Tutorial #10. AVL trees

Theoretical part

AVL trees are strictly balanced trees, that means it **always** stay in the state |h(left)-h(right)| <= 1. This leads us to **O(log n)** for read. There are also approaches balanced on probability, but they can promise only T(log n) for average case.

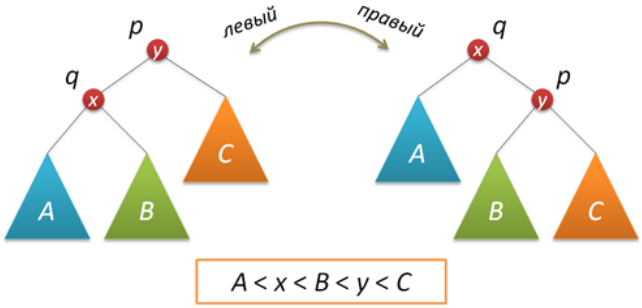
Search

AVL tree algorithms is a combination of plain binary tree and balancing algorithms. This means ***Search (find)*** operation will remain the same.

Rotations

Restructuring approaches for AVL trees are also called **rotations**. To perform rotation on AVL tree, there should be special conditions – tree should be in unbalanced state. This condition depends on **balance factor = h(left) – h(right)**. It should be {-1, 0, 1} for each node. If it is {-2, 2} for any node – tree starting from this node (and up to root) should be restructured to reduce this difference. To have this value you can either store height of subtree or balance factor itself in each node.

The simplest way to balance subtree is to do **simple rotation**.



But this will work only
(for right): ***h(q) = h(C) + 2*** *&&* ***h(B) <= h(A).***
(for left): ***h(p) = h(A) + 2*** *&&* ***h(B) <= h(C).***

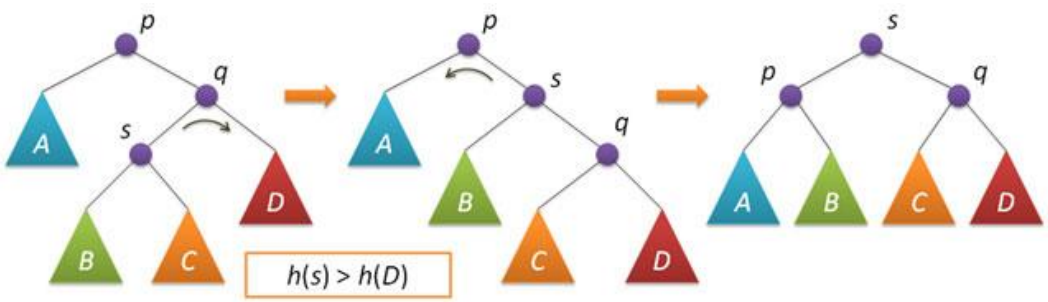
In other words “middle” subtree (B) should not be “higher” that side subtree (A for right rotation) of the same node before rotation. Else, it will again bring subtree in unbalanced state.

What should we do in other case?

Let’s do more complex things – **big rotation**. If “**middle**” subtree is “**higher**” than side subtree, we can firstly fix it!

For left rotation it will be as follows:

Rotate (**q**) subtree in *opposite direction* to make **left branch of it subtree shorter (or equal) than right branch**. Then the condition for simple rotation will become true.



Insert

Now, to insert the element to the tree, you have to

1) Follow BST rules
2) Rebalance all path from inserted element to the root node.

```
public static <T> Node<T> insert(Node<T> root, Node<T> newNode) {
	if (root == null) return newNode;
	if (newNode.key < root.key)
		root.left = insert(root.left, newNode);
	else
		root.right = insert(root.right, newNode);
	return root.balance();
}
```

Delete

To delete the element, you have to:

1) Follow BST rules
2) Rebalance starting from “replacing node” (that was inserted instead of deleted node) to the root node.

```
Node<T> remove(int k) {
	if (k < key) left = left.remove(k);
	else if (k > key) right = remove(k);
	else {  // if (k == p->key)
		Node<T> q = left;
		Node<T> r = right;
		if (r == null) return q;
		Node<T> min = r.findMin();
		min.right = r.removeMin();
		min.left = q;
		return min.balance();
	}
	return balance();
}
```

Practical part

Having BST implemented:

1) Implement rotation methods in your Linked Tree classes. Carefully add balancing operation to insert() and delete() implementations. Test your tree on with the following actions:
    a. **insertion sequence**: *March, May, November, August, April, January, December, July, February, June, October, September*.
    b. **deletion sequence**: April, August, December.
    c. **find sequence**: January-December (all months present except deleted).
2) (*) Measure algorithm complexity properties.
    a. Insert 1000000 (1M) values from 0 to 999999 into the tree.
    b. For each insertion, measure **operation time** and **tree height**.
    c. Delete elements from 0 to 999999.
    d. For each deletion, measure **operation time** and **tree height**.
    e. Build 2 graphs that show operation time and tree height depending on number of elements in a tree. Do they fit O(log n) estimation?