Joanna Klukowska
joannakl@cs.nyu.edu

Programming Project 4: Find Words - Revisited

Due date: December 6, 11:55PM EST.

You may discuss any of the assignments with your classmates and tutors (or anyone else) but all work for all assignments must be entirely your own. Any sharing or copying of assignments will be considered cheating. If you get significant help from anyone, you should acknowledge it in your submission.

In this project you will revisit the programming project #1. The objective of the program is still the same as in the first project. The program produces all possible words given a set of letters and a dictionary. Your program should display all words with lengths 2 up to the number of letters from the set. For example, if the letters are

```
atrac
```

your program should print in alphabetical order all of the following words (again, assuming that they are all in the dictionary):

```
acta
arcat
at
car
carat
carta
cat
catar
rat
ta
```

You can start with your own program for project 1 or you can use the code that I sent out - this is up to you.

The implementation of the Dictionary class should be different in this project. Instead of storing all the words in an ArrayList object, as you did in project 1, you will be using a binary search tree.

The Program Input

Input File Your program is given the name of the text file on its command line. The text file contains a dictionary that is used by the program. You may assume that the dictionary contains a sorted list of words, one per line. A word is any sequence of letters.

If the filename is omitted from the command line, it is an error. If the filename if given but it does not exist or cannot be opened for reading by the program, for any reason, it is an error. The program should display an error message if any of these conditions occur, and it should be as specific as possible (it is not enough for it to say ”could not open file”). Your program is NOT ALLOWED to hardcode the input filename in its own code. If any of these errors occur, the program should terminate.

Your program should read in all the words from the input file and store them in a suitable way for later use. Your program should read the input file only ONCE.

User Input The user should be prompted to enter a string of 2-10 characters (letters only, no spaces, commas, or any other characters). Your program should accept both upper case and lower case letters. If the user enters any characters other than letters, it is an error. If the user enters too few or too many characters, it is an error. The program should display an error message if any of these conditions occur, and it should be as specific as possible.

If the user enters any uppercase letters, your program should convert them to lowercase before proceeding.

Computational Task

Once the user has entered the letters, the program displays all the words in the dictionary that can be formed as combinations of *any subset* of the letters entered by the user.

Program Design and Implementation

Your program should, as before, contain classes that represent 1) the Find Words game, 2) the dictionary, 3) the collection of letters entered by the user.

The class representing the dictionary has to use binary search tree to store the words (instead of being array based). This means that you need to implement your own binary search tree class. It should be based on the recursive implementation of the BST class that we discussed in class (feel free to use any of the code posted on the website and/or textbook). Your implementation should provide an iterator for the class that returns data according to the inorder traversal. You need to implement contains method to allow for searching within the tree for words. You may also want to provide other method(s) that make it easier to search for prefixes in the dictionary.

Your design of the class representing the dictionary should make sure that the dictionary is not stored as a singly linked list.

Extra credit: Implement an AVL tree instead of the BST tree to store the words in a dictionary.

Programming Rules

You must document all your code using Javadoc. Your class documentation needs to provide a description of what it is used for and the name of its author. Your methods need to have description, specification of parameters, return values, exceptions thrown and any assumptions that they are making.

You must use recursion and backtracking to produce your list of words. If you did not do it correctly or efficiently in the first project, make sure that it is done correctly in this project.

You may use any classes for reading the input from the file and from the user. You may use any exception related classes.

You may use the sort method provided in the Arrays and/or Collections class.

The program has to be fast, i.e., it has to finish in just a second or two even if you are working with the largest of the dictionary files and the user enters 10 letters.

Grading

If your program does not compile or crashes (almost) every time it is ran, you will get a zero on the assignment.

30 points	design and implementation of the dictionary class
35 points	design and implementation of the binary search tree class
15 points	design and implementation of the program (that includes input handling)
20 points	proper documentation
15 points	extra credit part (this includes the balancing of the tree according to the AVL tree rules)

How and What to Submit

Your should submit all your source code files (the ones with .java extensions only) in a single **zip** file to NYU Classes. You should have at least three different files.