CS 232 Introduction to C and Unix

Project 2: Using a Trie to Store Word Counts

(Due on March 26, 11:59pm)

Note that this project is more substantial than the previous projects. That is why you have longer to work on this project. **Start as soon as possible.** If you get stuck, feel free to come to office hours. I'm here to help you, but I can't help if you don't ask.

READ THE ENTIRE PROJECT DESCRIPTION CAREFULLY BEFORE BEGINNING. THEN READ IT AGAIN! I know there is a lot of information here, but it is here to help you through this project.

Project goals:

This project will test your ability to design a data structure in C and write some functions to manipulate that data. After this project, you should be able to design self-referential structures, manipulate pointers, and use malloc and free appropriately. Furthermore, you will learn how to parse and manipulate strings as you use your data structure to index web pages. This indexing will be used as part of your web search engine in the next project.

Since one of the goals of this course (see the syllabus) is to "work with a team to take a narrative algorithm description and implement the algorithm and supporting data structures," this is a **group project**. You will be working with a partner to complete this project. Your partner will be emailed to you. (You do not get to choose your partner, just as you do not typically get to choose your coworkers at a job.) The final code should go to your own Bitbucket repository. That is, the code from your group will push to both Bitbucket repositories of group members.

Group work:

This might be the first time that some of you have worked on a group programming project, so here are some important instructions for a smooth group project.

- You should respond to your group member's messages (email, phone, text, or otherwise) within 12 to 16 hours. This means you check your email at least once in the morning and once in the evening, replying to any messages you've received from group members. There are two main reasons for this: to have enough time to complete the project, and to be courteous to your team members.
- Get together VERY SOON, in the next two or three days. Even if you think that you can't find a convenient time, you need to make a meeting happen somehow (even if it is a phone or video conference meeting).

Each group will do its work differently. However, here are some guidelines for steps you should take.

- The first thing to do is to make sure you each understand the project.
- Then talk about the overall design you will need: the data structures and functions you will use. Be specific as you are doing this. For example, it is a good idea to code all of your structures and your function declarations.
- After you have settled on the overall design, you can begin coding the function definitions (*i.e.*, the function bodies). You can either divide up the functions and have each person work on half of them, or you can have a group coding session where you take turns at the keyboard and discuss the code as you write it (*e.g.*, pair programming). Most people dislike the idea of a group coding session, and it admittedly takes more time to write the code. However, it can often save time when debugging, because having multiple pairs of eyes on the code will increase the chance of catching mistakes before you compile. Furthermore, if the person doing the typing explains what they are writing, it helps clarify the purpose of each line of code you write and allows the other group member to double-check the code.

You will be doing a group evaluation after this project. This is in keeping with the stated course and project goal of "working with a team to take a narrative algorithm description and implement the algorithm and supporting data structures." Thus, please make sure that the workload was (approximately) well-balanced between the members. This is important, and it will affect your project grade (which can be different for different group members). Both team members have a responsibility to distribute the workload among the team: you should neither contribute nothing to the project nor do everything yourself.

While several of you are more than capable of completing the project yourself, completing the project is not the only goal for this assignment. **Working as a part of a team** is another part of the assignment.

Project description:

When you enter a query in a web search engine, it is not really the web that is searched, but a database of statistics about web pages. Collecting those statistics is called "indexing" the web pages, which is done before the actual searches that people perform. In this project, you will implement a data structure to store how often each word occurs in a document, which is one of the most important statistics to gather about web pages. In the next project, you will use this functionality to index several web pages as part of a simple web search engine.

In Blackboard, please download the file “project2.tar”. Upload this file to your cloud9 workspace (the same Python workspace as for the project 1) under a new subdirectory “project2” under the “project” directory. Use the following command to unzip the tar file:

```
$  tar -xvf project2.tar
```

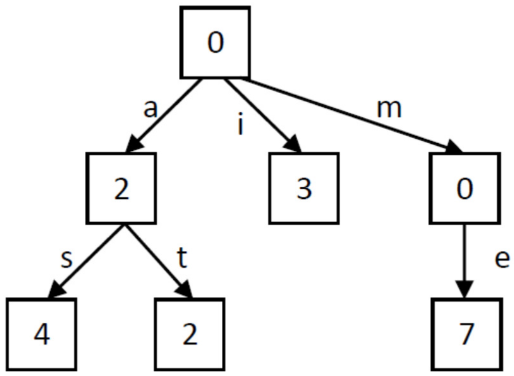
You can find four files:

- indexPage.c: the file that your group will complete and submit (part of the program already written)
- getText.py: the Python file for getting the text content of a web page
- solution: the executable sample solution to project 2
- runTestCases.sh: the shell script to run tests and compare your solution to the sample solution

To run “solution” or “runTestCases.sh”, you will need to change the executable permission of the files by using the “chmod” command.

The trie data structure:
To complete the program, you will create a data structure called a **trie** (pronounced as “try”) to store both the words that appear on a web page and how many times each word occurred. A trie is similar to a tree: each data structure consists of nodes. The starting point in the trie (or tree) is called the root node, which is typically drawn at the top when drawing a trie (or tree). Each node can have zero or more children nodes. If a node has no children, it is called a leaf; otherwise, it is an interior node.

In a *tree*, data items are associated with different *nodes* in the tree, so each node (typically) contains a data item. (See the binary tree example in Section 6.5 of K&R for a particular example of a tree.) In contrast, in a *trie*, the data associated with a node is stored along the entire path from the root to a node. For example, consider the following trie of word counts:



Each link in the trie is labeled with a letter. The letters along the path from the root to a node compose the word associated with that node. For example, the bottom right node is for the word “me”. The count there indicates that the word “me” has been seen 7 times. Note that some words have prefixes that are also words (*e.g.*, the prefix “a” of “as” is a word, which has been seen twice), but some prefixes will have 0 counts (*e.g.*, “m”). Each node can have up to 26 children (one for each letter), but children that are not needed are simply not created. For example, if there are no words that start with z, then the root node will not have a ‘z’ child. If there are no words that start with “aq”, then the ‘a’ child of the root node will have no ‘q’ child. Overall, this trie holds the following word counts: (a, 2), (as, 4), (at, 2), (i, 3), (me, 7).

Trie implementation instructions and hints:

- You are to define and use a self-referential structure for a trie node. Since you are dealing with lower-case words, each node in your trie will have from 0 to 26 children. Your trie node structure will have to keep track of
  - the count for its word
  - and its children (*i.e.*, pointers to its children, and which letter each child is associated with).

  You can also refer to the binary tree in Section 6.5 of K&R to get ideas about how to form the trie.
- You MUST understand the trie data structure and think about what your C structure will contain before you begin coding. **If you just start coding without thinking first, you will end up regretting it.**
- The nodes in the trie should be allocated on the heap using malloc and deallocated using free.
- You will probably want to use recursion in your functions (although you are not required to do so). To think about the recursion, it helps to notice that any node in the trie can be viewed as the root of a sub-trie: the node itself, and its children, and their children, and so on. For instance, in the trie above, the sub-trie starting at the ‘a’ node consists of the ‘a’ node, the ‘as’ node, and the ‘at’ node.

Parsing the web page:

To get the word counts for a web page, you should use the "getText" function (*e.g.*, the function in indexPage.c that uses getText.py). The getText function will take a URL, a buffer, and the buffer size as arguments. It will fill the buffer with the text from the page given by the URL. It will truncate the text, if needed, to avoid overflowing the buffer. Upon return, the buffer is guaranteed to be null-terminated. The getText function returns the number of bytes filled into the buffer. Your program should be able to handle pages up to 300000 characters long, indexing the first 300000 characters of any pages that are longer than that.

Once getText returns a string, part of your job is to parse the string into terms (*i.e.*, words), which you can count using your trie data structure. For this project, we are only dealing with terms that consist of alphabetic characters. Thus, when parsing the string, you should skip all non-alphabetic characters, including whitespace characters (' ', '\t', '\n', '\r'), punctuation, and digits. **Every contiguous substring of alphabetic characters forms a term.** Note that you should convert all alphabetic characters to lower-case, in order to use your trie.

Parsing example:

Suppose getText returns the string

```
" Feb. 27
                      all events at a glance for today
Events
                      calendar ...
&copy; 2010 IPFW | 2101 E. Coliseum Blvd., Fort Wayne, IN 46805
1-866-597-0010
IPFW is an Equal Opportunity/Equal Access University."
```

You should add the following words into your trie for that page:

```
feb
all
events
at
a
glance
for
today
events
calendar
copy
ipfw
e
coliseum
blvd
fort
wayne
in
ipfw
is
an
equal
opportunity
equal
access
university
```

Each addition will increment that term's count by one.

Program requirements:

The program should take a URL as a command-line argument and print out information as it indexes that page. The main function of indexPage.c should be very, very simple. By keeping "main" simple and putting the code in other functions, you will be able to use your code for the next project with few modifications. Specifically, the main function should do three things:

- Call a function that indexes the web page and returns a pointer to your trie
- Call a function that prints out the counts of all the words in the trie
- Call a function to destroy the trie (*i.e.*, free any allocated memory) when it is no longer needed

You should also create a function to add a word occurrence to the trie. It will be useful for the indexing function.

For compatibility, the functions to index the web page and print out the word counts must **use the exact formatting and messages** shown below.

When indexing the web page, print out the web address, followed by each term that you encounter, converted to lower-case, in the order they appear in the web page. Each term should appear on its own line, preceded by a tab. Here is a fictitious example:

```
http://www.cs.ipfw.edu
	computer
	science
	computer
	science
	department
	...
```

When printing out the counts of all the words in the trie, print out one line for each word with non-zero count, in alphabetical order. Do not print out words with 0 counts. To traverse the nodes in alphabetical order, you should print out the count for the root node of your trie (if non-zero), then recurse upon each child in alphabetical order. To keep track of the word associated with the current node, you can use a character buffer as a stack: every time you go down a level in the trie (*i.e.*, make a recursive call), you add the corresponding character to the end of the buffer (*i.e.*, push on the stack). When you go back up a level in the trie (*i.e.*, return from a recursive call), you remove a character from the end of the buffer (*i.e.*, pop off the stack). You should keep track of the buffer length and print an error message if the buffer is too small.

Here is an example of the output for the word counts for the example trie shown above:

```
a: 2
as: 4
at: 2
i: 3
me: 7
```

Use the format string "%s: %d\n" to get this output. Note that the first character on these lines must be a lower-case letter (*i.e.*, no leading spaces).

Additional instructions and hints:

- "TODO:" tags mark several places that you will need to edit the code in the source file, although you will need to make other edits as well.

- You may define other functions if you need them. The functions listed in the provided code are suggestions about how to structure the code, but you can change/add/omit some of them as you see fit.
- You should enclose a URL in quotes when you type it on the command line.
- After you have tested your program on a few web pages, you can run a set of test cases using the shell script "./runTestCases.sh" in your project2 directory. This will run your program on some simple web pages and check its output, notifying you if your program passed or failed the tests. Passing the test cases is a good sign that your program is correct.
- Be careful to avoid memory leaks! You should use the **valgrind** heap checker to make sure you are not leaking memory. Even if your program is otherwise correct, you will lose points for memory leaks.
- For this project, you should be prepared to spend a fair amount of time debugging your code even after you get it to compile. My guideline would be to expect **at least week of debugging after you get your code to compile**. Of course, it might take longer than that or shorter than that, depending upon how carefully you write your code.
- A reminder about some basic rules that you are expected to follow for all the code you write in this course: your variable names should be intelligible, giving some indication of what the variable does (e.g., no i's or j's except in very simple loops). Also, your code should have comments sprinkled throughout, describing what the code is doing. You need not comment every line.
- TA will be collecting only the indexPage.c file. Make sure that all of your work is in that file.
- While this is a group project, you will be using this code in the next project, which is an individual project. Thus, you should **make sure that everyone in the group understands what the code does! Do not hesitate to ask your group members to help you understand something.**
- You can refer to the source code of Project 1 (*i.e.*, crawler.c), such as how to allocate memory for structures and how to get the command-line arguments.

Submission

Before pushing the code indexPage.c and the evaluation file, please use Git to run

```
$ git pull origin master
```

TA may have updated the score file to your Bitbucket repository. Running the above command can get the latest score file and keep your cloud9 project directory in sync with your Bitbucket repository.

Remember that when you push the code for this project to your Bitbucket repository, please operate under the “project” directory, instead of “project2” subdirectory. You will upload two files:

- The completed indexPage.c, which is the same for both group members.
- A text file “evaluation.txt”, in which you will indicate the contributions of you and your group member. For example, simply indicate who implemented which function(s). Moreover, in this file please indicate how many hours you worked on this project.

If you have any questions, please let the instructor know.

Grading rubric:

- Code can be compiled – 100pts
- Implementation (if the code is compiled) – 200 pts
  - Passing 20 testing cases
  - freeTrieMemeory function – 20 pts
  - printTrieContents function – 40pts
  - addWordOccurrence function – 40pts
  - indexPage function – 60 pts
  - main function – 40pts
- No memory leaks (if the code is well implemented) – 70pts
- Coding style (if the most of the above are implemented) – 30 pts

Total: 400 points.