Project 4: Binary Trees and Tree Traversals

Due: 11/05/2007

Warning: This assignment is relatively challenging, so start early and ask questions right away!

Educational Objectives: Understand the binary tree ADT and various tree traversal algorithms. Understand the usages of iterators in accessing elements in containers.

Statement of Work: Implement various binary tree traversal algorithms using iterators.

Project Description:

A `Binary_Tree` is a type of data container/data structure that organizes data hierarchically. The `Binary_Tree` ADT is typically implemented using node structures that contain data and pointers to other nodes. The `Binary_Tree` ADT maintains a reference to a special node, called the `root` of the tree. From this node, other nodes can be located by traversing pointers. There are slightly different approaches to implementing trees, but we will use one based on each node maintaining 3 pointers.

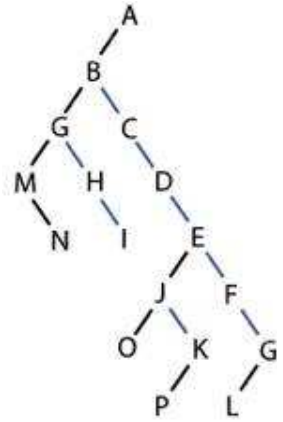
- Each node maintains three pointers, to nodes named parent, left child and right child, respectively.
- The `root` has an invalid parent pointer.
- If `A->lchild` refers to node `B`, then `B->parent` refers to `A`.
- Similarly, if `A->rchild` refers to node `C`, then `C->parent` refers to `A`.
- If a node `A` can be reached from a node `B` by following only parent pointers, `A` is said to be an ancestor of `B`.
- Any two nodes A and B in the Binary Tree have at least one common ancestor.
- There is always a unique shortest path from a node A to another node B in the Binary Tree. The path starts from A and traverses parent pointers until it reaches the first node that is a common ancestor of both A and B. Then, from that ancestor, it traverses either left child or right child pointers to reach B.

Trees are useful in many computing applications. Mostly they are efficient ways to maintain data that has a natural hierarchic order—for instance, file systems and databases. Binary trees are not as commonly used as related ADTs, such as **B-Trees**, but are conceptually simpler and therefore useful in teaching the basic concepts behind trees.

Further reading about trees and in particular binary trees, see the slides as well as the following article: http://en.wikipedia.org/wiki/Binary_tree

Tree Traversals:

Every ADT defines iterators that allow for efficient access to the data. To date, we have studied ADTs that maintain a linear ordering of data, and which allow for linearly-sequential traversals of the data. By comparison, the `Binary_Tree` ADT supports a greater variety of iterator types, and different ways to traverse the data.



The above picture was obtained from the Wikipedia article mentioned in the previous section. It will be used to illustrate some concepts about various tree traversals.

In order to understand the traversals, it is useful to think that each traversal defines an ordering between nodes of a tree. Each of these orderings is a function of the root-to-node path. More explicitly, consider the above picture. Then, the root-to-node paths of each of the nodes `A`, `E`, `I`, and `O`, are:

```
A:<empty>                    E:left-right-right-right
I:left-left-right-right      O:left-right-right-right-left-left
```

Remember that the depth of a node is the length of the path from the root to the node. The set of all nodes at a given depth is called a level of the tree. For instance, `A` is at level 0 (or has depth 0), `E` and `I` are at same level (depth 4) and `O` is at level 6.

Traversing a tree is performed to complete some computation on the stored data. Such computation can usually be factored as smaller tasks, performed each time a traversal accesses a particular node in the tree. A traversal generally accesses a node multiple times, once when it is first reached from another node (typically the parent). At that time, one executes the `onEntry()` task; the next task is executed according to the ordering specified by the particular traversal strategy. This is called the visit, and the task executed is also named `onVisit()`. Finally when the node is last accessed during the traversal there may be some final task, which we name `onExit()` task.

By defining only these three tasks differently, a multitude of tree-based algorithms can be implemented that re-use one of the four possible traversal modes described below. After each pseudo-code, we give the result that would be printed if `onEntry()` printed the character "(", `onExit()` printed ")", and `onVisit()` printed the data stored at the current node.

Level-order Traversal

The level order traversal visits the nodes in the following order. It visits first the root, then all nodes at level 1, and so on for each level. In a given level, it visits the nodes according to their root-to-node paths. If `M` and `N` are two nodes, consider where their paths from the root first diverge, and visit first the node corresponding to a left turn compared to the one corresponding to a right turn. For instance, node `I` precedes node `E` in the level order traversal because the second pointer in the path from root to node `I` is left while it is right for node `E`.

In order to implement a level order traversal, one uses a queue `Q`, initially empty. Then, in pseudo-code, you should perform the following tasks.

```
currNode = root;
onEntry();
onVisit();
Q.push(root); // Insert the root on the queue.
while Q has elements do {
    prior_node = Q.front();
    foreach child c of prior_node do {
        // left child first, then right child
```

```
            currNode = c;
            onEntry();
            onVisit();
            Q.push(c);
        }
        onExit();
        Q.pop();
    }
```

The full level ordering of the tree shown above using the above algorithm is:

```
(A(B)(G(C)(M(H)(D)(N)(I)(E)))(J(F)(O(K)(G))(P)(L)))
```

Note that the above pseudo-code is simplified. For instance, `currNode` should not be assigned invalid values (if a node has no left child, one shouldn't set `currNode = currNode -> lchild`, for instance).

The above code is very general. For instance, you may use this to actually build a tree from data that is stored in level-order, by defining `onEntry()` or `onVisit()` to do data-reading operations. In fact, you should use this pseudo-code as a starting point to implement the `Load()` method, as data is given in level-order (see the **Project Requirement** section below).

Pre-order Traversal

Pre-order traversal accesses first (and visits) the root. Recursively, after visiting the current node it accesses (and visits) the left child, then the right child. Pre-order traversal can be efficiently implemented via the stack-based DFS algorithm, or using the parent pointer in a loop, with the following pseudo-code.

```
currNode = root;
previous_node = root->parent ;
        // will not be a valid node here;
while(currNode is valid) {
    if (previous_node == currNode-> parent) {
        // condition_1
        onEntry();
        // This means we are first visiting the currNode
        onVisit();
           // Do the entry and visitation tasks before visiting the
children

        if currNode has a valid lchild {  // test_condition 2
            // we always try to visit the left child first
            previous_node = currNode;
            currNode = currNode -> lchild;
            continue;
        }
    }
    // At this point we have either failed
    // condition_1 or condition_2 and in
    // each case we do not have to visit a left child
    // (either because we are backtracking or //
    // because there is no valid left child).
    // In any case,
    // unless we are backtracking from a right child
    // we should try to access
    // that right child next

    if (previous_node != currNode -> rchild){
        if currNode has a valid rchild {
            previous_node = currNode;
            currNode = currNode-> rchild;
        }
    }
    // Now we are done;
```

```
    // there is no right child to visit either because
    // it is not valid or we were backtracking from
    // the rchild to start with
    else {
        onExit();
        previous_node = currNode;
        currNode = currNode ->parent;
        // Note that if current node is the root and
        // we are backtracking
        // currNode becomes root->parent, which is
        // invalid and we leave the loop, being done.
    }
} // end
```

Pre-order traversal for example:

```
(A(B(G(M(N))(H(I)))(C(D(E(J(O)(K(P)))(F(G(L)))))))
```

Post-order Traversal

Post-order traversal defines the opposite strategy of pre-order. Starting with the root, it tries to visit all the children of the current node, and finally visits the node itself. Post-order traversal can be implemented with a stack-based DFS, or using the parent pointer in a loop. The pseudo-code is very similar to the one provided for the pre-order traversal, except that the method `onVisit()` is invoked just before `onExit()` rather then just after `onEntry()`;

Post-order traversal for example:

```
(((((N)M)((I)H)G)(((((O)((P)K)J)(((L)G)F)E)D)C)B)A)
```

Inorder Traversal

Again, very similar to the prior two traversals, except that `onVisit()` is invoked before visiting the `rchild`.

In-order traversal for example:

```
(((M(N))G(H(I)))B(C(D(((O)J((P)K))E(F((L)G))))))A)
```

For more reading on tree traversals, consult the slides or the textbook.

Project Requirement

You are required to implement the `TBinaryTree` interface contained in file `tbt.h`. The most important tasks are related to the traversal iterators and the `load` function. The implementation of `TBinaryTree::LevelorderIterator` is given. It is a good clue as how to write the `Load` function as the data to construct the tree is given in level order.

The iterators interact with a helper class called a `Navigator`. The `Navigator` class keeps a pointer to the current node `currNode`. If `nodeN` is a `Navigator`, then `nodeN++` is overloaded to advance the `currNode` stored by `nodeN` to `currNode->rchild`; however, `++nodeN` is overloaded differently to advance `currNode` to `currNode->lchild`; finally, `nodeN--` and `--nodeN` are both overloaded to update `currNode` with `currNode->parent`.

In addition, a `Navigator` class defines the modes `onEntry()`, `onExit()`, and `onVisit()`. This enables the iterators to be generic. Any computation on a tree may be achieved by creating a particular Navigator for a specific task; then providing it as an argument during the construction of an iterator for a particular traversal. For instance, the class `printNavigator` is a Navigator class that implements `onEntry()` to print "(", `onExit()` to print ")" and `onVisit()` to print the data stored at `currNode`. This simple class can be used to provide the complex-looking printout given in the example of the Inorder Traversal by using simply the piece of code (here we print to standard output):

```
printNavigator<T> * printIt = new printNavigator<T>(cout);
```

```
printIt->SetNode(Root().getNode());
TBinaryTreeInorderIterator<T>itr;
for( itr.Initialize(printIt); itr.isValid(); ++itr);

//Yes, that is all!
```

Now, the trick is to overload the InorderTraversal `operator++` (prefix) to actually traverse the tree, performing the increment/decrement of the Navigator's position according to the traversal rules and invoking the functions `onEntry()`, `onVisit()`, and `onExit()` where appropriate. Again, for a good example, see the code for `TBinaryTree::LevelorderIterator`. Use the pseudo-code given above to guide your efforts in the implementation of the other iterators.

Apart from the implementations of the load method and the iterators, you should also fill in the code for all template-declarated methods that are not already implemented in the `tbt.cpp` file. For the most part, the behavior of the other functions are self-explanatory, using clues from the code comments or applying principles from generic programming that apply to ADTs and their iterators. However, if you are not sure what a particular function is supposed to do, ask!

Format of input files

An input file contains the node values of a tree with the following format:

1. Nodes are separated by the tab key '\t', or a new line charater '\n'. There could be optional white space (blank) between nodes and separators. However, it is the tab key '\t' or the new line character '\n' that separates nodes.
2. If a node does not have value (or more specifically, if there are holes in the tree), then the corresponding value is shown to be empty, that is, there is no element between two neighboring separators.
3. Nodes are given from left to right in the level order.
4. If a node is empty (does not have value), then the child nodes will not been shown in the file (i.e., they will not occupy position in the file)..

Download

Click here to download partial code, test cases, and executable code. The tar file contains the following files:

`tbt.h` // interface of node, tree, navigator, and iterators etc. it also contains the implementation of node/tree/navigators

`tbt.cpp` // You need to implement the interfaces of the class templates that are not implemented in tbt.h here.

// They are mainly related to the interface of iterators. The implementation of level order iterator is provided to you.

// You also need to implement the load() function in this file to load a tree.

`loadTest.{cpp,x}` // Test the load function (for char type trees)

`CharTest.{cpp, x}` // Test trees of type char

`IntTest.{cpp, x}` // Test trees of type int

`StringTest.{cpp, x}` // Test trees of type string

`test.c*` // test cases for trees of type char

`test.i*` // test cases for trees of type int

`test.s*` // test cases for trees of type string

`makefile` // the Makefile

Deliverable Requirements

Submit your `tbt.cpp` file, or, if you prefer, you can compress all related files in a tar file and submit the tar file.