z5416824 – Jaden Khuu

COMP3121 Assignment 1

Q3

3.1

Consider an algorithm which will compute the post-order traversal of a binary search tree (BST) given an array of its pre-order traversal, P. This algorithm must also run in $O(n)$ time.

To begin this algorithm, note that the first element of a pre-order traversal is the binary search tree's root and will always be the last element in a post-order traversal. To account for this, initialise a pointer *(mainRoot)* to the value of the first element in array P.

Then, determine which elements would belong to the left and right subtrees of *mainRoot* node (where values lesser than *mainRoot* go into the left subtree and greater go into the right subtree) and store them a left subarray or right subarray.

There will be 2 main subarrays, representing the left subtree (subtree A) and the right subtree (subtree B) of this main root node.

Starting with subtree A, set a pointer to the first element of the subarray A which will represent the root subtree A. Like the main tree, determine which values belong in the left and right subtrees of subtree A. This process should continue down the subtree until the resulting subarrays are of length 1 as this would represent a node with no children.

With these subarrays, we can construct a post-order traversal of subtree A. To do this, append the elements onto a new post-order traversal array *(postArrayA)* in this order (skipping the ones that do not exist):

1. Left most non-visited child in the left subtree of subtree A.
2. The opposing right child.
3. The root of those children.
4. Repeating from step 1 until the left child of subtree A is reached.
5. The left most non-visited child of the right subtree of subtree A.
6. The opposing right child.
7. The root of those children.
8. Repeating from step 5 until the right child of subtree A is reached.
9. Finally, appending the root of subtree A.

This should result in a post-order traversal of the subtree A. **This should be repeated for subtree B, starting with setting a pointer to the first element of subarray B.**

Once that process finishes, the algorithm should be left with a pointer to the main root node of the tree and 2 arrays containing the post order traversal of subtree A and subtree B.

To construct the final post-order traversal of the whole tree, initialise a final post-order traversal array (*finalPostArray)* and concatenate the mentioned elements in the order of:

1. Post-order traversal array of subtree A (*postArrayA*).
2. Post-order traversal array of subtree B (*postArrayB*).
3. Main root node (*mainRoot*).

This will result in the post-order traversal of the binary search tree given only its pre-order traversal. This algorithm has a time complexity of $O(n)$ as its main operations include:

1. Setting pointers - $O(1)$.
2. comparative operators - $O(1)$.
3. Appending onto arrays - $O(n)$.

Thus, these operations are run x number of times depending on the size of the array but are ultimately simplified due to big O notation to result in an algorithm with the time complexity of $O(n)$.

3.2

Consider an algorithm which will compute the post-order traversal of a binary search tree (BST) given an array of its pre-order traversal, P, and an array of its in-order traversal, I. This algorithm must also run in $O(n \log n)$ time.

To begin, note that the first element of a pre-order traversal is the binary search tree's root and will always be the last element in a post-order traversal. To account for this, initialise a pointer *(mainRoot)* to the value of the first element in array P.

With the value of this root node, use a binary search to find the index of its position in the I.

With the index of the root, the algorithm will divide array I into two subarrays, $leftI$ representing the left subtree and $rightI$ representing the right subtree.

Then, also using the index of the root from the previous operation, divide array P into two subarrays, $leftP$ representing the left subtree and $rightP$ representing the right subtree.

The algorithm will rely on constructing a new tree with this information to determine the post-order traversal. This will be done with the help of a function, $constructTree(I, P)$ which takes in the in-order and post-order traversal arrays and returns the root node of that tree.

To begin, create a new tree node and construct the left subtree by recursively calling $constructTree(I, P)$ with the inputs $I = leftI$ and $P = leftP$.

Do the same with for the right subtree to construct the right subtree by recursively calling $constructTree(I, P)$ with the inputs $I = rightI$ and $P = rightP$.

Once this process is done, return the new node as the root of the tree. Using a similar algorithm to the steps explained in **3.1,** use this tree to compute the post-order traversal of the tree.

Thus, the algorithm computes the post-order traversal of the tree constructed using the arrays containing its in-order and post-order traversal.

The overall time complexity of this algorithm will be $O(n \log n)$. This is because each recursive call will have an input of $\frac{n}{2}$ and the depth of the recursion tree will be $O(h)$ or $O(\log n)$. Therefore, the time complexity is $O\left(\frac{n}{2}\right) \times O(\log n)$ which results in $O(\frac{n}{2} \log n)$, simplifying to become $O(n \log n)$.