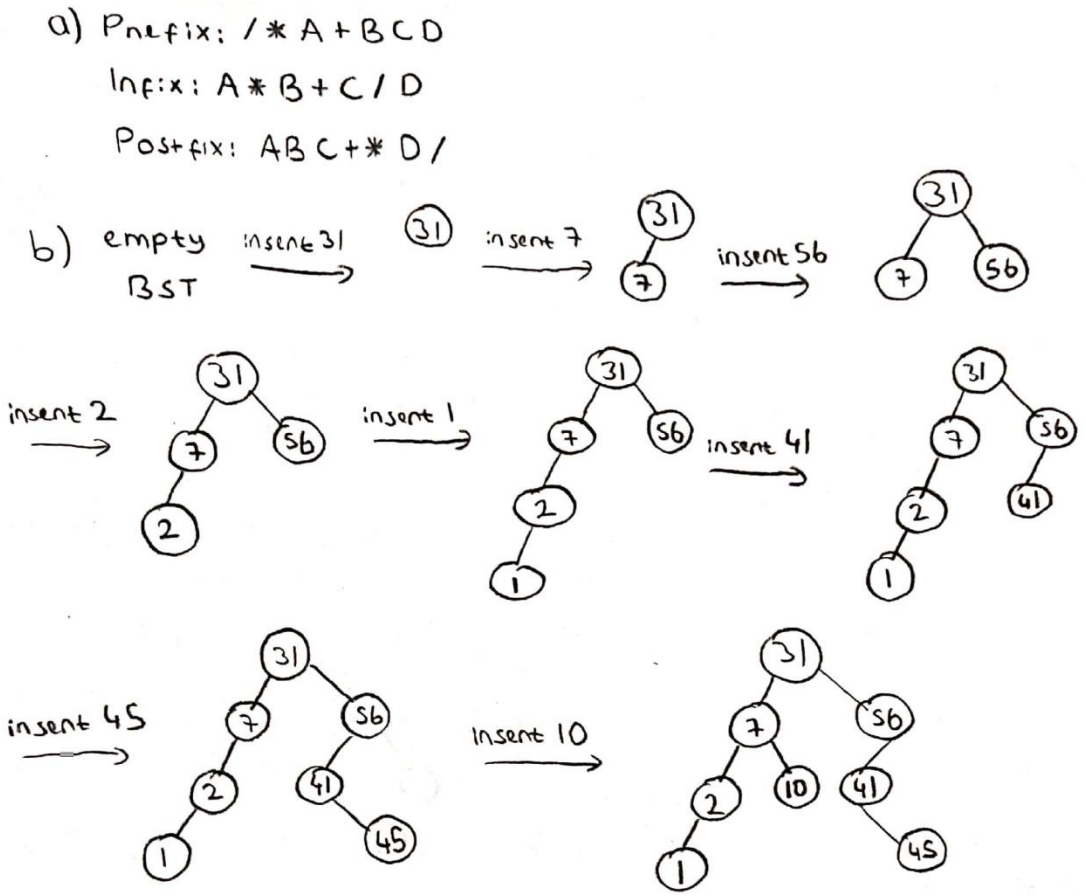
Cemhan Kaan Özaltan

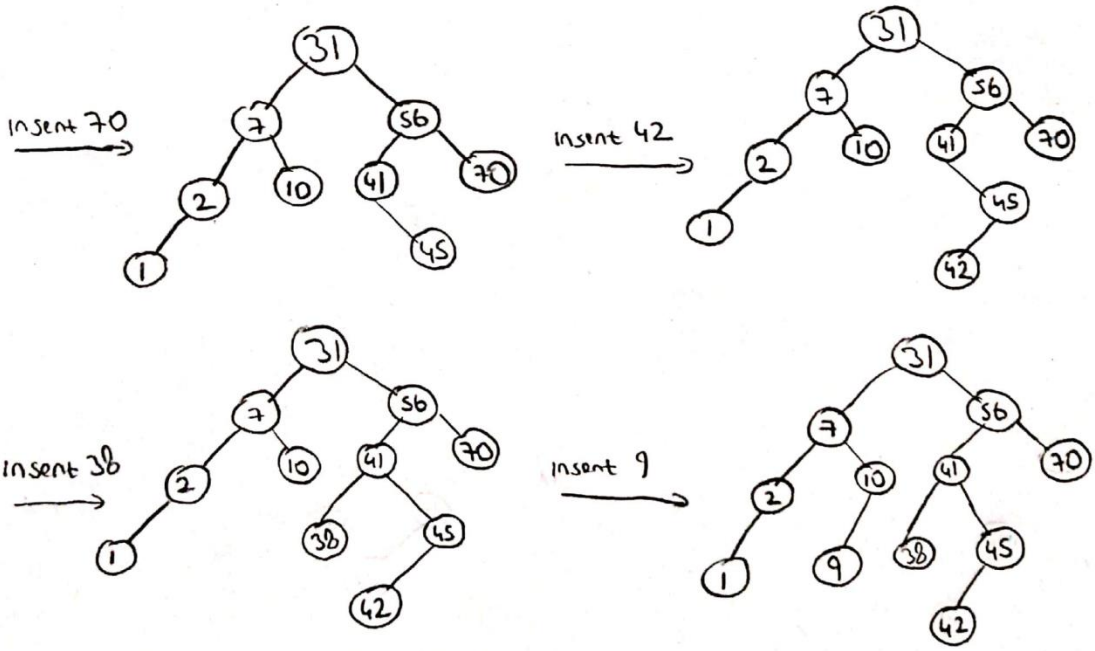
21902695

Section 3

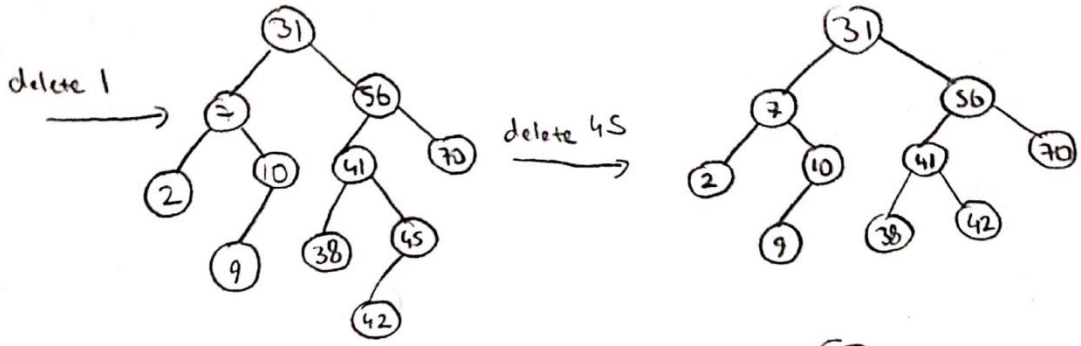
Homework 2 Report

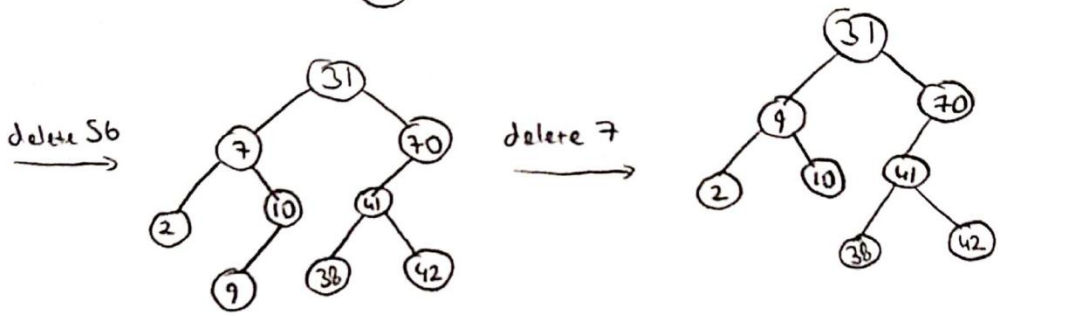
Question 1



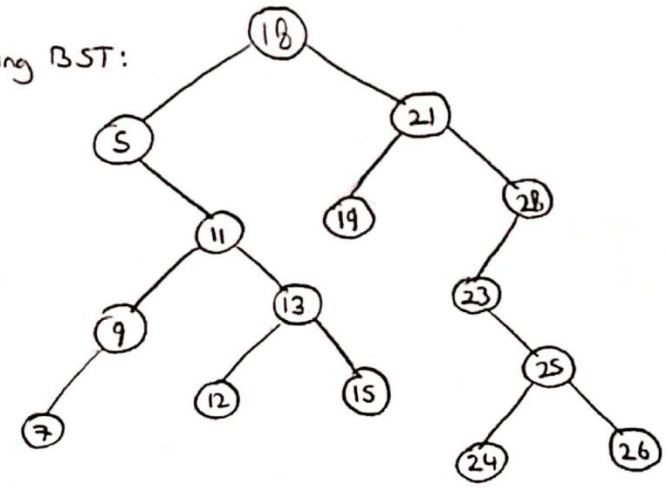

insert 70
31
7
56
2
10
41
70
1
45
insert 42
31
7
56
2
10
41
70
1
45
42

insert 38
31
7
56
2
10
41
70
1
38
45
42
insert 9
31
7
56
2
10
41
70
1
9
38
45
42


delete 1
31
2
56
2
10
41
70
9
38
45
42
delete 45
31
7
56
2
10
41
70
9
38
42


delete 56
31
7
70
2
10
41
9
38
42
delete 7
31
9
70
2
10
41
38
42

c) Corresponding BST:



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

Question 3

The levelOrderTraverse function is implemented with two helper functions. Initially, levelOrderTraverse calls levelTraverse with root parameter. Later on, levelTraverse calculates the height of the BST and calls levelDisplay for each level with treePtr and level parameters. Later, if treePtr is not NULL and the base case is not reached, levelDisplay makes recursive calls for the left and right subtrees with decremented level parameters until level reaches 1, at which it prints the item of treePtr. Decrementing level results with different level values and different amounts of recursive calls, therefore allowing the algorithm to reach the required level. The worst case is with a BST with children all at the same side. In this case, the height is equal to node count so levelTraverse is O(n). Similarly, levelDisplay goes from 1 to n operations, making it O(n). Therefore, the worst case time complexity is $O(n^2)$. It is possible to make this algorithm asymptotically faster by using additional data structures such as queues since BST does not allow direct access which increases the time complexity.

The span function is implemented with a helper function. First, span initializes a result variable to 0, which will be used tor return the result with pass by reference. Then, it calls spanTree with root, a, b and result parameters. If treePtr is not NULL and the base case is not reached, spanTree increments result if the item of treePtr is in the span. Then, if treePtr's item is greater than or equal to a, a recursive call for the left subtree is made. Similarly, if treePtr's item is less than or equal to b, a recursive call for the right subtree is made. This is done in order to prevent traversing unnecessary parts. The worst case is when the span includes every item in the BST. Since every node is visited in this case, the time complexity is O(n). Because

visiting the required nodes is necessary in every case in order to count them, this algorithm cannot be made asymptotically faster.

The mirror function is implemented with a helper function. First, mirror calls mirrorTree with root parameter. Later, if treePtr is not NULL and the base case is not reached, mirrorTree swaps leftChildPtr with rightChildPtr and makes recursive calls for the left and right subtrees. This operation ultimately mirrors the whole BST by swapping the left and right child pointers of each node. Since each node is visited in every case and each swap has a constant number of operations in it, the worst case time complexity is O(n). As visiting every node is necessary, this algorithm cannot be made asymptotically faster.