lab_trees

Tempestuous Trees

Due: Feb 24, 23:59 PM

Doxygen | Lab handout

Assignment Description

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

Lab Insight

Trees are a very powerful structure for lookups and finding data quickly. Examples of use cases for this data structure includes search enginer optimization and fast sorted data retrieval. CS 410, Text Information Systems, is a course that delves into topics involving text manipulation such as test search lookups. Trees can even be used for syntax and language grammar analysis which relates a lot with CS 421, Programming Lnaguages and Compilers.

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:

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

So we have to scope it:

```
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 '*' token
```

Clang gives a more helpful error message:

```
fatal error: missing 'typename' prior to dependent type name 'BinaryTree<T>::Node'
```

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:

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

Getting Set Up

From your CS 225 git directory, run the following on EWS:

```
git pull
git fetch release
git merge release/lab_trees -m "Merging initial lab_trees files"
```

If you're on your own machine, you may need to run:

```
git pull
git fetch release
git merge --allow-unrelated-histories release/lab_trees -m "Merging initial lab_trees
files"
```

Part of good "git hygene" is to run `git pull` before you do anything else.

Upon a successful merge, your lab_trees files are now in your `lab_trees` directory.

The code for this activity resides in the `lab_trees/` directory. Get there by typing this in your working directory:

```
cd lab_trees/
```

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

Testing Your Code

To test your code, compile using make:

```
make
```

Then run it with:

```
./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:

```
./treefun
```

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

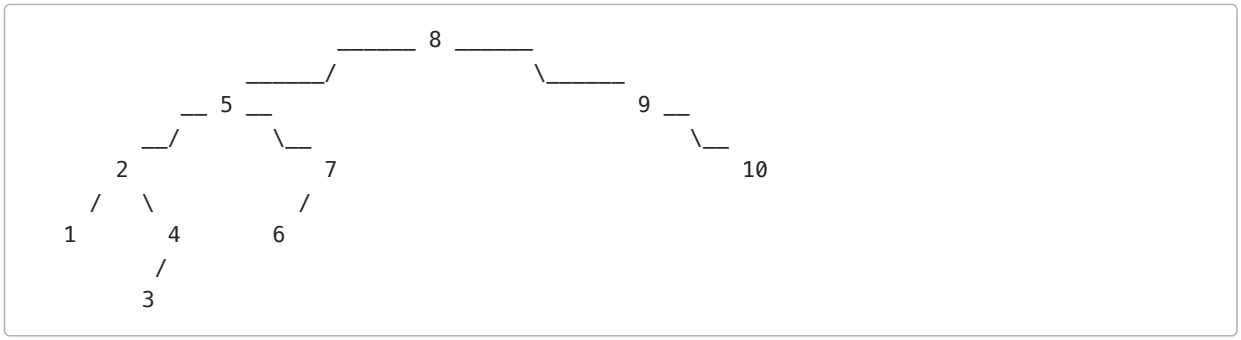
There is a function called `printLeftToRight()` that prints out the values of the nodes of a binary tree in order. That is, everything to the left of a node will be printed out before that node itself, and everything to the right of a node will be printed out after that node.

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.

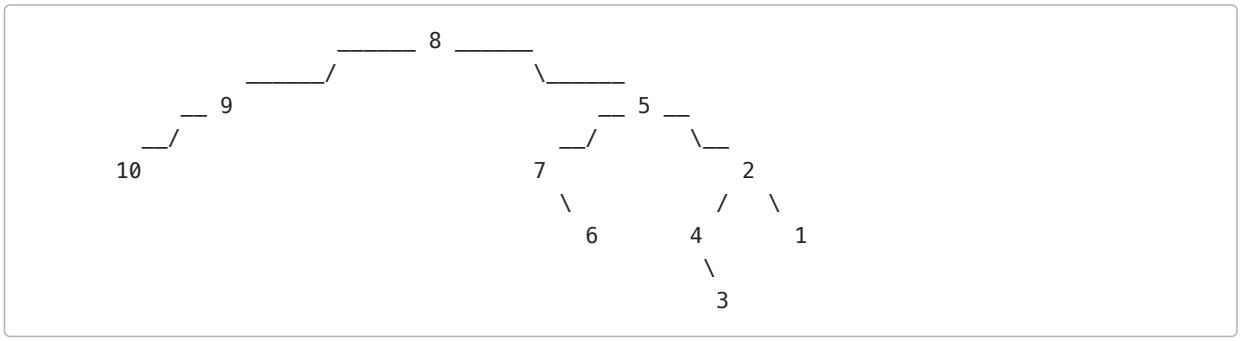
The `mirror()` Function

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 `TreeTraversals` Family

Class Hierarchy for TreeTraversals family:

We've already implemented PreorderTraversal class for you(see TreeTraversals/PreorderTraversal.h). Your task is to implement the the following constructors and functions for InorderTraversal class:

- `InorderTraversal(typename BinaryTree<T>::Node* root)`
- `void add(typename BinaryTree<T>::Node *& treeNode)`

Test your InorderTraversal class:

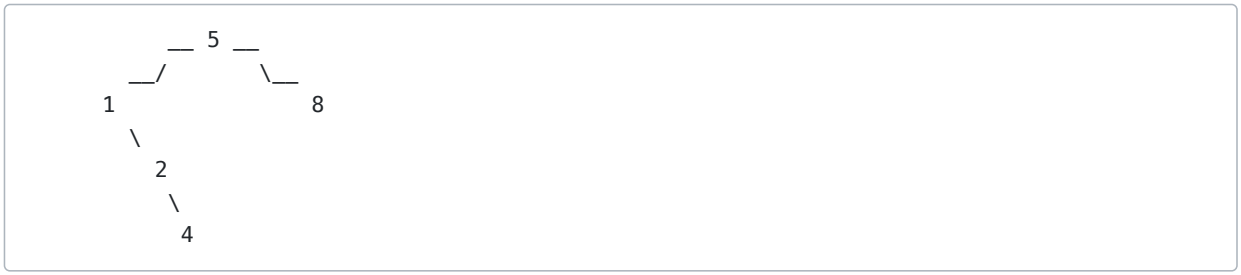
```
clang++ -std=c++1y -stdlib=libc++ main.cpp -o main #run these commands INSIDE
TreeTraversals folder
./main
```

The `isOrdered()` Family

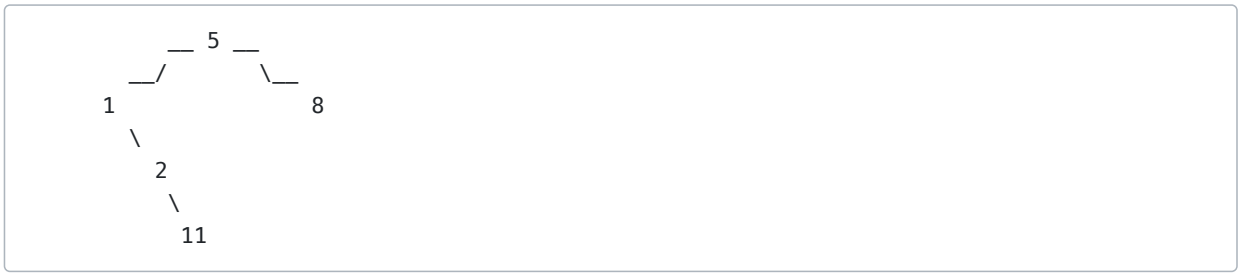
The `isOrdered()` family includes two functions, one should be implemented iteratively, the other should be implemented recursively.

Both functions return 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



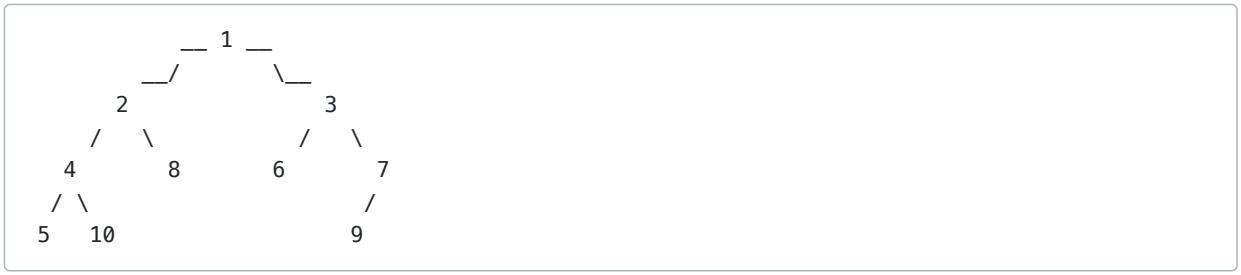
You'll need to implement the following functions:

- `bool isOrderedIterative() const`
- `bool isOrderedRecursive() const`

Hint: What conditions need to be true for a tree to be in order (as defined above)? How can we check this iteratively? (Your Iterator class might help) How can we check this recursively? You might want to write your own helper functions for this exercise.

The `getPaths(vector<vector<T>>& paths)` Function

Good work! We just have one more function for you to write. `getPaths(vector<vector<T>>& paths)` will use two dimensional vector `paths` to store 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 pushed before paths ending in a node further to the right. For example, for the following tree



`paths.size()` will be 5 (that means that `vector<vector<int>> paths` will contain 5 `vector<int>`), and each vector in `paths` will look like this:

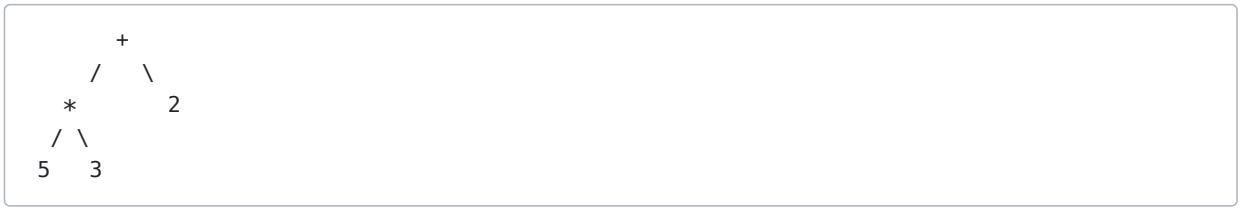
```
paths[0]: 1 2 4 5
paths[1]: 1 2 4 10
paths[2]: 1 2 8
paths[3]: 1 3 6
paths[4]: 1 3 7 9
```

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 add your entire path when you get to a leaf node. The `std::vector` class might be useful for that.

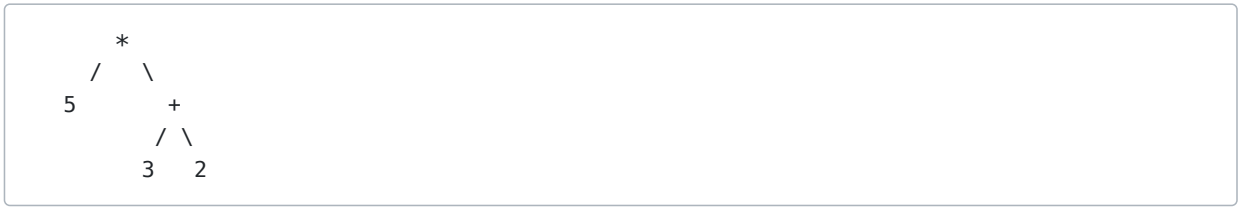
Abstract Syntax Trees

Binary trees are an amazing data structure with the flexibility to do an assortment of amazing things. One thing binary trees are amazing for is expressing the rules of syntax. One interesting syntax is PEMDAS (Parenthesis Exponentiation Multiplication Division Addition Subtraction). For example, the expression `5 * 3 + 2` evaluates to 17, while the expression `5 * (3 + 2)` evaluates to 25. We can use binary trees to express these two PEMDAS strings.

For `5 * 3 + 2`, we can generate the following binary tree



For `5 * (3 + 2)`, we can generate the following binary tree



What you'll notice between these two Abstract Syntax Trees is that you first have to calculate all the subtrees before you can do the intial math operation. Basically, you will first need to calculate the value of the operation of each previous subtree before you can compute the value for the current tree.

We have given you the function `double AbstractSyntaxTree::eval()`, which will take in a BinaryTree that has `std::string` for nodes and return the calculated result from that AST. A helper function may be of use. Also, keep in mind that `AbstractSyntaxTree::getRoot()` returns a variable of type `typename BinaryTree<std::string>::Node* const`.

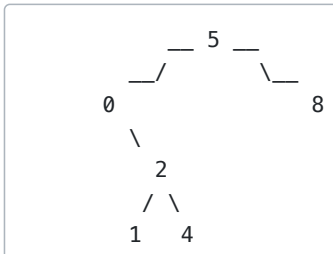
You do not need to worry about the exponential operator. We will only be using the addition, subtraction, multiplication, and division operations.

> ❗ Also, since the numbers you need for the calculations will be strings, you can use `std::stod(std::string str)` to convert strings into doubles.
>
> ```
> double x = std::stod("25"); // x = 25
> ```

(Extra Credit) The `sumDistances()` Function

> ☢ 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`.

Testing Your Code with Catch

Run the Catch tests as follows (this requires your code to compile when run simply as `make`):

```
make test
./test
```

Submitting Your Work

The following files are used in grading:

- `binarytree.h`
- `binarytree.cpp`
- `abstractsyntaxtree.h`
- `abstractsyntaxtree.cpp`
- `TreeTraversals\InorderTraversal.h`

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

Guide: How to submit CS 225 work using git

Good luck!

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