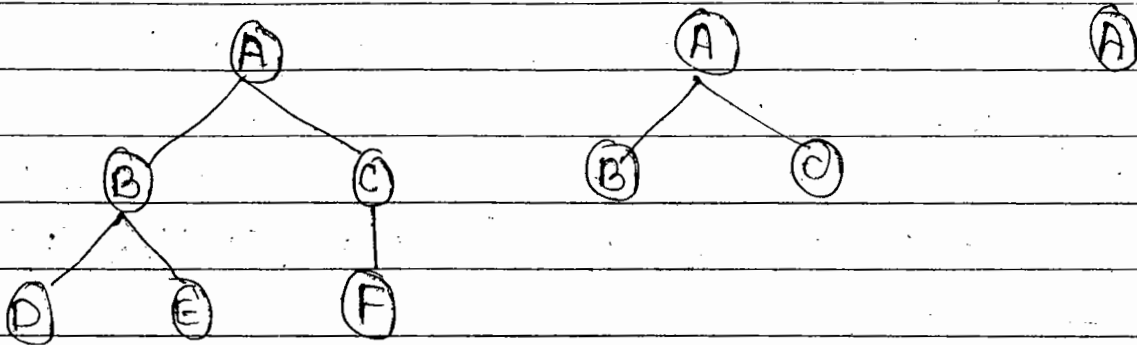
level 0 - 1 node
level 1 - 2 subtree
level 3 - may be two or one



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.

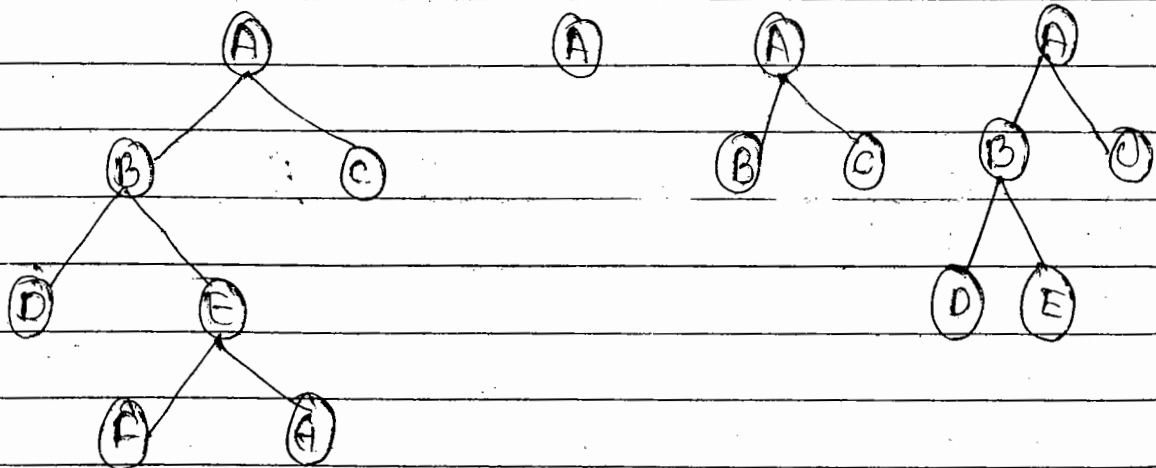
ex.



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.

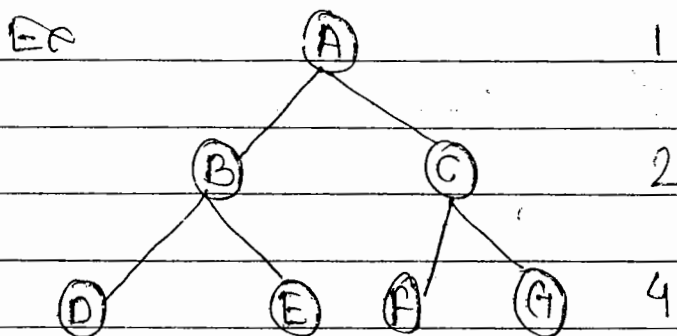
ex.



complete binary tree -

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

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



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

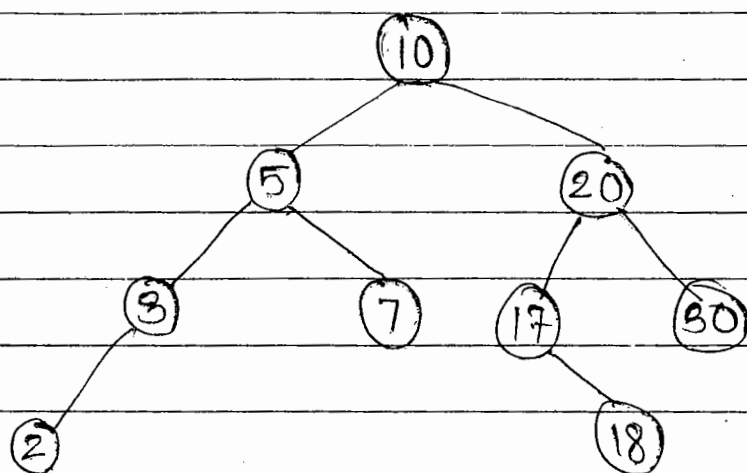
→ when we are organizing the b.T then need to follow following steps.

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

② Newly added element value is less than or

root element then it should be organised in left subtree.

③ If Newly entered element is greater than or root value then it should be organised in right subtree.



Tree traversal -

In fully developed binary tree there are three type of traversal is possible 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 subtree in preorder
- ③ travel the right subtree in preorder.

10 5 3 2 7 20 17 18 30

② Inorder traversal -

When we are travelling inorder procedure then

- ① first visit left sub-tree in inorder
- ② visit the root node
- ③ travel in 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 at last.

2 3 5 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
```

```
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("\n5 for Exit .....");
```

```
scanf("%d", &choice);
```

```
switch (choice)
```

```
{
```

```
case 1:
```

```
printf("Enter the data:");
```

```
scanf("%d", &num);
```

```
insert (&root, num);
```

```
break;
```

```
case 2:
```

```
printf("\n pre order traversal :");
```

```
preorder (root);
```

```
getch();  
break;
```

case 3:

```
printf("In In-order Traversal:");  
inorder (root);  
getch();  
break;
```

case 4:

```
printf("In post-order traversal:");  
postorder (root);  
getch();  
break;
```

case 5: tree (root);
return 0;

default: printf("Invalid option:");

}

}

}

void insert (node **temp, int num)

{

if (*temp == NULL)

{

*temp = malloc (sizeof (node));

(*temp) → left = NULL;

(*temp) → ~~data~~ → 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;
}

```

temp

400
400
400

void preorder (node * temp)

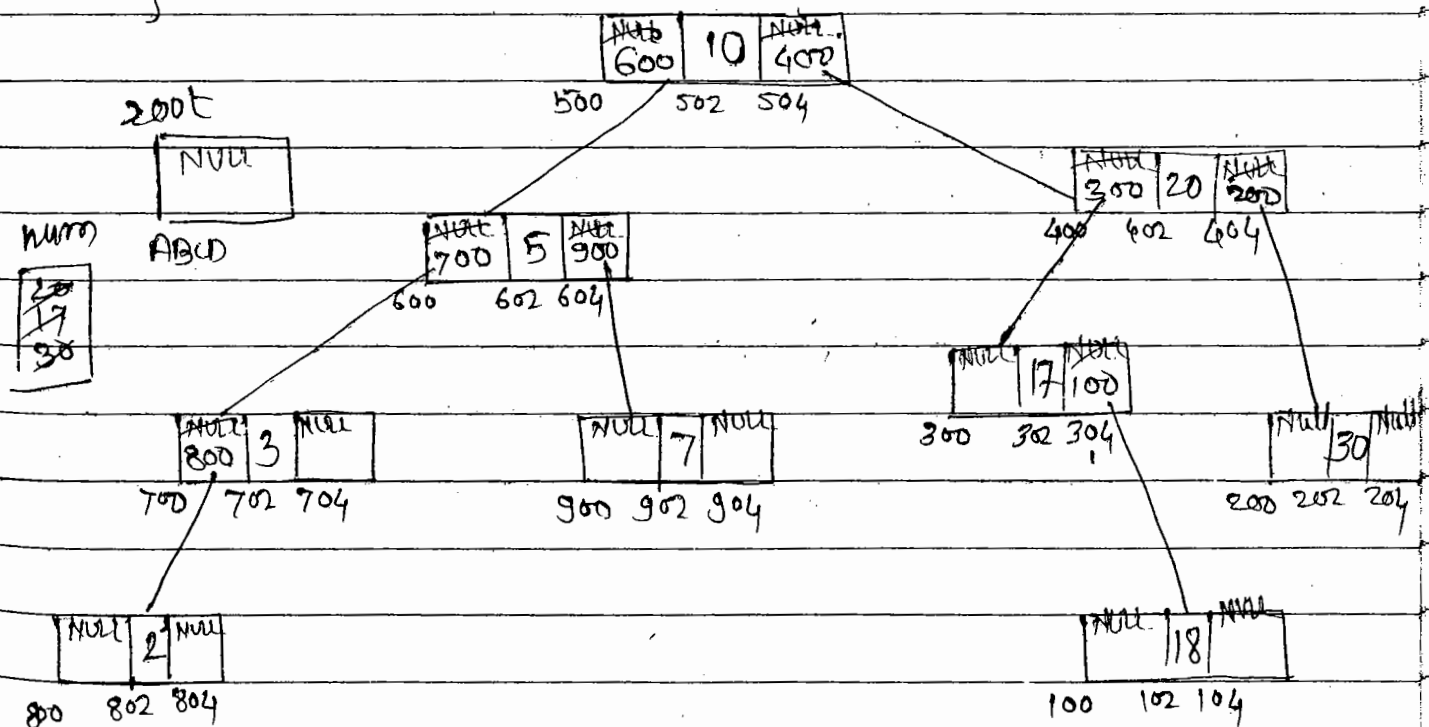
```

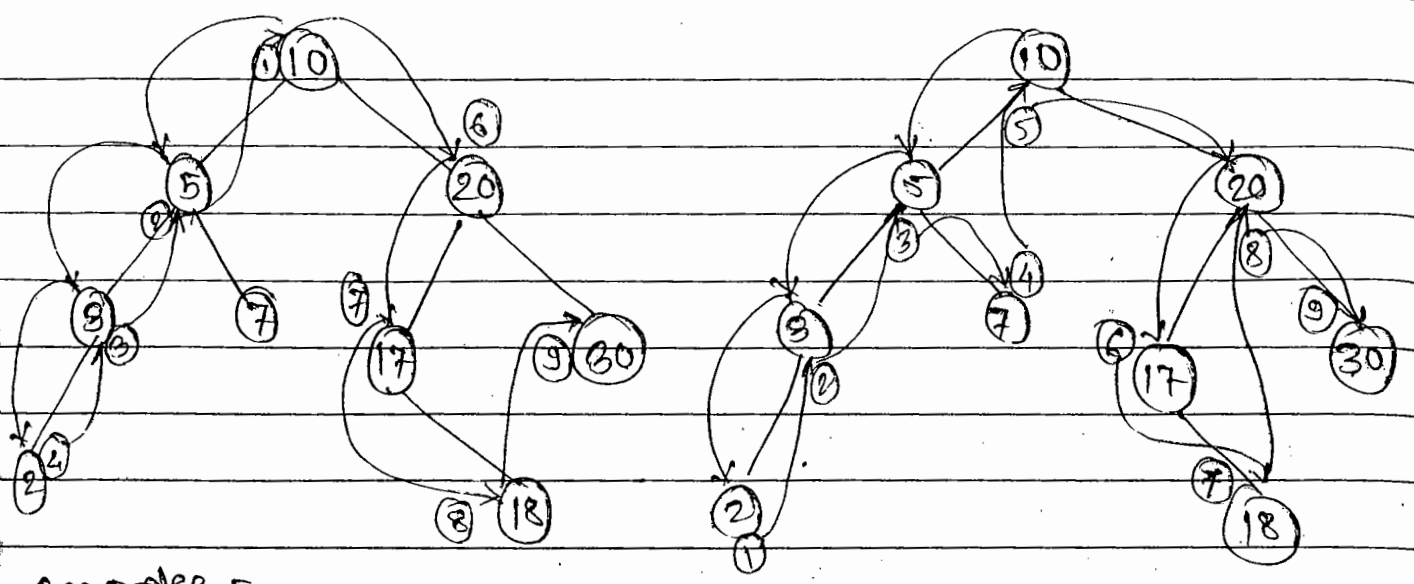
{
    if (temp != NULL)
    {
        printf ("%d", temp → data);
        preorder (temp → left);
        preorder (temp → right);
    }
}

```

```
else
return;
}
```

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

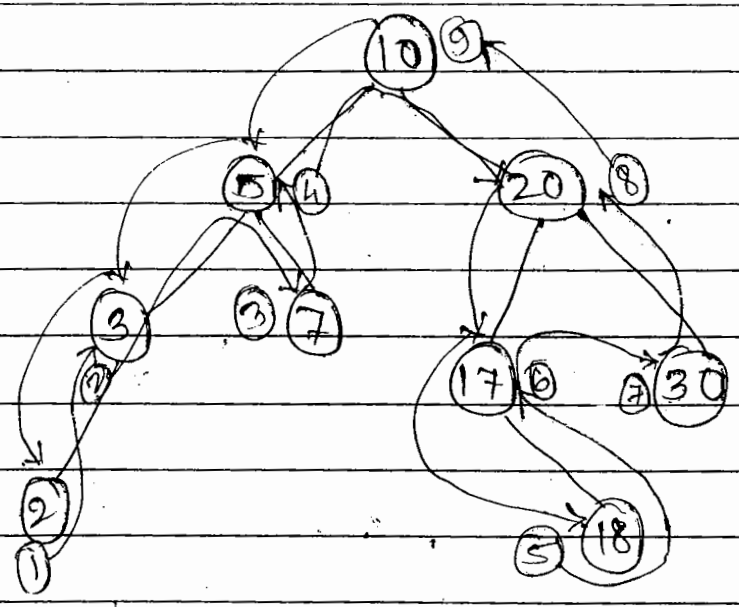




preorder -

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

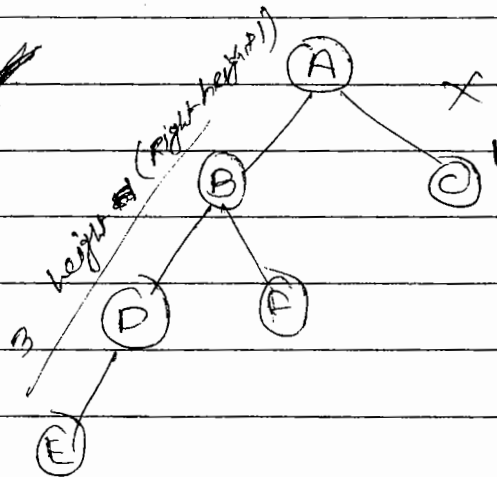
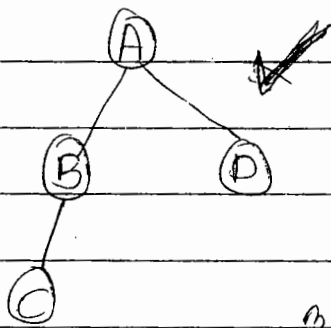
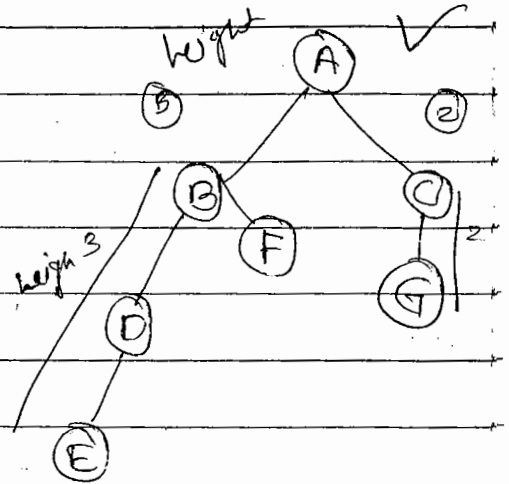
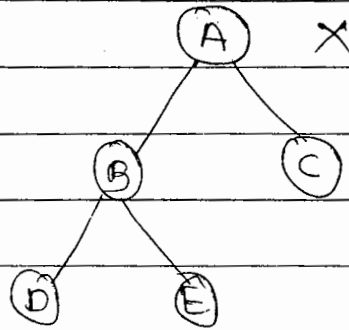
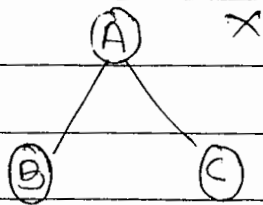
3-02-15

AVL Tree (Adelson-Velskii & Landis) -

it is a self balanced - binary search tree

② when we are working with AVL tree left subtree height should be equal to right subtree 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.

⑤ Implementation of AVL tree with ~~the~~ traversing process

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

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

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

```
typedef struct node
```

```
{
```

```
    struct node *left;
```

```
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 inorder (node *); // root address is parameter
void postorder (node *); // root address is parameter
int height (node *);
node *rotateRight (node *); // auto balance in right subtree
node *rotateLeft (node *); // auto balance 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 *); // balance factor.
int main ()
```

```
{
    node *root = NULL;
    int x, n, i, OP;
```

```
do
{
```

```
    printf ("\n 1 Create:");
    printf ("\n 2 Insert:");
    printf ("\n 3 Delete:");
    printf ("\n 4 Print:");
    printf ("\n 5 @wt:");
    printf ("\n Enter your choice:");
```



```
scanf("%d", &op);
```

```
switch (op)
```

```
{
```

```
case 1: 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;
```

```
case 2: printf("\n Enter a data :");
```

```
scanf("%d", &x);
```

```
root = insert (root, x);
```

```
break;
```

```
case 3: printf("\n Enter a data :");
```

```
scanf("%d", &x);
```

```
root = Delete (root, x);
```

```
break;
```

```
case 4: 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);

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 = Delete(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) // rebalance during

```

    if (BF(temp → right) <= 0)
        temp = RR(temp);
    else
        temp = RL(temp);
}
else
{

```

// data to be deleted found.

```

if (temp → right != NULL)
{

```

1 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);
```

```
}
```

temp

```
node * rotateRight (node *temp)
```

```
{
```

```
    node *y;
```

```
    y = x → left;
```

```
x →
```

```
    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);
```

```
}
```

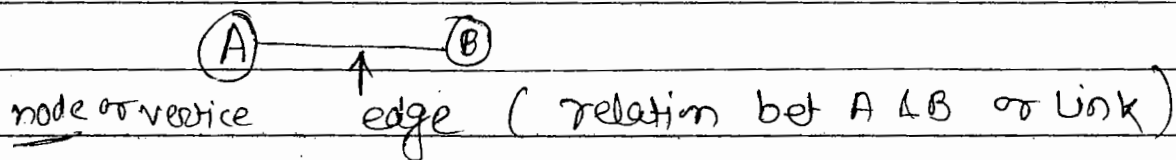
```
}
```

04-02-15

Graphs -

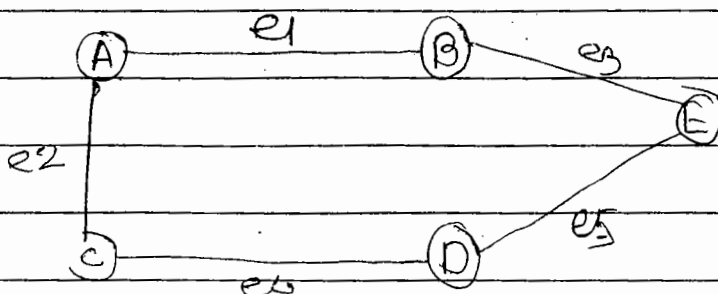
A graph 'G' consists of a set 'V' of vertices (nodes) & a set 'E' of Edge.

- 'V' is a finite & non-empty set of vertices
- 'E' is a set of ^{pairs of} vertices, these 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.



$$V(G) = \{V_1, V_2, V_3, V_4, V_5\}$$

$$E(G) = \{e_1, e_2, e_3, e_4, e_5\}$$

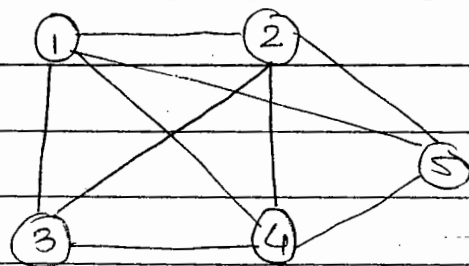


there are five vertices & five edges in the graph.

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

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

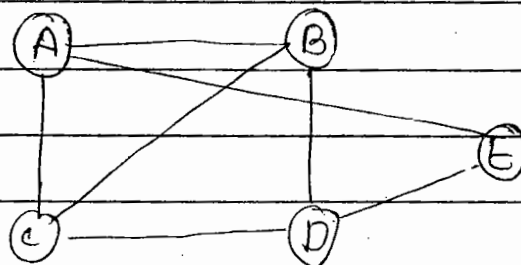
$$E(G) = \{(1, 2), (2, 4), (2, 3), (1, 4), (1, 5), (4, 5), (3, 4)\}$$



Graph terminology -

1) Adjacent vertices -

vertex v_1 is said to be adjacent to a vertex v_2 if there is an edge (v_1, v_2) or (v_2, v_1) .

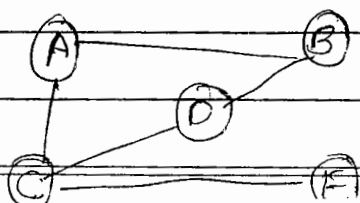


Adjacent

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 adjacent to the next.

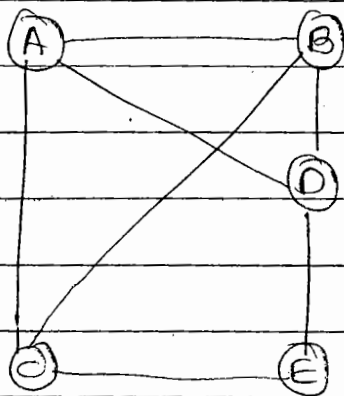


$$W(A, E) = \{ (A, B), (B, D), (D, C), (C, E) \}$$

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

Cycle -

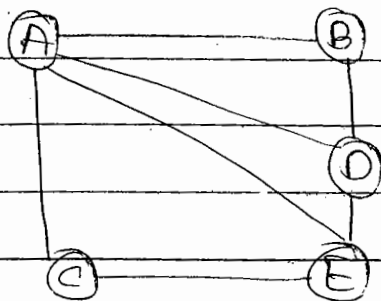
A cycle is the path in which first & last vertices are the same.



(reverse dirn is possible)

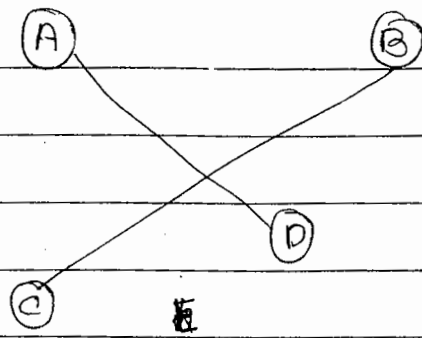
Unconnected graph -

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



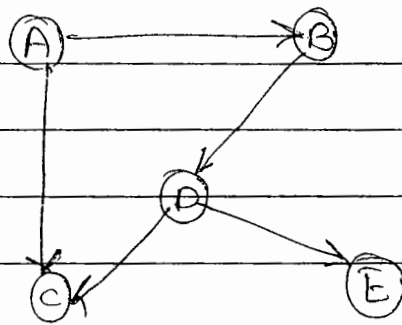
Unconnected graph -

→ graph is called unconnected if there no path is exists between any one of the vertex.



Directed graph -

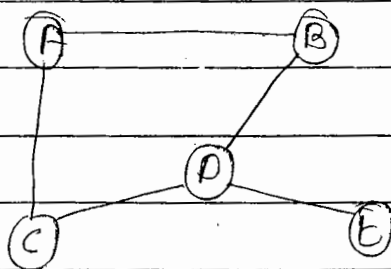
A graph is called directed if there exists a path direction from any vertex to any other vertex.



(it doesn't support reverse dir'n)

undirected graph

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

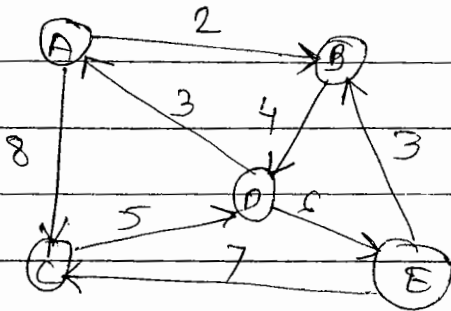


Degree -

The number of edges incident is called degree.

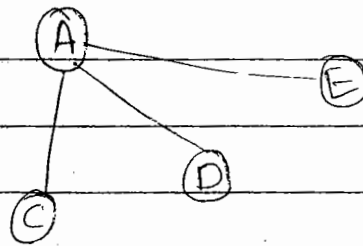
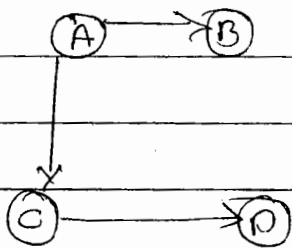
Weighted graph -

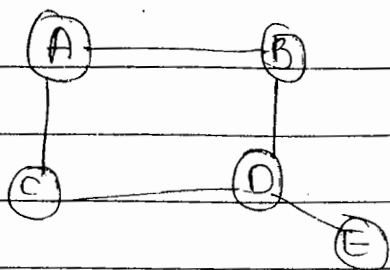
A graph is said to be weighted graph if every edge in the graph is assigned some weight or value.



Tree -

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





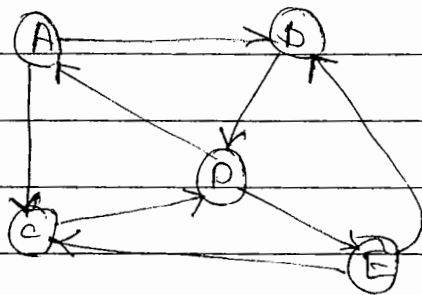
A degree $\rightarrow 2$ $((A,B), (A,C))$
 B degree $\rightarrow 2$ $((B,A), (B,D))$
 C degree $\rightarrow 2$ $((C,A), (C,D))$
 D degree $\rightarrow 3$ $((D,B), (D,C), (D,E))$
 E degree $\rightarrow 1$ $((E,D))$

Indegree -

Incoming edges of vertex is called indegree

outdegree -

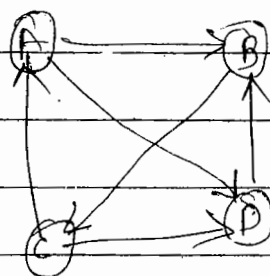
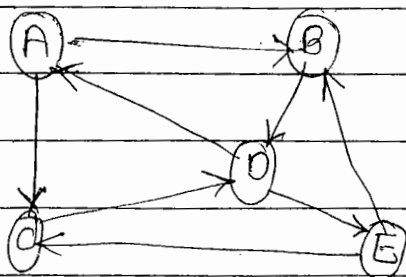
Outgoing edges of vertex is called outdegree



$IN(A) = 1$	$OD(A) = 2$
$IN(B) = 2$	$OD(B) = 1$
$IN(C) = 2$	$OD(C) = 1$
$IN(D) = 2$	$OD(D) = 2$
$IN(E) = 1$	$OD(E) = 2$

complete graph -

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



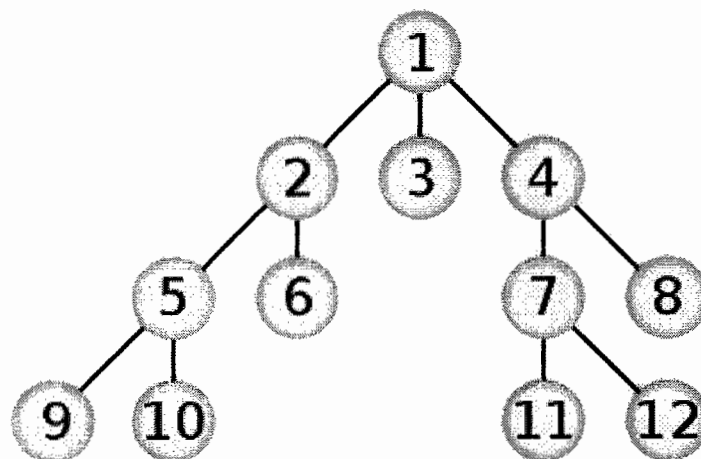
What's the difference between DFS and BFS?

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

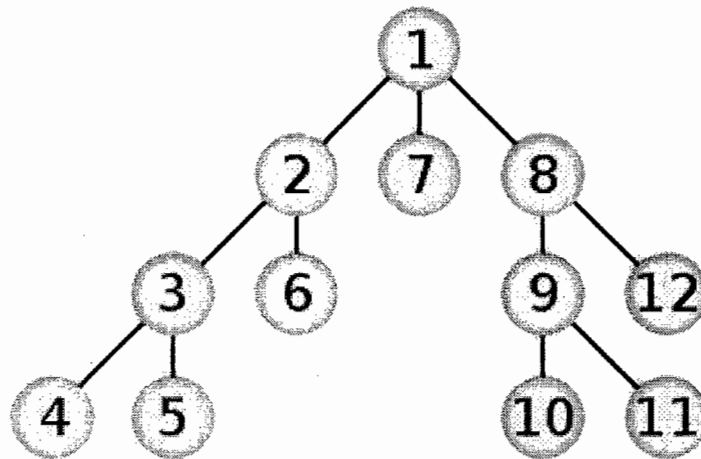
Here's an example of what a BFS would look like. The numbers represent the order in which the nodes are accessed in a BFS:



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:



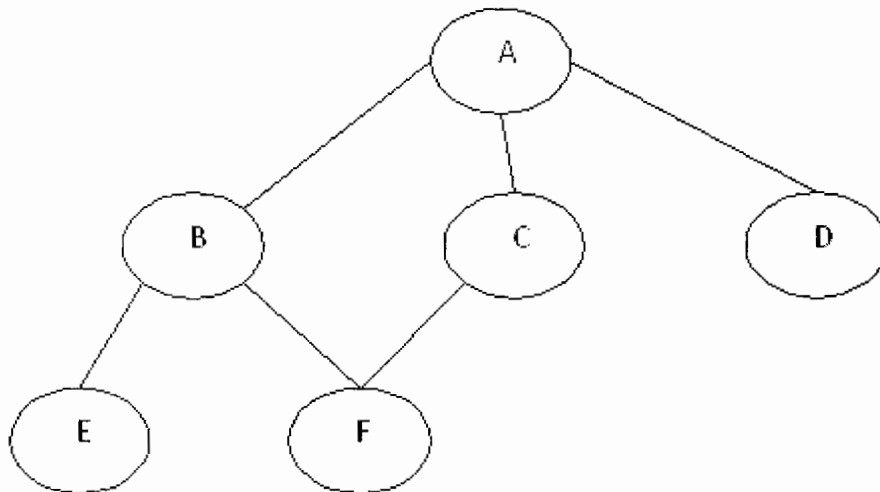
Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it's not necessary to store all of the child pointers at each level.

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();
}

```