**CS 223 – Data Structures and Systems Programming**

**PEX 2 – Scrabble**® **Words**

**Due @ 2300 on Lesson 24**

**M-Day: Tue, 14 Mar 2017**

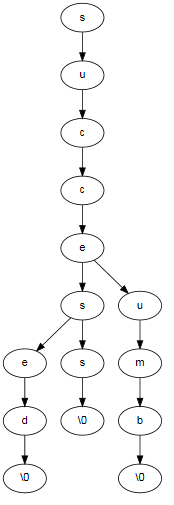
**T-Day: Wed, 15 Mar 2017**

|  |
| --- |
| 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). * **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**  Documentation Policy:  * You must document all help received from any source other than your instructor. * The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. * 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 time listed above. * Late penalties accrue at a rate of 25% per 24-hour period past the on-time turn-in date and time. The late penalty is a cap on the maximum grade that may be awarded for late work. * There is no early turn-in bonus for this assignment. |

1. Objectives

* Understand tree data structures and how they can be applied to a specific problem.
* Create a Ternary Tree data structure and use it to implement 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 Scrabble® word search application.
* Follow [CS223 coding standards](http://dfcs-moodle/mod/resource/view.php?id=5625)

1. Background

In this programming exercise you will implement a Scrabble® word search application using a Ternary Tree data structure to store a given dictionary of legal words.

In a game of Scrabble®, you score points by building a word using one or more letters from other words on the board. In order to do this, a player must be able to determine which words have certain letters in specified positions. For example, a game board with a “C” and a “T” separated by a space can support the words “CUT”, “COT”, and “CAT”. In order to determine which word will provide the maximum points in a Scrabble® game, a player would need to evaluate all the possible word scores and choose the word that gives the largest point value[[1]](#footnote-2). We will **not** be implementing a scoring mechanism in this exercise; we will implement the ability to generate all words that fit a given search pattern.

A search pattern consists of letters and wildcards. Your program must be able to find all words that have the specified letters in the pattern in the specified position. Using the example above, the pattern used to search for solutions would be “C.T”, where the period (.) is used as a wildcard position.

**Ternary Tree Definition**

A ternary tree is similar to a binary tree, except that nodes have 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 search using a pattern, see the tutorial provided on the course web site.

Figure

1. Programming Exercise

For the programming exercise you will implement the following methods of a **TernaryTree** data structure:

**TernaryTreeType \* ternaryTreeCreate();**

**void ternaryTreeDelete(TernaryTreeType \* tree);**

**void ternaryTreeInsert(TernaryTreeType \* tree, char \* word);**

**WordList \* ternaryTreeFindPattern(TernaryTreeType \* tree, char \* pattern);**

When launched, your program will create a Ternary Tree, read the dictionary file specified on the command line and then add the words from the dictionary into the tree. A full dictionary is provided in the file **dictionary.txt**, as well as a small test dictionary, **test\_dictionary.txt**. Your program will also read a test file provided as a second command line argument. This file consists of various word patterns 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 find all words that match the pattern, and output a comma-separated list of all matching words. The output for a single pattern should all be on the same line, with a carriage return at the end. (There should **not** be a comma after the last word.)

A pattern consists of letters and wildcards. The period (.) symbol is used as a wildcard, while characters should be matched exactly. For example, the pattern “C.T” will produce the output:

**CUT, COT, CAT**

Additionally, you will write a thorough set of unit tests for your **TernaryTree** data structure. In order to test your tree, you will need to write tests in an order that help you ensure you solve the problem in a logical sequence of steps. You will need to test your **ternaryTreeInsertBalanced** function, your **ternaryTreeInsertWord** function, and your **ternaryTreeFindPattern** function. (From CS210 you learned about "[test-driven development](https://en.wikipedia.org/wiki/Test-driven_development)" where you write the test cases before you implement the actual code. If you write the test cases first, then 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:

|  |  |  |  |
| --- | --- | --- | --- |
| 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. Pattern search

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.

1. Helpful Hints

A WordList data structure has been provided to help you store a list of strings. Study the WordList.h file to understand how to use this data structure. WordList.c.o is provided as an *object file* that your program can link to. In order to use WordList.c.o, you must include the file in your list of files in your CMakeLists.txt file:

**set(SOURCE\_FILE …, …, PEXs/PEX2/WordList.c.o, …)**

Write a **loadDictionary** method in your PEX2.c file that reads the entire contents of the dictionary file into a WordList and then use a recursive helper method that has extra parameters to determine the order words will 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***.

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 build 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.

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/PEX2 folder and use that relative path to load it at runtime.

When reading a word or a word pattern from a file, which are always on a separate line in the file, you can strip the newline character from the word using the following function:

**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[] = "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[])

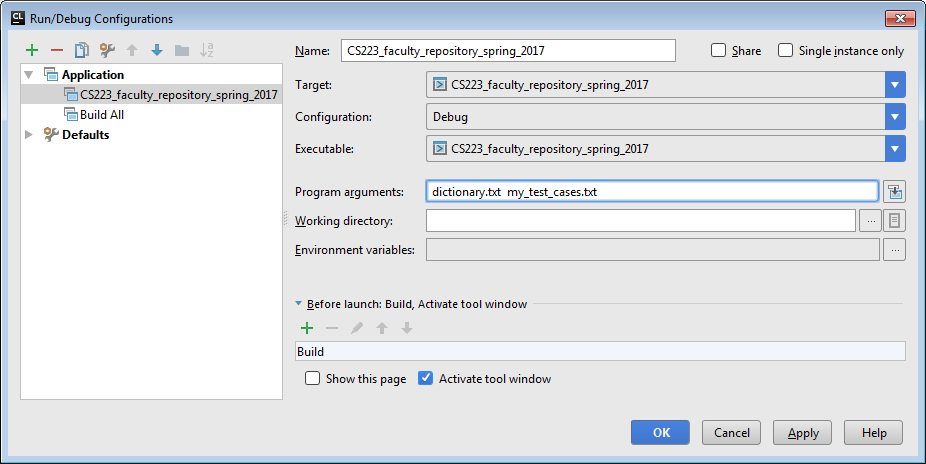
where:  
 argc contains the number of arguments on the command line, and  
 argv is an array of strings, where each element of the array is one command line argument.

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:



1. Submission Requirements

* Include your documentation statement in the comment header block at the top of the PEX2.c file.
* The PEXs/PEX2 folder of your source code repository must contain the following files. (Make sure you have used Git 🡪 Add to make them part of your repository.)
  + Code files: TernaryTree.h,TernaryTree.c,

TernaryTreeUnitTest.c,

Pex2.c

* + Binary files: WordList.c.o
  + Data files: dictionary.txt, test1.txt (and any other tests files you create)
  + Unit test inputs: UnitTests.docx

**Note**: The big idea is that your instructor can open your repository and execute your programs. If files are missing, the execution will fail, so please make sure that all of your PEX2 files have been added to your repository.

**PEX 2 Grade Sheet Name:**

Points

Criteria Available Earned

|  |  |  |
| --- | --- | --- |
| Coding standards | 10% |  |
| Your code follows the C coding standards. | **9** |  |
| Functionality | 90% |  |
| Submitted code compiles and runs (all or nothing in this category) | **10** |  |
| Program maintains a balanced tree when inserting from dictionary file | **15 ~~10~~** |  |
| Insert method functions properly | **20 ~~10~~** |  |
| Search method functions properly (i.e., pattern matching) | **11 ~~20~~** |  |
| Program reads input files ~~and runs tests~~ correctly | **15 ~~11~~** |  |
| Unit test inputs/outputs are provided in the unit test tables | **5** |  |
| All unit tests output the expected values as specified in the unit test tables | **5 ~~15~~** |  |
| Subtotal | **90** |  |
| Vague/missing documentation statement (-5%) | **− 4.5** |  |
| Submission requirements not followed (-5%) | **− 4.5** |  |
| Poor software development work habits (shown by your Git commit history) | **− 9** |  |
| Late penalties – 25/50/75/100% per 24-hours past due date/time | **− 23/45/67/90** |  |
| Total | **90** |  |

Comments:

1. Each letter in a Scrabble® game is assigned a point value (see <http://scrabble.hasbro.com/en-us/faq>) and the point value of a word is the sum of the points for the letters in the word. [↑](#footnote-ref-2)