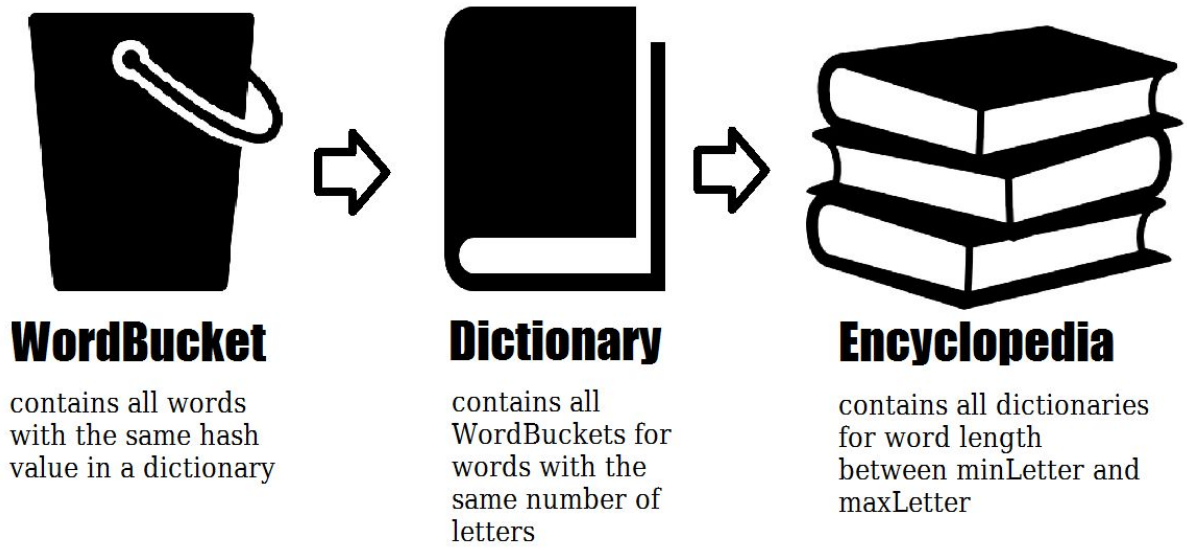
Lab Report - Exercise 12: Scrabble Cheater Deluxe
2016-07-05

Simon Fecke (s0544318)
Julius Deckert (s0545190)
Metawee Langka (s0551423)

Lab Report

1. Making a plan and refactoring

Since in this exercise we were supposed to add another layer to the dictionary data structure, which previously had only consisted of two nested `ArrayLists`, we chose to come up with a new plan. Adding a third nested `ArrayList` to the existing project would have left us with an object of `ArrayList<ArrayList<ArrayList<String>>>` which would not only have made it more difficult for us to solve the next exercise, but also would have made things less readable for anyone else looking at our source code.



The class replacing the most inner `ArrayList`, we chose to name `WordBucket`. We probably should have already implemented it in the previous exercise, but it seemed unnecessary at the time. This class is responsible for all words that generate the same hash value and are therefore placed into the dictionary at the same index. This includes, but is not limited to, permutations (or rather anagrams) which obviously have to be placed into the same bucket, as we expect to find all of them when we'll be searching for matching Scrabble tiles later on. All other words that are placed into the same bucket simply do so by accident and fall under the topic of "collision". When searching for matching Scrabble tiles later, they will be sorted out by checking whether they are actually a permutation of said tiles string.

The class replacing the middle `ArrayList` we named `Dictionary`. Since in this exercise, we want to find all words matching any possible combination of the Scrabble tiles drawn from the bag, we need one dictionary for each word length. In general these words have a length between two and seven letters, since two is the smallest word length in the english language and seven is the amount of tiles a Scrabble player has on his bank (except maybe for the last few turns when the bag is empty). We still decided to let the user choose these numbers (`int minLetter` and `int maxLetter` located in the `Encyclopedia` class) so the application remains flexible, instead of hardcoding the values. The class replacing the most outer `ArrayList`, we called `Encyclopedia`, since this contains all the dictionaries. The dictionary for words with length equal to `minLetter` is found at index 0, while the last index, equal to `maxLetters - (minLetters - 1)`, holds the dictionary for the longest words allowed.

2. The interface ScrabbleStorage

Since these three classes (`WordBucket, Dictionary, Encyclopedia`) all need some similar methods, which don't necessarily do the same thing, we decided to have them all implement the same interface, which we called `ScrabbleStorage`. Each of these methods will use the methods of the same name from the other class it is interacting with.

The methods `getCollisionCount()` and `countValuesStored()` retrieve information about the storage that we use later on to output some statistics which helps us improving on our hash function and choosing a sensible capacity.

The core of the whole application however are the methods `add()` and `getPossibleWords()`. The first of which is used to put words read from the file into the storage, and the latter looks for matches for a string of tiles in the storage.

3. The class Permutation

To generate all possible combinations from the Scrabble tiles drawn from the bag, we created a new class called `Permutation`. Its constructor receives a string as parameter and fills a `HashSet<String>` with the possible combinations. To do this, the string is being passed on to the method `findPermutations()` which first adds the string itself to the `HashSet` and then removes each letter one by one, using the `deleteCharAt()` method from class `StringBuilder`, and calls itself recursively for the remaining string, until the minimum word length(2) is reached. This way all possible combinations of letters (we call them permutations in this report) are generated and since we're using a `HashSet` we don't have to worry about duplicates.

Other than that, the class has only a few other methods. The method `size()` returns the size of the `HashSet`, so the number of permutations, while `getValues()` returns all permutations in the `HashSet`. The method `normalize()` (already explained in our last lab report) is used in the constructor before `findPermutations()` is called (which is actually somewhat redundant but keeps things in order) and we overrode the `toString()` method for more convenient output when debugging our program, writing first the original word and then all its (normalized) permutations to the console.

4. The class WordBucket

Actually nothing new, but since we didn't specifically write it for the last exercise we decided to briefly describe it here. Basically it's a wrapper for an `ArrayList<String>` with a few methods to handle it. The method `add()` puts an element into the `ArrayList`, `asList()` returns the complete `ArrayList` and `toArray()` returns the complete list as a string array. Again we overrode the `toString()` method to be output a complete dictionary more conveniently later on. Words are put into the bucket via the interface method `add()` and search results get returned by the interface method `getPossibleWords()`. At this level, this method checks whether the words found are permutations of the tiles string and returns them only if this is true. Furthermore there are methods like `isEmpty()` which will return a boolean value depending on if the bucket is empty or not. The method `isPermutation()` will test two strings of being permutations of each other by normalizing both strings.

5. The class Dictionary

The constructor for class Dictionary has to be called with an integer parameter determining its capacity. (The capacity may never ever change later on since it would impact the hash function.) It will then initialize an array of `WordBuckets` equal to that capacity. It also sets the `valuesStored` variable to zero, which will later be incremented whenever a word is added. This can be done via the interface method `add()`, which at this level will first generate a hash value for the word and then add it to the bucket at the index equal to the hash in the array. To lookup a match the interface method `getPossibleWords()` is used. At this level the method generates the hash value for the tiles string and then calls the `getPossibleWords()` method of the `WordBucket` object found at the index equal to the hash value calculated.

6. The class Encyclopedia

This is the most outer layer of our storages, and its constructor is being called from the main class with all necessary information:

The name for the file that contains the list of words, the values for `minLetter` and `maxLetter` determining the length of words that are allowed and the value for the capacity of each dictionary. *(This is still one thing that could be improved upon. It would be possible to go through the file once and count the number of words that will be stored in the Encyclopedia, divide it by the number of dictionaries and then choose the next highest prime number. But since this would mean our application would start up even slower than it already does, and since we're always using the same two files anyways we decided against it and used fixed values instead.)*

The method `readFile()` reads every word from the list and passes it to the interface method `add()` which checks its length and then passes it to the `add()` method of the corresponding dictionary. The interface method `getPossibleWords()` at this level initializes an `Permutation` object of the tiles string and then, for each permutation generated, calls the `getPossibleWords()` method of the dictionary corresponding to the permutations number of letters.

7. Evaluation

It really helped a lot, that we were the same group members that had worked together on the previous lab the week before, since it allowed us to simply use the same source code as our base for this project. It also made the whole working process, including the allocation of tasks, more smoothly as we were already used to working with each other and everybody knew the code quite well.

This exercise was also really interesting, and quite a lot of fun. We had already realized some elements of the exercise in our basic version last week by accident. So we used the leftover time to refactor the whole program making the source code more readable. The whole lab took us around 11 hours, three of which we spent on the report.

8. Screenshots

Statistics output:

```
Total buckets: 49074
Used buckets: 21066
Empty buckets: 28008
42.92700819171048% of capacity is being used.

Values stored: 49038
Collisions: 18015
36.73681634650679% collision rate.
```

Results for a draw of Scrabble tiles from the bag and a fixed string:

```
The 7 tiles you drew are: 'k' 'h' 'i' 'a' 'r' 'a' 'w'

The possible words for 'khiaraw' are:
hi
haika
hark
wair
aah
air
whir
hair
ka
rah
haar
ark
aha
haw
irk
aria
ah
ai
raw
war
awa
haik
khi
ar
wha
raia
wark
arak
aw
```

```
The possible words for 'kunnygr' are:
gun
rug
nun
nu
gunny
rung
run
guy
runny
yuk
gnu
urn
knur
un
gunk
```