Programming Assignment #5: Text Prediction with Tries

COP 3502, Spring 2023

Due: Sunday, April 23, *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 implement a text prediction algorithm.

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 your submission, 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 code doesn't get flagged for plagiarism). Of course, trying to write the whole program from scratch will lead to the greatest amount of growth.

Deliverables

TriePrediction.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

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:

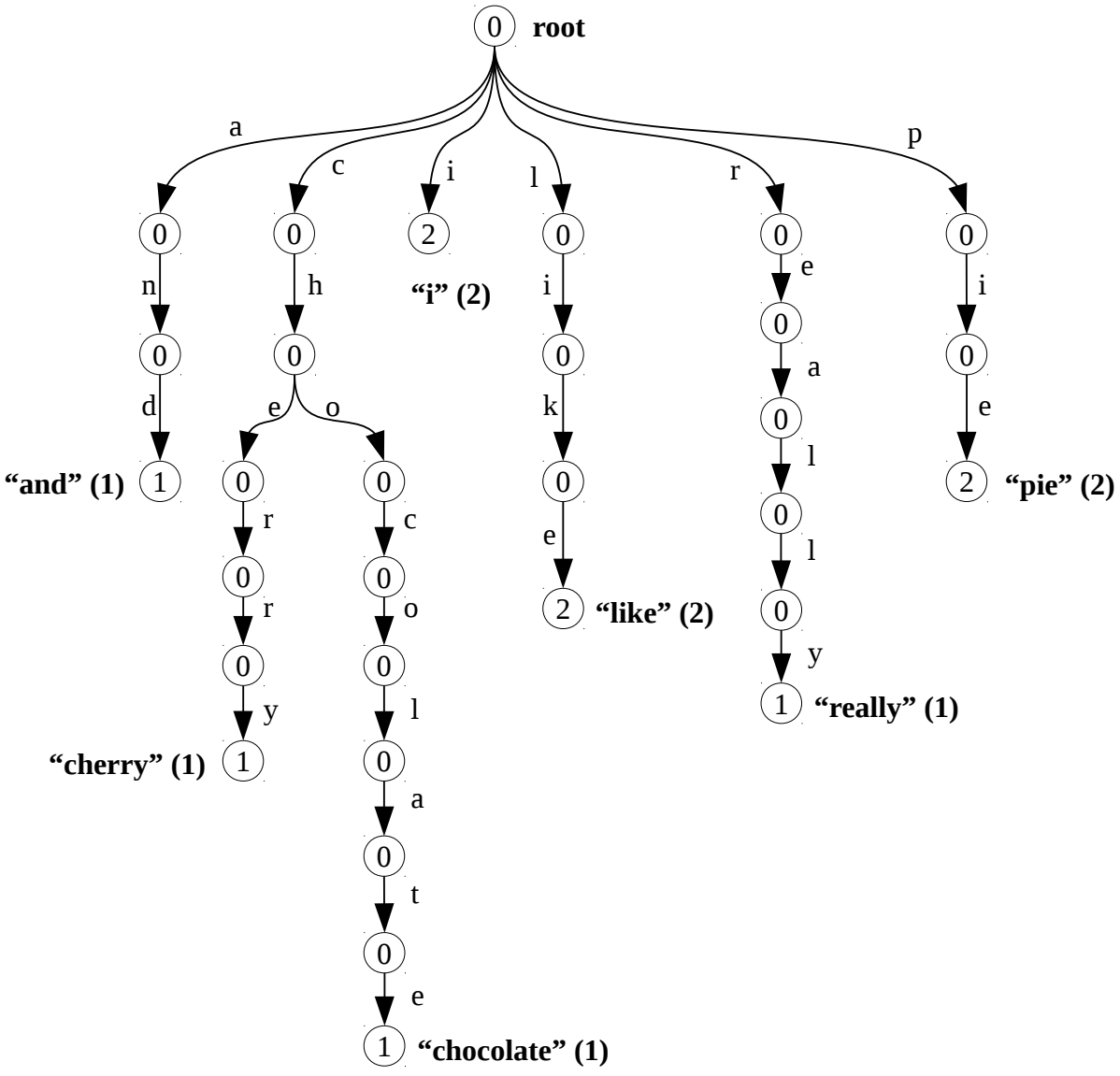

Figure 1:

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 **must** *#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.*)

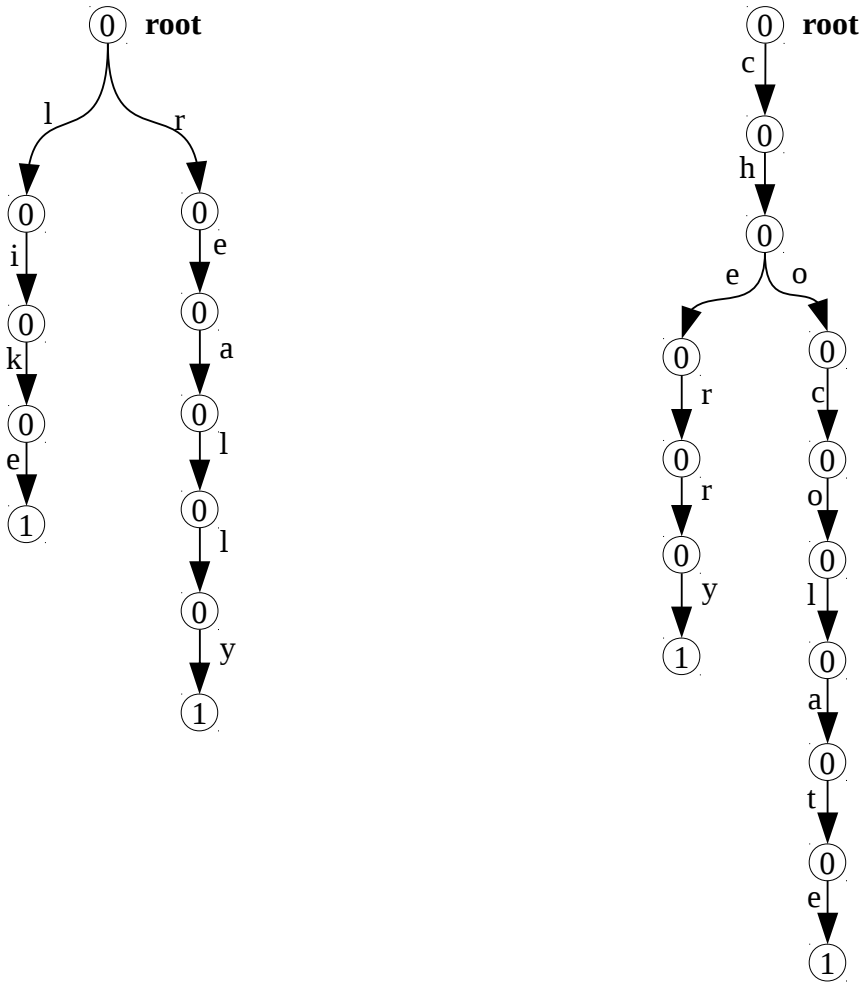

Figure 2:

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*). It contains counts of all words that immediately follow the word “I” in that sentence, and its root should be stored in the *subtrie* pointer field of the node marked **“i” (2)** 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” (2)** 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. 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. The specifications for these files are described in the sections that follow. You can assume that we will always specify valid corpus and input files when we run your program. For example:

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

For a refresher on how to process command line arguments in C, please refer to the PDF for Program #1 (Numeronym).

3. 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.
```

4. Building the Tries (and Subtries)

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.

For each sentence in the corpus, you must also 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.

5. Input File (Trie Processing Commands)

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. 7 and 8.)

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 8, "Function Requirements" (page 9).

6. Sample Input and Output Files

Consider, for example, the following corpus file and input (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 result in 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.

7. 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 for me 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!

8. 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 *processInputFile()*, but I do intend to deploy unit tests for that function 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 (or NULL if the specified file does not exist).

```
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 5, “Input File (Trie Processing Commands).” 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 5, “Input File (Trie Processing Commands).” 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*. You may assume *str* is not NULL.

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*.

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

Description: Counts how many new nodes would have to be dynamically allocated if we were to insert the given string (*str*) into the trie. If *str* is NULL, this function should simply return zero. Note that *root* could be NULL. Note also that this function should **not** add *str* to the trie. As usual, if there are any non-alphabetic characters in *str*, assume they would be ignored when inserting the string into the trie.

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 new nodes we would have to allocate if adding *str* to the given trie.

```
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 reasonable and realistic estimate (greater than zero) of the number of hours you spent on this assignment.

9. 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.

Note: Depending on your perspective, some people might find that this simplifies the assignment, while others might find this function unnecessary.

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

10. 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 *valgrind*, which is installed on Eustis.

Valgrind will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called *foo()* that has a nasty memory leak, but you run your program in such a way that *foo()* never gets called, *valgrind* won't be able to find that potential memory leak.

Also, note that if you do not use *fclose()* to explicitly close all open files before your program terminates, *valgrind* will most likely alert you that your program has a memory leak.

The *test-all.sh* script will automatically run your program through all test cases and use *valgrind* to check whether any of them result in memory leaks. If you want to run your program through *valgrind* manually, simply compile your program with the *-g* flag, and then run it like so:

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

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

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

For more information about *valgrind*'s output, see: http://valgrind.org/docs/manual/quick-start.html

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

As always, 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*.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *TriePrediction.c*, *TriePrediction.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*TriePrediction.c*, *TriePrediction.h*, *test-all.sh*, the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

> **Warning:** Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

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

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd TriePredictionProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

12. Checking the Output of Individual Test Cases

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. This section tells you how to do that.

There are two types of test cases included with this assignment: (1) the test cases where you compile your program and run it with a corpus file and input file (such as *corpus01.txt* and *input01.txt*) specified as a command line arguments, and (2) the test cases where you have to compile one of our source files (*UnitTest10.c* through *UnitTest17.c*) along with your source file (*TriePrediction.c*) in order to run.

If you want to run one of these test cases individually in order to examine its output outside of the *test-all.sh* script, here's how you do it:

1. **Instructions for running your program with corpus and input files specified as command line args:**

   a. Place all the test case files released with this assignment in one folder, along with your *TriePrediction.c* file.

   b. In *TriePrediction.h*, make sure line 18 is commented out *exactly* as follows, with no space directly following the "//". This is the *only* line of *TriePrediction.h* that you should ever modify:

   ```
   //#define main __hidden_main__
   ```

   c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program:

   ```
   gcc TriePrediction.c
   ```

   d. To run your program and redirect the output to *output.txt*, provide corpus and input filenames at the command line after `./a.out`, like so:

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

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/output01.txt
```

2. **Instructions for compiling your program with one of the unit test cases:**

a. Place all the test case files released with this assignment in one folder, along with your *TriePrediction.c* file.

b. In *TriePrediction.h*, make sure line 18 uncommented and appears *exactly* as follows. This is the *only* line of *TriePrediction.h* that you should ever modify:

```
#define main __hidden_main__
```

c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program with *UnitTestLauncher.c* and any *one* of the *UnitTestXX.c* files you would like to test:

```
gcc TriePrediction.c UnitTestLauncher.c UnitTest10.c
```

d. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/UnitTest10-output.txt
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

13. Style Restrictions (*Super Important!*)

These are the same as in the previous assignment. Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

★ Any time you open a curly brace, that curly brace should start on a new line.

★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.

★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

★ Please avoid block-style comments: /* *comment* */

★ Instead, please use inline-style comments: // *comment*

★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"

★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your *#include* statements.

★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be indented with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

14. Special Restrictions (*Super Important!*)

1. You must use tries in order to receive credit for this assignment.
2. Do **not** write to any files. You may read input files, but writing to files is prohibited in this assignment.
3. Note that there is no restriction on the lengths of the input file names that will be passed to your program; they can be arbitrarily long, so you should not hard-code any limits on those string lengths.
4. As always, you must avoid the use of global variables, mid-function variable declarations, *goto* statements, and system calls (such as `system("pause")`).
5. Do not write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
6. No crazy shenanigans.

15. 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. Be sure to include your name and NID in a comment at the top of your source file. Don't forget to *#include "TriePrediction.h"* in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also be able to compile and run on Eustis like so:

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

16. Grading

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

90%	Correct output for test cases (including unit tests)
10%	No memory leaks (passes *valgrind* tests)

Important Note! Additional point deductions may be imposed for poor commenting and whitespace. Significant point deductions may be imposed for violating the style restrictions or special restrictions listed above.

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!