12.2 - Week 12 - Applied - Practical

Introduction

Objectives

- Becoming better acquainted with the implementations of various Binary Tree and BST methods.
- Seeing the complexity benefits of precomputing values.

Recursive calls on Trees

Before we get started, we should talk a bit about the general structure of recursion on Trees. This can normally take two forms:

- Adding an `aux` method to the Tree class, to solve the recursion on any subtree
- Adding the same method to the TreeNode class, to solve the recursion on any subtree

In general though, we normally want to turn a problem of "Solve this on this particular tree" to "Solve this problem on any subtree". With this information we can then piece together the solution on the entire tree.

This is one of the cornerstones of Dynamic Programming, which is covered in FIT2004

Printing Trees

Let’s warm up with something simple but useful, pretty-printing trees.

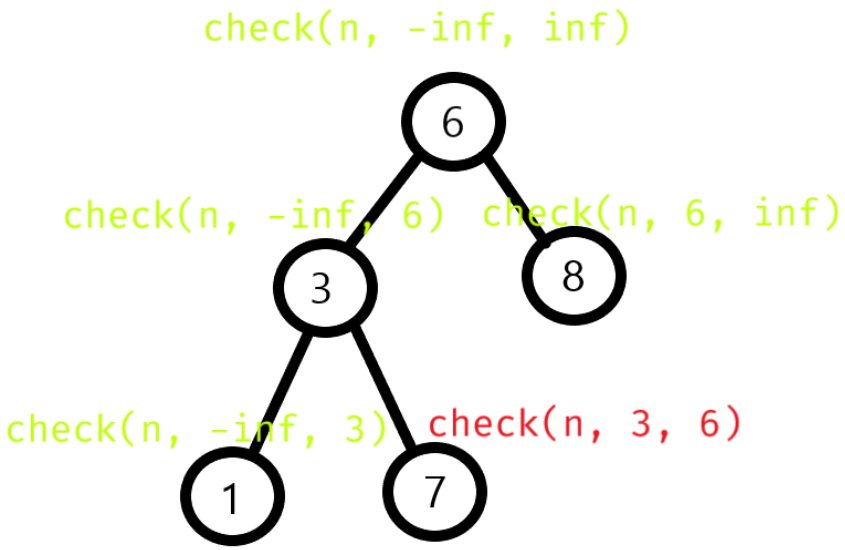
In the `bt.py` file given, try to fix the bugs present so that `__str__` works. (Try pressing "Run" to get an idea of what is being printed vs. What should be printed).

Implementation and Confirming Invariants

Next let's get comfortable by implementing some BST methods, and just confirm that a BST is in-fact a BST!

Given the BST implementation given, do the following:

- Implement a method `check_meets_bst_bounds(self, current: TreeNode, lower_bound: K, upper_bound: K)` which checks that all nodes in the `current` sub-tree satisfy the bounds given. These bounds should update when making recursive calls to reflect the requirements of the BST invariant. (See image below for clarity).
- After running the the code and confirming the BST implementation is incorrect, fix the `insert_aux` and `delete_aux` method to pass all tests.



Implementing TreeSort

Now that we have a valid BST implementation, let's work on TreeSort.

Remember from the Videos / Workshop that TreeSort has two main steps, insertion, and then inorder traversal.

In `bst.py`, implement the method `treesort(input_list: list) -> list`, and any required methods in the BinarySearchTree class from last Exercise.

Building Perfect BSTs

We've seen in the Workshops and Applied Theory classes that having perfectly balanced BSTs give us better complexity bounds

But what if *we could build the BST to be balanced from the beginning?* If we know everything that will go into the BST, surely we can keep it perfectly balanced.

In `balanced_bst.py` , add method `make_balanced(input_list: list) -> list`, that will make a reordered version of `input_list`, so that, when the output list is inserted into a BinarySearchTree, one-by-one, the resulting tree has height at most

$$\lfloor \log_2 \left(len \left(\text{input\_list} \right) \right) \rfloor$$

Advanced Question - Counting Perfectly Balanced Trees

You might've noticed in the previous exercise that we had a lot of choice in the ordering of elements, even with that strict height restriction.

In this exercise, we'll show that this number of choices is actually super-large, and that loosening the height constraint even a bit makes this number even more massive.

In `balance.py`, implement recursive function `num_orderings(n_nodes: int, max_height:int)`, which returns the number of orderings of n distinct nodes, such that the height of the resulting binary search tree does not exceed `max_height`.

If you're up for it, try graphing `num_orderings(n, math.floor(math.log(n, 2)))`, and `num_orderings(n, 2 * math.floor(math.log(n, 2)))`.