Project 2

COP 4530, Summer C 2021

Due July 9, 2020

1 Overview

In this project, you will develop code for a binary search tree variant called a Sequoia. Specifically, you will need to implement member functions for two classes, Sequoia and SequoiaNode. Just like a BST, Sequoia is the main class and represents the tree as a whole, while SequoiaNode represents a single node in the tree.

A header file and a driver have been provided. You are allowed to add member functions to the classes defined in the header; however, you are not allowed to change the data members of the Sequoia or SequoiaNode classes or modify the driver file. The driver file implements several validation functions for the Sequoia; these functions must be defined in the header, and you are not allowed to modify them.

2 Sequoia trees

Just like their real world counterparts, Sequoias are very tall trees. Specifically, every node in a Sequoia must be a *tall* node, where a *tall* node satisfies one of the following:

- the left subtree has at least double the height of the right subtree, **or**
- the right subtree has at least double the height of the left subtree

Sequoias are also BSTs, and every node must simultaneously satisfy the BST property:

- every value in the left subtree is less than or equal to the value of this node, **and**
- every value in the right subtree is greater than or equal to the value of this node

To maintain the tallness of the Sequoia, each node stores a height in addition to its value, parent, left and right children. The Sequoia itself stores a root pointer and size.

3 Sequoia insertion

Inserting a node in a Sequoia tree starts by inserting the node in the same way you would insert a node into a BST. Similar to an AVL tree, we then need to adjust the other nodes in the tree afterwards to maintain their tallness property. Unlike an AVL tree, though, a Sequoia is quite unbalanced in an effort to increase its height.

To fix the Sequoia nodes' tallness, iterate up the tree, starting at the parent of the newly inserted node. At each node, recalculate the height by examining the height of the left and right children. (I recommend writing an updateHeight() function for this purpose, though this function is not required.) Then, check whether the height of the left and right trees satisfy the tallness property. If so, continue on to the parent of this node. Otherwise, there are two cases:

Case 1: the left subtree height is greater than or equal to the right subtree height, but less than double

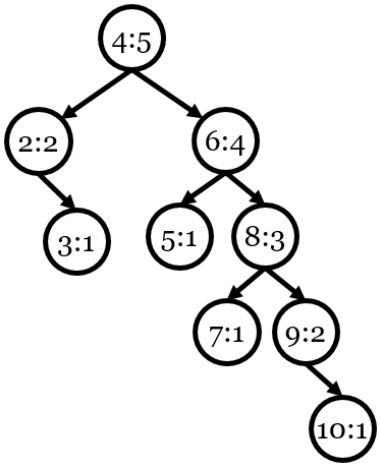
Case 2: the right subtree height is greater than the left but not double

In case 1, we simply rotate the right child of this node to the left, and we resolve case 2 by rotating the left child of this node to the right. In both cases, we will need to update the height of the rotated child; however, both this node and the rotated child will be tall afterwards, and we can continue moving up the tree. These rotations are performed exactly like the rotations for AVL trees.

Once we have iterated all the way to the root node, we need to check the tree to see whether the root node was rotated, and if so, update its root. The easiest way to do this is to include a loop in the `Sequoia::insert` function that updates the root to be its parent if its parent is not null.

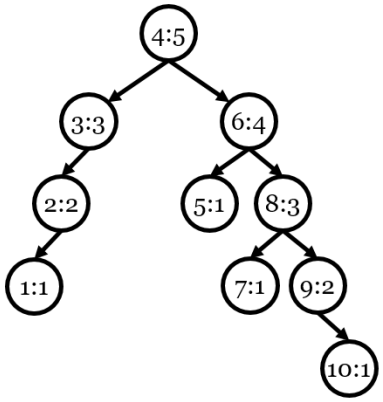
4 Sequoia insertion example

Suppose that we are inserting the value 1 into the Sequoia below. Note that each node is labelled as value:height, and every node in the tree is tall.

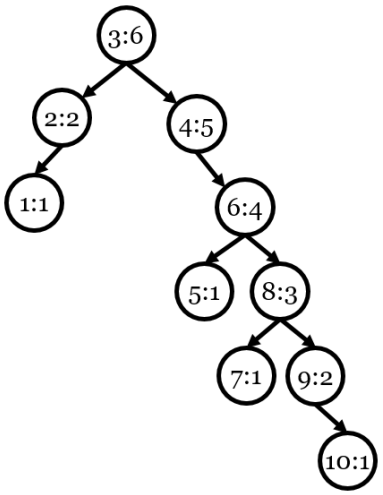


We start by adding the 1 as a left child of 2. This node has height 1.

We then start adjusting the tree, beginning with node 2. The height of node 2 is still 2; however, the node is no longer tall, as the height of its left and right subtrees are both 1 ($1 \not\leq 2(1)$). This qualifies as case 1 of the insertion procedure above since the left tree height is greater than or equal to the right height and but less than double the right height, so we fix it by rotating the right child (3) to the left and updating the height of node 3:



Afterwards, we see that both 2 and 3 are tall, so we continue on to node 4. Node 4 still has height 5; however, it is also no longer tall, as the height of its left subtree is 3 and the height of its right subtree is 4 ($3 \not\geq 2(4)$ and $4 \not\geq 2(3)$). This qualifies as case 2 of the insertion procedure above (since $4 > 3$ and $4 < 2(3)$), so we fix it by rotating the left child (3) to the right and updating the height of 3:



At this point, we have reached the root of the tree, so we can stop: every node satisfies the BST property, has the correct height, and is tall. Afterwards, we need to update the root of the Sequoia from 4 to 3.

5 Sequoia deletion

Deleting a node from a Sequoia is very similar to insertion; however, the tallness property will be restored after at most 1 rotation. To start, delete the given node from the Sequoia in the same manner as a BST. Then, begin iterating up the tree, starting from the parent of the deleted node.

At each node, update the height based on its children, then check whether the node is tall by comparing the left and right subtree heights. If the tree is tall, continue to the parent of this node, but otherwise we have the same two cases as insertion:

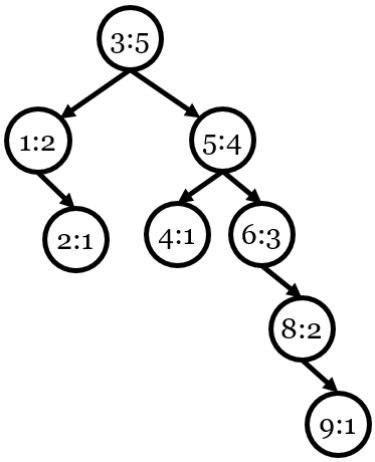
Case 1: the left subtree height is greater than or equal to the right subtree height, but less than double

Case 2: the right subtree height is greater than the left but not double

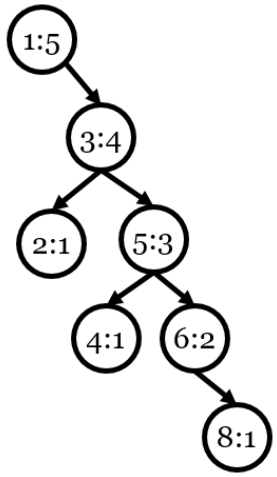
The cases are resolved in the same way as insertion: in case 1, we simply rotate the right child of this node to the left, and in case 2, we rotate the left child to the right. In both cases, we will need to update the height of the rotated child; however, this rotation will guarantee that all nodes in the tree are now tall. As with insertion, we need to update the root in case it was rotated.

6 Sequoia deletion example

Suppose that we are deleting 9 from the Sequoia below:



We start by deleting the 9, so 8 has no children. Then, we start moving up the tree at node 8. The height of 8 is now 1, and node 8 is tall because its left and right subtrees both have height 0 ($0 \geq 2(0)$). The height of 6 is now 2, and it's also still tall: its left subtree height is 0 and the right subtree is 1, and $1 \geq 2(0)$. Continuing to 5, the height of 5 is now 3. Its left subtree height is 1 and its right subtree height is 2, and $2 \geq 2(1)$, so node 5 is tall and we continue to 3. The height of 3 is now 4, but it is no longer tall, as its left subtree has a height of 2 and its right subtree a height of 3 ($3 \not\geq 2(2)$ and $2 \not\geq 2(3)$). According to case 2, we should rotate the 1 node to the right and update its height:



Every node now satisfies the BST property, has the correct height, and is tall. Lastly, we would update the root of this tree to 1. Note that the height of this tree has not changed, so if this tree were a subtree of a larger Sequoia, all of the other nodes would still be tall and have the correct height.

7 Driver file

You have been provided with a simple driver file that reads in values to insert and delete from `input.txt` and performs these operations on an empty Sequoia tree. The driver writes two trees to `output.txt`, one per line: the first is printed after all insertions and the second after all deletions are done.

The `input.txt` file will have 2 lines. The first line has all of the integers to be inserted into the Sequoia tree, separated by spaces, and the second line has all of the integers to be removed from the tree, separated by spaces. All of the values on the second line will also appear on the first line.

All insertions and deletions are performed in the order that the values are entered on each line. Moreover, the driver checks that every node in the tree has the correct height and is tall after every insertion and deletion and will print an error message and stop if some node is incorrect.

8 Submission and grading

You should your source code and header, `sequoia.cpp` and `sequoia.h`, as a zip archive. Your code will be evaluated based on whether it produces the correct output for some number of test cases. You may use any development environment that you like when developing the code; however, it will be compiled and run using `g++` in a Linux environment. Code that does not compile will not receive substantial credit, so be sure that your code can be compiled using `g++`.

The development environment used in class is VS Code (`https://code.visualstudio.com/download`). This IDE includes useful features like syntax highlighting and an integrated terminal window that you can use to invoke a compiler.

Windows users can download MinGW (`https://sourceforge.net/projects/mingw-w64`) to use `g++`, though you will need to add the MinGW bin directory to the Windows path in order for the terminal to recognize the command `g++` (or `gdb` or any of the other tools provided by MinGW).

9 Important note

You are allowed to use any code posted for this course on Canvas, and you are allowed to discuss this project with the course TAs or professor. However, you are *not* allowed to use code from the internet or your peers. Your code will be run through plagiarism-detection software, and violators will be dealt with accordingly.