CS 202

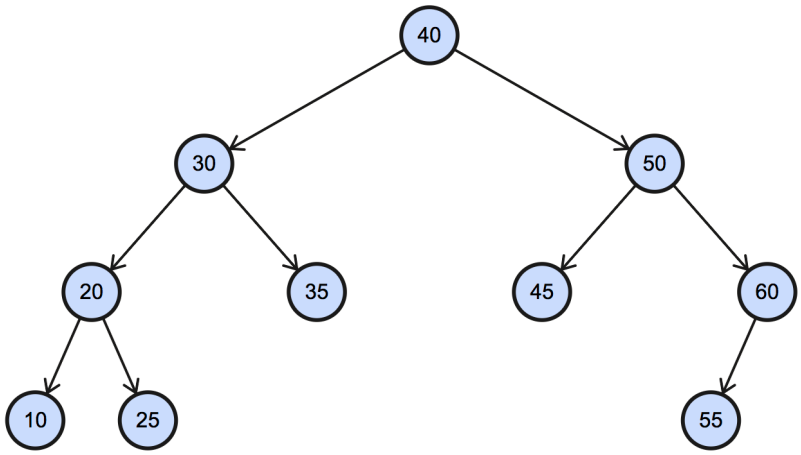
Homework 2

Kerem Ayöz
21501569 Section: 1

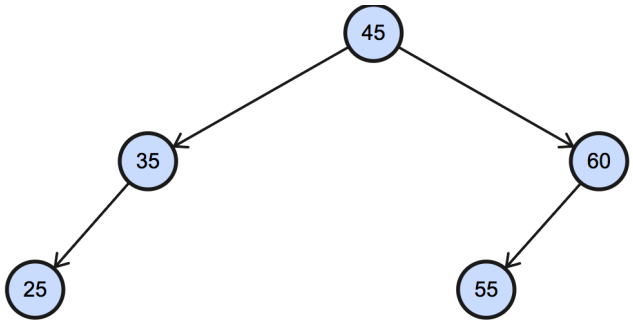
Question 1

a-) Preorder Traversal: *R, G, L, A, O, H, I, T, M*
Inorder Traversal: *A, L, G, O, R, I, T, H, M*
Postorder Traversal: *A, L, O, G, T, I, M, H, R*

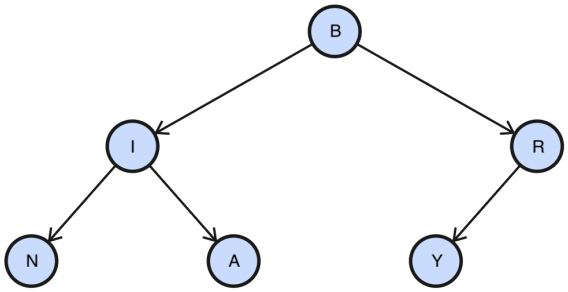
b-) After inserting 40,50,45,30,60,55,20,35,10,25

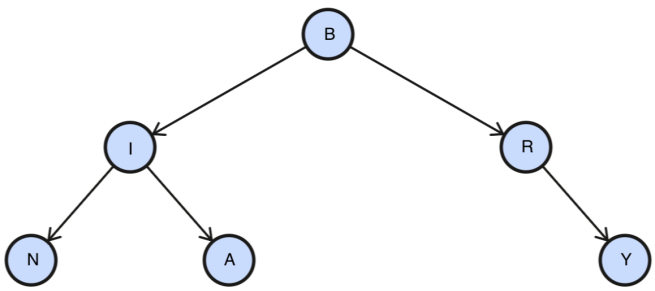


After deleting 10, 40, 50, 20, 30



c-) From that traversals we can build 2 different trees.



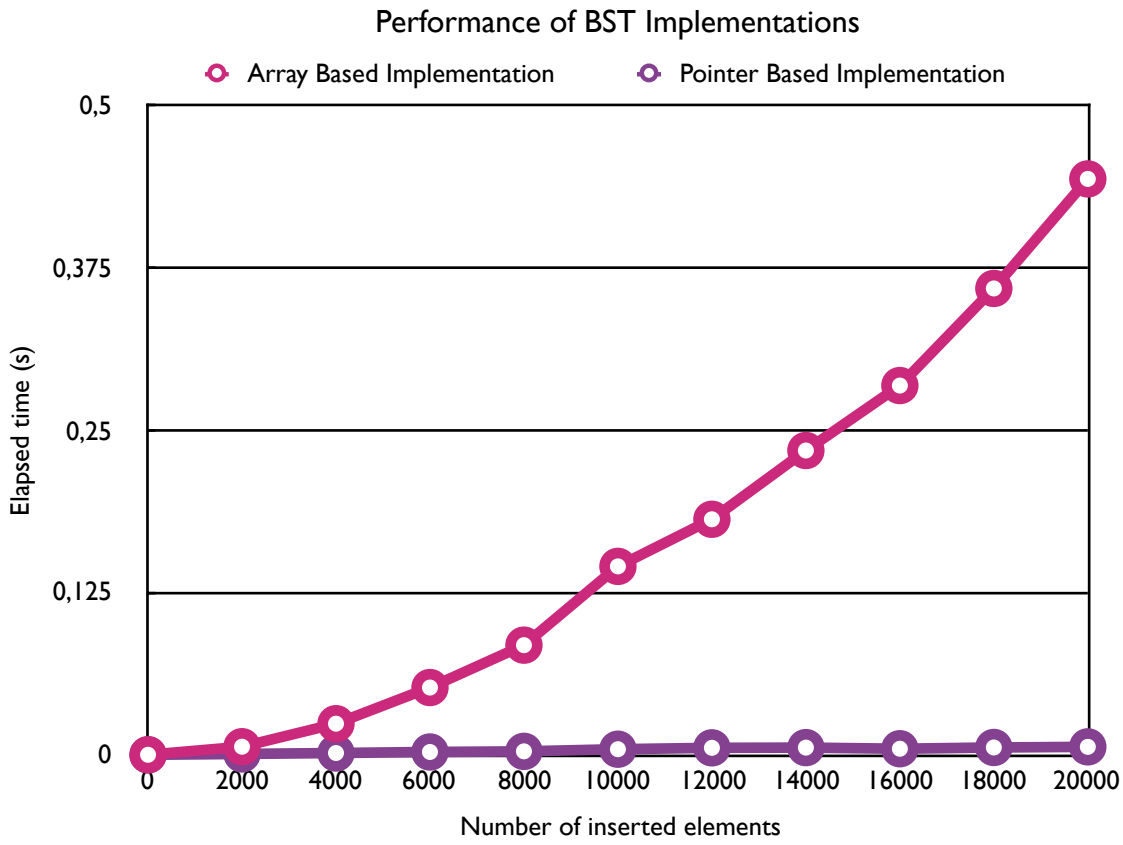


Inorder Traversal: N, I, A, B, Y, R

Inorder Traversal: N, I, A, B, R, Y

Question 3

1. First graph shows the elapsed time for insertions of different number of elements. Binary search tree is implemented in two different ways; array based and pointer based implementations. These implementations have different running times even they are both binary search trees.



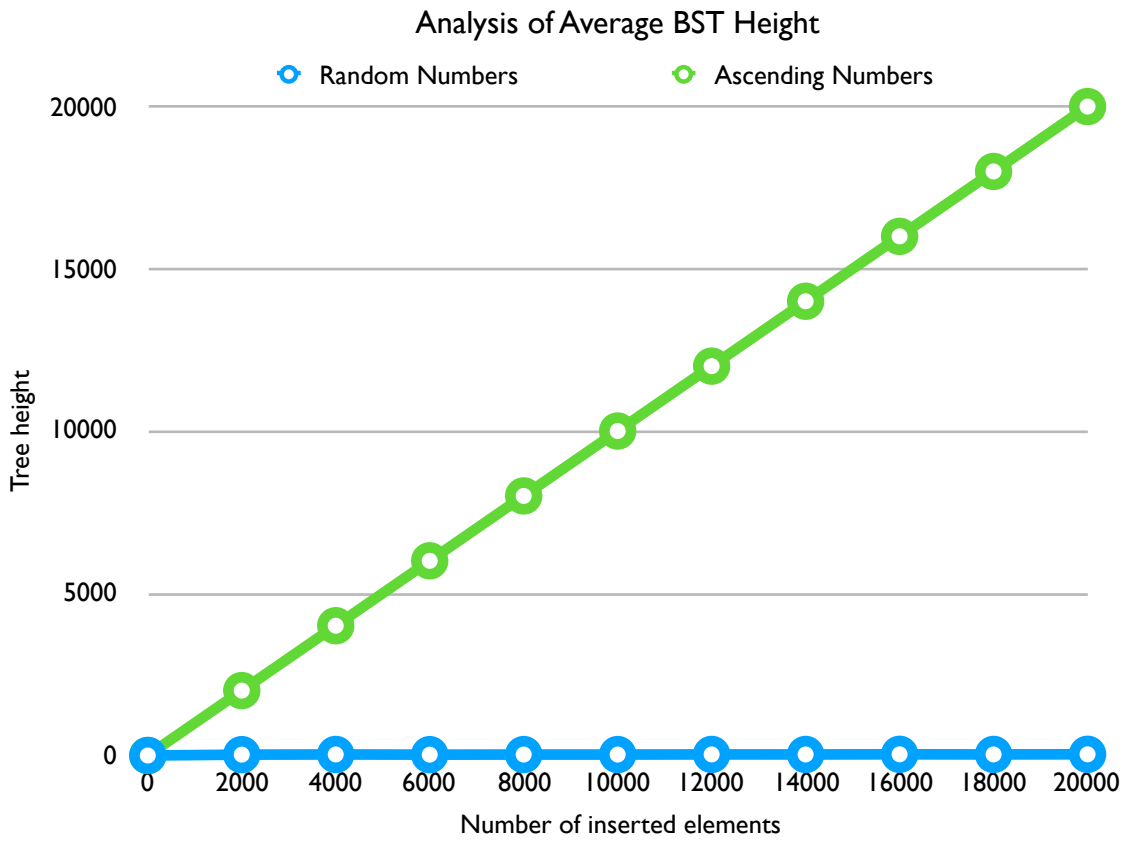
According to the graph, insertion into an array based implementation of binary search tree has bigger running times than insertion into a pointer based implementation of binary search tree. The complexity of the insertion operation to binary search tree have best-case $O(logn)$, worst-case $O(n)$ complexity in theory. Random numbers are inserted, so these results reflects the average case.

In the pointer based implementation, it is not needed to know the size of the tree because elements may be anywhere in the memory. New node is created and connected to the related node in this implementation, no need for any other operation while inserting. So the complexity of that operation is $O(logn)$.

However, because of the array feature that the size of the array is fixed; when the array becomes full, new array is created and all the elements in the old array is deep copied to the new array. Therefore, this deep copy operation slows down the insertion operation in array based implementation. New array is created when we need to insert 3rd, 5th, 9th, … , $(2^n + 1)$th element. So we copy the whole array $logn$ times and time complexity of that operation is $O(nlogn)$.

As it could be seen from the graph, insertion operation is more efficient in the pointer based implementation of binary search tree. It is equal to the theoretical result. However; array implementation could be made more efficient by creating a maximum sized array at the beginning. Deep copy is not needed in that situation.

2. Second graph shows the height change in the binary search tree while the insertion operation occurs. This time only pointer based implementation is used however; for the first case random numbers are inserted, for the second case ascending numbers are inserted to the tree.



According to the graph, height of the tree grows faster while inserting ascending integers to the binary search tree. Also the height is equal to the number of elements, which means tree becomes linear linked list. This is because every element is bigger than the last inserted element and it is inserted to the right child of the last inserted element. So binary search tree behaves just like linked lists in that situation and time complexity of the insertion operation gets the worst case time complexity which is $O(n)$. It is also linear according to graph.

When inserting random numbers, state of the tree may be as same as ascending integer insertion, however expected result is just the average case for this type of insertions. Therefore, height grows logarithmically in that type of insertions despite of the extreme combinations of random numbers. Time complexity of that type of insertions is the average case for the binary search trees and it is $O(logn)$.

The results show that the tree's height gets less if numbers are inserted randomly, which means in the average case scenario occurred. However, experimental results for random integer insertion are distinct from theoretically expected results. Height should be the $O(logn)$ according to theory but tree's height is more than than that number. For instance when 20000 random integer is inserted to the tree, height should be exactly 15 but in the experiments height gets no value less than 33. Even though there is a difference in the results, it could be said that experimental result still not as big as the ascending number insertion and it is still proportional to logarithm of the number of the elements. This difference is occurred because of the randomness of the elements.

As a result, height of the tree affects the performance of the binary search tree operations quite well. The order and the values of the inserted numbers and implementation of the binary search tree also affects the performance of the binary search tree. The order of the numbers could be made ineffective by using "Balanced Binary Tree's". They guarantee that the height of the tree is always proportional to the logarithm of the number of elements. They use different operations to keep tree balanced which means height difference between leaf nodes could be at most 1. This feature avoids the ascending integer insertion and tree is not affected by the order and values of the inserted elements. AVL Trees, 2'3 and 2'3'4 Trees, Red Black Trees and many others are the examples of the balanced binary search trees.