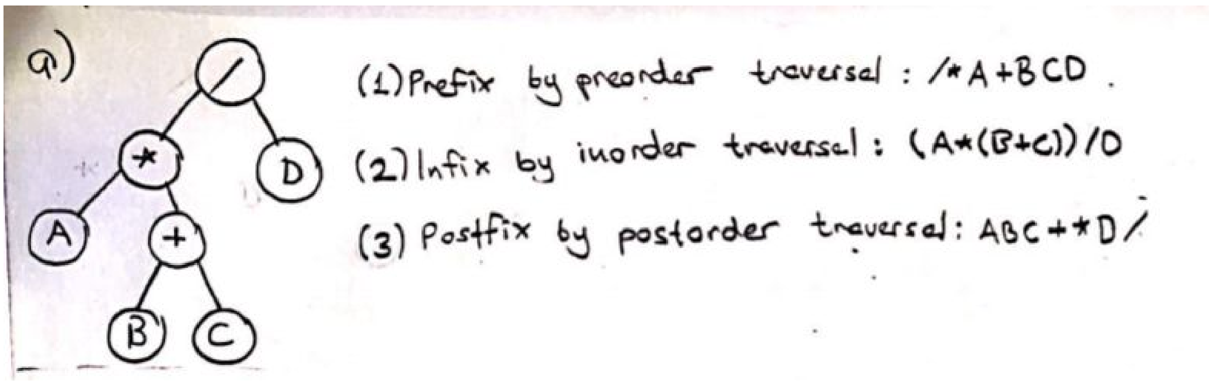
Berk Saltuk Yılmaz - 21903419 - Sec03

Homework 2: Binary Search Trees

Question-1

(a):



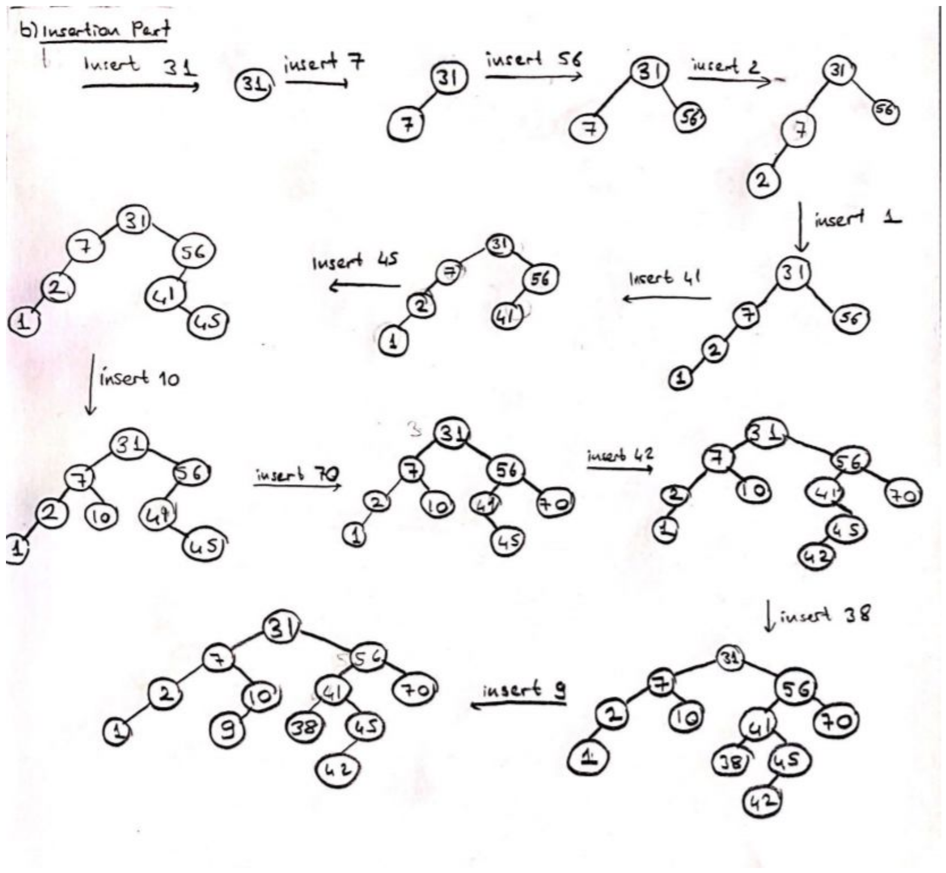
(1) Prefix by preorder traversal : /*A+BCD.

(2) Infix by inorder traversal : (A*(B+C))/D

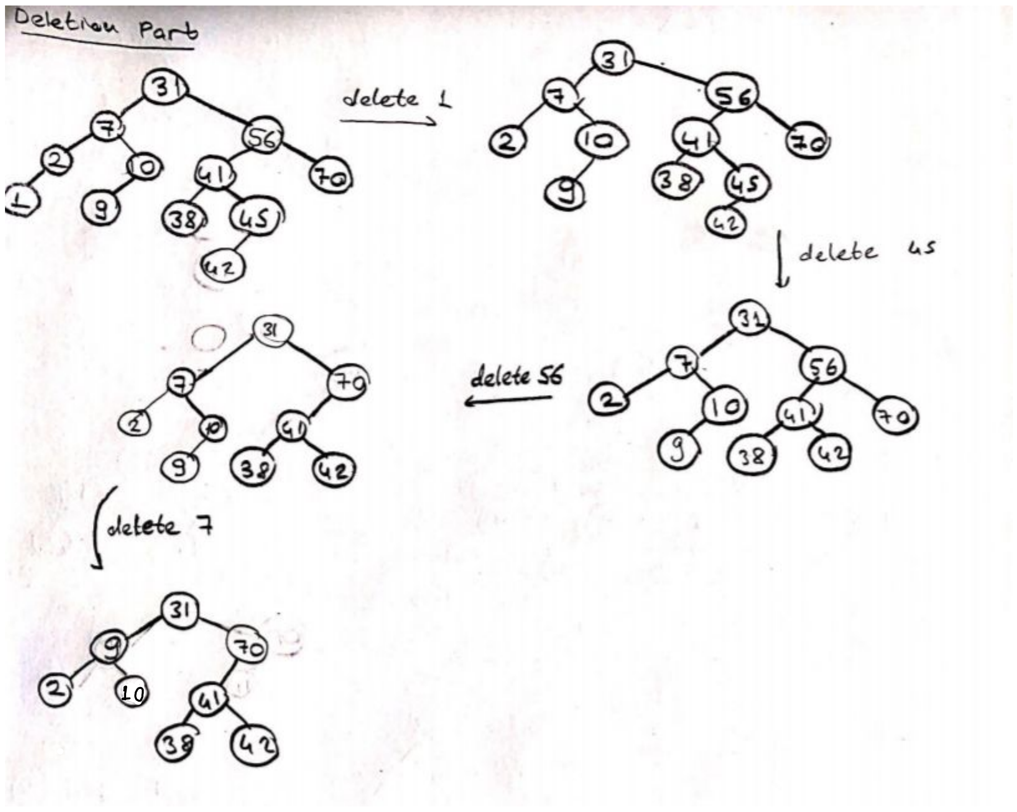
(3) Postfix by postorder traversal: ABC+*D/

(b):

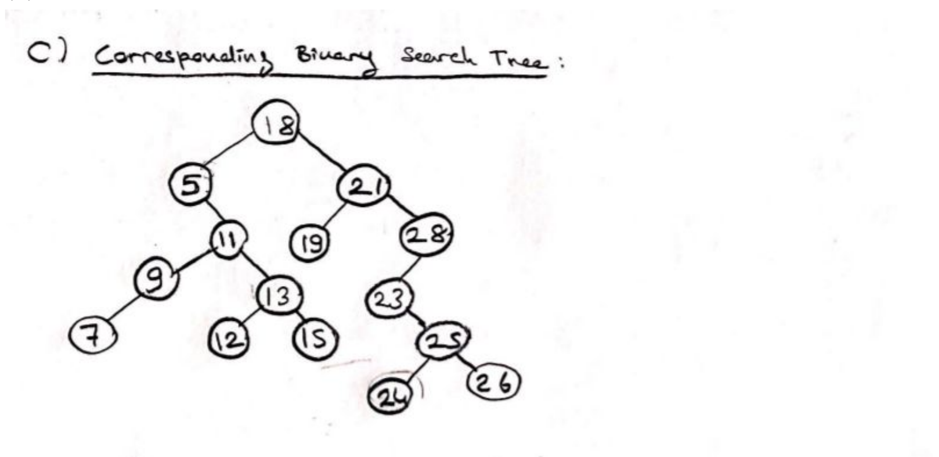
Insertion Part:



Deletion Part:



(c):



Postorder Traversal:

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

Question-3

Analyzing levelorderTraverse Function:

This function initially gets the tree’s height by calling the getHeight() function to find the number of levels of the tree. With a for loop, this function calls another method call levelorderTraverse2 with the root node and the i variable (which prints the items of the current level, the level is indicated with loop variable i) for “height” times. levelorderTraverse2 is a recursive method with the base case checks for the current node to be empty or not. When the base case is not reached, this method calls itself for left and right children until the current level (i) is equal to one. When the current level is equal to one, the function prints the item in the current node. For example, let’s say level order traverse is called as levelorderTraverse(root, i) (i = 3), function calls itself until it prints all the nodes at level 3 (two times in this case).

For each call of levelorderTraverse2, each node is visited once in any case, but only the items in the current level will be printed. Thus, it can be said that the time complexity is proportional to the number of nodes, namely, O(n). Moreover, this function will be called by levelorderTraverse for the number of its height. Such that for loop will be executed n times. As a result, it can be said that time complexity in all cases is $O(n^2)$.

Also, this function could have been implemented by using a queue;
Enqueue the root to the queue, use a while to check whether the queue empty, print the queue’s front, dequeue one element from the queue, enqueue the children and repeat!

Analyzing span Function:

The span function calls another function named span2, which takes two integers defining the span, plus the root node of the BST (In general form, it looks at the current node). The span2 function checks whether the current node is empty or not as the base case of recursion. If not, it checks if the current node's data is in the given span or not. If the data is in the given span, it increases the number of items within this span and calls itself for left and right children. If the data is not in the given span, it checks if the data is greater than the span's higher. If it is, the function is called for only the left sub-tree (since all the elements of the right sub-tree will also be greater than the higher bound of the span). If it is not, the function is only called for the right sub-tree (since the current node's data is not within the span and its left sub-tree also will not be in the span).

If we consider the time complexity, we will check the item in the current node each time, and we will traverse the top to the bottom as the number of height at max. Since if the given span is not satisfied, we will only consider half of the preceding sub-tree. So if the height of the tree is a, and the number of nodes in the given span is b, Big-O time complexity will be O(a + b) in the worst case.

This implementation is the neatest and most efficient way to write this, I believe. Maybe by visiting each node, even the ones we know are not in the span, an inefficient version might be coded.

Analyzing mirror Function:

The mirror function calls another function named mirror2, which takes the root node of the BST (In general, it operates on the current node). This function checks if the current node is empty as the base case. If the node is not empty, it calls itself, first for the left sub-tree, secondly for the right sub-tree. Then from bottom to top, it swaps left and right children as pairs. Finally, the whole tree becomes mirrored. If the method is called again, we obtain the initial tree. It can be seen that the tree is mirrored from the inorder traverse method.

This function works in the same logic as in/pre/postorder traversal functions. Two function calls will happen recursively at each time, and it will be equivalent to total n visits if we say there are n nodes in the tree. For each call, we can say

$$T(n) = T(0) + T(n-1) + 1$$

$$\vdots$$

$$T(n) = T(0)n + n$$

Via back substitution, the Big-O time complexity can be found as O(n).

This function could be implemented using a queue, too. First enqueuing the root and then enqueuing every pair and dequeuing nodes in reverse order, while the queue is not empty.