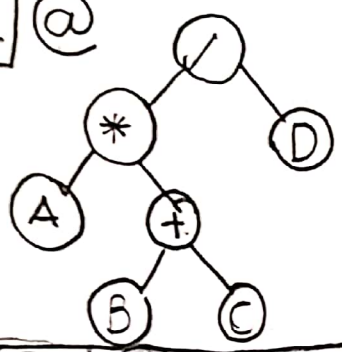
N. Onur Vural - 21902330 - CS202 - HW2 - SEC02

Q1

a)



Traversels:

Preorder: /*A+BCD (Root, Left, Right)

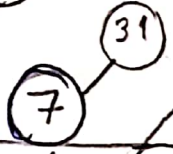
Inorder: A*B+C/D (Left, Root, Right)

PostOrder: ABC+*D/ (Left, Right, Root)

b)

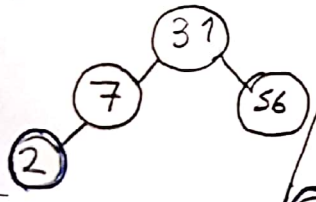
1) Insert 31

2) Insert 7

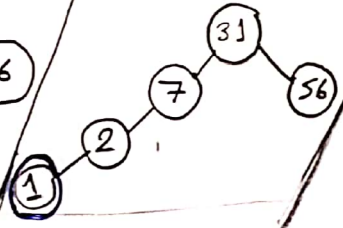


3) Insert 56

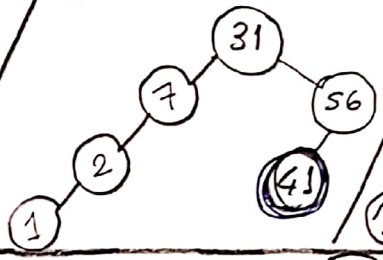
4) Insert 2



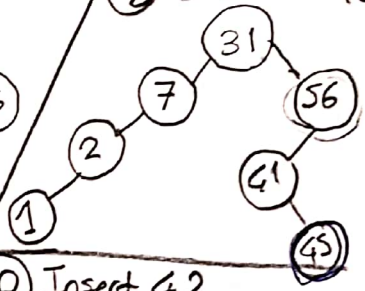
5) Insert 1



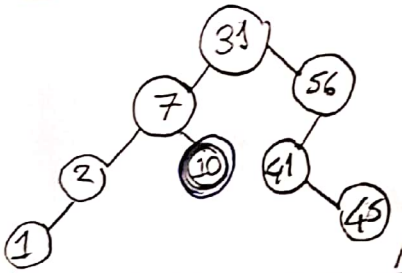
6) Insert 41



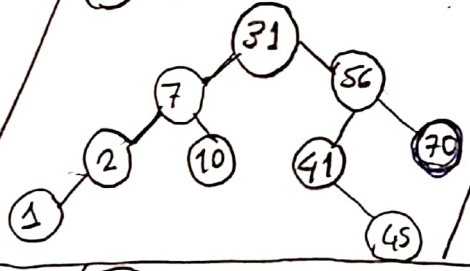
7) Insert 45



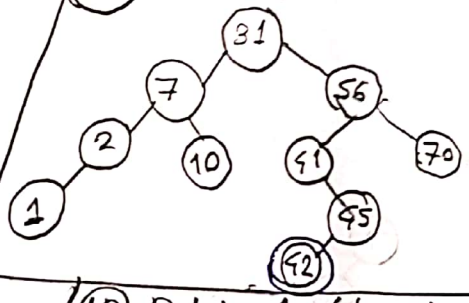
8) Insert 10



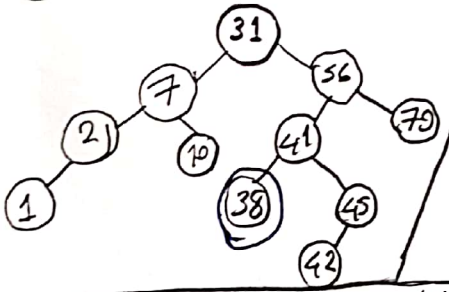
9) Insert 70



10) Insert 42

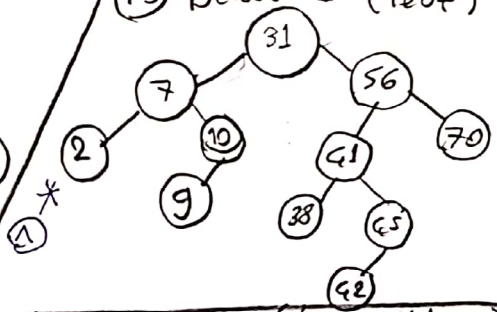


11) Insert 38

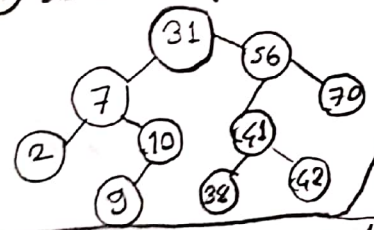


12) Insert 9

13) Delete 1 (leaf)

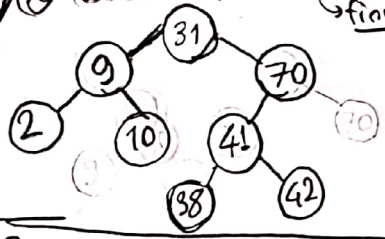


14) Delete 45 (one child)

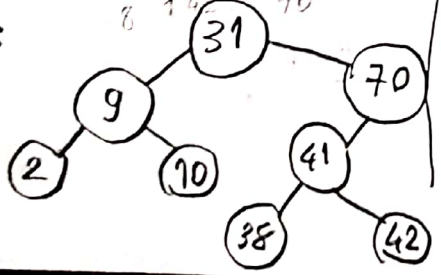


15) Delete 56 (two children)

16) Delete 7 (two children) → find successor



Finalized:



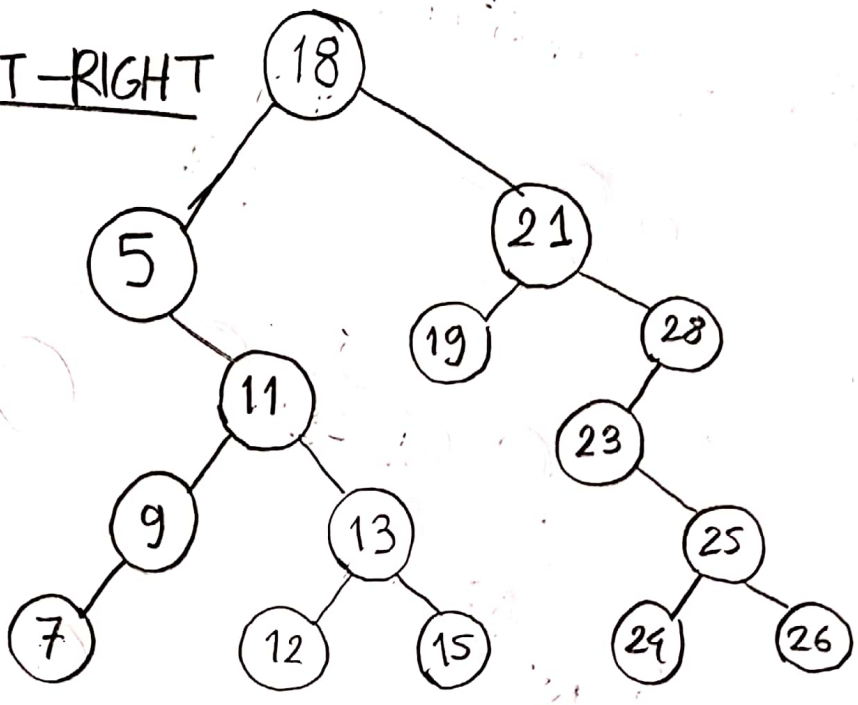
Pre: 31, 9, 2, 10, 70, 41, 38, 42

In: 2, 9, 10, 31, 38, 41, 42, 70

Post: 2, 10, 9, 38, 42, 41, 70, 31

C) given → preorder : 18, 5, 11, 9, 7, 13, 12, 15, 21, 19, 28, 23, 25, 24, 26

ROOT – LEFT – RIGHT



Postorder:

7, 9, 12, 15, 13, 11, 5, 19, 24, 26, 25, 23, 28, 21, 18

Question-3

- In **levelorderTraverse**, the traversel is done by the help of using a queue named **NodeQueue.** Essentially the **NodeQueue** is used to hold and remember two children of the current node that is being investigated. The algorithm begins the processing by first node, in other words root, and adding it to the front of queue (by enqueue() function). As a following step, it begins to get the BinaryNode from queue (by getFront(QueueItemType& queueFront) function), prints it's content (item), removes it from the queue (by dequeue() function) and proceeds by adding left and right children of that particular node to the queue (by enqueue() function) in an iterative fashion. This iteration continues until the queue becomes empty, which suggests that there is no node left in the binary search tree to check. The time complexity of this particular implementation is O(n) in worst case and this is mostly because the algorithm needs to visit each and every node in the BST while processing. All of the other operations in the algorithm are constant operations and because the loop goes until there is no unvisited element, the time complexity becomes directly proportional with the number of elements in the BST and therefore n elements yield a time complexity of O(n). Apparently it seems that the algorithm cannot be implemented to be asymtotically faster because due to the BST structure every item needs to be visited once. It must also be noted that the best and average case are also same, O(n) because of this reason to visit all elements in such a structure.

- In **span,** in order to determine the number of nodes having items within the range, a recursive approach is taken so that the algorithm recursively checks the current nodes. If the current node turns out to have an item within the range (a <= treePtr->item && b >= treePtr->item), the count is increased and span is called recursively for both children. In the second case, where the current node item is not within the range, we need to continue checking but in order to have more efficiency we need to provide such a solution that we do not need to visit each and every item. In this respect the lower bound and the upper bound is checked (else if (a > treePtr->item) and else if (b < treePtr->item)) and accordingly recursive traversdel is done. If the current item is smaller than the lower bound we do not need to check even smaller nodes but instead we need to branch into nodes with bigger elements and this is done by calling the recursive function for right child (span(treePtr->rightChildPtr, a, b)), node which has certainly larger item than current item and checking in this way. Similarly, when the current item is larger than the upper bound we do not need to check even larger items and therefore a recursive call is done for left subtree (span(treePtr->leftChildPtr, a, b)), since it is guaranteee that in the BST left children are always smaller than root. As the algorithm uses such a way by not needing to check every item, the time complexity is not O(n) but instead it turns out to be O(h+n) which suggests that the time complexity depends on the height of BST and the total node number n such that a <= n <= b. This means that having a more balanced tree will be much efficient although it does not change the overall time complexity. Therefore this implementation seems to give the most efficient approach possible.

- In **mirror,** the main goal is to symmetrically exchange the left and right pointers so that for each node items in the left subtree will be bigger and the items in the right subtree will be smaller. The algorithm proceeds until we arrive to a empty node and during this check it recursively calls the mirror function for both the left and right subtree (mirror(treePtr->leftChildPtr); mirror(treePtr->rightChildPtr);). When the recursion arrives to the base case, empty node (!treePtr), it begins to swap the current left and right node pointers where eventually the tree becomes symmetrical in accordance to y axis. As it comes to the time complexity, it turns out to be similar to any traversel due to the fact that every node is checked and swapped by its symmetrical counterpart thus it yields O(n). All of the operations are constant operations other than the loop which runs until all nodes are visited and because of that the number of operations are directly proportional with the number of nodes in a given BST. In this respect, an asymtotically faster apporach would not be less than O(n) as every item must be visited.