# DSA555 Assignment 3

Weight: 7%  
Total Marks: 10  
Assigned: November 21, 2013  
Due: December 4, 2013, 11:59 pm

Failure to complete assignment 3 by December 6th, 2013 will result in a failing grade for this course.

# Late/Resubmission Penalties

|  |  |
| --- | --- |
| On or Before | Penalty |
| December 4 | No penalty |
| December 6 | No penalty (**this is not a typo**) |
| After December 6 | Cannot be submitted (and therefore **CANNOT** pass the course) |
| Resubmitted assignments | Max 5 marks |

# Dictionary Tree (10 marks)

Consider that in the English language, many words are extensions of other words. For example, the word “termless” is an extension of the word “term”. Exploiting this property, we can build predictive structures that can be used to list all words in the English language that are extensions of a given word.

Your part in this is to implement classes that implement the **iWordNode** and **iDictionaryTree** interfaces. The following sections describe iWordNode and iDictionaryTree.

## Interface class iWordNode

**NOTE:** **You do not code this class/interface, it is given to you.**

iWordNode is an interface class that encompasses the functionalities required to represent a single word part. An iWordNode is not responsible for holding a complete word or even a single letter, it is only responsible for holding a table of immediate children and whether it forms a whole word. Each node has a potential 26 children, one for each letter of the alphabet. The children are accessible by the letter that they are stored under.

**HINT: This means that ideally, the children should NOT be stored in a list!** Perhaps something more “tabular”/“Random-Accessible” could be used…

So, we can represent any word in the English language given a “root” word node and a sequence of word nodes to represent the characters of the word. For example, to represent the word “treeless” we would need the following word nodes.

In the following example, each pair of parenthesis represents a word node. Inside each pair, the letter Y indicates that this word node represents a whole word while the letter N indicates that this word node is only a word part. Additionally, the node may be labeled for clarity if it contains a “quoted” word. The square braces indicate which letter the word node is stored under.

(“root” N) -> [t](N) -> [r](N) -> [e](N) -> [e](Y) -> [l](N) ->[e] (N) -> [s](N) -> [s](Y)

Notice that the highlighted word nodes are marked with Y’s because they sit at the last character of a whole word. The first highlighted word node sits at the end of the chain t,r,e,e which is a whole word. The second highlighted word node sits at the end of the chain t,r,e,e,l,e,s,s which is also a whole word.

The most immediate application of an iWordNode tree is text prediction. By knowing what the user has already entered, we can quickly eliminate huge amounts of words and show to the user relevant longer words that they may be going for.

**NOTE: All functions that accept characters or words in iWordNode and iDictionaryTree can expect the incoming parameters to be all lowercased**

This interface has the following public pure virtual member functions:

virtual iWordNode\* getChild(char letter) = 0

* Returns a pointer to an iWordNode instance at the given letter. If none exist, this function should return nullptr

virtual size\_t getNumChildren() const = 0

* Returns the **TOTAL** number of children that this node and all of its subtrees contain

virtual bool isWholeWord() const = 0

* Returns true if this instance is at the end of a whole word, false otherwise

## Interface class iDictionaryTree

**NOTE:** **You do not code this class/interface, it is given to you.**

iDictionaryTree is an interface class for the actual tree itself. The iDictionaryTree is responsible for holding a root iWordNode and managing the tree overall.

**NOTE: All functions that accept characters or words in iWordNode and iDictionaryTree can expect the incoming parameters to be all lowercased**

This interface has the following public pure virtual member functions:

virtual void insert(const char\* word) = 0

* Inserts a “whole word” into the tree
* If the word is not already in the tree, the required word node chain is created and stored appropriately
* If the word is already in the tree but not marked as a whole word, it is marked as a whole word

virtual iWordNode\* getNodeForWord(const char\* word) = 0

* Attempts to find the node associated with a given word
  + A word is found by descending character by character into the word node tree from the root
* If the word is not found in the tree, nullptr is returned
* If the word is found, a pointer to it is returned

virtual iWordNode\* getRoot() = 0

* Simply returns the root node of the tree
* The tree should **always** contain a root node, even if it is empty
  + Remember, the character associated with a word node is only present on the word node’s parent! Therefore, we need an uncharacterized root at the top of the tree at all times!

## Allowed STL Elements

You may use **ANY** STL elements that you desire in the following classes.

## class WordNode : public iWordNode

Write the **WordNode** class. This class is a concrete implementation of the iWordNode interface. It should implement all of the pure virtual functions of iWordNode. Additionally, it should implement **at minimum** the following member functions:

WordNode()

* Default constructor
* No explicit requirements here except that this constructor construct this object safely and correctly

virtual ~WordNode()

* Virtual destructor
* Ensure that this class correctly deallocates any memory that it is responsible for
  + In typical situations, that would be this word node’s children

## class DictionaryTree : public iDictionaryTree

Write the **DictionaryTree** class. This class is a concrete implementation of the iDictionaryTree interface. It should implement all of the pure virtual functions of iDictionaryTree. Additionally, it should implement **at minimum** the following member functions:

DictionaryTree()

* Default constructor
* No explicit requirements here except that this constructor construct this object safely and correctly

virtual ~DictionaryTree()

* Virtual destructor
* Ensure that this class correctly deallocates any memory that it is responsible for
  + In typical situations, that would be this tree’s root with the root then automatically deallocating all of its children (as long as WordNode’s destructor is written correctly)

# Rubric

|  |  |
| --- | --- |
| Component | Weight |
| Does Not Compile on VS nor Matrix | **RESUBMISSION** |
| Does Not Pass Tester | **RESUBMISSION** |
| Logical Errors | **RESUBMISSION UNLESS VERY MINOR** |
| Passes Tester | **5** |
| Code Quality | **4** |
| Comments | **1** |
| Total | **10** |

If your assignment must be resubmitted for any reason (see above), it **CANNOT GET OVER 50%**.

# Submission Requirements

* Save your DictionaryTree and WordNode classes in DictionaryTree.hpp (I don’t want a .cpp file!)
* Include the filled in Seneca academic honesty submission form included below in honesty.txt:

Student Assignment Submission Form  
==================================  
I/we declare that the attached assignment is my/our own work in accordance with Seneca Academic Policy. No part of this assignment has been copied manually or electronically from any other source (including web sites) or distributed to other students.  
  
Name(s) Student ID(s)  
  
-------------------------------------------------------------------------------  
  
-------------------------------------------------------------------------------  
  
-------------------------------------------------------------------------------  
  
-------------------------------------------------------------------------------  
  
-------------------------------------------------------------------------------

* Submit the assignment on blackboard. If you can’t submit on blackboard, please let me know by emailing me.

In addition to the requirements set out on the assignment, your program must meet the programming standard that is posted on the course web page. The functions/classes must pass the testing provided by the test main.

Feel free to add helper classes/functions in addition to what is strictly required above.

That’s all! Please let me know as quickly as possible if you run into any problems.