

Programming Assignment #5: Text Prediction with Tries

COP 3502, Spring 2017

Due: Monday, April 24, *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 process command line arguments with `argc` and `argv` and 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 of your filename matters!)

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:

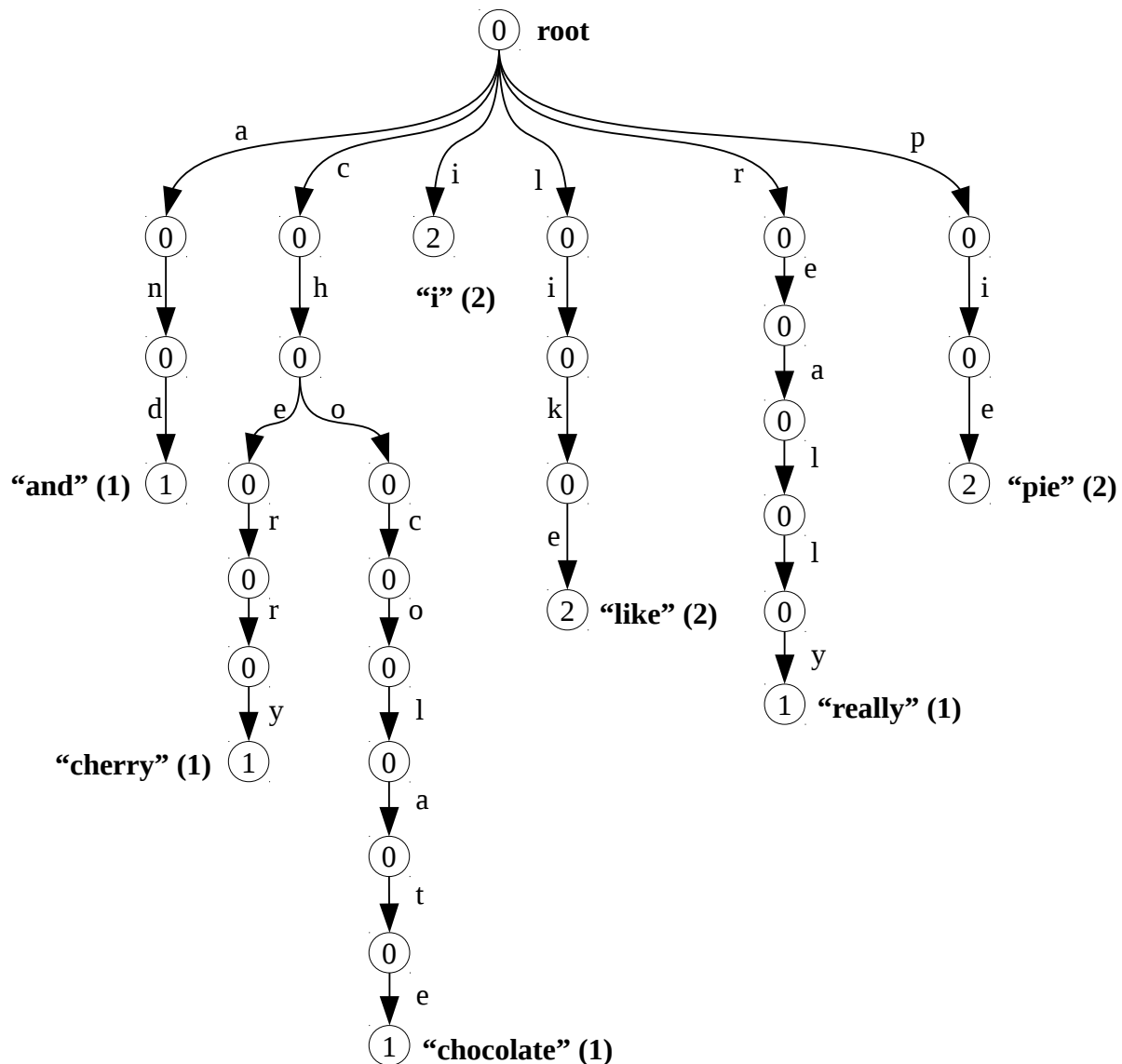


Figure 1:

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.)

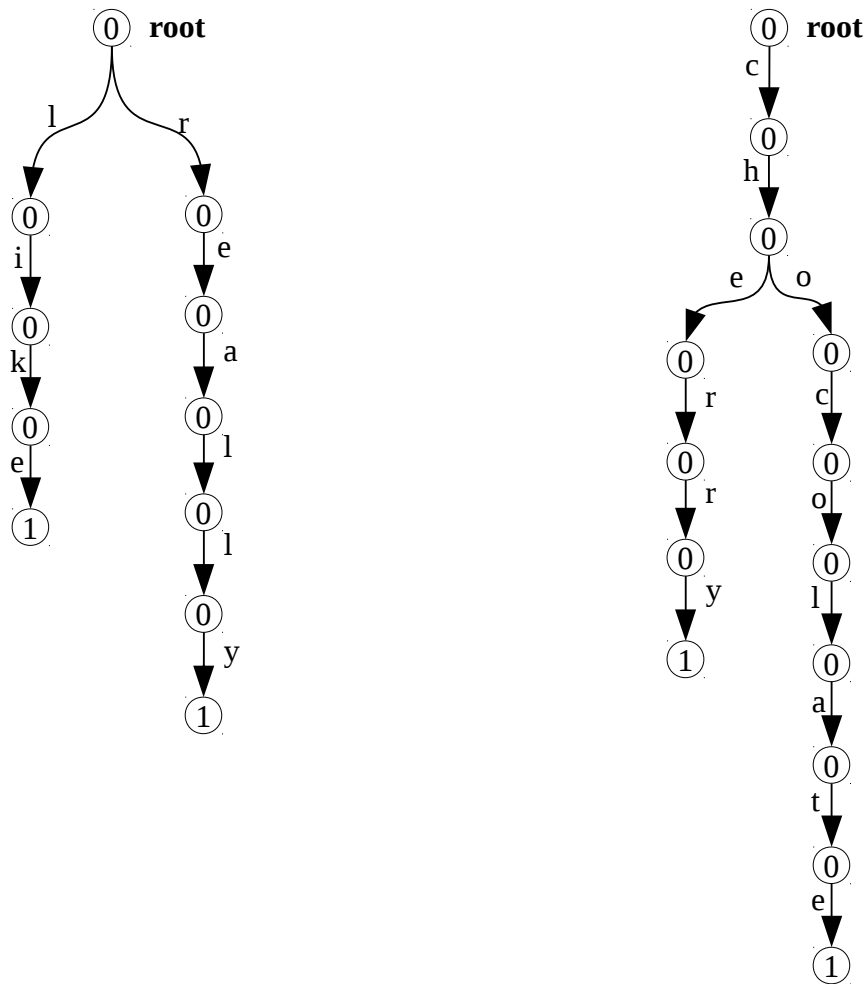


Figure 2:
Co-occurrence subtries for “I” (left) and “like” (right), based on the sentence, “I like cherry pie, and I really like chocolate pie.”

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. 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.

You can assume that I will *always* specify valid input files when I run your program.

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. Following is a simple program that print out all the command line arguments passed to a program. You don't need this in your programming assignment; it's just an example to show how `argc` and `argv` work.

```
#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;
}
```

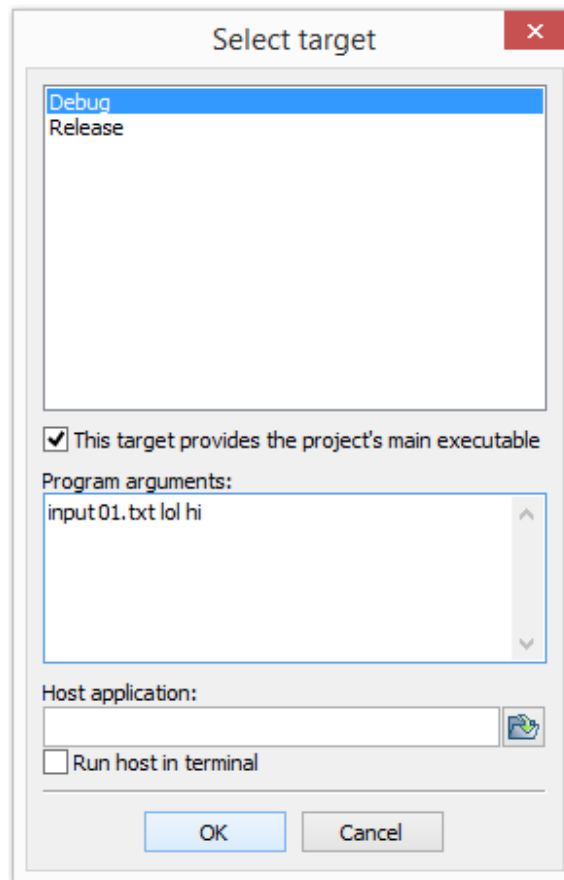
If we compiled this code into an executable file called `a.out` and ran it from the command line by typing `./a.out input01.txt lol hi`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: input01.txt
argv[2]: lol
argv[3]: hi
```

2.2. 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 programs' arguments...*, and in the box labeled *Program arguments*, type your desired command line arguments.

For example, the following setup passes `input01.txt lol hi` 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.")



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.

(Continued on the following page.)

2.3. Corpus File

2.3.1 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.
```

2.3.2 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.

2.3.3 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.

2.4. 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 correspond to one of the following three commands:

Command: <string>

Description:

<string> is a string containing fewer than 1024 characters. The string will contain alphabetical characters only ('A' through 'Z' and/or 'a' through 'z'), and it appears on a line by itself.

If <string> is in the trie, you should print the string, followed by a printout of its subtrie contents, using the output format shown in the sample outputs for this program. (Note that when printing a subtrie, the words in the subtrie are preceded by hyphens.)

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

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

Command: !

Description:

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 pg. 6.)

Command: @ <string> <n>

Description:

This is the text prediction command.

<string> is a string containing fewer than 1024 characters. The string will contain alphabetical characters only ('A' through 'Z' and 'a' through 'z'). <n> is an integer.

When you encounter this command, you should print the following sequence of $(n + 1)$ words:

$$w_0 \ w_1 \ w_2 \ \dots \ w_n$$

where w_0 is <string>, and for $1 \leq i \leq n$, 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 input file, but words w_1 through w_n should appear in all lowercase.

If `<string>` 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.

2.5. Sample Input and Output Files

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

corpus03.txt:

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

input03.txt:

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

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
```

(Continued 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 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 `TriePrediction.c` source file if you want to use them, 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.

(Continued on the following page.)

3. Standard Tests and Unit Tests

With this assignment, I 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 I have written, which will in turn call individual functions from your code to see if they're working as intended.

3.1. 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.

3.2. 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 me 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.

4. 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
```

5. Function Requirements

5.1. Required Function

You have a fair amount of leeway with how to approach this assignment. There are nine required functions. How you structure the rest of your program is up to you.

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

Description: You must write a `main()` function.

Returns: Your `main()` must return 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.

Note: I did not include a unit test for this function, but I do intend to write one to check whether your trie has been constructed properly.

```
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* string 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 I call this function manually when testing your code, I will ensure the string 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.

```
int containsWord(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 1. Otherwise, return 0.

Note: You might find that you don’t need this function to build out the text prediction functionality of your code.

```
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.

Returns: The number of strings in the trie that begin with 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.

```
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.

5.2. 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.

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.

```
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.

```
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.

6. Compilation and Testing (Code::Blocks)

Important Note! If you compile your project with Code::Blocks, 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.

The key to getting your program to include a custom header file in Code::Blocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in Code::Blocks, importing `TriePreidction.h` and the `TriePrediction.c` file you've created (even if it's just an empty file so far).

1. Start Code::Blocks.
2. Create a New Project (*File* → *New* → *Project*).
3. Choose “Empty Project” and click “Go.”
4. In the Project Wizard that opens, click “Next.”
5. Input a title for your project (e.g., “TriePreidction”).
6. Pause to reflect on life a bit. Isn't this amazing?
7. Choose a folder (e.g., Desktop) where Code::Blocks can create a subdirectory for the project.
8. Click “Finish.”

Now you need to import your files. You have two options:

1. Drag your source and header files into Code::Blocks. Then right click the tab for **each** file and choose “Add file to active project.”

– or –

2. Go to *Project* → *Add Files....* Then browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click “Open.” In the dialog box that pops up, click “OK.”

You should now be good to go. Try to build and run the project (F9).

Note! Even if you develop your code with Code::Blocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 7, “Compilation and Testing (Linux/Mac Command Line)”) for instructions on command line compilation in Linux. See also: “Super Important: Running the test-all.sh Script.”

(Continued on the following page.)

7. 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 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 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 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 unit_output01.txt
1c1
< fail whale :(
---
> Success!
seansz@eustis:~$ _
```

7.1. **Super Important:** Running the test-all.sh Script

To be absolutely certain that all your test cases are passing, you should place `TriePrediction.c` and `TriePrediction.h` in the `test_cases` directory included with this assignment and run the `test-all.sh` script, like so:

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


8. Grading Criteria and Submission

8.1. Deliverables

Submit a single source file, named `TriePrediction.c`, via Webcourses. 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 "TriePrediction.h"` in your source code. Your program must compile on Eustis with the following:

```
gcc TriePrediction.c test_launcher_std.c
```

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

8.2. Additional Restrictions: Use Tries; No Globals, Mid-Function Declarations, or System Calls

You must use tries to receive credit for this assignment. As always, you must also avoid the use of global variables, mid-function variable declarations, and system calls (such as `system("pause")`).

8.3. Grading

The expected scoring breakdown for this programming assignment is:

80%	Correct Output for Test Cases (Including Unit Tests)
10%	No Memory Leaks; Passes <code>valgrind</code> Tests
10%	Comments and Whitespace

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 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.

8.4. Closing Remarks

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