Callum Stanford-Hoare (cas113)

CS21120 Programming Assignment - Finding Rhymes of English Words

Hand in - 22 November 2024

Task 1: Implementing IPhoneme:

This was very straightforward as my first run of the tests achieved 7/9 passes. This was quickly resolved by putting in more conditions to check for null values which solved the issue, stress free.

Task 2: Implementing IPronunciation:

This section gave the obstacle of trying to implement the findFinalStressedVowelIndex() function, however after much planning of my implementation it was easy to put that plan into motion. I ultimately decided to opt for a backwards iteration of the list meaning that if 1 was found it would be able to stop immediately or it would have to keep traversing given no 1. Originally I aimed to implement a switch case but realized I needed a flag to disable looking for 2s as it would overwrite the position of the “last” secondary stressed vowel.

On the topic of time complexities, the average case time complexity would be O(n) for the add method, using online resources [1] as we must add all elements to our arraylist. And for the findFinalStressedVowelIndex() we must assume it to be O(n) as it entirely depends on the inclusion of a case of 1 which is the only case where it would stop early.

Task 3: Implementing IWord:

Task 3, took much planning as the wording confused my approach by a lot. I originally thought we were concatenating elements to a string but in reading the return types of the

getPronuncations() function I soon realized the task was to implement a collection, thus I choose a HashSet as upon reading the cases, the addSamePronunciationTwice read that our additions had to be unique. Thus I chose a hash set and when adding to our HashSet the average time should be a complexity O(1)[2].

Task 4: Implementing IDictionary:

Perhaps the hardest section thus far, section 4 gave much struggle around parseDictionaryLine. I began implementation revolving around the idea of using a HashMap to allow us instant retrieval of values plus the added benefit of ensuring that our values are entirely unique. The function in question however was hard to implement due to trying to understand how lines would be parsed. The original basic implementation to pass the two first tests had much difficulty indexing arrays which values are held in, I also received IllegalArgumentException due to passing in the wrong values when creating phonemes. Eventually however the solution was produced where it would take in a line, effectively strip away text we don't need then iterating for the addition of phonemes to be added to a pronunciation list to then be added to the word in the dictionary. The average and worst time complexities[3]:

1. Storing a word - O(1)
2. Getting a word - O(1)
3. Getting pronunciation count - O(n)
4. Loading dictionary - O(n)

Task 5: Rhyming:

Perhaps the task that provided the most challenge, task 5 only had its upsides in optimizing the code. Reaching that position on the other hand proved tumultuous. For starters the first method to implement - rhymesWith relied on a try catch to begin with where my original implementation was sloppy as it would cause IndexOutOfBounds exceptions whenever it compared each letter to one another. Say “orange” and “flor” both would have the same pattern of an “or” but they should not rhyme as there are letters still after it in “orange”. The issue was my code would not stop properly to begin with so it would test ‘a’ against something it couldn’t reach. This caused me to rely on a try catch which worked but upon some online research, was not ideal. However going back to it my implementation is a lot more efficient as it has removed that try catch reliance and actually has boundary testing and I really don’t think words can do it justice to show the obstacle this implementation provided. The getRhymes method has gone through a few iterations to where it is now. Originally it was more straightforward - a triply nested for each loop, each to search through each compartment of the dictionary in order to compare against our selected word. Much of the trouble came from trying to fix the other function but when resolved the tests were taking around 100ms to get through whereas now they are taking in ranges between 30ms to 70ms to complete. The main change I made to improve efficiency was to edit the first for each loop by changing it to a parallel stream[4] for the dictionary set. Research online said this was a more efficient method of searching through a map and this led to the improved time difference. Talking about time complexities directly the average time complexity should be (n) as although we are doing multiple for loops we treat them as looking over 1 object at a time. Additionally we don’t know without looking at every word in the dictionary the amount of pronunciations per each word. For all is currently known a majority of

these words have only 1 way to pronounce it thus when we look at it in our function these for each loop equate to doing O(1).

On the theoretical question of trying to calculate which word has the most rhymes, the fundamental idea is that you are running getRhymes each time and then comparing the returned sets size to another set, whichever is larger is the word with more rhymes. To be smarter about it we wouldn't need to check any of the words in those sets as those words rhyme with. Perhaps a map implementation where we store a word and the amount of rhyming words it possesses/so a <String,int> . When we look at our first word we will add each of its rhyming words to this map with the amount of words being the same. When we check the next word in our dictionary we can just do a simple contains(word) from this map to check if it has been checked and skip if it already has. Once the search has been done it is a simple case of iterating through our map and seeing which word has the highest int and saving that entry as our word with the most rhymes.

Arguably we want the **words** with the most rhymes meaning we need the collection of words that word rhymes with so we just grab the set from the dictionary again using that word and said set is our list of words with the highest amount of rhymes since they all rhyme to each other. Time complexity in the worst case would be O(n) due to the need to go over most of the words in the dictionary and the following operations only amount to complexities of O(1) so they have no effect.

Self-evaluation:

Overall I found that the earlier tasks were easier than the latter due to the more rudimentary implementations of interfaces correlative to the environments set up in the practicals. Compared to the earlier tasks, task 4 and 5 provided more of a challenge due to their

open-ended interpretations of how to make an efficient solution to their provided problems. The main difficulty was therefore trying to make my solution as efficient as possible in keeping the time down to process these tests. This all relied on choosing the right data structures and implementing efficient checks. I firmly believe my solution and report is deserving of a mark of around 80%.

References

1. baeldung (2019). *Time Complexity of Java Collections | Baeldung*. [online] Baeldung. Available at: https://www.baeldung.com/java-collections-complexity.
2. Stack Overflow. (n.d.). *collections - What is the time complexity performance of HashSet.contains() in Java?* [online] Available at: https://stackoverflow.com/questions/25247854/what-is-the-time-complexity-performance-of-hashset-contains-in-java.
3. Time complexity of maps (2024). *Mastering High Performance with Kotlin*. [online] O'Reilly Online Learning. Available at: https://www.oreilly.com/library/view/mastering-high-performance/9781788996648/43e3171a-5332-4f64-b1ab-10ee6214bba5.xhtml [Accessed 11 Nov. 2024].
4. Strmecki, D. (2021). *When to Use a Parallel Stream in Java | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com/java-when-to-use-parallel-stream.