<https://hackmd.io/s/r1fxH9TQQ>

# Project Gutenberg Solution

For this project there were many ways to solve one problem. Below are just a very small subset of the approaches that we can take and they are no in way the most efficient or succient solutions.

* A lot of these methods are lazily appending Strings with + . The better approach would be to use a StringBuilder <https://stackoverflow.com/questions/1532461/stringbuilder-vs-string-concatenation-in-tostring-in-java>
* Many of these methods also re-process the text file that contains the novel. This is not efficient. Ideally we will only need to perform steps like converting the .txt file into a string or parsing the novel into separate chapters once. One way to achieve this is with class variables that store these post-processed strings.

#### getTotalNumberOfWords()

For this method, we had to

1. Take a text file as input and convert it to something we would be able to process. One approach is to take the text file and convert the whole thing into a giant string.
2. Once we have a string that represents all the contents of the text file, we can parse it by splitting elements by the space character.

public int getTotalNumberOfWords() {

String text = parseNovelIntoString();

return text.split(" ").length;

}

public static String parseNovelIntoString() {

BufferedReader br = null;

String text = "";

try {

br = new BufferedReader(new FileReader("/path/to/text/file"));

StringBuilder sb = new StringBuilder();

String line = br.readLine();

while (line != null) {

sb.append(line);

sb.append(System.lineSeparator());

line = br.readLine();

}

text = sb.toString();

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

br.close();

} catch (Exception e) {

e.printStackTrace();

}

}

return text;

}

#### getTotalUniqueWords()

Once we are able to get all the words in the text file, we can get the unique number of words by utilizing the HashMap data structure. Every time we see a word that isn’t already in our HashMap, we add it to the HashMap and incrase our uniqueWordCount by 1.

The one tricky part about this problem is that we want don’t want to count words with different capitalizations as different words. For example : ‘The’ and ‘the’ should not count as two separate words. To solve this, we can convert all words to their lowercase equivalent both when we store it and when we’re accessing the HashMap for it.

We also want to make sure that when we are parsing the string, we don’t include punctuation marks. This could wrongly increase the number of words we thought were unique by counting ‘said’ and ‘said,’ as two separate words.

public int getTotalUniqueWords() {

String text = parseNovelIntoString();

String[] words = text.split(" ");

HashMap<String, String> seenWords = new HashMap<>();

int totalNumberOfUniqueWords = 0;

for (String s : words) {

String sanitizedString = s.toLowerCase()

.replaceAll("\"", "")

.replaceAll("\\?", "")

.replaceAll("\\.", "")

.replaceAll("!", "")

.replaceAll(";", "")

.replaceAll(":", "")

if (!seenWords.containsKey(sanitizedString)) {

totalNumberOfUniqueWords++;

seenWords.put(sanitizedString, sanitizedString);

}

}

return totalNumberOfUniqueWords;

}

#### get20MostFrequentWords()

To get the 20 most frequent words, we can utilize use the HashMap data structue we used in getTotalUniqueWords(). For getTotalUniqueWords() we used the word as both the key and the value when storing it in the HashMap. seenWords.put(sanitizedString, sanitizedString) .

Now that we want to also count the number of times we see the word, we should use the word as the key, and numberOfTimesSeen as the value. This way we can keep a record of how many times we see each word.

Once we’ve parsed through the novel and counted the number of times we’ve seen each word, we need to sort our HashMap by value. There are multiple ways to do this sorting. One of the ways is to use a custom Comparator as illustrated below.

public static List<Map.Entry<String, Integer>>

sortByValue(Map<String, Integer> map) {

List<Map.Entry<String, Integer>> sortedEntries =

new ArrayList<>(map.entrySet());

Collections.sort(sortedEntries,

new Comparator<Map.Entry<String, Integer>>() {

@Override

public int compare(Map.Entry<String, Integer> e1,

Map.Entry<String, Integer> e2) {

return e2.getValue().compareTo(e1.getValue());

}

}

);

return sortedEntries;

}

#### get20MostInterestingFrequentWords()

To filter out the most common words in the English language, we can utilize what we’ve done in the last method get20MostFrequentWords() . However, now instead of adding EVERY word we see into the HashMap, we will only add the word if it’s NOT one of the common words.

#### get20LeastFrequentWords()

Taking what we have from get20MostFrequentWords() we can modify how we’re sorting values. In get20MostFrequentWords() we were sorting by descending order. Now we can modify our custom Comparator method to sort by ascending order instead.

Collections.sort(sortedEntries,

new Comparator<Map.Entry<String, Integer>>() {

@Override

public int compare(Map.Entry<String, Integer> e1,

Map.Entry<String, Integer> e2) {

return e1.getValue().compareTo(e2.getValue());

}

}

);

#### getFrequencyOfWord()

Since this method requires us to parse the novel into separate chapters, we have to first implement a method such as parseTextByChapter(String text). This method would take the entire novel and return an array the same size as the number of chapters in the novel. Each element in the array would represent all the text from that chapter. For example, chapters[0] would repesent the entire text from Chapter 1 of the novel.

Based on how your novel was formatted, it could be possible to implement it as such:

public String[] parseTextByChapter(String text) {

return text.split("CHAPTER");

}

Once we have each chapter separated out, we can then split the chapter up by words and count the frequency of each word seen.

int[] getFrequencyOfWord(String word) {

String[] chapters = parseTextByChapter(parseNovelIntoString());

int[] frequencyByChapter = new int[chapters.length];

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

frequencyByChapter[i] =

getFrequencyOfWordInChapter(chapters[i], word);

}

return frequencyByChapter;

}

int getFrequencyOfWordInChapter(String chapter, String word) {

int count = 0;

String[] chapterSplitByWord = chapter.split(" ");

for (String s : chapterSplitbyWord) {

// sanitize(s) turns it to all lower case and removes all punctuation

if (sanitize(s).equals(word)) {

count++;

}

}

return count++;

}

#### getChapterQuoteAppears()

Once each chapter is stored as an element in a String array, we can iterate through the array to see if our quote is found inside the string that represents the chapter.

For many of the novels there were random line breaks \n throughout the text. One way to navigate around this was to remove all the line breaks and spaces so that a chapter is represented as one long string with no breaks.

public static int getChapterQuoteAppears(String s) {

String[] chapters = parseTextByChapter(parseNovelIntoString());

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

if (chapters[i].replaceAll("\n", "").replaceAll(" ", "")

.contains(s.replaceAll("\n", "").replaceAll(" ", ""))) {

return i;

}

}

return -1;

}

#### generateSentence()

To generate a random sentence we followed the simplied algorithm:

1. start sentence with ‘The’
2. Parse through book to look for all instances of ‘the’ and store all the words that comes after it
3. Randomly pick one of the words from step (2)
4. Repeat steps (2) and (3) with different words until we have a total of 20 words in our sentence.

Helper methods such as getRandomNumberInRange() and getAllWordsAfterWord(String word)makes its easier for us to implement generateSentence()

public static String generateSentence() {

String sentenceSoFar = "The";

String lastWordUsed = "The";

for (int i = 0; i < 20; i++) {

List<String> possibleWordsToUse = getAllWordsAfterWord(lastWordUsed);

lastWordUsed = possibleWordsToUse

.get(getRandomNumberInRange(0, possibleWordsToUse.size() - 1));

sentenceSoFar += " " + lastWordUsed;

}

return sentenceSoFar;

}

public static int getRandomNumberInRange(int min, int max) {

return ThreadLocalRandom.current().nextInt(min, max + 1);

}

public static List<String> getAllWordsAfterWord(String word) {

String s = parseNovelIntoString();

String[] splitByWord = s.split(" ");

List<String> wordsAfter = new ArrayList<>();

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

if (splitByWord[i].toLowerCase().equals(word.toLowerCase())) {

wordsAfter.add(splitByWord[i + 1]);

i++;

}

}

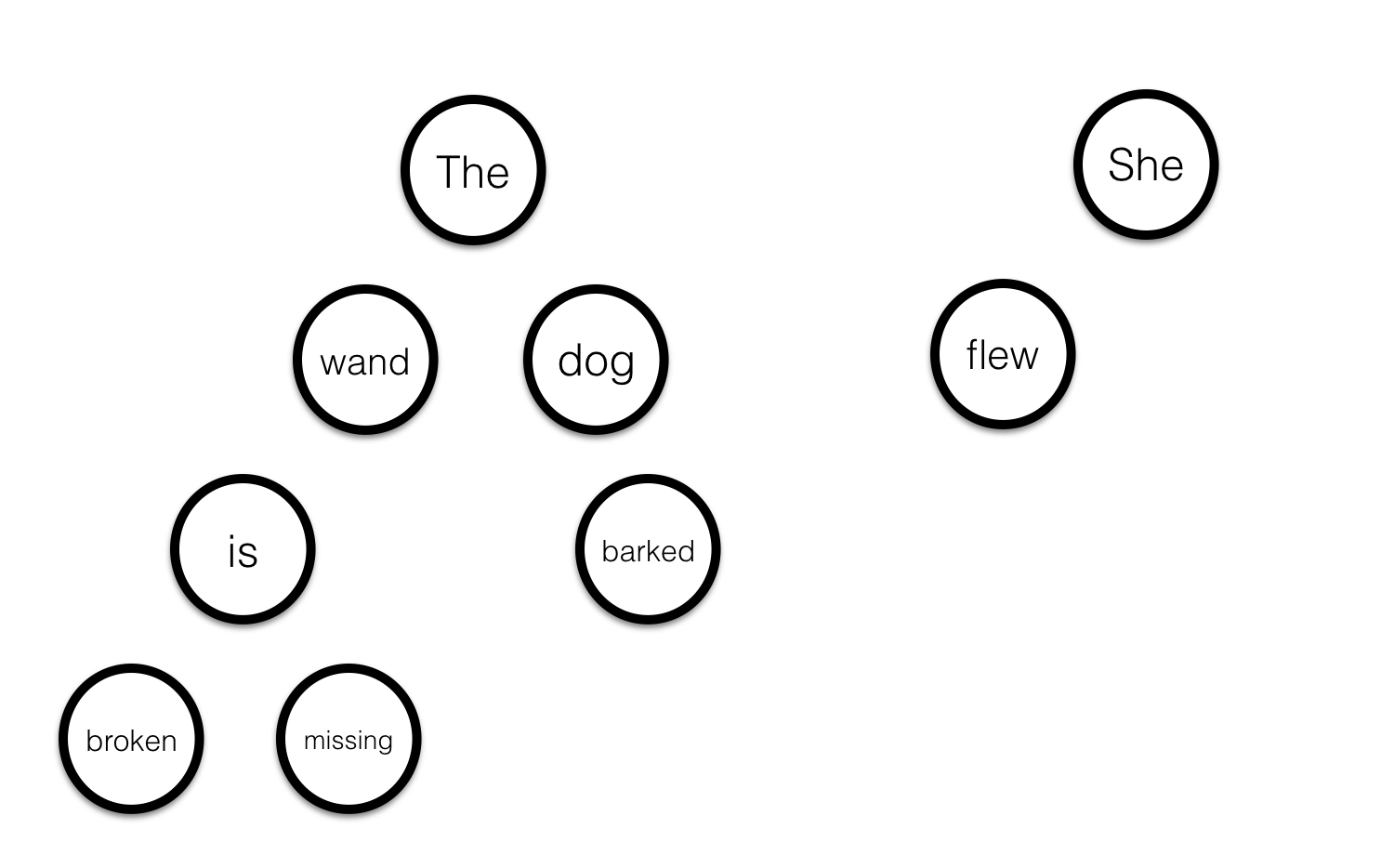
return wordsAfter;

}

#### getAutocompleteSentence()

Implementing autocomplete involved building out tries with the sentences of the book. For each sentence in the book, we should first 1) Iterate through our existing tries to see if there were other sentences that had the same beginning words and then 2) Either create a new trie to store the sentence, or append the difference of the sentence to an existing trie.

For example, if 4 of the sentences of our book was:  
The wand is broken.  
The dog barked.  
She flew.  
The wand is missing.

Our trie would look like:  


#### findClosestMatchingQuote()

One way to approach this problem is to create a method such as getSimilarityOfSentence(String sentence1, String sentence2) . This method would allow us to take two different sentences and get a score of how “similar” they are.

Once we have this method implemented, we can then parse through book and compare each sentence in the book to the quote we want to find. While doing our comparisons, we also save the most similar sentence to our quote that we’ve seen so far, and in the end return the sentence that has the highest similarity.