BinaryTree

• The **BinaryTree** component family allows you to manipulate values modeled as mathematical binary trees with any label type $T$ (i.e., *binary tree of $T$*)
  – Another generic type like *Sequence* and *Set*
Interfaces and Classes

`Stdard` extends `BinaryTreeKernel`

`Iterable` extends `BinaryTreeKernel`

`BinaryTreeKernel` has contracts for three methods:
- `assemble`
- `disassemble`
- `size`

`BinaryTree1` implements `BinaryTree`
BinaryTree has contracts for four methods (the last of which we will skip):

- root
- replaceRoot
- height
- inOrderAssemble
Interfaces and Classes

There is really an abstract class as usual in the chain here, but it is not shown because these slides describe the client view, and a client needs only interface BinaryTree and class BinaryTree1.
Mathematical Model

\text{type } BinaryTreeKernel \text{ is modeled by binary tree of } T
No-argument Constructor

• Ensures:

\[ this = empty\_tree \]
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BinaryTree&lt;NaturalNumber&gt; bn = new BinaryTree1&lt;&gt;();</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BinaryTree&lt;NaturalNumber&gt; bn = new BinaryTree1&lt;&gt;();</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bn = □</code></td>
</tr>
</tbody>
</table>
assemble

```c
void assemble(T root, BinaryTree<T> left, BinaryTree<T> right)
```

- Assembles in `this` a binary tree with root label `root` and subtrees `left` and `right`; the declaration notwithstanding, the `dynamic` type of `left` and `right` must be the same as the `dynamic` type of `this`.
- Aliases: reference `root`
- Replaces: `this`
- Clears: `left, right`
- Ensures:
  ```c
  this = compose(root, #left, #right)
  ```
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x = 70 ] ( bn = ? )</td>
<td>( lt = )</td>
</tr>
<tr>
<td></td>
<td>( rt = )</td>
</tr>
<tr>
<td>\texttt{bn.assemble(x, lt, rt);}</td>
<td></td>
</tr>
</tbody>
</table>
### Code

```java
x = 70    bn = ?
lt = ▲
rt = △
```

```java
bn.assemble(x, lt, rt);
```

### State

```
x = 70    bn = 70
lt = ▲
rt = △
```
Example

Note the alias created here, which you cannot see in the tracing table.

bn.assemble(x, lt, rt);

<table>
<thead>
<tr>
<th>State</th>
<th>x = 70</th>
<th>bn = ?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

lt = ✅
rt = ✅
**disassemble**

T disassemble(BinaryTree<T> left, BinaryTree<T> right)

- Disassembles **this** into its root label, which is returned as the value of the function, and subtrees *left* and *right*; the declaration notwithstanding, the *dynamic* type of *left* and *right* must be the same as the *dynamic* type of **this**.
- Replaces: *left, right*
- Clears: **this**
- Requires:
  
  ```
  this /= empty_tree
  ```
- Ensures:
  
  ```
  #this = compose(disassemble, left, right)
  ```
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lt = ?} \hspace{1cm} \texttt{bn = \textcolor{green}{13}} \hspace{1cm} \texttt{rt = ?}</td>
<td>\texttt{NaturalNumber root =} &lt;br&gt; \texttt{bn.disassemble(lt, rt);}</td>
</tr>
</tbody>
</table>

- \texttt{bn.disassemble(lt, rt)}
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| `lt = ?  bn = 13  rt = ?`
| NaturalNumber root =  
bn.disassemble(lt, rt); | `root = 13  bn = 13  lt = 13  rt = 13` |
size

int size()

• Reports the size of this.
• Ensures:

\[ size = |this| \]
iterator

Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

   ~this.seen * ~this.unseen = IN_ORDER(this)
Traversal Orders

• There are three named *traversal orders* for the nodes of a binary tree, named according to when the root is “visited” relative to the (recursive) traversals of the left and right subtrees
  – *Pre-order*: root is visited *before* left and right
  – *In-order*: root is visited *between* left and right
  – *Post-order*: root is visited *after* left and right
Traversal Orders

- There are three named traversal orders for the nodes of a binary tree, named according to when the root is visited relative to the (recursive) traversals of the left and right subtrees:
  - **Pre-order**: root is visited before left and right
  - **In-order**: root is visited between left and right
  - **Post-order**: root is visited after left and right

The `iterator` method returns an `Iterator<T>` that visits the node labels in this order.
- Pre-order traversal: \(<4, 2, 1, 3, 5>\)
- In-order traversal: \(<1, 2, 3, 4, 5>\)
- Post-order traversal: \(<1, 3, 2, 5, 4>\)
root

T root()

• Reports the root of this.
• Aliases: reference returned by root
• Requires:
  \( \text{this} \neq \text{empty\_tree} \)
• Ensures:
  \( \text{there exists } lt, rt: \text{binary tree of } T \quad (\text{this} = \text{compose}(\text{root}, lt, rt)) \)
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bn = bn.root();</code></td>
<td><code>bn = 13</code></td>
</tr>
</tbody>
</table>

```
NaturalNumber k = bn.root();
```
Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NaturalNumber k = bn.root();}</td>
<td>\texttt{bn = 13}</td>
</tr>
<tr>
<td></td>
<td>\texttt{k = 13} \texttt{bn = 13}</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>NaturalNumber</th>
<th>bn.root();</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note the alias created here, which you cannot see in the tracing table.

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>bn = 13</td>
</tr>
</tbody>
</table>

\[
k = 13 \quad bn = 13
\]
replaceRoot

T replaceRoot(T x)

• Replaces the root of this with x, and returns the old root.
• Aliases: reference x
• Requires:
  
  \( \text{this} \neq \text{empty\_tree} \)
• Ensures:
  
  \( \text{there exists } lt, rt: \text{binary tree of } T \)
  
  \( (#\text{this} = \text{compose}(\text{replaceRoot}, lt, rt) \text{ and this} = \text{compose}(\text{x}, lt, rt)) \)
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{n = 4} \texttt{bn = 13}</td>
<td></td>
</tr>
<tr>
<td>\texttt{NaturalNumber k =}</td>
<td></td>
</tr>
<tr>
<td>\texttt{bn.replaceRoot(n);}</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| \[
\text{NaturalNumber } k = \\
bn.\text{replaceRoot}(n);
\]
| \[
\begin{align*}
n &= 4 \\
bn &= 13
\end{align*}
\] | \[
\begin{align*}
n &= 4 \\
bn &= 4 \\
k &= 13
\end{align*}
\] |
Example

Note the alias created here, which you cannot see in the tracing table.

```java
NaturalNumber k = bn.replaceRoot(n);
```

<table>
<thead>
<tr>
<th>n = 4</th>
<th>bn = 13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n = 4</th>
<th>bn = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>k = 13</td>
<td></td>
</tr>
</tbody>
</table>
## Another Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n = bn.replaceRoot(n);</code></td>
<td><code>n = 4  bn = 13</code></td>
</tr>
</tbody>
</table>
Another Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 4 ) ( bn = 13 )</td>
<td></td>
</tr>
<tr>
<td>( n = bn.\text{replaceRoot}(n); )</td>
<td></td>
</tr>
<tr>
<td>( n = 13 ) ( bn = 4 )</td>
<td></td>
</tr>
</tbody>
</table>
Another Example

This use of the method avoids creating an alias: it **swaps** \( n \) with the old root.

\[
n = bn.replaceRoot(n);
\]

<table>
<thead>
<tr>
<th>State</th>
<th>( n = 4 )</th>
<th>( bn = 13 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 13 )</td>
<td>( bn = 4  )</td>
<td></td>
</tr>
</tbody>
</table>

9 January 2015
height

```c
int height()

• Reports the height of this.
• Ensures:
  \[ height = h(t)(this) \]
```
Resources

• OSU CSE Components API: `BinaryTree`