Assignment 3 – XD Report Template

Your Name

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

This programs function is to read from standard in, analyse the file, and return the number of unique lines in the file, followed by a newline. However, while this is the main function of the program, most of the code is oriented to build an array of linked lists, to help facilitate this unique line checking process.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- In Part I, you implemented garbage collection—routines that clean up dynamically-allocated memory. How did you make sure the memory was all cleaned up? How did you check?

  The use of valgrind to check if my coded free() statements were working was the main source of help when it came to checking for memory leaks, next to rubber duck debugging. If there were leaks valgrind would complain and I would fix it, and if not I knew I was good to go.

- In Part II, you made a major optimization to the linked list optimization. What was it, and why do you think it changed the performance of bench1 so much?

  Originally, when a new node was added to the linked list, it would be added at the end of the list. I optimised it by having the nodes be inserted at the beginning of the list, which avoided the issue of exponential adding times all together. If you keep adding nodes at the end of the list, it will take longer and longer to find it each pass, which follows an exponential increase for time it takes to add a node.

- In Part III, you implemented hash tables. What happens to the performance of bench2 as you vary the number of buckets in your hash table? How many buckets did you ultimately choose to use?

  As you increase the number of buckets, the program requires less time to process each iteration, so it goes faster. But, if you want to add buckets you cant be lazy, and there are also some diminishing returns to consider. If you had 1000 buckets with one node in each of them, that would be inefficient. I personally went with 2 buckets because it was easy to evenly sort keys between the 2, and it led to less repetitive coding down the line, only having t define and manipulate two things.

- How did you make sure that your code was free from bugs? What did you test, and how did you test it? In particular, how did you create inputs and check the output of uniqq?

  I programmed this code in stages, checking to make sure each stage was operational before moving onto the next. If there were bugs, I would analyse the code and fix them, and then go on my merry way. I also checked using valgrind, and scan-build. For uniqq, since songs have a good mix of repeating and non repeating lines, I will make 2 text files of 2 different songs, and test that against uniqq. Songs are also good since thy are short and you can figure out on your own how many unique lines there are.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays.

Ill probably just make a bunch of random text files, that are kind of short. by the time I get to uniqq the program will be pretty fast so there is not much sense in making files that contain, lets say, the entire script for the bee movie. Then Ill run those through uniqq and see what I get, keeping in mind lines I know are not repeated. if I get repeated lines, or do not get unique lines where I should, Ill look for issues. If uniqq is fully functional, then this proves all of the functions it relies on also work.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

Simply type in ./uniqq on the command line, followed by a file name you want to be analysed. Hit enter and you should get the number of unique lines in the file.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{ lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[2]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

This entire program is broken up in to files that contain chunks of the code. Bench2 is considered the main file at this point. If the user wishes to run the program multiple times, and they input a number like so) ./bench2 5, the program will run 5 times. This is achieved by reading in from argv, converting to a string, and passing that value to the unique words function. Everything having to do with unique words was given to us, so I will not go into detail on it. Then the main hashtable is defined with unique words from before, and we are off to the races. From there hash get (found in hash.c) will call list find (ll.c), which will finally call cmp (item.c). The next thing to be processed are some if statements that make sure this data mining tower previously described works. If these check out, the program prints Success, and terminates.

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

[2] This is my footnote.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

In ll.c you will find functions list create, list add, list find, list remove, and list destroy. ll.h defines the node struct. All of these functions contribute to generating and processing a linked list. Any issues with linked lists can be found here. ll.c depends on ll.h, and ll.h in addition to defining ll.c functions also defines the LL and Node structs, which are key to the linked list implementation. ll.h defines the linked list and node structs, along with the heads of the regular ll.c functions. item.c is in charge of making the cmp function, and item.h defines the item struct. hash.c contains the code for hash create, hash put, hash get, and hash destroy. hash.h defines the hashtable struct.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- list create: takes nothing, returns an LL * pointer. Allocates space for the given linked list in heap memory. If it cannot do that is returns NULL, if not it sets the head of the linked list to null
- list add: takes a pointer to a linked list and an item to put in the list, returns a boolean. Makes a new node, with the item, and inserts the node at the end of the given linked list. returns true of it works, and false if it does not.
- list find: takes a given linked list, and item, and a function cmp. Checks the given linked list if the item can be found within it. returns true of it found it, and false if it did not
- list remove: takes a given linked list, and item, and a function cmp. If the item is found in the linked list it removes it, frees the nodes memory, and returns true. If not, it returns false.
- list destroy: takes a linked list and returns void. frees every node in the linked list and sets the linked list pointer to NULL.
- cmp: takes two items are returns if they are equal (true) or not (false)
- append: takes a character and a character pointer, and returns an int. Appends the indivigual character to the char pointer. Returns 0 on sucsess and 1 on failure.
- list traverse: takes a linked list and returns an int. Goes through the linked list looking at all of the values stores in all of the keys. If the value of a key is 0, unique gets 1 added to it. returns unique. Meant to track repeated lines.
- hash create: takes nothing and returns a hashtable. Used to make a new hashtable
- hash put: takes a hash table, a char as a key, and an int as a value. makes a new node in the hashtable, node made in its respective bucket.
- hash get: Takes a hashtable and a char as a key. returns an int pointer. Looks through the hashtable for the key. Once it finds the key, it returns where the location of its stored value is.
- hash destroy: takes a hash table and returns nothing. Frees the hashtable memory.

https://stackoverflow.com/questions/19891962/c-how-to-append-a-char-to-char

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.