Description, Correctness and Time analysis

Description

I decided to implement a product of primes to uniquely represent each Anagram Class and then hash that product to associate each input word to each value of the hash table, which represent one Anagram Class. The Fundamental Theorem of Arithmetic states that any integer can be decomposed into a unique product of primes and from this we can conclude that each distinct product of primes will yield a unique integer value. We can extend this to our problem by assigning the first 26 primes to each letter of the alphabet. Let this unique value for each Anagram Class be denoted as $\boxed{K_i}$ and this will be our key for our hash table. The hash table takes a key K_i and our hash function $h(K_i)$ is simply $h(K_i) = K_i \mod n$, where n is the size of the input array in terms of the number of words. Each value stored by the hash table will be considered as one Anagram Class. We can simply assign to the hash table array at index $h(K_i)$ a Linked List that represents one Anagram Class. There exists a problem of collisions as although each Anagram class has its unique representation, the hash value that comes from $h(K_i)$ is not necessarily unique. This is why we simply implement linear probing and look for the next empty value. Furthermore, I also implemented something that's kind of redundant but for every word, a Word class is generated, and each Word object has a String that represents that word with its characters sorted with Counting Sort. In addition, each Linked List class also has this sorted word to ensure for extra security if the unique number representation of an Anagram Class fails for some reason.
After every word is added to the hash table, we simply extract them linearly from the table and write each class to a line in the output.
For the sake of simplification, here is the pseudocode for the entire program, including File I/O:

```
Anagram(File F):
    //Define hashtable and array of everything
    HashTable hashTable
    String[] all

    //File I/O
    for String s in F do:
        add s to all

    //Main script
    for String s in all:
        add s to hashTable

    //Extract from hashTable and write to file
    for LinkedList in hashTable:
        for Node in LinkedList:
            write Node.value to file + " "
        write new line

    //If curious
    write final count of anagram classes
    write final count of values
    write runtime in seconds
```

Correctness

The Anagram class classification hinges on the *put*() method of the HashTable/Map.
Since the Anagram Classes are stored in an array of LinkedLists in the HashTable class, there are two cases to consider when adding values to the HashTable. Define the array entries[] to be the array of LinkedLists/Anagram Classes.

- Case 1: entries[$h(K_i)$)] where $h(K_i) = K_i \mod n$ access a null value of the array. This means that there is currently no Anagram Class associated with the given input word and its key. We simply create a new LinkedList associated with this new Anagram Class and store it in the array

- Case 2: entries[$h(K_i)$)] results in a collision. This means that two Anagram class values K_i and K_j are distinct but $h(K_i) = h(K_j)$. We simply find the next empty node by incrementing the the output of $h(K_i)$ OR stop when we find a node of the array where the LinkedList at that node has a key and alphabetically sorted string that's the same as the input string. We can then just add our current input value to that node. If we increment too much, we simply loop back to the beginning.

Either case, each string is guaranteed to be stored in its desired LinkedList and therefore each word is correctly classified into its Anagram Class. Even if collision happens, the required LinkedList is guaranteed to be somewhere within the next few number of nodes as it that LinkedList was previously a result of linear probing.

Time Analysis

Let n be the number of words in the input file and let c_i be the number of letters of the ith word of the input file. Reading the file of course takes $\boxed{\Theta(n)}$ time. At a naive view, inserting the a word into the hash table will take $\Theta(1)$ time for each word. But to account for collisions, in a worst case analysis assuming the input file actually has a minimal number of Anagram Classes, the hash value will loop around the entire length of the array, which makes insertion take at maximum $O(n)$ per value and so insertion for every word will take $O(n^2)$ but in an average case assuming there are very few collisions which there should be, insertion will take on average $\boxed{\Theta(n)}$ time. Furthermore, with each insertion, each word has to be processed into its prime representation and its alphabetically sorted form with Counting Sort on the characters, each taking $\boxed{\Theta(c_i)}$. Finally, we simply write every Anagram class to the output file by scanning through the hash table and since we're still outputting n words, this runtime will also be $\boxed{\Theta(n)}$. Putting everything together, this program runs in $\Theta(3n + 2c_i)$ and dropping constants we have $\boxed{\Theta(n + c_i)}$