```
/**
* Title: Binary Search Trees
* Author : Mustafa Mert Gülhan
* ID: 22201895
* Section : 1
* Homework : 1
* Description : Answers to Question 1 and 3
*/
```

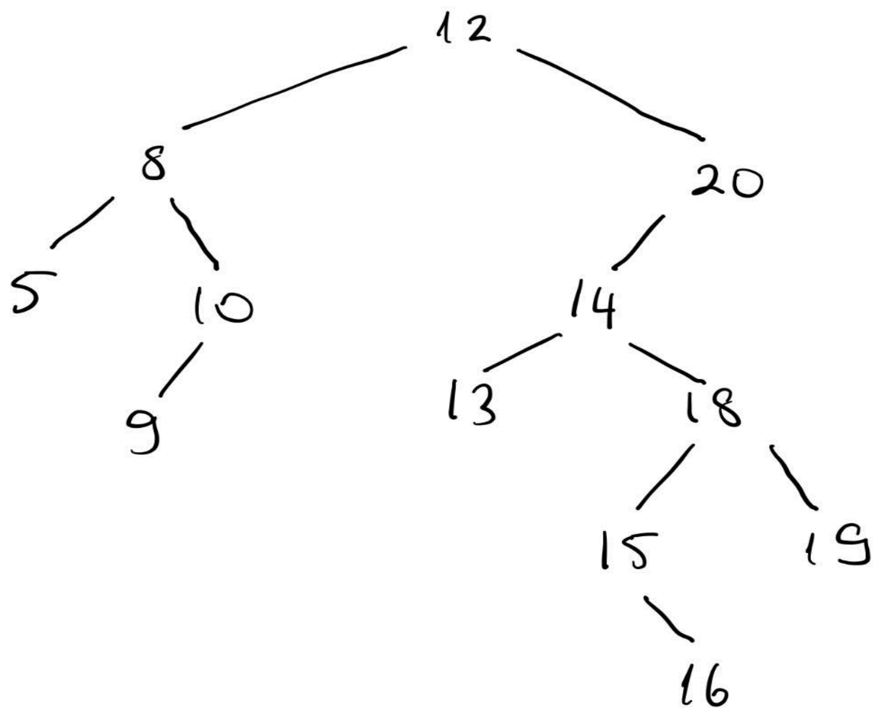
## Question 1:

Part a:

First sequence cannot represent a pre-order traversal of a binary search tree because the right subtree of 12 does not form a binary search tree.

Drawing of Second Sequence :

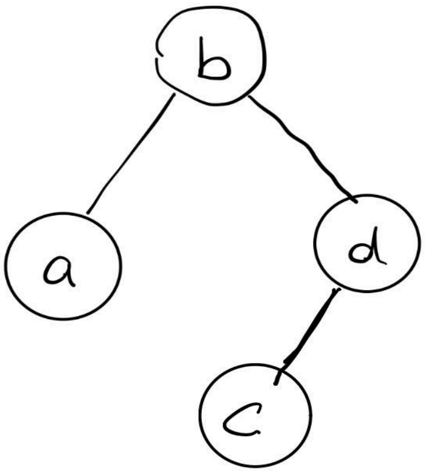
< 12, 8, 5, 10, 9, 20, 14, 13, 18, 15, 16, 19 >



Post Order: < 5, 9, 10, 8, 13, 16, 15, 19, 18, 14, 20, 12 >
In Order:   < 5, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19, 20 >

Part B:

We can form this structure with three different ways of insertions. These are: 1) b,a,d,c  2) b,d,c,a  3) b,d,a,c



Part C:

**Postorder: 5, 6, 15, 10, 23, 24, 22, 26, 20**
**Preorder: 20, 10, 15, 5, 6, 26, 22, 23, 24**

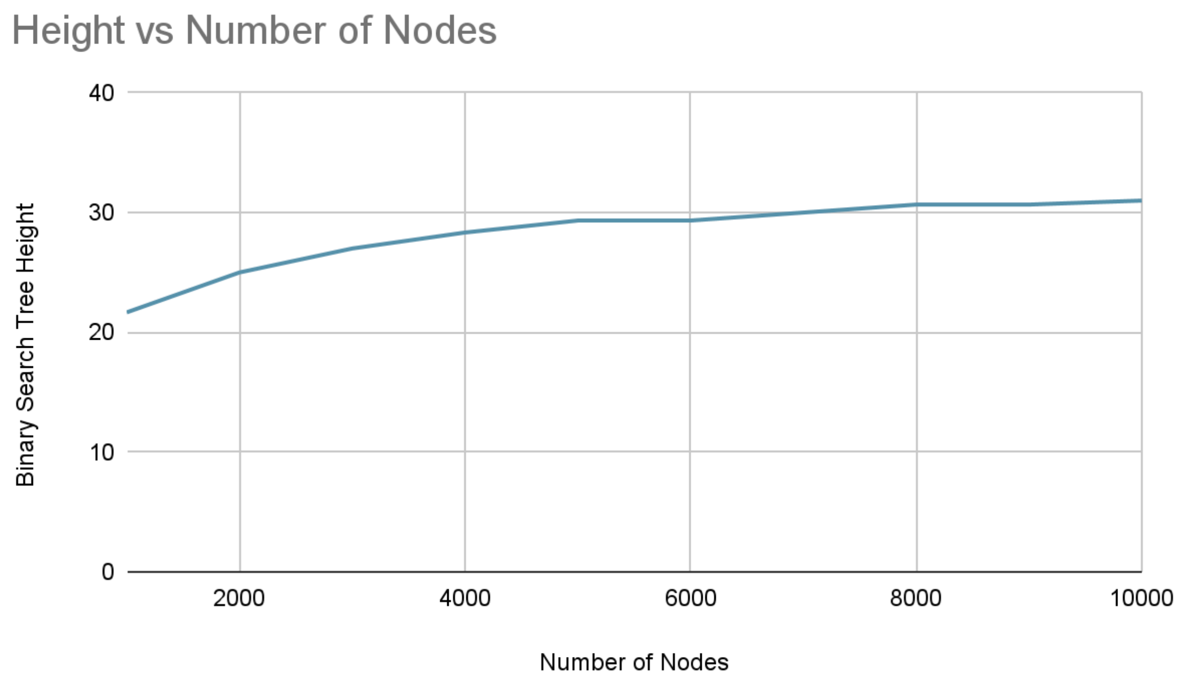
7th Element in Preorder is: 22

Part D:

Firstly 43 is our root so we don't care about the values bigger than 43 because 4,9 will go to the left subtree. Then if we need to compare 4 and 9 with each other we need to insert at least one of them before inserting any of the 5,6,7,8 which would lead to 4 and 9 going into different subtrees which avoids comparison. So if we consider all these 6 numbers {4,5,6,7,8,9} the total arrangement of these numbers is 6!. However we want the first element to be either 4 or 9. Thus the arrangements we want are 2 * 5! because 2 is either 4 or 9 for the first element then we don't care about the rest which is 5! because 4 and 9 will be compared no matter what. I didn’t consider the other numbers because after arranging {4,5,6,7,8,9} they can go anywhere in between them which would not be changing the comparison goal. So the answer is (2 * 5!) / 6! which is 1/3.

## Question 3:

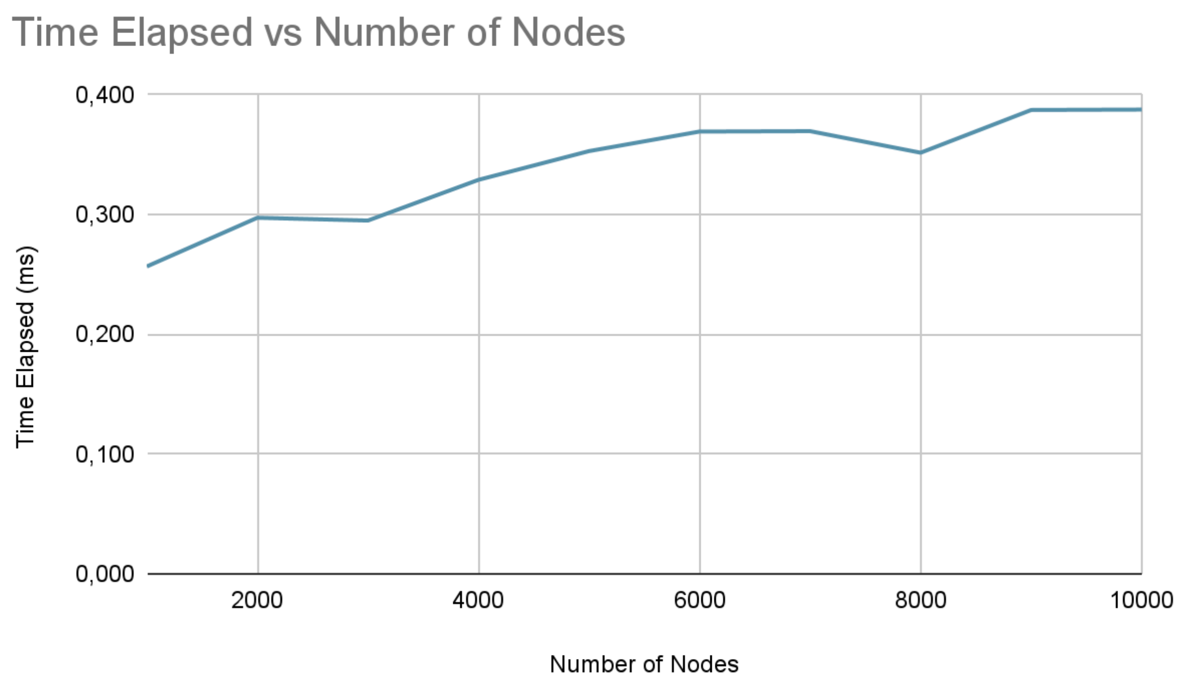
### Part 1:



Plot 1: Height vs Number of Nodes

At first look it looks very similar to a logarithmic function. Normally the expected behavior would be log(n) where n is the number of nodes in a balanced binary search tree. For an unbalanced BST created with random insertions we should expect the height to be slightly higher than the log(n) because the randomly built BST is probably not going to be fully balanced but approximately balanced.

### Part 2:



Plot 2: Time Elapsed (in milliseconds) vs Number of Nodes

If we insert the nodes in random order, the average case of inserting would be still around O(log(n)) , maybe slightly higher because of the randomness. However if we start inserting in sorted order the binary search tree becomes a linked list because we continuously add to the right child if the array is in ascending order or vice versa. So insertion will become O(n) per insertion. Thus, inserting in sorted order will decrease performance highly (from O(nlog(n)) to O(n^2) for total insertions) due to creation of a linked list rather than a BST.

## Part 3:

If we want to keep the tree fully balanced, we can sort the numbers we want to add in, then first we can insert the element in the middle of the array as a root. After that if we go left and right of the middle element 1 by 1 in every loop and add those elements we would get a completely balanced binary search tree which can minimize the height and the insertion cost. If we compare it with other approaches this solution would be very effective in comparison to sorted inserting, lowering the total insertion cost from O(n^2) to O(nlog(n)) and the height from n to log(n). When it comes to comparing with the random insertion this solution would get rid of the slight negative effects of randomness and assure a log(n) height.