University of North Carolina at Greensboro
CSC 330: Advanced Data Structures
Prof. Stephen R. Tate

Handout 4
September 2, 2009

Assignment 1 — Using Binary Search Trees

Due: Monday, September 14

Objectives: This assignment has two objectives: First, to get students started using and programming in the Unix environment, and second, to experiment with lookup tables based on lists and trees and consider performance under various scenarios.

High-Level Description: You will write a program that reads a large text file, and keeps track of how many times each word occurs in the file using a list and a binary search tree. You will measure the time required by both implementations on several different inputs, and describe what affects the performance in each case. The programming for this assignment is not terribly deep or complicated, but you will working on a new system and you should not underestimate the challenges that this can raise — do *not* wait until the last minute to start this assignment!

Details: For this assignment, you will use the binary search tree implementations from the book, as well as files supplied specifically for this assignment. Unfortunately, the code from the book is not compatible with the most modern compilers (such as the one on the `cmpunix` machine that you'll be using!), so do not download the code from the textbook publisher or authors. Instead, go to the assignments link on the class web page, and follow that to download corrected versions. Alternatively, you can copy the code directly on the `cmpunix` system from directory `~srtate/330/assign1`.

Your program will need to keep track of pairs consisting of a word and an integer count. In order to do this, you should define a class for these pairs, with a constructor and methods to retrieve the current count, increment the count, get the word, etc. You should also provide operator overloads for the `==` and `<` comparison operators. I'll call this "`pairClass`" in this handout, but you can feel free to name it however you'd like.

You will code up two alternative ways of keeping track of these pairs: using a list (use the STL `list` class) and a binary search tree (use the `stree` class from the book with the provided code). To see how to do this, look through the main file, `wordreader.cpp`: it reads a word at a time from standard input and then calls a function `findAndInc()` for each word it reads. This function should look for the word in your data structure, and if it finds the word it should increment the count for that word — if it doesn't find the word, then it's a new word and should be added to the data structure (with count 1). Since you will be using both a list and a tree to keep track of the words, you will need to write *two* implementations of this function with slightly different function signatures:

```
void findAndInc(stree<pairClass> &allWords, const string &word)
void findAndInc(list<pairClass> &allWords, const string &word)
```

By doing it this way, you can simply change the declaration of your data structure on the first line of the `main` function, and when you recompile the compiler will automatically link that call to the correct version of the `findAndInc()` function.

Finally, notice that wordreader.cpp calls a `processWords()` function (again, one of two functions, depending on the type of your data structure). You should use this function to print out the words in your data structure along with the count of each one — see the sample output below for an example. The words should be sorted, so for your list implementation you may want to use the `sort()` method provided by the STL `list` class before printing.

Testing and Analysis: When you are writing and debugging your code, you will probably want to use a small input file, such as the one in the "sample input" below. For testing with text from files (rather than entered from the keyboard) you should use input redirection. Once you are convinced that your code works correctly, you will use some large text files provided in directory `~srtate/data/assign1` for testing. There are three files, `test1.txt`, `test2.txt`, and `test3.txt`, and all three consist of the full text of *The Count of Monte Cristo*, plus information at the beginning that varies by test case.

You should run all three test files through your program using the list data structure, and then all three using the tree data structure. So that you are really timing just the data structure operations, you can comment out the call to `processWords()` in the main program. Use the Unix `time` command to time your program. For example, you might run the program like this:

```
time ./wordreader <~srtate/data/assign1/test1.txt
```

This will print 3 different times — the one you want to report is the "user" time, which reflects the amount of time your program actually spent executing on the CPU. Feel free to adjust the compiler flags in the `Makefile` to experiment with optimization settings (e.g., using "`-O4`" when compiling), but make sure you use consistent flags when comparing implementations.

Finally, look at the test data files, and think about why the different implementations performed as they did for the different input files. Write a brief report giving a table of your measured times and explaining the performance — explain why the different implementations performed well or didn't perform well on different inputs. It is important that you are as precise in your descriptions as possible, and use terminology and notation that is standard in computer science. For example, you should be referring to concepts "worst case," "best case," and "average case," and should be describing performance in terms of asymptotic notation ("big-oh notation", although if you want to really be precise you should use better notations such as "theta" notation).

To Turn In: Use the `330submit` program (see Handout 3) in order to turn in your code, using assignment name `assign1`. You should submit every file that you have created or modified (you can also include the `.h` files that were provided to you, but this isn't necessary — they should *not* be changed!). On the due date, you should turn in a printout of your code along with your report giving your performance analysis.

Sample Input/Output:

SAMPLE INPUT

```
Peter Piper picked a peck of pickled peppers,
A peck of pickled pepers Peter Piper picked;
If Peter Piper picked a peck of pickled peppers,
Where’s the peck of pickled peppers Peter Piper picked?
```

SAMPLE OUTPUT

```
a: 3
if: 1
of: 4
peck: 4
pepers: 1
peppers: 3
peter: 4
picked: 4
pickled: 4
piper: 4
the: 1
where’s: 1
```