CMPSCI 187: Programming With Data Structures

Lecture #29: Case Study -- Word Frequencies
David Mix Barrington
19 November 2012

Case Study -- Word Frequencies

- The Word Frequency Problem
- Choosing a Data Structure
- Input and Output in Project #5
- The Algorithm Using BST's
- The `WordFreq` Class
- The `FrequencyList` Class
- Analyzing `FrequencyList`

The Word Frequency Problem

- We conclude the chapter on Binary Search Trees by using them to compute the frequencies of words in a text.

- We define a word to be a sequence of letters and digits, with a delimiter at each end -- a space, line break, or punctuation mark. Our goal is to produce an alphabetical list of the words occurring in the text, with the number of occurrences of each word. We ignore the case of letters throughout.

- We also allow the user to control two parameters of the report. There is a **minimum word size** so that we don't count frequencies of words below that size. And in our report, we only list those words that occur at least some number of times, called the **minimum count to report**.

- Our basic object will contain a word and its count of occurrences (so far).

Choosing a Data Structure

- The basic idea is obvious -- we will read through our text with a `Scanner` object, identifying each word that we see. We need to keep track of each unique word we have seen, each with its count. When we read a word, therefore, we need to determine whether we have seen it before, and if so how many times. We then either create a new word-count pair for a new word, or replace the existing word-count pair with another, incrementing the count.

- If we keep the word-count pairs in a collection, we need to test whether a given word is represented, retrieve its word-count pair if it is, and insert the correct new word-count pair for that word. We also need to output the words from the pairs in the collection in alphabetical order.

- DJW choose to keep the collection as a binary search tree, which can easily be output in order and can be searched quickly as long as it stays balanced.

Using a Priority Queue

- In Project #5, we begin by having you solve the word frequency problem with a priority queue, where the word-frequency pairs are sorted by frequency rather than alphabetically.

- This means that you cannot use DJW's data structure for word-count pairs, because they have a `compareTo` method that compares the strings alphabetically, and you need a `compareTo` method that compares frequencies.

- But you are also required to output your words alphabetically. This means that after reading the entire text, you will need to sort your word-count pairs alphabetically. The simplest way to do this is to convert each of your pairs into a DJW pair, and put them into a second priority queue that keeps them in alphabetical order.

Input and Output in Project #5

- The `FrequencyList` class we're about to see takes input from the console and from a file, and gives its output to the console. The user specifies the two parameters, the program reads the file `words.dat`, and the program sends the report to the console. This fits DJW's application scenario of a tool to be used while developing other text analysis software. But it isn't really suitable for us to grade.

- Your program will be able to take its parameters from either the console or from the **command line**. You may or may not have used the command line before -- it is the reason for the parameter `String [ ] args` in the main method of any Java class. If we say `java MyClass foo bar` rather than just `java MyClass`, for example, the string `foo` becomes `args[0]` and the string `bar` becomes `args[1]`.

- Your program will allow the user to specify a third parameter, the name of the text file to read, and will give its output to a file named `report.dat`.

The Algorithm Using BST's

- So DJW's program has a collection of word-count pairs, and it is about to process the next word in the text file. If the word is too short, of course, we can just ignore it (except for our count of the total number of words). But if it is long enough we must either create a new pair for it with count 1, or replace the current pair with another, incrementing the count.

- The contains method will tell us whether the word is already there, as long as the compareTo method looks only at the strings. The get method is even better, as it returns us a pointer to the pair in question if it is there. The first thing we would think to do is to remove this pair and add a new one with the new count. But we can save significant time by just changing the count of the existing pair, in place. The new count doesn't affect where the pair belongs in the tree. (This won't work with your priority queue in Project #5.)

- It's easy to make a new pair and add it to the tree if this is an entirely new word.

The `WordFreq` Class

- We need to create new objects, increment the count of an object, compare two objects alphabetically by word, and produce a string in a particular format, with the frequency given by a string of exactly five digits.

```
public class WordFreq implements Comparable<WordFreq> {
   private String word;
   private int freq;
   DecimalFormat fmt = new DecimalFormat(“00000”);
   public WordFreq(String newWord) {
      word = newWord; freq = 0;}
   public void inc( ) {freq++;}
   public int compareTo (WordFreq other) {
      return this.word.compareTo(other.word);
   public String toString( ) {
      return (fmt.format(freq + “ “ + word);}
   public String wordIs( ) {return word;} // could be “getWord”
   public int freqIs( ) {return freq;}} // could be “getFreq”
```

The `FrequencyList` Class -- Header and Input

- This code sets up the variables we need and initializes some from the console.

```
public class FrequencyList {
   public static void main (String [ ] args) throws IOException {
      String word;
      WordFreq wordToTry, wordInTree, wordFromTree;
      BinarySearchTree<WordFreq> tree = new BST<WordFreq>( );
      String skip; // skip end of line after reading integer
      int numWords = 0 , numValidWords = 0,
          numValidFreqs = 0, minSize, minFreq, treeSize;
      FileReader fin = new FileReader("words.dat");
      Scanner wordsIn = new Scanner (fin);
      wordsIn.useDelimiter("[^a-zA-Z0-9]);
      Scanner conIn = newScanner(System.in);
      System.out.print("Minimum word size: ");
      minSize = conIn.nextInt( );
      skip = conIn.nextLine( );
      System.out.print("Minimum word frequency: ");
      minFreq = conIn.nextInt( );
      skip = conIn.nextLine( );
```

The `FrequencyList` Class: Processing the Text

```
while (wordsIn.hasNext( )) {
   word = wordsIn.next( ); // delimiter ensures we get a word
   numWords++;
   if (word.length( ) >= minSize) {
      numValidWords++;
      word = word.toLowerCase( );
      wordToTry = new WordFreq(word);
      wordInTree = tree.get(wordToTry);
      if (wordInTree == null) {
         wordToTry.inc( ); tree.add(wordToTry);}
      else wordInTree.inc( );}}
treeSize = tree.reset(BinarySearchTree.INORDER);
```

The `FrequencyList` Class: Output

- Your format in Project #5 will match the format in DJW's *code*, which may or may not match the book text exactly. For example, the string of 17 dashes in the code on page 596 matches to a string of 13 dashes on page 597.

```
S.o.pln (“The words of length “ + minSize + “and above,”);
S.o.pln (“with frequency counts of “ + minFreq + “and above:”);
System.out.println( );
System.out.println(“Freq  Word”);
System.out.println(“----- ----------------”); // 13 on p. 597
for (int count = 1; count <= treeSize; count++) {
   wordFromTree = tree.getNext (BinarySearchTree.INORDER);
   if (wordFromTree.freqIs( ) >= minFreq) {
      numValidFreqs++; System.out.println(wordFromTree);}}
System.out.println( );
S.o.pln (numWords + “ words in the input file.  “); // note spaces
S.o.pln (numValidWords + “ of them are at least “
         + minSize + “ characters.”);
S.o.pln (numValidFreqs + “ of these occur at least “
         + minFreq + “ times.”);
System.out.println(“Program completed.”);}}
```

Analyzing `FrequencyList`

- What is the running time of `FrequencyList`? The first question should be "in terms of what"? Let's let n be the number of characters in the text file. We will spend O(n) time reading the file, but the rest of the processing depends on characteristics of the file -- how many words, how many distinct words, and so on.

- Let's say that there are O(n) words in the file -- since words are usually a constant length, this is probably true. The worst case is actually when they are all distinct, because then we have O(n) nodes in our BST. The unsuccessful searches take O(log n) time each as long as the BST stays balanced, so we have a total time of O(n log n).

- What about the priority queue? We could take O(n) for each insertion into the queue, for a total time of $O(n^2)$. But the priority queue will take advantage of many words occurring often, because we don't have to search far into it to find the most common words. On average with English, our successful searches take about O(log n) time, but this analysis is beyond this course.