CS 220 – Data Structures and Systems Programming – Spring 2020

PEX 4 – Spell Checker – 75 Points

Due @ 2300 MDT on 20 May 2020

Help Policy:

AUTHORIZED RESOURCES: Any, except another cadet's program.
NOTE:

- Never copy another person's work and submit it as your own.
- Do not jointly create a program unless explicitly allowed.
- You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook).

Documentation Policy:

- You must document all help received from any source other than your instructor or instructor-provided materials, including your textbook (unless directly quoting or paraphrasing).
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance, and HOW it was used in completing the assignment.
- If no help was received on this assignment, the documentation statement must state "None."
- If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.
- **Vague documentation statements must be corrected before the assignment will be graded and will result in a 5% deduction on the assignment.**

Turn-in Policies:

- On-time turn-in is at the specific day and time listed above.
- There is no late turn-in for the final PEX.

OBJECTIVES

Upon completion of this programming exercise, students will:

- Demonstrate understanding of a tree ADT through application to a specific problem.
- Implement a ternary tree ADT to store a dictionary.
- Create a thorough set of unit tests and utilize these to ensure code correctness.
- Create a command line argument driven program that implements a rudimentary spell checker.

BACKGROUND

This is a partner programming exercise. You may select your partner from any of the class sections, then send a joint email to all instructors with the partner names for approval. Additionally, you are required to review the "Git Concepts" document from CS210 and read the "How to Use Git in a Team" document. Once you've set up a PRIVATE repository with your partner you must provide access to the repository for all instructors.

This programming exercise is two-part, which requires you to store a dictionary of legal words from a text file and implement a basic spell checker application. The dictionary of words will be stored using a ternary tree ADT.

TERNARY TREE DEFINITION

A ternary tree is similar to a binary tree, except that nodes may have up to three children instead of two. The tree is constructed in such a manner that the entire subtree anchored at the center child of a node contains all suffixes associated with the prefix that led to the current node. For example, if the prefix is "succ", then the current node contains the second "c" of the prefix, and the subtree attached to the center child of the current node would contain the suffixes "eed", "ess", and "umb" to create the words "succeed", "success", and "succumb" (see Figure 1). For examples on how to build a ternary tree, as well as searching the tree, see the tutorial provided on the course web site.

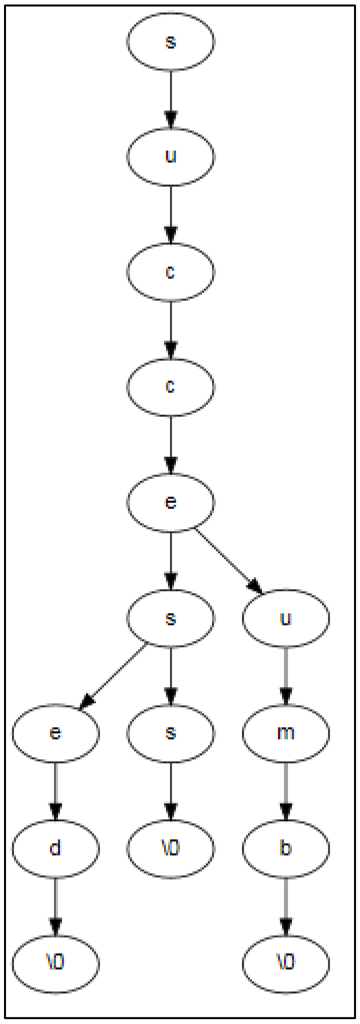


Figure 1

PROGRAMMING EXERCISE

For the programming exercise your executable will be launched from the command line and require two arguments in addition to the name of the executable. The first argument is the name of the dictionary file and the second is the name of the file containing the words to spellcheck.

You will decompose the problem and implement functions to perform the following operations on a ternary tree:

- Create a ternary tree
- Delete a ternary tree
- Insert a word into a ternary tree
- Check the spelling of a provided word using the dictionary stored in the ternary tree

**Note: An operation may require more than one function to be well decomposed.*

The order in which words are inserted into the ternary tree is ***crucial*** in maintaining a somewhat balanced ternary tree. Attempting to build the ternary tree by inserting the words in the order they are found in the text file (alphabetically) will result in an extremely unbalanced tree. What effect would this have on the Big-O of the operations performed on your tree?

To maintain a somewhat balanced tree you will need to begin by selecting the word in the middle of the provided dictionary as your starting point. Effectively, you have now split your dictionary into two and you'll continue selecting and inserting the middle word of successively smaller groups of words, much like searching a BST. However, you will need to return to the words not-yet inserted into your ternary tree. Think, recursion.

As an example, if the provided dictionary included the words shown in the image on page 2 of this write-up (succeed, success, succumb), they would be inserted in the ternary tree in the following order: success, succeed, succumb. Test 1, shown below, was written to verify the correct ternary tree insertion of these three words.

A full dictionary is provided in the file `dictionary.txt`, as well as a small test dictionary, `test_dictionary.txt`, for development purposes. The name of a dictionary file will be provided as a command-line argument when running your program. In addition to the dictionary file, your program must also accept a test file provided as a second command line argument. This file consists of various words to spell check, such as the ones provided in `test1.txt`. Your program will read the test file one line at a time, use the ternary tree to spell check all words, and output a list of the words checked and whether they were spelled correctly (Correct, or Incorrect). The output for a single word should be on the same line, with a carriage return at the end.

In addition to storing the dictionary as a ternary tree and implementing the functionality of the spell checker, you will write a thorough set of unit tests. In order to test your implementation, you will need to write a sufficient number of test cases to cover the required functionality including boundary conditions. Using a software development process called "test-driven development", where you write test cases before code implementation helps you know exactly what a function is supposed to do before you start implementing it. Please try to do test-driven development for this assignment.

A test consists of a specified input, and an expected output. For example, to write a test to ensure you are maintaining a balanced tree, you would first specify the inputs and expected outputs in a table (notice the binary search like selection of words to insert):

Test 1 Inputs	Test 1 Expected Outputs	Test 2 Inputs	Test 2 Expected Outputs
succeed success succumb	Inserting "success" Inserting "succeed" Inserting "succumb"	cab cad car cat cop cot cow cut	Inserting "cop" Inserting "car" Inserting "cad" Inserting "cab" Inserting "cat" Inserting "cow" Inserting "cot" Inserting "cut"

You will have to manually check your expected outputs against what your test program outputs. As you make changes to your program, make sure you run your test to ensure you haven't changed functionality that worked before.

Your unit test submission will consist of a single file named "ternaryTreeUnitTest.c" that runs a minimum of three tests:

1. Balanced tree insertion (this test doesn't have to actually insert words in the tree, just ensure that you are processing them in the right order given a list of words)
2. Tree insertion
3. Spell check

You will also submit a word document containing tables like the one above for each test that you perform. You should have multiple tests for each of the three items above.

HELPFUL HINTS

Write a **`loadDictionary`** method in your `pex3.c` file that reads the entire contents of the dictionary file into a list of words, and then uses a recursive helper function to determine the order words should be inserted into the dictionary. ***The logic of this code will be very similar to the binary search method used when searching a sorted array (see test cases on prior page and also be sure to follow the tutorial example).***

C is case-sensitive. The easiest way to avoid worrying about case is to use all capital letters in your tree. The provided dictionary file uses all upper-case characters, and you should do the same with any word patterns you use.

You can assume that all dictionary files that your program uses are saved in ascending sorted order.

Use helper functions where necessary to perform recursive tasks.

It is helpful to begin with a small dictionary in order to test your tree. Make sure you understand how words should be inserted into your tree before you start running tests against a large dictionary. NOTE: When you branch to a left, or right child the character of the parent node is NOT included in the word.

You can print every word in your tree using a pre-order traversal. This will give you an idea of the structure since the first word printed will be the one that terminates in the left-most subtree.

When you are inserting words from the dictionary into the ternary tree or searching for a word in the tree, remember to include the null character as part of your search. The null character is always the last character in a search pattern.

Remember to place the dictionary file in your PEXs/PEX4 folder and use that relative path to load it at runtime.

When reading a word from a file, you can strip the newline character at the end of the word using the following function:

```c
void stripNewline(char *word){
    char *newLine;
    if ((newLine = strchr(word, '\n')) != NULL) {
        *newLine = '\0';
    }
    if ((newLine = strchr(word, '\r')) != NULL) {
        *newLine = '\0';
    }
}
```

Passing arguments to a program using the command line:

Your program must get the name of the dictionary file and the name of the word-pattern file from the command line. However, do **not** program this functionality until after you have a fully working program. While you are developing and debugging your program, simply create appropriate variables in your program to represent the file names, such as:

```
char dictionaryFileName[] = "../PEXs/PEX4/dictionary.txt";
```

After you have completely tested the functionality of your program, modify it to get the two file names from the command line. Command line arguments are passed to the `main()` function using two parameters:

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

where:

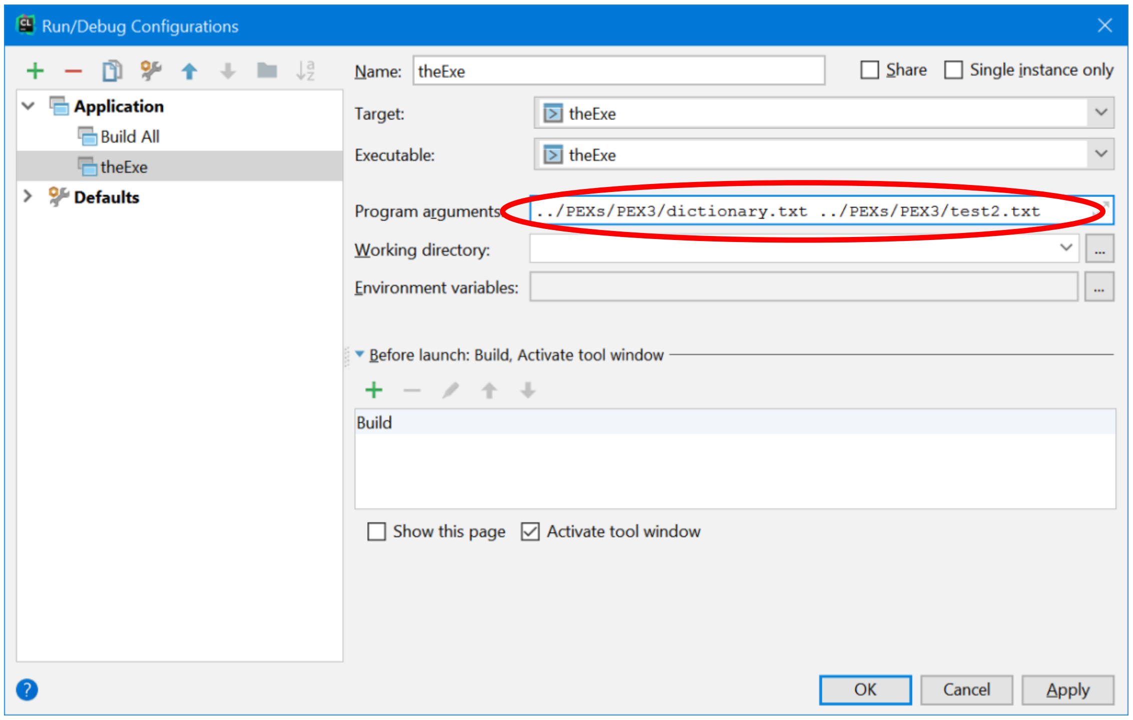
`argc` contains the number of arguments on the command line, and
`argv` is an array of strings, where each element of the array is one command line argument.

For example, if a command line contained:

```
myProgram.exe an example
```

then `argc` would be 3, `argv[0]` would be "myProgram.exe", `argv[1]` would be "an", and `argv[2]` would be "example".

If you want to specify command line arguments when executing a program from inside CLion, you must put the command line strings into the "Program arguments" edit box of the *run configuration*. Select `Run` → `Edit Configurations` and put the command line arguments in the indicated location below:


Run/Debug Configurations

- Application
  - Build All
  - theExe
- Defaults

Name: theExe

☐ Share ☐ Single instance only

Target: theExe

Executable: theExe

Program arguments: `../PEXs/PEX3/dictionary.txt ../PEXs/PEX3/test2.txt`

Working directory:

Environment variables:

Before launch: Build, Activate tool window

Build

☐ Show this page ☑ Activate tool window

OK Cancel Apply

SUBMISSION REQUIREMENTS

- Include your documentation statement in the comment header block at the top of the **`pex4.c`** file.
- Zip your code, data, and unit test documentation files. Submit your zipped file to the course website:
  - Possible code files: `ternaryTree.h`,`ternaryTree.c`, `ternaryTreeUnitTest.c`, `pex4.c`, `pex4.h`
  - Data files: `dictionary.txt`, `test1.txt` (and any other tests files you create), `CMakeLists.txt`
  - Unit test inputs: `UnitTests.docx`
- Each team member must fill out the pair programming analysis and email it to their instructor separately.
- Only one PEX submission per team.

PEX 4 Grade Sheet

Name:____________________________________

Criteria	Points Available	Points Earned
Coding standards	10%	
Your code follows coding standards (comments, naming & indention, etc)	**4**	
Partner programming analysis and shared Git repository	**4**	
Functionality	90%	
Submitted code compiles and runs (all or nothing in this category)	**10**	
Program reads input files correctly	**10**	
Program maintains a balanced tree when inserting from dictionary file	**10**	
Insert method functions properly	**10**	
Spell check method functions properly	**15**	
Sufficient number of unit test inputs/outputs are provided in the unit test tables	**12**	
Subtotal	**75**	
Vague/missing documentation statement (-5%)	**– 4**	
Submission requirements not followed (-5%)	**– 4**	
Late penalties – 50% Cap (may turn in 1 week later than due date / time)	**max score=37.5/75**	
Total	**75**	

Comments: