CS 220 - Homework 7

Due: Friday, November 16th by 11pm
This assignment is worth 60 points.

Learning Objectives

This homework will give you practice with operator overloading, C++ templates, linked structures, dynamic memory allocation, and recursion. You will use git for version control.

Important Note: This homework comes with three starter source files (“CTree.h”, “CTreeTest.cpp”, and “TTreeTest.cpp”) . You should get these files from the public repository under homework/hw7. Make sure you have a proper copy of these files before you start writing any code.

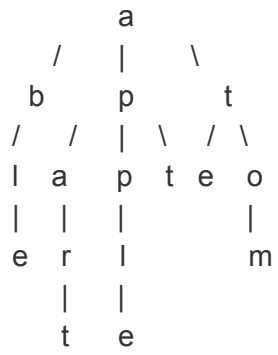
Part 1:

Create a class called **CTree** which is a dynamic tree of characters. A *tree* is a data structure which begins with a root node containing data (a character in this case), and then links to children nodes (like a single parent family)[1]. Each child node in turn contains data and may have its own children. We refer to all the children of a particular node as siblings of each other. (Other family terminology such as parent node, descendents, ancestors, etc. apply in the expected ways.)

In an ordered tree, the siblings are always ordered according to some metric (ASCII ordering in this case). Our tree will also be special because the children of any particular node must contain unique data. So essentially the children form a set of data. Your class must be declared and implemented in files "CTree.h" and "CTree.cpp", respectively. We will give you the “CTree.h” though.

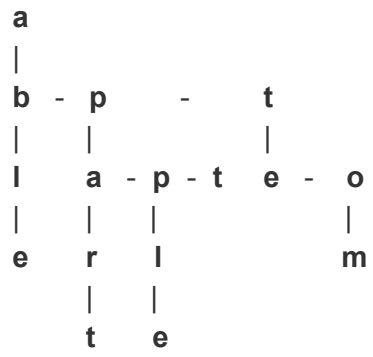
Visually, we may think of a character tree in a top-down fashion as depicted here. In this case 'a' is the root node of the entire tree, but each node is the root of the subtree below it. As such, trees are naturally recursive data structures. Note that the topmost root node in a tree should not have siblings.

[1] You can read more about trees here: https://en.wikipedia.org/wiki/Tree_(data_structure).



An ordered character tree that is used in a particular way to store string information is called a "Trie" (pronounced "try"). You can store multiple strings from some text in a Trie by using a path from the root to a leaf node (one without any children) in order to spell each string in the text. For example, the strings stored in the tree above are "able", "apart", "apple", "apt", "ate", and "atom". Notice that by iterating through the nodes in a depth-first, left to right manner, we get an alphabetical ordering of the strings it represents.

Your CTree class must have private data to store a character and its children information. Really we will treat each node of the tree as a (sub)tree itself. That way we only need one class instead of separate node and tree classes, and will be able to use recursion for much of the work. However, the implementation picture is somewhat more complicated than that above since we don't know how many children any given node in the tree will have. Therefore, we will basically use a linked list to keep track of the children of a node, transforming the picture above into the following:



Here, each node keeps a link to its first child, which then links across to the next sibling in the chain, and also down to its own first child if it has one.The last node in a chain points to NULL. Each node should also keep a link to its preceding node. That is the parent if the node is a first child, or its left sibling if the node is not a first child. **We are giving you the header file for this class in order to help with setting up the recursive structure properly.**

Minimally, here are the functions that your CTree must have, keeping in mind that at all times the siblings of any node must be ordered and unique. You should only use the assignment (=), equals (==), less than (<) and output (<<) operators to implement these functions. You are welcome to create helper functions and variations as well.

- A constructor that is passed the character to store in the root node of a new tree. Consider adding parameters for the links in the node also, all with default values of NULL.
- A destructor that clears all the children of the current node and all the siblings to the right of the current node, freeing that dynamically allocated memory. (Hint: use recursion!)
- An 'addChild' function, which is passed the character data to be stored in the child. This function returns false if it fails (because the data is already a child of this tree node) and true if it succeeds. Remember that the class must keep the children ordered.
- An (overloaded, second) 'addChild' function, which is passed the root of an existing tree to be stored as a child of this node. This function returns false if the root is invalid (has prev or siblings), or if the data in the root node is already a child of the tree node. Otherwise it adds the node in order and returns true.
- Two *private* 'addSibling' functions. One is passed a character to be stored in a new node, and the other is passed the root of an existing tree. Each of these functions returns false if it fails (because the data is already a sibling to the right of this tree node, or the root node is invalid) and true if it succeeds. Remember that siblings must always be ordered. (Hint: the character version can simply create the node, call the other version and then return the resulting value.) Note: these functions can be useful in implementing the add functions.
- A toString() function that creates and returns a string of all the elements in the nodes of the tree, separated by newline characters. The ordering of the nodes must be determined by a depth-first pre-order traversal from the root node (see below).
- An overloaded ^ operator that does the same thing as addChild. The difference is that instead of returning a boolean, it returns the resulting CTree. You should not change the function signature of *operator^* function, but you should make a call to the addChild function when implementing it. The “CTreeTest.cpp” has an example on how this is supposed to work. (i.e., look for testCaretOp() function).
- An overloaded << operator that displays the value returned by toString (i.e., function *operator*<<) on the output stream. In fact, when implementing this, you should make a call to the toString() function of the parameter object. Also, note this function is NOT a member of the CTree class. You need to implement this as a separate function in your cpp file. Again, there is a sample test in the “CTreeTest.cpp” that shows how this is supposed to work (i.e., look for testOutputOp() function).

Regarding the toString() function, we want it to do a depth-first pre-order traversal of the tree. This means that as you iterate through the tree, the first node you get is the root, then it goes through that node's children (recursively depth-first), and finally its siblings. As a related example, a simple (non-object-oriented) recursive depth-first traversal starting at some parameter node would look like this:

```
DFtraverse(node) {
    process data at node
    if node has children,
      do a depth-first traversal from its first child
    if node has siblings (to the right),
      do a depth-first traversal from its next sibling
}
```

You are given a file named CTreeTest.cpp that tests your implementation. Once you are done implementing CTree.cpp, you should compile and run CTreeTest to make sure that all test cases pass.

Part 2:

For the second and final part of the assignment you will generalize the CTree from part 1 to create a templated ordered tree with unique siblings. Note you write your own header file for this part. Your tree class must be called Tree and be declared in “Tree.h” and fully defined (i.e., implementation of functions should occur) in a file named "Tree.inc". Note that the file has the *.inc* extension and not .cpp. The “Tree.inc” will be included at the end of the “Tree.h” file after the class declaration (This is because we can't split templated classes into separate .h and .cpp files). Also, note the Tree class must be templated so that it works for various types as applicable.

Remember to keep data private, and const-protect objects (explicit and implicit parameters) whenever you can. Your Tree class must contain the same public functions and overloaded operators as the CTree class, except that parameters for the data stored in each node will now be of some generic type instead of char.

Because the templated Tree will need to keep siblings in order and unique, each base type will need to have the equals (==) operator and the less than (<) operator defined on them. If you used any operators other than these and the assignment operator (=) in the functions in CTree.cpp, you will need to rewrite those functions too.

Similar to the previous part, **You are given a file named TTreeTest.cpp that tests your implementation (for the templated version).** Once you are done implementing “Tree.h”

and “Tree.inc”, you should compile and run TTreeTest to make sure that all test cases pass.

Makefile & Compiling

Create and submit a Makefile with two rules (aside, perhaps, from the clean rule): 1) a rule that creates a target executable called `cTreeTest` for the first part of this homework, and 2) a rule that creates a target executable called `tTreeTest` for the second part of this homework (the templated version). These two targets should compile with no errors or warnings using the typical C++ compilation command: `g++ <source> -Wall -Wextra -std=c++11 -pedantic`. Do not use any special libraries that require different additional/different compiler options.

The autograder does an automatic check to see if it can compile the two targets. ***Thus, it is very important that you make sure your submission successfully compiles, otherwise your final score for the entire homework assignment will be 0.***

Git log

In the assignments folder of your private repository (*not* the team repository you made for the midterm project), create a new subfolder named `hw7`. Do your work in that subfolder and use `git add`, `git commit` and `git push` regularly. Use `git commit` and the associated comments to document your work. e.g. if you just modified `CTree.cpp` to add a child functionality, you might do `git add CTree.cpp ; git commit -m "addChild functions added!" ; git push`.

Your submission includes a copy of the output of `git log` showing at least four commits to the repository. Save the `git log` output into a file called `gitlog.txt` (e.g. by doing `git log > gitlog.txt`).

Specific Requirements

- Do not eliminate anything from “CTree.h” file. You can add more private (helper) functions if you want, but your “CTree.cpp” file should at least provide an implementation for all the functions that are declared in the given “CTree.h”.
- Your program should only use `new` and `delete` for dynamic memory allocation/deallocation. Do not make use of C functions such as `malloc`, `calloc`, `free`, etc. for that purpose.
- All variables must be declared inside functions. No variables should be global or `extern`.
- Do not use `auto`.
- Use header guards in all header (`.h`) files.
- Do not use `using` in header (`.h`) files.

- In C++ source files (`.cpp` files), you may import individual symbols using statements like "`using std::string`". *Do not* use "`using namespace <id>`", either in headers or in source files.

Hints and Suggestions

- Make use of **gdb** to debug and also run **valgrind** to make sure there is no memory leakage.
- The test files given to you are minimal sample test files. It would be best if you do not confine yourself to these and test your code even more thoroughly.