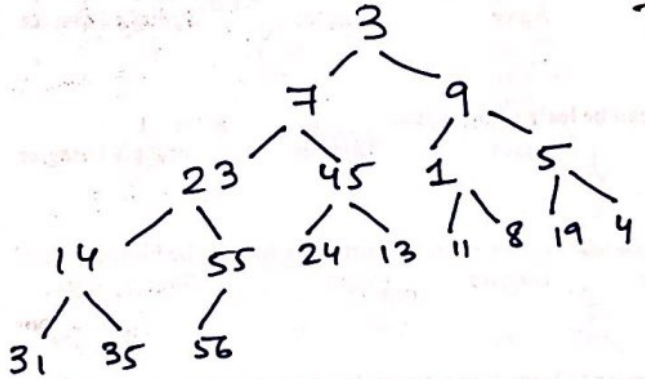
Data : {3, 7, 9, 23, 45, 1, 5, 14, 55, 24, 13, 11, 8, 19, 4, 31, 35, 56}

Binary Tree :-

For given array, elements from left in array will be filled in the tree level wise starting from level 0. Here, if parent node is at index 'i' in array, then left child of that node is at index $(2i+1)$ and right child of that node is at index $(2i+2)$ in array. Using this concept, we insert left & right nodes by choosing parent node. The 1st element in array is inserted as root node at level 0 in tree, then traverse the array and for every node 'i' insert its both left & right child in the tree recursively.

Tree depth = 4.



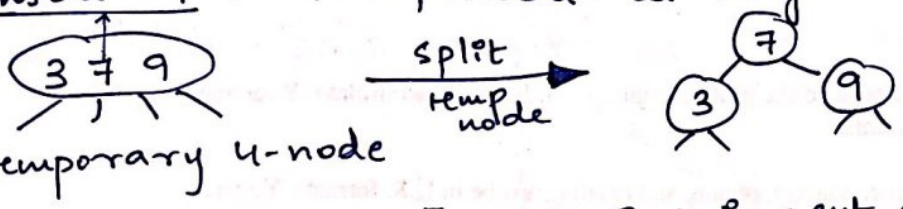
2-3 Tree :-

For a given non-leaf node, it can have either 2 or maximum of 3 child. All the leaf nodes should be at same level with all the data present in sorted order. If a node contains 1 data element, then it will have 2 children and if a node contains 2 data elements, then it will have 3 children / sub-trees. The created tree is thus ordered and balanced in nature.

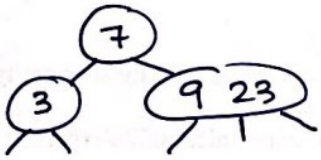
(i) Insert 3 :

(ii) Insert 7 : 7 > 3, insert at right of 3

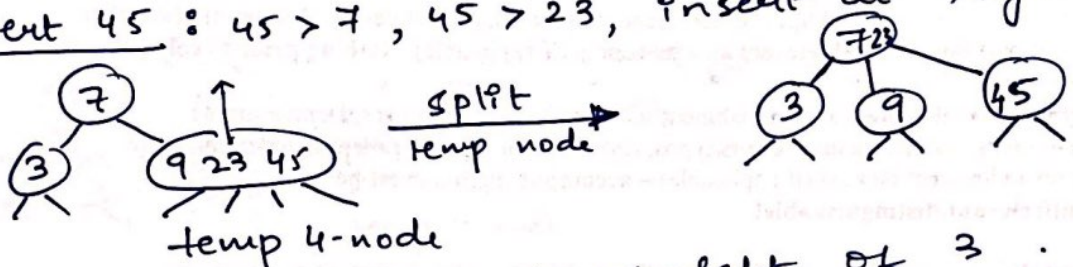
(iii) Insert 9 : 9 > 7, insert at right of 7



(iv) Insert 23 : 23 > 7, 23 > 9 ∴ insert at right of 9

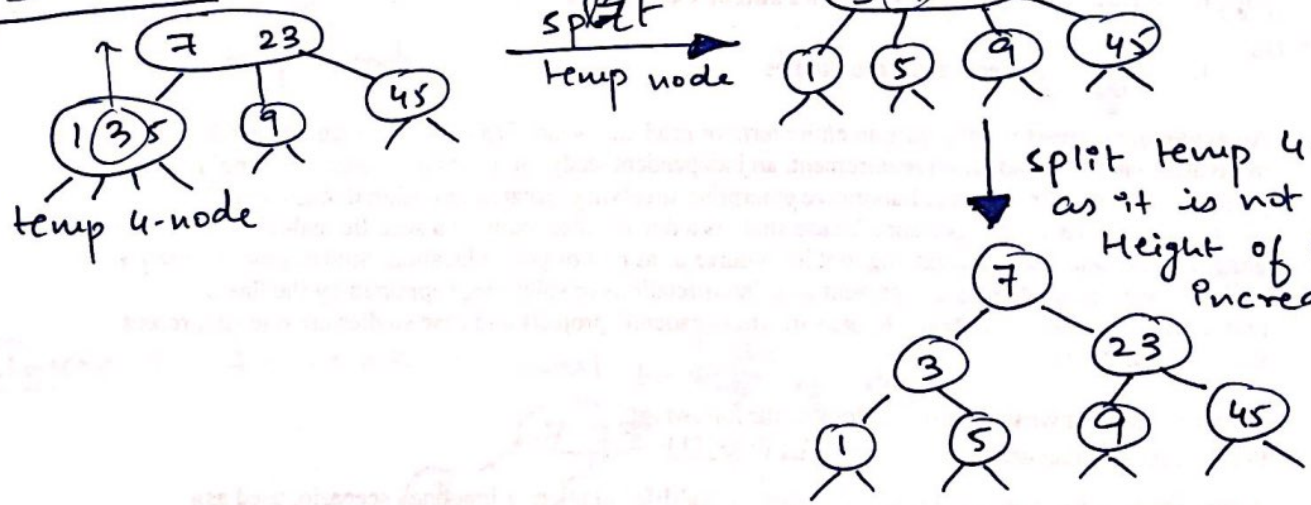


(v) Insert 45 : 45 > 7, 45 > 23, insert at right of 23.

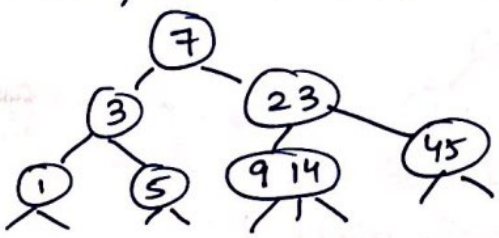


(vi) Insert 1 : 1 < 3, insert at left of 3.

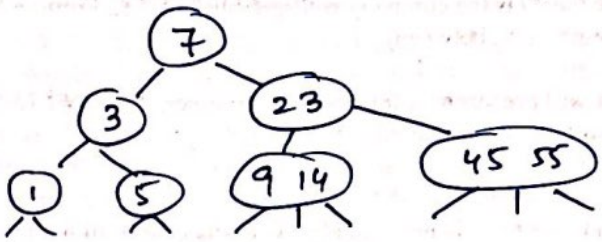
(vii) Insert 5 : 5 < 7, 5 > 3 ∴ insert at right of 3.



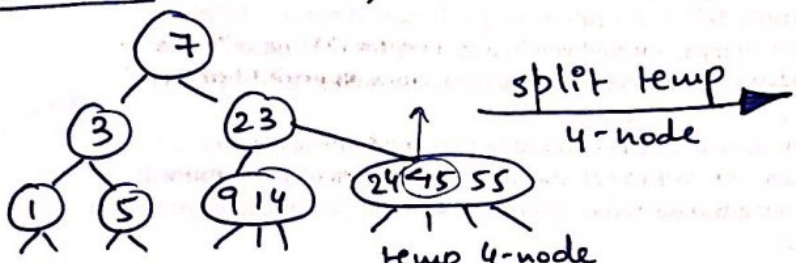
(viii) Insert 14: 14 > 7, 14 < 23, 14 > 9 ∴ insert 14 at right of 9.



(ix) Insert 55: 55 > 7, 55 > 23, 55 > 45 ∴ insert at right of 45

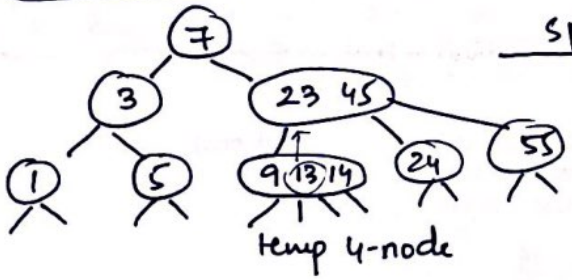


(x) Insert 24: 24 > 7, 24 > 23, 24 < 45 ∴ insert at left of 45.

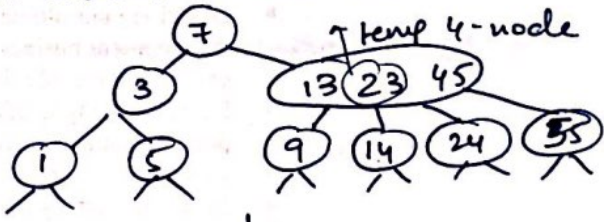


split temp 4-node

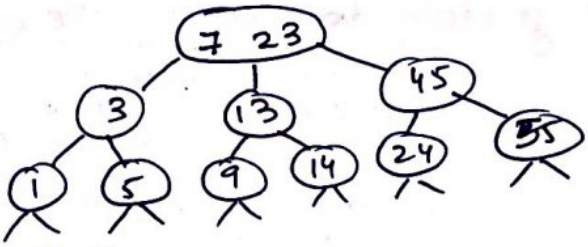
(xi) Insert 13: 13 > 7, 13 < 23, 13 > 9 & < 14 ∴ insert in between.



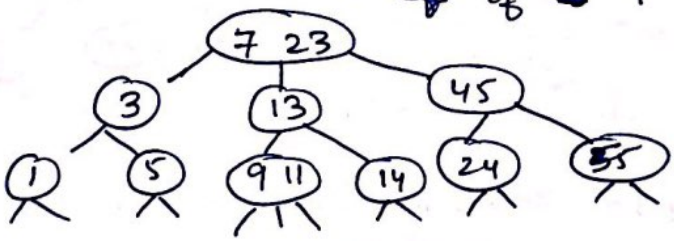
split temp 4-node



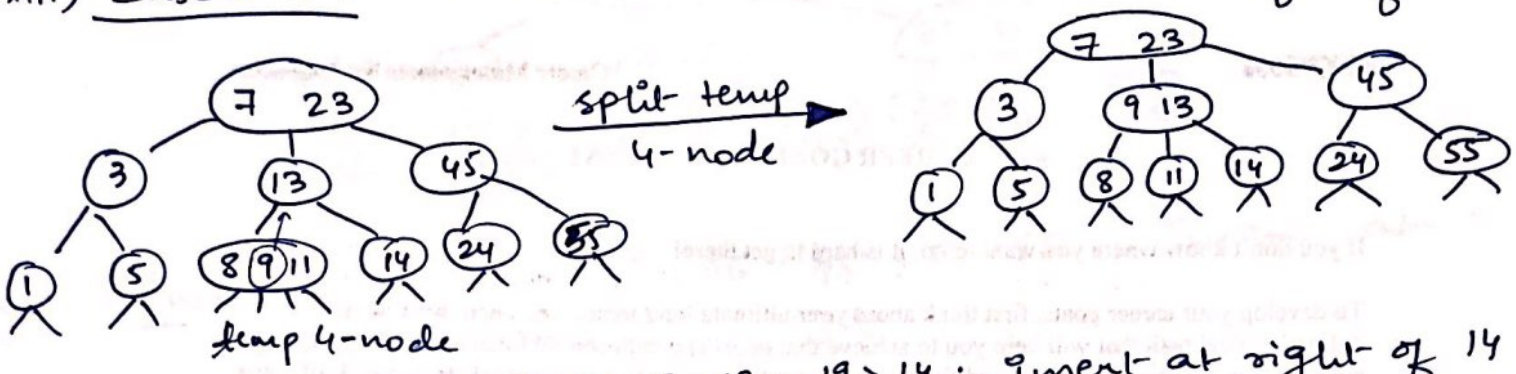
split temp 4-node.



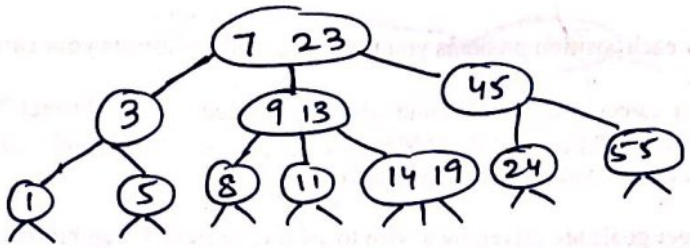
(xii) Insert 11: 7 < 11 < 23 ∴ insert to right of 9



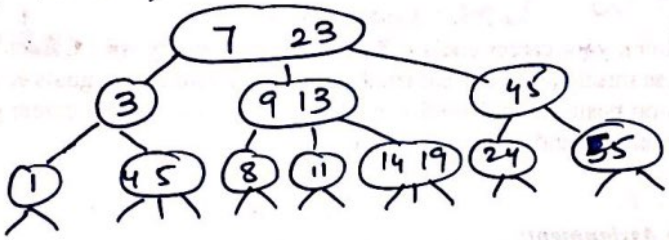
(xiii) Insert 8 : $7 < 8 < 23$; $8 < 13$; $8 < 9$ ∴ insert at left of 9



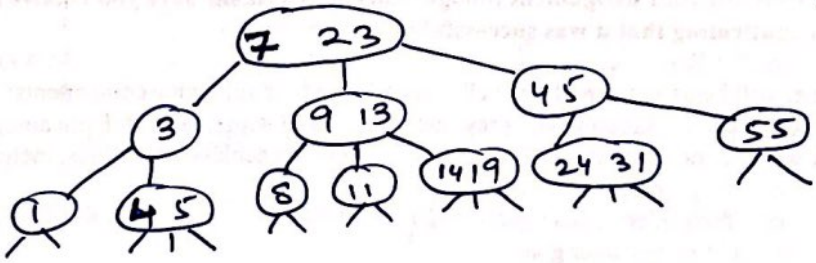
(xiv) Insert 19 : $7 < 19 < 23$; $19 > 13$; $19 > 14$ ∴ insert at right of 14



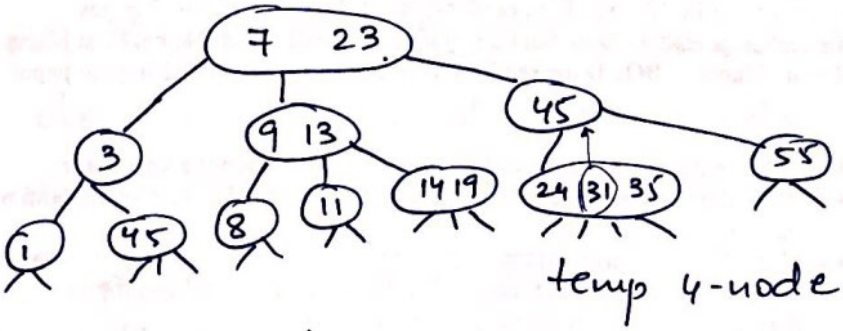
(xv) Insert 4 : $4 < 7$; $4 > 3$; $4 < 5$ ∴ insert at left of 5 .



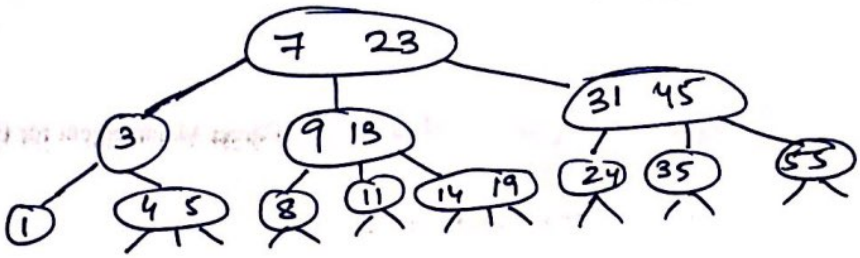
(xvi) Insert 31 : $31 > 23$; $31 < 45$; $31 > 24$ ∴ insert at right of 24 .



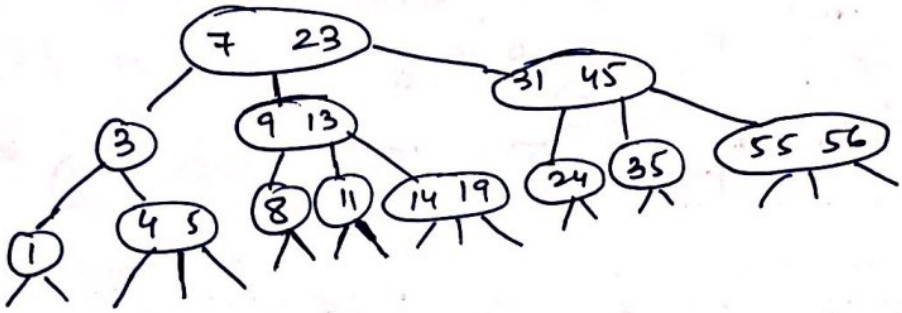
(xvii) Insert 35 : $35 > 23$; $35 < 45$; $35 > 31$ ∴ insert at right of 31



split temp. 4-node & promote 31 to parent node as it is not 2-3 tree



(xviii) Insert 56 : 56 > 23 ; 56 > 45 ; 56 > 55 ∴ insert at right of 55



Hence, 2-3 tree is constructed with height of tree = 2

2-3-4 Tree :-

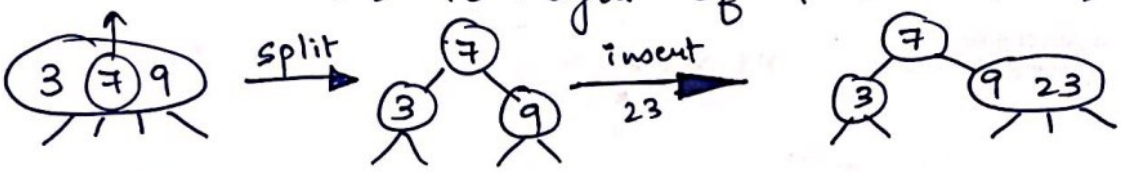
- If there are 2 nodes, it must contain 1 data element
- If there are 3 nodes, it must contain 2 data elements.
- If there are 4 nodes, it must contain 3 data elements.
- leaf node can have either 1, 2 or 3 data elements.
- During insertion, while traversing tree from root to leaf, split 4-nodes as it is encountered & move the middle element to parent node which cannot be a 4-node to accomodate another element.
- The above property of tree makes 2-3-4 tree insertion algorithm more efficient as it avoids the return path after reaching the leaf.

(i) Insert 3 :

(ii) Insert 7 : 7 > 3, insert at right of 3

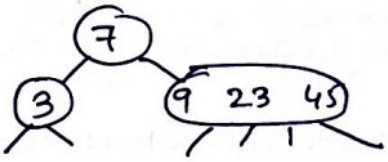
(iii) Insert 9 : 9 > 7, insert at right of 7

(iv) Insert 23 : 4-node encountered, ∴ split it & then insert 23 to right of 9 as 23 > 7; 23 > 9.

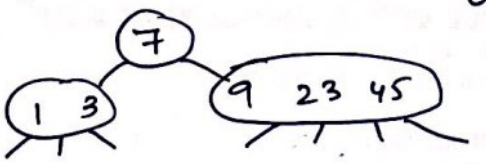


Height of tree increases by 1.

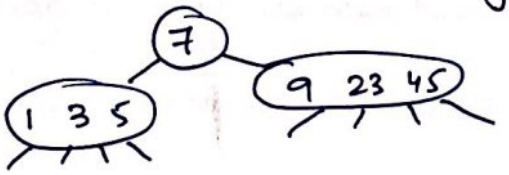
(v) Insert 45 : 45 > 7 ; 45 > 23 ; ∴ insert to right of 23



(vi) Insert 1 : 1 < 7 ; 1 < 3 ∴ insert to left of 3



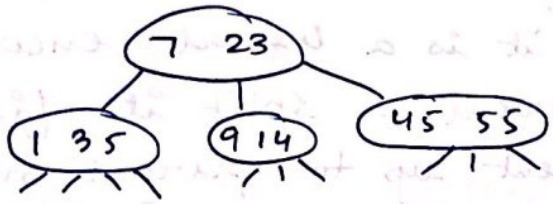
(vii) Insert 5 : 5 < 7 ; 5 > 3 ∴ insert at right of 3



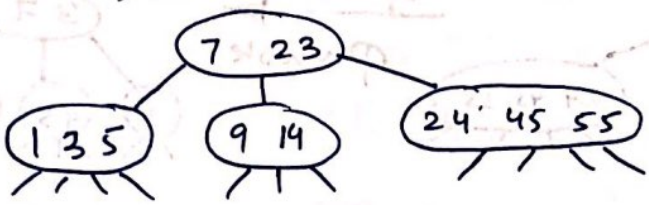
(viii) Insert 14 : 14 > 7 ; 9 < 14 < 23 so insert in between 9 and 23.

4-node encountered so split and move 23 up to parent node

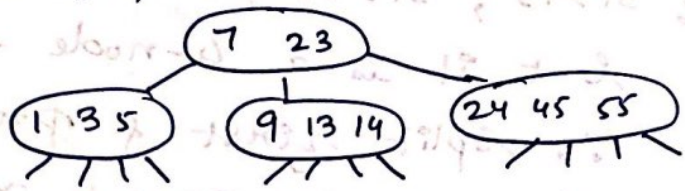
(ix) Insert 55 : 55 > 23 ; 55 > 45 ∴ insert to right of 45.



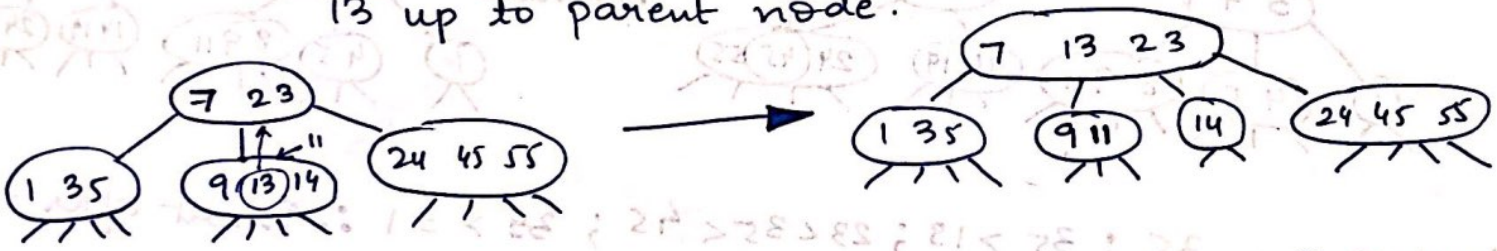
(x) Insert 24 : 24 > 23 ; 24 < 45 ∴ insert to left of 45.



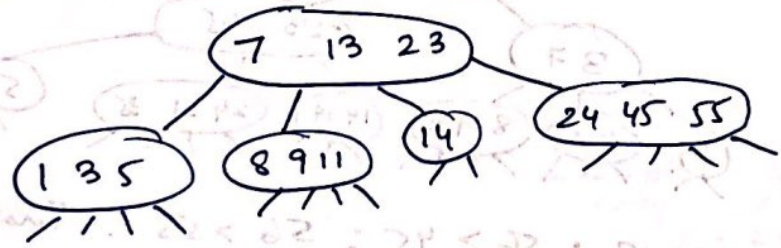
(xi) Insert 13 : 7 < 13 < 23 ; 9 < 13 < 14 ∴ insert between 9 & 14.



(xii) Insert 11 : 7 < 11 < 23 ; 9 < 11 < 13 ∴ insert there but (9, 13, 14) is a 4-node so first split it and promote 13 up to parent node.

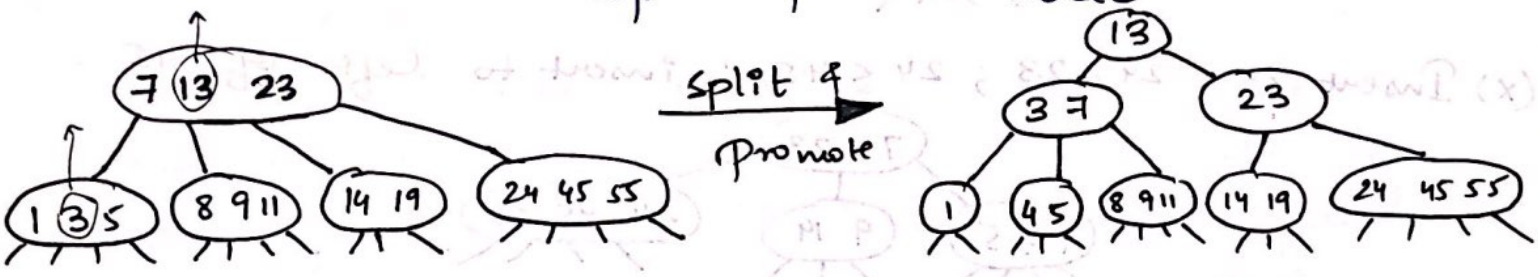


(xiii) Insert 8 : 7 < 8 < 13 ; 8 < 9 ∴ insert to left of 9.

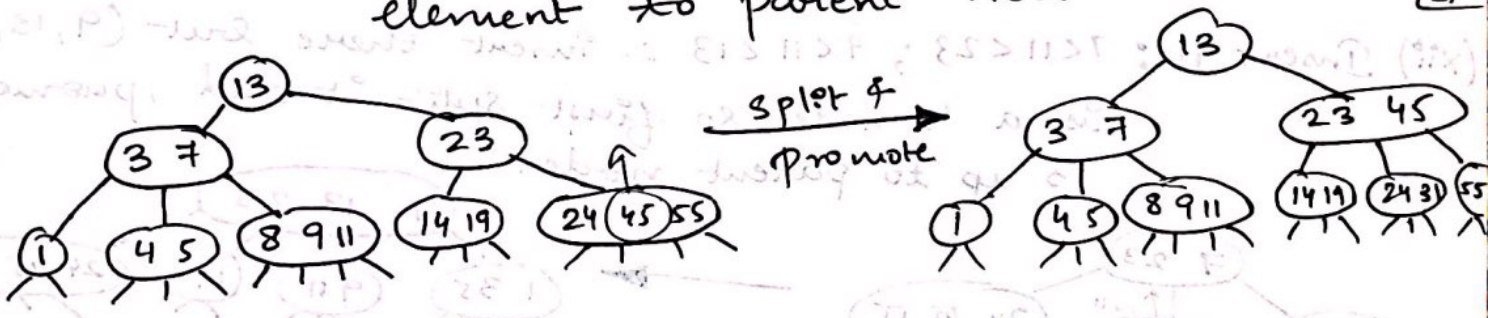


(xiv) Insert 19 : 13 < 19 < 23 ; 19 > 14 ∴ insert to right of 14

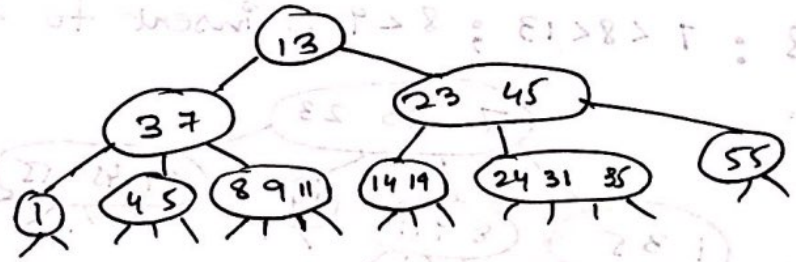
(xv) Insert 4 : $4 < 7$; $3 < 4 < 5$ ∴ we need to insert there but it is a 4-node encountered while tree traversal ∴ split it first & promote middle element up to parent node.



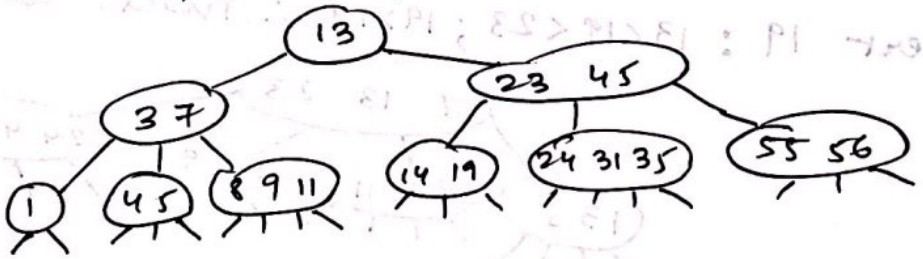
(xvi) Insert 31 : $31 > 13$; $31 > 23$; $24 < 31 < 45$ ∴ Insert in between but it is a 4-node encountered & ∴ split first & promote middle element to parent node. Then insert 31



(xvii) Insert 35 : $35 > 13$; $23 < 35 < 45$; $35 > 31$ ∴ Insert after 31.



(xviii) Insert 56 : $56 > 13$; $56 > 45$; $56 > 55$ ∴ Insert to right of 55.



∴ 2-3-4 tree is constructed with height = 2

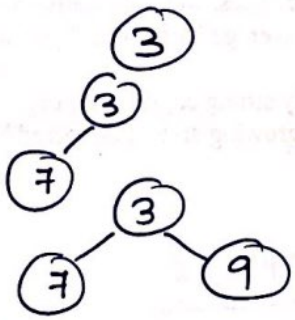
Binary Heap Tree :-

A binary heap is a complete binary tree which means all levels of tree are completely filled except possibly the last level. The nodes are filled from left to right. The value stored in each node is either greater than or equal to its child node in case of a max heap OR less than or equal to its child node in case of a min heap.

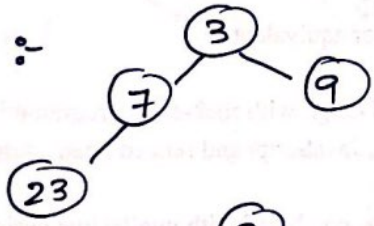
(i) Insert 3 :-

(ii) Insert 7 :-

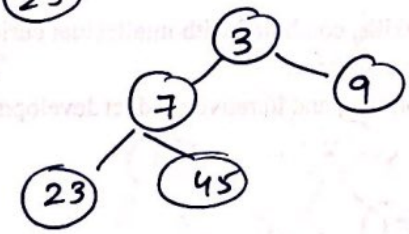
(iii) Insert 9 :-



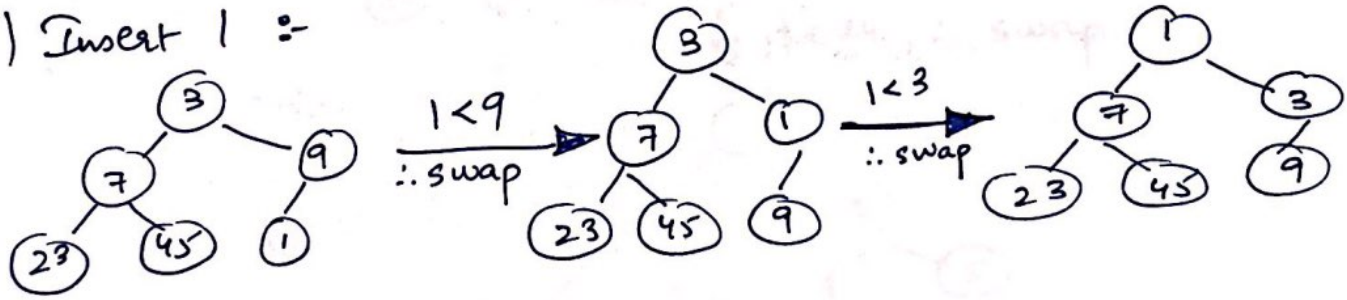
(iv) Insert 23 :-



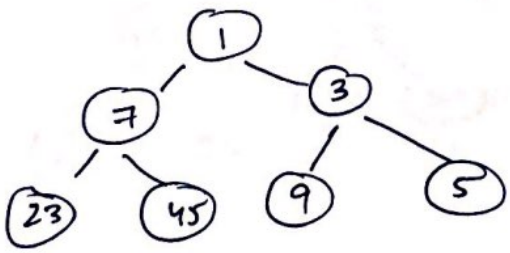
(v) Insert 45 :-



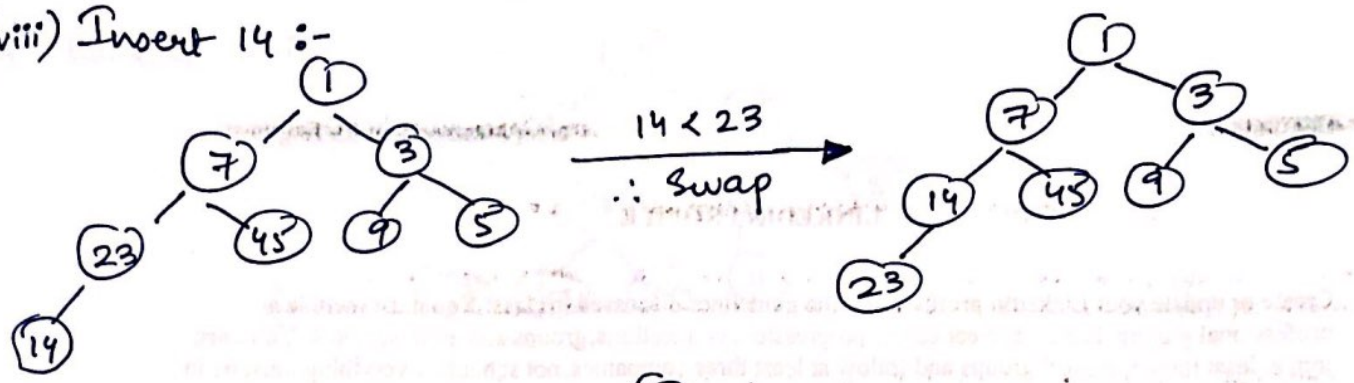
(vi) Insert 1 :-



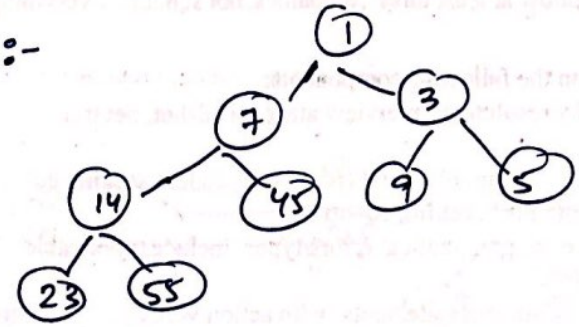
(vii) Insert 5 :-



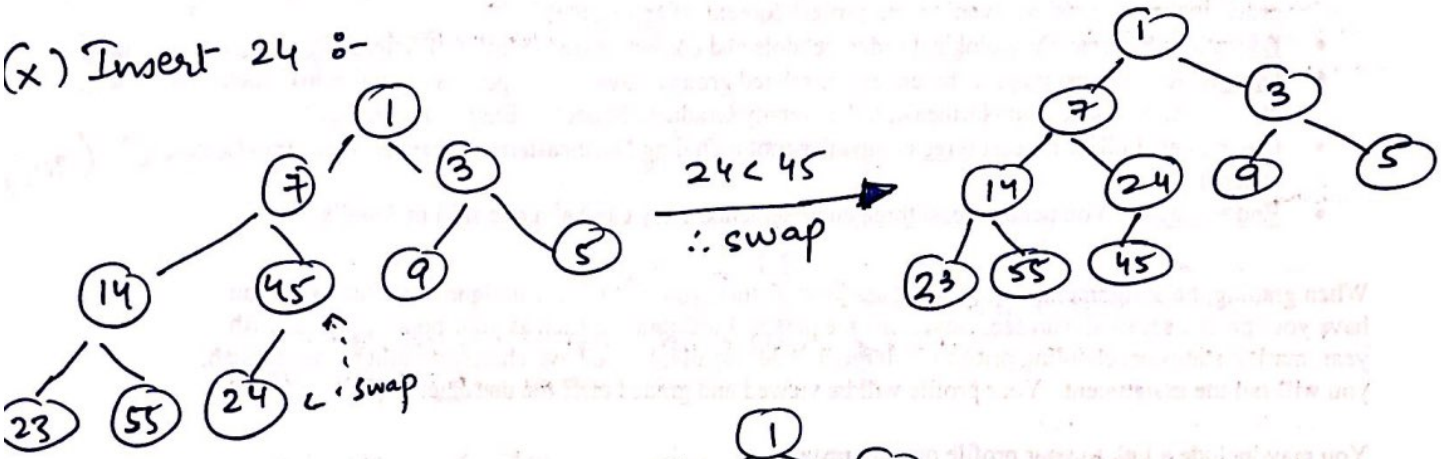
(viii) Insert 14 :-



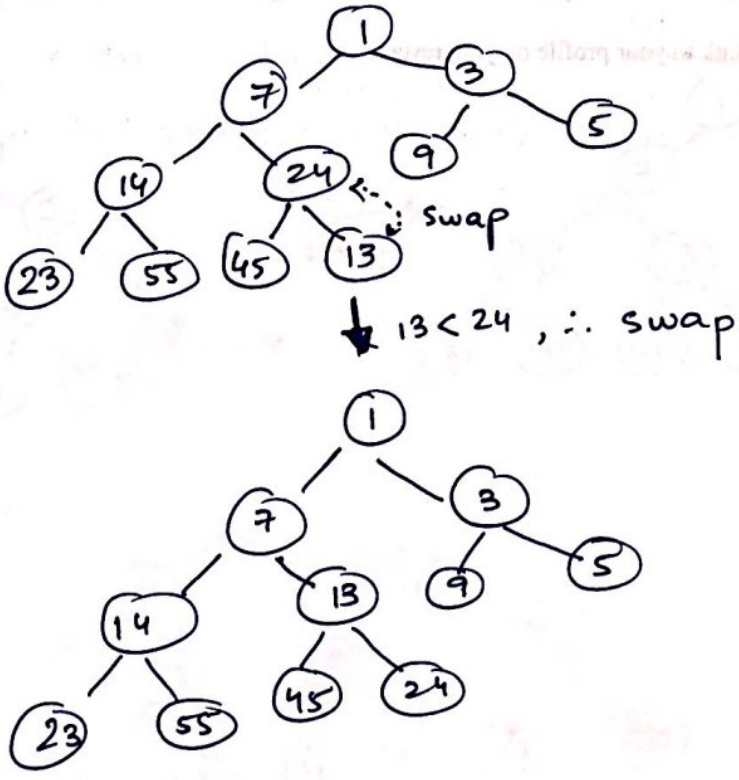
(ix) Insert 55 :-



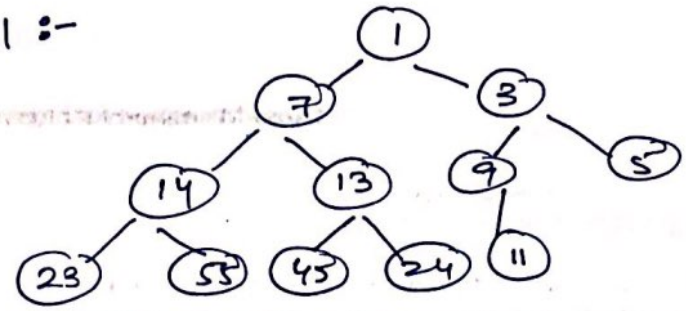
(x) Insert 24 :-



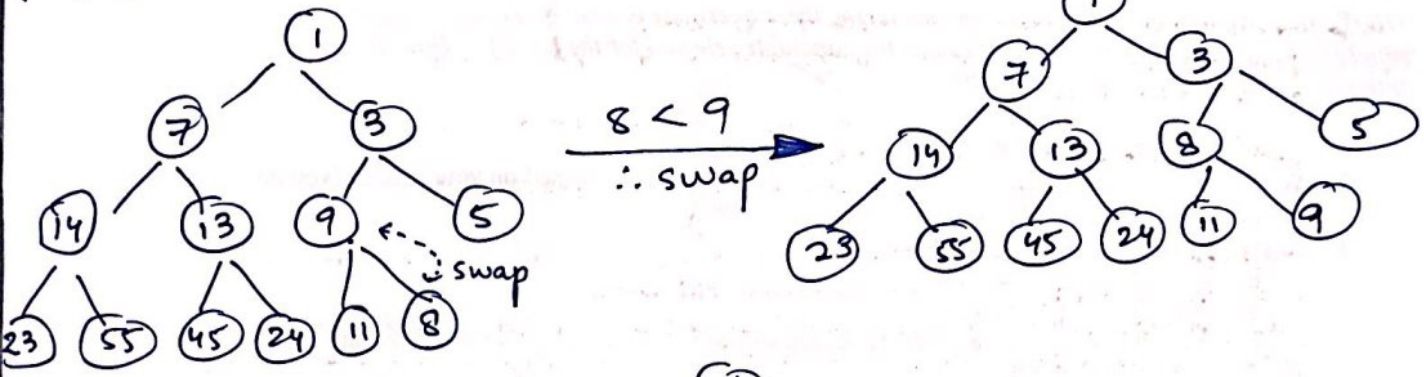
(xi) Insert 13 :-



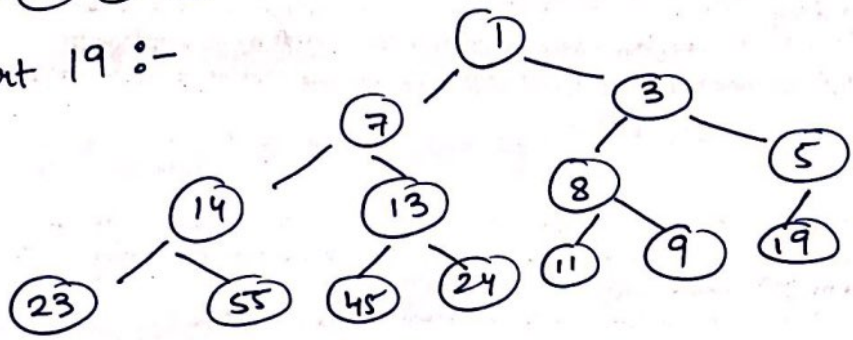
(x) Insert 11 :-



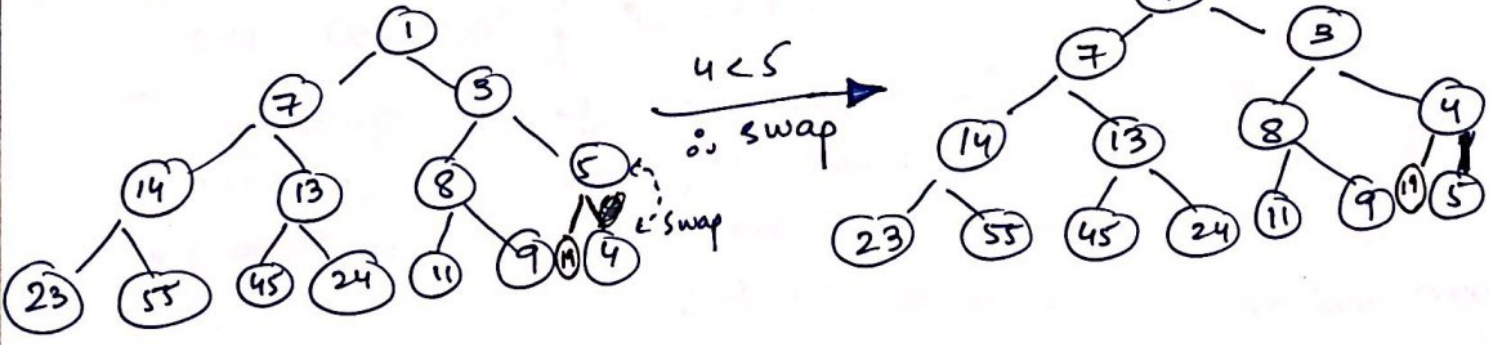
(xi) Insert 8 :-



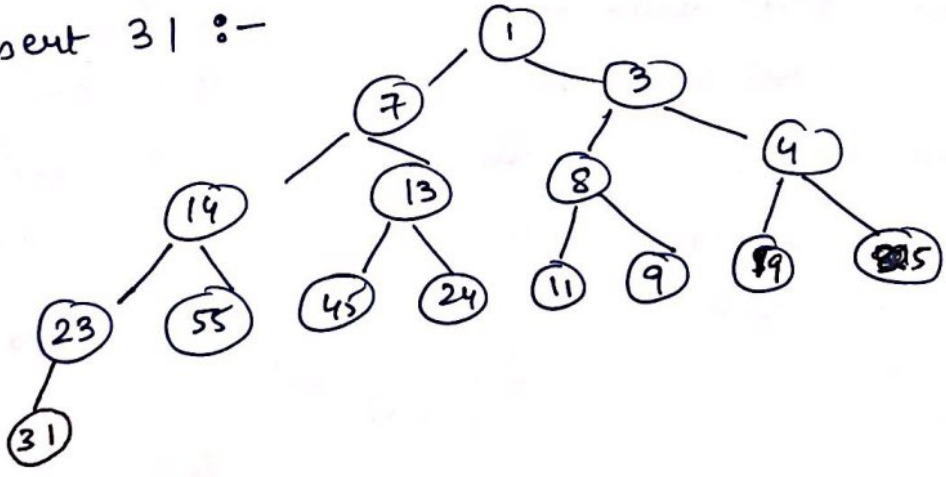
(xii) Insert 19 :-



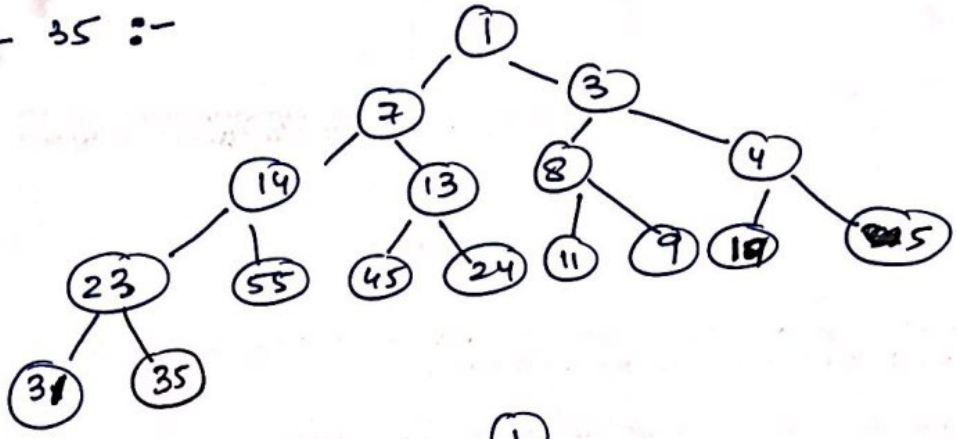
(xiii) Insert 4 :-



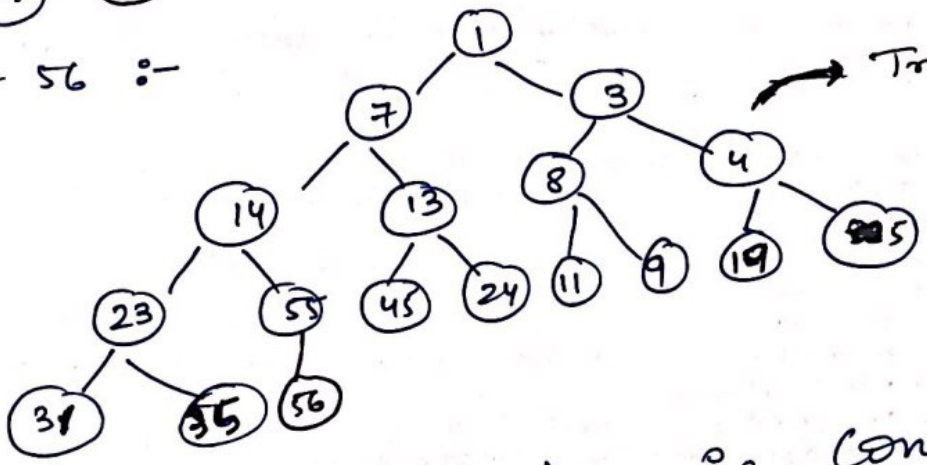
(xiv) Insert 31 :-



(xv) Insert 35 :-



(xvi) Insert 56 :-



Thus, binary min heap tree is constructed.

Time Complexity of Binary Tree – Since we have to traverse through all elements for insertion, worst case complexity is $O(n)$.

Time Complexity of 2-3 Tree – Since 2-3 tree is balanced in nature, the worst case complexity is $O(\log n)$ as height of tree (2-3 tree) is $O(\log n)$.

Time Complexity of 2-3-4 Tree – An insertion requires $O(\log n)$ node splits where each split takes constant time. Hence, worst-case time complexity is $O(\log n)$

Time Complexity of Binary Heap Tree – Since we need at most 1 swap on each level of a heap on the path from inserted node to the root, worst case runtime of the algorithm is $O(\log n)$.

However, insertion into 2-3-4 tree can give better efficiency than 2-3 trees. For 2-3 trees, insertion algorithm traces path from root to leaf and then backs up from leaf as it splits nodes. To avoid this return path after reaching a leaf, insertion algorithm of 2-3-4 trees splits the 4-nodes as soon as it encounters them as the tree is being traversed from root to a leaf. Thus, a 2-3-4 tree insertion algorithm is preferred.