Lab : Balanced Search Trees

Information

In-class labs are meant to introduce you to a new topic and provide some practice with that new topic.

Topics: Balanced Search Trees

Solo work: Labs should be worked on by each individual student, though asking others for help is permitted. Do not copy code from other sources, and do not give your code to other students. **Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.**

Turn in: Print out the worksheet and fill it out. Upload pictures of your work to Canvas once done.

Starter files: None

Contents

1.1 Introduction

There are multiple types of Balanced Search Tree data structures available for us to use, with the common feature of them being that they remain balanced after any kind of action (insert, remove, etc.) We will look at oneof these tree types and how it works, so that you can gain an understanding of their operations in general. Further down the road, if you need to implement them later, you'll at least know in general how they work.

Recall that with a **binary search tree**, if you happen to insert items in an ascending or descending order, you end up just getting a straight line...:

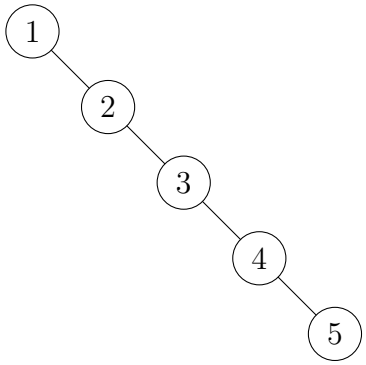


Figure 1.1: Pushing 1, 2, 3, 4, 5

Whereas if you insert things at more-or-less a random unsorted order, we generally will get a more balanced tree:

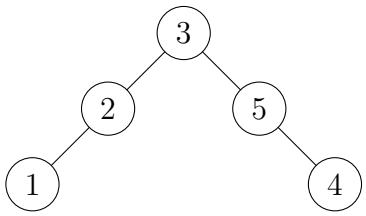


Figure 1.2: Pushing 3, 5, 2, 1, 4 (chosen at random) - more balanced

1.2 AVL Trees

For an AVL tree, the two subtrees of the root differ by no more than one. After an operation, if the height difference is more than one, we do rebalancing to fix the tree.

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(logn)$	$O(logn)$
Insert	$O(logn)$	$O(logn)$
Delete	$O(logn)$	$O(logn)$

(a) AVL Tree efficiency, from https://en.wikipedia.org/wiki/AVL_tree

Say that we have the following unbalanced tree. To balance it, we push the 20-node up, which pushes the 30-node down.

Figure 1.4: Diagram from Data Abstraction & Problem Solving with Walls and Mirrors, 7th ed, Carrano and Henry, Page 593

1.2.1 Height and Balance Factor

(Review) Height of a node: The height of a node is the amount of edges on the longest path between that node and a leaf. A leaf node has a height of 0.

(Review) Height of a tree: The height of a tree is the height of its root node. [1]

(Review) A height-balanced tree: A tree is height balanced (aka, simply *balanced*), if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1.

[1] https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology_used_in_trees

(Note) Left subtree of n: According to the book, "In a binary tree, the left child of node n plus its descendants [is a left subtree of node n]."

However, to work with the definition of a balance factor from *Wikipedia*, we need to use Wikipedia's definition of a subtree: "A subtree of a tree T is a tree consisting of a node T and all of its descendants in T."

So for our purposes, $LeftSubtree(n)$ **is going to be n plus its left descendants.**

Balance Factor: After an operation is performed on an AVL tree, the Balance Factor of each node is calculated. The equation for a node n's balance factor is:

$$BalanceFactor(n) = Height(RightSubtree(n)) - Height(LeftSubtree(n))$$

If the **absolute value** of the Balance Factor is more than 1, then we need to rebalance. In other words, the balance factor for *every node* must be -1, 0, or 1.

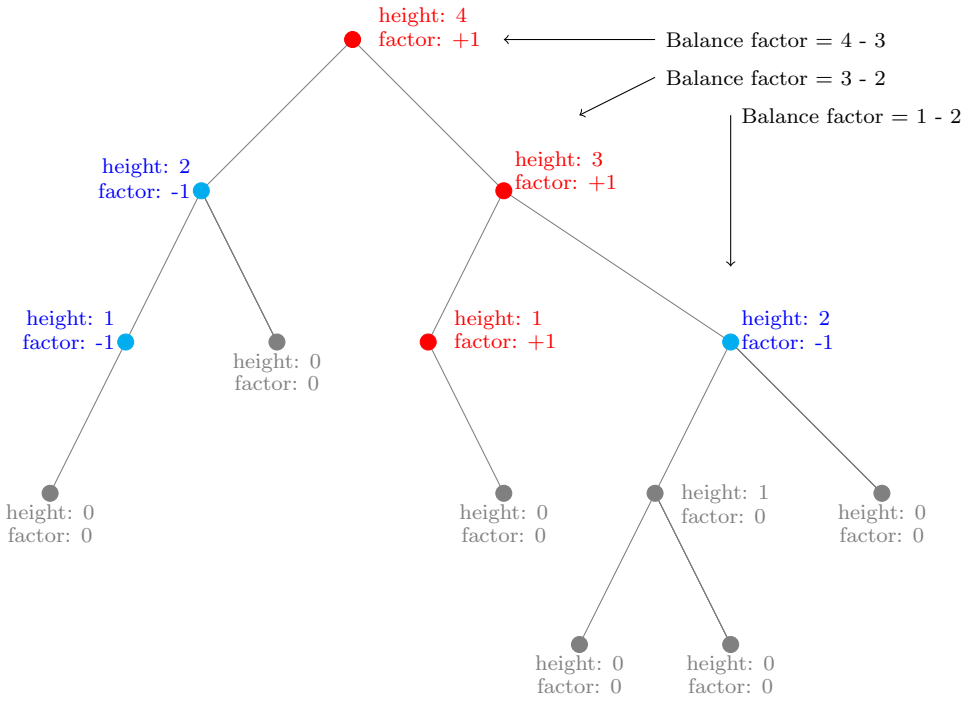


Figure 1.5: AVL tree with balance factors, from https://en.wikipedia.org/wiki/AVL_tree

Now let's cover the different types of rotations that may occur. (Examples from https://www.cise.ufl.edu/~nemo/cop3530/AVL-Tree-Rotations.pdf)

1.2.2 Left Rotation (LL)

Let's take a subtree that has a balance factor of +2 like this:

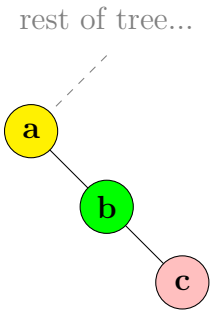
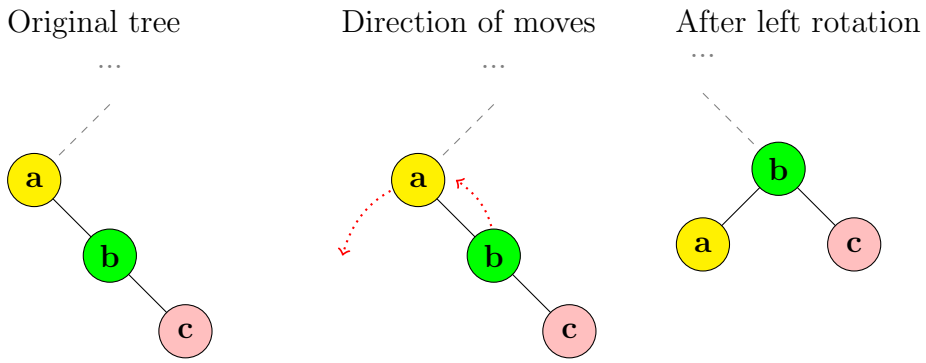


Figure 1.6: Unbalanced tree

Node a has a balance factor of +2, because it's left subtree is height 0 (no children), and its right subtree is height 2. Therefore, we are going to do a **left rotation on a.**

1. a's right child, b, becomes the new root (of this subtree).
2. If b has a left child, it becomes a's right child.
3. a becomes b's left child.

The steps look like this:



1.2.3 Right Rotation (RR)

Let's take a subtree that has a balance factor of -2 like this:

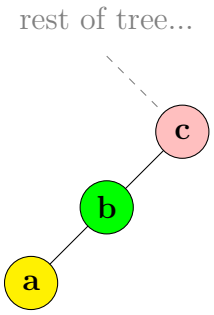
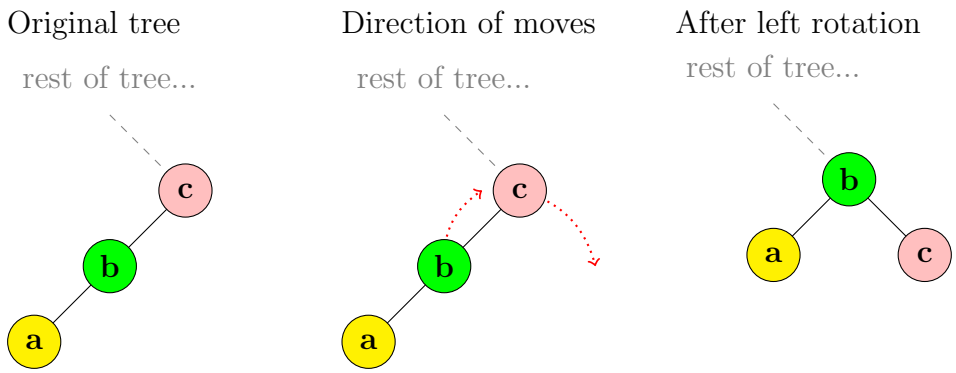


Figure 1.7: Unbalanced tree

This time, c has a balance factor of -2 because its right subtree is height 0, and its left subtree is height 2. We are going to do a **right rotation on c.**

1. c's left child, b, becomes the new root (of this subtree).
2. If b has a right child, it becomes c's left child.
3. c becomes b's right child.

The steps look like this:

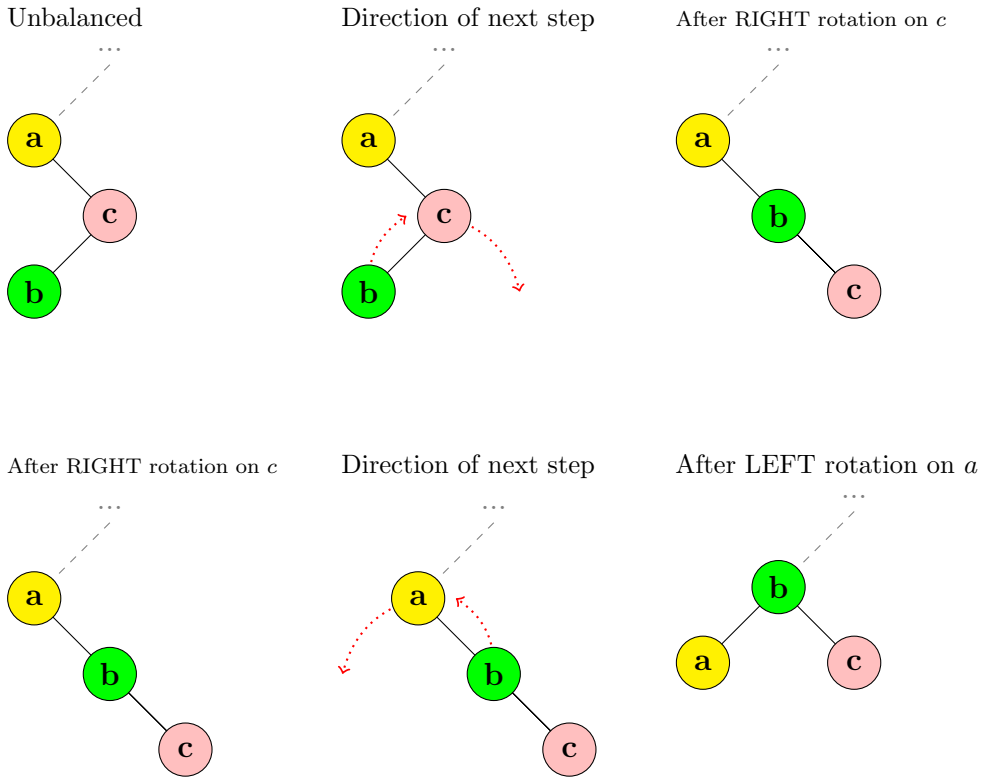


(Note) Sequence preservation: While rotating a set of nodes, keys are only moved "vertically", so that the "horizontal" in-order sequence is preserved. [2]

[2] https://en.wikipedia.org/wiki/AVL_tree#Rebalancing

1.2.4 Left-right rotation (LR) / Double-left

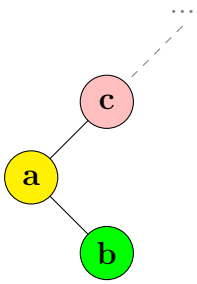
In some cases, we can't balance a tree with one rotation and have to rotate twice. We will need a "left-right" rotation in cases where our tree is right-heavy, but its right-subtree is left-heavy.



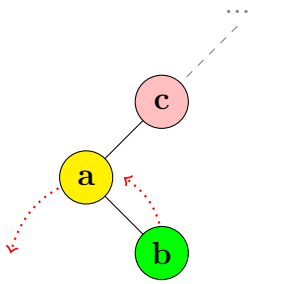
1.2.5 Right-left rotation (RL) / Double-right

We will need a "right-left" rotation in cases where our tree is left-heavy, and its left-subtree is right-heavy.

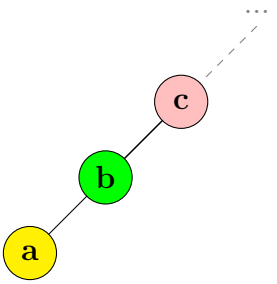
Unbalanced



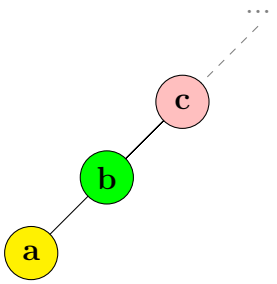
Direction of next step



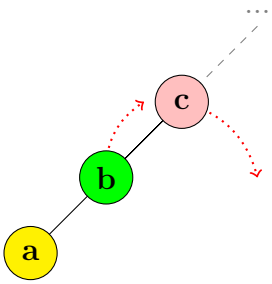
After LEFT rotation on a



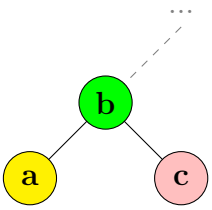
After LEFT rotation on c



Direction of next step



After RIGHT rotation on a



1.2.6 Pseudocode

Pseudocode also from https://www.cise.ufl.edu/ nemo/cop3530/AVL-Tree-Rotations.pdf

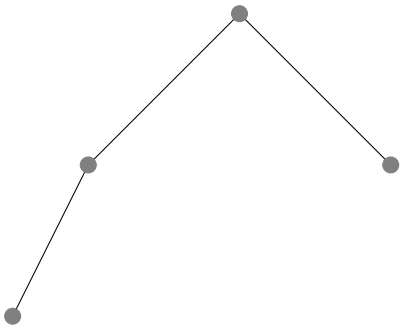
```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
    {
        DO double left rotation (LR)
    }
    ELSE
    {
        DO single left rotation (LL)
    }
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
    {
        DO double right rotation (RL)
    }
    ELSE
    {
        DO single right rotation (RR)
    }
}
```

1.3 Questions

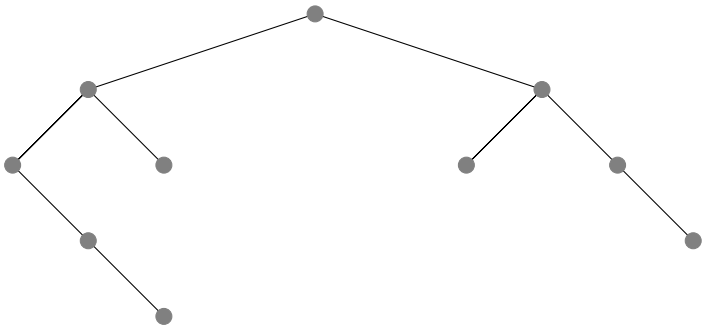
Question 1

For the given trees, label the **height** and **balance factor** of each node.

a.



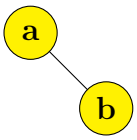
b.



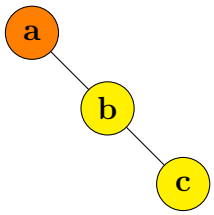
Question 2
Steps of building an AVL tree are listed. Do the first balance operation.

1. Push “a”

2. Push “b”



3. Push “c”



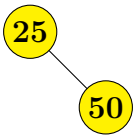
4. Balance the tree:

Question 3

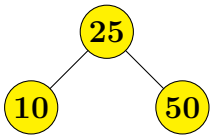
Steps of building an AVL tree are listed. Do the first balance operation.

1. Push "25"

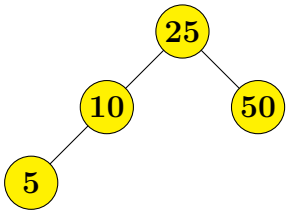
2. Push "50"



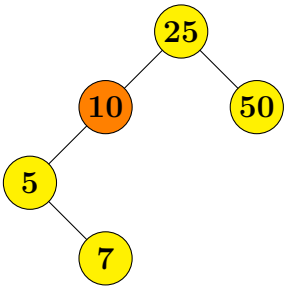
3. Push "10"



4. Push "5"



5. Push "7"



6. Balance the tree: