NAME: PRAGYA AGARWAL
STUDENT ID: 1001861779
ASSIGNMENT ID: 02

ASSIGNMENT - 02

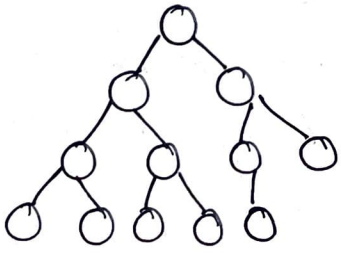
Ques 1 Answer the following questions regarding Binary trees:

a) What are the least and greatest number of leaf nodes in a binary tree with n nodes, show with examples?

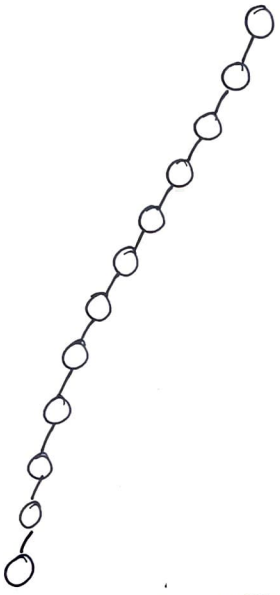
a) The least number of leaf nodes in a binary tree with n nodes is 1.

The greatest number of leaf nodes in a binary tree with n nodes is $\left\lceil \frac{n+1}{2} \right\rceil$, in case when binary tree is perfect, or complete, or full.

EXAMPLE: A binary tree with no. of nodes n = 12



→ This is a complete binary tree with 12 nodes.
The no. of leaf nodes here is 6, and that is the maximum too : $\frac{(n+1)}{2} = \left\lceil \frac{13}{2} \right\rceil = 6$



→ This is a left skewed binary tree, where each parent node has only one child node. The no. of leaf nodes in this case is 1, which is the minimum possible.

BINARY TREE: It is a non-linear, hierarchical data structure where each node has at most two child nodes.

LEAF NODE: A node in the tree is identified as leaf node when that node does not have any right and left child nodes.

b) What is the relationship between the number of nodes in a full binary tree and the number of leaf nodes, show with an example?

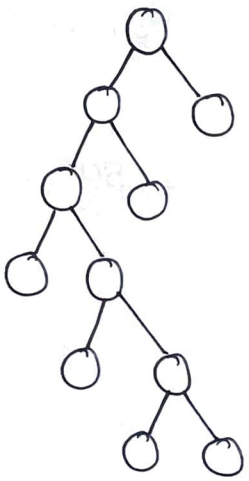
b) Relationship between the no. of nodes and no. of leaf nodes in case of a full binary tree is given by:

$$\text{no. of leaf nodes} = \frac{\text{no. of nodes} + 1}{2}$$

$$l = \frac{n+1}{2}$$

A binary tree is identified as a full binary tree when every node has either 0 or 2 child nodes.

EXAMPLE :



→ This is an example of a full binary tree, with no. of nodes equal to 11.
The no. of leaf nodes in this tree as can be counted from the figure as well are 6.

using the relationship, $l = \frac{n+1}{2}$, we obtain the same result.

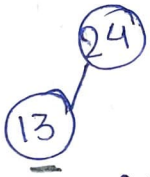
Ques 2 Answer the following questions regarding Binary Search trees:

a) Insert the following 15 randomly generated objects into a binary search tree in the order they are listed (show each step).

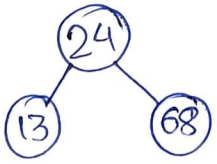
24, 13, 68, 62, 66, 22, 58, 80, 2, 59, 23, 54, 1, 15, 95

a) 1) Insert 24

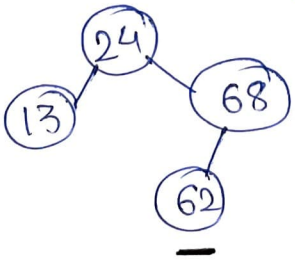
2) Insert 13 (13 < 24 → left)



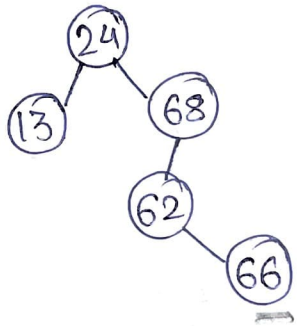
3) Insert 68 (68 > 24 → right)



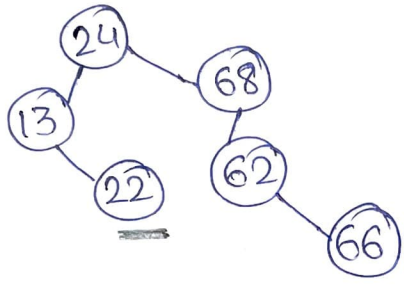
4) Insert 62 (62 > 24 → right, 62 < 68 → left)



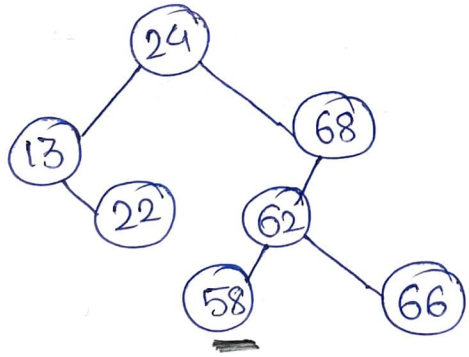
5) Insert 66 (66 > 24 → right , 66 < 68 → left,
66 > 62 → right)



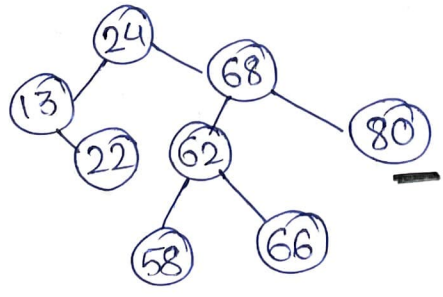
6) Insert 22 (22 < 24 → left , 22 > 13 → right)



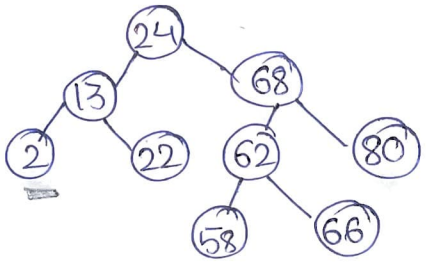
7) Insert 58 (58 > 24 → right , 58 < 68 → left,
58 < 62 → left)



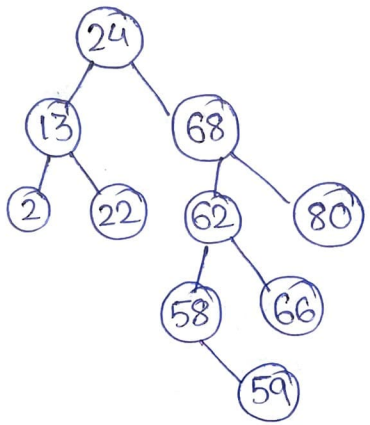
8) Insert 80 (80 > 24 → right , 80 > 68 → right)



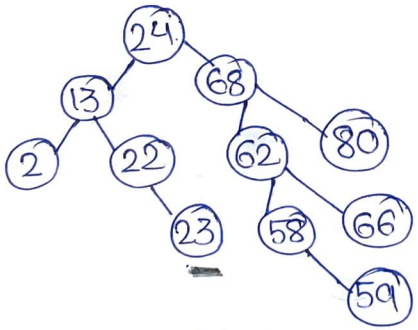
9) Insert 2 (2 < 24 → left , 2 < 13 → left)



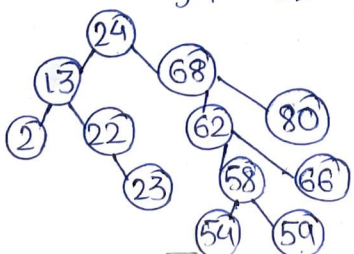
10) Insert 59 (59 > 24 → right , 59 < 68 → left ,
59 < 62 → left , 59 > 58 → right)



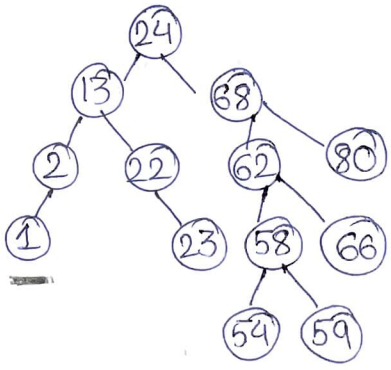
11) Insert 23 (23 < 24 → left , 23 > 13 → right ,
23 > 22 → right)



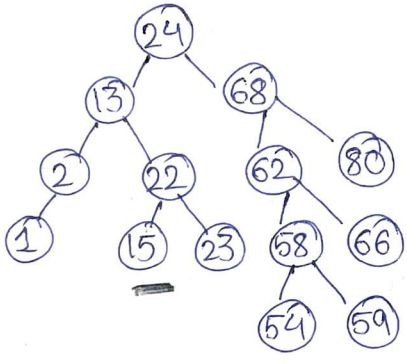
12) Insert 54 (54 > 24 → right , 54 < 68 → left ,
54 < 62 → left , 54 < 58 → left)



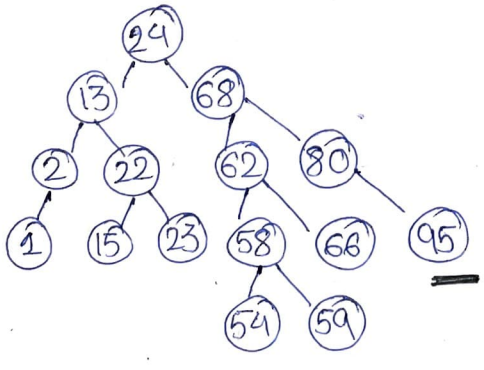
13) Insert 1 (1 < 24 → left, 1 < 13 → left, 1 < 2 → left)



14) Insert 15 (15 < 24 → left, 15 > 13 → right,
15 < 22 → left)



15) Insert 95 (95 > 24 → right, 95 > 68 → right,
95 > 80 → right)



→ Final binary search tree obtained by inserting the given 15 objects.

Binary Search Tree: The left subtree of a node in BST contains nodes with values smaller than node's value and right subtree of a node contains nodes with values greater than node's value.

b) Give five integers that could be inserted into this tree that would increase the height of this tree [5 integers that can (separately) increase the height of the tree by 1]. Explain your answer.

b) 25, 61, 57, 45, 52 are five integers out of some few others that can be inserted into the tree (obtained in part (a)), that would result in the increase in height of the tree.

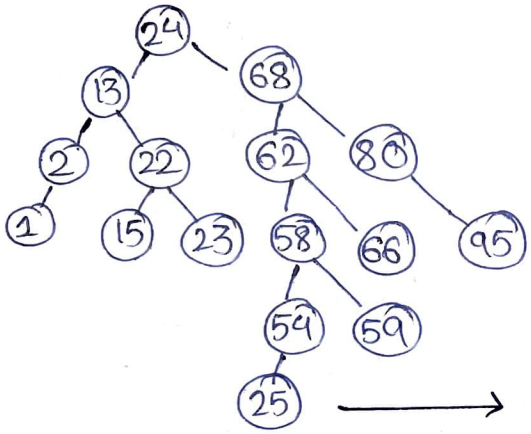
→ The integers that are to be inserted should be chosen such that they result in increase of height of tree by 1 level.

This can be done when the integer chosen is inserted as the child node of the leaf node present at the last level of the tree.
In the case of the obtained tree, these leaf nodes are (54) and (59).
Therefore the integers b/w 24 → 58 can be inserted as the child node of (54) resulting in increase of height and 60 and 61 in (59) as child nodes to increase one level, by following the properties of BST.
The insertion of nodes at other leaf nodes will not result in the increase in height of the tree as 1 level increase, will just make one object addition to the tree at the same level as the current last level of tree.

This can also be explained by drawing the BSTs after the proposed insertions :

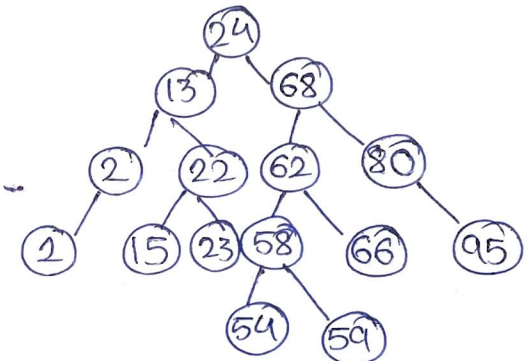
→ **insert 25**

(25 > 24 → right, 25 < 68 → left, 25 < 62 → left, 25 < 58 → left, 25 < 54 → left)



→ **insert 61**

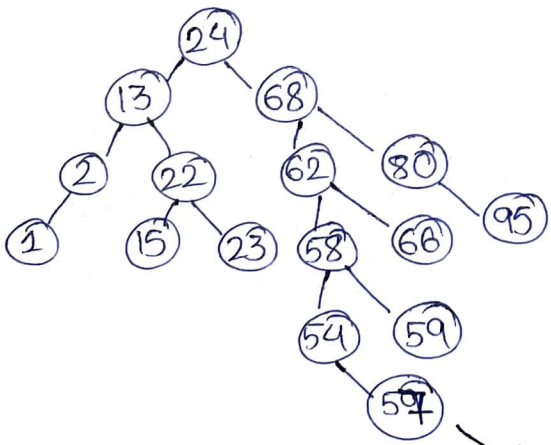
(61 > 24 → right, 61 < 68 → left, 61 < 62 → left, 61 > 58 → right, 61 > 59 → right).



increase in the height of tree by 1 on insertion by following BST properties.

→ **insert 57**

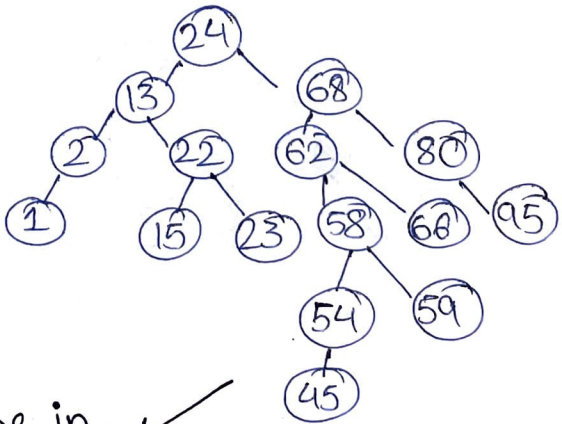
(57 > 24 → right, 57 < 68 → left, 57 < 62 → left, 57 < 58 → left, 57 > 54 → right)



increase in the height of the tree by 1 on insertion by following BST properties

→ **insert 45**

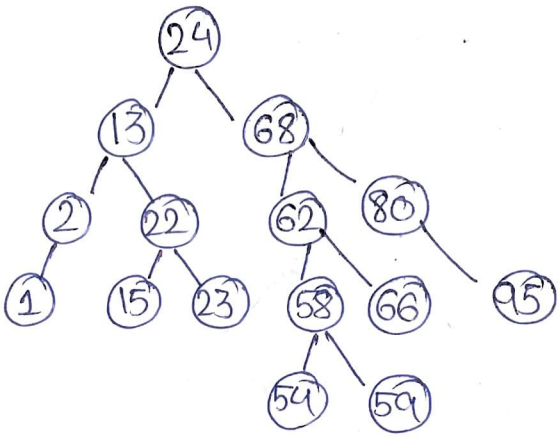
(45 > 24 → right, 45 < 68 → left, 45 < 62 → left, 45 < 58 → left, 45 < 54 → left)



and similarly insertion of 52

c) Remove the root node (from part (a)) four times by copying up the smallest element of the right sub-tree, show each step and the final tree.

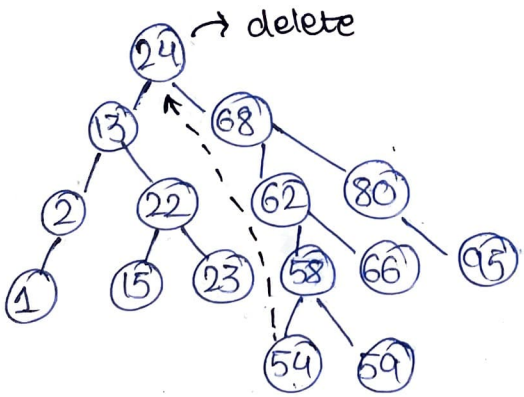
c) Initial BST :

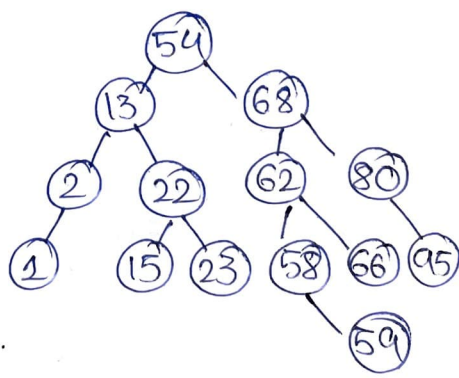


1) Removing the root node first time :

Smallest element of the right sub-tree : 54

→ The successor node does not have any child nodes, so we can delete the root node, replace it with it's successor (54), and as the successor is a leaf node, we can simply free the space allocated to it.

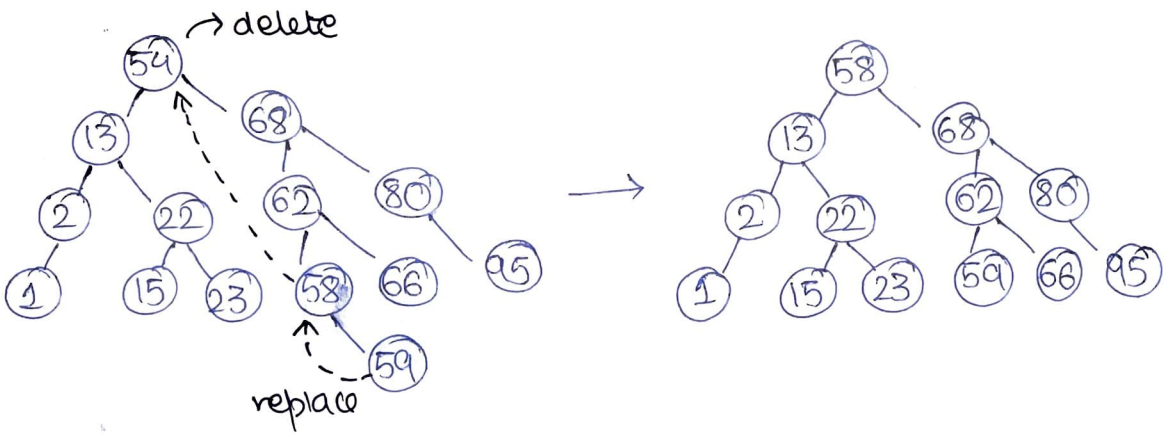




2) Removing the root node second time :

Smallest element of the right sub-tree : 58

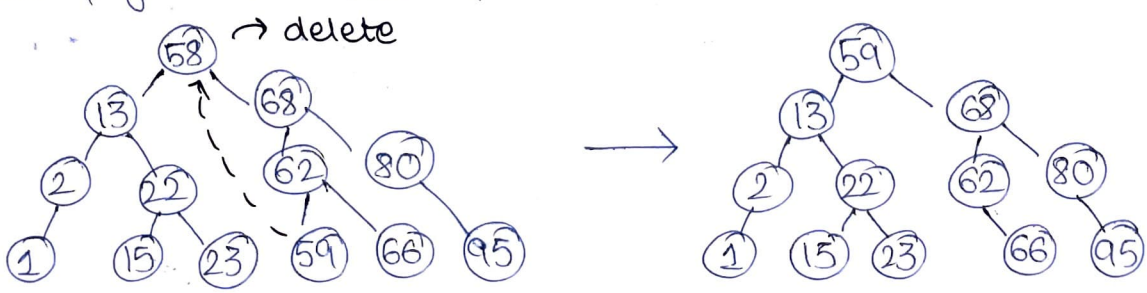
→ The successor node has one right child node. we will delete the root node and replace it with the successor node 58. And we will replace the successor node with it's child node and delete the space allocated to it.



3) Removing the root node third time :

Smallest element of the right sub-tree : 59

→ The successor node does not have any child nodes, so we can delete the root node, and replace it with it's successor (59), and as the successor is a leaf node, we can simply free the space allocated to it.

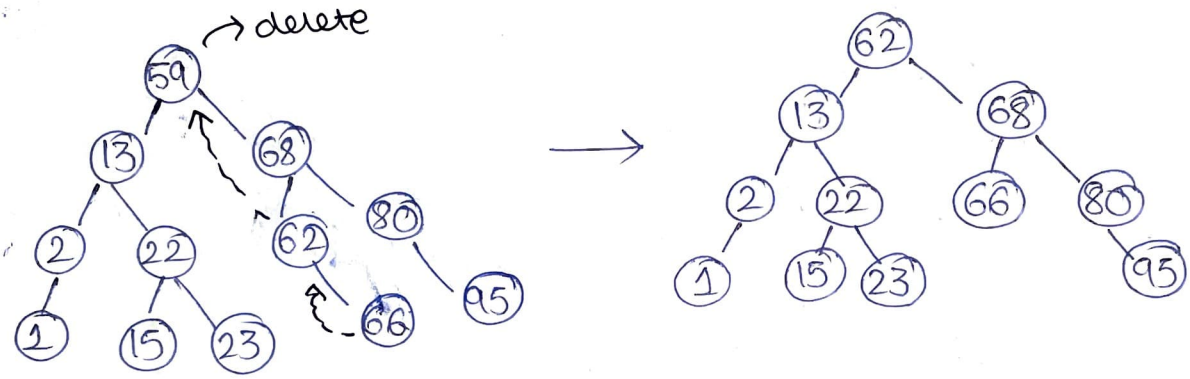


replace and free the allocated space

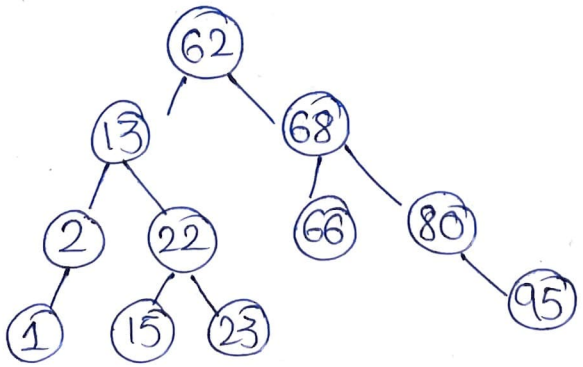
4) Removing the root node fourth time:

Smallest element of the right sub-tree : 62

→ The successor node has one right child node. We will delete the root node and replace it with the successor node 62. And we will replace the successor node with it's child node and free the space allocated to it.



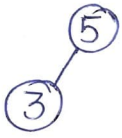
Final tree obtained after four deletion operations at the root node and replacing with the smallest element of right sub-tree are:



Ques 3 Insert the following n objects, in the order given, into a binary min-heap and place your answer into an array. [5,3,9,7,2,4,6,1,8]

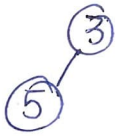
1) Insert 5

2) Insert 3

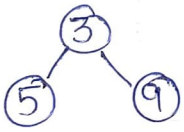


not following the min-heap property

heapify operation

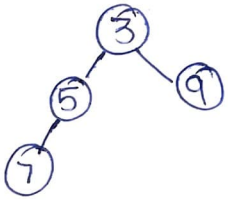


3) Insert 9



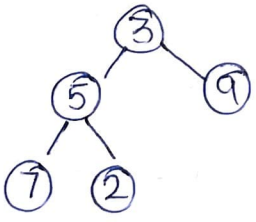
maintaining the binary tree property and min-heap property

4) Insert 7



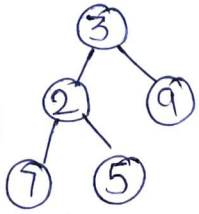
maintaining the complete binary tree and min-heap property

5) Insert 2

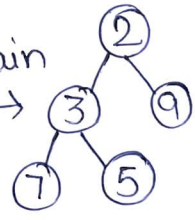


not following min-heap property

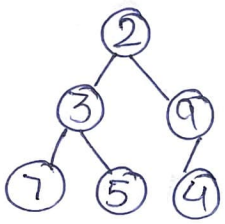
heapify operation



again

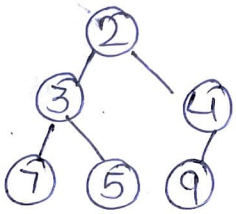


6) Insert 4

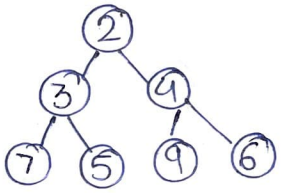


not following min-heap property →

heapify operation

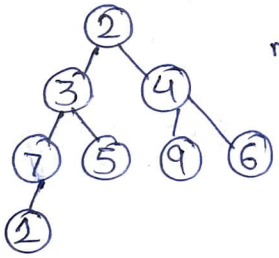


7) Insert 6



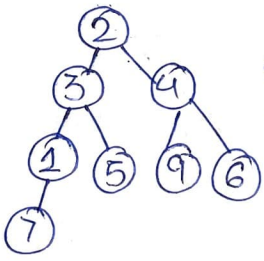
maintaining the complete binary tree and min-heap property

8) Insert 1

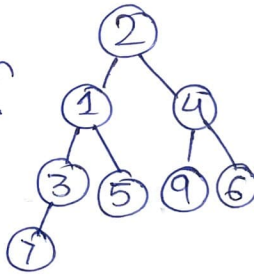


not following min-heap property →

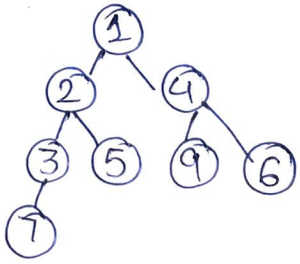
heapify operation



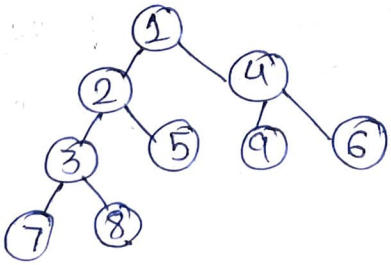
again →



again ↓

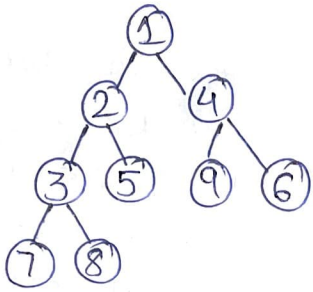


9) Insert 8



maintaining the complete binary tree and min-heap property

Binary min-heap obtained:



Array implementation:

The array representation of a binary min-heap is done as:

→ Root element will be at A[0]

→ A[(i-1)/2] will return the parent node for the element at ith index in the array.

→ A[(2*i)+1] will return the left child node for the element at ith index in the array.

→ A[(2*i)+2] will return the right child node for the element at ith index in the array.

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

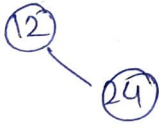
Question 4 Answer the following questions regarding AVL trees :

a) Insert the following sequence of elements into an AVL tree, starting with an empty tree (show each step) : 12, 24, 14, 27, 35, 17, 19, 22.

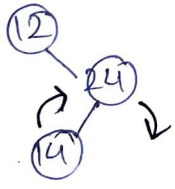
a) AVL is a self-balancing binary search tree where the difference b/w heights of left and right sub-trees cannot be more than 1 for every node.

1) Insert 12

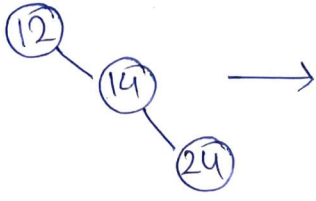
2) Insert 24



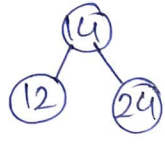
3) Insert 14



imbalance → fix AVL properties (right-left rotation)

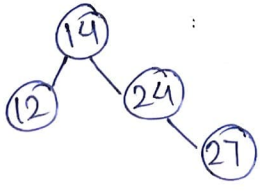


(right rotation)

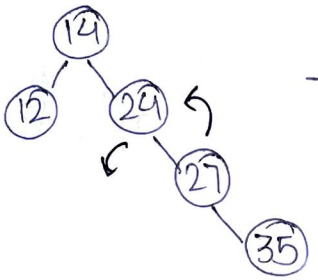


(left rotation)

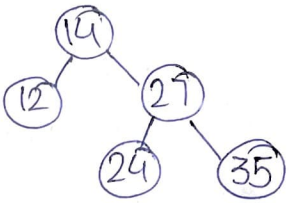
4) Insert 27



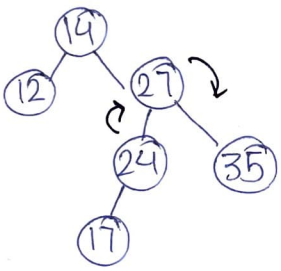
5) Insert 35



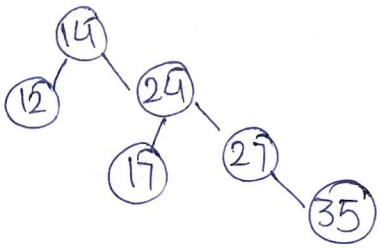
imbalance → fix AVL properties (left rotation)



6) Insert 17

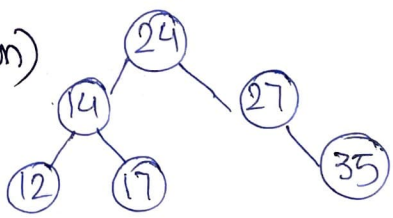


imbalance → fix AVL properties (right-left rotation)

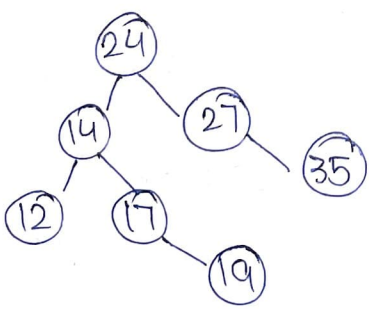


(right rotation)

(left rotation)

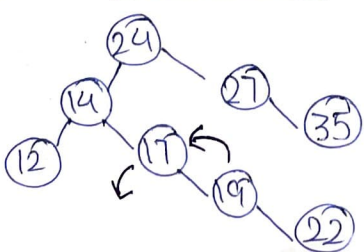


7) Insert 19

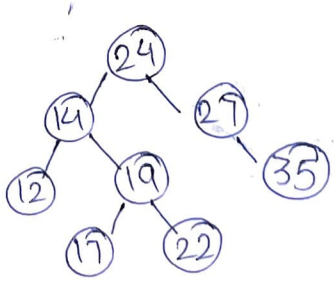


Final AVL tree after insertion of all the elements.

8) Insert 22

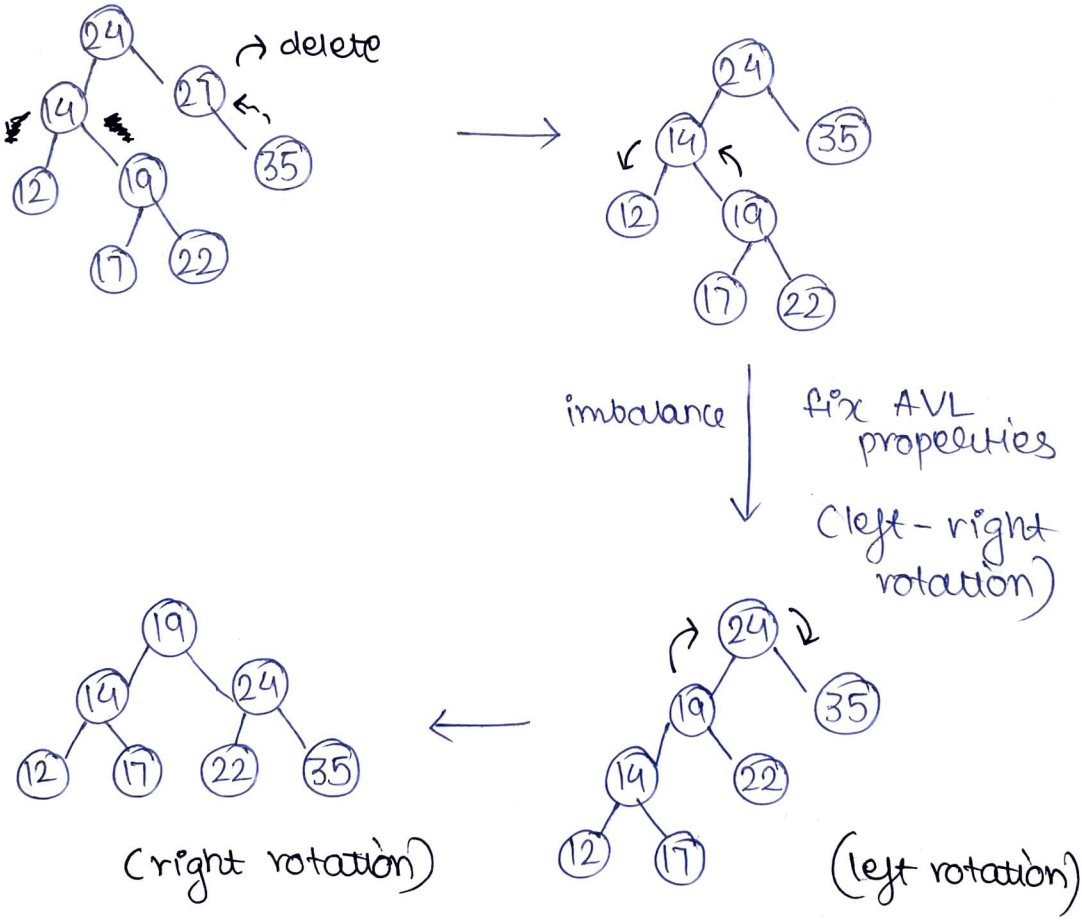


imbalance → fix AVL properties (left rotation)

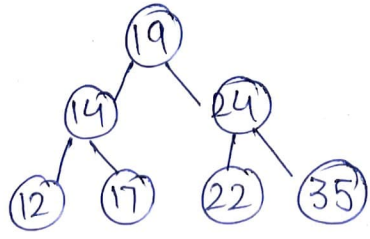


b) Delete 27 in the AVL tree that you got (show each step).

b) First performing standard binary search tree deletion operation in the obtained tree.



→ Final balanced AVL Tree after deletion of (27) element



c) What maximum difference in heights b/w the leafs of a AVL tree is possible?

c) At every level in the AVL tree, a tree can be formed with maximum difference between the heights of the left and right subtrees to be 1. If there are n nodes in the tree, the maximum no. of levels or height of AVL tree can be given by log(n) and since the difference b/w any leaves in the tree cannot exceed that so maximum height difference between the leafs of a AVL tree that is possible is log(n).

Ques 5 Answer the following questions regarding Red Black trees:

a) What are the operations that could be performed in O(logn) time complexity by red-black tree?

a) The following operations could be performed in O(logn) time complexity by red black tree:

→ Insertion
→ Deletion
→ Finding predecessor
→ Finding successor.

Some restrictions have to be imposed to achieve logarithmic time complexities, that are

→ root property should be black
→ every leaf is black
→ children of red node are black
→ all leaves have same black.

These restrictions are imposed to achieve self-balancing red-black trees with logarithmic complexities of time for insertion, deletion and search operations.

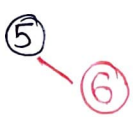
b) Insert the following sequence into a red black tree (show each step):

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

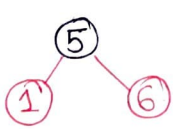
b) Red-Black Tree is a kind of self balancing binary search tree where each node has an extra bit and that bit is interpreted as the colour (red or black) which are used to make sure that the tree remains balanced during insertions and deletions.

1) Insert 5

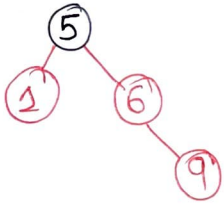
2) Insert 6



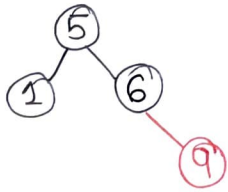
3) Insert 1



4) Insert 9



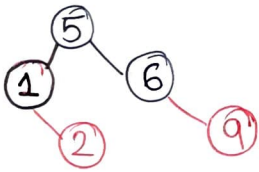
Two consecutive red nodes



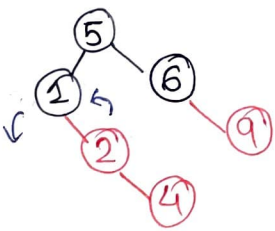
↓

The new node's parent sibling's colour is red and parent's parent is root node so we use RECOLOUR to make it Red Black Tree

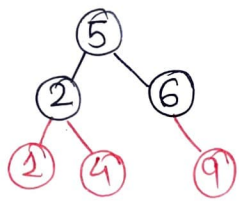
5) Insert 2



6) Insert 4



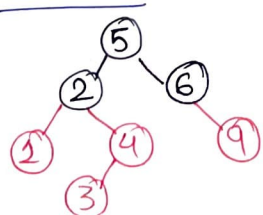
Two consecutive red nodes



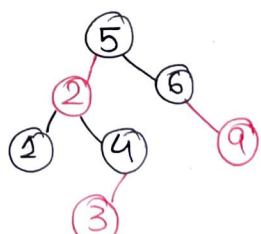
The newnode's parent sibling is NULL. So we need to perform rotation

Here we perform left rotation and Recolour.

7) Insert 3

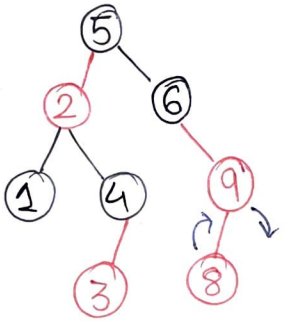


Two consecutive red nodes



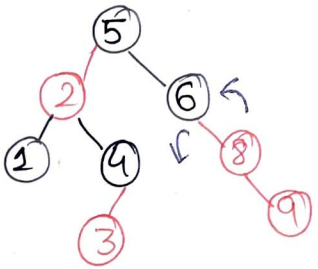
The new nodes's parent sibling is red and parent's parent is not root node so we perform Recolour & Recheck

8) Insert 8

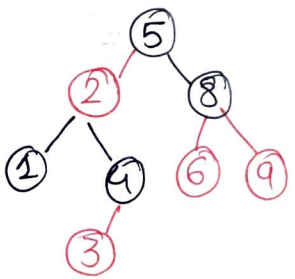


Two consecutive red nodes

The new node's parent sibling is NULL. So we need to perform rotation. Here we right-left rotation and recolour.

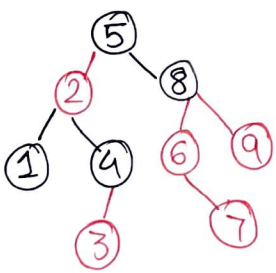


[right rotation]



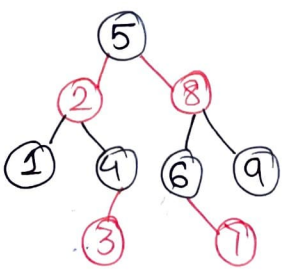
[left rotation and recolour]

9) Insert 7



Two consecutive red nodes

The new node's parent sibling is red and parent's parent is not root node so we need to perform RECOLOUR and recheck.



Final Red Black Tree after inserting all elements.