Lab 11 - AVL Trees

CS 251, Fall 2024

Copyright Notice

Learning Goals

By the end of this lab, you'll:

- Understand the necessity for performing rotations when modifying an AVL tree (balanced tree).
- Practice performing the appropriate rotations whenever the AVL property is broken after an insertion to an AVL tree.
- Implement the rotation functions to complete the implementation of an AVL tree class in C++.

Starter Code

lab11-starter.zip

Tasks

For this lab worksheet, work on the tasks in the **given order**. We strongly encourage collaboration in labs. Work with your peers and ask your TAs questions!

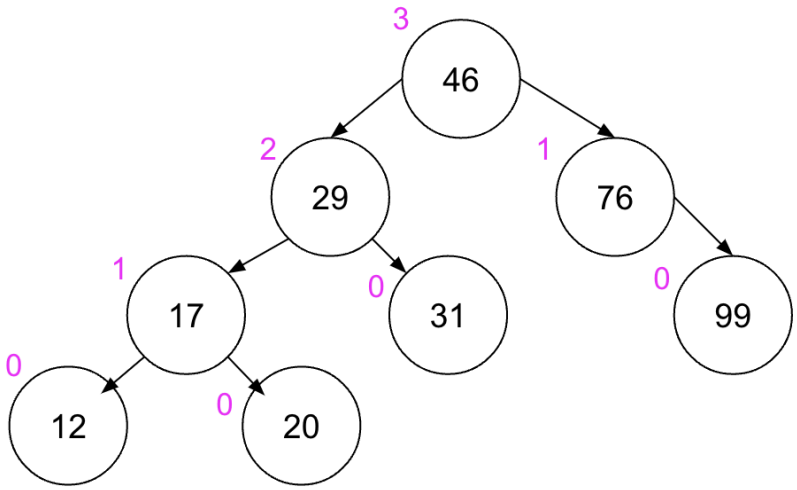
Throughout, we will use the following `Node` definition:

```
class Node {
  int data;
  int height;
  Node* left;
  Node* right;
};
```

Recall that with AVL trees, we need to keep track of the `height` of each `Node`, in order to be able to compute whether or not the balance property has been violated at a `Node`.

In this lab we will first perform some tasks on paper and in the end we will apply what we do in code, while implementing AVL tree insertion.

For all the on-paper exercises we will start by working off the following tree which in what follows we will call `T`:



For your convenience, we have added the height of each node to the left of the node, in pink.

Task 1: Right rotation

Right rotation on paper

Starting off with `T`, insert a node with `data` value **8**.

Question 1.1: Draw the updated tree with the proper BST insertion on `T`, with the updated heights for all pertinent nodes.

Question 1.2: At which node is the balance property broken? Recall that the balance of a node `N` is broken if `abs(height(N->left) - height(N->right)) > 1`. (Note: If you find that height breaks for multiple nodes, you should only choose the first one that breaks as you update heights from the bottom up, i.e. the broken node with the shortest height).

Since we've identified that our AVL tree is broken, we will need to update it by doing one of the rotations that we presented in class. In this case, we have a left-leaning tree so we will perform a right rotation.

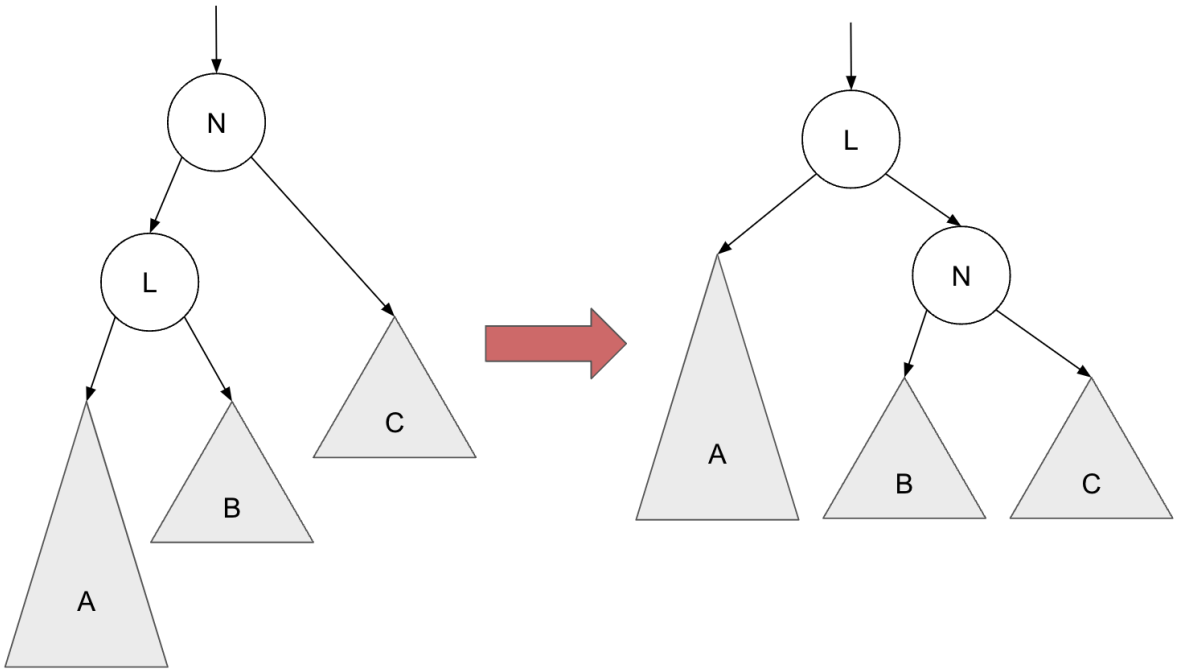
When doing a right rotation, there are three special nodes that we should identify/label:

1. N - the node at which the AVL tree broke.
2. L - the current left child of N
3. B - the current right child of L

Once we have identified these three special nodes, the rotation can be performed in three steps:

1. change N's left child to now be B
2. change L's right child to now be N
3. change N's parent node to now point to L instead of N

Here is a picture that illustrates this scheme:



Question 1.3: Draw the AVL tree that is produced after doing a right rotation on the tree you drew in question 1.1. The rotation is done on the "broken" node that you identified in question 1.2 (i.e. that node will be `N`).

Right rotation in code

Before we go ahead to implementing right rotation, let's first see the necessity of it in our program.

Open the starter code that we gave you. Your implementation work will be done in `avl.cpp`. Run the program by using the command: `make run_avl`.

When you run it, you will see this output:

```
Print format for each node ---> <data:height:balance>

10:0:0 20:1:1 30:2:2

Invalid AVL Tree, broken at 30
```

The middle line of the output is outputting an inorder traversal of the nodes in the tree and it is printing the data that is saved in the node, followed by its height, and then followed by its balance. In the last line it is explaining whether or not the tree is a valid AVL tree, and if it isn't it is telling you which is the node that it is broken in.

Since the balance at node 30 is +2, this is a left-leaning tree that needs a right rotation like the one explained above.

Question 1.4: Implement the `_rightRotate(Node*& N)` function in `avl.cpp`. To do it, we have broken down the steps that you will follow to complete the full rotation:

1. Assign pointers to nodes `L` and `B`
    a. You should note that we are passing `N` as a parameter to the function, thus, you already have a pointer to `N`. You will need to set the pointers to `L` and `B` as they are described above.
        i. As described above, `L` is the left child of `N` and `B` is the right child of `L`.
2. Perform the rotation
    a. This will *only* change the following two pointers:
        i. Change `N`'s left child to point to `B`

        ii. Change `L`'s right child to now point to `N`
    b. Note that we are not changing N's parent yet. That will be changed in step 4 below.
3. Update the heights of `N` and `L` (no other heights need to be updated)
    a. You should use the helper functions we've provided in `avl.h` to compute the new heights. Use the formula we gave in class for the height of a node.
    b. You should think about which of the heights you should now update first.
4. Update the pointer from `N`'s former parent so that it now points to `L`. Updating `N` will do the trick here, given that it is passed as a reference to a pointer.

NOTE: For testing your right rotate function, you should uncomment lines 36 to 38 in `avl.cpp`, so that your function is called when it needs to be called.

Make sure that after implementing your `_rightRotate` function, you run your program again (using `make run_avl`) and now your output should look like:

```
Print format for each node ---> <data:height:balance>


10:0:0 20:1:0 30:0:0


Valid AVL Tree
```

Task 2: Left rotation

Left rotation on paper

Starting off with `T`, insert a node with `data` value **100**.

Question 2.1: Present the drawing of the tree with the proper insertion on `T`, with the updated heights for all pertinent nodes.

Question 2.2: At which node is the balance property broken?

In this case, we have a right-leaning tree so we will perform a left rotation.

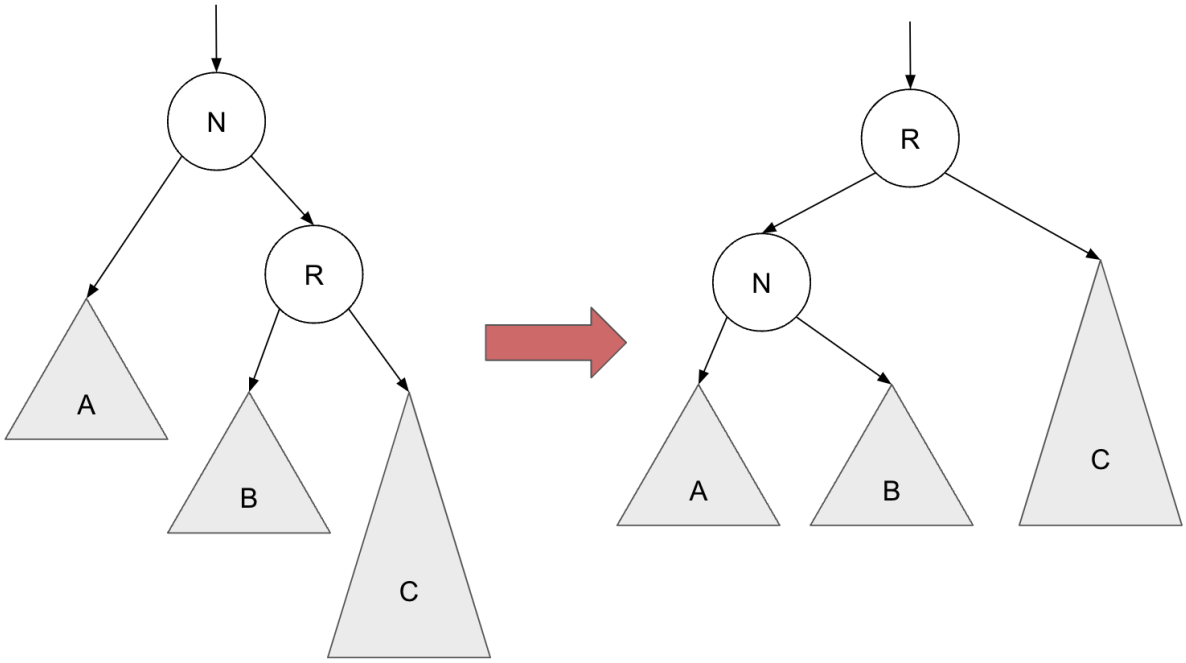
When doing a left rotation, there are three special nodes that we should identify/label:

1. N - the node at which the AVL tree broke.
2. R - the current right child of N
3. B - the current left child of R

Once we have identified these three special nodes, the rotation can be performed in three steps:

1. change N's right child to now be B
2. change R's left child to now be N
3. change N's parent node to now point to R instead of N

Here is a picture that illustrates this scheme:



Question 2.3: Draw the AVL tree that is produced after doing a right rotation on the tree you drew in question 2.1. The rotation is done on the "broken" node that you identified in question 2.2 (i.e. that node will be N).

Left rotation in code

Before we go ahead to implementing left rotation, let's first see the necessity of it in our program.

Uncomment the lines in the `main` method that are marked for uncommenting in part 2. These two lines will insert 40 and 50 to our AVL. Run the program by using the command: `make run_avl`.

When you run it, you will see this output:

```
Print format for each node ---> <data:height:balance>

10:0:0 20:3:-2 30:2:-2 40:1:-1 50:0:0

Invalid AVL Tree, broken at 30
```

We can see that the AVL tree is broken at node 30. Since the balance at node 30 is -2, this is a right-leaning tree that needs a left rotation like the one explained above.

Question 2.4: Implement the `_leftRotate(Node*& N)` function in `avl.cpp`. To do it, we have broken down the steps that you will follow to complete the full rotation which are analogous to those you completed in Task 1:

1. Assign pointers to nodes `R` and `B`
    a. As described above, `R` is the right child of `N` and `B` is the left child of `R`.
2. Perform the rotation
    a. This will *only* change the following two pointers:
        i. Change `N`'s right child to point to `B`
        ii. Change `R`'s left child to now point to `N`
3. Update the heights of `N` and `L` (no other heights need to be updated)
4. Update the pointer from `N`'s former parent so that it now points to `R`.

NOTE: For testing your left rotate function, you should uncomment lines 41 to 43 in `avl.cpp`, so that your function is called when it needs to be called.

Make sure that after implementing your `_leftRotate` function, you run your program again (using `make run_avl`) and now your output should look like:

```
Print format for each node ---> <data:height:balance>

10:0:0 20:2:-1 30:0:0 40:1:0 50:0:0

Valid AVL Tree
```

Task 3: RL Rotation (optional)

NOTE: This task (and the next) are marked as optional. If you finish the first two tasks early you can continue on to these tasks, in order to get some practice on doing these rotations which we will cover in our next lecture.

So far, we have covered two of the possible cases of insertions to an AVL tree, which trigger rotations. Now we take a look at a couple of cases that will necessitate two rotations instead of just one.

Starting off with `T`, insert a node with `data` value **87**.

Question 3.1: Present the drawing of the tree with the proper insertion on `T`, with the updated heights for all pertinent nodes.

Question 3.2: At which node is the balance property broken?

Since the balance at the node which is broken is negative, this would make us think that this is a right-leaning tree like the one we saw in Task 2. Let's try to do a left rotation, like we did then.

Question 3.3: Draw the tree that is produced after doing a left rotation on the tree you drew in question 3.1. The rotation is done on the "broken" node that you identified in question 3.2 (i.e. that node will be `N`).

This tree that you just drew is not an AVL tree. ***Why?***

In this case, we have to perform two rotations.

This case is one in which the balance at the broken node is negative, but the data value we are trying to insert is less than the data value stored in the right child of the broken node.

For this case, the two rotations we need to perform to produce an AVL tree are:

1. A right rotation at the right child of the broken node.
2. A left rotation on the broken node.

Question 3.4: Draw the tree that is produced after doing a right rotation at the right child of the broken node you identified in question 3.2, from the tree you produced in question 3.1.

Question 3.5: Draw the tree that is produced after doing a left rotation on the broken node, from the tree you drew in question 3.4.

Now we should have a valid AVL tree with the insertion of 87.

Task 4: LR Rotation (optional)

Starting off with `T`, insert a node with `data` value **18**.

Question 4.1: Present the drawing of the tree with the proper insertion on `T`, with the updated heights for all pertinent nodes.

Question 4.2: At which node is the balance property broken?

Since the balance at the node which is broken is positive, this would make us think that this is a left-leaning tree like the one we saw in Task 1. However, similarly to task 3, this will not produce an AVL tree.

This case is one in which the balance at the broken node is positive, but the data value we are trying to insert is greater than the data value stored in the left child of the broken node.

For this case, the two rotations we need to perform to produce an AVL tree are:

1. A left rotation at the left child of the broken node.
2. A right rotation on the broken node.

Question 4.3: Draw the tree that is produced after doing a left rotation at the left child of the broken node you identified in question 4.2, from the tree you produced in question 4.1.

Question 4.4: Draw the tree that is produced after doing a right rotation on the broken node, from the tree you drew in question 4.3.

Now we should have a valid AVL tree with the insertion of 18.

These two tasks can be tested and checked in `avl.cpp` by uncommenting the corresponding sections in the `main` function and in the `_insert` function, in the same way we did for the first two tasks.

Deliverables

Diagrams for questions 1.1 (with labeled broken node), 1.3, 2.1 (with labeled broken node), and 2.3.

Implementation for `_leftRotate` and `_rightRotate` functions in `avl.cpp`.

Acknowledgements

Content by Daniel Ayala with help from materials by Shanon Reckinger and Joe Hummel.