CSC 207 **Assignment 8: Trie** **Fall 2009**

Due: Tuesday 28 April 2009

Topics: Recursive data structures, trees, traversals.

Collaboration: This homework assignment may be completed individually or in pairs. The extra credit must be done individually (though you may use the collaborative homework as your basis).

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate test cases and a statement justifying their sufficiency. One only one submission (paper and digital) per group is necessary. If you are submitting extra credit, please indicate which work is your own, and which is that of your group.

Getting Started

As we discussed in class, a trie is an efficient recursive data structure for storing prefixes that supports fast pattern matching. On this assignment, you will complete an implementation of a trie. The skeleton implementation on the MathLAN may be found here:

```
~weinman/courses/CSC207/code/trie/
```

Details

You will need to decide how you want to implement your trie. There are several possibilities, with various ease/efficiency tradeoffs. One important thing to remember is that tries are space efficient because the prefixes are not stored at the nodes; they are kept track of by algorithms that traverse the trie. The one thing your trie must track, however, is whether or not the prefix associated with the root is in fact a member of the lexicon (a dictionary word).

You must provide implementations of the following methods (though you are certainly allowed to—and in some cases encouraged to—add more).

`WordTrie` Two constructors are given. You may add more if you find it necessary, but the two signatures given are (more than) sufficient.

`isWord` This method indicates whether the prefix (whatever it may be) associated with the trie is a dictionary word.

`insert` A core method that adds a dictionary word to the trie. Your insertion algorithm should be recursive. Note that Strings are immutable and calls to `substring` are rather expensive in Java. Thus, you may want to add a helper method that adds a "word" to a trie, starting from a particular character index of a string.

`contains` A core method to query the trie for a particular word. Your algorithm should be recursive. The same facts of immutability and `substring` apply, so you may want to add a (recursive) helper method that checks whether a "word" is in a trie, starting from a particular character index of a string.

`containsSub` A similar method to see if a string is a prefix that may be found in the tree. The prefix need not be a dictionary word. It should merely be a path to some node in the trie. Note that it is possible for both `contains` and `containsSub` to easily piggyback their similarities on top of the same recursive method.

`minLength` Calculates the length of the shortest word. Unless someone calls `insert("")` on your trie, `minLength()` should return 1, for instance, if the dictionary contains the word `I`.

`maxLength` A more interesting method to calculate the length of the longest word in the trie. For instance, if the dictionary contains the word `pneumonoultramicroscopicsilicovolcanoconiosis`, then `maxLength()` should return 45.

`numNodes` A count of the number of nodes in the tree. This corresponds to the total number of possible prefixes in the tree and gives a sense of the compression of the lexicon when stored as a sequence of words.

`subtrie` An important method (for the next assignment) exposing the recursive nature of the trie. Follows the single character prefix to a child trie.

`iterator` A method to iterate over the prefixes that are children of the trie. No particular order is specified. If you build your trie upon the Java Collections framework, this should be a very simple method.

`buildTrie` The most important static method. A factory that takes a list of words and builds a trie for that lexicon.

Important Note On Efficiency The methods `insert`, `contains`, and `containsSub` should be $O(K)$ on average, where K is the length of the query, NOT the number of words or nodes in the trie. In order for this to be true, your `subTrie` method must be $O(1)$ on average. A general implementation of `maxLength` is necessarily $O(N)$, because it will visit every node in the trie.

Extra Credit

I will give 20 points of extra credit on this assignment to those who add the method

```
public Iterator<String> wordIterator()
```

to their trie AND guarantee that the words are iterated in sorted order. This is not a trivial task. You must implement an inner class to create an implementation of `Iterator` that tracks the current traversal of the trie. You are allowed to use a slightly modified version of your trie implementation that no longer gives $O(K)$ performance for the three core methods or $O(1)$ for `subtrie`. Instead, it would likely be $O(K \log B)$ for the core methods, where B is the average branching factor of the trie, and $O(\log B)$ for `subtrie`. Given these modifications it is easy to write a recursive *routine* that amounts to an in-order traversal of the trie. However, an iterator encapsulates that computation. Thus, while a recursive routine tracks the computation using the program stack, you must use your own stack(s) to encapsulate the incremental progress between calls to your iterator's `next` method.

(It took me about 30 minutes to implement this; your mileage, of course, may vary greatly.)