CS 202 - Section 1 22.10.2024

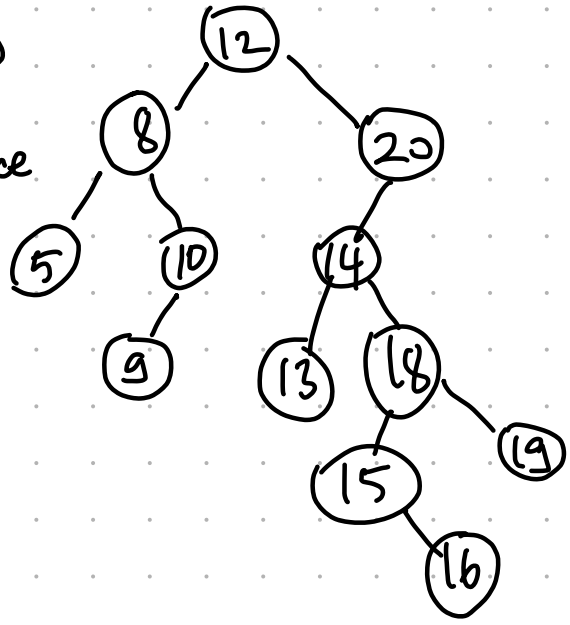
Homework 1

Sinay Uygur - 22203328

Question-1

a) Preorder traversal: root → left → right
inorder : left → root → right
postorder: left → right → root

BST => of the 2nd preorder sequence



The second sequence <12, 8, 5, 10, 9, 20, 14, 13, 18, 15, 16, 19> is the correct one.

Post-order traversal sequence: (of the BST above)

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

In-Order traversal sequence: (of the BST above)

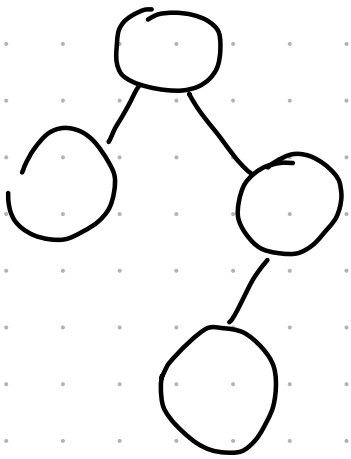
<5, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19, 20>

First sequence was not a correct pre-order traversal because in this type of traversal, after we encounter a node larger than the root, all subsequent nodes should be larger than the root. (because they are in the right subtree of that root.

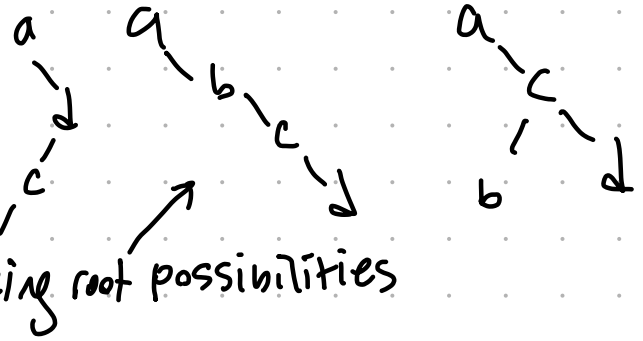
In sequence 1, after 14 we should see 13 (left subtree before right) → 13 appears after 15 and 18 and violates BST property.

But second sequence, we don't see that problem. only that satisfies BST property with that pre-order traversal.

b) tree structure



a<b<c<d



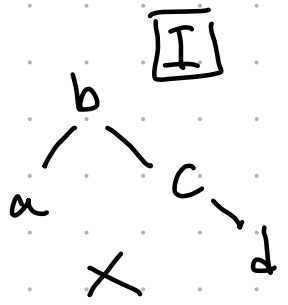
a being root possibilities

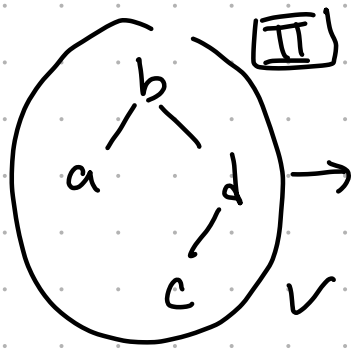
a as a root,

a cannot be a root because there is no element which is less than a. (That element must be on the left of the root, so there must be one.) a doesn't have a left child. So it is impossible for a to be root in this structure. (first element inserted is the root.)

b as a root,

b can be root because it has 1 left child (a) and 2 right children. So it's tree structure can be



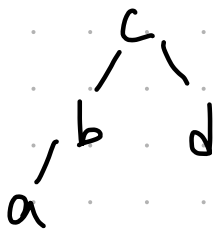


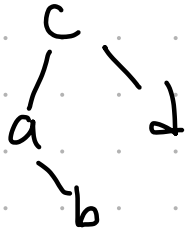
→ this fits the wanted structure.

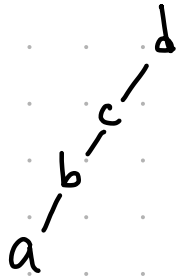
For b to be root it should be first inserted. Then a or d is inserted, at last c should be inserted otherwise tree structure becomes like the I. So insertion orders can be:

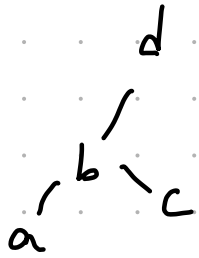
b-d-a-c , b-a-d-c , b-d-c-a ⇒ 3 ways

c and d also cannot be roots because they don't have 1 less and 2 more elements than themselves. Only b satisfies this tree structure.







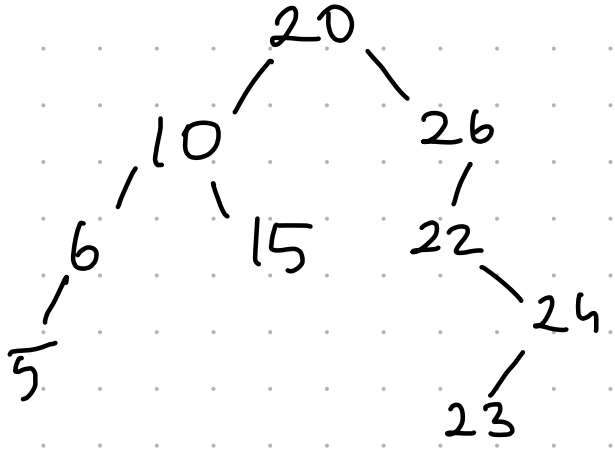


So these possibilities doesn't fit to the wanted tree structure.

Only b being the root, there are 2 ways of inserting elements to that specific structure.

c) Post-Order : 5, 6, 15, 10, 23, 24, 22, 26, 20

Tree representation:



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

↳ 22 is the 7th key in preorder traversal of a BST post-order

d) 100 keys

If the first key inserted is 43, it becomes the root of the tree. Therefore 4 and 9 will be on the left subtree of the root.

If 4 and 9 is compared during insertion, one of them should be on a high level of the other one. (one of them is the ancestor of the other one.

Whether 4 or 9 inserted first, that one must be inserted before any number between them. (5, 6, 7, 8)

43 inserted, then we have 99 numbers left to insert, 42 of them will go to the left subtree.

when 4 and 9 is compared it is obvious that

↳ 4 comes before 5, 6, 7, 8 and 9 or (i)

↳ 9 comes before 4, 5, 6, 7, 8 (ii)

for (i) there are 6 numbers total (including 4)

so the probability $\frac{1}{6}$ ← number 4 selected; 6 ↖ total: 4, 5, 6, 7, 8, 9

for (ii) it is the same $\frac{1}{6}$ ← number 9 selected; 6 ↖ total

So summing these will give the all probabilities 4 and 9 is compared during insertion process. If 4 and 9 doesn't get inserted before other ones (case (i) and (ii) won't happen.)

If 5 is inserted first, then 4 will be compared with 5 and 9 will be compared with 5 and 4-9 won't be compared. This also applies to the other numbers. So, first inserted one must be either 4 or 9.

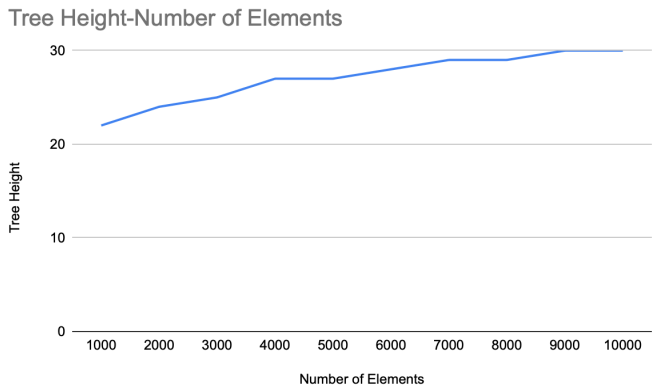
Probability: $\boxed{1/3}$

Simay Uygur CS-202 Homework 1 22203328

Question-3

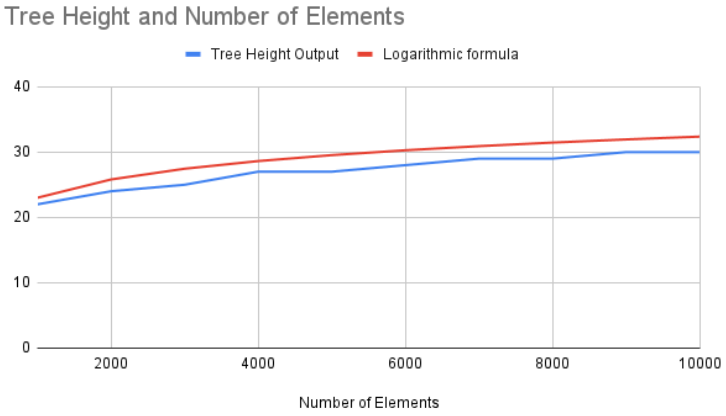
Part 1

Observing Graph 1 plotted from the data, the function for the tree's height doesn't increase linearly. In some points, heights are the same for different numbers of nodes.



Graph 1: BST height versus number of nodes (from the data using Google Sheets)

The expected behavior of unsorted insertion was logarithmic. It generally is calculated with this formula "4.311 * ln(N) - 1.953 ln(ln(N)) -3". The graph below is drawn using Google Sheets and this formula is written to draw the second graph. If this formula is applied and graphs are plotted again, we observe that the lines are close.



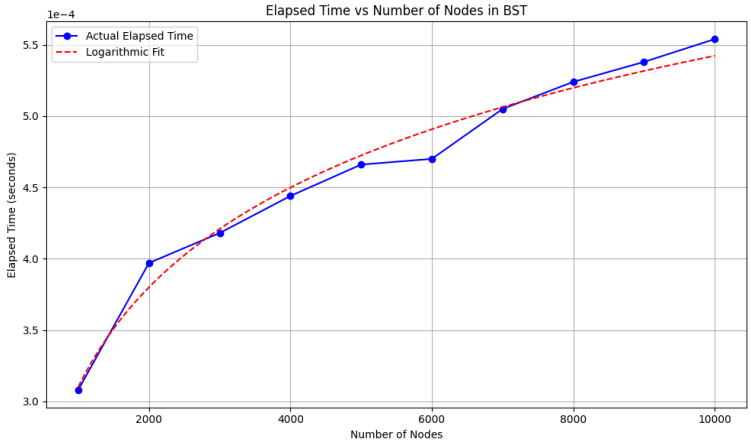
Graph 2: BST height versus number of nodes expected/calculated

Comparing the logarithmic line and output line in Graph 2, we can see that they are quite similar. If the function was linear, we should not see equal heights and there must be a constant slope. Therefore, the function is logarithmic and close to the expected values. This is because of the binary tree's structure. Nodes aren't placed in a linear structure, therefore a binary tree

with height h can have maximum $2^h - 1$ nodes. So the maximum capacity increases with a power of two. After random insertions, we can observe that the tree is close to a balanced tree. So the tree's height is approximately, for total n nodes h = logn. To sum up, because of the tree's binary form, its height doesn't increase linearly like an array or linked list. Unless insertions are not sorted, bst will have approximately logn height on average because it is mostly balanced.

Part 2

The unsorted random number insertion graph is below. Also with dotted lines, logarithmic fit is drawn using Python.

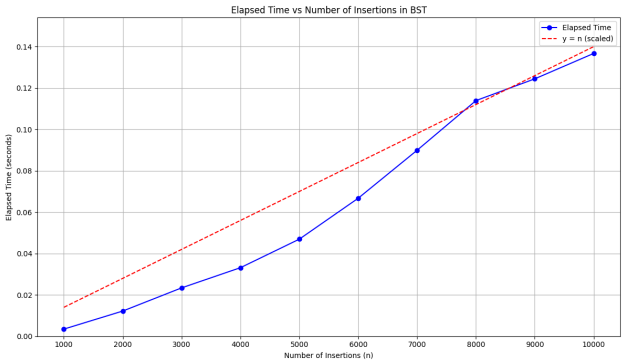


Graph 3: Elapsed Time vs Number of Nodes in BST (unsorted insertion)

In Graph 3, it is obvious that the elapsed time line fits to the logarithmic line. Because of binary tree structure, a search process in the tree starts and after its place is found, insertion is done. This search has the same logic as binary search. Binary search in a sorted array checks halves of the array by comparing with the middle element's value and then checks the right or left of that middle element. In binary search tree its insertion algorithm checks if the inserted key is bigger or less than the current node's key and goes to the right or left node accordingly and finds its possible insertion place. Every time it eliminates almost half of the subtrees. This search has O(logn) time complexity per insertion of one element because it eliminates the left or right half of the tree when comparing with a node. Searching for insertion place takes O(logn) time and inserting it takes O(1) time. Therefore, it takes O(logn) time per insertion to the binary search tree. Tree's height is logn in average, so that matches with the graph. As the tree grows bigger, it takes more time to maintain the insertion processes as it can be seen in the Graph 3. Total time complexity for n number elements is O(nlogn).

If the insertion order is sorted, the tree's root only will have a right or left subtree. Tree structure would be changed and become like a linear linked list (for pointer based bst implementation). To add a new element, every node would be compared with the insertion element. Searching for the possible insertion place takes O(n) time as we compare with every element. So with sorted insertion, the search process for inserting a key doesn't fit the binary search logic. Searching in

this new structure leads to linear search. For total n number of elements, insertion takes O(n^2) time. Graph from the data of sorted insertion is in the below and the dotted line is linear for better visualization.

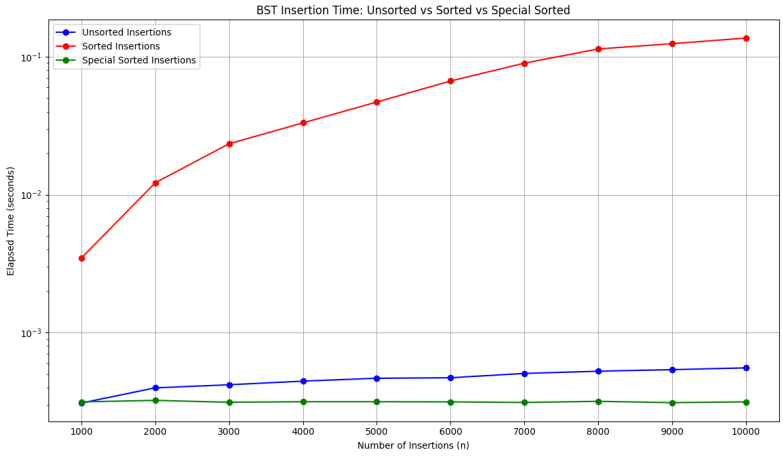


Graph 4: Elapsed Time vs Number of Nodes in BST (sorted insertion)

Part 3

Keeping the tree as flat as possible can be achieved by a specific order of insertions. With that specific order, the tree's height would be minimized and all insertion, deletion retrieval operations' time complexity becomes minimized. Therefore, with a sorted array algorithm must insert the elements in a way that it is searching with binary search logic. Array's middle element should be the root, then left and right subarrays' middle element is the root of right and left subtrees. Thus, it changes the insertion order so that middle elements in the array become the root of the tree and subtrees of the bst. Choosing middle elements would end up with right and left subtrees having close height because it has the same number of elements on its both sides. This also applies to selecting the root for the related subtrees. This approach is the best to keep the tree balanced.

Each 1000 traversal takes almost equal time if data is analyzed. Because it inserts in a special way so that each insertion interval takes minimum time. Also the time comparison plot shows a similar trend, with sorted insertion taking increasingly more time as the number of nodes grows.



Graph 5: BST Insertion Time Comparisons with Unsorted, Sorted and Special Sorted Way

As it can be seen in the Graph 5, sorted insertion is the worst case because of its linear structure. Insertion process traverses through all of the elements in the tree. Nevertheless, unsorted random insertion is the average case as the tree is close to a balanced three and that reduces the time and makes it logarithmic. The last way is the best case as it perfectly fits binary search tree structure and makes the tree almost full. So, bst height is logn in every time.Therefore, insertion operations take O(logn) time. That approach is the best as the tree's height is minimized and every operation done with that bst would take minimum time and becomes the most optimal one.

All of the related methods that output the data are in the analysis.cpp class. The test runs were done in Dijkstra Linux Machine. Each 1000 insertion is calculated separately using ctime library.