lab_trees Tempestuous Trees

Due: Sunday, October 11 at 11:59 PM

Doxygen for lab_trees

Assignment Description

In this lab we'll explore some cool helper functions for binary trees, learn about creating helper functions and thinking recursively, and hopefully see some fancy ascii trees on the terminal!

A new C++ thing

We'll be using templates again in this lab, and unfortunately, this means we have to walk into a dark scary corner of C++. But hopefully we can take our lantern and cast some light here before any compiler errors bite.

The following function definitions won't compile:

```C++
template <typename T>
Node * BinaryTree<T>::myHelperFunction(Node * node)
// The compiler doesn't know what a Node is, since the return type isn
// with the function.
```

So we have to scope it:

```C++
template <typename T>
BinaryTree<T>::Node * BinaryTree<T>::myHelperFunction(Node * node)
```

Using g++, the latter will show an error such as:

```
error: expected constructor, destructor, or type conversion before '*'
```

Clang gives a more helpful error message:

```
fatal error: missing 'typename' prior to dependent type name 'BinaryTr
```

Without going into the ugly details of it, this is happening because your compiler thinks `Node` is a member variable of the `BinaryTree<T>` class (since it's a template). We can resolve this issue simply by adding a nice, friendly, `typename` keyword before our `BinaryTree<T>::Node` type.

This lets the compiler know that `Node` really is a type, not a variable:

```C++
template <typename T>
typename BinaryTree<T>::Node * BinaryTree<T>::myHelperFunction(Node *
```

The above, fixed, definition compiles correctly. Since you'll probably want to create your own helper functions for this lab, this is important to remember when you see the strange error above. You won't be responsible for this nuance of templates on any exam in this class.

Checking out the code

To check out your files for this lab, run

```TERMINAL
svn up
```

from your `cs225` directory.

This will create a new folder in your working directory called `lab_trees`.

A reference for the lab is provided for you in Doxygen form.

Testing Your Code

To test your code, compile using make:

```TERMINAL
make
```

Then run it with:

```TERMINAL
./treefun color
```

You will see that the output is colored — green means correct output, red means incorrect output, and underlined red means expected output that was not present. This mode is a bit experimental, and it might cause problems with your own debugging output (or other problems in general). To turn it off, simply leave off the "color" argument:

```TERMINAL
./treefun
```

You may also diff your solution with our expected output:

```TERMINAL
./treefun > out
diff -u out soln_treefun.out
```

Helper Functions and Recursion

You'll want to be thinking about the following problems recursively. To do this, though, you'll have to make your own helper functions to help implement the functions, so that you can recursively act differently on different nodes. There is room in the `.h` file for you to declare these extra functions. A helper function stub for `height()` has been provided for you.

The `height()` Function

There is a function called `height()` that returns the height of the binary tree. Recall that the height of a binary tree is just the length of the longest path from the root to a leaf, and that the height of an empty tree is -1.

We have implemented `height()` for you (see `binarytree.cpp`) to help you get a sense of recursive functions. Please read through the code, and ask questions if you are unsure of how it finds the height of a tree.

The `printLeftToRight()` Function

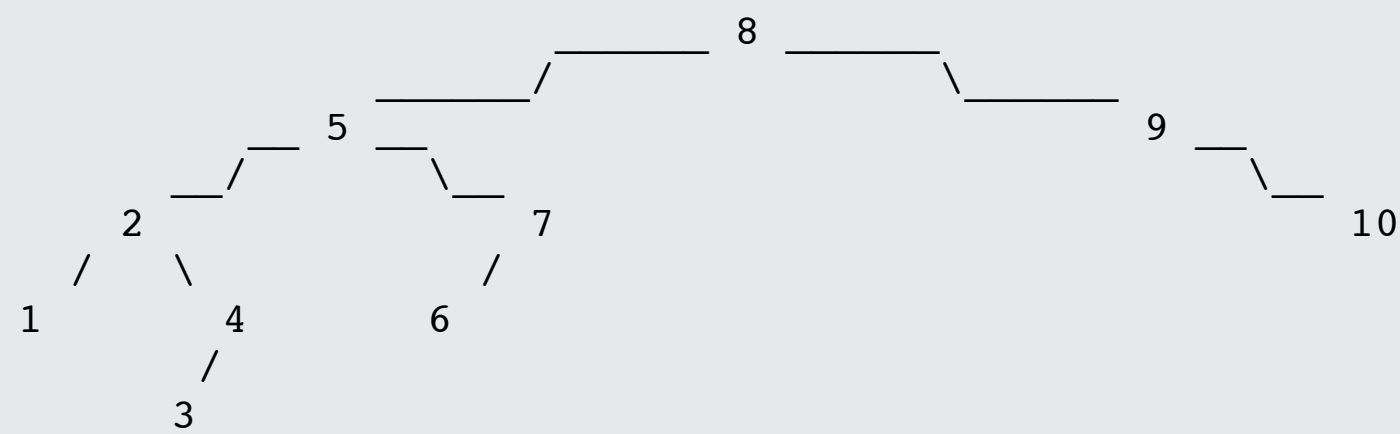
There is a function called `printLeftToRight()` that prints out the values of the nodes of a binary tree in order. That is, everything to the left of a node will be printed out before that node itself, and everything to the right of a node will be printed out after that node.

We have implemented `printLeftToRight()` for you (see `binarytree.cpp`). Please read through the code, and ask questions if you are unsure of how it works. Note that `printLeftToRight()` uses an in-order-traversal to print out the nodes of a tree. You will need to use one of the three traversals covered in lecture for some of the following functions.

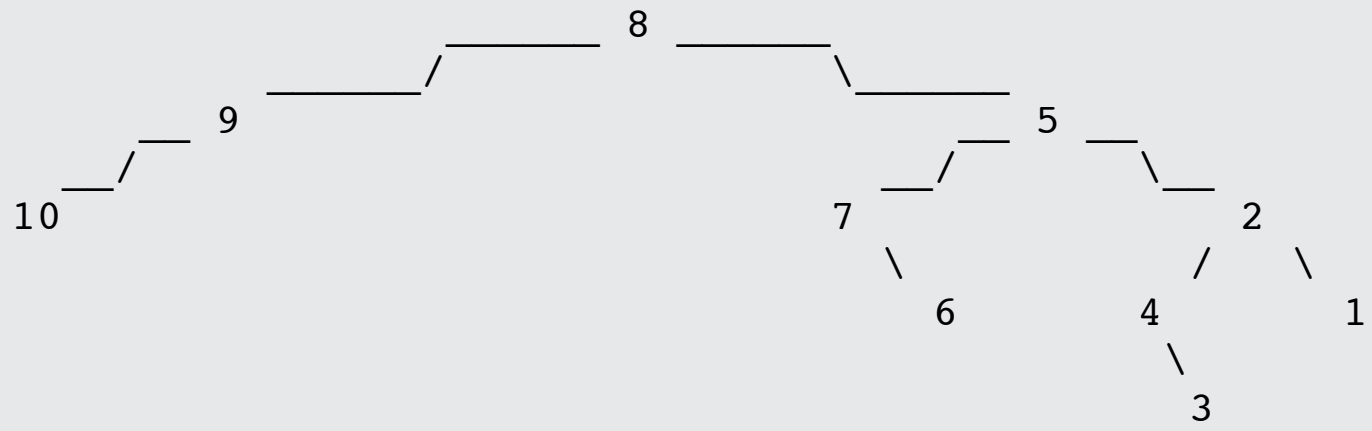
The `mirror()` Function

The `mirror()` function should flip our tree over a vertical axis, modifying the tree itself (not creating a flipped copy).

For example, if our original tree was



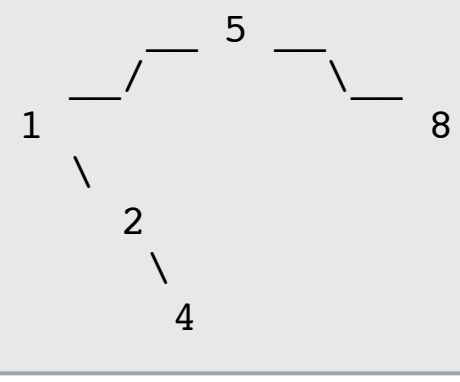
Our mirrored tree would be



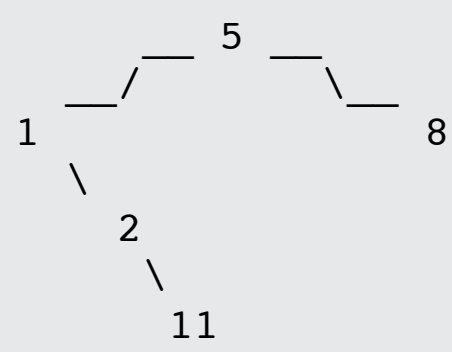
The `isOrdered()` Function

The `isOrdered()` function returns true if an in-order traversal of the tree would produce a nondecreasing list output values, and false otherwise. (This is also the criterion for a binary tree to be a binary search tree.)

For example, `isOrdered()` should return `true` on the following tree:



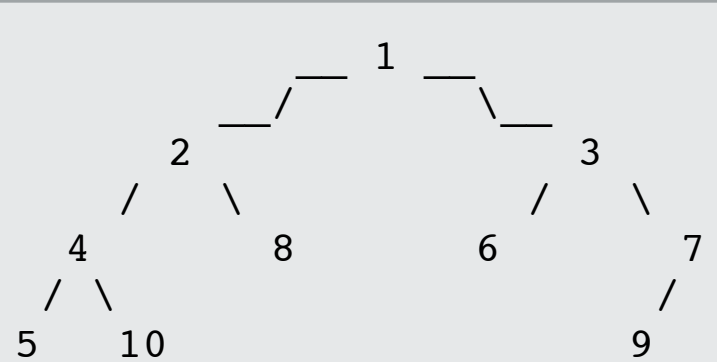
but `false` for



Hint: What conditions need to be true for a tree to be in order (as defined above)? How can we check this recursively?

The `printPaths()` Function

Good work! We just have one more function for you to write. `printPaths()` will print out all the possible paths from the root of the tree to any leaf node — all sequences starting at the root node and continuing downwards, ending at a leaf node. Paths ending in a left node should be printed before paths ending in a node further to the right. For example, for the following tree



`printPaths()` would output

```
Path: 1 2 4 5
Path: 1 2 4 10
Path: 1 2 8
Path: 1 3 6
Path: 1 3 7 9
```

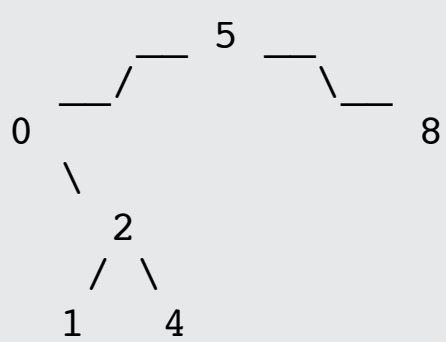
Hint: You'll need to keep track of the whole path that you took to arrive at any point in your traversal, so you can print out your path when you get to a leaf node. The `std::vector` class might be useful for that.

(Extra Credit) The `sumDistances()` Function

Complier errors

Submitting code that doesn't compile **will result in a zero** on the entire lab. This includes code in the extra credit portion and should be common sense. Be sure to test your code before you submit.

Good job getting this far! The following function is for extra credit — its output will not be used to grade the regular portion of the lab. Each node in a tree has a distance from the root node - the depth of that node, or the number of edges along the path from that node to the root. `sumDistances()` returns the sum of the distances of all nodes to the root node (the sum of the depths of all the nodes). Your solution should take $\mathcal{O}(n)$ time, where n is the number of nodes in the tree. For example, on the following tree:



`sumDistances()` should return `0+1+2+3+3+1 = 10`.

Committing Your Code

Commit your code in the usual way.

Grading Information

The following files are used in grading:

- `binarytree.h`
- `binarytree.cpp`

All other files including any testing files you have added will not be used for grading.

Good luck!

Thanks to Nick Parlante/Stanford, Princeton's CS 126, and CS 473 Spring 2011 for the exercises and inspiration.