DATA STRUCTURES

OBJECTIVES :

→ To Understand the concepts of ADTs.

→ To learn linear data structures - lists, Stacks and queues.

→ To Understand Sorting, Searching and hashing algorithms.

→ To apply Tree and Graph structures.

UNIT - I

LINEAR DATA STRUCTURES - LIST

Abstract Data Types (ADTs) - List ADT - array-based implementation - linked list implementation - Singly linked lists - circularly linked lists - doubly linked lists - applications of lists - Polynomial Manipulation - All operations (insertion, Deletion, Merge, Traversal).

UNIT - II

LINEAR DATA STRUCTURES - STACKS, QUEUES

Stack ADT - operations - Applications - Evaluating arithmetic expressions - Conversion of Infix to Postfix expression - Queue ADT - Operations - Circular Queue - Priority Queue - dequeue - Applications of Queues.

UNIT-III NON LINEAR DATA STRUCTURES - TREES

Tree ADT - tree traversals - Binary Tree ADT - Expression trees - Applications of trees - Binary Search tree ADT - Threaded Binary Trees - AVL Trees - B-Tree - B+ Tree - Heap - Applications of heap.

UNIT-IV

NON LINEAR DATA STRUCTURES - GRAPHS

Definition - Representation of Graph - Types of Graph - Breadth - first traversal - Depth - first traversal - Topological Sort - Bi - Connectivity - cut vertex - Euler circuits - Applications of graphs.

UNIT-V

SEARCHING, SORTING AND HASHING TECHNIQUES.

Searching - Linear Search - Binary Search, Sorting - Bubble Sort - Selection Sort - Insertion Sort - Shell Sort - Radix Sort - Hashing - Hash functions - Separate Chaining - Open Addressing - Rehashing - Extendible Hashing.

OUTCOMES: At the end of the course, the student should be able to;

→ Implement ADT for linear data structures.

→ Apply the different linear and non-linear data Structures to problem solutions.

→ critically analyze the Various sorting algorithms.

UNIT- I

LINEAR DATA STRUCTURES - LIST

1) Abstract Data Types (ADTs)
2) List ADT
3) Array Based implementation
4) Linked list implementation
5) Singly linked lists
6) circularly linked lists
7) Doubly - linked lists
8) Applications of lists
9) polynomial manipulation
10) All operations (Insertion, Deletion, merge, Traversal).

Data Structures in C :

They are used to store data in a Computer in an organized form. In C language different types of data structures are;

- Array
- Stack
- Queue
- Linked List
- Tree
- Graph.

Array: It is Collection of similar data type, you can insert, delete element from array without any order.

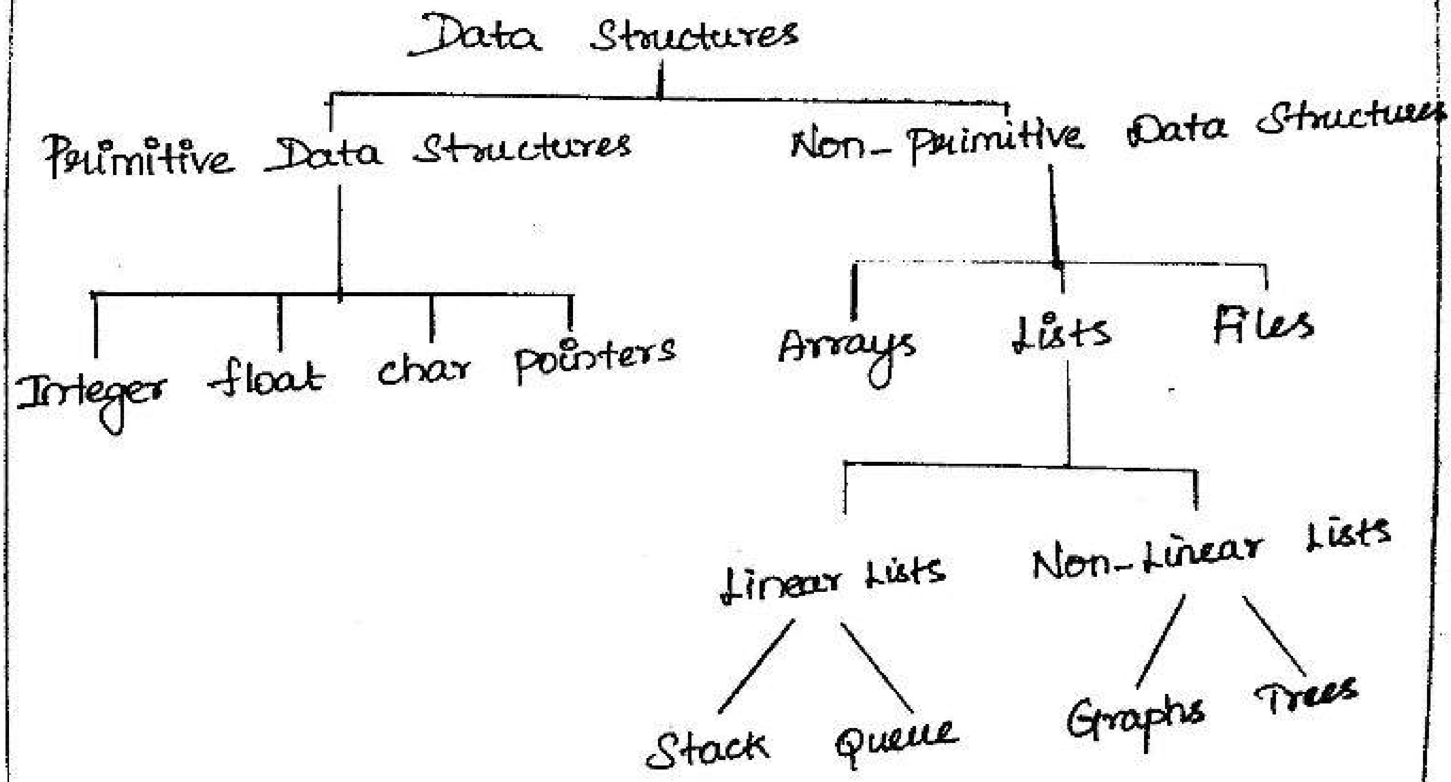
Stack: Stack work on the basis of Last-In-First-out (LIFO). Last entered element removed first.

Queue: Queue work on the basis of First-In-First-out (FIFO). first entered element removed first.

Linked List: It is the Collection of node, here we can insert and delete data is any order.

Tree: Stores data in a non linear form with One root node and sub nodes.

Graph: Graph is a non-linear form of Data structure with vertices and edges.



1) Abstract Data Types (ADTs) : (Nov/Dec - 2018)

An Abstract data type (or) ADT is a mathematical model of a data structure. It represents the implementation of the set of operations by using the modular design.

[N/D - 2019]
[A/m - 2019]

Eg ; Lists, sets and Graphs.

Modular programming is defined as organizing a large program into small, independent program segments called modules.

In C programming each module refers to a function, that is responsible for a single task.

2) List ADT :

List - Definition : A List is a dynamic collection of items (or) elements $E_1, E_2, E_3 \ldots E_n$ of size 'n' arranged in a linear sequence.

A List with size zero (or) with no elements is referred to as an empty List.

List - operations :

The operations that can be carried on a List are.,

Insert (X, L) - Insert an element X in the given List L.

Delete (X, L) - Delete an element X in the given List L.

Find (X, L) - Find the element X in the List L.

Make empty (L) - Delete all the nodes in the List.

Previous (x, L) - Find the Previous element of x in the given List L.

Next (x, L) - Find the next element of x in the given List L.

Implementation of List ADT

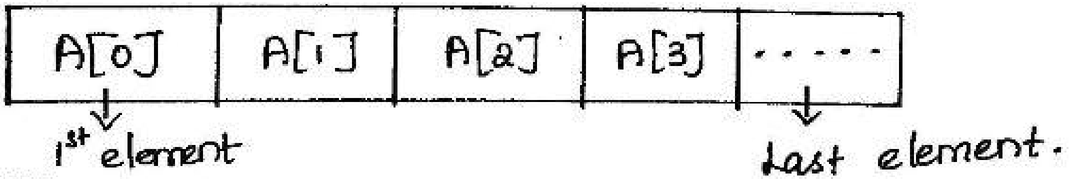
Array Based Implementation   Linked List Implementation

3) Array Based Implementation : (N|D-2018)

* An Array is a data structure which can store a fixed Sequential collection of elements of the same type.

* A specific element in an array is accessed by an Index.

* An array consists of contiguous memory Locations. The lowest address corresponds to the 1st element and the highest address corresponds to the Last element.



List Structure:

Index	Array	position
List [0]		1
List [1]		2
List [2]		3
List [3]		4
List [4]		5

Array Name : List
List size : 5
Start position : 1
End position : 5
Start Index : 0
End Index : 4
1st element referred by : List [0]
5th element referred by : List [4]
ith element referred by : List [i-1]

Operations

(1) Is Empty (List)

(2) Is Full (LIST)

(3) Insert element to end of the LIST

(4) Delete element from end of the LIST

(5) Insert element to front of the LIST

(6) Delete element from front of the LIST

(7) Insert element to n^{th} Position of the LIST

(8) Delete element from n^{th} position of the LIST

(9) Search element is the LIST

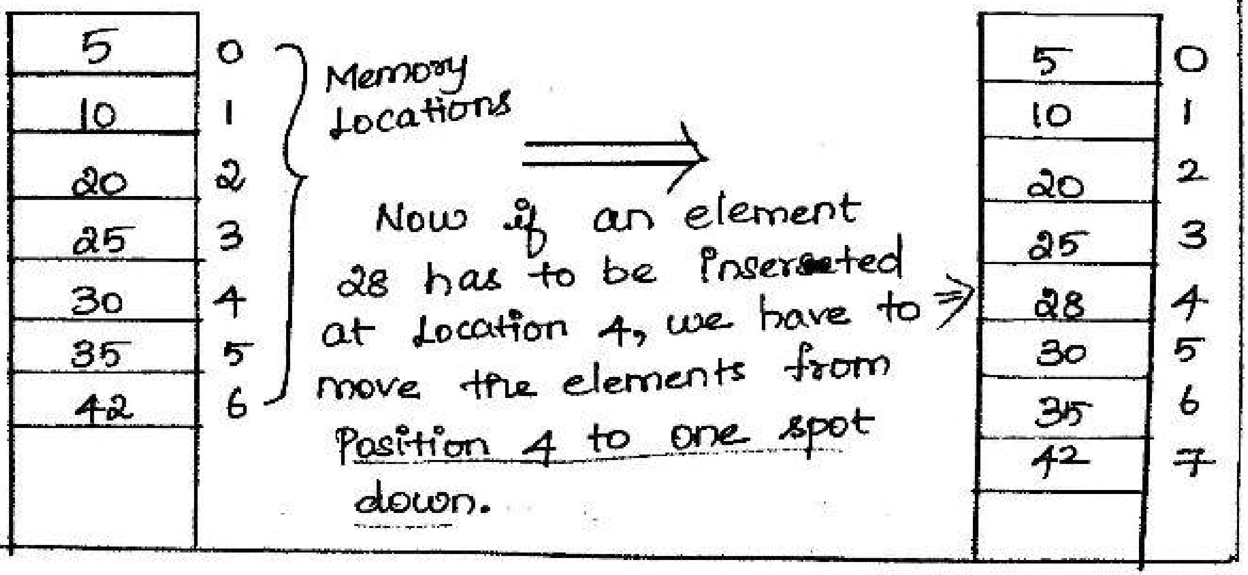
(10) Print the elements is the LIST.

Insertion :

Consider a List of size 'n'. If an element has to be inserted at the position i, then the element after i-1 (ie) i, i+1, i+2 .... requires pushing one spot down to make room. (ie) to make memory space.

Now the element which was previously at location i is now at i+1, the element previously at location i+1 will be at i+2 and so on.,

(Eg)



Now if an element 28 has to be Inserted at Location 4, we have to move the elements from Position 4 to one spot down.

Routine to insert an element into an array:

```
Void insert (elementtype x, int loc, elementtype
                                        a[])
{
    int i;
   if (n= =0)
 {
     a[0] = x;
      n = n+1;
   }
 else if (loc <=n)
{
    for (i=n ; i>= loc ; i--)
   {
         a[i+1] = a[i];
     }
   a[loc] = x;
Printf (" Insertion success");
 }
else     Printf (" Illegal Insertion Location");
}
```

Insert at position '0' requires first pushing the entire array down one spot to make memory space

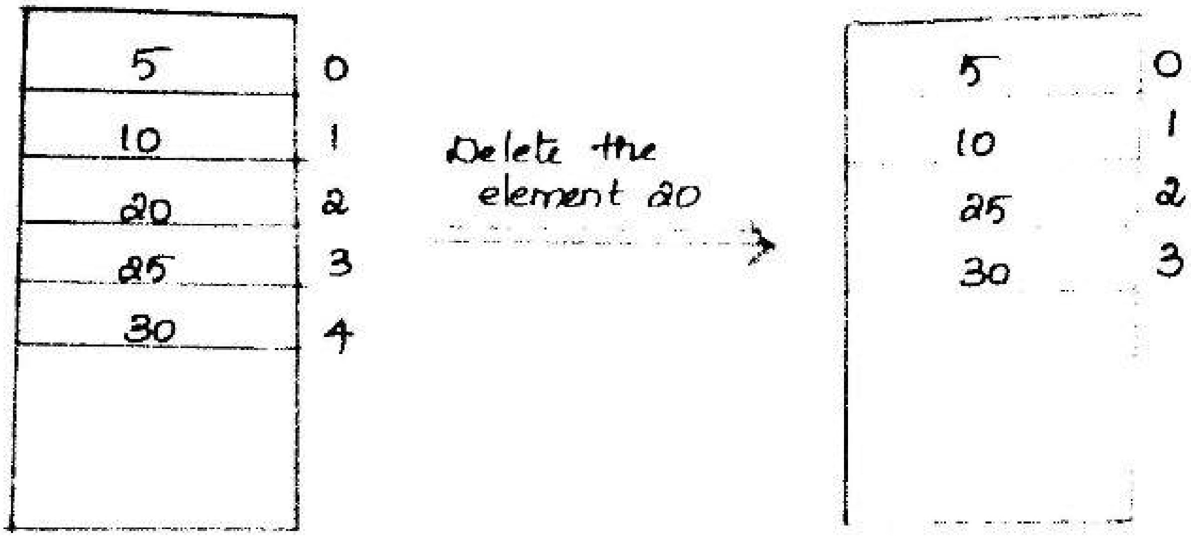
Deletion:

Using Delete operation an element from the list can be removed.

Deletion is done by shifting the elements in the list by one step up.

If the element at position i has to be deleted then the element at position i+1 is moved to position i.

Eg;



Routine to delete an element from an Array:

```
Void delete (elementtype x, elementtype a[])
 {   int i, j;
     for (i=0; i<n && a[i]!=x; i++)
     for (j=i; j<n; j++)
         a[j]= a[j+1];
  }
```

Print :

Print operation is used to display all elements in the list.

Routine to print the elements in an array.

```
  Void  Print (elementtype a[])
 {   int i;
     for (i=0; i<=n; i++)
     Printf("%d", a[i]);
  }
```

Find :

Find operation searches for a particular element in the LIST.

Routine to find the elements in an array.

```
int search (element type a[], element type x)
{   int i ;
    for (i= 0; i<n && a[i] ! =x ; i++)
    Return i;
}
```

Advantages :

- * Data Accessing is faster.
- * Array are easy to understand.

Disadvantages : (N/D - 2018)

- * Array size is fixed.
- * Array elements store continuously.
- * Insertion and deletion of elements in an array is difficult.
- * Unique type data is stored.

(4) Linked List Implementation

A Linked list is a linear data structure, which is a collection of nodes. [N/D-2019]

Each node can hold the data along with pointer field.

A Node (or) Structure Contains two parts

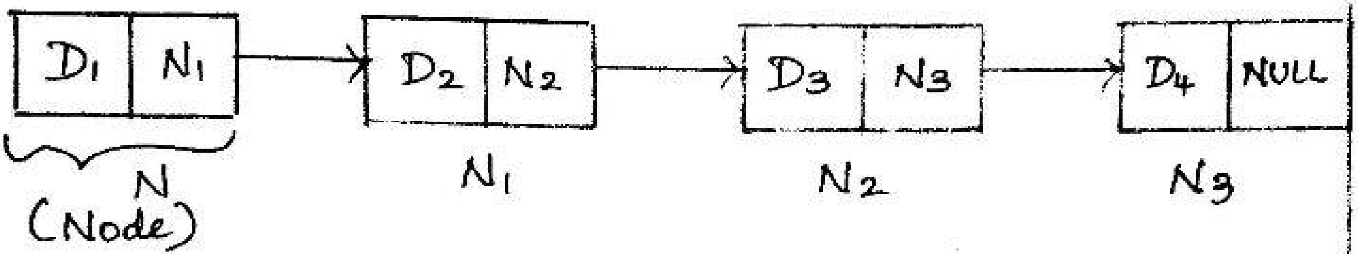
Node

Data field	Next Pointer

* Data field contains the actual element on the list.

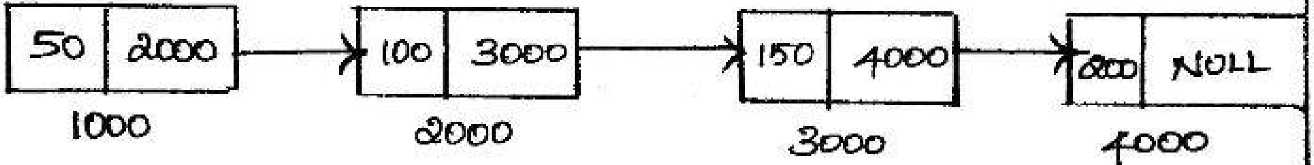
* Next pointer field contains the address of the next node in the list. The address which is used to access a particular node is known as pointer.

Representation of Linked List :



The link field of last node consists of NULL which indicates end of linked list.

Eg ;



The above list contains 4 nodes, which contains the (memory locations) 1000, 2000, 3000, 4000 respectively. The values stored in the data field are 50, 100, 150, 200, the next pointer field contains the address of the next node respectively.

Advantages : [A/m-2019]

* Dynamic Data structure.
* Size is not fixed.
* Data can store non-continuous memory blocks
* Insertion and deletion of nodes are easier and efficient.

Disadvantages :

- Memory allocation is more required.
- Does not support random (or) direct access.

C Representation of a node in linked list.

```
Struct node
{
   int data;
   Struct node *next;
};
```

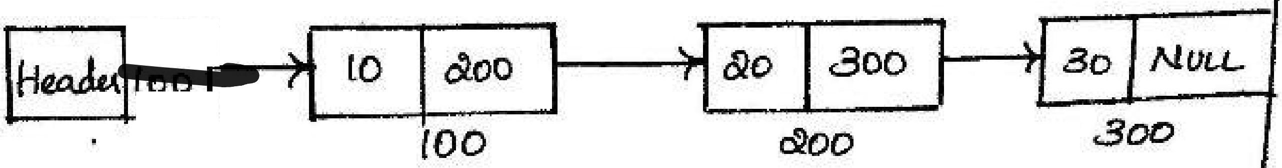
Types of Linked List.

(i) Singly Linked List
(ii) Doubly Linked List
(iii) Circularly singly linked list
(iv) Circularly doubly linked list.

(5) Singly Linked List. [N/D-2019]

Singly linked List is a Linear Data structure. It means Collection of nodes. Every nodes are linked together in some sequential manner and this type of linked list is called Singly linked List. Header points to the first node and the last field of last node is NULL.

Eg.,



Operations on List :

- Insertion
- Deletion
- Find
- IsLast
- IsEmpty.

① **Insertion** : This operation is used to insert a node in the Linked List. [N|D-2019]

List Insertion can be done is three ways.

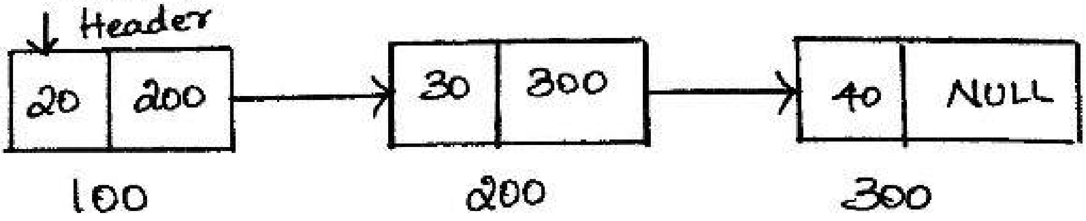
* Insertion in first node.
* Insertion in last node.
* Insertion in Intermediate node.

✓Insert a node in first place : (N|D-2018)

This operation is used to insert a new node into the existing singly linked list at the beginning of the list.

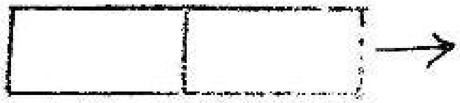
If there is an empty list then the newnode itself is the first node.

Initially the LinkedList contains the following nodes.



* Now we want to create a newnode and this node will be inserted first in the list.

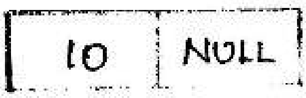
```
newnode = malloc (sizeof (struct node));
```



* This newnode store a data element 10 and the next pointer field will be NULL.

(ie.) newnode → data = 10

newnode → next = NULL



NewNode

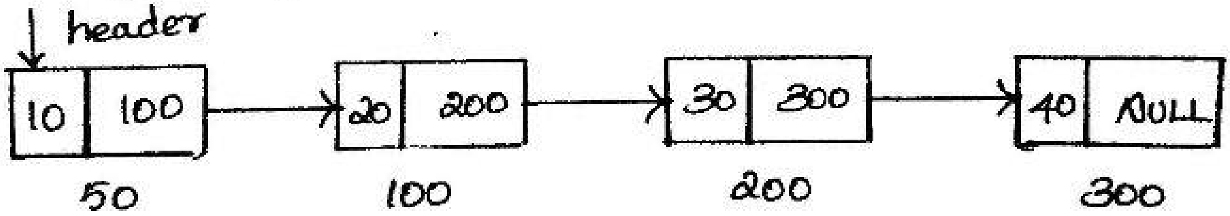
* Attach the link address field of the newnode to point to the starting node of the linked list.

```
if (header == NULL)
        header = newnode
else
        newnode → next = header.
```

* Set the external pointer to point to the new node.

```
header = newnode ;
```

finally we get the Linked List as follows.



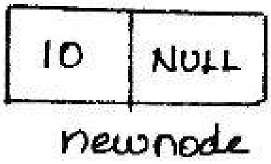
Routine to insert node at Front (first place).

```
Void insert (int x, List L)
{
    Position header ;
  newnode = malloc (size of (struct node));
   newnode → data = x ;
   newnode → next = NULL ;
```

```
if (header == NULL)
  {  header = newnode ;
  }
else   newnode -> next = header;
          header = newnode ;
  }
```

✓ Insert a node at the Rear End (Last place)

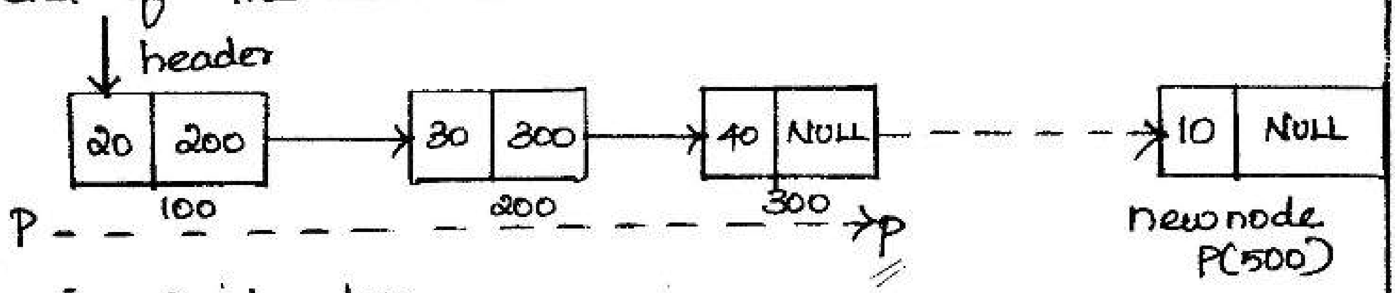
If the first node is null the newnode created.



(ie) newnode → data = x

newnode → next = NULL.

If it is not empty (ie) list contains 3 nodes. Now the pointer p is used to traverse to the end of the list as shown below.

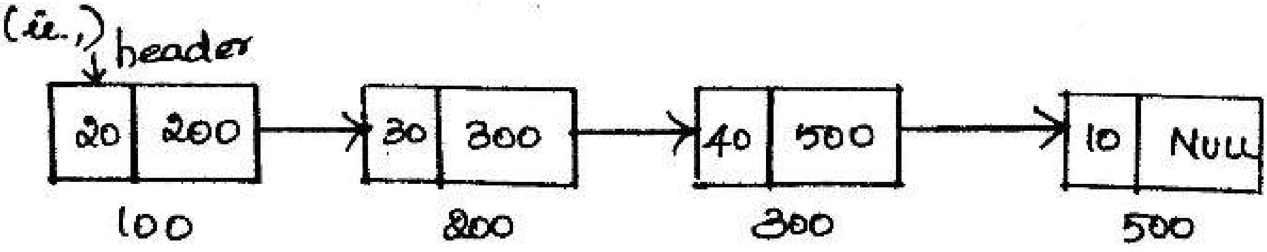


ie., P = header

find the last position.

P → next = newnode.

Now newnode is attached to the list.

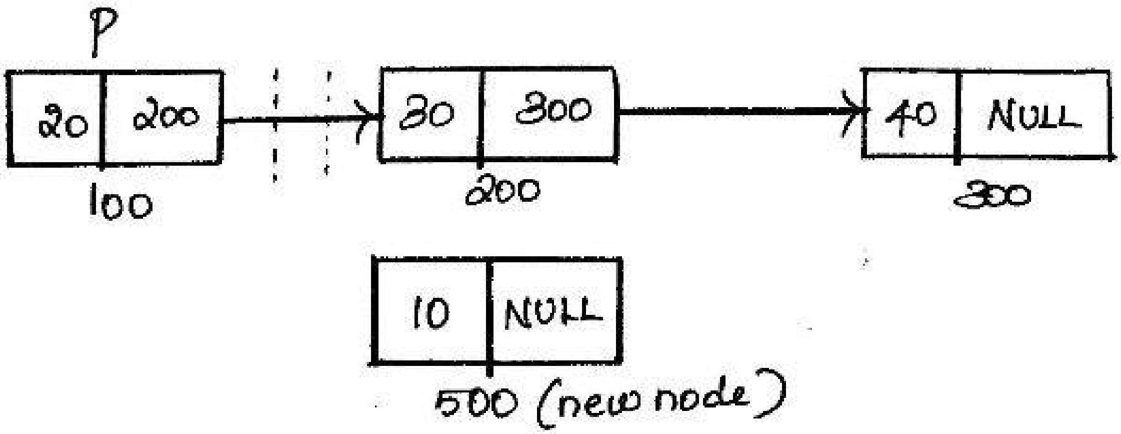


Routine to Insert node at Rear End (Last)

```
Void insert (int x, List L, Position P)
{
    Position header;
  newnode = malloc (size of (struct node));
  newnode -> data = x;
  newnode -> next = NULL;
  if (header = = NULL)
 {
      header = newnode;
    }
   P= header;
  while (P-> next ! = NULL)
  {    P= P->next;
    }  P-> next = newnode
  }
```

Insert a node in an Intermediate Place.

Find the position p where to insert.

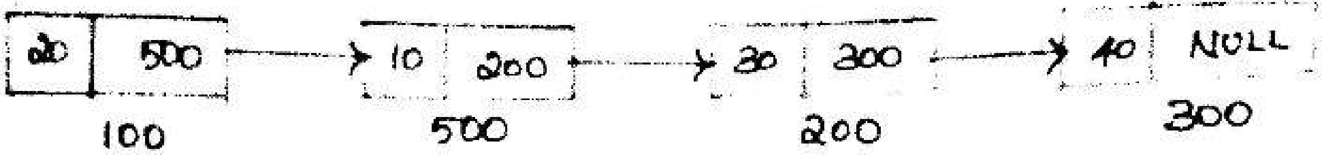


500 (new node)

now, newnode → next = P → next

null = 200

P → next = newnode



Routine to insert a node after the position n

```
void insert ( int x, List L, position n)
{
      Position newnode;
  newnode = malloc (sizeof (struct node));
   if (newnode ! = NULL)
  {
     newnode -> data = x
     newnode -> next = P -> next;
   }
     P -> next = newnode;
  }
```

```
p = header;
for (i=0; i<n; i++)
   p=p->next;
newnode->next=p->next;
p->next=newnode;
```

② find operation.

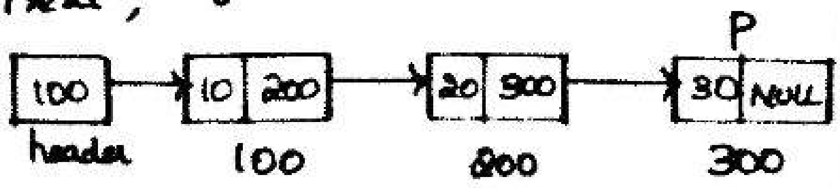
It is used to find the position of a Search element in the list.

(i) Find (ii) Findprevious (iii) Find Next.

Routine - find operation

```
  Position find (int x, List L)
{
     Position P;
      P = L -> next;
   While (P! = NULL && P -> data ! = x)
    {
        P = P -> next;
    }
    return P;
  }
```

Eg; find (30)

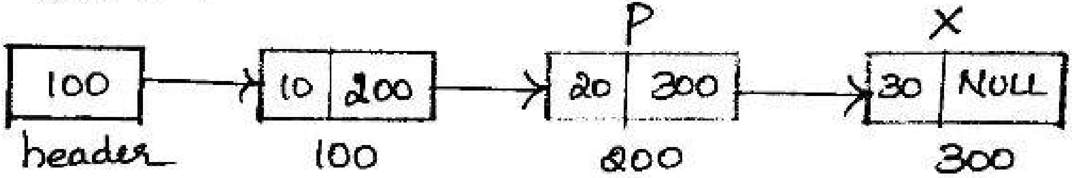


Find Previous : This operation returns the address of the node which is just previous to the data element x.

Routine - find previous :

```
    Position   Findprevious ( int x, List L)
  {   Position  P;
       P = L;
 while ( P-> next ! = NULL  && P-> next -> data ! = x)
  {    P = P-> next;
   }
   return P;
 }
```

find (30)



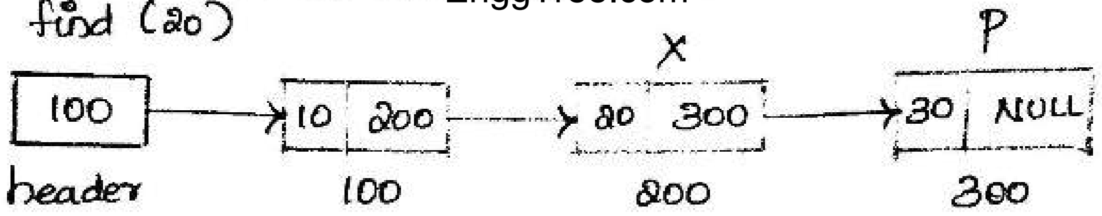
Find next :

This operation returns the address of the node which is next to the dataelement X.

Routine - find Next :

```
    Position   findnext (int X, List L)
  {   Position  P;
       P = L;
     While (P-> next ! = NULL && P-> data ! = x)
      {  P = P-> next;
       }
       return P-> next;
   }
```

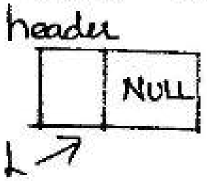
Eg; find (20)



In this list the find function returns the Value 30 as the next element of 30.

③ Check whether the list is Empty:

The empty function is used to check whether the list is empty (or) not.



Routine - empty :

```
 list IsEmpty (List L)
{
  return (L->next == NULL)
 }
```

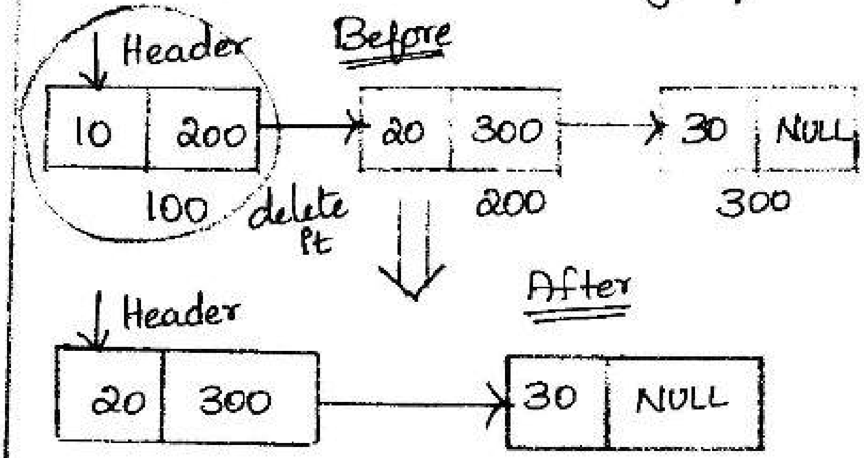
④ Delete Operation:

- * Delete operation removes a node from the list.
- * Delete operation is done by pointing to the Previous node then performing the delete operation.
- * Delete the first node from the list.
- * Delete the lastnode from the list.
- * Delete the intermediate node from the list.

Delete the first node from the list :

* The header pointer is move to the second node of the list.

* Release the memory of first node.



```
temp=header;
header=header->next;
free(temp);
```

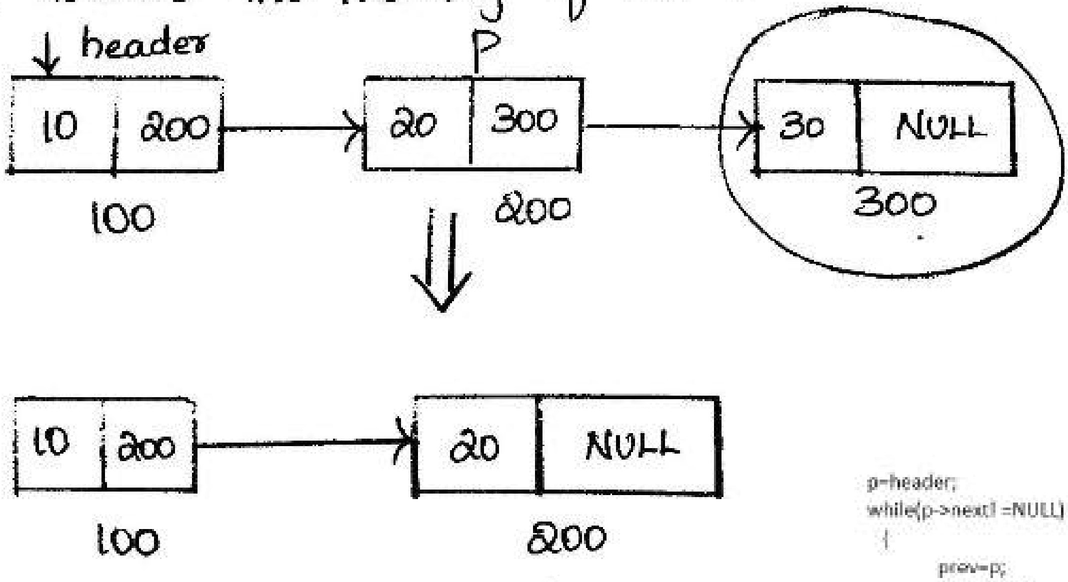
(ie.,) header = header → next

Delete the last node from the list :-

* Find the position of last node.

* Make the address of Predecessor node as NULL.

* Release the memory of the last node.



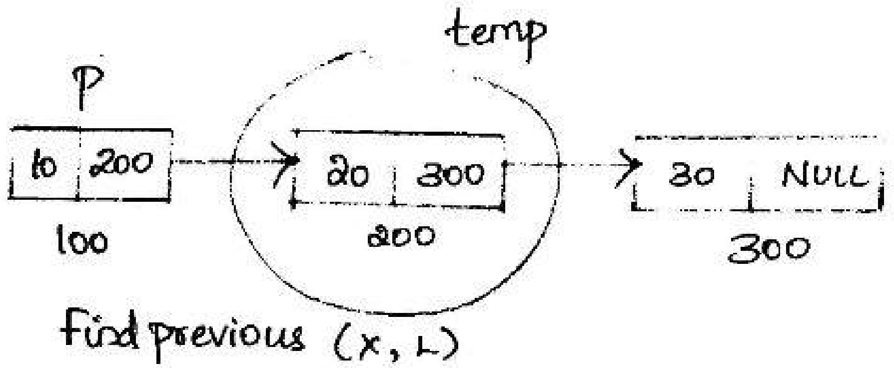
```
p=header;
while(p->next! =NULL)
  {
          prev=p;
          p=p->next;
  }
prev->next=NULL;
free(p);
```

(ie.,) Find~~previous~~position (P)

P → next = NULL.

Delete an intermediate node in the list :

- * Find the Previous position of the deleted node.
- * Make the link field of previous node to Points to the successor of deleted node.
- * Release the memory of deleted node.



```
findprevious (x, L)
P-> next = temp -> next
free temp.
```

Delete a node after position n

```
p = header;
for (i=0; i<n; i++)
  {
        prev=p;
        p=p->next;
   }
prev->next=p->next;
free(p)
```

Routine to delete a node from the list.

```
//find the elements Previous position.
// check the current Position is Last.
int islast (Position P, List L)
{
 //Returns 1 if the position is Last.
      if (P-> next = = NULL)
          return (1);
 }
Void delete (int x, List L)
{  //Deletion of x from List
     Position P, temp ;
```

```
   P = findprevious (x, L);
   if (!isLast (P, L))
  {  temp = P->next;
     P->next = temp->next;
     free (temp);
   }
 }
```

Routine to delete a List.

```
 Void delete list (List L)
 {
     Position P, temp;
     P = L -> next;  //header assumed.
    L -> next = NULL;
   while (P != NULL)
   {   temp = P -> next;
        free (P);
        P = temp;
    }
 }
```

(7) Doubly Linked List.

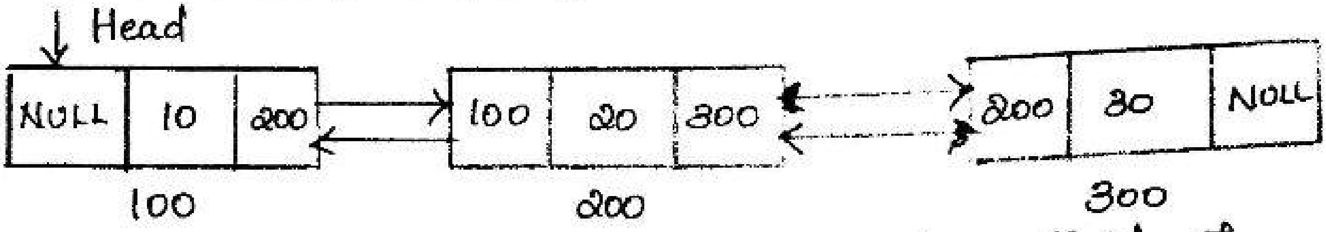
Doubly Linked List is a Linear Data structure. It is a collection of nodes. Each node consists of three fields. They are.,

- Previous Address field
- Data field
- Next Address field.

Prev Add	Data field	Next Add

Node

Representation of Doubly Linked List.



first node is assigned as head. The last field of last node is assigned as NULL, which indicate end of the list.

Operations of Doubly Linked List.

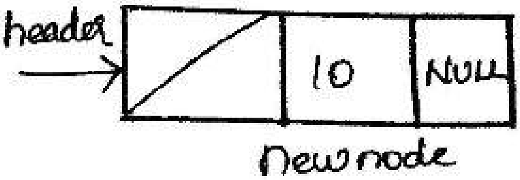
- Creation of Doubly Linked List.
- Insertion
- Deletion
- Display
- Search.

Creating a node for double linked list.

A node in a doubly linked list is declared using the following structure.

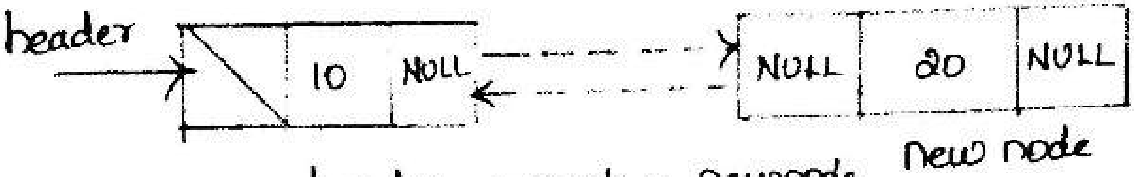
```
Struct dnode
{
    int data;
    Struct dnode *prev;
    Struct dnode *next;
};
```

If there is no nodes in the list, then the list is empty then assign the first node as head node.



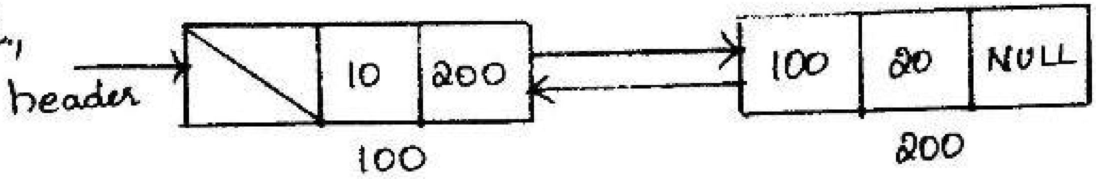
```
newnode → data = x
newnode → next = NULL
newnode → Prev = NULL
  header = newnode.
```

If we want to add one more node in the list, then create a newnode and attach that node to the end of the list.



```
header → next = newnode
newnode → Prev = header
```

ie,



Display of Doubly Linked List.

To display the information we have to traverse the list, node by node from the 1st node, Until the end of the list is reached.

- If list is empty then display Empty list message.
- If the list is not empty, follow the steps given below.

```
temp = header
while (temp! = NULL)
{
    Print temp → data;
     temp = temp → next;
}
```

Counting the number of nodes.

```
int Countnode (node *header)
{   if (header = = NULL)
        return 0;
  else
        return (1+ Countnode (header -> next));
```

Insertion of a node in the list.

Insertion of doubly linked can be done in three possible ways.

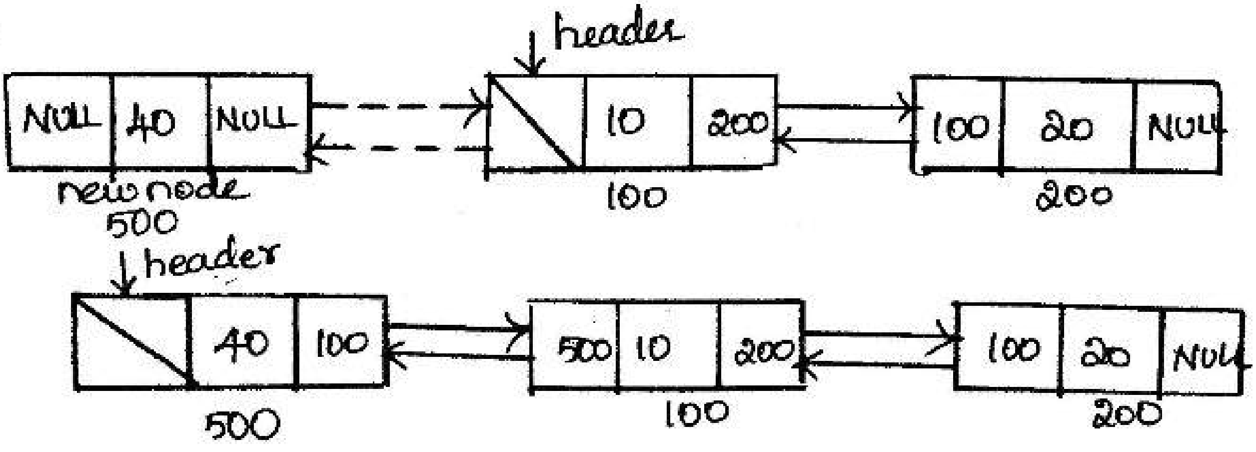
- Insertion in 1st position.
- Insertion in intermediate position.
- Insertion in last position.

Insert a node at the beginning.

- Get the newnode using getnode ()
  newnode = getnode ()
- If the list is empty then, header = newnode
- If the list is not empty, then follow the steps.

```
newnode -> next = header;
  header -> Prev = newnode;
    header = newnode.
```

(eg)



Insert a node at the end:-

Insertion in last position of the list, find the last node of the list, whose next pointer value is null.

* Get the newnode using getnode()

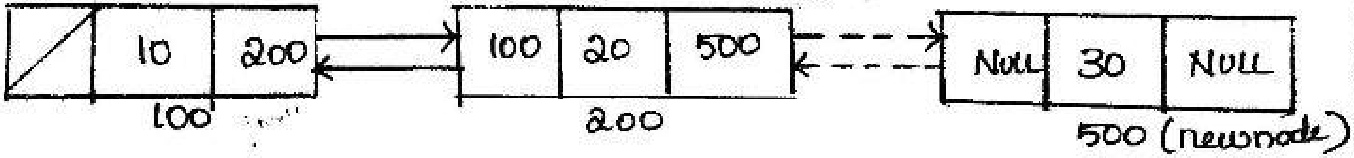
```
newnode = getnode();
```

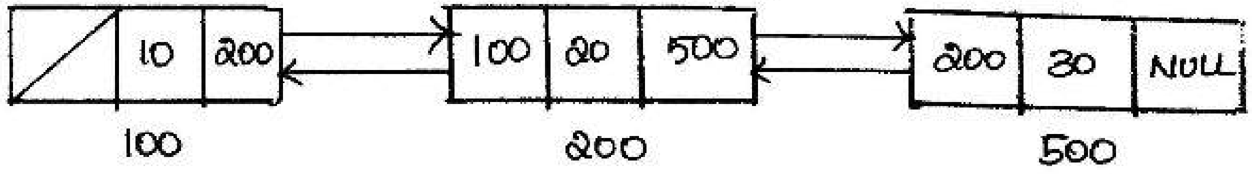
* If the list is empty then

```
header = newnode
```

* If the list is not empty, then follow the steps.

```
temp = header
while (temp -> next != NULL)
    temp = temp -> next;
temp -> next = newnode
newnode -> Prev = temp
```





Insert a node at an Intermediate position.

* Get the newnode using getnode()

```
newnode = getnode()
```

find the Previous position P, where the node to be inserted.

400 → 300

Newnode → next = P → next

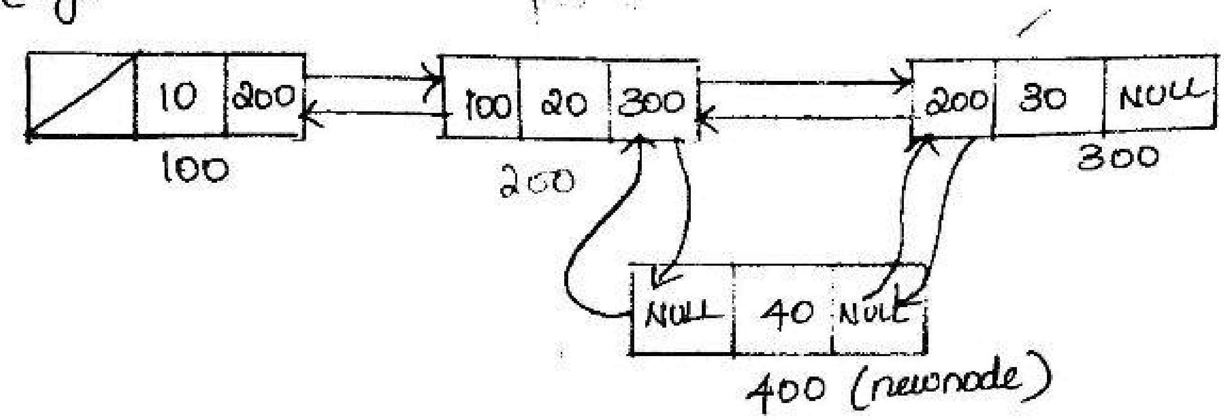
P → next → Prev = newnode

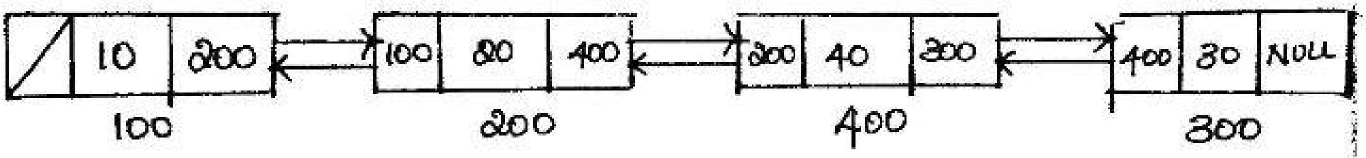
P → next = newnode

newnode → Prev = P.

400 → 200   200

(Eg)





Routine to insert an element in the list.

```
void insert (int x, List L, position P)
{
    Struct dnode *newnode;
    if (newnode != NULL)
   {
        newnode -> data = x;
        newnode -> next = P -> next;
    P -> next -> Prev = newnode;
            P -> next = newnode;
             newnode -> Prev = P;
        }
}
```

Deletion of a node from the List :-

Deletion means releasing the memory allocated for a node in the list. There are 3 possible cases available in deletion operation.

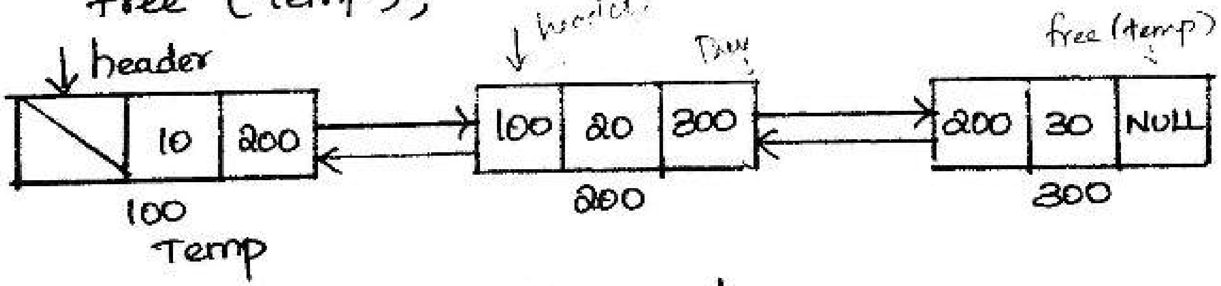
i) Deleting the 1st node.
ii) Deleting the intermediate node.
iii) Deleting the last node.

Deleting the first node:-

* If the list is empty then display "Empty list" message.

* If the list is not empty, follow the steps

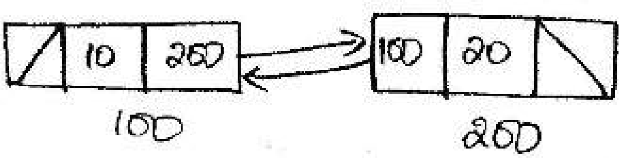
```
temp = header;
header = header -> next;
header -> Prev = NULL;
   free (temp);
```



Deleting a node at the end.

* If the list is empty then display "Emptylist" message.

* If the list is not empty then follow the steps.



```
temp = header
while (temp->next != NULL)
{
    temp = temp->next;
}
    temp->prev->next = NULL;
  free (temp);
```

Deleting a node at Intermediate position :

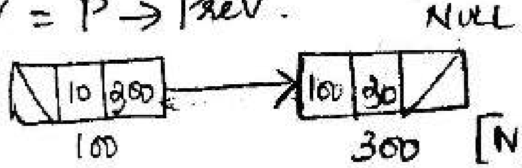
* If the list is empty then display "Empty list" message.

* If the list is not empty, follow the steps.

(i) Get the position of the node to delete.

(ii) Ensure that the specified position is inbetween 1st node and last node.

```
Find position P
          100     300     300
P -> Prev -> next  =  P -> next ✓
          300     100     100
P -> Next -> Prev  =  P -> Prev.
free (P);
      200
```



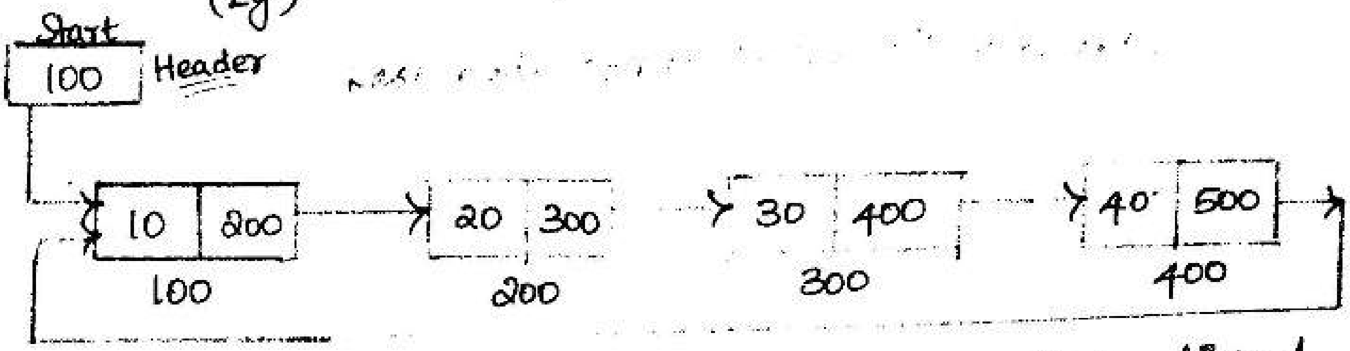
[N/D-18]

(6) Circularly Linked List [A/M-2019]

A circular list is a list in which the link field of the lastnode is made to point to the start / first node of the list. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list.

In circular linked list no NULL pointers are used.

(Eg)



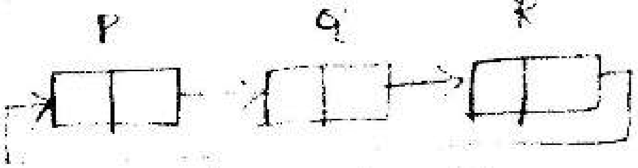
The Basic operations in a circular single linked list are

- Creation
- Insertion
- Deletion
- Traversing.

```
P->data = 20;
temp->data = 10;
[illegible]->next = temp;
temp->next = P;
```

```
Struct node
{
  int data;
  node *next;
};
```

Creating a circular single linked list with 'n' number of nodes.



```
P->next = Q;
Q->next = R;
R->next = P;
```

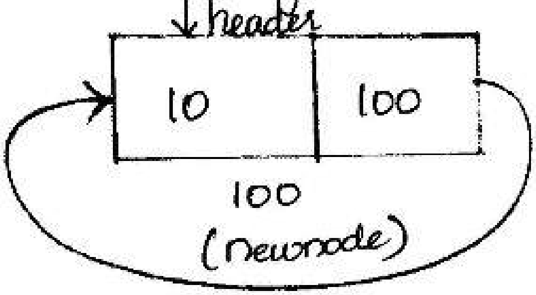
```
Void createlist (int n)
{
   int i;
   node *newnode , *temp;
   newnode = getnode ();
   header = newnode;
// If the list is not empty, follow the steps
      temp = header;
      while (temp->next ! = header)
           temp = temp->next
           temp-> next = newnode;
         Repeat the above steps 'n' times
   }
          Newnode ->next = header;
```

Initially we will allocate memory for newnode using getnode() function.

```
if (x==1)
{ header = newnode ;
  header -> next = header
  x=0;
}
```

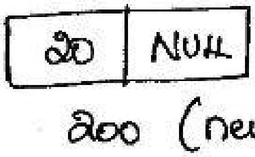
This is used to check whether first node is created (or) not.

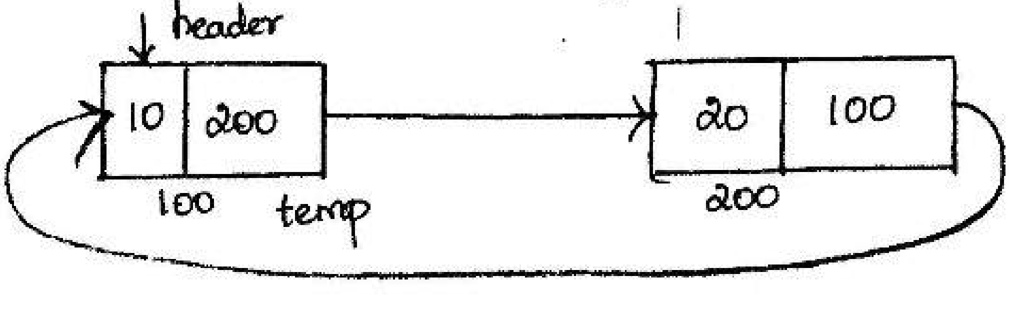
If the x value is 1 then first node is assigned as (header) (or) Start. After create header node then we have to assign zero to x.



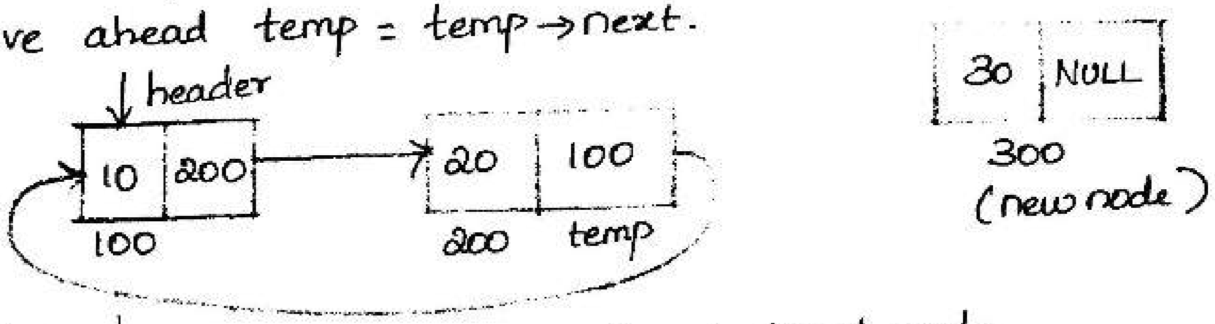
If we want to attach another node to the newnode it will call getnode() function, then assign temp = header, initially it will change as.,

```
|temp -> next = header|
temp -> next = newnode
newnode -> next = header.
```


(ie.,) add ⇒

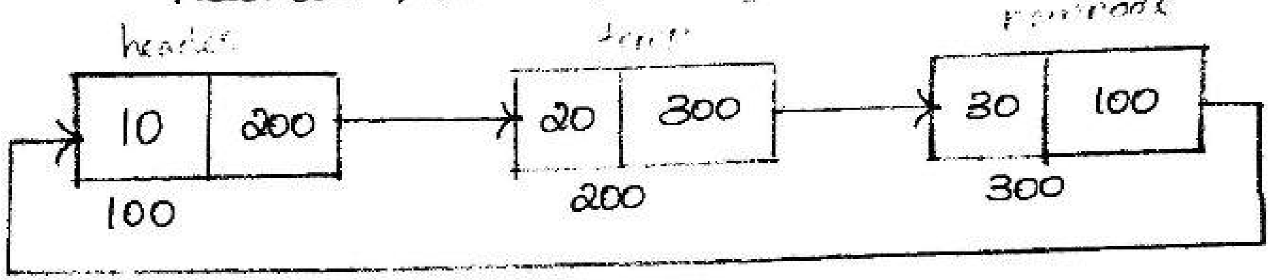


If we want to add one more node then it check temp → next ! = header if true means it move ahead temp = temp → next.



now ; ↳ temp → next is point to head node

[then; temp → next = newnode
newnode → next = header.]



Using the same procedure it can add 'n' number of nodes in the list.

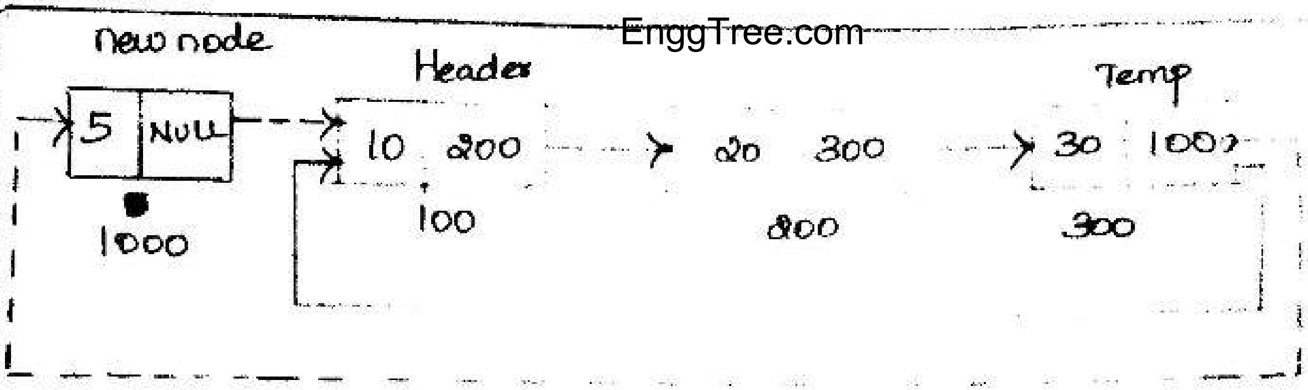
Insertion of circular linked list :-

There are '3' possible cases are available.

- first Insert
- Middle Insert
- Last Insert

first Insert :

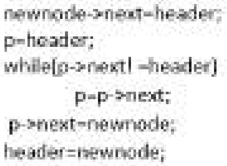
If we want to insert a node at first we will search last node in the list using search () function. Then that node is mentioned as temp.



newnode → next = header

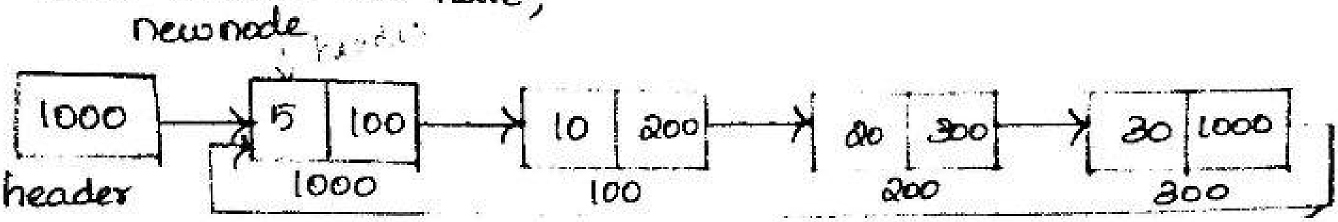
header = newnode

temp→next = header.



```
newnode->next=header;
p=header;
while(p->next! =header)
        p=p->next;
 p->next=newnode;
header=newnode;
```

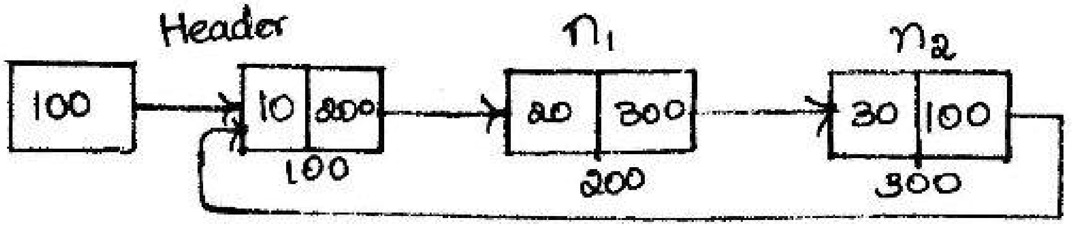
After insert we have;

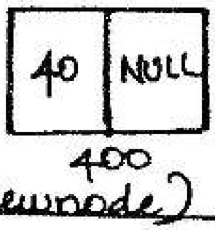


Insert a node at the Last position :-

- * Create a new node
- * Set the newnode's next to itself
- * If the list is empty return newnode.
- * So our new node's next to the front.
- * Set tail's next to our newnode
- * Return the end of the list.

If we want to insert a newnode 40 then.,

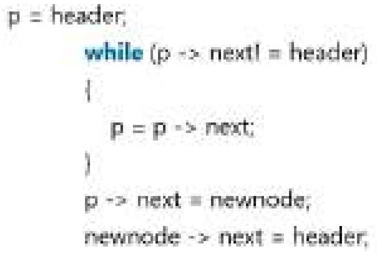




The newnode 40 insert after n_2.

$n_2 \rightarrow$ next = newnode

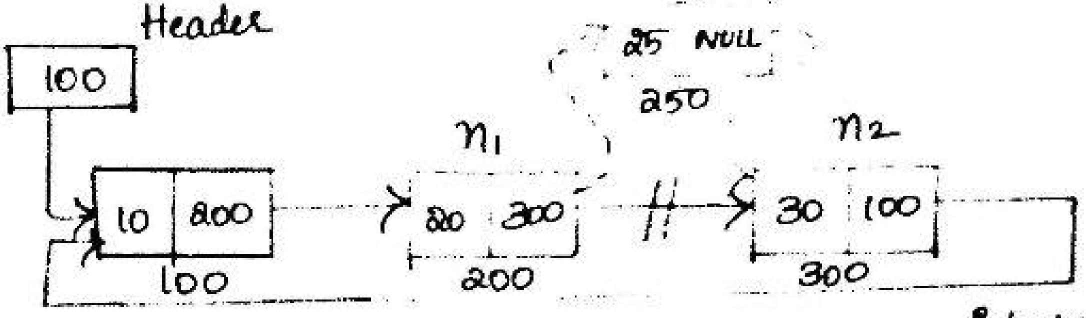
newnode $\rightarrow$ next = header



```
p = header;
while (p -> next! = header)
{
    p = p -> next;
}
p -> next = newnode;
newnode -> next = header;
```

After insert we have;

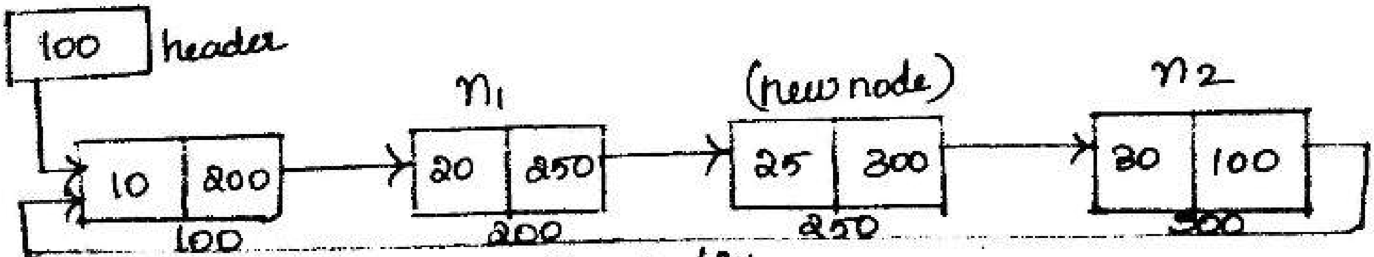
Insert a node at the middle of the circular list.,



If we want to insert a newnode 25 isbetween n_1 and n_2 then;

$n_1 \rightarrow$ next = newnode

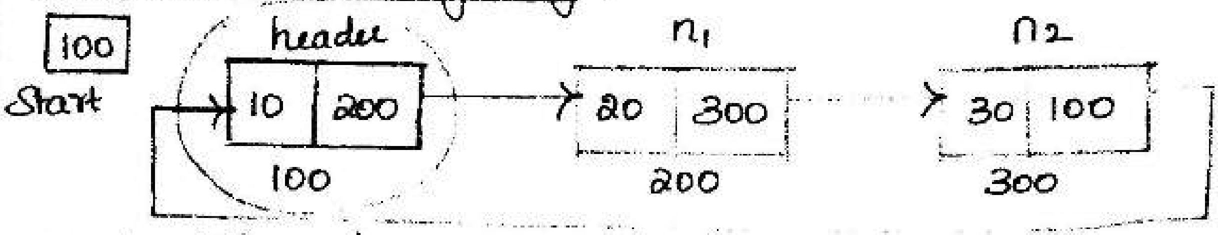
newnode $\rightarrow$ next = n_2.



Deletion in circular linked list.

1) Deletion at beginning of the circular linked list.
2) Deletion at the middle of the circular linked list.
3) Deletion at the end of the circular linked list.

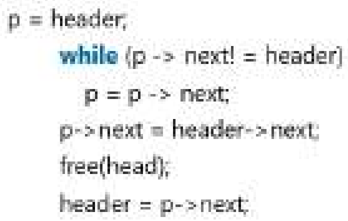
① Delete at Beginning;



Delete this node;

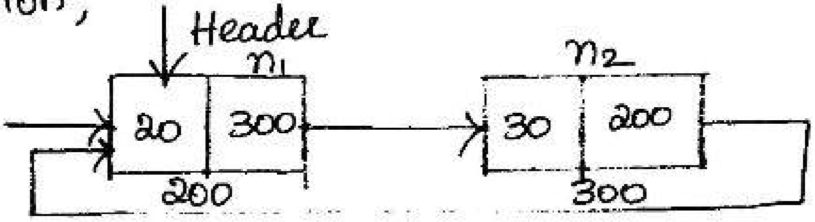
If we want to delete the node

temp = header;
header = header → next
n2 → next = header
free (temp);

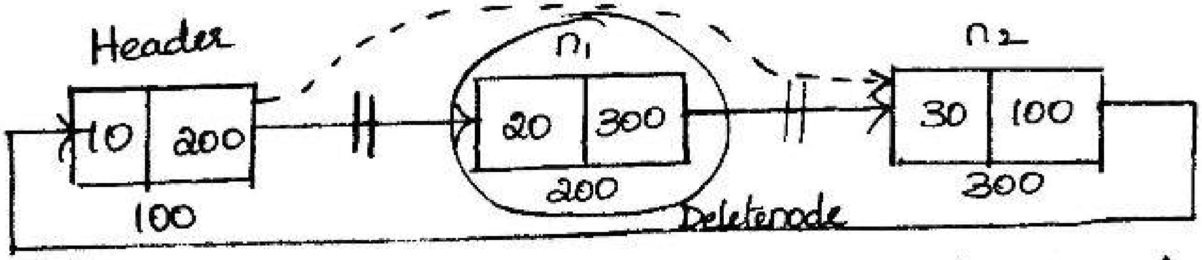


```
p = header;
    while (p -> next! = header)
        p = p -> next;
    p->next = header->next;
    free(head);
    header = p->next;
```

After deletion;



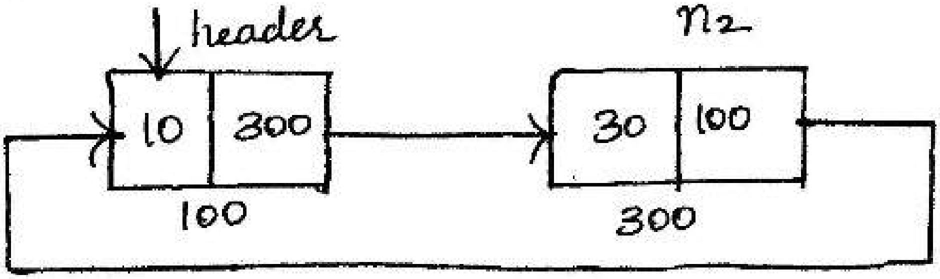
② Delete at middle;



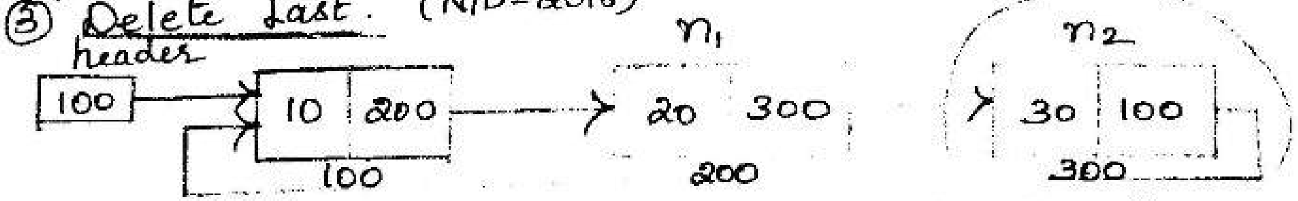
If we want to delete middle node n1 of linked list.

header → next = n2;
free (n1);

After delete we have;

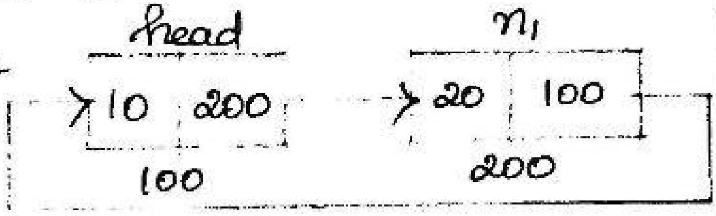


③ Delete Last. (N/D-2018)

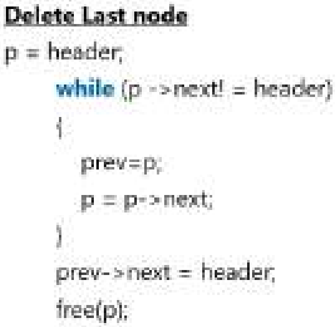


If we want to delete last node n2 then

n1 → next = header
free (n2);



Routine - Delete operation :



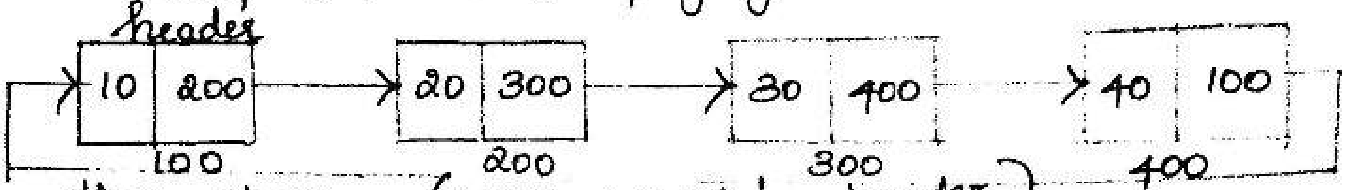
Delete Last node

```
p = header;
      while (p ->next! = header)
      {
         prev=p;
         p = p->next;
      }
      prev->next = header;
      free(p);
```

```
Void delete ()
{
    temp = header;
    if (temp = = NULL)
  {   Printf ("List is Empty \n");
  }
else
{
    int key;
   Printf (" Enter the delete data\n");
   Scanf ("%d", & key);
   temp = search (key);
   if (temp = = NULL)
{
 Printf ("The node is not found \n");
  }
else
  {   Prev = getprev (key);
      Prev → next = temp → next;
       free (temp);
    }
} }
```

Display of circular list :-

The head node is assigned as tempnode. If we have a CLL then the data will be displayed as temp → data // displaying 1 value (ie) 10.



then check (temp → next ! = header)

temp = temp → next.

When it reaches temp = = header; means it displayed all the data in the circular linked list.

Routine - Display.

```
   Void display()
 {
    temp = header;
    if ( temp = = NULL)
  { Printf ("CLL is Empty\n");
  }
 else {
       While (temp -> next != header)
    {  Printf ("%d\n", temp -> data);
       temp = temp -> next;
    }
     Printf ("%d\n", temp -> data);
   }
 }
```

Searching a node from circular linked list :

While searching a node from circular linked list; we go on comparing the data field of each node start from the head node. If the node is containing the desired data is found, then it will display given data is found otherwise return NULL.

```
  void search (int key)
 {
    int found = 0;
     temp = header;
   while (temp -> next ! = header && found = = 0)
  {   if (temp -> data ! = key)
   {    temp = temp -> next;
   }
 else    found = 1;
    }
   if (temp -> data = = key)
  {    found = 1;
      }
    if (found)
  {
    return (temp);
   }
  else
    return (NULL);
  }
```

[N/D - 2019]

(8) APPLICATIONS OF LISTS. [A/M - 2019]

(Polynomial Manipulation) (N/D - 2018)

All operations: Insertion, Deletion, Merge, Traversal.

Linked list is generally used to represent and manipulate polynomials. Polynomials are expressions containing teams with non-zero co-efficients and exponents.

A polynomial is of the form;

$$P(x) = a_0x^n + a_1x^{n-1} + \dots + a_n.$$

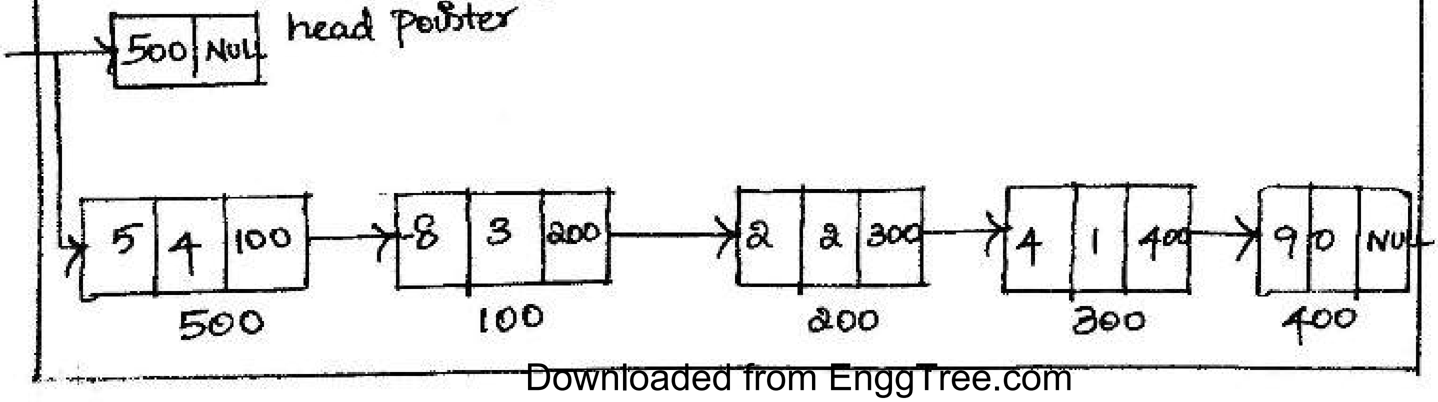
In the linked list representation of polynomials, each term/element in the list is referred as a node. Each node contains three fields namely

- Co-efficient field
- Exponent field
- Link-field.

(address of next node)

Co-efficient	Exponent	Link

The Co-efficient and exponent field stores the data of a polynomial. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x + 9$ shown in the figure.



Declaration for linked ~~List~~ Implementation of Polynomial ADT

```
Struct poly
{
  int coeff;
  int exp;
  Struct poly *next;
};
```

Creation of the polynomial :

```
Poly create (Poly *head 1, Poly *newnode1)
{
     Poly *ptr;
     if (head 1 == NULL)
   {
      head 1 = newnode1;
      return (head 1);
   }
else
   {
     Ptr = head1;
    while (Ptr -> next != NULL)
      Ptr = Ptr -> next;
      Ptr -> next = newnode1;
   }
    return (head 1);
}
```

Addition of two polynomial :

```
Void add ()
{
   Poly *ptr1, *ptr2, *newnode
   Ptr1 = list 1;
   Ptr2 = list 2;
```

Polynomial differentiation

```
Void diff ()
{
 Poly *Ptr1, *newnode;
 Ptr1 = list 1;
while (Ptr1 != NULL)
{
newnode = malloc (sizeof
       (struct poly));
newnode -> coeff = Ptr1 ->
   coeff * Ptr1 -> exp;
newnode -> exp = Ptr1 ->
               exp-1;
newnode -> next = NULL;
list 3 = create (list 3,
          newnode);
 Ptr1 = ptr1 -> next;
}
```

List1: $5x^3$

O/p : $15x^2$ 15 | 2 | → NULL

```
while (Ptr1! = NULL || Ptr2! = NULL)
{
  newnode = malloc (sizeof (struct poly));
  if (Ptr1 -> exp = = ptr2 -> exp)
  {
  newnode -> Coeff = ptr1 -> coeff + ptr2 -> Coeff ;
   newnode -> exp = ptr1 -> exp ;
   newnode -> next = NULL ;
     list 3 = Create ( list 3 , newnode);
      Ptr1 = ptr1 -> next ;
      Ptr2 = ptr2 -> next ;
   }
 else
   {  if (ptr1 -> exp > ptr2 -> exp)
 {  newnode -> coeff = ptr1 -> coeff ;
   newnode -> exp = ptr1 -> exp;
     newnode -> next = NULL;
       list 3 = Create ( list 3 , newnode);
         Ptr1 = Ptr1 -> next ;
   }
 else
 {  new -> Coeff = ptr2 -> Coeff ;
   newnode -> exp = ptr2 -> exp;
     newnode -> next = NULL ;
     list 3 = Create ( list 3 , newnode);
      Ptr2 = ptr2 -> next ;
   } }
}
```

Subtraction of two polynomial :

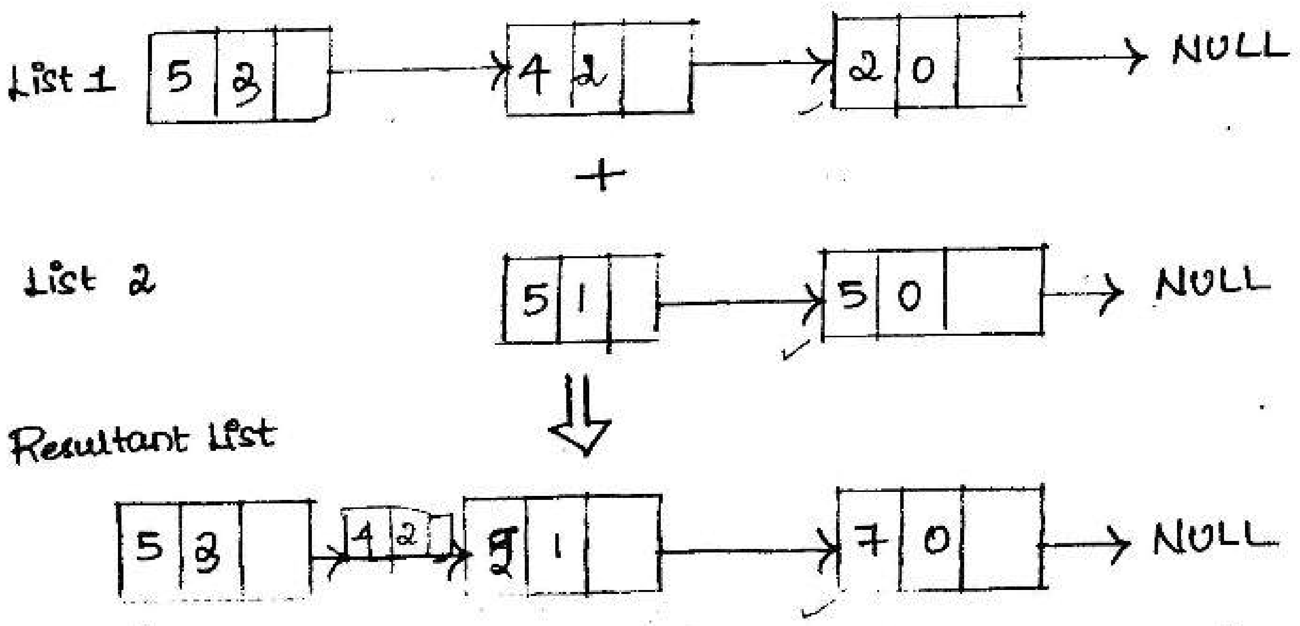
```
Void sub( )
{
    . . . . . .
    . . . . . .
//same as addition of two polynomial
    . . . .
    . . . .
  if ( . . . . . . . )
{
  newnode -> coeff = (Ptr1->coeff) - (Ptr2->coeff);
   . . . .
   . . . .
}
}
```

(Eg) I/P :

1^{st} number : $5x^3 + 4x^2 + 2x^0$

2^{nd} number : $5x^1 + 5x^0$

Output : $5x^3+4x^2+\overset{5}{\cancel{9}}x^1 + 7x^0$



Advantages of Singly linked list

* SLL is dynamic data structure. It means user can able to make change in number of nodes.

* We can access all nodes in forward direction in SLL.

* SLL uses only one pointer variable link so the node of SLL occupied less memory space than nodes of other linked list.

Disadvantages of Singly linked list :

* It is very difficult to access nodes of SLL in backward direction.

* We need to use traversing operation for accessing information from SLL, it is a time consuming process.

* It is very difficult to perform insertion (or) deletion of a node before given location in SLL.

Advantages of Doubly Linked List :

* We can traverse in both direction. i.e., from starting to end and as well as from end to starting.

* It is easy to reverse the linked list.

* If we are at a node, then we can go at any node.

Disadvantages of Doubly Linked List :

* It requires more space for node because extra field is required for pointer to previous node and next node.

* Insertion and deletion takes more time.

Advantages of circular linked list :

* If we are at node, then we can go to any node.

* It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes.

Disadvantages of circular linked list :

* They use more memory than arrays, because of the storage used by their pointers.

* If not traversed carefully, then we could end up in an infinite loop.

* Like singly and doubly linked list, it also doesnt supports direct accessing of elements. [A/M - 2019]

Applications of singly Linked List	Doubly linked list	circular linked list
* Stack	* Represent deck of cards in a game	* Timesharing problem in operating system.
* Queue	* undo functionality in phtoshop (or) word.	* shared printer, Printing process waiting.
* Graphs	* Application that have MRU (Most Recently Used) list.	

UNIT-II

LINEAR DATA STRUCTURES - STACKS, QUEUES.

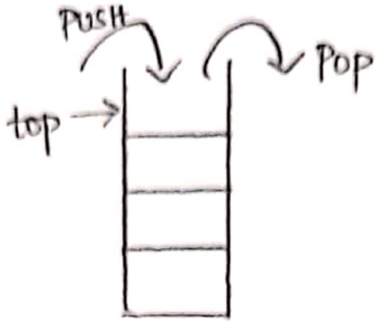
1) Stack ADT
2) operation
3) Applications
4) Evaluating arithmetic expressions
5) Conversion of Infix to postfix expression
6) Queue ADT
7) Operations
8) Circular Queue
9) Priority queue
10) dequeue
11) Applications of Queue.

1) Stack :

A Stack is a non-primitive linear data structure and is an ordered collection of homogeneous data elements.

It is an ordered list in which addition of a new data element (or) deletion of an existing data element is performed at the same end. This end is known as the top of the stack.

The last element inserted will be on top of the stack, since deletion is done from the same end, last element inserted will be the first element to be removed out from the stack. So stack is called Last In First Out (LIFO) data structure.



Stack:

Operations on Stack :

There are two fundamental operations on Stack. They are ;

- * PUSH
- * POP

PUSH / INSERT Operation :

This operation inserts an element alway on top of the stack. Inserting an element into the stack is called push operation.

When we add an item to a stack, we say that we push it on to the stack. For every push operation the top of the stack can be incremented by 1. top = top + 1

DELETE / POP operation :

Deleting an element from the top of the stack is called pop operation. The top of the stack can be decremented by 1 for every pop operation.

top = top - 1

Implementation of Stack :

Array Implementation

Linked list Implementation.

Array Implementation of Stack :

When an array is used to implement a stack, the push and pop operations are realized, by using the operations available on an array.

- The limitation of an array implementation is that the stack cannot grow and shrink dynamically as per the requirement.

Operations :-

Push ( )
Pop ( )
ISFull ( ).
Is Empty ( )

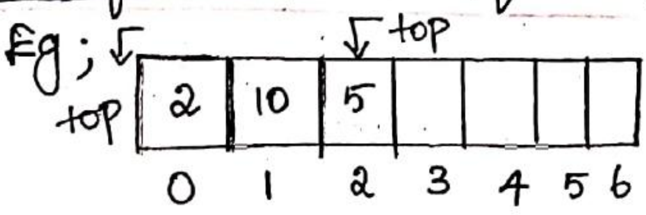
PUSH operation :-

* It adds a new element to the stack.

* Each time a new element is inserted in the stack, the top pointer is incremented, by one.

* When implementing the PUSH operation, Overflow condition of a stack is to be checked.

Eg ;



Push (2)
Push (10)
Push (5)

top = top + 1

Routine to check stack is FULL:

```
int isfull (stack s)
{
        if (TOP = = Arraysize)
            return (1);
  }
```

Routine to PUSH an element on to the stack:

```
  Void Push (int x, Stack s)
{
     if ( isfull (s))
   Error (" full stack");
 else  {  top = top+1;
          S[top] = x;
       }
}
```

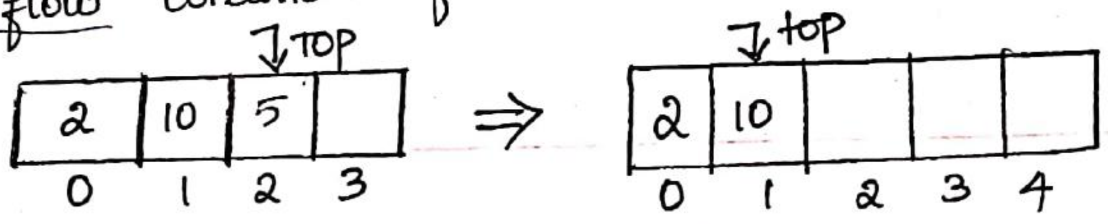
Pop operation:

* A pop operation deletes the topmost element from the stack.

* Each time an element is removed from the stack, the top pointer is decremented by one.

* Underflow condition of a stack is to be checked.

Eg;



Pop(S)

TOP = TOP−1;

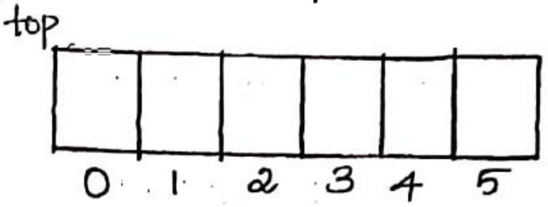
Routine to pop an element from the stack.

```
 void pop (stack s)
{
   if (IsEmpty (s))
       Error (" Empty stack");
else
     {
       x = s [TOP];
       TOP = TOP - 1;
     }
}
```

Is Empty :-

```
* int IsEmpty (stack s)
 {
    if (TOP == -1)
    return (1);
  }.
```

TOP = -1.



Empty Stack:

Routine to Return Top element of the stack.

```
   int Topelement (stack s)
{
    if (! IsEmpty (s))
        return s[TOP];
 else
        Error (" Empty stack");
  }     return 0;
```

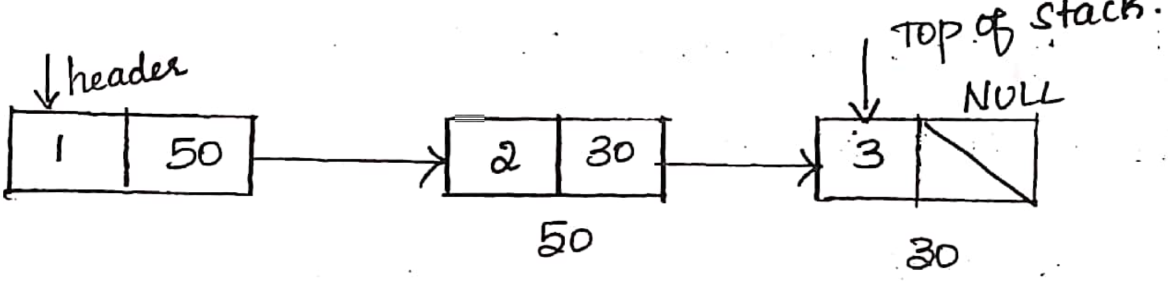
Linked List Implementation of Stack :

The List is a Collection of nodes. Each node Consists of two fields, data and Next pointer. Here we use singly linked list.

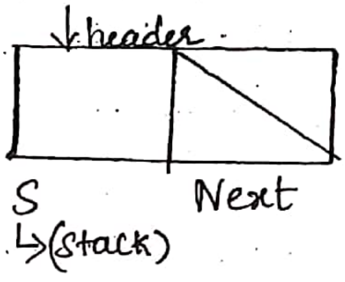
The PUSH & POP operations are done at the same end of the stack (ie) at the top of stack.

Declaration :

```
Struct node
{
    int data ;
    Struct node * next ;
};
```



Routine to test whether a stack is Empty.



```
int IsEmpty (stack s)
{
    return if(S->next = = NULL);
            return (1) ;
}.
```

Creating an Empty stack:

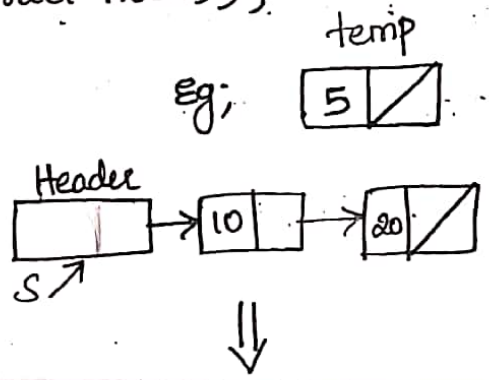
```
Stack createstack ( )
{
    Stack S ;
    S = malloc (sizeof (struct node)) ;
```

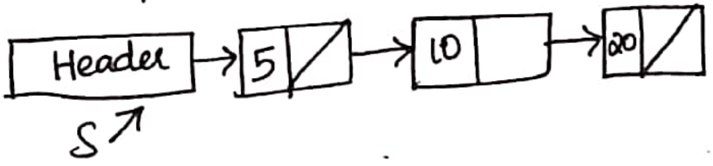
```
  if (S = = NULL)
   error ("out of space");
  makeempty (s);
   return s;
}
 int makeempty (stack s)
{
    if (S = = NULL)
   error ("Must use create stack first");
else
     while (!empty(s))
     Pop(s);
  }
```

Routine to push an Element on to the stack :-

```
 Void push (int x, stack s)
{
   Stack temp;
 temp = malloc (sizeof (struct node));
 if (temp = = NULL)
  error ("Out of space");
 else
  {
    temp -> data = x;
   temp -> next = S -> next;
    S -> next = temp;
  }
}
```

eg;



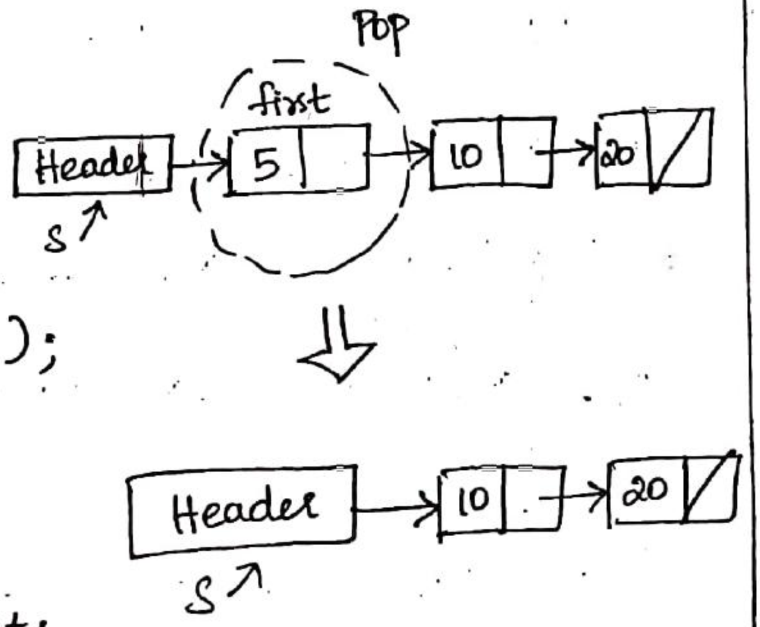


Routine to Return top of the stack:

```
int top (stack s)
{
  if (ISEmpty (s))
  {
  Printf ("Stack is Empty");
  return 0;
  }
else
  return s->next->data;
}
```

Routine to pop an Element from the stack:

```
Void pop (stack s)
{
  Ptr first;
  if ( ISEmpty (s))
Printf ("Stack is Empty\n");
else
{
  first = s->next;
  S->next = S->next->next;
   free (first);
 }
```



(3) Applications of Stack (N/D-2018)

(or)

Evaluating Arithmetic Expressions.

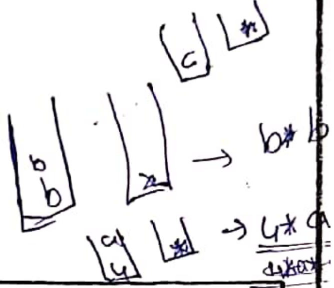
[Conversions of Infix to postfix Expressions]

Applications :-

- * Expression Evaluation
- * Backtracking (Game playing, Finding path)
- * Memory Management.

Evaluating arithmetic Expressions. (N/D-2018)

Eg;



Infix	Prefix	Postfix
a+b	+ab	ab+
a+b*c	+a*bc	abc*+
(a+b) * (c-d)	*+ab-cd	ab+cd-*
b*b-4*a*c	-*bb**4ac	bb*4a*c*-

Infix Notation: Operators are written between the operands they operate on, Eg., 3+4

(b*b)-((4*a)*(c))

Prefix Notation : Operators are written before the Operands. Eg., +34

Postfix Notation : Operators are written after the Operands. Eg; 34+

Conversions of Infix to postfix Expressions.

Steps :-

1) first, we have to take infix expression.

2) We read the expression from left to right.

3) We read this expression one by one and check whether it is operand (or) operator.

4) If it is operand, then we print it and if operator we store it into the stack.

5) In the end, we retrieve operator from the stack and print it.

Eg; Infix Postfix

Infix	Postfix
1) 2+3*4	234*+
a*b+5	ab*5+
(a/(b-c+d))*(c-a)*c	abc-d+/ca-*c*
(a/b)-c+(d*c)-(a*c)	ab/c-dc*ac*-
12+60-23	12 60+23-
5+6/3*(5+6)-7	563/56+*+7-

a bcd-+/ ca- *Cb·

Eg; 1)

Stack	Process	Output
	Now 2 is read and Placed on O/P field	2
+	The operator '+' is read & placed on stack.	2
+	Now 3 is read and Placed on O/P.	23
* +	Now operator '*' is read and placed on stack.	23
* +	Now 4 is read & placed on O/P	234*+

1) (a+b) * c/d + e/f

Step 1:

Input Stack	Process	Output Stack
(	The left paranthesis is encountered, then it is Pushed on to the stack.	
(	The operand 'a' is read, So it is placed on to the Output.	a
+ (	The operator '+' is read, then push it on to the stack.	a
+ (	The operand 'b' is read, so it is placed on to the output.	ab
) + (	The symbol ')' is read, now we pop all the operators from the stack, and place them in output field.	ab+
*	The operator '*' is read and place the stack.	ab+
*	The operand 'c' is read and placed on to the Output.	ab+c

Input	Process	output
/ *	The operator '/' is read and placed on to the stack.	ab+c
/ *	The operand 'd' is read, and Placed on the output.	ab+cd
+ / *	The next char '+' is scanned and it is an operator, so it check precedence inside stack operator which has high priority. (so pop) from the stack. Add Poped operator to the output.	ab+cd/*
+	The next character is 'e' operand, read it & Place it to output.	ab+cd/*e
/ +	The operator '/' is read & Placed on to the output.	ab+cd/*e
/ +	The operand 'f' is read and placed on to the output.	ab+cd/*ef
	Then the operator is poped & placed on to the stack.	ab+cd/*ef/+

Infix : (a+b) * c/d + e/f

Postfix : ~~ab+cd/*ef/+~~

ab+c*d/ef/+

2) (4+8) * (6-5) / ((3-2) * (2+2))

Stack I/P	Process	Stack Output
(	The left parantheis is encountered and then it is pushed on to the stack.	
(	Now 4 is read, so it is placed on to the output.	4
+ (	The operator ~~is~~ '+' is read, then push it on to the stack.	4
+ (	Now 8 is read, so it is placed on to the output.	48
) + (	The symbol ')' is read, now we pop all the operator from the stack, place them in o/p field.	48+
*	The operator * is read and placed on to the stack.	48+
(*	The symbol '(' is read and placed on to the stack.	48+
(*	~~The~~ Now 6 is read, so it is placed on to the output.	48+6
- (*	The operator '-' is read and placed on to the stack.	48+6

Stack I/p	Process	Stack output
- (*	Now 5 is read, and Placed on the output.	48+65
) - (*	The symbol ')' is read, now we pop all the operator from the stack, Place them in o/p field.	48+65-*
/	The operator '/' is read, and Placed on to the stack.	48+65-*
(/	The operator '(' is read, and placed on to the stack.	48+65-*
((/	The operator '(' is read, and placed on to the stack.	48+65-*
((/	Now 3 is read, and Placed on the output	48+65-*3
- ((/	The operator '-' is read, and placed on to the stack.	48+65-*3
- ((/	Now 2 is read, and placed on the output.	48+65-*32
) - ((/	The symbol ')' is read, now we pop all the operator from the stack, Place them in o/p.	48+65-*32-
* (/	The operator '*' is read, and placed on to the stack.	48+65-*32-
(* (/	The symbol '(' is read, and placed on to the stack.	48+65-*32-

Routine to convert Infix to postfix Expression.

```
Stack S
 Char ch
 Char element
While (Tokens are Available)
{
    Ch = Read (Token);
  if (ch is operand)
  {
     Print ch;
  }
 else
 {  while (Priority (ch) < = Priority (Top most stack))
    {
       element = Pop (s);
       Print (ele);
    }
 } Push (s, ch);
}
 While (! Empty (s))
{
   element = Pop(s);
   Print (ele);
}
```

Infix : (4+8)*(6-5)/((3-2)*(2+2))

Postfix: 48+65*32-22+*/

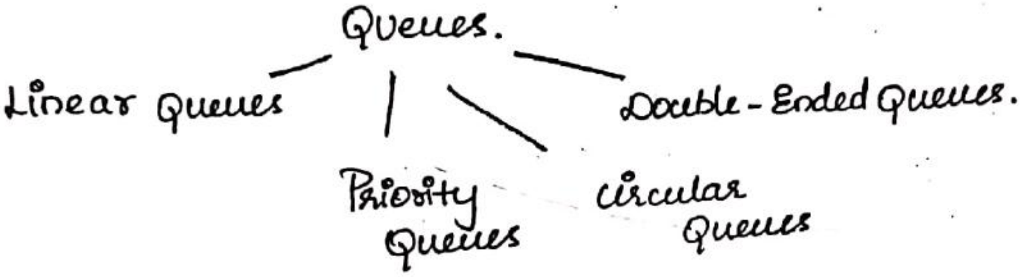
48+6*5-3/2*2+

(* (/	Now 2 is read, and Placed on the o/p field.	48+65-*32-2
+ (* (/	Now operator '+' is read and Placed on to the stack.	48+65-*32-2
+ (* (/	Now 2 is read, and Placed on the o/p field.	48+65-*32-22
) + (* (/	Now symbol ')' is read and we pop all the operator.	48+65-*32-22+
) * (	The symbol ')' is read.	48+65*32-22+*/

6) QUEUE ADT

A Queue is an ordered collection of elements in which insertion are made at one end is referred to as the REAR end, and the end from which deletions are made is referred to as the front end.

In a Queue the first element inserted will be deleted first. So a queue is referred as FIRST-IN-FIRST OUT (FIFO) lists.



Implementation of queues:-

* Array Implementation. (single dimensional array)

* Linked List Implementation.

Operations of Queue:-

1) Enqueue

2) Dequeue.

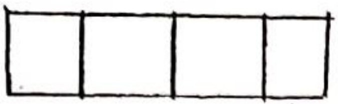
* Array Implementation of Queue:-

Enqueue Operation:

It is used to add a new element into a queue, at the rear end.

When implementing the enqueue operation overflow condition of the queue is to be checked.

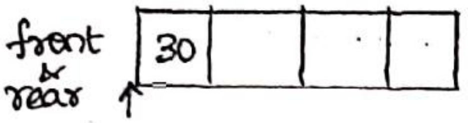
Assign the new element in the array by incrementing the rear.



Empty Queue
Array size is 4

rear = -1
front = 0.

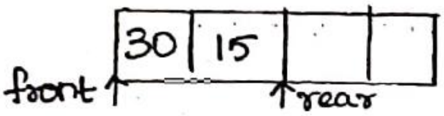
Insert an element 30 in the queue.



front = 0
rear = 0

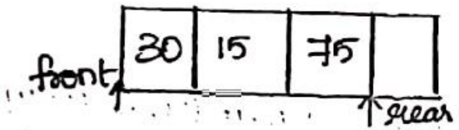
rear = rear + 1

Insert an element 15 in the queue.



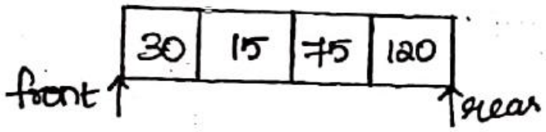
front = 0
rear = 1

Insert an element 75 in the queue.



front = 0
rear = 2

Insert an element 120 in the queue.



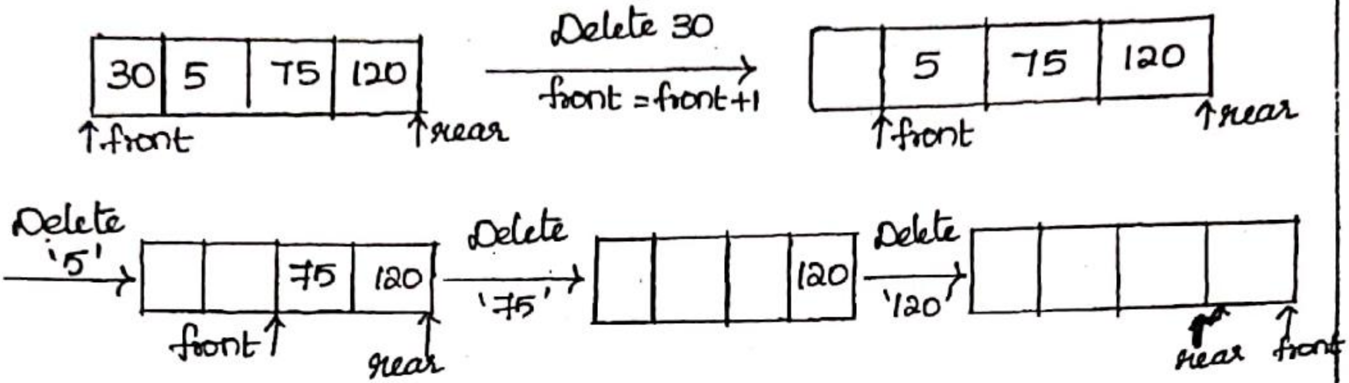
front = 0
rear = 3.

Routine to Enqueue Operation:-

```
rear = -1;
front = 0;
Void enqueue (int num)
{
 if (rear < arrsize-1)
 {
    rear = rear + 1
    Queue [rear] = num;
 }
 else
    Printf ("queue is full");
}
```

Dequeue Operation:

It is used to delete an element from the front end of the queue. When implementing the dequeue operation underflow condition of a queue, is to be checked. After the deletion increment the front variable by 1.



Routine to Dequeue Operation:

```
  Void   delete ()
{
   if (rear = =front -1)
  Printf ("queue is empty");
else {
    Printf ("Deleted element %d", Queue (front));
   } front = front+1;
 }
```

* Linked List Implementation of Queue.

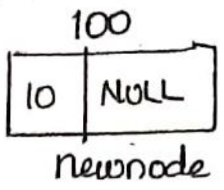
A Queue can be implements by singly linked list.

Enqueue Operation:

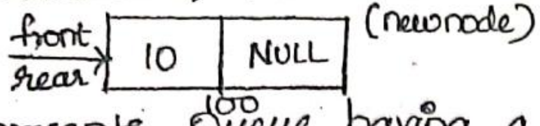
It is used to add a new element into a queue.

1) Allocate the memory for the newnode.
2) Assign the value for datapart of the newnode.
3) Assign the link of the rear to the newnode.
4) Assign the rear to the newnode.

rear = 0 }
front = 0 } It indicates Queue is empty.

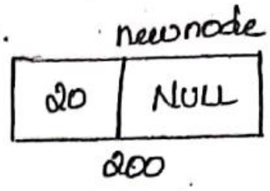


front = newnode ; rear = newnode.



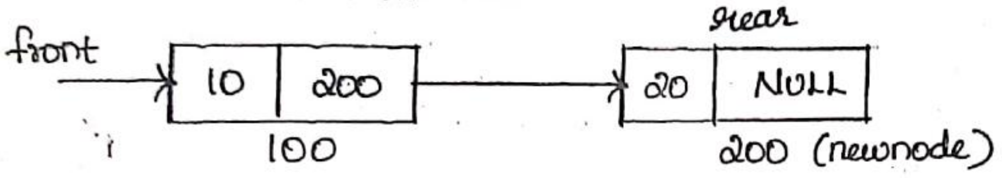
It represents Queue having a single node.

Insert a newnode to the existing node.



rear -> next = newnode

rear = newnode.



Routine to Enqueue an element in queue.

```
Void enqueue (int x)
{
  if (rear = =NULL)
{
  rear = (struct node *)malloc (sizeof (struct node));
 rear ->next = NULL;
  rear -> data = x;
} front = rear;
  else
{
 temp = (Struct node *)malloc (sizeof (struct node));
 rear ->next = temp;
 temp -> data = x;
 temp -> next = NULL;
  rear = temp
 }
}
```

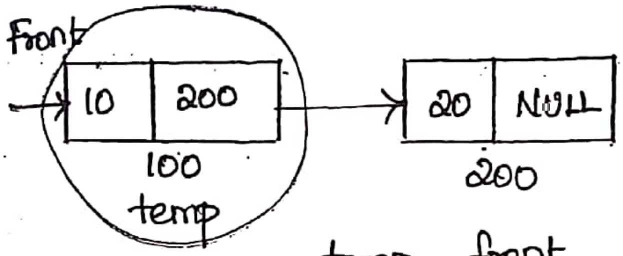
Dequeue operation:

It is an operation used to remove an element from the front of the queue.

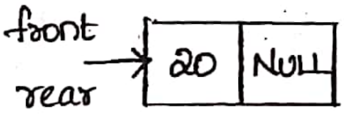
1) Assign the front pointer to the temp pointer.
2) The front pointer is made to point after the first node, and the other nodes remain unchanged.
3) Free the allocated memory of the temp pointer.

Routine - Dequeue an element in Queue.

```
Void dequeue ()
{
  temp = front;
  if (temp = = NULL)
 Printf ("Queue is empty");
else if (temp->next! = NULL)
{ temp = temp->next
Printf ("Dequeue value %d",
              front->data");
  free (front);
  front = temp;
}
else
{ Printf ("Dequeue value : %d",
              front->data)
  free (front);
  front = NULL;
} rear = NULL;
}
```



temp = front
front = front -> next



CIRCULAR QUEUES. (RING BUFFER) (N/D-2018)

Circular queue is another form of a linear queue in which the last position is connected to the first Position of the list. Initially the front and rear ends are at the same position. When we insert an element the rear pointer moves one by one until the front end is reached. If the next position of the rear is front, the Queue is said to be fully occupied. Beyond this we cant add (Insert) any data. When we delete the elements the front pointer moves one by one, until the rear pointer is reached.

Array Implementation of Circular Queue.

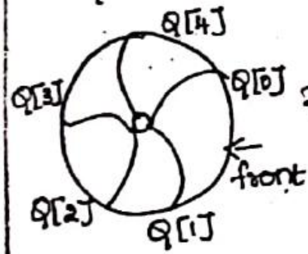
To insert an element to the queue, the position of the element is calculated as.,

rear = (rear + 1) % maxsize
Queue (rear) = value.

To perform the deletion the position of front is calculated as front = (front + 1) % arraysize.

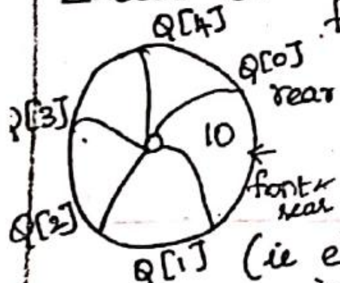
10 20 30 40
20 70 90
30 10
40

Enqueue.



front = 0
Rear = -1
Queue is empty

Insert 10:

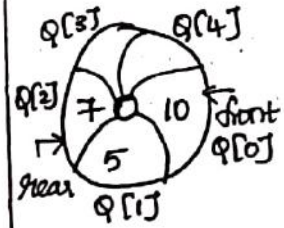


front = 0, max size is 5.
rear = (rear+1) % maxsize
= (-1 + 1) % 5
= 0
(ie element 10 is inserted in 0th position)

Insert 5:

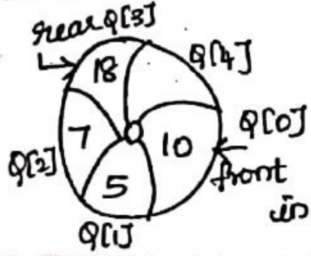
front = 0
rear = (0+1) % 5
= 1
(ie, element 5 is inserted in 1st position)

Insert 7:



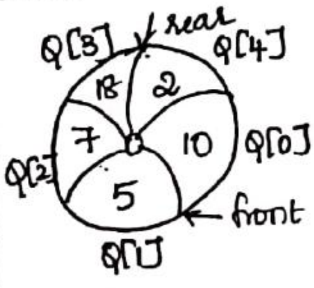
front = 0
rear = (1+1) % 5
= 2
element 7 is inserted in 2nd position.

Insert 18:



front = 0
rear = (2+1) % 5
= 3.
element 18 is inserted in 3rd position

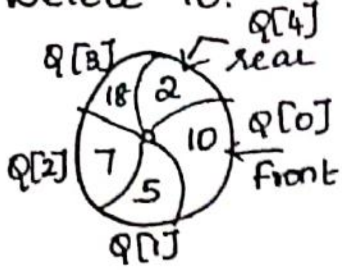
Insert 2:



front = 0
rear (3+1) % 5
= 4
ie., element 2 is inserted in 4th position.

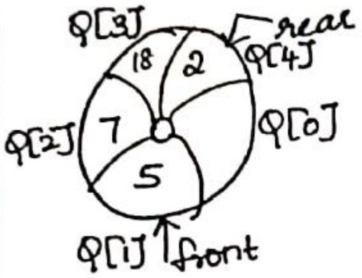
Dequeue:

Delete 10.



front = 0
rear = (0+1) % 5
= 1.

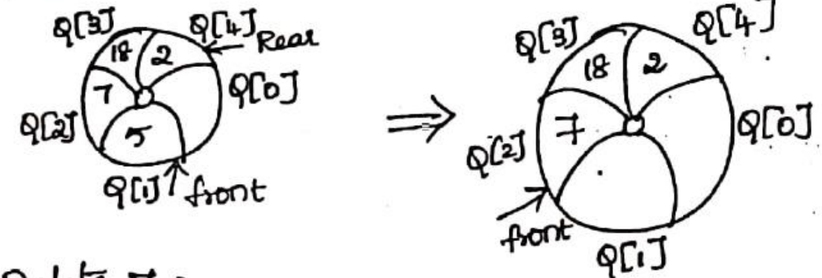
The element which is pointed by the front pointer is dequeued and it is moved to first position.



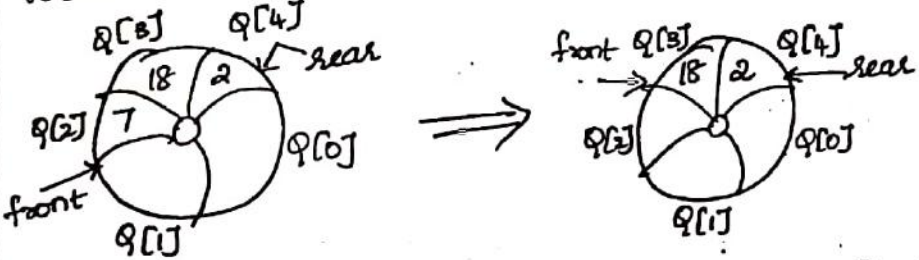
If we want to delete another element in the queue, then first dequeue the element and then move the front Pointer to next position.

ie., front = (1+1) % 5
= 2.

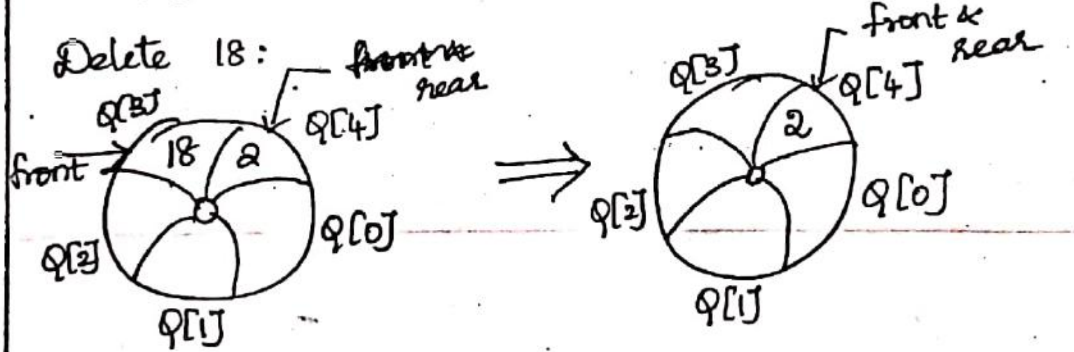
Delete 5:



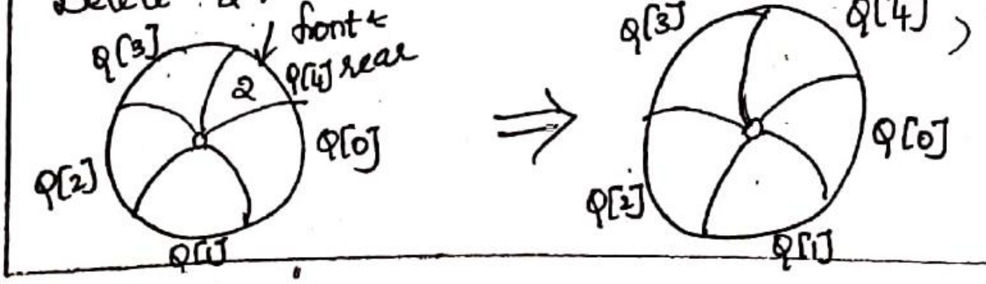
Delete 7:



Delete 18:



Delete 2:



rear = −1
front = 0
(queue is empty)

Routine for Enqueue operation in circular Queue.

```
front = 0
rear = -1
void enqueue ()
{
   rear = (rear+1) mod maxsize.
   if (rear == maxsize)
        Print queue is full.
   else
        Queue [rear] = value;
   }
```

Routine - Dequeue operation in circular queue.

```
void   dequeue ( )
{
     if ( rear == -1)
     Print queue is empty
else
     { Print ("Deleted element   Q[front]");
     if ( front == rear)
   {
     front = 0;
     rear = -1;
   } else  front = (front+1) % maxsize;
}  }
```

Double Ended Queue (Dqueue)

A dqueue is a special type of data structure, is a homogenous list of elements in which insertion and deletion operations are performed from both the ends.

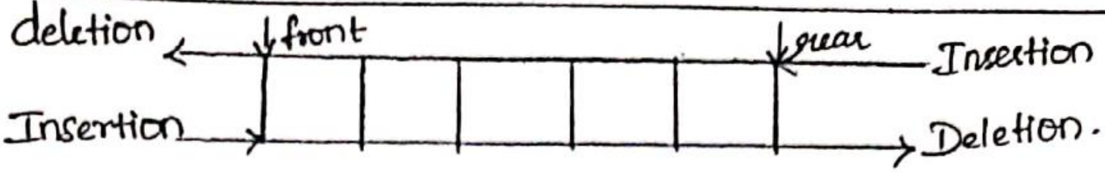
The 4 operations performed on dqueue are.,

① Insert an element from front end. insertfront()

② Insert an element from rear end. insertlast()

③ Delete an element from front end. deletefront()

④ Delete an element from rear end. deletelast()



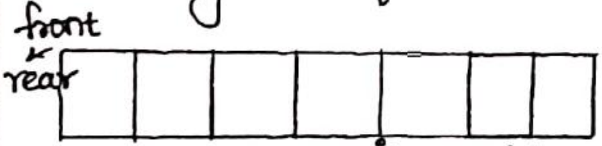
Representation of Dqueue.

Dqueue — I/p restricted dqueue
\ O/p restricted dqueue.

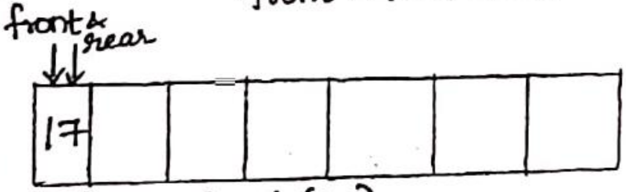
The I/p restricted dqueue allow insertions at one end and deletions from two ends.

The O/p restricted dqueue allow deletions from one end and insertions from two ends.

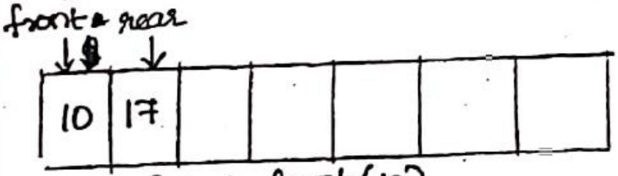
Building a dequeue.



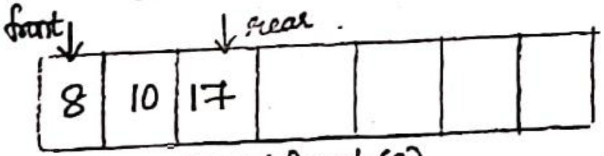
Dequeue is empty.
front = rear = -1.



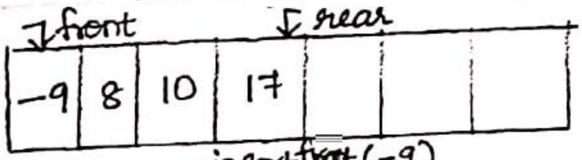
insert front (17)



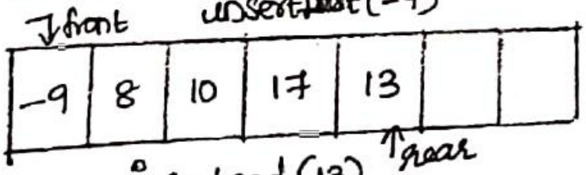
insert front (10)



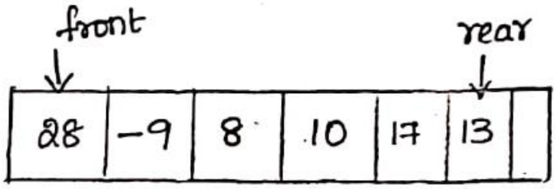
insertfront (8)



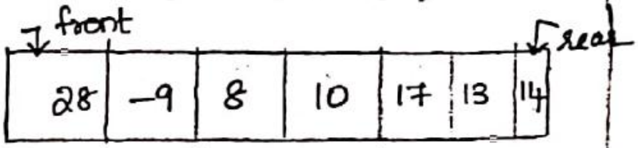
insertfront (-9)



insertend (13)

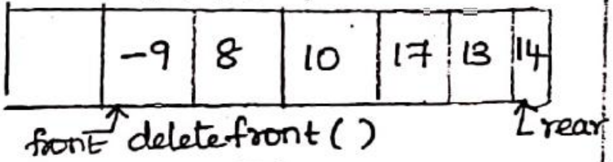


insert front (28)

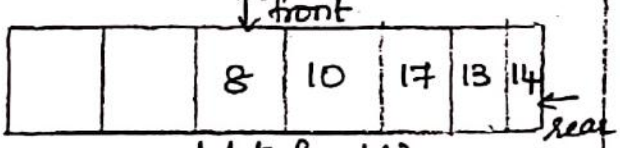


insertend (14)

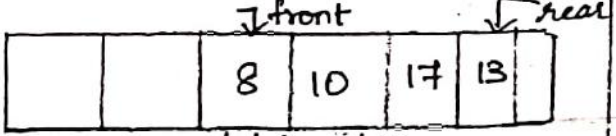
insertend (21) → full, insertfront (6) → full.



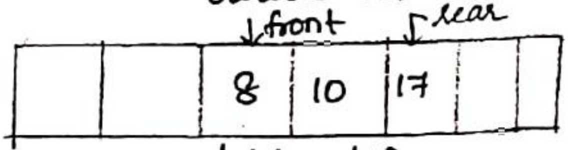
deletefront ( )



deletefront ( )



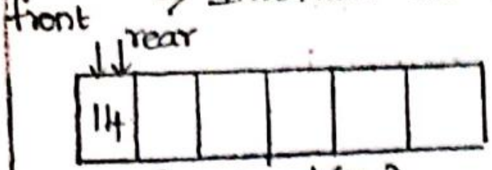
deleteend ( )



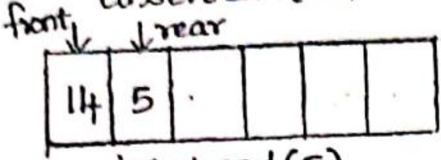
deleteend ( )

Building an I/p restricted Dequeue.

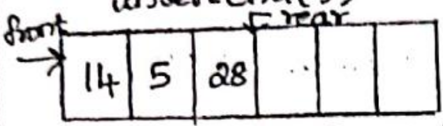
⇒ Insertion at one end and deletions from two ends.



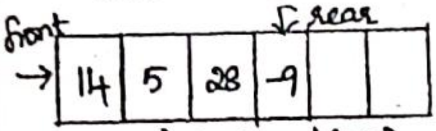
insertend(14)



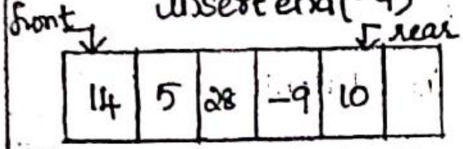
insertend(5)



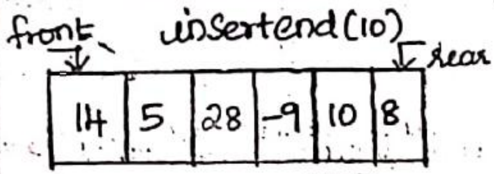
insertend(28)



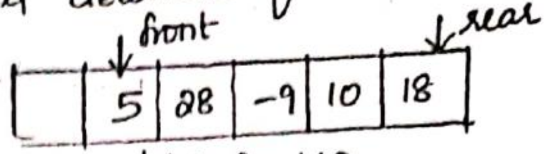
insertend(-9)



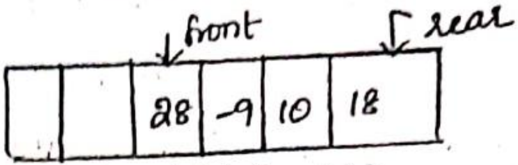
insertend(10)



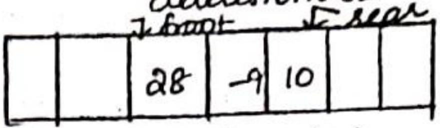
insertend(8)



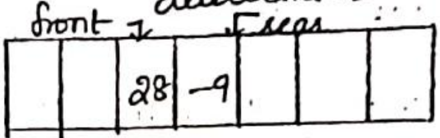
deletefront()



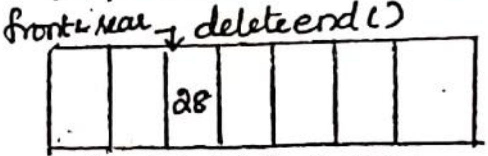
deletefront()



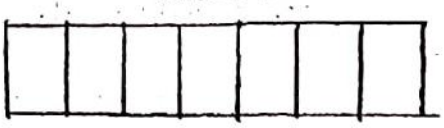
deleteend()



deleteend()



deleteend()

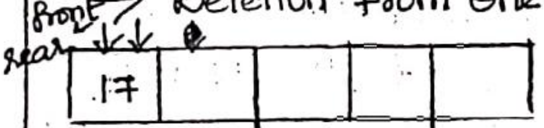


deleteend()

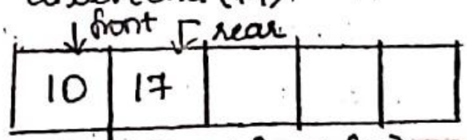
front & rear = -1.

Building an O/p restricted Dequeue.

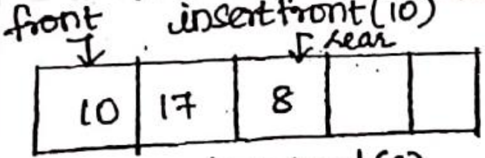
⇒ Deletion from one end & Insertion at both ends.



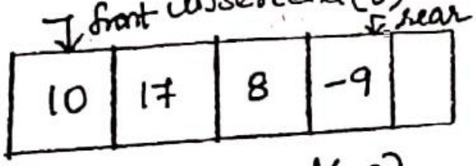
insertend(17)



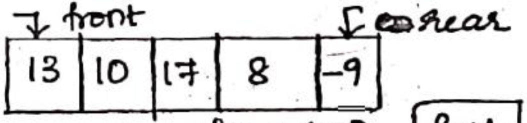
insertfront(10)



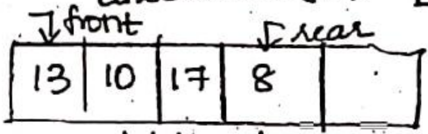
insertend(8)



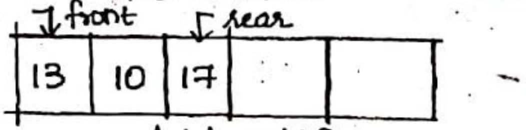
insertend(-9)



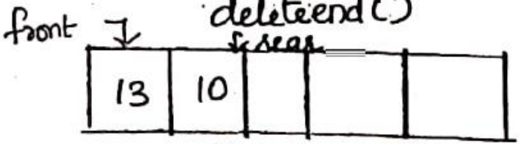
insertfront(13) full



deleteend()



deleteend()



deleteend()

PRIORITY QUEUE

A Priority Queue is an abstract data type which is like a regular Queue (or) Stack data structure, but where additionally each element has a "Priority" associated with it. In a Priority queue, an element with high Priority is served before an element with low priority.

Properties :

* Every item has a Priority associated with it.

* An element with high priority is dequeued before an element with low priority.

* If two elements have the same Priority, they are served according to their order in the queue. [FCFS Basis]

Operations :

insert (item, Priority) : Inserts an item with given priority.
getHighestpriority ( ) : Returns the highest priority item.
deleteHighestpriority ( ) : Removes the highest priority item.

Implementation : (N|D-2018)

* Using Array
* Using Linked list.
* Using Heaps
    * Binary Heap
    * Fibonacci Heap.

Applications :

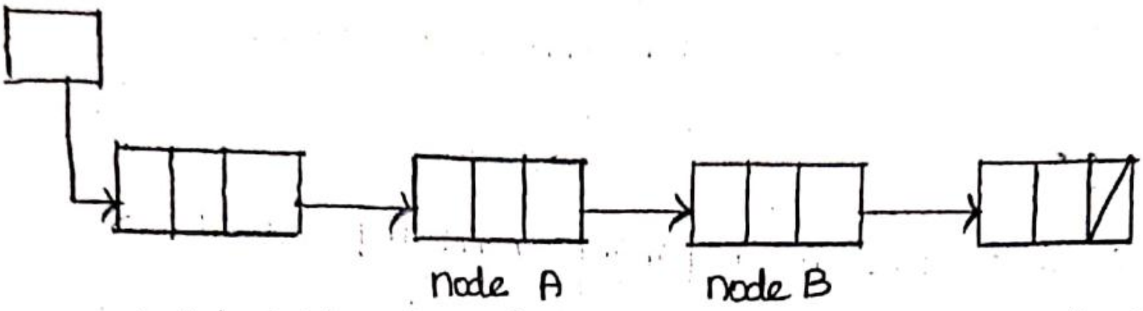
* CPU scheduling.

* Graph algorithms like Dijkstra's, shortest Path algorithm, Prim's, minimum spanning tree etc.,

* All Queue applications where priority is involved. (BFS)

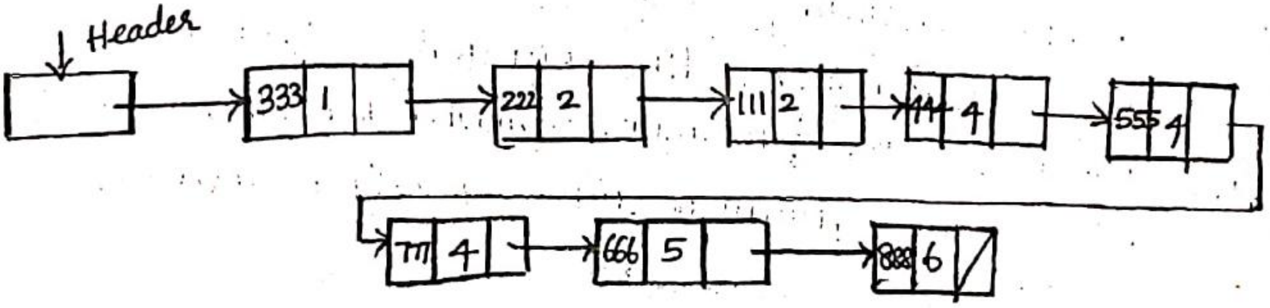
Linked list representation of Priority Queue.

Information	Priority number	Address of next node



Priority of node A is higher than priority of node B. (or) Both have same priority → node A was added before node B.

Location	Info	Priority	Link
1	222	2	
2			
3	444	4	
4	555	4	
5	333	1	
6	111	2	
7			
8	666	5	
9	777	4	
10	888	6	
11			



Application of Stack.

Infix to postfix Expression.

1) A+(B*C-(D/E^F)*G)*H

Scanned I/p	Stack	O/p Expression.
A	Empty	A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(*-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
^	+(-(/^	ABC*DE
F	+(-(/^	ABC*DEF
)	+(-(/^)	ABC*DEF^/
*	+(-*	ABC*DEF^/
G	+(-*	ABC*DEF^/G
)	+(-*)	ABC*DEF^/G*-
*	+*	ABC*DEF^/G*-
H	+*	ABC*DEF^/G*-H
Empty	+*	ABC*DEF^/G*-H*+
Empty	Empty	ABC*DEF^/G*-H*+

A BC* DEFF^/G*- H*+

	Infix	Postfix	
2)	((A+B)*C-D)*E	AB+C*D-E*	AB-C*D-E*
3)	a*(b+c)*d	abc+*d*	a bc+*d*
4)	A/B^C-D	ABC^/D-	ABC^/D-

Applications of Queue.

- When data is transferred asynchronously between two Process. eg., IO Buffers.
- When resource is shared among multiple consumers. Eg include CPU and Disk scheduling.
- In recognizing palindrome.
- Keyboard buffer.
- JOB scheduling.
- Round Robin scheduling.
- Simulation.
- Graph theory.

ABD↑+ EF-/G+
ABD+* E/FGHK/+*-
Postfix 457* 836^/9*-2*+

Infix to Postfix.

1) (A+B↑D)/(E-F)+G   ABD↑+EF-/G+

2) A*(B+D)/E-F*(G+H/K)   ABD+*E/FGHK/+*-

3) 4+(5*7-(8/3^6)*9)*2   457*836^/9*-2*+

4) A*(B+C)*D.

Scanned I/P	Stack	Expression o/p
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*(+)	ABC+
*	**	ABC+*
D	*	ABC+*D
Empty	Empty	ABC+*D*

UNIT-III

NON LINEAR DATA STRUCTURES - TREES.

1) Tree ADT
2) Tree Traversals
3) Binary Tree ADT
4) Expression Trees
5) Applications of Trees
6) Binary Search Tree ADT
7) Threaded Binary Trees
8) AVL Trees
9) B-Tree
10) B^+ Tree
11) Heap.
12) Applications of Heap.

A tree is a data structure made up of nodes (or) Vertices and edges without having any cycle. The tree with no nodes is called the null (or) empty tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

Terminologies Used in Tree:

Root :

The top node in a tree. (A)

Child :

A node directly connected to another node when moving away from the root. Except Root node all are child node.

Parent :

The converse notation of a child. ie., parent node that has an edge to a child node. (A, B, C)

Siblings :

A group of nodes with the same parent. (D and E are siblings for B)

Descendant : (children, grandchildren, great grand children)

A node reachable by repeated proceeding from parent to child. Also known as subchild. (lower hierarchy)

Ancestor : [Parent, Grandparent, Great-GP and so on]

A node reachable by repeated proceeding from child to parent. (higher hierarchy)

Leaf : (Terminal node) Eg; D, E, F, G

External node (not common). A node with no children.

Branch node (Internal node) : A node with atleast one child. Eg., Root node. (Non-Terminal) nodes. Eg; A, B, C

Degree : for a given node, its number of children. A leaf is necessarily degree zero. Eg; Degree (A) = 2.

Edge : The connection between one node and another.

Path : A sequence of nodes and edges connecting a node with a descendant.

Level : The level of a node is defined as : 1 + the number of edges between the node and the root. Level = Depth + 1.

Level of Root = 0.

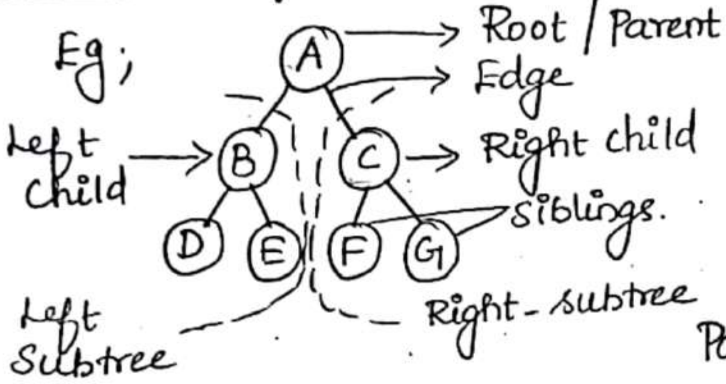
Height of tree : The height of a tree is the (N/D-2018) height of its root node. [Starts from 0]. Leaf don't have height.

Height of node : The height of a node is the number of edges on the longest path between that node and a leaf.

Depth : The depth of a node is the number of edges from the tree's root node to the node. [Starts from 0]

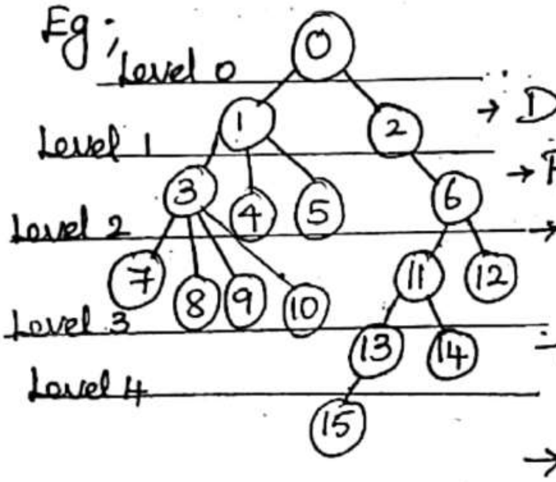
Forest : A forest is a set of n ≥ 0 disjoint trees.

Eg;



Depth of node C : 01
Height of tree : 2.
Height of Root : ~~3.~~ 2
Height of node C : 01
Depth of tree : 2.
Depth of Root : 0.
Level : 3 levels. (1&2)
Path from A to F : A – C – F

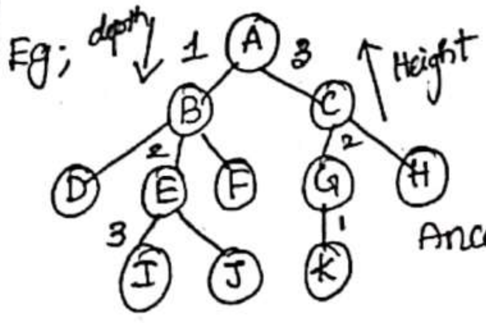
Eg;



→ It has 16 nodes.
→ Depth : 5
→ Degree of Tree :- 5
→ Degree of node 0 :- 2
→ Root : Node 0
→ Node 4 is a leaf and Node 4 is the child of 1.
→ Root node 0 is the Grandparent of node 4.
→ Nodes 3, 4 and 5 are siblings. since it has same parent 1.
→ Height of tree : 6 5

In a tree with 'N' nodes, there will be maximum 'N–1' edges.

Eg;



Depth of tree : 3
Depth of A : 0
Depth of B : 1
Depth of K : 3

Height of tree :- 3.
Height of A : 3
Height of B : 2
Height of K : 0

Ancestor :- (A) is ancestor for all nodes.

(2) Tree Traversals.

Tree traversal (also known as tree search) is a form of Graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

- * Inorder
- * Preorder &
- * Postorder.

(N/D-2018)

Unlike linear data structures (Array, Linked List, Stacks, Queues, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

DFS: Depth First Search.

Eg.,

Inorder (Left, Root, Right):
4 2 5 1 3

Preorder (Root, Left, Right):
1 2 4 5 3

Postorder (Left, Right, Root):
4 5 2 3 1

BFS: (Breadth first Search)

1 2 3 4 5.

Inorder:

1) Traverse the left subtree.
2) Visit the root.
3) Traverse the right subtree.

```
Void inorder (Tree T)
{
if (T! = NULL)
{
inorder (T->left);
PrintElement (T->Element);
inorder (T->right);
}
}
```

Preorder :

1) visit the root
2) Traverse the left subtree
3) Traverse the right subtree.

```
void Preorder (Tree T)
{
  if (T! = NULL)
  {
   PrintElement (T->Element);
   Preorder (T->left);
   Preorder (T-> right);
} }
```

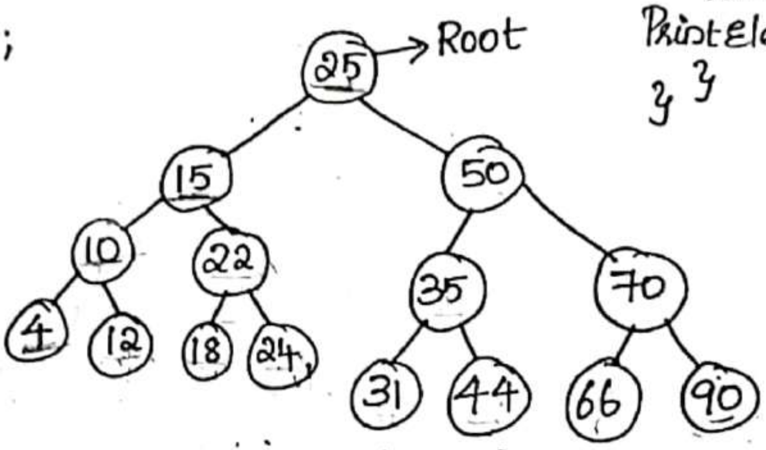
Postorder :

1) Traverse the left subtree
2) Traverse the right subtree
3) Visit the root.

```
   void Postorder (Tree T)
{
    if (T! = NULL)
 {
   Postorder (T->left);
   Postorder (T-> right);
 PrintElement (T-> element);
} }
```

L-R-Ri
R-L-Ri
L-Ri-R

Eg;



Inorder : 4-10-12-15-18-22-24-25-31-35-44-50-66-70-90

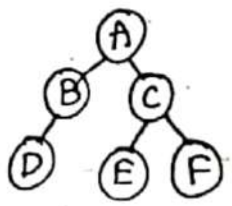
Preorder: 25-15-10-4-12-22-18-24-50-35-31-44-70-66-90

Postorder: 4-12-10-18-24-22-15-31-44-35-66-90-70-50-25

(N/D-2018)

BFS - Breadth first Search.

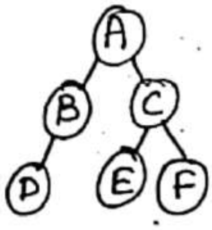
* It uses the Queue for storing the nodes.
* Constructs wide and Short tree.
* Vertex-Based algorithm.



BFS : A-B-C-D-E-F

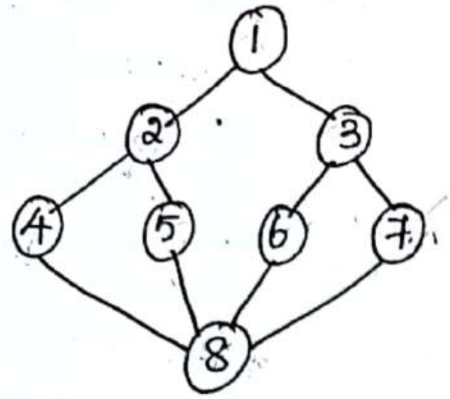
DFS - Depth first Search.

* It uses the stack for traversal of the nodes.
* constructs narrow and long trees.
* Edge-based algorithm.



DFS : A-B-D-C-E-F.

Eg;



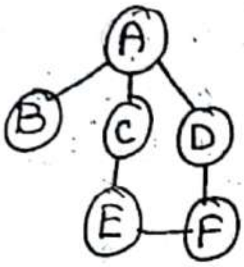
(we use all connections at one time)

BFS: 1-2-3-4-5-6-7-8

(we use only one connection at a time)

DFS: 1-2-4-8-5-6-7-3

Eg;



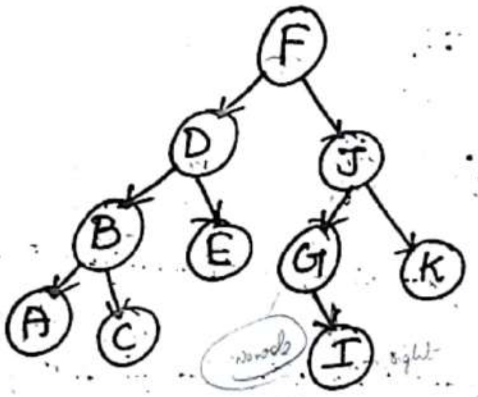
BFS: A B C D E F

DFS: A B C E F D (or)

A D F E C B (or)

A C E F D B

Eg;



L R Ri

Inorder : ABCDEFGIJK

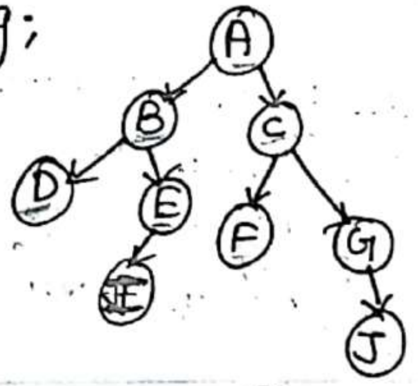
Ri L Ri

Preorder : FDBACEJGIK

L Ri R

Postorder: ACBEDIGKJF

Eg;

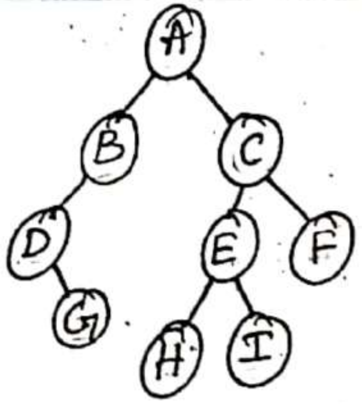


Inorder: DBIEAFCGJ

Preorder: ABDEICFGJ

Postorder: DIEBFJGCA

Eg;



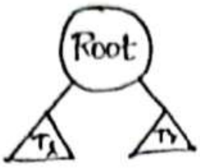
Inorder : DGBAHEICF

Preorder: ABDGCEHIF

Postorder: GDBHIEFCA

(3) BINARY TREE ADT

A Binary tree is a tree in which no node can have more than two children. The maximum degree of any node is two. This means the degree of a binary tree is either zero (or) one (or) two.



T_l – Left Tree
T_r – Right Tree.

In the above fig., the binary tree consists of a root and two sub trees T_l and T_r. All nodes to the left of the binary tree are referred as left subtrees and all nodes to the right of a binary tree are referred to as right subtree.

Implementation:

A Binary tree has atmost two children, we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the key information plus two pointers (left and right) to other nodes.

Binary tree node declaration:

```
typedef Struct tree_node *tree_Ptr;
Struct tree_node
{
  element_type element;
  tree_ptr left;
  tree_ptr right;
};
typedef tree_ptr Tree;
```

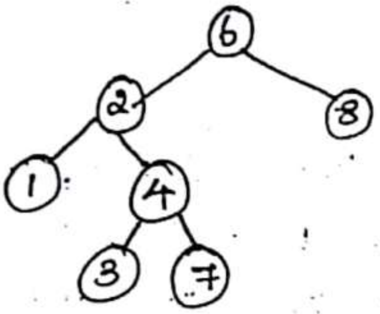
(or)

```
Struct node
{
 int data;
Struct node *left;
Struct node *right;
};
```

Types of Binary Tree :

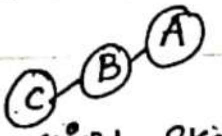
- * Strictly binary tree.
- * Skew tree
- * Left Skewed binary tree.
- * Right Skewed binary tree.
- * Fully Binary tree (or) Proper binary tree
- * Complete Binary tree.
- * Almost Complete Binary tree.

*** Strictly Binary tree :** It is a Binary tree where all the nodes will have either zero (or) two children. It does not have one child in any node.

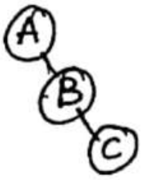


*** Skew tree :** It is a binary tree in which every node except the leaf has only one child node. There are two types of skew tree, they are left skewed binary tree and right skewed binary tree.

*** Left Skewed Binary tree :** A left skew tree has node with only the left child. It is a binary tree with only left subtrees.



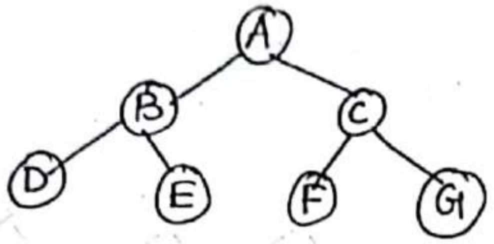
*** Right Skewed Binary tree :** A right skew tree has node with only the right child. It is a Binary tree with only right subtrees.



* Full Binary tree (or) Proper Binary tree :

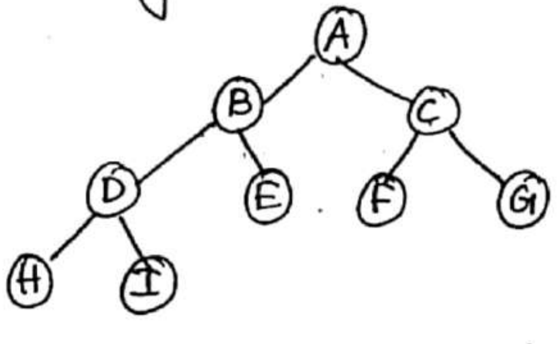
A Binary tree is a full binary tree if all leaves are at the same level and every non leaf node has exactly two children and it should contain maximum possible number of nodes in all levels. * A full Binary tree of height h has $2^{h+1}-1$ nodes.

$\boxed{2^{h+1}-1}$

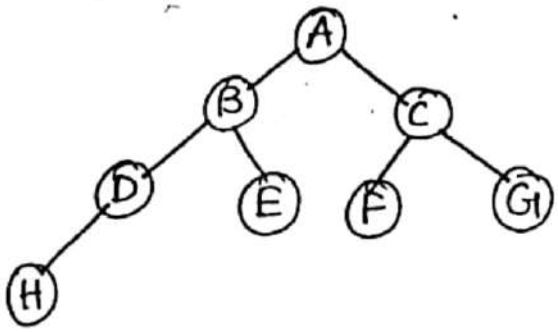


$2^{h+1}-1$ nodes.

*** Complete Binary tree:** Every non-leaf node has exactly two children but all leaves are not necessary at the samelevel. A complete Binary tree is one where all levels have the maximum number of nodes except the last level. The last level elements should be filled from left to right.



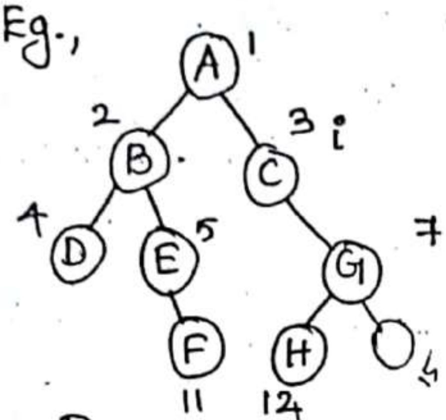
*** Almost Complete Binary tree** : An almost complete Binary tree is a tree in which each node that has a right child also has a left child. Having a left child does not require a node to have a right child.



Implementation of Binary tree : (In memory)

- Array Implementation
- Linked list Implementation.

Eg.,



Using Array.

7

0	A	B	C	D	E		G				F	H
0	1	2	3	4	5	6	7	8	9	10	11	12

Parent :- $[i/2] \Rightarrow \lfloor i/2 \rfloor \Rightarrow \lfloor 3/2 \rfloor \Rightarrow \lfloor 1.5 \rfloor \Rightarrow 1$

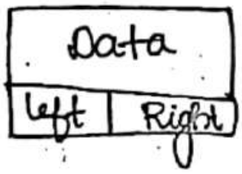
If c is parent for eg., then,

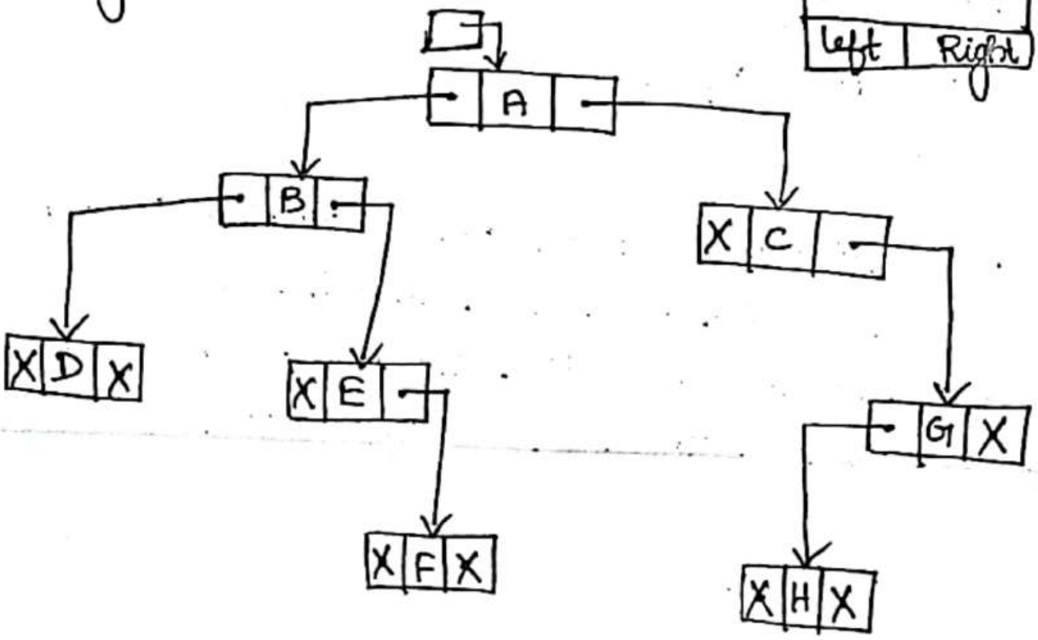
leftchild = $2 * i \Rightarrow 2 * 3 \Rightarrow 6$

Right child = $2 * i + 1 \Rightarrow 2 * 3 + 1 \Rightarrow 6 + 1 \Rightarrow 7$

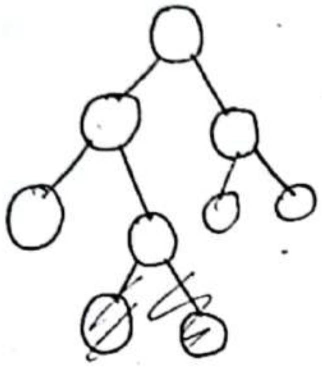
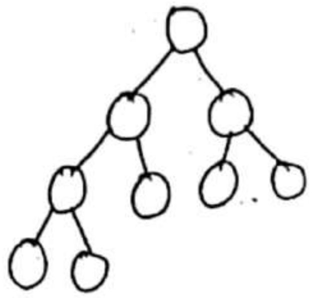
So if parent is c, then leftchild is at 6th Position and right child is at 7th position.

Using Linked list :-





Left child Address	Data	Right Child Address

Full Binary tree	Complete Binary tree
* A full binary tree (Sometimes proper binary tree (or) 2-tree) is a tree in which every node other than the leaves has two children.	* A Complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
	
* Each node has zero (or) two children.	* All levels of the tree is full, except possibly the last level, where nodes are filled from left to right.

(4) Expression Trees. N/D-2018

An expression tree is a representation of expressions arranged in a tree-like data structure. In other words, it is a tree with leaves as operands of the expressions and node contain the operators.

Similar to other data structures, data interaction is also possible in an expression tree.

Expression trees are mainly used for analyzing, evaluating and modifying expressions, especially complex expressions.

Types

- * Algebraic expressions
- * Boolean expressions.

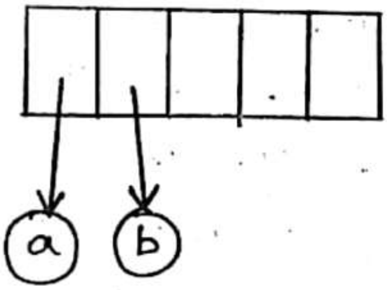
Construction of an expression tree.

The evaluation of the tree takes place by reading the postfix expression one symbol at a time.

* If the symbol is an operand, a one-node tree is created and its pointer is pushed onto a stack.

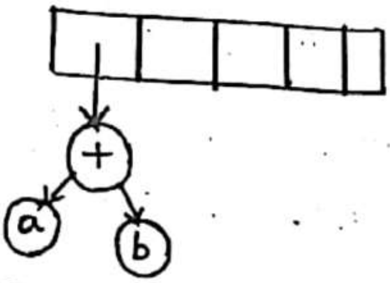
* If the symbol is an operator, the pointers to two trees T_1 and T_2 are popped from the stack and a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively is formed. A pointer to this new tree is then pushed to the stack.

Example: The input is: ab+cde+**.
Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack. for convenience the stack will grow from left to right.



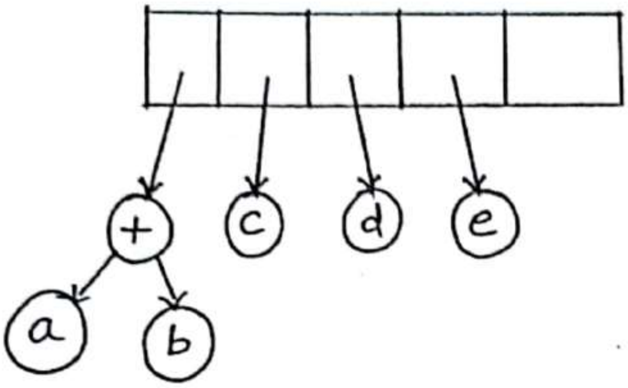
Stack Growing from Left to Right.

⇒ The next symbol is '+'. It pops the two pointers to the trees, a new tree is formed, and a Pointer to it is pushed onto the stack.



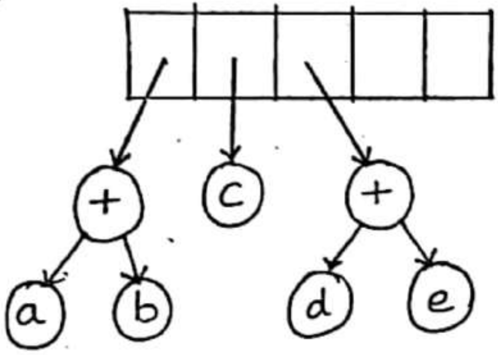
Formation of a new tree.

⇒ Next c, d and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



Creating a one-node tree.

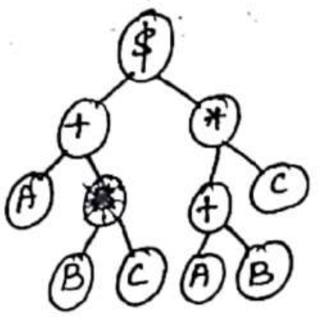
⇒ Continuing, a '+' is read, and it merges the last two trees.

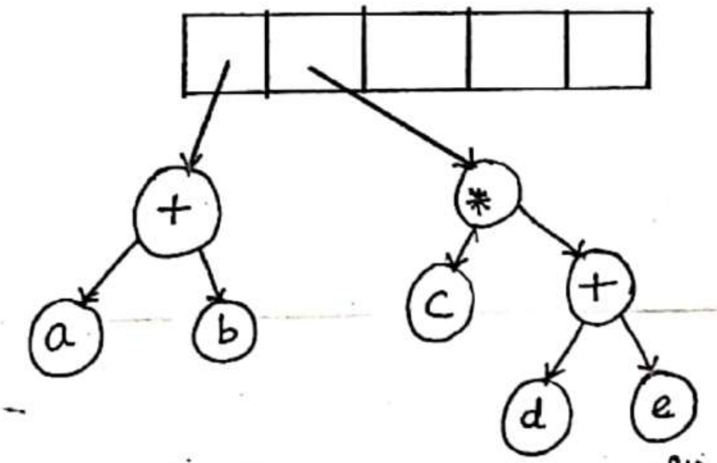


Merging two trees.

⇒ Now, a '*' is read, the last two tree pointers are popped and a new tree is formed with a '*' as the root.

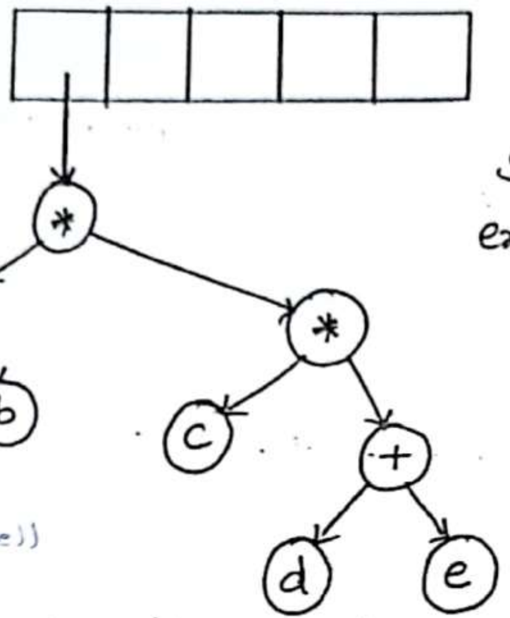
Eg (A+B*C) $ ((A+B)*C)





forming a new tree with a root.

⇒ finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.

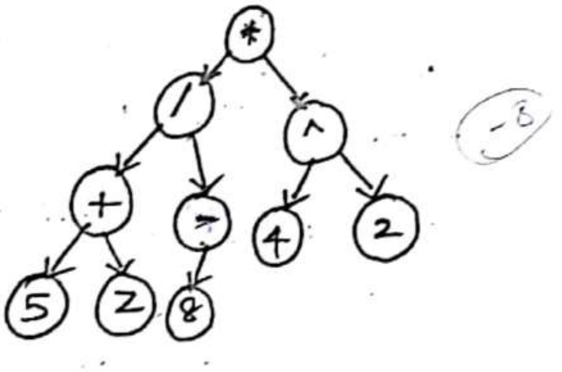


Steps to construct an expression tree. ab+cde+**.

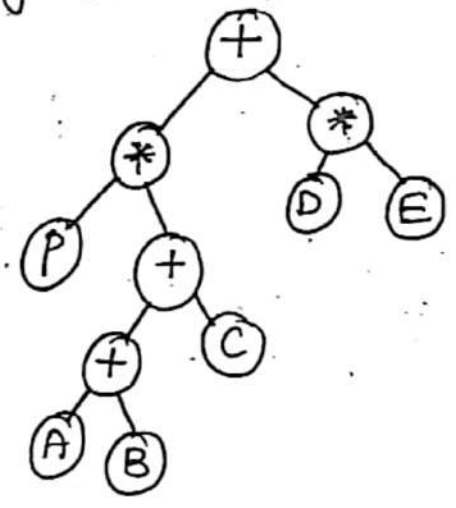
$(a+b)*(c+(d+e))$

Algebraic expressions :-

$((5+z)/-8) * (4 \wedge 2)$

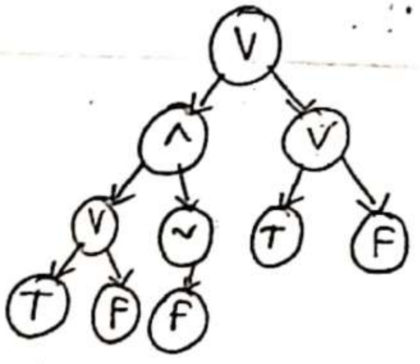


Eg (2) $P*((A+B))+C)+(D*E)$



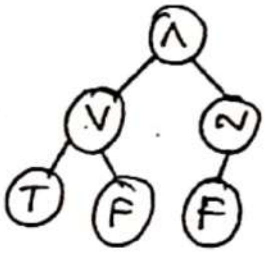
Boolean Expressions :-

$((\text{true} \vee \text{false}) \wedge \sim\text{false}) \vee (\text{true} \vee \text{false}))$

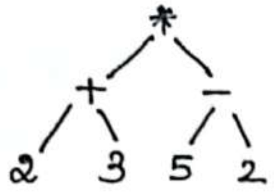


Eg (2)

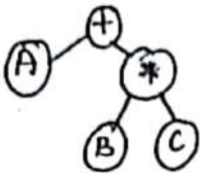
$(\text{true} \vee \text{false}) \wedge \sim \text{false}$



Eg (2) $(2+3) * (5-2)$

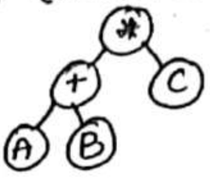


Eg A+B*C

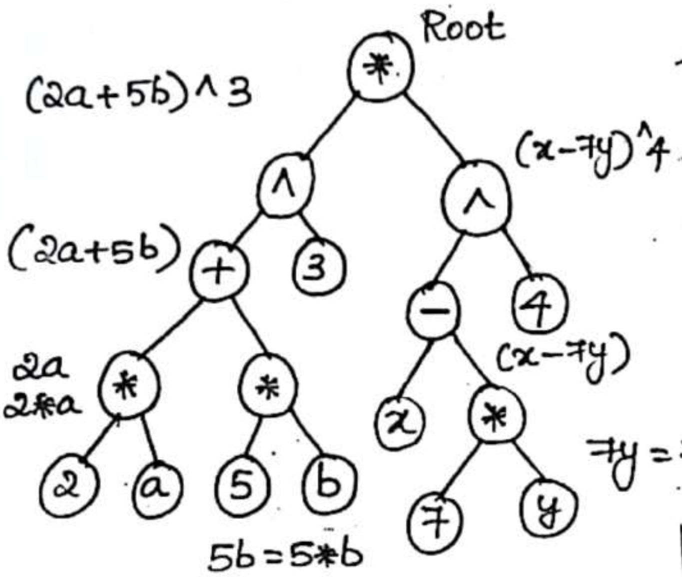


Note :- Internal node – operator
External node – operand

Eg (A+B)*C



Eg (3) $(2a+5b)^\wedge 3 * (x-7y)^\wedge 4$



7y = 7*y

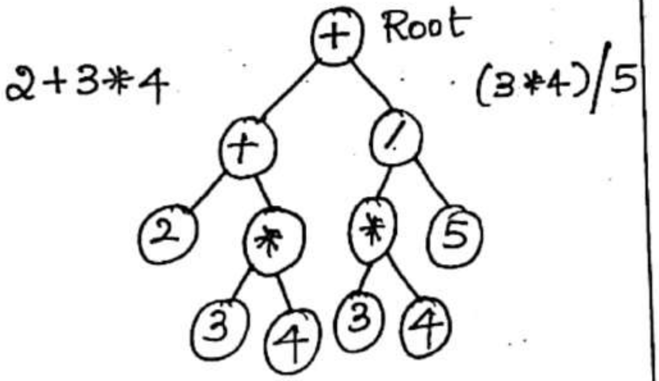
5b = 5*b

1) Divide expressions into two small mathematical term to get Root operator.

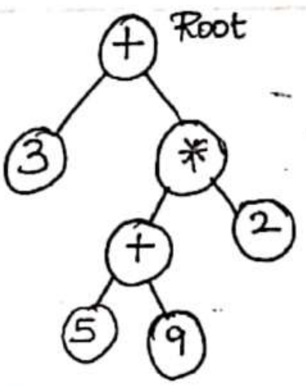
2) Scan symbol from right to left.

3) Place operator then expression on to the node.

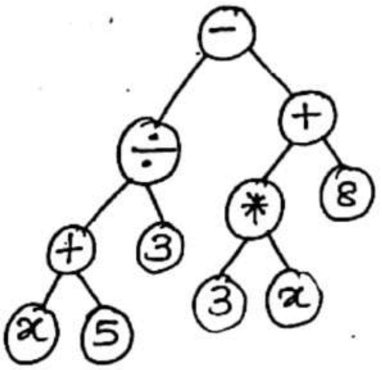
Eg (4) 2+3*4+(3*4)/5



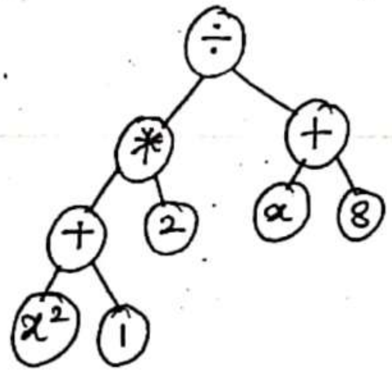
Eg (5) 3+((5+9)*2)



Eg (6) $((x+5) \div 3) - (3x+8)$



Eg (7) $((x^2+1)*2) \div (x+8)$

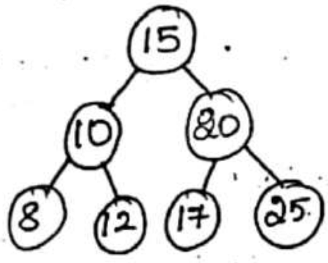


(5) Applications of trees.

- * Binary Search Trees / Binary Sorted trees.
- * Decision Trees.

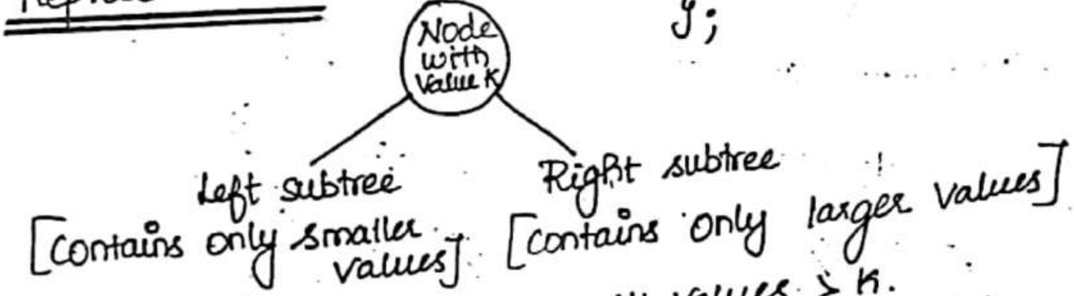
Binary Search Tree (BST) :- It is a binary tree where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

Left subtree ≤ Right subtree.



```
Struct node
{
  int data;
Struct node *leftchild;
Struct node *rightchild;
};
```

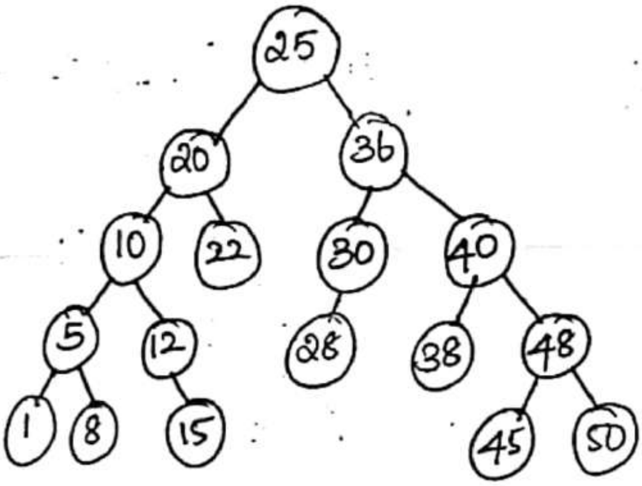
Representation:-



All Values <= K

All Values > K.

Eg.,



Left Subtree (Keys) ≤ node (key) ≤ right-subtree (keys)

Operations on BST.

*Search *Insertion *Deletion.

SEARCH in BST: (N|D-2018)

Step 1 : Read the search element from the user.

Step 2 : Compare, the search element with the value of Root node.

Step 3: If both matches, then display "Given node found".

Step 4 : If both not matches, check whether search element is smaller (or) larger than that node value.

Step 5 : If larger, then continue search process at right subtree

Step 6 : If smaller, then search at left subtree.

Step 7 : Repeat the same until we found exact element.

Step 8 : If we reach the node, then display "Element found", if not then display "Element not found".

INSERTION in BST:

Step 1 : Create a Newnode with given value and set its left and right to NULL.

Step 2 : Check whether tree is empty.

Step 3 : If Empty, then set root to newnode.

Step 4 : If the tree is not empty, then check whether value of newnode is smaller (or) larger than the node. (here it is root node)

Step 5 : If Newnode is smaller (or) equal to root node then move to left else move to right child.

Step 6 : Repeat the above step until we reach to a leaf node.

Step 7: After reaching a leaf node, then insert the newnode as left child. If newnode is smaller (or) equal to that leaf, else insert it as right child.

DELETION IN BST :

Step 1 : Find the node to be deleted using Search operation.

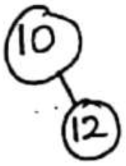
Step 2 : Delete the node using free function (if it is a leaf) and terminate the function.

Insert the following into a Binary Search Trees.

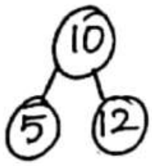
10, 12, 5, 4, 20, 8, 7, 15 and 13.

Insert 10.

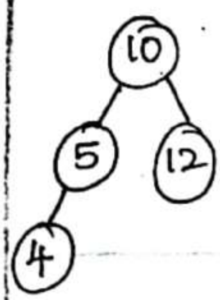
Insert ⑫



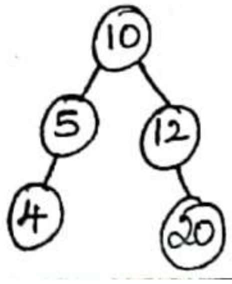
Insert 5



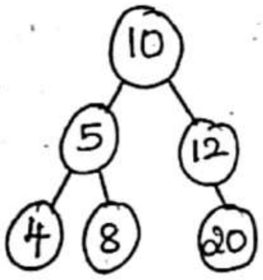
Insert 4



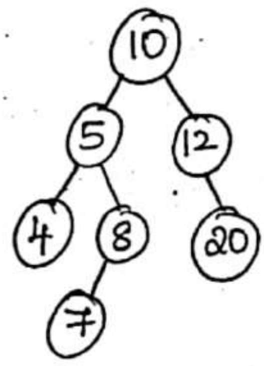
Insert 20



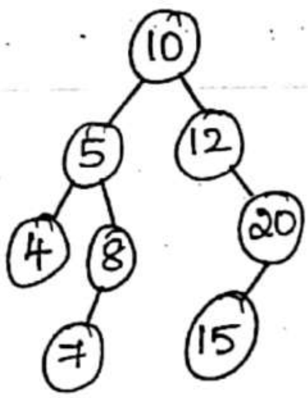
Insert 8.



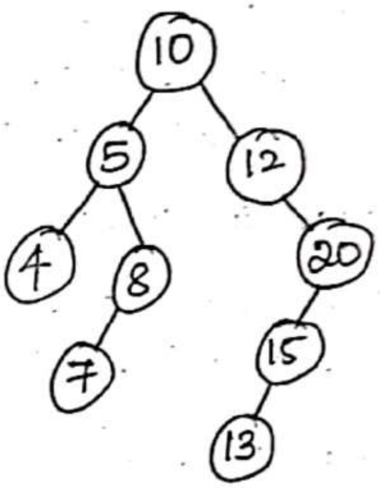
Insert 7.



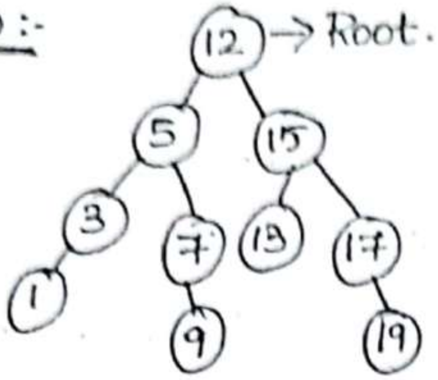
Insert 15.



Insert 13.

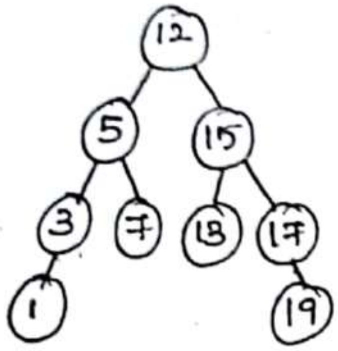


Deletion :-



Case 1 : No child
Case 2 : One child
Case 3 : Two children.

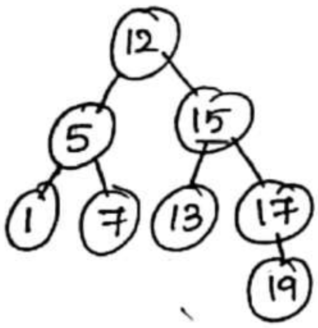
Delete Node 9 :-



Case 1 : No child.

9 is a leaf node, so we just cut the link and wipe off the node that is clear it from memory.

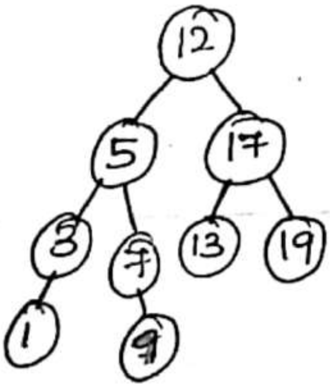
Delete Node 3 :-



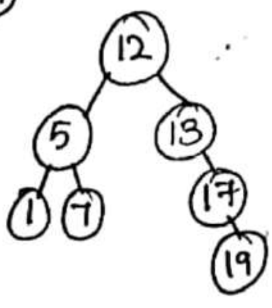
Case 2 : One child.
Wipe it off from memory.

Delete Node 15 :

Case 3 : Two child.



Wipe it off from memory.
(minimum element from right side deleted)
(or) largest element from left side.



Algorithm: INSERTION

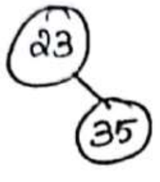
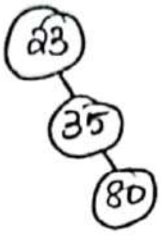
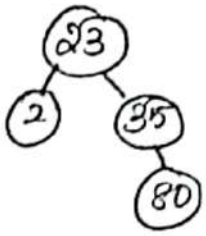
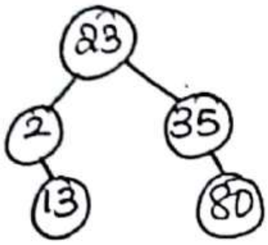
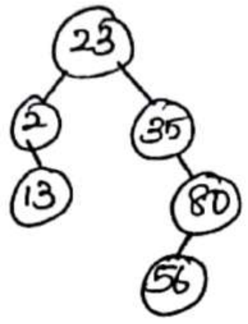
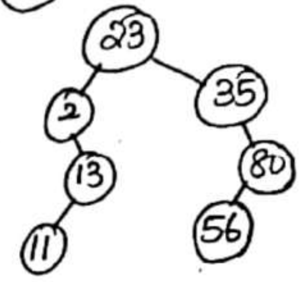
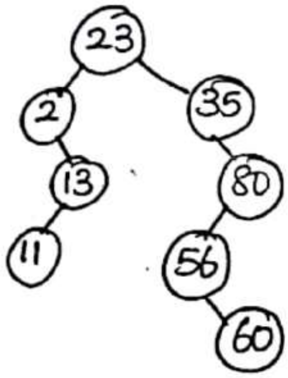
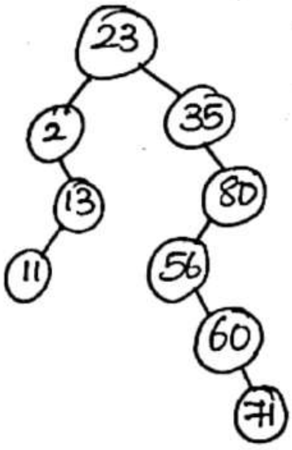
```
Void insert (int data)
{
  Struct node *tempnode = (struct node *) malloc
                          (sizeof (struct node));
   Struct node * current ;
   Struct node * Parent ;
   tempNode -> data = data ;
   tempNode -> leftchild = NULL ;
   tempNode -> rightchild = NULL ;
   if (root == NULL)
 {
     root = tempNode ;
} else {
   current = root ;
   Parent = NULL ;
While (1) {
  Parent = current ;
 if (data < parent -> data)
{ current = current -> leftchild ;
     if (current == NULL)
   {
   Parent -> leftchild = tempNode;
   return;
   }
  }
  else {
   current = current -> rightchild;
   if (current == NULL)
 {
 Parent -> rightchild = tempNode;
 return ;
}}}
}
}
```

Algorithm for Search:-

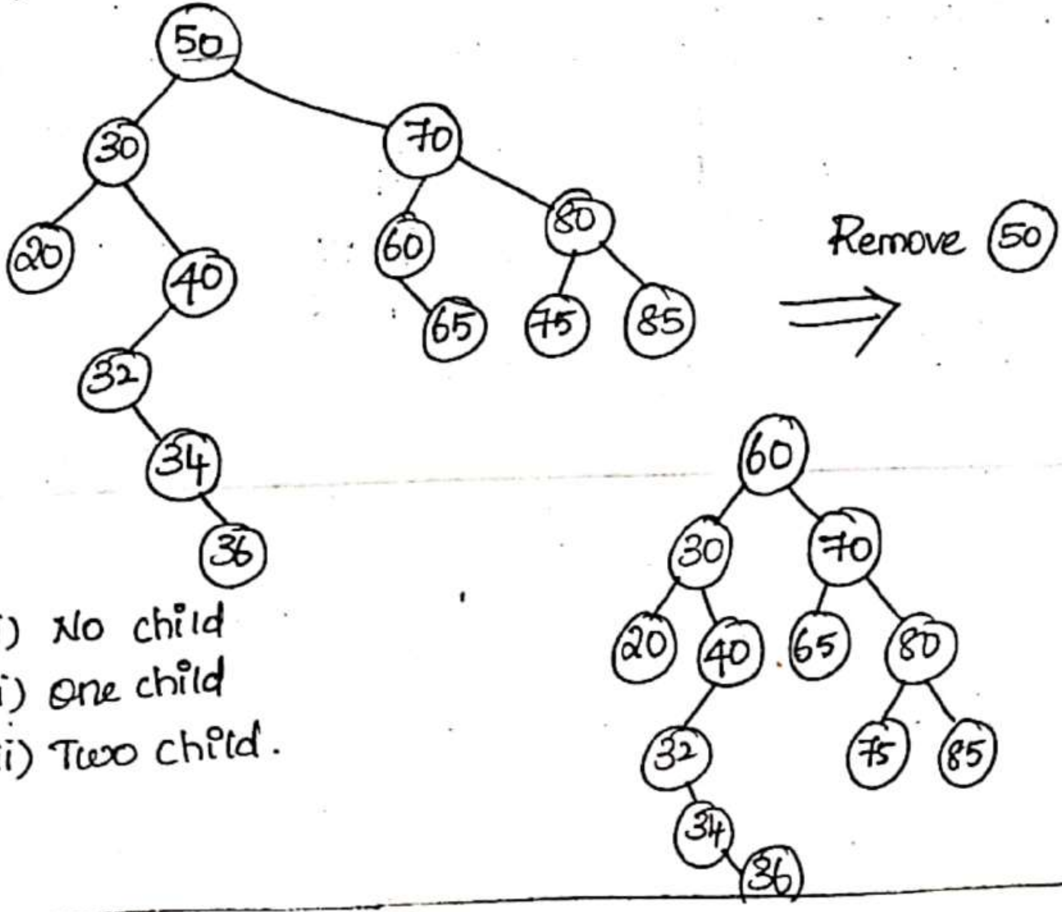
```
Struct node* Search (int data)
{
 Struct node* Current = root ;
Printf ("visiting elements :");
while (current -> data != data)
{  if (current != NULL)
{  Printf ("%d", current -> data);
 //goto left tree.
 if (current -> data > data)
{ current = current -> leftchild ;
 }
else  //goto right tree.
 {
current = current -> rightchild;
 }
 if (current == NULL)
 {
  return NULL ;
  }
 }
}
```

Eg (2): <u>Binary Search tree.</u>

23, 35, 80, 2, 13, 56, 11, 60, 71.

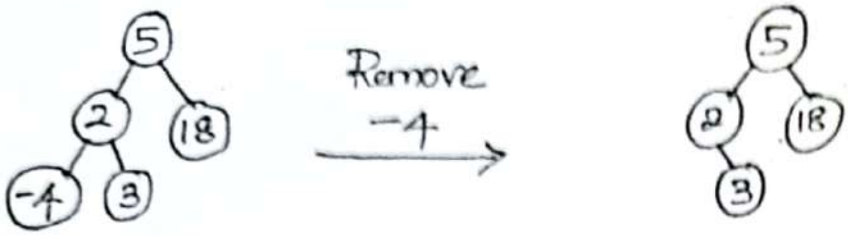
Eg (3)



(i) No child
(ii) One child
(iii) Two child.

(i) No child :-

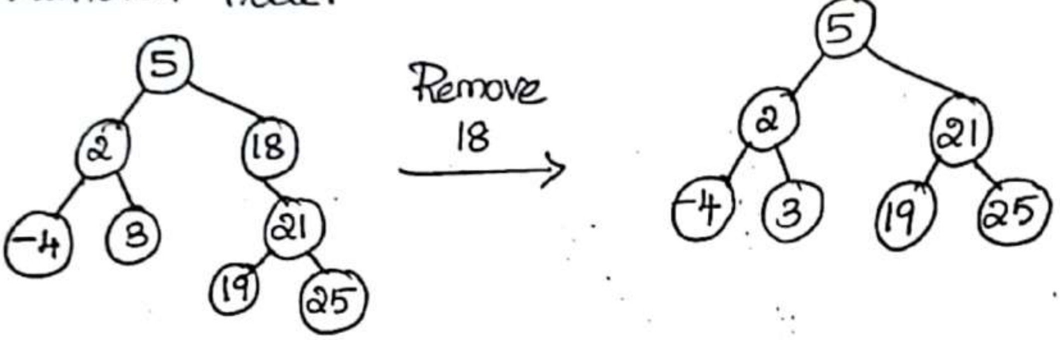
Node to be removed has no children.



(ii) One child :-

"Node to be removed has one child."

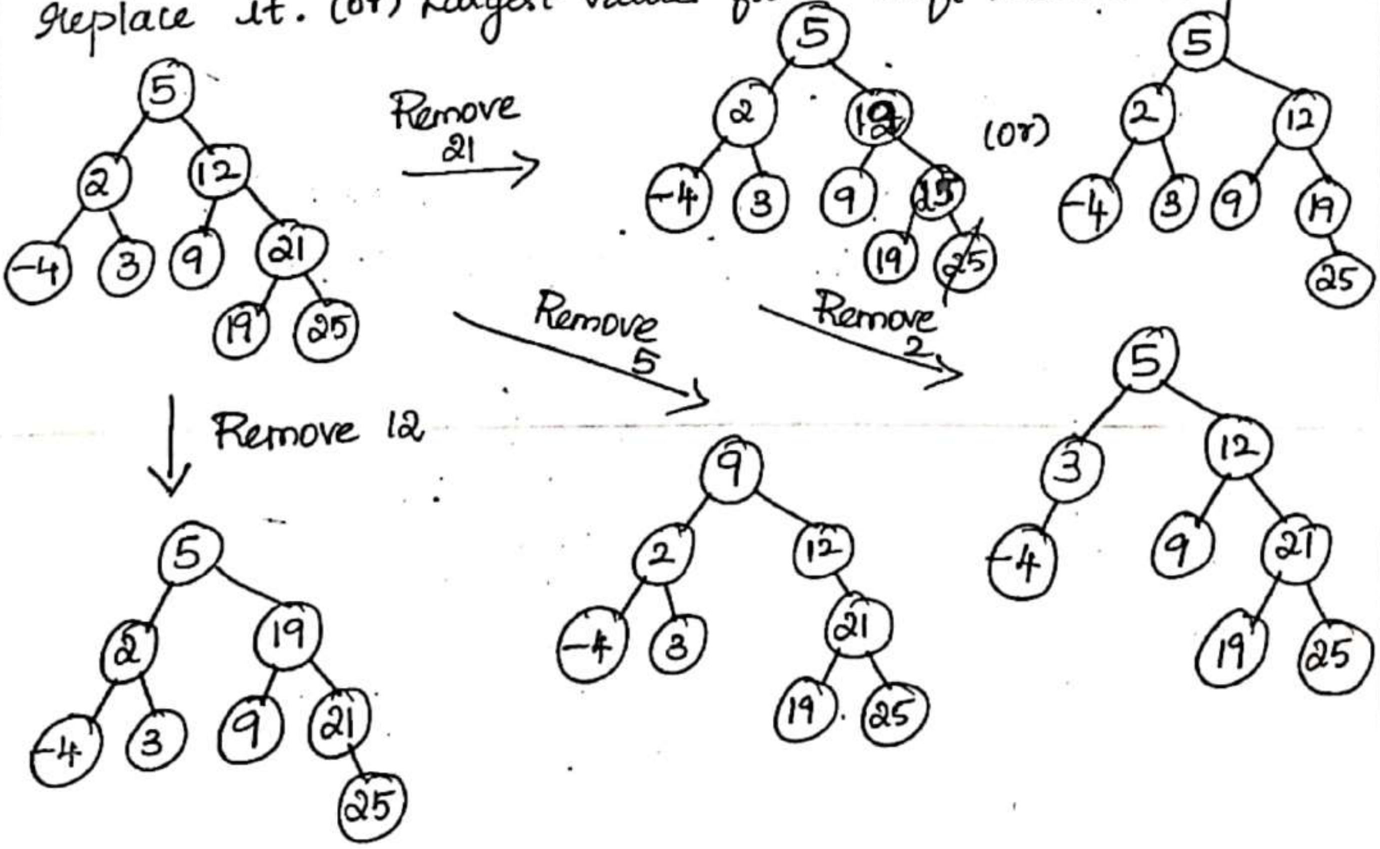
→ Links single child directly to the parent of the removed node.



(iii) Two child :-

Node to be removed has two children.

→ find minimum values in the right subtree & replace it. (or) Largest value from left subtree & replace it.



(7) Threaded Binary Trees.

A Binary tree is threaded by making all right Child Pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be NULL point to the inorder predecessor of the node.

* We have the pointers reference the next node in an inorder traversal; called threads.

* We need to know if a pointer is an actual link (or) a thread, so we keep a boolean for each pointer.

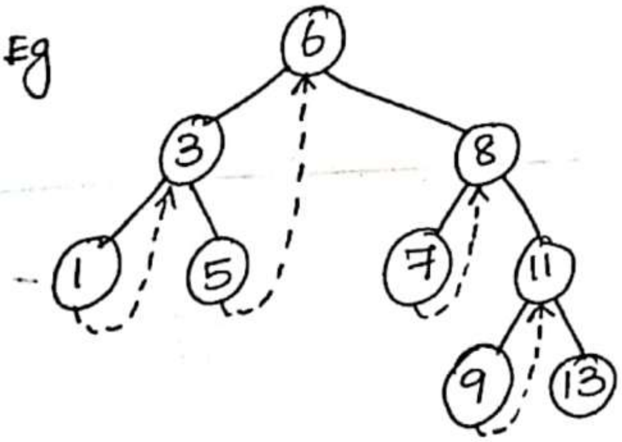
NEED: Threaded binary tree makes the tree traversal faster since we do not need stack (or) recursion for traversal.

Types:-

(i) Single threaded:

* Each node is threaded towards either the in-order, Predecessor (or) successor (left (or) right) means all right NULL pointers will point to inorder successor OR all left NULL pointers will point to inorder Predecessor.

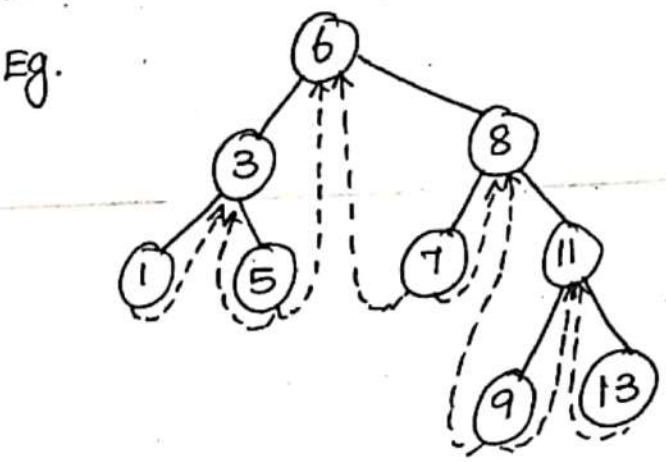
Eg



(ii) Double threaded.

* Each node is threaded towards both the in-order Predecessor and sucessor (left and right) means all right NULL pointers will point to inorder successor AND all left NULL pointer will point to inorder Predecessor.

Eg.



Representation of Threaded Binary Node:

```
  Struct Node
{
   Struct Node *left, *right;
   int info;
// True if left pointer points to Predecessor
// in Inorder Traversal
 boolean lthread;
// True if right pointer points to successor
// in Inorder Traversal
 boolean rthread;
};
```

Let tmp be the newly inserted node :- There can be 3 cases during insertion.

Case (i) Insertion in Empty tree.

Both left and right pointers of tmp will be set to NULL and new node becomes the root.

```
root = tmp;
tmp->left = NULL;
tmp->right = NULL;
```

Case (ii) When new node inserted as the left child.

```
tmp->left = par->left;
tmp->right. = par;
```

Before insertion, the left pointer of parent was a thread, but after insertion, it will be a link pointing to the new node.

```
Par -> lthread = false;
Par -> left = temp;
```

Case (iii) : When new node is inserted as the right child.

tmp -> left = par ;

tmp -> right = par -> right ; Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

Par -> rthread = false ;
Par -> right = tmp ;

Deletion :

Case (1) : Leaf Node need to be deleted.

* If it is left child of parent then after deletion, left pointer of parent should become a thread pointing to its predecessor of the parent node after deletion.

Par -> lthread = true ;
Par -> left = Ptr -> left ;

* If it is right child of parent, then after deletion, right pointer of parent should become a thread pointing to its successor. The Node which was inorder succesor of the leaf node before deletion will become the inorder successor of the parent Node after deletion.

Case (2) : Node to be deleted has only one child.

The Inorder successor and inorder predecessor of the node are found out.

S = insucc (ptr);
P = inpred (ptr); If node to be deleted has left subtree, then after deletion right thread of its predecessor should point to its successor.

P -> left = S ;

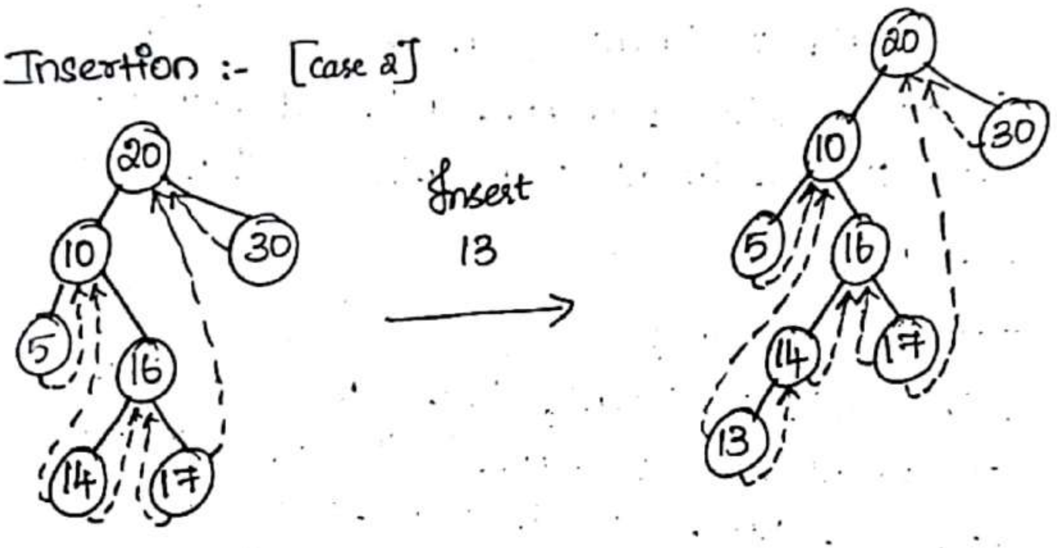
* If Node to be deleted has right subtree, then after deletion left thread of its successor should point to its predecessor.

S → left = P;

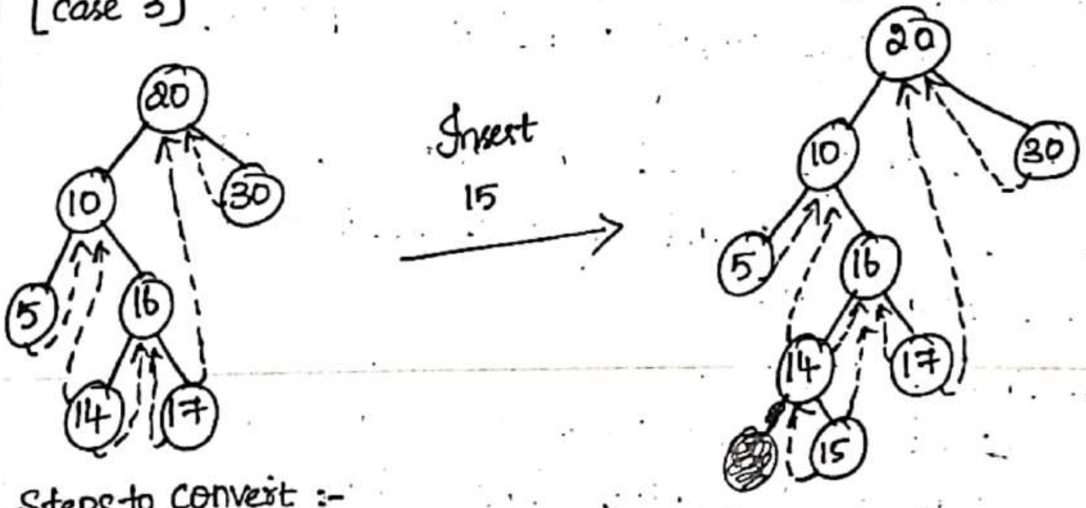
Case (3) : Node to be deleted has two children.

We find inorder successor of Node ptr (Node to be deleted) and then copy the information of this successor into Node ptr. After this inorder successor Node is deleted using either case (1) or case (2).

Insertion :- [case 2]



[case 3]

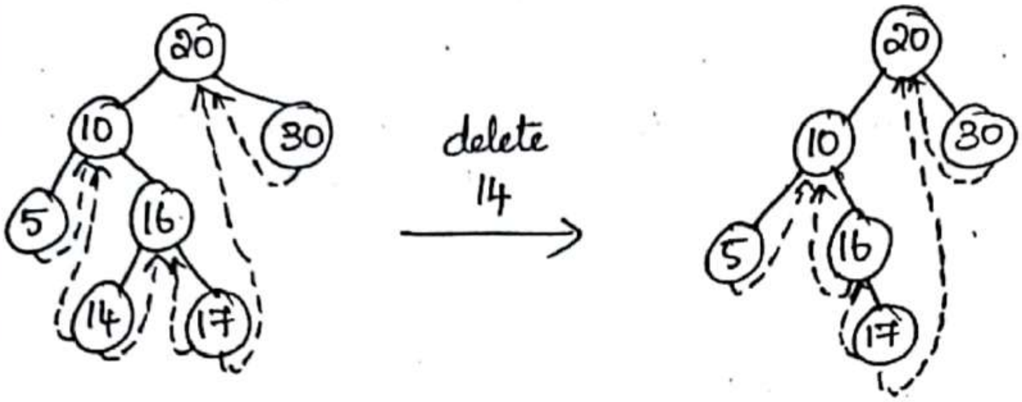


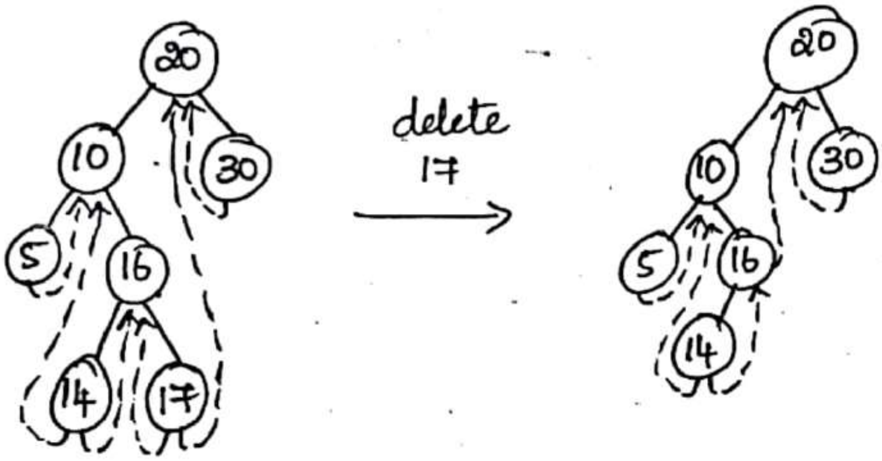
Steps to convert :-

1) Keep the leftmost & right most NULL pointer as NULL.

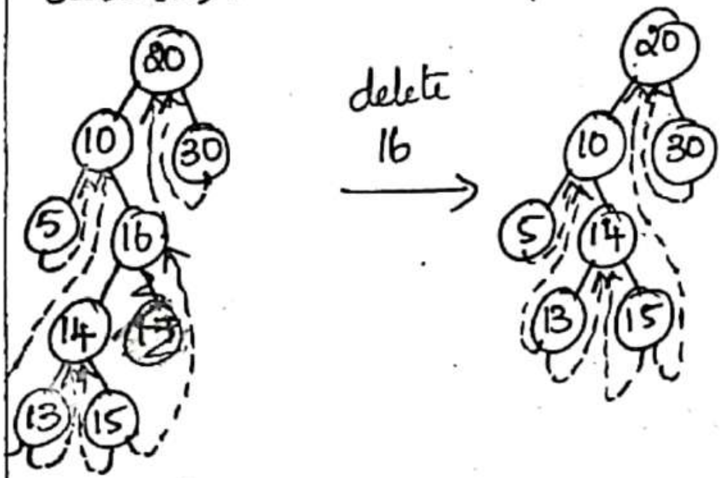
2) Change all other NULL pointer as left ptr (Inorder Predecessor) and Right pointer (Inorder successor)

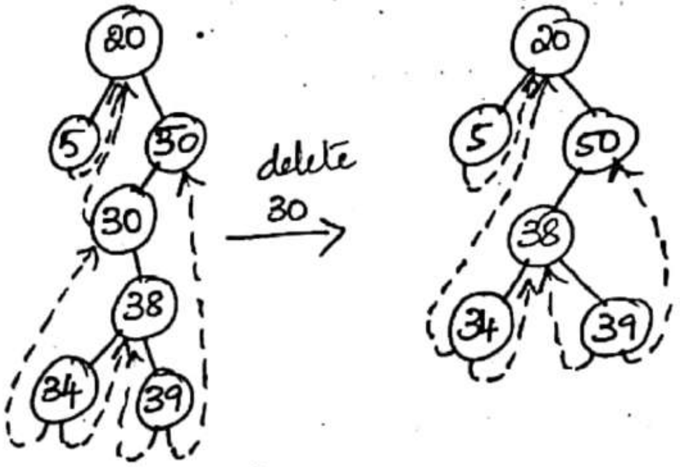
Deletion: Case (1)



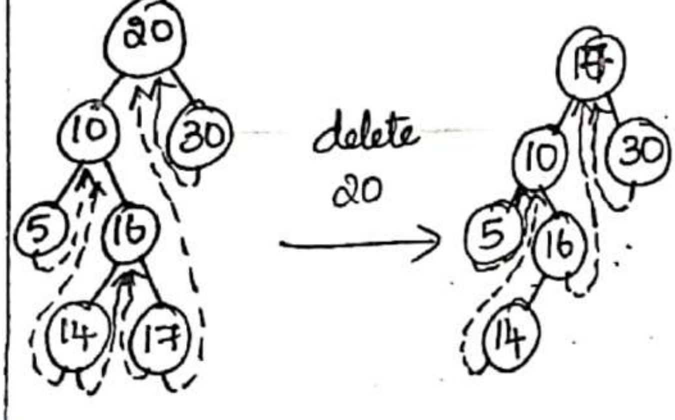


case (2):





case (3):



(8) AVL Trees.

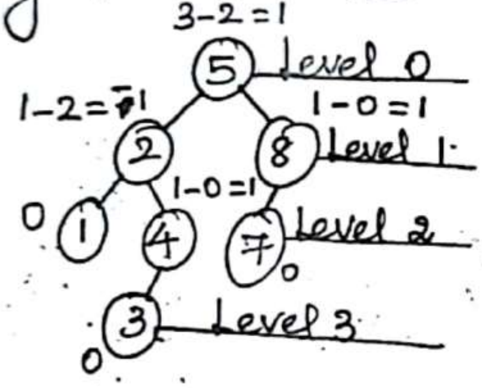
An AVL trees is another balanced binary Search tree, named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be Proposed. Height of the two Subtrees of a node differs by atmost one.

Height of AVL Tree (Balance factor)
= Height of left Subtree − Height of right subtree.

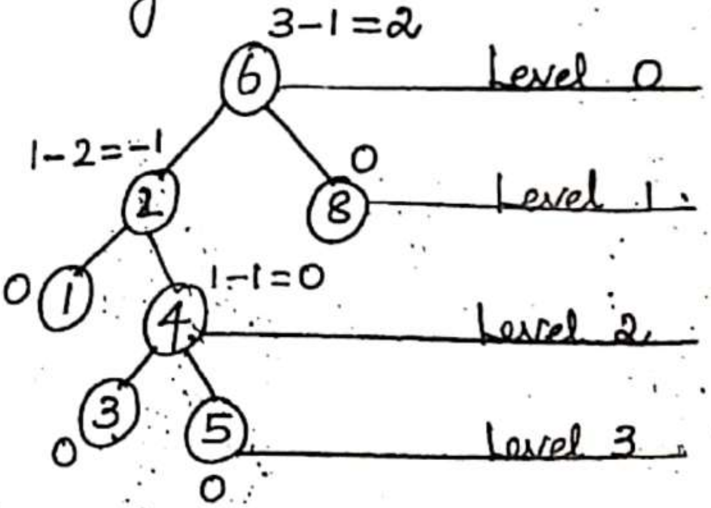
In AVL tree, Balance factor of a node is either −1, 0, +1. So BF cannot be more than one.

Height of an empty tree is defined to be (−1).

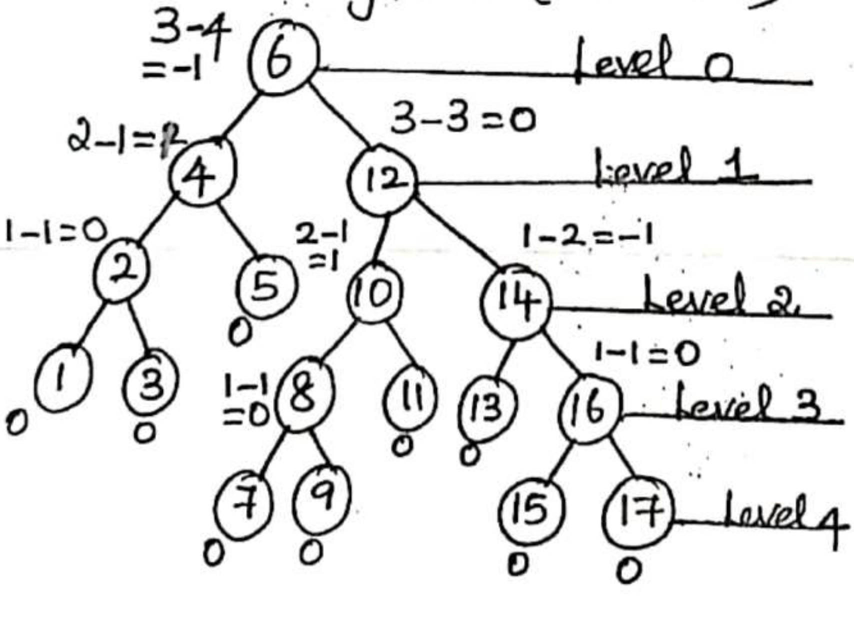
Eg for AVL Tree.



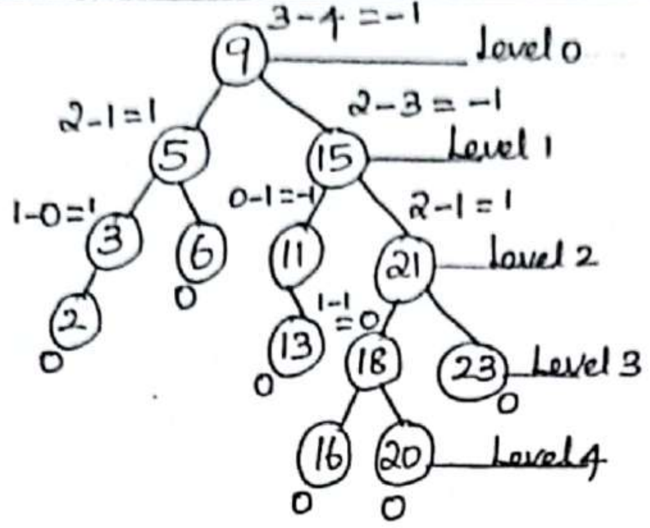
Eg for Not an AVL Tree



Eg (2) A Balanced Binary tree (AVL Tree)



Eg (3)



Operations :-

(i) Insertion.

(ii) Deletion.

Insertion :-

Let the newly inserted node be W.

1) Perform Standard BST insert for W.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z.

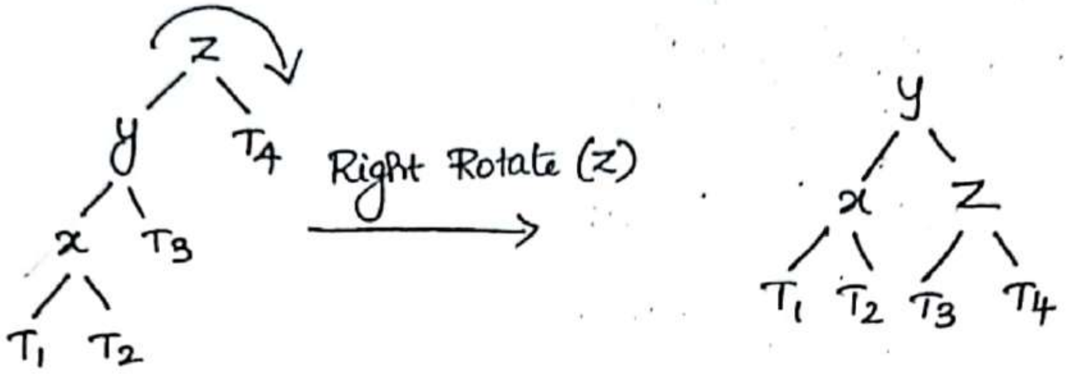
4 possible cases.

1) Left - Left rotation
2) Left - Right rotation.
3) Right - Right rotation.
4) Right - Left rotation.

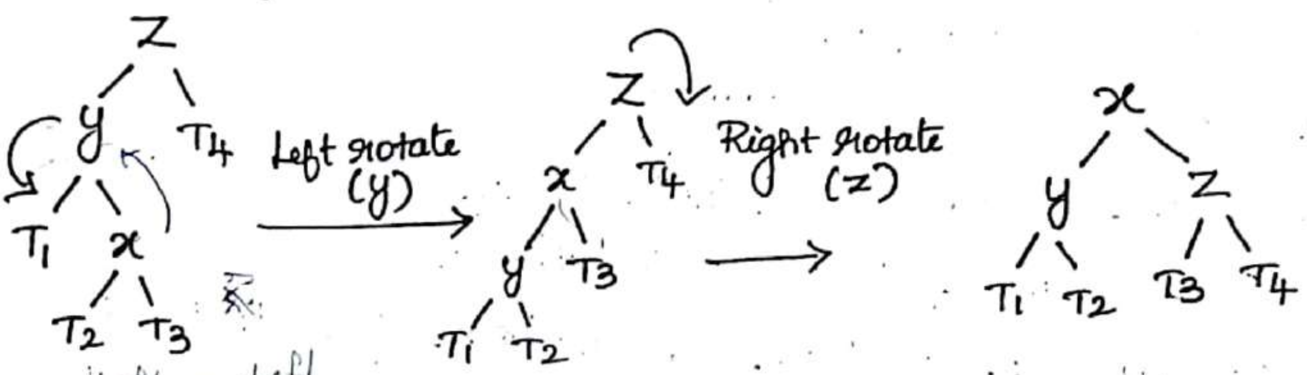
Right - Right

1) Left - Left Rotation. [single rotation]

T1, T2, T3 and T4 are subtrees.

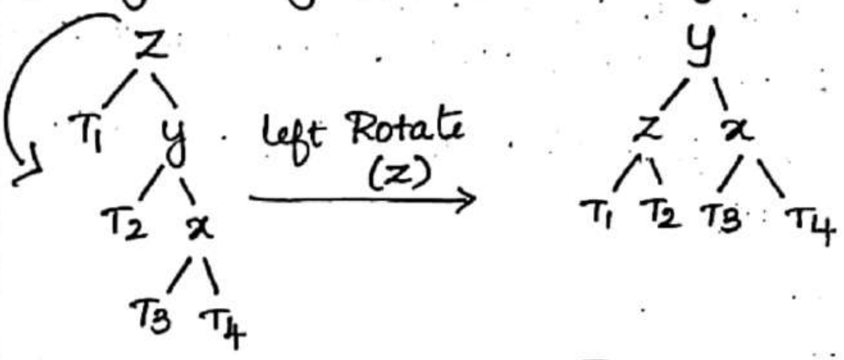


2) Left - Right rotation. [Double rotation]

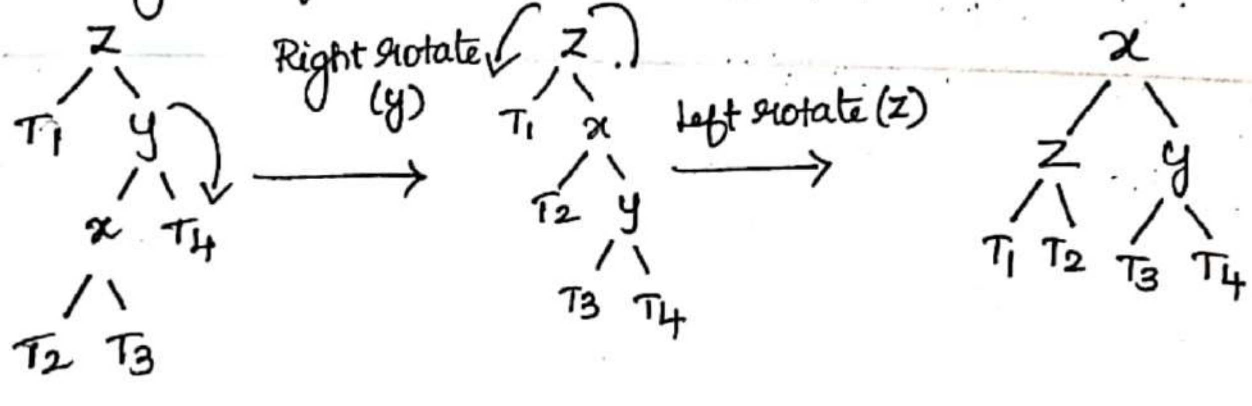


Left - Left

3) Right - Right rotation [Single rotation]



4) Right - Left rotation [Double rotation]

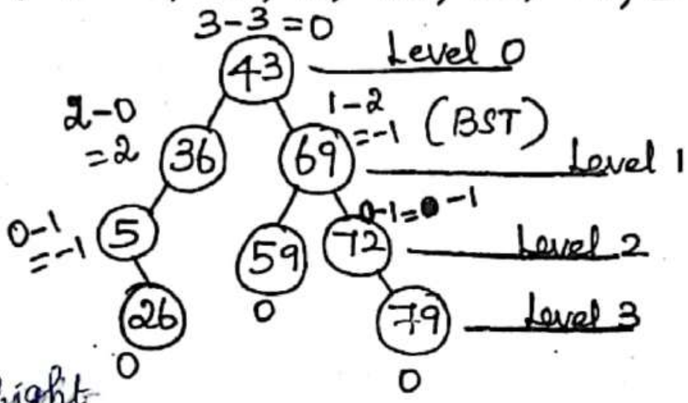


Representation of AVL trees.

```
Struct AVLnode
{
   int data;
   Struct AVLnode *left, *right;
   int balfactor;
};
```

Create an AVL tree by inserting the following.

43, 69, 36, 5, 72, 26, 79, 59.



1) Left - ~~Left~~ Right Rotation.

```
if (balance >1 && key < node->left->key)
    return rightrotate(node);
```

2) Right - ~~Right~~ Left ~~rotation.~~

```
if (balance <-1 && key > node->right->key)
    return leftrotate(node);
```

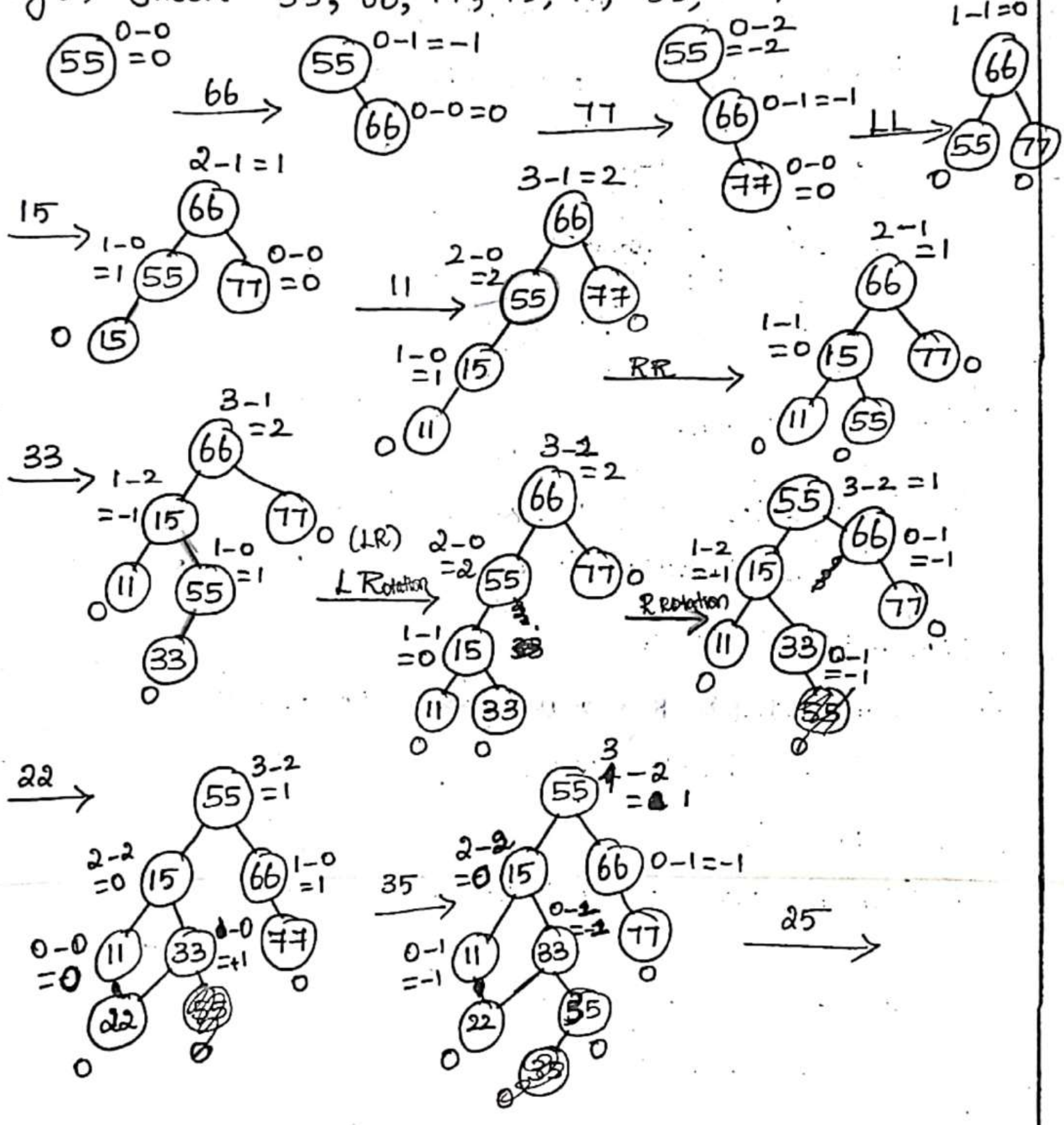
3) Left right case.

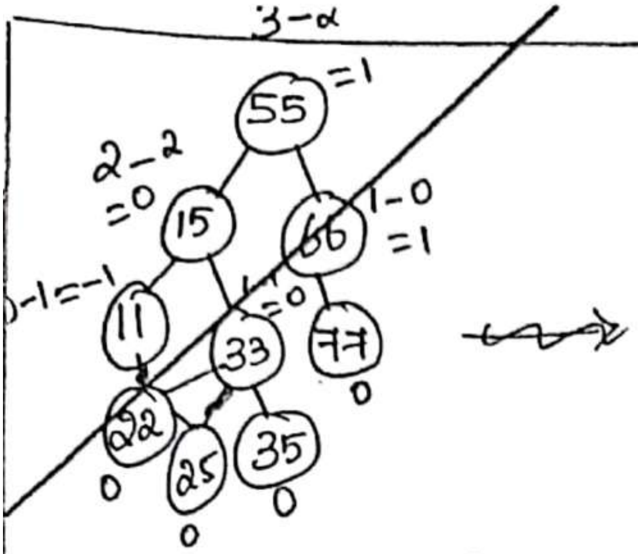
```
if (balance >1 && key > node->left->key)
{
    node->left = leftrotate (node->left);
    return rightrotate(node);
}
```

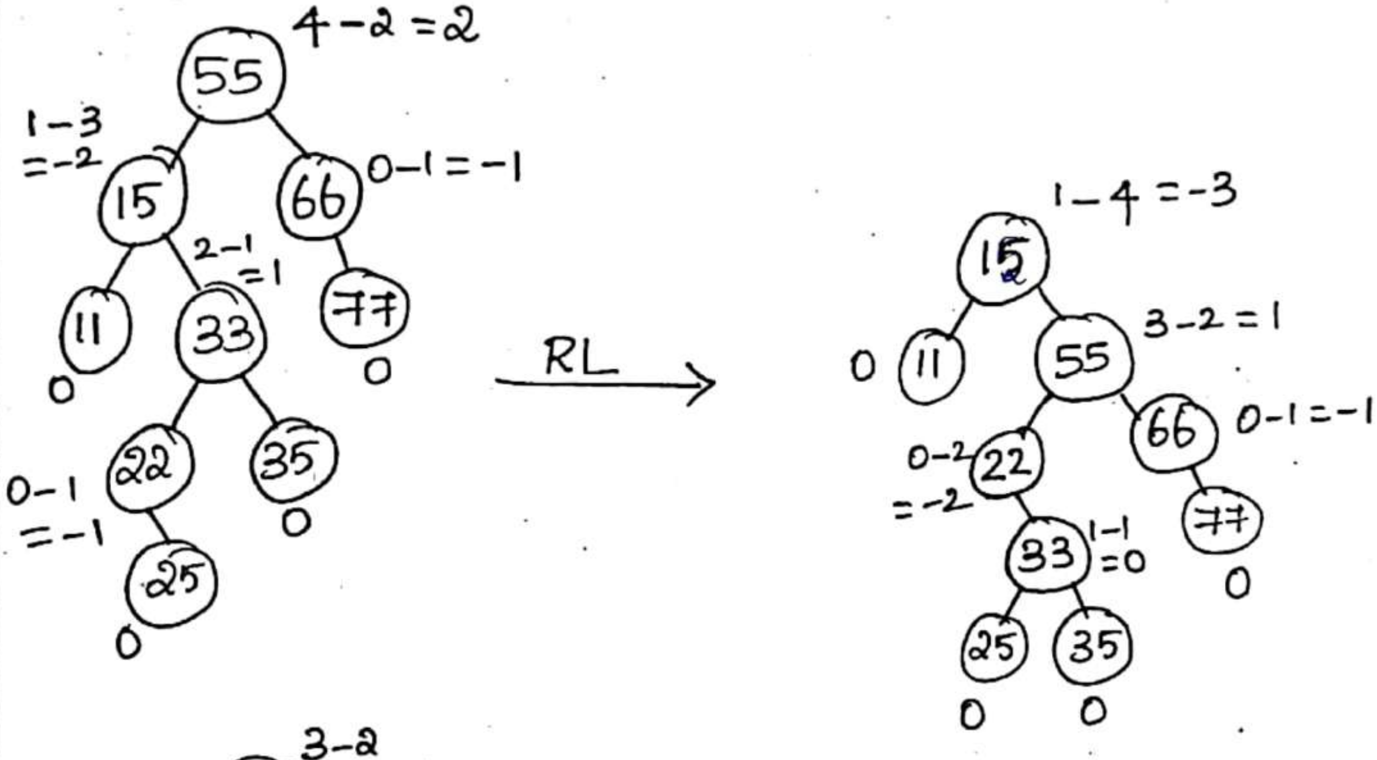
4) Right - left case.

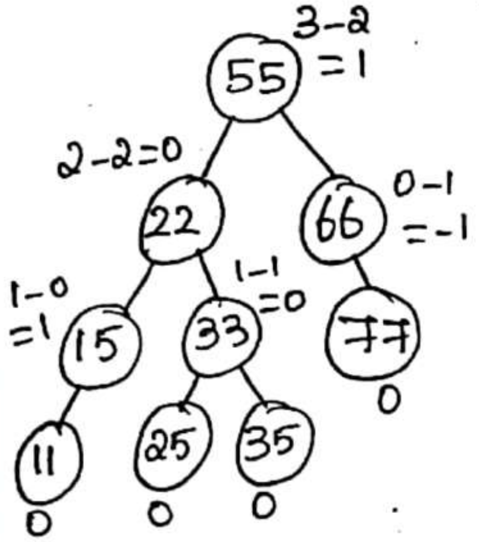
```
if (balance < -1 && key < node -> right -> key)
{
    node -> right = rightrotate (node -> right);
    return leftrotate (node);
}
```

Eg (1) Insert 55, 66, 77, 15, 11, 33, 22, 35, 25.

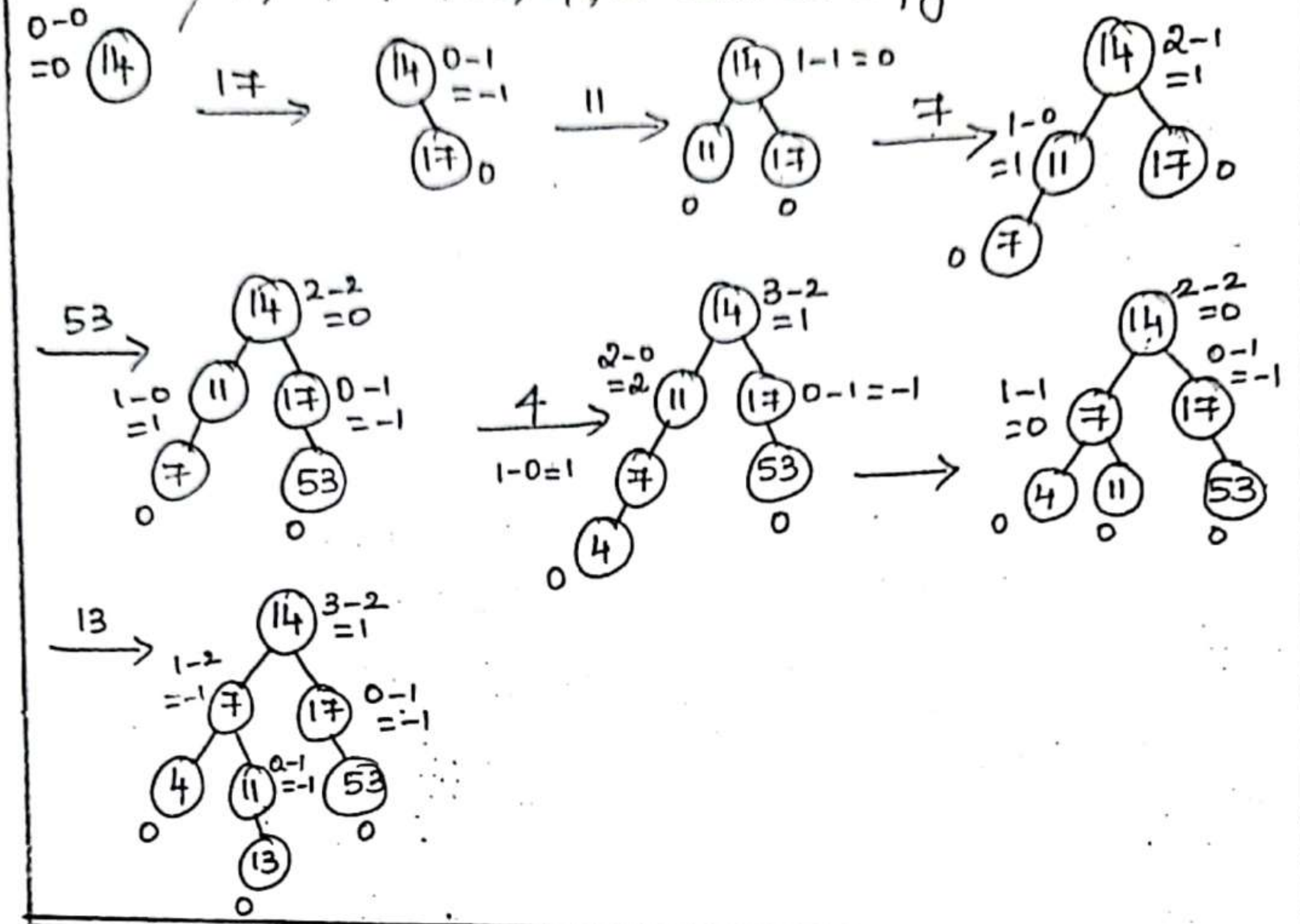



3-2
=1
55
2-2
=0
15
66
1-0
=1
0-1=-1
11
1-1
=0
33
77
0
22
25
35
0
0
0

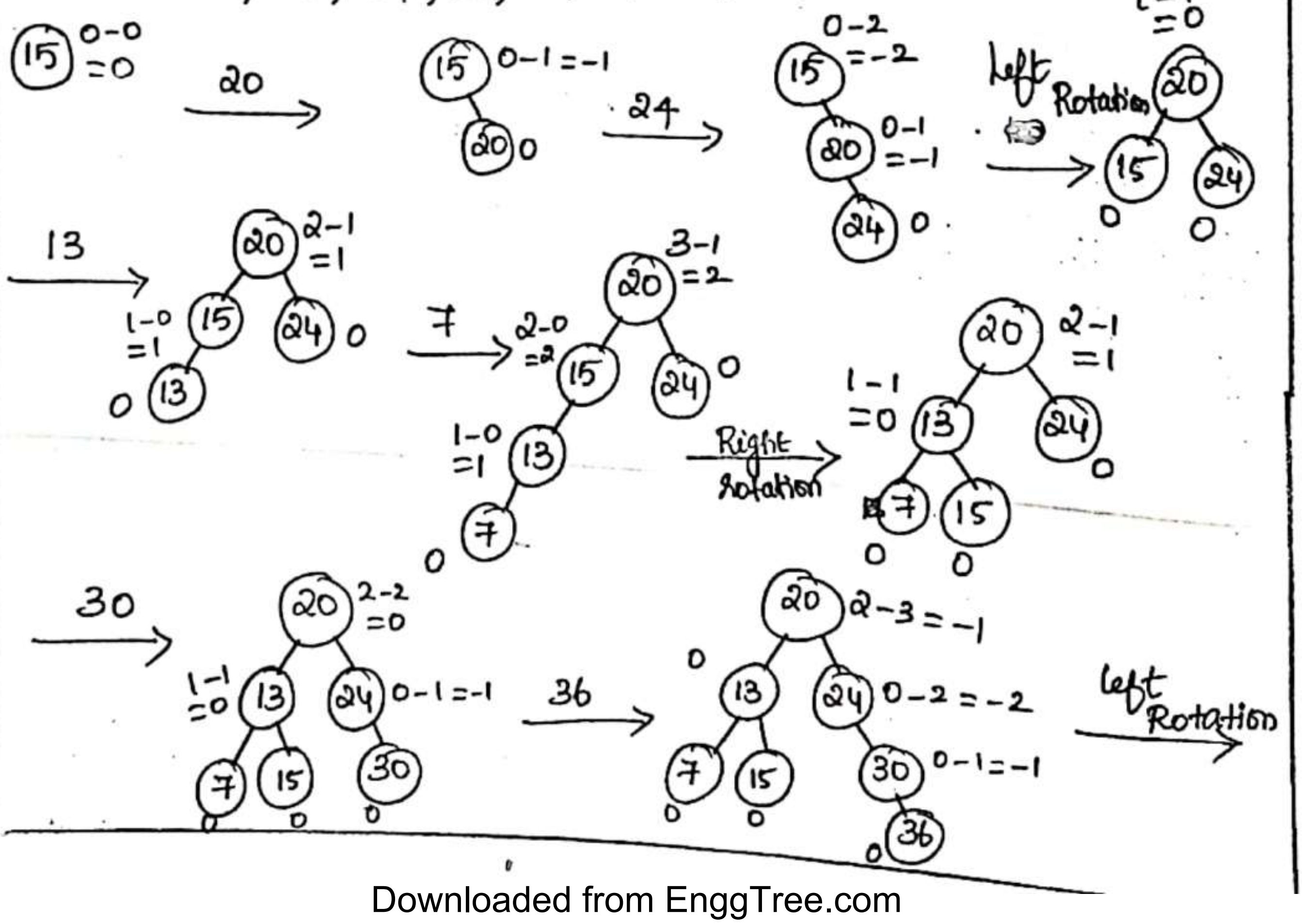

4-2=2
55
1-3
=-2
15
66
0-1=-1
2-1
=1
11
33
77
0
0
22
35
0-1
=-1
0
25
0
RL
1-4=-3
15
0
11
55
3-2=1
66
0-1=-1
0-2
=-2
22
33
1-1
=0
77
0
25
35
0
0

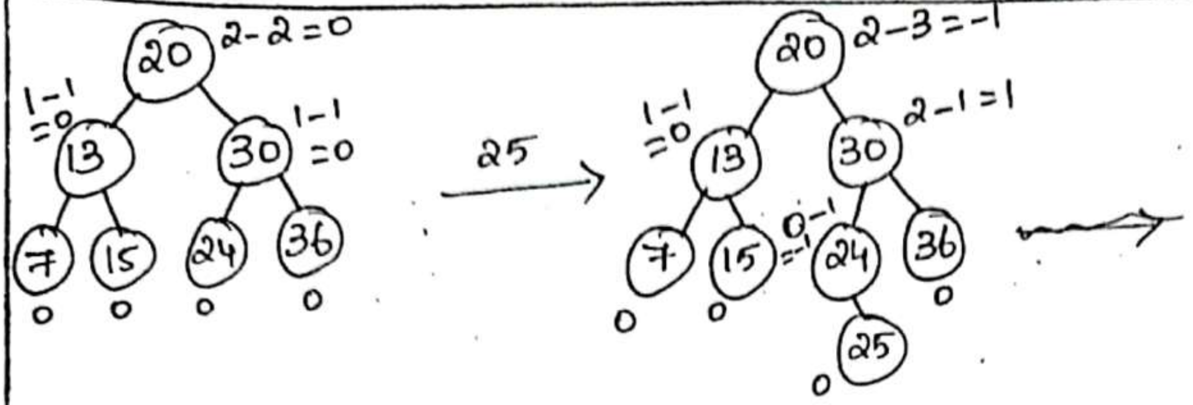

3-2
=1
55
2-2=0
22
66
0-1
=-1
1-0
=1
15
33
1-1
=0
77
0
11
25
35
0
0
0

Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL Tree.

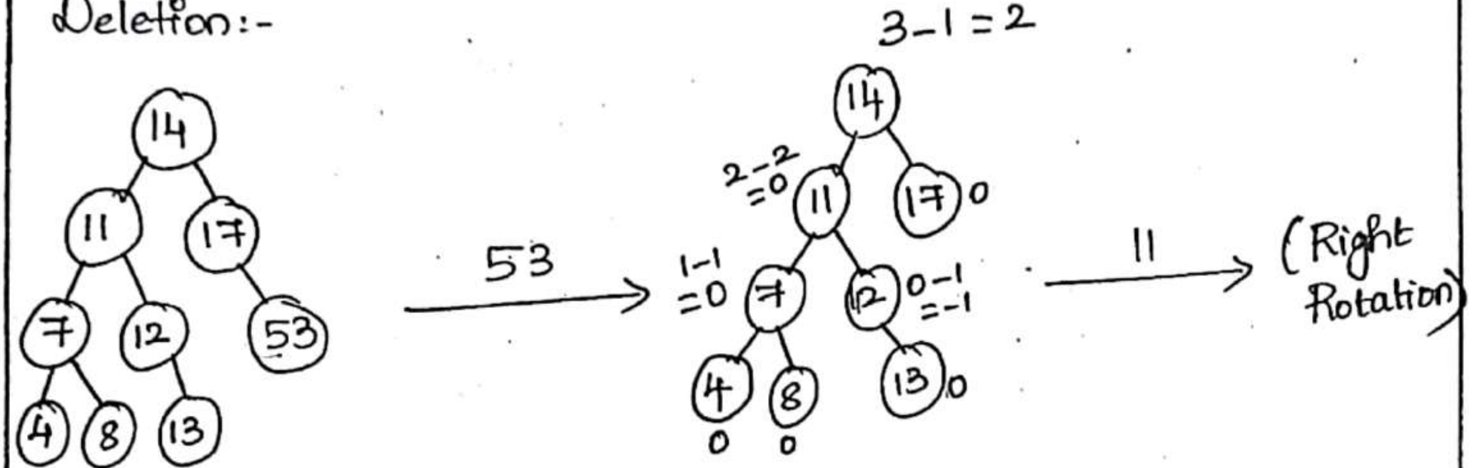


Insert 15, 20, 24, ~~10~~, 13, 7, 30, 36, 25

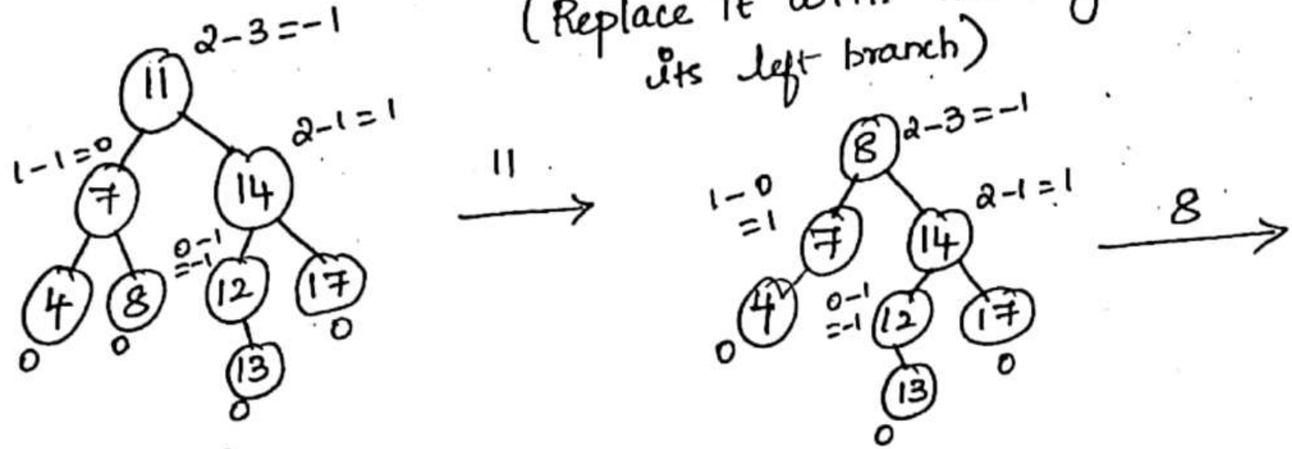


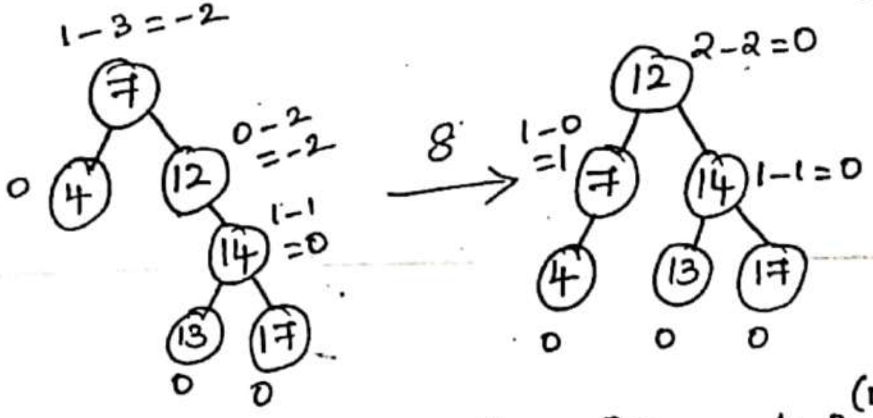


Deletion:-



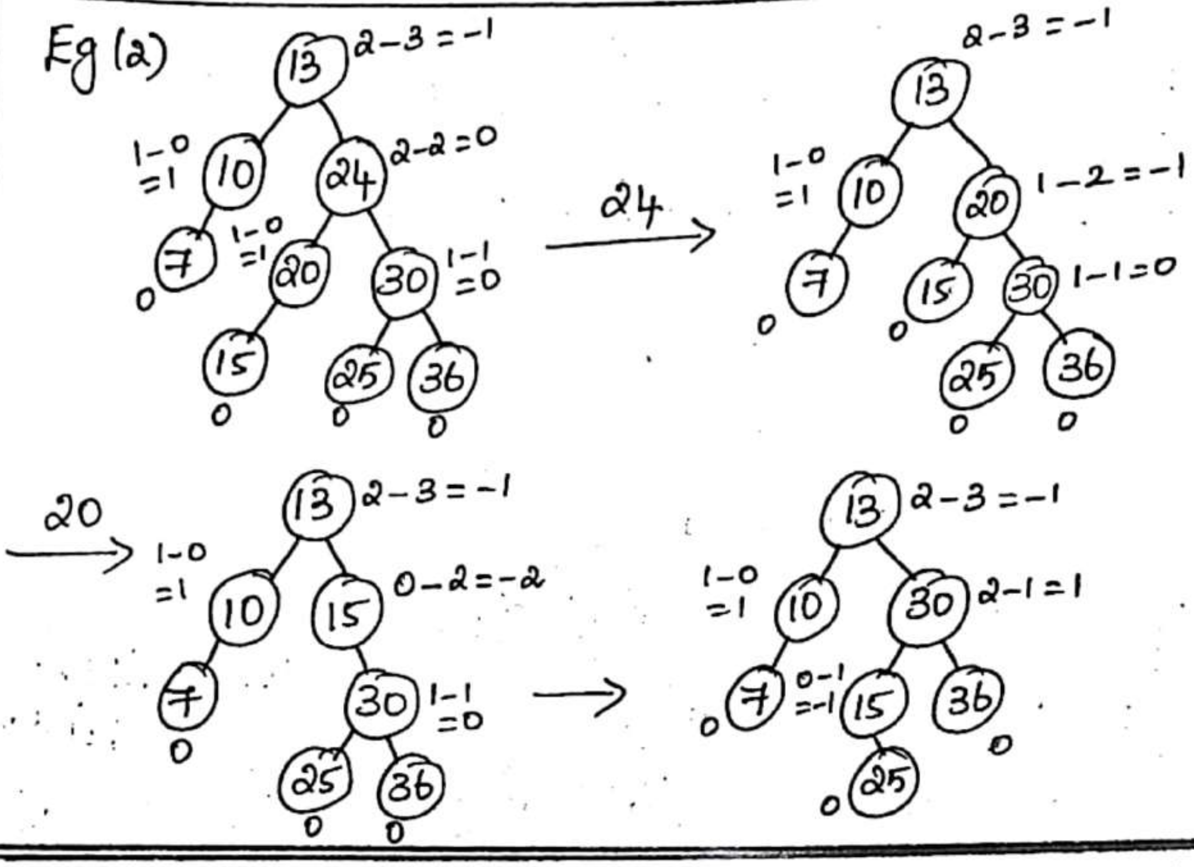
(Replace it with the largest in its left branch)



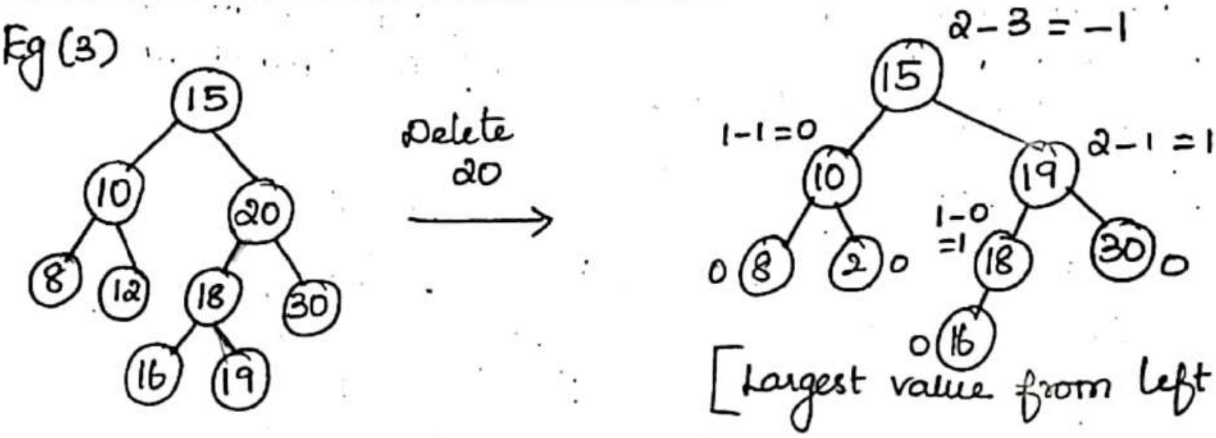


(or) Replace it with ~~maximum~~ (min) minimum value of Right branch.

Eg (2)

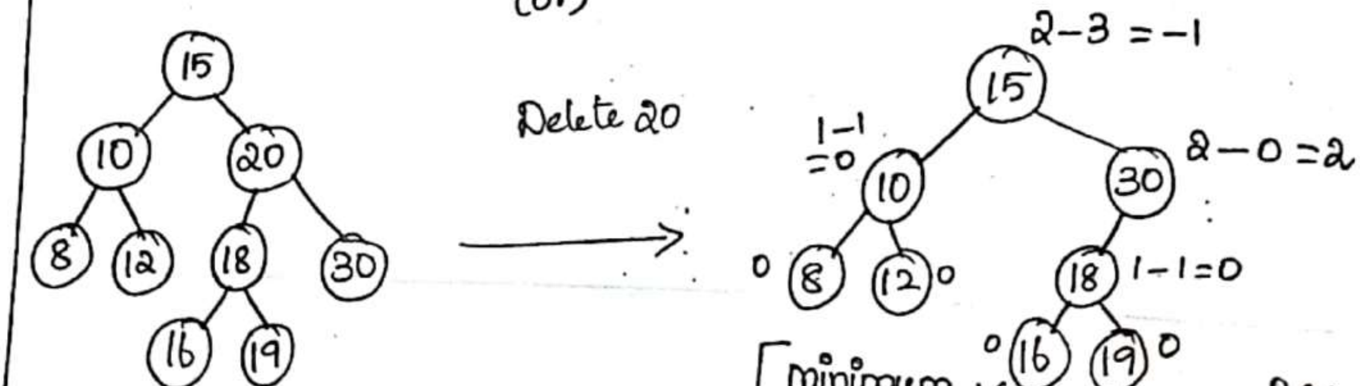


Eg (3)



[Largest value from left subtree]

(or)



[minimum value from right subtree]

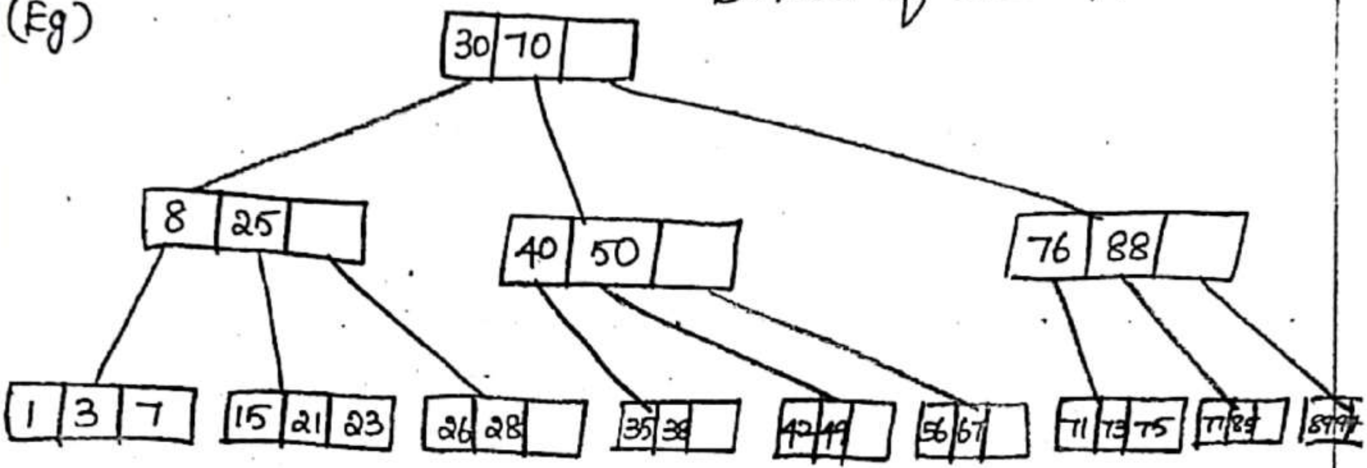
This is not an AVL tree so need rotations.

(9) B-Tree

A B-tree is a self-balancing tree data structure that keeps data structure stored and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children.

(Eg)

B-Tree of Order 4.



B-tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Properties :-

B-tree of order m has the following properties.

Property #1 : All the leaf nodes must be at same level.

Property #2 : All nodes except root must have atleast $\lceil m/2 \rceil - 1$ keys and maximum of $m-1$ keys.

Property #3 : All non leaf nodes except root (ie., all internal nodes) must have atleast $(m/2)$ children.

Property #4 : If the root node is a non-leaf node, then it must have atleast 2 children.

Property #5 : A non leaf node with n-1 keys must have n number of children.

Property #6 : All the keys values within a node must be in Ascending order.

for Eg., B-Tree of order 4 contains, maximum 3 key values in a node and maximum 4 children for a node.

Eg; m-ary search tree
Order m=3.
max. no of child = 3(m)
max. no of key = 2(m-1)
min. no of key = (m/2)-1 = 1
min. no of child = (m/2) = 2.

Operations on a B-Tree.

1) Search
2) Insertion
3) Deletion.

Search :

Step 1 : Read the search element from the user.

Step 2 : Compare, the search element with first key value of root node in the tree.

Step 3 : If both are matching, then display "Given node found !!!" and terminate the function.

Step 4 : If both are not matching, then check whether search element is smaller (or) larger than that key value.

Step 5 : If search element is smaller, then continue the search process in left subtree.

Step 6 : If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match (or) comparision completed with last key value in a leaf node.

Step 7 : If we completed with last key value in a leaf node, then display "Element is not found", and terminate the function.

Insertion :- In a B-tree, the new element must be added only at leaf node. That means, always the new keyvalue is attached to leaf node only.

Step 1 : Check whether tree is Empty.

Step 2 : If tree is Empty, then create a new node with new key value and insert into the tree as a root node.

Step 3 : If tree is Not Empty, then find a leaf node to which the new key value be added using BST logic.

Step 4 : If that leaf node has an empty position, then add the new key value to that leaf node by maintaing ascending order of key value within the node.

Step 5 : If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6 : If the spiting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Definition :- (Order of a B-tree)

According to Knuth's definition, a B-tree of order m is a tree which satisfies the following properties.

(i) Every node has atmost m children.

(ii) A non-leaf node with K children contains K-1 keys.

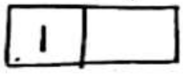
(iii) All leaves appear in the same level.

Eg; Construct a B-tree of order 3 by inserting numbers from 1 to 10.

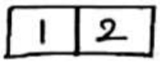
max child = m = 3
min child = m/2 = 3/2 = 1.5 = 2
max key = m-1 = 2
min key = m/2-1 = 1

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

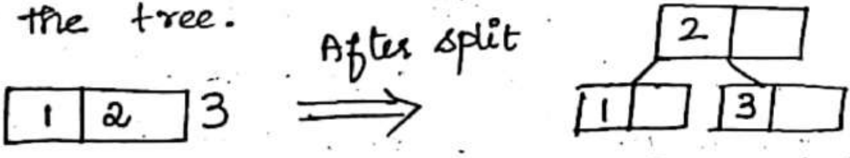
Insert 1 : Since '1' is the 1st element inserted into a new node, it act as the root node.



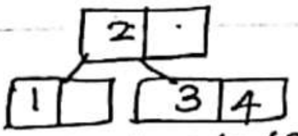
Insert 2 : We have only one node that acts as root and also leaf. This leaf node has an Empty position. so add 2.



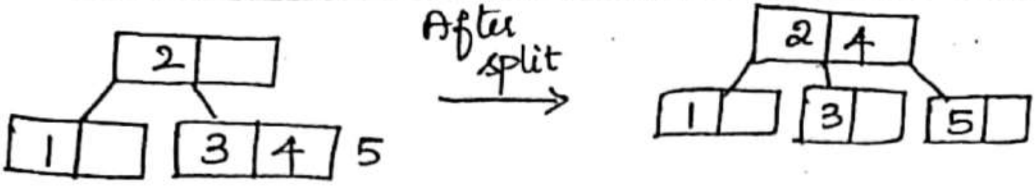
Insert 3 : It doesn't have Empty position. So split that node and now middle value becomes a new root nodes for the tree.



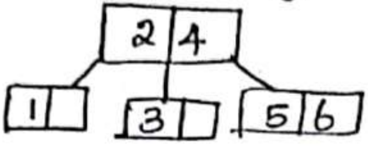
Insert 4 : 4 > Root node (2). so insert at right of (2), in the Empty position.



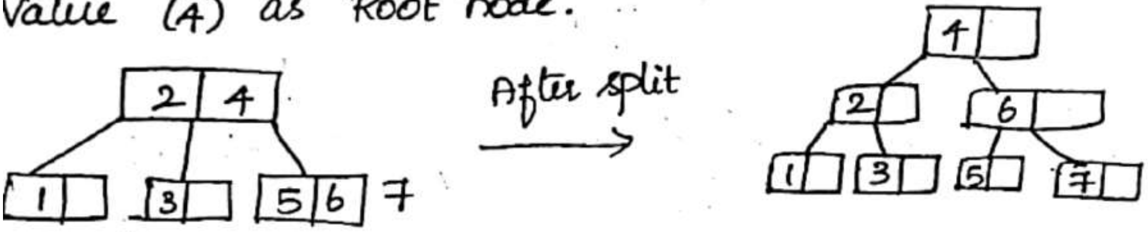
Insert 5 : 5 > Root node (2). Leaf node is already full, so split that node, now send middle value (4) as parent node, near the empty position in parent node. So new element (5) is added as a new leaf node.



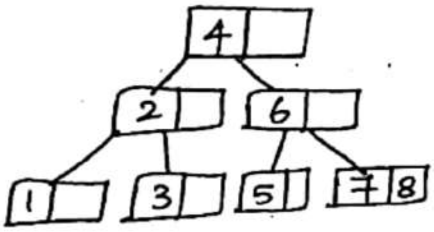
Insert 6 : 6 > Root (2) and (4). Move to right of 4, and add into the empty position.



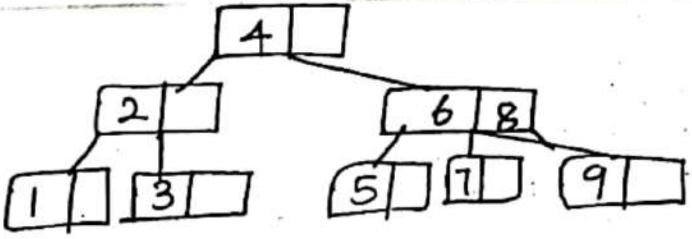
Insert 7 : 7 > Root (2) and (4). Move right and it is already full, split the node by sending middle value (6) to its parent node. But parent node is also full. so again split the node (2) and (4) by sending middle value (4) as Root node.



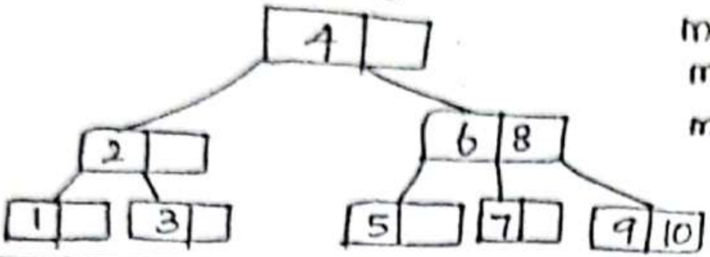
Insert (8) : 8 > Root node (4). Move right and insert at Empty position.



Insert (9) : 9 > Root node (4). Move right of (4). 9 > 6, move right (6). The leaf node is already full so split node by sending middle value (8) as parent node. The parent node (6) has Empty position. So Insert (9).



Insert (10) : 10 > Root node (4). Move right.
10 > Root node (6). Move right and insert at empty position.



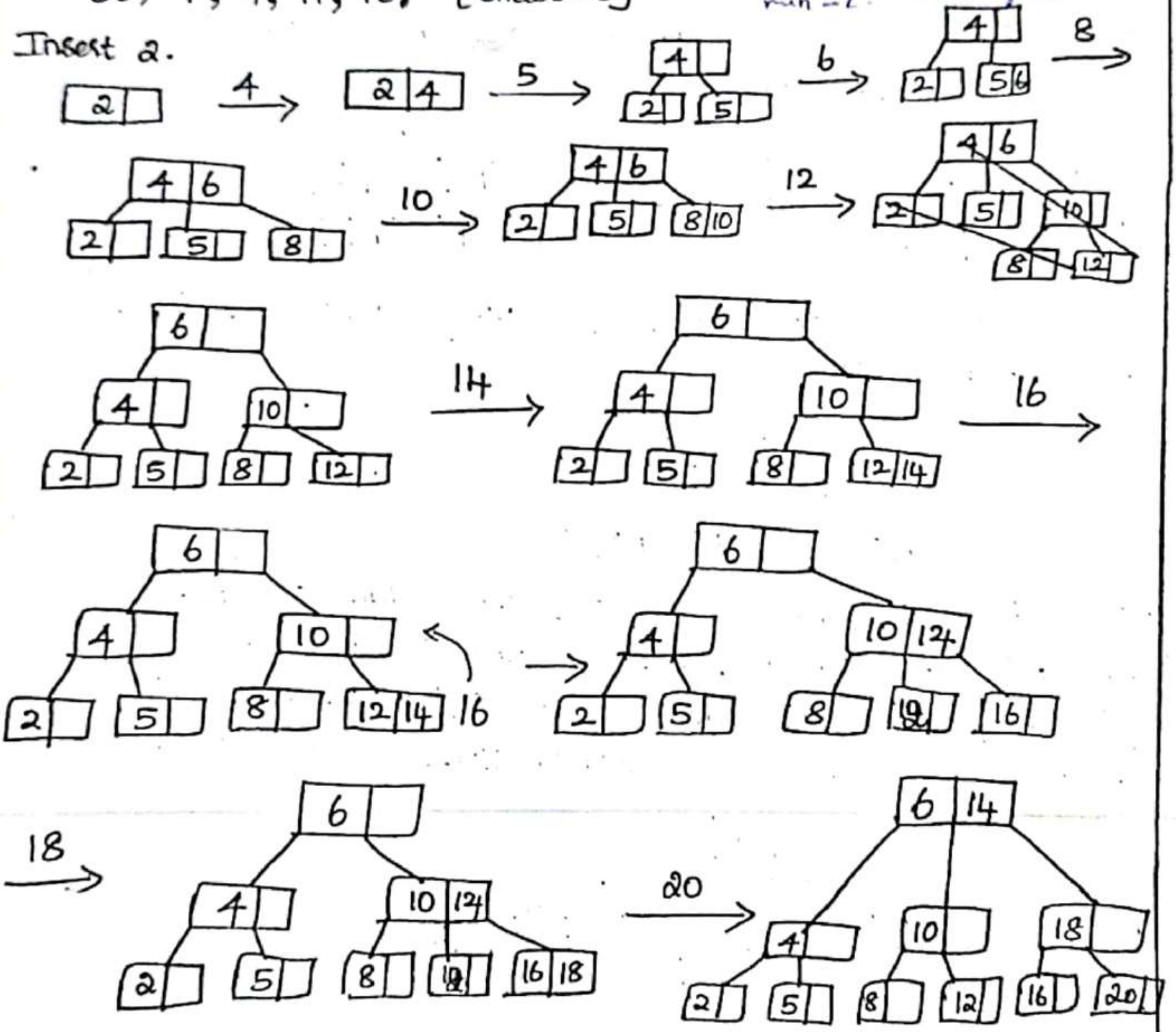
Order 3.
max. no. of child = m = 3
min. no. of child = (m/2) = 2
max. no. of key = (m-1) = 3-1 = 2
min. no. of key = (m/2-1) = 2-1 = 1.

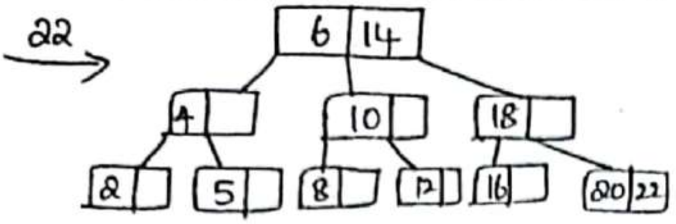
Eg (2) Construct B-tree by inserting 2, 4, 5, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 7, 9, 11, 13, [Order - 3]

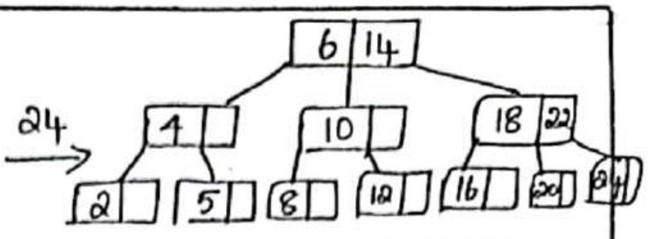
max - 3
min - 2

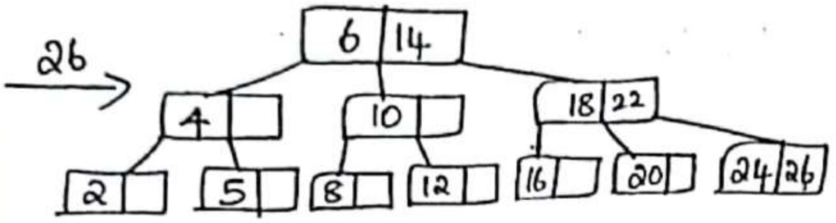
max key = 2
min key - 1

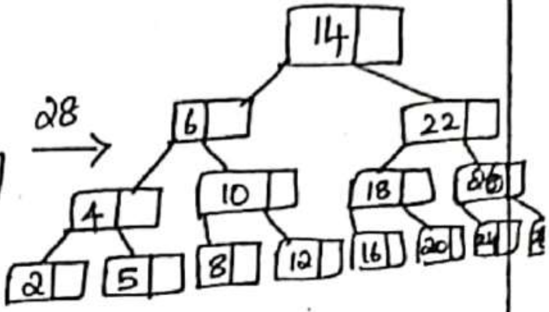
Insert 2.

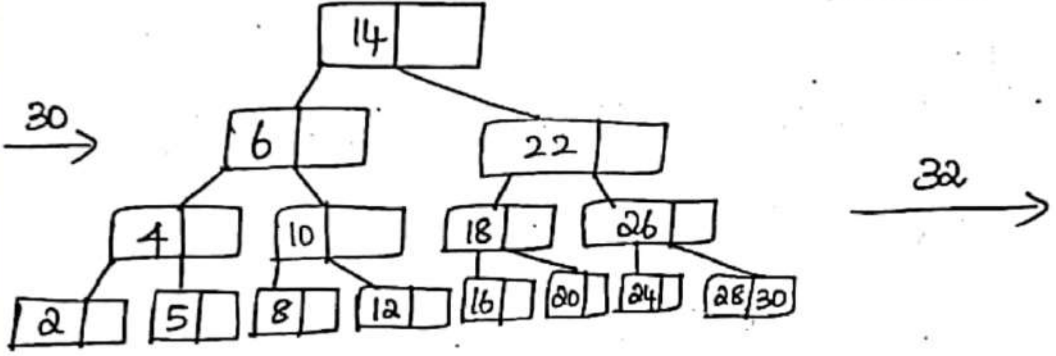


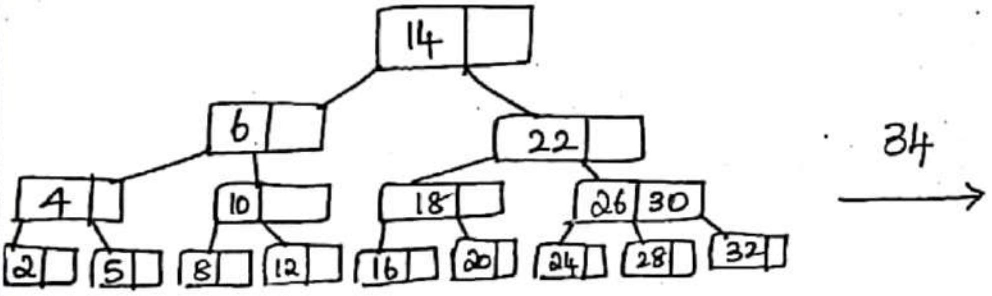


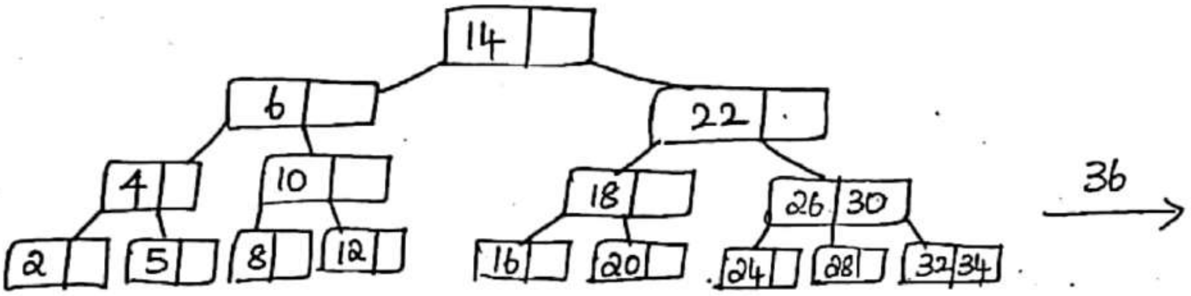


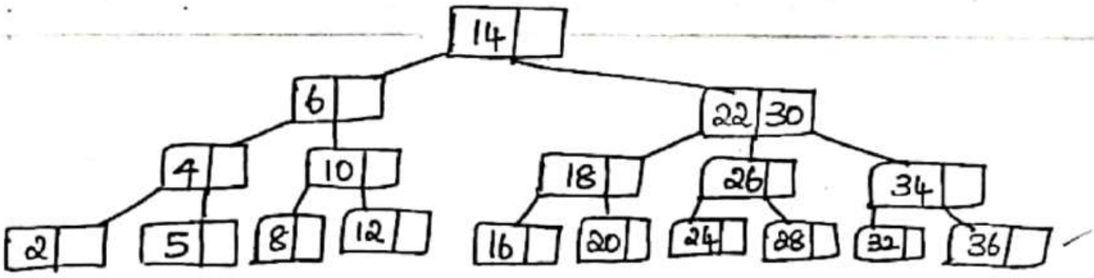






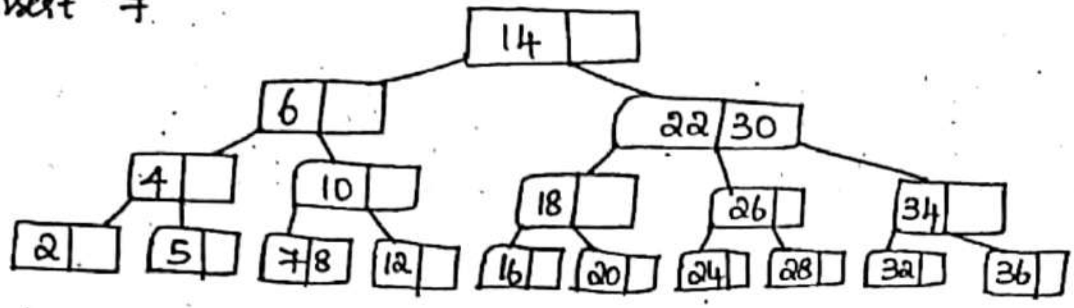




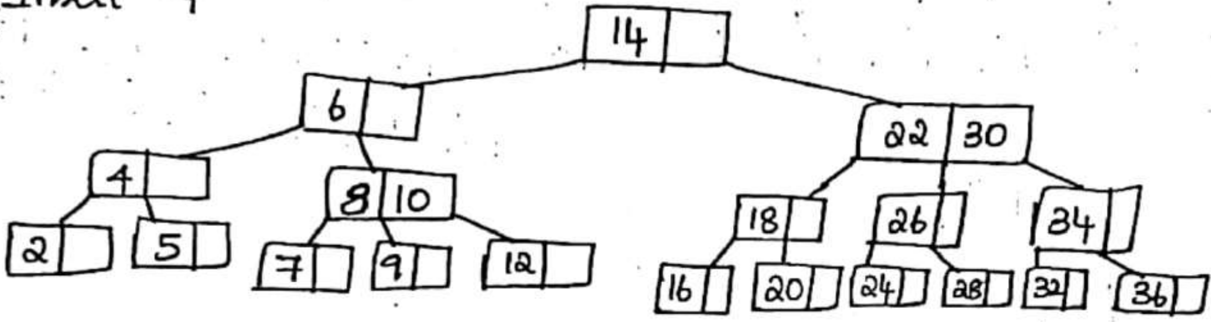


Key → ~~Levels~~ Node.
Child → Levels.

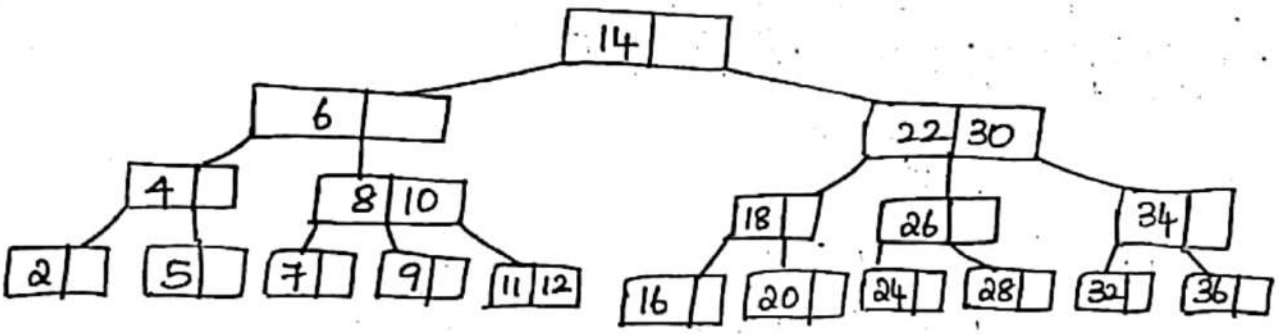
Insert '7'



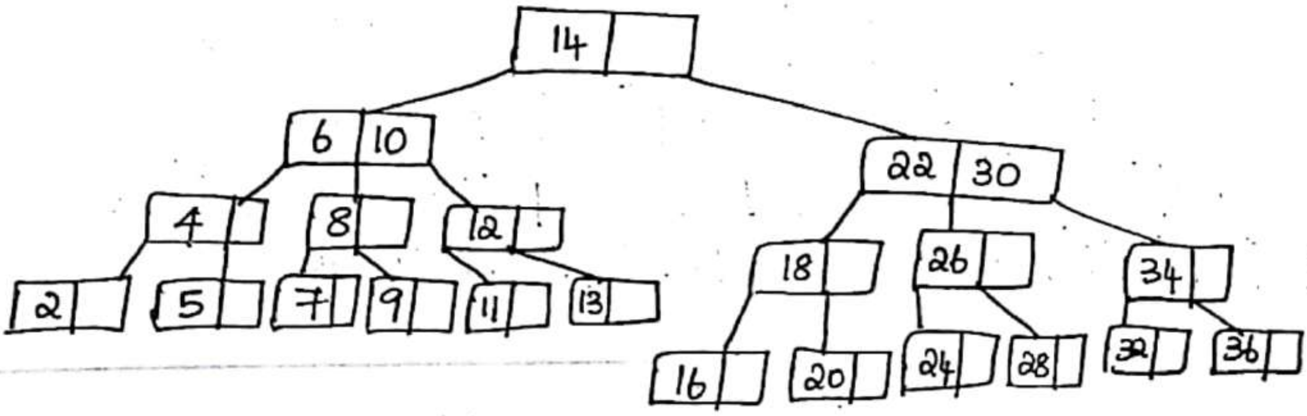
Insert '9'



Insert 11.



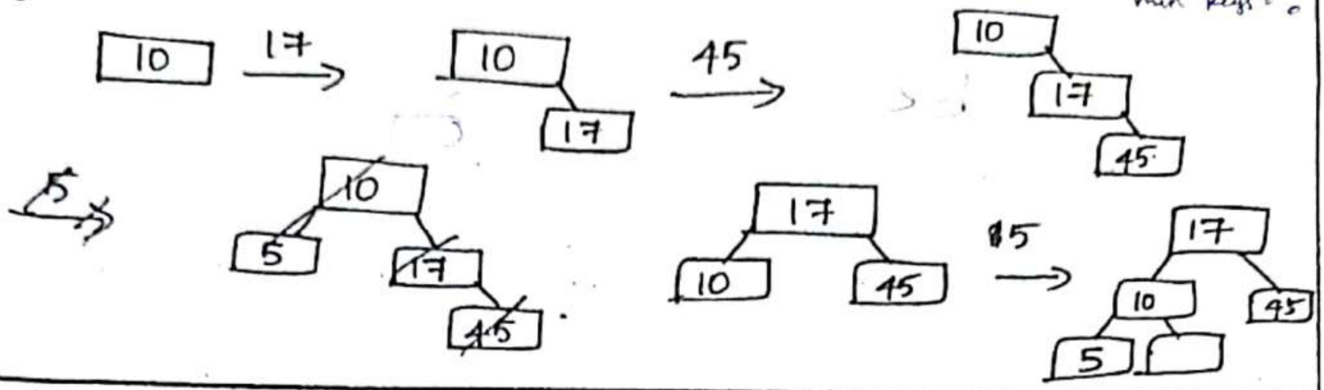
Insert 13.



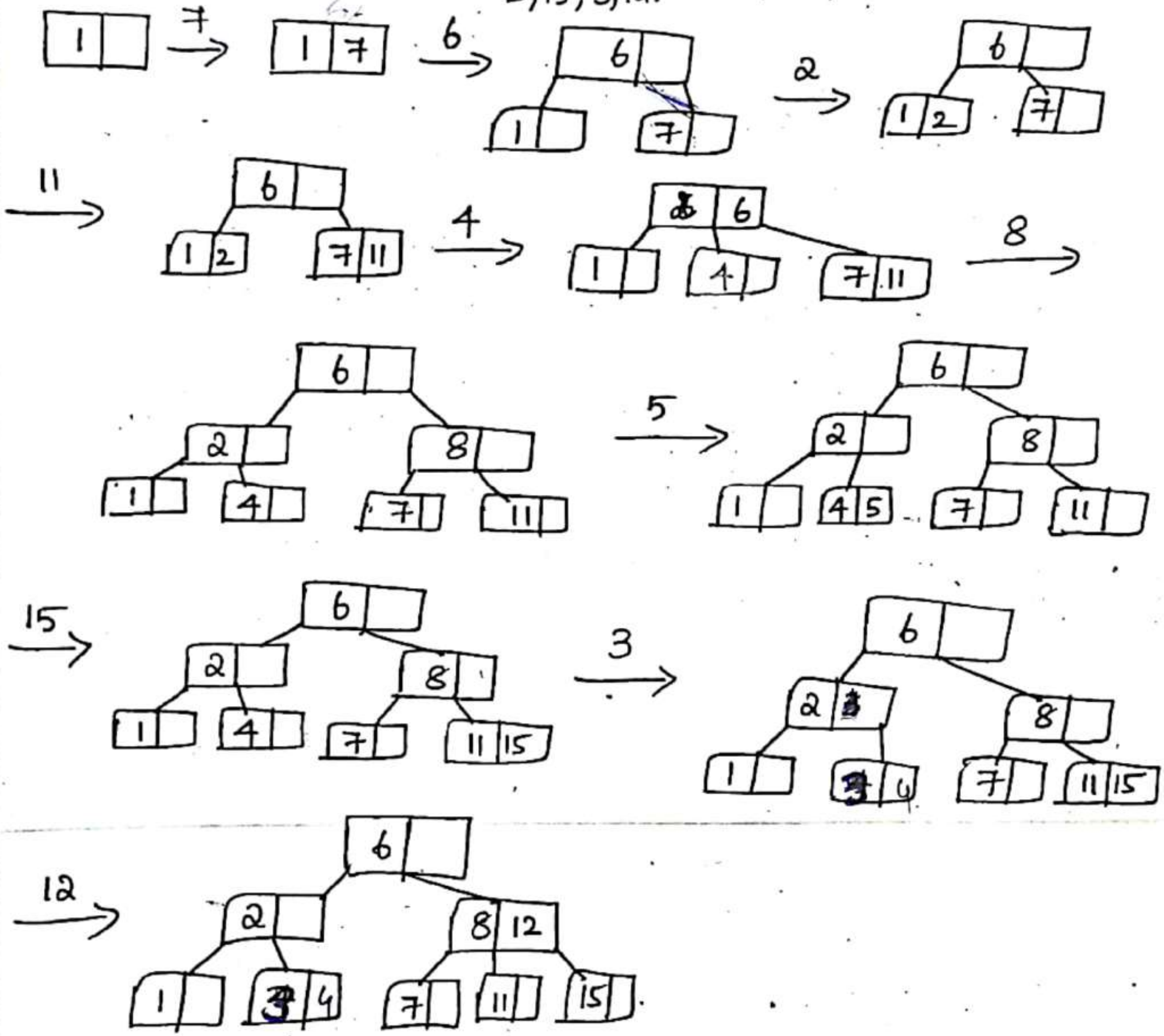
Eg Insert F, S, Q, K, C, L, H, T, V, W, K, R, V
of Order (2).

max child = 2
min = 1
max keys - 1
min keys = 0

Eg Insert 10, 17, 45, 5 in a B-Tree of Order (2)



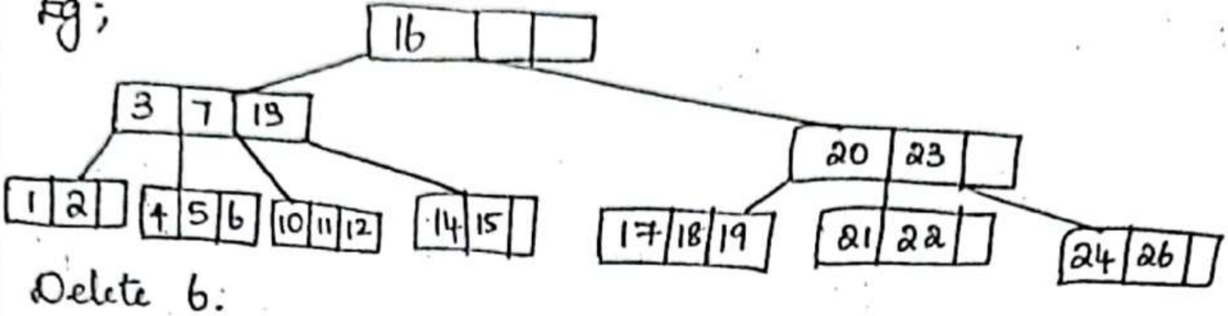
Insert 1, 7, 6, 2, 11, 4, 8, 5, 15, 3, 12. (order 3) ie., m=3.



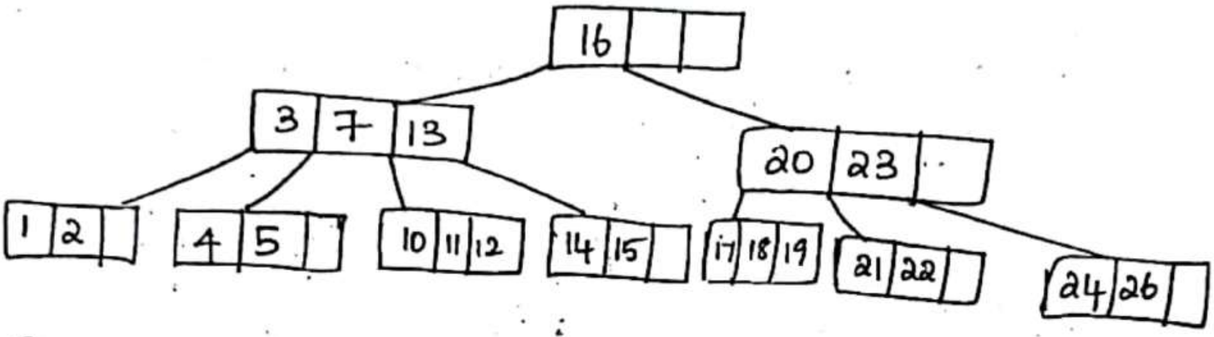
Deletion (B-Tree).

Case 1: If Key K is in node x, and x is a leaf, simply delete K from x. [No child]

Eg;

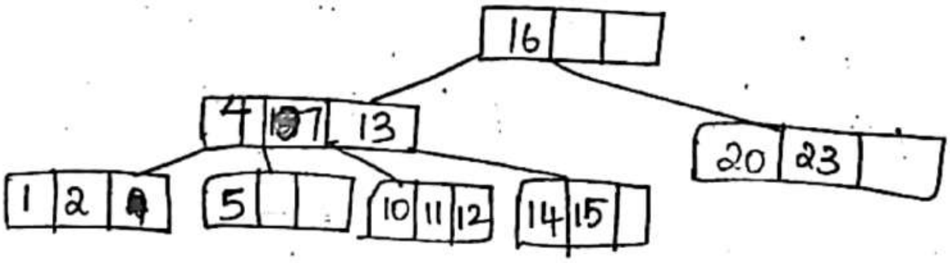


Delete 6.

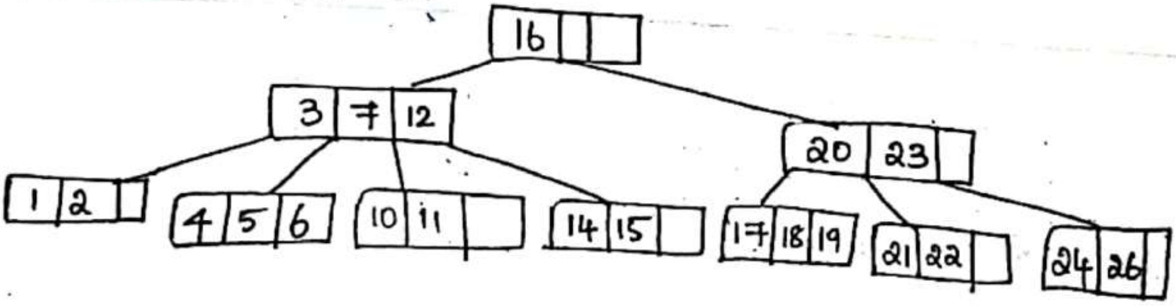


Case 2: If Key K is in node x, and x is an internal node, there are three cases to consider.

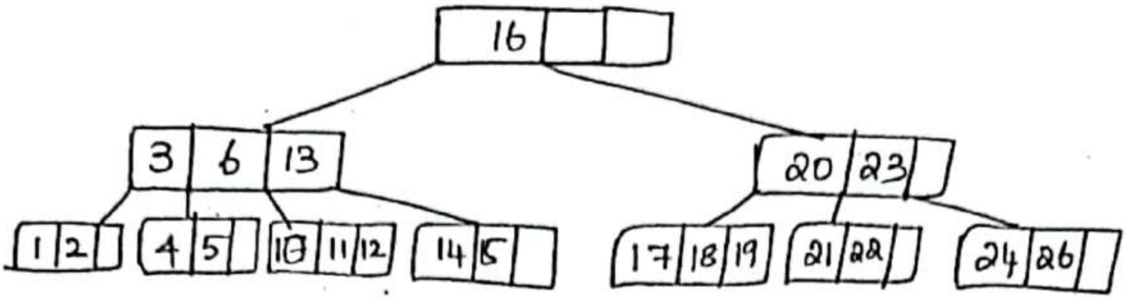
Case 2(a): Delete 3.



Case 2(b): Delete 13.



Case 2(c): Delete 7.



(or)

Eg(2): (N/D-2018)

Insert 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8 to form a B-Tree and delete 2, 21, 10, 3 and 4 of order '4'.

Given; Order '4'.

Key → Node
child → Levels.

max. no. of child = (m) ⇒ 4

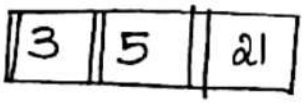
min. no. of child = $(m/2) = (4/2) \Rightarrow 2$

max. no. of key = $(m-1) = (4-1) \Rightarrow 3$

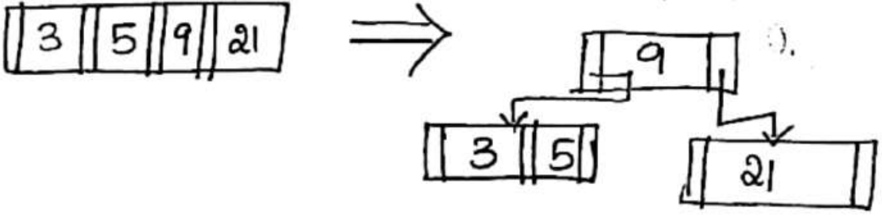
min. no. of key = $(m/2 - 1) = (4/2 - 1)$

$= (2-1) \Rightarrow 1$.

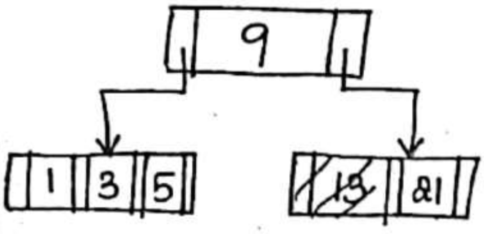
Insert 5, 3, 21. (Arrange in ascending order)



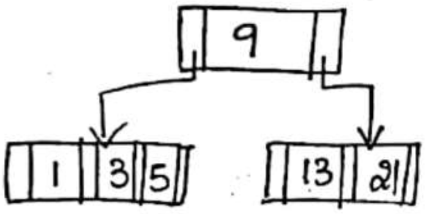
Insert 9. (make middle nodes as root, if there are two values in the middle choose biggest nodes as middle, ie., as root)



Insert 1.

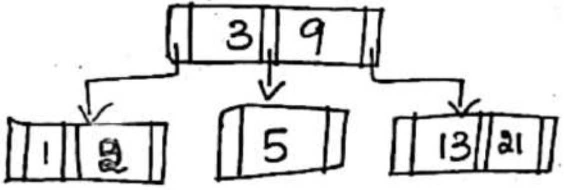


Insert 13.

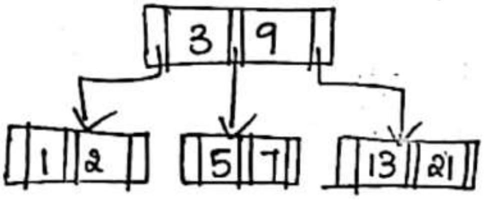


Insert 2.

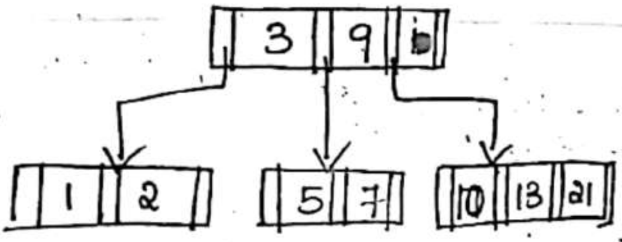
1 2 3 5 → Root



Insert 7.

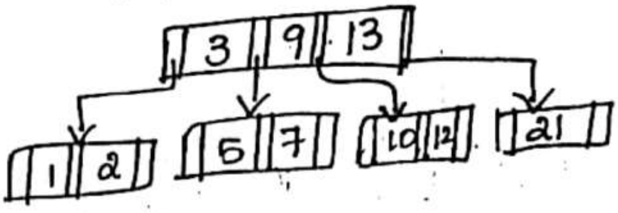


Insert 10.

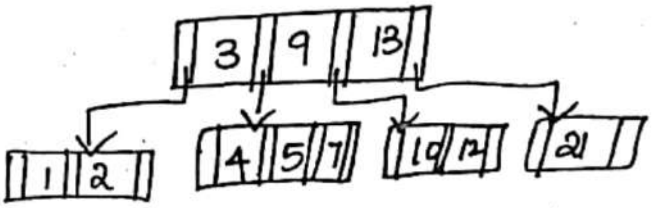


Insert 12. → Root

7, 12, 13, 21



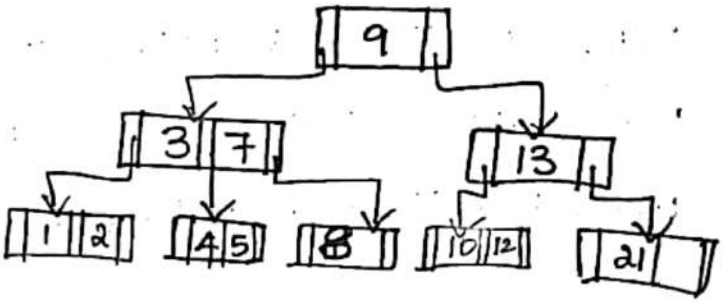
Insert 4.



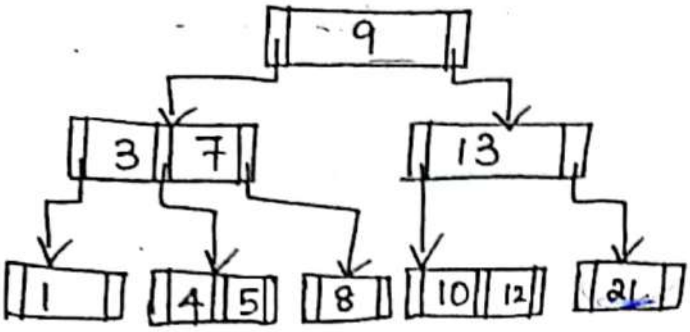
Insert 8. → Root

4, 5, 7, 8 → Root

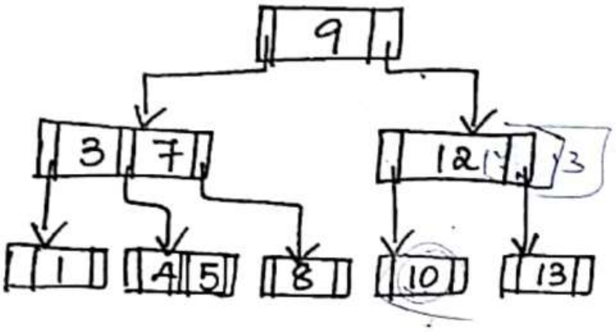
3, 7, 9, 13



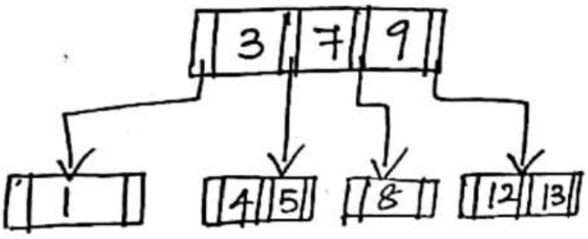
Delete 2.



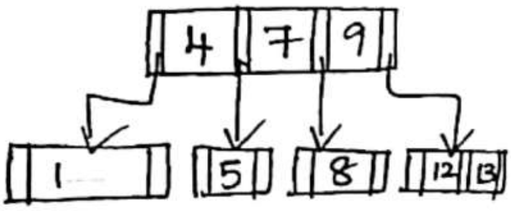
Delete 21.   10, 12 13 → Node



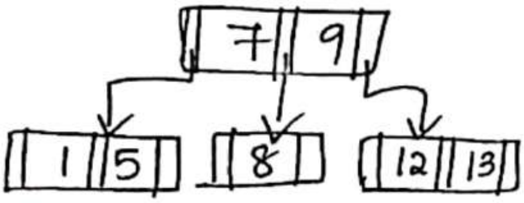
Delete 10.



Delete 3.



Delete 4.



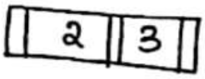
Eg (3): Construct a B-tree with order m=3 for key values 2, 3, 7, 9, 5, 6, 4, 8, 1 and delete the values 4 and 6.

Given; Order m = 3.

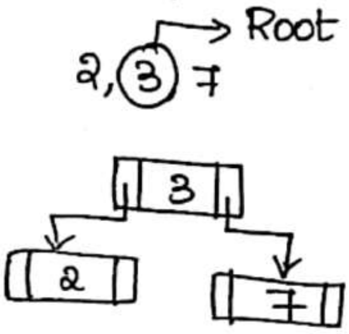
Key → node
child → levels.

max. no. of child = (m) ⇒ 3.
min. no. of child = $(m/2) = (3/2) \Rightarrow 1.5 \Rightarrow 2$.
max. no. of key = $(m-1) = (3-1) \Rightarrow 2$
min. no. of key = $(m/2 - 1) = (3/2 - 1) = 2 - 1$
⇒ 1.

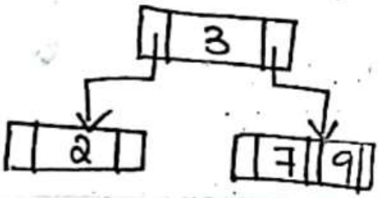
Insert 2, 3



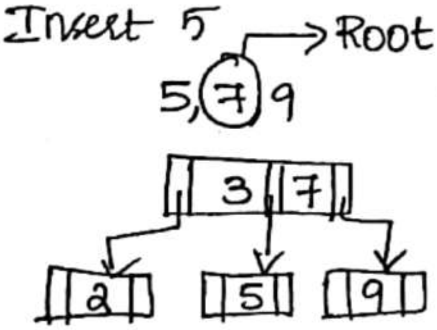
Insert 7.



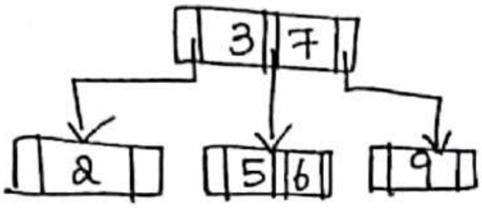
Insert 9.



Insert 5



Insert 6 :-

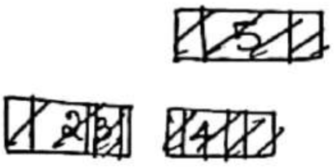


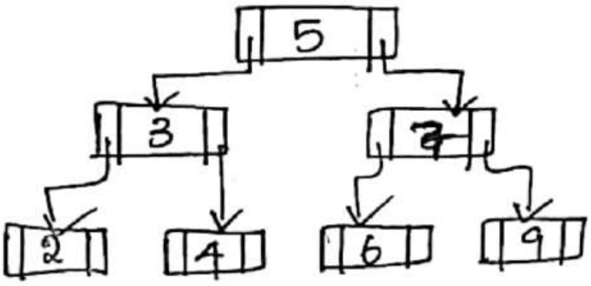
Insert 4 :- → Root

4, ⑤, 6

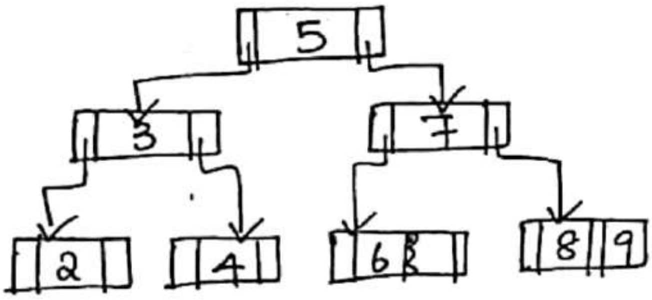
→ Root

3, 5, 7

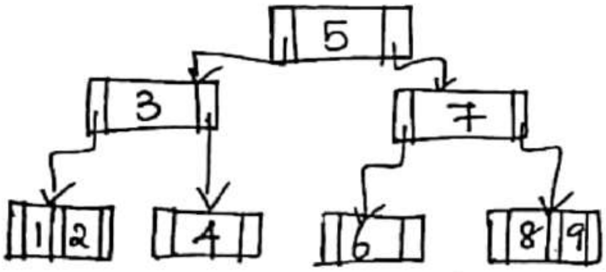




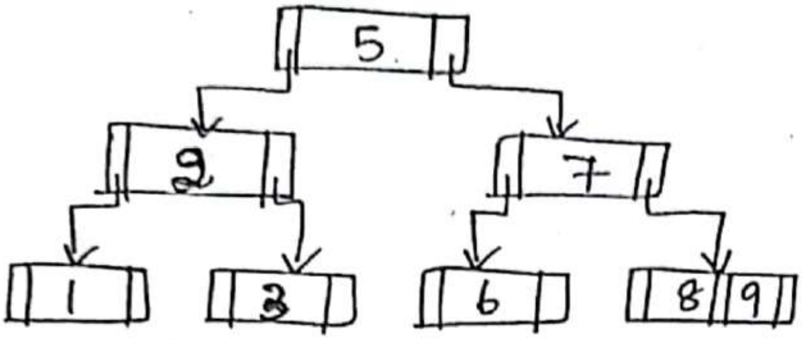
Insert 8.



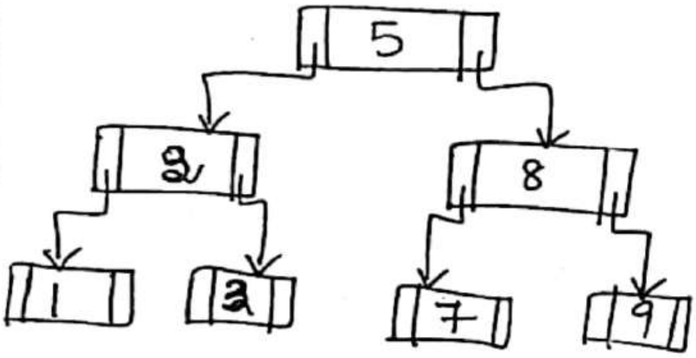
Insert 1.



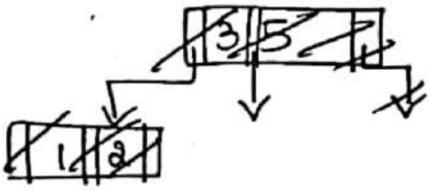
Delete 4.

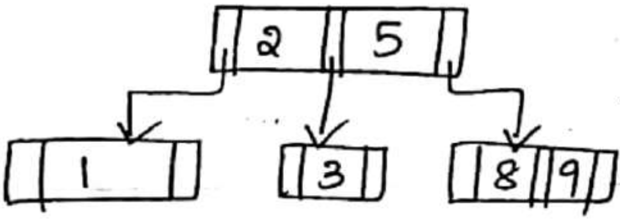


Delete 6.

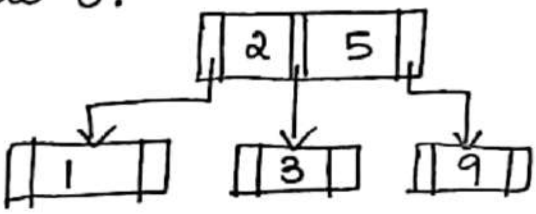


Delete 7.

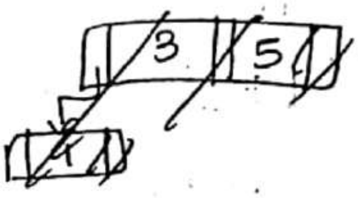


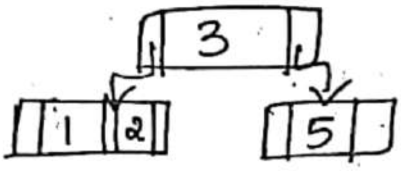


Delete 8.



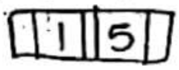
Delete 9.



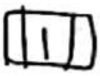


Delete 3.

Delete 2.



Delete 5.



Delete 1.

Empty tree.

(10) B^+- Tree.

A B^+ tree is a balanced tree in which every Path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between $[n/2]$ and $[n]$ children, where n is fixed for a particular tree.

In a B^+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in internal nodes.

All the leaf nodes are interconnected for faster access.

Key may be duplicated; every key to the right of a particular key is >= to that key.

B^+ Tree characteristics.

1) Data records are only stored in the leaves.
2) Internal nodes stores just keys.
3) keys are used for directing a search to the Proper leaf.
4) If a target key is less than a key in an internal node, then the pointer just to its left is followed.

5) If a target key is greater (or) equal to the key in the internal node, then the pointer to its right is followed.

6) B$^+$ tree combines features of ISAM (Indexed Sequential Access Method) and B-trees.

B$^+$ tree Node Structures :

P_1	K_1	P_2	....	P_{n-1}	K_{n-1}	P_n

→ Contains upto $n-1$ search key values (K_1 to K_{n-1}) and n pointers ($P_1, P_2, \ldots, P_n$)

→ Search key values within a node are kept in sorted order, thus, if $i < j$, then $K_i < K_j$.

Special cases :

↳ Root node is the only node in the tree. i.e., Root node becomes the leaf node.

Min. no. of children → 1 → Pointing to the only single link of records.

Max. no. of children → b-1 → same as that of a leaf node.

Let 'b' be the branching factor (or) order of B$^+$ tree.

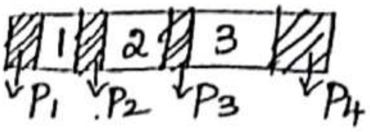
$\lceil \rceil \rightarrow$ celing function

Non-Leaf Node :-

$\lceil b/2 \rceil \leq m \leq b.$

Let K represent the no. of search keys in a node.

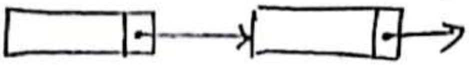
$\Rightarrow \lceil b/2 \rceil - 1 \leq K \leq b-1.$



No. of pointers (or) children =

No. of Search Keys +1.

LEAF NODE :-



Leaf nodes occuring at the last level of the B^+ tree use their own pointer (last one) to connect with each other to facilitate the sequential access at leaf level.

↳ Max. no. of children : $\lceil b/2 \rceil - 1 \leq m \leq b-1$

(Pointers ~~Bam~~ remain same)

↳ Max. no. of Search keys :

$\lceil b/2 \rceil - 1 \leq K \leq b-1.$

Since no. of pointers are same, only one of them in each node is used for connecting each other among leaf nodes.

ROOT NODE :

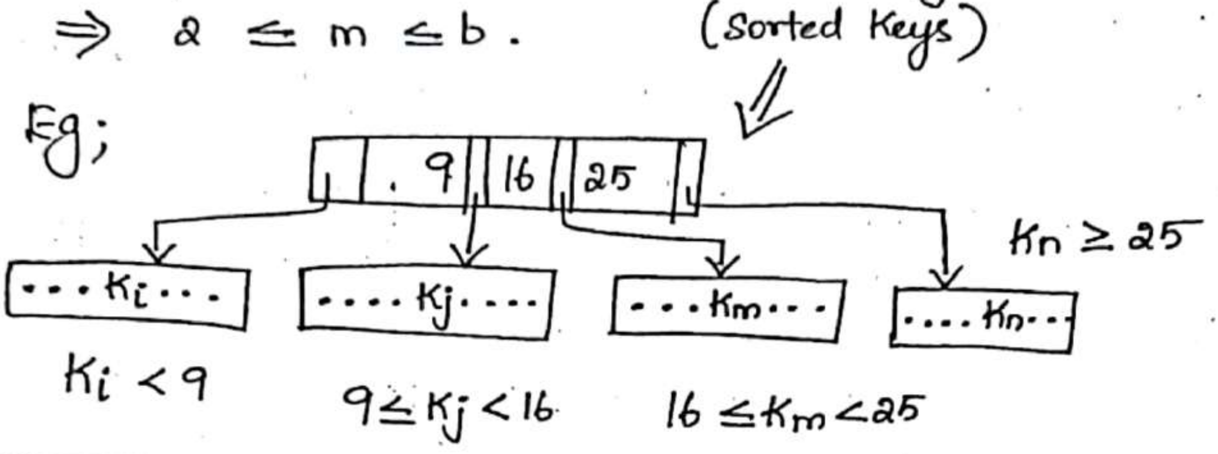
Max. no. of children :

↳ b (counted as Internal node only)

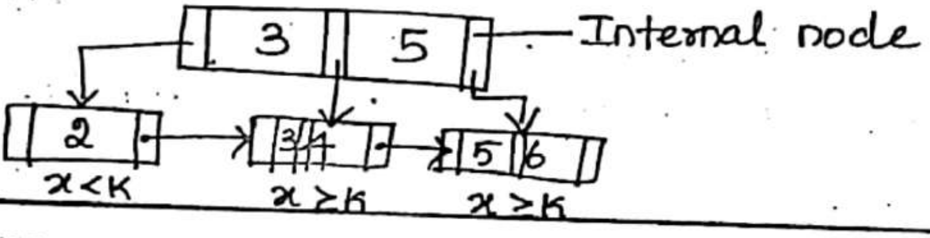
↳ Min. no. of. children → 2 (It must have atleast 2 pointers in case it has only one search key)

⇒ $2 \leq m \leq b$.

Eg;

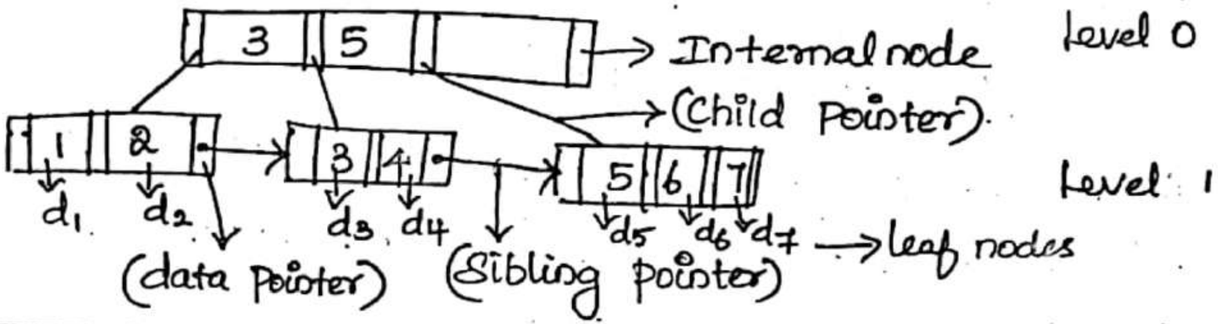


B⁺ tree occupy a little more space than B-tree.



B^+ tree = Index Sequential access method (features)
+
B- tree (freatures)

Eg;



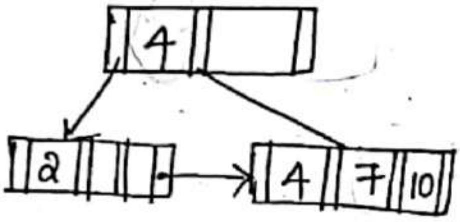
Insert 2, 4, 7, 10, 17, 21, 28 Order m=4;

Given : Order m = 4

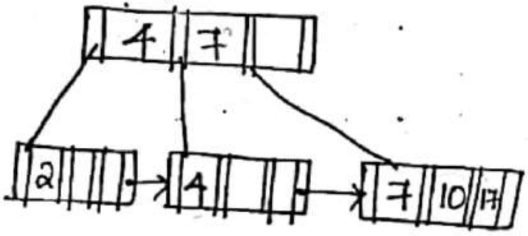
Key ⇒ $(m-1) = (4-1) = 3$.

Insert 2, 4, 7

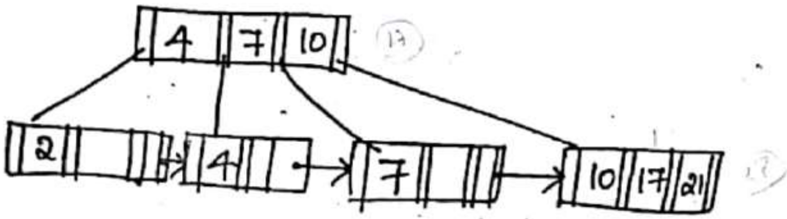
Insert 10



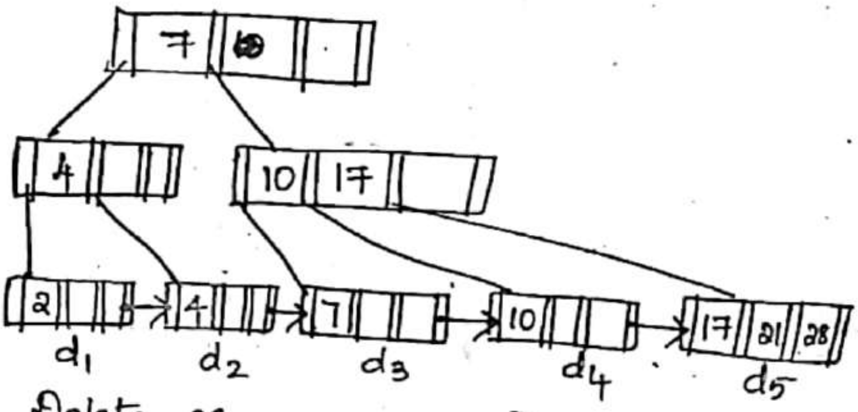
Insert 17



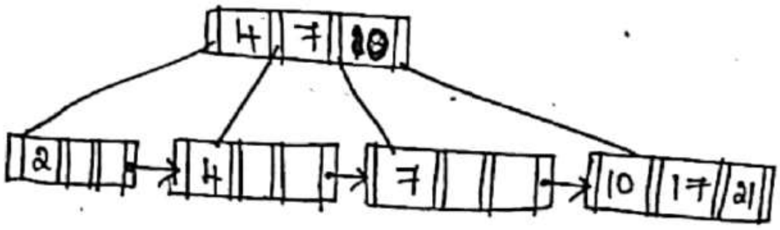
Insert 21



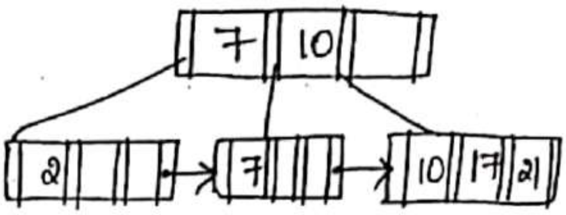
Insert 28



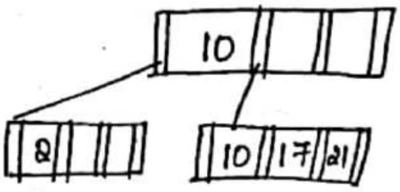
Delete 28



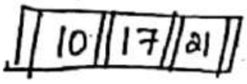
Delete 4 :



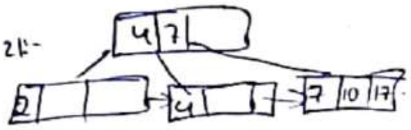
Delete 7 :



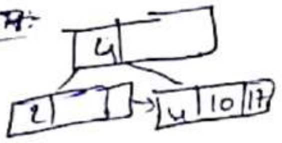
Delete 2 :



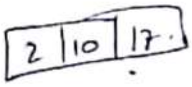
Delte 21:-



Delete 7:-



Delete 4:-



B+ tree : Insertion.

① If data page full (No) and Index page full (No), then Place the record in sorted position in the appropriate leaf page.

② If data page full (yes) and Index page full (no), then split the leaf page, Place middle key in the index page in sorted order. Left leaf contains records with keys below the middle key and Right leaf page contains records with keys equal to (or) greater than the middle key.

③ If data page full (yes) and Index page full (yes), then split the leaf page, Records with keys < middle key go to the left leaf page, Records with keys >= middle key go to the right leaf page, split the Index page, If key < middle key go to left Index (or) if >= middle go to right Index. The middle key goes to next (higher level) index. If next level index page is full, continue splitting the index pages.

B+ tree : Deletion.

① If data page below fill factor (NO), Index page Below fill factor (NO), then delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.

② If data page below fill factor (yes), Index page below fill factor (No), then Combine the leaf page and its sibling. change the Index page to reflect the change.

③ If data page below fill factor (yes), Index page below fill factor (yes), then combine the leaf page and its sibling, adjust the index page to reflect the change, Combine the index page with its sibling, continue combining index pages until we reach a page with the correct fill factor of you reach the root page.

(II) Heap.

A Binary heap is a complete binary tree which satisfies the Heap Ordering Property.

* the min-heap property: The value of each node is greater than (or) equal to the value of its parent, with the minimum-value element at the root.

* the max-heap property: The value of each node is less than (or) equal to the value of its parent, with the maximum value element at the root.

Heap Implementation. (using Array)

Shape is similar to complete Binary tree.

Heap

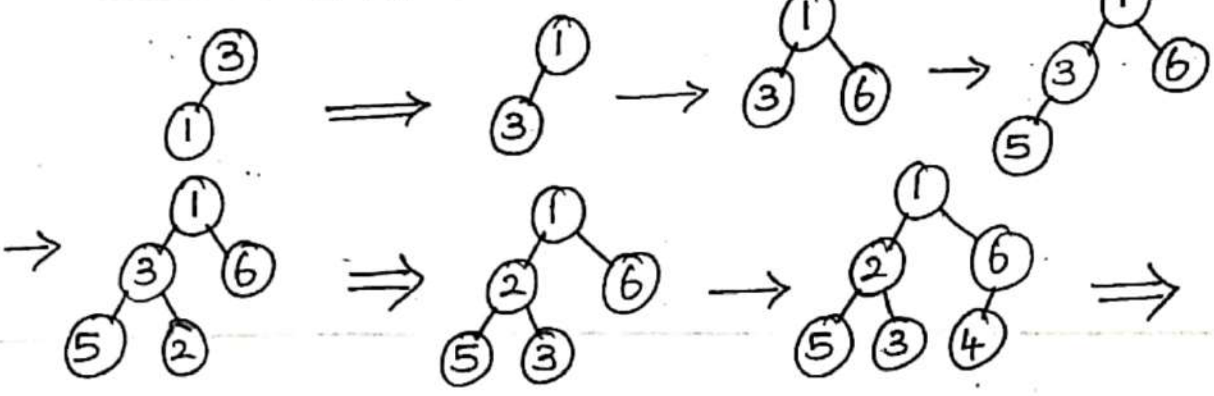
Min Heap (Root is lesser than left and right child)

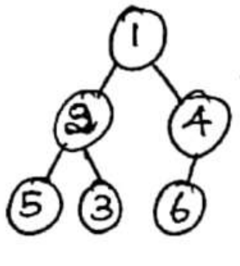
Max Heap (Root is higher than left and right child).

Note :- Always insert from left to right.

Build min Heap.

3	1	6	5	2	4

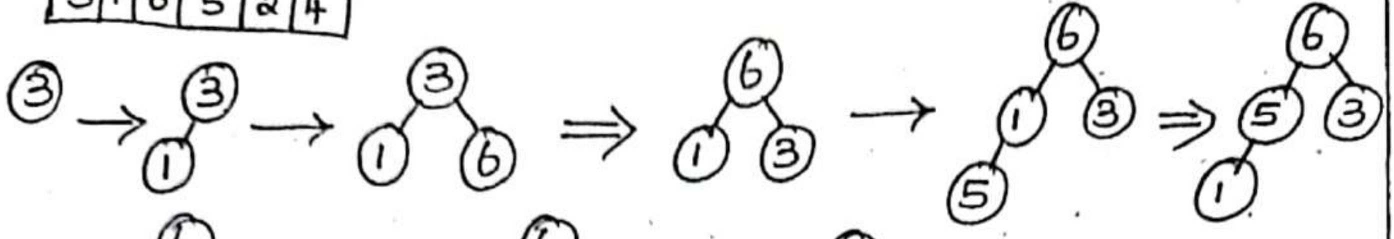


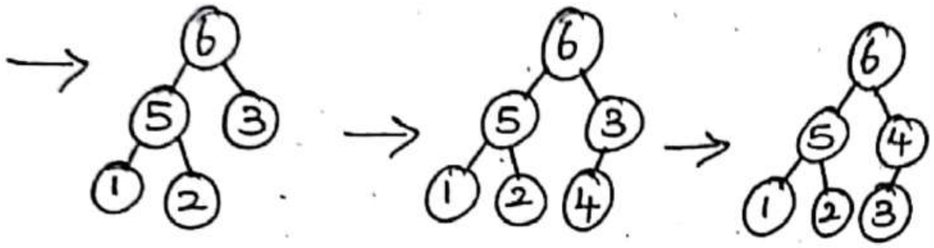


1	2	4	5	3	6

Build Max heap :

3	1	6	5	2	4

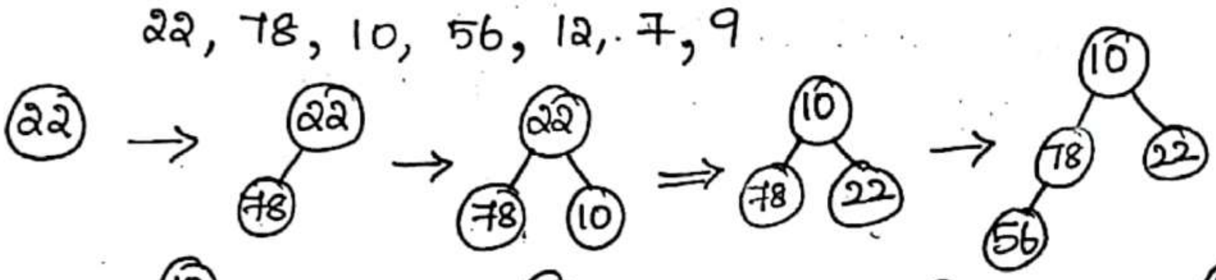


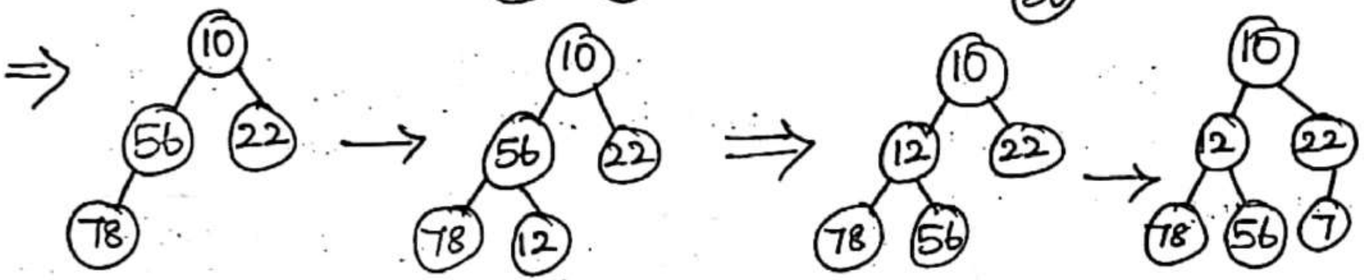


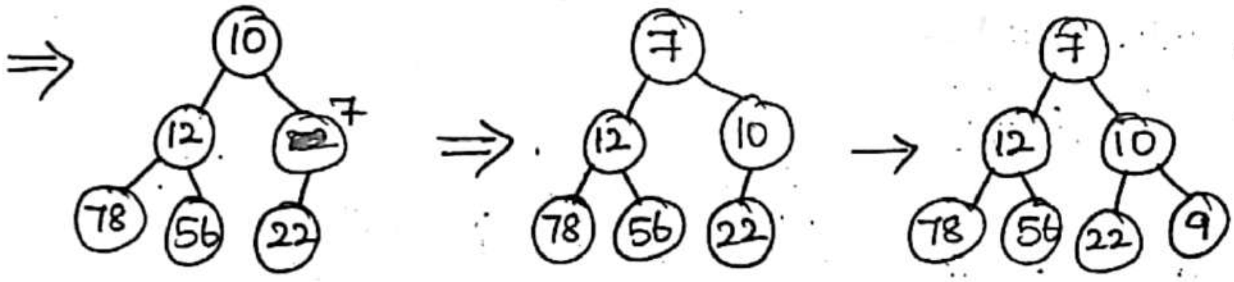
6	5	4	1	2	3

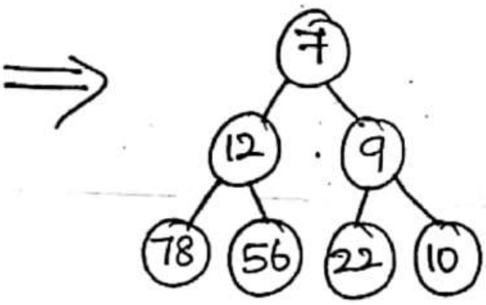
Build min heap :

22, 78, 10, 56, 12, 7, 9







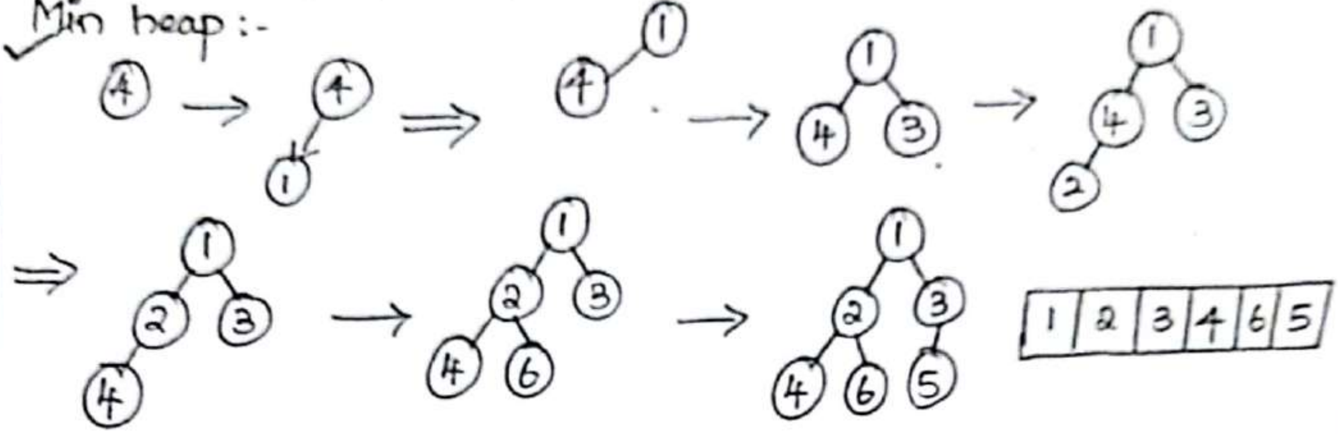


7	12	9	78	56	22	10

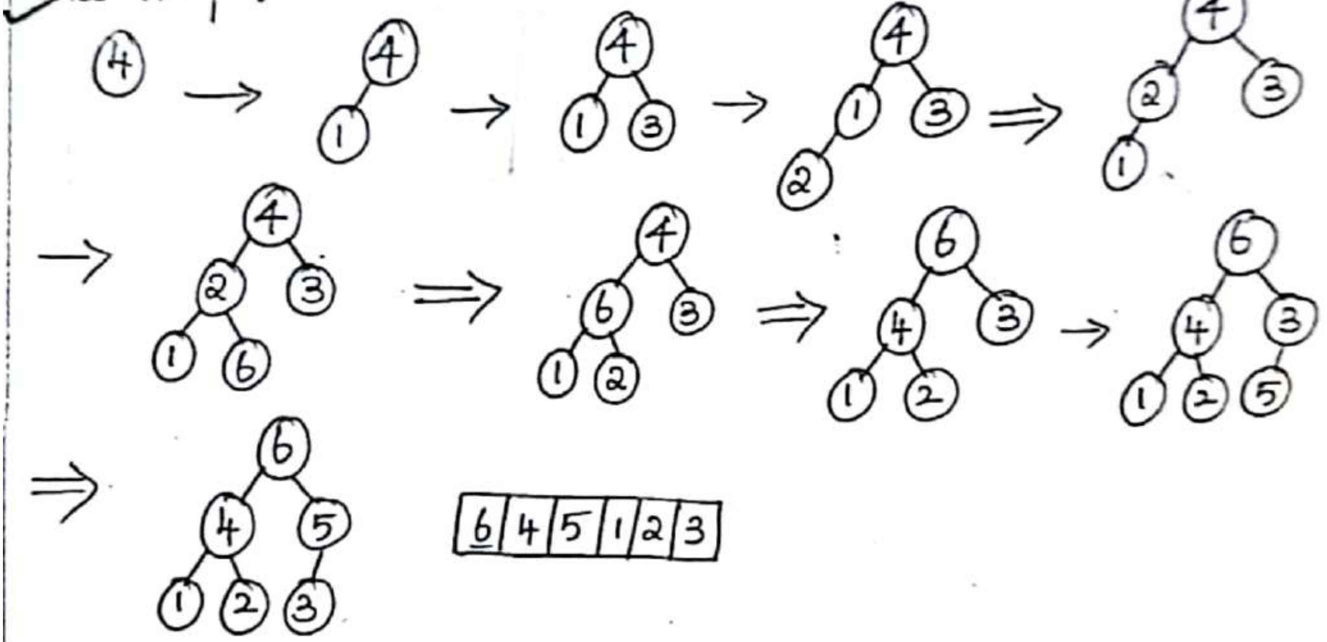
Build Min and Max heap for the following.

4, 1, 3, 2, 6, 5

Min heap :-



Max Heap :-



Parent = (i/2)
Left = 2i
Right = 2i+1

} Array implementation of Heap

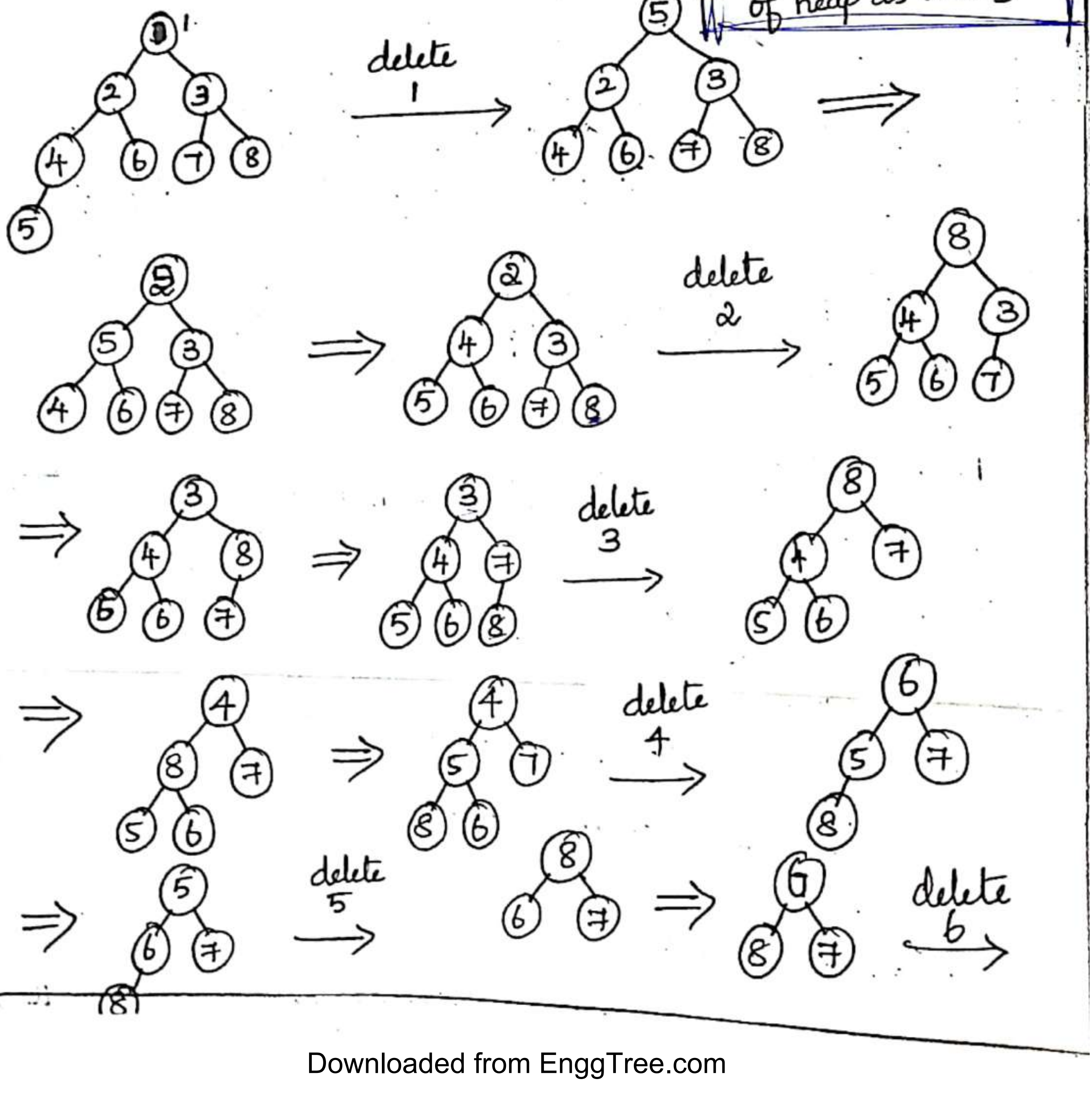
Insertion algorithm :

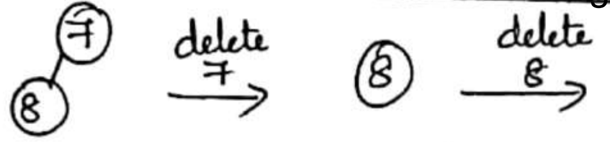
```
Void insert (int a[], int *size, int Key)
{
    if (*size >= max. Element)
  return;
  a[(*size)++] = key;
```

```
shift-up (0, size-1);
return;
void shiftup (int a[], int size).
{
  int parent = i/2 && Parent >0
if (a[Parent] > a[i])
{
  swap (a[parent], a[i]);
 Shiftup (a, Parent);
}
```

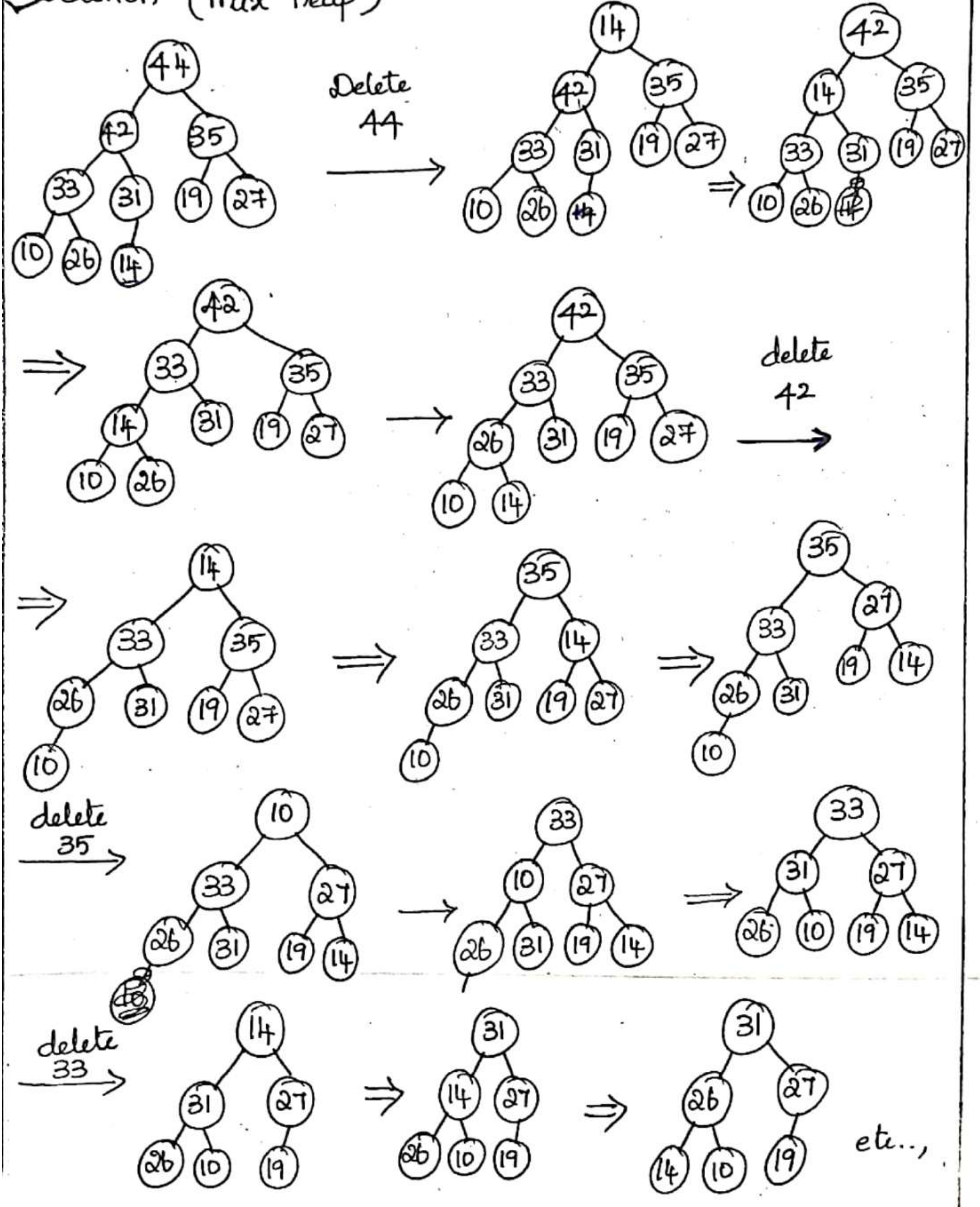
Deletion :- (min Heap)

[Place last element of heap as root]





Deletion (max heap)



ALGORITHM :

Insertion (min & max)

① Create a new node at the end of heap.
② Assign new value to the node.
③ Compare the value of this child node with its parent.
④ If value of parent is less than child, then swap them. → max heap.
⑤ If value of parent is less than child, then do not distrub → min heap.
⑥ Repeat the above steps until heap property holds.

Deletion (min & max)

① Remove the root node.
② Move the last element of last level to root.
③ Compare the value of this child node with its parent.
④ If value of parent is less than child, then swap them → max heap
⑤ If value of parent is less than child, then do not distrub → min heap
⑥ Repeat the above steps until heap Property holds.

(12) Applications of Heap.

* Used in Graph algorithms like

✓ i) Prim's algorithm
✓ ii) Dijkstra's algorithm.

UNIT-IV

NON LINEAR DATA STRUCTURES - GRAPHS

1) Definitions
2) Representations of Graph
3) Types of Graph
4) Breadth first Traversal
5) Depth first Traversal.
6) Topological Sort
7) Bi-connectivity
8) cut vertex
9) Euler circuit
10) Applications of Graphs

1) Definitions :

In this datastructure, data is organized without any sequence. Eg - Tree, Graph etc. All the data elements in non-linear data structure can not be traversed in single run.

Eg ; Trees and Graphs.

Graph is a collection of finite number of vertices and an edges that connect these vertices. Edges represent relationships among vertices that stores data elements.

$$G = (V, E)$$

Linear Data Structure	Non-Linear Data Structure
1) Every item is related to its previous and next item.	1) Every item is attached with many other items.
2) Data is arranged in linear sequence.	2) Data is not arranged in sequence.
3) Data items can be traversed in a single run.	3) Data can not be traversed in a single run.
4) Eg; Array, Stack, Queue, Linked List.	4) Eg; Tree, Graph.
5) Implementation is Easy.	5) Implementation is difficult.

(2) Representations of Graph. (N/D-2018)

Graph data structure is represented using following representations.

i) Adjacency matrix
ii) Incidence matrix
iii) Adjacency List.

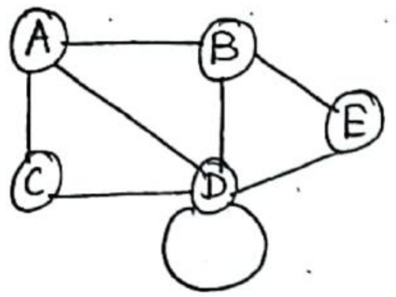
Adjacency matrix :

A Graph with 'n' vertices can be represented using a matrix of nxn class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 (or) 0. Here,

1 → There is an edge from row vertex to column vertex.

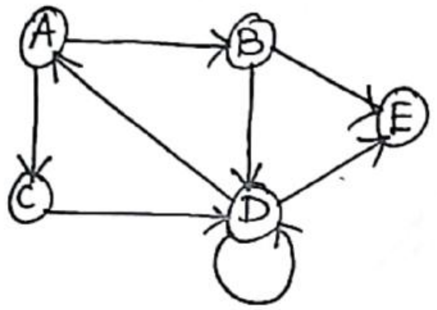
0 → There is no edge from row vertex to column vertex.

Undirected Graph:



$$\Rightarrow \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed Graph:



$$\Rightarrow \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Incidence Matrix: A Graph with n vertices and E edges can be represented using a matrix of n×E class. Row → vertices & column → Edges.

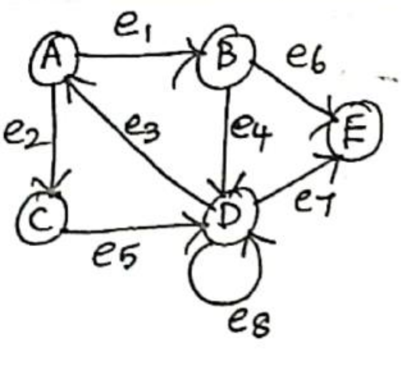
Matrix is filled with eith 0, 1, −1.

0 → row edge is not connected to column Vertex

1 → row edge is connected as outgoing edge to column Vertex.

−1 → row edge is connected as incoming edge to column vertex.

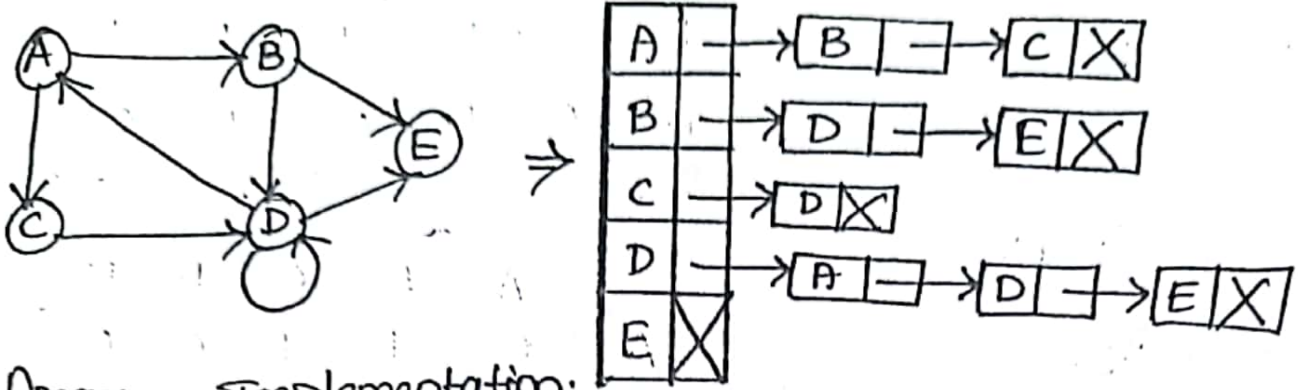
Directed Graph:-



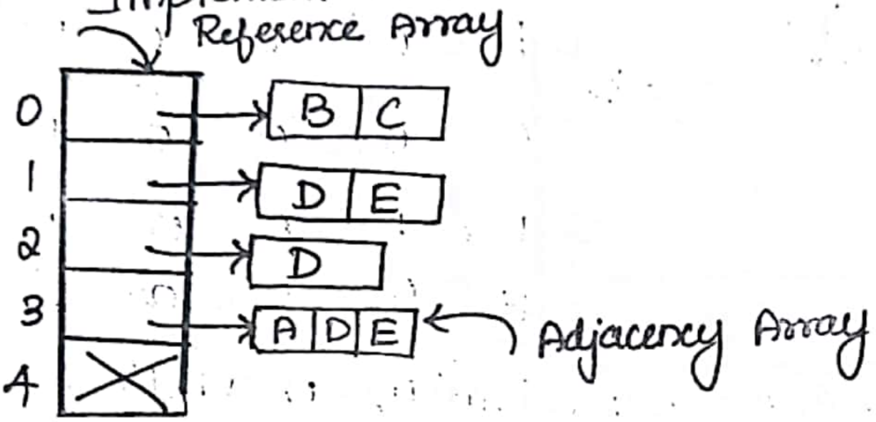
$$\Rightarrow \begin{array}{c|cccccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

Adjacency List : In this representation, every vertex of graph contains list of its adjacent vertices.

Linked list Implementation :-



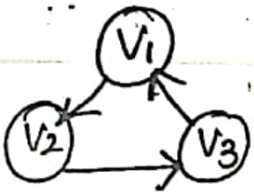
Array Implementation:



(3) Types of Graph.

- * Directed Graph
- * Undirected Graph.

Directed Graph: It is a graph which consists of directed edges. All the edges in E are unidirectional. Sometimes it is also called as digraph.

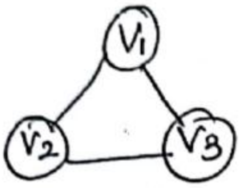


(V_1, V_2), (V_2, V_3) and (V_3, V_1) are directed graph.

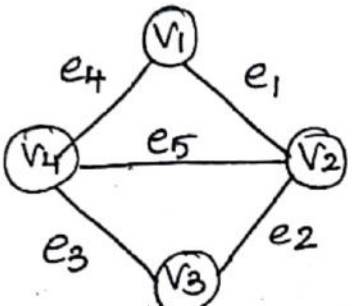
Note : $(V_1, V_2) \neq (V_2, V_1)$

Undirected Graph: It is a graph which consists of undirected edges. In the below fig, (v_1, v_2), (v_2, v_3) and (v_3, v_1) are edges. It is noted that

$(v_1, v_2) = (v_2, v_1)$ and so on.



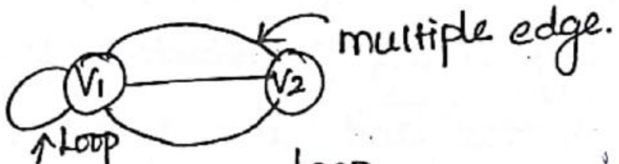
Eg;



Vertices. $V = \{v_1, v_2, v_3, v_4\}$ and

Edges $E = \{(v_1,v_2), (v_2,v_3), (v_3,v_4), (v_4,v_1), (v_4,v_2)\}$

Basic Terminologies :-

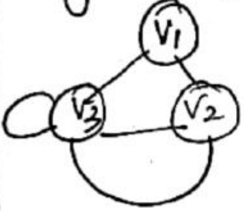


→ Edge from (v_1, v_1) is called as loop.

→ A graph with no edge (ie E is empty) is empty.

→ A Graph with no vertices is called NULL Graph.

Degree of a vertex :- [No. of edges incident with it]



$d(v_1) = 2$

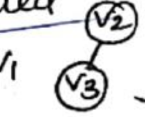
$d(v_2) = 3$

$d(v_3) = 5$

A pendent Vertex is a Vertex whose degree is 1. [

]

whose degree is 0 called Isolated vertex (v1) [

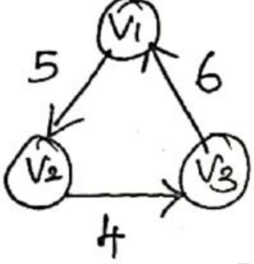


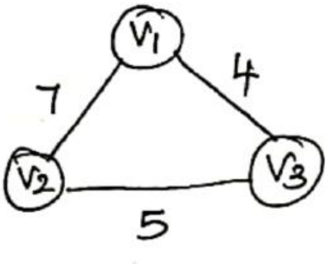
]

Weighted Graph:

A graph G is said to be weighted graph if every edge in a graph is assigned a value.

Eg.,





Graph Traversals.

→ BFS (Breadth first Search)
→ DFS (Depth first Search)

BFS :- BFS Traversal of a graph, Produces a Spanning tree as final result. Spanning tree is a graph without any loops. We use Queue data structure with maximum size of total number of Vertices in the graph to implement BFS traversal of a Graph.

Steps to Implement BFS:

① Define Queue of size total number of Vertices in the graph.

② Select any Vertex as starting point for traversal. Visit that vertex and insert it into the queue.

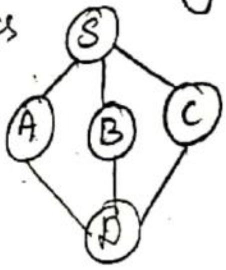
③ Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

④ When there is no new Vertex to be Visit from the vertex at front of the Queue then delete that Vertex from the Queue.

⑤ Repeat step ③ and ④ until Queue becomes Empty.

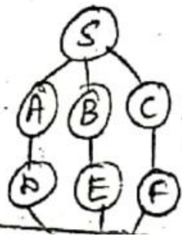
⑥ When Queue becomes empty, then Produce final "spanning tree" by removing unused edges from the graph.

BFS



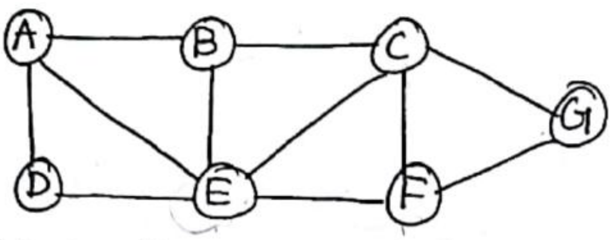
S → A → B → C → D

DFS

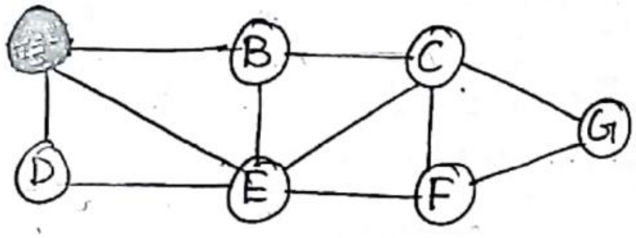


S → A → D → G → E → B → F → C

Eg; consider the following example graph to Perform BFS traversal.



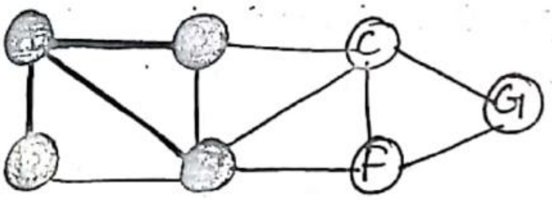
Step 1: Select the vertex A as starting point (visit A) Insert A into the Queue.



Queue

A						

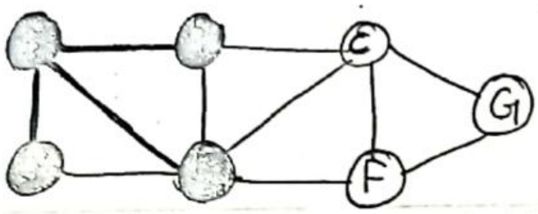
Step 2: Visit all adjacent Vertices of A which are not visited (D, E, B). Insert newly visited vertices into the Queue and delete A from the Queue.



Queue

	D	E	B			

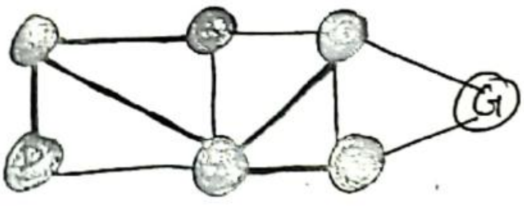
Step 3: Visit all adjacent Vertices of D which are not Visited (there is no vertex). Delete D from the Queue.



Queue

		E	B			

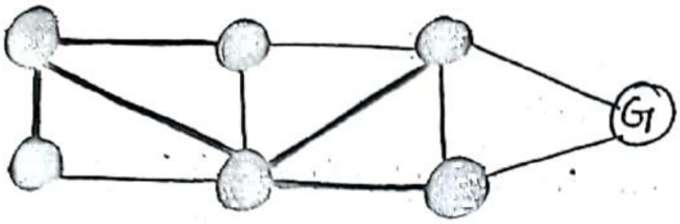
Step 4: Visit all adjacent Vertices of E which are not Visited (C, F). Insert newly visited Vertices into Queue and delete E.



Queue

			B	C	F	

Step 5 : Visit all adjacent vertices of B which are not visited (there is no vertex). Delete B from the queue.

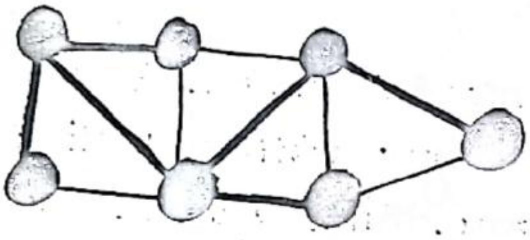


Queue

				C	F	

Step 6 : Visit all adjacent vertices of C which is not visited (G)

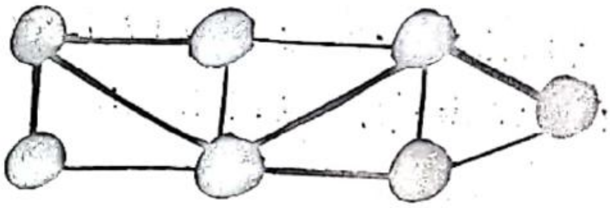
Insert newly visited vertex into the queue and delete C from the queue.



Queue

					F	G

Step 7 : Visit all adjacent vertices of F which are not visited (there is no vertex). Delete F from the queue.



Queue

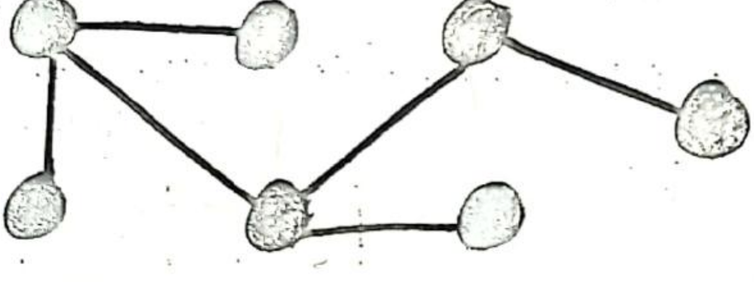
						G

Step 8 : visit all adjacent vertices of G (there is no vertex). Delete G from queue.

Queue

(Queue becomes Empty) so stop BFS process

BFS



A → B → C → D → E → F → G

DFS (Depth First Search)

It produces a spanning tree as final result. Spanning tree is a graph without any loops. We use stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a Graph.

Steps to Implement DFS.

① Define a stack of size total number of vertices in the Graph.

② Select any vertex as starting point for traversal. Visit that vertex and push it on to the stack.

③ Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

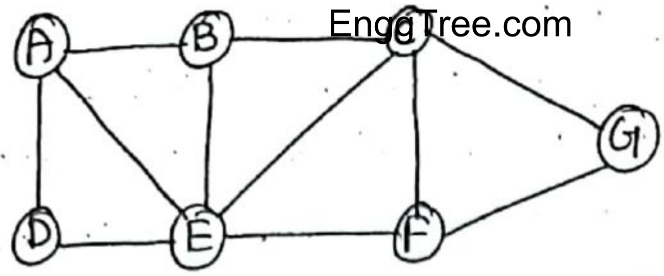
④ Repeat Step ③ until there are no new vertex to be visit from the vertex on top of the stack.

⑤ When there is no new vertex to be visit then use "BACK TRACKING" and pop one vertex from the stack.

⑥ Repeat steps 3, 4, and 5 until stack becomes Empty.

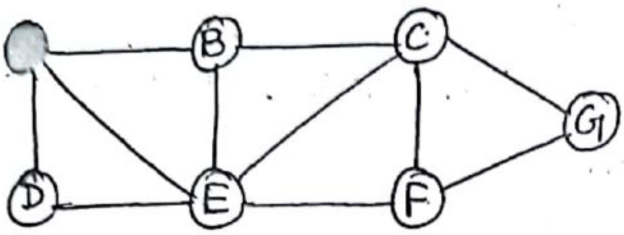
⑦ When stack becomes empty, then Produce final spanning tree by removing unused edges from the graph.

Eg;

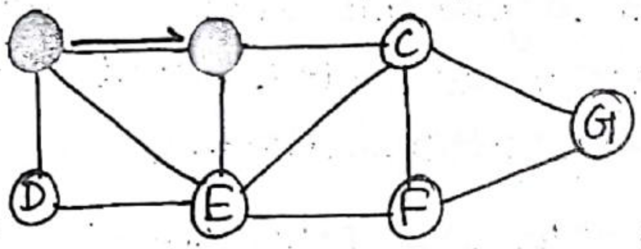


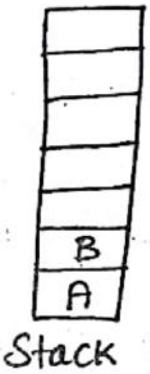
Step 1: Select the vertex A as starting point (visit A)

Push A on to the stack.

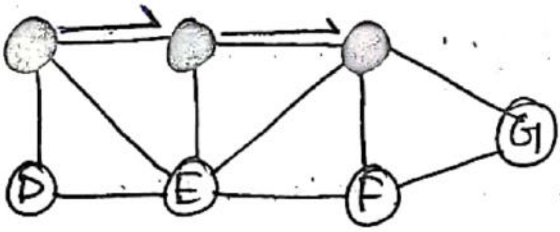


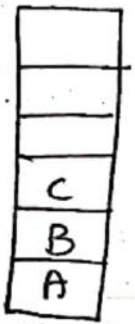
Step 2: Visit any adjacent vertex of A which is not visited (B). Push newly visited vertex B on to the stack.



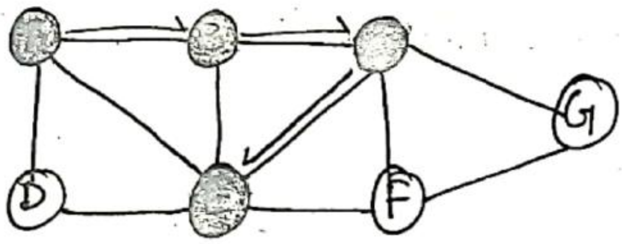


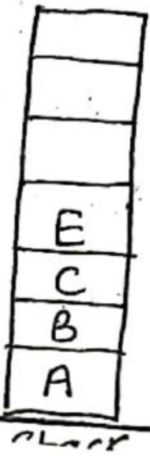
Step 3: Visit any adjacent vertex of B which is not visited (C). Push C onto the stack.



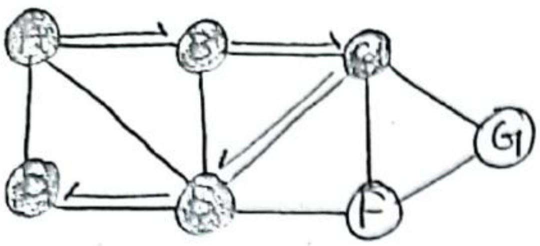


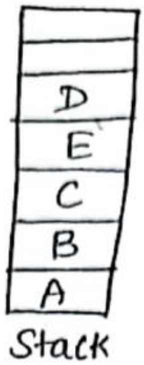
Step 4: Visit any adjacent vertex of C which is not visited (E). Push E on to the stack.



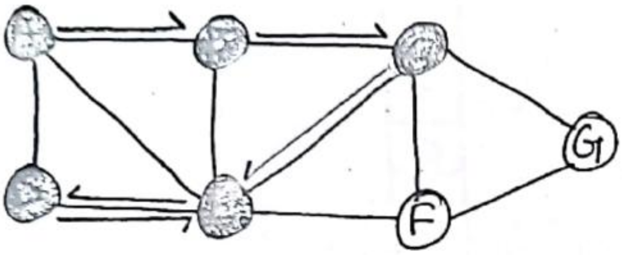


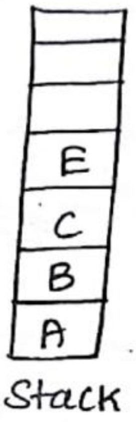
Step 5 : Visit any adjacent vertex of E, which is not visited (D). Push D on to the stack.





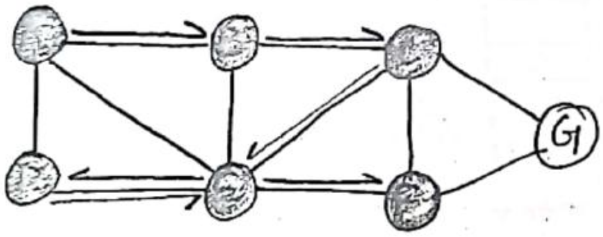
Step 6 : There is no new vertex to be visited from D. So use back track. pop D from the stack.

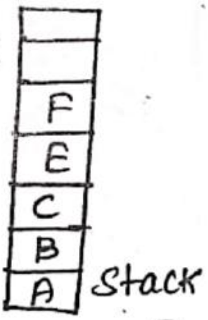




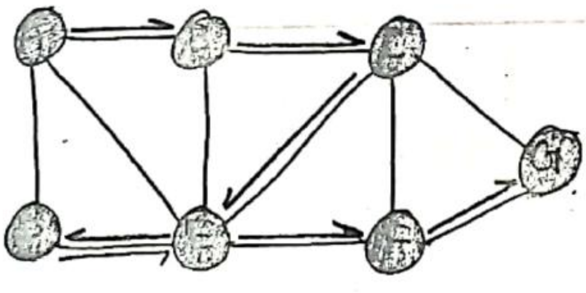
Step 7 :

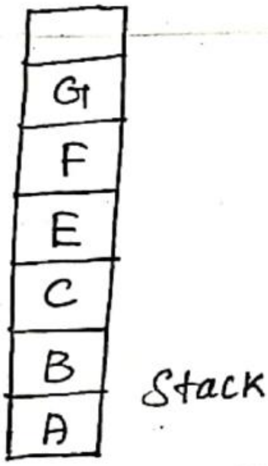
visit any adjacent vertex of E which is not visited (F). push F on to the stack.



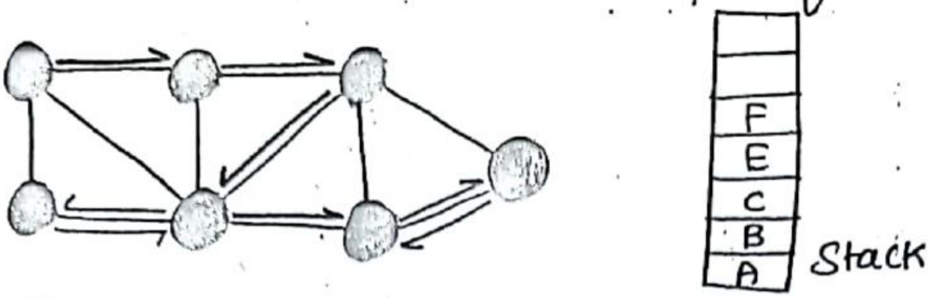


Step 8: visit any adjacent vertex of F which is not visited (G). push G on to the stack.

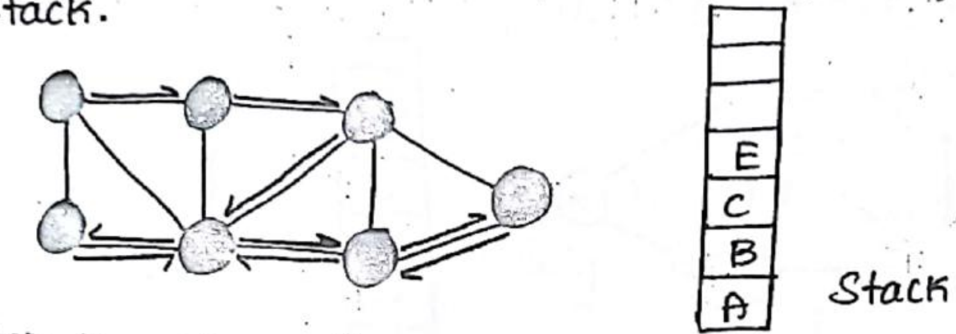




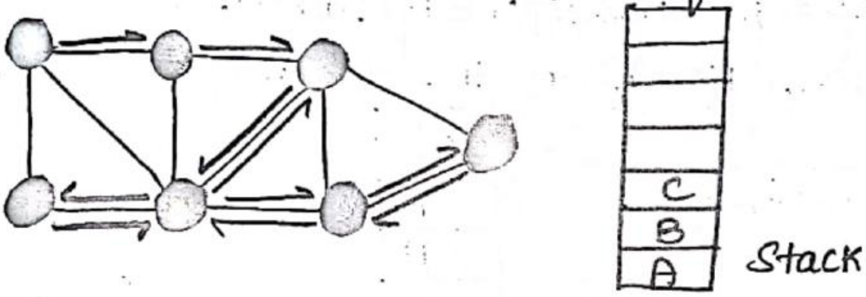
Step 9 : There is no new vertex to be visited from G. So use back track. Pop G from the stack.



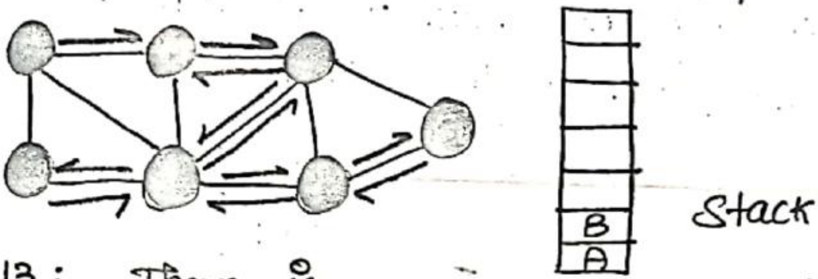
Step 10 : There is no new vertex to be visited from F. So use back track. pop F from the Stack.



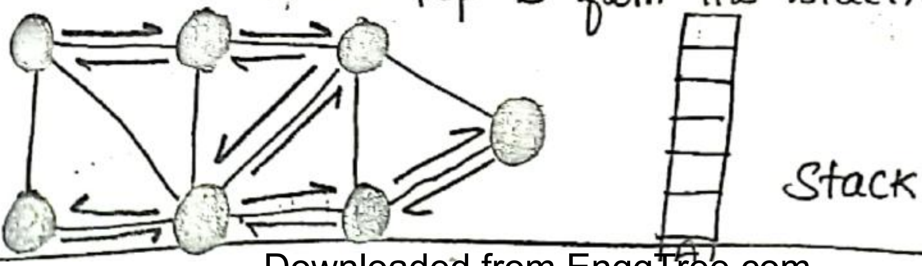
Step 11 : There is no new vertex to be visited from E. So use back track. Pop E from the stack.



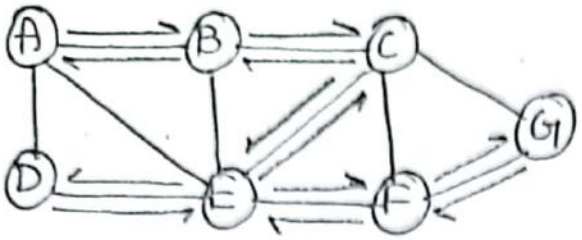
Step 12 : There is no new vertex to be visited from C. So use back track. Pop C from the stack



Step 13 : There is no new vertex to be visited from B. So use back track. Pop B from the stack.

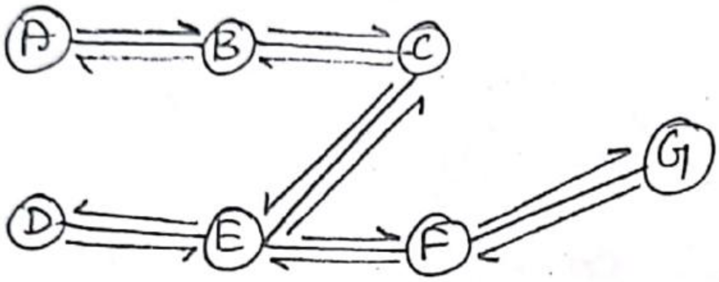


Step 14 : There is no new vertex to be visited from A. So use back track. Pop A from the stack.



Stack became Empty. So stop DFS Traversal. final result of DFS traversal is following Spanning tree.

A → B → C → E → D → F → G



Eg Problems :- BFs and DFS.

Algorithm for BFs.

```
Visited [v] = True
Q. enqueue(v)
while not Q. isEmpty() do
  v ← Q. dequeue()
  for all w adjacent to v do
    if visited [w] = false then
      Visited [w] = True
      Q. enqueue (w)
```

Algorithm for DFS.

```
Algorithm dfs from Vertex (G, v)
  S. push (v)
  while not S. isEmpty () do
    v ← S. pop( )
    if visited [v] = false then
      Visited [v] = True
      for all w adjacent to v do
        S. push (w).
```

(6) Topological Sort.

A Topological sort is an ordering of the Vertices in a directed graph with the Property that if there is a path from V_i to V_j, then V_i must come before V_j in the Ordering.

⊗ If there are cycles in the graph, then topological ordering is impossible.

Algorithm:

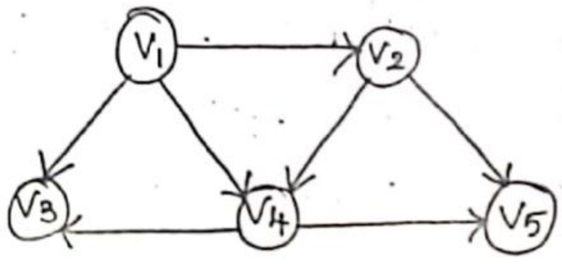
(1) Scan the adjacency lists to find a Vertex V with 0 in degree.

(2) If there is no such Vertex stop. otherwise

(3) Assign a topological number to the Vertex V, and remove all edges (u, v).

(4) Goto step (1)

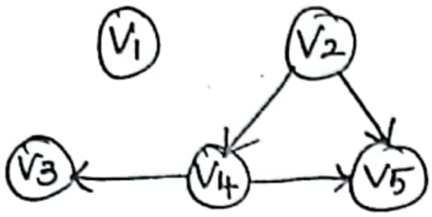
Eg.,



Solution:

Vertex	Indegree before dequeue 1	2	3	4	5
V_1	0	1	2	2	2
V_2	—	0	1	1	2
V_3	—	—	1	0	1
V_4	—	—	0	—	0
V_5	—	—	—	—	0

Step 1 :



V_1 , V_2

Indegree. (1st time)

$V_1 \rightarrow 0$

$V_2 \rightarrow 1$

$V_3 \rightarrow 2$

$V_4 \rightarrow 2$

$V_5 \rightarrow 2$

Step 1

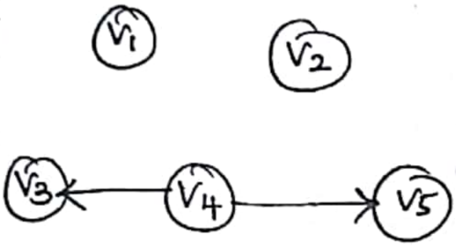
$V_2 \rightarrow 0$

$V_3 \rightarrow 1$

$V_4 \rightarrow 1$

$V_5 \rightarrow 2$

Step 2 :



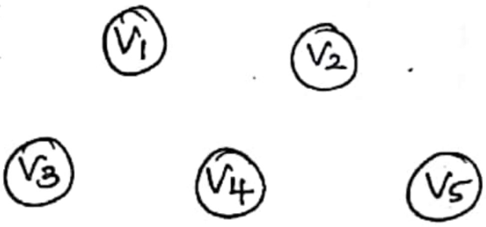
V_1, V_2, V_4

$V_3 \rightarrow 1$

$V_4 \rightarrow 0$

$V_5 \rightarrow 1$

Step 3 :



V_1, V_2, V_4, V_3, V_5

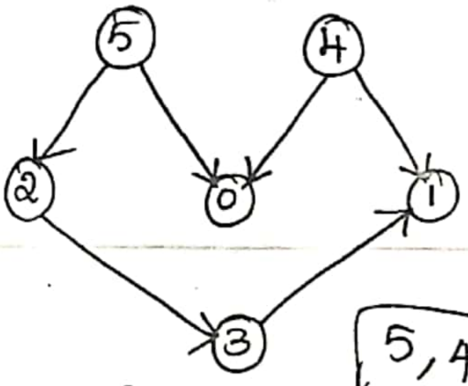
V_1, V_2, V_4, V_5, V_3.

$V_3 \rightarrow 0$

$V_5 \rightarrow 0$

The topotogical ordering of above graph is V_1, V_2, V_4, V_3 and V_5 (or) V_1, V_2, V_4, V_5 and V_3.

Eg (2)



$\boxed{5, 4}$

④

5 , 2, ~~4~~ 0 . 1

Indegree.

$5 \rightarrow 0$

$4 \rightarrow 0$

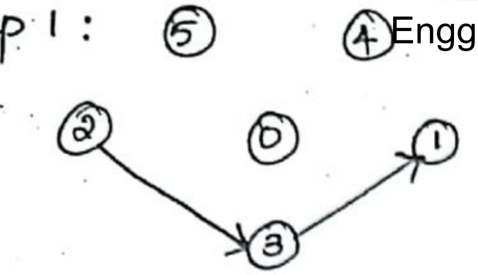
$0 \rightarrow 2$

$2 \rightarrow 1$

$1 \rightarrow 2$

$3 \rightarrow 1$

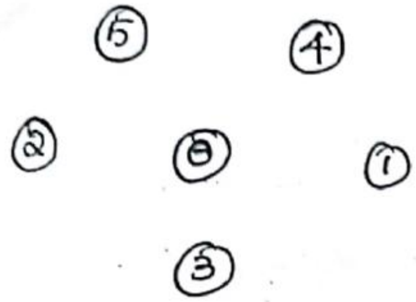
Step 1 :



0 → 0
2 → 0
1 → 0
3 → 1

5, 4, 0, 2, 1.

Step 2 :



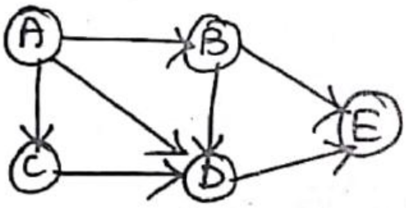
3 → 0

5, 4, 0, 2, 1, 3.

5, 4, 0, 2, 1, 3
5, 4, 1, 2, 0, 3
5, 4, 2, 1, 0, 3

Solution :- 5, 4, 0, 2, 1, 3 (or) 5, 4, 2, 0, 1, 3.

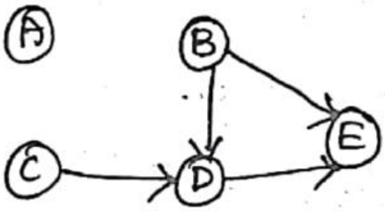
Eg (3)



Indegree.

A → 0
B → 1
C → 1
D → 3
E → 2.

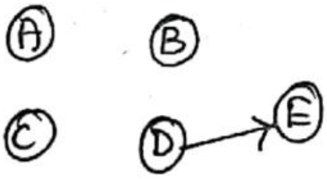
Step 1 :-



B → 0
C → 0
D → 2
E → 2

A, B, C (or)
A, C, B

Step 2 :-



D → 0
E → 1

A, C, B, D

Step 3 :

E → 0

A, C, B, D and E.

Solution :- A, C, B, D and E.

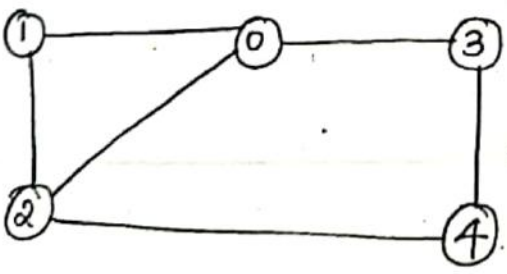
Algorithm :

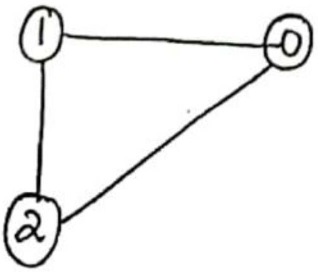
```
    Void topsort (Graph G)
  {
     int counter;
     Vertex v, w ;
  for (counter = 0; counter <numvertex ; counter ++)
 {
    V= find _ NewVertex of Degreezero ();
    if ( V = NotAVertex)
  {
    Error ("Graph has a cycle");
  }  break ;
     TopNum [V] = Counter ;
  for each w adjacent to V
        Indegree [w]- - ;
  } }
```

(7) Bi-connectivity

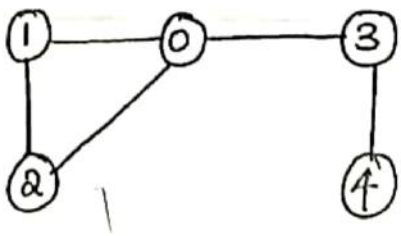
An Undirected Graph is called Biconnected if there are two vertex disjoint paths between any two vertices.

Eg.,





Not - Biconnected.

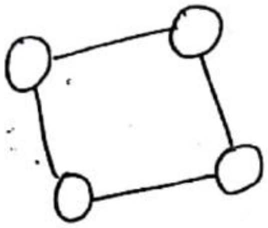


A Connected graph is Biconnected if it is Connected and doesn't have any Articulation point. We mainly need to check two things in a graph.

1) The graph is Connected.
2) There is no articulation point in Graph.

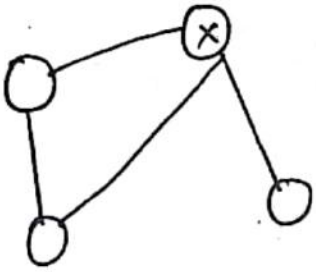
Articulation point → Also called cut vertex, iff removing it (and edges through it) disconnects the graph.

(Eg)



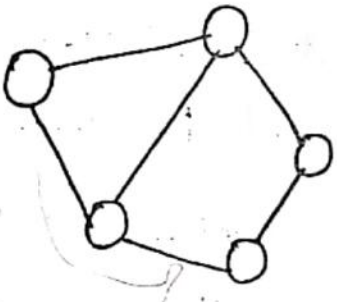
A biconnected graph on 4 Vertices and 4 edges.

(Eg)



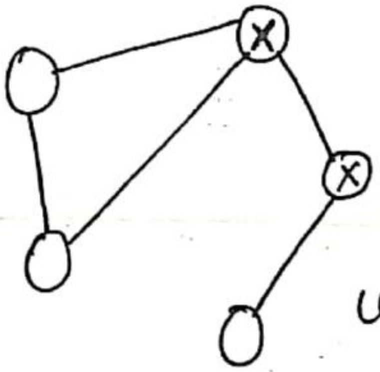
A Graph that is not biconnected. The removal of vertex X would disconnect the graph.

(Eg)



A biconnected graph on five vertices and six edges.

(Eg)



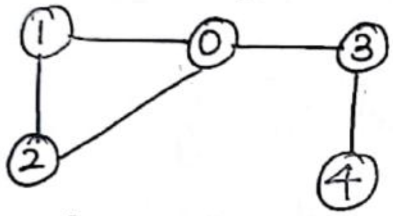
A graph that is not biconnected. The removal of vertex X would disconnect the graph.

(8) Cut-Vertex

Cut-vertex (Articulation point) :-

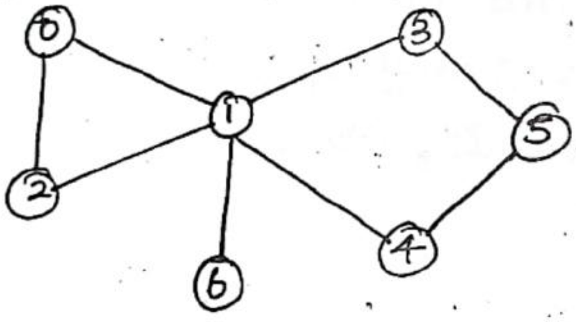
A vertex is an undirected connected graph is an articulation point, iff removing it (and edges through it) disconnects the graph.

Articulation points represent Vulnerabilities in a connected network single points whose failure would split the network into 2 (or) more disconnected components. They are useful for designing reliable networks.



Articulation points are
0 and 3.

Articulation points
are 1 and 2.



Articulation point is 1.

How to find all articulation points in a given graph:-

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph.

for a Vertex V, do following.

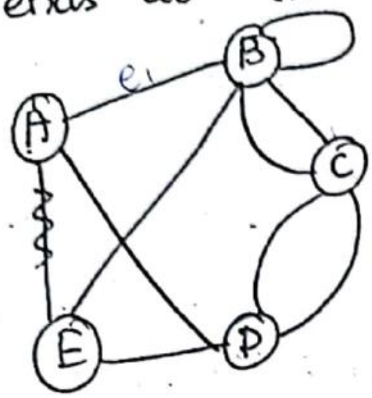
a) Remove V from graph.

b) See if the graph remains connected (BFS or DFS)

c) Add V back to the graph.

(9) Euler circuit. (N/D-2018)

An Euler circuit is a circuit that uses every edge of a graph exactly once. An euler path circuit starts and ends at the same vertex.



An Euler circuit :

CDCBBADEBC

Under what condition(s) is a graph guaranteed to have an Euler circuit?

iff every vertex has an even number of edges.

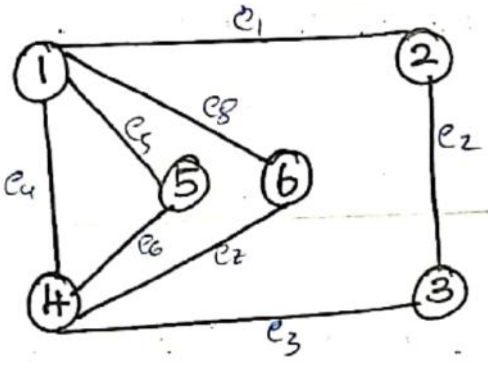
To compute an Euler circuit, (or) show there is not one:

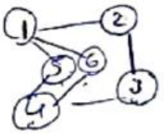
① pick a vertex and perform a depth-first traversal, marking edges that are traversed, and marking any edges. vertex that is reached and has no remaining untraversed

② If this does not lead to a circuit, there is no Euler circuit in the graph.

③ pick a vertex with an untraversed edge, and repeat the first step.

Eg ;





1-5-4-6-1-2-3-4-1.

Pick vertex 1:

V	Edges			
1	2	4	5	6
2	1	3		
3	2	4		
4	1	3	5	6
5	1	4		
6	1	4		

Path 1 : 1-2

V	Edges			
1	~~2~~	4	5	6
2	~~1~~	3		
3	2	4		
4	1	3	5	6
5	1	4		
6	1	4		

Path 1 : 1-2-3

V	Edges			
1	~~2~~	4	5	6
~~2~~	~~1~~	~~3~~		
3	~~2~~	4		
4	1	3	5	6
5	1	4		
6	1	4		

V	Edges			
1	~~2~~	4	5	6
~~2~~	~~1~~	~~3~~		
~~3~~	~~3~~	~~4~~		
4	1	~~3~~	5	6
5	1	4		
6	1	4		

Path 1 : 1-2-3-4

V	Edges			
1	~~2~~	~~4~~	5	6
~~2~~	~~1~~	~~3~~		
~~3~~	~~3~~	~~4~~		
4	~~1~~	~~3~~	5	6
5	1	4		
6	1	4		

Path 1 : 1-2-3-4-1

Pick vertex 1 again

V	Edges			
1	~~2~~	~~4~~	~~5~~	6
~~2~~	~~1~~	~~3~~		
~~3~~	~~3~~	~~4~~		
4	~~1~~	~~3~~	5	6
5	~~1~~	4		
6	1	4		

Path 2 : 1-5

V	Edges			
1	~~2~~	~~4~~	~~5~~	6
~~2~~	~~1~~	~~3~~		
~~3~~	~~3~~	~~4~~		
4	~~1~~	~~3~~	~~5~~	6
~~5~~	~~1~~	~~4~~		
6	1	4		

Path 2 : 1-5-4

V	Edges			
1	~~2~~	~~4~~	~~5~~	6
~~2~~	~~1~~	~~3~~		
~~3~~	~~3~~	~~4~~		
4	~~1~~	~~3~~	~~5~~	~~6~~
~~5~~	~~1~~	~~4~~		
6	1	~~4~~		

Path 2 : 1-5-4-6

V	Edges
~~1~~	~~2~~ ~~4~~ ~~5~~ ~~6~~
~~2~~	~~1~~ ~~3~~
~~3~~	~~3~~ ~~4~~
4	~~1~~ ~~3~~ ~~5~~ ~~6~~.
~~5~~	~~1~~ ~~4~~
~~6~~	~~1~~ ~~4~~

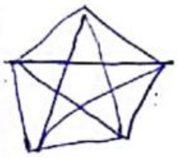
Path 1 : 1 – 2 – 3 – 4 – 1

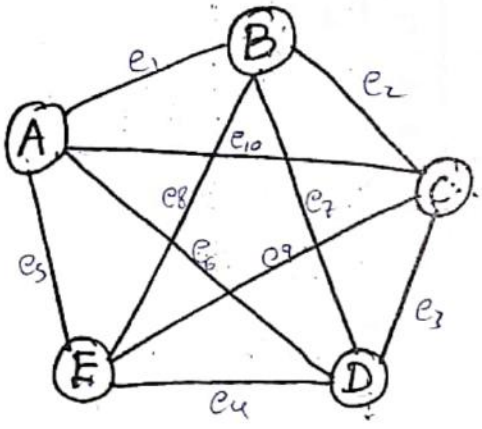
Path 2 : 1 – 5 – 4 – 6 – 1

Path : 1 – 5 – 4 – 6 – 1 – 2 – 3 – 4 – 1

Path 2 : 1 – 5 – 4 – 6 – 1

Eg (2)





Path 1 : A → B → C → D → E → A

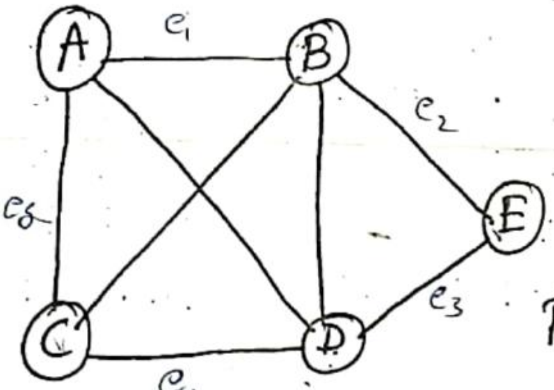
Path 2 : A → C → E → B → D → A

e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8
A – B – C – D – E – A – D – B – E
e_9 e_{10}
– C – A

10 Edges exactly once,

Path : A → B → C → D → E → A → C → E → B → D → A.

Eg (3)



Path 1 : A – B – E – D – C – A

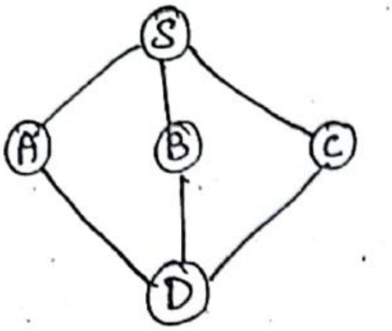
Path 2 : A – C – D – E – B – A

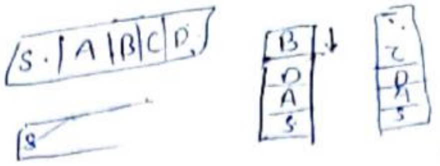
Path : $A \xrightarrow{e_1} B \xrightarrow{e_2} E \xrightarrow{e_3} D \xrightarrow{e_4} C \xrightarrow{e_5} A \to$ e_5
C → D → E → B → A.

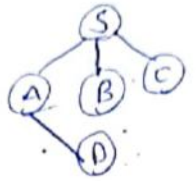
Graph Traversals.

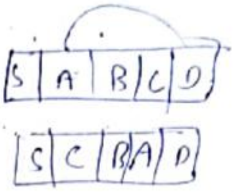
Anna univ. Questions.

BFS :



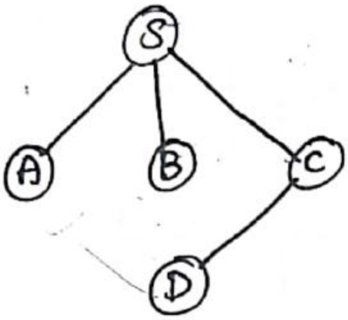


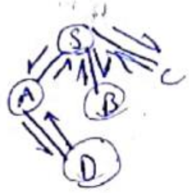




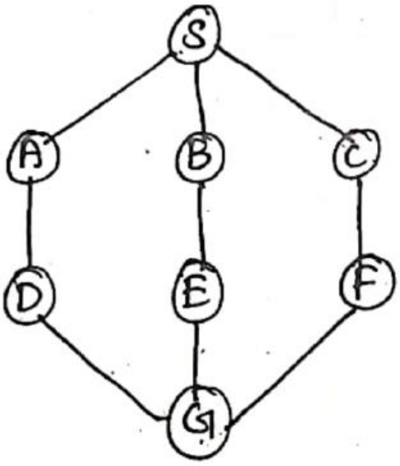
Solution :

S → A → B → C → D



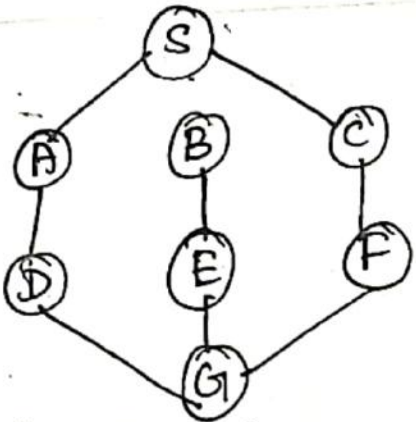


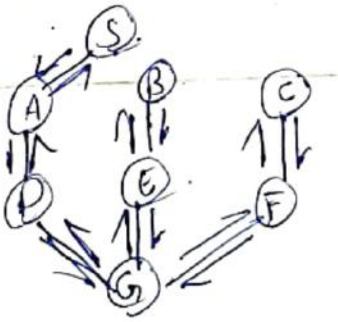
DFS :



Solution:-

S → A → D → G → E → B → F → C
→ S





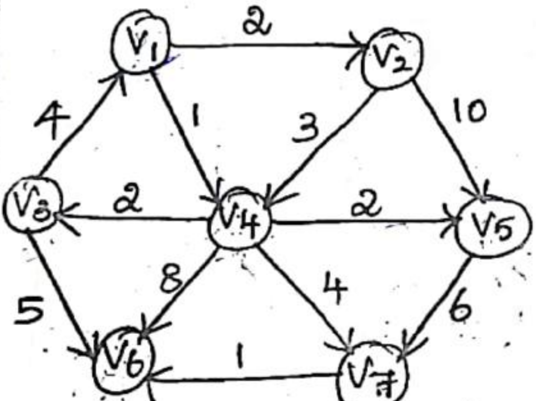
(10) Applications of Graph

* Dijkstra's Algorithm. (find shortest path from source vertex to all other vertices)

It is used to find the single source shortest path and it proceeds in stages. At each stage dijkstra's algorithm selects a vertex V, which has the smallest dv among all the unknown vertices, and declares that the shortest path from S to V is known.

* It could be used for both directed and undirected graphs.
* All edges must have non-negative weights.
* Graph must be connected.

Eg (1)



Initial Configuration

V	Known	dv	Pv
V1	0	0	0
V2	0	∞	0
V3	0	∞	0
V4	0	∞	0
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

After V1 is declared known

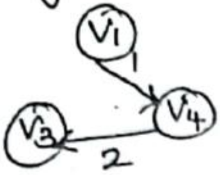
V	Known	dv	Pv
V1	1	0	0
V2	0	2	V1
V3	0	∞	0
V4	0	1	V1
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

After V4 declared known

V	Known	dv	Pv
V1	1	0	0
V2	0	2	V1
V3	0	3	V4
V4	1	1	V1
V5	0	3	V4
V6	0	9	V4
V7	0	5	V4

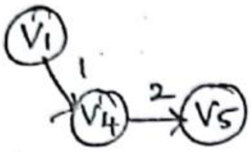
When updating the cost of V_3, add the cost with already selected vertex.

ie.,

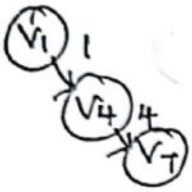


$V_3 = 1+2 = 3.$

III^{rly}



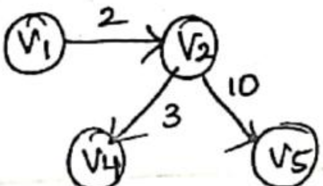
$V_5 = 1+2 = 3.$



$V_7 = 1+4 = 5.$

V_2 declared as known.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

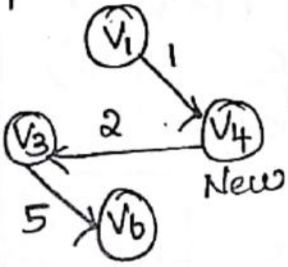


Adjacent to V_2 is V_4 and V_5.

V_4 already visited and $V_5 = 2+10$ which is >3, the old value. so no change in dv. The next minimum unvisited vertex is V_3, declare it as known.

V_3 declared as known.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4



Adjacent of V_3 is V_6.

Old value V_6 : 9

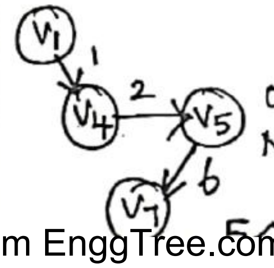
New value V_6 : 8 (1+2+5)

$8 < 9.$

So $dv = 8$. ∴ Minimum unvisited vertex is V_5.

V_5 declared as known.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	8.	V_3
V_7	0	5	V_4



Adjacent of V_5 is V_7.

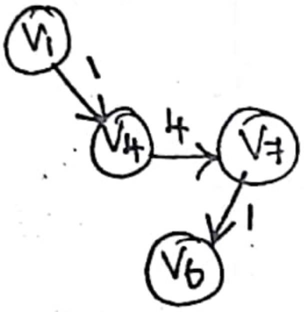
old value of $V_7 = 5$.

New value of $V_7 = 9$ (1+2+6)

$5 < 9$ so dv ⇒ No change

V7 declared as known

V	Known	dv	Pv
V1	1	0	0
V2	1	2	V1
V3	1	3	V4
V4	1	1	V1
V5	1	3	V4
V6	0	6	V7
V7	1	5	V4



Old value :

V6 = 8

New value :

V6 = 6 (1+4+1)

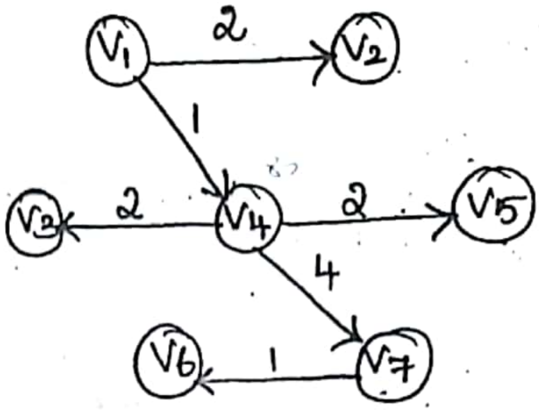
dv = 6. (8<6)

V6 declared as known

V	Known	dv	Pv
V1	1	0	0
V2	1	2	V1
V3	1	3	V4
V4	1	1	V1
V5	1	3	V4
V6	1	6	V7
V7	1	5	V4

Algorithm terminated.

Solution :



Algorithm :-

```
void dijkstra (table T)
{ Vertex V, w;
  for( ; ;)
  {
    V= Smallest Unknown distance Vertex;
    if (v = = NotAVertex)
      break;
    T[V]. Known = True;
    for each W adjacent to V
      if [!T[w]. Known]
        if (T[V]. Dist + Cvw < T[w]. Dist)
        {
          Decrease (T[w]. Dist to T[V]. Dist + Cvw)
          T[w]. path = V;
        }}
}
```

UNIT-V

SEARCHING, SORTING AND HASHING TECHNIQUES.

1) Searching
2) Linear Search
3) Binary Search
4) Sorting
5) Bubble Sort
6) Selection Sort
7) Insertion Sort
8) Shell Sort
9) Radix Sort
10) Hashing
11) Hash functions.
12) Separate chaining.
13) open Addressing.
14) Rehashing.
15) Extendible Hashing.

Searching: It is an operation (or) a technique that helps finds the place of a given element (or) Value in the list. Any search is said to be successful (or) Unsuccessful depending upon whether the element that is being searched is found (or) not. Some of the standard searching technique that is being followed in the data structure is listed below.

- Linear Search (or) Sequential search.
- Binary search.

} N/D - 2018

Linear Search (or) Sequential Search:

It is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, Otherwise the search continues till the end of the data Collection.

Steps :

1) Start from the leftmost element of arr[] and one by one compare x with each element of arr[].

2) If x matches with an element, return the index.

3) If x doesn't match with any of elements, return -1.

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99

Search (12)

(12) ⇒ Both not match. Move to next element.

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99
		(12)						

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99
			(12)					

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99
				(12)				

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99
					(12)			

	0	1	2	3	4	5	6	7
List	65	20	10	55	32	12	50	99
						(12)		

Both are matching. So we stop comparing and display element found at Index 5.

arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

X = 175.

Output :- −1.

Element x is not present in arr[], so it returns −1.

Assume that K is an array of 'n' keys, K(0) through K(n−1), and r an array of records, r(0) through r(n−1), such that K(i) is the key of r(i).

```
for (i=0 ; i<n ; i++)
  if (key = = K(i))
     return (i);
else
     return (-1);
```

If match found then index value is returned.
If no match found then, −1 is returned.

Advantages:

1) The Linear search is simple. It is very easy to understand and implement.

2) It does not require the data in the array to be stored in any particular order.

DISADVANTAGES :

1) This method is insufficient, when large number of elements is present in list.

2) It consumes more time and reduces the retrieval rate of the system.

(Eg) The Linear search is inefficient - If array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order, to find a value in the last element.

(3) BINARY SEARCH

Binary Search, is also known as half-interval search, Logarithmic search, (or) binary chop, is a search algorithm that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array. I/P : 65, 20, 10, 55, 32, 12, 50, 99

	0	1	2	3	4	5	6	7	8
List	10	12	20	32	50	55	65	80	99

search(12)

(Arranged in ascending order)

	0	1	2	3	4	5	6	7	8
List	10	12	20	32	50	55	65	80	99

(12)

8/2 = 4

7/2 = 3

Search element (12), compared with middle element (50).

12 < 50. So search only in the left sublist (10, 12, 20, 32)

	0	1	2	3
List	10	12	20	32

5/2 = 2

	0	1	2	3
list	10	12	20	32

(12) 3/2 = 1

Search element (12), compared with middle element (12). Both are matching. So result is " Element found at Index. 1".

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Search (80)

80 >50

5	6	7	8
55	65	80	99

5	6	7	8
55	65	80	9.9

↑ 80 4/2 = 2

7	8
80	99

7	8
80	99

↑ 80

Both are matching. So Element 80 is found at Index 7.

Algorithm:

```
   low = 0;
   hi = n-1;
 while (low <= hi)
{
 => mid = (low + hi)/2;
 if (key = = k(mid))
     return (mid);
 if (key < k(mid))
     hi = mid-1;
 else   low = mid+1;
 } /* end while */
 return (-1);
```

Advantages :

(1) faster, because does not have to look at every element.

(2) Disadvantages :

(1) This algorithm requires the list to be sorted. Then only the method is applicable.

(4) SORTING

Sorting refers to arranging data in a particular format. It specifies the way to arange data in particular order.

Eg; Telephone directory - The telephone directory stores the telephone numbers of people sorted by their name, so it is easily searched by their names.

Some of the Sorting techniques are.,

1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Shell Sort
5) Radix Sort.

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison - based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

ALGORITHM :

We assume list is an array of 'n' elements. We further assume that swap function swaps the values of the given array elements.

```
begin Bubble Sort (list)
for all elements of list
if list [i] > list [i+1]
swap ( list [i], list [i+1])
   end if
   end for
return list
end bubble - Sort.
```

(Eg) 14, 33, 27, 35, 10.

Bubble sort starts with very first two elements, Comparing them to check which one is greater.

14, 33, 27, 35, 10. [14 < 33 Already Sorted no swap]
14, 33, 27, 35, 10. [33 > 27. So swap it]
14, 33, 27, 35, 10. [33 < 35. Already Sorted]
14, ~~33~~. 27, 33, 35, 10 [35 > 10. so swap it]
14, 27, 33, 35, 10 [33 > 10. so swap it]
14, 27, 33, 10, 35 [27 > 10. so swap it]
14, 27, 10, 33, 35 [14 > 10. so swap it]
14, 10, 27, 33, 35 [Last Iteration]
10, 14, 27, 33, 35

And when there is no swap required, bubble sort learns that an array is completly sorted.

SELECTION SORT

Selection sort is a simple algorithm. This sorting algorithm in-place comparison based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted Path at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

14	33	27	10	35	19	42	44

The whole list is scanned sequentially.

✓ 14 33 27 ✓ 10 35 19 42 44 (swap)
↑min

So, we replace 14 with 10. 10 is the minimum value in the list, appears in the 1st position of the sorted list.

10, (Sorted) | ✓ 33 27 ✓ 14 ← MIN 35 19 42 44
(unsorted)

We find that 14 is the second lowest value in the list and it appear at the 2nd place. Swap the values.

10, 33, 27, 14, 35, 19, 42, 44
(After swapping two least values are positioned)

10, 14, | ✓ 27, 33, 35, ✓ 19, ← MIN 42, 44 (swap it)
SORTED | UNSORTED

10, 14, 19, | 33, 35, 27, 42, 44 (swap it)
SORTED | UNSORTED | MIN

10, 14, 19, 27, | 35, 33, 42, 44 (swap it)
SORTED | UNSORTED | MIN

10, 14, 19, 27, 33, | 35, 42, 44 (No Need swapping)
SORTED | UNSORTED | MIN

10, 14, 19, 27, 33, 35, | 42, 44 (No Need swapping)
SORTED | UNSORTED
↑ MIN

So the sorted Order is

10, 14, 19, 27, 33, 35, 42, | 44.
SORTED | ↑ MIN (UNSORTED)

Solution: 10, 14, 19, 27, 33, 35, 42, 44 [Sorted array]

Eg; (2)

7 5 4 2 ⇒ 2 | 5 4 7
↑ min element (2) | Sorted array | Unsorted Array

2 5 4 7 ⇒ 2 4 | 5 7
↑ min element (4) | Sorted array | unsorted array

2 4 5 7 ⇒ 2 4 5 | 7
↑ min element (5) | Sorted array | Unsorted array

2 4 5 7
↑ min element (7) ⇒ 2 4 5 7
Sorted array.

ALGORITHM :

```
void Selection_Sort (int arr[], int n)
{  int i, j, min_idx;
for ( i=0;  i<=n-1; i++)
   {
```

```
        min_idx = i;
       for (j = i+1 ; j<n ; j++)
       if (arr[j] < arr [min_idx])
         min_idx = j;
       Swap (&arr [min_idx], &arr[i]);
      }
   }
```

INSERTION SORT

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. for eg., the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted here. Hence the name, insertion sort.

Steps :

1) If it is the 1st element, It is already sorted. Return 1.
2) Pick next element.
3) Compare with all elements in the sorted sub-list.

4) Shift all the elements in the sorted sub-list that is greater than the value to be sorted.

5) Insert the value.

6) Repeat until list is sorted.

Eg (1)

(Pass)							Position Moved
Original	34 ✓	8 ✓	64	51	32	21	
After P=1	8	34 ✓	64 ✓	51	32	21	1
After P=2	8	34	64 ✓	51 ✓	32	21	0
After P=3	8	34	51	64 ✓	32 ✓	21	1
~~After P=4~~	~~8~~ ~~8~~ ~~8~~ ~~8~~	~~34~~ ~~34~~ ~~34~~ ~~21~~	~~51~~ ~~51~~ ~~21~~ ~~34~~	~~64~~ ~~21~~ ~~51~~ ~~51~~	~~21~~ ~~64~~ ~~64~~ ~~64~~	~~32~~ ~~32~~ ~~32~~ ~~32~~	3
After P=4	8 8 8	34 34 32	51 32 34	32 51 51	64 64 64 ✓	21 21 21 ✓	3
After P=5	8 8 8 8	32 32 32 21	34 34 21 32	51 21 34 34	21 51 51 51	64 64 64 64	4

No. of elements : 6.
n−1 : 6−1 => 5 passes.

ALGORITHM :

```
Void Insertionsort ( ElementType A[], int N)
{
    int j, P;
```

```
Element Type Tmp;
 for (P=1; P<N; P++)
{
    Tmp = A[P];
  for (j=P; j>0 && A[j-1] > Tmp; j--)
   A[j] = A[j-1];
   A[j] = Tmp;
  }
}
```

Eg (2): No. of elements : 8
(n-1) : 8-1 ⇒ 7 passes.

Original	14	33	27	10	35	19	42	44	Position Moved
After P=1	14	33	27	10	35	19	42	44	0
After P=2	14	27	33	10	35	19	42	44	1
After P=3	14	27	10	33	35	19	42	44	3
	14	10	27	33	35	19	42	44	
	10	14	27	33	35	19	42	44	
After P=4	10	14	27	33	35	19	42	44	0
After P=5	10	14	27	33	19	35	42	44	3
	10	14	27	19	33	35	42	44	
	10	14	19	27	33	35	42	44	
After P=6	10	14	19	27	33	35	42	44	0
After P=7	10	14	19	27	33	35	42	44	0

SHELL SORT N/D-2018

Shell sort named after its inventor, Donald shell, also referred as diminishing increment sort.

Algorithm:

```
Void
 Shellsort (ElementType A[], int N)
{
    int i, j, Increment;
    ElementType Tmp;
for (Increment = N/2; Increment >0; Increment/=2)
 for (i= Increment ; i<N; i++)
 {
      Tmp = A[i];
   for (j=i; j>= Increment ; j- = Increment)
      if (Tmp < A [j- Increment])
      A[j] =A [j- Increment];
    else
      break;
    A[j] = Tmp;
   }
}
```

Compare elements that are distant apart rather than adjacent. So if there are N elements then we start to solve gap <N.

gap = floor (N/2)

⇒ gap1 = floor (gap/2)

gap2 = floor (gap1/2)

Eg; Original list : 77 62 14 9 30 21 80 25 70 55

N = 10.

gap = floor (N/2) = floor (10/2) = 5.

Pass 1 : 77 62 14 9 30 21 80 25 70 55

So checking the elements distance apart.

⇒ 21 62 14 9 30 77 80 25 70 55

(77, 21) ⇒ swap

(62, 80) (14, 25), (9, 70) (30, 55) ⇒ No swap.

Pass 2 : Gap = floor (5/2) = 2.

⇒ 21 62 $\overset{21}{14}$ 9 30 77 80 25 70 55

⇒ 14 9 21. 25 30 55 70 62 80 77

Pass 3 : Gap = floor (2/2) = 1.

⇒ 14 9 21 25 30 55 70 62 80 77

⇒ 9 14 21 25 30 55 62 70 77 80

(swap) (swap) (swap)

Sorted List :-

9 14 21 25 30 55 62 70 77 80

RADIX SORT

Radix sort is a non-Comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits, which share the same significant position and value.

Radix sort
- Least significant Digit (LSD)
- Most Significant Digit (MSD)

LSD → Process the integer representations starting from least digit and move towards the MSD.

MSD → Process the integer representations starting from maximum digit and move towards the LSD.

```
radix sort (inout the Array : Item Array,
            in n : integer , in d : integer)
// Sort n/ d-digit integers in the array the
for (j=d down to 1)
{
    Initialize 10 groups to empty
    Initialize a counter for each group to 0.
    for (i=0 through n-1)
    {
        k = j th digit of the Array[i]
        Place the Array[i] at the end of group k
        Increase k th counter by 1
    }
    Replace the items in the Array with all the items in group 0, followed by all the items in group 1, and so on.
```

Eg (1) 170 45 75 90 802 24 2 66

$17\underline{0}$ $04\underline{5}$ $07\underline{5}$ $09\underline{0}$ $80\underline{2}$ $02\underline{4}$ $00\underline{2}$ $06\underline{6}$

Consider one's place.

$1\underline{7}0$ $0\underline{9}0$ $8\underline{0}2$ $0\underline{0}2$ $0\underline{2}4$ $0\underline{4}5$ $0\underline{7}5$ $0\underline{6}6$

Consider 10th place.

$\underline{8}02$ $\underline{0}02$ $\underline{0}24$ $\underline{0}45$ $\underline{0}66$ $\underline{1}70$ $\underline{0}75$ $\underline{0}90$

Consider 100th place.

002 024 045 066 075 090 170 802

Sorted array: 2 24 45 66 75 90 170 802.

Eg (2) : 53 89 150 36 633 233 733.

$05\underline{3}$ $08\underline{9}$ $15\underline{0}$ $03\underline{6}$ $63\underline{3}$ $23\underline{3}$ $73\underline{3}$

Consider one's place.

$1\underline{5}0$ $0\underline{5}3$ $6\underline{3}3$ $2\underline{3}3$ $7\underline{3}3$ $0\underline{3}6$ $0\underline{8}9$

Consider 10th place.

$\underline{6}33$ $\underline{2}33$ $\underline{7}33$ $\underline{0}36$ $\underline{1}50$ $\underline{0}53$ $\underline{0}89$

Consider 100th place

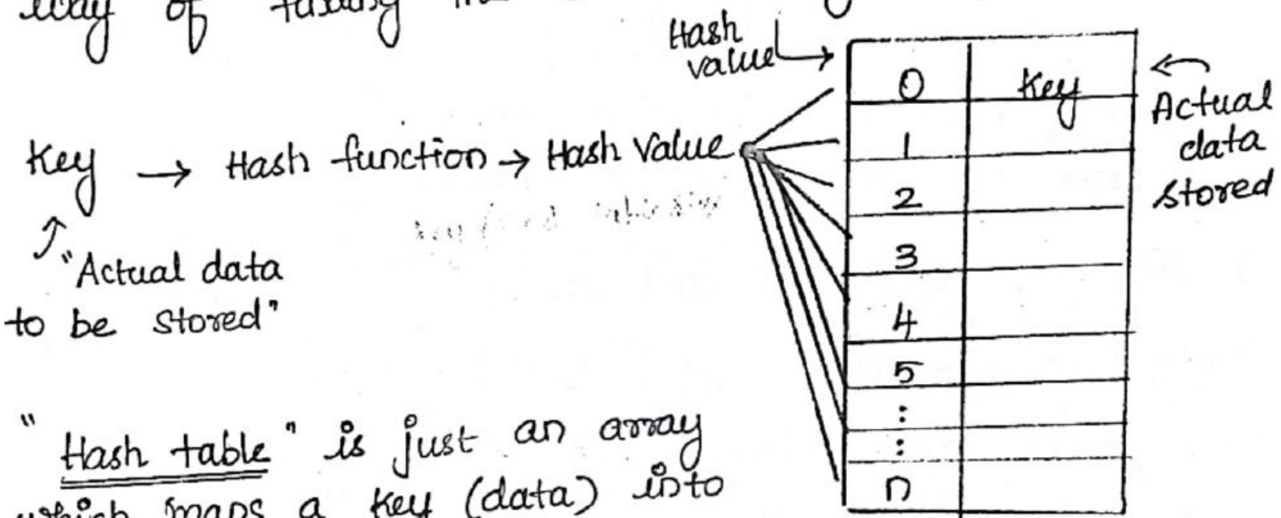
036 053 089 150 233 633 733

Sorted array: 36 53 89 150 233 633 733

HASHING

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 nextlevel searching method when compared with the linear (or) Binary search.

It is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.



"Hash table" is just an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity.

"Hash function" is a function which takes a pice of data (ie., key) as input and outputs an integer (ie., hash value) which maps the data to a particular index in the hash table.

Every entry in the hash table is based on the key value generated using a hash function. The values returned by a hash function is called hash values, hash codes, digests (or) simply hashes.

HASH FUNCTION.

The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Shows the hash table of size m = 11. In other words, there are m slots in the table, from 0 to 10.

Eg; 54, 26, 93, 17, 77 and 31.

Simply takes an item and divides it by the table size, returning the remainder as its hash value.

h(item) = item % m

Item	Hash value	
54	10	(54 mod 11)
26	4	(26 mod 11)
93	5	(93 mod 11)
17	6	(17 mod 11)
77	0	(77 mod 11)
31	9	(31 mod 11)

11) 54 (4
44
10

11) 26 (2
22
4

Once the hash values have been computed, we can insert each item into the hash table at the designated position.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Note that 6 out of 11 slots are occupied. This is referred to as "LOAD FACTOR", and commonly denoted by $\lambda = \frac{\text{number of items}}{\text{Table size}}$.

Eg., $\lambda = 6/11$.

for eg., if the item 44 had been the next item in our collection, it would have a hash value (44 mod 11) = 0. since 77 also had a hash value of 0, we would have a problem.

According to the hash function, two (or) more items would need to be in the same slot. This is referred to as a "collision" also called as "clash". So collision create a problem in hashing technique.

Two methods of Hash functions.

- folding method.
- mid - square method.

folding method :

If our item was the phone number

436-555-4601.

(43, 65, 55, 46, 01) groups of 2.

43+65+55+46+01 ⇒ 210.

If hashtable has 11 slots, then (210 mod 11) = 1.

So, the phone number 436-555-4601 hashes to slot 1.

Some folding methods go one step further and reverse every other piece before the addition.

Eg; 436-555-4601

Group of 2 $(43, \underbrace{65, 55, 46}, 01) \longrightarrow$ Reverse

Reverse $\Rightarrow$ $\underset{\text{No change}}{\underline{\underline{43}}}, 56, 55, 64, \underset{\text{No change}}{\underline{\underline{01}}}$

$43 + 56 + 55 + 64 + 01 = 219.$

$(219 \bmod 11) = 10.$

MID - SQUARE METHOD :

Another numerical technique for constructing a hash function is called the mid - square method. We first square the item, and then extract some portion of the resulting digits.

If the item were 44, then $44^2 = 1936$.

By extracting the middle two digits, 93 and Performing the remainder step we get $(93 \bmod 11) = 5$.

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

We can also create hash functions for character-based items such as strings. The word "cat" can be thought of as a sequence of ordinal values.

Ord ('c') = 99
ord ('a') = 97
Ord ('t') = 116

a	b	c	d	e	f	g	h	i	j	k
97	98	99	100	101	102	103	104	105	106	107
l	m	n	o	p	q	r	s	t	u	v
108	109	110	111	112	113	114	115	116	117	118
w	x	y	z							
119	120	121	122.							

I/p String : c a t (99 + 97 + 116) = 312

(312 mod 11) = 4.

(or)

To use positional value as a weighting factor.

Position :

1 2 3

C A T

$(99\times1)+(97\times2)+(116\times3) = 641$ (641 mod 11) = 3.

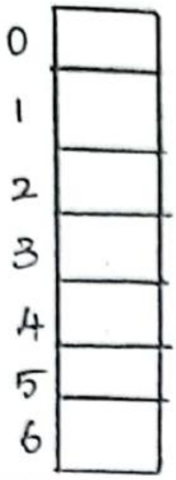
SEPARATE CHAINING (N/D – 2018)

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

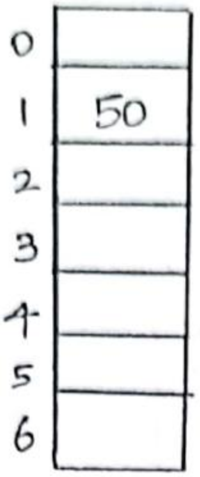
Let us consider a simple hash function as "key mod 7" and sequence of

Keys as

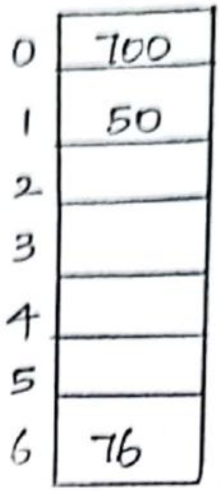
Eg (1) 50, 700, 76, 85, 92, 73, 101.



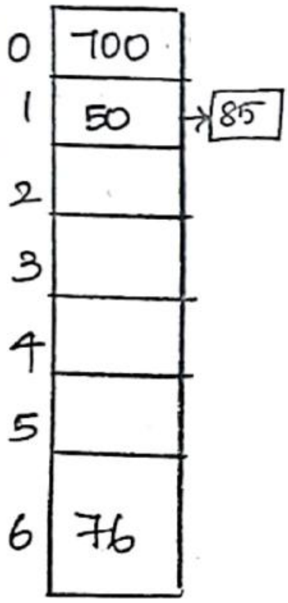
Initial Empty Table



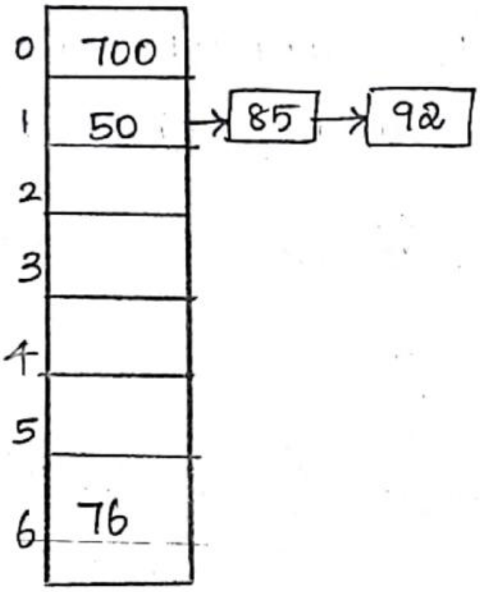
Insert 50
(50 mod 7)
= 1



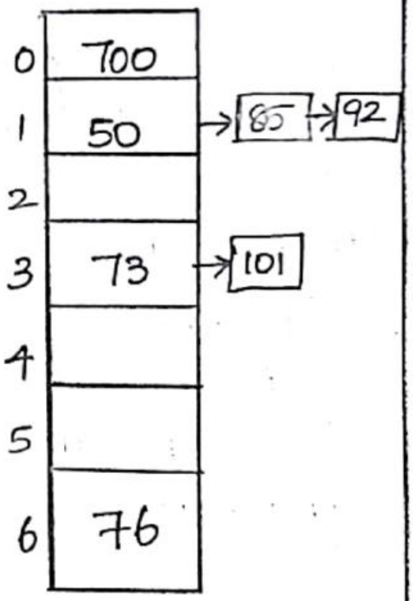
Insert 700 and 76
(700 mod 7) = 0
(76 mod 7) = 6



Insert 85
(85 mod 7)
= 1
Collision occurs,
add to chain



Insert 92
(92 mod 7)
= 1
Collision occurs,
add to chain



Insert 73 and 101
(73 mod 7)
= 3
(101 mod 7)
= 3
Collision occurs,
add to chain.

OPEN ADDRESSING (N/D-2018)

Open addressing (or) closed hashing is a method of collision resolution in hash tables.

formula : h_0 (Key) = (Key mod arraysize).

With this method a hash collision is resolved by Probing (or) searching through alternate locations in the array, until either the target record is found, (or) an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequence includes;

→ Linear probing (N/D-2018)
→ Quadratic probing
→ Double Hashing.

Operations : Insert (Key)
Search (Key)
Delete (Key)

Linear probing :

In which the interval between probes is fixed - often set to 1.

Quadratic probing :

In which the interval between probes increases linearly (hence, the indices are described by a Quadratic function).

Double Hashing : In which the interval between probes is fixed for each record but is computed by another hash function.

LINEAR PROBING.

Eg ; Let us consider a simple hash function as "Key mod 7" and sequence of keys as.

50, 700, 76, 85, 92, 73 and 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	50
2	
3	
4	
5	
6	

(50 mod 7) =1

Insert 50

0	700
1	50
2	
3	
4	
5	
6	76

(700 mod 7) =0

(76 mod 7) =6

Insert 700 and 76

0	700
1	50
2	85
3	
4	
5	
6	76

(85 mod 7) =1 collision occurs

Insert 85. Collision occurs so insert 85 at next free slot.

0	700
1	50
2	85
3	92
4	
5	
6	76

(92 mod 7) =1

Insert 92. Collision occurs as 50 is there at index 1. so insert at next free slot.

0	700
1	50
2	85
3	92
4	73
5	101
6	76

(73 mod 7) =3

(101 mod 7) =3.

Insert 73 and 101. Collision occurs. so insert at next free slot.

QUADRATIC PROBING

$h_0(x) = (\text{Hash}(x) + i \wedge 2)$ % Hash Table size

Eg; Keys : 7, 36, 18, 62. and use Hash-table size as 11.

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

Insert 7.
(7 mod 11)
= 7

0	
1	
2	
3	36
4	
5	
6	
7	7
8	
9	
10	

Insert 36.
(36 mod 11)
= 3.

0	
1	
2	
3	36
4	
5	
6	
7	7
8	18
9	
10	

Insert 18
(18 mod 11)
= 7.
Collision occurs.

0	62
1	
2	
3	36
4	
5	
6	
7	7
8	18
9	
10	

Insert 62.
(62 mod 11)
= 7.
Collision occurs

$\frac{18+1}{11}$ (7 + 1*1) % 11
8.
(18 + 1*1) % 11   18 % 11 + 1 % 11

$$h_1(18) = [(18 + 1 * 1) \bmod 11]$$
$$= [19 \bmod 11]$$
$$= 8$$

$$h_{62} = [(62 + 2 * 2) \bmod 11]$$
$$\Rightarrow [66 \bmod 11]$$
$$\Rightarrow 0.$$

DOUBLE HASHING.

$h_i(x) = (\text{Hash}(x) + i * \text{Hash } 2(x))$ % HashTablesize

$H_1(\text{Key})$ = Key mod tablesize.

$H_2(\text{Key}) = M - (\text{Key} \bmod M)$

M is the prime number < table size

Eg., 37, 90, 45, 22, 17, 49, 55

Table size =10.

0	
1	
2	
3	
4	
5	
6	
7	37
8	
9	

Insert 37
(37 mod 10)
=7

0	90
1	
2	
3	
4	
5	
6	
7	37
8	
9	

Insert 90
(90 mod 10)
=0

0	90
1	
2	
3	
4	
5	45
6	
7	37
8	
9	

(45 mod 10)
=5

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

Insert 22
(22 mod 10)
= 2

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	

Insert 17
(17 mod 10) = 7

Prime no. less than 10

$H_2(17) = 7 - (17 \bmod 7)$
$= 7 - 3$
$= 4.$

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	49

Insert 49
(49 mod 10)
= 9

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37
8	
9	49

Insert 55
(55 mod 10)
= 5.

$H_2(55) = 7 - (55 \bmod$
$= 7 - 6$
$= 1.$

Place 17 in such a way that ④ position from ⑦

REHASHING. (N/D-2018)

Rehashing is the process of re-calculating the hashcode of already stored entries (Key value pairs) to move item to another bigger size Hashmap when load factor threshold is reached.

LOAD FACTOR : It is a measure, "till what load, hashmap can allow elements to put in it before its size is increased.

When the number of item in map, crosses the load factor limit at that time Hashmap double its capacity and hashcode is re-calculated of already stored elements for even distribution of key value pairs accross new buckets.

Steps :

1) create a large table.

2) create a new hash function (for example, the table size has changed)

3) Use the new hash function to add the existing data items from the old table to the new table.

Rehashing Techniques :

1) Linear probing
2) Two-pass file creation
3) Separate overflow area
4) Double Hashing
5) Synonym chaining
6) Bucket Addressing
7) Bucket chaining.

Eg; 13, 15, 24 and 6 are inserted into an open addressing hash table of size 7.

hash function h(x) = x mod 7.

Linear probing with i/p 13, 15, 24 and 6.

0	6
1	15
2	
3	24
4	
5	
6	13

(13 mod 7) = 6

(15 mod 7) = 1

(24 mod 7) = 3

(6 mod 7) = 0

After 23 is inserted.

0	6
1	15
2	23
3	24
4	
5	
6	13

(23 mod 7)

= 2

∴ Table is 70% full.

So a new table is created.

The size of the table is 17, because this is the 1st prime number that is twice as large as the old table size.

∴ New has function is then.,

h(x) = x mod 17.

The old table is scanned, and elements 6, 15, 23, 24 and 13 are inserted into the new table. This entire operation is called as rehashing.

Open addressing hash table after rehashing.

6, 15, 23, 24 and 13.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing is expensive operation with running time O(N).

Extendible Hashing.

Used when the amount of data is too large to fit in main memory and external storage is used.

N records in total to store.

M records in one disk block.

In ordinary hashing several disk blocks are examined, to find an element, a time consuming process.

Extendible hashing: No more than two blocks are examined.

Idea:

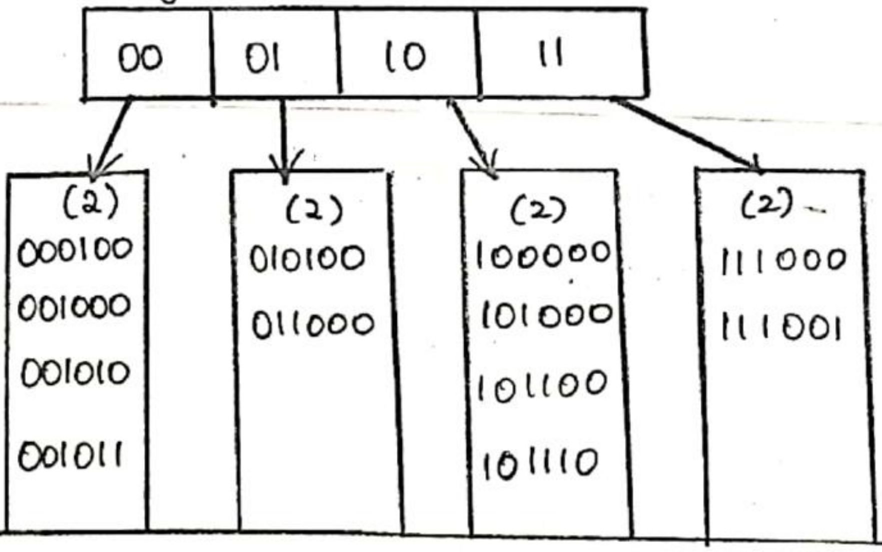
→ Keys are grouped according to the first m bits in their code.

→ Each group is stored in one disk block.

→ If the block becomes full and no more records can be inserted, each group is split into two, and m+1 bits are considered to determine the location of a record.

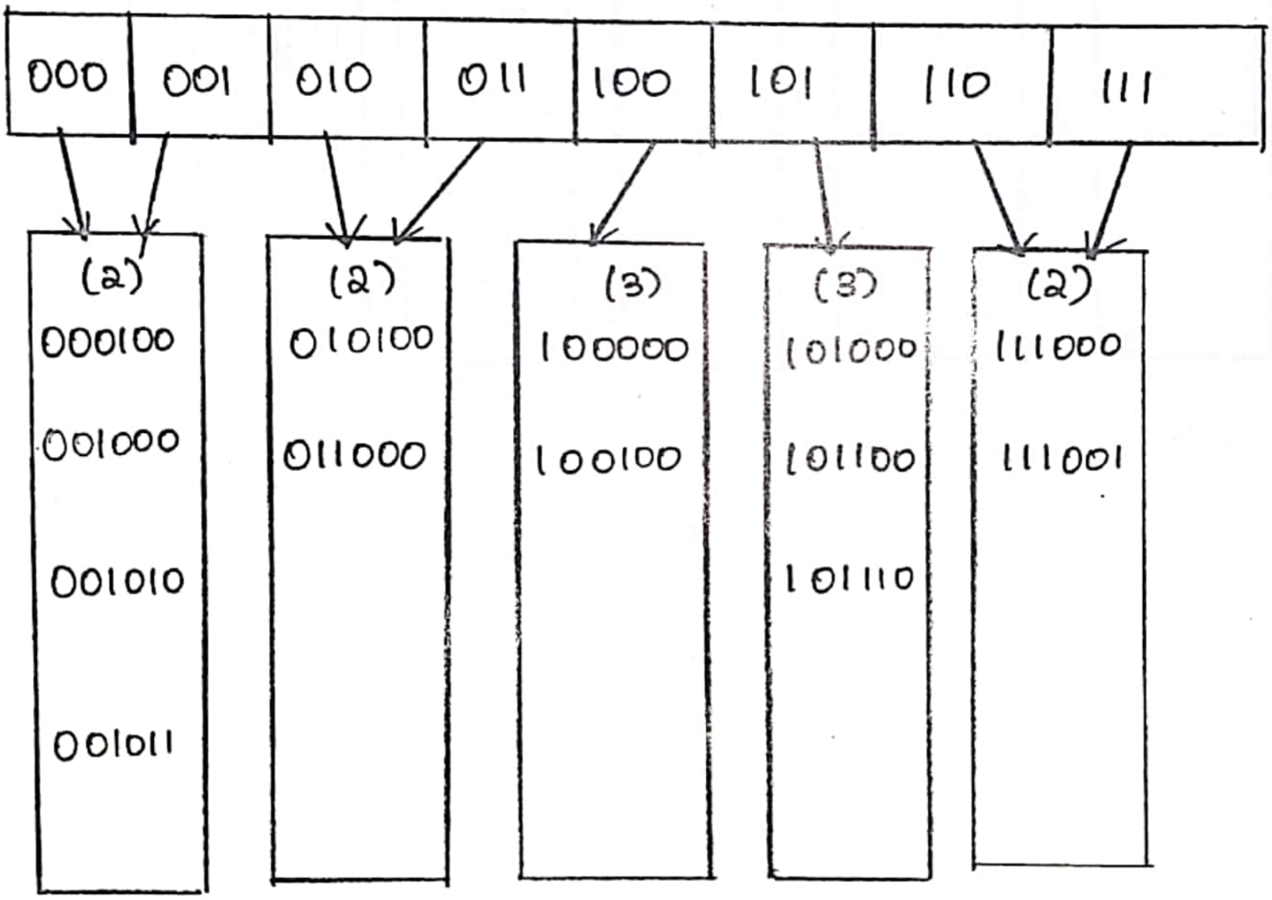
Eg., Suppose data consists of several six-bit integers. Each leaf has up to M = 4 elements.

Original Data



D → Directory (no. of bits used by the root).
The no. of entries in the directory is thus 2^D.

Suppose we need to insert 100100. This would go into third leaf, but it is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing the directory size to 3.



Extendible Hashing after insertion of 100100 and directory split.

Extendible hashing after inserting of
000000 and leaf split.