CS21120 Assignment

Introduction

In this assignment we were tasked to output a list of words that rhymes within a given word. To do this we were given a CMU dictionary text file that contains ~135,000 lines of words and pronunciation.

For this coding assignment I didn't use any coding AI-assistance, only resources online such as java documentation, Stack Overflow, W3Schools and other websites. As for the documentation I will be using a grammar checker website QuillBot (R4).

Task 1: Implementing IPhoneme

Implementing IPhoneme was relatively simple because I had to create a Phoneme class that complied with the IPhoneme interface, which then generated methods from the interface and added extra codes, such as the constructor arguments are stored as attributes and checks if the vowel or non-vowel phoneme stress value is valid. For example, a vowel phoneme cannot have a stress value of -1, whereas a non-vowel phoneme must have a stress value of -1.

Task 2: Implementing IPronunciation

Implementing IPronunciation was a bit tricky for me at first because I couldn't quite understand how the rhyming process worked therefore coding findFinalStressedVowelIndex() took some time.

To find the final stressed vowel, I had two choices as to either go through the list of Phonemes front to back or vice versa. When looking through the CMU dictionary text file, I realized not all primary stressed vowel words are towards the end of the pronunciation of the word, but rather it is randomised. Since the idea is to find the final stressed vowel, I decided to search back to front where upon finding a primary stressed vowel it will instantly return the index otherwise it will go through the whole list of Phonemes and return the index with priority to former highest stress vowel.

Initially I used a LinkedList data structure to store the list of Phonemes however, upon reevaluating my code, I realized an ArrayList is a more suitable data structure because of two specific operations I used in the pronunciation class, addPronunciation() which adds phoneme to listOfPhonemes and findFinalStressedVowelIndex().

One thing I would like to point out is that when using .add() on a Phoneme to an ArrayList, it will have a default capacity of 10 (R2) and when reading from the CMU dictionary text file, majority of the words will have a list of Phonemes of 10 or less therefore, making the average-case time complexity of θ(1) and in the rare cases O(n) will happen.
As for ArrayList indexing, the average-case time complexity is θ(1) which is constant time.

LinkedList on the other hand, holds references to the first and last element, so adding will have an average-case time complexity of θ(1) however, as for LinkedList indexing, it has to traverse the list until it finds the element, therefore it has an average-case time complexity of θ(n) since it still needs to search for at least half of the list which is still linear time.

As for findFinalStressedVowelIndex() with an ArrayList average-case time complexity, it would be θ (n) when it comes to both ArrayList and LinkedList, because final stressed vowel can only be found anywhere within the list of Phonemes therefore on average it will take n/2 indexing time to find it which is linear time.

Task 3: Implementing IWord

IWord class was relatively simple to implement because it only needed a constructor and 3 methods, so there weren't many moving components to think about. The only notable method to mention is the addPronunciation() where a passed pronunciation object would be added to a setOfPronunciation. As for the setOfPronunciation implementation of the data structure, there were HashSet, LinkedHashSet and TreeSet. I decided to use a HashSet because it will be faster compared to other set frameworks, for example LinkedHashSet, which implements pointers or TreeSet which rearranges data (R3).

addPronunciation() average-case time complexity is θ(1) because storing pronunciation objects in setOfPronunciation will be based on its hash code generated by the hashing algorithm. Assuming the hashing algorithm can produce unique hash codes.

Task 4: Implementing IDictionary and parsing the CMU file

Initially I used a HashSet to store all the Word objects because each of their own had their own .getWord() which outputs an English word so every line from the CMU dictionary text file would be split into words and pronunciations seconds. Then the pronunciation section will be stored into a pronunciation object, which will be stored within a setOfPronunciation within the new Word object.

However, upon coding and testing loadDictionary() it would take 4 minutes and 28 seconds to load the whole CMU dictionary and I deemed that unoptimized and not a suitable solution. This was due prior to storing the English word, looping all stored word objects used .getWord() and checks if the English words matched.

Upon reevaluating, I concluded that a HashMap (R1) needed to be used. I did think about using a HashMap at the very start however, I didn't know why I would need a key and value when a Word object already has all the functionality. But implementing a HashMap with an English word as a key and a word object as a value would make the check if the English word exists within the HashMap quicker because HashMap look-up average-case time complexity is θ(1) and worst-case time complexity is O(n) if there are multiple collisions. Once the HashMap implementation was done and some minor changes happened, then the dictionary can load in 600ms which is considerably a lot faster than the previous version with a HashSet.

With the implementation of HashMap it allows getWord() an average-case time complexity of θ(1) and a worst-case time complexity of O(n) if there are multiple collisions. The addWord() has an average-case time complexity of θ(1) whereas the worst-case time complexity is O(n^2) because checking if the word exist within the dictionary would take O(n) assuming there is multiple collisions within the HashMap and then it would use .put() the pass Word object into the HashMap which will also be O(n) making it O(n^2).

As for getWordCount() it returns the amount of keys which would take an average-case time complexity of θ(1) and getPronunciationCount() returns the amount of values by iterating through the values and using .getPronunciation.size() to get the amount of pronunciation making average-case time complexity of O(n).

Task 5: Rhyming!

For the Dictionary class, I decided to use a HashSet to store all English words that rhyme because HashSet will only keep unique elements therefore there won't be duplicate words and a .put() with average-case time complexity of θ(1).

Upon testing rhymesWith() JUnit tests I realized testRhymesWith_OneSyllableCVC() test kept failing and I found that upon finding the last stressed vowel of both words, past it must have matching phonemes. So I created isPronunciationRhymesWithPronunciation() which checks if two given pronunciation and finalStressIndex past phonemes matches.

Another difficulty I came across was when testing testGetRhymes() JUnit specifically testGetRhymes_Cat() where upon getting all the rhymes with the word "cat" it should have a set of 77 English words that rhyme with "cat" but my code could only find 67 words. I used temporary regex code to find words within the CMU dictionary matches with regex "AE1 T$" and compare it to the 67 words. I found out that the missing words contained two primary stressed vowels within the pronunciation, and it would only return the left-most primary stress index, to fix this I altered the code to return the right-most primary stress index.

As for getRhymes() average-case time complexity of θ(n) because majority of the English words within the CMU dictionary only have one pronunciation so majority of word's pronunciation will only be checked once.

In order to find the English words with the most rhymes, I would use a HashMap containing the English word and number of rhymes respectively as key and value. Then I would create a loop that goes through the dictionary HashMap keys and pass it into the getRhymes(), of which will return a set of English words that rhyme. By using .size() on the HashSet it would return the amount of rhymes. Then with the current English word in the loop and size of rhymes found will be put into a HashMap. Once the loop is finished, iterate through the whole HashMap entry set and return the key with the highest value.

Self-evaluation

Task 4 was the most difficult for me because parseDictionaryLine() had a lot of moving components, such as splitting the line into word and pronunciation objects, removing comments, and using HashMap. But overall, I completed everything to the best of my ability in terms of readability and efficiency (loadDictionary() and getRhymes() worst-case scenario ~600ms). With extra JUnit tests and implementing Javadoc, I believe I should get ~75%.

References

Java HashMap documentation (R1):
https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html

Difference with ArrayList and LinkedList (R2):
https://www.javatpoint.com/difference-between-arraylist-and-linkedlist

Set in Java (R3):
https://www.geeksforgeeks.org/set-in-java/

Asymptotic Notation:
https://www.programiz.com/dsa/asymptotic-notations
https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/

QuillBot(R4):
https://quillbot.com/grammar-check