Reading LDC Ch 17.1, 17.2

Each tree node can be defined using a separate class – similar to LinearNode or linked lists.

Nodes contain a reference to the data stored in the node, and references for each of the possible children of the node:
- Binary tree: 2 references required – left and right children
- n-ary tree: n references required – one for each possible child

Trees organized this way lend themselves to recursive processing for many operations.
A possible set of operations for a binary tree is shown in the BinaryTree interface.

BinaryTree has no methods to add a particular element, or to remove a particular element from the tree.

Refined versions of binary tree (such as binary search trees) will define those methods based on specific characteristics.

BinaryTree is still useful in certain situations.

```java
package javafoundations;

public class BTNode<T> {
    protected T element;
    protected BTNode<T> left, right;

    // Creates a new tree node with the specified data.
    public BTNode (T elmt) {
        element = elmt;
        left = right = null;
    }

    // Returns the element stored in this node.
    public T getElement() {
        return element;
    }

    // Sets the element stored in this node.
    public void setElement (T element) {
        this.element = element;
    }

    // Returns the left subtree of this node.
    public BTNode<T> getLeft() {
        return left;
    }

    // Sets the left child of this node.
    public void setLeft (BTNode<T> left) {
        this.left = left;
    }

    // Returns the right subtree of this node.
    public BTNode<T> getRight() {
        return right;
    }

    // Sets the right child of this node.
    public void setRight (BTNode<T> right) {
        this.right = right;
    }
}
```

```java
// Returns the element in this subtree that matches the specified target. Returns null if the target is not found.
public BTNode<T> find (T target) {
    BTNode<T> result = null;
    if (element.equals(target)) {
        result = this;
    } else {
        if (left != null) {
            result = left.find(target);
        }
        if (result == null && right != null) {
            result = right.find(target);
        }
    }
    return result;
}
```

```java
// Returns the element stored in this node.
public T getElement() {
    return element;
}
```

```java
// Returns the element in this subtree that matches the specified target. Returns null if the target is not found.
public BTNode<T> find (T target) {
    BTNode<T> result = null;
    if (element.equals(target)) {
        result = this;
    } else {
        if (left != null) {
            result = left.find(target);
        }
        if (result == null && right != null) {
            result = right.find(target);
        }
    }
    return result;
}
```
*An iterator is an object that provides a **means of processing a collection of objects**, one at a time.

**Method Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>Returns true if the iteration has more elements.</td>
</tr>
<tr>
<td>next()</td>
<td>Returns the next element in the iteration.</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes from the underlying collection the last element returned by the iterator (optional operation).</td>
</tr>
</tbody>
</table>

*By implementing the Iterator interface, a class formally establishes that: objects of that type are iterators.*

*Now, the **for-each** version of the **for** loop can be used to process the items in the iterator.*
public void add(T item)
{
    if (count == items.length)
        expandCapacity();
    items[count] = item;
    count++;
}

private void expandCapacity()
{
    T[] larger = (T[]) (new Object[items.length*2]);
    int location = 0;
    for (T element : items)
        larger[location++] = element;
    items = larger;
}

public boolean hasNext()
{
    return (current < count);
}

public T next()
{
    if (! hasNext())
        throw new NoSuchElementException();
    current++;
    return items[current - 1];
}

public void remove()
throws UnsupportedOperationException
{
    throw new UnsupportedOperationException();
}
public LinkedBinaryTree<T> getLeft()
{
    if (root == null)
        throw new EmptyCollectionException("Get left failed. Tree is empty.");
    LinkedBinaryTree<T> result = new LinkedBinaryTree<T>();
    result.root = this.root.getLeft();
    return result;
}
**A decision tree** is a tree whose nodes represent decision points, and whose children represent the options available.

The **leaves** of a decision tree represent the possible conclusions that might be drawn based on the answers.

Decision trees are used in **expert systems** – software that attempts to represent the knowledge of an expert in a particular field.

Decision tree with yes/no binary tree

**Expertise examples**
- a doctor
- a car mechanic
- accountant
- PC help desk!?!?

----

**IMPORTANT:**
To test this file you need to create a driver OUTSIDE javafoundations. You cannot do it by adding a main() method in LinkedBinaryTree.java.

I’ll pause for a moment so you can let this information sink in.

The left child represents the answer “No”
The right child represents the answer “Yes”
public class BackPainAnalyzer
{
    // Asks questions of the user to diagnose a medical problem.
    public static void main (String[] args)
    {
        BackPainExpert expert = new BackPainExpert();
        expert.diagnose();
    }
}

public class BackPainExpert
{
    private LinkedBinaryTree<String> tree;

    public BackPainExpert() {
        String e1 = "Did the pain occur after a blow or jolt?";
        String e2 = "Do you have a fever?";
        String e3 = "Do you have difficulty controlling your arms or legs?";
        String e4 = "Do you have persistent morning stiffness?";
        // (etc)
        n7 = new LinkedBinaryTree<String>(e7);
        n2 = new LinkedBinaryTree<String>(e2, n4, n5);
        n3 = new LinkedBinaryTree<String>(e3, n6, n7);
        tree = new LinkedBinaryTree<String>(e1, n2, n3);
    }

    public void diagnose() {
        Scanner scan = new Scanner(System.in);
        LinkedBinaryTree<String> current = tree;
        System.out.println ("So, you’re having back pain.");
        while (current.size() > 1) {
            System.out.println (current.getRootElement());
            if (scan.nextLine().equalsIgnoreCase("N"))
                current = current.getLeft();
            else
                current = current.getRight();
        }
        System.out.println (current.getRootElement());
    }
}