CS 314H Algorithms and Data Structures — Fall 2012	Programming Assignment #6
Treaps	Due November 11/14/16, 2012

In this assignment you will work in pairs to implement a map (associative lookup) using a data structure called a treap, which is a combination of a tree and a heap. Your key challenge in this assignment will be to carefully and thoroughly test your data structure, so you will also be asked to design a testing program for your code. For this assignment, you should not discuss testing strategies with other teams.

You will again be writing Peer Reviews, with the following schedule.

Program and Report Due	5:00pm Sunday November 11
Peer Reviews Due	5:00pm Wednesday November 14
Testing Report and Test Code Due	5:00pm Friday November 16

1 Treaps

A *treap* is a binary search tree that uses randomization to produce balanced trees. In addition to holding a key-value pair (a map entry), each node of a treap holds a randomly chosen priority value, such that the priority values satisfy the *heap property*: Each node other than the root has a priority that is at least as large as its parent's priority. An example treap is shown in Figure 1, where the keys are shown at the top of each node and the priorities are shown at the bottom of each node. Notice that the keys obey the binary search tree (BST) property and the priorities obey the heap property. Because the keys obey the BST property, a lookup operation can be performed just as with any BST. However, the insert and remove operations are more complex.

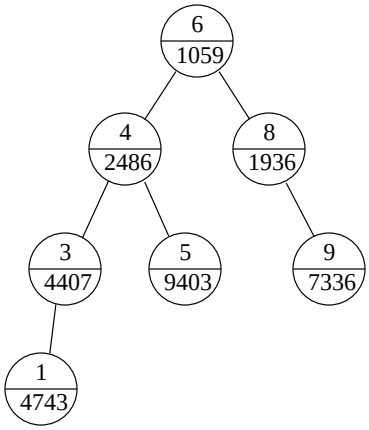

Figure 1: A treap for a map with key set $\{1, 3, 4, 5, 6, 8, 9\}$. For each node, the key is shown in the top half, while the priority is shown in the bottom half. The priority values are chosen at random, with smaller numbers indicating higher priority.

To insert a new node x with key k, we first perform the insertion at the appropriate leaf position according to the BST property, exactly as in a binary search tree (See Figure 2). The node is assigned a randomly chosen priority p, and because x's parent y may have priority greater than p, the heap property may be violated. To restore the heap property, we perform a rotation, making x the parent of y, as shown in Figure 2(b). Specifically, if x is the left child of y, then we rotate right around y, and if x is the right child of y, then we rotate left around y. Node x now has a new parent, and the heap property may still be violated, requiring another rotation. In general, the heap property is restored by rotating the new node x up the treap as long as it has a parent with higher priority. Figure 2 shows an insertion requiring 2 rotations.

To remove a node x, we "reverse" the insertion. We rotate x down the treap until it becomes a leaf, and then we simply clip it off. At each step, the decision to rotate left or right is governed by the relative priority of the children.

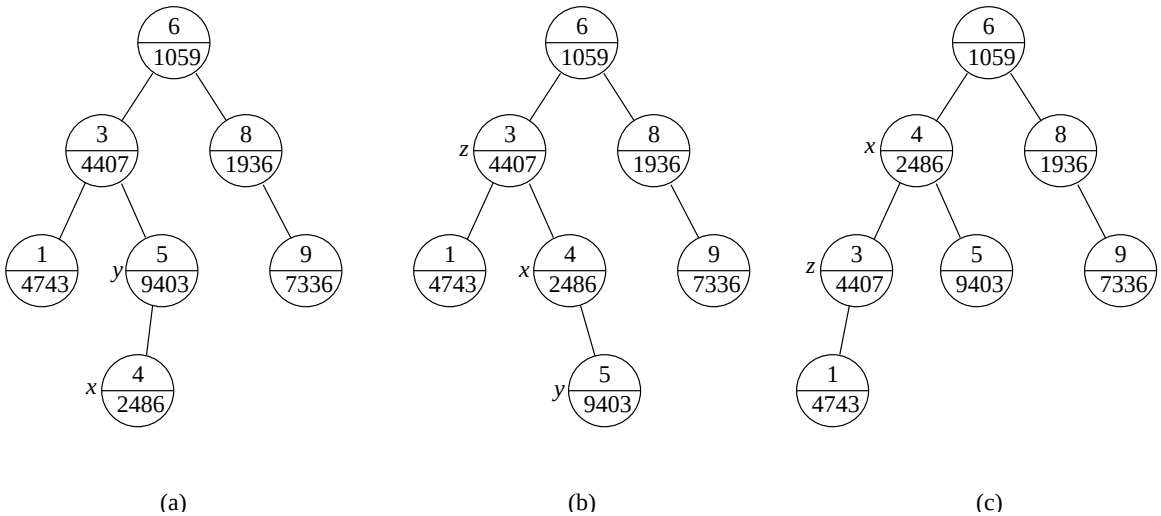

Figure 2: Inserting new node *x* into a treap. (a) The new node *x*, with key *k=4* and priority *p=2486*, is added as a leaf according to the BST property. The heap property with respect to *x*'s parent *y* is violated. (b) The situation after a right rotation around *y*; the heap property with respect to *x*'s new parent *z* is violated. (c) After a left rotation around *z*, the heap property is restored.

The child with the higher priority should become the new parent. Thus, if *x*'s left child has higher priority than *x*'s right child, then we rotate right around *x*. Conversely, if *x*'s right child has smaller priority than *x*'s left child, then we rotate left around *x*. Figure 3 illustrates a removal requiring 2 rotations. This removal reverses the insertion of Figure 2.

All three map operations—lookup, insert, and remove—run in time $O(h)$, where *h* is the height of the treap. It is not hard to show that a treap with *n* nodes has expected height $\Theta(\log n)$. Note that the root of a treap is determined by the randomly chosen priorities. The node with the highest priority (smallest key value) is the root. Thus, the root node is equally likely to contain any of the map entries, regardless of the order in which the entries are inserted or removed. Consequently, we expect that half of the entries will be in the left subtreap and the other half in the right subtreap. The analysis of treap height is therefore similar to the analysis of recursion depth in quicksort.

2 Your Assignment

Implement a map using a treap. In particular, you should implement the following interface. Your treap should store entries with keys that are `Comparable` objects and values that are `Objects`. The `lookup(k)` method should return null if no entry with key *k* is in the map.

```
public interface Treap {
    Object lookup(Comparable key);
    void insert(Comparable key, Object value);
    void insert(Comparable key, Object value, int priority);
    Object remove(Comparable key);
    Treap [] split(Comparable key);
    void join(Treap t);
    void printTreap(PrintWriter o);
    void resetIterator();
    Comparable nextKey();
    double balanceFactor() throws OperationNotSupportedException;
    void meld(Treap t) throws OperationNotSupportedException;
    void difference(Treap t) throws OperationNotSupportedException;
}
```

A more detailed description of the interface is in `Treap.java`. Implement your treap-based map in `TreapMap.java`

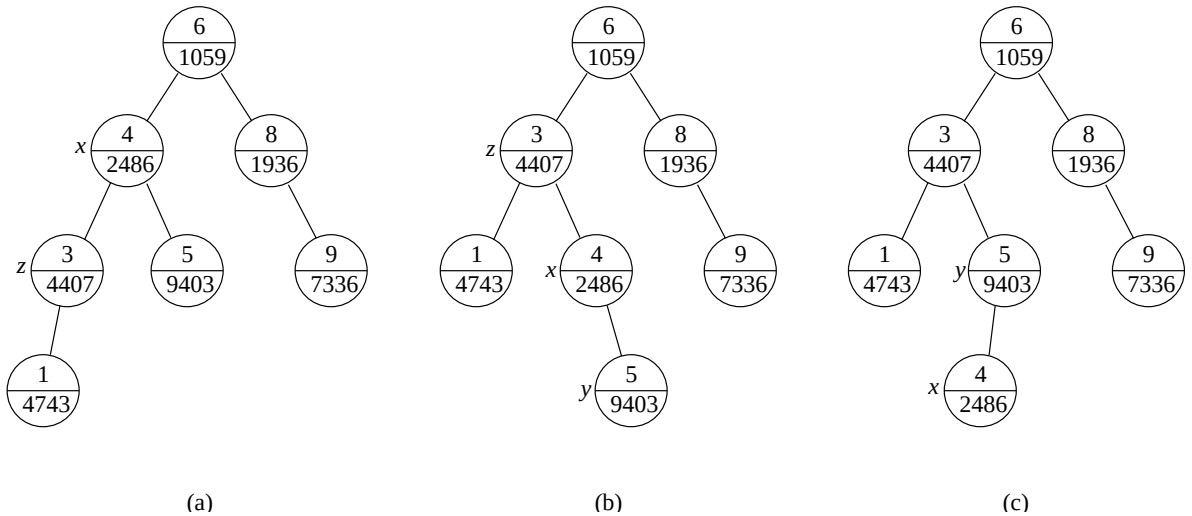

Figure 3: Removing a node *x* from a treap. (a) Node *x* has two children, of which the left child *z* has smaller priority. (b) After a right rotation around *x*, node *x* now has only one child, *y*. (c) After a left rotation around *x*, node *x* is now a leaf and can be clipped off like an excessively long toenail.

The insert method. Insertion into the treap should be implemented as outlined earlier.

The remove method. Removal from the treap should be implemented as outlined earlier.

The split method. A treap *T* can be *split*, using a key *k*, to produce two treaps, T_1 and T_2, such that T_1 contains all of the entries in T with key less than k, and T_2 contains all of the entries in T with key greater than or equal to k. To perform the split, we insert into T a new entry x with key k and priority $p = 0$, forming a new treap T'. (We assume that 0 is the smallest possible priority value.) Because x has the smallest possible priority, x is the root of T', so the split has been accomplished with T_1 being the left subtreap and T_2 being the right subtreap. You should not "lose" any value associated with k if k is already in the treap, although it is ok if you destroy the old treap.

The join method. The inverse of a split is *join*, in which two treaps, T_1 and T_2, with all keys in T_1 being smaller than all keys in T_2, are merged to form a new treap T. To perform the join, we create a new treap T' with an arbitrary new root node x and with T_1 and T_2 as the left and right subtreaps. We then remove x from T' to form the joined result T.

Split and join both take time $O(h)$, where h is the height of the T (the input to split or the result of join). The expected height is $\Theta(\log n)$, where n is the size of T, so split and join both run in $O(\log n)$ expected time. More interestingly, split and join can be used as subroutines to $meld$ two treaps or take the $difference$ between two treaps. You may implement these for additional karma.

3 Testing

Since the treap in this assignment is not part of a larger application, you will not be able to use or test your treap without writing your own test program. Write a program (in `TreapTest.java`) to test your treap for correctness.

Your test program should work with any implementation of the `Treap` interface. It should not find any errors in a correct implementation. For errors that it does find, it should produce output that would be useful to a human. You are not required to test the portions that you do not implement yourself, but you should test everything that you do implement.

4 Karma

Three of the operations in the interface (`balanceFactor()`, `meld()` and `difference()`) are optional. Implement them for extra karma. If you do not implement them, throw an `OperationNotSupportedException`

4.1 Meld

A *meld* takes two treaps, T_1 and T_2 and merges them into a new treap T, much like the Vulcan mind meld for which it is named[1]. Unlike a join, a meld does not require any relationship between the keys in T_1 and T_2. Meld is a naturally recursive procedure and should be able to meld two treaps of size n and m ($m \leq n$) in $O(m \log(n/m))$ time. Describe how you meld treaps and how your algorithm meets the specified asymptotic time bound.

4.2 Difference

The *difference* between two treaps, T_1 and T_2, is a treap T containing the keys of T_1 with any keys in T_2 removed. The difference can also be computed recursively and also runs in $O(m \log(n/m))$ time. Describe how you take a difference and how your algorithm satisfies this time bound.

4.3 Diagnosing Problems Through Testing

Typically, the goal of a test program is to identify bugs. With some additional work, you can attempt to diagnose common problems by using the observed behavior of the program. For example, if the iterator misses one key, it is likely that the missing key is the first or last key added. A test program can attempt to verify this hypothesis and provide a suggestion to the user. Can you use your test program to provide assistance in finding common mistakes?

4.4 Balance statistics

It would be useful to know how balanced or imbalanced your treap is. The balance factor is the ratio between the height of the treap and the minimum possible height. A perfectly balanced treap will have a balance factor of 1.0. Include observations on how well the treap seems to keep itself balanced in your report.

5 What to turn in

Except for the specific names of the files, the directions for what to turn in are the same as for Assignment 5, but we repeat them below for your convenience.

5.1 Deadline 1

Your code and assignment report are due, as normal. If you turn in your code late, your reviewers will have the choice of whether or not to review your code, so it is in your best interest to not take any late days for this assignment. Regardless of whether you have submitted your assignment yet or not, on Sunday night, you will be assigned several projects to review.

Turn in `TreapMap.java` and `TreapTest.java` and any other files necessary for your implementation. Be sure to comment any non-obvious portions of your assignment. **Please do not use packages.** When you use a package, your Peer Reviewer, who only has access to your bytecode, cannot easily run your code.

Make sure that your code compiles and that all classes have the correct names and are in the correct files. If you do not submit your solution on time, you will lose points for being late, and your reviewers get to choose whether or not they will review your solution. Since the reviews can help you test your program to improve your final submission, it's in your best interest to submit your solution on time.

5.2 Deadline 2

Your peer reviews are due. If you turn these in late, your grade will be penalized.

[1] Not really.

5.3 Deadline 3

Your revised code, testing report, and reviewer grades are due. You have the option to fix your code based on the feedback you received, if you have time. The testing report should cover all insight into the review process, what you did, how you did it, what you learned, what you learned from the reviews of your own code, what this revealed about your code, and what changes you made/would make given more time.

Acknowledgments. We thank Bobby Blumofe, now at Akamai, for the original version of this assignment, and Walter Chang for his subsequent modifications.