Zijian Gao
26319560
3/24/2016
CS311 Anagrams Project

Jar Details:

- Simple scanner that reads the file at the location of the first argument given to the file (run using the command "java -jar anagrams.jar [dictlocation]"
- All output is dumped to stdout, an output redirection ">" can be used to direct output to a file.

Data Structures:

- This uses a Binary Search Tree, **AnagramBST**, to hold anagram classes. Each **AnagramClassNode** can have a left and right child, and stores a data object, **AnagramClassNodeData**.
- **AnagramClassNodeData** represents an anagram class, with a linked list of strings representing each member of the class, and a "tag" value, which is the alphabetically sorted version of the anagrams. The "tag" value is an absolute representation of any possible anagram for that "tag", such that if any anagram of this "tag" value exists in the dictionary, when it is sorted, it will match the respective "tag" value for its respective anagram class.
- Both **AnagramClassNodeData** and **AnagramClassNode** implements **Comparable**<**?**> for convenience determining whether a given node is less than/equal/greater than a node.
- Adding a member to the **AnagramBST** involves searching for the anagram class that the word belongs in, if it exists. If the class does not exist, a new AnagramClassNode is created and added to the search tree. The convention used for if a member is equal to its parent node is to set as the left child node of the parent.
- Only a limited subset of standard BST operations are required for this, namely addNode, searchNode and and inorder traversal for printing the nodes.
- The BST root node itself will never have a left subtree due to there not existing a string that is less than an empty string, but its right subtree can have a left sub-sub tree.

Algorithm Design:

This anagram algorithm uses the fact that one string is equal to another string if and only if it contains the same characters in the same size sequence. Before deciding whether a read word belongs in an anagram class, the word is first sorted (the original value is preserved, too) such that if a class is already existing in the BST, it can be uniquely identified by that sorted string "tag".

Steps:

1. While there are still words in the dictionary, read a word from the dictionary, save this value
2. Sort the word alphabetically using counting sort, call this the "tag", retaining the original value as well.
3. Add the member to the binary search tree
    1. A binary search is done using string comparison of the "tag" value against the "tag" value of the anagram classes on the search path.
        1. If the tag is found, an anagram class for the read member has already been created, and the member is added to the anagram class with the matching tag.
        2. If the tag is not found, a new anagram class is created and added to the proper location by again using a binary search to search for the location that it belongs in.
4. End loop

5. Dump each anagram class, one per line, each anagram separated by one whitespace character, to standard out using in-order tree traversal to visit each node besides the sentinel root node.

Proof of Correctness:
It is a fact of string equality that one string equals another string if and only if they contain the same characters with the same length and in the same sequence. Additionally for some anagram, call it *a*, of some tag *t*, there exists exactly one distinct alphabetical sorting(from a to z). Counting-sort is a sort that is proven correct and works on numbers, and since a string is simply an array of characters, which take values from 0 to 255, counting sort can be utilized to alphabetically sort a string.

Because of this, the only members that when sorted can possibly match a certain tag value, are members that contain exactly the same characters in the same order and size when sorted as that tag value.

Searching and adding to a binary search tree are trivially correct operations. When a member matches an anagram class, that class is guaranteed to be found and the member is guaranteed to be added by the properties of a binary search tree and linked list. When a member does not match a class, the node is guaranteed to be added to the tree barring issues related to lack of memory.

The loop invariant for the main loop is conditioned on the existence of more lines to read in the dictionary file, and so is guaranteed to terminate when there are no more lines to read.

Time Analysis:

Average case analysis:
I perform the average case analysis in depth instead of worst-case here because it is assumed that by running the anagram program, a dictionary that contains at least one anagram will be used, otherwise, the dictionary would trivially already be a list of one-word anagram classes.

Let k be the average letters per word in some dictionary
Let w be the number of words in that dictionary.

Step 1 takes $O(k)$ time to read the word. The loop invariant will terminate in $O(w)$ time
Step 2 takes $O(k)$ time as counting sort is a linear sort.
Step 3 takes $O(k \log w)$ breakdown:

Part 1: The first search takes $O(k \log w)$ time because the average height of the binary search tree would be achieved if there existed some anagrams in the dictionary. To search the tree, since string equality requires matching each character, each node that is visited needs to be compared against k letters on average. The height of the tree would be $O(\log w)$, which is the maximum amount of nodes that will be searched on average.
Part 1.1: Adding a member to an existing class takes $O(1)$ time as adding an element to a linked list is a constant time operation
Part 1.2: Creation of the anagram class and adding the member to the empty list are both constant time operations. Finding the correct location to add the new anagram class node takes $O(k \log w)$ time as it is a search.

Step 4 takes $O(kw)$ time. $O(kw)$ to print each anagram class, because each character of each word must be printed and the traversal will reach at most w nodes, but on average if half of the words are anagrams, there will be $O(\log w)$ nodes, so $O(kw)$ dominates.
Total runtime: $O(wk \log w)$, the loop dominates the runtime of this algorithm.

Worst case analysis:
In the worst case, the height of the tree would be $O(w)$ in the case that there do not exist any words that are anagrams of any others, and this is the maximum count of nodes in the tree. So in the worst case, the runtime is $O(w^2 k)$