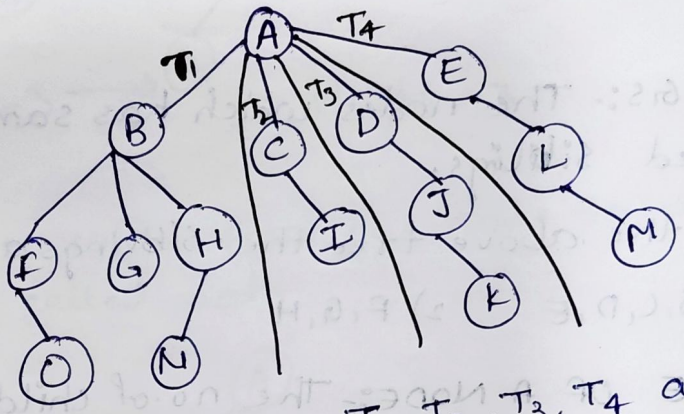
# MODULE-III

## TREES

TREE:- A tree is a non linear data structure in which there should be no loops or cycles.

* A specially designated node called root.
* We have sub-trees like $T_1, T_2, T_3 ---- T_n$




* In the above figure $T_1, T_2, T_3, T_4$ are called Sub-Trees.
* In the above tree A, B, C, D, E, F, G, H, I, J, K, L, M are called nodes and AB, AC, AE, BF, CI, DJ, EL are called edges.

⇒ ROOT:- A node which has no predecessor is called as root.

Ex: In the ~~root~~ above ~~the~~ tree the root is A.

⇒ PARENT:- A node which has successor is called parents node

Ex: In the abv tree the parents nodes are A, B, C, D, E, F, H J, L

⇒ CHILD:- A node which has predecessor is called child node.

Ex: In the abv tree all the nodes except the root 'A' are called child nodes

⇒ LEAF:- A node which has no successor is call. leaf node.

Ex: In the abv tree the leaves are G, I, L, O, N K, M

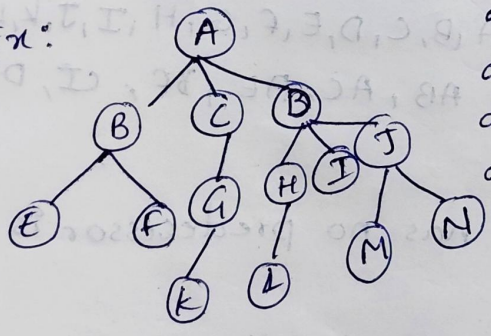
⇒ SIBLINGS:- The nodes which has same parent are called siblings.

Ex: In the above tree the siblings are:

1) B, C, D, E 2) F, G, H

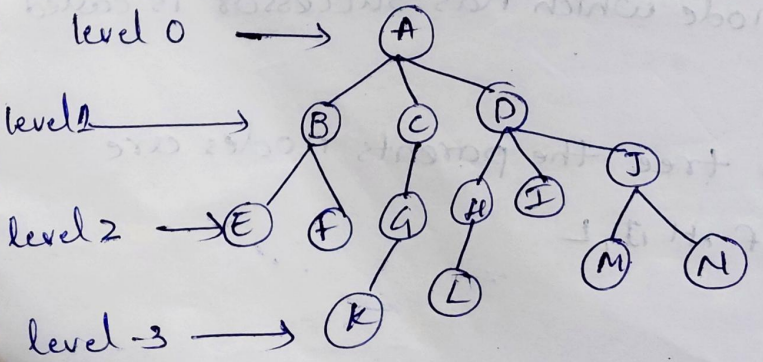
⇒ DEGREE OF A NODE:- The no. of children of a node is called degree of a node

Ex:

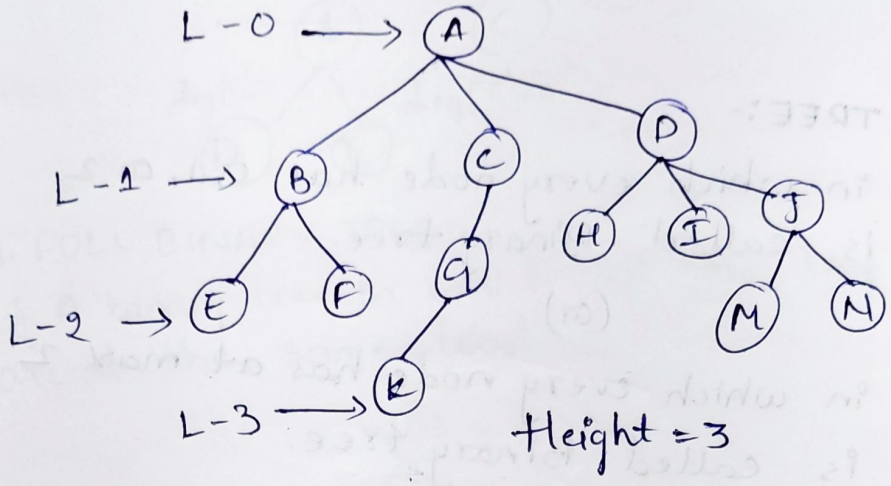



deg A = 3 deg G = 1
deg B = 2 deg H = 1
deg C = 1 deg I = 0
deg D = 3 deg J = 2
deg E = 0 deg K = 0
deg F = 0 deg L = 0

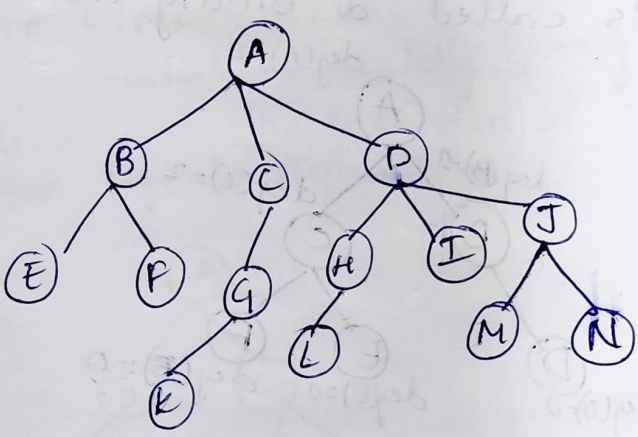
LEVEL OF A NODE:- The no. of edges exits from the node to the root is called level of a node.



HEIGHT OF A TREE:- The no.of levels of a tree is called height of a tree




PATH:- The route from one node to another node is called path.




1) A to K = A → C → G → K

2) D to L = D → H → L

Length of the path:. The no. of edges existing in the path is called length of the path.

1. BINARY TREE:-

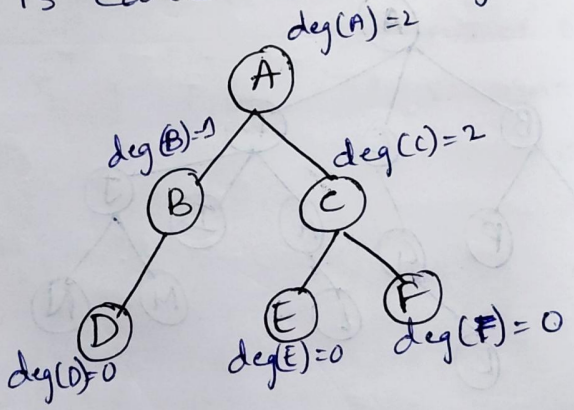
* A tree in which every node has 0, 1, or 2 children is called Binary tree.

(or)

* A tree in which every node has at max 2 children is called Binary tree.

(or)

* A tree in which every node has degree of 0, 1 or 2 is called a binary tree.

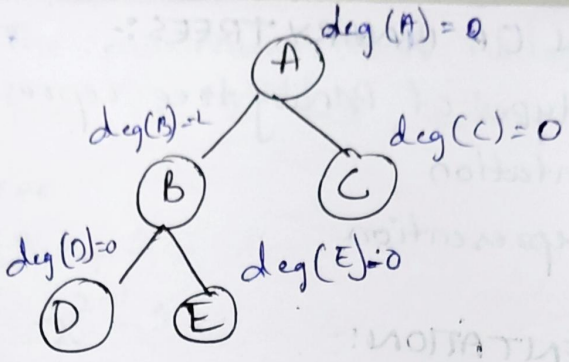



2. STRICTLY BINARY TREE:-

* A tree in which every node has exactly 2 children except the leaf nodes is called Strictly Binary tree.

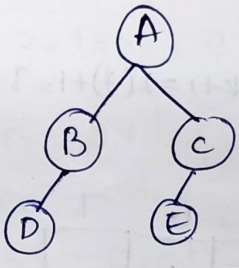
(or)

* A tree which has degrees of each as 0, 2 is called strictly binay tree.



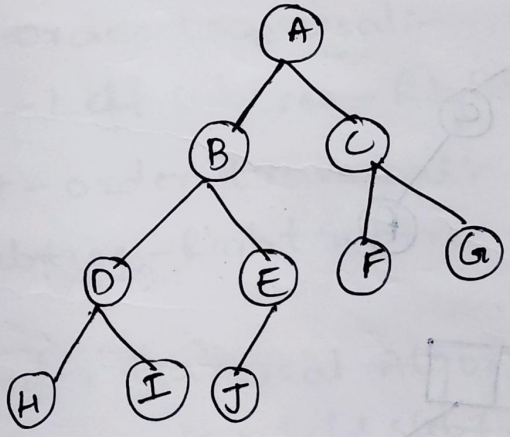

3. FULL BINARY TREE:-

* A binary tree in which all the leaf nodes are at the same level.




4. COMPLETE BINARY TREE:-

* A binary tree which is strictly Binary tree except the last level and the insertion of nodes must be done from left to right.

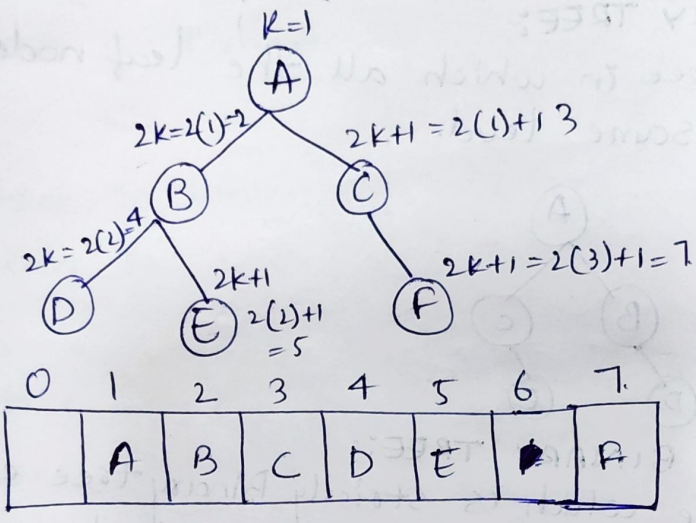


## ⇒ REPRESENTATION OF BINARY TREES:-

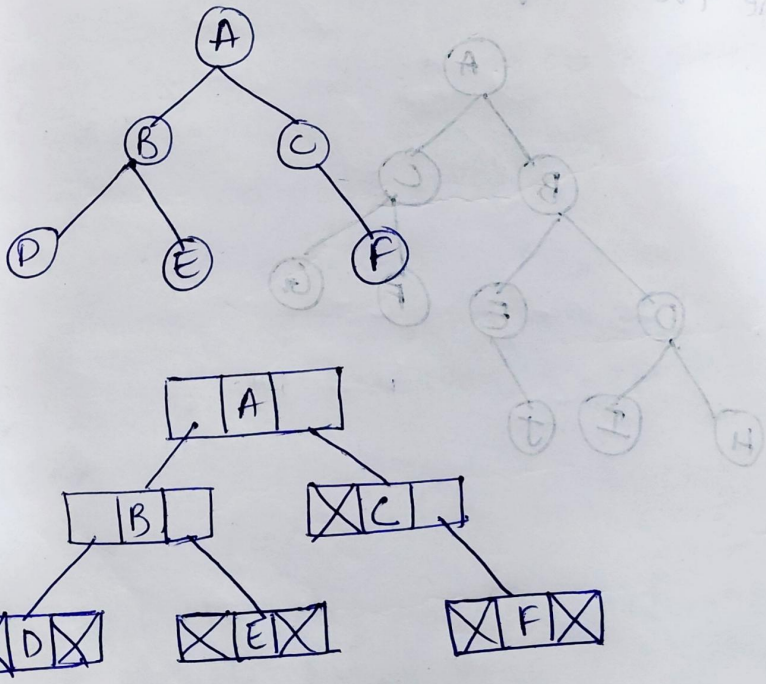
There are two types of Binary tree representation.

1. Array representation
2. Linked list representation

### 1. ARRAY REPRESENTATION:




### 2. LINKED LIST REPRESENTATION:-



NOTE:- The maximum no. of nodes present in a binary tree is $2^{n+1}-1$, where n is no. of levels of a tree.

Ex: If n = 2.

$$2^{2+1}-1$$
$$= 2^3-1$$
$$= 7$$

⇒ BINARY TREE TRAVERSALS:-

* There are 3-types of binary tree traversals

1) In-order traversal
2) pre-order traversal
3) post-order traversal.

1) In-order traversal:-

Left Subtree - Root - Right Subtree (LTR)

2) Pre-order traversal:-

Root - Left subtree - Right subtree (TLR)

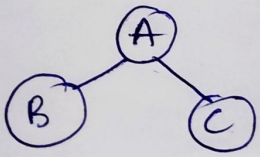
3) post-order traversal:-

Left Subtree - Right subtree - Root (LRT)

⇒ Inorder traversal Algorithm:-

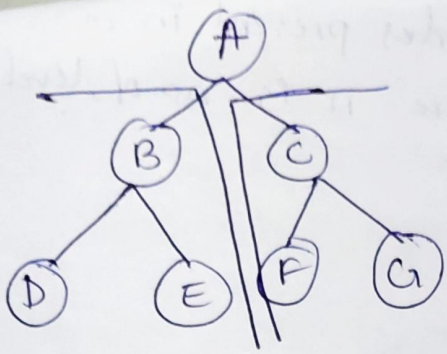
1. Traverse the left subtre in inorder
2. Visit the root
3. Traverse the Right-subtree in inorder

Ex:

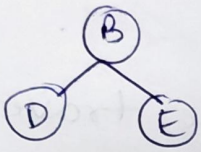



~~Inorder~~

Inorder : BAC




1) Traverse the left subtree in in-order:

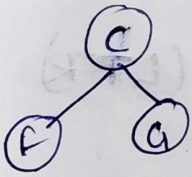



In-order : DBE

2) Visit the root:

Root: A

3) Traverse the Right subtree in inorder



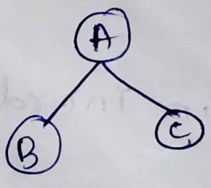

Inorder: FCG

The in-order traversal is : DBE A FCG

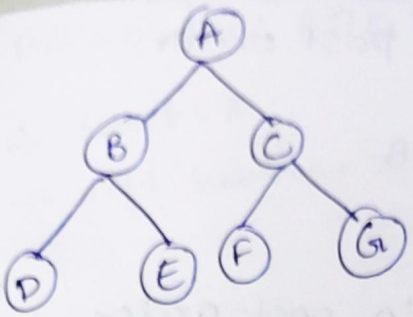
⇒ Pre-order traversal algorithm:-

1) Visit the root
2) Traverse the left subtree in pre-order
3) Traverse the Right subtree in preorder.

Ex:




pre-order traversal: ABC




1) Visit the root

Root : A

2) Traverse the left subtree in pre-order

Pre-order : BDE

3) Traverse the Right subtree in pre-order

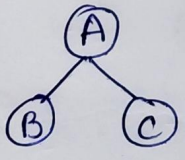
pre-order : CFG

The pre-order traversal is : ABDECFG

⇒ Post-order traversal algorithm:-

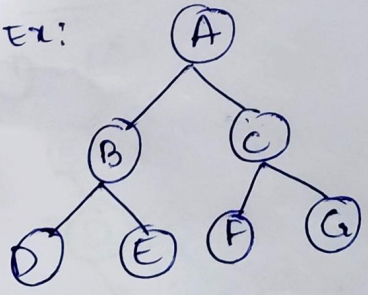
1) Traverse the left subtree in post-order
2) Traverse the Right subtree in post-order
3) Visit the root.

Ex:

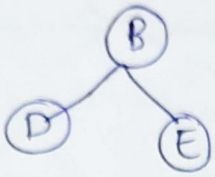



Post-order : BCA

Ex:

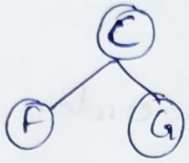


1. Traverse the left subtree in post order




post-order : DEB

2. Traverse the Right subtree in post-order



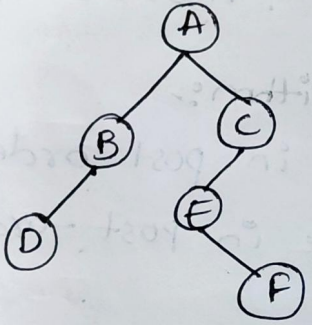

post order : FGC

3. Visit the ~~node~~ root.

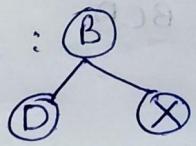
root : A

The traversal ~~of~~ in post order is DEBFGCA.

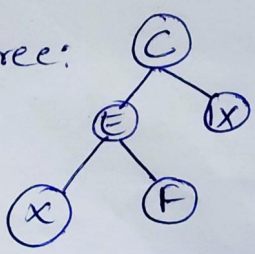
Ex:




In-order: DBAEFC.

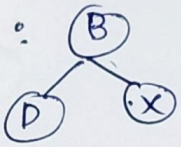
→ left subtree :  DB

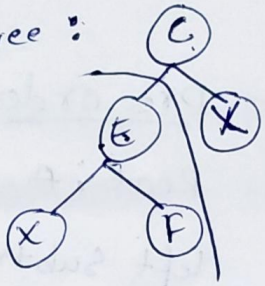
→ Root : A.

→ Right subtree:  EFC.

Pre-order: ABDCEF

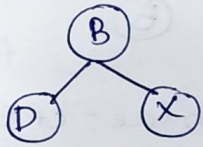
→ Root: A

→ left subtree:  = BD

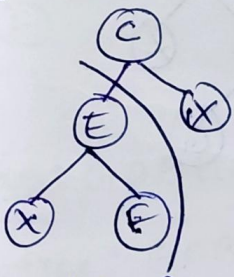
→ Right subtree:  = CEF

Post-order: DBFECA.

→ left subtree: DB

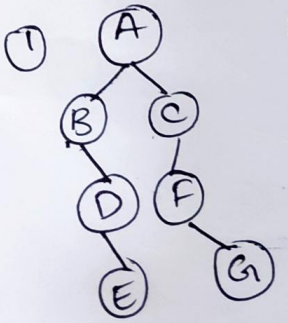


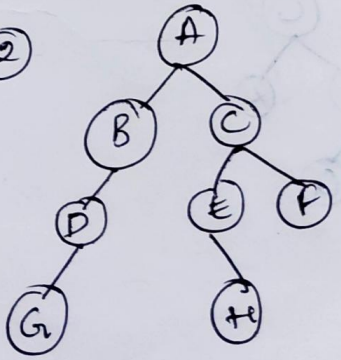
→ Right subtree: FEC

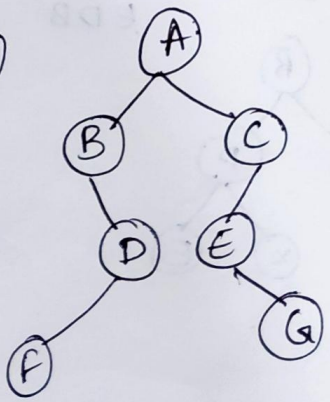


→ Root: A

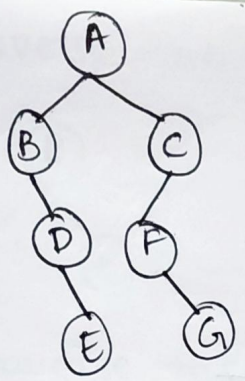
WORK:

(1) 

(2) 

(3) 

1)



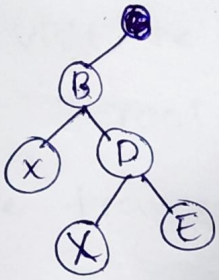

In-order: DEBAFGC

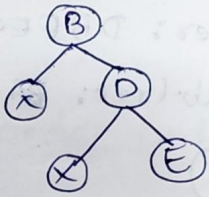
Pre-order: ABDECFG

left sub-tree: DEB

Root: A

left subtree: BDE

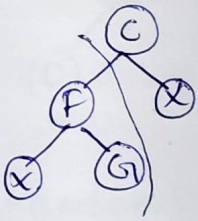


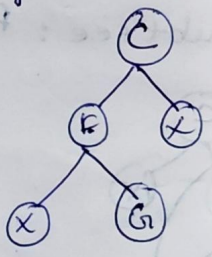




Root: A

Right sub-tree: FGC

Right-sub-tree: cfG.



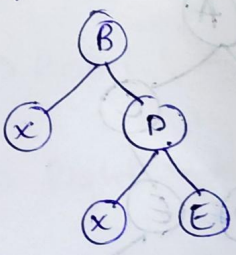



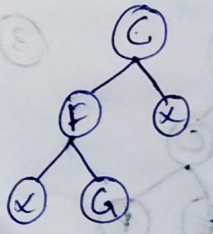

Post-order: EDBGFCA

left sub-tree: ~~DEB~~ EDB

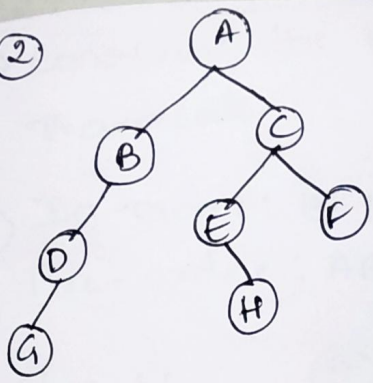
Right sub-tree: GFC

Root: A



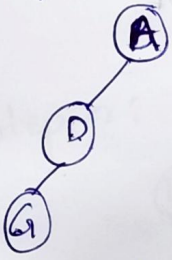



(2)



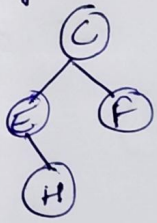

In-order: GDBAEHCF

left-subtree: ~~DGA~~GDB




Root: A

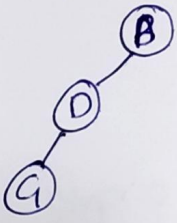
Right-subtree: EHCF



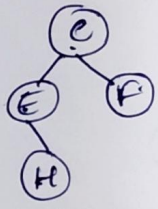

Pre-order: ABDGCEHF

Root: A

left-subtree: BDG

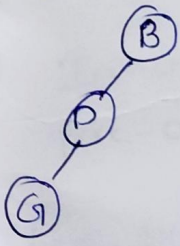


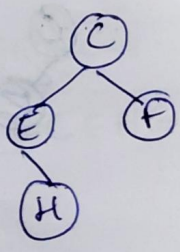

Right-subtree: CEHF



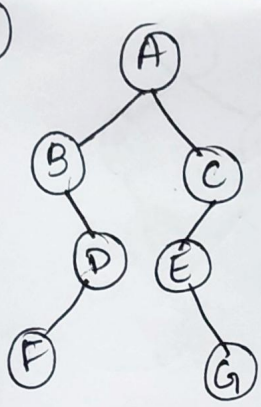

Post-Order: GDBHEFCA

left subtree: GDB Right-subtree: HEFC Root: A



3




In-order: BFDAEGC

left subtree: BFD

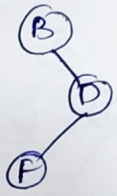

Root: A

Right-subtree: EGC


Pre-order:- ABDFCEG

Root: A

left subtree: BDF

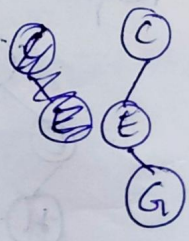



Right-subtree: CEG


Post-order: FDBGECA

left sub tree: FDB


Right-subtree: GEC

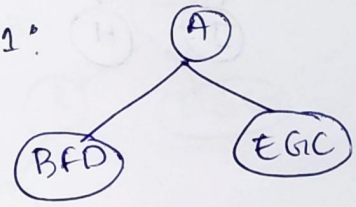



Root: A

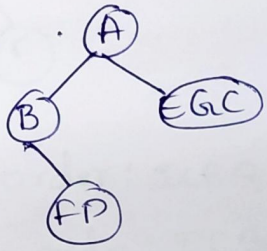
⇒ Construct the binary tree for the given tree traversals.

1) In-order : BfDAEGC
Pre-order : ABDfCEG

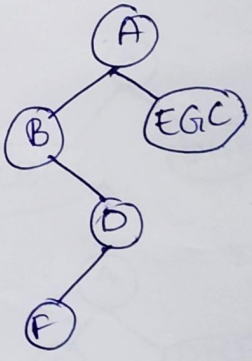
Step-1:



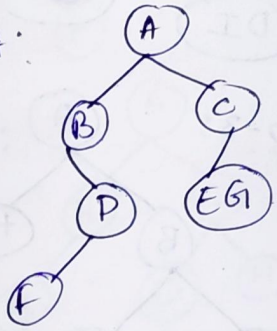

Step-2:



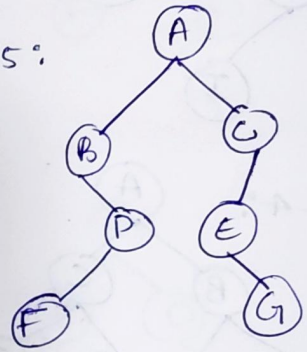

Step-3:




Step-4:

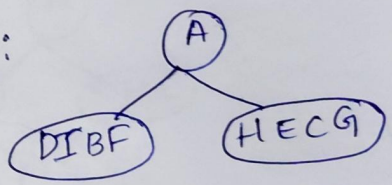



Step-5:

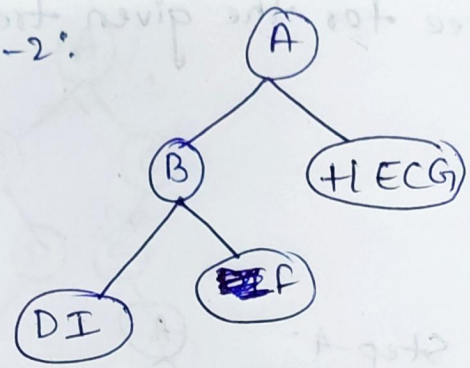



2) In-order : DIBfAHECG
pre-order : ABDIFCEHG

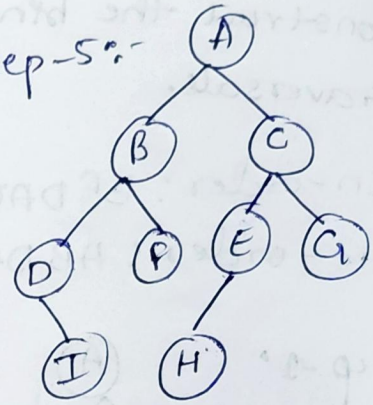
Step-1:



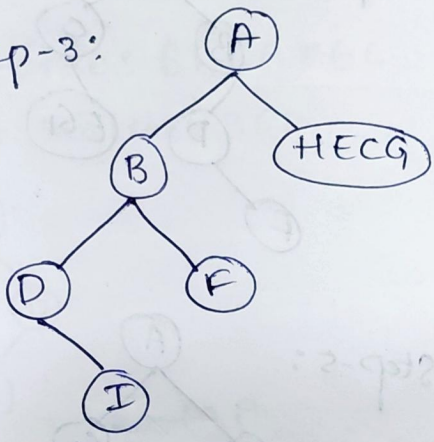
Step-2:



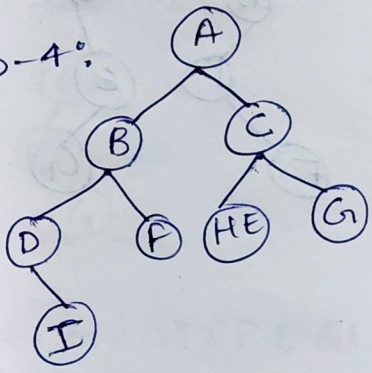

Step-5:-




Step-3:



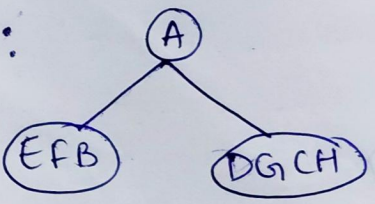

Step-4:



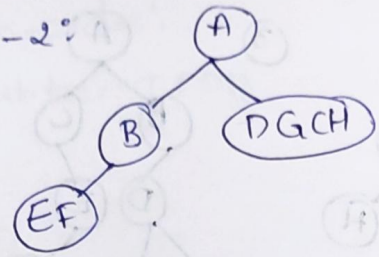

3) In-order: EFBADGCH

Pre-order: ABEFCDGH.

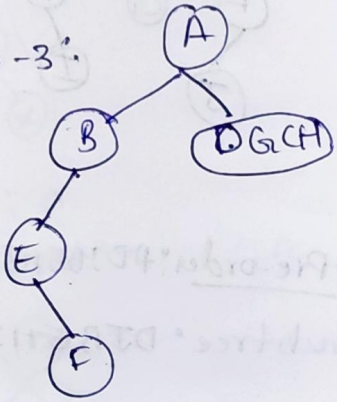
Step-1:



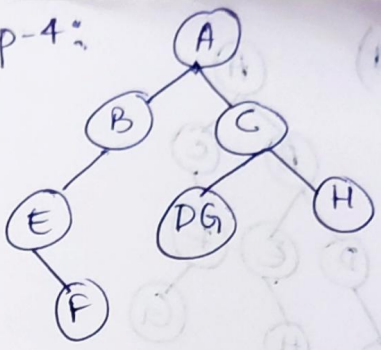
Step-2:



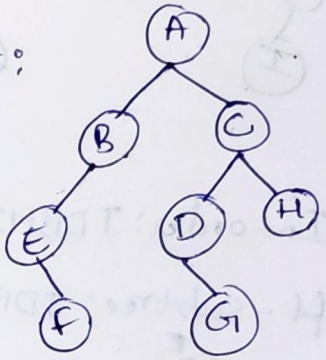

Step-3:




Step-4:



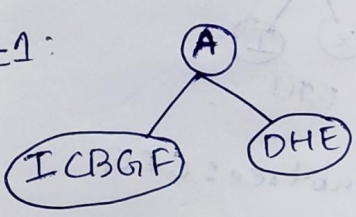

Step-5:



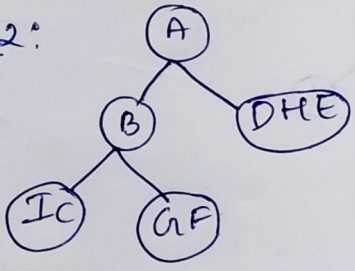

④ In-order : ICBGFADHE

Post-order : ICGFBHEDA

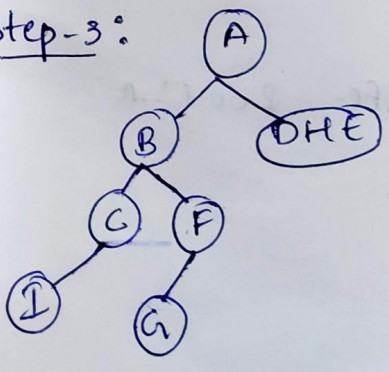
Step-1:



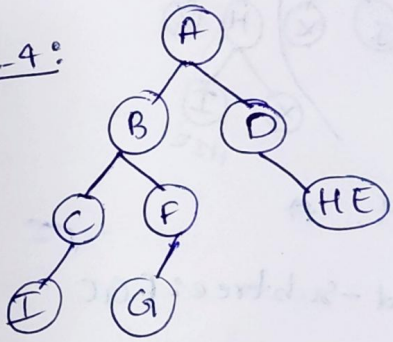

Step-2:



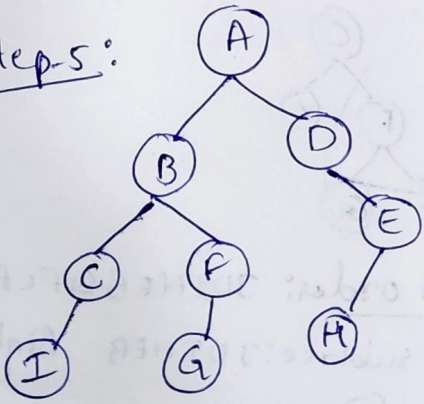

Step-3:



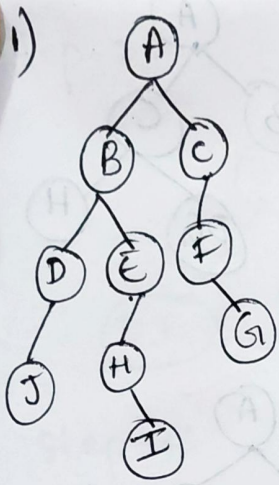

Step-4:



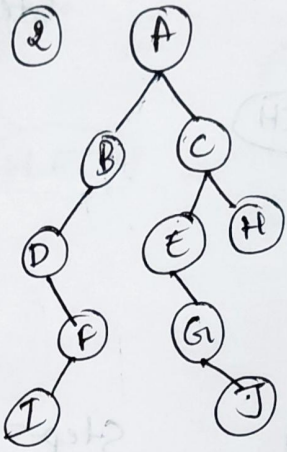

Step-5:



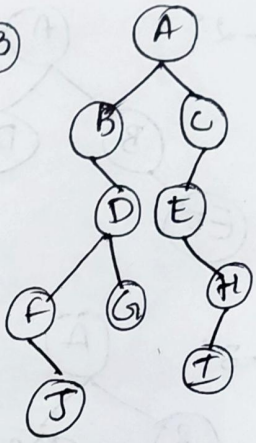
1)




2)

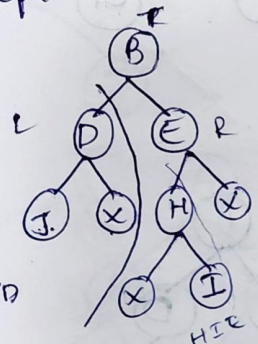


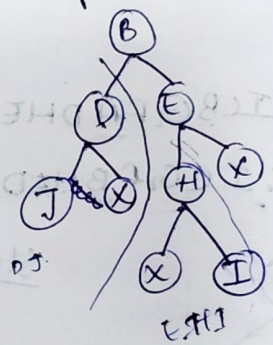

3)




1) In-order: JDBHIEAFGC ~~Post~~ Pre-order: ADJBEHICFG

Left-subtree: JDBHIE Left subtree: DJBEHI

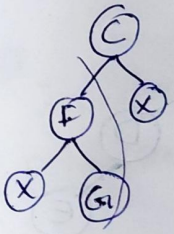


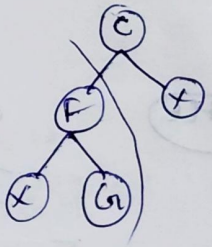




Root: A

Right-subtree: FGC.

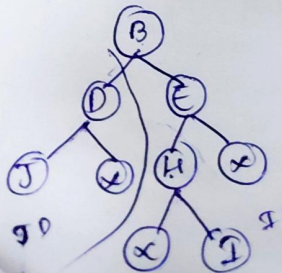
Right subtree: CFG

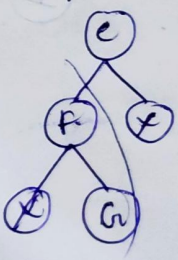



Post order: JDIHEBGFCA.

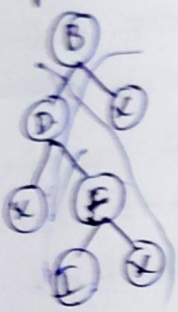
Left subtree: JDIHEB Right-Subtree: GFC. Root: A



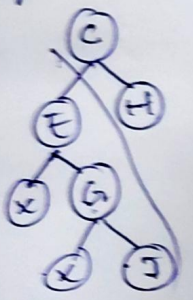
2) In-order:

left-subtree: IFDB




Root: A

Right subtree: GJEC



## ⇒ BINARY SEARCH TREE:- (BST)

* A binary tree in which the elements less than or equal to the root are present at the left subtree of the root and the elements greater than the root are present at the right subtree of the root is called binary search tree.

## ⇒ OPERATIONS ON BINARY SEARCH TREES:-

The different types of operations performed on binary search trees are:

1) Insertion
2) Deletion
3) Lookup | search.

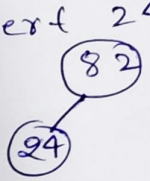
## ⇒ INSERTION:-

In this operation the elements less than the root will be inserted at the left subtree of the root and if the element is greater or equal then the root it will be inserted at the right subtree

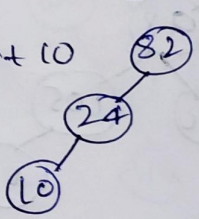
82, 24, 10, 65, 98, 105, 12, 75, 90, 112, 65

Step-1: Insert 82

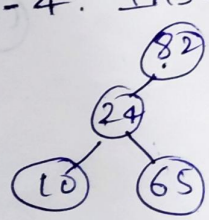

Step-2: Insert 24



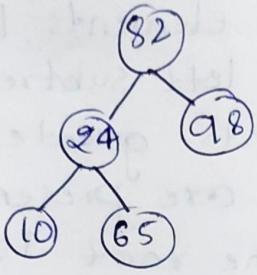

Step-3: Insert 10



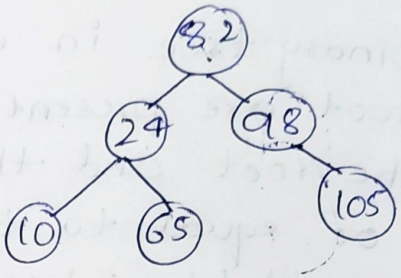

Step-4: Insert 65



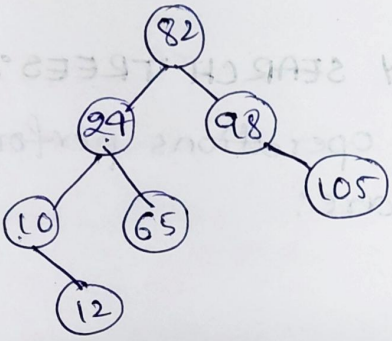
Step-5: Insert 98



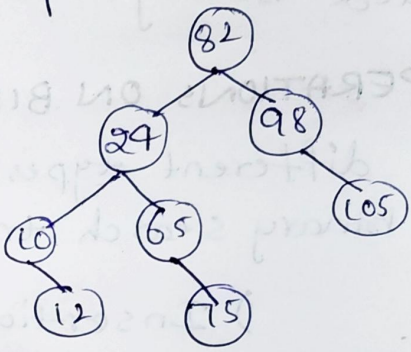

Step-6: Insert 105



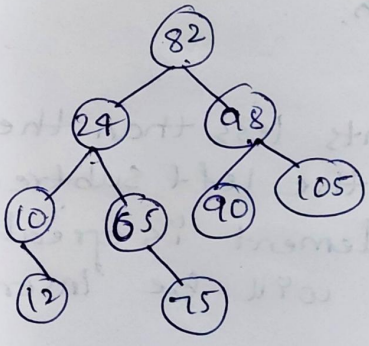

Step-7: Insert 12



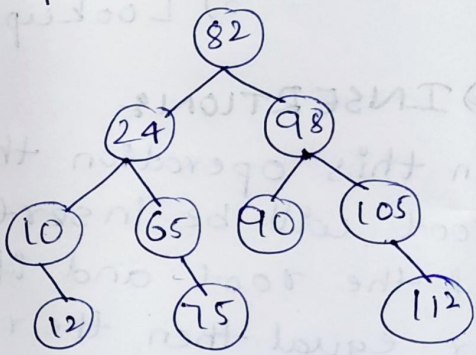

Step-8: Insert 75



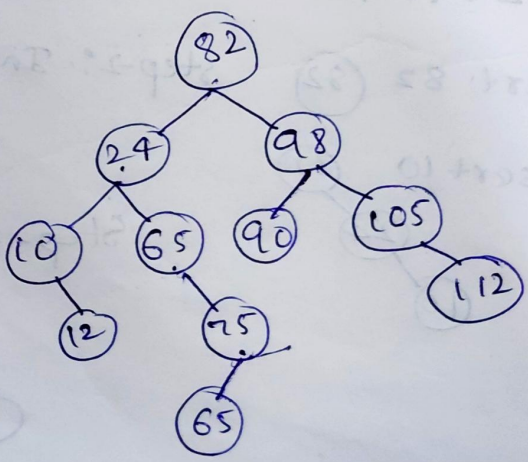

Step-9: Insert 90




Step-10: Insert 112




Step-11: Insert 65

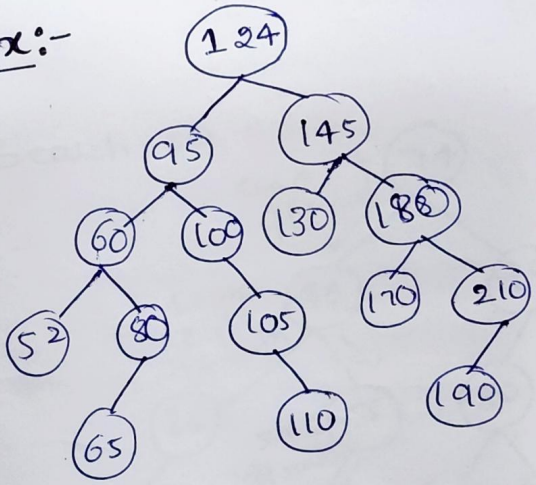


## ⇒ DELETION OPERATION:-

* In this operation if we want to delete an element from BST, then we need to follow the below rules:

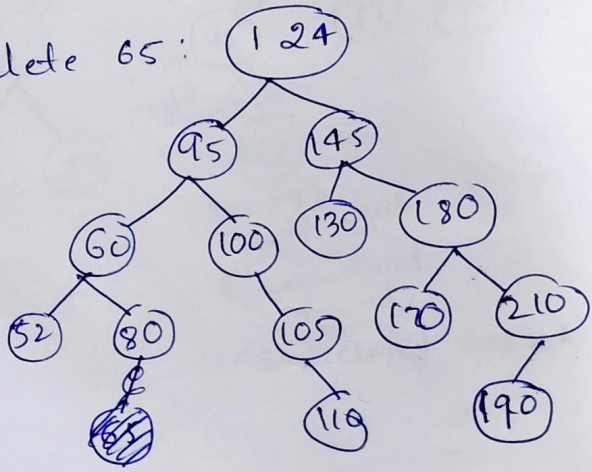
1. If the element to be deleted is a leaf node then, we can delete it directly.
2. If the element to be deleted is a parent node then: replace it with:
   a) greatest element from its left subtree
   b) smallest element from its right subtree

**Ex:-**

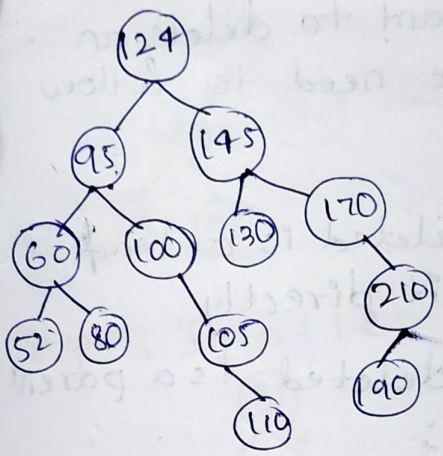



1. Delete 65
2. Delete 180
3. Delete 95
4. Delete 145
5. Delete 80
6. Delete 60
7. Delete 124

1. Delete 65:



2. Delete 180

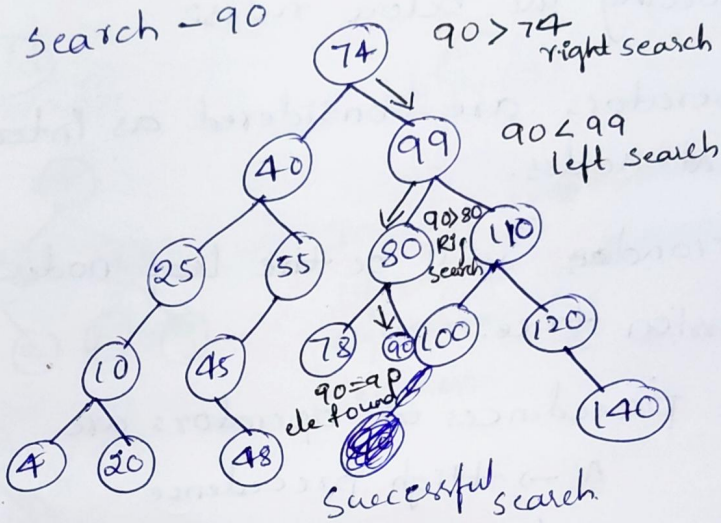



3. Delete 95

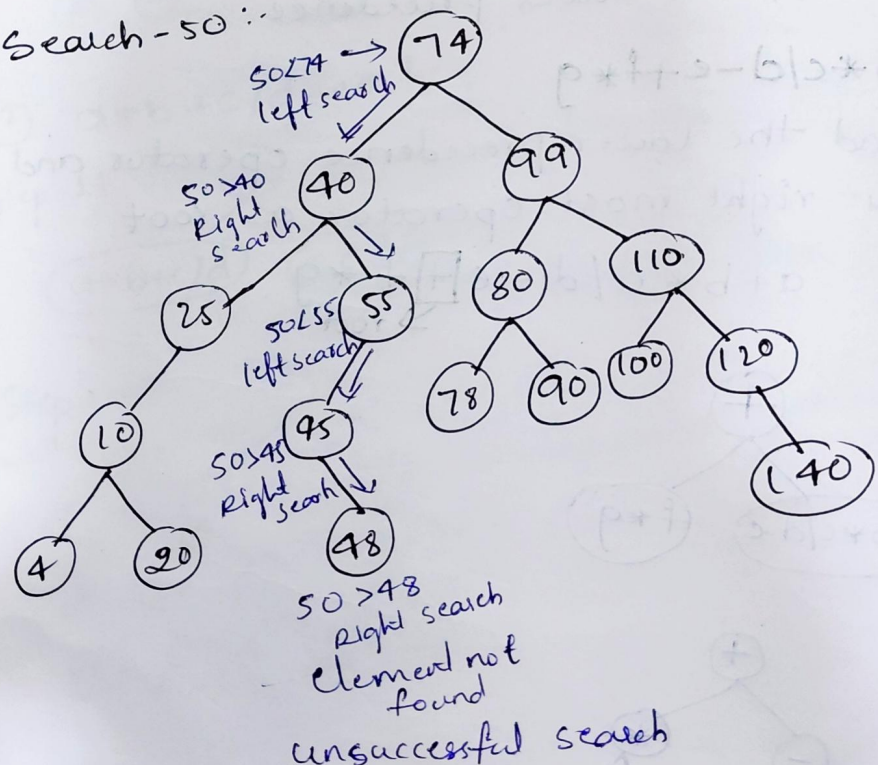
## ⇒ LOOKUP / SEARCH OPERATION:-

This operation is used to find whether an element is present in BST or not.

Ex:- Search - 90




Search - 50 :-



## ⇒ EXPRESSION TREES:-

The binary tree which is a combination of operators & operands can be constructed by following the below rules:

1) Operators are considered as internal nodes or parent nodes.

2) Operandas will be the leaf nodes of the expression tree.

3) The precedences of operators are

^ → High precedence

*, / → next high precedence

+, - → lower precedence

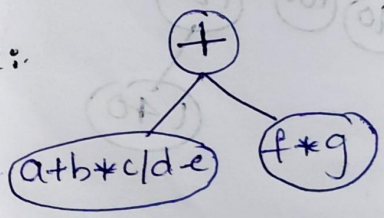
Ex: a+b*c/d-e+f*g

Step-1: find the lower precedence operator and take the right most operator as root

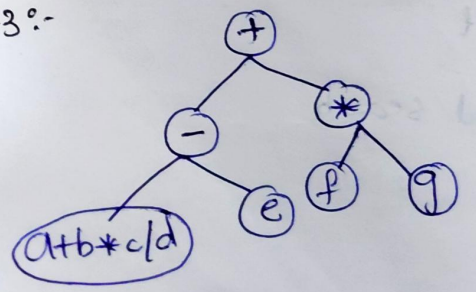
a+b*c/d-e[+]f*g

↘ root

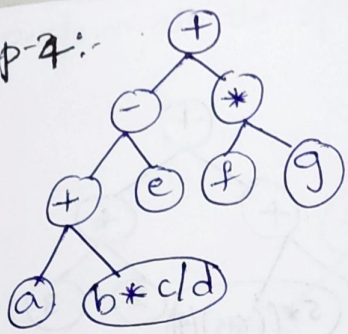
Step-2:-



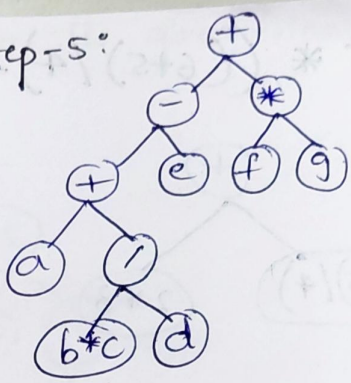

Step-3:-



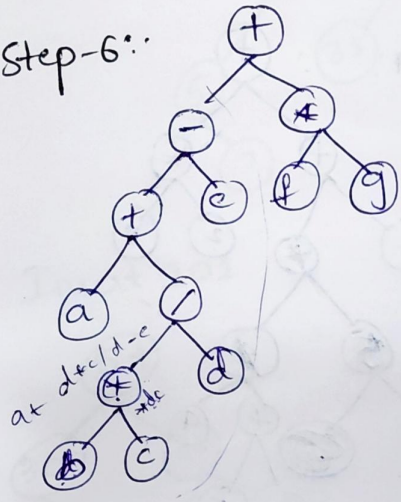
Step-4:-




Step-5:




Step-6:




Inorder: a+b*c/d-e+f*g

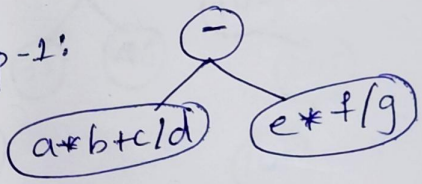
left subtree: a+b*c/d-e

Right subtree:- f*g

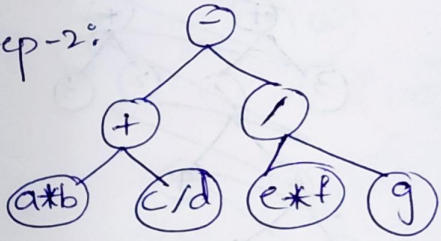
preorder:-

(2) a*b+c/d-e*f/g

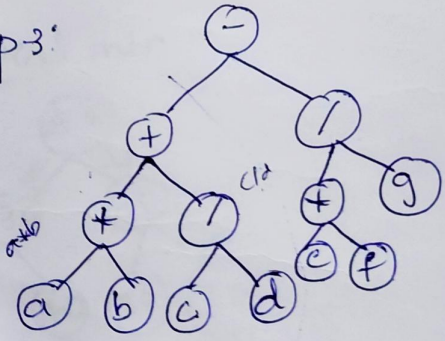
Step-1:




Step-2:



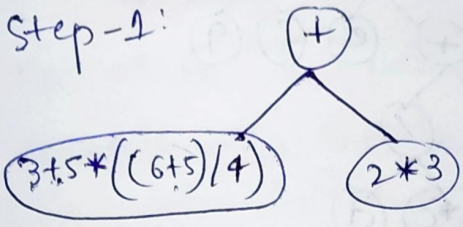

Step-3:



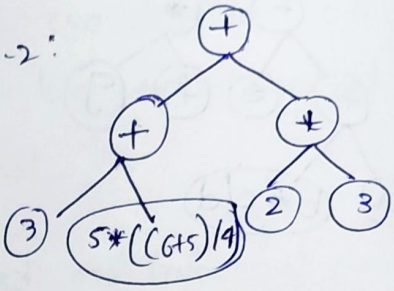

Inorder:- a*b+c/d-e*f/g

③ 3+5 * ((6+5)/4)+2*3

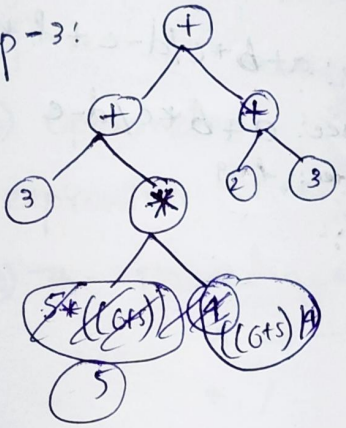
Step-1:



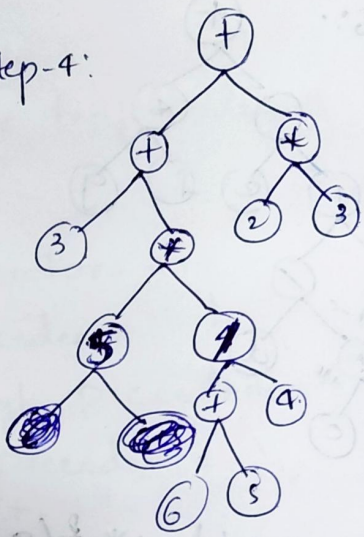

Step-2:



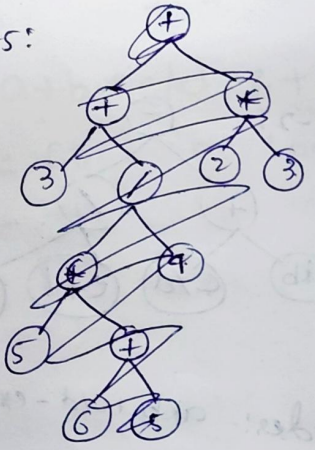

Step-3:



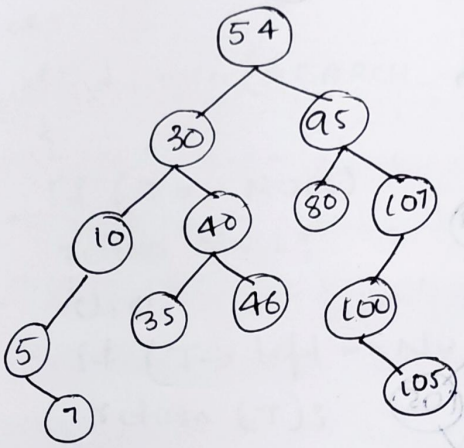

Step-4:



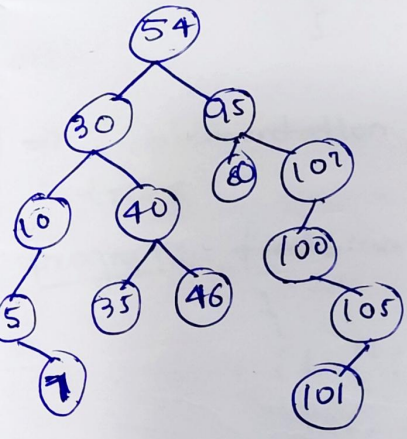

Step-5:



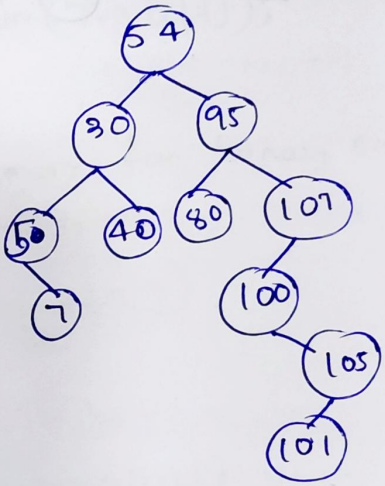
⇒ Perform all the operations on the given binary tree.



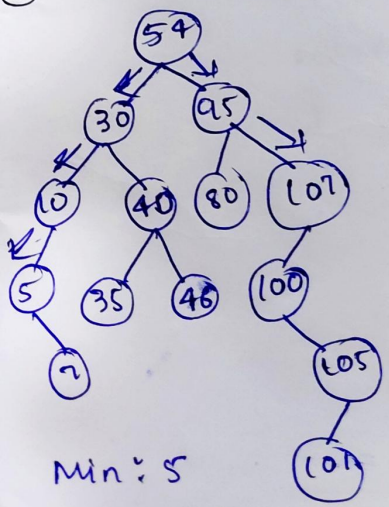

1. Insert 101




2. Delete 10

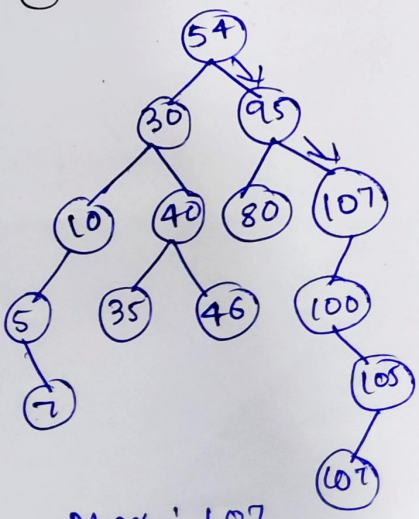



(3). Find min.



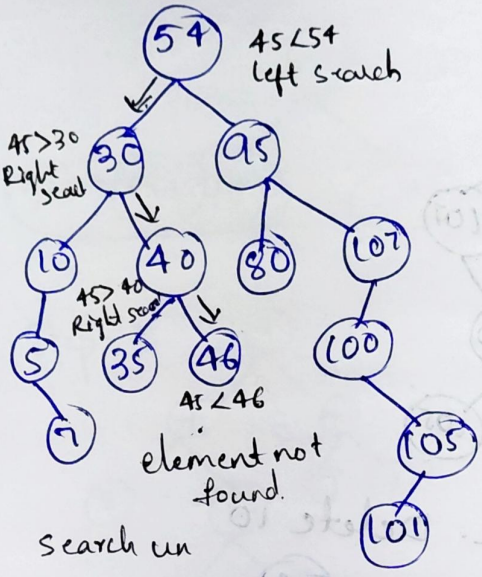

Min: 5

(4) Find maximum




Max: 107.

(5) Search: 45




search un

(6) Search 80

⇒ Recursive Implementation of find_min for binary search trees.

Program:

```
find_min (SEARCH_TREE T)
{
if (T==NULL)
return NULL;
else
if (T->left==NULL)
return (T);
else
return (find_min (T->left));
}
```

⇒ Implementation of find_max for binary search trees

Program:

```
find_max (SEARCH_TREE T)
{
if (T!=NULL)
while (T->right!=NULL)
T=T->right;
return T;
}
```

⇒ Make_null :-

This operation is mainly used for initialization. Some programs prefer to initialize the first element as a one-node tree, but our implementation follows the recursive def of trees more closely. It is also a simple routine, as evidenced below.

```
make_null (void)
{
return NULL;
}
```

⇒ Find operation :-

```
find (element_type, x, search_tree)
{
if (T == NULL)
return NULL;
if (x < T -> element)
return (find (x, T -> left));
else
if (x > T -> element)
return (find (x, T -> right));
else
return T
```

⇒ AVL TREES:

Adelson | Landis

Velskii

* AVL tree is a balanced binary search tree.
* Balanced means the difference btw the height of left subtree and height of right subtree which is called the balance factor, is 0, 1 or -1.

* ⇒ OPERATIONS ON AVL TREES:-

The diff types of operations performed on AVL trees are:

1) Insertion
2) Deletion
3) Search.

⇒ INSERTION OPERATION:-

Inserting an element in an AVL tree is same as insertion in BST.

* When we insert a new in AVL tree the tree may become unbalanced, to make the tree balanced we use rotations.
* The diff types of rotations used for the insertion of the elements are:

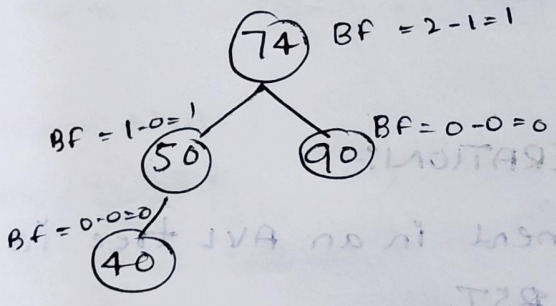
1. L L rotation
2. R R rotation
3. L R rotation
4. R L rotation.

1) L.L Rotation:-

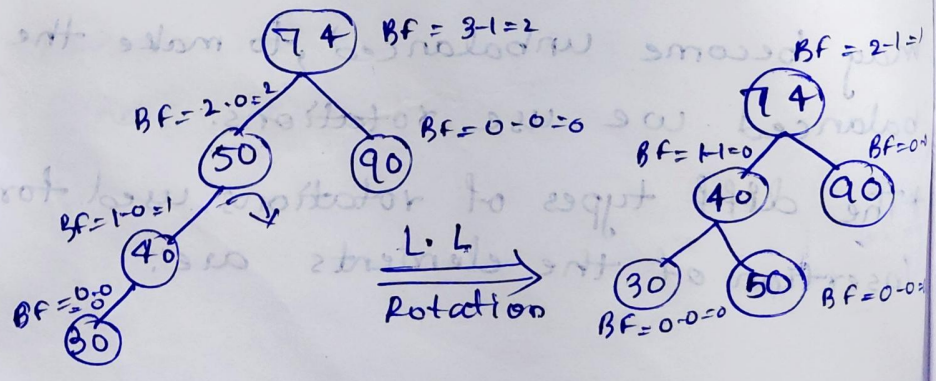
When we insert a new node in AVL tree, at left subtree as left child the tree may get unbanlanced in this case to make the tree balanced we apply L.L rotation at the unbalanced node.

* The L.L rotation is also known as right rotation.

Ex: Let the given tree is:




After inserting 30 the AVL tree is:



2) R.R Rotation:-

When we insert a new node in AVL tree, at Right subtree as right child the tree may get unbalanced in this case to make the tree balanced we apply R.R rotation at the unbalanced node.

* The R.R rotation is also known as left rotation.

Ex:

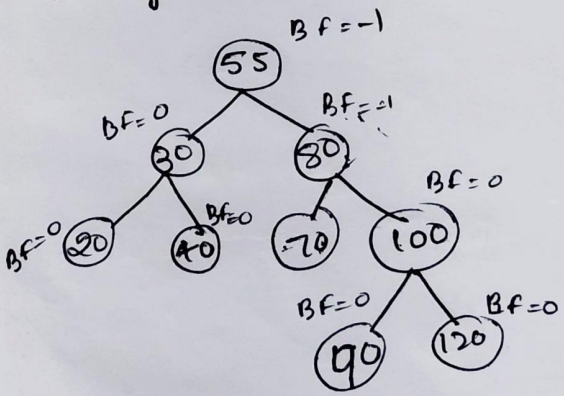
## ⇒ DELETION OPERATION:-

* In this operation when we delete an element from the AVL tree the tree may get unbalanced. To balance the tree we use rotations.
* If we delete an element and the deleted element is present at the left subtree of the unbalanced node and the deleted node balance factor is 0, 1 or -1 then we use the following rotations:

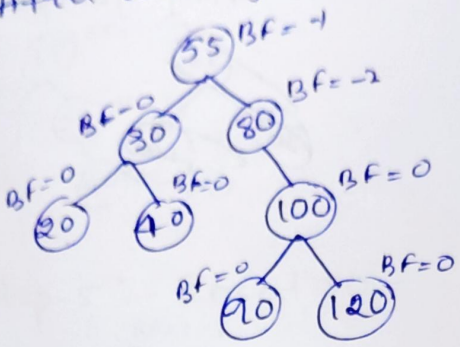
$L_0 \longrightarrow$ Same as RR rotation
$L_1 \longrightarrow$ Same as RL rotation
$L_{-1} \longrightarrow$ Same as RR rotation

*where, $L_0$ means the deleted is present at the left subtree of the unbalanced node and the deleted node B.F is '0' Similarly $L_1$ and $L_{-1}$

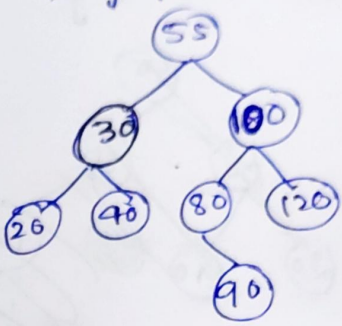
Ex: Let the given AVL tree is:



After deleting 70




After applying LO rotation



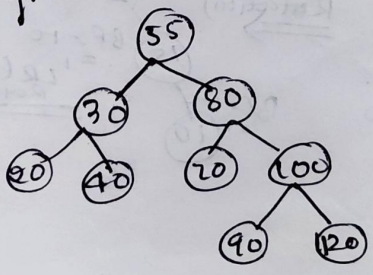

* If we delete an element and the deleted element is present at the right sub tree of the unbalanced node and the deleted node balance factor is 0, 1 or -1 then we use following rotations:

R0 } ⟶ LL Rotation
R1 }

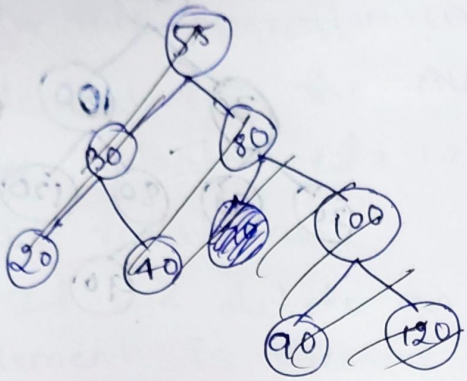
R-1 ⟶ LR rotation

* where, R0 means the deleted node is present at the right subtree of the unbalanced node and the deleted node B.f is 0 Similarly L1 and L-1

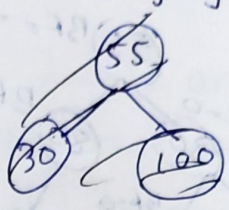
Ex:- Let the given AVL tree is:



After deleting 20




After applying RO rotation



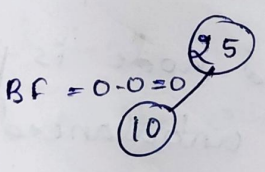

## ⇒ CONSTRUCTION OF AVL TREE :-

25, 10, 15, 4, 0, 50, 8, 70, 6, 12, 15

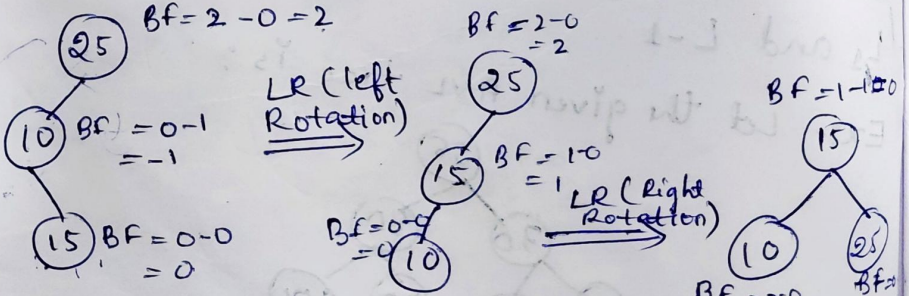
Step-1 :- Insert 25

BF = 0-0
(25) = 0 (where, BF means Balance factor)

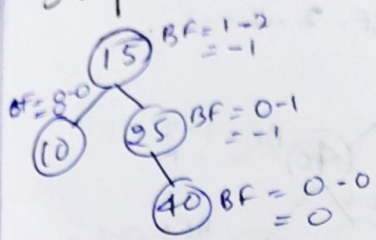
Step-2 :- Insert 10



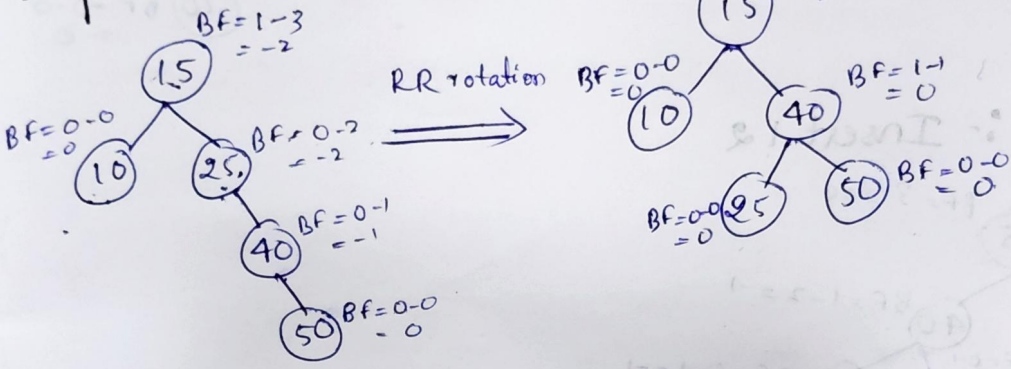

Step-3 :- Insert 15



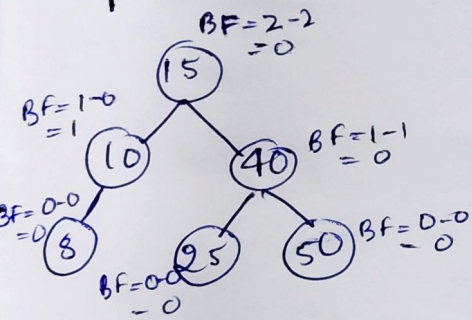
Step 4 :- Insert 40



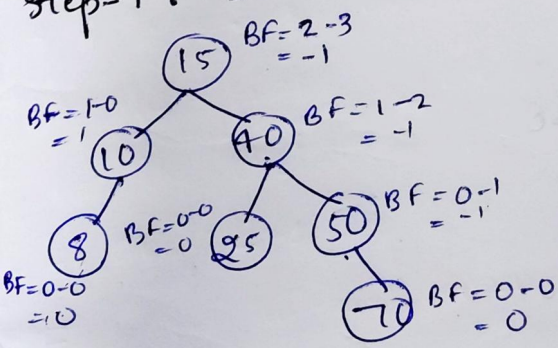

Step-5 :- Insert 50



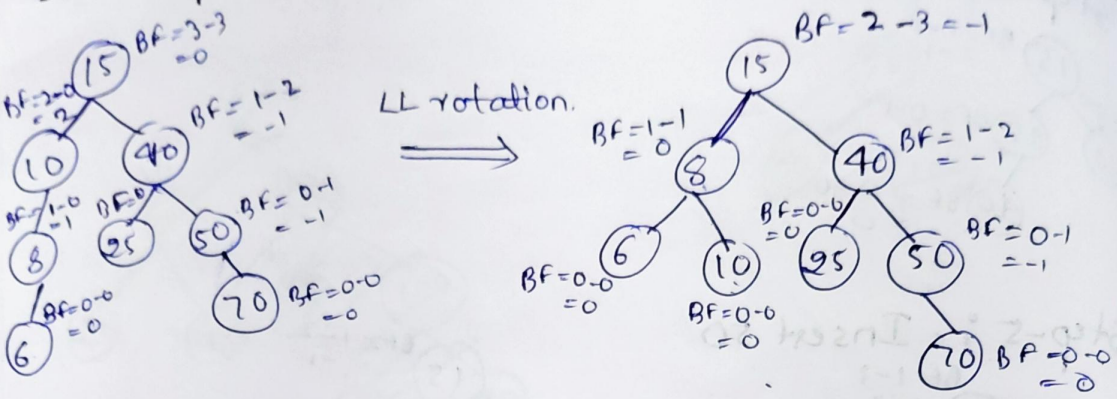

Step-6 :- Insert 8



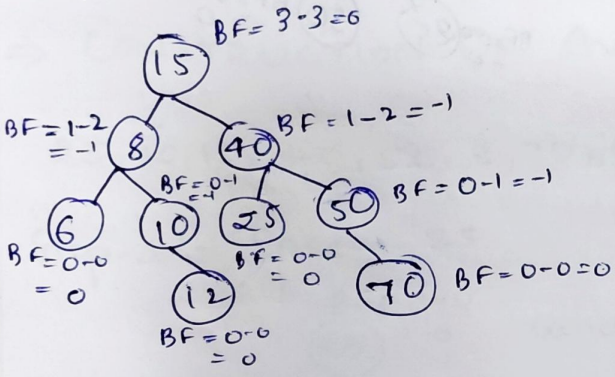

Step-7 :- Insert 70



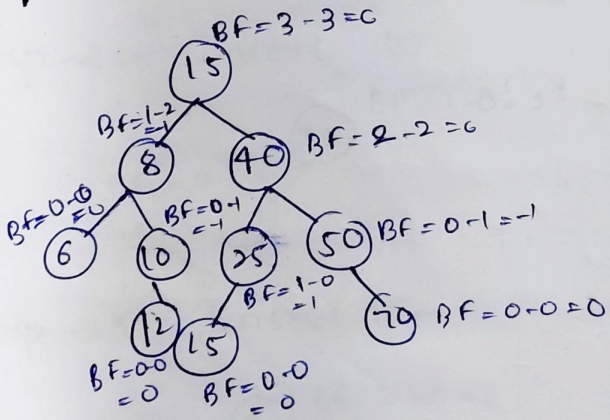
## Step-8:- Insert 6




## Step-9:- Insert 12

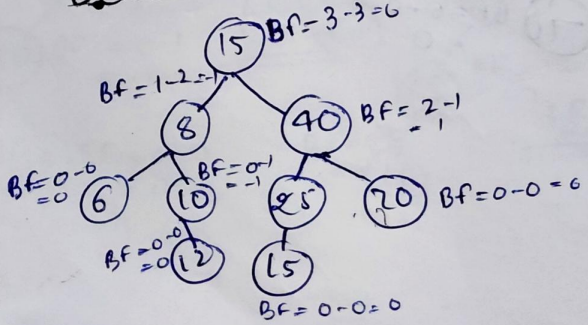


## step-10: Insert 15



Delete 50
Delete 40
Delete 10
Delete 12
Delete 70
Delete 8

Delete 50



## ⇒ B-TREES

* It is a balanced M-way tree. Where M is order
* It is generalisation of BST in which a node can have more 1 key and more than two children.
* It maintains sorted order.
* All the leaf nodes must be at the same level.

⇒ B-tree order M has following properties.

* Every node has maximum M children.
* Min children leaf node → 0
  Root node → 1
  all the other internal nodes have $\lceil m/2 \rceil$ (ceiling)
* Every node has at max M−1 keys.
* Min keys are:
  for root node: 1
  All the other nodes = $\lceil \frac{m}{2} \rceil - 1$

## ⇒ OPERATIONS ON B-TREES:-

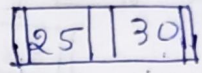
* Operations performed on B-Trees are
  1) Insertion
  2) Deletion
  3) Search

⇒ CONSTRUCTION OF B-TREE

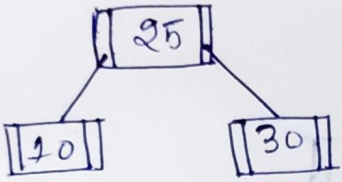
Ex: 25, 30, 10, 40, 5, 70, 60, 80, 15, 35, 28, 72, 45, 82, 101, 112. Order-3

Step-1: Insert 25

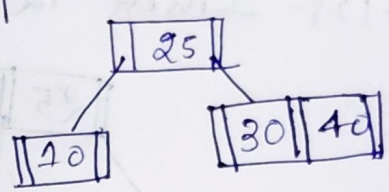

Step-2: Insert 30



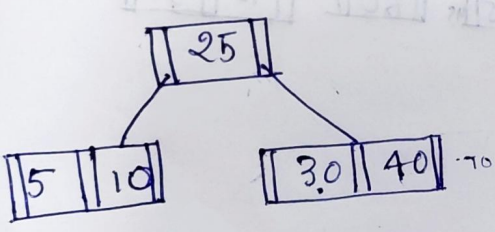

Step-3: Insert 10



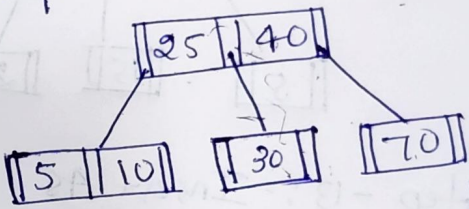

Step-4: Insert 40



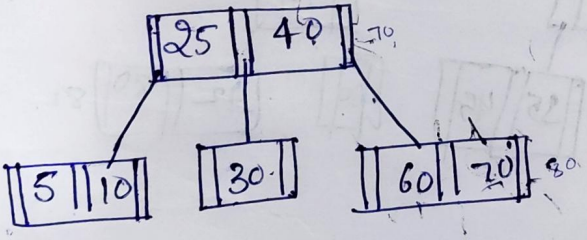

Step-5: Insert 5




Step-6: Insert 70

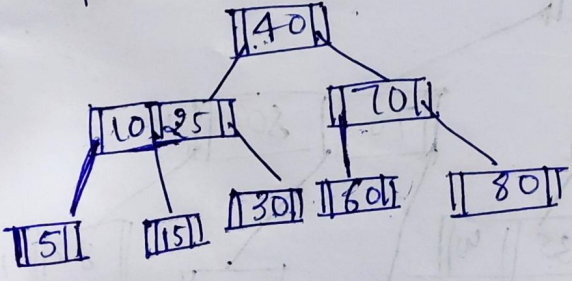



Step-7: Insert 60

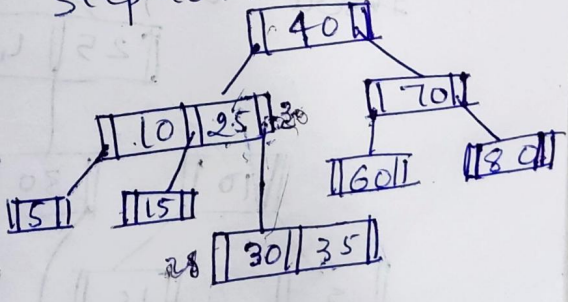



Step-8: Insert 80

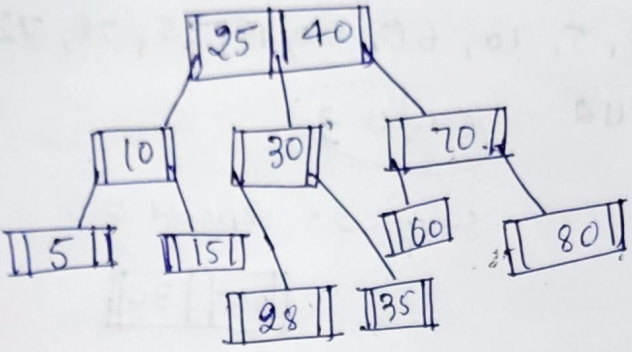

Step-9: Insert 15



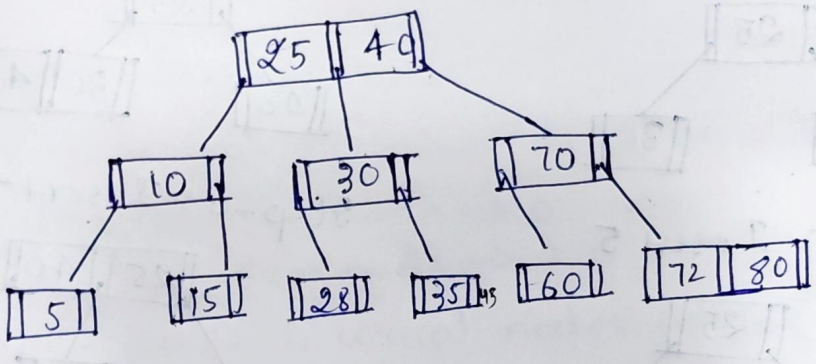

Step-10: Insert 35



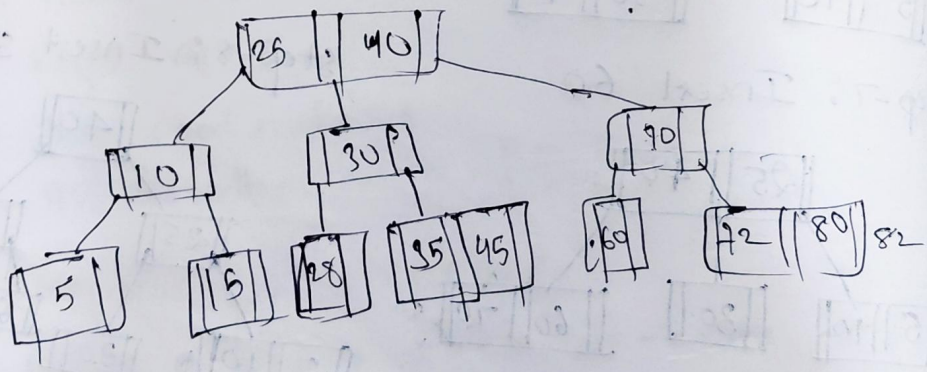
Step-11:- Insert 28



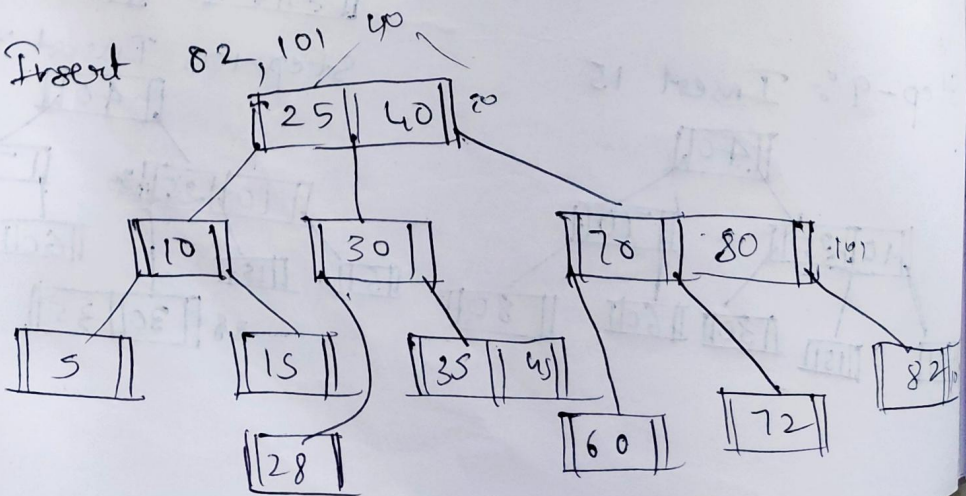

Step-12:- Insert 72




Step-13:- Insert 45



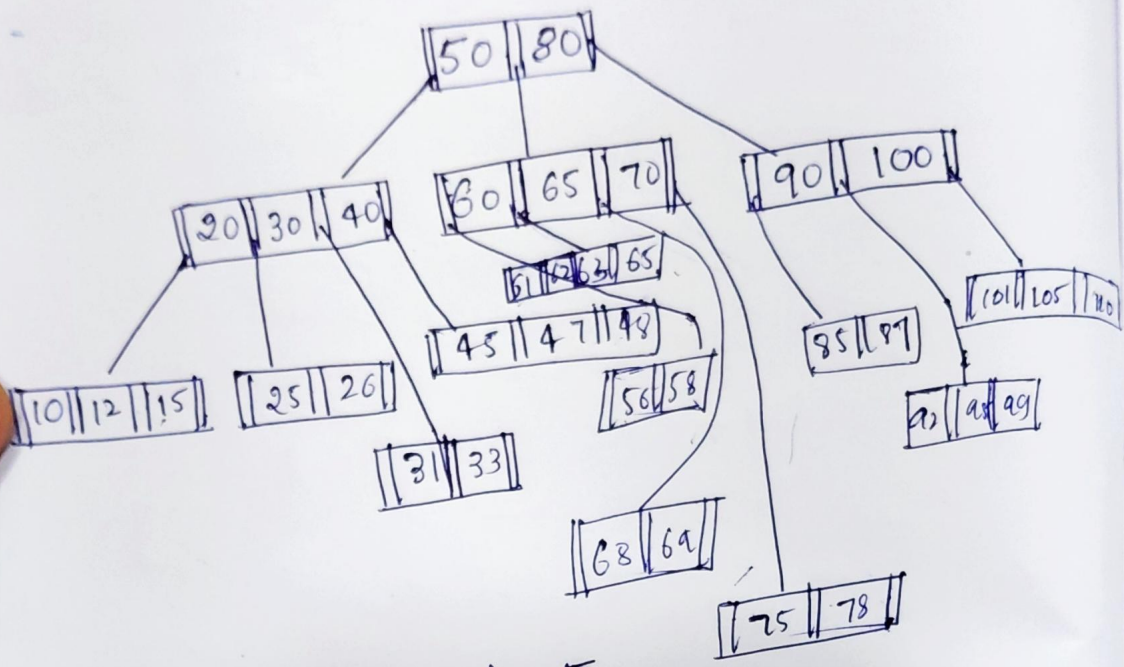

Insert 82, 101



4 - order B-Tree :-

25, 30, 10, 40, 5, 70, 60, 80, 15, 35, 28, 72, 65,
32, 42, 79, 89, 45, 82, 101, 112

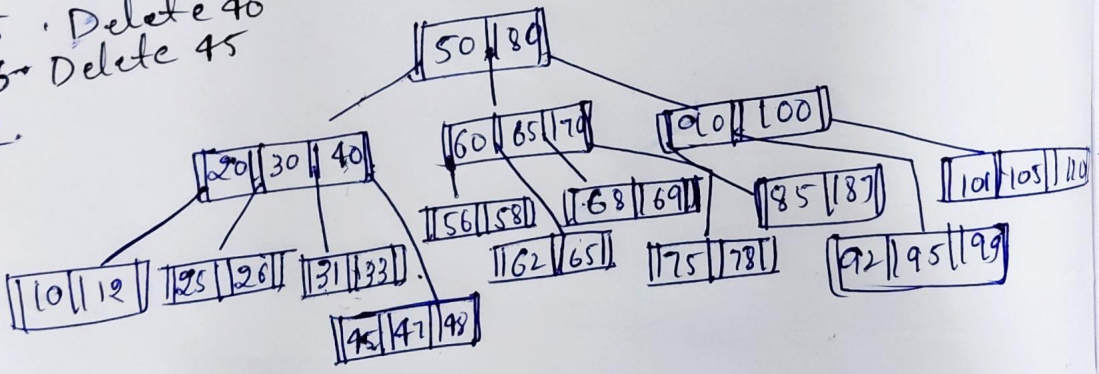
⇒ DELETION :-



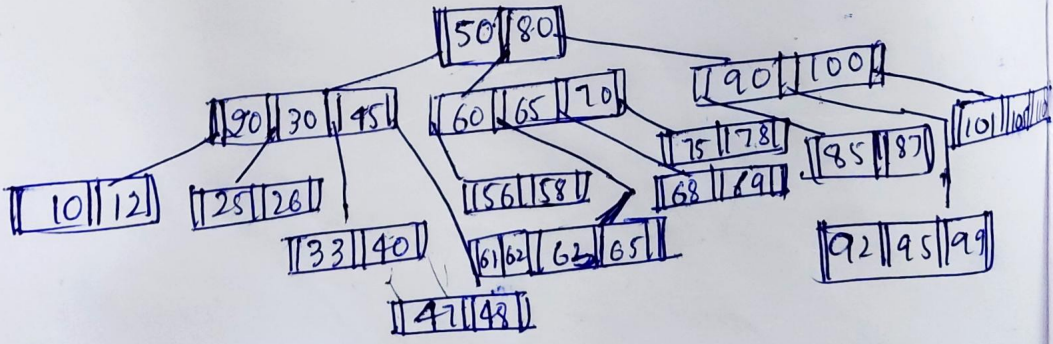

1. Delete 15
2. Delete 31
3. Delete 61
4. Delete 78
5. Delete 40
6. Delete 45

order = 5
$m = 5$
Min children → $\lceil \frac{m}{2} \rceil = 3$
Max children → $5 \rightarrow m$
Min keys → $\lceil \frac{m}{2} \rceil - 1 = 3 - 1 = 2$
Max keys : $m - 1 = 5 - 1 = 4$

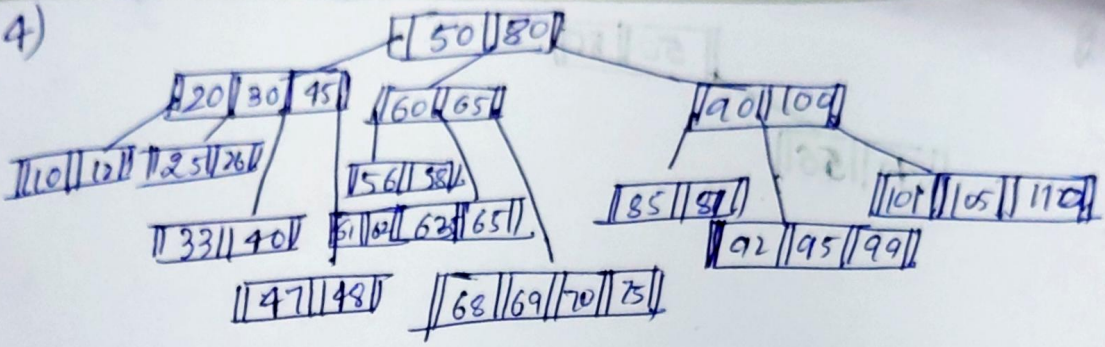
1.




2.



4)


50
80
20
30
45
60
65
90
100
10
12
25
26
56
58
85
87
101
105
110
33
40
60
63
65
92
95
99
47
48
68
69
70
75

# UNIT-IV

# HASHING, SEARCHING AND SORTING

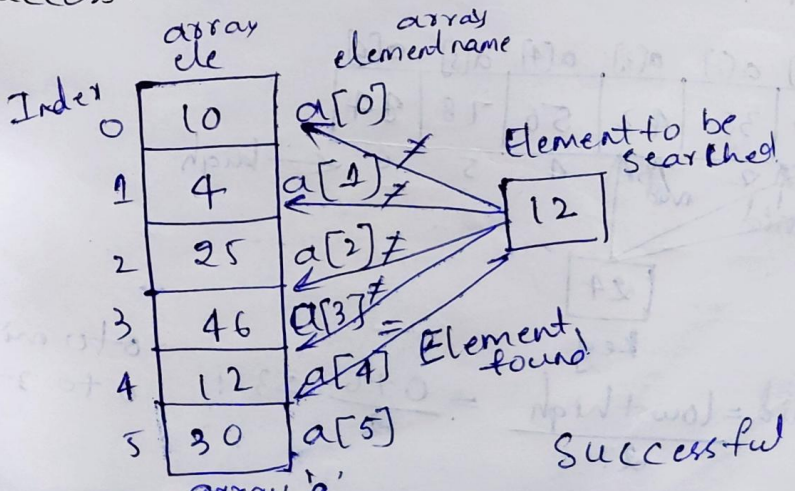
## ⇒ SEARCHING:-

Finding an element whether it is present or not is called searching.

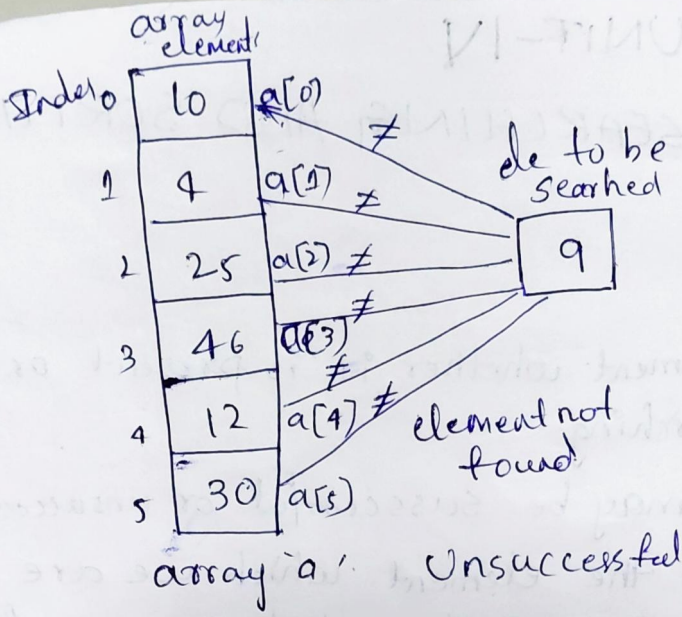
* The search may be susscessful or unsuccessful
* If we find the element which we are searching for then it is called a successful search
* If we don't find the element which we are searching for then it is called a unsuccessful search.
* The types of searching techniques:-
  1. Linear search
  2. Binary search.

* Linear search:- A search which starts from the first element and continue sequentially till we find the element is called linear search

⇒ Successful search:-




Successful search.

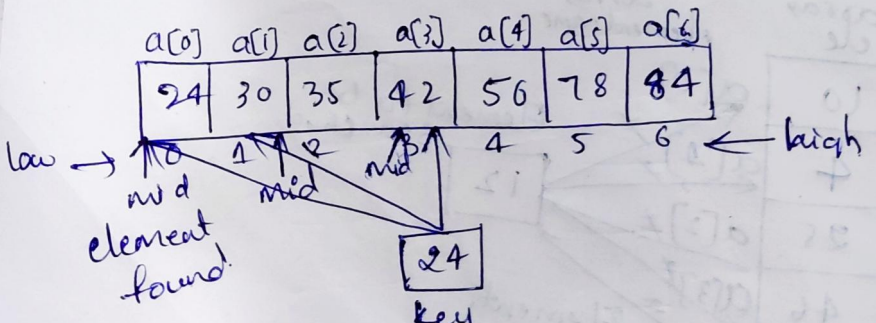



array 'a' Unsuccessful Search

Code:

```
for (i=0; i<n; i++)
  {
  if (a[i]==key)
    {
    printf ("Element found at %d position", i+1);
    break;
    }
  }
if (i==n)
  printf ("Element not found");
```

⇒ BINARY SEARCH:-

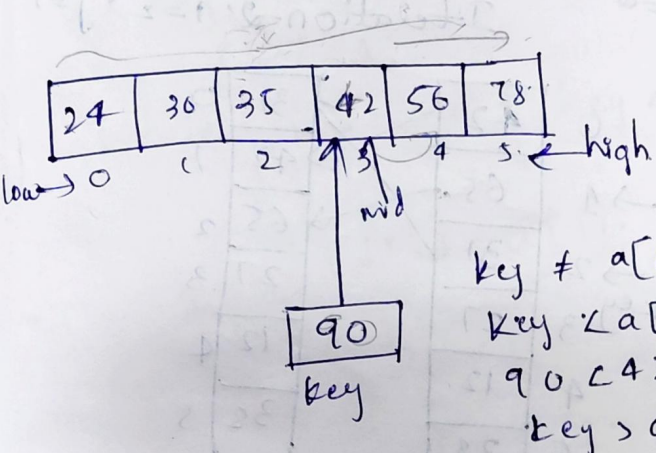



$$Mid = \frac{low + high}{2} = \frac{0+6}{2} = 3$$

0 to mid-1
0 to 3-1 = 2

$mid = \frac{0+2}{2} = 1$

0 to mid-1
0 to 1-1 = 0




key ≠ a[mid]
key < a[mid]
90 < 42 ✗
key > a[mid]
90 > 42

$mid = \frac{low + high}{2}$

$= \frac{0+5}{2}$

$= 3$

$mid = \frac{4+5}{2} = 5$

mid+1 to n-1.
5+1 to 5
6 to 5
element not found.

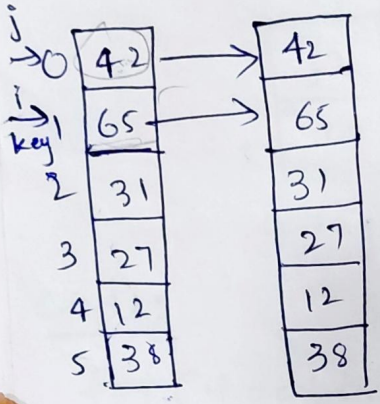
## ⇒ INSERTION SORT :-

Let the given array is

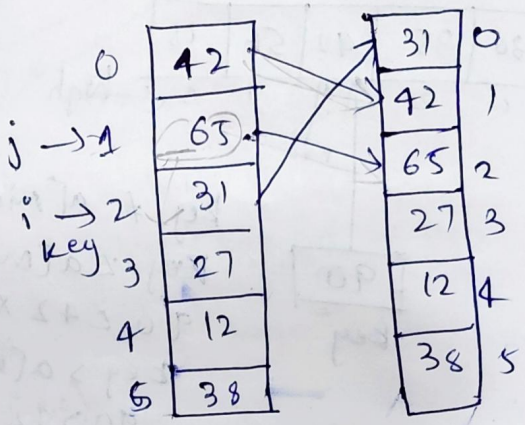
| 42 | 65 | 31 | 27 | 12 | 38 |
|---|---|---|---|---|---|

$n = 6$

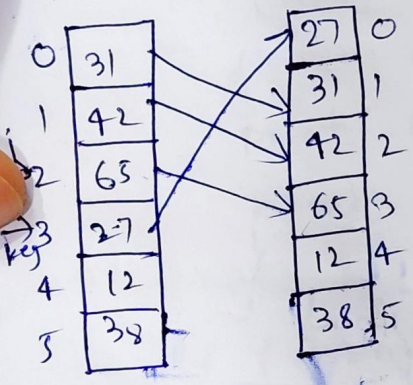
Iteration 1 :- $i=1, j=0$



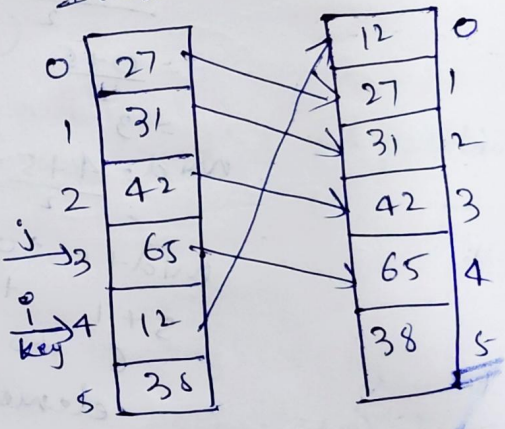

Iteration-2 : $i=2 \rightarrow j=1$



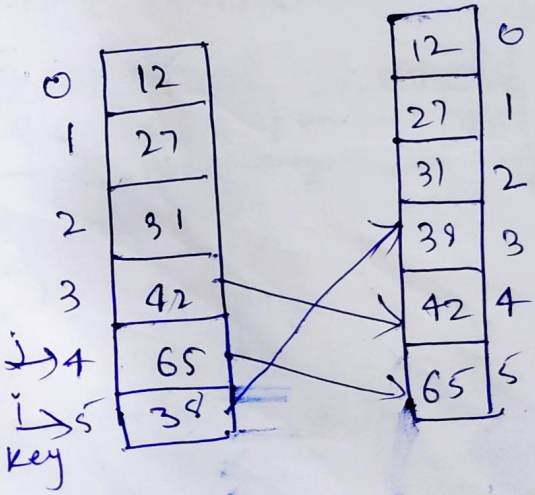

Iteration-3 : $i=3, j=2$




Iteration-4 :- $i=4, j=3$




Iteration-5 :- $i=5$ $j=4$



CODE:-

```
for (i=1; i<n; i++)
{
  key = a[i]
  j = i-1;
  while (j >= 0 && key < a[j])
  {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

a[0+1]

a[1] = 65;

a[j+1] = a[j]

⇒ MERGE SORT:-

i = 1 ; i < 6 ; i++

key = 65;

j = 0

while (0 >= 0 && 65 < 42

i = 2 ; i<6 ; i++

key => 31

j = 1 key < a[j]

while

if

j--

## ⇒ HEAP SORT :-

* To sort the given set of elements using heap sort method we should follow two steps.

step-1 :- Construct the heap

step-2 :- Delete the root till the tree becomes empty

1) Construct the heap.

* A heap tree is a complete binary tree where we insert the elements from left to right.
* A heap tree is of two types :
  1. Min heap tree
  2. Max heap tree.

### 1) Min-heap tree :-

* A min heap tree is a CBT where the parent nodes are smaller than its children.
* A min heap tree is used to sort the elements in ascending order.

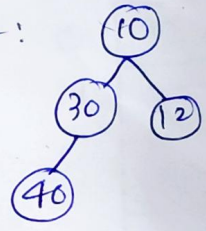
Ex : 12, 30, 10, 40

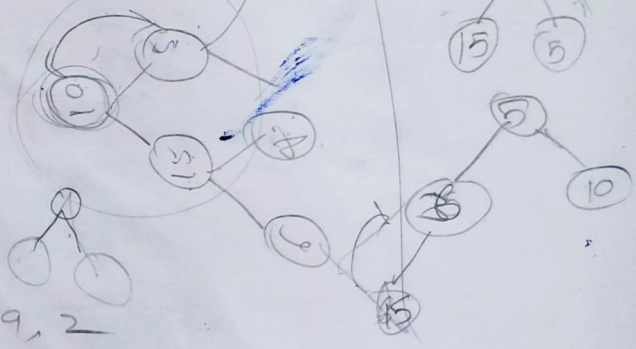
step-1 : (12)

step-2 :- (12) (30)

step-3 :- (12) (30) (10) → (10) (30) (12)

step-4 :



10, 15, 5, 6, 7, 9, 2

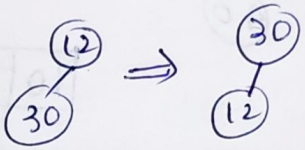
10, 15, 5, 6, 7, 9, 2

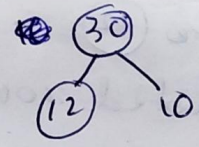
2) Max heap tree:-

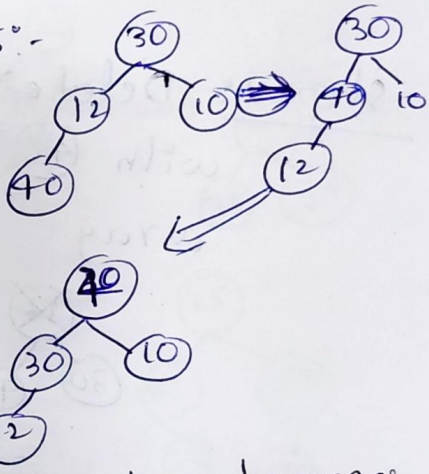
* A max heap tree is a CBT where the parent nodes are greater than its children.
* A max heap tree is used to sort the elements in descending order.

Ex: 12, 30, 10, 40

step-1: 

step-2 

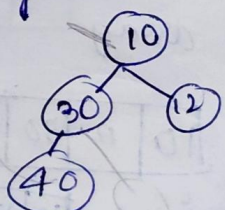
step-4:- 

step-5:- 

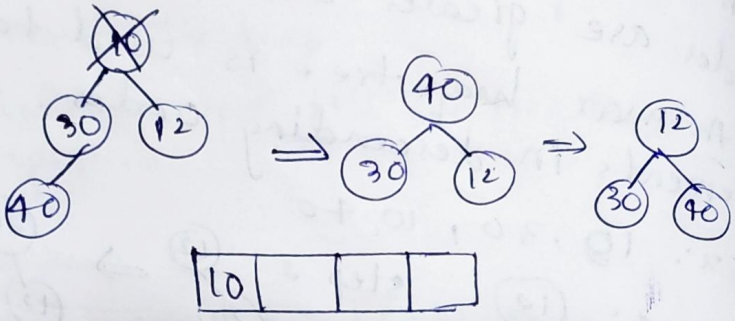
Step-2: Delete the root till the tree becomes empty.

* In this step we delete the root of min/max heap tree and it will be replaced with the right most element at the last level of min/max heap tree. The deleted root will added to the sorted output array which contains the sorted list of elements. This process will continue till the tree becomes empty.

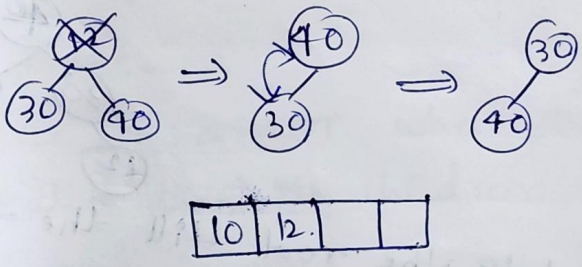
Ex: let the given ~~max~~ min. heap tree is



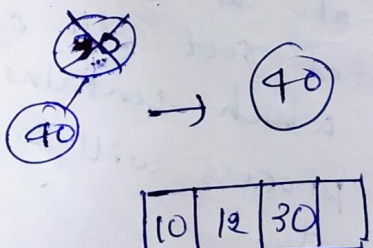
Step-1: Delete the root (10) and add it to o/p array & replace it with last ele (40)



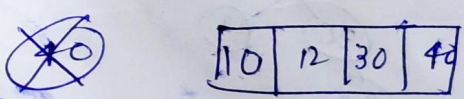

Step-2: Delete the root ~~40~~ (12) replace root with (40) add deleted root to o/p array




Step-3: Delete the root 30, add it to o/p array, replace with 40




Step-4:- Delete the last element and add it to o/p array



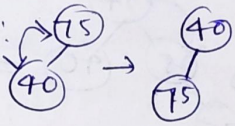

The Sorted o/p array is 10 | 12 | 30 | 40

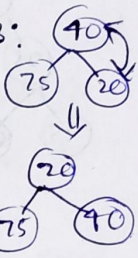
Ex: Sort the below elements using heap sort

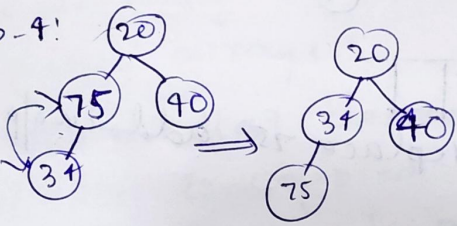
75, 40, 20, 34, 25, 15

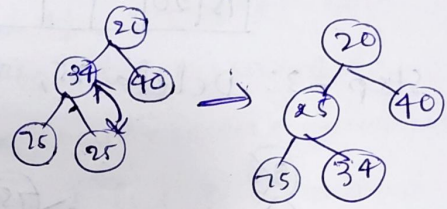
Step-1: construct the min heap tree.

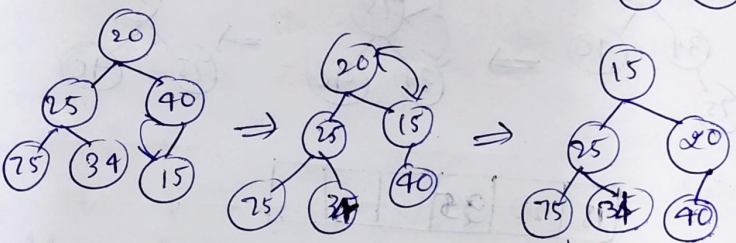
step-1: 

step-2: 

step-3: 

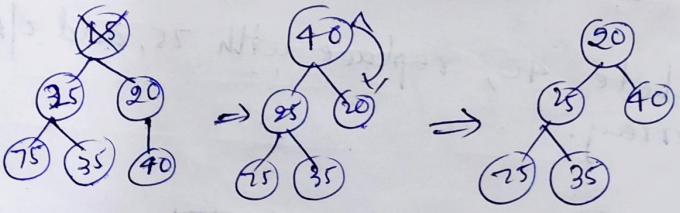
step-4: 

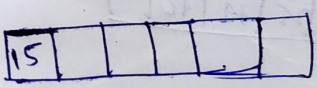
step-5: 

step-6: 

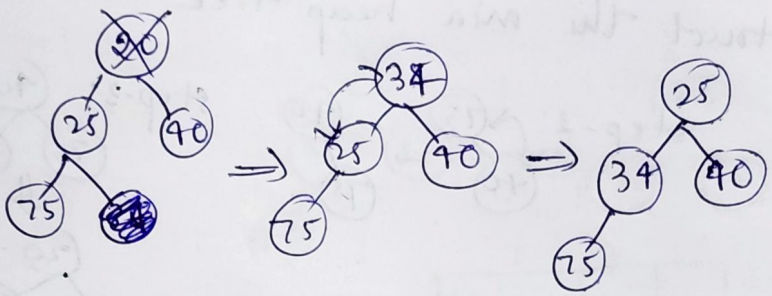
Step-2: Delete the element till the tree is empty.

Step-1: Delete 15, replace with 40, add in o/p array



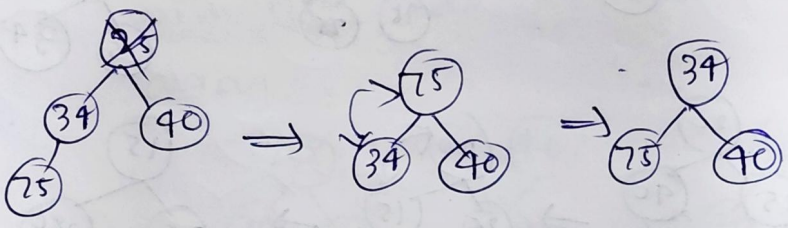


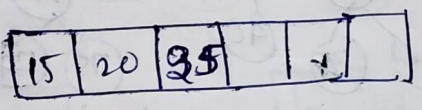
Step-2: Delete 20, replace with (35) add in o/p array



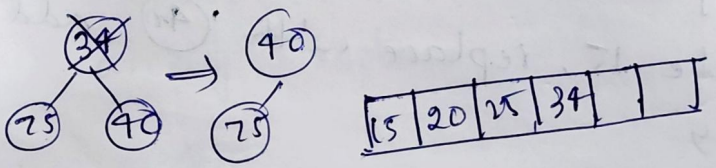


Step-3: Delete 25, replace 75 add in o/p array

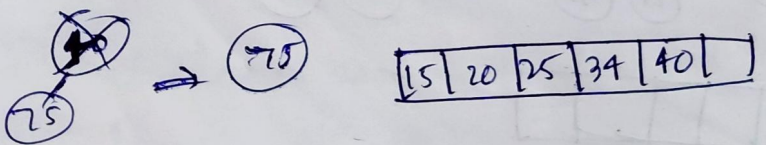



Step-4: Delete 34, replace 40, add o/p array

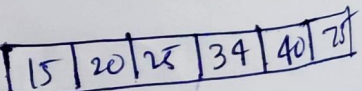



Step-5: Delete 40, replace with 75, add o/p array.




Step-6: Delete 75

The sorted o/p array is



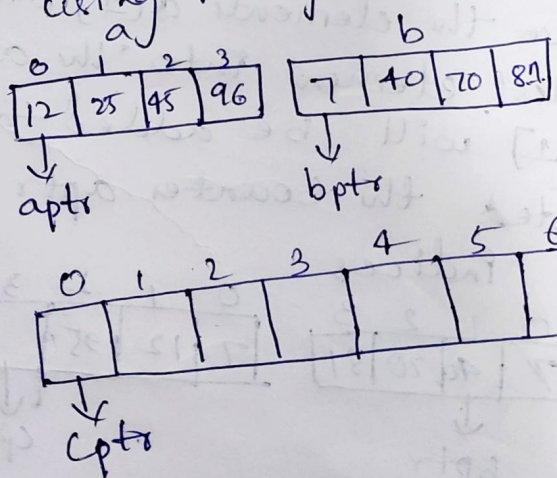
# MERGE SORT :-

The fundamental operation in this algorithm is merging two sorted arrays. The basic merging algorithm takes 2 input arrays 'A' & 'B' and output Array 'C' and, 3 pointers [aptr], [bptr], [cptr] which are initially set to the beginning of their respective arrays.

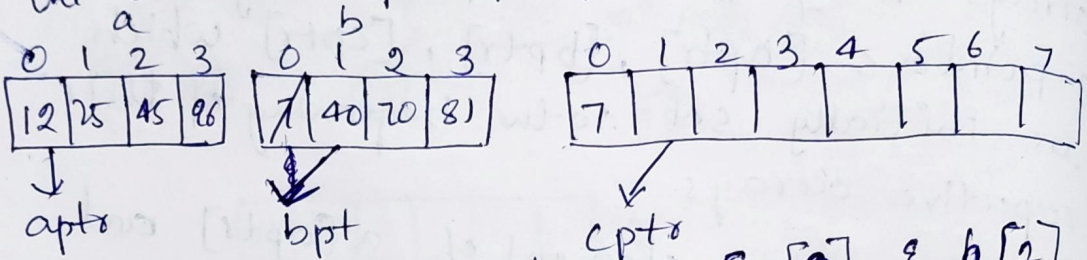
* The smallest element of a[aptr] and b[bptr] is copied to the array C, and the appropriate counters are incremented
* When either of the input list is exhausted. the remainder of other list is copied to the o/p array C

Ex: let the given array 'a' = {12, 25, 45, 96,} b = {7, 40, 70, 81} then merge the two arrays using merge sort

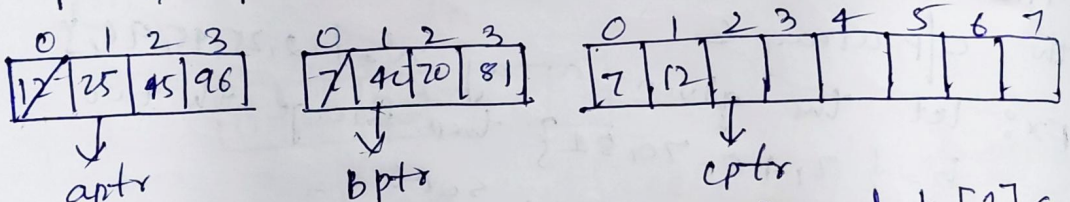
Step 1:



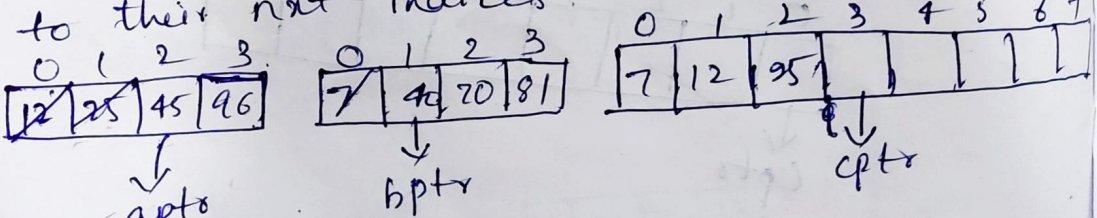
Step-1:- Compare the elements a[0] i.e 12 and b[0] i.e 7 and copy the smallest element into the array c. in this case b[0] i.e 7 will be added to the array c and increment the counter bptr & cptr. to the next indices



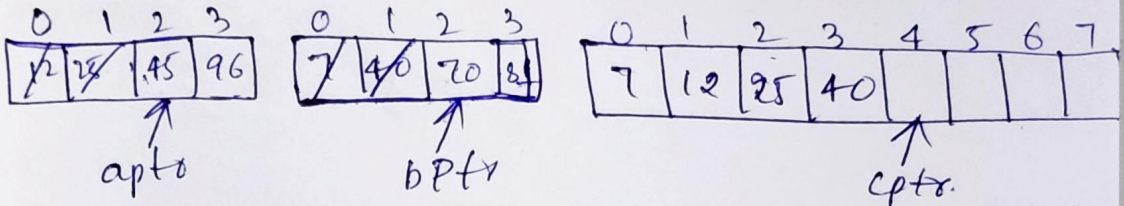

Step-2:- Compare the elements a[0] & b[1] and copy the smallest element into the array c. in this case a[0] i.e 12 will be added to the array c and increment the counter aptr & cptr to their next indices.



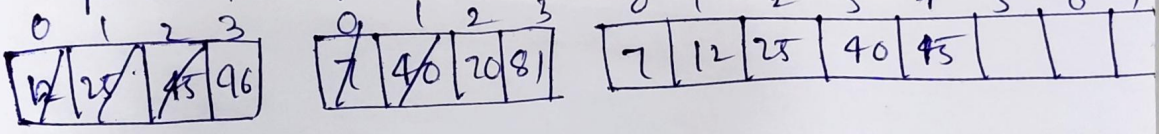

Step-3:- Compare the elements a[1] and b[1] & copy the smallest element into the array c. in this case a[1] will be added to array c and incremented the counter aptr & cptr to their next indices.



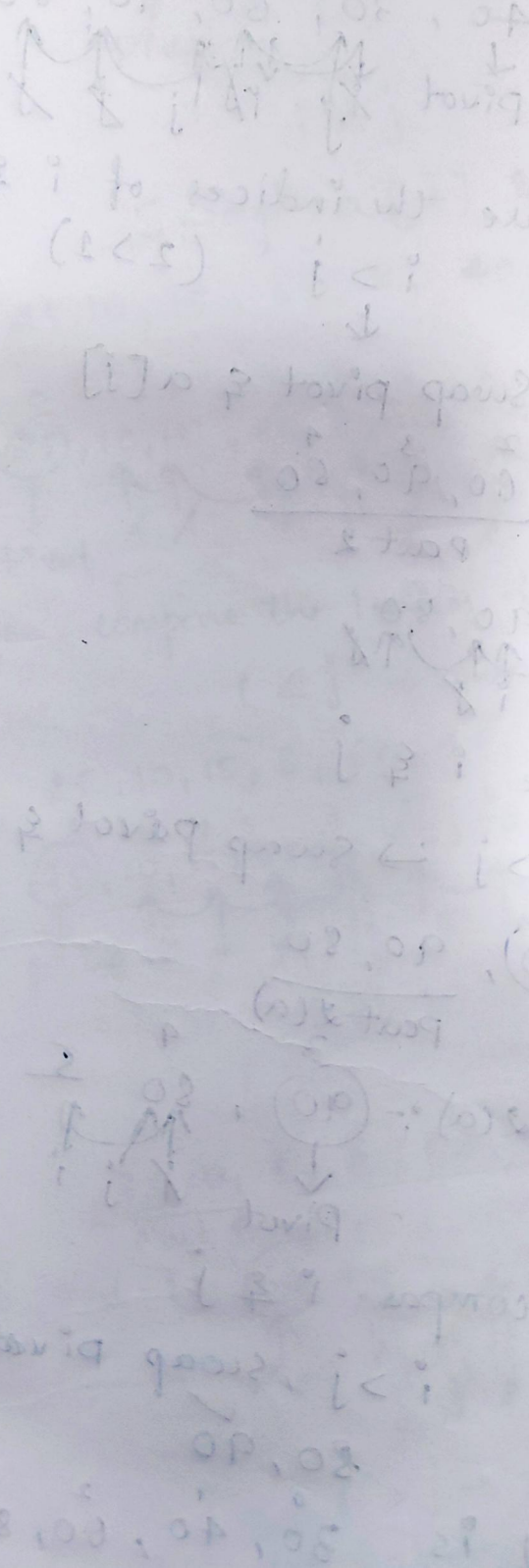
Step-4: Compare the elements a[1] & b[1] & copy the smallest element into the array c. in this case b[1] will be added to array c. and incremented the counter bptr & cptr. to their next indices.




Step-5:- Compare the elements a[2] & [2] & copy the smallest element into the array c. In this case a[2] will be added to array c. and incremented the counter aptr & cptr to their next indices.



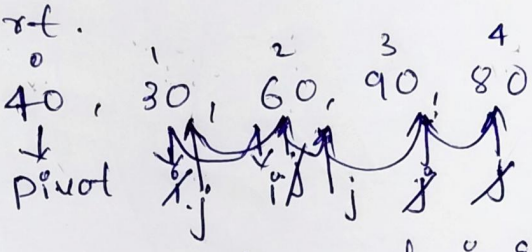
8) A = { 5, 10, 30, 40, 50 }  B = { 2, 8, 50, 60, 90 }



## ⇒ QUICK SORT:-

* Ex: Let an array contains the elements 40, 30, 60, 90, 80 sort the above elements using Quick sort.

Sol: Step-1:-

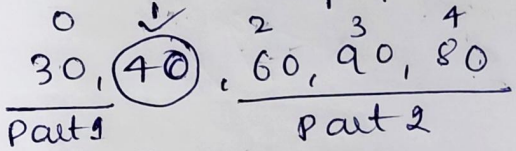



compare the indices of i & j

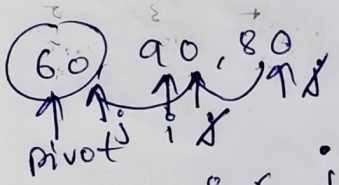
$i > j \quad (2 > 1)$

↓

Swap pivot & a[j]

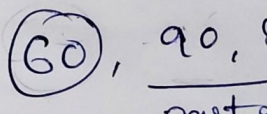



Part-2:-
Step-2:-




compare i & j

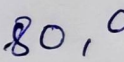
$i > j \Rightarrow$ swap pivot & a[j]



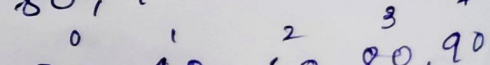

Step-3:- part 2(a):-


compar i & j

$i > j$, swap pivot & a[j]

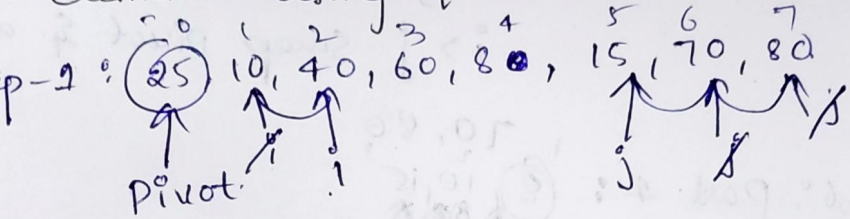



The sorted array is



Ex:- Let the array contains the elements ~~40, 20,~~ 25, 10, 40, 60, 8, 15, 70, 80 sort the above elements using quick sort.

Sol: step-1 :

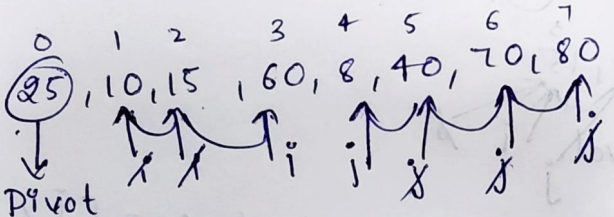



$i < j$

Swap $a[i]$ & $a[j]$

25, 10, 15, 60, 8, 40, 70, 80.

Step-2:

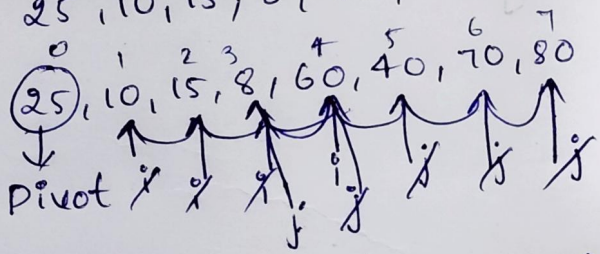



compare the indices of i & j

$i < j$ → swap $a[i]$ & $a[j]$

25, 10, 15, 8, 60, 40, 70, 80

Step-3:

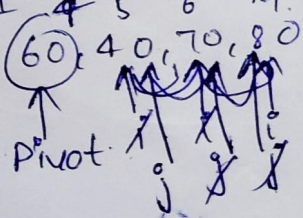



$i > j$ → swap pivot & $a[j]$

8, 10, 15, (25), 60, 40, 70, 80

part 1 part 2.

Step-4:- part-2:-

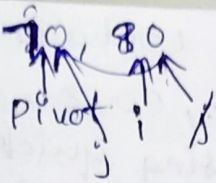



$i > j$ → swap pivot & $a[j]$

40, (60), 70, 80

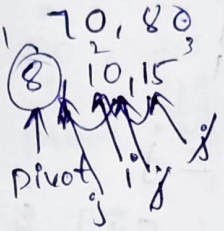
part 2(a)

Step-5: Part 2(a):




i > j swap pivot & a[j]

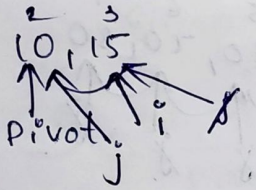
Step-6: Part 1:




i > j swap pivot & a[j]

8, 10, 15

Step-7:




i > j swap pivot & a[j]

10, 15

The sorted array is 8, 10, 15, 25, 40, 60, 70, 80

# MODULE-V

## GRAPHS

⇒ GRAPH:- Graph is a set of vertices and edges

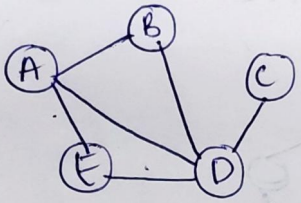
Let G is a graph then it can be written as

$G = \{V, E\}$

where, V = Set of vertices

E = Set of Edges.

Ex:-




In the above graph, $V = \{A, B, C, D, E\}$

$E = \{AB, AD, AE, ED, BD, DC\}$

⇒ EDGE:- A straight line which connects two vertices is called an edge.

In the abov graph A-B is an edge where A is called source/origin & B is destination.

* There are 2-types of edges

1. Directed
2. Undirected

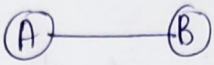
* Directed edge:- A edge which has a direction or arrow mark is called directed edge.

Ex: 


In the above graph the edge can be written as AB but not BA

* Undirected edge:- A edge which no direction or arrow mark is called Undirected edge.

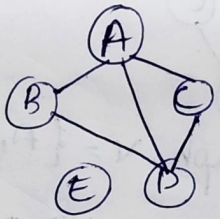
Ex:




In the above graph the edge can be written either AB or BA.

⇒ ISOLATED VERTEX:- A vertex which is not connected to any of the vertex ie. the degree of the vertex is '0'.

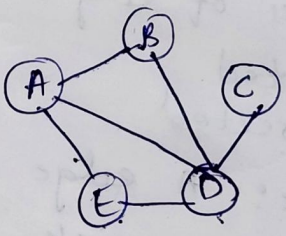
Ex:-




In the abv graph the vertex E is not connected any other vertex.

⇒ PATH:- A route from one vertex to another vertex is called path.
A path may contain more than 1 edge

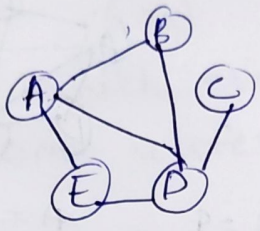
Ex:




In the abv graph the paths from A to C are:

A - B - D - C
A - D - C
A - E - D - C

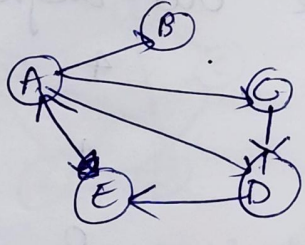
⇒ ADJACENT VERTICES:- The vertices which are connected to the other vertices are called adjacent vertices.




The adjacent vertices of A = {B, D, E}
B = {A, D}
C = {D}
D = {A, E, B}
E = {A, D}

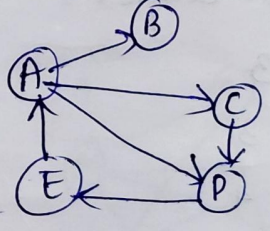
⇒ IN-DEGREE:- The no. of edges reaching that vertex is called indegree of vertex.

Ex:




In degree of A = 1
B = 1
C = 1
D = 2
E = 1

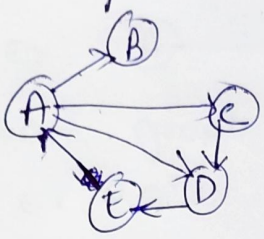
⇒ OUT DEGREE:- The no. of edges leaving that vertex is called out degree



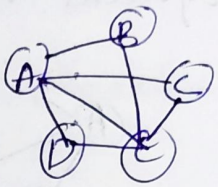

out degree of A = 3
B = 0
C = 1
D = 1
E = 1

⇒ DEGREE OF A VERTEX: The sum of In degree and Out degree it's called degree of a vertex.

Ex:

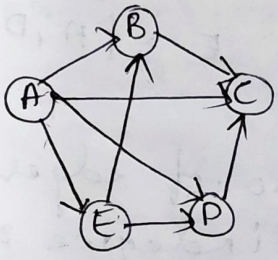



degree of A = 1+3 = 4
B = 1+0 = 1
C = 1+2 = 3
D = 1+1 = 2
E = 1+1 = 2



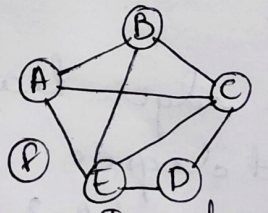

A = 4+4 = 8
B = 2+2 = 4
C = 2+2 = 4
D = 2+2 = 4
E = 3+3 = 6

Ex:




| Vertex | In-degree | Out-degree | degree |
|---|---|---|---|
| A | 0 | 4 | 4 |
| B | 2 | 1 | 3 |
| C | 3 | 0 | 3 |
| D | 2 | 1 | 3 |
| E | 1 | 2 | 3 |

Ex:

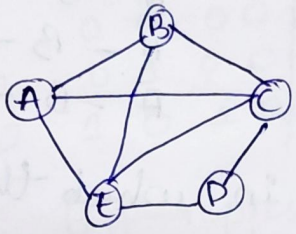



| Vertex | In-degree | out-degree | degree |
|---|---|---|---|
| A | 3 | 3 | 6 |
| B | 3 | 3 | 6 |
| C | 4 | 4 | 8 |
| D | 2 | 2 | 4 |
| E | 4 | 4 | 8 |
| F | 0 | 0 | 0 |

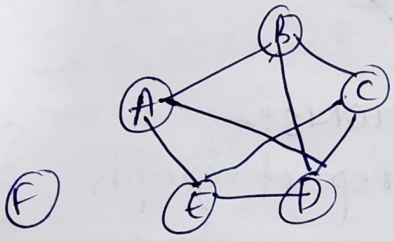
⇒ CONNECTED GRAPH:- A graph in which there exists a path from any vertex to any other vertex. is called connected graph

(or)

A graph in which every vertex is connected to atleast one vertex is called connected graph.



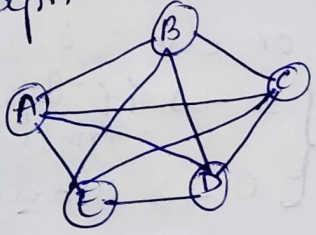

⇒ DISCONNECTED GRAPH:- A graph in which there is a isolated vertex (vertex which is not connected to any vertex) is called disconnected graph.




⇒ COMPLETE GRAPH:- A graph in which every vertex is connected to every other vertex in the graph.

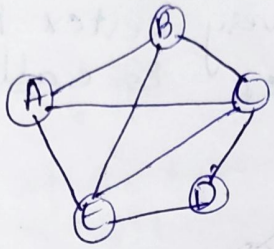
(or)

A graph in which there exist an edge from every vertex to every other vertex in the graph.



⇒ CYCLE:- A path in which the starting vertex (source) and ending vertex (destination) are same then it is called cycle.

Ex:-

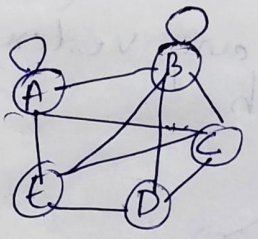



In the given graph some the cycles are

A - B - C - D - E - A
A - B - E - A
A - B - C - E - A.

⇒ LOOP:- A edge in which the starting vertex (source) and ending vertex (destination) are same then it is called loop.

Ex:




⇒ GRAPH REPRESENTATIONS:-

There are 3 types of rep of graph

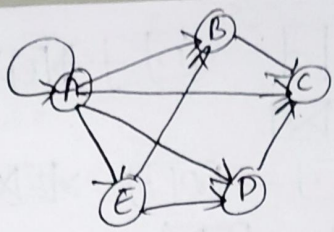
1) Adjacency Matrix rep
2) Adjacency List rep
3) Linked rep.

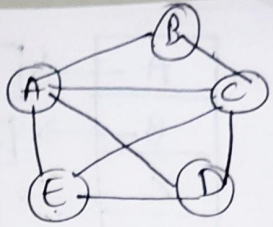
⇒ Adjacency Matrix representation:-

In this representation, if there exists an edge from vertex i to vertex j then we write it as 1 or else 0.

$$V_{ij} = \begin{cases} 1 \rightarrow \text{if there exists an edge frm } i \rightarrow j \\ 0 \rightarrow \text{otherwise} \end{cases}$$

Ex:




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 0 | 1 | 0 | 0 |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 0 | 0 |
| E | 0 | 1 | 0 | 1 | 0 |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 0 | 1 | 1 |
| D | 1 | 0 | 1 | 0 | 1 |
| E | 1 | 0 | 1 | 1 | 0 |

⇒ Adjacency list representation:-

* In this representation we list out all the adjacent vertices of each vertex.

for the abv graph the adjacency list rep is:

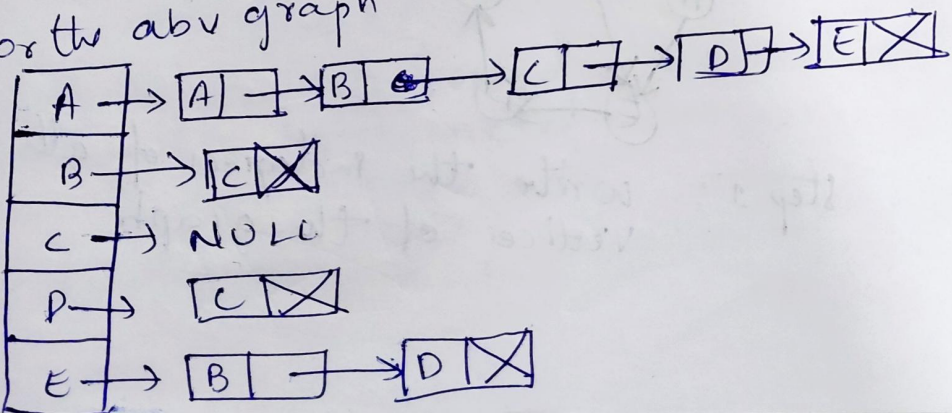
A → A, B, C, D, E
B → C
C → NULL
D → C
E → B, D

A → B, C, D, E
B → A, C
C → A, B, D, E
D → A, C, E
E → A, C, D

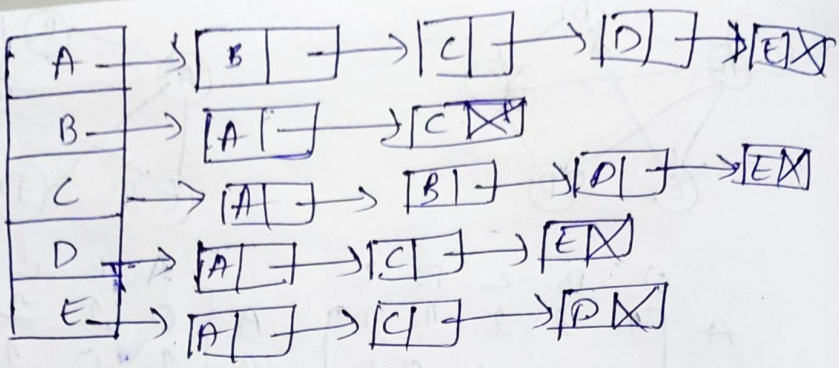
⇒ Linked representation:-

In this rep every vertex is rep as a node and all the adjacent vertices of each vertex are linked together.

Ex: for the abv graph linked rep is:-



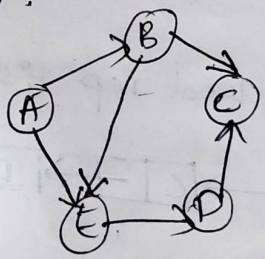
Ex:




⇒ TOPOLOGICAL SORT :-

* It is ordering of vertices.
* when there is an edge uv, we must write u first followed by v.
* This sorting can be applied only for Directed acyclic graphs.
* we can have more than one topological order for one graph.
* In this algorithm first we select a vertex which has no incoming edges i.e a vertex with in-degree '0'.
* Write that vertex in sorted order and delete it along with its connected edges.
* This process is repeated till the graph is empty.

Ex:




Step-1: write the indegrees of all the vertices of the graph.

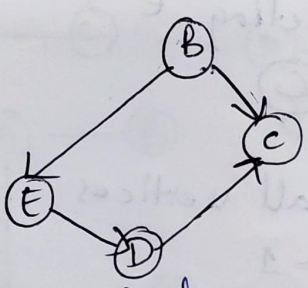
In-deg (A) = 0
In-deg (B) = 1
In-deg (C) = 2
In-deg (D) = 1
In-deg (E) = 2

* Selected a vertex whose in-degree is '0' (i.e A)
* write it in the sorted order and delete it along with its connected edges

sorted order : A

* After deleting the vertex A the graph bems

step-2 :-




write the indegrees of all vertices

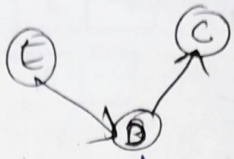
Indeg (B) = 0
Indeg (C) = 2
Indeg (D) = 1
In deg (E) = 1

* Select the vester whose in-deg is '0' (i.e B)
* write it in the sorted order and delete it along with its connected edges

sorted order : AB

* After deleting the vester B the graph becomes.

Step 3:-

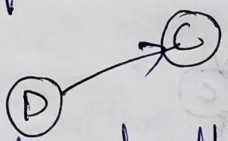



* Write the indeg of all vertices

Indeg(C) = 1
Indeg(D) = 1
Indeg(E) = 0

* Select the vertex whose ~~sorted~~ indeg is '0'
(i.e. E)
* Write it in the sorted order and delete it with its connected edges
sorted order: A B E
* The graph after deleting E

Step-4:




* Write the indeg of all vertices
Indeg(C) = 1
Indeg(D) = 0
* Select the vertex whose indeg is '0' (i.e D)
* write it in the sorted order and delete it along with is connecting edges.
sorted order: A B E D.
* The graph after deleting:

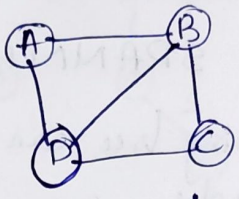
Step-5:- (C)

* Write the indeg : indeg(C) = 0
* Select the vertex whose indeg is '0' (i.e C)
* Write it in the sorted order and delete it along with its connecting edges.
Sorted order: A B E D C
* The graph is empty.

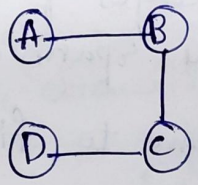
⇒ SPANNING TREE:-

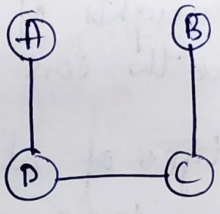
* It is a sub graph of the given graph which has:
  1) No loops / cycles
  2) n-1 edges (n → no. of vertices)

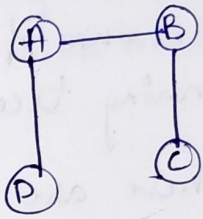
Ex: Let the given graph is

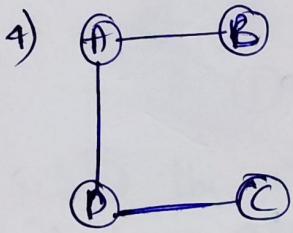


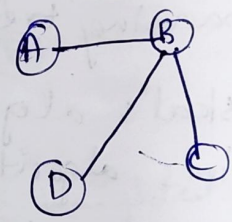

then the spanning trees for the graph are

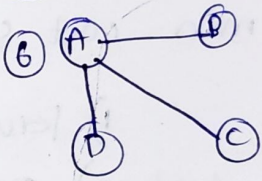
1) 

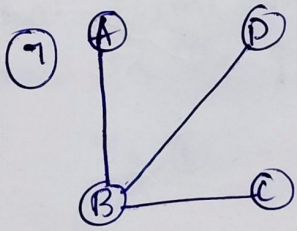

2) 


3) 


4) 

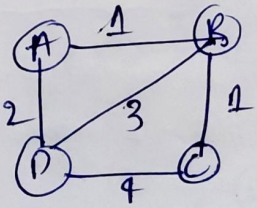

5) 


6) 


7) 


⇒ WEIGHTED GRAPH:-

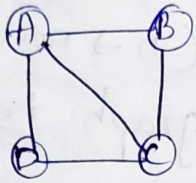
A graph in which all the edges have an integer value called weight on it, is called weighted graph.



⇒ UNWEIGHTED GRAPH:-

A graph which has no weights on the edges is called unweighted graph.

Ex:



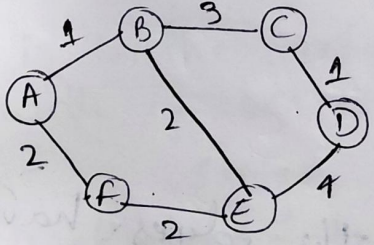

⇒ MINIMUM COST SPANNING TREE:-

* The cost of the spanning tree can be calculated only for weighted graphs.
* The sum of the weights of the edges of the spanning tree gives the cost of the spanning tree.
* There are two types of algorithms to find the min cost spanning tree.

1) Kruskal's algorithm
2) prim's algorithm.

⇒ KRUSKALS ALGORITHM:-

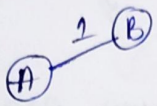
Ex:-



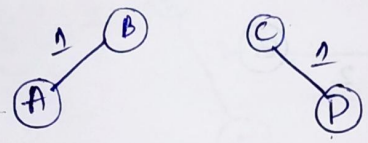

Step-1:- Write the weights of the edges in A.O

AB = 1
CD = 1
AF = 2
FE = 2
BE = 2
BC = 3
ED = 4

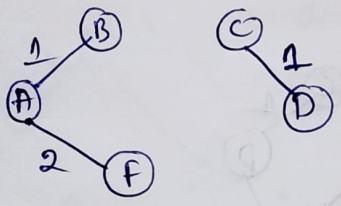
Step-2:- Select the least weight edge AB=1 & draw in the graph.



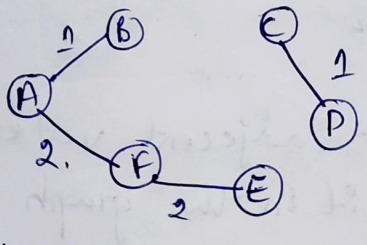

Step-3:- Select the nxt least weight edge which should not form any loop / cycle. i.e CD=1 & draw in the graph



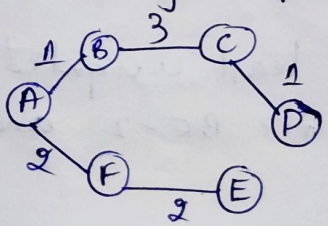

Step-4:- Select the nxt least weighted edge which shld not form any loop / cycle i.e AF=2 & draw in the graph.




Step-5:- Select the nxt least weighted edge which shld not form any loop / cycle i.e AE=2 & draw in the graph.

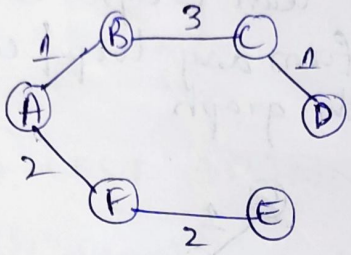



Step-6:- Select the nxt least weighted edge which shld not form any loop / cycle i.e BC=3



In the above graph we have n-1 edges and by adding any one of the remaining edges it forms cycle.

So, the spanning tree for the given graph is:

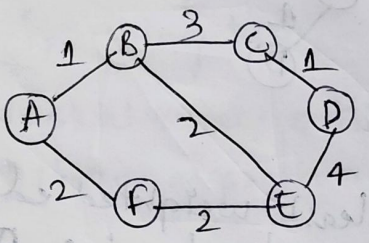



~~The cost of~~

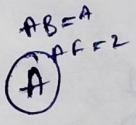
The minimum cost of the spanning tree ⇒

$1+1+2+2+3 = 9$

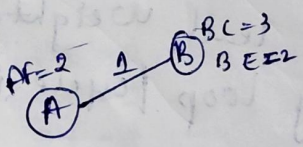
⇒ PRIM'S ALGORITH:-



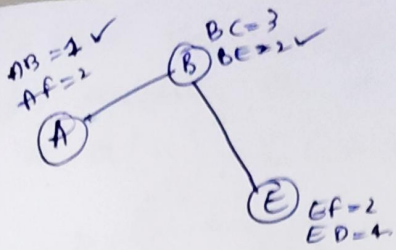

Step-1: Select any one vertex (A) and draw it in the graph.



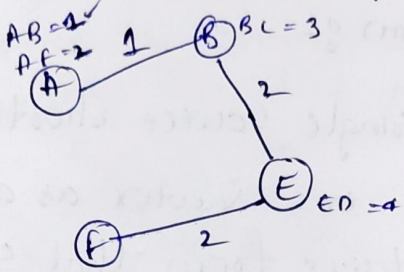

Step-2: Select any least adjecent vertex of A i.e AB=1 & draw it in the graph



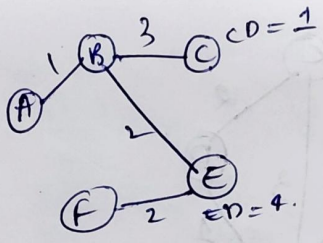

Step-3: Select any least weighted adjacent vertex of A or B i.e BE=2 and draw it in the graph



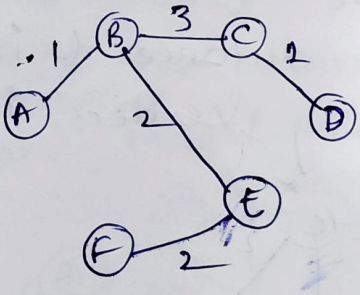

step-4 :- Select any least weighted adjacent vertex of A, B, or E i.e EF=2 and write it in the graph




step-5 : Select any least weighted adj vertex of A, B, E, or F ie BC=3 & draw it in the graph

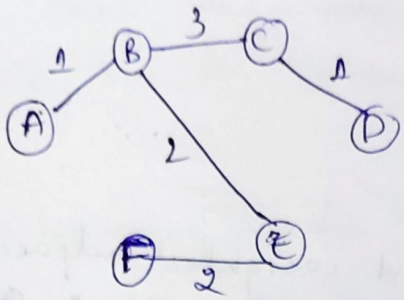



step-6 :- Select any least weighted adj vertex of A, B, E, F, or C i.e CD=1 & draw it in the graph




In the above graph we have n-1 edges and by adding any one of the remaining edges it forms cycle.

So, the spanning tree for the given graph is:

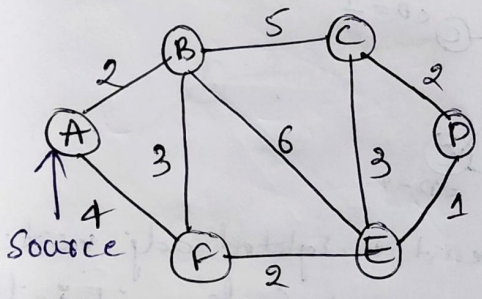



The minimum cost of spanning tree is ⇒ 1+1+2+2+3 = 9

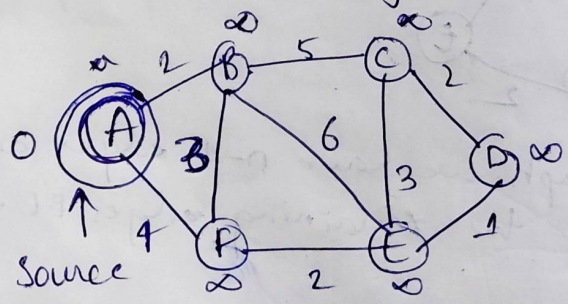
⇒ Dijkstra's Algorithm :-

* It is used to find single source shortest path.
* In this we assume one vertex as a source and will find the distance from that source to all the remaining vertices in the graph.

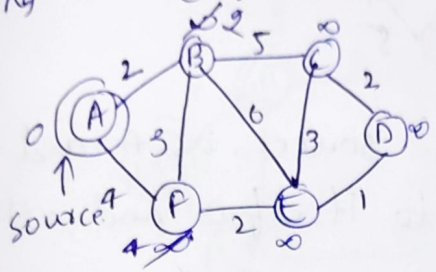
Ex:-




Step-1: Let us take the vertex A as the source and initialize the distance to itself is 0 and all the remaining vertices is ∞.



step-2: Find the connected vertices of the vertex A and modify the distance to those vertices using ★ $d(u) + c(u,v) < d(v)$ → relaxation

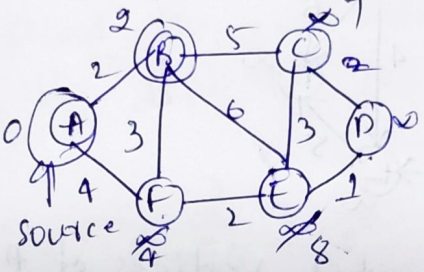



$d(u) + c(u,v) < d(v)$

$0 + 2 < \infty$

$2 < \infty$

step-3: Find the connected vertices of the vertex B and modify the dist to those vertices

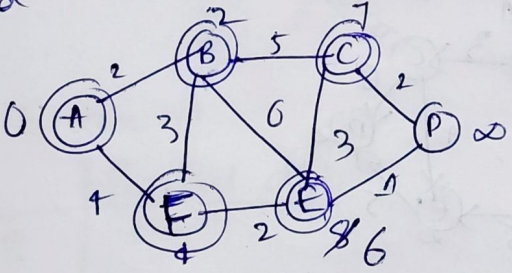



$d(u) + c(u,v) < d(v)$

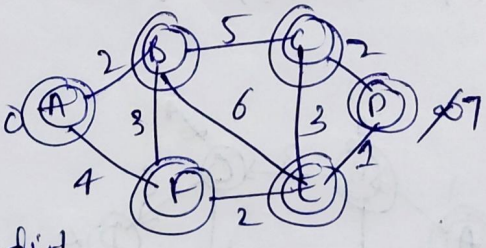
$2 + 5 < \infty$

$7 < \infty$

step-4: Find the connected vertices of the vertex F and modify the dist to those vertices



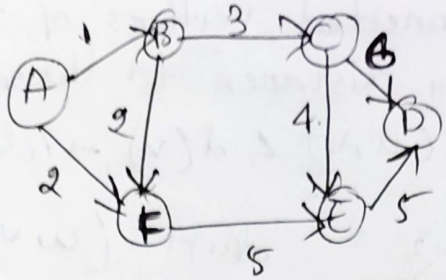

Step-5: Find the connected vertices of the vertex E and modify the dist to those vertices



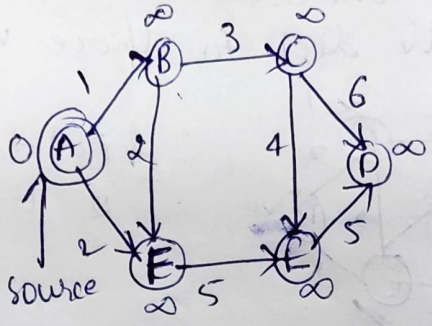

source → 1

| | Shortest dist |
|---|---|
| B | 2 |
| C | 7 |
| D | 7 |
| E | 6 |
| F | 4 |

Eg-2:

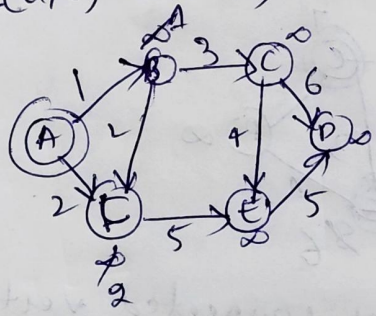



Step-1: Let us take the source as A and initialize the distance to itself 0 and all other vertices ∞.

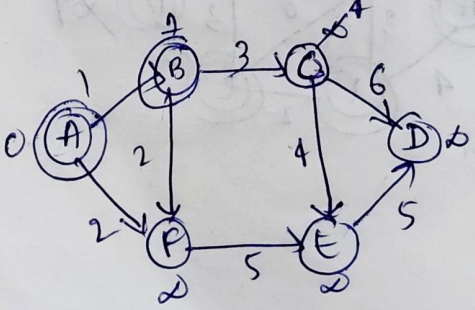



Step-2:- Find the connected vertices of A and modify the dist to those vertices during

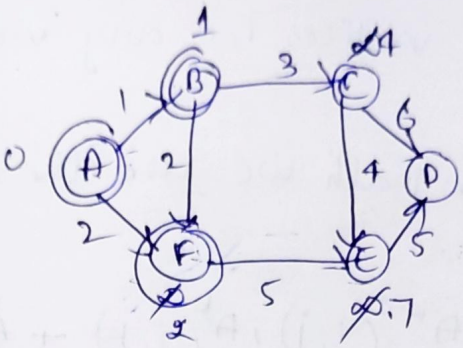
$d(u) + c(u,v) < d(v)$



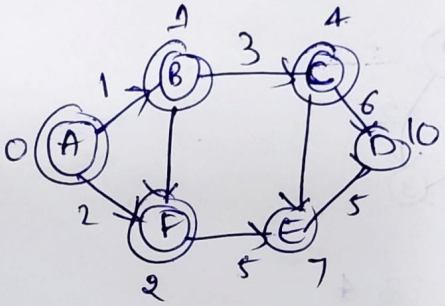

Step-3:- Find the connected vertices of B and modify the dist to those vertices.



step-4:- find the connected vertices of F and modify the dist to those vertices.




step-5:- find the connected vertices of E and modify the dist to those vertices.




Soure - A

| | Shortest dist. |
|---|---|
| B | 1 |
| C | 4 |
| D | 10 |
| E | 7 |
| F | 2 |

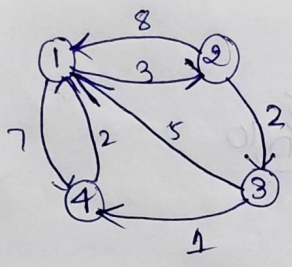
## ⇒ ALL PAIRS SHORTEST PATH

* This algorithm can be used to find the shortest path for any pair of vertices. i.e any vertex to any other vertex.

* To find the minimum path we use the following formula:

$$A^k(i,j) = \min\left(A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\right)$$

Ex:- Find all the pairs shortest path for the following path



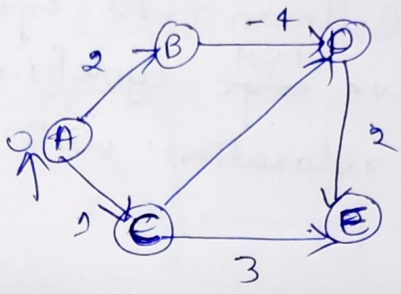

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$A^1$ =

## ⇒ NEGATIVE WEIGHTED EDGE COST:-

* A graph which has -ve weights for the edges is called -ve weighted edge cost.
* To find the shortest path for the -ve weighted edge cost we use the algorithm called Bellman ford algorithm.

Ex:-




In the abv algorithm we have -ve weights to find the S.P from one source to vertex

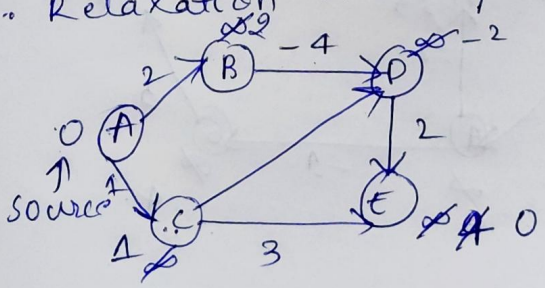
* In this algorithm ~~first~~ we list out the edges.

(A,B) (A,C) (B,D) (C,D) (C,E) (D,E)

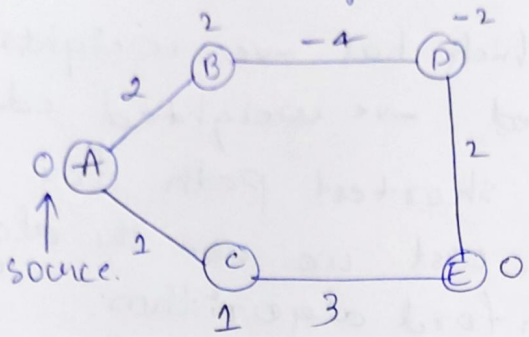
* Then, find the shortest path from the source vertex A to all other vertices. using the relaxation formula ( $d(v) = d(u) + c(u,v)$ )
* The relaxation process shdd continue $n-1$ times where n is no. of vertices

$5-1 = 4$ relaxations

Step-1 :- Relaxation first-lym.



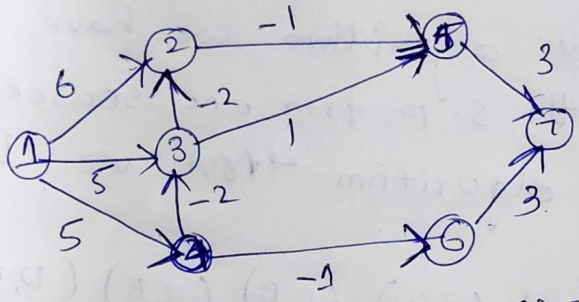
Step-2: Relaxation 2nd tym.




NOTE:-

The disadvantage of Bellman ford cycle is, if the graph has -ve ~~edge~~ cost cycle [i.e sum of edges is -ve] the relaxation will not stop for n-1 tyms

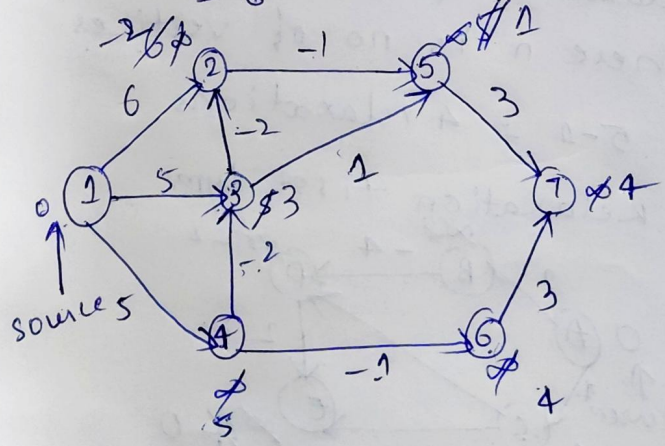
Ex:



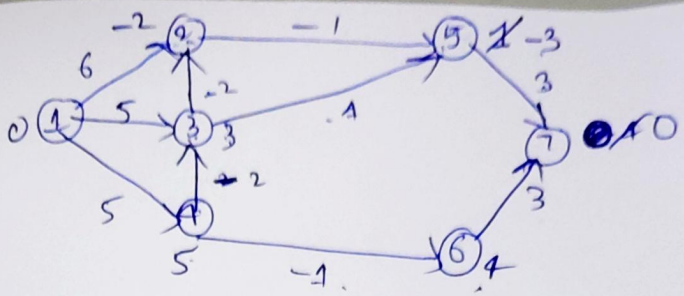

(1,2) (1,3), (1,4) (2,5) (3,2), (3,5) (4,3) (4,6), (5,7) (6,7)

Relaxation = n-1
= 7-1
= 6

Step-1:



step-2 :-




step-3 :-