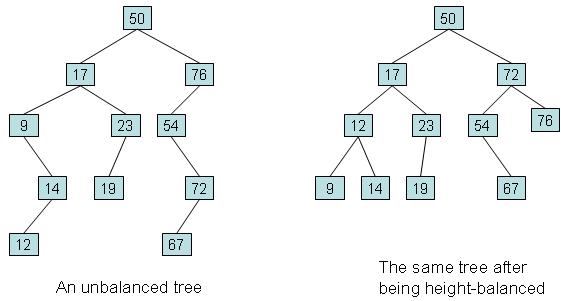
HW7: Implement Tree Rotations and Self-updated Height

The goal of this homework is to implement the base methods that will be used to implement self-balancing binary search trees, in particular during the midterm.

|  |  |  |
| --- | --- | --- |
| As we have seen in the last lecture, binary search trees organize data by separating the keys into those that are *smaller* than the root (on the left) and *larger* than the root (on the right). Some trees are *worst-case trees* (where none of the nodes have two children - for instance when the tree is just a straight line, or a list) and *best-case trees* (where all the levels are filled except the last one).  The “balancedness” of a tree depends on the order in which keys are inserted into a tree. For a same set of keys, depending on the order in which they are inserted into the binary tree, the tree can end up being well-balanced (best case) or extremely unbalanced.  In particular, inserting keys in a sorted (decreasing or increasing order) produces worst-case trees. |  |  |

It is worth asking ourselves: *suppose the elements were inserted in a particularly bad order, and that the tree is poorly balanced -****is there anything that can be done to rebalance the* tree?**

Rebalancing makes the height of the left subtree and right subtree of all nodes in a tree be different of no more than 1. (The height of a binary tree, deepest you can go: for instance the height of the tree on the left below is 5 because: 50 is at height 1, 17 is at height 2, 9 is at height 3, 14 is at height 4, 12 is at height 5.)



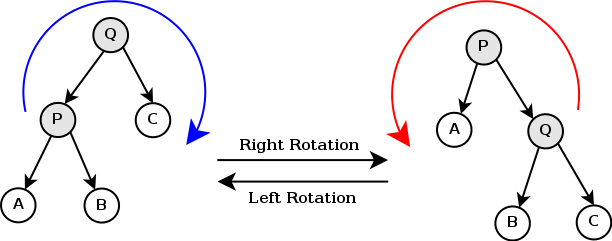
The answer is of course: yes, there is. Trees can be rebalanced quite easily, but because binary search trees have conditions on the way the keys are ordered (smaller on the left; larger on the right), this condition has to be maintained when elements are moved around. The operations which move data around while preserving these conditions are called **tree rotations**.

These operations are interested in shifting around the height of binary trees

There are two tree rotations:

* one **right rotation**, which is appropriate when the left subtree is deeper than the right subtree (and it shifts one depth to the right);
* the other, **left rotation**, is appropriate in the opposite case.

Both operations are illustrated in the following diagram: P and Q are nodes, and A, B and C are their *subtrees* (they are not nodes, but subtrees - which can be any of the following: empty, only one node, or big large trees, etc.).



Your goal is to take the BinaryTreeSkeleton project, and extend the BinaryTree class in the following way:

* in the **BinaryTree** class, extend the protected class BinNode by adding the field height

**protected** **class** BinNode {

  KeyType key;

  ValueType value;

  BinNode left, right;

**int**height;

**public** BinNode(KeyType k, ValueType v) { key = k; value = v; height = 1; }

}

* in the **BinarySearchTree** class, modify the insert operation so that it maintains the new **height** field with the height of the subtree (4 points)
* in the **BinaryTree** class add the two following operations, which take a tree *x*, and return the left rotated tree or right rotated tree (6 points)
  + protected BinNode rotateLeft(BinNode x);
  + protected BinNode rotateRight(BinNode x);
* Because these methods manipulate the protected class BinNode, they cannot be made public. *SO FOR TEST PURPOSES ONLY (these methods are not useful)*, also implement public void rotateLeftRoot(), and public void rotateRightRoot(), which apply a rotateLeft to the root of the tree, or a rotateRight. Then call these functions from **Main.java** to do some limited testing that your methods work: try rotating the tree left a few times, then right, then left, and see that it still is the same tree (5 points).

Export the entire project from Eclipse and submit the archive file. Include the names of everybody that has worked on the assignment in the Main.java.