CSE326, Project 2

Due: Wednesday, July 27

1 Overview

In this project you will be implementing a very basic search engine. The program begins by parsing a collection of text files. Then the user can give single word queries, and the program returns the names of the files that contain the query word. The "front-end", which handles all the file IO and parsing, is written for you. Your job is to implement the "back-end" data structures. The back-end is a Dictionary ADT that maps words to a list of files containing that word. A quick and dirty implementation based on an unsorted array is included. You will add the following implementations of the Dictionary ADT:

1. an unbalanced binary search tree,
2. an AVL tree, and
3. a splay tree.

As in project 1, you will collect data on on how the performance of the algorithms scales with input size. Work in pairs and turn in one write up.

2 Provided Code

The code can be downloaded from www/326/projects/proj2.tar. To compile the provided code, run

```
gcc -o search *.c.
```

The first argument to `search` is the implementation to use. This is one of `-array, -unbalTree, -avlTree, -splayTree`. All other arguments should be text files to parse. For example, without any modifications you should be able to run

```
search -array *.c
```

to search through the code. After parsing the program waits for user input. Type in single words, followed by return. To quit, enter a single period on a line. When testing performance you can run many searches at once by putting words in a text file, one per line, and piping to file to the program by

```
search -array testFiles/* < testQueries.
```

The front-end code is in search.c. The important methods are main(), which parses the command arguments and handles user input, and parseFile() which reads a file into the dictionary. For each word in the file, parseFile() inserts the word as a key, with the file name as a value. To find all files that contained a given word, you just do a find on that word. To remain implementation independent, the functions newDictionary(), insert(), and find() serve as wrappers for the particular implementation. The unsorted array implementation is in array.c. Stubs for the three tree types are in the other three files.

The dictionary is slightly complicated in that the value stored is a list of strings (e.g. the names of files that contained the key word.) When doing an insert you don't want to overwrite the value, you want to add a name to the list, taking care not to allow duplicates. The list is implemented as a basic linked list using the StringList structure. See insert_array() for an example.

3 Requirements

- Fill in the new(), insert(), and find() functions for each of the three tree types. You will need to define a node structure for each of the tree types. See array.c for an example of defining a structure in C.

- Fill in the test methods. You should give enough test cases to convince yourself and the grader that your data structure is working correctly. Writing a print routine and printing the data structure after a few insertions is a good way to do this.

- Design and carry out a performance study. For source documents see http://www.gutenberg.org. There you can download large text files, such as the complete works of Shakespeare, or the Bible. Download a document corpus large enough to show performance differences between the various algorithms. Graph the performance of the 4 algorithms for various corpus sizes. Show insertion and lookup performance separately. You can use the Unix command `wc` to measure the document size. Turn in a write up as in project 1.

4 Hints

- The provided code is brand new, and will most likely contain a bug or two. If you find something wrong email me, or just email the class if you know how to fix it.

- Memory management in C is tricky, especially if you're used to languages with automatic memory management like Java. Pay very close attention to how you use pointers. One example: when a key is passed to insert(), the string needs to be copied, not just the pointer. This is because the string is a local variable in the calling method, and won't exist once the method returns. This is also true for the file names, but we can be sloppy here and just copy the pointer, since we know the file names came from the command line arguments and aren't going to go away. I'd recommend understanding the details of insert_array() before writing your own code.
- Debuggers are very useful, especially in C, where a crash returns no information without a debugger. To debug, use the -g option with gcc, then run the program with `gdb search`. From there type 'run' and the arguments to the program.
- To get valid timing measurements you need to do as little I/O as possible. I'd recommend commenting out all the printf() calls when testing performance.
- The unbalanced binary tree is the easiest tree. Get it working completely before you start on the others. Code for all the trees is in Weiss. Your code will be somewhat different, but if its looking significantly more complicated than the code in Weiss, you're probably doing something wrong.

5 Extra Credit

Have fun with this project. There's no end to the enhancements that could be made. A small amount of extra credit will be given for extra features. Be sure to clearly explain what you enhancement is and demonstrate its use in your write up. Some ideas:

- A real search engine would perform *stemming*. This is the process of removing word endings so that for example, "house", "housing", and "houses", all reduce to "hous". A Google search for Porter Stemming will give you some ideas on how to do this.
- Documents that contain a word many times should be ranked higher. Modify the file list to also maintain the number of times a word occurred in the document, and return the file list in ranked order.
- Allow multiple word queries. You can implement logical OR by doing a find on each query word and merging the lists. A logical AND is similar, but you only keep files which were in all the lists.
- **Difficult:** A real web search engine gets its pages by *crawling*. This is a process of finding URLs in documents you already have and downloading those pages. You could do a simple version of this by looking for href tags,

or the word http, and trying to parse the URL. You can download the page by calling the command line tool w3c (see http://www.w3c.org/ComLine). The process continues recursively on the newly downloaded pages.