CS2040S: Data Structures and Algorithms

Problem Set 4

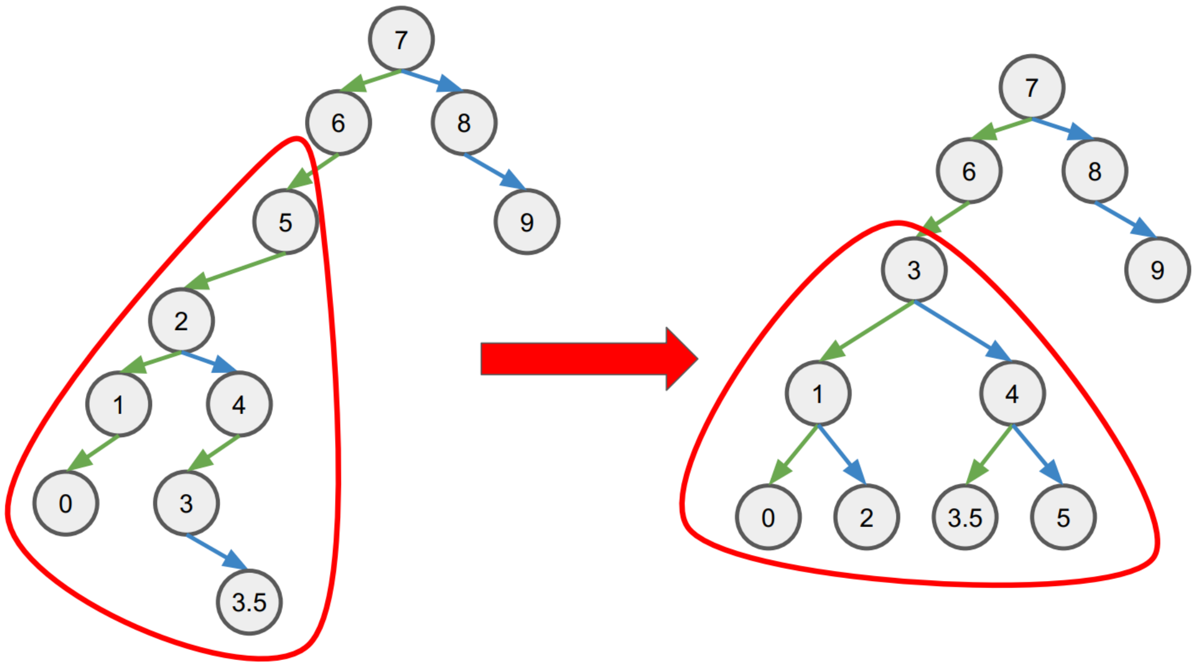
Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person you talked to about the problem (even if you only discussed it briefly). You can do so in Coursemology by adding a Comment. Any deviation from this policy will be considered cheating and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. **(Scapegoat Trees, Part 1)**

Self-balancing trees can seem complicated at times, with various rotations and splits and merges needed to keep everything in balance. Scapegoat Trees, by contrast, are simple. On insertion and deletion, you do not do any anything extra at all! Just do the insert or delete operation as you would in an unbalanced binary search tree.

We do, however, need to do some work when and if parts of the tree become too unbalanced. Whenever things get too bad, a scapegoat node is chosen, and the entire subtree rooted at that node is rebuilt. So most of the time, our tree operations are very fast and simple. And every so often, we have to do some cleanup work to compensate.

Your goal in this problem set is to implement a Scapegoat Tree. In Part 1, you will implement the key step in a Scapegoat Tree: rebuilding a subtree. In Part 2, we will complete the implementation by choosing which nodes to "scapegoat" and when to perform this rebuild operation, building on the work you did in Part 1.



Beware that a key property of the rebalance operation is that it has to be efficient. (Otherwise, the cost of the occasional rebalancing might dominate the cost of the entire data structure!) In particular, if we are rebuilding a subtree containing k nodes, it should only take $O(k)$ time. Thus your implementation of the following methods should be sufficiently efficient to satisfy this goal. (As always, you should make your solutions as efficient as possible.)

Problem 1.a. First, implement an operation `int countNodes(TreeNode node)` that counts the number of nodes in the subtree rooted at the specified `node`.

Problem 1.b. Second, implement an operation `TreeNode[] enumerateNodes(TreeNode node)` that returns an array of nodes in the **inorder traversal** of the subtree rooted at the specified node. (You should make no assumption about the ordering of keys in the tree, but instead simply rely on the tree structure to infer the order. The tree itself may or may not be sorted in the way you expect.)

You are only allowed to use default Java arrays for your solution, i.e. Java API such as `List` and `ArrayList` will not be allowed.

Hint: You may want to use the `countNodes` method from the previous part to determine the size of the array to return in advance. If you resize the array frequently, it is likely to not be as efficient.

Problem 1.c. Finally, implement an operation `TreeNode buildTree(TreeNode[] nodeList)` that builds a perfectly balanced tree (definition below) from the list of nodes previously constructed. Combined with the previous parts, this should give you everything you need to rebuild a subtree.

Do remember that for each node, after you've balanced the two subtrees directly below it, you will need to update that node's left and right child to be the new root of the balanced left and right subtrees respectively. Also, remember to update the parent of the new root of the balanced subtree to be the correct node.

Let us define a perfectly balanced tree. For any two sibling nodes in a perfectly balanced tree, their size (the number of nodes in their subtree, not the height of their subtree) should differ by at most 1.

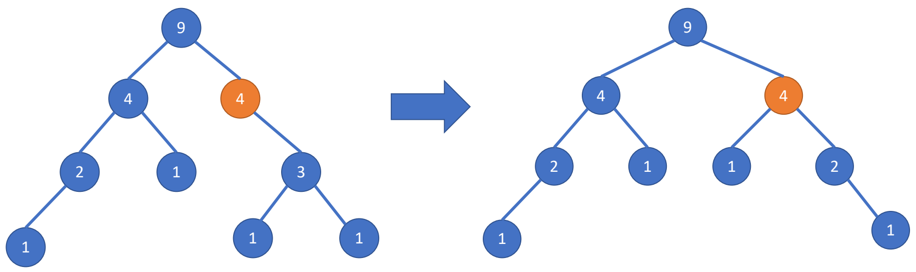


Figure 1: Example of a tree being rebalanced. The number in each node represents its weight. Notice the orange node is violating the balance condition as $3 > (2/3) \times 4$, and that all the other nodes are in balance. The rebalance operation rebuilds the subtree rooted at the orange node, using the rebuild routine from Part1.

Problem 2. (Scapegoat Trees, Part 2)

Next, you will complete the implementation of Scapegoat Trees, building on the balancing routine that you wrote in Part 1. The basic idea of a Scapegoat Tree is that whenever a node is out of balance, we simply rebuild the entire subtree as a balanced tree.

To decide if a node is out of balance, we are going to maintain the ***weight*** of every node: the weight of a node u is the number of nodes in the subtree rooted at u (including u itself). In other words, leaf nodes have a weight of 1 by definition.

We say that a node u is ***unbalanced*** when one of its children has more than 2/3 of the total weight, i.e., if v is a child of u and $weight(v) > (2/3)weight(u)$.

After every insertion, we can check whether any of the nodes on the insertion path became unbalanced, and if so, rebuild the relevant subtree. Your main job in this problem is to modify the implementation of the `insert` routine to perform such checks and rebuild as needed.

A few important notes about solving this problem:

- The solution to this problem will rely on your solution to Part 1 to rebuild the tree. However, mistakes from Part 1 will not be considered mistakes in this problem, so you can solve this problem even if your previous code had mistakes. It may, though, be harder to debug your solution if your `rebuild` routine does not work correctly, and you may want to implement a simpler and less efficient version for the purpose of this problem.
- It is possible to use the `rebuild` routine from Part 1 as a black-box without changing it at all, and that is acceptable. However, you may find it easier (or slightly more efficient) to modify the `rebuild` routine from the previous problem set in small ways. That is also okay. (Please **do not modify nor remove any method signature**, however.)
- Be careful when multiplying or dividing integers by `float` or `double` (non-integer) numbers. Typically, Java correctly interprets the result as a `float` or a `double`, but occasionally

problems can arise if the result is automatically converted back into an `int`.

- For the purpose of this problem, you may assume that every integer inserted into the tree is unique, and there will never be a duplicate insertion. (For more of a challenge, you can implement your code so that a duplicate insertion has no effect on the tree, i.e., it leaves the tree unchanged with a single copy of the node in the tree.)

Problem 2.a. The first step is to modify the implementation of the `TreeNode` class so that each node maintains the weight of its subtree. (Unlike Part 1, we do not want to have to repeatedly count the number of nodes in a subtree, as that can be slow.) Add a variable `weight` to the class, ensure that it is properly initialized whenever a new node is created, and update the `insert` method to ensure that after every insertion, the weights of all relevant nodes have been updated.

Problem 2.b. You will need to update the weights of the nodes when a subtree is rebuilt. You may want to modify the `rebuild` method from Part 1 to update the weights. Alternatively, you may want to implement a new routine `fixWeights(TreeNode u)` that updates the weights of every node in the subtree rooted at the specified node.

Problem 2.c. Implement the method `boolean checkBalance(TreeNode u)` which determines whether the specified tree node is unbalanced.

Problem 2.d. Update the `insert` method so that it works as follows:

- Insert the specified key in the tree using a typical binary search tree insertion (notice that this new node will be inserted as a leaf).
- Identify the **highest** unbalanced node on the root-to-leaf path to the newly inserted node. (Why is it sufficient for us to check only the nodes along the root-to-newly-inserted-leaf path?)
- If there is no such unbalanced node, then we are done. If there is an unbalanced node, then rebuild it.

Note that you already have all the parts you need to solve this last part, just that you will have to now put them together.