TCSS 342B Fall 2004

Word Counting Project, version 1.1 writeup.

Project #3, (Homework #6)

Out: Wednesday, November 10, 2004.
Due: Submission in three stages:
Part one due November 17, 10:30 AM; use e-submit to turn-in code.
Part two due November 24, 10:30 AM; use e-submit to turn-in code and written report. Also hand-in written report in class.
Part three due November 29, 10:30 AM; use e-submit to turn-in code and written report. Also hand-in written report in class.

Background and Introduction

Word counting is used in all sorts of applications, including text analysis, creating indexes, and even cryptography. In this programming assignment, you will take a text file and compute what words appear in that file and how many times they appear.

In this assignment, you will use a Map data structure to keep track of words, and to associate each word with an Integer counter. You will update the Map to reflect the occurrence of each string as it is read from a file. If a word is seen for the first time, it is inserted into the map with associated Integer 1. Otherwise, the Integer associated with the word in the Map is increased by one. The dictionary will be implemented in five different ways: as a binary search tree, as an AVL tree, as a red-black tree, as a chaining implementation of a hash table, and as a quadratic probing version of a hash table. You will run the code on input files of different sizes and different contents to see which data structures seem to perform better in practice.

In addition, when the binary search tree and AVL trees are used, you will print an alphabetically first set of words, as well as which words appeared in the file most frequently.

Learning Objectives

- Increase your familiarity with binary search trees, AVL trees, red-black trees, and hash tables. (In particular, you should know how to write a method that recursively traverse a binary search tree.)
- Learn how to conduct experiments to compare the performance of data structures and their implementations.
- Understand how different inputs can affect performance.
- Reading and modifying other people's code.

Starter Files

Much of the code is provided for you. Portions of the starter code was written by a variety of people, including: the authors of our java software structures book, Mark Allen Wiess, Donald Chinn, and myself (Ed Hong). The code is located in the wordcount.zip file on the website.

Here is a breakdown of the provided files, along with some explanatory comments:

- BinaryTreeNode.java
  BinaryTree.java
  BinaryTreeADT.java
  BinarySearchTree.java
  BinarySearchTreeADT.java
  ElementNotFoundException.java
  EmptyCollectionException.java
  These files above are based provide by Lewis & Chase as part of your book. I have made the following modifications:
  - There are no more package statements. Stick all files in one directory, and it should compile.
  - The BinarySearchTree no longer stores duplicate items; an add of a duplicate object does not change the tree. Duplicate objects are ignored.
  - Some explanatory comments have been added; you are expected to provide more comments (see Assignment below).
  - Note that BinaryTree.java is incomplete; the inorder iterator needs to be implemented.
- DictionaryApp.java – This is a driver program that parses the command line argument to determine which algorithm is supposed to be used, and what input file is to be used. It also sets up a timer to time how long it takes to read in and insert the strings into the Map. There is a method called getWord, which gets the next sequence of contiguous alphanumeric characters from the input stream. This is what we will consider a word to be for the purposes of this assignment. You should a look at the main method to see how it works. This file is complete as is; the only change you would need to make to this is in lines 177 and 178 to use quadratic probing hash tables; this cannot be done until you create this data structure in part 3 of the assignment.
  Note that DictionaryApp uses the standard Java Map interface, and the implementations TreeMap and HashMap for the red-black tree implementation and the chaining hashtable implementation of the Map. However, DictionaryApp uses the SimpleMap interface for the other data structures. Ideally we would use the same interface for all 5 data structures, so that all our timing results would measure the difference between the data strucutres, as opposed to the difference in the interface. Since implementing the full java Map interface is prohibitively time-consuming and not that instructive, DictionaryApp uses a simplified Map interface (SimpleMap) for the data structures that are not part of the Java Collections API. SimpleMap only specifies those methods from the Map interface that DictionaryApp actually needs.
- SimpleMap.java – This is a simplification of the standard Java Map interface; this interface has just two methods: a put method and a get method. These two methods are similar to those in the Map interface. The implementations of binary search trees, avl trees, and quadratic probing hashing will use this SimpleMap interface to connect with the DictionaryApp program.
- BinarySearchTreeMap.java – This is an implementation of the SimpleMap interface that uses a BinarySearchTree. It stores KeyValuePairs in the BinarySearchTree. This file is incomplete; you are expected to complete it.
- AvlNode.java
  AvlTree.java
  AvlTreeMap.java – These files implement the Avl Trees. Avl Trees are implemented as an extension of binary search trees; thus all previous operations on binary trees and on binary search trees should also automatically work on avl trees. AvlNode extends BinaryTreeNode to include a height instance variable. AvlTree extends BinarySearchTree and is a very incomplete implementation. AvlTreeMap extends BinarySearchTreeMap to use AvlTrees instead of BinarySearchTrees; it should be complete enough to use once AvlTree is working correctly.

- king_james_bible.txt
  shortkjv.txt
  a-sorted.txt
  b-sorted.txt
  f-sorted.txt
  s-sorted.txt
  w-sorted.txt

  These are some sample files in **Unicode Big-Endian** format that can be used as input into DictionaryApp. shortkjv is just the first few lines of king_james_bible.txt. The King James Bible text was obtained from Project Gutenburg, whose website is http://promo.net/pg . You might find that website a good source of text. Web pages (saved as text files) and source code are other possible sources of text. Note that **DictionaryApp requires text files in Unicode Big-Endian** format, and will not work on standard ANSI text files (in ASCII code). Standard ANSI text files can be easily converted to Unicode Big-Endian format using TextPad or NotePad; use the Save As command, and select a different format. On the Mac, TextEdit can save text files into Unicode format; UTF-16 is the appropriate format.
- At some point, some starter code for hashing with quadratic probing may be provided, keep checking your email and the website.

The Assignment

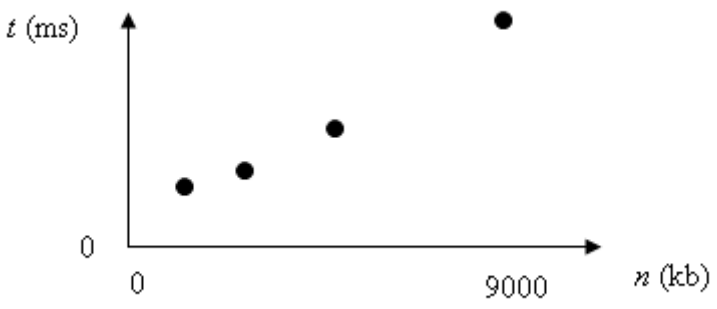
Here is a step-by-step list of things you need to do for this assignment, in the approximate order that you should do them:
Part One:

- Put all the starter files in the same directory.
- Understand all the code implementing binary search trees.
- In BinaryTree.java, implement iteratorInOrder().
- Add additional comments to BinarySearchTree.java so that it is more understandable. As a guide, note the comments already added to removeElement. Short methods (10 or less lines) need no comments beside the header comment at the beginning of the method. Longer methods should have some comments, but you do not need to go into extreme detail.
- Understand the Map and the SimpleMap interface and how it is used by DictionaryApp.
- Understand how BinarySearchTreeMap implements SimpleMap.
- Finish the implementation of get and put in BinarySearchTreeMap.java. To implement put, use a find and update the Object you found; do not use the binary search tree remove method. This is so that our AVL trees will work with the SimpleMap even without an AVL remove method. Note that because BinarySearchTree is defined to throw exceptions, you will have to catch exceptions in your implementation. After this step, DictionaryApp should work with the –b option, although the printing out of words may not work yet.
- Implement printFirst10() in BinarySearchTreeMap.java. (This will require a working iteratorInOrder implementation). This method should print out the first 10 KeyValuePairs, (in order by Key), along with their associated values.
- Finish the implementation of BinarySearchTreeMap.java by writing print10MostFrequent(). print10MostFrequent should print out to the terminal window the ten strings (along with their counts) that have the highest counts; it should assume that the values stored in the KeyValuePairs were all Integer objects. If there is a tie for tenth place, you need only print out one of them. You should design your code so that if we wanted to print out the 50 most frequent words, it would be easy to change your code to do so without having very many lines of code to add. After this step, DictionaryApp should be printing both the first 10 and the 10 most frequent words and frequency counts out to the terminal, when it is run with the –b option.

- For testing purposes, be sure to use only text files in Unicode Big Endian format; otherwise getWord method will not read the words properly. (You can check that it is the right format by running DictionaryApp on the binary search tree. If the file was not in the right format, the strings that are printed out won't look right.)
- Although part one is not due until Wednesday Nov. 17, you should be able to finish it by Monday Nov. 15. Since part two will take longer than part one, you should start working on part two before part one is due. Otherwise you may not have enough time to complete part two.

Part two:

- Complete AvlTree.java so that the AVL tree implementation works. AvlTree.java is the only file you need to modify; the rest of the files should make it work with Dictionary.app. You only need to implement insertion into the Avl tree. You **do not need to implement remove** from the tree (although no one is stopping you from doing so…). Because there is no parent node pointer in our trees, the easiest method of implementing insertion is to make a recursive subcall that return the new root node of the subtree, and have the calling function assign the new root node to the appropriate variable. This is the same coding style as was in the recursive addElement method for binary search trees in the slides. Also, it is only the left rotation and the right rotation needs to do significant work in reassigning links. For left-right and right-left rotations, you can make calls to the left rotation and right rotation methods with the appropriate parameters, and it should all work.
- Run the program using all three algorithms on three different kinds of input files: (1) sorted input, (2) standard English text, and (3) a type of text of your choosing (some examples are: poetry, Java code from any source). Also, run the algorithms on texts of different sizes (enough range to be able to plot the results on a graph to detect any trends in performance). For each algorithm/input file pair, run the program three times and take the average. Be sure the files you have are in **Unicode big endian format**.
- Write a short report describing what types of files you performed your experiments on and what conclusions you made. More report details are given in the turn-in section below. **Your report should include several graphs, where the x-axis is the size of the file being used as input (label each point by its filename) and the y-axis is the running time.** For each type of input (English text, sorted words, Java code, etc.), you should plot the running times of all three algorithms on the same graph so that you can see which performs better. (Thought question: what other quantities might be appropriate to be used on the x-axis instead of file size? Why?) For example, an AVL tree on the sorted files might look like this:



- You will have a set of points for each data structure on the graph. For a given set of related files, graph the running times of different data structures on the same graph so that it is easy to make a direct comparison of their performance.

Part three:

- Make DictionaryApp work with a Quadratic Probing implementation of a Hash table. Some starter code may be provided later.
- Repeat the above process for part two, this time focusing on the two additional implementations, Java's hash table, and your quadratic probing hash table. This means you should be timing the two hash table implementations on the texts you chose before, you should be creating more graphs, and you should be describing some additional conclusions, including ones about how well hash tables perform compared to trees.

What to Turn In

For each part, you should e-submit all the code you have so far. It will be easiest to put all your files in one directory, and then zip them all up (using a program like WinZip). Then you can e-submit just one .zip file. Whenever a written report is due, you should e-submit the report **and** print it out and turn it in during class. To save space, you **should not e-submit** the King James Bible nor any of the sorted files originally given as sample input. You should, however, e-submit the text files you have chosen (in Unicode big Endian format). Try not to choose too many text files that are more than 1 MB in size.

For part one: e-submit your code. You should have added comments to BinarySearchTree.java, and your DictionaryApp program should correctly work on BinarySearchTrees (with the –b option).

For part two: e-submit your code and your chosen text files. Your implementation should now correctly work with AVL trees. Also e-submit your report, and turn in a hard copy of the report in class. Some more report requirements are given below.

For part three: e-submit your code and your report, and turn in a hard copy of the report in class. See report requirements below.

Your reports should describe the results of your experiments. Each report should include (at least) three graphs, one for each type of text input (English text, sorted input, and the type of your choice). The King James Bible should be one of the English text files you use. The three graphs will compare the insertion times of the different data structures on different text size. See "The Assignment" section for a description of how the graphs should be plotted.

For your report, here are some questions you should answer; feel free to comment on more:

1. For each of the files you ran tests on, what were the ten most frequent words that appeared in the file? What were ten first words in alphabetical order? Do *not* report these results for the already sorted files.

2. How long does it take just to read in the King James Bible file without doing any kind of insertions into a data structure? (Hint: Use the –x option. The –x option is just a dummy method.)

3. How do the algorithms perform as the size of the text and the size of the trees grows? (Note that the text size might be large, but the trees might be small if there are a lot of repeated strings.)

4. On the worst case input for BST (a file of sorted words, for example), about how big does the tree have to get before the AVL tree implementation performs better?

5. For normal text inputs (for example, fiction, web pages, etc.), do the most frequently used words tend to be the same words?

6. Are there any differences in word frequencies from different input files (for example, English text versus Java code)?

7. Are there any other patterns or phenomena you observed?

Software Engineering Tips

First read through the code, and make sure you understand it. Look through DictionaryApp.java to see what it is doing to help you conduct your experiments. Write the easy code first and then the harder code (PrintMostFrequent). **Most importantly**, *understand* what the code is doing before modifying any of it.

When it comes time to do experiments, think about how many files and what kinds of files you want to test. Don't just run the program on a bunch of random files and draw conclusions from that. Be scientific about your experiments by deciding *beforehand* what kinds of things you are looking for and then collecting text files of a particular kind.

Evaluation

As usual, you will be graded on two things: the *correctness* of your program (i.e., how well you followed directions, and how well your program does what we asked), and on the *clarity* of your program (i.e., how readable it is, and how well it communicates the intent of what you were trying to accomplish to the human reader). Clarity will benefit from good explanatory comments, good choice of variable names, good use of local names (if needed), and good use of indentation to group things.

You will also be evaluated on the quality of your **test cases** (how systematic you were in choosing your test files) and the quality of your **analysis** (graphs, observations).