# HW 4: Red-Black Tree

I have written a new program to analyze word counts in texts, with more functionality than in the last assignment. It relies on a `RedBlackTree` object, that you must write.

As before, the program loads up a text, and counts the number of unique words it contains. It also has new commands that are based on the orderability of keys (such as returning the first key, or the next key after a given key). Here is a typical run of the program using "Alice's Adventures in Wonderland":

```
$ java WordFreqs2 alice.txt
The text contains 2629 unique words.
Please enter a word to query, "!help" for help, or "!exit" to exit.
> !stats
Hash table statistics:
    Size (n): 2629
    Height: 15 (9 black)
    Avg node depth: 9.773
    # red nodes: 681 (25.9%)
    Root key: "in"
> >
The last word (alphabetically) is "zigzag".
> <zigzag
"zealand" comes before "zigzag" alphabetically.
> #616
Word #616 is "dream".
> &dream
"dream" is word #616.
> !exit
Goodbye!
```

Here is the API of the `RedBlackTree` object, with its methods' expected time complexities:

- `RedBlackTree()` (constructor) ($O(1)$). Creates an empty red-black tree.
- `void put(K key, V value)` ($O(\log n)$) Inserts a new key-value pair. You may assume neither is null.
- `V get(K key)` ($O(\log n)$) Returns the value corresponding to the given key, or `null` if the key is not present.
- `V delete(K key)` ($O(\log n)$) Removes a key-value pair, returning the deleted value. Returns `null` if the key wasn't present.
- `boolean containsKey(K key)` ($O(\log n)$) Returns true if the key is present.
- `boolean containsValue(V value)` ($O(n)$) Returns true if the value is present.
- `boolean isEmpty()` ($O(1)$) Returns true if the tree is empty.
- `int size()` ($O(1)$) Returns $n$, the number of key-value pairs in the tree.

- `K reverseLookup(V value)` ($O(n)$) Finds a key that maps to the given value, or returns `null` if there is none.
- `K findFirstKey()` ($O(\log n)$) Returns the key that is less than all the others (or `null` if none).
- `K findLastKey()` ($O(\log n)$) Returns the key that is greater than all the others (or `null` if none).
- `K getRootKey()` ($O(1)$) Returns the key contained in the root (or `null` if none).
- `K findPredecessor(K key)` ($O(\log n)$) Returns the predecessor of the given key, or `null` if the key is not present or has no predecessor.
- `K findSuccessor(K key)` ($O(\log n)$) Returns the successor of the given key, or `null` if the key is not present or has no successor.
- `int findRank(K key)` ($O(\log n)$) Returns the rank of the given key, or `-1` if the key is not present.
- `K select(int rank)` ($O(\log n)$) Returns the word with the given rank, or `null` when the rank is invalid.
- `int countRedNodes()` ($O(n)$) Returns the number of red nodes in the tree.
- `int calcHeight()` ($O(n)$) Returns the height of the tree, where an empty tree has height 0.
- `int calcBlackHeight()` ($O(\log n)$) Returns the black height of the tree, or 0 for an empty tree.
- `double calcAverageDepth()` ($O(n)$) Returns the average distance of the nodes from the root. Empty trees should return `NaN`.

In Java, the key type `K` should implement the `Comparable<K>` interface, so that you can sort keys as needed. You may make any private methods you want, though the lecture notes and the text provide some good suggestions as to which ones to make. And of course, you must adjust the syntax and names as appropriate if you are using a language other than Java.

Remember that color flips and rotations can only happen in certain circumstances. You might find it useful to check conditions before doing these actions, and to throw exceptions when they are violated. You might also want to make additional debugging methods that check your tree, to make sure it is following all the rules. These ideas can save you a lot of time, in debugging.

This is probably the hardest assignment of the semester. Please start as early as you can.

As always, style matters!