CS 15 Lab 6: Tree Traversals

Introduction

In this lab, you will practice traversing binary trees. Specifically, you'll gain practice with:

- Three traversal orders:
  - in-order
  - pre-order
  - level-order
- Two implementation strategies:
  - Recursive, using the function call stack
  - Iterative
    - For in and pre order traversals, using an `std::stack`
    - For level order traversal, using an `std::queue`
- The *C++ Standard Template Library* (STL).
- *Universal Polymorphism*, because we're using the STL. The STL implements a variety of standard containers which are templated - thus, you can use them for any type of data!

Traversal Algorithms Review

We will have talked about traversals in class, but here is a review (in case we don't get to all of them).

In-Order Algorithm

This algorithm is naturally recursive, and we'll do a recursive implementation in this lab.

1. Traverse left
2. Process current Node (whatever that means for the application)
3. Traverse Right
4. Don't forget to combine and return results if necessary!

Pre-Order Algorithm

This algorithm is also naturally recursive, **but we won't do it recursively in this lab**:

1. Process current Node (whatever that means for the application)
2. Traverse left
3. Traverse Right
4. Don't forget to combine and return results if necessary!

How can we do it non-recursively? We will mimic the storage behavior the function call stack gives us with an `std::stack` data structure: that is, rather than make a recursive call, you'll push the object (actually a pointer to it) onto a `std::stack`.

1. Push root on the stack if the tree isn't empty
2. As long as there is anything in the stack:
   (a) Pop an object off the stack
   (b) Process that node (whatever that means for the application)
   (c) If there's a right child, push it on the stack
   (d) If there is a left child, push it on the stack
   (e) Don't forget to combine and return results if necessary!

You must be careful to push the right child onto the stack first so that the left subtree is handled first (because of the LIFO nature of the stack). Be sure you understand this: Draw out an example for yourself (not to turn in).

Observe how much easier the recursive version is!

Level-Order Algorithm

In a level-order traversal, we process nodes in "level order". That is, we process the root, then the root's children, then the root's children's children, etc. If you draw the tree, you process all the nodes on one line (at one level) before proceeding the the next level.

This traversal cannot be done easily with recursion. You cannot use a stack, because nodes are not processed in a LIFO order.

Instead, we'll use a queue - in particular, we will use an instance of `std::queue`. The algorithm is essentially the same as for the non-recursive, stack-based pre-order algorithm above, except it uses a queue to process nodes in a FIFO order.

The Standard Template Library (STL)

The *Standard Template Library* (STL) is a set of `C++` libraries that provide containers and other functionality that can be used with most objects (e.g., to contain `Node` pointers). You've already seen `std::vector`. In this lab, we will be using `std::stack` and `std::queue`.

- The STL demonstrates *modularity*:

  Since all queues look the same, why re-implement one for every application? There is one, ready-to-use, well-debugged queue implementation that we can all share!

- The STL demonstrates *universal polymorphism*:

  Since a queue of integers and a queue of patrons of the Olympic gymnastics finals are exactly the same except for the type of the data, why re-implement a queue for each data type? If we did that, we couldn't satisfy our desire for modularity either except for certain very simple cases. So, there is one ready-to-use, welll-debugged queue implementation that we can use for any type of data!

To create a `std::stack` or `std::queue` object, you need to tell the compiler what type of object or fundamental data type you will be putting into the container. To do this, you use angle brackets, as follows:

```cpp
#include <stack>
#include <queue>
#include <string>

int main() {
    std::stack<int> my_stack_of_ints;
    std::queue<std::string> my_queue_of_strs;
    return 0;
}
```

Like all data abstractions, the STL containers have **interfaces**: They come with a set of functions that define the abstract type. You'll find lists of functions for STL stacks and queues below for your reference.

STL Stack Interface

Function	Description
`void pop()`	Removes element from the top of the stack
`bool empty() const;`	Returns true if stack is empty
`TYPE& top();`	Returns a reference to the top element of stack
`void push(const TYPE &val);`	Puts `val` on top of stack
`size_type size() const;`	Return number of elements on the stack

Code Example

```cpp
#include <stack>         // load declarations from STL

int main() {
    std::stack<int> s;   // make stack of integers
    s.push(14);          // push on two items
    s.push(12);

    int n = s.top();     // get copy of top item
    s.pop();             // remove top item
    return 0;
}
```

STL Queue Interface

Function	Description
`void pop()`	Removes element from front of queue
`bool empty() const;`	Return true if queue is empty
`TYPE& front();`	Returns a reference to the front element of queue
`void push(const TYPE &val);`	Puts `val` on back of queue
`size_type size() const;`	Return number of elements on the queue

Code Example

```
#include <queue>         // load declarations from STL

int main() {
    std::queue<int> q;   // make queue of integers
    q.push(14);          // push on two items
    q.push(12);

    int n = q.front();   // get copy of front item
    q.pop();             // remove top item
    return 0;
}
```

Implementation Specifics

Getting Started

Get the files from `/comp/15m1/files/lab06`. The items you have to complete have TODO comments. You will write the traversal functions as specified above for a `BST` class[1]

Node struct

The `Node` structure for the `BST` class is as follows:

```
struct BSTNode {
        int      value;
        BSTNode *left;
        BSTNode *right;
};
```

[1]Note that these traversals can be generalized to work on any type of tree!

Functions to Implement

You will write three functions, each implementing one of the traversals.

1. `void BST::inOrder(BSTNode *root)`

   You must use recursion for this function, following the rules for printing the nodes in the **In-Order Algorithm** section of this document. (Make sure you use spaces appropriately when you print — you want your solution to be the same as the solution that we provide).

2. `void BST::preOrder(BSTNode *root)`

   Do a preorder traversal of the tree using an `std::stack`, following the rules for printing the nodes in the **Pre-Order Algorithm** section of this document, using the `std::stack` interface above.

3. `void BST::levelOrder(BSTNode *root)`

   Do a level-order traversal of the tree using an `std::queue`, following the rules for printing the nodes in the **Level-Order Algorithm** section of this document, using the `std::queue` interface above.

Things to Notice

There are a few unusual things worth noticing in the code.

- Although the three functions above are private, there are also three public functions that are defined as:

  - `void BST::inOrder();`
  - `void BST::preOrder();`
  - `void BST::levelOrder();`
  - `void BST::postOrderDelete();`

  These functions are the ones called in `main()`. They will then call their corresponding private version. These are called wrapper functions. Outside the `BST` class, e. g., in `main()`, we do not have access to the private data member `root` (nor should we!).

- Also notice that the private functions you're going to write take in a `BSTNode *` (pointer to a `BSTNode`) as a parameter. You'll need to consider how your `std::stack` and `std::queue` objects will need to be declared to work with pointers!

- Bonus point: `postOrderDelete()` is called by the destructor, and should delete all nodes in a postorder fashion. *Exam-type question: Can we delete in another order, say by doing a pre-order traversal?*

Tips

- Test as you go!

  Don't do everything at once! Pick a function, and do that one first. You may not even write the whole function. For example, if you're not used to using the STL types, you might write a draft of the function that just pushes and pops one item on the `std::stack` or `std::queue`. Test that. Slowly build up your function out of pieces that you completely understand and that work.

- Talk!

  You can't share code with your classmates, but you can share ideas. Draw pictures, discuss.

Submitting Your Work

Submit your work as discussed in class.