Lab 19 Binary Trees

Goal

In this lab you will modify a binary tree so that its nodes have parent references. Post-order traversal will be implemented using the parent references.

Resources

- Chapter 23: Trees
- Chapter 24: Tree Implementations

In `javadoc` directory

- *`BinaryTreeAccessInterface.html`*—API documentation for a binary tree with an embedded current node
- *`BinaryTreeInterface.html`*—API documentation for the binary tree ADT
- *`BinaryWithParentsTreeAccessInterface.html`*—API documentation for a binary tree with an embedded current node that supports moving to the parent
- *`TreeInterface.html`*—API documentation for the tree ADT

Java Files

- *`PreToPost.java`*
- *`TestBasicAccess.java`*
- *`TestBinaryTree.java`*
- *`TestParentAccess.java`*
- *`TestPostorderIterator.java`*

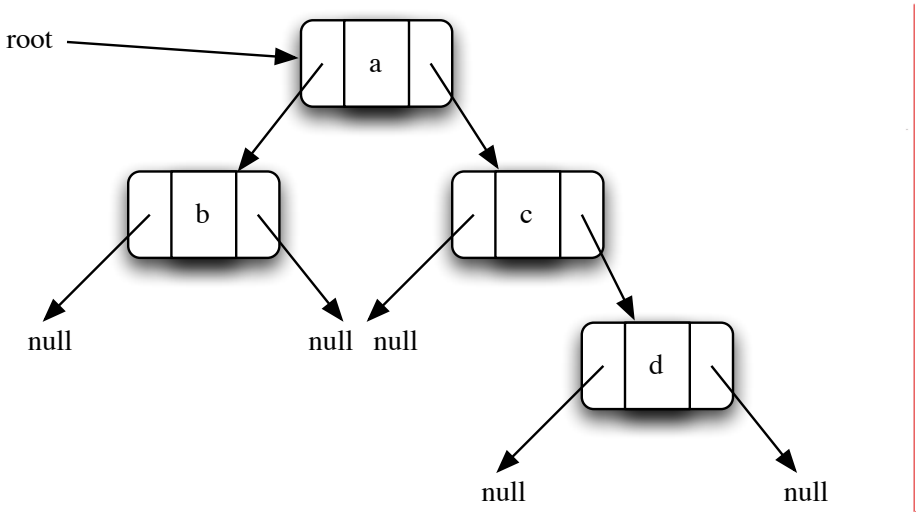
In `TreePackage` directory

- *`BinaryNode.java`*
- *`BinaryTree.java`*
- *`BinaryTreeAccessInterface.java`*
- *`BinaryTreeInterface.java`*
- *`BinaryWithParentsTreeAccessInterface.java`*
- *`TreeInterface.java`*

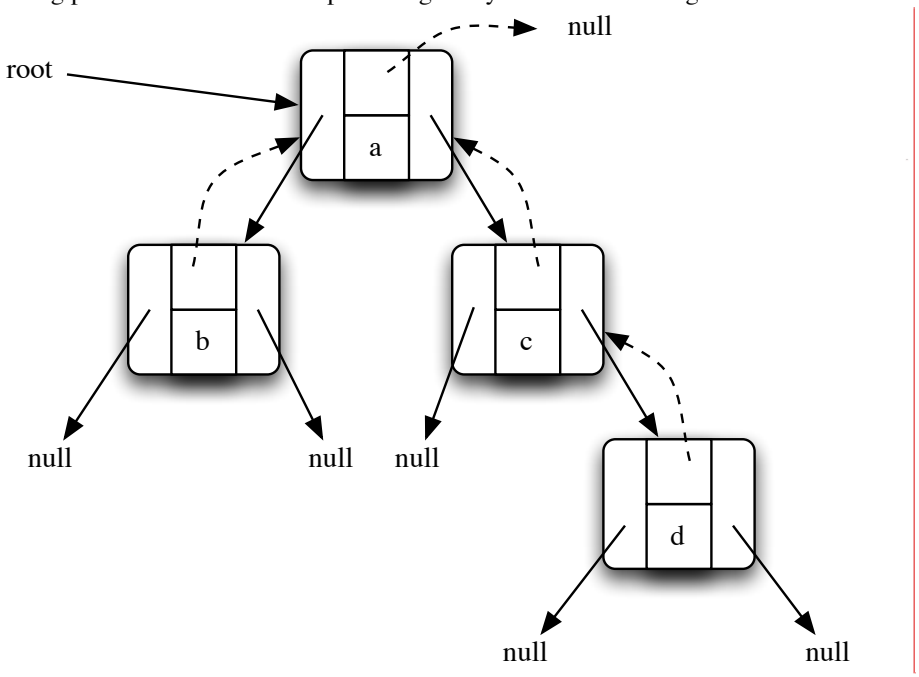
Introduction

Binary Tree

A binary tree is a tree where every node has zero, one, or two children. Each node will have two links, one for each child. If the there is no child, the link will be null. The following is an example of a binary tree.



To travel in the tree, you must start at the root and traverse either right or left links until you reach the desired node. Just as a chain can be doubly linked to make it easier to move backward, you can add parent references to the nodes. Adding parent references to the preceding tree yields the following tree.



Pre-Lab Visualization

Constructing a Binary Tree

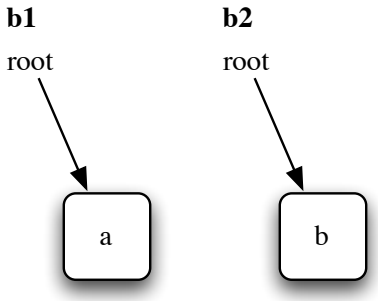
There is one detail about the implementation of `BinaryTree` that is important to understand for this lab. Consider the following chunk of code.

```
BinaryTree b1 = new BinaryTree("a");
BinaryTree b2 = new BinaryTree("b);
BinaryTree b3 = new BinaryTree("c", b1, b2);
BinaryTree b4 = new BinaryTree("d", b3, b3);
```

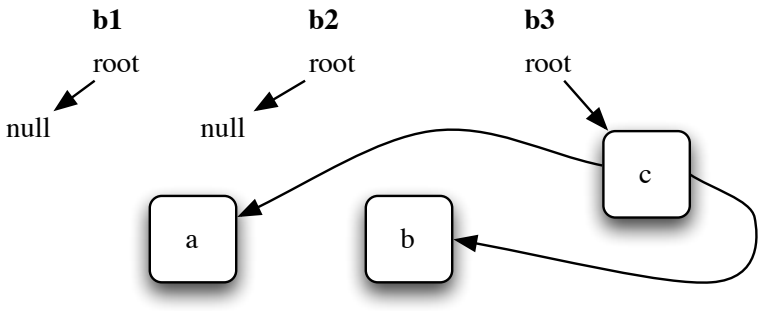
Make a prediction of the contents and structure of each tree at the end of the code.

b1 **b2** **b3** **b3**

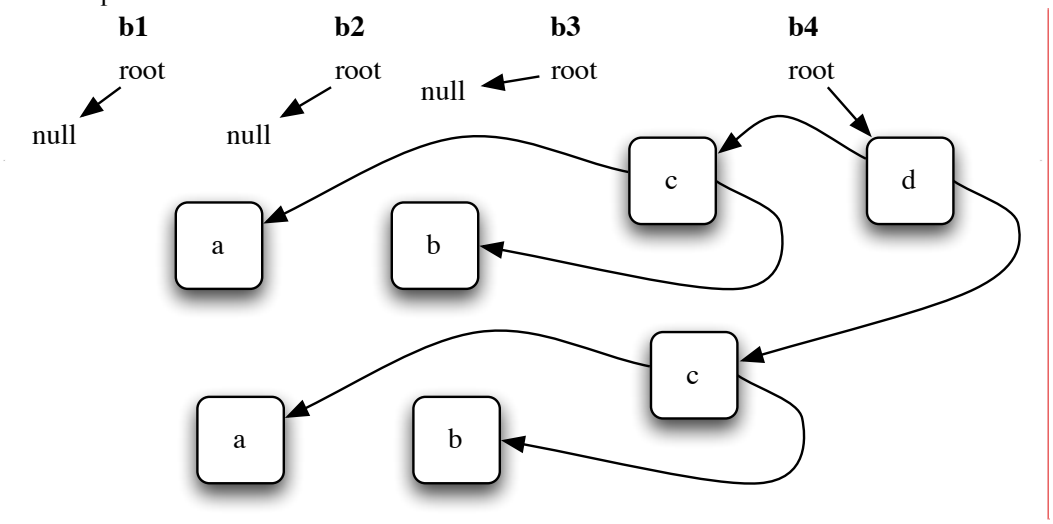
The first two statements each create a tree containing one node. In the following trees, null references out of a node are not drawn.



The third statement does not necessarily do what one might expect. The nodes from the two trees `b1` and `b2` are used to build the tree `b3`. It might not be expected that `b1` and `b2` no longer have access to those trees, but it is important for the integrity of `b3`. (Refer to the code in method `privateSetTree` in the class `BinaryTree`.)



The final statement is requesting `b3` twice. Since both subtrees cannot share the nodes, a copy must be made. The final picture is



Basic Access to a Binary Tree

As defined in the class `BinaryTree`, there is not much you can do with the binary tree except to build and traverse it. For this lab an interface `BinaryTreeAccessInterface` has been defined that will allow a client the ability to move through the tree. This is similar to the `DecisionTreeInterface`, which was defined in Chapter 25.

There is a current reference, which is kept by the binary tree. It can be reset to the root of the tree. Query methods are provided to allow the client to ask if the current node has right or left children. There are methods that allow the client to move the current reference to the right or left child. Finally, there is a method that will allow the client to get the data in the node that is the current reference.

There are two fundamental questions that must be answered for the access.

What happens if the reference is advanced but the child does not exist?
What happens if the tree is empty?

One possibility is for the bad advance to throw an exception. Another possibility is to leave the reference where it is. A third possibility is to let the reference become null. All three are reasonable choices. In this implementation, the third option is chosen.

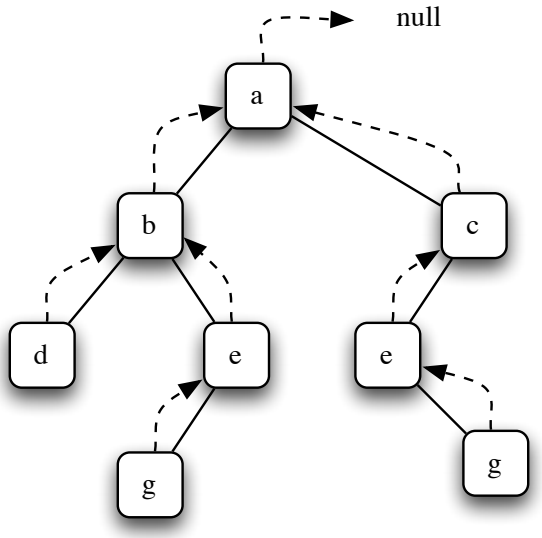
Once the current access becomes null, it should not be used. If the client asks for data, just return null. If the client attempts to move, leave it null. If the tree is empty, the current access will be null and it will behave in a reasonable way.

Once parent references have been added to the tree, the access interface will be extended with a method to query if the current node has a parent and a second method to move the current reference to the parent.

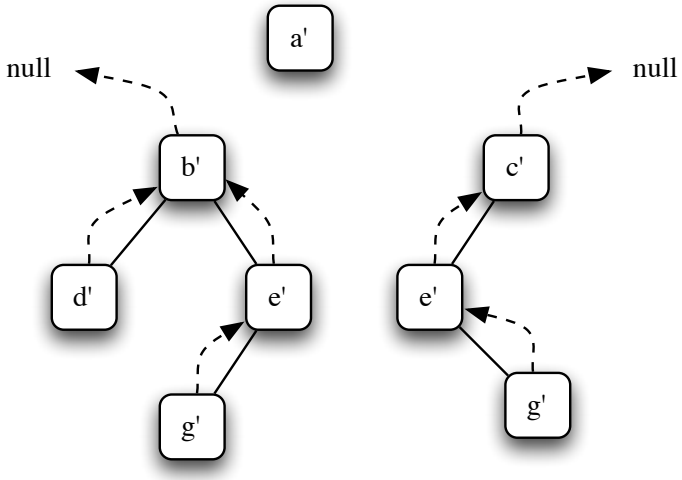
Parent References

For the most part, adding parent references to a binary tree is a simple operation. There are only two places where a parent reference will need to be changed. The first is `privateSetTree()` in `BinaryTree`. It is the place where two trees are composed together to create a larger tree. The parent references of the roots of the two subtrees must be fixed to point to the new root. The other place where parent references must be fixed is not as obvious.

Consider the copy operation from `BinaryNode`. It will make a copy of the top node, then recursively copy the subtrees. Note that the links cannot just be copied. If they were, they would point to the original nodes. Suppose that the following tree is going to be copied.



We assume that copies of the subtrees will be made recursively. Once this is done, it just remains to link up the subtrees with a copy of the root node. The following picture shows the state of the copy at that point.

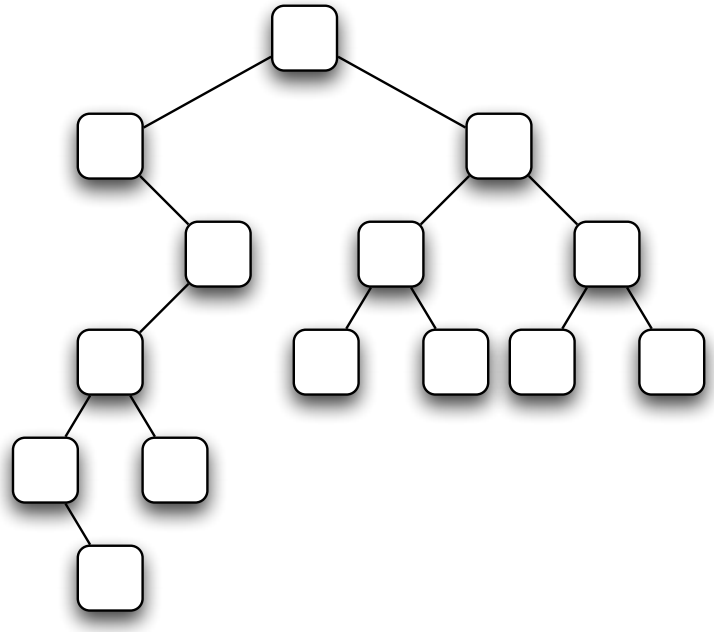


Show the links that must be formed. In the lab, a second version of the copy operation will be created that takes a single parameter, which will be the parent node for the copy.

Post-Order Iterator

The strategy employed by the iterator is to keep a reference to the node that will be produced by the `next` method. When `next` is called, the value to be returned is retrieved from the referenced node. Then the reference will be moved to the next node.

In the following tree, label the nodes numerically with the order that they will be visited by a post-order traversal.



Constructor:
The constructor for the post-order iterator must find the first node in a post-order traversal. Give an algorithm for that operation.

HasNext():
This method is nearly trivial. What is the condition for `hasNext()` to return true?

Next():
The method `next()` must move to the next node in the post-order traversal. Examine the preceding tree and answer the following questions.

Can the next node ever be below the current node?

Is the next node ever not the parent of the current node?

How can those cases be recognized? Write down a condition.

If the next node is not the parent, give an algorithm for finding it.

Pre-Order Expressions

One way to construct an arithmetic expression is to use a prefix notation. The operations come before the arguments they will be applied to. Let's look at the problem of constructing a binary tree from a prefix expression. Consider the following three prefix expressions

```
*  +  1  /  2  3  -  4  5
+  1  +  +  3  4  5
-  +  3  4  /  2  +  3  4
```

Draw a binary tree for each of the expressions where the leaf nodes are values and the interior nodes are binary operations. A pre-order traversal must produce the prefix expression.

In the preceding prefix expressions; circle the tokens (values and operators) that are in the left subtree. Circle the tokens in the right subtree. This hints that a recursive algorithm can be used to construct the binary tree from the prefix expression.

Give a recursive design for `getTree` that constructs a binary tree from a prefix expression.

 **Identify the problem**:

 **Identify the smaller problems**:

 **Identify how the answers are composed**:

 **Identify the base cases**:

 **Compose the recursive definition**:

Show the operation of your definition on the prefix expression

```
* + 1 / 2 3 - 4 5
```

Directed Lab Work

The main classes that you will be working with today are `BinaryTree` and `BinaryNode`. Take a look at them now, if you have not done so already. To verify that these classes work, you can run `TestBinaryTree`.

Modifying BinaryTree for BasicAccess

The first task is to add methods to the BinaryTree class to implement the basic access given by BinaryTreeAccessInterface.

Step 1. Change the declaration of the `BinaryTree` so that it implements `BinaryTreeAccessInterface`.

Step 2. In `BinaryTree`, add the six methods required by `BinaryTreeAccessInterface`.

Step 3. In `BinaryTree`, add a private variable of type `BinaryNode<T>` that will reference the current access node.

Step 4. Implement each of the methods just added. Refer to the pre-lab.

Step 5. Anytime the structure of the binary tree is changed, the access needs to be reset. Call `resetAccess()` in the constructors, `setTree()`, and `clear()`.

Checkpoint: Compile BinaryNode and BinaryTree. Run TestBinaryTree and TestBasicAccess. All tests should pass.

Modifying BinaryTree with Parent References

Step 6. In the class `BinaryNode`, add a private variable that will hold the parent reference.

Step 7. Add a new constructor, which has four arguments: `data`, `left`, `right`, and `parent`.

Step 8. Modify the constructor that takes three arguments to use the new constructor with `null` for the parent.

Step 9. Create and fully implement three new methods in `BinaryNode`:

```
public BinaryNode<T> getParent()
public void setParent(BinaryNode<T> p)
public boolean hasParent()
```

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

Step 10. Make a duplicate of the method `copy()` in `BinaryNode` and add an argument `BinaryNode<T> p` to the duplicate.

Step 11. In the duplicate, set the parent of `newRoot` to be `p`.

Step 12. In the original, set the parent of `newRoot` to be `null`. (We will assume that if the original version of the copy method is being called, it is the top of the tree being copied and the parent should be null. If not, it is the responsibility of the method that called copy to set it correctly.)

Step 13. In both the original and the duplicate, change the two recursive calls to `copy()` so that they pass `newRoot` as the parameter.

Checkpoint: BinaryNode should compile successfully.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

The modification of BinaryNode is finished. The next goal is to modify BinaryTree appropriately. Any time a new binary tree is created, parent references for children may need to be set.

Step 14. Anywhere in `BinaryTree` that a left or right child is set, set a parent reference in an appropriate fashion.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

It is now time to add in the methods that will allow a client to move the current access to follow a parent reference.

Step 15. Change the declaration of the `BinaryTree` so that it implements `BinaryWithParentsTreeAccessInterface`.

Step 16. Add and implement the three additional methods required by the interface.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

It is now time to see if the changes to BinaryNode and BinaryTree have really worked. All tests in TestParentAccess should pass.

Implementing a Post-Order Iterator with Parent References

Step 17. In the class `BinaryTree`, create a copy of the private inner class `InorderIterator`. Rename the copy to `PostorderIterator`.

Step 18. Keep the code in the methods for future reference. The private variable `currentNode` will be kept, but `nodeStack` will be deleted.

Step 19. Refer to the pre-lab exercises and create a method in `PostorderIterator` that will move the current node to the first node to be printed in a post-order traversal.

Step 20. Call the new method in the constructor just after setting the `currentNode` to the root.

Step 21. Complete the `hasNext()` method.

Step 22. Refer to the pre-lab exercises and complete the `next()` method. Don't forget to throw `NoSuchElementException` when there are no more elements to be iterated over.

Step 23. Remove `nodeStack` from `PostorderIterator.`

Step 24. Change the method `getPostorderIterartor()` so that it returns a new `PostorderIterator` instead of throwing an exception.

Checkpoint: Compile BinaryTree. All previous test code should still pass.
All tests in TestPostorderIterator should pass.

Now that BinaryTree has a post-order iterator using parent references, it will be used to produce a post-order traversal of a binary expression tree.

Converting Prefix Expressions to Postfix

Step 25. The application `PreToPost` exists but needs to be completed.

*Checkpoint: The application should run. Enter the prefix expression * 3 4. The application will read the expression, then quit.*

Step 26. In the `main`, create a new `Scanner` that will break up the string that was read into tokens. Use `next()` to get the tokens as strings.

Step 27. Complete the method `getTree()` in `PreToPost`. Refer to the recursive design in the pre-lab exercises.

Step 28. Call the method `getTree()` in `main` to create the binary expression tree.

Step 29. Create a post-order iterator for the expression tree.

Step 30. Use the iterator to print out the values in the expression tree.

*Final checkpoint: The application should run. Enter the prefix expression * 3 4. The result should be 3 4 *.*

Run the application again. Enter the prefix expression - - / 4 3 + 8 7 2. The result should be 4 3 / 8 7 + - 2 -.

Test the application for other inputs. Try expressions that have too many or too few tokens.

Post-Lab Follow-Ups

1. Change the access interface so that it will throw an exception if there is no data value to return. Also change it so that if the client attempts to move to an empty child another exception is thrown. Modify `BinaryNode` and `BinaryTree` to satisfy the new access methods. Modify the `TestBasicAccess` and `TestParent` to test for the exceptions.

2. You can implement an in-order traversal without a stack by using the parent references. If the current node has a right child, you search down the tree for the next node. If the current node doesn't have a right child, you search up the tree for the next node. Implement this version of an in-order iterator.

3. Use the parent references along with checking the direction of backtracking to implement the pre-order iterator.

4. Another way that the iterators can be implemented is to mark the nodes that have been visited by the iterator. This will require an extra variable stored in the node to remember the marking. One problem with this implementation is that there can only be one iterator at a time. Use the parent references along with marking to implement the post-order iterator. Change `BinaryTree` so that there is a single instance of the iterator which is held by an instance variable. It should be the only object returned by the `getPostorderIterator()` method. You will need to have a `reset` method for the iterator that will unmark all of the items.

5. Use the parent references along with marking to implement the in-order iterator.

6. Use the parent references along with marking to implement the pre-order iterator.

7. Modify the application to use a string tokenizer to break up the arithmetic expressions using spaces and the four operators as delimeters.

8. Develop and implement a recursive method that will evaluate a binary expression tree.

9. Modify the application to work with Boolean expressions that use the binary operators ^ (and) and v (or) and the unary operator ~ (not).