Programming Assignment #4: Text Prediction with Tries

COP 3502, Fall 2017

Due: Sunday, November 19, *before* 11:59 PM

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! You will then use this data structure to complete a text prediction task.

In this assignment, you will also learn how to use `valgrind` to test your programs for memory leaks.

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 management, 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.

In this program, you are welcome to use any code I have given you so far in class, as long as you include a comment to give me credit (primarily so your submission doesn't get flagged for plagiarism). The intellectually curious student will, of course, try to write the whole program from scratch.

Deliverables

TriePrediction.c

(***Note!*** Capitalization and spelling of your filename matter!)

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:

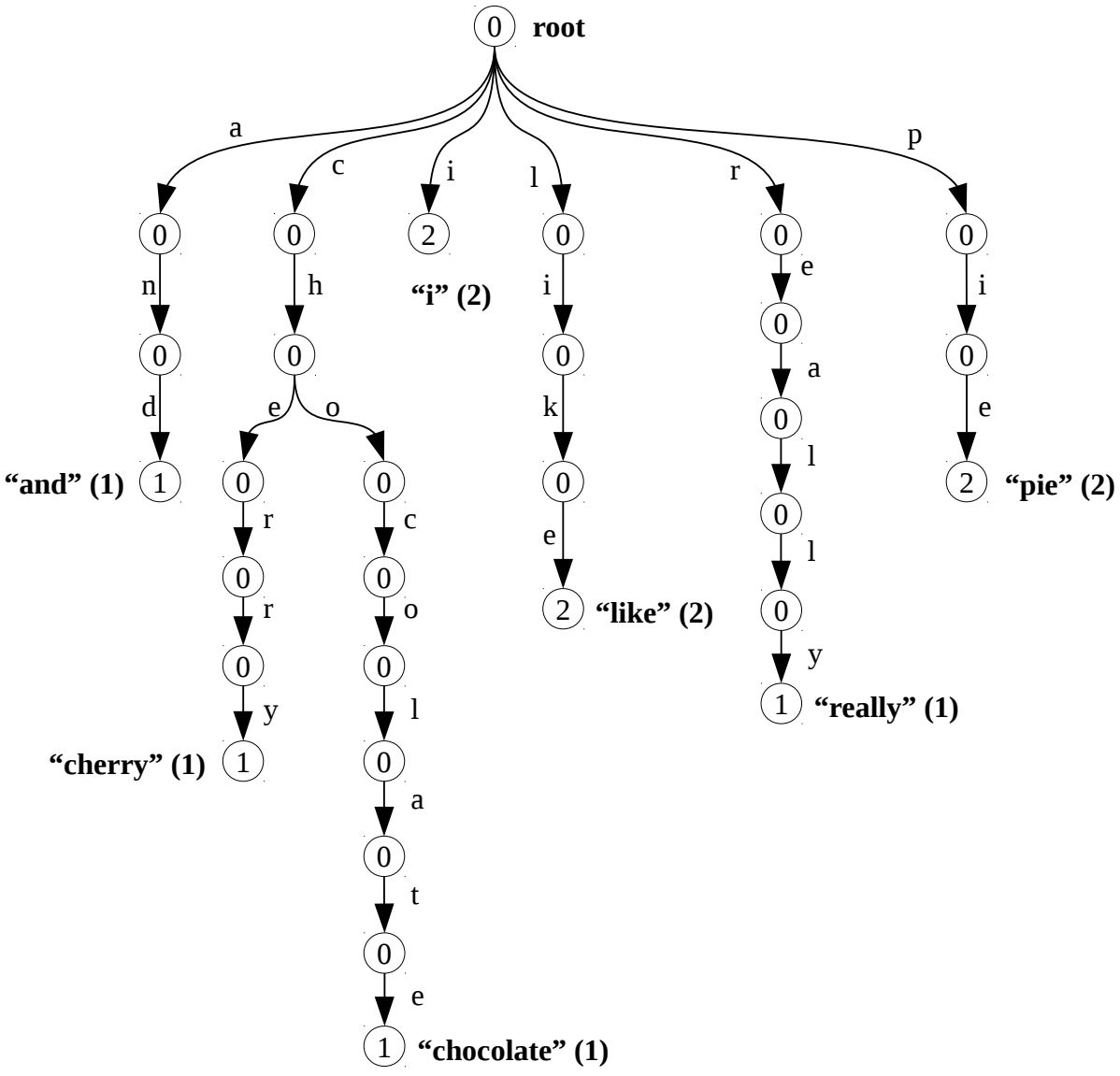


Figure 1:
This is a trie that codifies the words “and,” “cherry,” “chocolate,” “I,” “like,” “really,” and “pie.” The strings “I,” “like,” and “pie” are represented (or counted) twice. All other strings are counted only once.

1.1. TrieNode Struct (TriePrediction.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 `TriePrediction.h`, without any modifications. You should `#include` the header file from `TriePrediction.c` like so:

```
#include "TriePrediction.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. Subtries: Contextual Co-occurrence and Predictive Text

Words that appear together in the same sentence are said to “co-occur.” In this program, we’ll be interested in contextual co-occurrence and predictive text – namely, given some word, *w*, what are all the words that we see immediately after *w* in the sentences in some corpus? Consider, for example, the following sentence:

> *I like cherry pie, and I really like chocolate pie.*

In the example sentence, the word “I” is followed by the words “like” and “really”, the word “pie” is followed by the word “and”, and so on.

To track co-occurrence, for each word *w*, we’ll place each word that directly follows *w* into the subtrie of *w*. For example, if we place these terms (and their associated counts) into their own tries, those tries will look like this:

(*See following page for diagram.*)

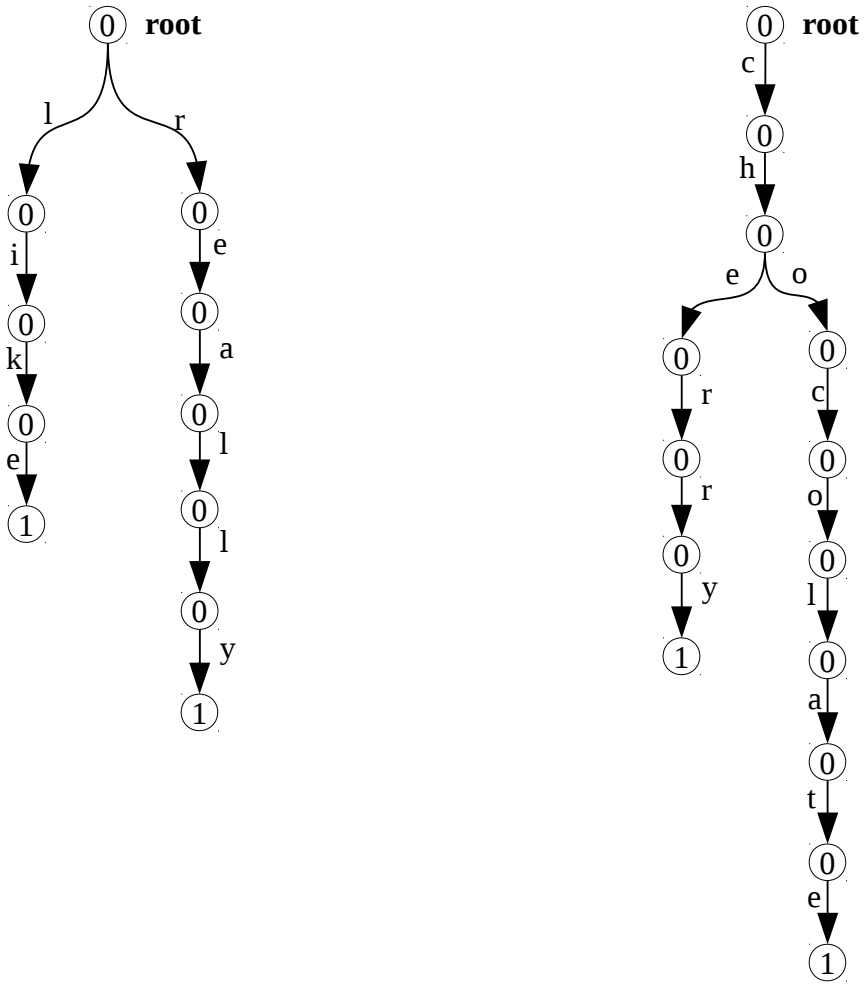


Figure 2:
Co-occurrence subtries for “I” (left) and “like” (right), based on the sentence, “I like cherry pie, and I really like chocolate pie.”

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

2.1. Command Line Arguments

Your program will take two command line arguments, specifying two files to be read at runtime. The first filename specifies a corpus that will be used to construct your trie and subtries. The second filename specifies an input file with commands to be processed based on the contents of your trie. You can assume that we will always specify valid input files when we run your program. For example:

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

It's super easy to get command line arguments (like the string "corpus01.txt" in this example) into your program. You just have to change the function signature for `main()`. Whereas we have typically seen `main()` defined using `int main(void)`, you will now use the following function signature instead:

```
int main(int argc, char **argv)
```

Within `main()`, `argc` is now an integer representing the number of command line arguments passed to the program (including the name of the executable itself). `argv` is an array of strings that stores all those command line arguments. `argv[0]` stores the name of the program being executed.

2.2. Example: A Program That Prints All Command Line Arguments

For example, here's a simple program that would print out all the command line arguments passed to a program:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i;
   for (i = 0; i < argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);
   return 0;
}
```

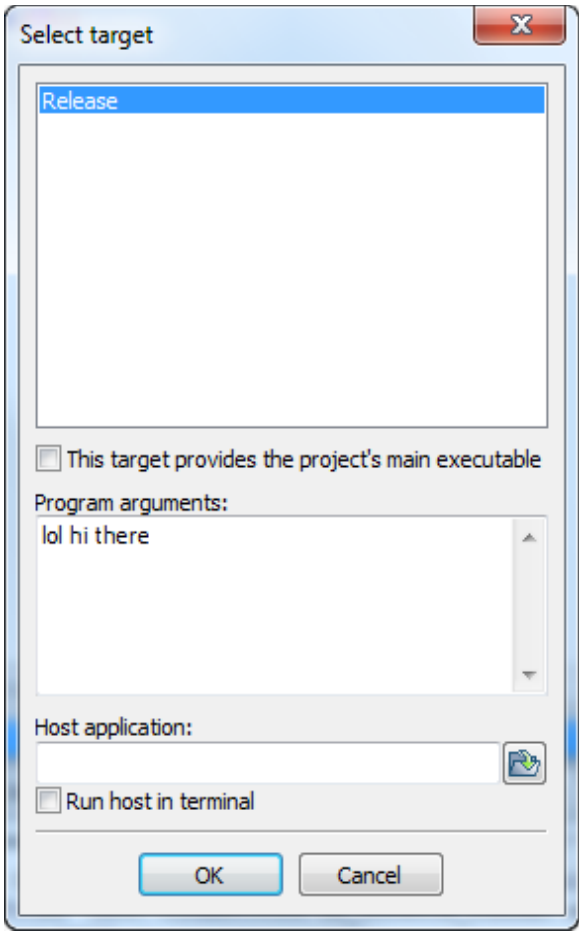
If we compiled that code into an executable file called `a.out` and ran it from the command line by typing `./a.out lol hi there!`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
argv[3]: there!
```

2.3. Passing Command Line Arguments in CodeBlocks

You can set up CodeBlocks to automatically pass command line arguments to your program every time you hit *Build and run (F9)* within the IDE. Simply go to *Project → Set program's arguments...*, and in the box labeled *Program arguments*, type your desired command line arguments.

For example, the following setup passes `lol hi there` as command line arguments to the program when compiled and run within CodeBlocks. (I don't think you need to check off the box next to "*This target provides*...." CodeBlocks seems to take care of that automatically after you hit "OK.")



(Of course, you will want to provide more reasonable command line arguments than `lol hi there`, such as `corpus01.txt` and `input01.txt`.)

If you compile your program like this in CodeBlocks and later run your executable from the command line, you will still need to supply fresh arguments at the command line.

At any rate, don't forget to test your code on Eustis before submitting, even if you do most of your development in CodeBlocks. As always, the safest way to test your program is by using the test-all.sh script on Eustis.

3. Input File Formats and Output Format

3.1. Corpus File

3.1.1 Corpus File Format

The corpus file contains a series of sentences. Each line contains a single sentence with at least 1 word and no more than 30 words. Each word contains fewer than 1024 characters, some of which may be punctuation characters.

Each sentence will contain exactly one of the following punctuators, and it will occur at the very end of the sentence: period (‘.’), exclamation point (‘!’), or question mark (‘?’). Other punctuators may occur throughout the sentence, including (but not limited to) commas (‘,’), colons (‘:’), semicolons (‘;’), apostrophes (‘’ and “’), quotation marks (‘“’ and ‘”’), and so on.

All words will have at least one alphabetic character (‘a’ through ‘z’ or ‘A’ through ‘Z’). Words will *not* contain any numeric characters (‘0’ through ‘9’). All punctuators should be stripped away from all words before entering them into the trie. So, for example, the word “don’t” will be entered into the trie as “dont”.

For example:

`corpus01.txt`:

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

`corpus02.txt`:

```
'tweren't my fault, I swear!
And now we're on to the second line of text.
```

3.1.2 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.

3.1.3 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, “Subtries: Contextual Co-occurrence and Predictive Text.”

If a string in the main trie is not followed by any other words in the corpus, its subtrie pointer should be NULL.

3.2. Command File

One of the required functions for this program needs to open and process an input file (the second filename specified as a command line argument) that contains commands for you to process after building your trie. Each line in this file will correspond to one of the following three commands:

Command	Description
`@ str n`	This is the text prediction command. When you encounter this command, you should print the following sequence of (n + 1) words: $w_0\ w_1\ w_2\ \ldots\ w_n$ In that sequence, w_0 is `str`, and for $1 \le i \le n$, w_i is the word that most frequently follows word w_{i-1} in the corpus. Note that the words are separated by spaces, and there is never a space after the last word on one of these lines of output. Furthermore, w_0 is always capitalized in the output exactly as it was capitalized in the command file, but words w_1 through w_n should appear in all lowercase. If `str` does not appear in the trie, it should appear on this line of output by itself. If some w_i does not have any words in its subtrie, the sequence should terminate prematurely. Again, this line of output should *not* have a trailing space at the end of it, even if it terminates prematurely.
`str`	If str is in the trie, print the string, followed by a printout of its subtrie contents, using the output format shown in the sample output files for this program. Note that when printing a subtrie, the words in the subtrie are preceded by hyphens. If `str` is in the trie, but its subtrie is empty, you should print that string, followed on the next line by "(EMPTY)". If `str` is not in the trie at all, you should print that string, followed on the next line by "(INVALID STRING)".
`!`	The character '!' will appear on a line by itself. When you encounter this command, you should print the trie using the output format shown in the sample outputs for this program. (When printing the main trie, there are no hyphens preceding the words on each line. See sample output files, or see the sample output below on pgs. 9 and 10.)

Important note: In the commands listed above, `str` is always a string, and `n` is a non-negative integer. Note that `str` is guaranteed to contain alphabetical characters only ('A' through 'Z' and 'a' through 'z'). Not counting the need for a null terminator ('`\0`'), the length of `str` can range from 1 through 1023 characters (inclusively). So, with the null terminator, you might need up to 1024 characters to store `str` as a char array when reading from the input file.

For more concrete examples of how these commands work or how your program's output should be formatted, see the attached input/output files and check out the function descriptions below in Section 4, "Function Requirements" (page 11).

3.3. Sample Input and Output Files

Consider, for example, the following corpus file and command file:

`corpus03.txt`:

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

`input03.txt`:

```
!
chocolaTE
apricot
@ I 11
@ chocolate 1
@ persimmon 20
```

If passed to your program, those files should produce the following output:

`output03.txt`:

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

The fun continues on the following page!

Note that a word might be in the trie but have an empty subtrie. Consider the following example:

`corpus04.txt`:

```
Spin straw to gold.
Spin all night long.
Spin spin spin.
Spindle.
```

`input04.txt`:

```
spin
spindle
nikstlitslepmur
```

`output04.txt`:

```
spin
- all (1)
- spin (2)
- straw (1)
spindle
(EMPTY)
nikstlitslepmur
(INVALID STRING)
```

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

3.4. Trie Printing Functions (Included with Assignment!)

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 `TriePrediction.c` (attachment) for those functions. You're welcome to use those functions in the source file you submit, or you can write your own.

Also, studying and understanding those functions will serve as a launching point for you to write the other functions required to get this program working.

The fun continues on the following page!

4. Function Requirements

In the source file you submit, `TriePrediction.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

Note: I did not include any unit tests for `buildTrie()` or `processInputFile()`, but I do intend to deploy unit tests for those functions when grading your assignment.

```
int main(int argc, char **argv);
```

> **Description:** You must write a `main()` function for this program. It should do the following five things: (1) capture the names of the corpus and input files (passed as command line arguments), (2) call the `buildTrie()` function, (3) call the `processInputFile()` function, (4) call the `destroyTrie()` function, and (5) return zero.
>
> **Returns:** 0 (zero).

```
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.

```
int processInputFile(TrieNode *root, char *filename);
```

> **Description:** This function takes in the root of a trie and the name of an input file, and processes that file according to the description above in Section 3.2, “Command File.” While we will always specify valid filenames as command line arguments, we might pass invalid filenames when unit testing. In the event that a bad filename is passed to this function (i.e., the specified file does not exist), this function should simply return 1 without producing any output.
>
> **Output:** This function should produce output according to the specification described above in Section 3.2 (“Command File”). For additional details or clarification on the output, please be sure to refer to the input/output files included with this assignment. Note that this function should not produce any output if the input file does not exist.
>
> **Returns:** If the specified input file does not exist, return 1. Otherwise, return 0.

```
TrieNode *destroyTrie(TrieNode *root);
```

> **Description:** Free all dynamically allocated memory associated with this trie.
>
> **Returns:** `NULL`.

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

Description: Searches the trie for the specified string, `str`.

Returns: If the string is represented in the trie (with *count* ≥ 1), return a pointer to its terminal node (the last node in the sequence, which represents that string). Otherwise, return `NULL`.

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

Description: Searches the trie for the most frequently occurring word and copies it into the string variable passed to the function, `str`. If you are calling this function yourself, you should create the `str` char array beforehand, and it should be (at least) long enough to hold the string that will be written to it. (For this, you can use `MAX_CHARACTERS_PER_WORD` from TriePrediction.h.) If we call this function manually when testing your code, we will ensure the `str` char array is created ahead of time and that it is long enough to hold the longest string in the trie. Note that there is no guarantee that `str` will contain the empty string when this function is first called; the string might contain garbage data.

If there are multiple strings in the trie that are tied for the most frequently occurring, populate `str` with the one that comes first in alphabetical order. If the trie is empty, set `str` to the empty string (“”).

Hint: You might find it easier to write helper functions that you call from within this function.

Returns: Nothing. This is a `void` function.

```
int containsWord(TrieNode *root, char *str);
```

Description: Searches the trie for the specified string, `str`.

Note: You might find that you don’t need this function to build out the text prediction functionality of your code, but you still need to implement it as part of this assignment.

Returns: If the string is represented in the trie (with *count* ≥ 1), return 1. Otherwise, return 0.

```
int prefixCount(TrieNode *root, char *str);
```

Description: Counts the number of strings in the trie (with *count* ≥ 1) that begin with the specified string, `str`. Note that if the specified string itself is contained within the trie, that string should be included in the count. If one of these strings occurs more than once, its entire *count* should be added to the return value.

Note: You might find that you don’t need this function to build out the text prediction functionality of your code, but you still need to implement it as part of this assignment.

Returns: The number of strings in the trie that begin with the specified string, `str`.

```
double difficultyRating(void);
```

Returns: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Returns: An estimate (greater than zero) of the number of hours you spent on this assignment.

5. 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.

Note: You should try to implement this yourself, but there's a copy of this function in our notes in Webcourses, should you really need it.

Returns: A pointer to the new node.

```
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.

Note: You should try to implement this yourself, but there's a copy of this function in our notes in Webcourses, should you really need it.

Returns: Nothing. This is a `void` function.

```
void stripPunctuators(char *str);
```

Description: Takes a string, `str`, and removes all punctuation from the string. For example, if `str` contains the string "`Hello!`" when the function is called, then `str` should contain the string "`Hello`" when the function returns. When writing this function, you might find C's built-in `isalpha()` function (from `ctype.h`) to be helpful.

Returns: Nothing. This is a `void` function.

6. Special Requirement: Memory Leaks and Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called `valgrind`, which is installed on Eustis.

To run your program through `valgrind` at the command line, compile your code with the `-g` flag, and then run `valgrind`, like so:

```
gcc TriePrediction.c test_launcher_std.c -g
valgrind --leak-check=yes ./a.out corpus01.txt input01.txt
```

For help deciphering the output of `valgrind`, see: http://valgrind.org/docs/manual/quick-start.html

Note that if you do not use `fclose()` to explicitly close all open files before your program terminates, `valgrind` might alert you that your program has a memory leak.

In the output of `valgrind`, the magic phrase you're looking for to indicate that you have no memory leaks is:

```
All heap blocks were freed – no leaks are possible
```

7. Test Cases and the test-all.sh Script

We've included multiple test cases with this assignment to show some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, `test-all.sh`, that will compile and run all test cases for you. The script will also automatically test for memory leaks using `valgrind`.

The `test-all.sh` script is the safest, most sure-fire way to test your code.

You can run the script on Eustis by placing it in a directory with `TriePrediction.c`, `TriePrediction.h`, the `sample_output` directory, and all the test case files, and typing:

```
bash test-all.sh
```

Please note that these test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

8. Standard Tests and Unit Tests

With this assignment, we have included two kinds of test cases: standard test cases, which you can process from the command line, and unit test cases, which bypass your `main()` function and instead call functions we have written, which will in turn call individual functions from your code to see if they're working as intended.

8.1. Running Standard Test Cases

To run the standard test cases, you must compile your source code with `test_launcher_std.c` and then run your program from the command line like so:

```
gcc TriePrediction.c test_launcher_std.c
./a.out corpus01.txt input01.txt
```

Compiling with `test_launcher_std.c` ensures that it is *your* `main()` function that executes when you run the program.

8.2. Running Unit Test Cases

Unit tests are designed to override your `main()` function and directly call one or more functions in your code. This allows us to determine whether you implemented individual functions correctly.

To run unit test cases, you must compile your source code with `test_launcher_unit.c`, as well as the source file for the unit test you want to run. Afterwards, you can run the program without any command line arguments. For example, you can compile and run `unit_test01.c` like so:

```
gcc TriePrediction.c test_launcher_unit.c unit_test01.c
./a.out
```

Compiling with `test_launcher_unit.c` essentially overrides your main() function and ensures that the unit test code you compiled into your program will be executed when you run the program.

8.3. CodeBlocks

If you compile your project with CodeBlocks, you must import *either* `test_launcher_std.c` *or* `test_launcher_unit.c` in your project (but not both). If you are compiling a unit test, you must also import the unit test's source file.

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

To compile your code at the command line and then run it with a standard (corpus-based) test case:

```
gcc TriePrediction.c test_launcher_std.c

./a.out corpus01.txt input01.txt > myoutput.txt

diff myoutput.txt sample_output/output01.txt
```

Of course, you should switch out *corpus01.txt* and *input01.txt* for other corpus and input files.

To compile and run a unit test case at the command line:

```
gcc TriePrediction.c test_launcher_unit.c unit_test01.c

./a.out > myoutput.txt

diff myoutput.txt sample_output/unit_output01.txt
```

If the contents of `myoutput.txt` and `unit_output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff myoutput.txt sample_output/unit_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 myoutput.txt sample_output/unit_output01.txt
1c1
< fail whale :(
---
> Success!
seansz@eustis:~$ _
```

10. Deliverables

Submit a single source file, named `TriePrediction.c`, via Webcourses. The source file should contain definitions for all the required function (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "TriePrediction.h"` in your source code.

Your program must work with the test-all.sh script, and it must be able to compile and run like so:

```
gcc TriePrediction.c test_launcher_std.c

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

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

11. Special Restrictions

You must use tries to receive credit for this assignment. As always, please do not use global variables or mid-function variable declarations. Doing so may result in a huge loss of points.

12. Grading

The *expected* scoring breakdown for this programming assignment is:

80%	correct output for test cases (including unit tests)
10%	no memory leaks (passes `valgrind` tests)
10%	adequate comments and whitespace; source includes student name and NID

Note! Your program must be submitted via Webcourses, and it 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 certain required functions, such as your `buildTrie()` function, do exactly what they are 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.

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