CMPT 225 Assignment 3 - Translator

Please, read the entire assignment first before starting it!

This assignment has two parts. Part 1 must be done first before you move on to Part 2.

Part 1

Objectives

The objectives of Part 1 of our Assignment 3 are for you to practise ...

- implementing recursive and iterative methods for a Binary Search Tree (BST) ADT class.
- using C++ exception handling mechanism.
- solving a problem going through the steps of the software development process.

Requirements

1. In Part 1 of our Assignment 3, your task is to design, implement and test a Binary Search Tree (BST) as a ADT class using the provided files.
   This Binary Search Tree (BST) class is a link-based implementation in which duplicated elements are not allowed.

It makes use of wrappers (as discussed in class) calling recursive methods.

2. Download this zip file into your sfuhome/cmpt225/A3/Part_1 folder and extract its content. You will see the following files:

   - A complete EmptyDataCollectionException class
   - A complete ElementAlreadyExistsException class
   - A complete ElementDoesNotExistException class
   - A complete UnableToInsertException class
   - A complete BSTNode class (BSTNode.h and BSTNode.cpp)
   - A complete WordPair ADT class (WordPair.h, WordPair.cpp), which can be used to test your BST class.
   - A complete makefile
   - An incomplete testDriver.cpp, which you can make use of or create your own. If you make use of this provided test driver, you will need to add more code to it.
   - A test input file dataFile.txt, which contains several word pairs (first an English word, then a colon (:), then the corresponding Klingon word).
     We shall discuss word pairs in more details in Part 2 of this Assignment 3.
     Feel free to create your own test input files. When you do so, make sure you follow the same format:
     <word1>:<word2>
   - An incomplete BST ADT class (BST.h, BST.cpp) for you to complete.
     Notice the public methods insert(...), retrieve(...) and traverseInOrder(...) are wrapper methods, wrapping around their recursive counterparts which are the private utility methods insertR(...), retrieveR(...) and

traverseInOrderR(...), respectively. This signifies that you must recursively implement insertR(...), retrieveR(...) and traverseInOrderR(...).

3. Open these files and read their documentation, comments and code. All the instructions you need in order to do this first part of our Assignment 3 are included in these files.

4. Lastly, your code must not print anything if the documentation does not requires it to do so.

5. Once you have successfully implemented and tested your BST class, move on to the next part of this Assignment 3.

Part 2

Objectives

The objectives of Part 2 of our Assignment 3 are for you to practise ...

- solving a problem going through the steps of the software development process, and in doing so

- implementing a data collection Dictionary ADT class, i.e., an appropriate data collection for the problem we are asked to solve.

Problem Statement

In this part of our Assignment 3, we are asked to solve the problem of translating English words into words of other languages.

The problem can be generically described as follows: given a key (in this assignment: an English word), return its associated value (in this assignment: a word in another language).

This type of problem is best solved using a Dictionary, i.e., a data collection that associates a value to a given search key, just like a dictionary does in the real world. Indeed, we look up a word (the search key) in a dictionary to find its value, i.e., information about the given word such as its definition, its origin, its pronunciation, etc.

Requirements

1. Select the *other language* into which your Translator will translate English words.
   You can select a natural language such as Spanish, French, Italian, Mandarin, Japanese, etc... Or you can select a language from other galaxies/planets such as Vulcan, Cardasian (Kardasi), etc...
   Note that you cannot select the Klingon language since I have already chosen this language and will be using it to illustrate the behaviour of our Translator Application in this part of our Assignment 3.

2. Create an input data file (must be called **myDataFile.txt**) containing a minimum of 40 pairs of words: an English word and its translation in the other language you have chosen. Follow the format of the test input file (dataFile.txt) provided in Part 1 of this assignment: one pair of words per line and separate the words using a colon ":", i.e., <English word>:<word in the other language>

3. Download Dictionary.h, open it and read its content. Create its implementation file **Dictionary.cpp** using all the files you downloaded and completed in Part 1.
   As you can see, this Dictionary class is implemented using a BST object as its underlying data structure (CDT). This is to say that when we implement the public methods of this Dictionary class, all we shall do is call the public methods of the BST class.
   This is very similar to the List-based implementation of the Stack class and of the Queue class we saw in class.
   Also, make sure you add appropriate documentation to the methods of the Dictionary class such as description, precondition (if any), etc.

4. Download this complete makefile - It is not the same as the makefile you downloaded for Part 1.

5. Finally, create a Translator Application (must be called **Translator.cpp**) which must instantiate and use an object of your Dictionary class.
   Hint: The testDriver.cpp provided in Part 1 can certainly inspire you when you are creating this Translator.cpp.

   This Translator Application must have the following behaviour:
   `Translation Algorithm:`

6.
7. `While not EOF`
8. `  Ask the user for an English word.`
9. `  Read the English word the user entered.`
10. `  Translate this English word using the dictionary object and print the English word's translation on the computer monitor screen.`
11. `  If the English word was not found, print ***Not Found!*** instead.`
12.
    Hint: Typing **CTRL+D** at the command line creates an **EOF** character.
    Here is an example using the Klingon language as the other language:
    `uname@hostname: ~$ ./translate`
13. `laser`
14. `laser:'uD'a'`
15. `today`
16. `today:jajvam`
17. `apple`
18. `***Not Found!***`
19. `<CTRL+D>`
20.
    If the user enters the word **display** on the command line, the Translator Application then prints all the pairs of words in

ascending alphabetical sort order based on the English word of the pair and terminates.
Here is an example, if the data file only contained the following 4 pairs of words (again, using the Klingon language):

```
top:yor
telephone:ghogh HablI'
horn:gheb
today:jajvam
```

then entering the following at the command line

```
uname@hostname: ~$ ./translate display
```

would produce the following output:

```
uname@hostname: ~$ ./translate display
horn:gheb
telephone:ghogh HablI'
today:jajvam
top:yor
```

Questions

- What is the time efficiency of the **display** feature described above?

- Notice that there is no code actually doing the printing in the BST ADT class i.e., this BST class is totally i/o free. How can we achieve this? How is it done?

- Why is it advantageous to keep classes i/o free?

- We know that the time efficiency of the best case scenario of inserting an element into a BST and retrieving an element from a BST is O($\log_2$ n. How can you guarantee that each BST insertion and retrieval will be done in O($\log_2$ n)?
  In other words, how can you guarantee that your BST will not degenerate into a linear data structure as you insert elements into it, hence producing the worst case scenario of BST insertion: a time efficiency of O(n)?

One way to avoid the BST's worst case scenario would be to implement an AVL tree instead of a BST. But since we are not asked to implement an AVL, there must be another way.

Marking Scheme

When marking Assignment 3, we shall consider some or all of the following criteria:

- Data file: Is the data file structured in such a way that BST's insert and retrieve methods both have a time efficiency of $O(\log_2 n)$.
- Compilation: Does your code compile?
- Execution: Does your code solve the Translator problem described in this assignment by producing the expected results also described in this assignment?
- Correctness: Do your ADT classes abide to their given public interface?
- Requirements: Does your solution satisfy the requirements described in this assignment?
- Coding style: Has *Good Programming Style*, described on the Good Programming Style web page of our course web site, been used?
- Documentation: Does your solution satisfy the documentation requirements described in this assignment?
- Note that you cannot make use of the C++ STL library nor can you use code that has not been written by you in the context of this course (this semester) or provided by this instructor.

Submission

Assignment 3 is due Monday, Oct. 31 at 23:59:59 on CourSys.

For Part 1: you need to submit **BST.h**, **BST.cpp**.

For Part 2: you need to submit **Dictionary.h**, **Dictionary.cpp**, **myDataFile.txt** and **Translator.cpp**.

Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted the following Thursday) in order to provide feedback to the students.

Remember our *target machine*?

You can use any C++ IDE you wish to do your assignments in this course as long as the code you submit compiles and executes on our ***target machine***, i.e., the CSIL workstations running the Ubuntu Linux platform (O.S.).

Why? Because your assignments will be marked on the ***target machine*** (Ubuntu Linux, C++ and g++ versions running on the CSIL workstations).

Also, you must use the provided makefile(s) when compiling your code, if there is one provided. Otherwise, you are free to create your own.