CS 202 - Fundamental Structures of Computer Science 2

Homework 2 – Binary Search Trees

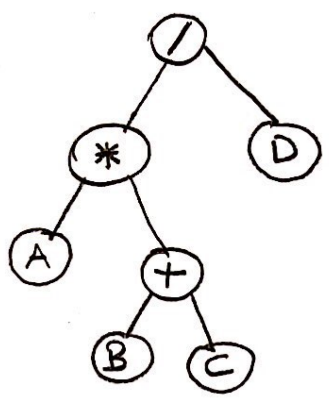
Student Name: Esad İsmail Tök

Student ID: 21801679

Section: 2

Question 1

Part a)



Prefix: $\boxed{/ * A + B C D}$

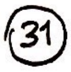
Infix: $\boxed{A * B + C / D}$

Postfix: $\boxed{A B C + * D /}$

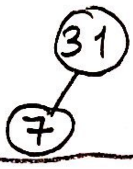
Question 1

Part b)

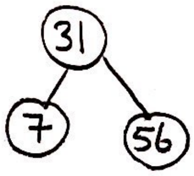
Insert 31 :



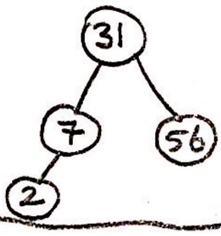
Insert 7 :



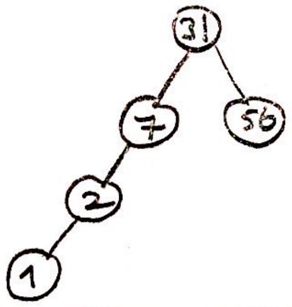
Insert 56:



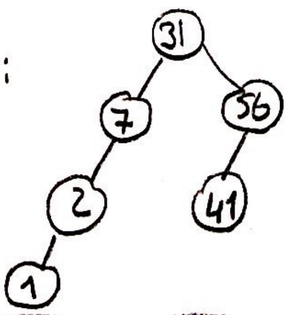
Insert 2:



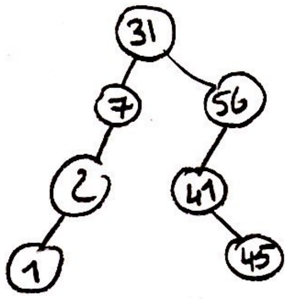
Insert 1:



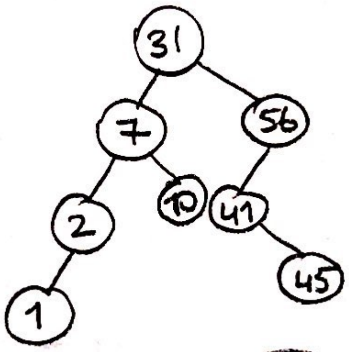
Insert 41:



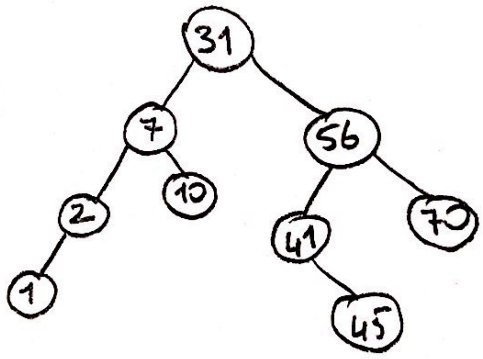
Insert 45:



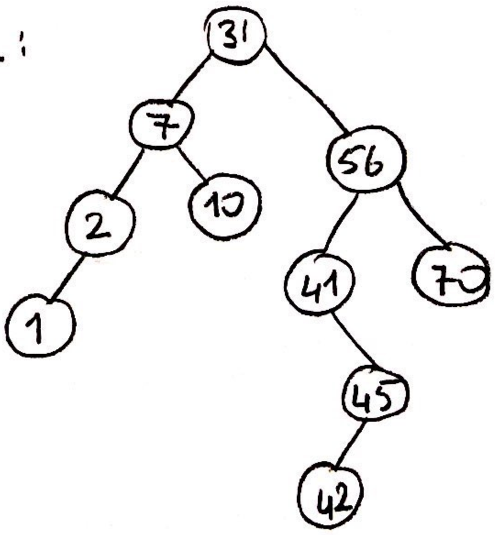
Insert 10:



Insert 70:



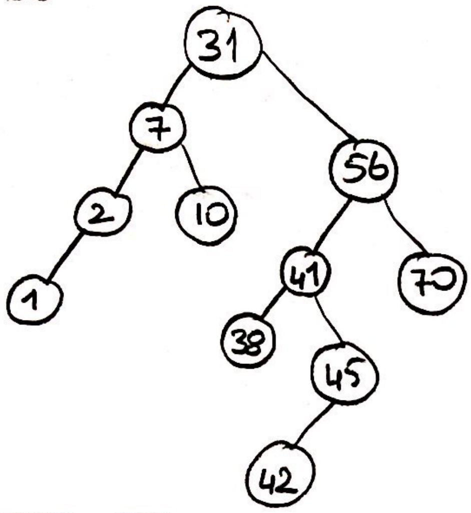
Insert 42:



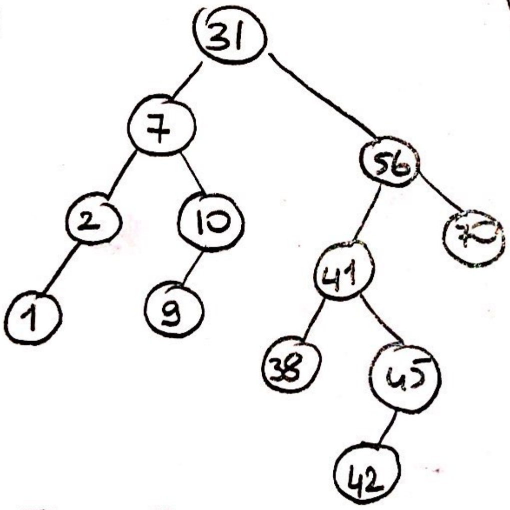
Question 1

Part b continued)

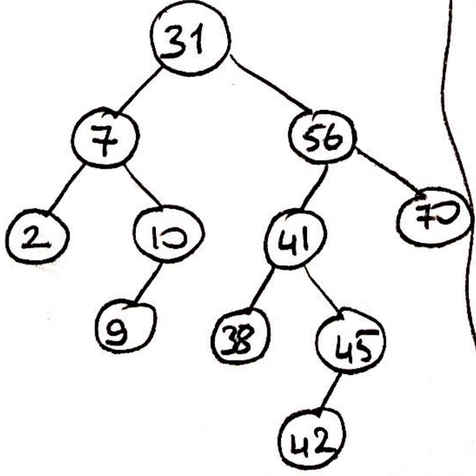
Insert 38:



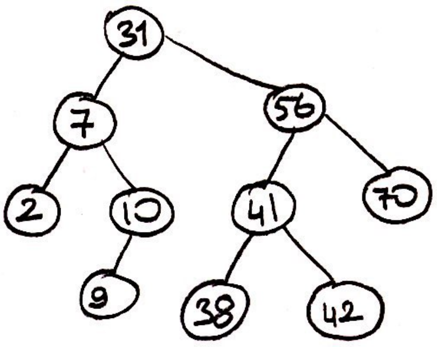
Insert 9



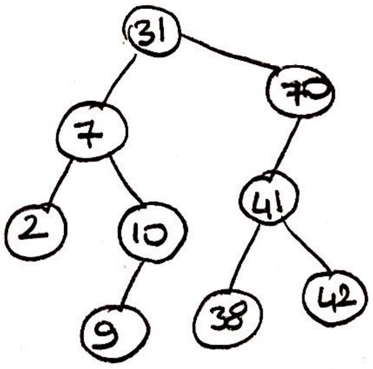
Delete 1:



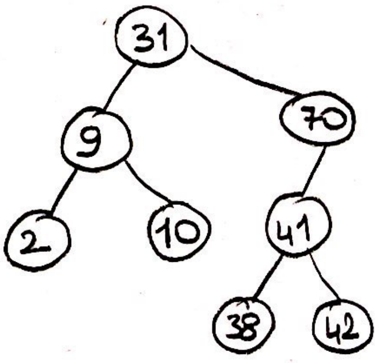
Delete 45:



Delete 56:



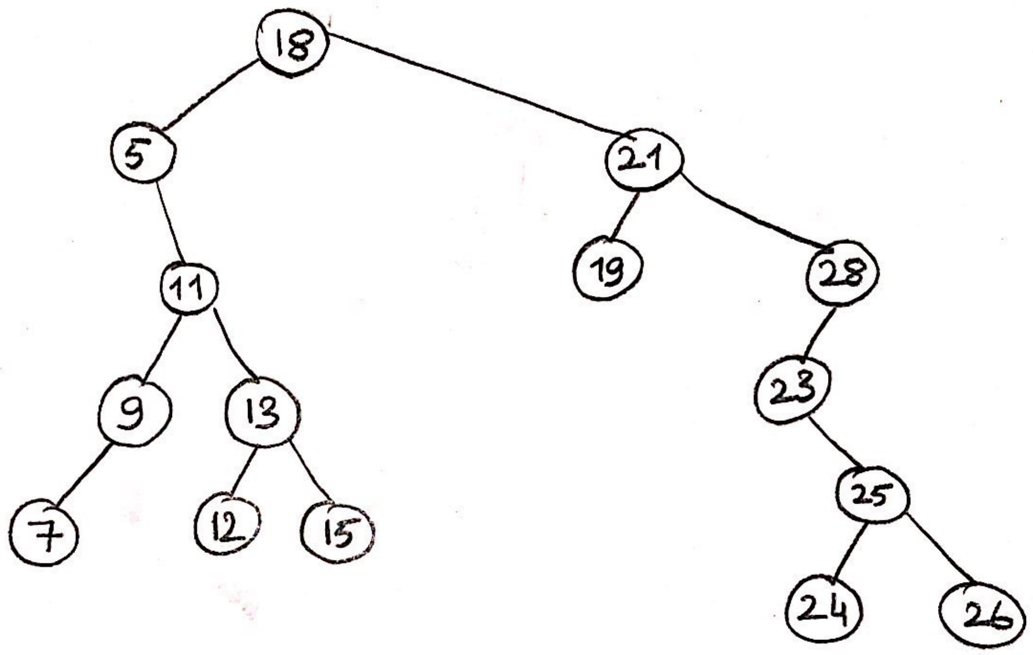
Delete 7:



Question 1

Part c)

Corresponding Binary Search Tree :



Postorder Traversal :

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

Question 3

Level Order Traverse Function:

In order to implement the levelorderTraverse() function, a Queue is used as a helper data structure and the ADT Queue is implemented in the code.

It is an iterative function. If the root is null, then function returns immediately because there is no node to traverse. If there is at least one item in the tree, an empty queue, containing BinaryNode pointers, is instantiated and the root of the tree is enqueued initially. The function will traverse the tree as long as the queue is not empty. In each iteration, the front item of the queue is retrieved and printed, then the queue is dequeued. So that a node is printed and removed from the queue. After the print process, the left and right child pointers of the current node is enqueued respectively, if they are not null. By this implementation we preserve the level order of the tree and print each item only when it is their turn to print. The function returns when the queue become empty.

If the tree has “n” items then since it is a traversal, the function will traverse all the nodes in the tree. Each print, enqueue, and dequeue operation has O(1) time complexity and there are total of “n” nodes to traverse, therefore the worst-case time complexity of the level order traverse is O(n).

Since this is a traversal method and must traverse all the nodes of the tree, O(n) is the best time complexity to achieve. Therefore this method cannot be implemented asymptotically faster.

Span Function:

In order to implement the span(const int a, const int b) function, a recursive helper function spanHelper(BinaryNode* treePtr, const int a, const int b) is used.

The root of the tree is passed to the helper function together with lower and upper bounds. The function has 3 base cases: If the lower bound is greater than the upper bound, there is an input fault and -1 is returned to indicate this exception. If the passed pointer is null, then it is not counted and 0 is returned. If the upper and lower bounds are equal and if the item equal to those values is found, then the function directly returns 1 since the tree cannot include the same value more than one times. As the recursive conditions: If the checked node is within the range, then function adds 1 and checks the left and right subtree recursively to add them into the return value as well. If the checked node item is below the range, then the right subtree of this node is passed discarding the left subtree. If the checked node item is above the range, then the left subtree of this node is passed discarding the right subtree. Therefore the total count of the nodes within the range is found.

In the worst case in which all the items in the tree are within the range of the inputs, the algorithm would traverse all the nodes of the tree in order to be able to count all of them just like a traversal algorithm. However considering the average, the algorithm does not need to visit all the nodes in the tree and if we consider “h” as the height of the tree and “k” as the total number of node items within the range in the tree, the time complexity of the algorithm becomes O(h + k).

It is an efficient algorithm to implement this function since the algorithm does not require to traverse all the nodes in the tree. The algorithm is fast enough and there is no asymptotically faster way of implementing this function as far as I can think of.

Question 3

Mirror Function:

In order to implement the mirror() function, a recursive helper function mirrorHelper(BinaryNode* treePtr) is used.

The root of the tree is passed to the helper function. If the passed pointer is null, then the function returns as the base case. As long as the pointer is not null, the algorithm recursively calls itself with passing the left and right child pointers respectively. When the two recursive calls in the function returns, the left and right child pointers are exchanged using a temporary Binary Node pointer. The algorithm keeps exchanging the left and right children of each node and at last the root's children are exchanged and the function returns. Thus an opposite binary search tree is obtained in which all items in the left subtree of a node are greater than that node; and all items in the right subtree of a node are smaller than that node. Moreover the inorder traverse now prints the items in the tree in descending order, preorder traverse of the mirrored tree is the reverse of the posterder traverse of the original tree, and the postorder traverse of the mirrored tree is the reverse of the preorder traverse of the original tree.

The algorithm is considerably similar to a postorder tree traversal because the function recursively calls itself with the left and right child pointers of the node and at last visits the node itself. The operations done in each recursion has O(1) time complexity since the basic assignments takes 1 unit of time; and since the algorithm has to traverse all the nodes in the tree, the time complexity of the algorithm is O(n), in which "n" is the item size of the tree.

Since the algorithm must traverse all the nodes of the tree in order to exchange the children O(n) time complexity is fair for this algorithm and this function could not be implemented asymptotically faster as far as I can think of.