Solutions to Week 10 Exercises

Ibrahim Chehab

March 25, 2024

1 Questions from FSG Slides

1.1 TreeTheory Pro Max

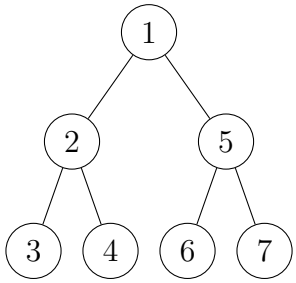
1.1.1

Which of the following statements is true? Select all that apply:

1. The in-order traversal of a binary tree is always sorted
2. The post-order traversal of a binary search tree *might* be sorted
3. The pre-order traversal of a binary tree *might* be sorted
4. The pre-order traversal of a binary search tree is always sorted
5. **All** ancestors of a node in a binary search tree are less than the node
6. **All** ancestors of a node in a binary tree *might* be less than the node

Solution
We will tackle each statement one by one.

1. The first statement is false because the in-order traversal of a binary tree is not always sorted. It is only sorted if the binary tree is a binary search tree.
2. The second statement is true if and only if the binary search tree is filled with all duplicates. Otherwise, the post-order traversal cannot be sorted, as we know children are visited before parents.
3. The third statement is true. Consider the following binary tree:



The pre-order traversal of this tree is `1, 2, 3, 4, 5, 6, 7`. This is sorted.

Note: This is equivalent to a "there exists" statement in MAT102, hence proving it for one case is enough.

4. The fourth statement follows similar logic to the second statement. If the binary search tree is filled with all duplicates, then the pre-order traversal will be sorted. Otherwise, it will not be sorted.

5. The fifth statement is false. Consider an arbitrary `Node` in the left subtree of the root. All ancestors of this node will be greater than the node.

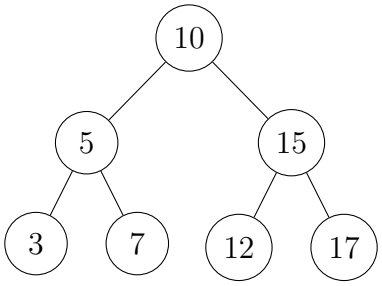
6. The sixth statement is true. Consider the same example from the third statement. The ancestors of node 4 are 1 and 2, which are less than 4.

1.1.2

Given the in-order traversal of an arbitrary binary *search* tree, is it possible to reconstruct the original tree? If so, how? If not, why not?

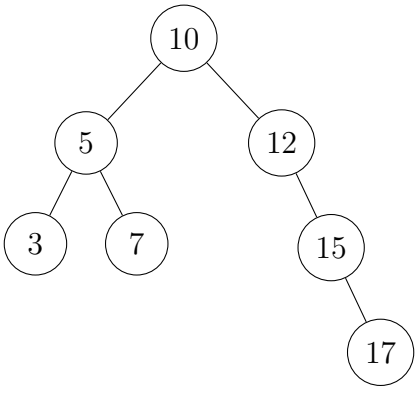
Assume for sake of contradiction that we can reconstruct a BST given only the in-order traversal. That is, define a function $f(lst)$ that takes in a list of integers lst and returns a BST. We would like to prove that this is a *bijective* function, meaning that for every BST, there is a unique in-order traversal, and for every in-order traversal, there is a unique BST.

Consider the following BST:



Running pre-order on this tree would output `[3, 5, 7, 10, 12, 15, 17]`.

Recall our tree rotation methods from the lab. Assume we rotate the *right subtree* to the right. Then, the tree would look like this:



Notice, by rotating the tree, we have not violated the BST properties, hence it is still a BST. However, the *in-order* traversal of this tree remains the same, as `[3, 5, 7, 10, 12, 15, 17]`.

Hence, we have proven that the function $f(lst)$ is not bijective, and therefore, it is not possible to reconstruct a BST given only the in-order traversal.

This becomes more apparent when we consider *AVL Trees* (Out of the scope of the course, but a good exercise for the reader). AVL Trees are a type of BST that are *self-balancing*. This means they maintain the BST property (i.e: their in-order traversal is always sorted), however the tree itself is not unique. Therefore, it is not possible to reconstruct an AVL Tree given only the in-order traversal.

1.1.3

Given a sorted list of integers from 1 to $n, n \in \mathbb{N} \setminus \{0\}$, how do you construct a balanced binary search tree (i.e: height $\log(n)$) from this list? Outline the order of insertion, and explain why this works.

Solution:

Recall, a BST is considered balanced if and only if the height of the left and right subtrees differ by at most 1.

To maintain this invariant, we will insert the median element (i.e: the element at index $\frac{n}{2}$) as the root of the tree. We will then split the list into two parts: the first half of the list will be the left subtree, and the second half of the list will be the right subtree.

This leaves us with two subproblems: constructing the left subtree and the right subtree. We can recursively apply this process to the left and right halves of the list to construct the left and right subtrees, respectively.

The order of insertion is as follows:

1. Insert the median element as the root of the tree, remove this element from the list.
2. Split the list into two halves: the left half and the right half.
3. Recursively insert the new median element of the left half of the list as the left child of the root. Remove this element from the list.
4. Recursively insert the new median element of the right half of the list as the right child of the root. Remove this element from the list.
5. Repeat steps 2 and 3 until the list is empty

This works because after each insertion, we are left with equal amounts of elements to be inserted in the left and right subtrees respectively. This ensures that the height of the left and right subtrees differ by at most 1, hence the tree is balanced.

1.2 IbraNatchi

This question is from Week 6, and as such the solution can be found in the Week 6 solutions document.

1.3 List Comprehensions

This question can easily be solved by running the code in a Python interpreter. This is left as an exercise to the reader.