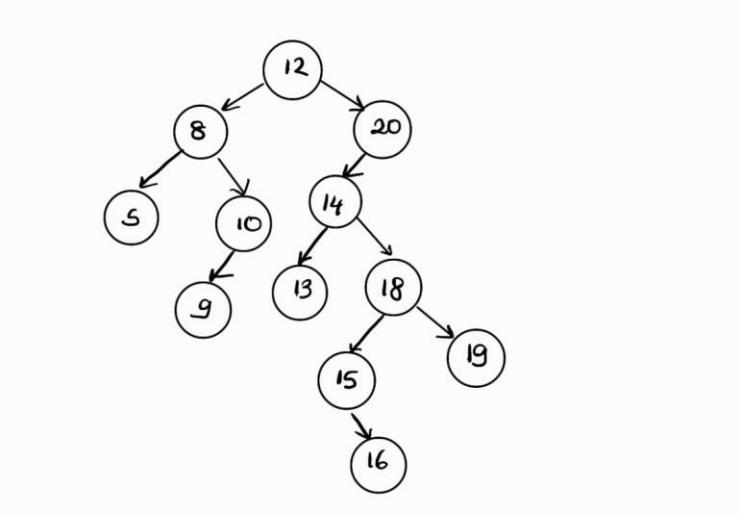
CS202 Homework 1

Hilal Coşgun

22201571

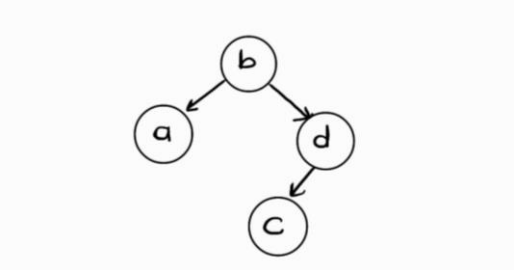
Question 1:

a) In pre-order traversal, recursively, first the root data is printed then the left subtree and then the right subtree are traversed.
In the first sequence we can detect that 12 is the root since the root data is printed first. Then, the elements smaller than 12 should be placed in the left subtree, and the elements greater than 12 should be placed in the right subtree. For each root, we should complete the left subtree before moving to the right subtree. However, in the first sequence, after 14, 15 comes next, which means we start placing elements in the right subtree. But the following element, 13, should have been placed in the left subtree of 14. This is why this sequence cannot be a pre-order traversal.
The second sequence, however, could be the pre-order traversal of the following tree:

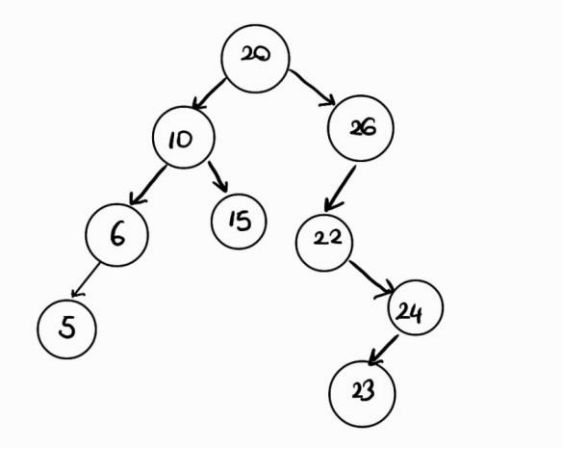


Post-order Traversal: 5 9 10 8 13 16 15 19 18 14 20 12
In-order Traversal: 5 8 9 10 12 13 14 15 16 18 19 20

b) There is only one possible way to arrange the elements.
There must be one number smaller than the root value in the left subtree and two numbers greater than the root value in the right subtree; therefore, b should be the root value. Since a is smaller than b, it is inserted into left subtree and c and d, which are greater than b, into the right subtree. As c is smaller than d, it is placed in the left subtree of d. To conclude, there is only one way to insert a<b<c<d into the given BST.



c) In post-order traversal, recursively, first the left subtree and right subtree are traversed, and then the root data is printed. Since we know that the root of the tree is printed last, the value at the end of the given sequence, which is 20, is the root. It is understood that the values smaller than 20 go to the left subtree, and the others go to the right subtree. The part of the sequence to the left of 23 is smaller than 20, and since 10 is the last element in this part, the root of the left subtree of 20 is 10. 15 is greater than 10, thus it goes to right subtree. The remaining 5 and 6 are smaller than 10, but 6 is printed later, then the root of the left subtree of 10 is 6. Following similar steps, the right subtree can be constructed. The result is:



Pre-order traversal: 20,10,6,5,15,26,**22**,24,23

22 is the seventh key.

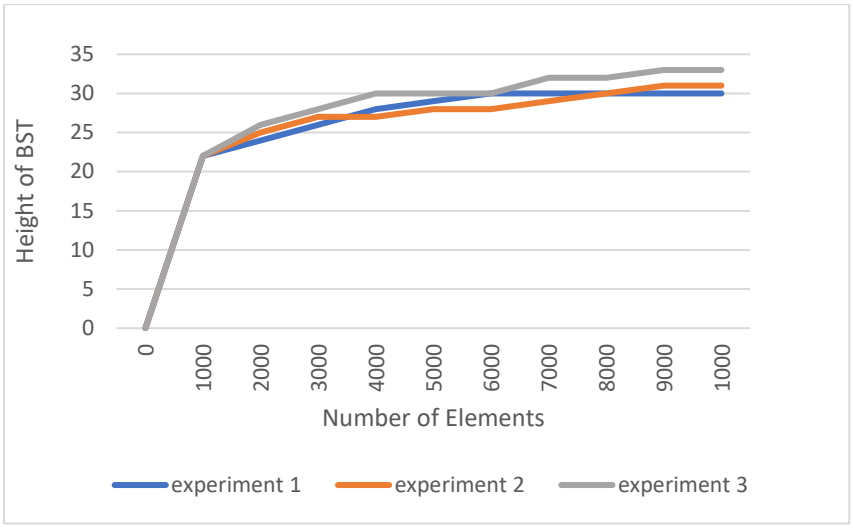
d) The first inserted key is 43, them both 4 and 9 will remain in the left subtree of the tree. For them to be compared, none of the numbers 5, 6, 7, or 8 should be inserted before either 4 or 9. Otherwise, the following scenario may occur: Suppose 5 is inserted before both 4 and 9. When we try to insert 4, it will be compared with 5 and placed in the left subtree. And when we try to insert 9, it will be compared with 5 and placed in the right subtree. In this case, 4 and 9 will not have been compared. To avoid this, at least one of 4 or 9 must be inserted before any of 5, 6, 7, or 8.

$A = \{$ One of $\{4,9\}$ is chosen first from the set $\{4,5,6,7,8,9\}$

$$P(A) = \frac{\binom{2}{1}}{\binom{6}{1}} = \frac{1}{3}$$

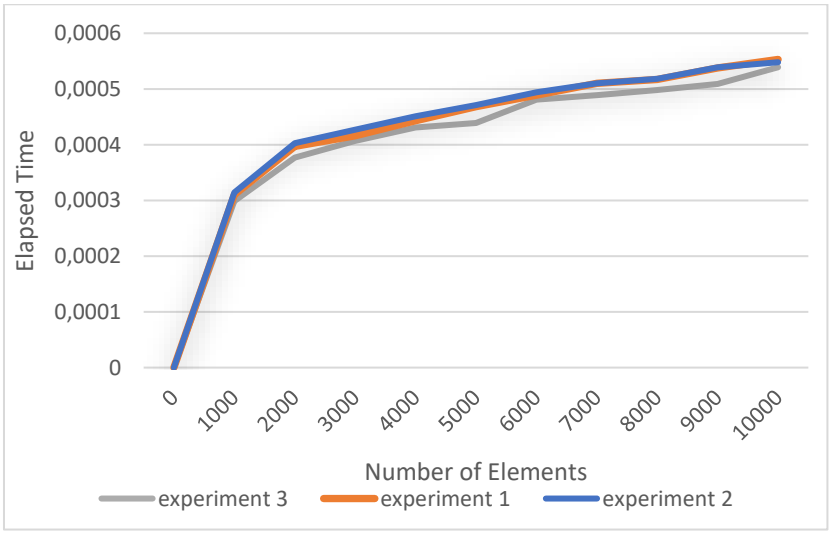
Question 3:

1)



I inserted 10,000 random numbers into a BST in parts of 1,000 and calculated the height after each part. I repeated this process three times and used the data to create the graph above. As can be seen, the resulting function resembles a logarithmic function. This was the expected scenario because of the nature of binary search trees. If we consider a full binary tree, the number of nodes at each level doubles the number at the level above, and the total number of nodes is 2^(height – 1). Therefore, the height would be equal to log2(n) + 1. Since a full binary tree is not achieved in our experiments with random numbers, in other words, some nodes do not have two children, the result is higher but still in a logarithmic shape. Then height of a BST with randomly inserted keys can be considered as an average case, resulting in h = log(n).

2)



I inserted 10,000 random numbers into a BST in parts of 1,000 and measured the elapsed time between each part. I repeated this process three times and used the data to create the graph above. The function in the graph resembles a logarithmic function. The reason is as follows: To find the proper position for insertion, one comparison is needed at each level. Then, the

cost of inserting a node into a BST is proportional to height of the BST. According to first graph, height of the randomly created binary search trees increase in a logaritmic way, then the insertion time complexity also increases in a logaritmic way, meaning O(logn).

If I inserted sorted numbers instead of random ones, BST would resemble a linked list. For instance, if the numbers were sorted in ascending order, in every insertion I would insert to the right subtree and the height would increase by one. As a result, height of tree would be equal to the number of nodes and for each insertion, I would have to traverse all the way to the end of the list. Then the insertion time complexity would be O(n).

3)

Keeping a tree balanced ensures that the tree remains as short as possible, making insertions efficient. To balance the tree, we should ensure that the number of nodes in the left and right subtrees is equal or very close. We should insert the median value of the available data as the root, so that half of the remaining values go to the left subtree and half go to the right subtree. Then, we should insert the median value of the values going to a subtree as the root of that subtree. In this way, the root of the subtrees should always be the median value. However, constantly finding the median in a random list can be costly, so we can sort the keys to be inserted first. Using an efficient method like quicksort for this is even better. After that, a recursive function can be written. The function receives the sorted array and inserts the node at the middle index as root. Then, makes a recursive calls with left half of the array for left subtree and right half of the array for right subtree. This way, by continuously inserting middle elements with a recursive function, we can obtain a balanced tree.

In this approach, we acquire the best case and cost of insertions is almost proportinate to $\log_2 n$. This is better than random insertion because random insertion results in an average case complexity O(logn) since it is unlikely to acquire a balanced tree with random keys. When it comes to sorted insertion, it creates a linked list rather than a BST, which makes it the worst case with O(n) time complexity.