Programming Assignment #4: Tries

COP 3502, Spring 2014

Due: Sunday, April 20, 11:59 PM on Webcourses@UCF

Abstract

In this programming assignment, you will gain experience with an advanced tree data structure that is used to store strings, and which allows for efficient insertion and lookup: a trie!

By completing this assignment and reflecting upon the awesomeness of tries, you will fortify your knowledge of algorithms and data structures and solidify your mastery of many C programming topics you have been practicing all semester: dynamic memory allocation, file I/O, processing command line arguments, dealing with structs and pointers to structs, and so much more.

In the end, you will have implemented a tremendously useful data structure that has many applications in text processing and corpus linguistics.

Attachments

Trie.h, corpus{01-05}.txt, input{01-05}.txt, output{01-05}.txt, printTrie.txt

Deliverables

Trie.c

1. Overview: Tries

We have seen in class that the trie data structure can be used to store strings. It provides for efficient string insertion and lookup; insertion into a trie is O(*k*) (where *k* is the length of the string being inserted), and searching for a string is an O(*k*) operation (worst-case). In a trie, a node does not store the string it represents; rather, the *edges* taken to reach that node from the root indicate the string it represents. Each node contains a *flag* (or *count*) variable that indicates whether the string represented by that node has been inserted into the trie (or *how many times* it has been inserted). For example:

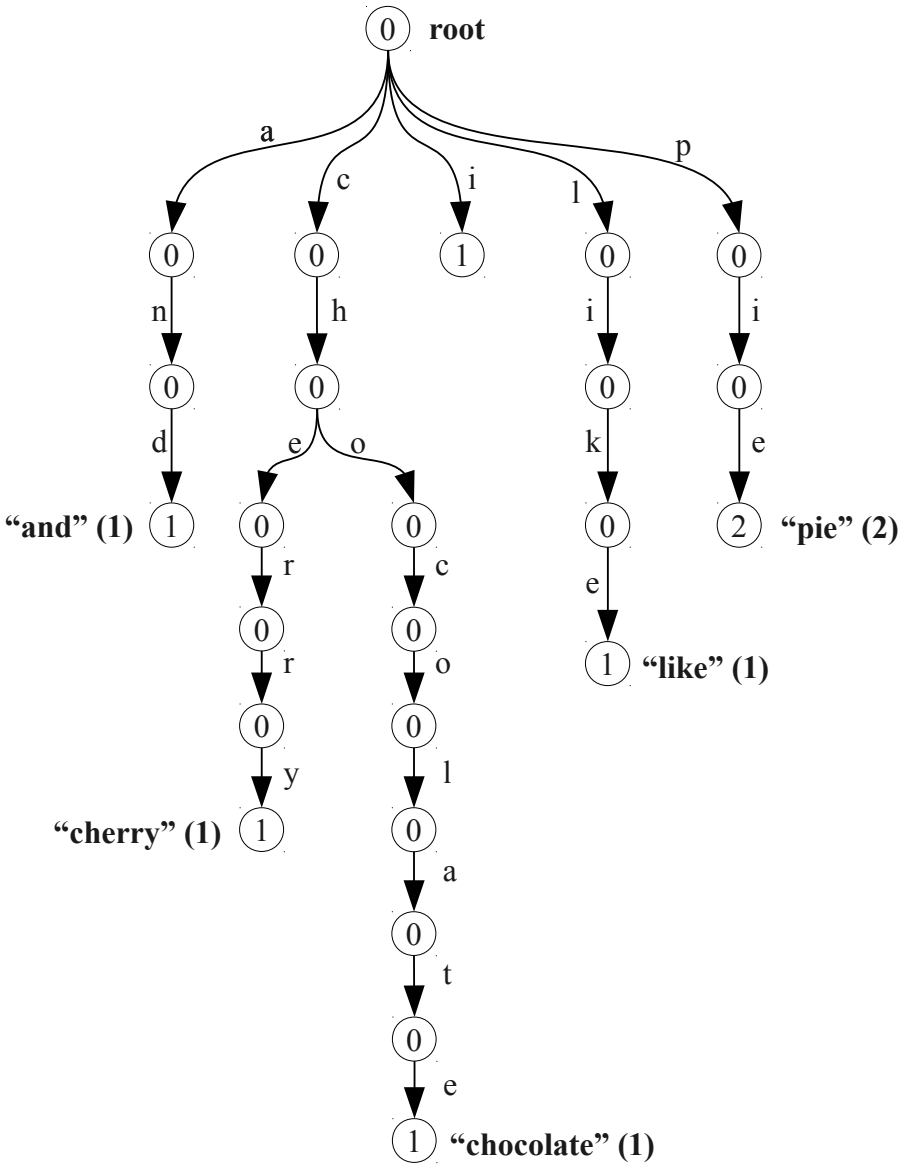

Figure 1:
This is a trie that codifies the words "and," "cherry," "chocolate," "like," and "pie." The string "pie" is represented (or counted) twice. All other strings are counted only once.

1.1. TrieNode Struct (Trie.h)

In this assignment, you will insert words from a *corpus* (that is, a body of text from an input file) into a trie. The struct you will use for your trie nodes is as follows:

```
typedef struct TrieNode
{
   // number of times this string occurs in the corpus
   int count;

   // 26 TrieNode pointers, one for each letter of the alphabet
   struct TrieNode *children[26];

   // the co-occurrence subtrie for this string
   struct TrieNode *subtrie;
} TrieNode;
```

You must use this trie node struct, which is specified in `Trie.h` without any modifications. You should `#include` the header file from `Trie.c` like so:

```
#include "Trie.h"
```

Notice that the trie node, because it only has 26 children, represents strings in a case insensitive way (i.e., “apple” and “AppLE” are treated the same in this trie).

1.2. Co-occurrence Subtries

Words that appear together in the same sentence are said to “co-occur.” Consider, for example, the following sentence:

> *I like cherry pie and chocolate pie* .

In the example sentence, the word “chocolate” co-occurs with the following terms (with counts in parentheses): *and* (1), *cherry* (1), *i* (1), *like* (1), *pie* (2).

For the purposes of this assignment, we will never consider a word to co-occur with itself. So, in the example sentence above, “pie” co-occurs with: *and* (1), *cherry* (1), *chocolate* (1), *i* (1), *like* (1). Notice that “pie” itself is not in the list of words co-occurring with “pie.”

If we place these terms (and their associated counts) into their own tries, those tries will look like this:

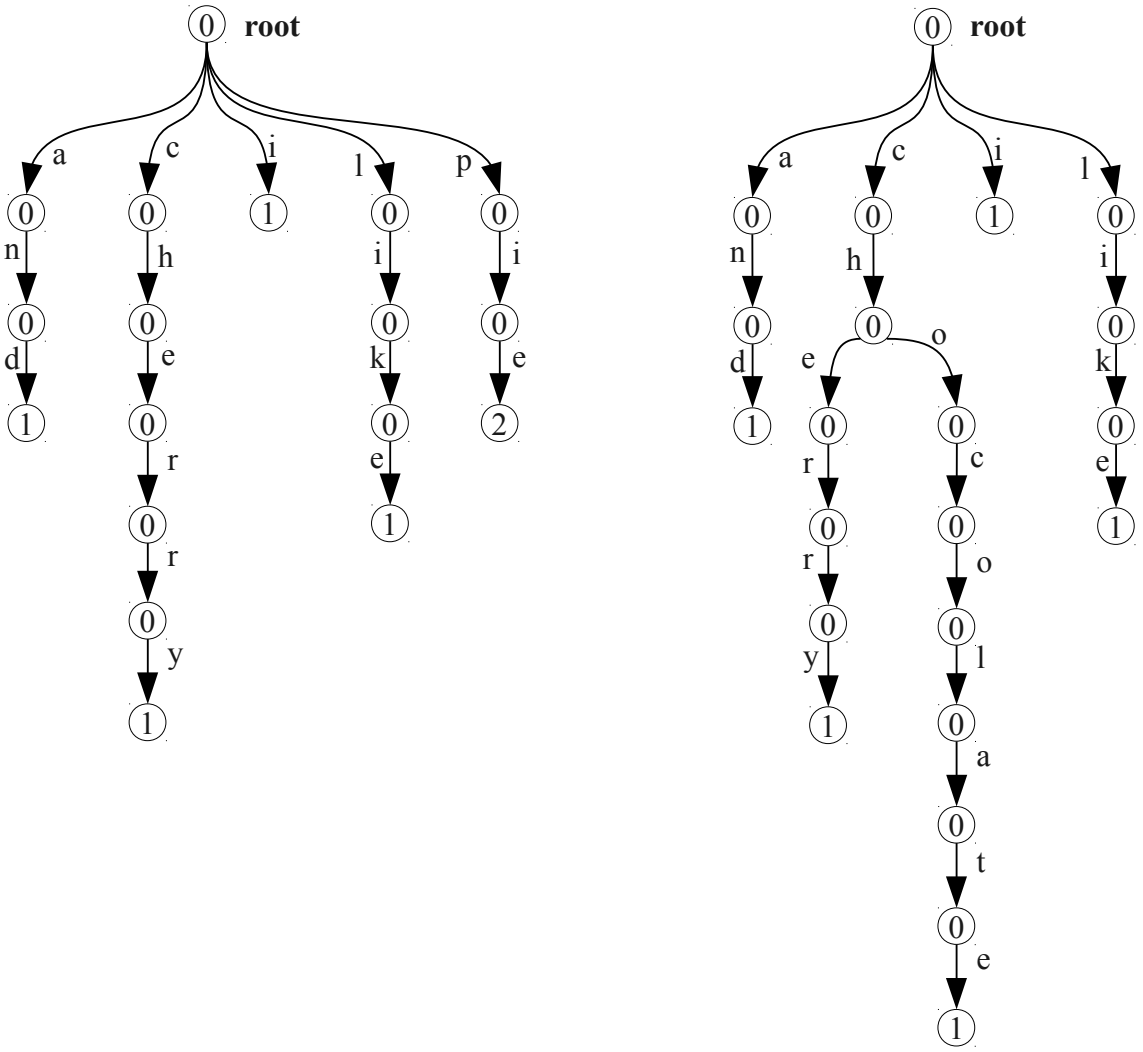

Figure 2:
Co-occcurrence subtries for “chocolate” (left) and “pie” (right), based on the sentence, “I like cherry pie and chocolate pie.”

In Figure 2, the trie on the left is what we will call the *co-occurrence subtrie* for the word “chocolate,” based on the example sentence above (*I like cherry pie and chocolate pie*). Its root should be stored in the *subtrie* pointer field of the node marked “chocolate” in the original trie diagram in Figure 1 (see page 2 above).

Similarly, the trie on the right in Figure 2 is the co-occurrence subtrie for the word “pie.” Its root should be stored in the *subtrie* pointer field of the node marked “pie” in the original trie diagram in Figure 1 (see page 2, above).

Within these subtries, all *subtrie* pointers should be initialized to NULL, because we will NOT produce sub-subtries in this assignment!

2. Input Files and Output Format

2.1. Command Line Arguments

Your program will take two command line arguments, specifying two files to be read at runtime:

```
./a.out corpus01.txt input01.txt
```

The first filename specifies a corpus that will be used to construct your trie and subtries. The second filename specifies an input file with strings to be processed based on the contents of your trie.

For more information on processing command line arguments with `argc` and `argv`, see the instructions from the Program #3 PDF. You can assume that I will always specify valid input files when I run your program.

2.2. Corpus File

2.2.1 Format

The corpus file contains a series of sentences. Each line contains a single sentence with at least 1 word and no more than 20 words. Each word contains fewer than 1024 characters. Each sentence is terminated with a single period, and that period is always preceded by a space. For example:

`corpus01.txt`:

```
I like cherry pie and chocolate pie .
```

2.2.2 Building the Main Trie

First and foremost, each word from the corpus file should be inserted into your trie. If a word occurs multiple times in the corpus, you should increment *count* variables accordingly. For example, the trie in Figure 1 (see page 2, above) corresponds to the text given in the `corpus01.txt` file above.

2.2.3 Building Subtries

For each sentence in the corpus, update the co-occurrence subtrie for each word in that sentence. The structure of the co-occurrence subtries is described above in Section 1.2, “Co-occurrence Subtries.”

If a string in the main trie does not co-occur with any other words, its subtrie pointer should be NULL.

2.3. Input File

The input file (the second filename specified as a command line argument) will contain any number of lines of text. Each line will contain either a single word, or an exclamation point (!). If you encounter

an exclamation point, you should print the strings represented in your main trie in alphabetical order, with the occurrence count in parentheses.

Otherwise, if you encounter a string, it will consist strictly of alphabetical characters. They may be upper- or lowercase, but they will not contain punctuation, spaces, numbers, or any other non-alphabetical characters. The string will contain fewer than 1024 characters. Process these strings as follows:

- If the string you encounter is in the trie, you should print the string, followed by a printout of its subtrie contents, using the output format shown below.

- If the string you encounter is in the trie, but its subtrie is empty, you should print that string, followed on the next line by "(EMPTY)".

- If the string you encounter is not in the trie at all, you should print that string, followed on the next line by "(INVALID STRING)".

For example, the following corpus and input files would produce the following output:

corpus01.txt:

```
I like cherry pie and chocolate pie .
```

input01.txt:

```
!
chocolaTE
apricot
```

program output:

```
and (1)
cherry (1)
chocolate (1)
i (1)
like (1)
pie (2)
chocolaTE
- and (1)
- cherry (1)
- i (1)
- like (1)
- pie (2)
apricot
(INVALID STRING)
```

For an example of what would cause an empty co-occurrence subtrie, consider the following input files:

`corpus02.txt`:

```
spin straw to gold .
spin all night long .
spin spin spin .
spindle .
```

`input02.txt`:

```
spin
spindle
nikstlitslepmur
```

`program output`:

```
spin
- all (1)
- gold (1)
- long (1)
- night (1)
- straw (1)
- to (1)
spindle
(EMPTY)
nikstlitslepmur
(INVALID STRING)
```

You must follow the output format above precisely. Be sure to consult the included text files for further examples.

I have included some functions that will help you print the contents of your trie(s) in the required format, because I think those functions are a bit too tricky to expect you to write them on your own. See `printTrie.txt` (attachment) for those functions. You're welcome to copy and paste them into your `Trie.c` source file if you want to use them!

2.4. Final Thoughts on Test Cases, Input, and Output

I have attached a few sample input and output files so you can check that your program is working as intended. Be sure to use `diff` on Eustis to make sure your output matches ours exactly. I also encourage you to develop your own test cases; ours are by no means comprehensive.

3. Function Requirements

3.1. Required Function

You have a lot of leeway with how to approach this assignment. Other than `main()`, there is only one required function. How you structure the rest of your program is up to you.

```
TrieNode *buildTrie(char *filename);
```

> **Description:** `filename` is the name of a corpus text file to process. Open the file and create a trie (including all its appropriate subtries) as described above.
>
> **Returns:** The root of the new trie.

3.2. Suggested Functions

These functions are not required, but I think they will simplify your task immensely if you implement them properly and call them when processing your corpus/input files. Think of these function descriptions as hints at how to proceed. If you want, you can even implement these functions with different parameters, return types, and so on.

```
TrieNode *createTrieNode(void);
```

> **Description:** Dynamically allocate space for a new TrieNode struct. Initialize all the struct members appropriately.
>
> **Returns:** A pointer to the new node.

```
TrieNode *getNode(TrieNode *root, char *str);
```

> **Description:** Searches the trie for the specified string, `str`.
>
> **Returns:** If the string is represented in the trie, return its terminal node (the last node in the sequence, which represents that string). Otherwise, return `NULL`.

```
void insertString(TrieNode *root, char *str);
```

> **Description:** Inserts the string `str` into the trie. Since it has no return value, it assumes the root already exists (i.e., `root` is not `NULL`). If `str` is already represented in the trie, simply increment its `count` member.

4. Compilation and Testing (Linux/Mac Command Line)

To compile at the command line:

```
gcc Trie.c
```

By default, this will produce an executable file called `a.out` that you can run by typing, e.g.:

```
./a.out corpus01.txt input01.txt
```

If you want to name the executable something else, use:

```
gcc Trie.c -o Trie.exe
```

...and then run the program using:

```
./Trie.exe corpus01.txt input01.txt
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./Trie.exe corpus01.txt input01.txt > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
14c14
< INVALID STRING
---
> (INVALID STRING)
seansz@eustis:~$ _
```

5. Grading Criteria and Submission

5.1. Deliverables

Submit a single source file, named `Trie.c`, through Webcourses@UCF. The source file should contain definitions for the required function (listed above), as well as any auxiliary functions you decide to implement. Don't forget to `#include "Trie.h"` in your source code. We will compile your program using:

```
gcc Trie.c
```

Be sure to include your name and PID as a comment at the top of your source file.

5.2. Additional Restrictions: Use Tries; Do Not Use Global Variables

You must use tries to receive credit for this assignment. Also, please do not use global variables in this program. Doing so may result in a huge loss of points.

5.3. Grading

The expected scoring breakdown for this programming assignment is:

80%	Correct Output for Test Cases
20%	Unit Testing of `buildTrie()` Function

Your program must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization, punctuation, or whitespace errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

For this program, we will also be unit testing your code. That means we will devise tests that determine not only whether your program's overall output is correct, but also whether your `buildTrie()` function does exactly what it is required to do. So, for example, if your program produces correct output but your `buildTrie()` function is simply a skeleton that returns `NULL` no matter what parameters you pass to it, or if it produces a totally funky, malformed trie, your program will fail the unit tests.

5.4. Closing Remarks

Start early. Work hard. Ask questions. Good luck!