CAS CS 112. Assignment 6

Due 11:59pm on Tuesday, November 22, 2011

Problem 1. (40 points) Your task is to implement an iterator for binary search trees, starting with the binary search tree code from lecture. Your iterator will return the data (not the keys) each time `next` is called. Note that iterators have to implement three methods: `hasNext`, `next`, and `remove`. You may assume that the tree will not be modified while your iterator is active, except by the iterator's own `remove` method.

For binary search trees, it is natural to have nodes iterated inorder, because that way they arrive in increasing order of keys, which is what you typically want. The tricky part for the iterator is that you have to do it without recursion. We have augment the binary search tree with parent pointers to make your job easier.

Hint: Store, in a separate `Node` variable within the iterator object, the most recently returned node (initially, null). You need it in case the user wants to call `remove` (note that the user cannot call remove before `next` was called, or twice in a row without calling `next` in between—if that happens, you should throw `IllegalStateException`). Store also, in another `Node` variable within the iterator object, the next node you are going to return when the user of your iterator calls `next`. That variable is, essentially, one step ahead of the user, anticipating the call to `next` with a ready answer. Initialize these two variables in the iterator constructor. Each time `next` is called, you return the node, and advance the two variables. One of those easy—it just catches up to the other. The other one is tricky: you need to figure how to find an inorder successor of a node (this is easy if the node has a right child, but more complicated if it doesn't).

We provide code to test your iterator in TreeClient.java and a data file data.csv (which contains the rosters of BU and BC hockey teams). You should, of course, test more special cases than we do, to make sure everything works correctly. We also provide you all of the `BinarySearchTree` code: you just have to fill in parts of the iterator. Submit only your modified `BinarySearchTree.java`.

Problem 2. (60 points) In this assignment you will implement a program that deals with anagrams. Two strings are anagrams of one another if by rearranging letters of one of the strings you can obtain the other. For example, the string "toxic" is an anagram of "ioxct." For the purposes of this assignment, we are only interested in those anagrams of a given string that are single dictionary words (i.e., no multi-word anagrams). The dictionary you will use is the Tournament Word List (TWL), a list of 178,691 words used by Scrabble players.

Write a program that, on an input word, prints out all anagrams of that word that appear in the TWL. For example, on input "salt", your program would print "slat salt lats last alts", on input "tblea" your program would print "table bleat blate", and on input "lkj" your program wouldn't print anything.

To solve this problem, you will build a data structure that allows you to efficiently search for anagrams of a given word. It will be a hash table with chaining, augmented by the following clever trick: before computing the hash value of a word, sort its letters to produce its key, and apply the hash function to the key. For example, the key for the string "toxic" is "ciotx", similarly the key for both "star" and "rats" is "arst". (Expert Scrabble players will recognize this procedure as computing the alphagram of a set of letters.) You will convert all letters to lowercase before doing any work on them, to avoid having to deal with different cases.

This approach guarantees that all words that are anagrams of one another are stored in the same bucket of the hash table. When you are searching for anagrams, you will first compute the key of the

word you are searching for, then hash the key, then search that bucket for anagram matches. You should feel free to use the methods described in the text and in class for appropriate hash functions for hashing strings (but please cite any source which you use), or use Java's built-in hashCode method.

Structure your code as follows. Produce a `Word` class, which stores a word and its alphagram. Its constructor will take a string, convert it to lowercase, and compute its alphagram (you will find `String.toCharArray()` method helpful; to sort the characters of the word, you can use a simple insertion sort, since the number of characters in a word is small and any fancier sorting algorithm is probably not worth it). The `Word` class will have public methods `boolean isAnagram(Word w)`, `String toString()`, and `int hashCode`.

Then write your `main` in the `Anagrams` class. Then create a table of `LinkedList<Word>` (a reasonable size is about $100,000$). Since Java has problems with generic arrays, here's the incantation to create the table:

```
LinkedList<Word> [] table = (LinkedList<Word>[]) new LinkedList[TABLE_SIZE];
```

(Note that, like any array of objects, this creates the array itself, but not the linked lists—they are all null for now and you have to create them.)

Then read each word from the dictionary file (make sure the file you open is called TWL06.txt in the current directory, whatever that is—else, we won't be able to grade your work), create a `Word` for it, and add it to the appropriate LinkedList in your table. You will find the file input code from the previous problem helpful. Once your data structure is created, read inputs from the console, and print the anagrams. You will find the `LinkedList` iterator useful in this part of the code. Loop forever—don't worry about quitting. Test your code, and, in particular, make sure that you are not mistakenly putting all the words into a single linked list.

Submit your files `Word.java` and `Anagrams.java`.

Problem 3. (**For fun, not for credit—only if you have time**). Compute the average square of the list length in your table (this will be related to the efficiency of the table lookup). Compute the average number of anagrams of a word in your table. Compute the average number of anagrams of a word in Jane Eyre.