CAS CS 112 – Spring 2008, Assignment 5

Programming part due at 10:00 pm on Tuesday, April 15

Written part due at beginning of class on April 15

Anagram Solver (75 pts)

In this assignment you will implement a program which prints out all anagrams of a specified string. Two strings are anagrams of one another if by rearranging letters of one of the strings you can obtain another. For example, the string "toxic" is an anagram of "ioxct". For the purpose of this assignment, we are only interested in those anagrams of a given string which appear in the dictionary. The dictionary you will use is the Tournament Word List (TWL), a list of 178,691 words used by Scrabble players.

Algorithm and Implementation

Since we will be performing multiple anagram queries, our first step is to load all of the words in the TWL into an appropriate data structure. A primary requirement is that one must be able to efficiently search this data structure to look for anagrams of a given word. A clever trick that we will use to facilitate this is to work with *alphagrams* of words, not just the words themselves. The alphagram of a word (or any group of letters) consists of those letters arranged in alphabetical order, and is the way expert Scrabble players sort the tiles on their rack. So, the alphagram for the string "toxic" is "ciotx", similarly the alphagram for both "star" and "rats" is "arst".

Now what we will do is store words and their alphagrams into a hash table. For each word, we will compute its alphagram. We will treat the alphagram as the key, and the word as the value of a <key, value> pair. Each pair is then inserted into a hash table, where the hash is applied on the key, but the entire pair is stored. This approach guarantees that all words which are anagrams of one another are stored in the same bucket of the hash table, since those words have common alphagrams. It will be helpful to define a new **Word** class to store the pairs and provide any additional Word functionality.

Once the entire TWL has been stored in a hash table, it is time to solve anagram queries. Prompt the user to entire a string, and output all of the anagrams of that string, or a message if none are found. To find the anagrams, you will have to compute the alphagram of the input string, hash it, and scan the corresponding bucket to find matching alphagrams. You should feel free to use the methods described in class and in the text for appropriate hash functions for hashing strings (but please cite any source which you use).

Submit the following files:

1. Word.java – a Word class as described above.
2. WordList.java – list to store Words (feel free to reuse code from the book or from earlier assignments, and you can only implement the functionality that you will use).

3. HashTable.java – basic functionality of a hash table with separate chaining.
4. Anagrams.java – a main program which loads words from the dictionary into the hashtable and then answers anagram queries.

Additional Details

The hash table code which you provide only needs to have the minimum functionality needed to do this assignment. You may fix a size for your hash table – for efficient searching, I recommend that the final hash table you submit contain around 100,000 buckets. For debugging your code, I suggest you work with a much smaller practice dictionary, perhaps 10 words, and a much smaller hash table, perhaps 8 buckets. You will receive up to 40 points for the HashTable, 15 for the WordList, and 10 each for your Word class and main.

Written Questions (25 pts)

1. Question 5.1 in the textbook, page 194 (10 points)
2. Determining a (Max) Heap (15 Points)

   A binary tree can be printed by using the following function:

   ```
   void printTree (Node t){
     if (t == null)
        return;
     System.out.print("[ " + t.element + " ");
     printTree(t.left);
     printTree(t.right);
     System.out.print(" ] ");
   }
   ```

   Notice that because of the brackets the output uniquely defines the shape of the tree. Write in pseudo-code a boolean function **isHeap(char[] str)**, which takes an array of characters output by the printTree function, and determines if the tree represented by **str** obeys the heap-order property. For example, the string "[36 [24 [21 [19] [17]] [20]] [14]]" obeys the heap-order property, but the string "[36 [14] [21 [30]]]" does not. You may assume that your input is a well-formed expression representing a binary tree, you just need to check the heap-order property.