Title: Binary Search Trees
Author: Dilara Mandıracı
ID: 22101643
Section: 1
Homework: 2
Description: This homework aims to inspect the fundamentals of Binary Search Trees, learning insertion and deletion principles, comparing elapsed time of these operations.
Date: 24.03.2023

QUESTION 1

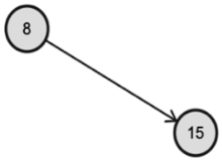
a)

Inserting 8,15,10, 4, 9, 6, 16, 5, 7, 14 to an empty Binary Search Tree:

1

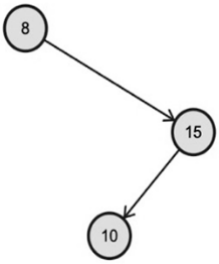
Insert 8 to empty BST

2



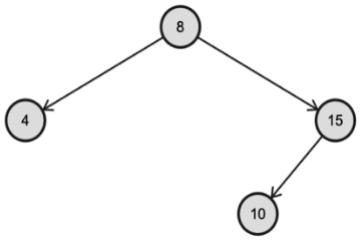
Insert 15 to the right child of 8.
Larger values go right, and smaller values go left to the binary search tree.

3



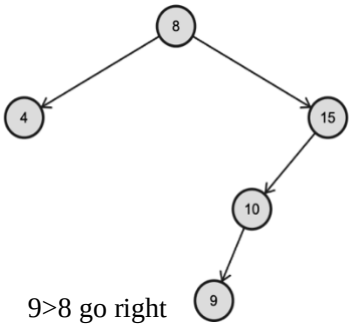
10>8 so go right.
10<15 so go left.
Insert 10.

4



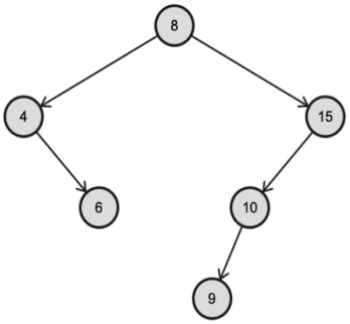
4<8 go left and insert

5



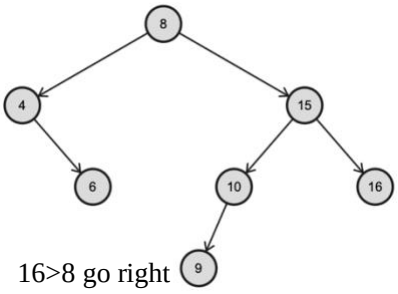
9>8 go right
9<15 go left
9<10 go left and insert

6



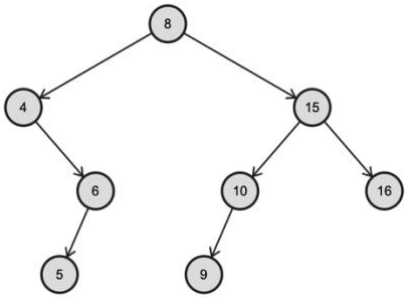
6<8 go left
6>4 go right and insert

7



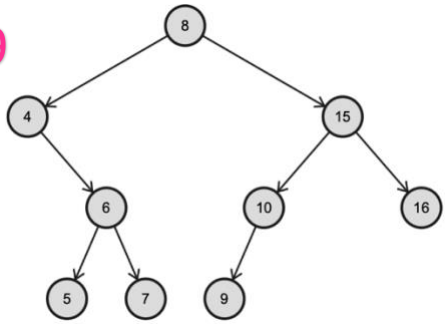
16>8 go right
16>15 go right and insert

8



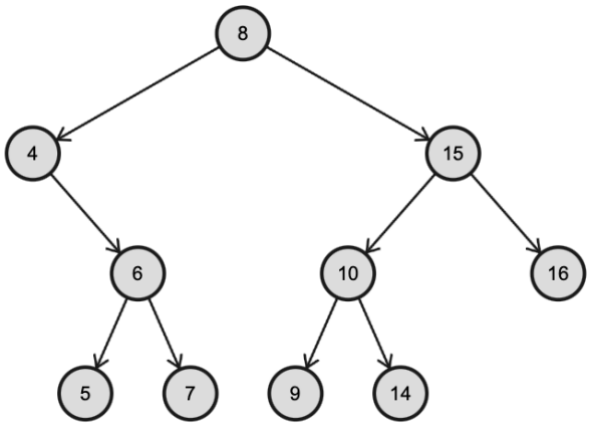
5<8 go left
5>4 go right
6<6 go left and insert

9



7<8 go left
7>4 go right
7>6 go right and insert

10



14>8 go right
14<10 go left
14>10 go right and insert

b)

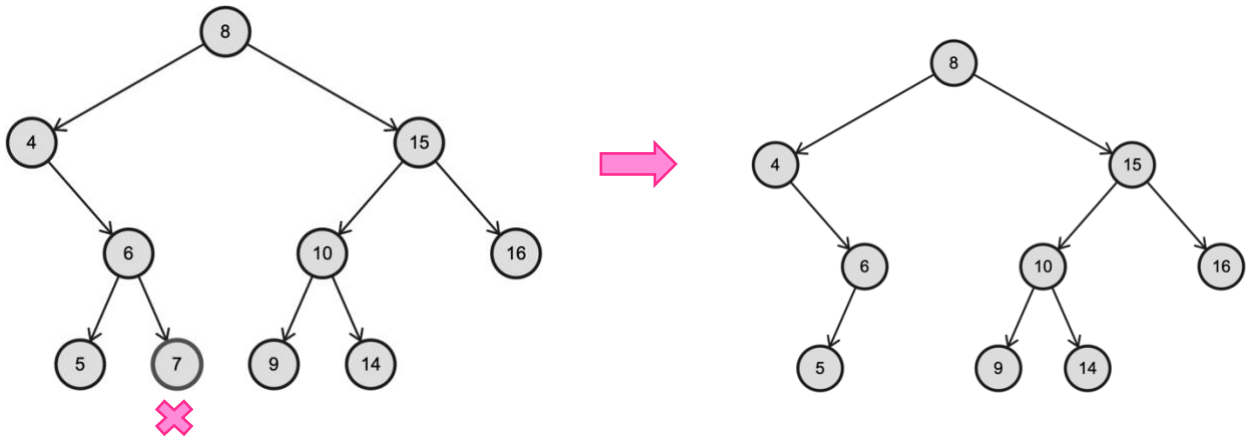
preorder traversal: **8,4,6,5,7,15,10,9,14,16**

inorder traversal: **4,5,6,7,8,9,10,14,15,16**

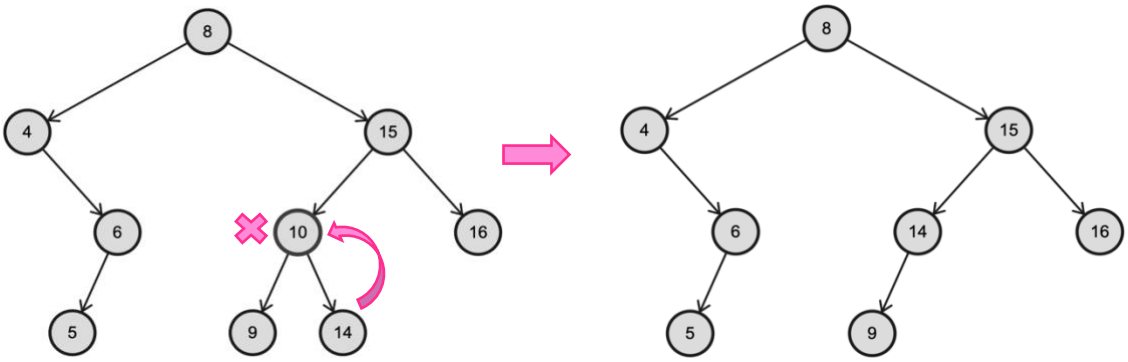
postorder traversal: **5,7,6,4,9,14,10,16,15,8**

c)

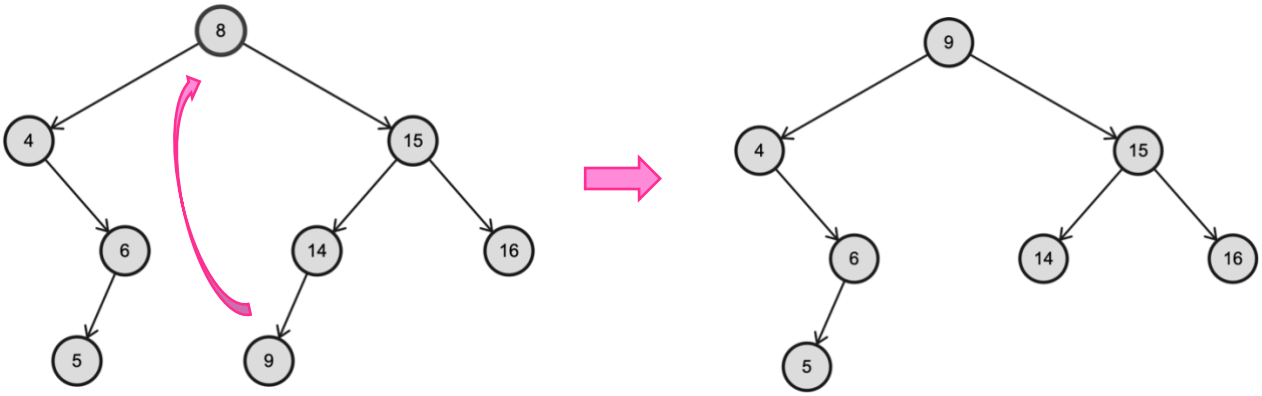
Delete 7: Since 7 is the leaf directly can be removed.



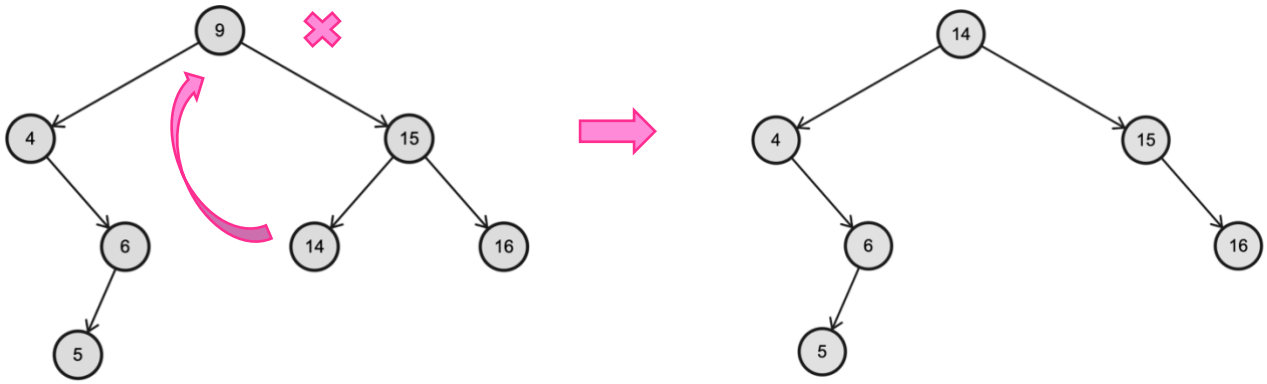
Delete 10: Node 10 has left and right children, when deleting 10, min element of its right sub-tree will be replaced to its place.



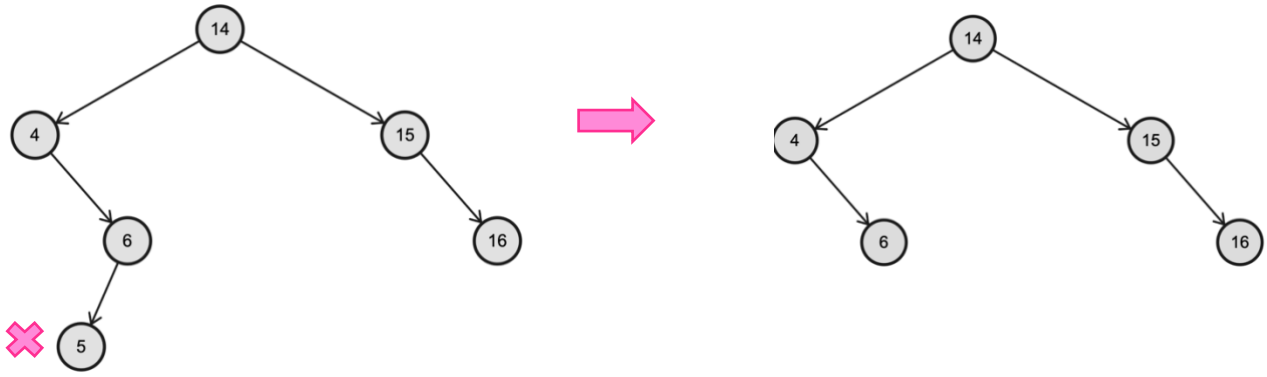
Delete 8: while deleting 8, its position will be filled by the min of its right sub-tree, which is 9



Delete 9: minimum element of a right subtree is 14, so after deleting 9, 14 will be placed its position.



Delete 5: since 5 is a leaf it can be deleted easily without changing any element's position.



d)

Recursive pseudocode implementation for finding the minimum element in a binary search tree:

While constructing binary search tree, smaller elements always put to the left side. So, while searching the minimum element always go left until reaching null pointer. When a node's left child is null it will be the minimum element of the binary search tree.

searchMinVal(node)

If node->left == null //if there is nothing at the left of that node
    return node->data //return its value
else //go left
    return **searchMinVal**(node->left) //call function again for the left node

e)

Maximum and minimum height of a binary search tree that contains n items are:

Maximum height: n

If all elements are inserted in a increasing or decreasing order it will create a maximum height of a binary search tree. N elements will construct n heighted binary tree

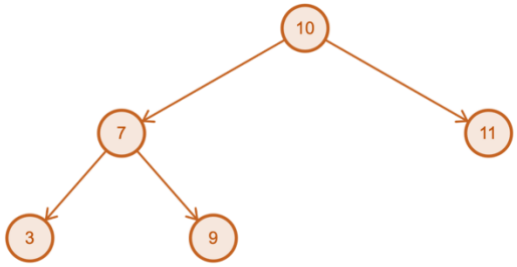
For example: n = 4, height = n => 4

Minimum height: minimum height can be obtained by making insertion in the correct order and trying to complete each row with nodes. Which can be said as ceil($\log_2(n+1)$).
Note: ceil() function rounds the double value to the upper integer. For example, ceil (2.4) = 3

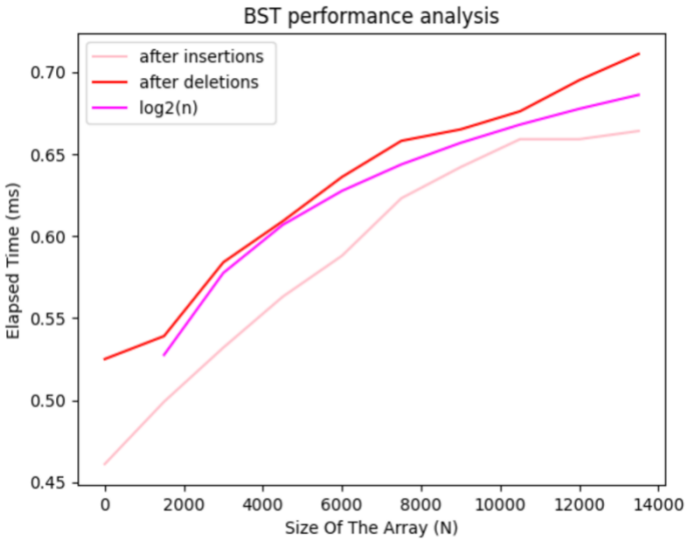
For example: n = 5

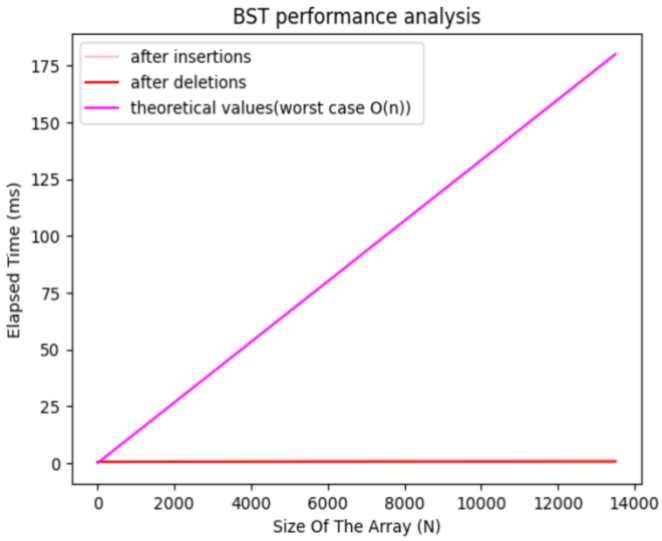
$\text{Log}_2 6 = 2.5$

Ceil(2.5) = 3



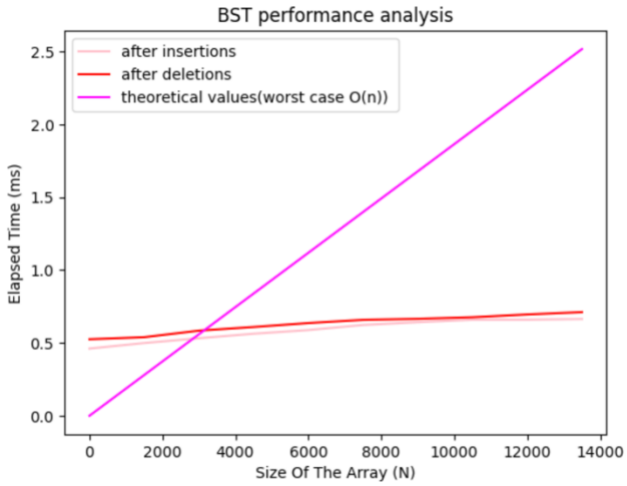
Conclusion





The average case of insertion and deletion to a binary search tree is O(logn). If we think about the insertion first, even if we insert 1500 random numbers for each set, the necessary time to insert the same amount of data is increasing. The time for inserting exactly the same amount of number is increasing because, in every couple of insertions, the height of the binary search tree will increase. When the tree has a higher height, it means the current pointer needs to traverse more to find the correct place to insert a new element. So, it can be concluded that if the height of the tree increases, then insertion time increases for the same amount of insertion. In addition, it behaves in a way that the average case, which is O(logn), because we use the random data set. If the data set were sorted (in any way ascending or descending), the worst case would

occur since it would always insert to the left or always insert to the right. This case would make the height maximum, as discussed in part e. It can be seen from the graph on the rights side, which compares the worst-case behavior and the random case. The worst-case elapsed time is much faster and higher than random results, and it is hard to show them in the same graph. So, I divided them into two graphs. The above right graph shows the comparison between the worst case and the experimental values which is average case. I showed the experimental values in the left graph by comparing their behavior with the log n function (putting some constant value to observe behavior). Below graph is also used to show the growth rate difference between O(N) and O(logN) by putting some constant to theoretical values.



The same discussion is valid for deletion. If the total elements and the height of the binary search tree decrease, the deletion time decreases. Since the height of the tree decreased, it made it easier and faster to reach the node that will be deleted. So, even if we delete the same number of nodes (1500) from a binary search tree when the height is low, deletion will take shorter since it takes shorter to find the correct node. According to need, it also takes shorter to find the minimum element of its child. Deleting the elements in the order of their insertion, is the best-case scenario because in that case always deleting the top element is not required to traverse the tree, and its height is not important. Hereby, the data set inserted is shuffled before the deletion process starts so that this situation doesn't happen. To conclude, if we think about the worst case of deletion, it would be an insertion data in ascending order and a deletion in descending order or vice versa.