CS 127: Data Structures **Winter 2006**
Handout 6: Problem Set #4 (Due: Thursday, 9 February 2006, 11:59p) *2 February 2006*

I like trees because they seem more resigned to the way they have to live than other things do.
— Willa Cather (1873–1947), *O Pioneers.*

This is a group assignment, to be done in your newly assigned group.

We're going to take a break from the web browser and turn to the "search results with context" problem: given a large text, we want to preprocess the text so that we can search it rapidly. Furthermore, when we find a result, we want to give the searcher a snippet of text around the sought word. This problem arises in search engines—when you Google for "macrame", you get about a twenty-word window including the search term as a summary of the page. (Google hit #1: "This page contains an introduction to the craft of **macramé** and links to my ...") We're going to live in the realm of pure text for now, mostly because it's a real pain to deal with HTML tags. The overall plan for implementing this search capability is the following:

- Take a huge text file—say the complete works of William Shakespeare.
- Build a giant array of the words contained in the text, so that `words[i]` is the ith word in the text.
- Build a binary search tree on the set of words in the text (i.e., without duplicates), where the node in the tree for the word "sapphire" stores a list of all indices i such that `words[i] = "sapphire"`.
- To search for a word w, we'll get the list of indices where the word is found using the tree, and then we'll use the array to look up the surrounding words.
- We'll also implement searching for a two-word phrase.

Create a directory called `ps4`, and a file `ps4.txt` in that directory. Describe anything interesting about your programming experience there. There will also be a question or two in the following that you'll need to answer in your `ps4.txt`. (Make sure that both names appear there and in your comments!)

0. Estimate the amount of time you spent on this problem set, and write it at the top of your `ps4.txt`.

1. Write a class `BinarySearchTree` whose nodes contain two pieces of data: a `String` (that's the word) and a `LinkedList<Integer>` (that's the list of indices where the word appears). Write at least the following methods in this class:

   - `void insert(String word, int index)`, which either adds a new word to the tree, or updates the node for word to include `index` in its list of indices.
   - `LinkedList<Integer> lookup(String word)`, which returns the linked list stored in the word's node, or an empty linked list if the word does not appear anywhere.

2. Write a class `TextSearcher` that includes the following:

   - A private `ArrayList<String>`, and a private `BinarySearchTree`. (Okay, so I said it would be an array before, but an `ArrayList` is much easier to deal with.)
   - `public TextSearcher(File file) throws IOException`, a constructor that, given a `File` input (see the Java API and below), does the following: it scans the words in the file one at a time, loading up the `ArrayList`'s i slot with the ith word found in file and inserting the ith word with index i into the BST. Here's some code that I found helpful as part of my constructor:

```
Scanner scanner = new Scanner(file);   // note:  may throw an IOException
scanner.useDelimiter("[\\s\\W]+");     // "separate tokens using whitespace"
while (scanner.hasNext()) {            // while (there's a word left) {
    String nextWord = scanner.next();  //    get the next word
    System.out.println(nextWord);      //    print it out (for now)
}                                      // }
```

- `public String search(String word, int range)`, which returns a `String` containing all "hits" for the word in the file, in a format that's appropriate to display to the user. A "hit" consists of a $(2 \cdot \texttt{range} + 1)$-word window with the query word in the middle. (Be careful! If you have a hit in word 5 of the document and a range of 10, you don't want to access word -1.)
- `public String search(String word1, String word2, int range)`, which is like the one-word search above, but here you only have a match if you have `word1` immediately followed by `word2` in the text.

  Think about how you want to implement this feature! If you do this without thinking, searching for "this osier"[1] will be slow, because you'll be going through every occurrence of "this" in the complete works of Shakespeare. Think about how you want to do this to make it as efficient as possible. Discuss your choice in `ps4.txt`.
- `public static void main(String[] args)`, which does the following: given a command-line argument of a filename, it should construct a TextSearcher object using that filename. Then it should run some test searches on the file.

  Optionally (but you're encouraged to do this), you can include a loop asking the user to enter queries in your main method.

3. Test your `TextSearcher` class on some large text files. I recommend that you use Project Gutenberg (`http://www.gutenberg.org/`) as the source of your test files. In your `ps4.txt`, analyze the running times of your two search methods, and of the preprocessing phase (the constructor).

Some tips:

- *This assignment has several complex pieces! Be sure to develop and test your code incrementally.*
- Start early! You can start much of question 2 even before you know anything about trees.
- There's a sample run of my code on the website. It's worth comparing to what you produce.
- If you use the complete works of Shakespeare for testing, as I did, you may run out of memory in the virtual machine. Run something like `java -Xmx200m TextSearcher shaks12.txt` to increase the memory allocation for the JVM.
- Helpful things to look up in the Java API: `StringTokenizer`, `File`, `IOException`.
- A good way to test your BST: print out the contents of each node when you add an entry to it. Try testing on a small text file first, so that you know what words you should find in the tree.
- If you haven't dealt with command-line arguments before, they're relatively straightforward: the `main(String[] args)` method takes one argument, which is an array of strings. So if you run `java TextSearcher shaks12.txt`, then `args[0]` will hold the string `"shaks12.txt"`.
- Exit gracefully if the user doesn't type in the right number of command-line arguments.
- My solutions do not do upper/lower-case conversions, so "but soft" and "But soft" will yield different results. But you're welcome to implement this feature!
- Of course, you can't use the built-in tree classes in Java for this assignment, but you're welcome to use built-in `LinkedList`s and `ArrayList`s.

[1] *Romeo and Juliet*, Friar Tuck: "I must up-fill this osier cage of ours/With baleful weeds and precious juiced flowers."