CS367 Programming Assignment 3

Lecture 1, Spring 2018

Due by 11:59 pm on Tuesday, March 20, 2018

In this page: Overview | Specifications | Handing in Related pages: Assignments

Overview

Why are we doing this program?

Description

In this assignment you will be writing a Java program that creates a word cloud for a text file. A word cloud is a way to visually represent information in a text file; key words from the text file are displayed with the importance of each word indicated by font size and/or color. For the purposes of this assignment, key words will be any words that show up in the text file that are not in a provided list of words to ignore. The importance of a word will be determined by how many times the word appears in the text file. The word cloud created from the text file will be saved to a webpage. Your program will take four command-line arguments: the name of the input text file, the name of the output file (i.e., the webpage), the maximum number of words to include in the word cloud, and the name of the text file containing the words to ignore.

To construct the word cloud, your program will first go through the text file and determine the key words that appear and how many times each key word shows up. Each key word can be thought of as a (word, # of occurrences) pair. Note that, by definition, the words in the pairs are unique. A Dictionary is an ADT that stores unique key values and provides operations to add and remove information as well as to traverse the key values in order. This makes the Dictionary a useful ADT to store the key word information. For this program, you will implement a Dictionary ADT using a binary search tree. After collecting all the key word information, your program will find the N key words with the most occurrences to include in the word cloud (where N is the maximum number of words to include, specified by the user as a command-line argument). To do this, your program will put all the the key words into a Priority Queue (prioritized by the number of occurrences) and then remove the required number of key words. For this program, you will implement a Priority Queue using an array-based heap.

Goals

The goals of this assignment are to:

- Code classes that implement specified interfaces.
- Gain experience using javadoc documentation to get information about a class.
- Gain more experience dealing with command-line arguments.
- Code a class that implements a `Dictionary` interface using a binary search tree (BST).
- Gain experience using `Comparable` objects and the `compareTo` method.
- Gain experience using a stack to implement an iterator for a binary search tree.
- Code a class that implements a `PriorityQueue` interface using an array-based heap.
- Gain experience reading and generating HTML files.

Specifications

What are the program requirements?

The `KeyWord` Class

The dictionary created from the input file will store `KeyWord` objects, each of which contains a word and a non-negative integer representing the number of times the word occurs in the input file. For the purposes of the `KeyWord` class, a word is a non-empty sequence of characters in which all the letters have been converted to lower-case (we'll add some more restrictions on what we consider to be a word in the main class). The javadoc documentation for `KeyWord` contains the complete details for each method and constructor. Note that the `KeyWord` class implements both the `Comparable<KeyWord>` interface and the `Prioritizable` interface (more below) and that you will need to override the `equals` method inherited from the `Object` class.

The Dictionary

We have specified how dictionaries are to work in the `DictionaryADT` interface (see javadoc documentation, `DictionaryADT.java` source). Note that the `insert` method throws a `DuplicateException` (see javadoc documentation, `DuplicateException.java`).

Total path length

One of the methods in the `DictionaryADT` is `totalPathLength`. The total path length is the sum of the lengths of the paths to each key in the dictionary. This can be used to give us a measure of how many keys must be searched, on average, to find a specific key (by taking the total path length and dividing by the number of keys stored in the dictionary).

For example, if we implement a dictionary using a singly-linked chain of nodes kept in sorted order, then the total path length of a dictionary containing seven keys is:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 = (7 \times 8) / 2 = 28$$

and the average path length is 28 / 7 = 4. If our dictionary containing the seven keys is a full binary tree, then the total path length is:

$$1 + 2 + 2 + 3 + 3 + 3 + 3 = 1 + (2 \times 2) + (4 \times 3) = 17$$

and the average path length is 17 / 7 = 2.42857... In general, a singly-linked chain containing N nodes has a total path length of N × (N + 1) / 2 and an average path length of (N + 1) / 2. For a binary tree, the total path length is the sum of the depths of the nodes (since the depth of each node is the length of the path from the root to that node). This leads us to the following recursive definition for the total path length for a binary tree starting at a node N that is at a depth D:

- The total path length is 0 if N is null.
- The total path length is D if N is a leaf.
- Otherwise, the total path length is D plus the total path lengths of the right and left subtrees (each of which have their root at depth D+1)

The `BSTDictionary` class

In a file, named `BSTDictionary.java`, you will code a class that implements the `DictionaryADT` interface (see `BSTDictionary.java` shell) using a binary search tree of `BSTnodes` (see javadoc documentation, `BSTnode.java`). Note the following:

- Your binary search tree implementation is expected to perform the `insert`, `delete`, and `lookup` operations with a worst-case complexity of O(*height of BST*).
- In order to receive full credit, your implementation of the `totalPathLength` operation must use recursion (by calling a private recursive auxiliary method).
- All necessary comparisons must be done using the `compareTo` method (for the type of key value being stored in the BST).
- You may make use of the code given in the on-line reading on Binary Search Trees (and modify it as needed).

The `BSTDictionaryIterator` class

The `BSTDictionary` class also has an iterator. In a file, named `BSTDictionaryIterator.java`, you will code the iterator (see `BSTDictionaryIterator.java` shell). You need only implement the `hasNext` and `next` methods of Java's `Iterator` interface. Note that the iterator returns the key values in order from smallest to largest. In order to receive full credit:

- Your `BSTDictionaryIterator` class must not use recursion in any of its methods or constructor.
- The constructor must have a worst-case complexity of O(*height of BST*).

Implementation hint: an implementation of the constructor that pushes ***all*** of the nodes in the binary search tree onto a stack in the constructor will not get full credit (since that will have a complexity of O(*N*) where *N* is the number of nodes in the tree). Instead, the constructor should make a stack and only push enough nodes to get to the first item to return when `next()` is called. When `next()` is called, it should get a node off the stack and push any necessary nodes needed so the next time `next()` is called, it will return the next value in order.

Do not change the contents of the `BSTnode.java`, `DictionaryADT.java`, or `DuplicateException.java` source files!

The Priority Queue

We have specified how priority queues are to work in the `PriorityQueueADT` interface (see javadoc documentation, `PriorityQueueADT.java` source). Note that the `getMax` and `removeMax` methods throw `NoSuchElementException` when they are called on an empty priority queue. For this assignment (in order to make the coding simpler), priority queues contain items that implement the `Prioritizable` interface (see javadoc documentation, `Prioritizable.java` source). A class (such as `KeyWord`) that implements

`Prioritizable` must provide a `getPriority` method that returns an integer value representing the priority of an item (where larger values correspond to higher priorities).

The `ArrayHeap` class

In a file, named `ArrayHeap.java`, you will code a class that implements the `PriorityQueueADT` interface (see `ArrayHeap.java` shell) using an array-based implementation of a max heap. In addition to the methods specified in the `PriorityQueueADT` interface, the `ArrayHeap` class provides two constructors: a default (no argument) constructor and a constructor that takes an initial size (an integer) for the underlying array. Your `ArrayHeap` class must compare elements in the heap using the values returned by `getPriority`.

Implementation hint: because the generic type `E` must be something that is `Prioritizable`, to create an array of type `E[]`, use the following (where *`size_of_array`* is the size of the array you are creating):

```
(E[])(new Prioritizable[size_of_array])
```

Do not change the contents of the `Prioritizable.java` or `PriorityQueueADT.java` source files!

The `WordCloudGenerator` Class

The `WordCloudGenerator` class is the main class of the program. The `main` method will do the following:

1. Check whether there are exactly four command-line arguments; if not, display "Four arguments required: inputFileName outputFileName ignoreFileName maxWords" and quit.

2. Check whether input and ignore files (given as command-line arguments) exist and are readable; if not, display "Error: cannot access file *fileName*" where *fileName* is the name of the appropriate file and then quit.

3. Check whether the maxWords command-line argument is a positive integer; if not, display "Error: maxWords must be a positive integer" and quit.

4. Read in the ignore file and create a dictionary of words to ignore.

5. Read in the input text file and create a dictionary of key words for it (leaving out any words listed in the ignore dictionary).

6. Print out information about the dictionary of key words in the following format:

   ```
   # keys: keys
   avg path length: average
   linear avg path: linear
   ```

   where *keys* is the number of keys in the dictionary, *average* is the average path length, i.e., *(total path length)/(# keys)*, and *linear* is the average path length if the underlying data structure is linear (like a chain of linked nodes), i.e., (1 + # keys)/2

7. Put the dictionary of key words into a priority queue.

8. Use the priority queue to create a list of the key words with the most occurrences. The number of key words in the list is the maximum words value passed in as a command-line argument (or the total number of key words if there are fewer key words than the maximum number of words). Since this list will be used to create the word cloud, and the word cloud (in simple display mode) displays words in alphabetical order, the list should be represented using a `DictionaryADT<KeyWord>`.

9. Generate an html page using the list of key words and print the output to the output file given as a command-line argument.

The `WordCloudGenerator.java` file contains the outline of the `WordCloudGenerator` class. Download this file and use it as the starting point for your `WordCloudGenerator` implementation.

Breaking English text into individual words is not as straight-forward as it might seem (for example, just using the `String.split` method to parse text using white-space to identify where words begin and end results in words that contain punctuation, like " or ?). To make things easier, we have provided the code that divides (i.e., parses) `Strings` into individual words. For our purposes, we will consider a *word* to be a non-empty sequence of characters that starts and ends with either a letter or a digit, contains no white-space, and contains at least one letter. The `WordCloudGenerator` class includes a `parseLine` method that takes a `String`, breaks it up into individual words, and returns a list of those words in the order they appear in the `String`. *Do not change the `parseLine` method.*

A method to generate the appropriate html code for your word cloud is provided for you in the `WordCloudGenerator` class. The `generateHtml` method takes as its parameters a dictionary of key words and a `PrintStream` to which to send output. It determines the

minimum and maximum number of occurrences in the given dictionary of key words. It then uses linear interpolation to map the number of occurrences for each key word to the appropriate font size and color.

`WordCloudGenerator` operates in two modes. If `simpleDisplay` is set to true, the generated HTML will place all words in alphabetic order with size and colour denoting word frequency (example here). If `simpleDisplay` is set to false, a more attractive two dimensional cloud will be generated. Popular words are placed in the centre with less popular words at the edges. The layout is targeted toward an 8.5 by 11 format so that the cloud can be easily printed. (example here).

Do not change the `generateHtml` method.

How to proceed

After you have read this program page and given thought to the problem we suggest the following steps:

1. Review these style and commenting standards that are used to evaluate your program's style.

2. You may use the Java programming environment of your choice in CS 367. *However, all programs must compile and run (using the Java 8 SE) for grading.* We recommend that you use Eclipse. You may want to review the Eclipse tutorial to learn the basics.

3. **Download** the following files:
   - `ArrayHeap.java`
   - `BSTnode.java`
   - `BSTDictionary.java`
   - `BSTDictionaryIterator.java`
   - `DictionaryADT.java`
   - `DuplicateException.java`
   - `Prioritizable.java`
   - `PriorityQueueADT.java`
   - `WordCloudGenerator.java`

4. Implement and test your `KeyWord` class, as described above.

5. Implement and test your `BSTDictionary` class, as described above.

6. Implement and test your `BSTDictionaryIterator` class, as described above.

7. Implement and test your `ArrayHeap` class, as described above.

8. Implement your `WordCloudGenerator` class, as described above.

9. Test your main program by developing test files. The main program you will run is in class `WordCloudGenerator`. It is called with four command line parameters: *inputFile*, *outputFile*, *ignoreFile* and *wordCount*. *inputFile* is a text file that will be analyzed to build the word cloud. *outputFile* is an html file that will contain the generated word cloud. *ignoreFile* is a text file containing words to be ignored in building the word cloud. *wordCount* is an integer defining the number of words to include in the word cloud. The following table contains a variety of test files and results. You can compare your results against ours. Note that if a word cloud doesn't contain all of the non-ignored words in a document, it is likely that the word cloud you generate will be slightly different from ours. This is because of slight differences in how priority queues return items with the same priority. You should find that the differences between your word cloud and ours are only with words with the lowest priority (i.e., the words with the smallest font).

Subject of Word Cloud	Input File	Output File (basic)	Output File (advanced)	Ignore File	Size of Word Cloud
First names of Epic students	`names.txt`	`Epic25.html`	`Epic25.pdf`	`emptyIgnore.txt`	25
Mary had a little lamb	`mary.had.txt`	`Mary13.html`	`Mary13.pdf`	`ignore.txt`	13

Who's on first?	on.first.txt	Wh035.html	Wh035.pdf	ignore2.txt	35
On Wisconsin	Wisconsin.txt	Wisconsin35.html	Wisconsin35.pdf	ignore2.txt	35
Merchant of Venice	merchant.txt	Merchant25.html	Merchant25.pdf	ignore.txt	25
Gettysburg Address	gettysburg.txt	Gettysburg35.html	Gettysburg35.pdf	ignore.txt	35
Declaration of Independence	declaration.txt	Declaration30.html	Declaration30.pdf	ignore.txt	30

10. Submit your work for grading.

Handing in

What should be handed in?

Make sure your code follows the style and commenting standards used in CS 302 and CS 367.

Electronically submit the following files to the Program 3 tab on Canvas:

- "`KeyWord.java`" containing your `KeyWord` class,
- "`BSTDictionary.java`" containing your modified `BSTDictionary` class,
- "`BSTDictionaryIterator.java`" containing your modified `BSTDictionaryIterator` class,
- "`ArrayHeap.java`" containing your modified `ArrayHeap` class,
- "`WordCloudGenerator.java`" containing your modified `WordCloudGenerator` class, and
- "`*.java`" only if you implemented additional classes for your program.

Please turn in only the file named above. Extra files clutter up the Dropbox directories.