1. A solution overview: use a block diagram an/or an algorithm you used.

**General Algorithm Used:**

Initialize data structures of a HashMap<String, ArrayList<String>> *dictionary*

Initialize an array *testing* of Strings for words to find anagrams for

Initialize an ArrayList<String> for *result*

//build the *dictionary* that maps the sorted version of a string to all other string that match that sorted value; essentially, map the sorted word to all anagrams of that word

for each *word* in the file

let *sortedWord* be the sorted version of *word*

if *sortedWord* is in *dictionary* already as a key

then add *word* to the existing ArrayList *anagrams* of *sortedWord*

else *sortedWord* is not in the *dictionary* as of yet

create a new ArrayList for *anagrams* of *sortedWord*

then add to the *anagrams* list and insert into *dictionary*

//search for the anagrams of words following the preprocessing done in the prior step

for each *entry* in *testing*

let *sortedEntry* be the sorted version of *entry*

index into *dictionary* looking for *entry* as a key and store result in *result*

if *result* is not NULL

print the *result* list out in a formatted manner

exit

**Algorithm for Word Sorting:**

Initialize a NUM\_CHARS variable to be 265, the number of ASCII characters, total

Initialize a *charCounts* array of characters of size NUM\_CHARS

Initialize a StringBuilder *string*

for each *character* in *word*

increment *charCount[character]* by one //this records that we have seen *character*

for *i*=1…NUM\_CHARS

for *j*=1…*charCount[i]*

add the char at *i* to *string*

return *string* as the sorted input

1. State the space and time complexity of your solution.

In this analysis, let *n* be the number of words in the file, let *l* be the length of the longest word in the file, let

**Space Complexity:**

In the word sorting algorithm, the space complexity is O(1) for the variables, and O(1) for the *charCounts* array, since NUM\_CHARS is a fixed variable. Too, the empty *string* will be at most size O(*l*).

In the main algorithm, the HashMap *dictionary* has size of O(*n*), since in the worst case, all words in the file are anagrams of the same word or no words have anagrams other than themselves, so we store O(*n*) entries each case, since in the first case we have one entry with an ArrayList value of O(*n*) entries and in the second case we have O(*n*) entries each with O(1) in their ArrayList’s value. Intermediate cases would have O(*n*) distribution in general. Also, *result* has at most O(*n*) entries. Too, *testing* takes O(*n*) space at most, if we wanted to test all words in the dictionary for anagrams.

Overall, we need O(*1*) for every word in the list upon sorting, so the worst case space complexity total is O(*nl*), since we store every word in the dictionary and O(*nl*) dominates the analysis.

**Worst-Case Time Complexity Analysis:**

In this analysis, I will be looking at the worst case performance of the aforementioned algorithms.

In the worst case, the time it takes to sort a word is O(*l*) for the first for loop, since access into an array and incrementing a value is simply O(1). The time it takes to concatenate the string, assuming adding a character is done in O(1) time is O(*l*), again, since the outer loop iterates at most O(1) times and the inner loop iterates at most O(*l*) times, since we have at most *l* characters in the longest string in the file. As a result, the time complexity overall for the word searching algorithm is O(*l*).

For the general algorithm, we iterate through *n* words and it takes us O(*l*) to sort the word, and O(*1*) to do the conditional statements, create a new ArrayList, and, finally, inserting and accessing elements of a HashMap takes O(1). Overall, the first loop takes O(*nl*) time. The second loop takes, assuming we end up testing each word in the dictionary, *n* iterations doing O(*l*) time to sort, O(1) to index into the *dictionary* and O(*n*) in the worst case to print out all the anagrams, to this loop takes O(*n2*+*nl*) time in the worst case. Depending upon the magnitude of *n* versus *l*, we could end up with either O(*n2*) or O*(nl*) time.

Overall, in the worst case, this algorithm takes either O(*n2*) or O*(nl*) time depending upon the magnitude of *n* versus *l*.

1. A listing of the most important data structures you employed.

Most important data structures:

* HashMap - this data structure is what allowed me to map the sorted strings to all anagrams and quickly look up all anagrams
* ArrayList - this data structure allowed me to store the anagrams
* Array - this data structure allowed me to know which words to test

1. Sample input and outputs (as required above) in the form of a table.

Please note that for running this program, I used powerpuff and the associated dictionary.

|  |  |
| --- | --- |
| **Input** | **Output** |
| Anagrams for plekic: | pickle |
| Anagrams for diapers: | aspired, despair, praised |
| Anagrams for teardrop: | parroted, predator, prorated |
| Anagrams for nameless: | lameness, maleness, salesmen |
| Anagrams for allergy: | gallery, largely, regally |
| Anagrams for deepak: | peaked |
| Anagrams for impressions: | permissions |
| Anagrams for restrain: | retrains, strainer, terrains, trainers |
| Anagrams for calligraphy: | graphically |
| Anagrams for nepal: | panel, penal, plane |
| Anagrams for stale: | least, slate, steal, tales, teals |
| Anagrams for parliaments: | paternalism |
| Anagrams for sucrose: | courses , sources |
| Anagrams for persist: | priests, spriest, sprites, stripes |
| Anagrams for disintegration: | disorientating |

1. A discussion of the entire process.

In this lab, I started with

1. A complete printout of the program(s) you wrote.

**Code:**

import java.io.BufferedReader;

import java.io.IOException;

import java.nio.charset.StandardCharsets;

import java.nio.file.FileSystems;

import java.nio.file.Files;

import java.nio.file.Path;

import java.util.ArrayList;

import java.util.HashMap;

public class Lab05 {

/\* Global variables \*/

public static HashMap<String, ArrayList<String>> dictionary = new HashMap<>();

public static int NUM\_CHARS = 256;

public static ArrayList<String> result = new ArrayList<>();

private static String testing[] = {"plekic", "diapers", "teardrop", "nameless", "allergy", "deepak", "impressions", "restrain", "calligraphy", "nepal", "stale", "parliaments", "sucrose", "persist", "disintegration"};

/\* Main method that runs algorithm \*/

public static void main(String args[]) {

//for all words, read word, sort, and insert into hashmap

readWords();

for(int i=0; i<testing.length; i++) {

//get a word and search for it

result = searchWord(testing[i]);

//print list of anagrams

if (result != null) {

printList(testing[i], result);

}

}

//exit/done

}

/\* Method that prints a row of the output table. \*/

private static void printList(String input, ArrayList<String> anagrams) {

System.out.print("Anagrams for " + input + ":\t\t\t");

for (int i = 0; i < anagrams.size(); i++) {

System.out.print(anagrams.get(i)+ "\t");

}

System.out.println();

}

/\* Method that searches for the word in the dictionary. \*/

public static ArrayList<String> searchWord(String word) {

ArrayList<String> returnList = dictionary.get(sortString(word));

return returnList;

}

/\* Method that reads from the file and loads the dictionary \*/

public static void readWords() {

Path path = FileSystems.getDefault().getPath("/usr/share/dict/", "words");

try {

BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.ISO\_8859\_1);

String line;

String word;

String sorted;

while ((line = reader.readLine()) != null) {

word = line.trim();

sorted = sortString(word);

if (dictionary.containsKey(sorted)) {

ArrayList<String> anagrams = dictionary.get(sorted);

anagrams.add(line);

dictionary.put(sorted, anagrams);

} else {

ArrayList<String> anagrams = new ArrayList<>();

anagrams.add(line);

dictionary.put(sorted, anagrams);

}

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

/\* Method that sorts the strings in alphabetical order. \*/

private static String sortString(String unsortedInput) {

int[] charCounts = new int[NUM\_CHARS]; //array to count the number of characters

StringBuilder sortedInput = new StringBuilder();

for (int i = 0; i < unsortedInput.length(); i++) {

charCounts[(int) unsortedInput.charAt(i)]++;

}

for (int i = 0; i < NUM\_CHARS; i++) {

for (int j = 0; j < charCounts[i]; j++) {

sortedInput.append((char) i);

}

}

return sortedInput.toString();

}

}