

## Assignment # 6

### ***Purpose***

The purpose of this assignment was to implement and test the `TreapMap` class, which is a binary search tree that uses randomization to maintain a somewhat balanced tree regardless of the input. The testing process should rigorously test the code in order to find faults.

### ***Implementation***

#### **The Treap Representation**

The `TreapMap` is represented as a tree (ha!) of `TreapNodes`. Each `TreapNode` contains a priority, a key, a value, and references to its parent and two children. Although heaps are easily represented as arrays, the treap's binary search tree property makes a node-based structure more appealing.

#### **The Lookup Method**

The `lookup()` method finds the value associated with a certain key using a standard binary search algorithm. Starting at the root, if the key value is less than the root, go to the root's left child. Otherwise, go to the root's right child. Repeat this process using the current node in place of the root until either the key is found or a null node is reached.

#### **The Insert Method**

The insertion method places a `TreapNode` with a certain key and priority into the `TreapMap` while preserving the `Treap` properties. This is accomplished by first placing the `TreapNode` as a leaf in a location that preserves the binary search tree property. Then, the `TreapNode` is rotated upwards until the heap property is met. There is one special case of the rotation that becomes important when the `split()` method is used.

When rotating the `TreapNode` upwards, there is a possibility that its priority is equal to its parent's priority. There are two ways of dealing with this situation, both of which produce a valid `Treap`: do nothing or rotate it so that the inserted `TreapNode` is now higher in the `Treap`. In the `split()` method, a `TreapNode` with a priority of 0 is inserted in the `TreapMap`. It is important that this `TreapNode` becomes the root. If another node already has a priority of 0 and the "do nothing" option is chosen, the inserted `TreapNode` will become one of the root's children and the `split()` method fails to work. If, however, an inserted `TreapNode` is placed so that its priority is strictly greater than its parent, instead of being greater-than or equal to, it is guaranteed to become the root. Note that another way of solving this is just to make it so that, outside of the `split()` method, no node can ever have a priority of 0. The implementation

assumes that 0 is the smallest priority value, so I have decided to err on the side of caution. Either way, the probability of this actually having an effect is quite small because the possible priority values are uniformly distributed between 0 and the maximum int value.

### The Remove Method

There was nothing tricky encountered in implementing the `remove()`, method, so it is in the exact form described in the handout (rotate the node that is supposed to be removed down until it becomes a leaf and then set the reference its parent has to it null).

### The Split Method

The `split()` method splits the `TreapMap` into two `TreapMaps`, one for values less than a certain key and one for values greater than a certain key. The implementation provided is to insert a `TreapNode` with the key value and a priority of 0 into the `TreapMap`, causing the root's left child to be the root of the low `TreapMap` and the root's right child to be the root of the right `TreapMap`. With just this implementation used, the `split()` method runs in  $O(h)$ , where  $h$  is the height of the tree.

This implementation has consequences that force the `split()` method to be destructive. By simply creating two new `TreapMaps` and using as their roots the current `TreapMap`'s root's children, many nodes will be shared across the multiple `TreapMaps`. This means that inserting a `TreapNode` into the original `TreapMap` will have an effect on one of the `TreapMaps` resulting from the split, which violates every expectation as to the behavior of the `TreapMap`. To fix this, the `TreapMap` on which the split operation is performed has its root nullified, removing any overlapping of `TreapNodes`.

### The Iterator

The iterator allows for a sequential traversal of the `TreapMap`'s nodes based on the key values. This functionality is completely provided by two methods: `resetIterator()` and `nextKey()`. The interaction between the iterator and the `insert()` / `remove()` methods was specified as implementation-dependent. I decided to allow nodes to be inserted and removed at any point in the iteration without interfering with the iterator's progress. This means that when `nextKey()` is called, the key immediately following the key returned by the previous `nextKey()` call is returned, regardless of whether it was inserted after the iteration process began.

To handle this, I made use of two `TreapNode` objects: `iterator` and `lastIterator`. `iterator` references the `TreapNode` whose key would be returned if `nextKey()` were to be called and `lastIterator` references the `TreapNode` whose key was returned from the previous `nextKey()` call. Thus, the process for retrieving the next key is simple: `make lastIterator`

reference `iterator`, set `iterator` to be the `TreapNode` with the next key value, and return `lastIterator`'s key value.

Finding the next node for `iterator` to reference is essentially a single step in an inorder traversal of the tree. If `iterator`'s right child is not null, the next node is the node with the lowest value in the binary search tree with `iterator`'s right child as the root (which is handled by traversing to `iterator`'s right child once and then traversing the left children until the leftmost node has been reached). If `iterator`'s right child is null, the next node is the `iterator`'s parent. If `iterator`'s parent is null (i.e. `iterator` references the `TreapMap`'s root) and `iterator`'s right child is either null or has already been traversed, every node has been traversed and `iterator` is set to null.

This process assumes that `iterator` starts off referencing the correct node. This is not the case when `nextKey()` is first being called or `nextKey()` is first being called after a call to `resetIterator()`. If this is the case, `iterator` must first be set to the smallest value in the `TreapMap` (the leftmost node) and then the process continues normally.

This iterator implementation allows for an elegant handling of the `remove()` and `insert()` methods. If the `TreapNode` referenced by `iterator` is removed through the `remove()` method, a call to `nextKey()` is made while preserving the `lastIterator` reference. If a `TreapNode` with a key value between `lastIterator`'s key value and `iterator`'s key value is inserted through the `insert()` method, `iterator` is then made to reference the new `TreapNode`. This is all that is needed to ensure that the iterator works with `TreapMap` manipulation.

There is one more case that appears as if it would need to be handled separately but actually does not. The implementation for the `insert()` method requires that the `TreapMap` only have one node for each distinct key. When a node with a key already found in the `TreapMap` is inserted, the old node is replaced. The special case arises when `iterator` references the old node. After the insertion, it should reference the new node. Before the new node is added, the old node is removed which progresses `iterator` to the next node in the `TreapMap`. Then the new node is inserted. Since the new node's key value is necessarily between `lastIterator`'s key value and `iterator`'s key value, `iterator` is made to reference this key value and the iteration works properly.

## The PrintTreap Method

The `printTreap()` method prints the `Treap`. It accomplishes this by performing a preorder traversal of the `Treap`, printing the current element and then printing its children.

## Karma

### Balance Statistics

The `balanceFactor()` method provides the ratio of the current `Treap` height over the optimum (minimum) `Treap` height with the same number of elements. In this case, the height of a `Treap` refers to the maximum number of edges needed to be crossed to get from the root to a leaf. Thus, a `Treap` with only a root has a height of 0 and an optimum height of 0. Since this `Treap` is optimally balanced, it has a balance factor of 1.0 (instead of 0 / 0). Similarly, an empty `Treap` also has a balance factor of 1.0.

When these special cases aren't met, however, the `balanceFactor()` method is solved using a recursive method. The height of a node is equal to one plus the greater height of its two children. The value produced by finding the root's height using this method and then subtracting by one is equal to the `Treap`'s height (the subtraction of one accounts for the use of edges, and not nodes, as a count of height). This is the current `Treap` height. The optimum `Treap` height is found by the formula  $\text{floor}(\log(\text{count}) / \log(2))$ , where `count` is the number of elements in the `Treap`. The ratio of these two values is then returned.

### Testing

`TreapTest`, the testing class, runs the `TreapMap` class through plenty of different situations in an attempt to expose bugs. It uses the contents of the `words.txt` (the dictionary) file provided for Assignment # 5 as elements and their corresponding keys (so the element "aardvark" has "aardvark" as its key). Perhaps the one aspect of the `TreapMap` class that is getting the most testing is the iterator because iteration is the easiest way to validate that correct changes have been made to the `TreapMap`.

In order to determine whether the `TreapMap` is correct at a certain time, an `ArrayList` usually containing identical elements is used. Ideally, the two should react identically for identical commands. If they don't, the fault is in `TreapMap`.

- 1.) Correct handling of identical elements is tested. Every word from the dictionary is added twice and then the contents of the `TreapMap` are compared with the contents of the `ArrayList`. They should be identical. The elements are found using the `lookup()` method. Since `resetIterator()` hasn't been called yet, this also tests whether the iterator works without first being reset, as it should.
- 2.) Removal is tested by itself. Every word from the dictionary is removed twice (the second time to make sure that removing something not in the `Treap` does not have unintended consequences) and tested to see whether it still exists in the `TreapMap` (it shouldn't).

- 3.) The `split()` and `join()` commands are tested together. First, the `TreapMap` is split based on a certain `String` (this is repeated for various `Strings`). Every value in the lower half of the split is tested to see whether it is indeed lower than the split `String`. Similarly, every value in the higher half is tested to see whether it is greater than or equal to the split `String`. Then, if the split `String` was originally in the `TreapMap`, the higher half is tested to see whether the split `String` is also in it. Finally, the two halves are joined and the resulting `TreapMap` should have every element that the starting `TreapMap` has.

This tests `lookup()`, `insert()`, `remove()`, `split()`, `join()`, and the iterator.

Although `TreapTest` is required to work properly for any implementation of the `Treap` interface, I temporarily modified it to test that the iteration worked with insertion and removal, which it passed. Additionally, I went throughout my `TreapMap` code and modified individual lines to see if `TreapTest` caught the changes. Every time, testing an incorrect version of `TreapMap` resulted in an instant error. By doing this, I guess I was testing the tester.