Lab 15: More on Trees

A. Work with Tree Iterators

Stacks and Queues

Two linear data structures that represent objects in the real world are a *stack* and a *queue*.

Stack

A stack data structure models a stack of papers, or plates in a restaurant, or boxes in a garage or closet. A new item is placed on the top of the stack, and only the top item on the stack can be accessed at any particular time. Stack operations include the following:

- *pushing* an item onto the stack;
- accessing the top item of the stack;
- *popping* the top item off the stack;
- checking if the stack is empty.

Queue

A queue is a waiting line. (The term is more popular in England than it is here.) As with stacks, access to a queue's elements is restricted. Queue operations include:

- adding an item to the *back* of the queue;
- accessing the item at the *front* of the queue;
- removing the front item;
- checking if the queue is empty.

Digression on Stacks

Working with Stacks and Queues

Exercise 1

Suppose that the following sequence of operations is executed using an initially empty **stack**. What ends up in the stack?

```
push A
push B
pop
push C
push D
pop
push E
pop
```

Exercise 2

Suppose that the following sequence of operations is executed using an initially empty **queue**. What ends up in the queue?

```
add A
add B
remove
add C
add D
```

```
remove
add E
remove
```

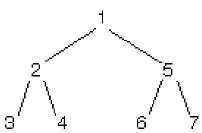
Solutions to Exercise 1 and Exercise 2

Exercise 1: `AC` Exercise 2: `DE`

General Design Concerns for a Tree Iterator

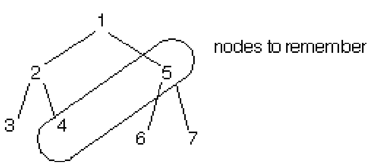
We now consider the problem of returning the elements of a tree one by one, using an iterator. To do this, we will implement the interface `java.util.Iterator`.

We will also use a nested iteration class to hide the details of the iteration. As with previous iterators, we need to maintain state saving information that lets us find the next tree element to return, and we now must determine what that information might include. To help work this out, we'll consider the sample tree below, with elements to be returned depth first as indicated by the numbers.



The first element to be returned is the one labeled "1". Once that's done, we need to somewhere keep track of the fact that we have to return at some point to element "2" and "5".

We return to "2" next and once we return element "2", we have to remember that element "3" and "4" are yet to return. Once we return "3" we still remember that we need to return to "4" and "5" as in the diagram below.



That means that our state-saving information must include not just a single pointer of what to return next, but a whole *collection* of "bookmarks" to nodes we've passed along the way.

More generally, we will maintain a collection that we'll call *fringe* or *frontier* of all the nodes in the tree that are candidates for returning next. The `next` method will choose one of the elements of the fringe as the one to return, then add its children to the fringe as candidates for the next element to return. `hasNext` is true when the fringe isn't empty.

The iteration sequence will then depend on the order we take nodes out of the fringe. Depth-first iteration, for example, results from storing the fringe elements in a *stack*, a last-in first-out structure. The `java.util` class library conveniently contains a `Stack` class with `push` and `pop` methods. We illustrate this process on a *binary tree* in which tree nodes have 0, 1, or 2 children named `myLeft` and `myRight`. Here is the code.

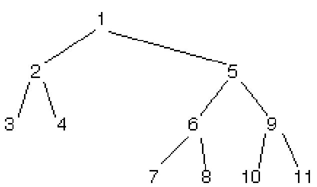
```
public class DepthFirstIterator implements Iterator {

    private Stack fringe = new Stack ( );

    public DepthFirstIterator ( ) {
        if (myRoot != null) {
            fringe.push (myRoot);
        }
    }

    public boolean hasNext ( ) {
        return !fringe.empty ( );
    }

    public Object next ( ) {
        if (!hasMoreElements ( )) {
```

```
            throw new NoSuchElementException ("tree ran out of elements");
        }
        TreeNode node = (TreeNode) fringe.pop ( );
        if (node.myRight != null) {
            fringe.push (node.myRight);
        }
        if (node.myLeft != null) {
            fringe.push (node.myLeft);
        }
        return node;
    }

    // We've decided not to show it for this example
    public void remove(){
        throw new UnsupportedOperationException();
    }
}
```

Stack Contents during Depth-First Iteration

Exercise 1

What numbers are on the stack when element 4 in the tree below has just been returned by `nextElement`?



Exercise 2

For the same image above, what numbers are on the stack when element 6 in the tree below has just been returned by `nextElement`?

Solutions to Exercises

Exercise 1: `5` Exercise 2: `7, 8, 9`

Effect of Pushing Left Child Before Right

Suppose the `nextElement` code pushes the left child before the right:

```
if (node.myLeft != null) {
    fringe.push (node.myLeft);
}
if (node.myRight != null) {
    fringe.push (node.myRight);
}
```

In what order are the elements of the tree below returned? Put a space between numbers.

Solution

Answer: `1 5 7 6 2 4 3`

A Depth-First Amoeba Iterator

Complete the definition of the `AmoebaIterator` class (within the `AmoebaFamily` class in ~cs61bl/su14/code/lab14). It should successively return names of amoebas from the family in preorder, with children amoebas returned oldest first. Thus, for the family set up in the `AmoebaFamily` main method, the name "Amos McCoy" should be returned by the first call to `next`; the second call to `next` should return the name "mom/dad"; and so on. Do not change any of the framework code.

Organizing your code as described in the previous step will result in a process that takes time proportional to the number of amoebas in the tree to return them all. Moreover, the constructor and `hasNext` both run in constant time, while `next` runs in time proportional to the number of children of the element being returned.

Uncomment the code at the end of the `AmoebaFamily` main method to test your solution. It should print names in the same order as the call to `family.print`, though without indenting.

A Breadth-First Amoeba Iterator

Now rewrite the `AmoebaIterator` class to use a *queue* (first in, first out) instead of a *stack*. (You might want to save your previous code too.) This will result in amoeba names being returned in breadth first order. That is, the name of the root of the family will be returned first, then the names of its children (oldest first), then the names of all their children, and so on. For the family constructed in the `AmoebaFamily` main method, your modification will result in the following iteration sequence:

```
Amos McCoy
mom/dad
auntie
me
Fred
Wilma
Mike
Homer
Marge
Bart
Lisa
Bill
Hilary
```

You may use either a list (your own `List` or the builtin `LinkedList`) to simulate a *queue*, or any subclass of the builtin `java.util.Queue` interface.

B. Build and Check a Tree

Printing a Tree

We now return to binary trees, in which each node has either 0, 1, or 2 children. Last lab, we worked with an implementation of binary trees using a BinaryTree class with a nested TreeNode class, as shown below.

```
public class BinaryTree {

    private TreeNode myRoot;

    private static class TreeNode {

        public Object myItem;
        public TreeNode myLeft;
        public TreeNode myRight;

        public TreeNode (Object obj) {
            myItem = obj;
            myLeft = myRight = null;
        }

        public TreeNode (Object obj, TreeNode left, TreeNode right) {
            myItem = obj;
            myLeft = left;
```

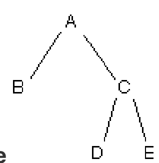
```
            myRight = right;
        }
    }
    ...
}
```

The framework is available in `~cs61bl/code/lab14/BinaryTree.java`. Fill in the blanks in the following code to print a tree so as to see its structure.

```
public void print ( ) {
    if (myRoot != null) {
        printHelper (myRoot, 0);
    }
}

private static final String indent1 = "    ";

private static void printHelper (TreeNode root, int indent) {
    ____ ;
    println (root.myItem, indent);
    ____ ;
}

private static void println (Object obj, int indent) {
    for (int k=0; k<indent; k++) {
        System.out.print (indent1);
    }
    System.out.println (obj);
}
```

The `print` method should print the tree in such a way that if you turned the printed output 90 degrees clockwise, you see the tree. Here's an example:

Tree



Printed Version

```
           E
        C
            D
    A
        B
```

Checking a `BinaryTree` object for validity

A legal binary tree has the property that, when the tree is traversed, no node appears more than once in the traversal. A careful programmer might include a `check` method to check that property:

```
public boolean check ( ) {
    alreadySeen = new ArrayList ( );
    try {
        isOK (myRoot);
        return true;
    } catch (IllegalStateException e) {
        return false;
    }
}
```

```
// Contains nodes already seen in the traversal.
private ArrayList alreadySeen;
(IllegalStateException is provided in Java.)
```

Write and test the `isOK` method, using variations of the sample trees you constructed for earlier exercises as test trees. Here's the header:

```
private void isOK (TreeNode t) throws IllegalStateException;
```

You may use any traversal method you like. You should pattern your code on earlier tree methods, for example, `size` and `busiest`.

Incidentally, this exercise illustrates a handy use of exceptions to return from deep in recursion.

Analyzing isOK's running time

Complete the following sentence.

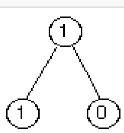
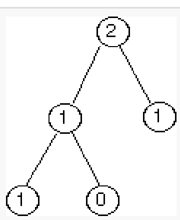
The `isOK` method, in the worst case, runs in time proportional to ______ , where `N` is ____________________ . Briefly explain your answer to your partner.

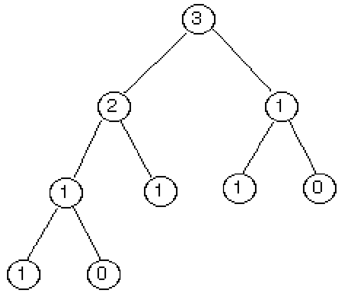
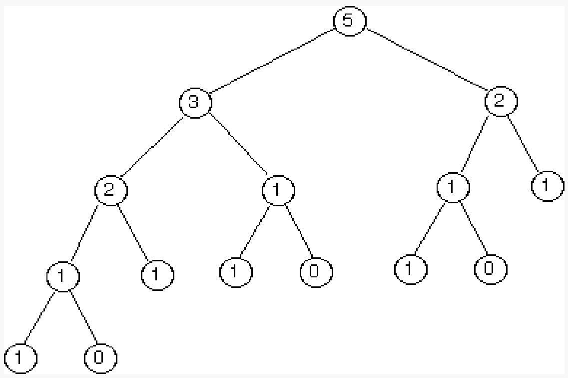
Solution

`N^2`, number of nodes in the tree

Building a Fibonacci Tree

This exercise deals with "Fibonacci trees", trees that represents the recursive call structure of the Fibonacci computation. (The Fibonacci sequence is defined as follows: F0 = 0, F1 = 1, and each subsequent number in the sequence is the sum of the previous two.) The root of a Fibonacci tree should contain the value of the `n`th Fibonacci number the left subtree should be the tree representing the computation of the `n-1`st Fibonacci number, and the right subtree should be the tree representing the computation of the `n-2`nd Fibonacci number. The first few Fibonacci trees appear below.

Function	Graph
`fibtree(0)`	
`fibtree(1)`	
`fibtree(2)`	
`fibtree(3)`	
`fibtree(4)`	

	
`fibtree(5)`	

Supply the `fibTreeHelper` method to go with the `BinaryTree` method below.

```
public static BinaryTree fibTree (int n) {
    BinaryTree result = new BinaryTree ( );
    result.myRoot = result.fibTreeHelper (n);
    return result;
}
```

```
private TreeNode fibTreeHelper (int n) ...
```

Note: primitive types like `int` are not objects. Java provides wrapper classes that allow primitive values to be stored as objects. In some situations, Java will automatically convert one to the other. However, for this exercise, you should use the `Integer` wrapper class. To create an `Integer` object, evaluate

```
new Integer (n)
```

where `n` is the integer you want to store. To access the stored integer value, use the `intValue` method.

Building an Expression Tree

Compilers and interpreters convert string representations of structured data into tree data structures. For instance, they would contain a method that, given a String representation of an expression, returns a tree representing that expression.

Copy the following code into your `BinaryTree` class. Complete and test the following helper method for `exprTree`. Your homework will build upon this code.

```
public static BinaryTree exprTree (String s) {
    BinaryTree result = new BinaryTree ( );
    result.myRoot = result.exprTreeHelper (s);
    return result;
}
// Return the tree corresponding to the given arithmetic expression.
// The expression is legal, fully parenthesized, contains no blanks,
// and involves only the operations + and *.
private TreeNode exprTreeHelper (String expr) {
```

```
        if (expr.charAt (0) != '(') {
            ____; // you fill this in
        } else {
            // expr is a parenthesized expression.
            // Strip off the beginning and ending parentheses,
            // find the main operator (an occurrence of + or * not nested
            // in parentheses, and construct the two subtrees.
            int nesting = 0;
            int opPos = 0;
            for (int k=1; k<expr.length()-1; k++) {
                ____ ; // you supply the missing code
            }
            String opnd1 = expr.substring (1, opPos);
            String opnd2 = expr.substring (opPos+1, expr.length()-1);
            String op = expr.substring (opPos, opPos+1);
            System.out.println ("expression = " + expr);
            System.out.println ("operand 1  = " + opnd1);
            System.out.println ("operator   = " + op);
            System.out.println ("operand 2  = " + opnd2);
            System.out.println ( );
            ____; // you fill this in
        }
    }
```

Given the expression `((a+(5*(a+b)))+(6*5))`, your method should produce a tree that, when printed using the `print` method you just designed, would look like

```
        5
    *
        6
+
                b
            +
                a
        *
            5
    +
        a
```

C. Homework

Readings

The next lab (lab #16) will cover binary search trees.

- JRS: Binary Search Trees
- CIDS: Chapter 9

Optimizing an Expression Tree

Given a tree returned by the `exprTree` method, write and test a method named `optimize` that replaces all occurrences of an expression involving only integers with the computed value. Here's the header.

`public void optimize ( )` It will call a helper method as did BinaryTree methods in earlier exercises. For example, given the tree produced for

```
((a+(5*(9+1)))+(6*5))
```

your optimize method should produce the tree corresponding to the expression

```
((a+50)+30)
```

Don't create any new `TreeNodes`; merely relink those already in the tree.

You will turn in your `BinaryTree.java` file as hw12.

© 2014 GitHub, Inc. Terms Privacy Security Contact