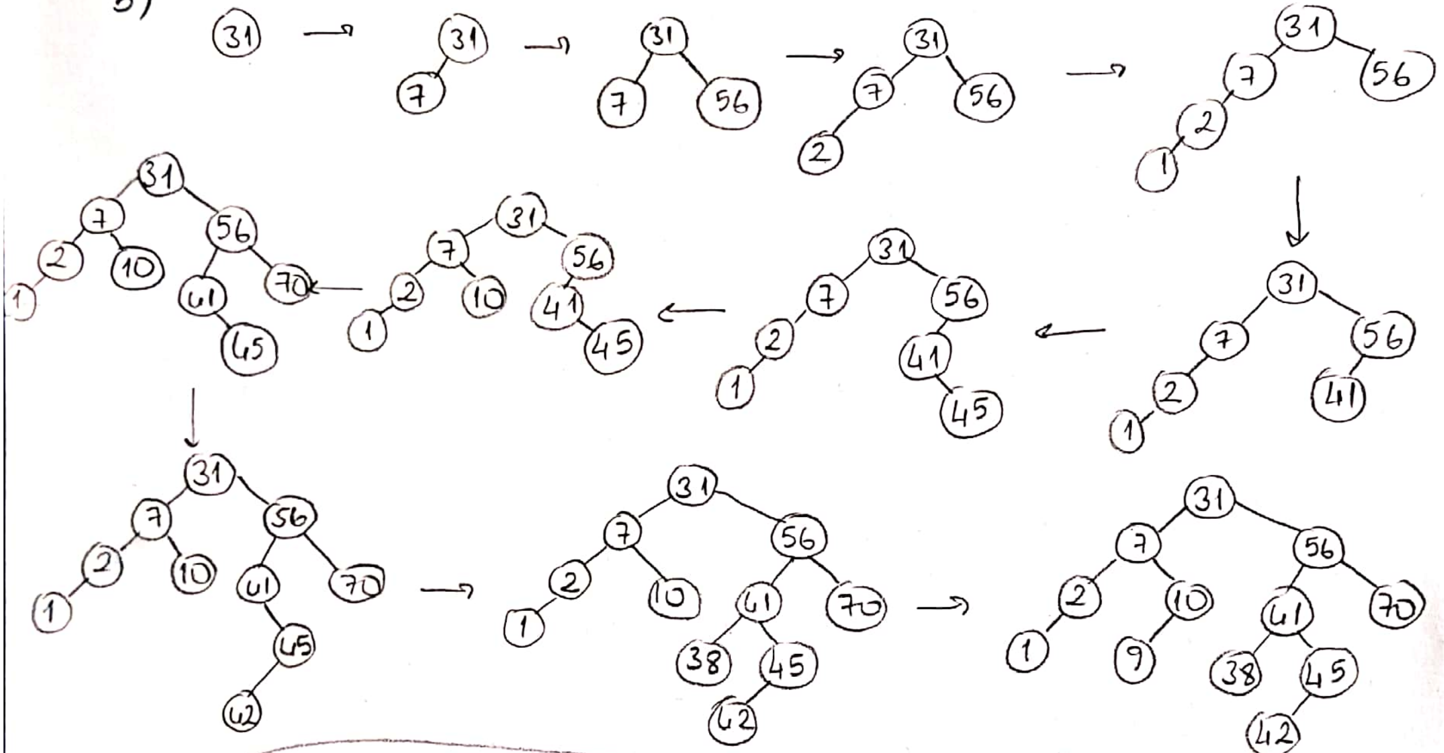
Question 1:

a) Prefix (preorder traversal): / * A + B C D

Infix (inorder "): (A * (B + C)) / D

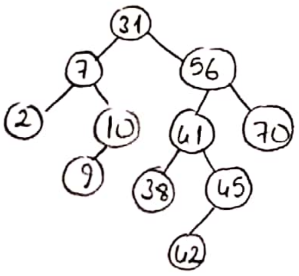
Postfix (postorder "): A B C + * D /

b)

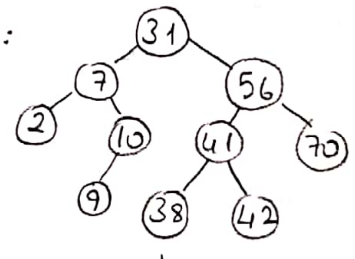


Insertions are completed.

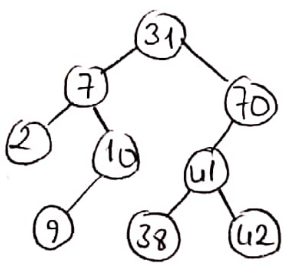
Delete 1:



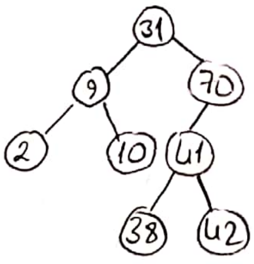
Delete 45:



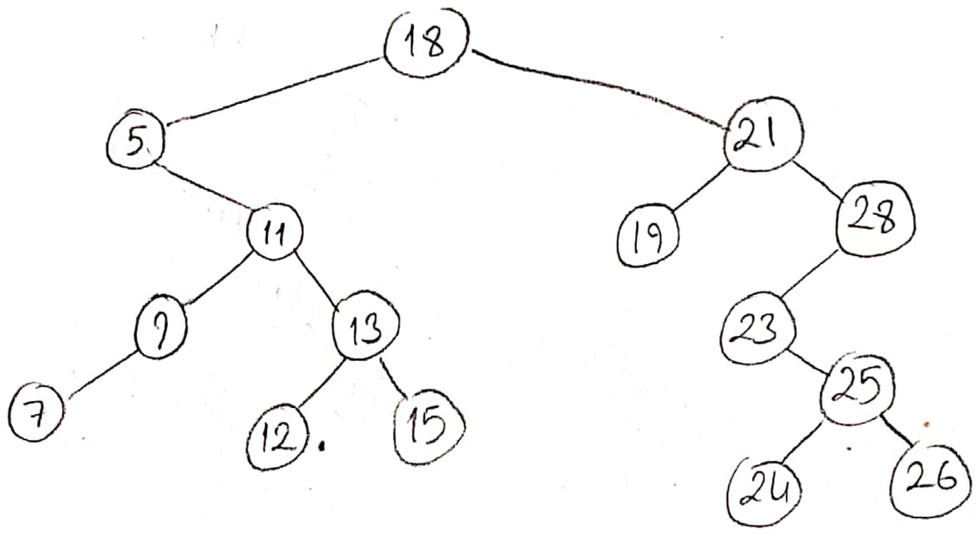
Delete 56:



Delete 7



c)



Postorder :

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

Ayda Yurtoğlu
21903153
CS 202, Section 1

Name: Ayda Yurtoğlu

Student ID: 21903153

Section No: 1

Question 3:

LevelorderTraverse:

I have created two additional functions, void levelorder(BinaryNode *&treePtr) and void levelorder2(BinaryNode *&treePtr, int level). In the original function, levelorder(root) is called, which calls the levelorder2 function for all its levels. It returns if the root is NULL. Levelorder2 is a recursive function and it calls itself until the level input is 1, meaning that, for a node on level a, levelorder2 will go down the tree until it comes to this level. If level input is greater than 1, the function first calls levelorder2 for left subtree and then calls right subtree, which makes the function print the level from left to right.

Complexity:

The worst case is when each node only has one child, so the tree has maximum height. In this case, if the tree has n nodes, then its height (level number) is also n. The for loop in levelorder will run for O(n) time, for each of the levels. The recursive function levelorder2, will call itself until n is equal to 1, which has O(n) complexity per level. So, by multiplying these complexities, we can say that the time complexity is O(n^2) for the worst-case scenario. The time required for the algorithm to finish will be smaller if the tree is a full or a complete tree, for the same number of nodes.

A better algorithm can be implemented, by using additional classes such as queue and using it iteratively. By enqueueing the children of all nodes (starting with the root) and dequeuing them after this operation, we can obtain a faster algorithm. A loop will operate until it reaches the last node of the tree and in each loop it will enqueue the node's children and dequeue the node itself. Since the loop will operate for all nodes, the time complexity will be O(n), which is better than O(n^2).

Span:

I have created an additional function, void span2(const int a, const int b, Binary Node *&treePtr, int& count), which uses recursion. It returns if the given tree is NULL. If the item of a node is in [a,b], it increases the count and calls itself for both left and right subtrees. If the item of a note is smaller than a, it only calls the right subtree and doesn't call left subtree as it contains items smaller than the item of the node. Since we know that the node's item is smaller than a, we don't have to scan its left subtree. Same goes for the node that has a greater item than b. The function doesn't scan its right subtree as it only contains items that are already greater than b. It’s only called for the left subtree.

Complexity:

The function is recursive and for each node that is not NULL, it calls itself. Since it doesn’t contain any loops and contains only operations with O(1) time complexity, the overall time complexity will be O(n) for a tree with n nodes.

The worst-case scenario would be when the items of the nodes are always between [a,b], so the function has to search for all nodes. I don’t think that there is a faster solution, because for the worst-case scenario the function is called only once for each node and we have to visit each of the nodes at least once for this algorithm. The best-case scenario would be if none of the nodes belong to [a,b], so the algorithm doesn’t look at most of the nodes.

Mirror:

I have again created an additional function void mirror2(BinaryNode *&treePtr), which recursively mirrors the tree. The base case is when the node is NULL, so it only functions when node is not NULL. Otherwise, it calls itself for its children, left and right subtrees. Then, to swap the left and right nodes, it creates a temporary node, which is equal to left subtree. It then equalizes the left subtree to the right and right subtree to the temporary node, so that the nodes are swapped.

Complexity:

Since there are no for loops, for each call we will have O(1) complexity. If our tree has n nodes, then we have O(1) + O(1) + … + O(1) for n times, which makes the time complexity n*O(1), that is O(n).

I think a faster implementation is not possible and worst-case scenario doesn’t exist, because no matter the height of a tree, the function has to go through all nodes. So, the existence of the right or left children doesn’t matter and the function only depends on the number of nodes.