Assignment 5

Binary Search Tree (or arbiny arches teer)

Purpose

You will need to use C++11 support for this exercise. For Eclipse and other GCC based compilers you will need to use the `-std=c++11` or `-std=c++0x` compiler option. This is not needed if you are using Visual Studio since it does not distinguish between the various versions of C++.

This assignment is worth 100 points. There is an optional part that is worth an extra 20 points. So, the maximum you can get for this project is 120 out of 100 points.

You will be creating a program that allows you to quickly solve Jumble or Scram-lets types of puzzles. To do this be using a binary search tree you will be creating. Your application will read in a list of words (over 35,000). You will sort the letters of each word to create a key. You will then add the key (and the word) to the tree (as one item called **Word**).

Example:

The word you read in from the file is the word “Binary”. You need to create a key from this. The key will be the word folded to lower case and ordered by its letters. So the word “Binary” first gets folded to lower case (binary) and then ordered by letter order – “abinry”. The key is “abinry” and the word is “Binary”. You need to create a class for this item (I will call it **Word**). The **Word** contains both the key (“abinry”) and the word text (“Binary”). This is added to the binary search tree in *key* order.

Once the dictionary of look up words has been build your application will prompt for a jumbed word. You will fold the jumbled word to lower case and sort the letters in the word. This will be what you look up in your dictionary of lookup words. If it is found you get the word text and that is your word unscrambled.

Example:

You type in Arbiny. This is folded to lower case (arbiny) and sorted (abinry). You look up the key abinry and it finds the word text Binary. So arbiny maps to Binary.

BinarySearchTree and **TreeNode** classes

First you need to create and test the **BinarySearchTree** class. The **BinarySearchTree** will contain **TreeNode** objects. These are both template classes.

These will be a template classes:

```
template<class DataType>
class TreeNode

template<class DataType>
class BinarySearchTree
```

If you want to make **BinarySearchTree** a friend class of the **TreeNode** you will need a forward declaration of the **BinarySearchTree**. So your header file may look similar to the following:

```
// forward declaration of the template class BinarySearchTree
template<class DataType>
class BinarySearchTree;

// TreeNode class
template <class DataType>
class TreeNode
{
      friend class BinarySearchTree<DataType>;
      // the rest of the TreeNode classs declaration goes here
};

// BinarySearchTree class
template <class DataType>
class BinarySearchTree
{
      // Your BinarySearchTree goes here
};
```

The **BinarySearchTree** class needs to implement the following member functions:

```
bool empty() const
```

The empty member function will return true if the binary tree is empty.

```
std::size_t size() const
```

Returns the number of items in the binary tree.

```
void print() const;
```

Display the contents of the binary tree in order by key (inorder traversal). Whatever is being stored in the binary tree will have to have overload the operator<< operator.

```
void debug(std::ostream &out) const;
```

Display the contents of the binary tree for debugging purposes. Use whatever traversal algorithm makes sense to you. Make sure you output the left and right pointers from the **TreeNode**. Do this for all nodes in the binary tree. Note, this could be a lot of output for a binary tree that has hundreds or thousands of entries. You may want to output this to a file (that is, you may want to pass an **std::ofstream** to the debug member function). Since **std::ostream** is a parent to **std::ofstream** you can do this and you should not change the signature of the debug member function. See the debug code provided with assignment-3 for help in writing this debug member function.

```
bool find(const DataType &searchItem,
          void (*foundNode)(const DataType&));
```

The first parameter passed to the find member function is the item we are looking for. The second parameter is the address of a function that will be called if the **searchItem** is found in the binary tree collection. The found item is passed to the function. If the item is found, the function is called, and the **find** member function will return **true**. If the item is not found, the function is not called, and **find** returns **false**.

Here is an example that uses the **find** member function:

```
// prototype
void processFound(const std::string &item);
…
// declaration of binary tree
BinarySearchTree<std::string> tree;
…
bool result = tree.find("555-122-3333", &processFound);
// result will be true if item "555-122-3333" was found and
// the processFound function was called.
```

Note that the function you pass to **find** is either a stand-along function (such as **processFound** above) or a **static** member function in a class. It cannot be a regular member function, it has to be a **static** member function.

```
bool erase(const DataType &deleteItem);
```

The **erase** member function will remove the specified item from the binary tree (if it finds the entry in the tree) The **erase** member function will return **true** if the item was found and removed and **false** if it was not found.

```
void insert(const DataType &newItem);
```

Insert the item into the binary tree. If the entry already exists it should be replaced by the new one.

```
void insert(const DataType &newItem,
            void (*duplicateItemFound)(DataType &existingItem,
                                       const DataType &newItem));
```

Insert the item into the binary tree. If the item already exists the **duplicateItemFound** function will be called. This is the function passed as the 2nd parameter to the **insert** function. Note this must either be a stand-alone function or a **static** member function. The signature of the function is as follows (the function name can be different):

```
void update(DataType &exsitingItem, const DataType &newItem);
```

Your **update** function will get a modifiable version of the found item (**existingItem**) and a **const** reference to the new item. You can update the **existingItem** as needed in the **update** function.

Here is an example using the `update` function above:

```
// declaration of binary tree
BinarySearchTree<std::string> tree;
…
tree.insert("555-122-3333", &update);
```

The `update` function is called if item `"555-122-3333"` already exists in the tree.

```
void traverse(void (*itemFound)(const DataType& foundItem)) const;
```

The traverse function will do an inorder traversal of the binary tree. For every item found it will call the **itemFound** function and pass it a **const** reference to the found item.

Constructors and destructor

You will need a default constructor, a copy constructor, and a destructor.

The copy constructor needs to make a deep copy of the tree. You should be able to use other member functions to do most of the work for you.

Make sure your destructor removes any remaining items from the tree.

Other information

You can add other member functions as needed. Make sure you free up any memory you allocate.

You should write stubs for all of the functions as you are creating the classes and then incrementally implement the member functions for the **TreeNode** and **BinarySearchTree** classes. By doing this incremental development you will have a better idea which function is failing (because it is probably the code you just wrote). Also, make sure you test and fix a member function before you go onto the next one. Also, look for opportunities to reuse code you already have in the class to implement other member functions.

Use good formatting in your programs. Use meaningful variable names and comment your code.

You can inline trivial member functions in the **TreeNode** and **BinarySearchTree** classes (trivial being 1 or 2 lines of code).

You must implement all of the member functions shown here. You can implement additional private, and protected member functions if you need them.

Create a header file named **BinarySearchTree.h** that contains your **BinarySearchTree** and **TreeNode** class definitions and any implementation code needed. Include the required header file including any guards needed.

Please write a driver function (main) to test your **BinarySearchTree** class. In the next part you will be creating an application that uses the **BinarySearchTree** and **TreeNode** classes.

Word Unscrambling Program

Now that you have a working **BinarySearchTree** class you need to create your word unscrambling program. The way this program will work is as follows:

You will be reading a file, "**english_words.txt**", that contains a number of words (over 35,000 words). You will read in a word and make a copy of it. The copy of the word needs to be folded into lower case (see the **std::tolower** function in header file **cctype**. This will take one character and convert it to lower case. You will need to do this for every character in the copy of the word you read in. After you have folded the copy of the word to lower case you need to sort the characters in the string in character order. So if the word is "Help" you will make a copy of this word (call it key) and fold it to lower case giving you "help". Now you sort the characters and you end up with "ehlp". Now you have a key, "ehlp" and the original word "Help".

You need to create a class to contain both of these values. You will add an instance of this class to your tree.

For our discussion assume we call this class **Word**. The constructor for the **Word** class will take the word passed to it and determine the key for that word. The resulting **Word** object will contain both the key and the original value for the word.

Assume we can create a **Word** as follows:

```
Word newWord("Help");
```

The **Word newWord** will have a key of **ehlp** and a value of **Help**.

We can insert the **Word** item into our binary tree. Assume the tree is defined as follows:

```
BinarySearchTree<Word> dictionary;
```

We can add the word with:

```
dictionary.insert(newWord);
```

For insert to work properly you will have to have your **Word** class override **operator==** and **operator<** (at the very least). If your **BinarySearchTree** class uses comparison operators in addition to these you will need to implement them as well.

For the **print** and **debug** member functions in your **BinarySearchTree** class you work you will also need to implement:

```
std::ostream& operator<<(std::ostream &out, const Word &outputWord)
```

When working with a **std::string** you can access and update individual characters using either the **at()** member function or the subscript operator. Here is an example:

```
std::string text("Hello");
text[0]= 'h';
std::cout << text << std::endl;
```

will output

```
hello
```

Also, the algorithms header file has a function called **std::sort**. This required two iterators to sort a collection of values.

The **std::string** class is a type of collection. You can get an interator to the first character in the **std::string** by calling the **begin**() member function. You get the end of the string with the **end**() member function. Note that **end**() does not give you the last character in the **std::string** but an indication that you are beyond the end of the string. These two member functions are exactly what you want for the **std::sort** function.

So you can sort the characters in a **std::string** as follows:

```
std::string myText("dallas");
std::sort(myText.begin(), myText.end());
std::cout >> myText >> std::endl;
```

will output

```
aadlls
```

You will need processing similar to the above in your **Word** class. You can create the class with a different name if desired.

Once you have read in all of the words, created your **Word** objects and added them to your **BinarySearchTree** you need to do the main processing for the application.

You need to read in scrambled words and see if they are in your binary tree.

You need to read in the scrambled word and sort the characters of the word into order by characters (you are creating the key for the scrambled word). If that key exists in your tree you now know the word that that key maps to and you have unscrambled the word.

Assume the word “Help” is in the tree with a key of “ehlp”.

Your program asks for a scrambled word and the user types in:

Pelh

Your program should fold this to lower case and sort the letters. The resulting key is “ehlp”. When you look for this in your binary search tree it will be found and the resulting word will be “Help”. You have unscrambled “Pelh” into “Help”.

Doing all of the above is worth 100 points

Optional (20 extra points):

But, what about duplicate words? The scrambled word **sdie** could map to **side**, **dies**, **ides** and **desi**. All of these are in the input file. Which one should you keep? The default behavior of insert is to replace the

existing entry with a new entry. So, by default, the last one of these 4 words that are in the input text file would be word that is found.

You can also update your **Word** class (or whatever you called it) to support multiple words for one key. You could then use the 2nd version of the insert to have it call a function you write. You could them update the found entry and add any new words to it.

You would do this by changing your word class to use an array or a **std::vector** to allow more than one word to be mapped to a key.

Add support to your Word class to allow multiple words to be associated with one key. This will allow you to display multiple solutions.

Here is some sample output:

```
Enter a word to unscramble: sdie[Enter]
The scrambled word sdie maps to words: side dies ides desi
```

There are some sample scrambled words you can try out. These are in file “scambled_words.txt”.

Submitting your work

Submit all of your header and source files to eLearning to get credit for this project. If you do the optional part you need to submit the original files in one submission to eLearning and the optional version in a second submission to eLearning.

Let me know if you have any questions.