Assignment 5 - TypeAhead

Submit Assignment

Due Monday by 11:59pm **Points** 50 **Submitting** a file upload
File Types hpp, cpp, txt, and in **Available** Oct 8 at 12am - Oct 23 at 11:59pm 16 days

The purpose of this assignment is to give you a chance to write some C++ code using classes, smart pointers, and exercising some data structure muscles. It will also give you some additional exposure to using some more parts of the STL. The inspiration for the program came from using the text messaging app on my phone and thinking about its use of prediction to aid my typing. For this assignment, you'll be doing one part of a text messaging app word prediction, a lexicographical search. A fuller featured predictor would also include frequently used words based on sentence structure and previous words used in context by the app user. It would be fun to do all of those things, but that's more than necessary for right now in this class.

When you see the program demonstrated you might be underwhelmed at first, but after you get it written, I think you'll be quite pleased at what it does and how fast it runs.

Assignment

Write a program that allows the user to begin typing a word and see a list of predicted words they may be attempting to type. The program begins by reading a list of words from a text file and storing them into an n-ary tree. Then, the user is presented with a blank screen, showing a cursor in the upper left. The user can begin typing a word and seeing a list of predicted words. After each key-press, the program uses the word tree to predict the next set of possible words they may be typing and displays them below the input line. See the **Sample View** section below for an example of how this looks.

The core of this assignment is a `WordTree` class, used to store the words in an efficient data structure. Read the section on **N-Ary Word Tree Structure** for how to build the tree. The `WordTree` class must expose the following public methods:

- `add(std::string word);`
  - Adds a new word to the tree. If the word already exists, it doesn't result in a duplication. If the tree is written correctly, no special code for handling a duplicate is necessary.
- `find(std::string word);`
  - Searches the tree to see if the word is found. If found, `true` is returned, `false` otherwise. (https://en.wikipedia.org/wiki/Breadth-first_search)
- `std::vector<std::string> predict<std::string partial, std::uint8_t howMany);`
  - Given the `partial` (or possibly complete word) word, returns up to `howMany` predicted words. The prediction must be a lexicographical prediction of the next possible words. This requires a breath-first search of the tree, after the node where the `partial` word ends.
  - Here is a wiki link on how to perform a breadth first search: https://en.wikipedia.org/wiki/Breadth-first_search (https://en.wikipedia.org/wiki/Breadth-first_search)
- `std::size_t size();`
  - Returns a count of the number of words in the tree.

The implementation of the `WordTree` must use smart pointers, no use of raw pointers is allowed.

The number of words to predict is the vertical size of the console window minus a few lines for the user input and the "--- prediction---" title. I also found it convenient to not display a word on the last row of the console to prevent drawing problems in my Linux console.

Console Rendering

You'll need to use a third-party header file that contains code for drawing to a console window. I have included the **rlutil.h** header file for your use. The license for the source code has some vulgar language, but it is a BSD-like license. I've made a few modifications to eliminate some warnings and add one keyboard definition. Use this code for drawing to the console. A few items you'll want to use from it:

- `rlutil::cls()` - Clears the console
- `rlutil::locate(...)` - Move the cursor to the specified location
- `rlutil::getkey()` - (blocking) Wait for the user to press a key
- `rlutil::setChar(...)` - Write a specific character at the cursor location
- `rlutil::KEY_DELETE`, `rlutil::KEY_BACKSPACE` - Keyboard codes for the delete and backspace keys
- `rlutil::trows()` - Number of rows in the console

The github location for this code is located at: **https://github.com/tapio/rlutil** **(https://github.com/tapio/rlutil)**

N-Ary Word Tree Structure

An n-ary tree (also m-ary or k-ary) is one where the number of children for each node can be 'n'. A bi-nary tree can have two children per node, a tri-nary tree can have three children per node. The following links provide additional information on n-ary trees:

- **https://en.wikipedia.org/wiki/M-ary_tree** **(https://en.wikipedia.org/wiki/M-ary_tree)**
- **https://leetcode.com/articles/introduction-to-n-ary-trees/** **(https://leetcode.com/articles/introduction-to-n-ary-trees/)**

For this assignment, 'n' is 26! A 26-ary tree, meaning 26 children per node. Each node in the tree represents a single letter of the alphabet. Additionally, each node contains a boolean indicating if it is the end of a word or not (true if it is, false otherwise). Each node (`TreeNode`) in the tree contains two data fields:

1. `bool endOfWord` - true if the node is the end of a word, false otherwise.
2. `std::array<std::shared_ptr<TreeNode>, 26>` children - Pointers to the children of this node. There are 26 children, each child represents one letter of the alphabet, 'a' to 'z'.

To store a word in the tree, you start at the root of the tree (a `TreeNode` itself), then select the child corresponding to the first letter. If this node is null, create the node, if it already exists, traverse. Then, select the child, of the new node, corresponding to the second letter of the word and do the same...if the node is null, create it, otherwise traverse. Continue this through the last letter of the word and then on that node, set the `endOfWord` boolean to true.

There is no need to store the letter the node represents, although you can if it helps you work with the tree. Implicitly, the position of the node as a child is the letter the node represents.

Using an n-ary tree in this way, creates an efficient data structure for both storage and searching of a large word list.

STL, Unit Tests, & Word List

A few items from the STL you might find useful or notice that I've used in the provided code:

- std::isalpha
- std::tolower
- std::transform
- std::accumulate
- std::all_of

There are unit tests associated with this assignment, you need to include and pass them. The source file for the unit tests is located in this file: **link**

You'll need a list of words to load into your tree. A relatively small one (a few thousand words) in this file: **link**. You can find a much larger dictionary (hundreds of thousands of words) at this location: **https://github.com/dwyl/english-words/** (https://github.com/dwyl/english-words/) (use words_alpha.txt).

Reading Word List

You can use the following code to read the words into your `WordTree`. This code assumes you have written the `WordTree` class and have the `add` method available.

Necessary includes: `<fstream>`, `<memory>`, `<algorithm>`

```
std::shared_ptr<WordTree> readDictionary(std::string filename)
{
    auto wordTree = std::make_shared<WordTree>();
    std::ifstream inFile = std::ifstream(filename, std::ios::in);

    while (!inFile.eof())
    {
        std::string word;
        std::getline(inFile, word);
        // Need to consume the carriage return character for some systems, if it exists
        if (!word.empty() && word[word.size() - 1] == '\r')
        {
            word.erase(word.end() - 1);
        }
        // Keep only if everything is an alphabetic character -- Have to send isalpha an unsigned char or
        // it will throw exception on negative values; e.g., characters with accent marks.
        if (std::all_of(word.begin(), word.end(), [](unsigned char c) { return std::isalpha(c); }))
        {
            std::transform(word.begin(), word.end(), word.begin(), [](char c) { return static_cast<char>(std::tolower
(c)); });
            wordTree->add(word);
        }
    }

    return wordTree;
}
```

Submission Notes

- Turn in the following files:
  - **main.cpp**
  - **WordTree.hpp**, **WordTree.cpp**
  - **TestWordTree.cpp** (provided for you)
  - **CMakeLists.txt**, **CMakeList.txt.in** (if you have one)
- Please have your code read ***EXACTLY*** "dictionary.txt". This way the grader can use the same word list for everyone while grading.
- CMakeLists.txt targets
  - Use ***EXACTLY*** `TypeAhead` as the name for the `add_executable` target for the application.
  - Use ***EXACTLY*** `UnitTestRunner` as the name for the `add_executable` target for the unit tests.
- Your code must compile without any warnings or compiler errors. See syllabus regarding code that has compiler errors.
- Your code must adhere to the CS 3460 coding standard: **link**
- Your code must be formatted through the use of the following clang-format configuration file: **link**

Sample View

The following shows an example of the program running. At this point, I have typed the partial word, "compu", the program is showing the predicted words below.

```
compu

--- prediction ---
compute
computed
computer
computes
computers
computing
compulsion
compulsory
computation
computational
```

Assignment 5 - TypeAhead

Criteria	Ratings		Pts
Compiles without any warnings (MSVC and g++). on Linux (-Wall -Wextra -pedantic) on Windows (/W4 /permissive-)	**2.0 pts** **Full Marks**	**0.0 pts** **No Marks**	2.0 pts
Follows required course code style. Must also be formatted using clang-format.	**2.0 pts** **Full Marks**	**0.0 pts** **No Marks**	2.0 pts
Passes all unit tests.	**2.0 pts** **Full Marks**	**0.0 pts** **No Marks**	2.0 pts
Correctly implemented 'add' method	**15.0 pts** **Full Marks**	**0.0 pts** **No Marks**	15.0 pts
Correctly implemented 'find' method	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Correctly implemented 'predict' method	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Correctly implemented 'size' method	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
User interface * Predict words after each keypress * Can use delete or backspace * Cursor is correct location etc	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
			Total Points: 50.0