Compuer Science 62
Assignment 10

Due 11:59pm on Thursday, April 22, 2010

This assignment is the C++ version of the animal game.

The program will have three parts, which have been carefully specified. You will write three C++ files:

`BinaryTree.cpp` will implement the `BinaryTree` class. The interface is provided in the file `BinaryTree.h`. This binary tree is *not* generic; each node contains a single string.

`BinaryTreeIO.cpp` will provide functions to read and write the file data file, `animals.tree`. The declarations are provided in the file `BinaryTreeIO.h`.

`AnimalGame.cpp` has the `main` function that interacts with the human and maintains the game tree.

Getting started

Read through this entire document to make sure you understand what we're asking of you.

To make it easier to develop your code incrementally, we have provided object files (.o files) for each part. From your work on Assignment 7, you already know the logic of the program; the focus of your work will be to cast it into C++.

The starter files for this assignment (which include the header files and the object files) can be found at:

```
/common/cs/cs062/assignments/assignment10/
```

Look at the different header files and make sure you understand the basics of what the interface for each class is doing. Note: you may *NOT* change the header files provided and your program must compile with these header files.

In addition, I have provided you with a Java solution to the animal game from assignment 7 at:

`/common/cs/cs062/assignments/ass10/ass7solution/`

Feel free to refer to this or your own code when you are developing your program.

How to proceed

You are now ready to start coding. Pick the file that seems easiest and complete it before moving on to the next. Below, we've provided more details on each of the files and some hints/suggestions. Once you have one of the .cpp files implemented, you can check it by compiling with our provided binaries.

For example, let's say you start with `BinaryTreeIO.cpp`. To compile this, you type:

```
g++ -c BinaryTreeIO.cpp
```

which will generate a new object file (.o) file for your implementation (note, that if you ever want to use our implementation again, you will need to copy it again from `/common`). Then, you can compile all of the object files to create a binary:

```
g++ -o animalgame BinaryTreeIO.o BinaryTree.o AnimalGame.o
```

You should also consider writing your own main method either in that class or in a separate file to test the functionality (make sure when you're done though, to remove any main methods in your .cpp files except the one in `AnimalGame.cpp`).

Requirements, suggestions and hints

In general, all the nodes/trees will be created with `new` and all the references will be of type `BinaryTree*`.

The main C++ libraries that you will use are `string`, `iostream`, and `fstream`. Look at the link on the course resources page for C++ documentation on these classes.

Make sure there is a file called `animals.tree` in the directory where you are running it (see `AnimalGame.cpp`) below or you will get an error!

`BinaryTreeIO.cpp`

This file will be the shortest and simplest. The header file specifies two functions whose actions should be straightforward: open the file, call a recursive function to do the work, and then close the file. As in our Java version, the easiest way to do this is to write recursive helper methods. The recursive functions will reflect the steps in a preorder traversal. The only parts to fill in are the file operations and the `BinaryTree` manipulations. Auxiliary functions are permitted.

As an example, here is a function that carries out a preorder traversal and counts the number of characters in all the strings.

```
int charCount(BinaryTree* root) {
  if (root == NULL){  // just in case
    return 0;
  }else if (root->isEmpty()){
    return 0;
  }else{
    return root->getString().length() +
           charCount(root->leftChild()) +
           charCount(root->rightChild());
  }
}
```

Notice that there is no class named `BinaryTreeIO`! This "compilation unit" simply provides two functions. There are no internal variables. (In Java, the two functions would have to be attached to a class and would be `static`.)

Finally, remember to put the include directive at the top of the file.

```
#include "BinaryTreeIO.h"
```

`BinaryTree.cpp`

This one may be the longest because there are many functions to implement. Most of them, however, are short and simple. Completing this part is mostly a task in translating your knowledge of binary trees into C++. Do not forget the include directive

`#include "BinaryTree.h"`

You may not have to use some of the functions, like `height()` and `depth()`, but include them anyway (we may test them :)

Be very clear on how the different types of nodes are identified. Here is one possible convention:

- If one of `left` or `right` is `NULL`, then they both are `NULL`.
- An empty tree is a node whose `left` and `right` members are `NULL`.
- An empty tree has a `NULL parent`. That way, we can use just one empty node throughout the program, which you can define in your C++ class as a standalone, static variable, say `empty`.
- A leaf is a non-empty node whose left and right children are empty.
- A root is a non-empty node whose `parent` is `NULL`.

One of the consequences of this design decision is that each node has a unique parent and may therefore appear in only one tree. It leads to the "deep copy" requirement discussed below.

The tree nodes contain answers and questions. Answers are at leaf nodes and questions are at non-leaves. As before, do not add any additional information to the nodes to tell them apart. In particular, do not store in a node the leading character, `A` or `Q`, from the `animals.tree` file.

Notice that there is no `setString()` function. We want you to construct the trees by detaching subtrees and reassembling them. The greatest challenge is to maintain the parent pointers properly when executing `setLeftChild()` and `setRightChild()`. As we discussed in class when we covered binary trees, the `setParent()` method is private to insure the integrity of the parent pointers. It *may* be useful to look at the Java binary tree implementation explored in class, which is in the notes on the class web page.

Finally, we have included code below for some of the methods of this class. You should read through and make sure to understand what it going on, but you may copy and use this code in the version you submit.

`AnimalGame.cpp`

This file, as before, implements the interaction with the player and updates the actual tree. Your organization from Assignment 7 ought to serve you well.

One variation from our Java version is that we won't deal with input and output from the command-line. Instead, when your game starts it should read from a file named "`animals.tree`" and when the game finishes it should update this file, that is, write the final tree to "`animals.tree`".

`AnimalGame.cpp` will have more include directives than the other files.

```
#include <iostream>
#include <string>
#include "BinaryTree.h"
#include "BinaryTreeIO.h"
```

Submisson

Submit your three .cpp files in a folder with you name, assignment number, etc. Make sure your code compiles with the included files *as they are provided* and that your files are well-documented files in the usual way with Javadoc comments, etc.

```
AnimalGame.cpp
BinaryTree.cpp
BinaryTreeIO.cpp
```

Grading

criterion	points
BinaryTreeIO	4
BinaryTree	6
AnimalGame	5
assert error checking	2
appropriate comments (including JavaDoc)	2
style and formatting	1
submitted correctly	1

Memory management in the BinaryTree class

One of most delicate aspects of C++ programming is making sure that all objects created with `new` are properly deleted. One must avoid dangling pointers and memory leaks. The copy constructor, the destructor, and the assignment operator are the critical functions for these tasks.

The animal game does not exercise the binary tree functions enough to expose deficiencies in these functions; you could probably write a seemingly correct game without implementing the functions at all. But I would like you to see a complete example and provide one in your assignment. Please *study* the functions carefully; understanding the functions and their interrelationships is part of this assignment.

In our `BinaryTree` class, the only job of the destructor is to delete all dynamically-constructed subtrees. We can do that with a simple call to `clear()`.

```
BinaryTree::~BinaryTree() {
  clear();
}
```

Although nice, this solution pushes the hard work off onto `clear()`, which recursively deletes the left and right subtrees and leaves the current object empty.

Below is the code for `clear()`. The implementation is mostly straightforward; there is just one subtlty. In the sample code, we keep a special empty tree, called `empty`, and use it wherever we can to avoid creating lots of empty trees. (Bailey, in an earlier version of his code, tried to *guarantee* that there is only one empty tree by making the default constructor private. That does not work here because the `clear()` method will turn any tree into an empty one.) We must be careful, however, not to delete the tree named `empty`.

```
void BinaryTree::clear() {
  // don't change an empty tree
  if (isEmpty()){
    return;
  }

  // detach this tree from its parent, if any,
  // to avoid dangling pointers from the parent
  if (isLeftChild()){
    parent->setLeft(empty);
  }else if (isRightChild()){
    parent->setRight(empty);
  }

  // delete the children recursively
  if (left != empty){
    delete left;
  }if (right != empty){
    delete right;
  }

  // make this an empty tree
  value = "";
  left = NULL;
  right = NULL;
  parent = NULL;
}
```

Assignment operator

The assignment operator “copies” data into the (already constructed) current object. The first task is to recycle all the current descendants—which is exactly what `clear()` does. The next step is to assign new values to the fields, calling the copy constructor when necessary to obtain a deep copy.

```
const BinaryTree & BinaryTree::operator=(const BinaryTree & rhs) {
  // avoid self-copies
  if (this != &rhs) {

    // destroy the nodes in the current tree
    clear();

    // assign default values
    value = rhs.getString();
    left = NULL;
    right = NULL;
    parent = NULL;

    // copy the children, as necessary
    if (!rhs.isEmpty()) {
      if (rhs.getLeftChild()->isEmpty()){
        left = empty;
      }else{
        setLeft(new BinaryTree(*rhs.getLeftChild()));
      }

      if (rhs.getRightChild()->isEmpty()){
        right = empty;
      }else{
        setRight(new BinaryTree(*rhs.getRightChild()));
      }
    }
  }

  // return the newly-copied object
  return *this;
}
```

Copy constructor

The copy constructor is nearly identical. In fact, similar enough, that we can use our trick of:

```
BinaryTree::BinaryTree(const BinaryTree & other){
  *this = other;
}
```

If you wanted to implement it for some reason, the only major difference is that the object under construction is a brand new object, so there is no need to call `clear()`.

As mentioned earlier, every non-empty node has a unique parent and can trace its ancestry to a unique root. This requirement leads to a "no sharing" policy and forces us to make deep copies. When we copy a subtree, either through the copy constructor or assignment, the copy is the root of its own tree. It cannot, and does not, share a parent with the object being copied.

Other implementations are possible, of course, and often the overhead of deep copying would be intolerable. Think about the options.