Solutions to Week 9 Exercises

Ibrahim Chehab

March 25, 2024

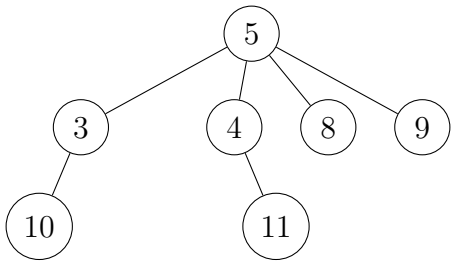
1 Questions from FSG Slides

1.1 TraversalOverflow

In this problem, you are tasked with matching the trees to their respective traversals.

1.1.1

1. `5, 3, 10, 7, 11, 8, 9.`
2. `5, 11, 10, 9, 8, 4, 3.`
3. `10, 3, 11, 7, 8, 9, 5.`
4. `10, 3, 11, 4, 8, 9, 5.`

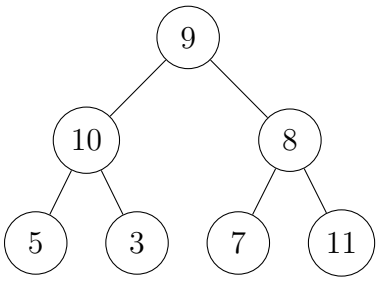


Solution:
Immediately, we can rule out option `1`, since that traversal includes a `7`, yet the tree does not. We can also eliminate option `2`, since it jumps from the *root* directly to a *leaf*, skipping the internal node `4`.

This leaves us with options `3` and `4`. Once again, we notice option `3` contains a `7`, which is not present in the tree. Therefore, the correct answer is option `4`, where option 4 is a *post-order* traversal of the tree.

1.1.2

1. `5, 3, 10, 7, 11, 8, 9.`
2. `9, 10, 8, 5, 3, 7, 11`
3. `10, 3, 11, 7, 8, 9, 5.`
4. `10, 3, 11, 4, 8, 9, 5.`



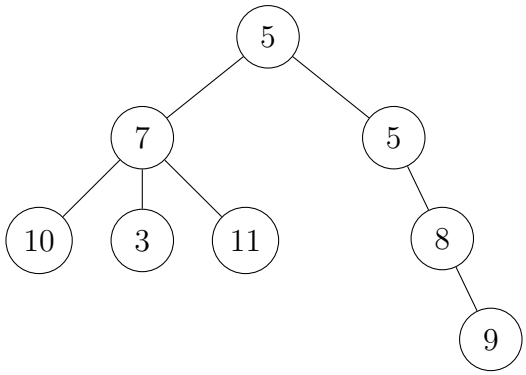
Solution:
We can immediately eliminate options `3` and `4`, because:

- Option `3` is in random order and doesn't follow any traversal known in CSC148H5.
- Option `4` contains a `4`, which is not present in the tree.

This leaves us with options `1` and `2`. We can see that option `1` is a *post-order* traversal of the tree, while option `2` is a *level-order* traversal.

1.1.3

1. `5, 3, 10, 7, 11, 8, 9.`
2. `9, 10, 8, 5, 3, 7, 11`
3. `10, 3, 11, 7, 8, 9, 5.`
4. `10, 3, 11, 4, 8, 9, 5.`



Solution:
Immediately we can eliminate all the options because none of them have two `5`s in the tree. Hence, this question has no answer.

1.2 TreeTheory

In this problem, you are tasked with answering questions about trees.

1.2.1

Consider an arbitrary binary tree of height n. What is the maximum number of leaves that a binary tree of height n can have? What is the minimum number of leaves that a binary tree of height n can have?

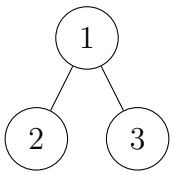
Solution:
Like all theory questions, it is important to first draw out some examples. This strategy will help you in upper-year courses (Especially CSC236).

Clearly, the minimum number of leaves a binary tree of height n can have is 1. This is because a tree of height n can be a single node, which is a leaf. 1 leaf can also occur if every node in the tree has one child (i.e: The tree is a glorified linked list).

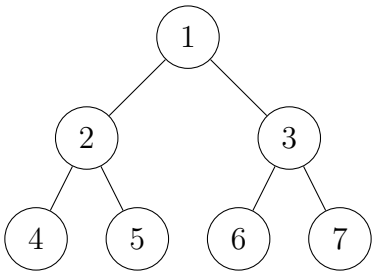
For the maximum number of leaves, we can see this happens when the tree is a *complete* binary tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. Let's draw a few examples and see if we can find a pattern.

Height 1: 1 leaf.

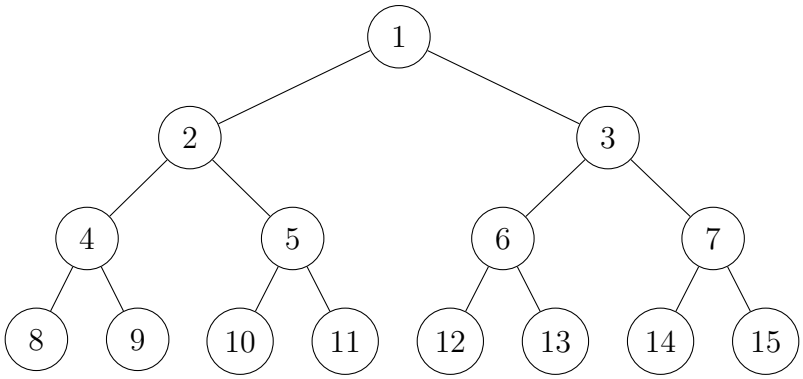
Height 2: 2 leaves.



Height 3: 4 leaves.



Height 4: 8 leaves.



We can quickly see that the number of leaves in a complete binary tree of height n is 2^{n-1}. Therefore, the maximum number of leaves a binary tree of height n can have is 2^{n-1}.

1.2.2

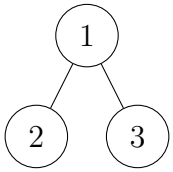
Consider an arbitrary Binary Tree of size n. What is the maximum height h_1 that can be achieved? What is the minimum height h_2 that can be achieved? Justify your answer.

Solution:
The maximum height h_1 that can be achieved is n. This is because a tree of size n can be a glorified linked list, where each node has one child. This is a tree of height n.

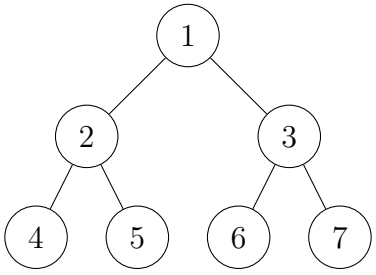
The minimum height once again involves a complete binary tree. A complete binary tree of size n has height $\log_2(n)$. Let's draw a few examples to see if we can find a pattern.

Height 1: Size 1.

Height 2: Size 3.



Height 3: Size 7.



Very quickly, we can see that the number of nodes given a height is $2^h - 1$. Solving this equation for h gives us $h = \log_2(n+1)$. Therefore, the minimum height h_2 that can be achieved is $\lceil \log_2(n+1) \rceil$. Note that we use the ceiling function because the height of a tree must be an integer.

1.2.3

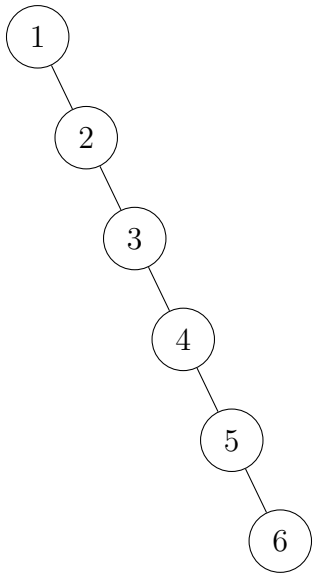
Consider the `search` operation for a *Binary Search Tree*. What is the **worst case** time complexity for the `search` operation? When does it happen? Give an example of a tree that would cause this to happen.
Are BST operations *in general* always $\mathcal{O}(\log n)$? Why or why not?

Solution:
The answer for this question comes from the lecture slides.

The worst case of the `search` operation for a *Binary Search Tree* is $\mathcal{O}(h)$, where h is the height of the tree. h is bounded by

$$\log_2(n) \leq h \leq n$$

Where n is the size of the tree. Consider the following BST:



Searching for the value 6 in this tree would take $\mathcal{O}(n)$ time. This is because the tree is a glorified linked list, and the value 6 is at the very end of the list.

1.2.4

You are working on a new sorting algorithm known as *IbraSort*. *IbraSort* works by first inserting all the elements into a *Binary Search Tree*, and then performing an *In-Order Traversal* to retrieve the sorted elements. What is the time complexity of *IbraSort*? Is it ever possible for *IbraSort* to have a complexity of $\mathcal{O}(n \log n)$? Why or why not?

Solution:

The complexity of *IbraSort* depends on the order of insertion into the BST. If we insert an already sorted (or mostly sorted) list into the BST, since it's not an AVL Tree, it will become unbalanced very quickly and will increase the time complexity of the `insert` operation (See lecture slides).

To understand the time complexity of *IbraSort*, let's consider the worst case scenario. If the BST is unbalanced, the height of the tree will be equal to the number of elements inserted. This means that the `insert` operation will take $\mathcal{O}(n)$ time for each element, resulting in a total time complexity of $\mathcal{O}(n^2)$ Inserting n elements into the BST, where each element takes n steps to insert $\implies \mathcal{O}(n \cdot n) \implies \mathcal{O}(n^2)$.

However, if the BST is balanced, the height of the tree will be $\log n$, where n is the number of elements inserted. In this case, the `insert` operation will take $\mathcal{O}(\log n)$ time for each element, resulting in a total time complexity of $\mathcal{O}(n \log n)$.

Note that we can disregard the complexity of the `in-order traversal` since it is $\mathcal{O}(n)$, which is less than the complexity of the `insert` operation. Hence, it does not contribute to the big-oh notation.

Fun fact: In CSC263, you will learn about HeapSort and TreeSort, and will also learn about Amoritzed Analysis, which applies to IbraSort

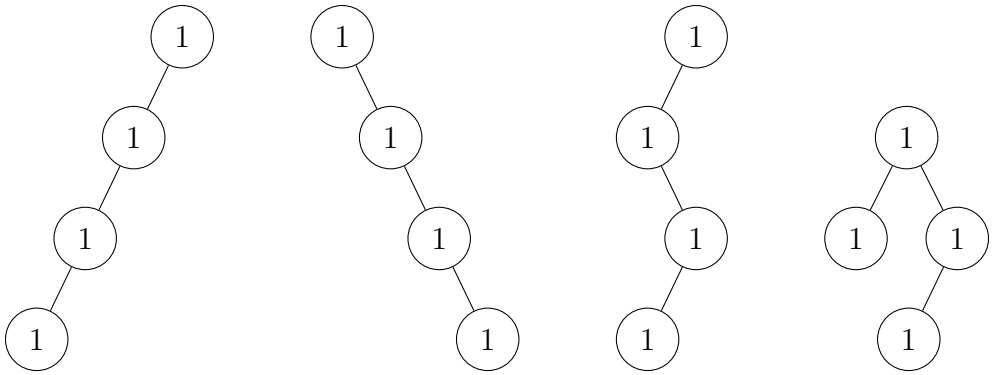
Therefore, the time complexity of *IbraSort* is $\mathcal{O}(n^2)$ in the worst case, but it can be improved to $\mathcal{O}(n \log n)$ if the BST is balanced. Achieving a balanced BST requires inserting the elements in a specific order that follows a *normal distribution*, where the median is chosen as the root and the elements to the left are less than the root, while the elements to the right are greater than the root.

1.2.5

Consider an arbitrary *Binary Tree*. Given it's preorder traversal and postorder traversal, is it possible to reconstruct the tree? Why or why not? What if we consider a *BST* instead?

Solution:

This is not possible. Consider the following counter-example.
Pre-Order: `[1, 1, 1, 1]` Post-Order: `[1, 1, 1, 1]`
The following can be a mulitude of trees, including:



Hence, it is not possible to reconstruct the tree given only the pre-order and post-order traversals.

This *is* possible with BSTs however. This is due to the BST properties that allow us to determine the root of the tree, and the left and right subtrees, **assuming there are no duplicate values in the tree**.

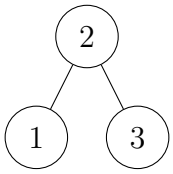
2 Homework Questions

2.1 Pre-order to BST

This question requries us to reconstruct a BST given its pre-order traversal, assuming it has no duplicate values. As a challenge, it also asks us to raise an `Exception` if the pre-order traversal is invalid.

We will first solve this problem without solving the challenge, then we will modify our solution to include the challenge.

To solve this probem, we must remember the format which pre-order outputs. Consider the following BST:



The pre-order traversal of this tree is `[2, 1, 3]`

Very quickly, we notice that the *first* element in the list, call it n, is the *root* of the current level of the tree. Now we must figure out how to differentiate the rest of the tree's nodes. To do so, we can utilize a technique known as *partitioning*, where we split up the elements remaining in the list into two *sublists*, one with elements greater than n, and one with elements smaller than n.

We now have a root element, and two valid pre-order traverals of the two subtrees. We can recursively call our function on these two lists to re-build the remainder of the tree:

```
def reconstruct_from_preorder(lst: list[int]) -> BinarySearchTree:
    """Return the BinarySearchTree that is reconstructed from the
   given
    preorder traversal <lst>.
    """
    if len(lst) == 0: # We've reached the leaves
        return BinarySearchTree(None) # Return an empty tree
    else:
        root = lst[0] # The first element is the root
        left_lst = [] # The two lists for the partition
        right_lst = []
        for i in range(1, len(lst)):
            # Actually partition the list
            if lst[i] < root:
                left_lst.append(lst[i])
            else:
                right_lst.append(lst[i])
        # Recursively reconstruct the left and right subtrees
        left_tree = reconstruct_from_preorder(left_lst)
        right_tree = reconstruct_from_preorder(right_lst)
        # Reassemble the tree
```

```
        new_tree = BinarySearchTree(root)
        new_tree._left = left_tree
        new_tree._right = right_tree
        return new_tree
```

2.2

A similar process can be used for *postorder*:

```
def reconstruct_from_postorder(lst: list[int]) -> BinarySearchTree:
    """Return the BinarySearchTree that is reconstructed from the
   given
    postorder traversal <lst>.
    """
    if len(lst) == 0:
        return BinarySearchTree(None)
    else:
        root = lst[-1]
        left_lst = []
        right_lst = []
        for i in range(len(lst) - 1):
            if lst[i] < root:
                left_lst.append(lst[i])
            else:
                right_lst.append(lst[i])
        left_tree = reconstruct_from_postorder(left_lst)
        right_tree = reconstruct_from_postorder(right_lst)
        new_tree = BinarySearchTree(root)
        new_tree._left = left_tree
        new_tree._right = right_tree
        return new_tree
```

The challenges are left as an exercise to the reader. The hint is to use a helper function that takes in the list, the minimum and maximum values that the current node can have, and the current index in the list. Remember, as you go *down* the tree, the restrictions on the values of the nodes get *tighter*. If you find a node that is out of the tighter bound, you know that the pre-order traversal is invalid.

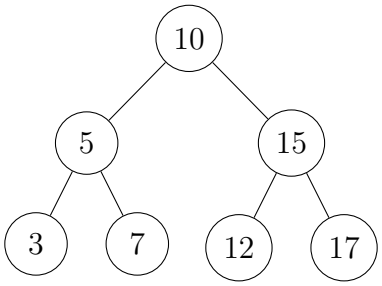
2.3

In the above examples, we have shown that given a pre-order or post-order traversal, we can reconstruct a BST (assuming no duplicates). However, we have not considered an *in-order* traversal. Let's prove that this is not possible.

Proof. 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

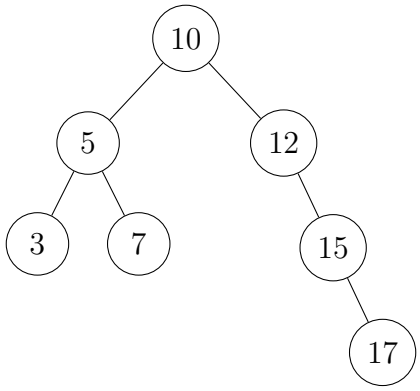
a BST. We would like to prove that this is a *bijective* function, meaning that for every BST, there is a unique in-order traversal, and for every in-order traversal, there is a unique BST.

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.

□