Binary Trees

Due Sunday March 16, 2014

Problem Description

Recall that a binary tree is *complete* if all levels in the tree are full[1] except possibly the last level which is filled in from left to right. In this assignment, you will create a `CompleteBinaryTree` data structure that stores data in a complete binary tree. Nodes are added and removed from the binary tree in such a way that the completeness of the tree is always preserved.

In class, we discussed three ways to traverse a binary tree: pre-order, in-order, and post-order. A *level-order traversal* is a fourth way of traversing a binary tree that visits all of the nodes at a depth of i before visiting the nodes at a depth of $i + 1$. The notion of a level-order traversal will be *extremely helpful* to you as you think about how to add and remove nodes from the binary tree in such a way that the tree is always complete. For example, assume that the binary tree looks like:

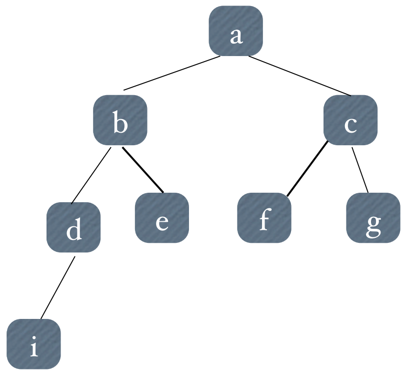


Figure 1: A complete binary tree

The nodes in this tree were added in the order: a, b, c, d, e, f, g, i. If we wanted to add another node (j) to the tree, this new node would be added as the right child of node d. In other words, we are adding nodes to the tree in a level-order manner: we finish adding nodes to depth i before we add nodes to depth $i + 1$. Similarly, if we wanted to remove a node from this tree, we would remove the node i.

It is recommended that you skim section 12.6.4 entitled *Level-order Traversal* in the Java Structures textbook before starting this assignment. Note that this section discusses how to make an `Iterator` for a binary tree that returns nodes according to a level-order traversal. You are **not** being asked to create an `Iterator`. Instead, you should take note of *how* a level-order traversal is performed. In particular, you should take note of *what additional data structures are needed* to perform a level-order traversal!

Unlike the other traversals we looked at in class, a level-order traversal can be performed without using recursion. Instead, an auxiliary data structure is used to keep track of what nodes should be visited next. In a similar manner, you will use an auxiliary data structure to keep track of where to add (and remove) nodes next.

[1]A binary tree is *full* if every node is either a leaf node or has two children

Class structure

There is only one class for this assignment: `CompleteBinaryTree`. You will see in the starter code that this class already contains two instance variables:

```
protected BinaryTree<E> root;
protected Queue<E> queue;
```

As you implement the `CompleteBinaryTree` class, you should add additional instance variables that are helpful.

You are responsible for implementing three methods: `add(E value)`, `E remove()`, and `printTree()`. In addition, you must add a `main` method where you illustrate the functionality of your code.

adding a node

The `add` method adds a node to the binary tree in such a way that the completeness property of the tree is always ensured: from left to right and from top to bottom. In order to do this, you should use a `Queue` to store the nodes in the tree. When `add` is called,

- The new node is created.
- If the tree is empty, the new node becomes the root
- Otherwise, we need to determine which node should be the parent of this new node. To do this, the front of the `Queue` is examined. If the node at the front of the `Queue` does not have two children, the new node is added as a child. If the node at the front of the `Queue` already has two children, it is dequeued. What should happen to the new node at this point?

Please work through a few examples of adding nodes to the `CompleteBinaryTree` class so that you understand the purpose of the `Queue`. When are nodes added to the queue? When are nodes removed from the queue? The order of the nodes in the queue should be the order in which the nodes would be visited in a level-order traversal.

As a final note, you may be tempted to perform a level-order traversal of your binary tree every time `add` is called. In other words, to figure out where to add a new node you might be tempted to start a brand new level-order traversal of the tree starting from the root. This is not a good idea! This would mean that adding a new node is O(N) where N is the number of nodes in the tree!

Instead, the root node should only be put on the `Queue` one time (when it is first created) and from that point on the `Queue` should keep track of the nodes in the tree. That is why the `Queue` is an instance variable instead of a local variable inside of `add`. In this way, adding a new node is O(1)!

removing a node

The `remove` method removes a node from the binary tree in such a way that the completeness property of the tree is always ensured: from right to left and from bottom to top. The simplest way to do this is to model the `remove` method after the `add` method. In particular, you can use another data structure (just like we used a `Queue` for the `add` method) to keep track of the order in which nodes should be removed.

Once you have a pointer to the node that should be removed, you should:

- Get the parent of this node

- Update the parent node so that it no longer points to this child node[2]
- Return the value of the removed node

You can look at the source code of Bailey's `BinaryTree<E>` class to see what methods you can call on a `BinaryTree<E>` node (`left`, `right`, `parent`, `setLeft`, etc). This might give you some ideas on how to correctly remove a node from a binary tree.

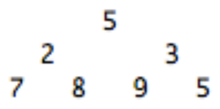
You can, in fact, *use the same data structure* for both the `add` and `remove` methods! There are data structures (ones that we discussed in class and ones that we did not discuss in class) that are similar to a `Queue` and a `Stack` but allow the user to modify (i.e. add and remove) from both ends of the data structure. You will receive extra credit if you implement the `add` and `remove` methods so that they use only one data structure together!

`printTree`

The `printTree` method prints a textual representation of the binary tree using `System.out.println` and related methods (e.g. `System.out.format`). The minimum requirements of the `printTree` method are as follows:

- Each level (depth) of the tree should be printed on a separate line
- Each parent node should be centered above its two children

Note that printing the tree will (again) require a level-order traversal of the tree! You can create a local variable `Queue` to perform the level-order traversal and help you print the tree[3]. The following is one possible output of calling `printTree` on a `CompleteBinaryTree` of `Integers`:



```
      5
  2       3
7   8   9   5
```

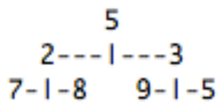
Figure 2: Textual print out of a `CompleteBinaryTree` of `Integers`

Note that each level of the tree is on a separate line and that the parent nodes are centered above the children. The root node contains the value 5. Its left and right child contain the values 2 and 3 respectively. The last level contains the leaf nodes: 7, 8, 9, and 5.

For extra credit, you can make a fancier `printTree` method. See Figure 2 for an example of a slightly better print out of the binary tree that shows the edges as well. Feel free to explore other styles and to make your `printTree` method as sophisticated and informative as you'd like!

[2]This step, if not done correctly, can introduce the dreaded `NullPointerException`. Why? How do we indicate that a node does not have a left or right child?

[3]What is the Big-O complexity of `printTree`?



```
      5
  2---|---3
7-|-8   9-|-5
```

Figure 3: A slightly fancier print out of the `CompleteBinaryTree`

The `main` method

Please provide a `main` method in your `CompleteBinaryTree` class that:

- Creates a `CompleteBinaryTree` of `Integers`
- Adds at least 7 integers to the binary tree and prints out the binary tree in between each call to `add`
- Removes integers from the binary tree and prints out the binary tree in between each call to `remove`

The `main` method is a great place to test the correctness of your code! You are being asked to provide a `main` method (1) to encourage the habit of using the `main` method as a place to unit test your code and (2) as a way of grading your assignment. Your `CompleteBinaryTree` class will be run and part of your grade will be based on what your `main` method prints out! So don't forget to include the `main` method!

Getting Started

All startup code is available in `/common/cs/cs062/assignments/assignment06`. A great way to get started on this assignment is to first read about level-order traversals, and then draw examples of adding nodes to the binary tree (using a Queue) until you are comfortable with how the Queue can be used. Once you understand how to add nodes, it will be easier to understand how to remove nodes.

Extra Credit

There are two extra credit opportunities for this assignment: (1) using only one data structure to help add and remove nodes and (2) a fancier, more sophisticated `printTree` method.

Grading

You will be graded based on the following criteria:

criterion	points
`main` method prints integer example	5
`add` correctly implemented O(1)	5
`remove` correctly implemented O(1)	5
General correctness	3
Appropriate comments (including JavaDoc)	2
Style and formatting	2

What to hand in

As usual, export your entire folder from Eclipse to your desktop. Make sure the exported folder on your desktop contains both a `bin` folder with your .class files and a `src` folder with your .java files.

Please remember to rename your folder to "Assignment7_LastNameFirstName"! Then drag your renamed folder into the dropbox. Be sure that your code is clear, formatted properly, and commented appropriately using Javadoc. See the "Style Guide" on the handouts webpage for what is expected of you.