O COMP 215 Algorithms Θ

Project #1

The submit date of this project is not yet set. Keep looking on onCourse, this file will be periodically updated with additions to the project (there's no point having the whole thing right now since you wouldn't know how to do some of the operations). I think it is unlikely that it will be due before November 19.

For this project, I ask you to write all your code *on your own.* Feel free to discuss with your classmates about the *concepts*, but if you have questions about your *code*, ask me. Likewise, feel free to look on the internet for inspiration about the code, but you *have to* use the class names, function names and variable names I give you, and you should implement the functions *as I described them in class*, so do not get *too much* inspiration from the code you see on the web.

In this project, you will be implementing a dictionary. This data structure will have 3 main functions: searching for a word (which will return a definition), adding an entry to the dictionary and removing an entry from the dictionary. There will also be a few functions allowing the storage of the dictionary to a file and reading dictionary entries from a file. The data structure used by the dictionary will be a Red-Black Tree. But let's do things one at a time.

Since this is your first big project, we will do things properly and break it into multiple files. You *must* break the project into various files exactly as I describe here. Also use exactly the same names for files, classes and variables (when applicable) as I describe here.

First, let's create the data structure to hold the dictionary entries.

1. In a file called `dictEntry.h`, write the prototype for a class `dictEntry`. This file must not contain the implementation of *any of the functions* (not even the constructors).

2. This class will have 2 protected data members: a string `word` and a string `definition`.

3. The class will also have the following public functions:

   - a constructor that takes 2 strings, `w` and `d` which initializes `word` and `definition` respectively
   - a function `getWord` which takes no input and returns the word string
   - a function `getDefinition` which takes no input and returns the definition string
   - a function `updateDefinition` which takes a string input `def` and updates the definition string
   - all the operators ==, <, >, <= and >= overridden. Each operator should take a *string* as input and compare that string (using lexicographic order – the order given by the usual string comparison function) to the word string of `this`.

4. In a file called `dictEntry.cpp`, implement all the functions described in `dictEntry.h`

We are going to do things pretty generally here, and our dictionary will be allowed to use any kind of search tree as its data structure (not even necessarily a binary search tree), so we are going to define `abstract classes` for search trees.

5. In a file called `treeNode.h`, write a class `treeNode`. This class only has a public data member: a pointer to dictEntry `data`.
6. In a file called `searchTree.h`, write an abstract class `searchTree`. This will be a fully abstract class with none of the functions implemented.
7. The class contains the following *pure virtual functions* (see the section on Abstract Base Classes in the C++ tutorial):
   - an `insert` function that takes a pointer to dictEntry as its input and has no output.
   - a `search` function that takes a string as input and returns a pointer to a treeNode.
   - a `remove` function that takes a string as input and returns no output.
   - functions `preOrder`, `postOrder` and `inOrder` that take no input and, for now at least, return no output.

Next, we need to finish the implementation of the binary search tree we started a while ago. Why complete it? because Red-Black trees will be a derived class from binary search trees of course!

First, the tree nodes:

8. In a file called `searchTreeNode.h`, write the prototype for a class `searchTreeNode` that is derived from `treeNode`. This file must not contain the implementation of *any of the functions* (not even the constructors).
9. This class will have 3 additional public data members: 3 searchTreeNode pointers `left`, `right` and `parent`.
10. The class will have 3 public constructors:
    - a constructor that takes only a pointer to a dictEntry `d` as input, and initializes `data` to `d` and sets all the pointers to NULL
    - a constructor that takes a pointer to a dictEntry `d` and a pointer to a searchTreeNode `p` as input, and initializes `data` to `d`, `parent` to `p` and `left` and `right` to NULL
    - a constructor that takes a pointer to a dictEntry `d` and 3 pointers to a searchTreeNode `l`, `r` and `p` as input and sets `data`, `left`, `right` and `parent` in the obvious way
11. In a file `searchTreeNode.cpp`, implement all the constructors described in `searchTreeNode.h`

Now for the binary search tree itself.

12. In a file called `binarySearchTree.h`, write the prototype for a class `binarySearchTree` which is derived from `searchTree`. This file must not contain the implementation of *any of the functions* (not even the constructors).

13. The class has only 1 protected data member: a pointer to a searchTreeNode `root`.
14. The class has only 1 public constructor: a constructor that takes no input and sets the root to NULL.
15. The class has lots of functions:
    - A public virtual function `insert` that takes a pointer to a dictEntry `in` and inserts it in the tree. This function has a corresponding protected helper function `insert_h`.
    - A public virtual function `search` that takes a string `w`, searches for it into the tree and returns a pointer to the node whose dictEntry's word member is equal to `w`. This function has a corresponding protected helper function `search_h`.
    - A public virtual function `remove` that takes a string `w` and removes the node in the tree whose dictEntry's word member is equal to `w`. If no such node exists, the function does nothing.
    - Public virtual functions `preOrder`, `postOrder` and `inOrder` with their corresponding protected helper functions `preOrder_h`, `postOrder_h` and `inOrder_h` (though, for in-Order, the helper function is not strictly necessary, feel free to do everything in the public function if you feel like it)
16. In a file `binarySearchTree.cpp`, implement all the functions described in `binarySearchTree.h`

And now for the Red-Black tree. First, we have to make a class for its nodes:

17. In a file called `RBsearchTreeNode.h`, write the prototype for a class `RBsearchTreeNode` that is derived from `treeNode`. This file must not contain the implementation of *any of the functions* (not even the constructors).
18. This class will have an additional public data member: a string `color`.
19. This class will have the same constructors as the searchTreeNode, so each constructor might as well call the parent constructor, but in addition, all the constructors should set the color field to the string "RED".
20. In a file `RBsearchTreeNode.cpp`, implement all the constructors described in `RBsearchTreeNode.h`

Finally, the Red-Black trees themselves. You might think if would make sense to have Red-Black trees be derived from binary search trees, but since the nodes in red-black trees are different from the ordinary binary search trees, it makes more sense to have them derived only from search trees to make sure that the root has the proper type.

21. In a file called `RBSearchTree.h`, write the prototype for a class `RBSearchTree` which is derived from `searchTree`. This file must not contain the implementation of *any of the functions* (not even the constructors).
22. The class has only 1 protected data member: a pointer to a RBsearchTreeNode `root`.
23. The class has only 1 public constructor: a constructor that takes no input and sets the root to NULL.

24. The class will have the same functions as binarySearchTree (though, you don't know how to implement those yet)

25. In a file `RBSearchTree.cpp`, implement all the functions described in `RBSearchTree.h` (though, you don't know how to implement those yet)

At long last, we have the class for the dictionary itself.

26. In a file called `dictionary.h`, write the prototype for a class `dictionary`. This file must not contain the implementation of *any of the functions* (not even the constructors).

27. The class has 1 protected data member: a pointer to a searchTree `dict`.

28. The class has 1 public constructor that takes a pointer to a searchTree and initializes `dict` to that pointer (this will let the user choose the type of search tree he wants the dictionary to use).

29. The class has the following functions:
    - a `search` function that takes a string and returns a string
    - an `add` function that takes a dictEntry and returns nothing
    - a `remove` function that takes a string and returns nothing

30. In a file `dictionary.cpp`, implement the constructor. Do nothing for the other functions for now, I will fill in the details later

That's it for now. More details to come later!