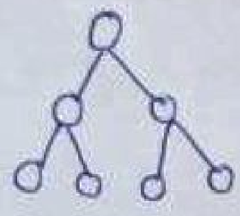
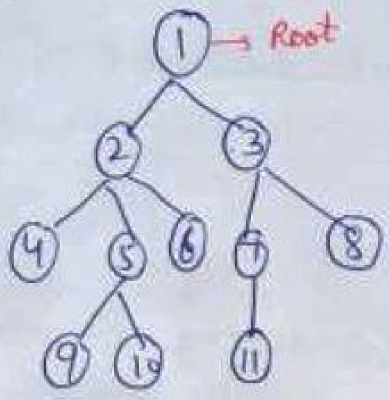
Unit-V

Tree :- It is a non- linear data structure because it doesnot store in a sequential manner. It is a hierarchical structure which is defined as collection of objects or entities known as nodes that are linked together to represent hierarchy.



Terminologies :-



a) **Root** :- It is the topmost node that doesnot have any parent. Root → Node 1 in above eg.

b) **Child node** :- If the node is a descendant of any node, then it is child node. For eg. 2 & 3 are child of 1.

c) **Parent** :- If the node contains any sub-node, then that node is said to be the parent of that sub-node.
For eg. '1' is the parent of 2 & 3.
'2' is the parent of 4, 5, 6. and so on.

d) Sibling :- The nodes that have the same parent.
Eg:- 2,3 are siblings, 4,5,6 are siblings etc.

e) Leaf node :- It is the bottom most node of the tree which doesnot have any child node. It is also known as external nodes. 6,4,8,9,10,11 are leaf node.

f) Internal node :- A node has atleast one child i.e. internal node or non-leaf node (apart from leaf node). 1,2,3,5,7 are internal nodes.

g) Ancestor node :- It is any predecessor node on a path from the root to that node. Eg:- 1, 2 & 5 are ancestors of 10.

h) Descendant :- The immediate successor of the given node is descendent of a node. Eg:- 10 is the descendent of 5.

Properties of tree :→

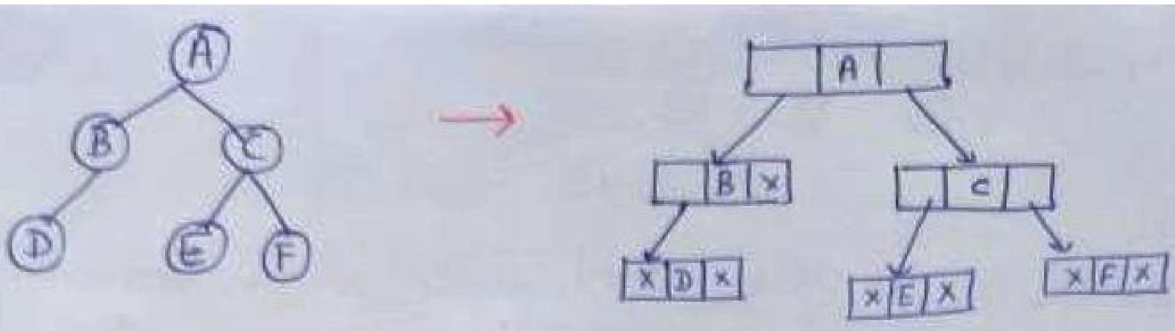
i) Recursive data structure (Reducing something in a self-similar manner).

ii) No. of edges will be (n-1) if there are 'n' nodes.

iii) Depth of node x = length of the path from the root to the node x.

iv) Height of node x = longest path from the node x to the node x.

Implementation of tree :→ Tree is created by creating the nodes dynamically with the help of the pointers.



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

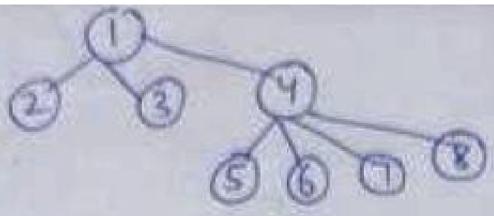
Applications of trees :-

- Storing data hierarchically.
- Organizing data .
- Trie (it is a dictionary to store the data] .
- Heap
- Routing table (Store the data in routers for sending the data) .

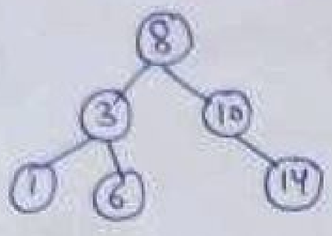
Types of tree :-

- General
- Binary
  - Full
  - Complete
  - Perfect
  - degenerate
  - balanced
- Binary Search Tree
  - AVL Tree
  - Red black Tree
  - Splay Tree
  - Treap
  - B-Tree
- Expression Tree

i) General tree :- In this, a node can have either '0' or max. 'n' no. of nodes. There is no restriction imposed on the degree of the node .

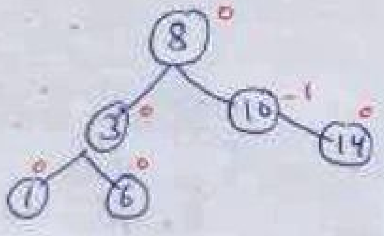


2) Binary tree :→ In this, each node in a tree can have utmost two child nodes.



3) Binary Search tree :→ In this, a node can be connected to the utmost two child nodes, so the nodes in the left subtree contain a value less then root node & the right subtree contain a value bigger than root node. (eg. same as above).

a) AVL tree :- It is a binary search tree with self-balancing factor (height of left subtree - height of right subtree) of 0, -1, +1. (eg. same as above).



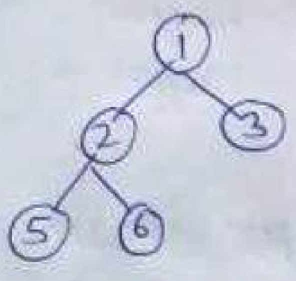
b) Red-black tree :- It is almost same as AVL tree, except in red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents the color of a node i.e. red or black for ensuring the balancing factor.

c) Splay tree → Splay means recently accessed node. In this, the recently accessed element is placed at the root position of tree by performing rotations.

d) Treap → It comprises the properties of both Tree & Heap data structure.

e) B-Tree → It is a balanced m-way tree, where m defines the order of tree.

Binary tree :→ It means that node can have maximum two children.



Properties of Binary tree :→

a) At each level of 'i', the max. no. of nodes = 2^i.

b) Min. no. of nodes at height 'h' = h+1

c) Min. height of tree(h) = $\log_2(n+1) - 1$

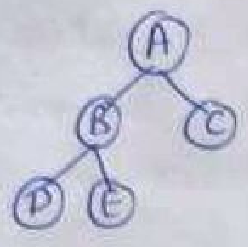
d) Max. height of tree (H) = n-1

e) Max. no. of nodes at height 'h' = $2^{h+1} - 1$

f) If no. of nodes is min., then height of tree would be max., if no. of nodes is max., then height of tree would be min.

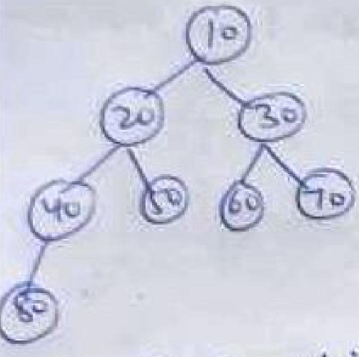
Types of Binary tree :→

i) Full / proper / Strict :→ Each node must contain either '0' or '2' children.



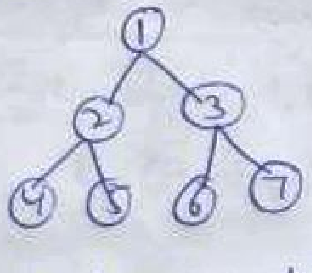
- No. of leaf nodes = Internal nodes + 1
- Max. no. of nodes = $2^{h+1} - 1$
- Min. no. of nodes = $2h - 1$
- Max. height = $(n+1)/2$
- Min. height = $\log_2(n+1) - 1$

ii) Complete binary tree :→ All the nodes are completely filled except the last level, and in the last level, all the nodes must be as left as possible.



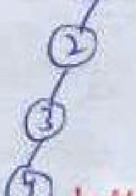
- Max no. of nodes = $2^{h+1} - 1$
- Min. no. of nodes = 2^h
- Max. height = $\log n$
- Min. height = $\log_2(n+1) - 1$

iii) Perfect binary tree → All internal nodes have 2 children & all leaf nodes are at same level.

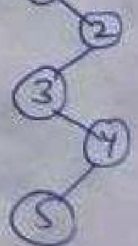


iv) Degenerate binary tree :→ internal nodes have only one child.

Right skewed
(i.e. Right child only)



left skewed
(i.e. left child only)



v) Balanced binary tree :→ like AVL & red-black tree i.e. balancing factor must be −1, 0, +1

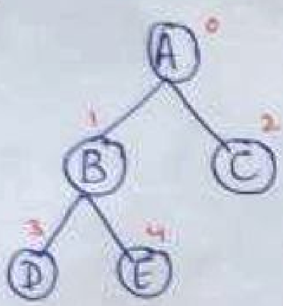
Implementation / Representation of Binary tree :→

Sequential OR Array representation

Dynamic OR Linked list Representation

1. Array representation :- There are two cases :-

- Case I :- start index with 0.



0	1	2	3	4
A	B	C	D	E

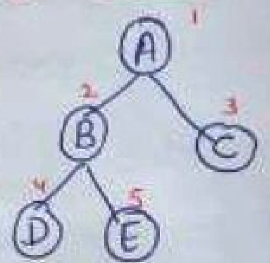
- If a node is at i^{th} index, then

→ left child of that node = $[(2 \times i)+1]$

→ right child = $[(2 \times i)+2]$

- Parent node = $\left\lfloor \frac{(i-1)}{2} \right\rfloor$

- Case II :- Start with 1.



1	2	3	4	5
A	B	C	D	E

- If a node is at i^{th} index, then

→ left child = $(2 \times i)$

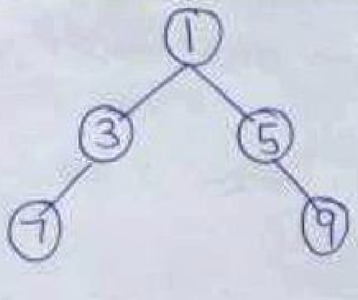
→ right child = $[(2 \times i)+1]$

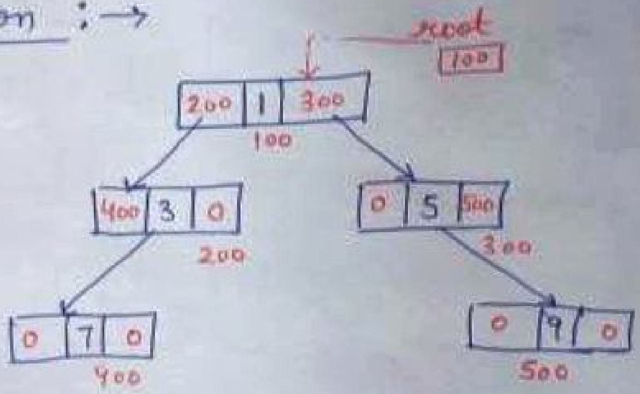
- Parent node = $\left\lfloor \frac{i}{2} \right\rfloor$

```
char tree[10];
int root (char x)
{
      if ( tree[0] ! = '\0')
                printf (" Tree has already root");
      else
             tree[0] = x;
      return 0;
}
int left ( char x,  int parent)
{
```

```
    if ( tree [parent] = = '\0' )
            printf(" No parent found");
    else
            tree [(parent * 2) + 1] = x;
    return 0;
}
int Right (char x, int parent)
{
    if ( tree [parent] = = '\0')
            printf (" No parent found");
    else
            tree [(parent * 2) + 2] = x;
    return 0;
}
```

2. Linked list representation :→






```
struct node
{   int data;                              // creation of Structure
    struct node * left, * right;
};
void main ()
{
    struct node * root;
    root = create ();
}
struct node * create ()
{
```

```c
struct node * temp;
int data;
temp = (struct node *) malloc(size of (struct node));
printf("Press 0 to exit");
printf("Press 1 to new node");
printf("enter your choice");
scanf("%d", & choice);
if (choice == 0)
  {      return 0;
  }
else
  {
        printf("enter data");
        scanf("%d", & data);
        temp -> data = data;
        printf("enter left child of %d", data);
        temp -> left = create();
        printf("enter right child of %d", data);
        temp -> right = create();
        return temp;
  }
}
```

Tree traversal :-> It means traversing or visiting each node of a tree. There are multiple ways:-

a) Preorder traversal → Root left right

b) Inorder traversal → left Root right

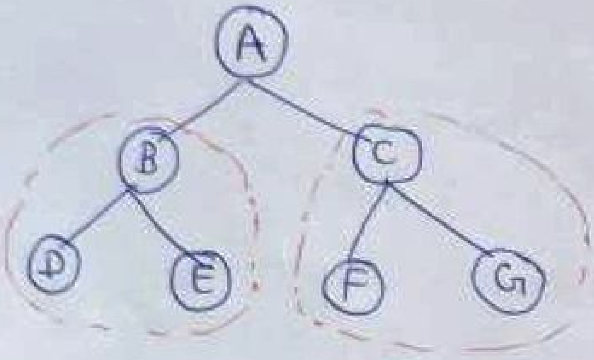
c) Postorder traversal → left right Root

a) Preorder traversal :→ The first root node is visited after that left subtree is traversed recursively, and finally right subtree is recursively traversed.

Algorithm :-

i) Visit the root node.
ii) Traverse the left subtree recursively.
iii) Traverse the right subtree recursively.

Eg :-



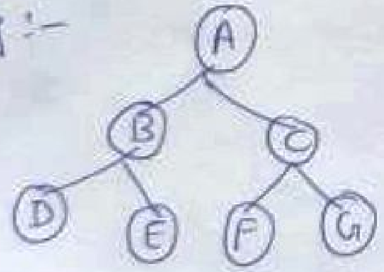
A → B → D → E → C → F → G

b) Postorder traversal :→ The first left subtree of the root node is traversed, after that recursively traverse the right subtree & finally the root node is traversed.

Algorithm :-

i) Visit the left subtree recursively.
ii) Traverse the right subtree recursively.
iii) Visit the root node.

Eg :-



D → E → B → F → G → C → A

c) Inorder traversal :- The first left subtree is visited after that root node is traversed and finally, the right subtree is traversed.

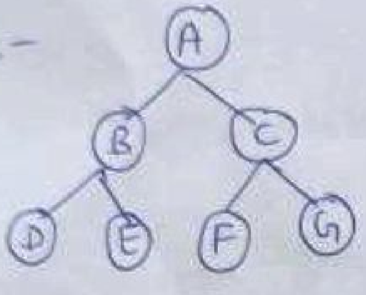
Algorithm :-

i) Traverse the left subtree recursively.

ii) Visit the root node.

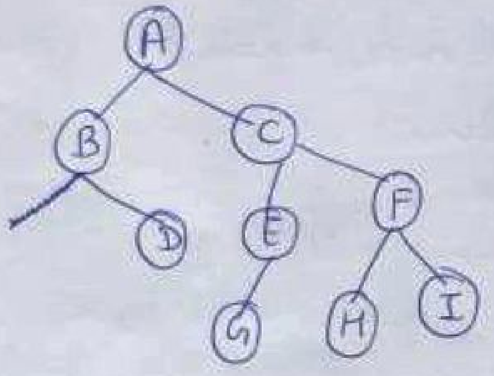
iii) Traverse the right subtree recursively.

Eg:-



D → B → E → A → F → C → G.

Q.



Find inorder, preorder & postorder.

Sol

Inorder → B D A G E C H F I

Preorder → A B D C E G F H I

Postorder → D B G E H I F C A

```
struct node
{   int data
     struct node * left, * right;

  };
  struct node * create (int x)
  {
```

```
    struct node * new-node = (struct node *) malloc(sizeof(struct node));
    new-node -> data = x;
    new-node -> left = NULL;
    new-node -> right = NULL;
    return (new-node);
}
void Preorder( struct node * root)
{
        if ( root == NULL)
              return;
        printf("%d", root -> data);
        preorder ( root -> left);
        preorder (root -> right);
}
void inorder ( struct node * root )
{
        if ( root == NULL)
              return;
        inorder (root -> left);
        printf("%d", root -> data);
        inorder ( root -> right);
}
void postorder ( struct node * root )
{
        if (root == NULL)
              return;
        postorder ( root -> left);
        postorder ( root -> right);
        printf("%d", root -> data);
}
```

Construct a binary tree from preorder & inorder:-

Check values from both the orders.

Algorithm :→

i) Pick an element from preorder. Increment a preorder index variable to pick the next element in the next recursive call.

ii) Create a new tree node 'tnode' with the data as the picked element.

iii) Find the picked element's index in inorder. Let the index be inIndex

iv) Call buildTree for elements before inIndex & make the built tree as a left subtree of 'tnode'.

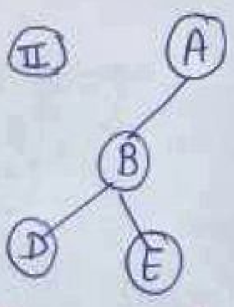
v) Call buildTree for elements after inIndex & make the built tree as a right subtree of 'tnode'.

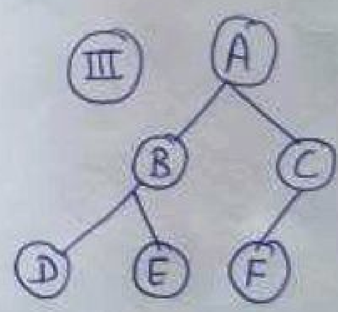
vi) Return 'tnode'.

Eg:- Preorder → A B D E C F

Inorder → D B E A F C

Sol Scan preorder from left to right & then check subtrees from inorder.





Construct a binary tree from postorder & inorder :→

Check values from both the orders.

Algorithm :-

i) Make a variable postIdx initialized to pick the next required node in the recursive call i.e. check the last element or scan from right to left.

(ii) Initialize a new tree node with the value picked.

iii) Traverse and find the node in the inorder traversal.

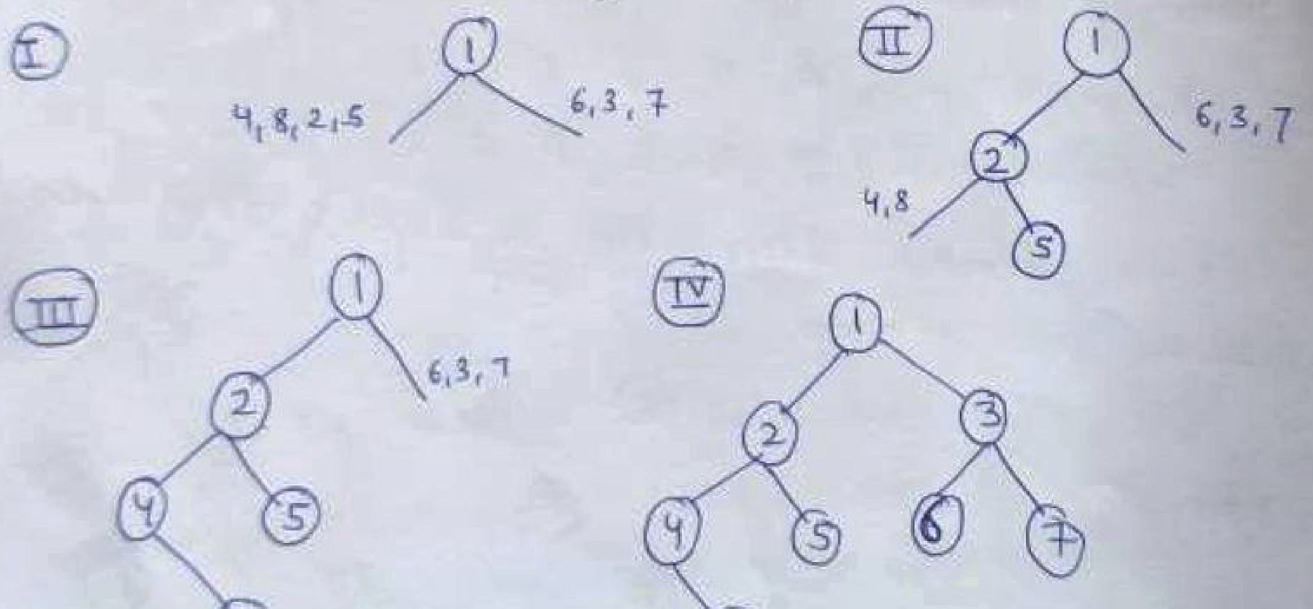
iv) Now, make two recursive calls, one for the left subtree passing value before the inorder index & one for the right subtree with value after the inorder.

v) Return the initialized node.

Eg:- Inorder → 4, 8, 2, 5, 1, 6, 3, 7

postorder → 8, 4, 5, 2, 6, 7, 3, 1

Sol. Scan postorder from right to left & then check inorder values for left & right subtree.



Construct Binary tree from preorder & postorder

From preorder & postorder, we are unable to construct unique binary tree, but full binary tree is possible.

Algorithm :-

i) Find the root from postorder & preorder.

ii) Rearrange the nodes in the tree by finding the root again at another level.

Eg:- Preorder → 1, 2, 4, 8, 9, 5, 3, 6, 7

Postorder → 8, 9, 4, 5, 2, 6, 7, 3, 1

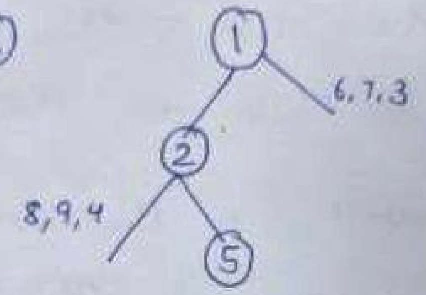
Sol.

(I)

(II)

Pre → 2, 4, 8, 9

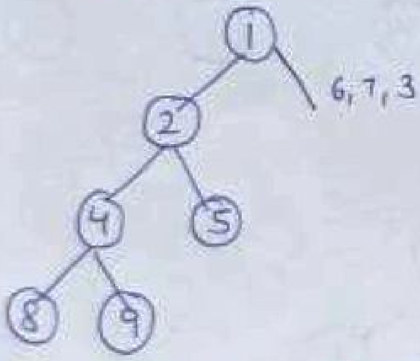
Post → 8, 9, 4, 5, 2



(III)

Preorder – 4, 8, 9

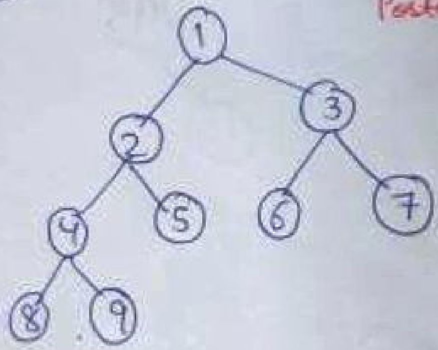
Postorder – 8, 9, 4



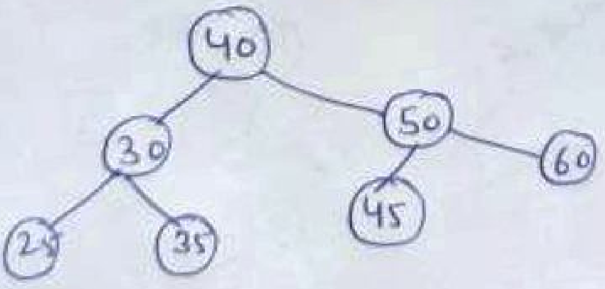
(IV)

Preorder – 3, 6, 7

Postorder – 6, 7, 3



Binary Search tree :→ It follows some order to arrange the elements i.e. the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.



Creation of binary searchtree :-

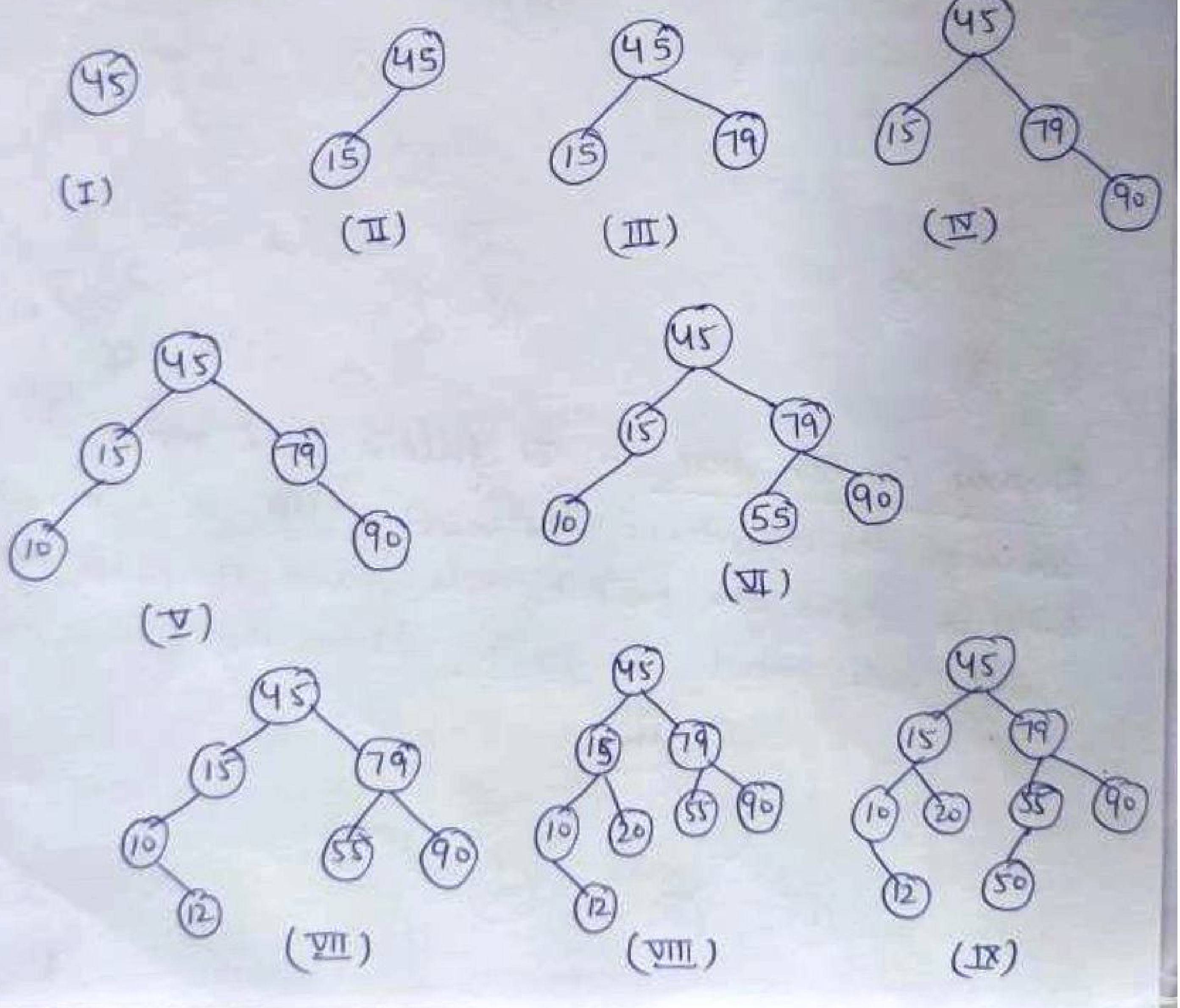
45, 15, 79, 90, 10, 55, 12, 20, 50

Algo :-

i) First, insert the first element in series at root node i.e. (45 in eg)

(ii) Then, read the next element, if it is smaller than the root node, insert it as the root of the left subtree & move to the next element.

iii) Otherwise, if the element is larger than root node, then insert it as root of the right subtree.



Insertion :- A new key in BST (Binary Search tree) is always inserted at leaf. To insert an element, start searching from root node :-

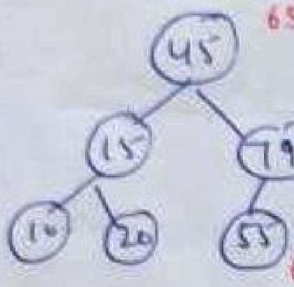
i) If it is less than root node, then search for the empty location in ~~rig~~ left subtree,

ii) else, search for empty location in right subtree.

Eg:-

65>45, then move to right subtree.

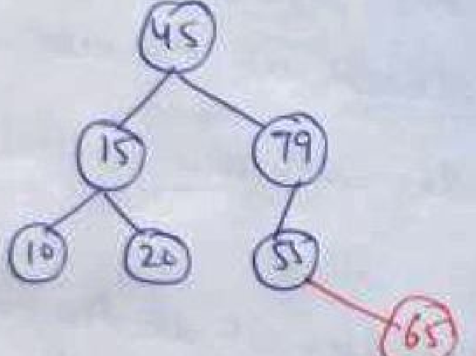
Insert 65



65<79, move to left subtree

65>55, move to right subtree, its empty then place over it.

Sol

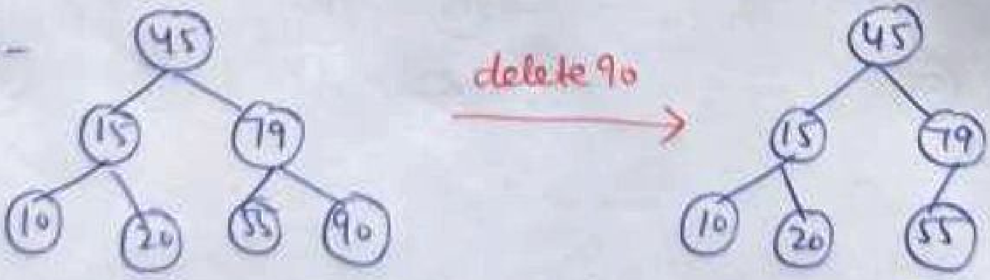


Deletion :- There are three possible situations :-

i) The node to be deleted is the leaf node :-

It is the simplest case. Just replace the leaf node with Null & simply free the allocated space.

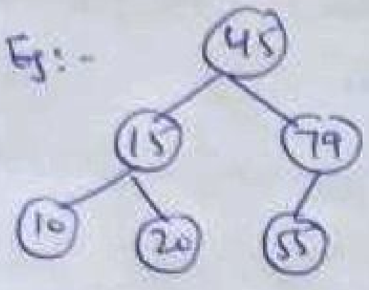
Eg :-



(ii) The node to be deleted has only one child :-

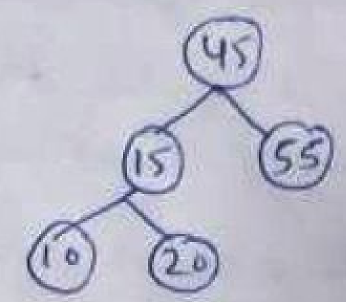
Replace the target node with its child & delete the child node.

Eg:-



Delete 79 →

Replace it with 55 & then delete.



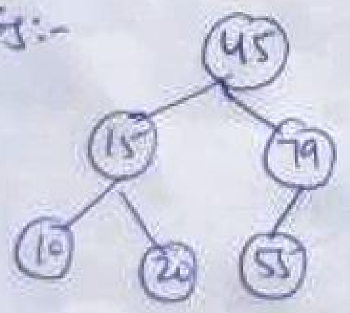
iii) The node to be deleted has two children :- There are some steps :-

- Find the inorder successor of the node to be deleted.
- Replace the node with the inorder successor until the target node is placed at the leaf of tree.
- At last, replace the node with NULL & free up the allocated space.

* Inorder successor = min. element in the right child of the node.

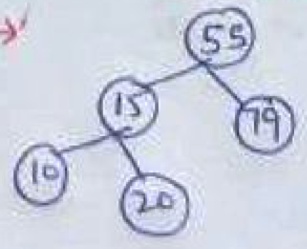
* Inorder predecessor = max element in the left child of the node.

Eg:-



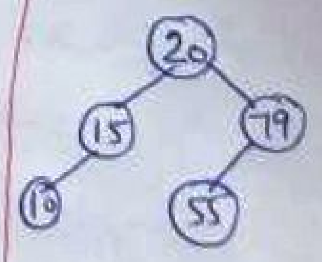
delete 45 →

Inorder Successor



OR

Inorder Predecessor



Searching :- Find or locate a specific element or node

Algo:-

i) Compare the element to be searched with the root element of the tree.

ii) If root is matched with the target element, then return the node's location.

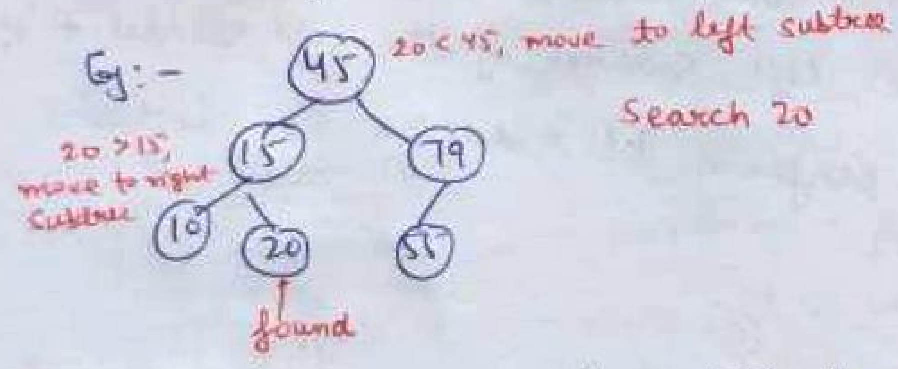
iii) If not matched, then check whether the item is less than root element, if it is smaller than root element, then move to the left subtree.

iv) If it is larger than root element, then move to the right subtree.

v) Repeat the above procedure recursively until the match is found.

vi) If the element is not found or not present in the tree, then return NULL.

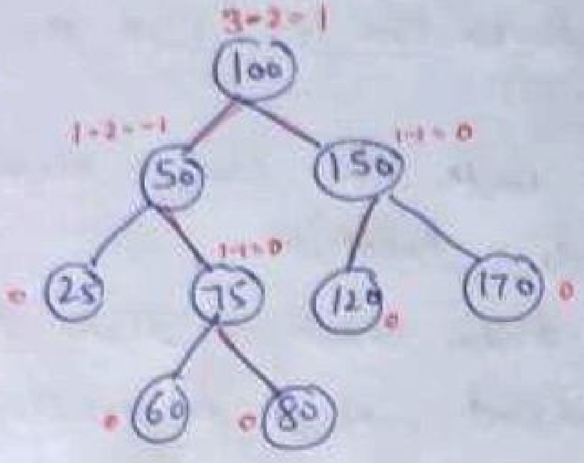
Eg:-



AVL Tree :- It is invented by GM Adelson-Velsky & EM Landis in 1962, that's why it is named AVL tree. It is defined as height balanced binary search tree in which each node is associated with a balance factor (height of left subtree - height of right subtree).

Tree is balanced if balancing factor of each node is in between -1 to 1 or (-1, 0, 1).

Eg:-



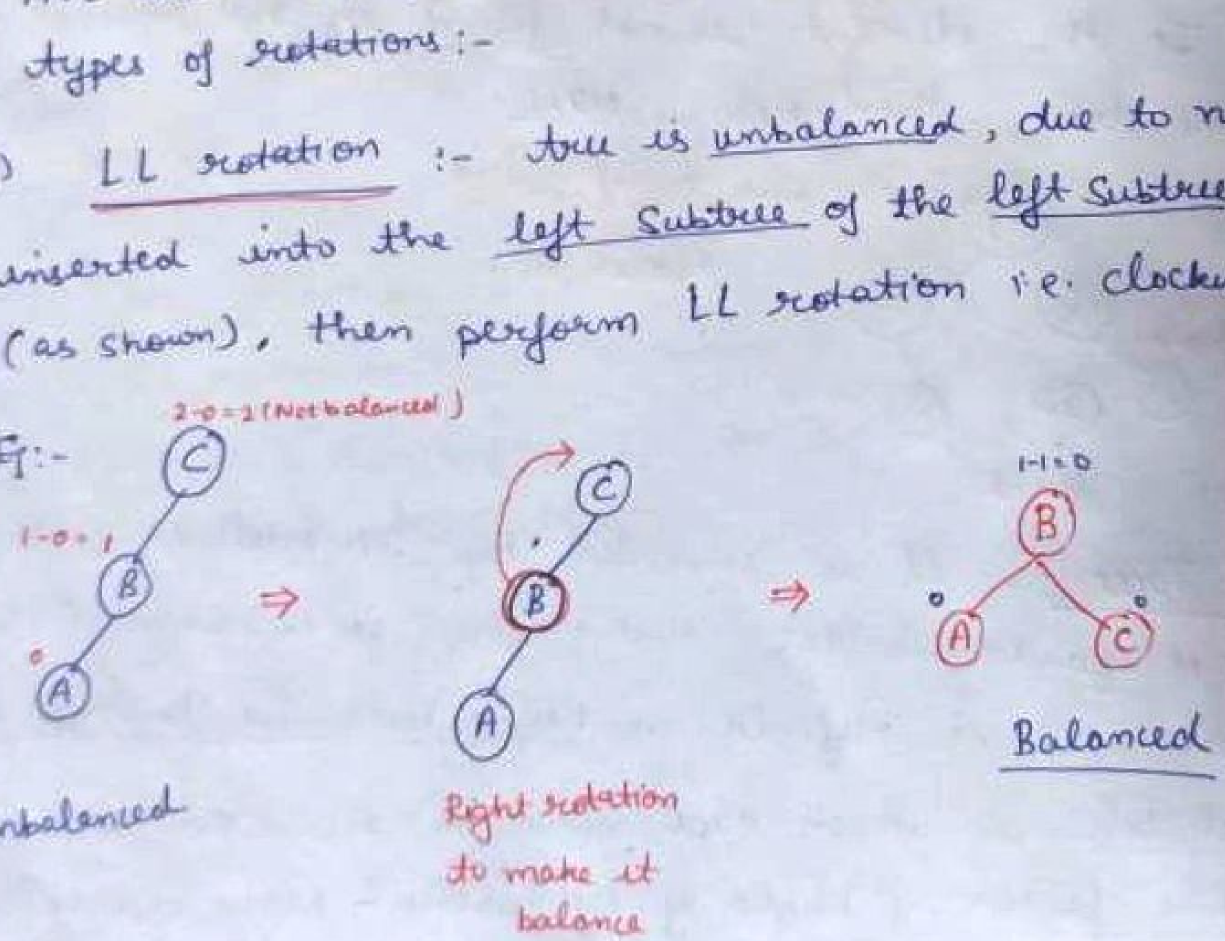
Operations on AVL:-

- Insertion
- deletion
- Searching

AVL rotations:- If tree is not balanced, then perform AVL rotation to make it balance. There are four types of rotations:-

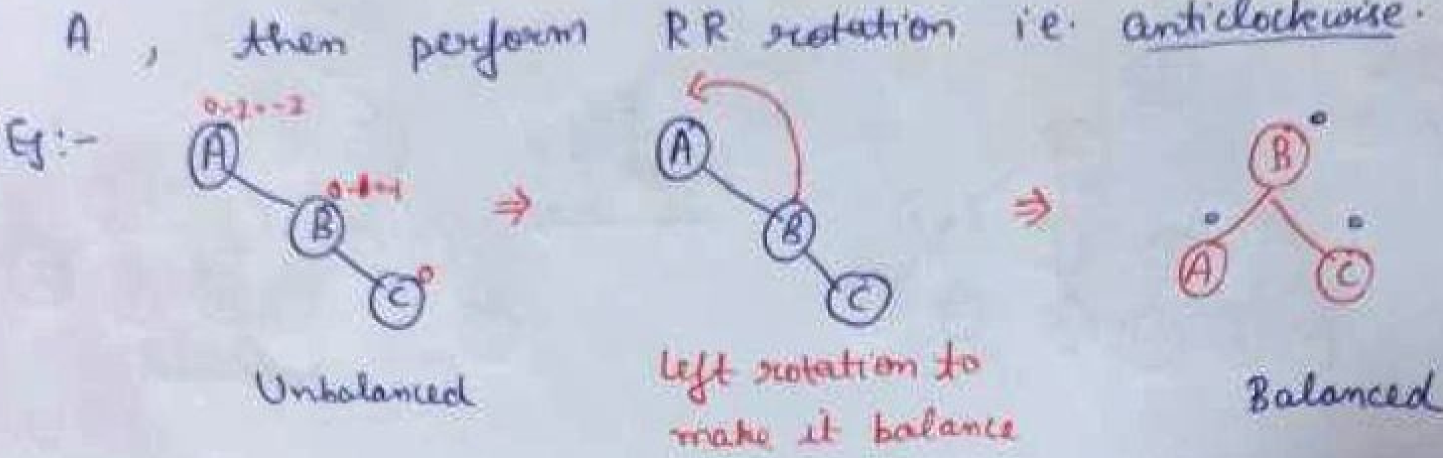
i) **LL rotation** :- tree is unbalanced, due to node is inserted into the left subtree of the left subtree of C (as shown), then perform LL rotation i.e. clockwise.

Eg:-



ii) <u>RR rotation</u> :- tree is unbalanced, due to node is inserted into the right subtree of the right subtree A, then perform RR rotation i.e. anticlockwise.

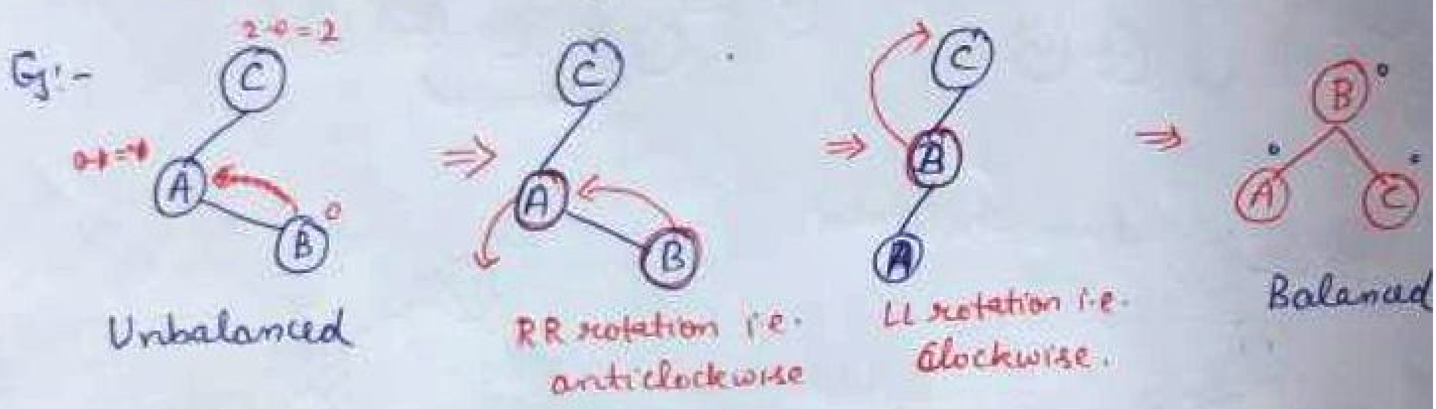
Eg:-



iii) <u>LR rotation</u> :-

$\boxed{\text{LR rotation = RR rotation + LL rotation}}$ i.e. first RR rotation is performed on subtree & then LL rotation is performed on full tree to make it balance.

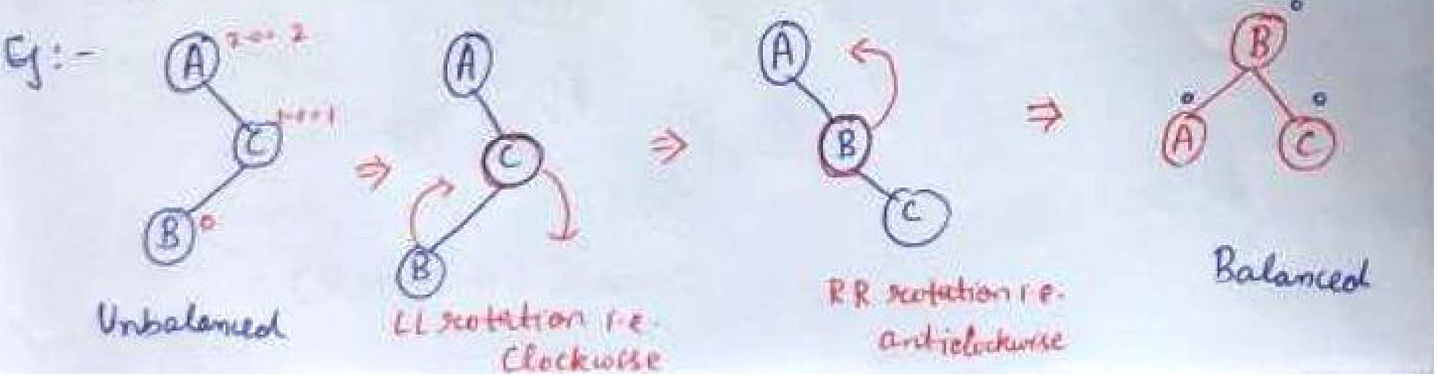
Eg:-



iv) <u>RL rotation</u> :-

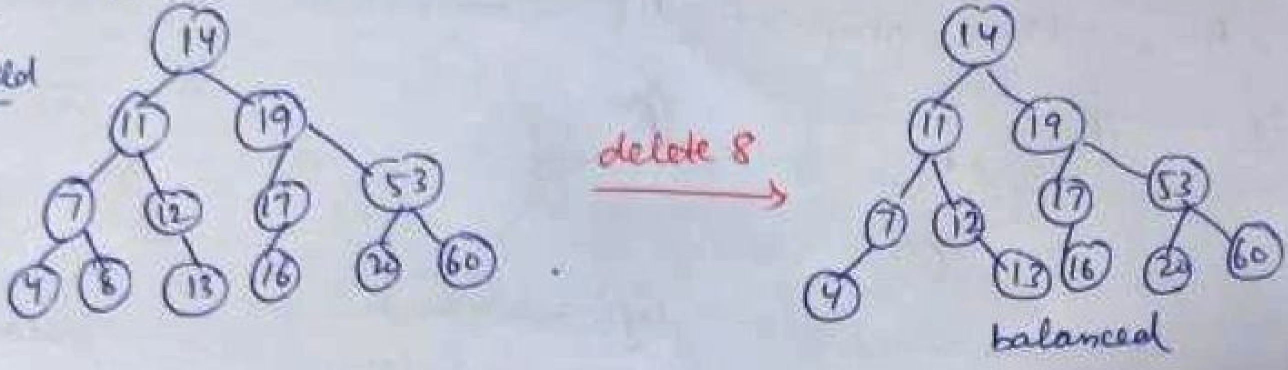
$\boxed{\text{RL rotation = LL rotation + RR rotation}}$ i.e. first LL rotation is performed on subtree & then RR rotation is performed on full tree to make it balance.

Eg:-

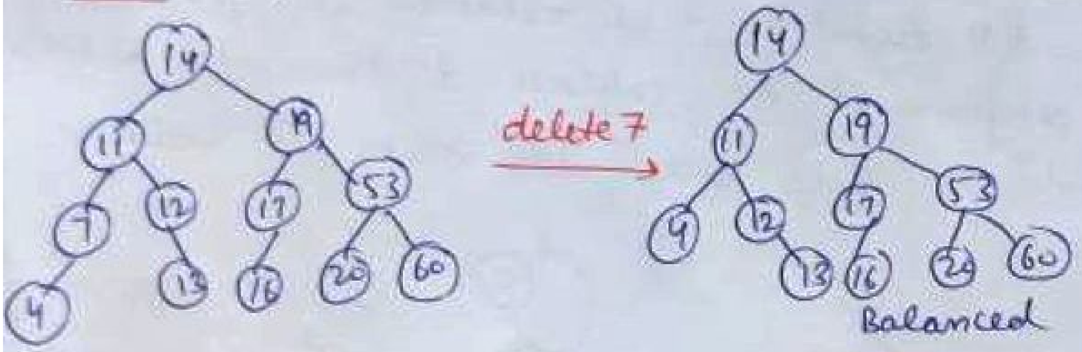


Deletion in AVL tree:- Same as BST, but check the balancing factor too.

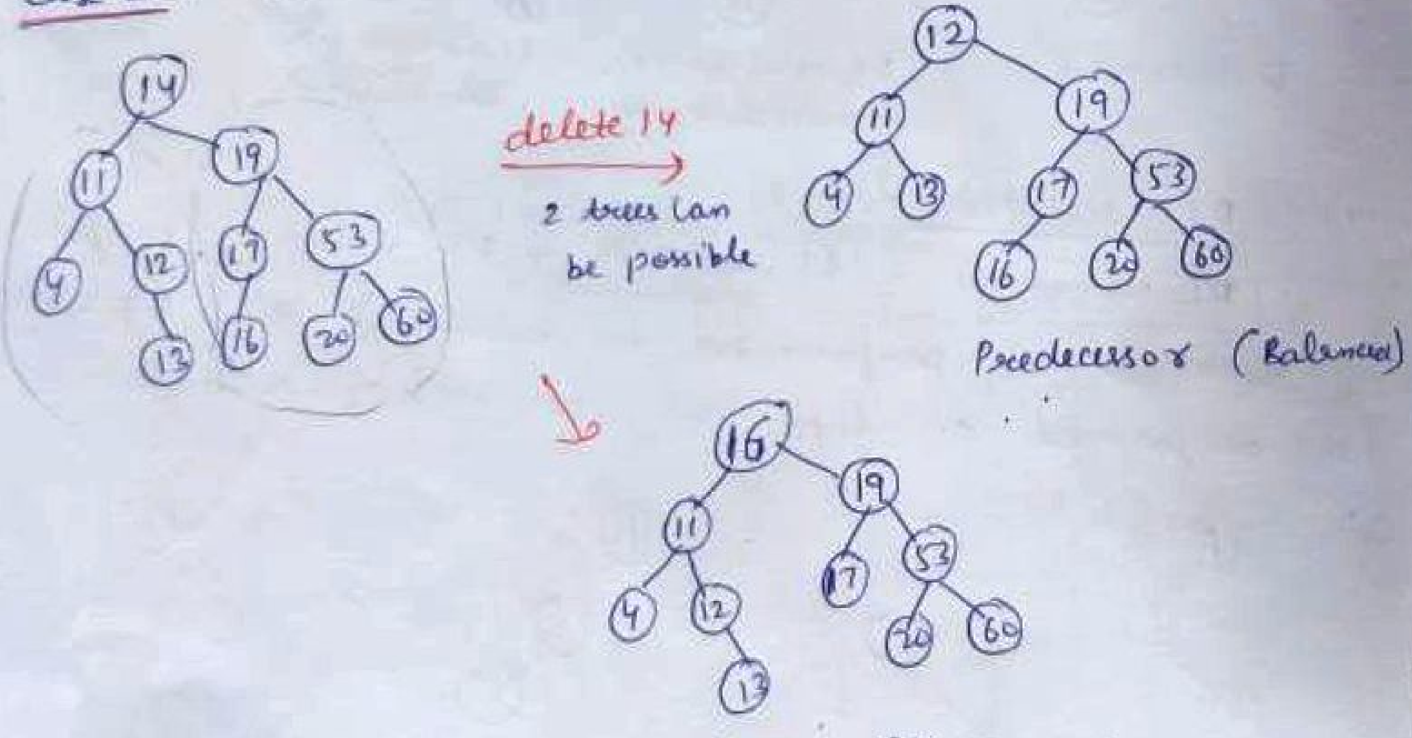
Case I:-
No child



Case II:- one child



Case III:- two children

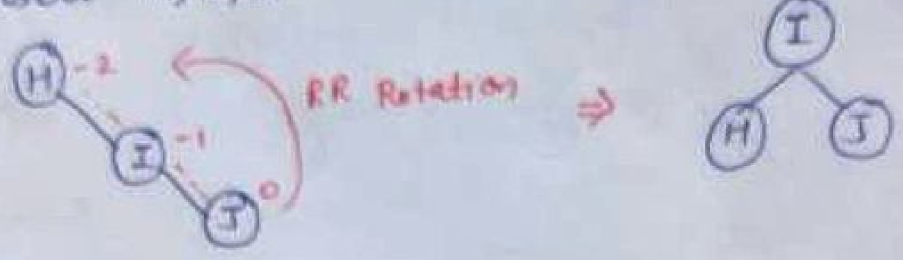


Sucessor (Balanced)

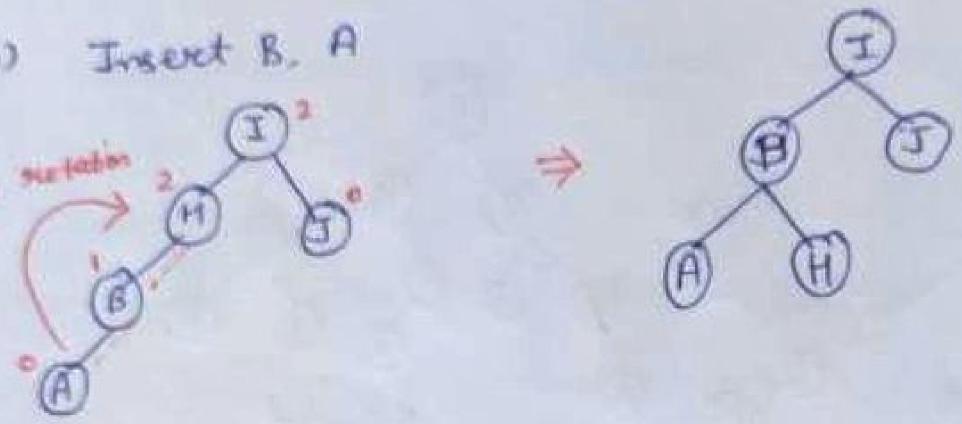
Q. Construct an AVL tree having the following elements:-

H, I, J, B, A, E, C, F, D, G, K, L.

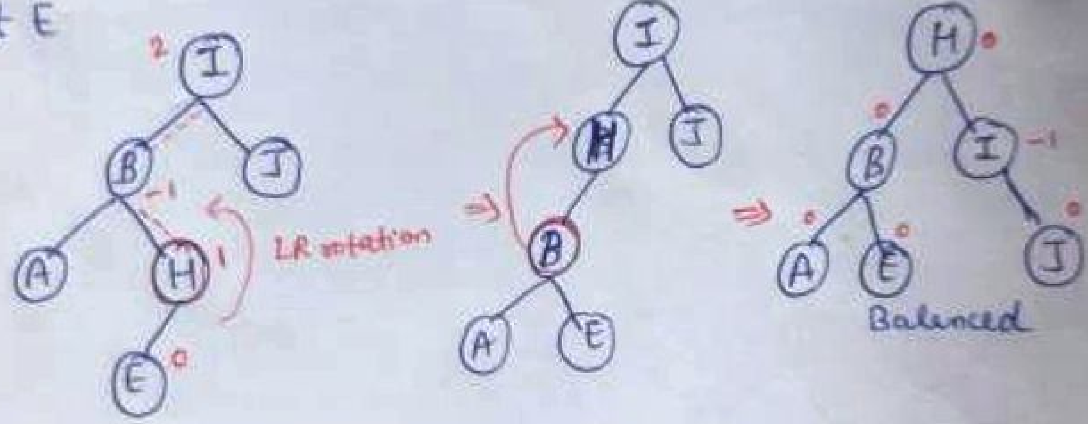
Sol (i) Insert H, I, J



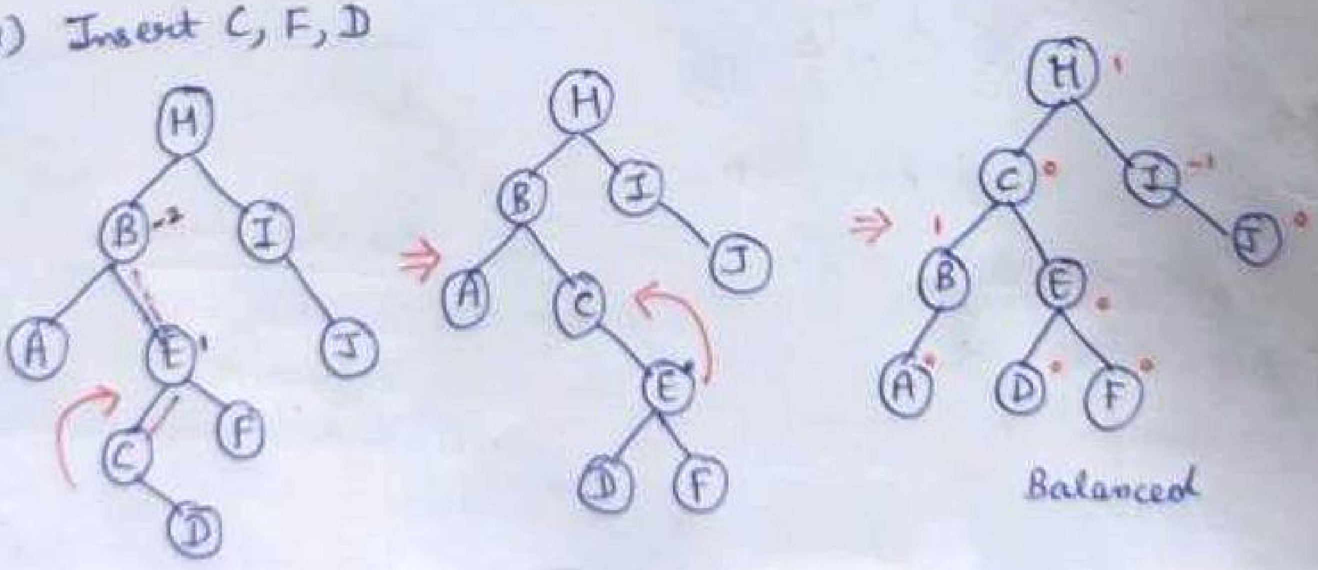
(ii) Insert B, A



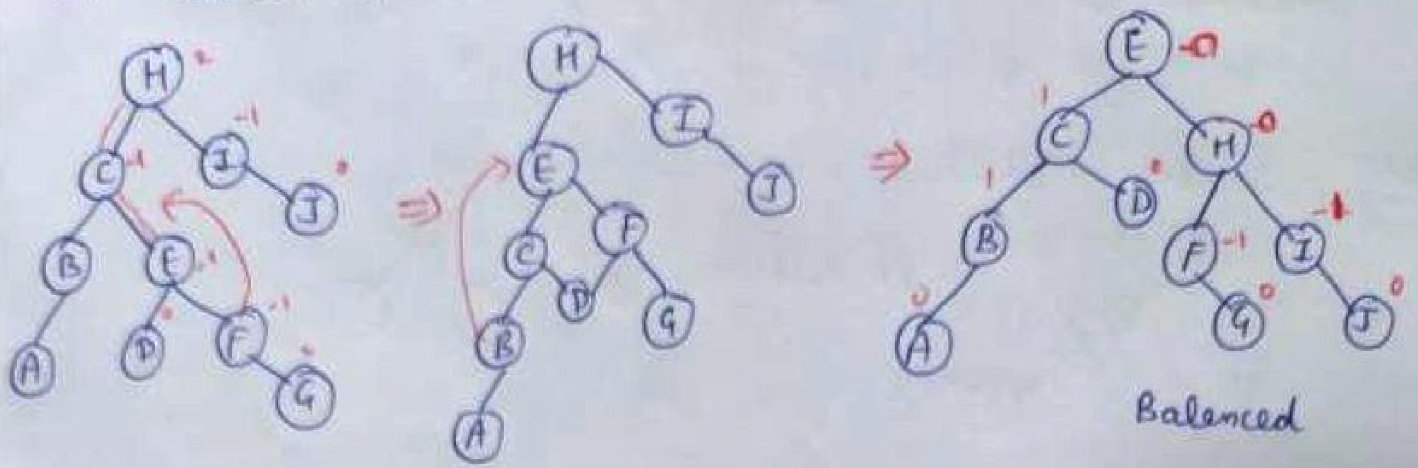
(iii) Insert E



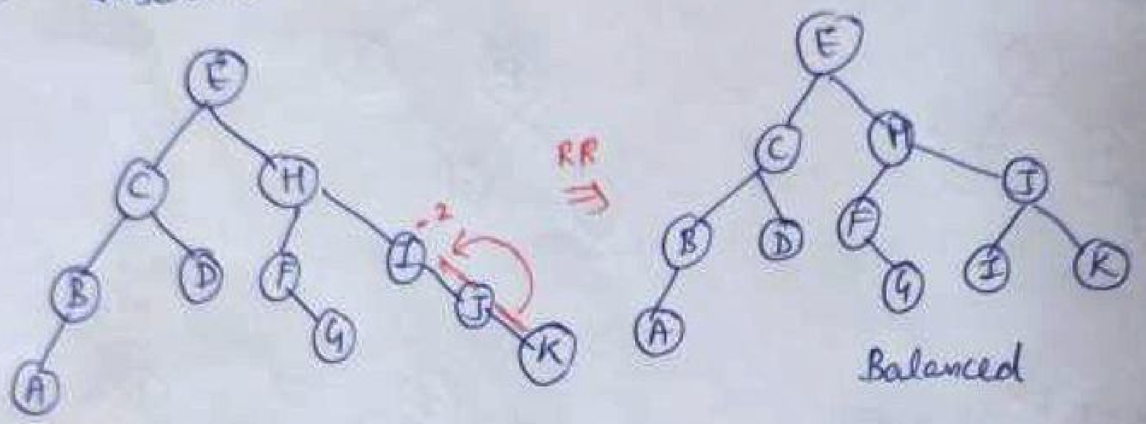
iv) Insert C, F, D



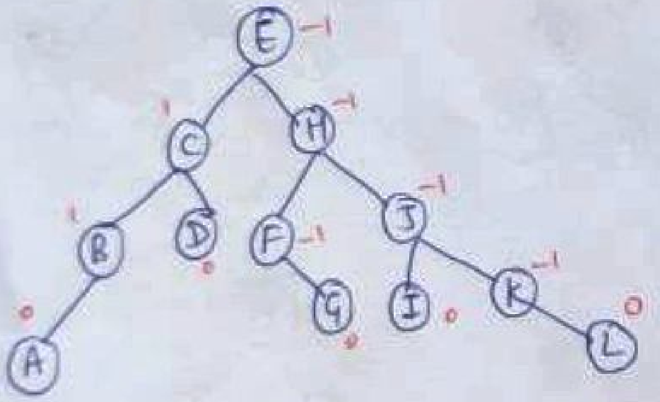
v) Insert G



vi) Insert K



vii) Insert L



Final AVL tree

Advantages:-

1) Self - balancing.
2) Efficient.
3) Quick.

Disadvantages:-

1) Complex.
2) Expensive.
3) Extra Space is required.

B-Tree :→ It is a specialized m-way tree that can be widely used for disk access. A B-Tree of order 'm' can have atmost m-1 keys and m children. One of the main reason of using B-tree is its capability to store large no. of keys in a single node and large key values by keeping the height of the tree relatively small. It contains following properties:-

i) Every node in a B-Tree contains at most 'm' children.

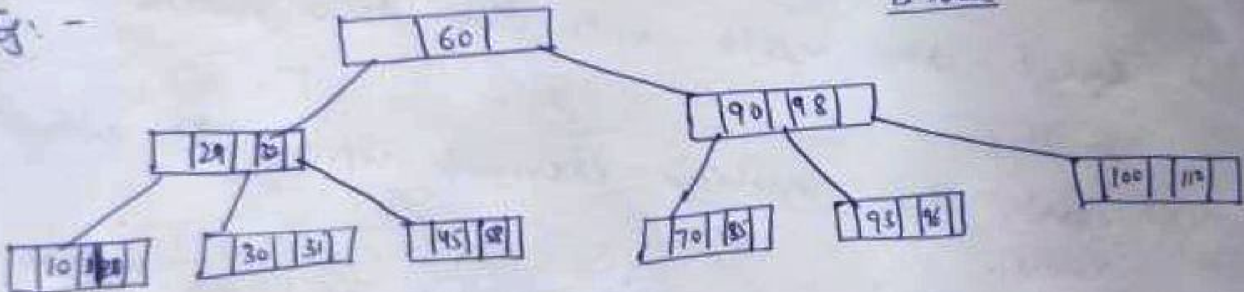
ii) Every node in a B-Tree except the root node & leaf node contain atleast m/2 children.

iii) The root nodes must have atleast 2 nodes.

iv) All leaf nodes must be at the same level.

B-tree

Eg:-



Operations :→

a) Searching :→ It is similar to that in Binary Search tree. For eg., if we search for an item 10 then the process will like:-

i) Compare item 10 with root node 60. Since 10 < 60, hence move to its left subtree.

ii) Since, 10 < 29 < 32, traverse ~~right~~ left subtree of 29.

iii) 10 < 29, move to left.

iv) Match found, return.

(b) Inserting :- Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B-tree :-

i) Traverse the B-Tree in order to find the appropriate leaf node at which the node can be inserted.

ii) If the leaf node contain less than m-1 keys, then insert the element in the increasing order.

iii) else, if the leaf node contains m-1 keys, then follow the following steps :-

   i) Insert the new element in the increasing order of elements.

   ii) Split the node into the two nodes at the median.

   iii) Push the median element upto its parent node.

   iv) If the parent also contain m-1 no. of keys, then split it too by following the same steps.

Eg :- Create a B-Tree of order '3' i.e. m = 3 by inserting the values from 1 to 10.

Sol

m = 3 i.e. (m-1) keys will be possible.

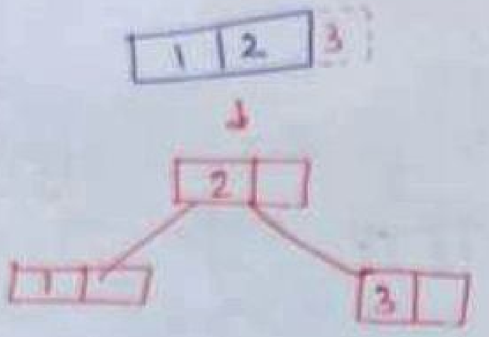
(i) Insert 1

1	

ii) Insert 2

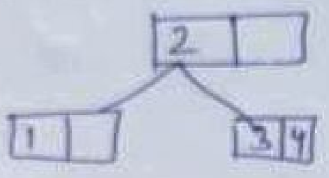
1	2

iii) Insert '3'

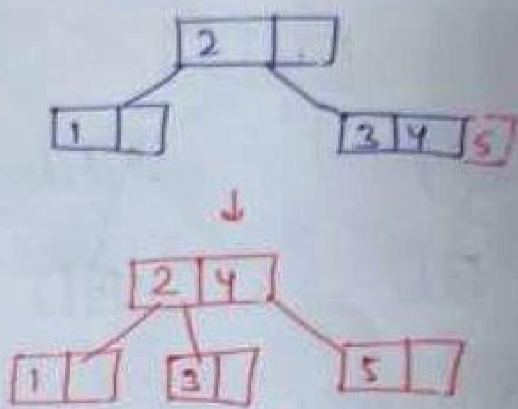


'3' can be inserted here, as we can have (m-1) i.e. 2 keys only. So, median value i.e. '2' will go up & become parent.

iv) Insert '4'

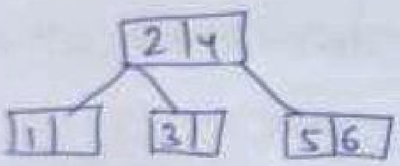


v) Insert '5'

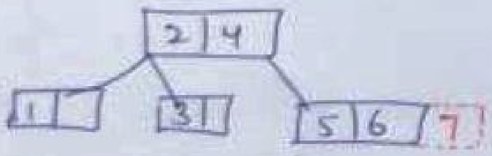


median i.e. 4 will go up

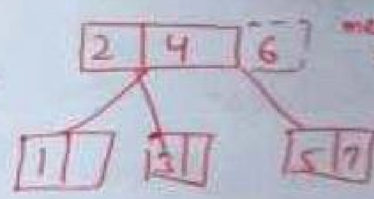
vi) Insert '6'



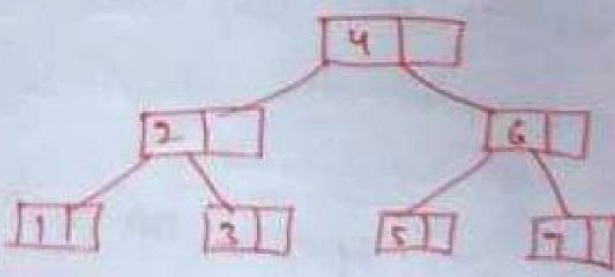
vii) Insert '7'



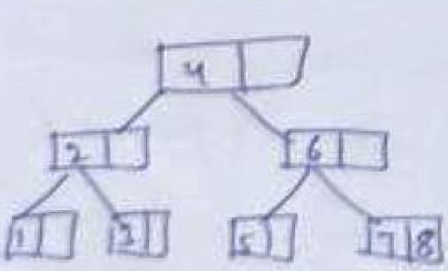
median i.e. 6 go up.



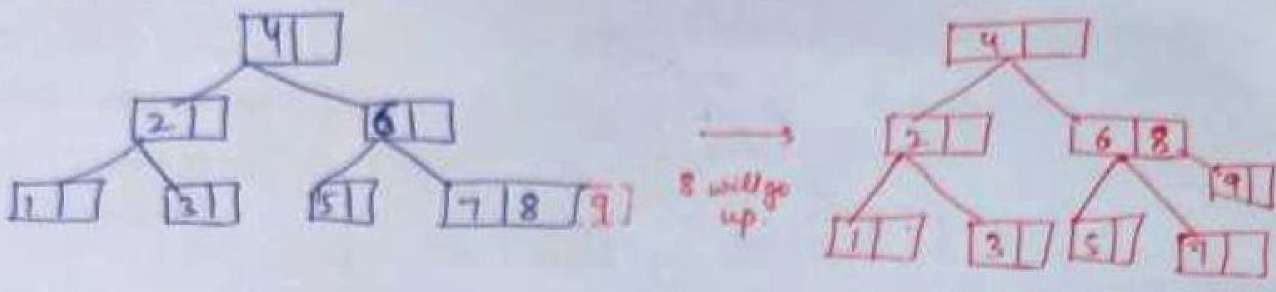
median i.e. 4 will go up



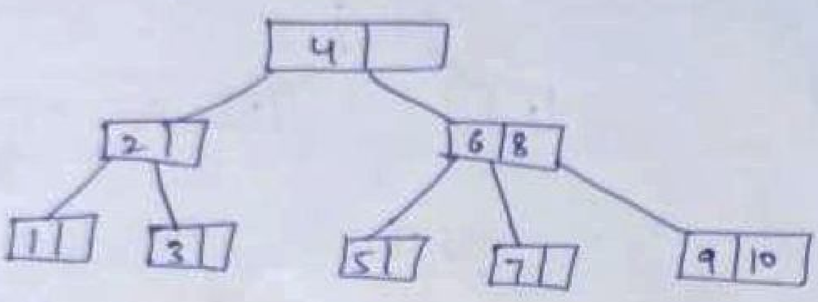
viii) Insert '8'



ix) Insert 9.



x) Insert 10



ⓒ <u>Deletion</u> :→ It is also performed at the <u>leaf</u> nodes. The node which is to be deleted can either be a leaf node or an <u>internal node</u>.

<u>Algorithm</u> :→

i) locate the <u>leaf</u> node.

ii) If there are <u>more then m/2 keys</u> in the leaf node then delete the desired key from the node.

iii) If the leaf node doesnot contain m/2 keys then Complete the keys by taking the element from <u>right</u> or <u>left Sibling</u>

a) If the <u>left Sibling</u> contains more than m/2 elements then push its largest element upto its parent & move the interwening element down to the node where the key is deleted.

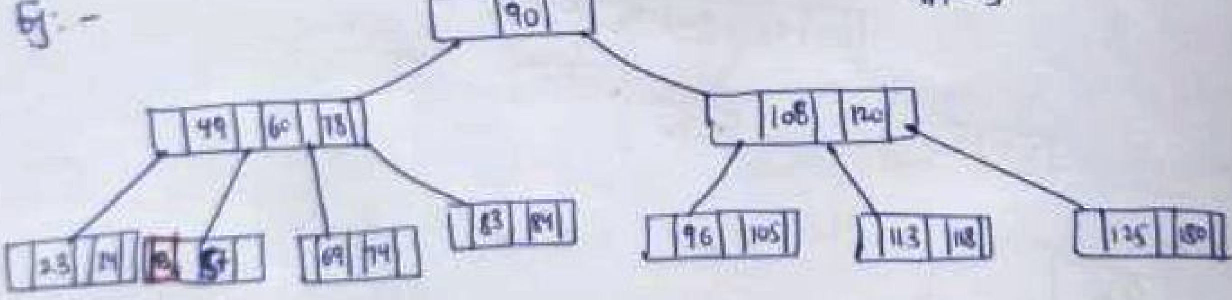
b) If the right sibling contains more than m/2 elements then push its smallest element upto the parent & move intervening element down to the node where the key is deleted.

iv) If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes & the intervening element of the parent node.

v) If parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Eg:- m=5



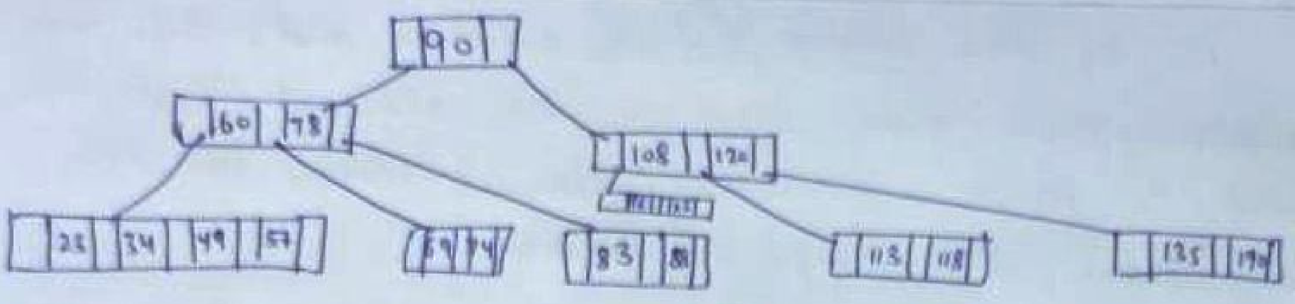
↓delete 53

if we delete 53, then it voilate the property of B-Tree.

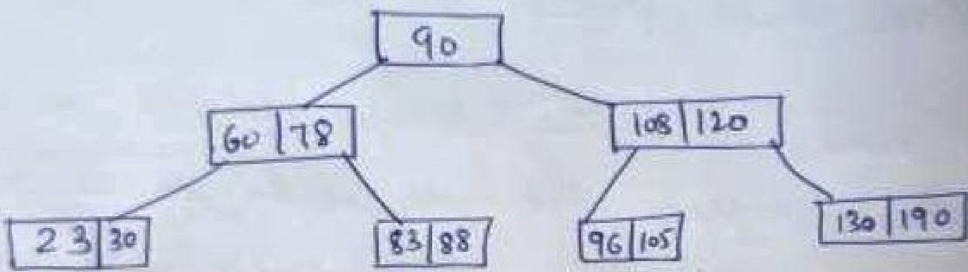
i.e. Min children = $\lceil \frac{m}{2} \rceil$, Max children = m

Min keys = $\left(\lceil \frac{m}{2} \rceil - 1\right)$, Max keys = m - 1

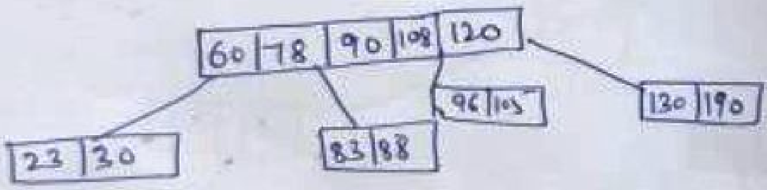
So, merge left node & then parent (i.e. 49 down)



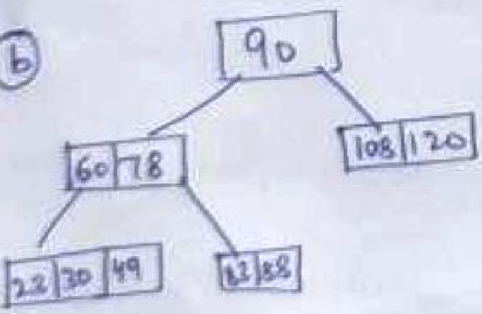
Case II :- If the deleted node is internal node, then take values from right & left siblings (after inorder predecessor & successor). If right & left siblings have min. keys then root node comes down & merge its left & right child.



ⓐ If we delete 60, then it voilates the condition of min. keys. So, root node i.e. '90' comes down & merge left & right subtree.

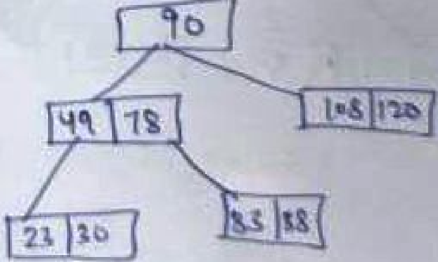


ⓑ

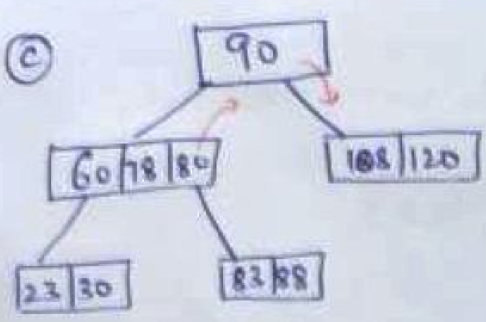


delete 60 →

inorder predecessor (highest value from left subtree)

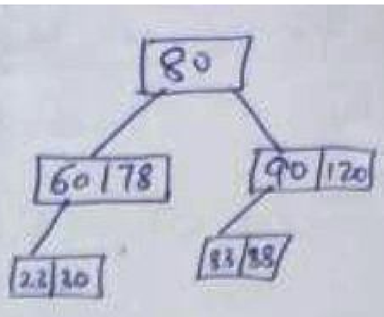


ⓒ



delete 108

Take value from left sibling i.e 80 (value transfered from root node).



Huffman coding using Binary tree :- It is a lossless data compression mechanism which was proposed by David A. Huffman in 1950. Generally, each character stores 8-bits of 0's & 1's, then this is fixed-length encoding. But, we have to reduce the amount of space that is possible by using variable-length encoding. It has two major steps:-

a) Huffman tree construction

i) Create a leaf node contains frequency of each character.

ii) set all nodes in sorted order acc. to their frequency.

iii) If two nodes have same frequency.

- Create a new internal node.
- frequency of node will be sum of frequency of those nodes that have same frequency.
- Make the first node as the left child & another as right child of new node.

iv) Repeat step (ii) & (iii) until all nodes form.

b) Assign Huffman code to each character by traversing a tree.

Eg:- Encode a b r a c a d a b r a using Huffman coding.

(a) Create Huffman tree

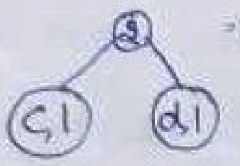
Step 1:- Find frequencies & make pairs with characters.

(a, 5), (b, 2), (c, 1), (d, 1), (x, 2).

Step 2:- sort in ascending order w.r.t. frequencies.

(c, 1), (d, 1), (b, 2), (x, 2), (a, 5).

Step 3:- pick first two characters & join them under a parent node.

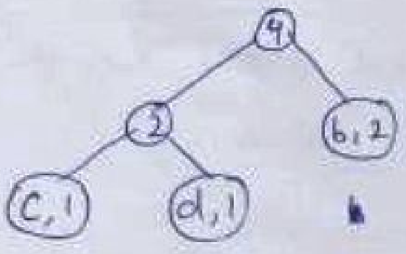


=) c+d = 1+1 = 2

(c,d, 2), (b, 2), (x, 2), (a, 5).

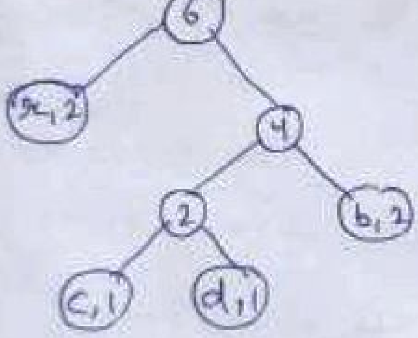
Step 4:- Repeat steps 2 & 3 until a tree is formed.

(i)



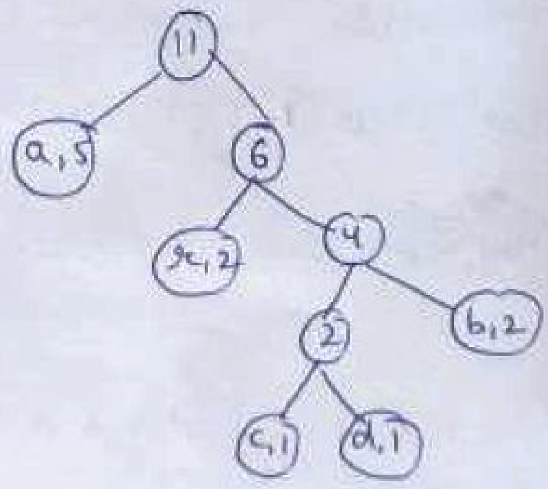
(b,c,d, 4)
(x, 2) ; (a, 5)

(ii)

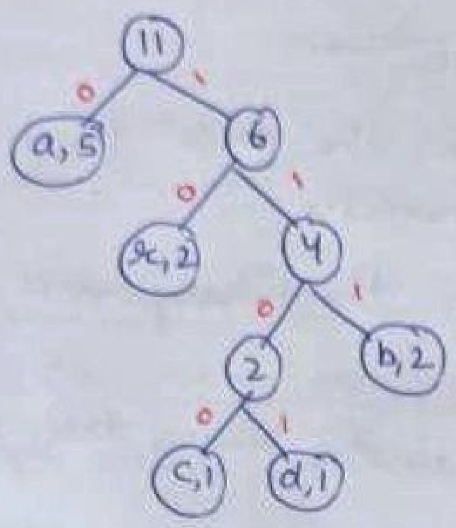


(a,5) (b,c,d,x,6)

(iii)



ⓑ Assign Huffman Code. (Assign '0' to left edge & '1' to right edge).



Character	frequency	code	code length
a	5	0	1
b	2	111	3
c	1	1100	4
d	1	1101	4
r	2	10	2

Encoding →

ⓒ 0 111 10 0 1100 0 1101 0 111 10 0

ⓒ Average code length = Σ(frequency × code length) / Total frequency

$$= \frac{\{(5\times1)+(2\times3)+(1\times4)+(1\times4)+(2\times2)\}}{5+2+1+1+2}$$

$= 2.09$

ⓓ Length of the encoded string

length = total no. of characters × Average

$= 11 \times 2.09$

$=$ 23 bits

Huffman decoding is a technique that converts the encoding data into initial data. Following steps are involved in decoding process:-

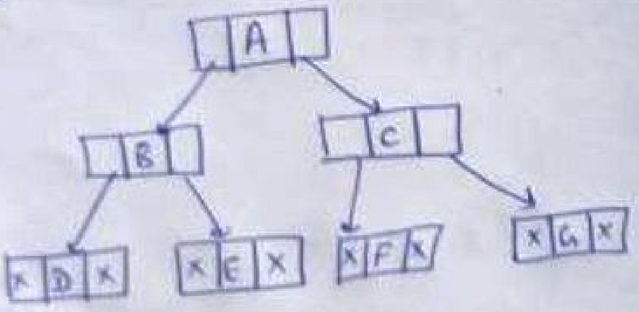
i) Start traversing over the tree from the root node & search for the character.

ii) If we move left in the binary tree, add 0 to the code.

iii) If we move right in the binary tree, add 1 to the code.

Threaded Binary tree :-

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. So, in order to effectively manage the space, a method was devised by Perlis & Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as threaded binary trees.

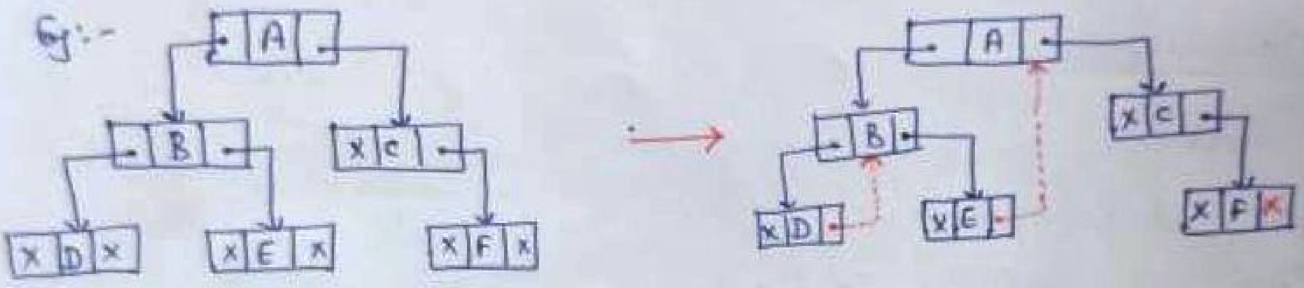


Types of threaded binary tree

- One-way threaded
  - left threaded
  - right threaded
- two-way threaded

i) **One-way threaded** :- In this, thread will appear either in the right or left link field of a node. If it appears in right link then tree is right threaded, if it appears in left link then tree is left threaded

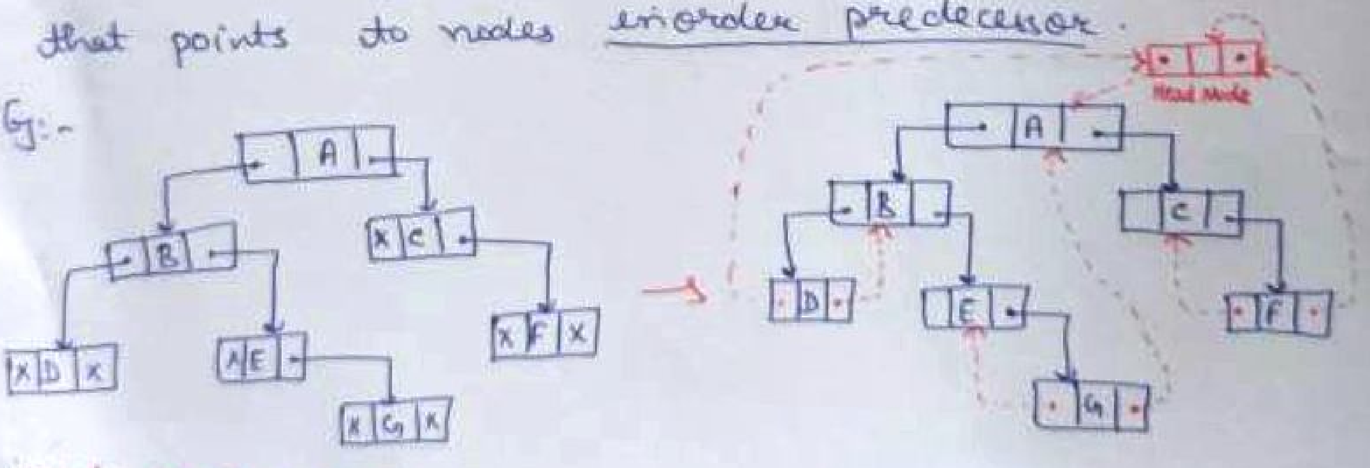
Eg:-



Find inorder first → D B E A C F

ii) **Two-way threaded** :- In this, right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor & left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.

Eg:-



inorder → D B E G A C F

D & F don't have inorder predecessor & successor so we create a special node called header node that doesnot contain any data part & its left link field points to the root node & its right link field points to itself. If this header node is included in two-way threaded binary tree then this node becomes the inorder predecessor of the first node & inorder successor of the last node. Now, threads of left link fields of the first node & right link fields of the last node will point to the header node.

Advantages :-

1) Fast.
2) It is linear.
3) No requirement of stack i.e. it saves lot of memory & time.
4) Efficient

Disadvantages :-

1) Need to maintain the extra information for each node.
2) Insertion & deletion is more time consuming because both threads & ordinary links need to be maintained.