Tree
Tree

- The *Tree* component family allows you to manipulate values modeled as mathematical trees with any label type $T$ (i.e., *tree of* $T$)
Interfaces and Classes

- Standard
- Iterable

extends

TreeKernel

extends

Tree

implements

Tree1
**Interfaces and Classes**

- **Standard**
- **Iterable**
- **TreeKernel** extends **Standard** and **Iterable**

- **Tree** implements **TreeKernel**
- **Tree1** extends **Tree**

**TreeKernel** has contracts for four methods:
- newSequenceOfTree
- assemble
- disassemble
- size
Tree has contracts for six methods:
  root
  replaceRoot
  height
  addSubtree
  removeSubtree
  numberOfSubtrees
Mathematical Model

type TreeKernel is modeled by
tree of $T$
No-argument Constructor

• Ensures:

\[ \textit{this} = \textit{empty\_tree} \]
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Tree&lt;NaturalNumber&gt; tn = new Tree1&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>Tree&lt;NaturalNumber&gt; tn = new Tree1&lt;&gt;();</code></td>
<td><code>tn = □</code></td>
</tr>
</tbody>
</table>
nullSequenceOfTree

Sequence<Tree<T>> newSequenceOfTree()

• Creates and returns an empty Sequence<Tree<T>> of the dynamic type needed in assemble and disassemble.

• Ensures:

    newSequenceOfTree = < >
newSequenceOfTree

Sequence<Tree<T>>

• Creates and returns an empty Sequence<Tree<T>> needed in assemble and disassemble.

• Ensures:

  newSequenceOfTree = < >

The mathematical model of Sequence<Tree<T>> is string of tree of $T$, so this value (the empty string) has the right type.
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sequence&lt;Tree&lt;NaturalNumber&gt;&gt; st = tn.newSequenceOfTree();</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>`Sequence&lt;Tree&lt;NaturalNumber&gt;&gt; st =</td>
<td><code>st = &lt; &gt;</code></td>
</tr>
<tr>
<td>tn.newSequenceOfTree();`</td>
<td></td>
</tr>
</tbody>
</table>
Example

Sequence<Tree<NaturalNumber>> st =
  tn.newSequenceOfTree();

Note that the value of the receiver does not matter, only its type.
assemble

```cpp
tvoid assemble(T root,
    Sequence<Tree<T>> children)
```

- Assembles in `this` a tree with root label `root` and subtrees `children`; the declaration notwithstanding, the `dynamic` type of each entry of `children` must be the same as the `dynamic` type of `this` and the `dynamic` type of `children` must be the same as that returned by `newSequenceOfTree`.

- Aliases: reference `root`
- Replaces: `this`
- Clears: `children`
- Ensures:
  ```
  this = compose(root, #children)
  ```

27 February 2019 OSU CSE
## Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 70$</td>
<td>$tn = ?$</td>
</tr>
<tr>
<td>$st = &lt;\bigtriangleup, \bigtriangledown, \square&gt;$</td>
<td></td>
</tr>
<tr>
<td>$\text{tn.assemble}(x, st);$</td>
<td></td>
</tr>
</tbody>
</table>
Example

\begin{align*}
\text{State} & \quad x = 70 \quad tn = ? \\
\text{st} & = < \text{ }, \text{ }, \text{ } > \\
\text{tn.assemble}(x, \text{st}); & \\
\end{align*}

How could \text{st} get a non-empty value? By adding trees to it using the \text{add} method for \text{Sequence}. 
## Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 70$</td>
<td>$x = 70$</td>
</tr>
<tr>
<td>$tn = ,$</td>
<td>$tn = 70$</td>
</tr>
<tr>
<td>$st = &lt; \boxed{3}, \boxed{2}, \boxed{1} &gt;$</td>
<td>$st = &lt; &gt;$</td>
</tr>
<tr>
<td>$tn$.assemble$(x, st)$;</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing the state after assembling](image)
Example

```
Code State
x = 70    tn = ?
= < △, ▲, □ >
```

```
tn.assemble(x, st);
```

```
State
x = 70    tn = 70
st = < >
```

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

T disassemble(Sequence<Tree<T>> children)

• Disassembles \textit{this} into its root label, which is returned as the value of the function, and subtrees in \texttt{children}; the declaration notwithstanding, the \textit{dynamic} type of \texttt{children} must be the same as that returned by \texttt{newSequenceOfTree}.

• Replaces: \texttt{children}

• Clears: \texttt{this}

• Requires:
  \[
  \texttt{this} \neq \texttt{empty\_tree}
  \]

• Ensures:
  \[
  #\texttt{this} = \texttt{compose}(\texttt{disassemble}, \texttt{children})
  \]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>st = ?</code></td>
<td><code>tn = 13</code></td>
</tr>
<tr>
<td><code>NaturalNumber root =</code></td>
<td></td>
</tr>
<tr>
<td><code>tn.disassemble(st);</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>st = ?  tn = 13</td>
</tr>
<tr>
<td><code>NaturalNumber root = 13</code></td>
<td>root = 13  tn = 13</td>
</tr>
<tr>
<td><code>tn.disassemble(st);</code></td>
<td>st = &lt; ■, △ &gt;</td>
</tr>
</tbody>
</table>
size

```c
int size()
```

- Reports the size of `this`.
- Ensures:
  
  ```
  size = |this|
  ```

27 February 2019 OSU CSE
iterator

Iterator<T> iterator()

- Returns an iterator over a multiset of elements of type T.
- Ensures:

\[ ~this.seen \ast ~this.unseen = \text{PRE\_ORDER}(this) \]
root

T root()

• Reports the root of this.
• Aliases: reference returned by root
• Requires:
  \( this \neq empty\_tree \)
• Ensures:
  \( there\ exists\ children:\ string\ of\ tree\ of\ T \)
  \( (this = compose(root, children)) \)
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>NaturalNumber k =</td>
<td></td>
</tr>
<tr>
<td>tn.root();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$tn = 13$</td>
</tr>
</tbody>
</table>
**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tn = NaturalNumber k = tn.root();</code></td>
<td><code>tn = 13</code></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td><code>k = 13</code> <code>tn = 13</code></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Example

```
NaturalNumber tn;
qn.root();
```

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

<table>
<thead>
<tr>
<th></th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>tn</td>
<td>13</td>
</tr>
<tr>
<td>k</td>
<td>13</td>
</tr>
</tbody>
</table>
replaceRoot

T replaceRoot(T x)

• Replaces the root of this with x, and returns the old root.
• Aliases: reference x
• Requires:
  
  this /= empty_tree

• Ensures:
  
  there exists children: string of tree of T
  
  (#this = compose(replaceRoot, children) and
  
  this = compose(x, children))
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 4$</td>
<td>$tn = 13$</td>
</tr>
<tr>
<td><code>NaturalNumber k =</code></td>
<td></td>
</tr>
<tr>
<td><code>tn.replaceRoot(n);</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

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

<table>
<thead>
<tr>
<th>$n = 4$</th>
<th>$tn = 13$</th>
</tr>
</thead>
</table>

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

```
NaturalNumber k =
  tn.replaceRoot(n);
```

<table>
<thead>
<tr>
<th>$n = 4$</th>
<th>$tn = 4$</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>$n = tn.replaceRoot(n);$</td>
<td>$n = 4 \quad tn = 13$</td>
</tr>
</tbody>
</table>
Another Example

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

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

<table>
<thead>
<tr>
<th>State</th>
<th>$n = 4$</th>
<th>$tn = 13$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = tn.replaceRoot(n);$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n = 13$</td>
<td>$tn = 4$</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing state transitions](image-url)
**height**

```c
int height()
```

- Reports the height of `this`.
- Ensures:
  
  ```latex
  \text{height} = ht(\text{this})
  ```
addSubtree

```cpp
void addSubtree(int pos, Tree<T> st)
```

- Adds `st` at position `pos` in the subtrees of `this`; the declaration notwithstanding, the `dynamic` type of `st` must be the same as the `dynamic` type of `this`.
- Updates: `this`
- Clears: `st`
- Requires:
  ```
  this /= empty_tree and
  0 <= pos and
  pos <= [number of children of this]
  ```
- Ensures:
  ```
  this = [#this with subtree #st inserted at position pos]
  ```
removeSubtree

Tree<T> removeSubtree(int pos)

- Removes and returns the subtree at position \( pos \) of \( this \).
- Updates: \( this \)
- Requires:
  
  \[
  this \neq empty_tree \quad \text{and} \quad \\
  0 \leq pos \quad \text{and} \quad \\
  pos < \text{[number of children of } this]\]

- Ensures:
  
  \[
  this = [#this \text{ with subtree at position} \\
  pos \text{ removed and returned as result}]\]
numberOfSubtrees

```c
int numberOfSubtrees()
```

- Reports the number of subtrees of the root of `this`.
- Requires:
  ```c
  this /= empty_tree
  ```
- Ensures:
  ```c
  there exists root: T, 
  children: string of tree of T 
  (this = compose(root, children) and 
  numberOfSubtrees = |children|)
  ```
Resources

• OSU CSE Components API: Tree
  – http://cse.osu.edu/software/common/doc/