Programming Project 3
CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development
November 25, 2019

This is the last programming project for the course!

0. Introduction.

An **anagram** is a word made by rearranging the letters of another word. For example, the English words `dearer`, `reader`, `reared`, and `reread` are anagrams of each other, and so are `present`, `repents`, and `serpent`. For this project, you will write a Java program that finds all sets of anagrams in a text file of English words. Your program will use an algorithm based on binary search trees, so it will be very fast.

How fast? The Russian author Leo Tolstoy's ***War and Peace*** is one of the longest published novels ever written, with over a thousand pages, and more than a half million words. My version of the program reads the text of this novel in about half a second, finding 793 sets of anagrams. (Your computer, and your program, may be faster or slower.) Among other things, this demonstrates that efficient programs require efficient data structures.

1. Theory.

We'll start by considering why the obvious algorithms for this problem are too slow. Suppose we read all the words from a file, and then compare each word with every other word, testing if each pair of words are anagrams. Although this algorithm would indeed find all the anagrams among these words, it is very inefficient. If there are N words, then it requires $O(N^2)$ time: too slow for large files. For example, if *War and Peace* has 5×10^5 words, then we might need 2.5×10^{11} tests to find all its anagrams.

We also need a way to test if each pair of words are anagrams. We might generate all possible rearrangements of one word, then test if one of them is equal to the other word. Unfortunately, if a word has w letters, then it can have at most $w!$ distinct rearrangements, so comparing words this way takes $O(w!)$ time: too slow for long words. For example, if a word has 10 letters, then we might test $10! = 3{,}628{,}800$ rearrangements.

We conclude that we can't compare each word with every other word, and we can't compare words by rearranging their letters. We can avoid both these limitations by using a version of the K-and-R string comparison algorithm discussed in the lectures, and by using binary search trees. Here's how to do that.

Summaries. Suppose that a *word* is a string of one or more lower case Roman letters `a` through `z`, without accents, blanks, or punctuation marks. Also suppose that the letter `a` corresponds to 0, that `b` corresponds to 1, etc., up to `z` that corresponds to 25. (These are *not* the ASCII or Unicode codes for those letters!) For each word, we set up an array of 26 integers, called a *summary* of that word. In the summary, the element at index k tells how many times the letter corresponding to k appears in the word. For example, if we have the word `present`, then its summary looks like this, where the large numbers are array elements, and the small numbers are array indexes. Note that `present` has two `e`'s (at index 4), one `n` (at index 13), one `p` (at index 15), one `r` (at index 17), one `s` (at index 18), and one `t` (at index 19).

0	0	0	0	2	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Now, two words are anagrams if and only if their summaries are equal. For example, `repents` and `serpent` have the same summary as `present`. To test if two words are anagrams, we construct summaries for both words, then test their summaries for equality. We can construct a summary of a word with w letters in $O(w)$ time, and we can test two summaries for equality in $O(26) = O(1)$ time. As a result, we can test if two words are anagrams in linear time.

Comparing summaries. We can also test summaries S_1 and S_2 using a version of the K-and-R algorithm. It visits the elements of S_1 and S_2 one at a time, from left to right, until it finds the first pair of unequal elements. Then it subtracts the element of S_2 from the element of S_1 and returns their difference. If it never finds unequal elements, then it returns 0. In either case, if the returned difference is less than 0, then $S_1 < S_2$. If it is equal to 0, then $S_1 = S_2$. If it is greater than 0, then $S_1 > S_2$.

Anagram trees. Because of the K-and-R algorithm, summaries are totally ordered, so we can use them as keys in a binary search tree. We'll call it an *anagram tree.* The anagram tree's keys are summaries, and its values are linear singly linked lists of strings, in which each string represents a word. The anagram tree is the basis for our efficient anagram finding program.

Here's how the program works. We start with an empty anagram tree. Then we read words as strings from a text file. Every time we read a word, we construct its summary. Then we search for a node in the tree that has the summary as its key, using the K-and-R algorithm to control the search. If we find the node, then we add the word to the node's list (if it's not already there). If we don't find the node, then we add a new node to the list. The new node's key is the summary, and its value is a list that contains the word.

After we've read all the words, there will be many nodes in the anagram tree, each with a list of one or more words. The words in each list are anagrams of each other, because they all have the same summary. We then traverse the tree, visiting each node. If we find a node whose list has only one word, then we ignore it, because the word is an anagram only of itself. However, if we find a node whose list has two or more words, then we print those words on the same line, because they are anagrams of each other.

Suppose the program reads N words. If we assume the words are read in random order, then it takes $O(\log N)$ time to add a word to the anagram tree. Since we add N words, it takes $O(N \log N)$ time to build the entire tree. Since the tree has N nodes, it takes $O(N)$ time to traverse it. As a result, the program runs in $O((N \log N) + N) = O(N \log N)$ time.

3. Implementation.

Here's how to write the program. You must use the Java class **`Words`** to read words from a text file. Java source code for `Words` is available on Canvas. You don't have to write it yourself. You don't even have to know how it works internally. All you need to know is that it acts like an iterator, and that it has the following public methods.

`public Words(String path)`

> Constructor. Initialize an instance of `Words` that reads words from a text file whose name is `path`. Throw an `IllegalArgumentException` if the file doesn't exist, or if it can't be read for some reason.

`public boolean hasNext()`

> Test if there is a word waiting to be read from the text file.

`public String next()`

> Read the next word from the text file, convert all its letters to lower case, and return it as a `String`. It isn't `null`, it isn't empty, and it contains only lower case letters.

You must write a class called `AnagramTree` that represents an anagram tree as described in the previous section. It must have the following members. To simplify grading, you must use the same names as are shown here.

`private class TreeNode`

> (5 points.) A node in the `AnagramTree`. It must have four private slots and a private constructor. The slot `summary` points to an array of 26 `byte`'s: it's the key. The slot `words` points to a linear singly linked list of `WordNode`'s: it's the value. The slots `left` and `right` point to `TreeNode`'s, or to `null`: they're subtrees.

`private class WordNode`

> (5 points.) A node that contains a word. It must have two private slots and a private constructor. The slot `word` must point to a `String` that represents the word. The slot `next` must point to a `WordNode`, or to `null`; it's the next node in a singly linked linear list.

`public AnagramTree()`

> (5 points.) Constructor. Initialize an empty instance of `AnagramTree`. It must have a head node.

`public void add(String word)`

> (20 points.) Add `word` to the anagram tree, as described in the previous section. This `String` isn't `null`, it isn't empty, and it has only lower case letters. You must use `compareSummaries` to control the search through the tree. You must use the tree's head node to avoid a special case when you add `word` to an empty tree.

`public void anagrams()`

> (10 points.) Traverse the anagram tree, visiting each of its `TreeNode`'s exactly once. You must skip the tree's head node. Every time you visit a `TreeNode`, you must print all the words from its linked list of `WordNode`'s on one line, separated by blanks. However, if the linked list has only one node, then you must ignore it.

`private int compareSummaries(byte[] left, byte[] right)`

> (10 points.) Here `left` and `right` are summaries: arrays of 26 `byte`'s. Compare `left` to `right` using the K-and-R algorithm. If `left` is less than `right`, then return an `int` less than 0. If `left` equals `right`, then return 0. If `left` is greater than `right`, then return an `int` greater than 0.

`private byte[] stringToSummary(String word)`

> (10 points.) Return a summary for `word`. This `String` isn't `null`, it isn't empty, and it has only lower case letters. The summary must be represented as an array of 26 `byte`'s. If `c` is a character from `word`, then you must use the Java expression `(c - 'a')` to compute `c`'s index in that array. You must not use `if`'s, `switch`'es, or loops to compute `c`'s index.

You must also write a class called `Anagrammer`. It's the driver, and it must have only a `main` method.

`public static void main(String[] args)`

> (5 points.) Make an instance of `Words` that reads words from a text file. Make an empty `AnagramTree`. Read all the words from the text file and add them to the tree. Finally, traverse the tree to print all its anagrams.

Here are some hints, notes, and threats.

- Your `AnagramTree` class must have two nested classes: `TreeNode` and `WordNode`. Do not try to use only one nested class, because that won't work.
- You may add as many private variables to the class `AnagramTree` as you want. However, you must not use a private variable when

a local variable would work instead.

- You may write as many private helper methods as you want. You must write at least one helper for the method `anagrams`.
- Your anagram tree must use a head node. You may design the head node however you want. However, recall that the head node's key must appear nowhere else in the tree.
- You must represent summaries as arrays of 26 `byte`'s. Do not use arrays of `int`'s, because that would take more memory. Recall that `byte` is Java's smallest integer type: it can hold integers from −128 to 127. We assume that no word will have more than 127 appearances of the same letter, but you don't have to check for this.
- I tested my version of this program using a text file that contains Tolstoy's ***War and Peace.*** It's available for free from the **Project Gutenberg** web site. It's also available on Canvas.
- When I ran my program on the text of *War and Peace*, it produced this **output,** with one set of anagrams on each line. The anagrams appear in arbitrary order. If you run your program on *War and Peace,* then your anagrams may appear in a different order. You may also want to test your program on a shorter file.
- Some anagrams don't look like English, or even like Russian. This may be because `Words` considers a word to be a series of one or more Roman letters. As a result, it incorrectly reads a word with punctuation, like `don't`, as `don` and `t`. It also incorrectly reads a word with accents, like `Kurágin`, as `kur` and `gin`. There may be misspellings in the original text too. Don't worry about these things.

4. Deliverables.

Unlike the lab assignments, you must work on this project individually, without a partner. **IT MUST BE WRITTEN ENTIRELY BY YOURSELF, ALONE!** The project is worth 70 points, and you must submit it to Canvas by **11:55 PM** on the last day of class: **December 11, 2019.**

The TA's will read your Java code and award partial credit where possible. As a result, you must submit Java source code for the classes `AnagramTree` and `Anagrammer` together in one `.java` file. Do not submit source code for the class `Words`. If you have questions about how or where to turn in your work, then please ask your lab TA's.

Do not submit the text of *War and Peace!* Do not submit the list of anagrams from *War and Peace!* These files are too big for the TA's (or anyone else) to read. However, if you have tested your program on a *short* input file, and have a *short* list of anagrams from that file, then you may include both in comments at the end of your `.java` file.