Lab 7
AVL Tree

Quang D. C.
dungcamquang@tdtu.edu.vn

November 15, 2021

Note

In the previous lab, we learned how to build up a BST. And in this tutorial, we continue to extend Binary Search Tree to the Balanced Binary Search Tree, one of them called AVL Tree.

Part I
Classwork

In this part, lecturer will:

- Summarize the theory related to this lab.
- Instruct the lesson in this lab to the students.
- Explain the sample implementations.

Responsibility of the students in this part:

- Students practice sample exercises with solutions.
- During these part, students may ask any question that they don't understand or make mistakes. Lecturers can guide students, or do general guidance for the whole class if the errors are common.

1. What is AVL Tree?

AVL is the abbreviation of the Russian authors Adelson-Velskii and Landis (1962) who defined the balanced tree.

An AVL tree is the same as a binary search tree, except that for every node, the height of the left and right subtrees can differ only by 1 (and an empty tree has a height of -1)[1]. This difference is called Balance Factor. When the Balance Factor > 1, the tree will be rebalanced.

[1] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1176/lectures/20-BinarySearchTrees/20-BinarySearchTrees.pptx

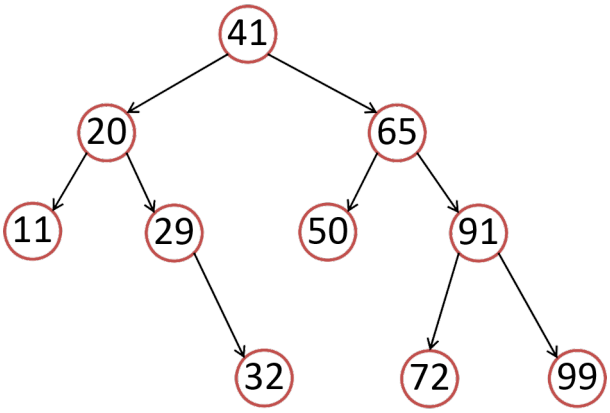

Figure 1: AVL tree

2. Node of a AVL tree

In this lab, we add one more attribute which is called *height* to the Node class (same with BST). This attribute will help us access the height of the node faster.

```
public class Node{
    Integer key;
    Node left,right;
    int height;

    public Node(Integer key){
        this.key = key;
        this.height = 0;
        this.left = this.right = null;
    }
}
```

In AVL class:

```
public int height(Node node){
    if (node == null)
        return -1;
    return node.height;
}
```

3. Check balance factor

A Balance Factor of a AVL tree is defined by the height difference of the left subtree and the right subtree.

```
private int checkBalance(Node x) {
    return height(x.left) - height(x.right);
}
```

4. Rotation

The insert or delete operation may make the AVL tree come to imbalance. A simple modification of the tree, called **rotation**, can restore the AVL property. We have 4 types

of rotation corresponding to 4 violation cases that make the tree imbalance.

4.1. Left rotation

When a new node is inserted into a AVL tree and makes it a right-right-unbalanced-tree. The tree can be re-balanced using left rotation as following:

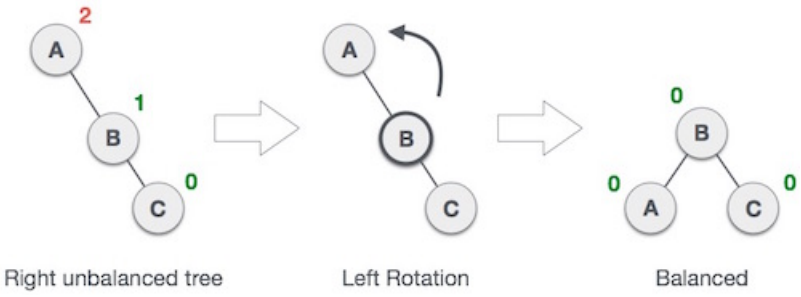

Figure 2: Left rotation

This is the code of left rotation:

```
private Node rotateLeft(Node x) {
    Node y = x.right;
    x.right = y.left;
    y.left = x;
    x.height = 1 + Math.max(height(x.left), height(x.right));
    y.height = 1 + Math.max(height(y.left), height(y.right));
    return y;
}
```

4.2. Right rotation

When a new node is inserted into a AVL tree and make it a left-left-unbalanced-tree. The tree can be re-balanced using right rotation as following:

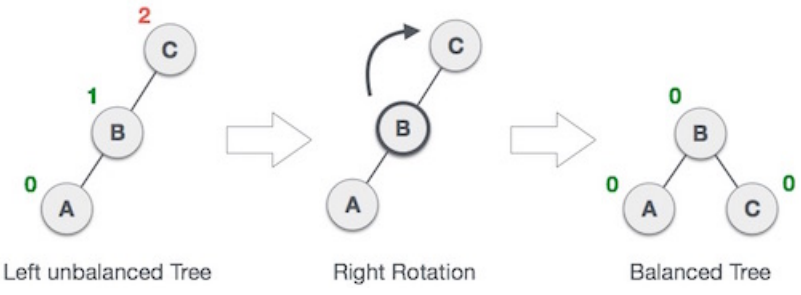

Figure 3: Right rotation

```
private Node rotateRight(Node x) {
    //your turn
}
```

4.3. Left-Right rotation

When a new node is inserted into a AVL tree and make it a left-right-unbalanced-tree. The tree can be re-balanced using left-right rotation.

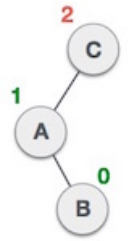

Figure 4: Left-Right rotation

First, we perform a **left rotation** on node A (the left subtree of C).

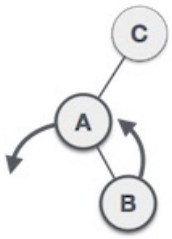

Figure 5: Left-Right rotation

This makes A come to the left subtree of B. And B replaces A to become the left subtree of C.

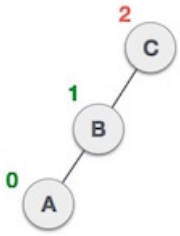

Figure 6: Left-Right rotation

Now, node C is remaining unbalanced but it has become case left-left-unbalanced-tree and **right rotation** can be used to balance the tree.

4.4. Right-Left rotation

When a new node is inserted into a AVL tree and make it a right-left-unbalanced-tree. The tree can be re-balanced using right-left rotation.

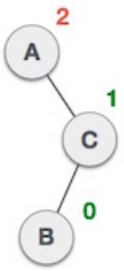

Figure 7: Right-Left rotation

First, we perform a **right rotation** on node C (the left subtree of A).

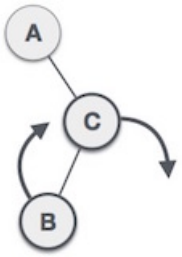

Figure 8: Right-Left rotation

This makes C come to the right subtree of B. And B replaces C to become the right subtree of A.

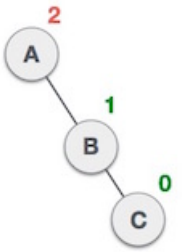

Figure 9: Right-Left rotation

Now, node A is remaining unbalanced but it has become case right-right-unbalanced-tree and **left rotation** can be used to balance the tree.

5. Balance

This is the code to re-balance the tree:

```
private Node balance(Node x) {
    if (checkBalance(x) < -1) {
        if (checkBalance(x.right) > 0) {
            x.right = rotateRight(x.right);
        }
        x = rotateLeft(x);
    }
```

```
    else if (checkBalance(x) > 1) {
        if (checkBalance(x.left) < 0) {
            x.left = rotateLeft(x.left);
        }
        x = rotateRight(x);
    }
    return x;
}
```

Part II
Excercise

Responsibility of the students in this part:

- Complete all the exercises with the knowledge from **Part I**.
- Ask your lecturer if you have any question.
- Submit your solutions according to your lecturer requirement.

Exercise 1

Complete the class to build the AVL tree. You can re-use the BST code in the previous lab (insertion, deletion).

THE END