We have learned that a binary search tree can store a list so that an item can be found in \(O(\log N)\) time (where \(N\) is the number of items in the tree), but only if the tree is balanced. We studied one structure that guarantees balance, the red black tree. In this assignment, you will implement and test an alternative structure that maintains balance, the Treap\(^1\).

The treap structure combines the properties of a binary search tree and a minheap. Each node contains both data (same as the binary search tree) and a priority, which is randomly chosen when the node is created. The Treap is both a binary search tree (looking at the data) and a minheap (looking at the priorities). In the example below, the data are A, F, R, T, V, and X and the priorities are 5, 10, 32, 17, 25 and 50.

```
    T:5
     /  \
    F:10  U:32
   /    /    \
  A:17  R:25  X:50
```

To search a treap for a given value, use the usual binary search tree algorithm on the data. The priorities are ignored.

To add an item to a treap, create a node consisting of the item and a random priority value. First insert the item into the binary search tree according to the data. Then, check the path from the new node back up to the root, recursively moving items with lower priority up the tree. You cannot move items by swapping, because that will destroy the binary search tree property. On the other hand, left and right rotations move nodes up the tree without disturbing the binary search tree order. Therefore, you can move a left child up by doing a right rotation on its parent, and you can move a right child up by doing a left rotation on its parent. In the following example, \(V\) with priority 21 is added to the treap above.

First, \(V\) is added to the tree in the usual manner. It ends up left of \(X\). Next, we note that \(V\)’s priority (21) is smaller than its parent (50). To fix this, we rotate \(X\) to the right.

---

Now V has moved up one level in the tree, but its priority is still smaller than its parent (U:32). Since V is a right child, we need to rotate U to the left.

At this point, our heap property is restored. V (and everything below V) is now larger than T. Our tree has also improved in balance, due to the priority value causing the rotations.

Your assignment is to create and test a Treap class that includes the following public methods:

- Treap() – constructs an empty tree
- E find(E item) – find an item in the treap (same as Binary Search Tree Find)
- boolean add (E item) – add an item to the treap, using the method explained above. Use double values for your random numbers (getDouble() method from the Random class) to minimize the possibility of generating identical priorities for different nodes.
- String toString() – preorder printing with indentation, same as in Binary Tree
- int height() – return the height of the tree
- int totalPath() – return the total path length of the tree. This is the sum of the depth of every node in the tree. For example, the final treap above has totalPath(1+2+2+3+3+3+3) = 17. Before U was inserted, the totalPath was (1+2+2+3+3+3) = 15.

You are encouraged to reuse code from the book’s BinaryTree, BinarySearchTree, and BinarySearchTreeWithRotation classes as needed (with proper attribution). For your convenience, these classes are included on the Sakai assignment page. Testing may be done using JUnit, a main class, or a combination of methods.

Once you have working versions of these classes, you should perform (at least) the following experiments.
1. Generate 1000 random numbers and add them to a BinarySearchTree. Record the height and totalPath of the tree. You will have to modify the BinarySearchTree class to add these methods. Add the same 1000 random numbers to a Treap, and record its height and totalPath. Repeat the experiment for 4 more sets of 1000 random numbers (a total of 5 BinarySearchTrees and 5 Treaps). Explain your results.

2. Instead of using random numbers, insert the numbers 0 through 999 into a BinarySearchTree in order and record the height and totalPath. Do the same for a Treap 5 times (so you will have 1 BinarySearchTree and 5 Treaps). Record the totalPath values from all trees and explain your results.

Submit all code (including test classes), clearly commented, including clear indications of which code is original and which borrowed from the textbook. You may not use any other code sources. Also submit a writeup that explains the design of your Treap class, its testing, and the results of your experiments.