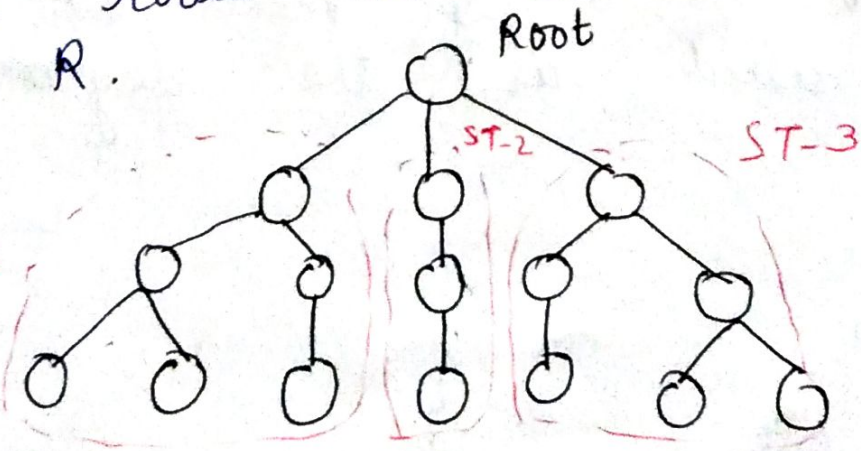
Trees

* A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like arrays, stacks, queues & linked list organize data in linear order.

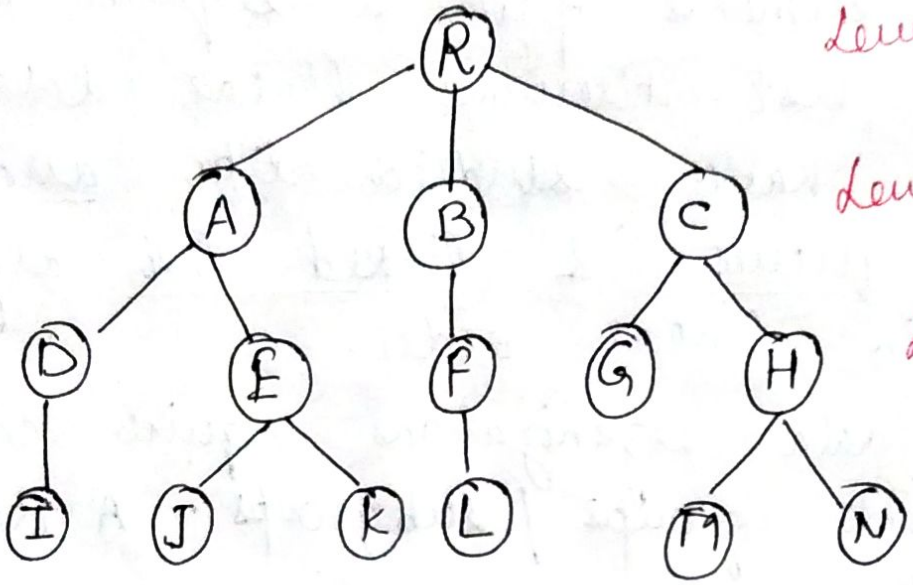
* Some data organizations require categorizing data into groups / subgroups. A data structure is said to be non-linear if its elements form a hierarchical classification where, data items appear at various levels.

Recursive def^{n} of trees :-

* A tree is a collection of nodes. The collection may be empty. Otherwise tree consists of distinguished node R called as the root node & zero or more non-empty subtrees $T_1, T_2 \dots T_n$ each of whose roots are connected by an edge from R.



* If a tree contains n no. of nodes, out of which one is the root node, then there will be (n-1) no. of edges.



Tree Terminologies :-

* ① Siblings :- nodes with same parent are called siblings. e.g.,
A, B, C - Parent R , D, E - Parent A etc.

* ② Degree of node :- The no. of subtrees of a node in a given tree is called as the degree of that node.
e.g., deg(R) = 3 , deg(A) = 2

* ③ Degree of Tree :- The maximum degree of nodes in a given tree is called as the degree of the tree.

④ Leaf node / terminal node / External node :-

* A node whose degree is zero. i.e., node without a child. e.g; I, J, K, L etc.

⑤ Non-leaf / Non-terminal / Internal node :-

* A node whose degree is not zero.

⑥ Path :-

A path is a sequence of consecutive edges from the source node to the destination node.

Path from R to M

R → C → C → H , H → M (Path length = 3)

* There exists only one path from one source to a destination. otherwise it is not a tree.

⑦ Path length :- Path length is determined by the no. of edges beth two nodes.

⑧ Depth of a Node :-

* depth of a node n is the length of the unique path from the root node to the node.

e.g., depth of (I) = 3
depth of E = 2

* In a tree, the depth of the root is always zero (0).

(9) Height of a Node :-

Height of any node n is the length of the longest part from n to a leaf node.

Height of node (C) = 2
" " " (A) = 2

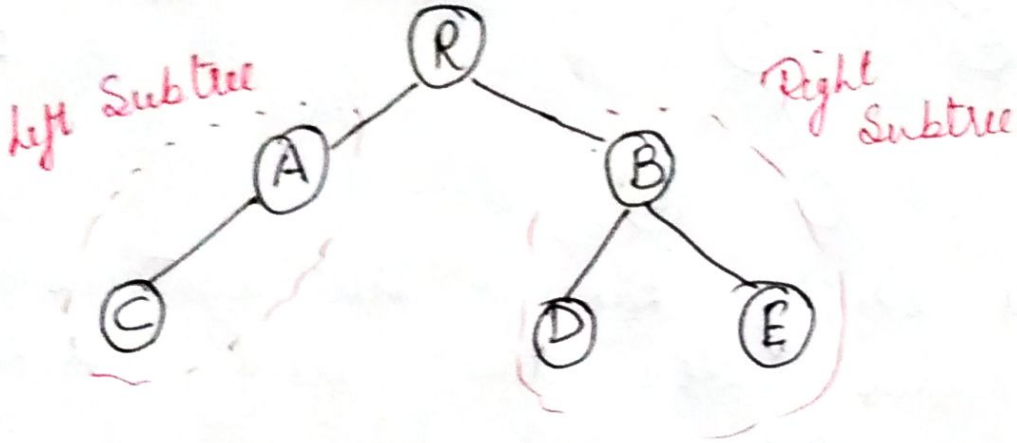
* Height of any leaf node = 0
* Height of a tree is same as height of the root node.
* Height of a tree = Depth of a tree
* Depth of a node = level of the node.

(10) Ancestor & Descendents :-

* if there is a path from node n_1 to node n_2, then n_1 is called as ancestor of n_2 & n_2 is called as the descendent of n_1.

(11) Binary Tree :-

* A Binary tree is a tree in which no node can have more than two children.
* Each node in binary tree contains either zero or 1 or max. two children.



Recursive Def^n for Binary Tree :-

* A binary tree is a tree which is either empty otherwise tree contains a distinguished node called as Root & two non-empty binary Trees called as Left Subtree & Right Subtree.

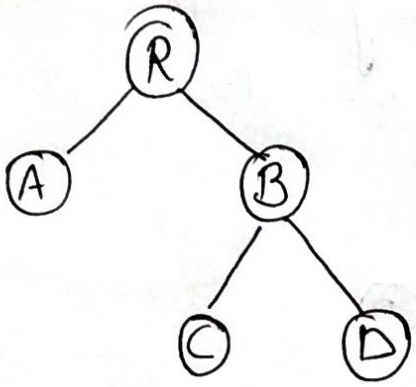
Strictly Binary tree / Extended Binary tree / 2-tree :-

* If every non-leaf node in a Binary tree has non-empty left subtree & right sub tree, then the tree is called as strictly binary tree.

OR

*A Binary tree is called as strictly Binary tree if each node of the tree contains either zero or exactly 2 children.

Eg.,



Leaf nodes = 3

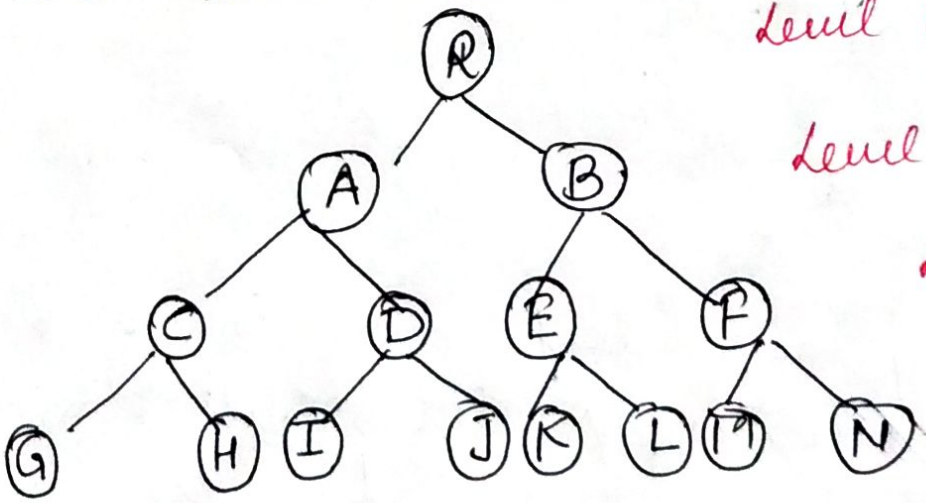
Total nodes = $2 \times 3 - 1$

$= 5$

* If a strictly Binary tree contains n leaf nodes, then the tree will have $(2n-1)$ no. of nodes

Complete Binary tree / Full tree :-

* A complete binary tree is a binary tree in which all internal nodes have degree 2 and all the leaf nodes are present at the same level.



Level 0

Level 1

Level 2

Level 3

* In a complete binary tree at any level i, there will be 2^i no. of nodes.
* In a complete binary tree if there are n no. of nodes at level l, then there will be $2n$ no. of nodes at level $l+1$.

Q. Find out the total no of nodes $t(n)$ of a complete binary tree of depth d.

$$t(n) = 2^0 + 2^1 + 2^2 + \text{---} + 2^d$$

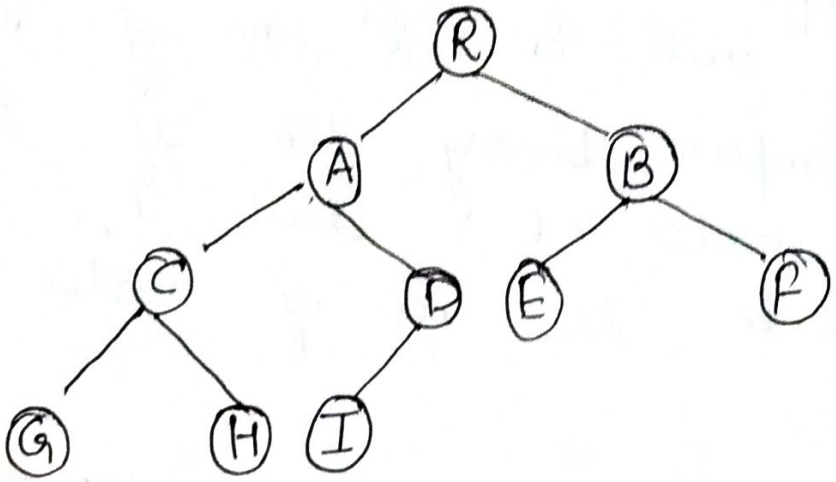
$$\boxed{t(n) = 2^{d+1} - 1}$$

Q If the complete binary tree contains t_n no. of nodes then its depth will be

$$\boxed{d = \log_2 (t_n + 1) - 1}$$

* Almost complete Binary Tree :-

* A binary tree is said to an almost complete B.T. if all its levels except the last level have maximum no. of possible nodes and all the nodes in the last level appear as far left as possible.



Level 0

Level 1

Level 2

Level 3

Q. Find out the minimum & the maximum no. of nodes of an almost complete binary tree the depth d.

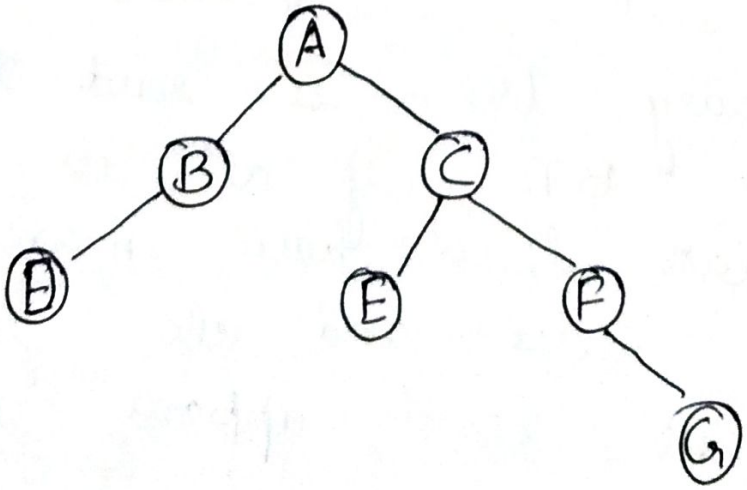
$$\text{Min } t_n = 2^d$$

$$\text{max } t_n = 2^{d+1} - 2$$

Representation of Binary tree in memory:-

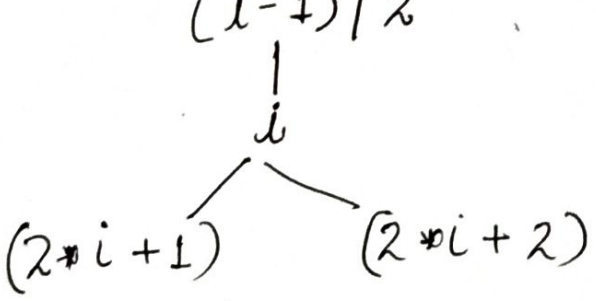
Two types of representation

1) Linear / Sequential representation using array
2) Linked representation using double linked list



(1) Linear / Sequential representation using array

* In sequential representation, if depth of the tree is d, then we need an array of size approximately 2^{d+1}.
* Always the root node of the tree will be stored at location 0.
* If any node is stored at location i, then its left child will be stored at $(2*i+1)$ location and its right child will be stored at $(2*i+2)$ and its parent will be stored at $(i-1)/2$ position.



Here, depth = Height = d = 3

so, array size = $2^{3+1} = 16$

for the above example;

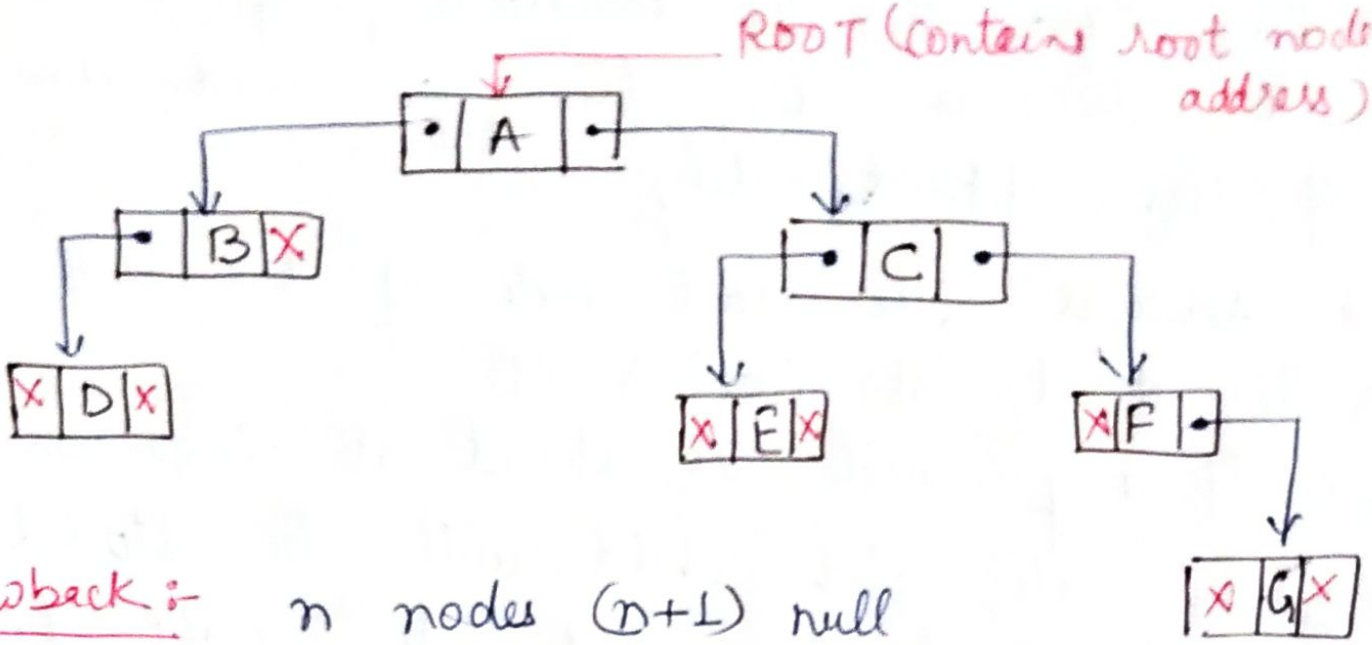
A = 0, B = 2*0+1 = 1, C = 2*0+2 = 2

D = 2*1+1 = 3, E = 2*2+1 = 5, F = 2*2+2 = 6

G = 2*6+2 = 14

A	B	C	D		E	F								G	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

② Linked Representation using double linked list :-



Drawback :- n nodes (n+1) null pointers.

Traversing a Binary Tree

Three standard ways of traversal :-

1) Pre-order traversal
2) In-order ,,
3) Post-order ,,

* Recursive algo. for Pre-order traversal :-

Step 1 :- Process the root node

Step 2 :- Traverse the left subtree in pre-order.

Step 3 :- Traverse the right subtree in preorder.

* Recursive algo. for In-order Traversal :-

Step 1 :- Traverse the left subtree in In-order.

Step 2 :- Process the root node.

Step 3 :- Traverse the right subtree in In-order.

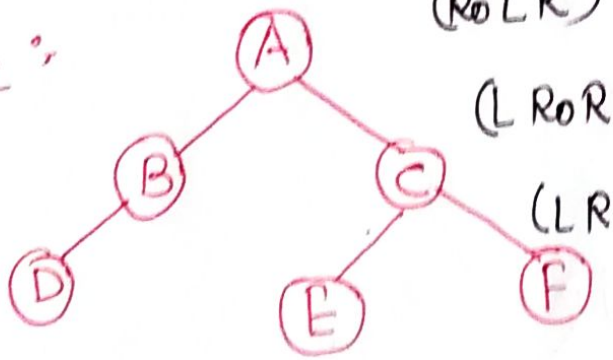
* Recursive algo. for Post-order Traversal :-

Step 1 :- Traverse the left subtree in Post-order.

Step 2 :- Traverse the right subtree in Post-order.

Step 3 :- Process the root node.

Ex 1 :-

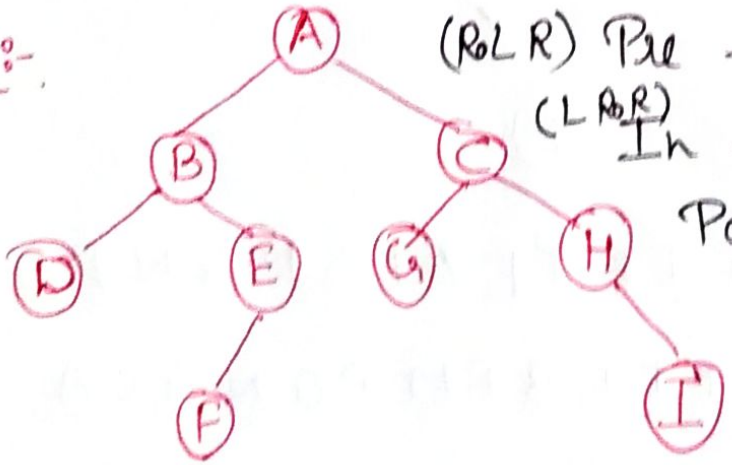


(RoLR) Preorder - A B D C E F

(LRoR) In-order - D B A E C F

(LRR) Post-order - D B E F C A

Ex 2 :-

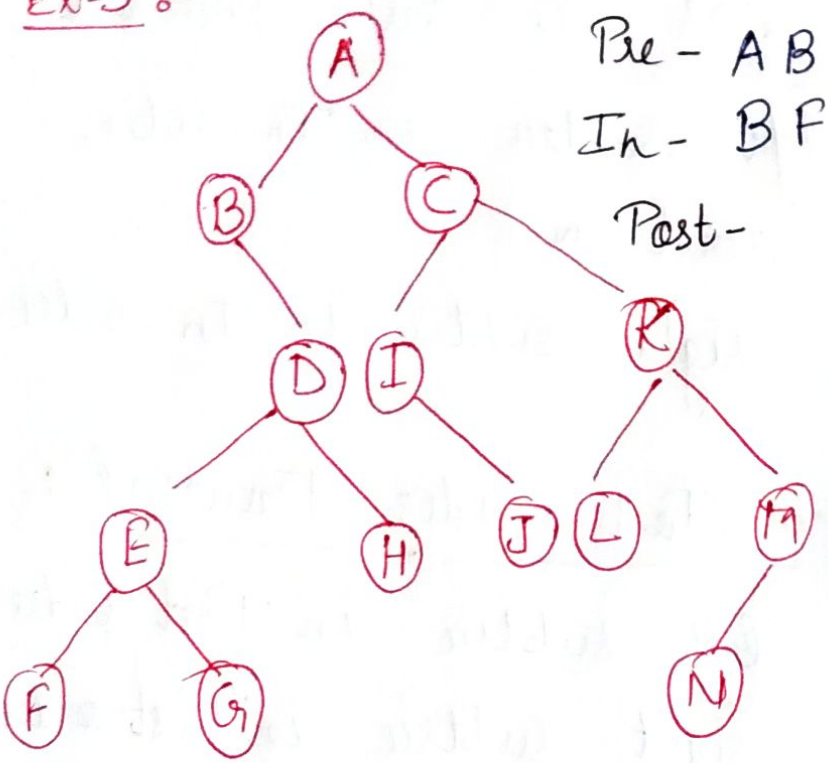


(RoLR) Pre - A B D E F C G H I

(LRoR) In - D B F E A G C H I

Post - D F E B G I H C A

Ex-3 :-

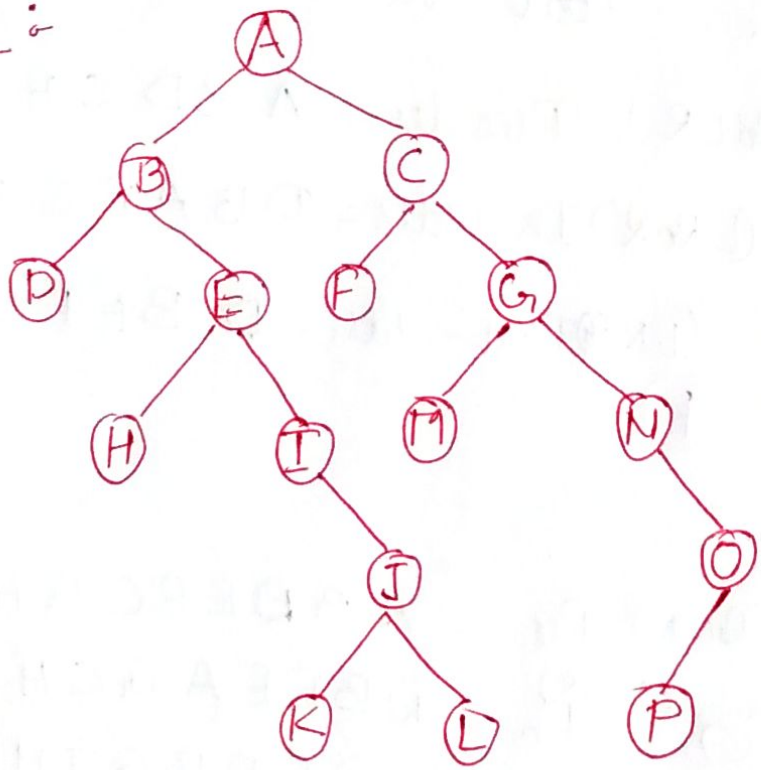


Pre - A B D E F G H C I J K L M N

In - B F E G D H A I J C L K N M

Post -

Ex 4 :-



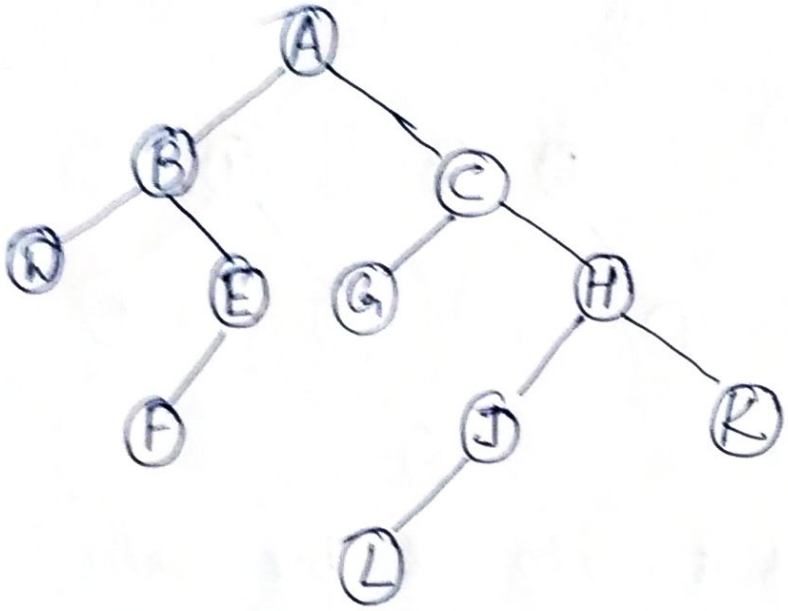
In-order :- D B H E I K J L A F C M G N P O

Post-order :- H K L J I E B F M P O N G C A

Q: Pre-order : A B D E F C G H J L K (RoLR)

In-order : D B F E (A) G C L J H K

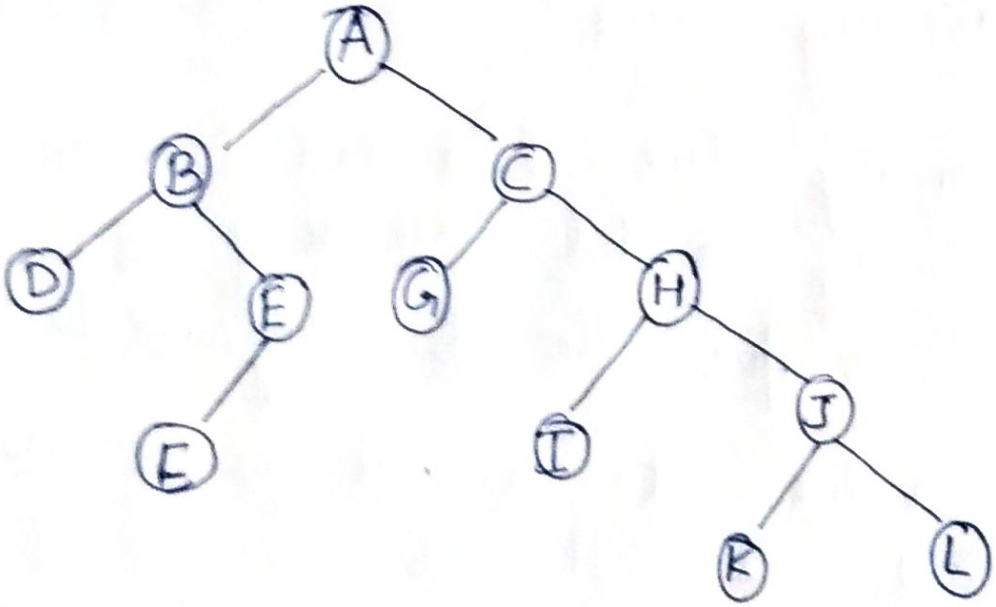
Left — Ro — Right



Q: Pre-order : A B D E F C G H I J K L (RoLR)

In-order : D B F E (A) G C I H K J L

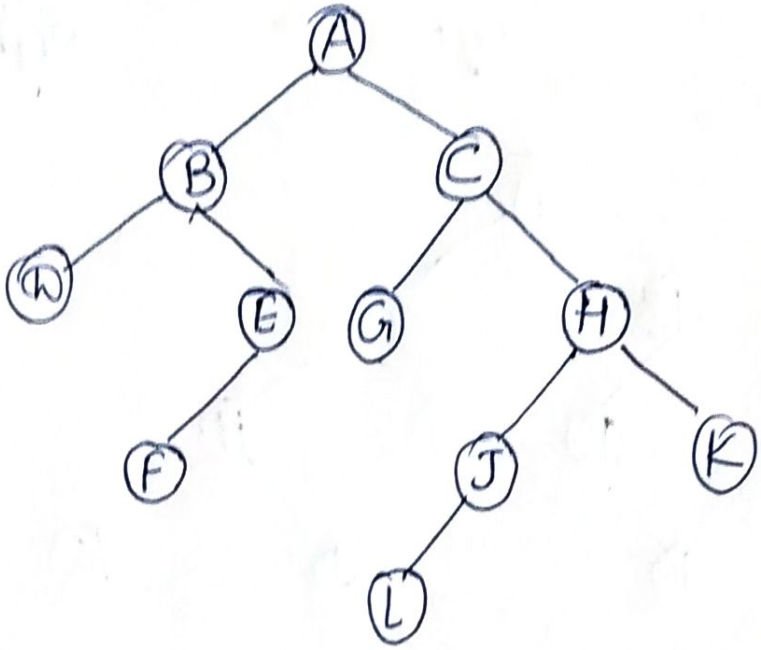
L — Ro — R



Q: Post : D F E B G L J K H C A (L R Ro)

In : D B F E A G C L J H K (L Ro R)

L R



Note :- When post-order & pre-order traversal of a binary tree is given, then the constructed binary tree is not a unique tree.

Binary Search Tree (BST)

* A binary tree T is said to be BST or Binary Sorted Tree if each node n of 'T' satisfies the following property :-

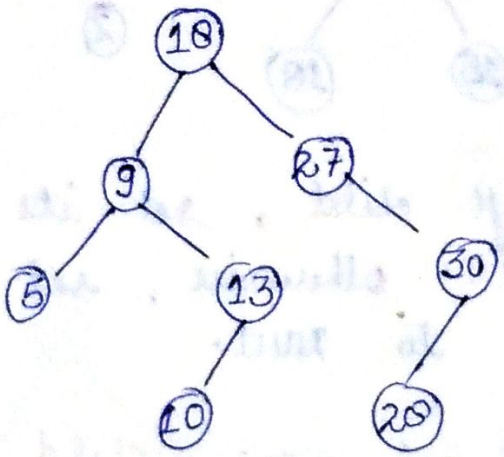
* The value at n is greater than every value of its left sub-tree & is less than or equal to every value of its right subtree.

Binary Search Trees

* A Binary Tree is a B.S.T. if every node x, it fulfills the following :-

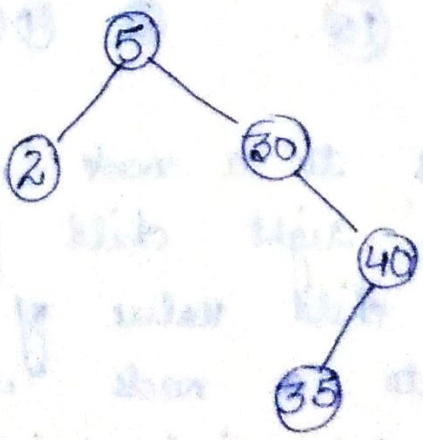
(i) the value of left-subtree of 'x' is less then the value at 'x'.

(ii) the value of right-subtree of 'x' is greater than the value at 'x'.



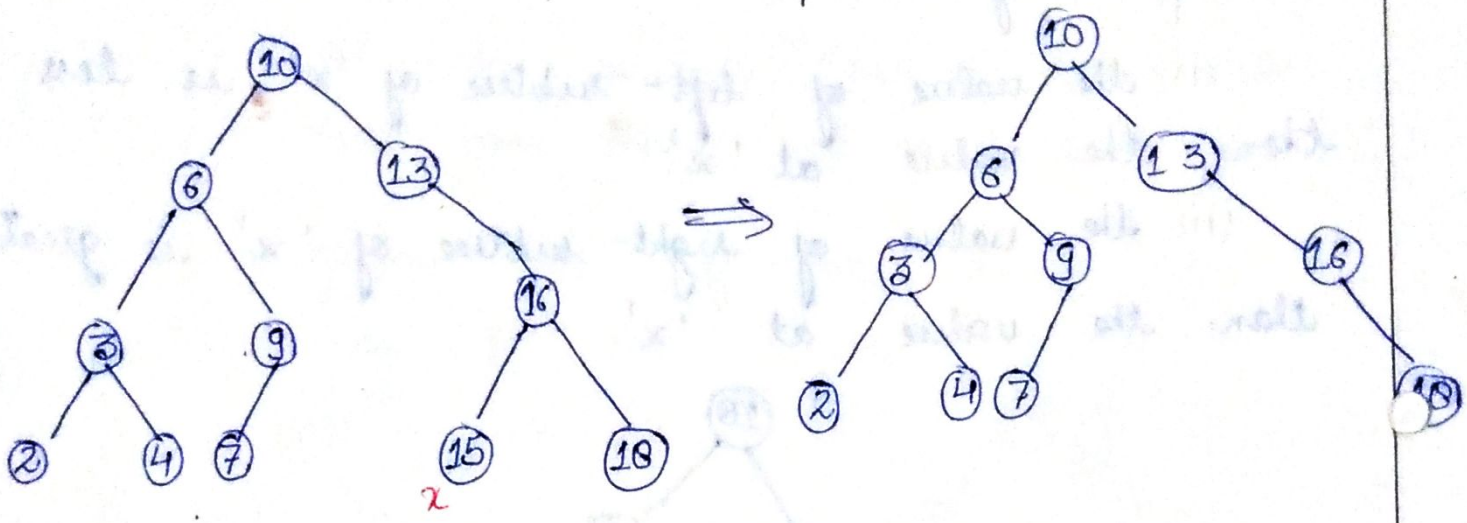
Insertion in B.S.T. :- Suppose T is an empty B.S.T. & now we have to insert the following data items.

5, 30, 2, 40, 35



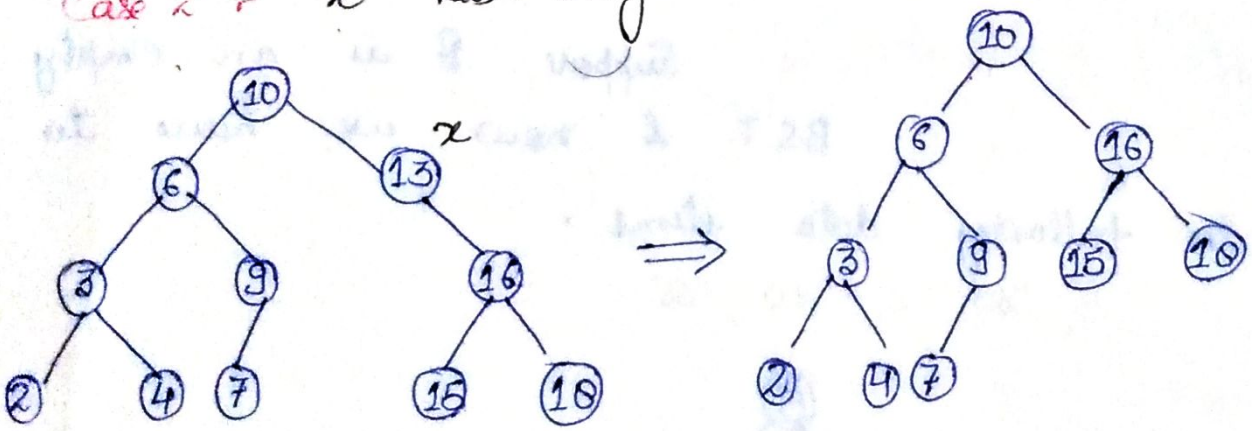
Deletion in B.S.T. :- There are 3 cases that arises, when we delete 'x' from B.S.T.

Case 1 :- 'x' is a leaf node.



* If 'x' is left child, set its parent's left pointer to NULL otherwise, set its parent's right pointer to NULL.

Case 2 :- 'x' has only one child

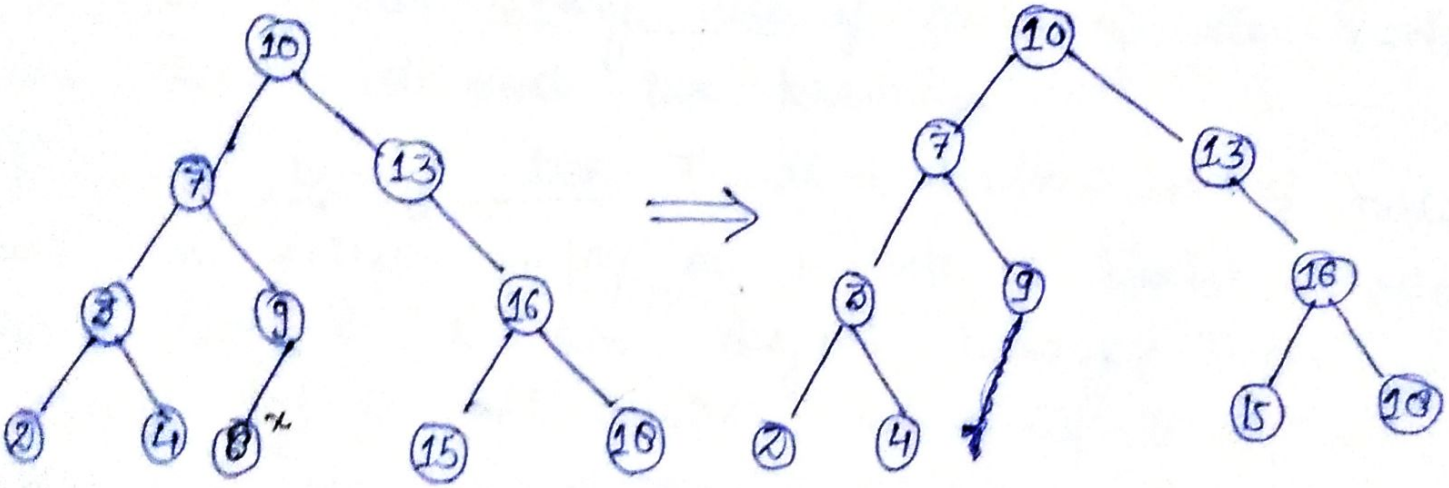


* If 'x' points to a node that has a right child then 'x' right child pointer is assigned to the right child value of its parent but if 'x' points to a node that has a left child then 'x' left child pointer value is assigned to the left child value of its parent.

Case 3 :- 'x' has both the children.

* (i) Find Successor of 'x'.
  (ii) Interchange the values of 'x' with its successor.
  (iii) Delete the Successor. (case 2:- As Successor of a node can have at most a single child.

For the given tree if we delete '6', its successor is '7'.



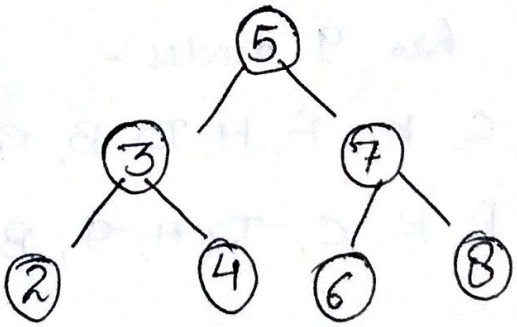
Resulting Tree

* Search trees are data structure that support many dynamic-set operations, including Search, Min, Max, Predecessor, Successor, Insert & Delete.

* A B.S.T. is a binary tree which is represent by a linked data structure. Each node contains Info, pointers to left, right & parent (p).

* The Info in BST are always stored in such a way as to satisfy the BST property :

  * the value at every node N is greater than every value in the left subtree of N & is less than every value in the right subtree of N.



Operations on B.S.T. :

1) Searching :

* In the following procedure given is a pointer to the root of the tree & a key k.

```
Tree-Search (x, k)
1. if x = NIL or k = Info[x]
2.     then return x
3. if k < Key[x]
4.     then return (Tree-Search (left[x], k))
5.     else return (Tree-Search (right[x], k))
```

2) Minimum & Maximum :

```
Tree-Minimum(x)
1. while (left[x] ≠ NULL)
2.     do x = left[x]
3. return x
```

```
Tree-Maximum (x)
1. while right[x] ≠ NULL
2.     do x = right[x]
3. return x
```

Insertion & Deletion :

③ Insertion

To insert a new value v into a BST T, we use the procedure Tree-Insert : The procedure is passed a node z for which key[z] = v, left[z] = right[z] = NULL. It modifies T in such a way that z is inserted into an appropriate position in the tree.

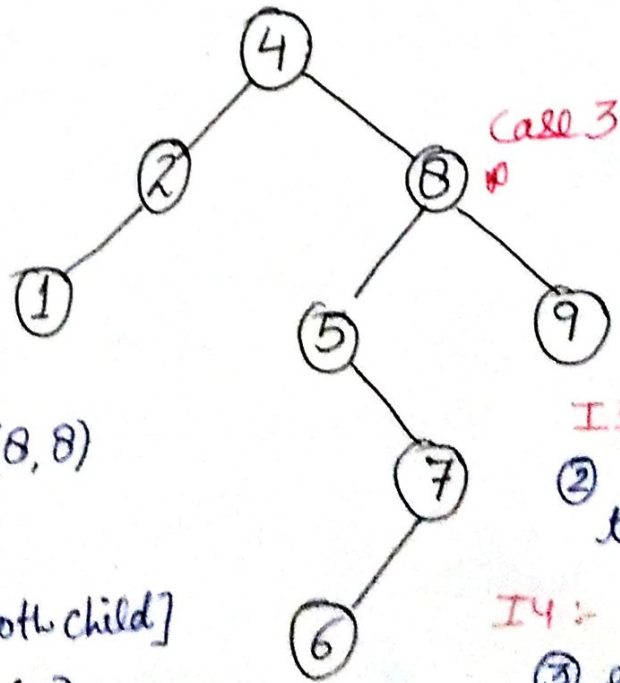
C- implementation for BST :- (using linked list)

```
struct node {
    struct node * left;
    int info;
    struct node * right;
}
typedef struct node * bstnode;

bstnode insert ( bstnode t, int x)
{
    if ( t == NULL)
    {
    t = (bstnode) malloc (sizeof(struct node));
      if (t == NULL)
      { printf ("Out of memory");
        return t;
      }
      else
      {
        t -> info = x
        t -> left = t -> right = NULL;
      }
    }
    else
    {
        if (x < t -> info)
            t -> left = insert(t -> left, x);
```

```
    else
    {
        if (x > t -> info)
            t -> right = insert (t -> right, x);
        }
    }
    return t;
}

bstnode delete (bstnode t, int x)
{
    bstnode temp;
    if (t == NULL)
        printf("BST is empty");
    else
    {
    ① if (x < t -> info)
            t -> left = delete (t -> left, x);
      ① else
        {
        ② if (x > t -> info)
            t -> right = delete (t -> right, x);
        ② else // element found
          {
          ③ if (t -> left && t -> right) // both children
              {
                temp = findmax (t -> left);
                t -> info = temp -> info;
              }  t -> left = delete (t -> left, t -> info);
```

```
③ else      // one or no child
  {
     temp = t ;
     ④ if (t -> left == NULL)
            t = t -> right ;
     ④ else
            if ( t -> right == NULL)
                  t = t -> left ;
            free (temp);
        }
  } }
  return t ;
} // end of delete ()
```

Example :-



I1:- delete (4, 8)
4<8 ; t -> right = delete(8, 8)

I2:- delete (8,8)
② else ③ if [as both child]
temp = add (7)
t -> data = 7
t -> left = delete (5, 7)

I3:- delete (5, 7)
② else
t -> right = delete(7, 7)

I4:- delete (7, 7)
③ else
temp = 7
④ else t = t -> left (6)

⑤ Successor & Predecessor :-

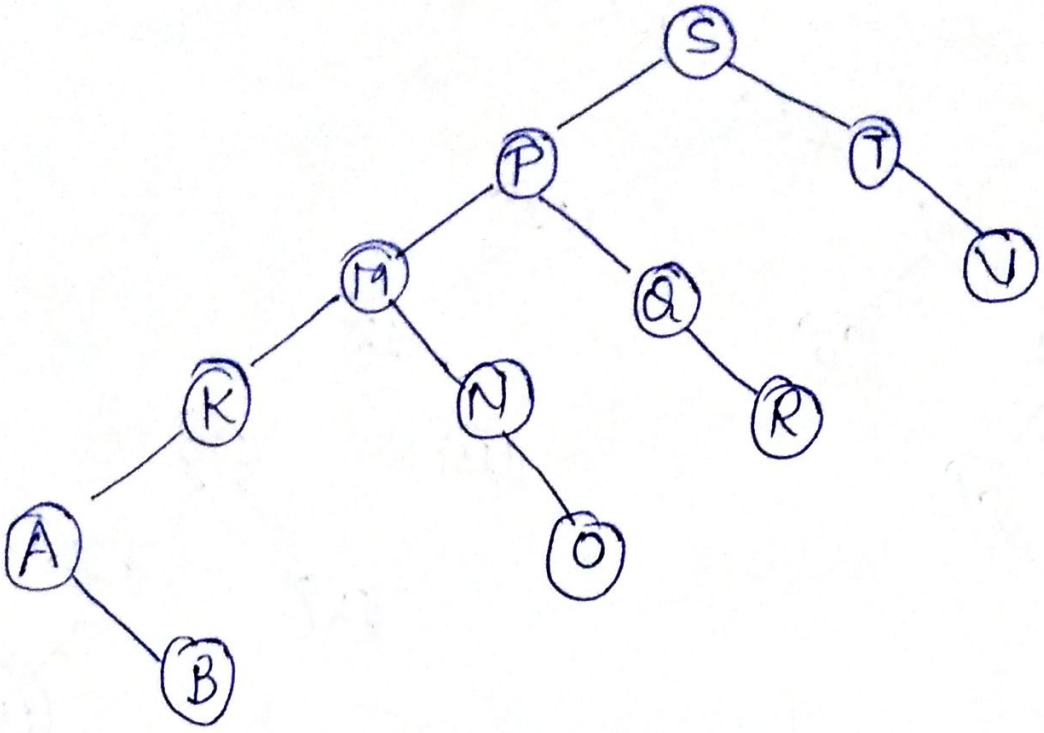
* Given a node in BST, it is sometimes important to be able to find its successor in the sorted order determined by an inorder walk.
* If all the keys are distinct, the successor of a node x is the node with smallest key greater than key [x].

```
Tree - Successor (x)
1.  if right [x] ≠ NULL
    then return (Tree - Minimum (right [x]))
2.
3.      y = p [x]
4.      while y ≠ NULL & x = right [y]
5.          do x = y
6.             y = p [y]
7.      return y
```

(lines 4–6: while-loop)

Q. Suppose the following list of letters is inserted in an empty BST.

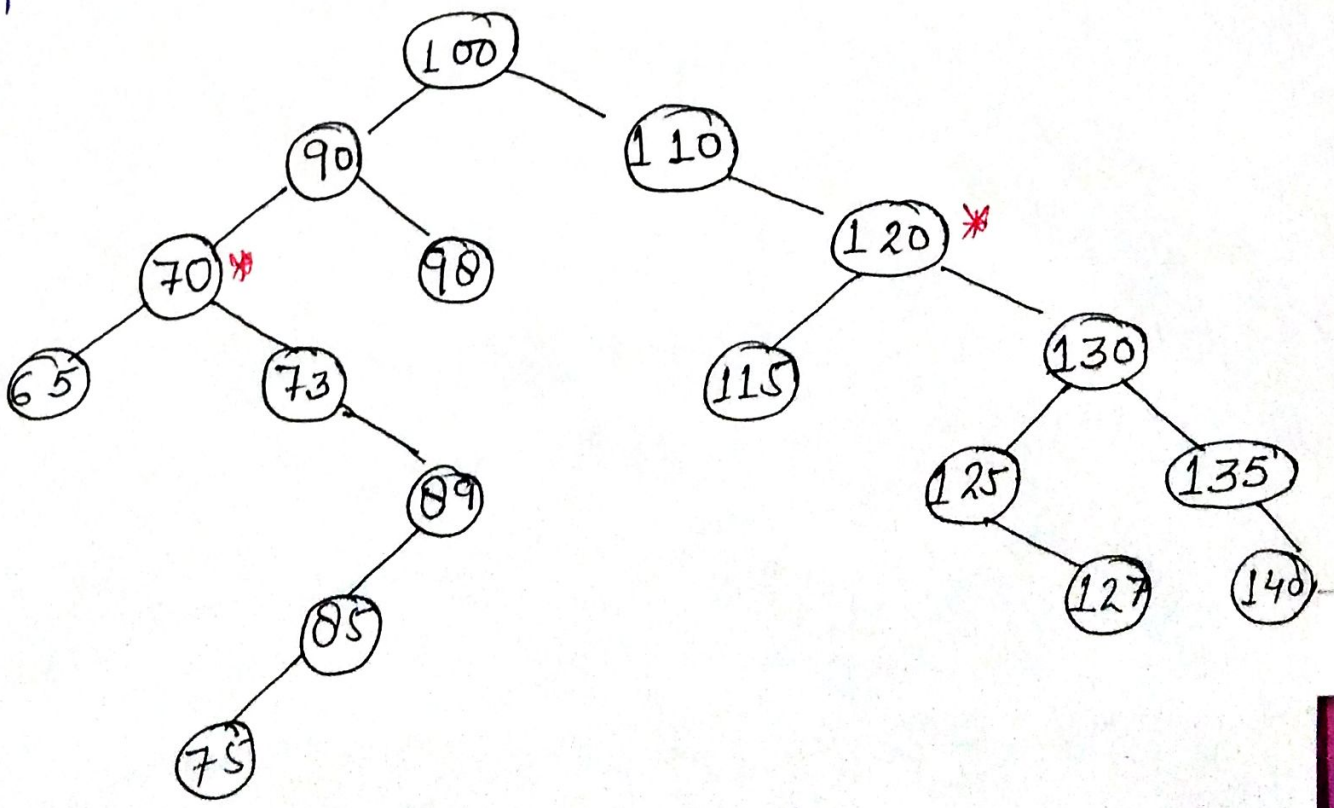
S, T, P, Q, M, N, O, R, K, V, A, B



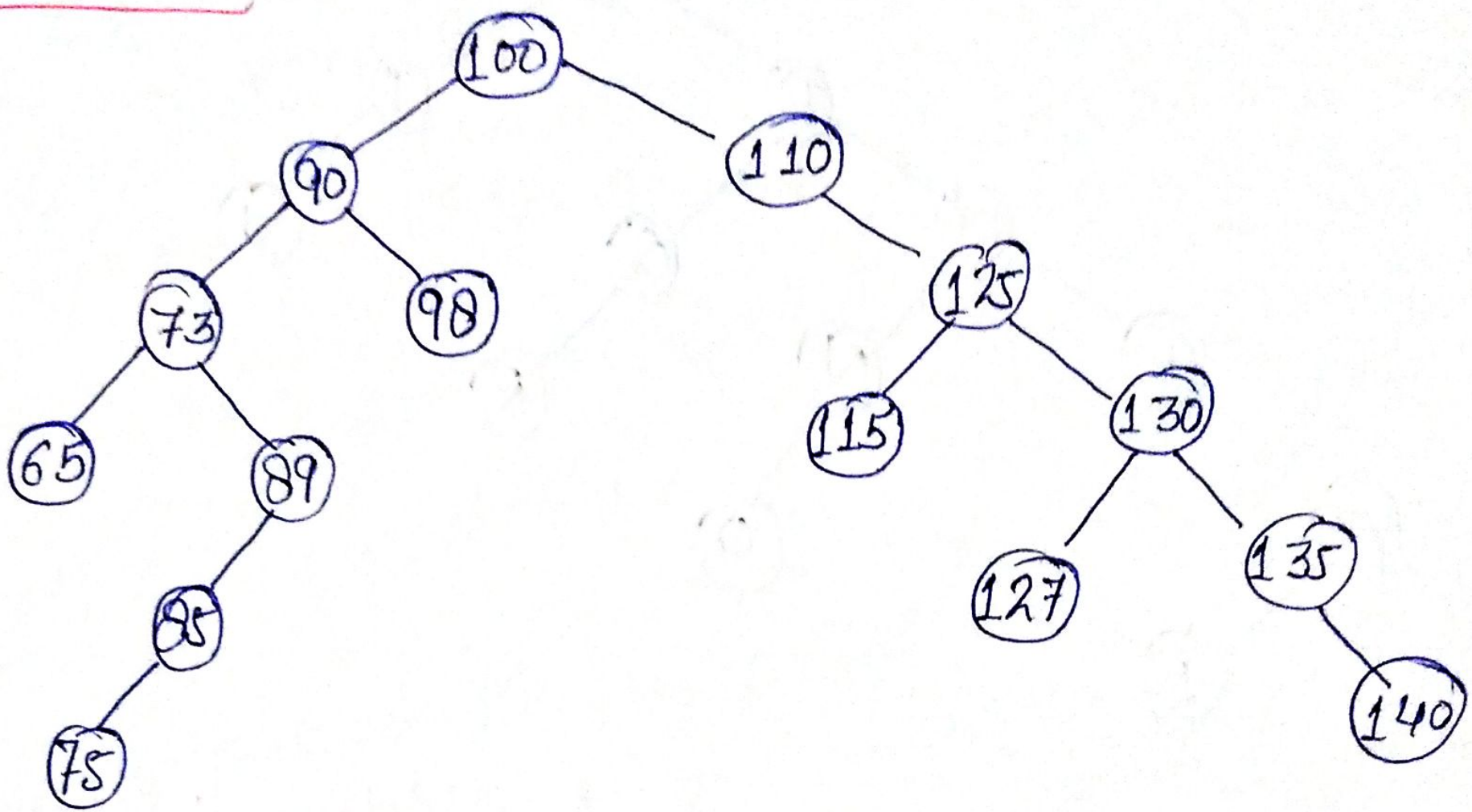
Q construct a BST with the following elements :-

100, 90, 110, 120, 70, 65, 73, 89, 130, 125, 135, 127, 140, 98, 85, 75, 115

then delete :- 120 & 70 from constructed BST



Resultant tree is :-



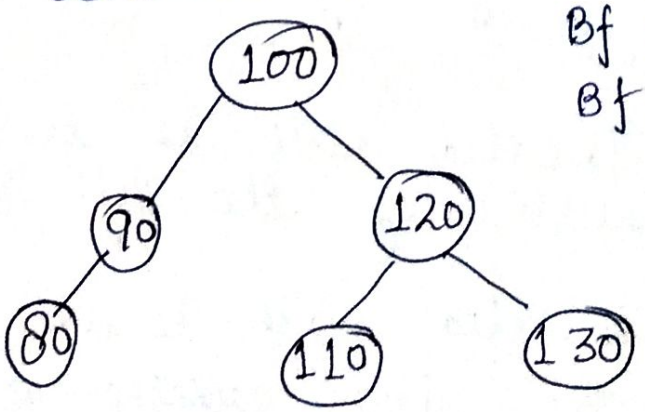
Height - Balanced Tree / AVL Trees :-

* One of the most popular balanced trees was introduced in 1962 by Adelson-Velskii and Landis and was known as AVL trees

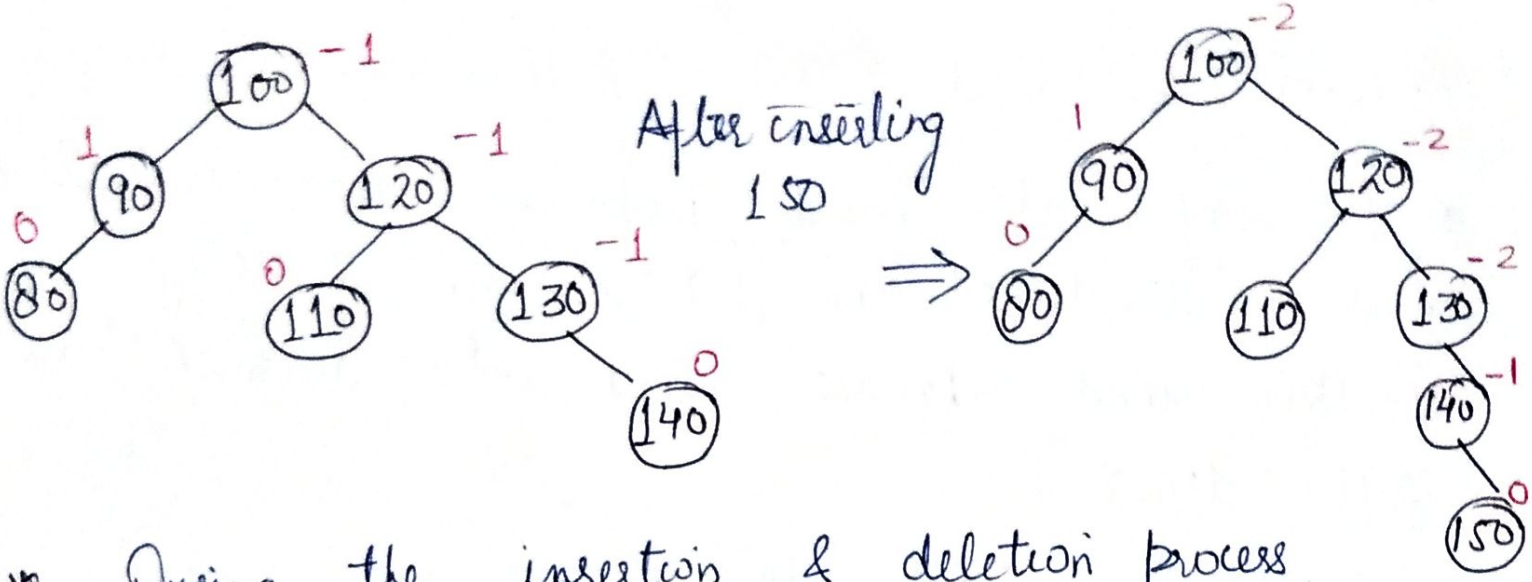
Definition :- Let T be a non-empty binary tree. T_L and T_R be the left subtree and right subtree of T. $H(T_L)$ and $H(T_R)$ be the height of left & right subtree respectively, then T will be called as an AVL Tree if,

$$|H(T_L) - H(T_R)| \leq 1$$

* $H(T_L) - H(T_R)$ is called as the Balance factor.
* If an AVL tree, every node balance factor is either 0, 1 or -1.
* An AVL tree is also a BST but with a balanced condition.



Bf (100) = 2-2 = 0
Bf (90) = 1-0 = 1



* During the insertion & deletion process, if some node's balance factor will be changed to 2 or -2 from 0, 1 or -1 then the AVL tree becomes unbalanced.

Rotation :- It is a technique to convert any unbalanced AVL tree to balanced AVL tree.

There are 4 types of rotation :-

(1) LL :- Inserted node is in the left subtree of left subtree of the reference node.

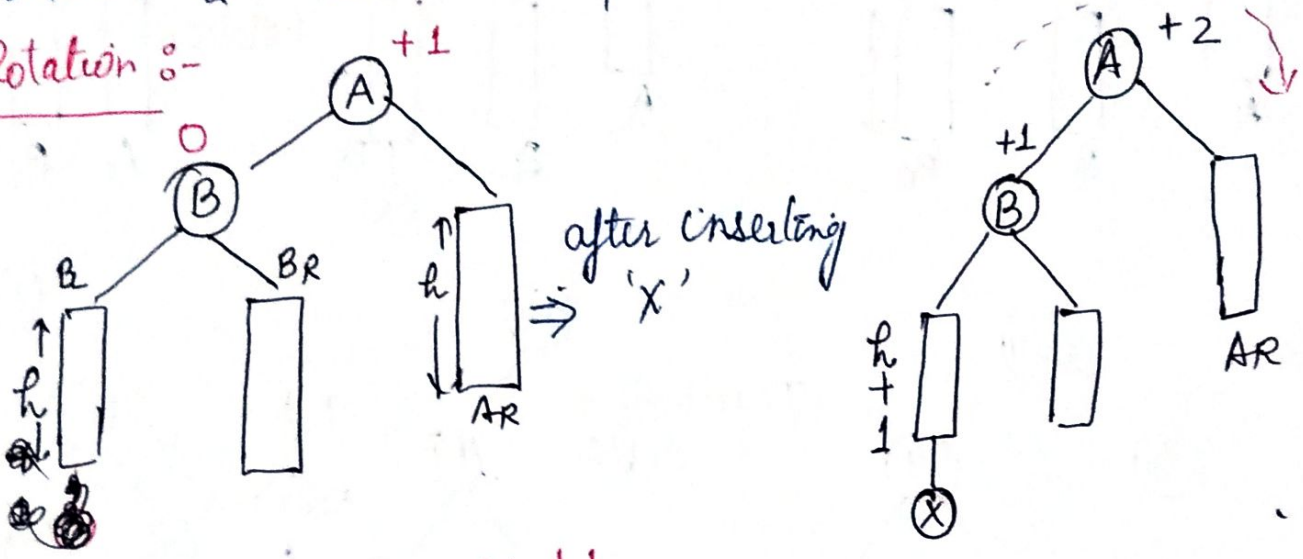
(2) RR :- Inserted node is in the right subtree of right subtree of the reference node

(3) LR :- Inserted node is in the right subtree of the left subtree of reference node.

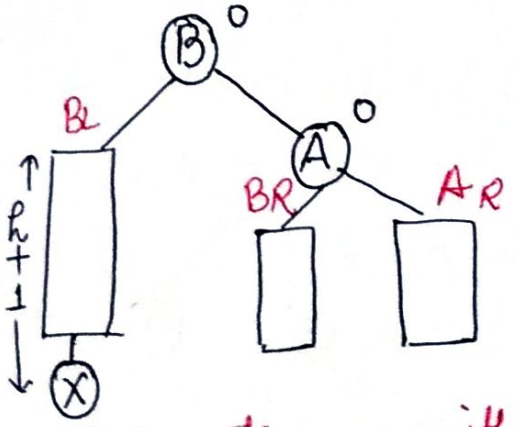
(4) RL :- Inserted node is in the left subtree of the right subtree of reference node.

Reference Node :- is a node whose balance factor has been changed from 0, 1 or -1 to 2 or -2 and which is nearest to the insertion & deletion point.

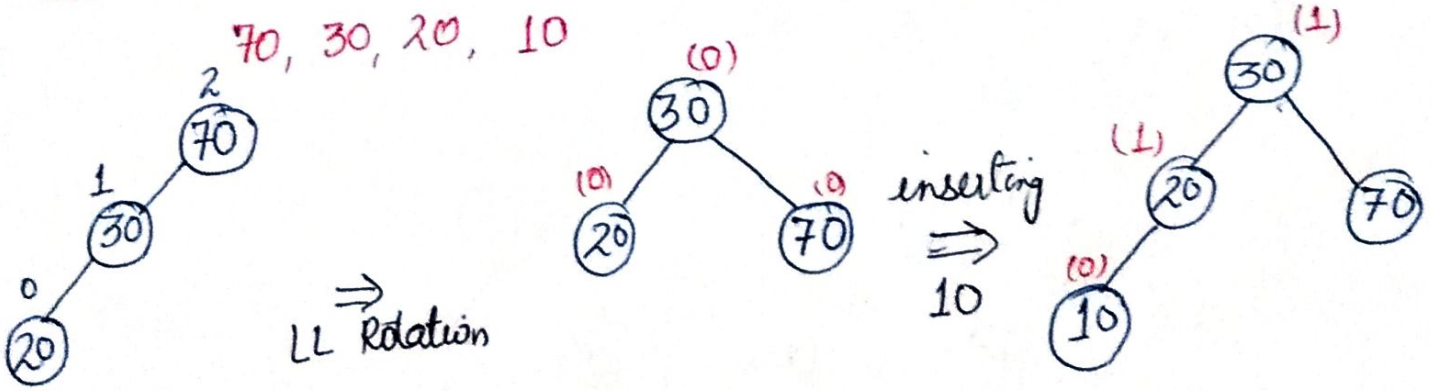
① L L Rotation :-



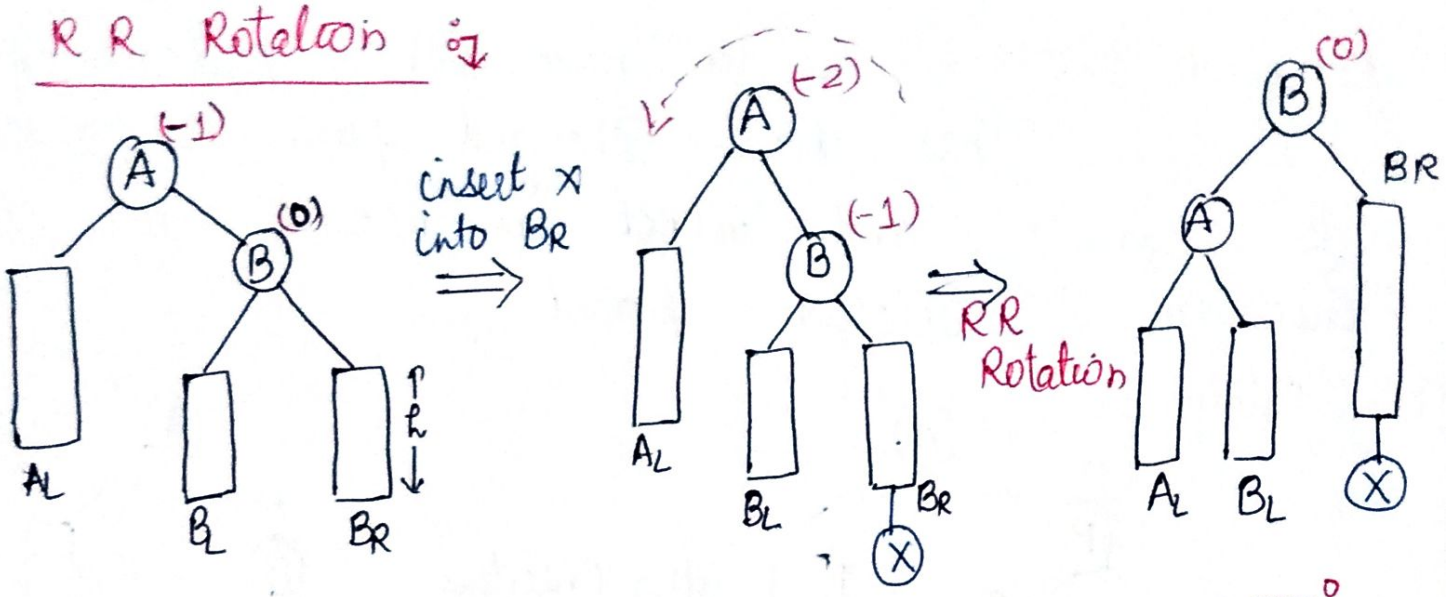
LL Rotation



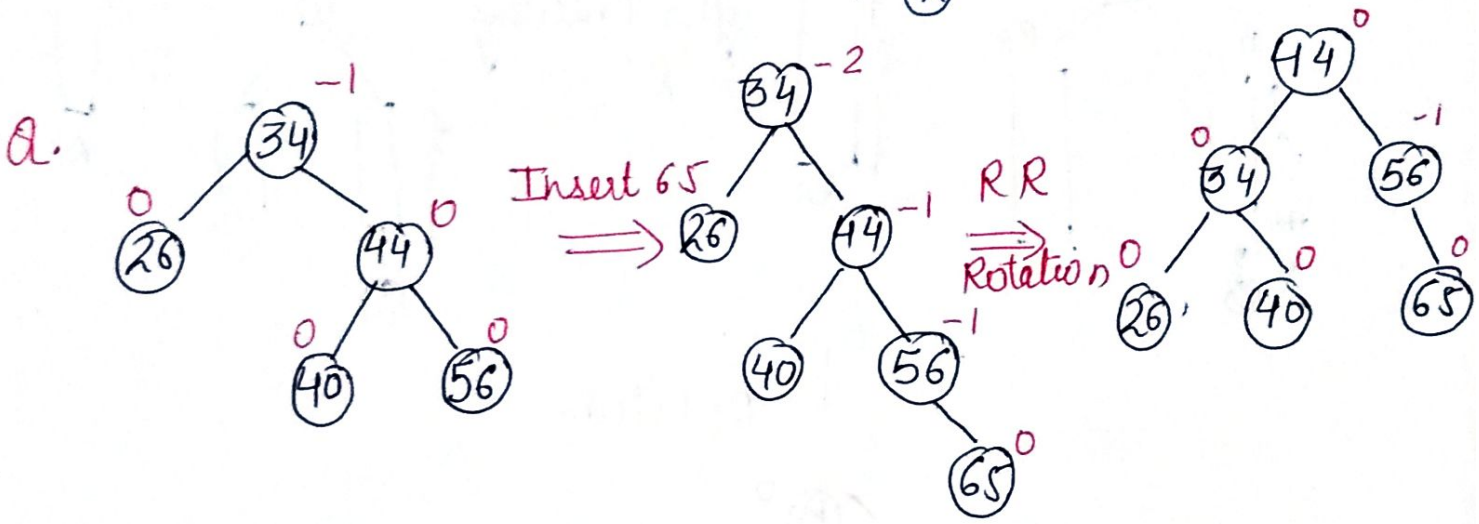
Q1. Create an AVL tree with the following Keys-
70, 30, 20, 10



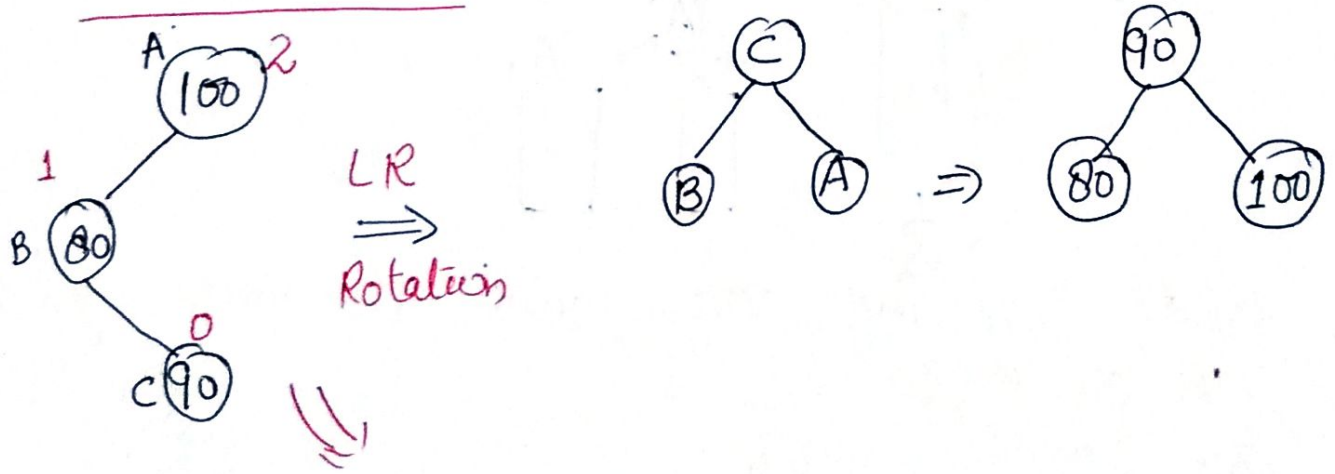
② RR Rotation :-



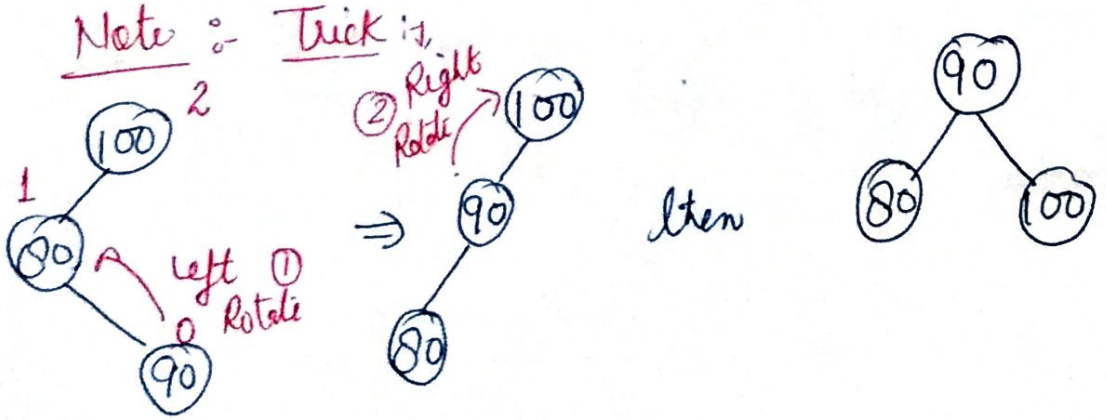
Q.



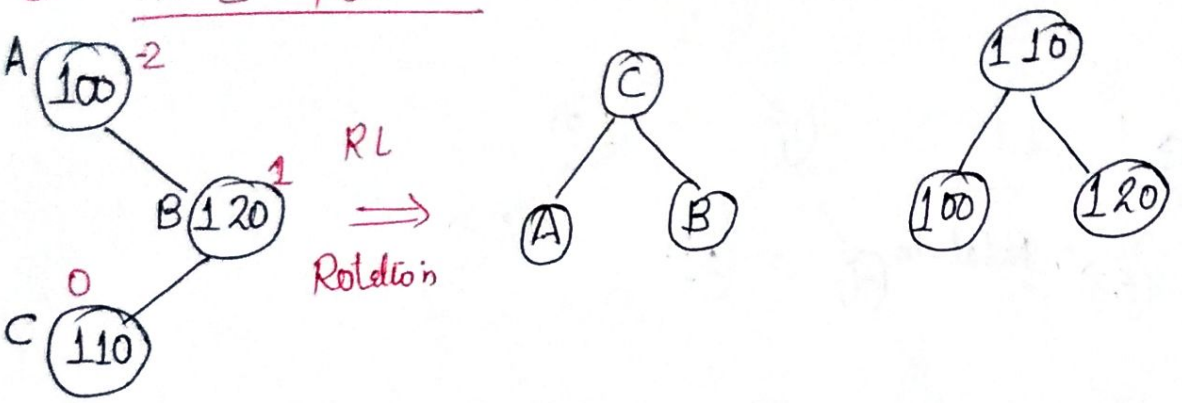
③ LR Rotation :-



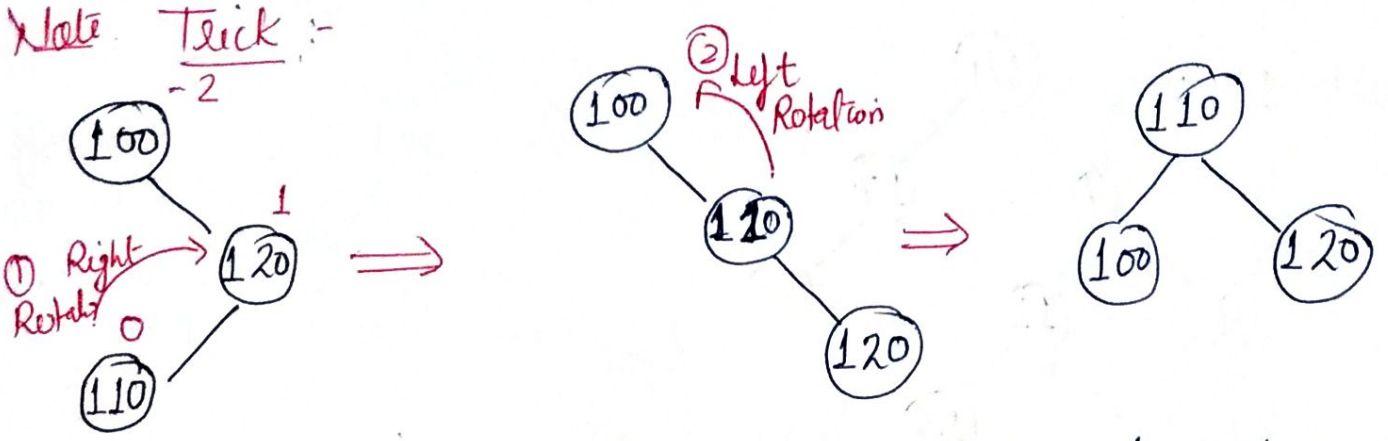
Note :- Trick :-



(4) R L Rotation :-



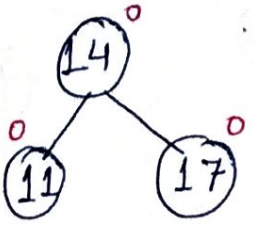
Note Trick :-



Question : construct an AVL search tree by inserting the following elements in the order of their occurrence

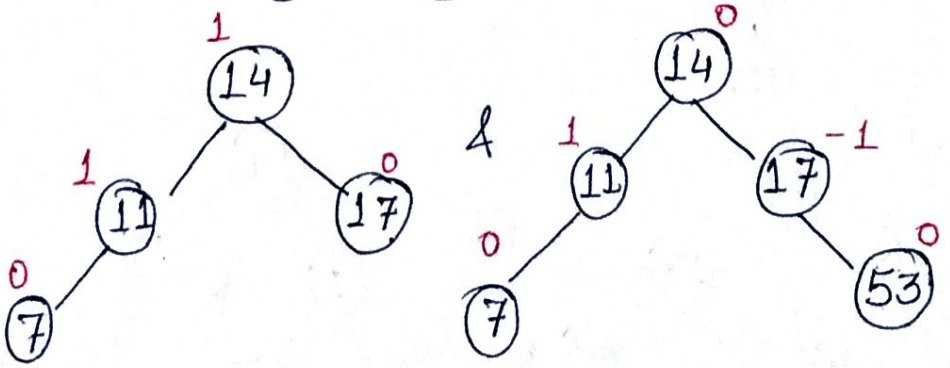
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

Insert 14, 17, 11

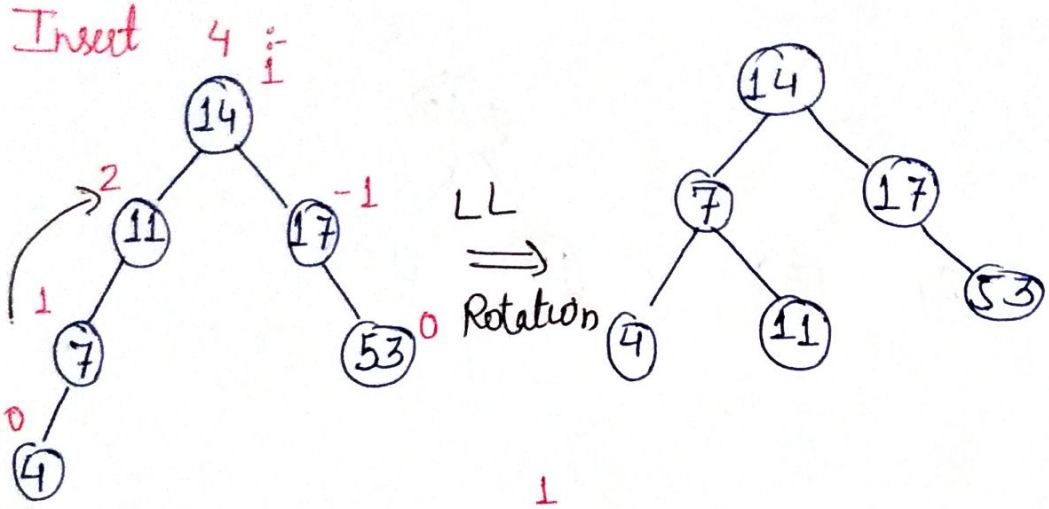


(4)

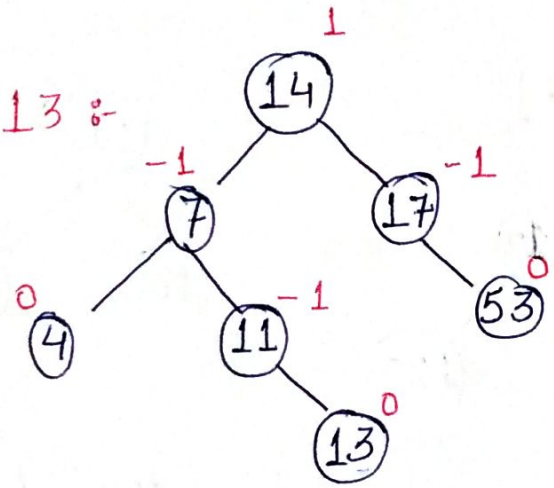
Insert 7, 53



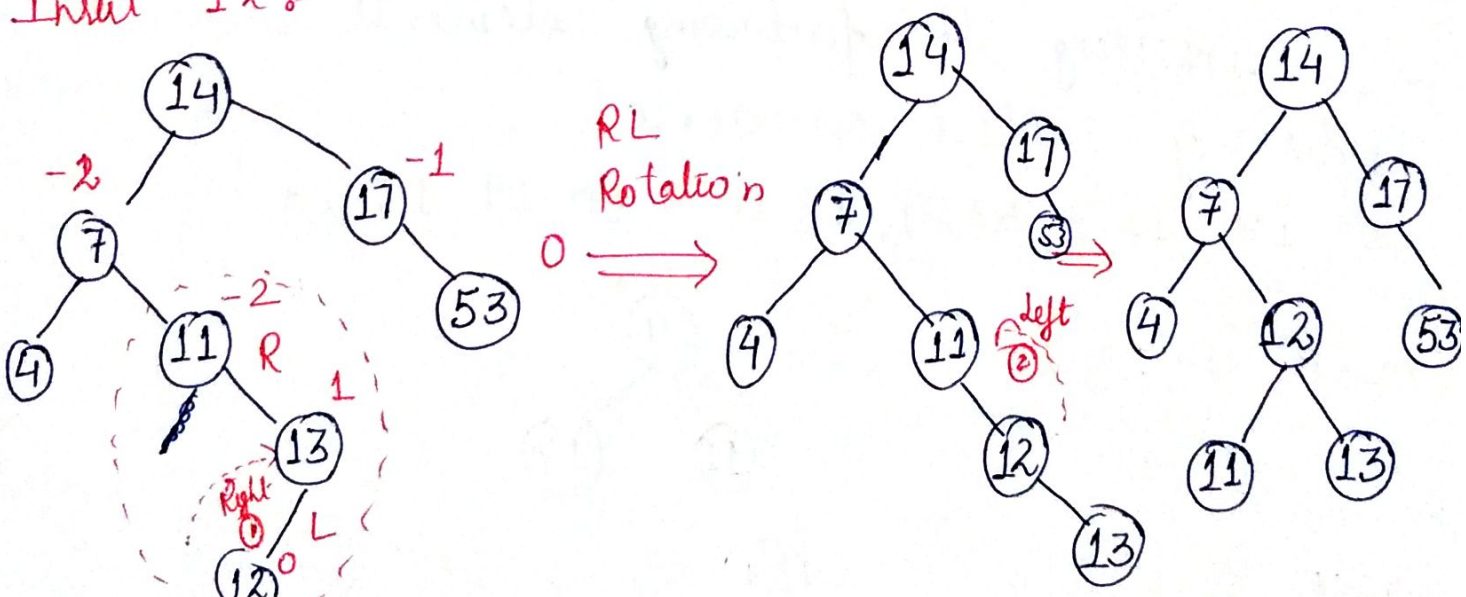
Insert 4 :-



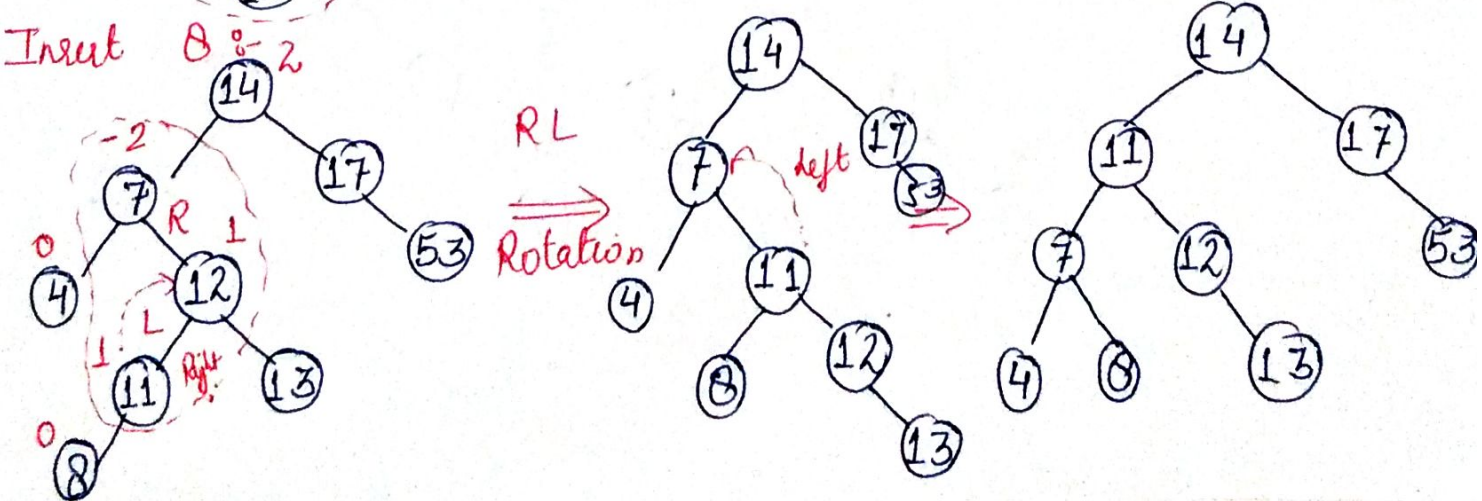
Insert 13 :-



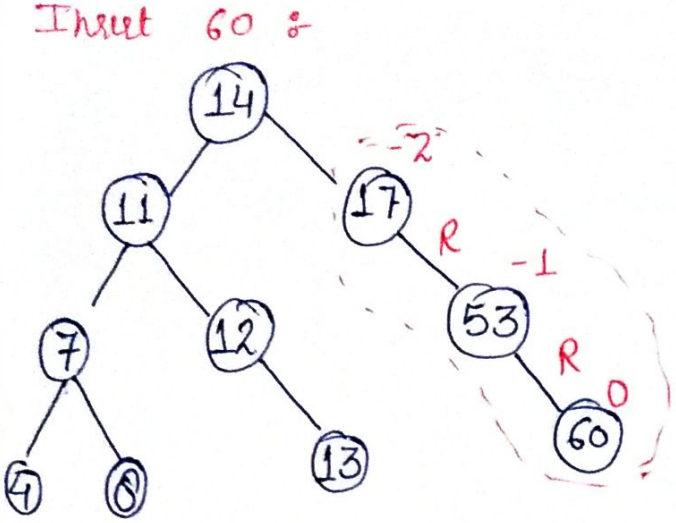
Insert 12 :-



Insert 8 :-

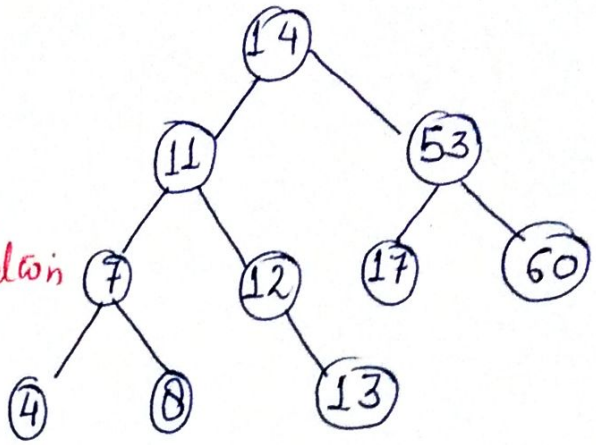


Insert 60 :-

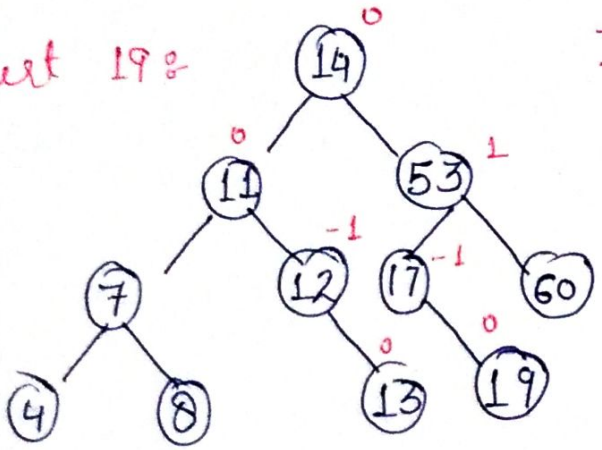


RR

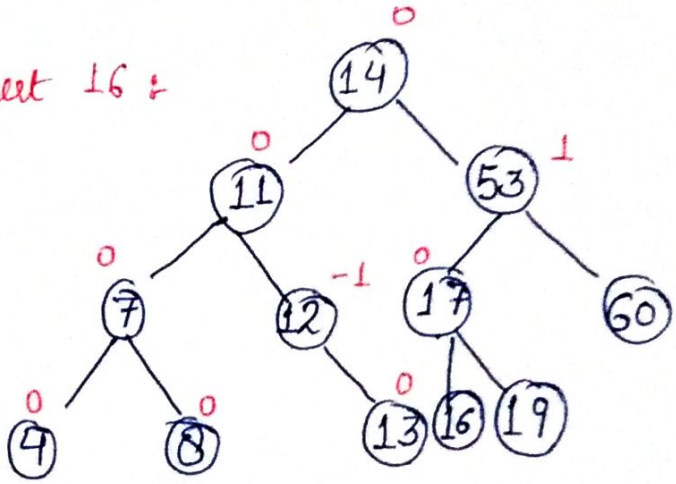
⇒ Rotation



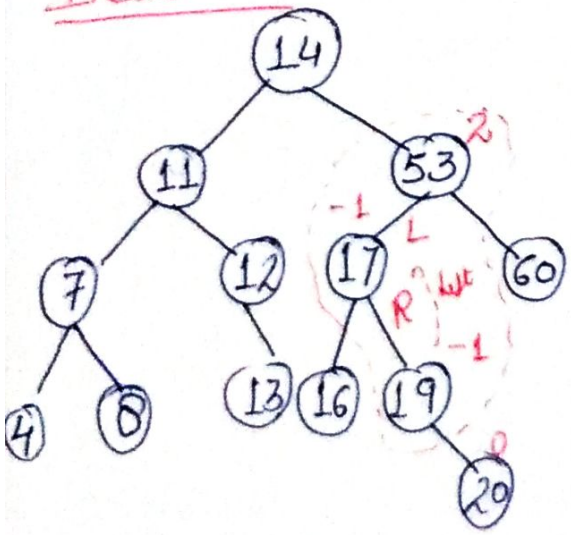
Insert 19 :-



Insert 16 :-

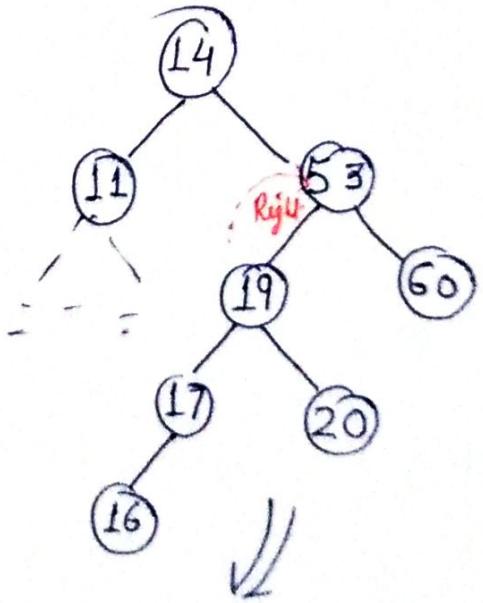


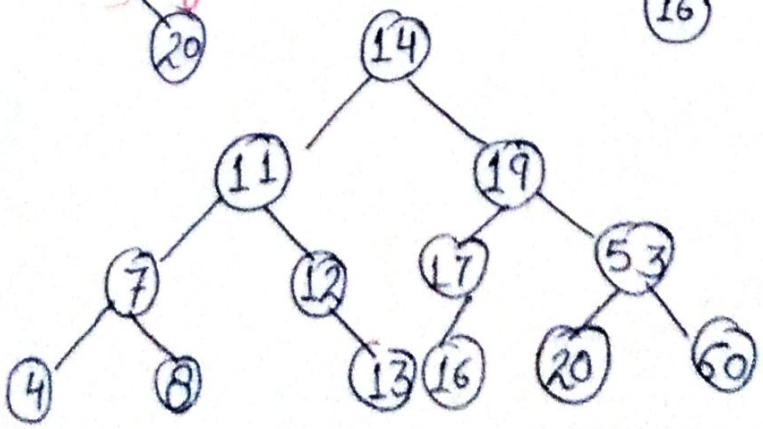
Insert 20 :-



LR

⇒ Rotation





Final AVL tree

Q :- 64, 1, 44, 26, 13, 110, 98, 85

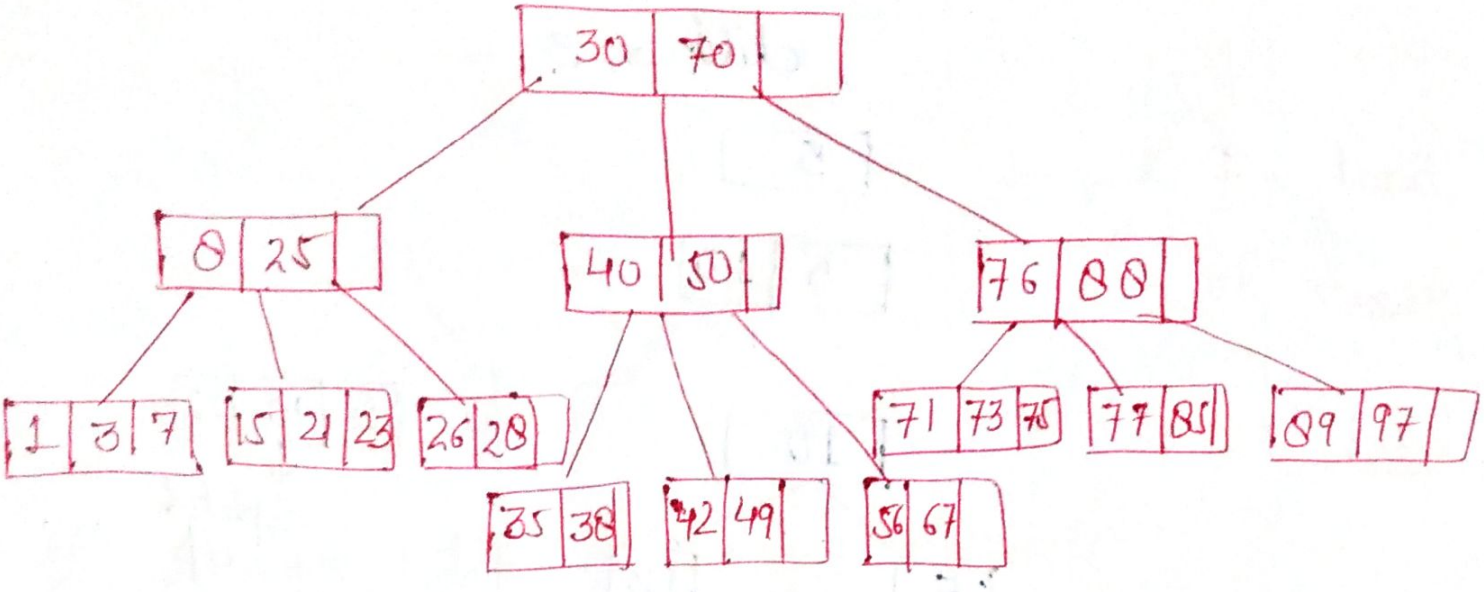
B-Tree & B+-Trees

* B-Trees are specialized m-way search tree. This can be widely used for disc access.
* A B-tree of order m can have max. (m-1) keys and m children.
* B-Tree is a special type of search trees in which a node contains more than one value (key) & more than two children.

B-tree of order 'm' has following properties:-

1) All leaf nodes must be at same level.
2) All nodes except root must have at least $(\lceil \frac{m}{2} \rceil - 1)$ keys and max. of (m-1) keys.
3) All non-leaf nodes except root must have at least $m/2$ children.
4) Root node must have at least two children
5) A non-leaf node with (n-1) keys must have n no. of children.
6) All the key values in a node must be in ascending order

for example :- B- tree of order 4 contains a maximum of 3 key values in a node and max. of 4 children for a node.



Note :

		max	min
Root	child	m	0
	data / key	m-1	1
Internal node	child	m	$\lceil m/2 \rceil$
	key	m-1	$\lceil m/2 \rceil - 1$
Leaf node	child	0	0
	key		

Q Consider the following elements

5, 10, 12, 13, 14, 1, 2, 3, 4

insert them into empty b- tree of order 3.

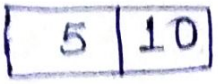
$m = 3$

data → 3-1 = 2

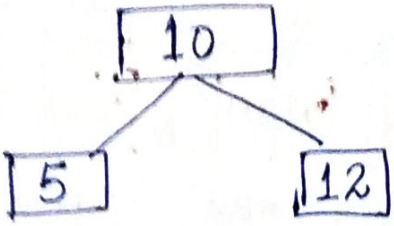
child → 3

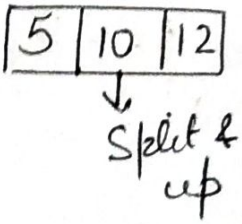
Insert 5 :

Insert 10 :-

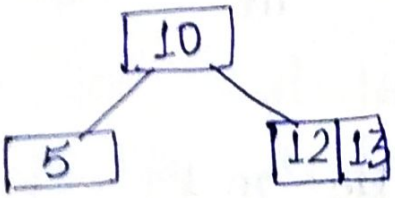


Insert 12 :-

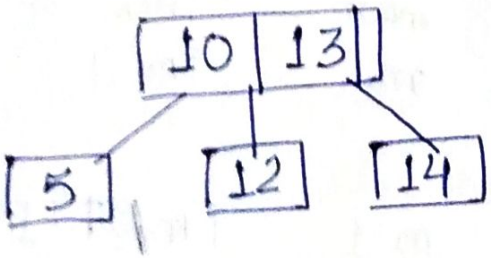


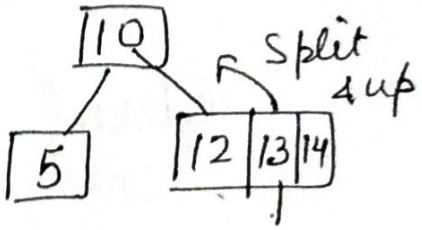


Insert 13 :-

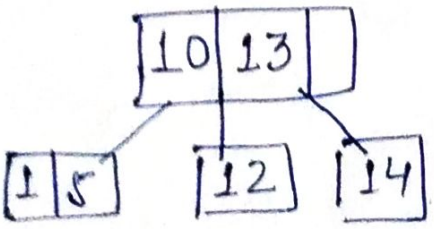


Insert 14 :-

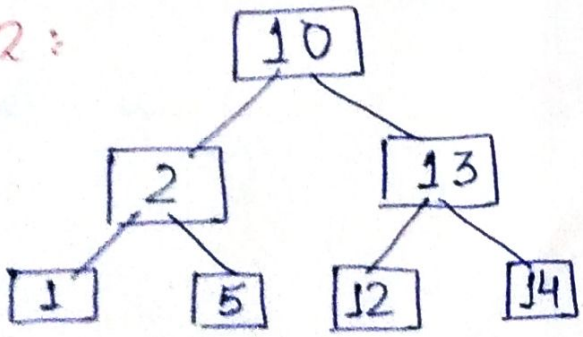


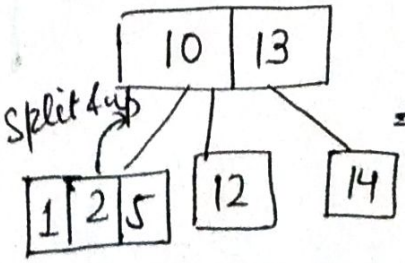


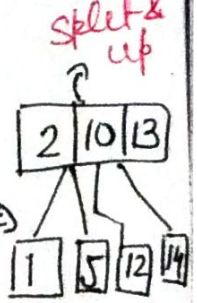
Insert 1 :-



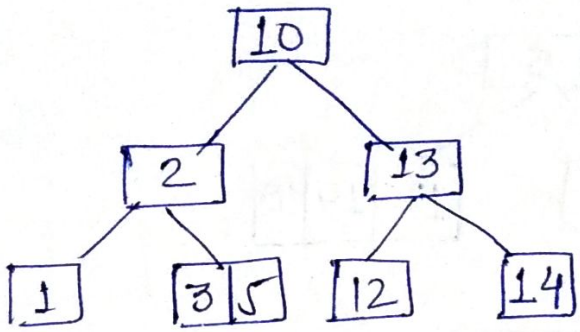
Insert 2 :-



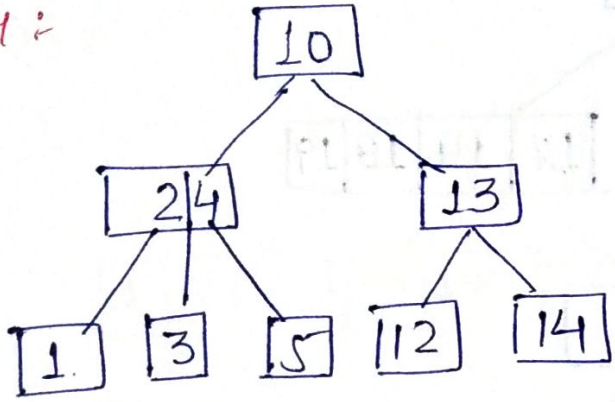


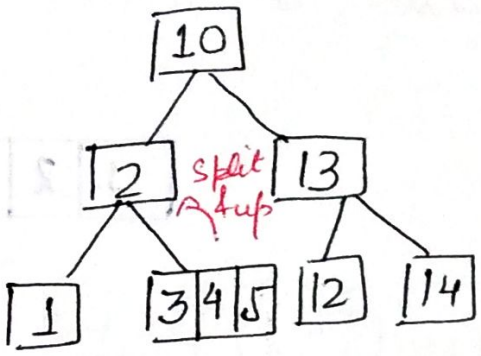


Insert 3 :



Insert 4 :





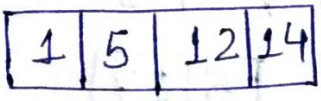
Q consider the following elements

5, 12, 14, 1, 2, 4, 18, 19, 17, 15, 25, 24, 22, 11, 30, 31, 28, 29, 13

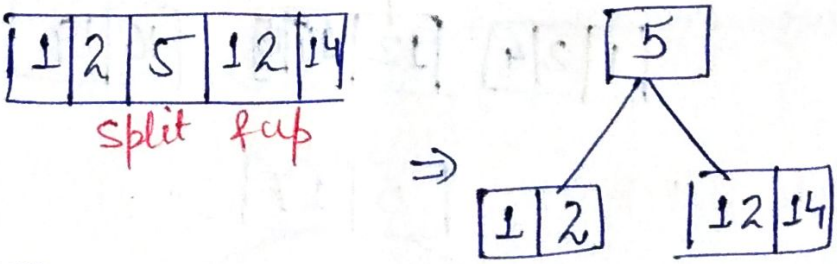
insert them into empty b-tree of order 5.

$m = 5, \quad D = 4$

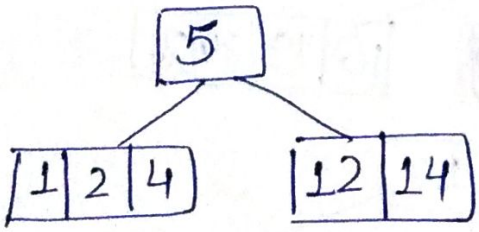
Insert 5, 12, 14, 1



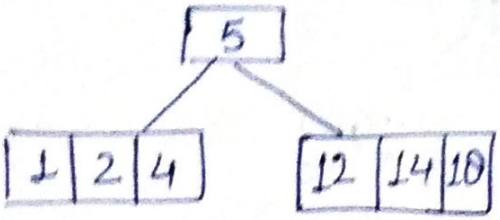
Insert 2 :-



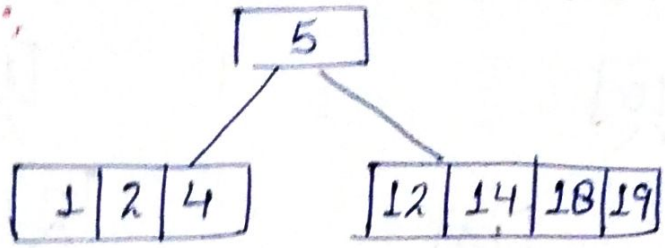
Insert 4 :



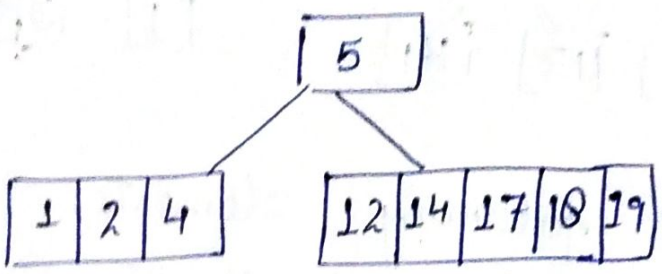
Insert 18:



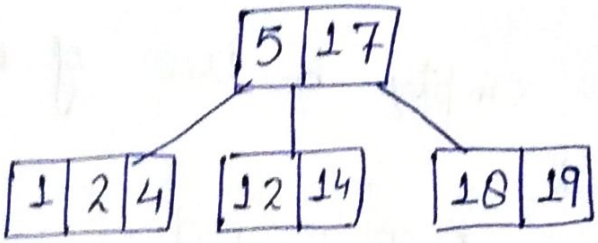
Insert 19:



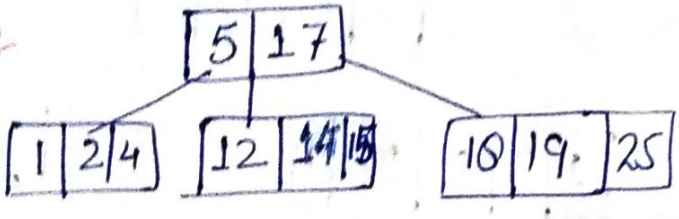
Insert 17:



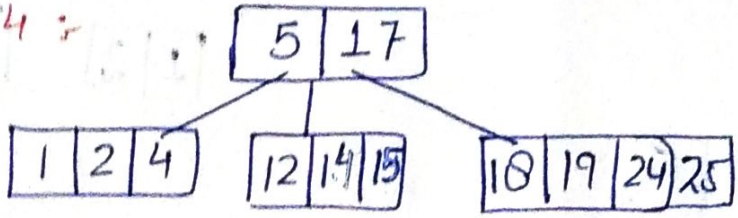
Split up



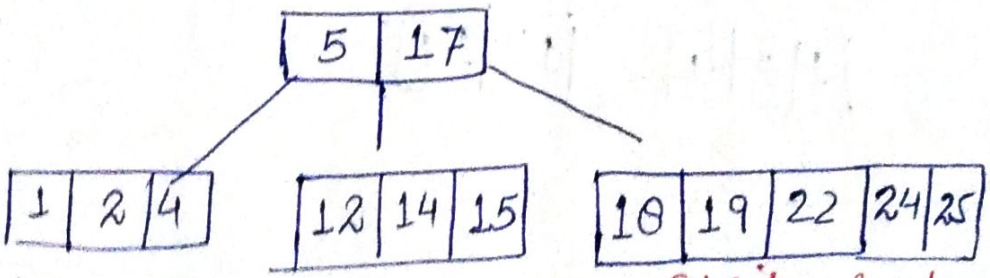
Insert 15 & 25:



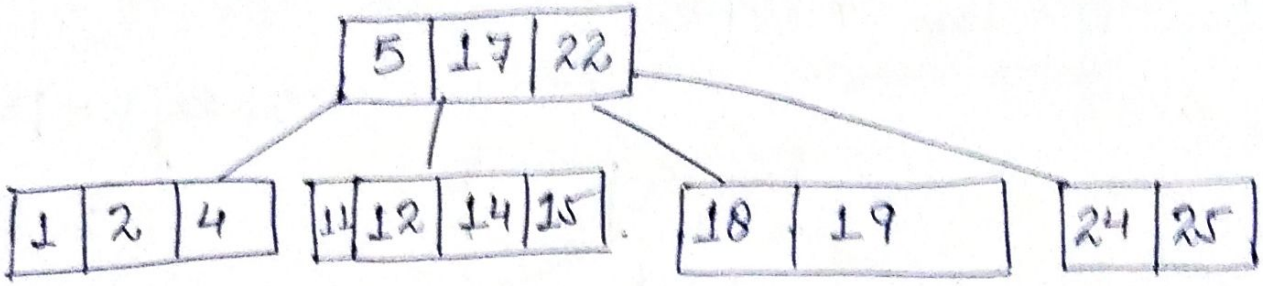
Insert 24:



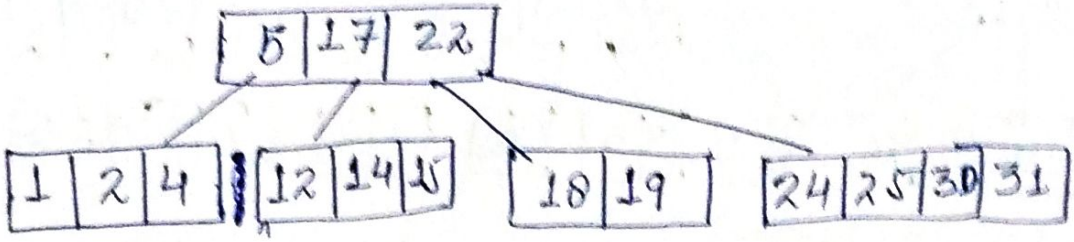
Insert 22:



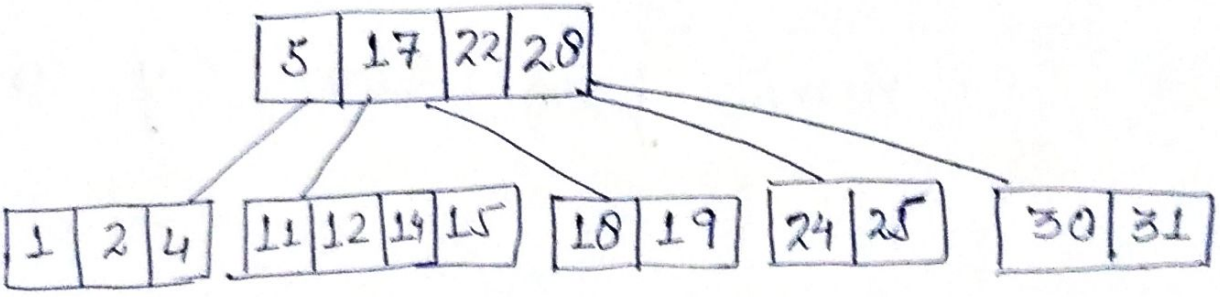
Split up



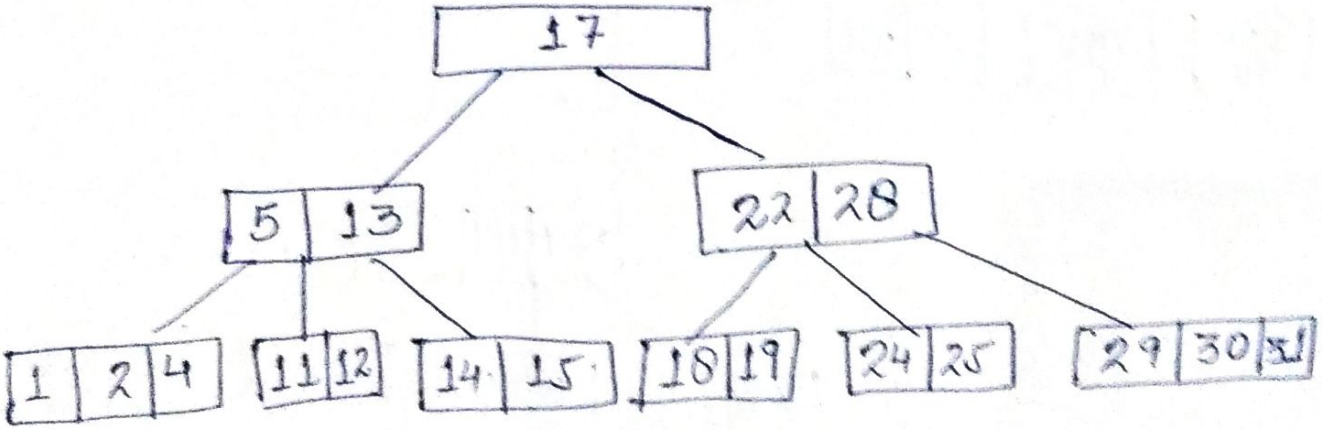
Insert 30, 31



Insert 28



Insert 29 & 13



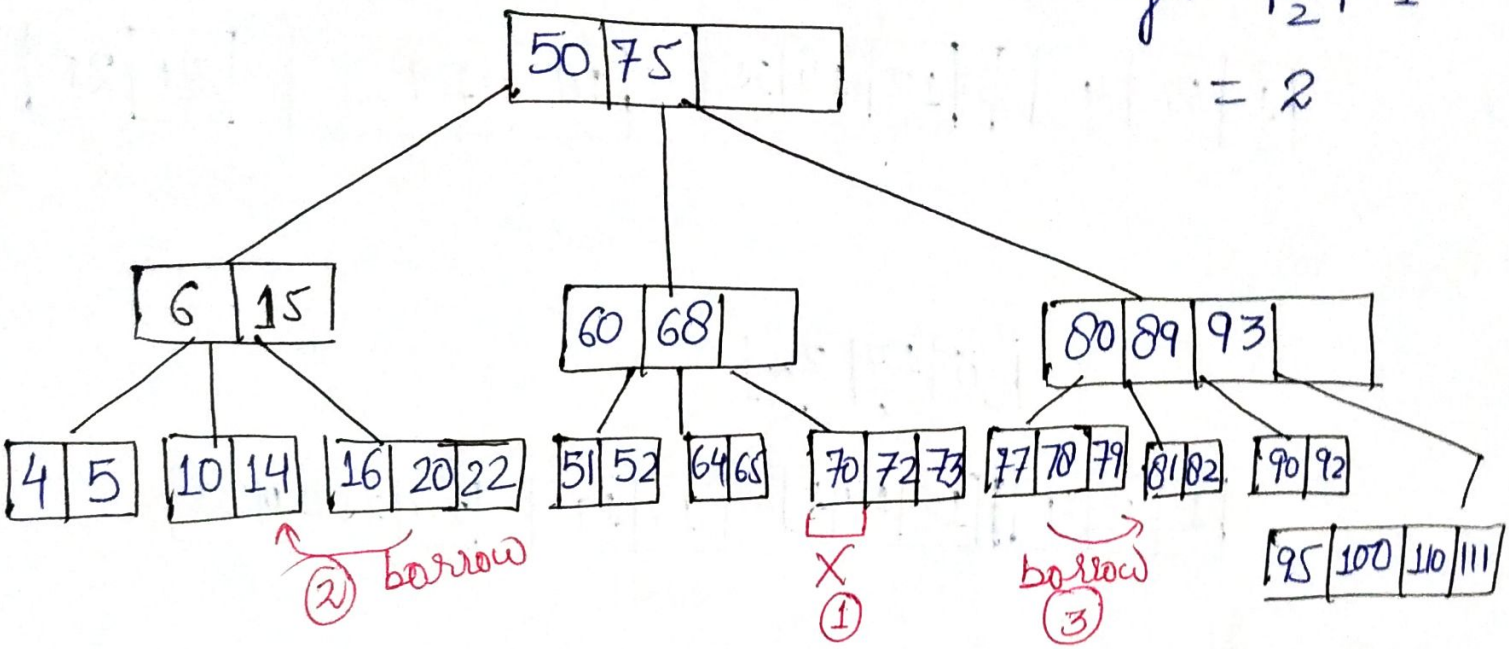
B-Tree Deletion :-

① ② ③ ④ ⑤ delete 70, 10, 81, 65, 75

$m = 5$

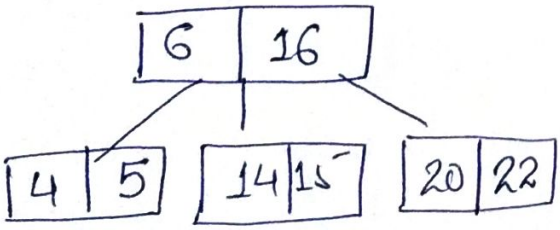
Min. Keys $= \lceil \frac{5}{2} \rceil - 1$

$= 2$



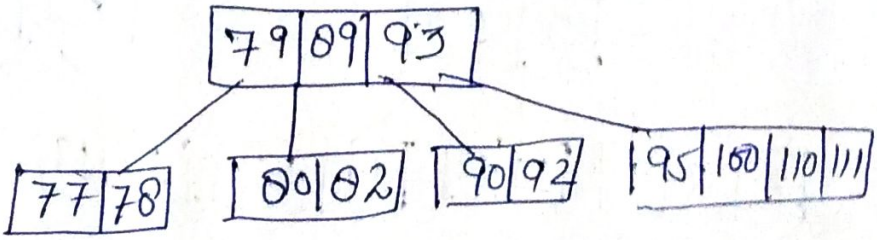
delete 10 :- Borrow from Sibling (if extra)

②

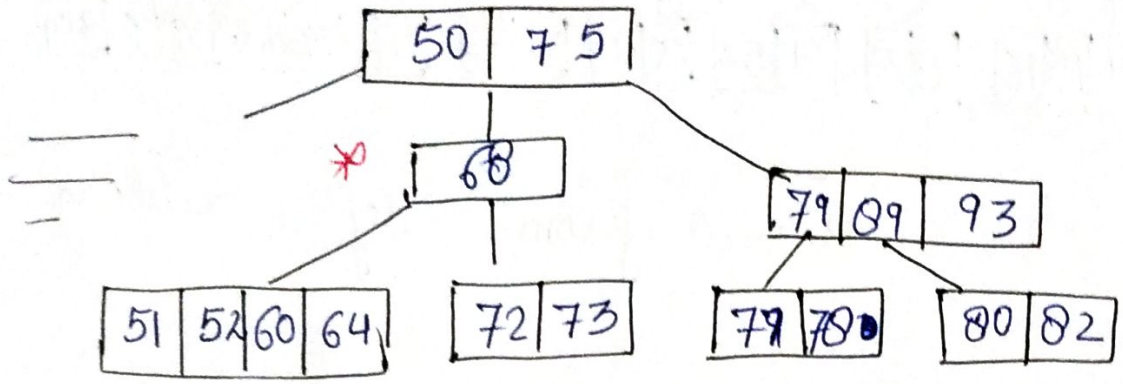


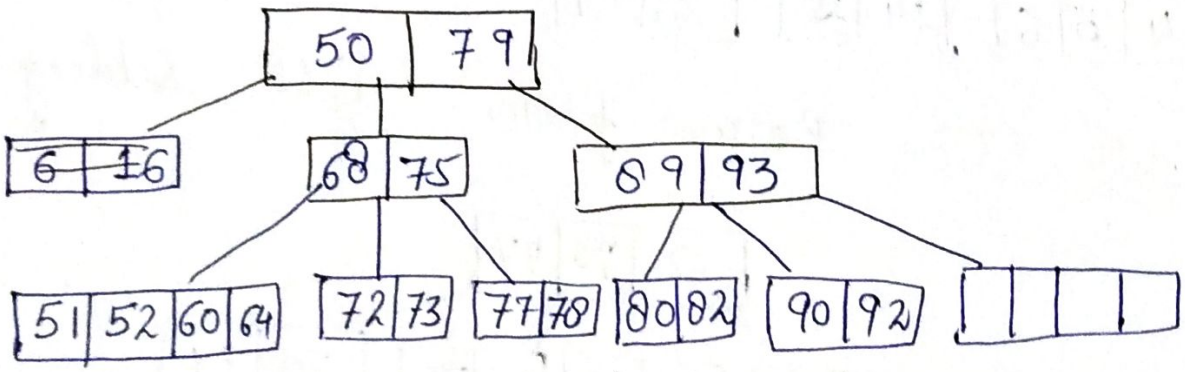
delete 81 :-

③

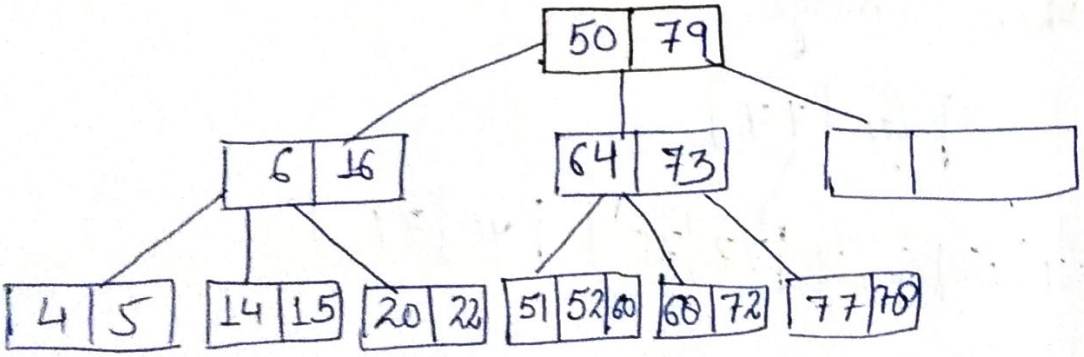


delete 65 :- ④ as can't borrow from L or R Sibling, Thus we'll merge



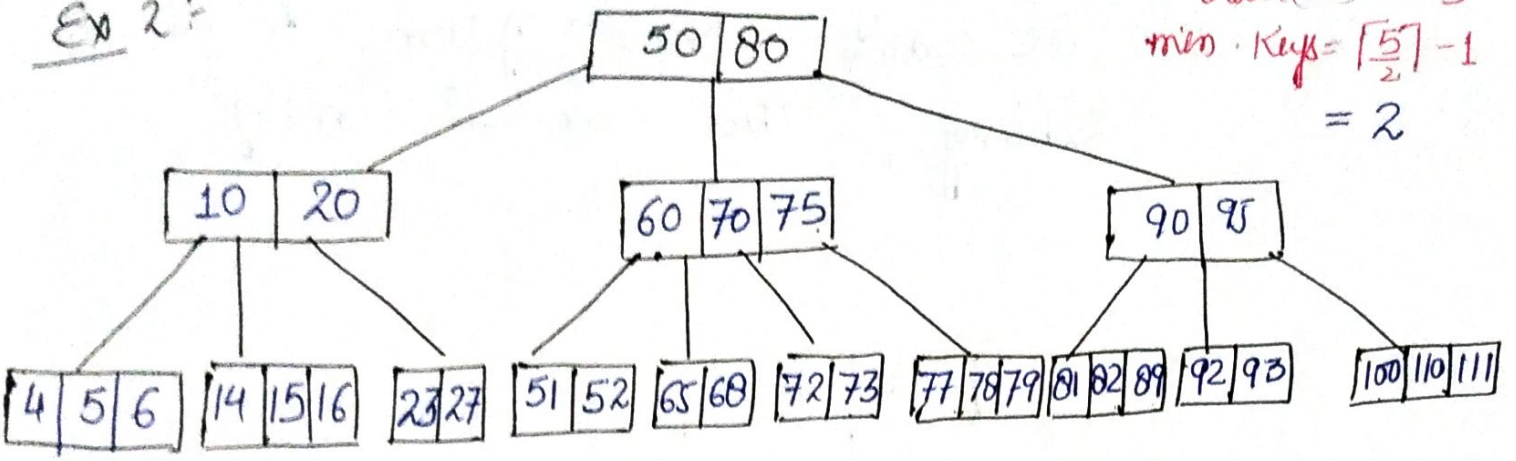


delete 75 :- ⑤ Replace 75 by its inorder Predecessor 73

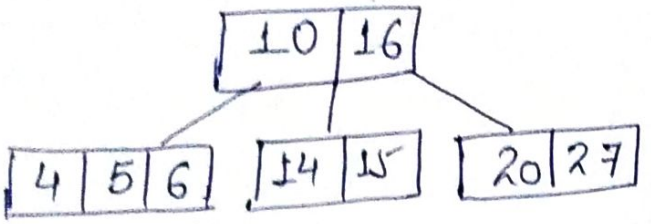


Ex 2:

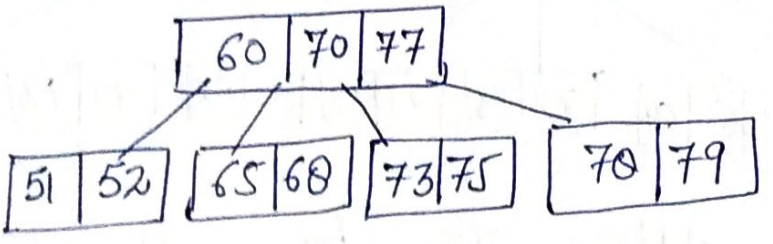
Order(m) = 5
min. Keys = $\lceil \frac{5}{2} \rceil - 1$
= 2



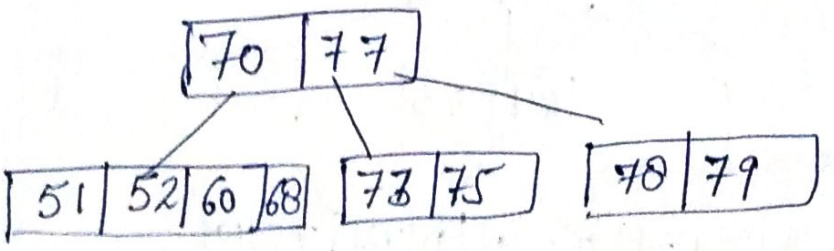
delete 23 :- Borrow from left sibling



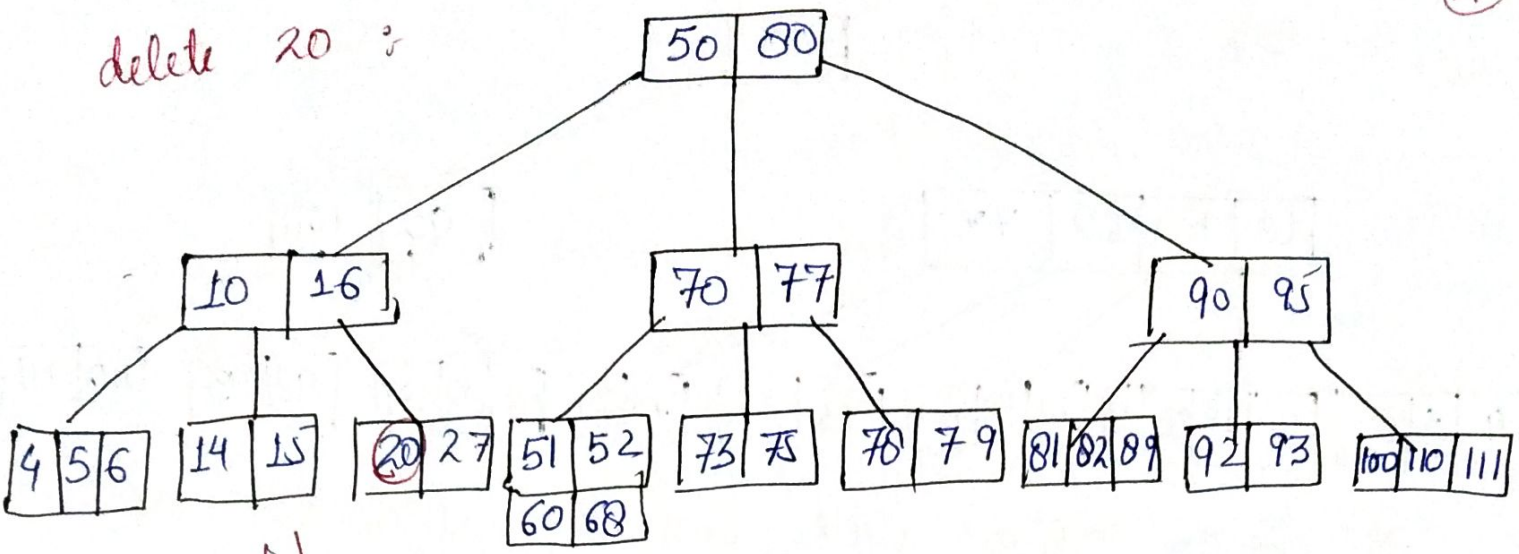
delete 72 : Borrow from right sibling



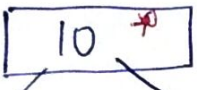
delete 65 : can't borrow either from L nor R.
then merge :



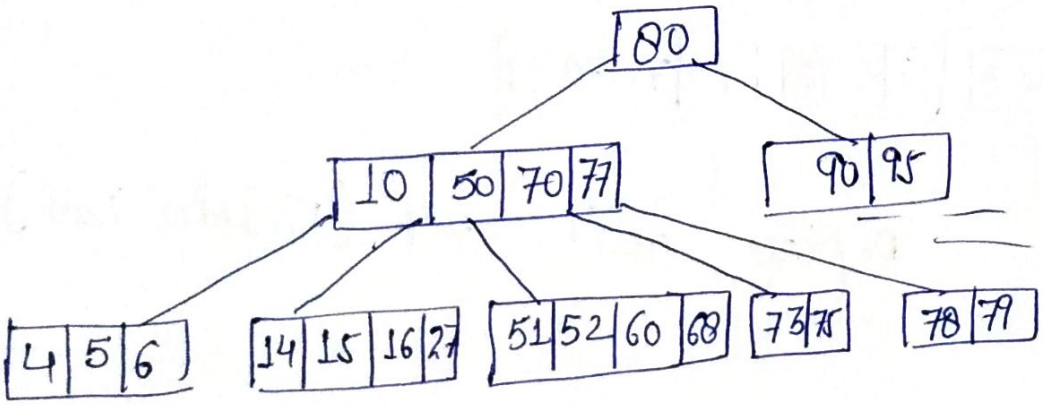
delete 20 :



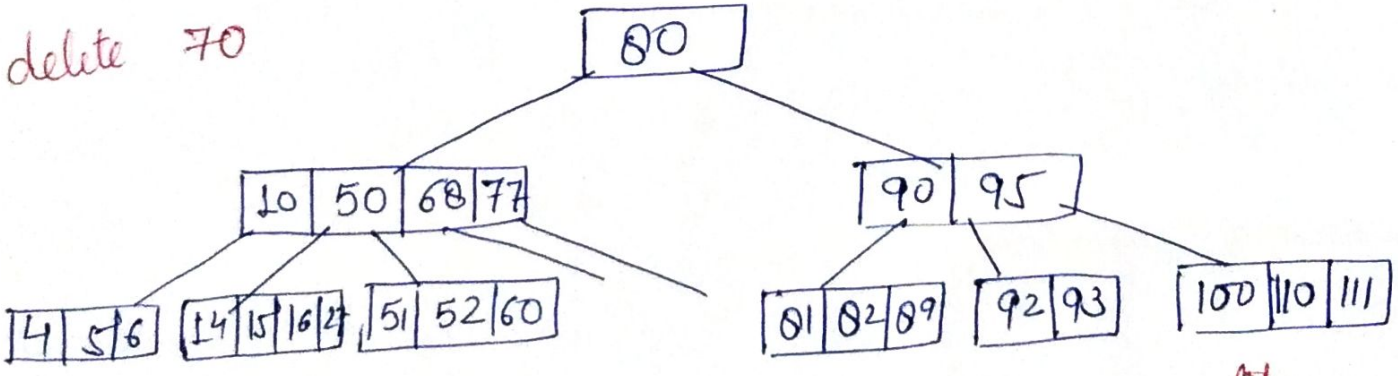
merge



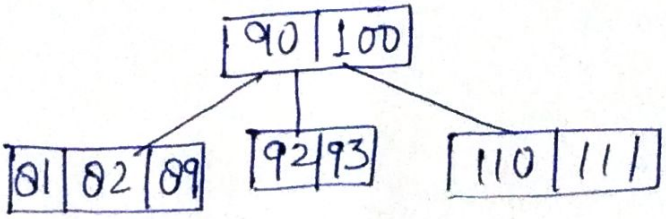
violate ⇒ can't borrow from Sibling than merge



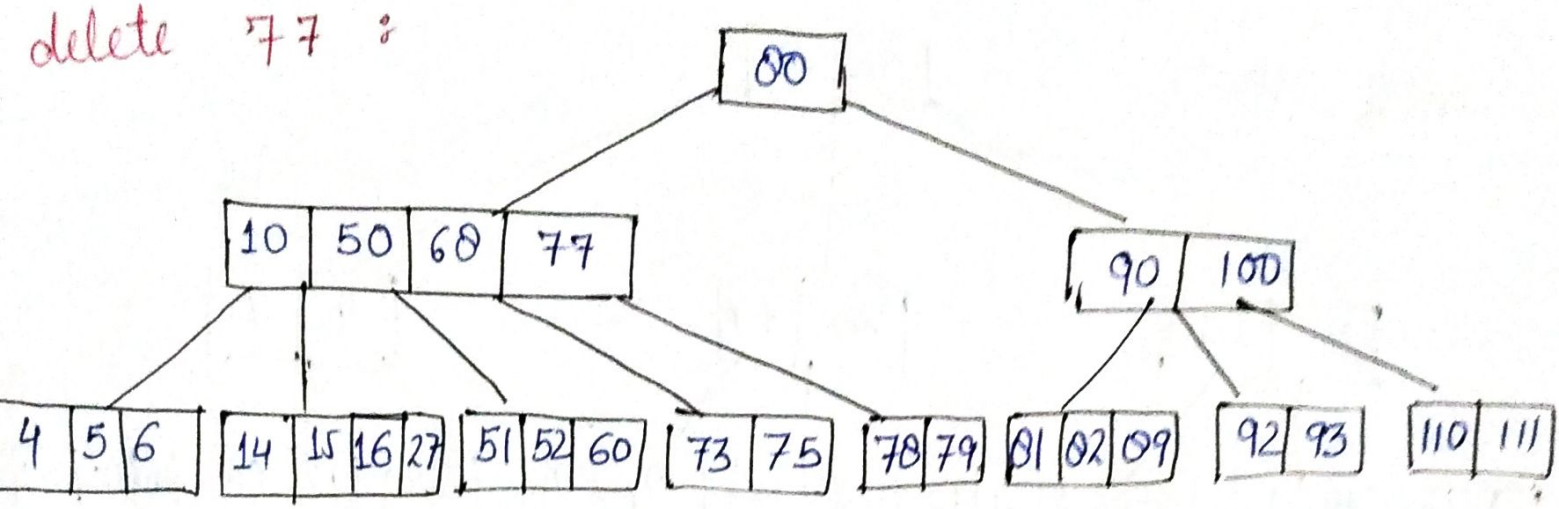
delete 70



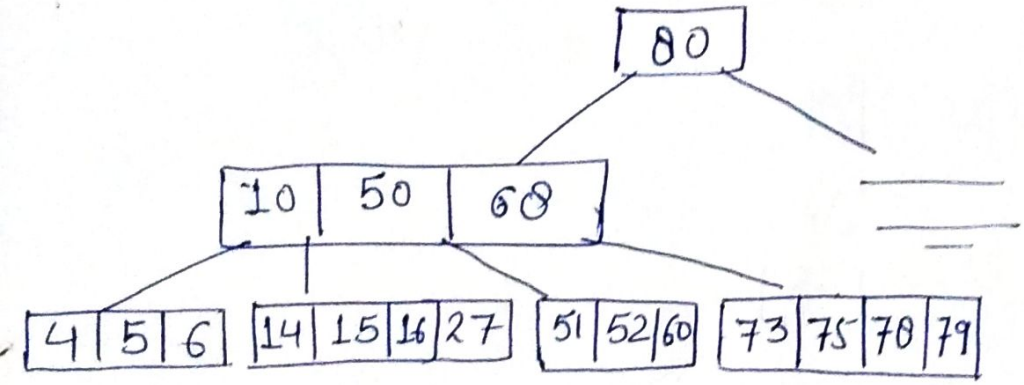
delete 95 : can't replace Replace with Predecessor than with Successor :



delete 77 :



* can't Replace with inorder successor or predecessor, than Merge



* delete 80 : Replace with 79 (Inorder Pred.)

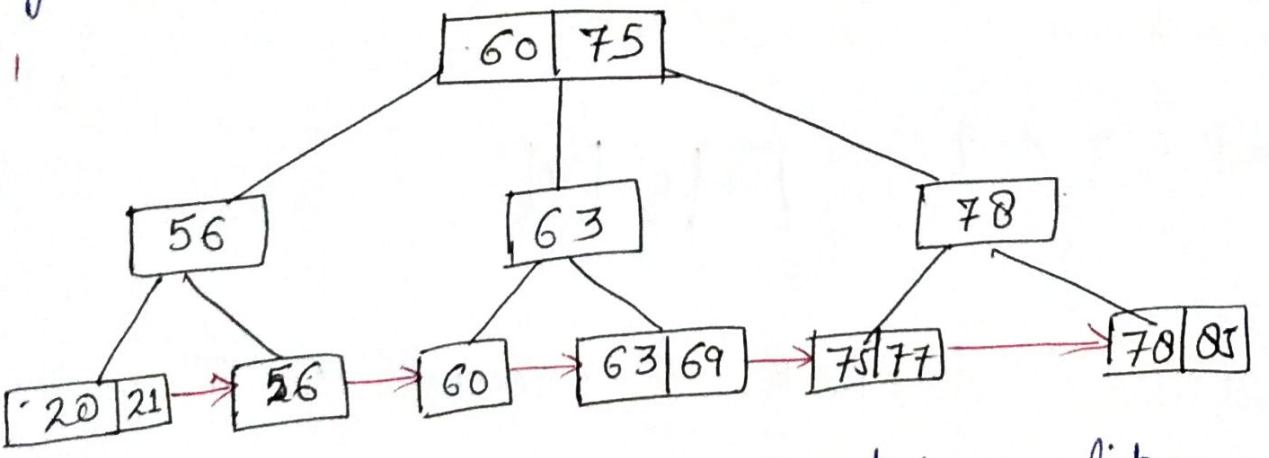
B^+ Trees :-

* B+ trees are extended version of B-Trees. This tree supports better insertion, deletion and searching over B-tree.

* In B^+ tree records can be stored at the leaf node; internal nodes will store key values only.

* The leaf nodes of B+ tree also linked together as linked list.

e.g.:



* This supports basic operations like searching, insertion & deletion. In each node, items will be sorted.

Advantages over B-Tree :-

* As leaf are connected like linked list, we can search elements in sequential manner also.

* Searching is faster.

* Records can be fetched in equal no. of disc accesses.

Insertion :-

Q. Insert the following values into an empty B+ tree of order 4.

3, 6, 9, 12, 19, 23, 33, 27, 21, 22, 30, 44

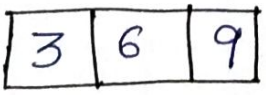
order $m = 4$, max. child $= 4$

min " $= \lceil \frac{4}{2} \rceil = 2$

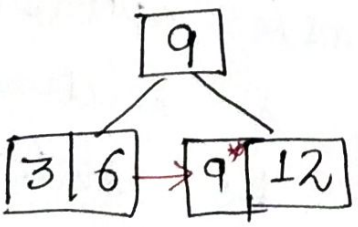
Max Keys $= 3$

Min Keys $= \lceil \frac{3}{2} \rceil - 1 = 1$

Insert 3, 6, 9 :-

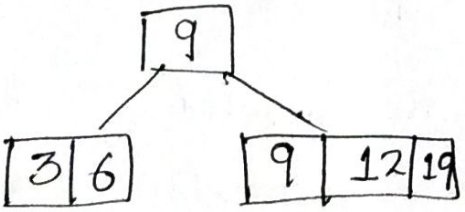


Insert 12 Split at second middle

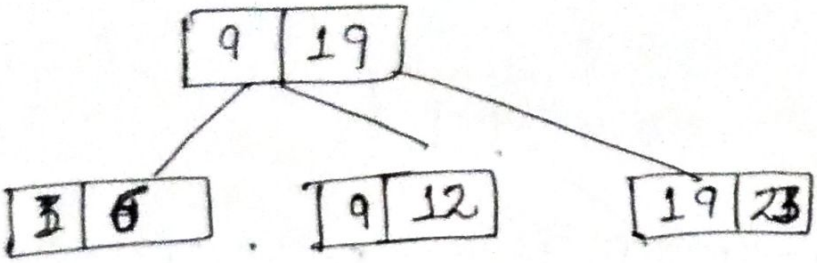


* all data must present in leaf node.

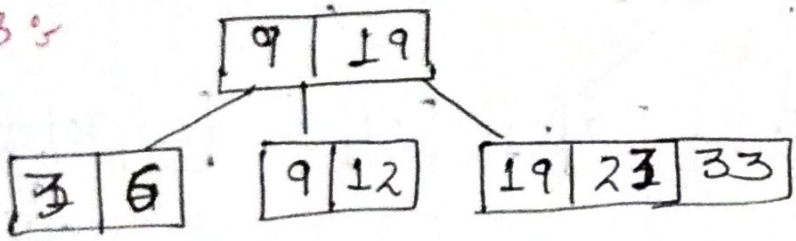
Insert 19



Insert 23:

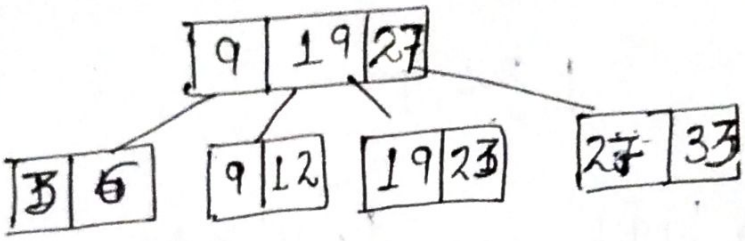


Insert 33:

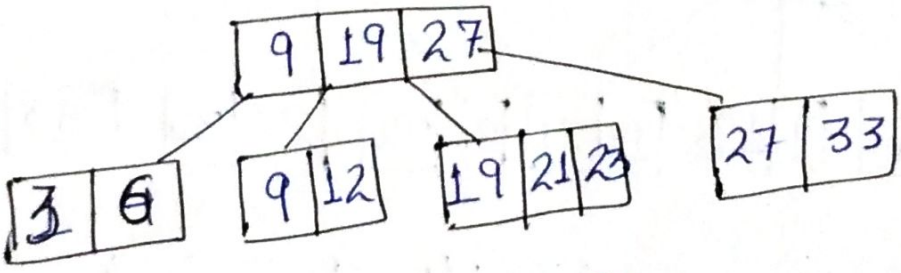


Insert 27:

Split at 27 & up

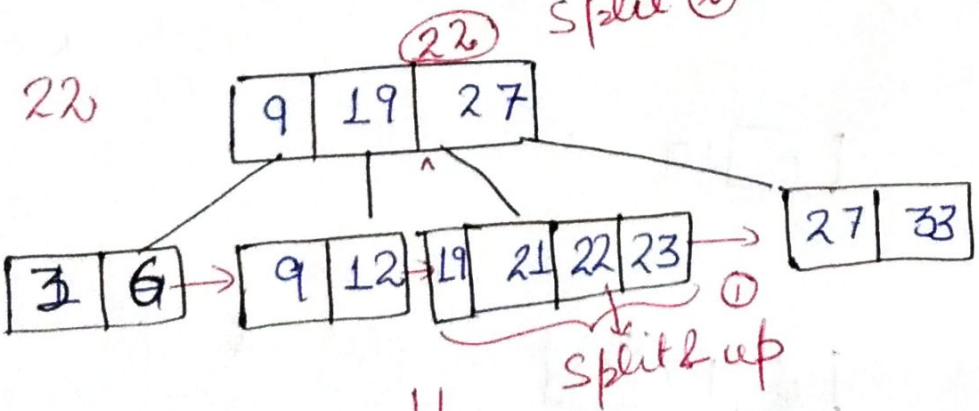


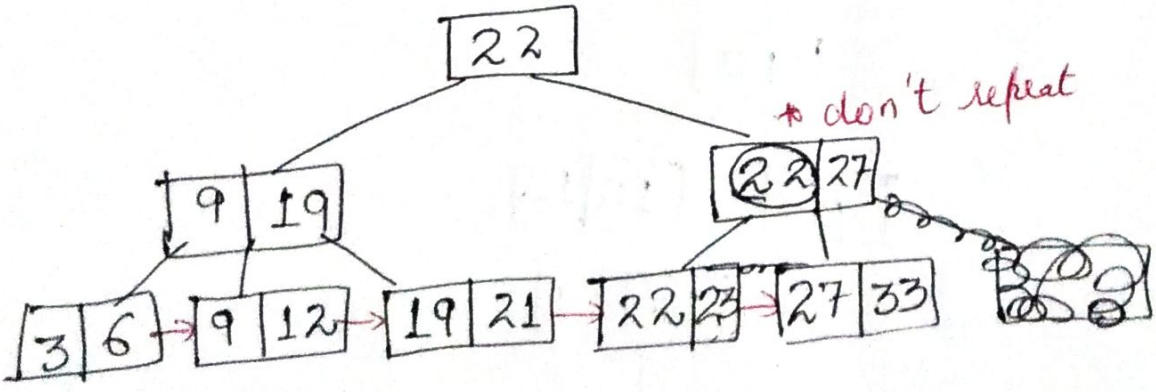
Insert 21



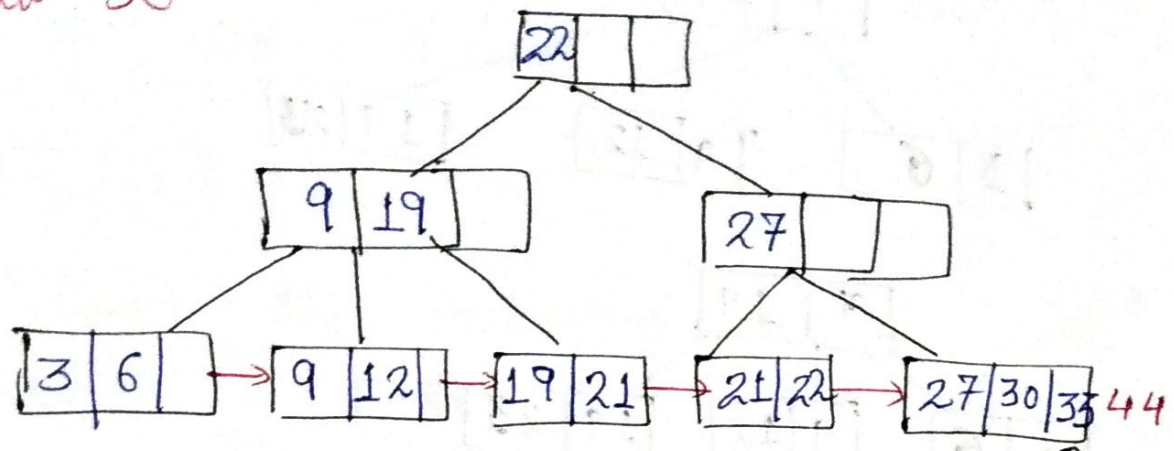
Insert 22

Split ②

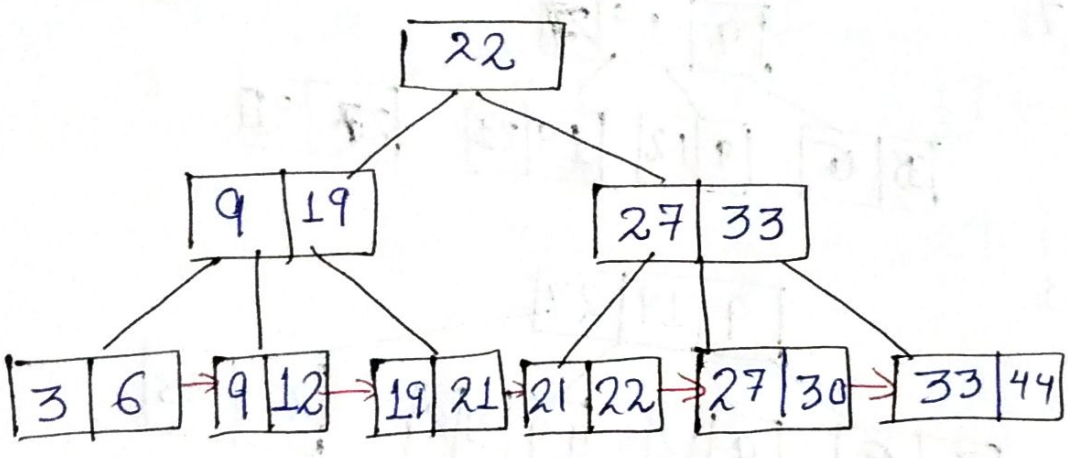




Insert 30



Insert 44



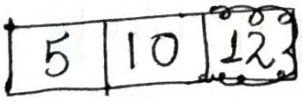
Q 5, 10, 12, 13, 14, 1, 2, 3, 4 $m = 3$

$k = 2$

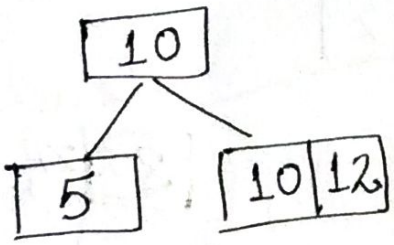
Insert 5 :

& 10

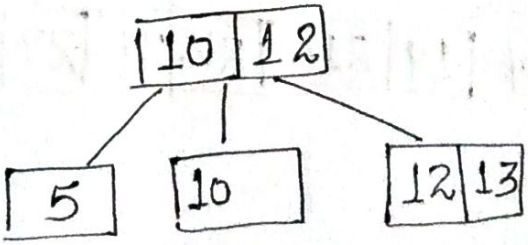
Insert 12 :-



split & up

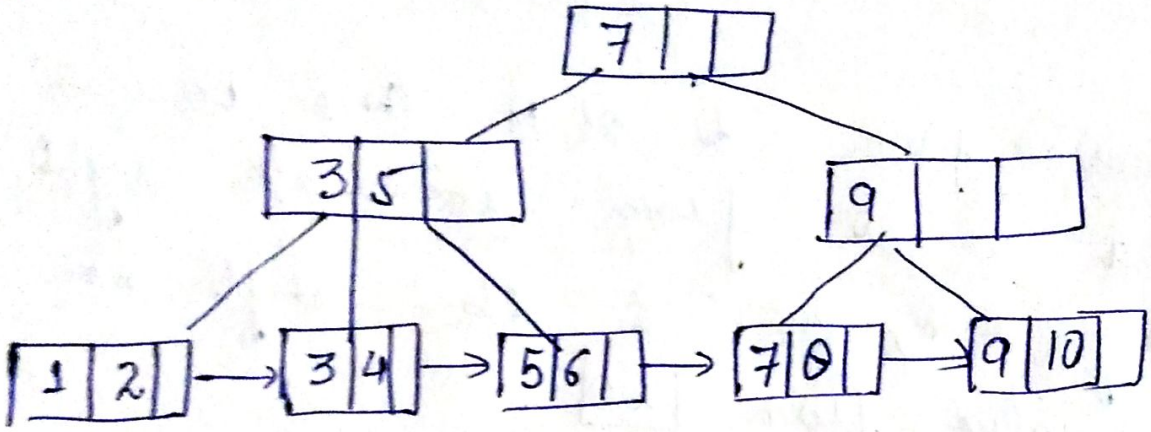


Insert 13 :-



Q :- Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 into an empty B+ tree of order 4.

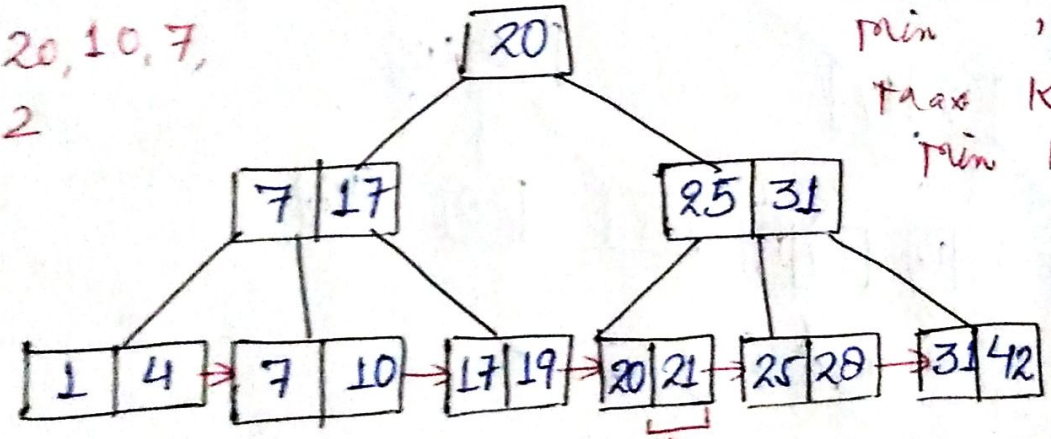
Ans



Deletion in B+ Tree :-

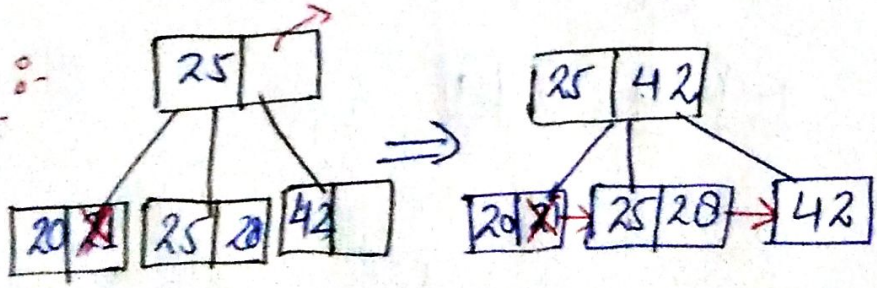
Order (m) = 4
max child = 4
min " = $\lceil \frac{4}{2} \rceil = 2$
max Key = 3
min Key = $\lceil \frac{4}{2} \rceil - 1 = 1$

21, 31, 20, 10, 7, 25, 42

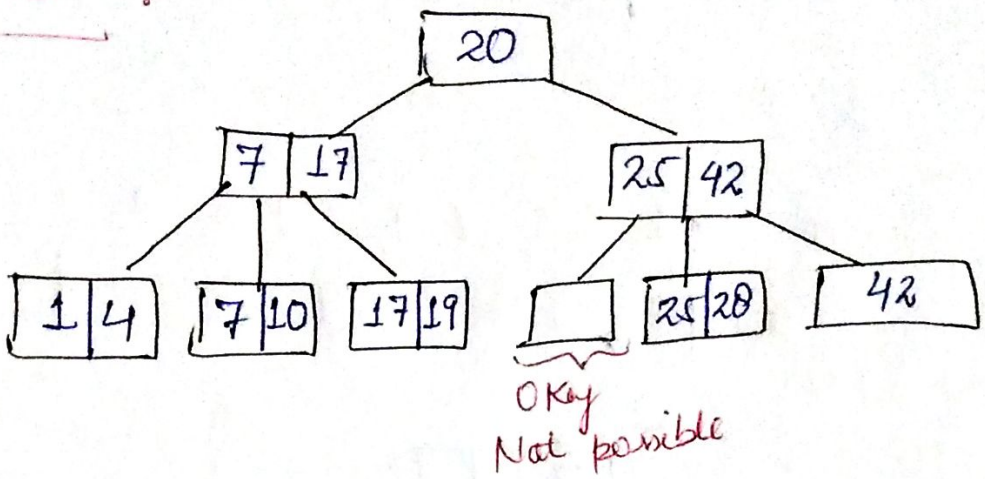


Delete 21 :- No Problem, as min. 1 Key will remain. copy min value from right child

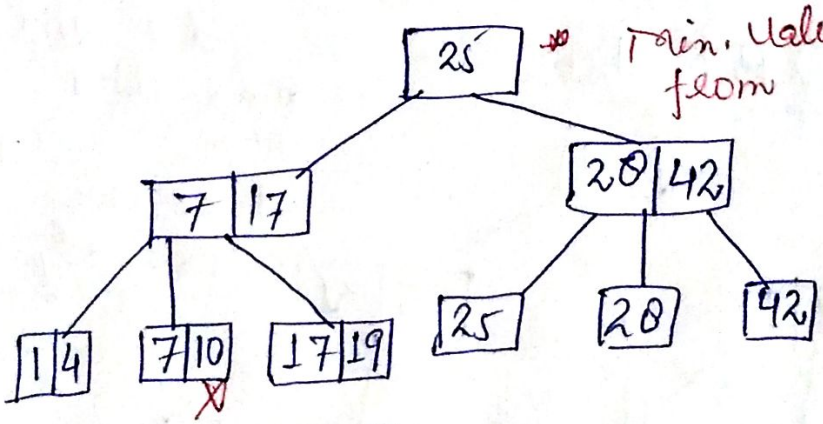
Delete 31 :-



Delete 20 :-



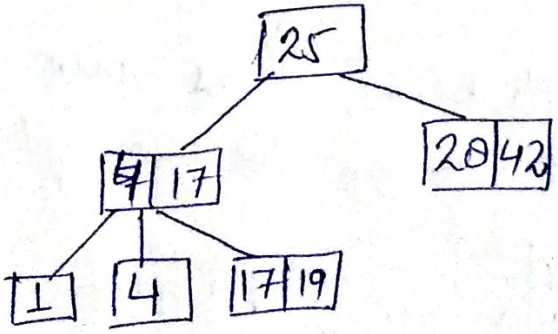
* Borrow from L or R Thus borrow smallest element from immediate right sibling, i.e, 25, & then shift new min. value from 20.



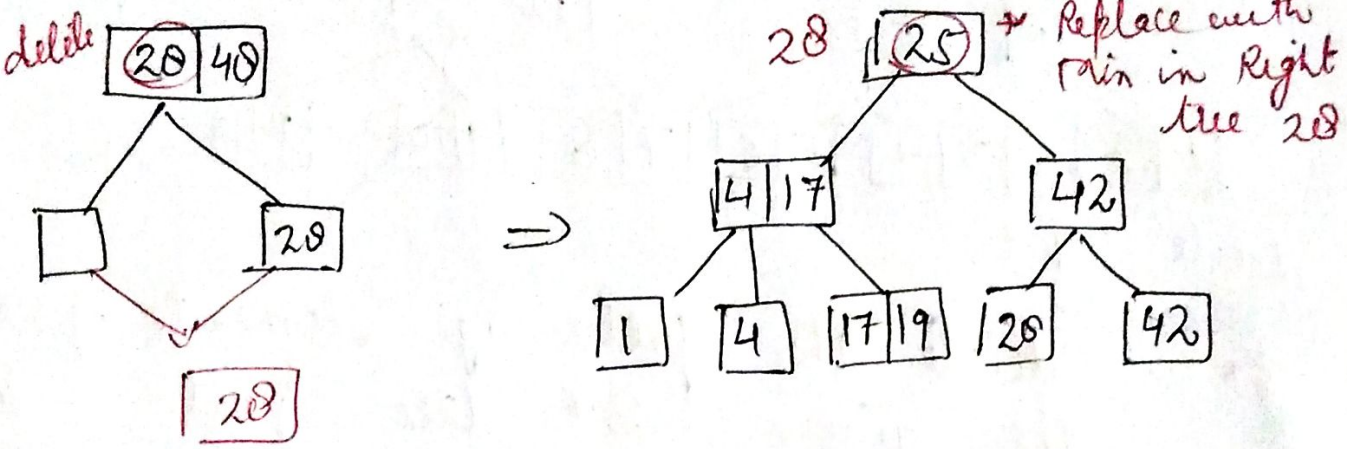
* Min. value at leaf nodes from right subtree.

Delete 10 :- No Problem

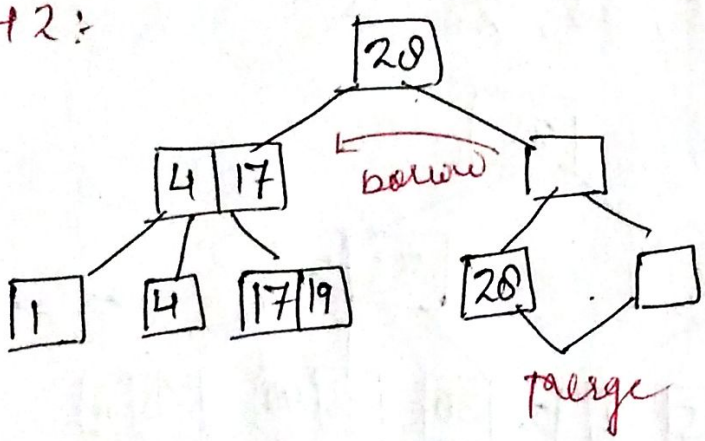
Delete 7 :-

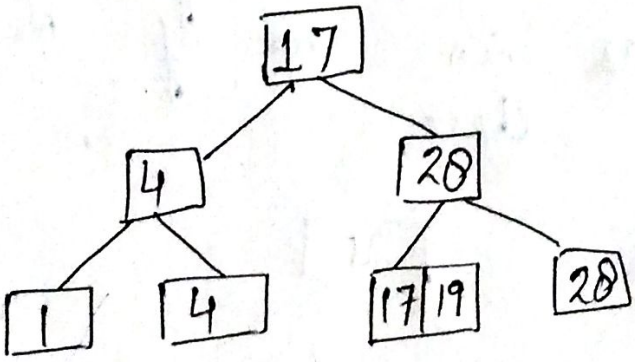


Delete 25: Can't borrow from any of the sibling, then merge

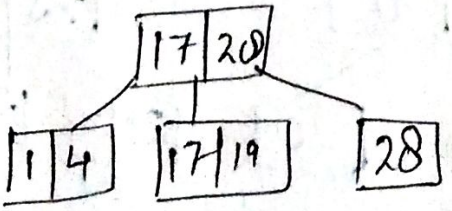


Delete 42:





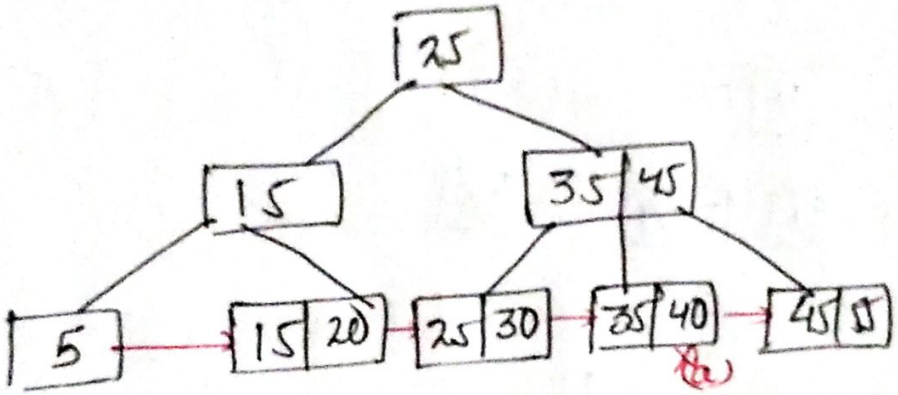
Delete 4:-



delete 9 :

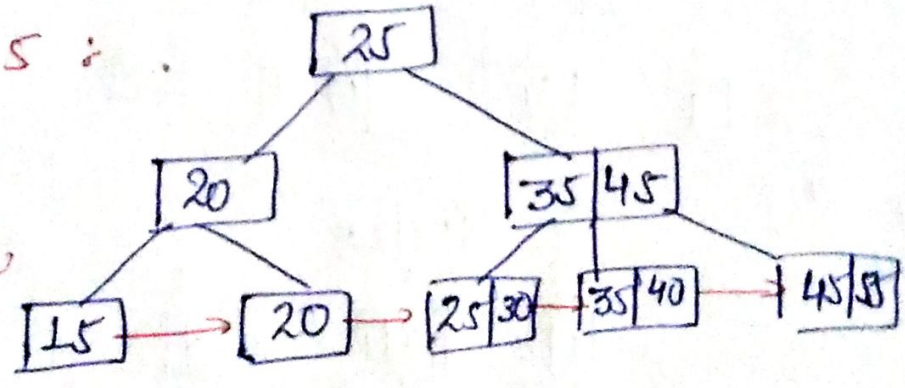
Ex 2:- delete the following elements from the given B+ tree
40, 5, 45, 35, 25, 15

order 3
min = $\lceil \frac{3}{2} \rceil - 1$
= 1



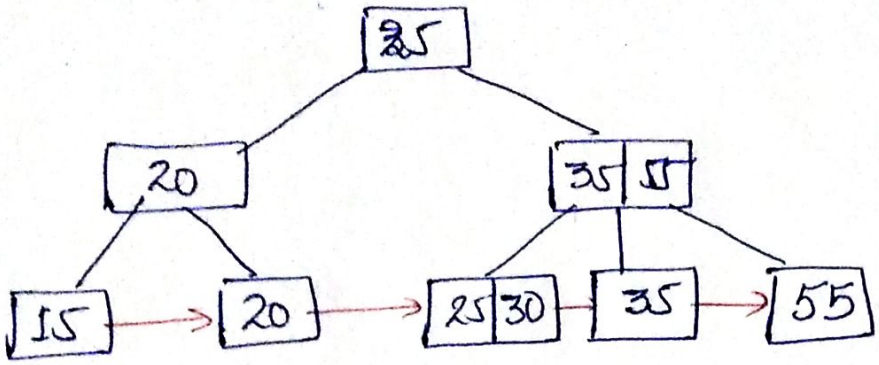
(a) delete 40 :- No changes required, as min. no. of keys are 1 & 35 remain there.

(b) delete 5 :-

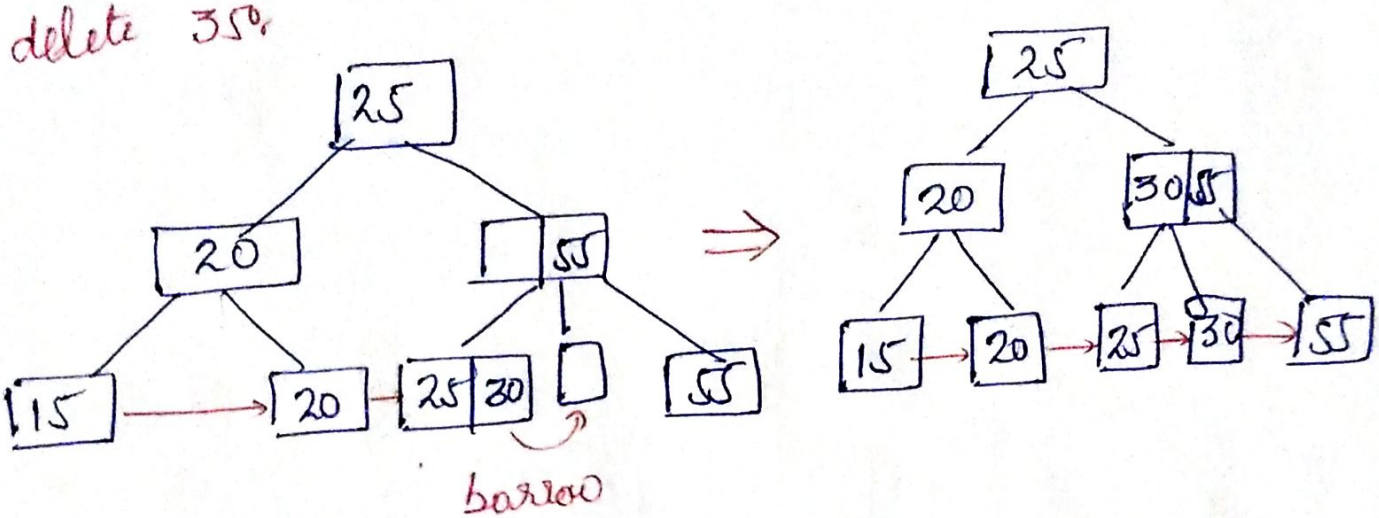


borrow 15 from Sibling

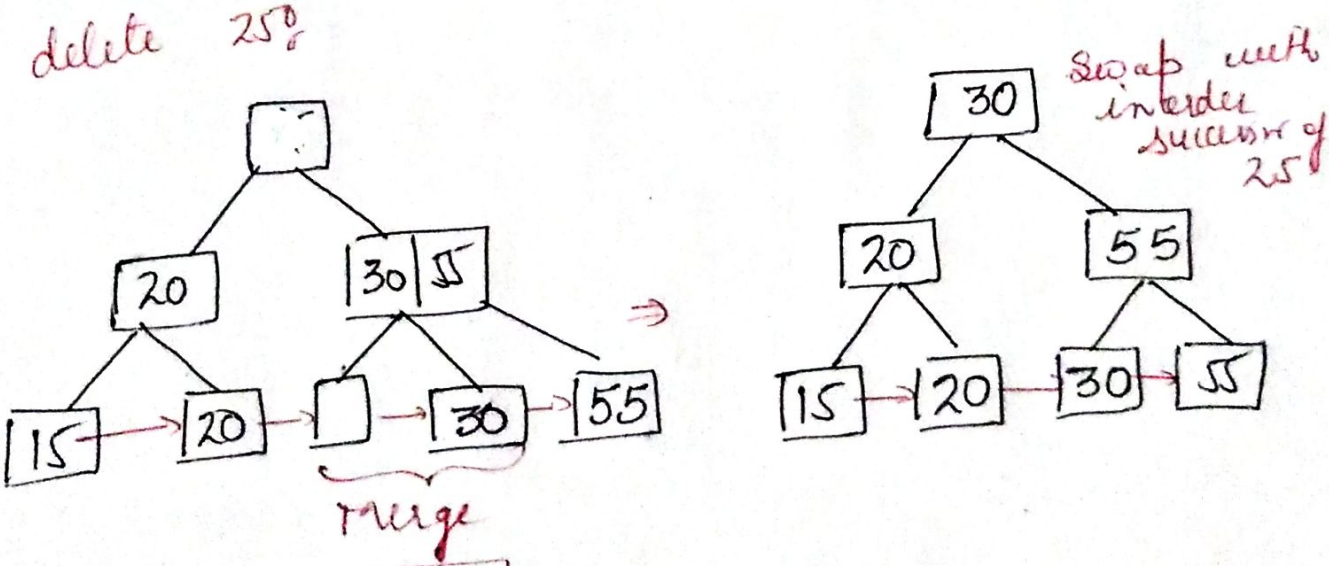
(c) delete 45 :- 45 is in internal node.



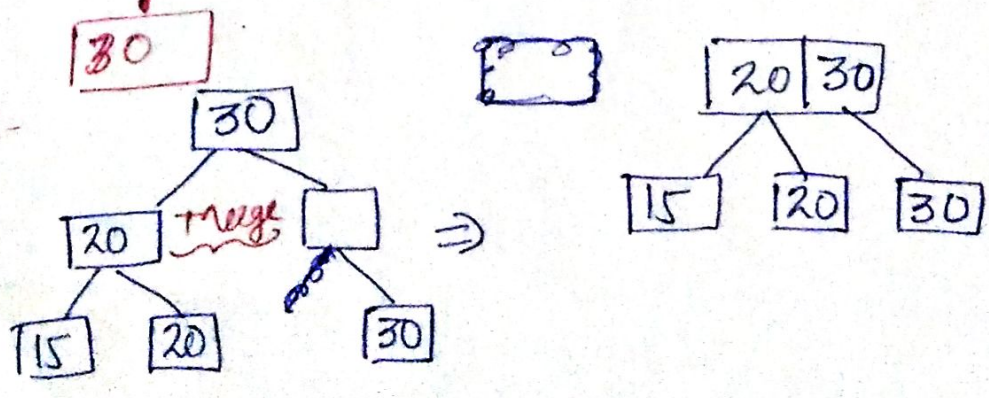
(d) delete 35 :-



(e) delete 25 :-



(f) delete 55 :-



Threaded Binary Tree

* As in case of ~~memory~~ binary tree, memory is not efficiently utilised. For storing 'n' values, (n+1) values are null values (leaf nodes).
* Hence, we have the concept of threaded binary tree.
* A binary tree with threaded pointers is called a threaded binary tree.
* In the linked representation of any binary tree if there are n no. of nodes, then there'll be (n+1) no. of null pointers. So in order to efficiently utilize the memory those null pointers can be replaced by some special pointers called as threads, which will point to the nodes higher in the tree.
* A binary tree with these thread pointers is called a threaded binary tree. Diagramatically, a thread can be represented by a dotted line

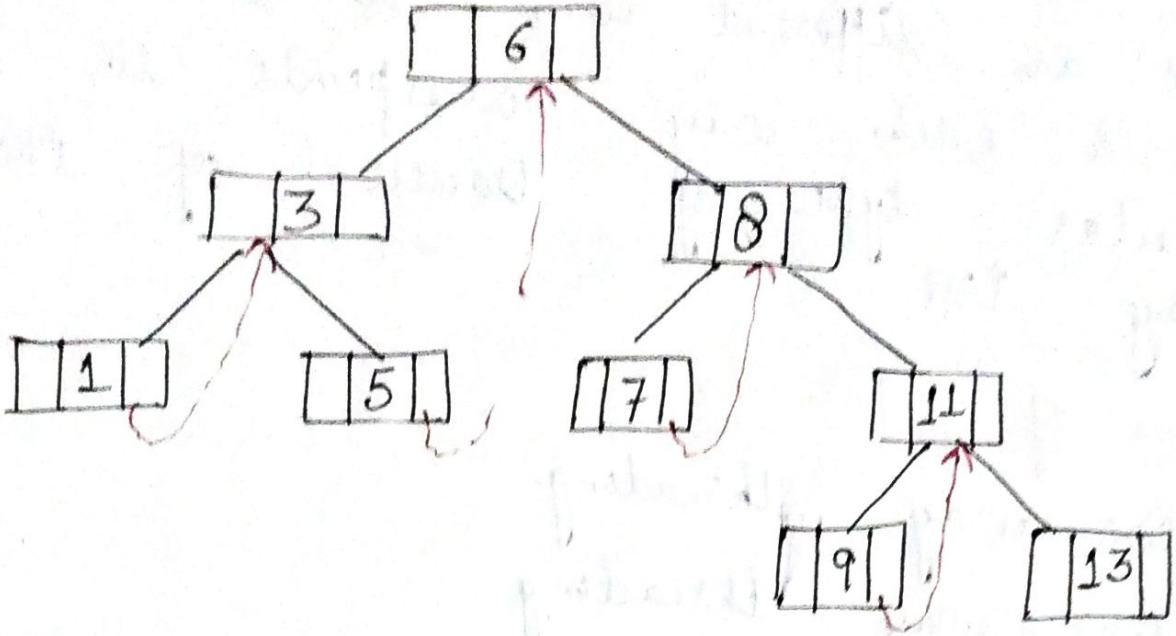
* There are different ways to thread a binary tree & each way corresponds to a particular type of traversal of the binary tree.

Types of Threads :-

1. One-way threading
2. two-way threading

1. One-way Threading :-

* ~~In one way threading a thread will appear in the right / null pointer field~~
* Each node is threaded towards either the in-order predecessor or successor (left or right).
* All right NULL pointers will point to inorder successor OR all the left NULL pointers will point to inorder predecessor.



② Two-way Threading :-

Each node is threaded towards both in-order predecessor & successor (left & right). Thus all right NULL pointers will point to inorder successor AND all left pointers will point to inorder predecessor.