Immutable BST

Static methods:
- emptyTree : java.util.Comparator<String> -> BST
- node : java.util.Comparator<String> x String x BST x BST -> BST

Dynamic methods:
- size : -> int
- height : -> int
- isEmpty : -> boolean
- insert : String -> BST
- contains : String -> boolean

BST.emptyTree(c).size() = 0
BST.node(c, s, t1, t2).size() = 1 + t1.size() + t2.size()

BST.emptyTree(c).height() = 0
BST.node(c, s, t1, t2).height() = 1 + max(t1.height(), t2.height())

BST.emptyTree(c).isEmpty() = true
BST.node(c, s, t1, t2).isEmpty() = false

BST.emptyTree(c).insert(s1) = BST.node(c, s1, BST.emptyTree(c), BST.emptyTree(c))
BST.node(c, s, t1, t2).insert(s1) = BST.node(c, s, t1, t2) if c.compare(s, s1) == 0
BST.node(c, s, t1, t2).insert(s1) = BST.node(c, s, t1.insert(s1), t2) if c.compare(s, s1) > 0
BST.node(c, s, t1, t2).insert(s1) = BST.node(c, s, t1, t2.insert(s1)) if c.compare(s, s1) < 0

BST.emptyTree(c).contains(s1) = false
BST.node(c, s, t1, t2).contains(s1) = true if c.compare(s, s1) == 0
BST.node(c, s, t1, t2).contains(s1) = t1.contains(s1) if c.compare(s, s1) > 0
BST.node(c, s, t1, t2).contains(s1) = t2.contains(s1) if c.compare(s, s1) < 0
Red-Black Trees (RBT)
Red-Black Trees

Red-black trees are binary search trees in which each node has a color (either red or black) and the following balancing invariants are preserved by all operations:

1. No red node has a red child.

2. Every path from the root to an empty tree/node contains the same number of black nodes.
Benefits of RBT

• Balanced BST
• Efficiency
Theorem

In a red-black tree, the longest possible path from the root to an empty tree is no more than twice the length as the shortest possible path from the root to an empty tree.
Lemma

A red-black tree with $n$ internal nodes has height at most $2 \times \lg(n + 1)$. 
Functional Red-Black Trees

Okasaki
Functional Red-Black Trees

Okasaki

data Color = R | B

data Tree elt
    = E | T Color (Tree elt) elt (Tree elt)
Binary Search Tree (BST)

• t is empty
• t is a node
  - a label
  - the left subtree of t is a BST,
  - the right subtree of t is a BST,
  - every label within the left subtree of t is less than the label of t,
  - every label within the right subtree of t is greater than the label of t
Binary Search Tree (BST)

- t is empty
- t is a node
  - a label
  - the left subtree of t is a BST,
  - the right subtree of t is a BST,
  - every label within the left subtree of t is less than the label of t,
  - every label within the right subtree of t is greater than the label of t

data Tree elt = E | Tree elt elt (Tree elt)
Binary Search Tree (BST)

data Tree elt
  = E | T Color (Tree elt) elt (Tree elt)

BST.emptyTree(c)  BST.node(c, s, t1, t2)
member/contains

member \( x \in E = \text{False} \)
member \( x \in (T \_ a y b) \mid x < y = \text{member} \ x \ a \)
\mid x == y = True
\mid x > y = \text{member} \ x \ b

BST.emptyTree(c).contains(s1) = false
BST.node(c, s, t1, t2).contains(s1)
\quad = true \quad \text{if c.compare(s, s1)} == 0
BST.node(c, s, t1, t2).contains(s1)
\quad = t1.contains(s1) \quad \text{if c.compare(s, s1)} > 0
BST.node(c, s, t1, t2).contains(s1)
\quad = t2.contains(s1) \quad \text{if c.compare(s, s1)} < 0
Insertions
Okasaki

```haskell
insert :: Ord elt => elt -> Set elt ->
    Set elt insert x s = makeBlack (ins s)
where ins E = T R E x E
    ins(T color a y b) | x < y  = balance color (ins a) y b
                        | x == y = T color a y b
                        | x > y  = balance color a y (ins b)
    makeBlack (T _ a y b) = T B a y b
```
Insertions

\[
\begin{align*}
\text{insert} & \::= \text{Ord elt} \Rightarrow \text{elt} \Rightarrow \text{Set elt} \Rightarrow \\
\text{Set elt insert x s} & = \text{makeBlack (ins s)} \\
\text{where ins E} & = \text{T R E x E} \\
\text{ins(T color a y b)} | x < y & = \text{balance color (ins a) y b} \\
| x == y & = \text{T color a y b} \\
| x > y & = \text{balance color a y (ins b)} \\
\text{makeBlack (T _ a y b)} & = \text{T B a y b}
\end{align*}
\]

\[
\begin{align*}
\text{BST.emptyTree(c).insert(s1)} & = \text{BST.node(c, s1, BST.emptyTree(c), BST.emptyTree(c))} \\
\text{BST.node(c, s, t1, t2).insert(s1)} & = \text{BST.node(c, s, t1, t2)} & \text{if c.compare(s, s1) == 0} \\
\text{BST.node(c, s, t1, t2).insert(s1)} & = \text{BST.node(c, s, t1.insert(s1), t2)} & \text{if c.compare(s, s1) > 0} \\
\text{BST.node(c, s, t1, t2).insert(s1)} & = \text{BST.node(c, s, t1, t2.insert(s1))} & \text{if c.compare(s, s1) < 0}
\end{align*}
\]
Balance
Okasaki

\[
\text{balance } B\ (T\ R\ (T\ R\ a\ x\ b)\ y\ c)\ z\ d = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)
\]

\[
\text{balance } B\ (T\ R\ a\ x\ (T\ R\ b\ y\ c))\ z\ d = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)
\]

\[
\text{balance } B\ a\ x\ (T\ R\ (T\ R\ b\ y\ c)\ z\ d) = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)
\]

\[
\text{balance } B\ a\ x\ (T\ R\ b\ y\ (T\ R\ c\ z\ d)) = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)
\]

\[
\text{balance color } a\ x\ b = T\ \text{color } a\ x\ b
\]
Figure from Okasaki
balance $B \ (T \ R \ (T \ R \ a \ x \ b) \ y \ c) \ z \ d$
$\quad = T \ R \ (T \ B \ a \ x \ b) \ y \ (T \ B \ c \ z \ d)$

balance $B \ (T \ R \ a \ x \ (T \ R \ b \ y \ c)) \ z \ d$
$\quad = T \ R \ (T \ B \ a \ x \ b) \ y \ (T \ B \ c \ z \ d)$

balance $B \ a \ x \ (T \ R \ (T \ R \ b \ y \ c) \ z \ d)$
$\quad = T \ R \ (T \ B \ a \ x \ b) \ y \ (T \ B \ c \ z \ d)$

balance $B \ a \ x \ (T \ R \ b \ y \ (T \ R \ c \ z \ d))$
$\quad = T \ R \ (T \ B \ a \ x \ b) \ y \ (T \ B \ c \ z \ d)$

balance color $a \ x \ b = T \ color \ a \ x \ b$

from Okasaki
balance \( B \left( TR \left( TR a x b \right) y c \right) z d \)
\[
= TR \left( TB a x b \right) y \left( TB c z d \right)
\]

balance \( B \left( TR a x \left( TR b y c \right) \right) z d \)
\[
= TR \left( TB a x b \right) y \left( TB c z d \right)
\]

balance \( a x \left( TR \left( TR b y c \right) z d \right) \)
\[
= TR \left( TB a x b \right) y \left( TB c z d \right)
\]

balance \( a x \left( TR b y \left( TR c z d \right) \right) \)
\[
= TR \left( TB a x b \right) y \left( TB c z d \right)
\]

balance color \( a x b = T \) color \( a x b \)

from Okasaki
balance B (T R (T R a x b) y c) z d
= T R (T B a x b) y (T B c z d)

from Okasaki
balance \( B \ (T \ R \ (T \ R \ a \ x \ b) \ y \ c) \ z \ d \)

= \( T \ R \ (T \ B \ a \ x \ b) \ y \ (T \ B \ c \ z \ d) \)

\( a \ x \ b \): left left
\( y \): left data
\( c \): left right
\( z \): data
\( d \): right

from Okasaki
if (isBlack() &&
  !(left.isEmpty()) &&
  !(((Node) left).left.isEmpty()) &&
  (left.color == RED) &&
  ( ((Node) left).left.color == RED))
if (isBlack() &&
   !(left.isEmpty()) &&
   !(((Node) left).left.isEmpty()) &&
   (left.color == RED) &&
   (((Node) left).left.color == RED))
Law of Demeter
Examples
Translating into Java
My Implementations

StringByLex on lexicographically_ordered.txt
build operation (Worst Case)
My Implementations

StringByLex on lexicographically_ordered.txt contains operation (Worst Case)
Other Approaches to Red-Black Trees
Fig. 2. Alternative balancing transformations. Subtrees a–d all have black roots unless otherwise indicated.

Figure from Okasaki
-- color flips
balance B (T R a@(T R _ _ _) x b) y (T R c z d)
  || B (T R a x b@(T R _ _ _)) y (T R c z d)
  || B (T R a x b) y (T R c@(T R _ _ _) z d)
  || B (T R a x b) y (T R c z d@(T R _ _ _)) = T R (T B a x b) y (T B c z d).

-- single rotations
balance B (T R a@(T R _ _ _) x b) y c = T B a x (T R b y c)
balance a x (T R b y c@(T R _ _ _)) = T B (T R a x b) y c

-- double rotations
balance B (T R a x (T R b y c)) z d
  || B a x (T R (T R b y c) z d) = T B (T R a x b) y (T R c z d)

-- no balancing necessary
balance color a x b = T color a x b

Fig. 3. Alternative implementation of \texttt{balance}.
Mutable Binary Search Trees

[CLR01]

To insert a new value $v$ into a binary search tree $T$, we use the procedure $\text{TREE-INSERT}$. The procedure takes a node $z$ for which $z.key = v$, $z.left = \text{NIL}$, and $z.right = \text{NIL}$. It modifies $T$ and some of the attributes of $z$ in such a way that it inserts $z$ into an appropriate position in the tree.

```plaintext
\text{TREE-INSERT}(T, z)
1 \text{\textbf{y = NIL,}}
2 x = T.root
3 \textbf{while } x \neq \text{NIL} \textbf{\{while } T \text{ is not empty} \textbf{\}}
4 \text{\textbf{y = x,}}
5 \textbf{if } z.key < x.key
6 \text{\textbf{x = x.left,}}
7 \textbf{else } x = x.right
8 z.p = \text{y}
9 \textbf{if } y \neq \text{NIL}
10 \text{\textbf{T.root = z \textbf{\{with empty tree}\}}}
11 \textbf{elseif } z.key < y.key
12 \text{\textbf{y.left = z,}}
13 \textbf{else } y.right = z
```
Red-Black Tree

[CLR01]

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
Rotation

[CLR01]

```
LEFT-ROTATE(T, x)
1  y = x.right  // set y
2  x.right = y.left  // turn y’s left subtree into x’s right subtree
3  if y.left ≠ T.nil
4     y.left.p = x
5  y.p = x.p  // link x’s parent to y
6  if x.p == T.nil
7     T.root = y
8  elseif x == x.p.left
9     x.p.left = y
10  else x.p.right = y
11  y.left = x  // put x on y’s left
12  x.p = y
```
Insert into RBT

We can insert a node into an $n$-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the Tree-INSERT procedure (Section 12.3) to insert node $z$ into the tree $T$ as if it were an ordinary binary search tree, and then we color $z$ red. (Exercise 13.3-1 asks you to explain why we choose to make node $z$ red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT($T, z$) inserts node $z$, whose key is assumed to have already been filled in, into the red-black tree $T$.

```
RB-INSERT($T, z$)
1  $y = T.nil$
2  $x = T.root$
3  while $x \neq T.nil$
4      $y = x$
5          if $z.key < x.key$
6              $x = x.left$
7          else $x = x.right$
8  $z.p = y$
9  if $y == T.nil$
10     $T.root = z$
11  elseif $z.key < y.key$
12     $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$
15  $z.right = T.nil$
16  $z.color = RED$
17  RB-INSERT-FIXUP($T, z$)
```

The procedures Tree-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in Tree-INSERT are replaced by $T.nil$. Second, we set $z.left$ and $z.right$ to $T.nil$ in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color $z$ red in line 16. Fourth, because coloring $z$ red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP($T, z$) in line 17 of RB-INSERT to restore the red-black properties.
When are rotations needed?

[CLR01]

```plaintext
RB-INSERT-FIXUP(T, z)
1   while z.p.color == RED
2      if z.p == z.p.p.left
3         y = z.p.p.right
4         if y.color == RED
5            z.p.color = BLACK  // case 1
6            y.color = BLACK  // case 1
7            z.p.p.color = RED // case 1
8            z = z.p.p         // case 1
9      else if z == z.p.right
10         z = z.p           // case 2
11         LEFT-ROTATE(T, z) // case 2
12         z.p.color = BLACK // case 3
13         z.p.p.color = RED // case 3
14         RIGHT-ROTATE(T, z.p.p) // case 3
15 else (same as then clause
16         with “right” and “left” exchanged)
17   T.root.color = BLACK
```
Assignment 6