CSC 115: Fundamentals of Programming II
Assignment #3: Recursion

Due date

~~Friday, June 29th, 2015 at 9:00 pm via submission to conneX.~~
Tuesday, June 30th, 2015 at 9:00 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #3 link on the CSC 115 conneX site. Please ensure you follow all the required steps for submission (including confirming your submission).

Learning outcomes

When you have completed this assignment, you should have become acquainted with:

- The use of *recursion* (specifically using a style of programming using *recursive backtracking*)
- Using linked-lists to support recursive backtracking.

Anagrams

Problem description

One form of word play is called *anagramming*. Some English phrase is given, such as:

```
I love programming!
```

and the letters in that phrase are jumbled about into different words:

```
VIM romping galore.
```

("vim" is a programming editor used by many programmers.) To be an anagram of a phrase the exact letters (but perhaps with different capitalization) must appear in both phrases. Punctuation is permitted to differ (i.e., spaces, commas, periods, etc.). There are anagram servers on the web that accept a phrase and then, after searching through a dictionary, produce a long list of anagrams—mostly nonsense combinations of words, but often containing wonderful ones which just require a bit of re-ordering to be clever and funny. The most established website is the *Internet Anagram Server* (http://wordsmith.org/anagram/).

Your task for this assignment is to write a program that takes both a phrase and a dictionary of words, and then generates the anagrams for that phrase given the dictionary.

A sample run

Here is one sample run of the finished program:

```
$ java A3driver --dict=files/small.txt --phrase="Justin Trudeau"
unjust, duet, air
duet, unjust, air
unjust, air, duet
air, unjust, duet
duet, air, unjust
air, duet, unjust
```

The dictionary of words in `files/small.txt` is purposely created to provide the anagram for phrase shown above (i.e., `small.txt` has only four words). Note the six lines of output have the same words but in different permutations. (The Internet Anagram Server would suppress permutations, but the algorithm you will implement is simpler that the one on that Server.) *The order of the permutations does not matter; rather what matters is that all permutations appear.* Please note: I've chosen "Justin Trudeau" for an anagram-phrase example as his name is topical and it has lots of vowels, i.e., my choice does not indicate my political persuasion!

AlphabetStats.java (provided to you)

When are two phrases anagrams of each other? **When their letter-frequencies match.** Keeping track of such frequencies can be a bit tedious, however, and you are provided with a class named *AlphabetStats*. It has the following methods:

- `public AlphabetStats(String phrase):` Computes the letter frequencies for the letters in `phrase`. It ignores characters that are not part of the alphabet, i.e., ignores whitespace and punctuation.
- `public String toString():` Produces a string representing letter frequencies currently in the instance of *AlphabetStats*. (This means that an instance of *AlphabetStats* can be given as an argument to System.out.print.)
- `public boolean contains(AlphabetStats as):` Compares the letter frequencies in the instance with those of `as`. For example, `as1.contains(as2)` returns *true* if all the letter frequencies in `as1` are greater than or equal to those in `as2` and returns *false* otherwise.
- `public boolean add(AlphabetStats as):` Adds all of the letter frequencies in `as` to the *AlphabetStats* instance. For example, after `as1.add(as2)` is performed, all of the letter frequencies in `as1` will have been incremented by the letter-frequency values in `as2`; `as2` is unchanged. `boolean subtract(AlphabetStats as)` decrements letter-frequency values.

- `public boolean isEmpty()`: Returns false if the *AlphabetStats* instance has 0 for all letter frequencies.

This class should be used as part of your recursive-backtracking algorithm for finding anagrams.

So to go back to the original question in this subsection, "When are two phrases anagrams of each other?", an answer is suggested below:

```
String phrase1 = "aabbcc";
String phrase2 = "cba cba";
AlphabetStats as1 = new AlphabetStats(phrase1);
AlphabetStats as2 = new AlphabetStats(phrase2);
as1.subtract(as2);

System.out.println("The two phrases are " +
    (as1.isEmpty() ? "" : "NOT ") +
     "anagrams of each other"
);
```

WordList.java, WordListNode.java, Anagrammer.java

Your task is to write three classes. *WordList* and *WordListNode* will be used to store intermediate results during recursion, and *Anagrammer* contains the code actually performing the recursive-backtrack search for a phrase's anagrams. I will spend some time in lectures explaining how recursive backtracking works (and the text also describes such backtracking for the "eight-queens" problem).

WordList is a ref-based linked list with six methods (other than the constructor):

- `void insertHead(String w)`: Inserts at the head a node storing `w`.
- `void removeHead()`: removes the node from the head of the list.
- `boolean contains(String w)`: returns *true* if a node in the list already contains a string equal to `w`.
- `boolean isEmpty()`: returns *true* if the list is empty (i.e., its head is equal to `null`).
- `int getLength()`: returns the length of the list..
- `String retrieve(int index)`: returns a string from node in the list at position `index`; if the index is not legal, then `null` is returned.
- All test routines for *WordList.java* are to appear in the `main()` method of *WordList*. (There must be at least eight test cases.)

Anagrammer has at most two public methods (although you a free to add as many private methods as needed) and one private method:

- `Anagrammer(String dictionary[], String phrase, int maxWords)`: A constructor to which is passed in some already-input dictionary in the form of a string array, the phrase string, the maximum length of acceptable anagrams for the phrase.

- `public void generate():` Using the phrase, dictionary, and maximum length of anagrams provided to the constructor, this method generates the anagrams that result from the combination. Note that `generate()` itself is not recursive, but rather it will call the (private) recursive method you add to the class.
- `private void findAnagram(WordList words, AlphabetStats phraseStats):` This method is meant to be recursive. The first parameter is the list of words so far in current anagram search; the second parameter represents the letters from the original phrase which are still left out of the list of words. One base case for `findAnagram` is when `phraseStats.isEmpty()` returns `true` (i.e., the contents of `words` is an anagram of the given phrase). There is at least one more base case and of course there must, of course, be a recursive call to `findAnagram`. (Some of this will be discussed during lectures and perhaps labs.) *For this assignment you must assume that anagrams do not contain repeated words.*
- All test routines for *Anagrammer.java* are to appear in the `main()` method of *Anagrammer*. (There must be at least eight test cases, and none of these may be one of the provided tests.)

You are provided with *A3driver.java* which does the work of reading command-line arguments, opening and reading the contents of the dictionary file, and invoking *generate()* on an instance of *Anagrammer*. Do not modify *A3driver.java*.

Three different dictionaries (*files/small.txt*, *files/medium.txt*, *files/large.txt*) are provided to you. You are, of course, welcome to explore the assignment with additional dictionaries you may find. (Keep it clean, folks!) The content of *files/README.txt* describes how the three test outputs in *files/* were created.

What you are to write

1. Please keep all code within a single directory. We will take advantage of what is called the *default package* in Java, i.e., all Java files in the same directory are considered to be in the same package. This will permit you to use package-private access for the assignment without a complex directory structure.

2. Write *WordListNode.java* and *WordList.java*. Ensure *WordList.java* has a *main()* with tests.

3. Complete *Anagrammer.java* using recursive backtracking (as outlined in lectures). Do not delete the redundancy in the output—for example, if four words are found to make up an anagram, you will see that combination printing in 4! (i.e., 24) lines of output. Ensure *Anagrammer.java* has a *main()* with tests.

4. Create a file named *test_output.txt* that describes your approach to testing.

5. In all of your coding work, please follow the coding conventions posted at conneX (i.e., a document in the "Lectures & stuff" section).

Files to submit (four in total):

1. *WordList.java*
2. *WordListNode.java*
3. *Anagrammer.java*
4. *test_output.txt*

Grading scheme

- "A" grade: An exceptional submission demonstrating creativity and initiative going above and beyond the assignment requirements. The program runs without any problems and implements the required classes and methods in those classes, and uses recursive backtracking. Required methods are tested with output showing tests and their results (i.e., pass or fail). Any extra work appears in class files having the name *<file>Extra.java*, and identified within the file (i.e., Class comment) are details of your extension to the assignment demonstrating creativity and initiative.

- "B" grade: A submission completing the requirements of the assignment. The program runs without any problems and implements the required classes and methods in those classes, along with recursive backtracking. Required methods are tested with output showing tests and their results (i.e., pass or fail). Public methods are tested with output showing tests and their results.

- "C" grade: A submission completing most of the requirements of the assignment. The program runs with some problems and uses recursive backtracking but does implement the required classes and methods, and yet might not have the expected output. Required methods are tested with output showing tests and their results (i.e., pass or fail).

- "D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not contain implementations of all required classes and methods.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.