Balanced Trees
Four height-balanced trees:

- Red-black binary trees
  Faster than AVL for insertion and removal

- Adelsen-Velskii Landis (AVL) binary trees
  Faster than red-black for lookup

- B-trees
  n-way balanced tree

- Nguyen-Wong B-trees
  n-way balanced tree
Properties of a red-black tree

1. Every node is either red or black.
2. The root is black.
3. Every leaf (T.nil) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
Demo RBTree
1. Every node is either red or black.
2. The root is black.
3. Every leaf (T.nil) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:

```
26
17
41
30
38
47
50
T.nil
```

<table>
<thead>
<tr>
<th>h</th>
<th>bh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

[Nodes with bold outline indicate black nodes. Don't add heights and black-heights yet. We won't bother with drawing T:nil any more.]

Height of a red-black tree:

- Height of a node is the number of edges in a longest path to a leaf.
- Black-height of a node x: bh(x) is the number of black nodes (including T:nil) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

Claim: Any node with height h has black-height \( h = 2 \).

Proof: By property 4, \( h = 2 \) nodes on the path from the node to a leaf are red. Hence \( h = 2 \) are black. (claim)

Claim: The subtree rooted at any node x contains \( 2^{bh(x)} \) internal nodes.

Cormen Figure 13.1
**Black-height** of a node $x$: $bh(x)$ is the number of black nodes (including $T.nil$) on the path from $x$ to leaf, not counting $x$. By property 5, black-height is well defined.
Lemma 1

No path from the root to a leaf is more than twice as long as any other. That is, $bh(x) \geq \frac{h}{2}$ where $h$ is the height of the tree rooted at $x$.

Example: $bh(x) = 4$
Lemma 2

The number of internal nodes of a tree rooted at $x$ is $\geq 2^{bh(x)} - 1$.

Proof by mathematical induction on height of tree.
Base case, height = 0

\[
\#\text{internal nodes} \geq 2^{bh(x)} - 1
\]

= <base case, root x is NIL, no internal nodes>

\[
0 \geq 2^{bh(x)} - 1
\]

= <bh(x) = 0>

\[
0 \geq 2^0 - 1
\]

= <math>

true //
Induction case

Must prove $\#\text{internal nodes} \geq 2^{bh(x)} - 1$ for height $h$ tree assuming $\#\text{internal nodes} \geq 2^{bh(x)} - 1$ for height $h - 1$ tree as the inductive hypothesis.

Let the tree rooted at $x$ have $bh(x) = b$.
If child is red then $bh($child of $x) = b$.
If child is black then $bh($child of $x) = b - 1$. 
#internal nodes for height \( h \) tree

\[
\begin{align*}
\text{=} & \quad \langle \text{x is an internal node} \rangle \\
& 1 + \#\text{internal nodes of left} + \#\text{internal nodes of right} \\
\ge & \quad \langle \text{math} \rangle \\
& 1 + 2 \cdot (\#\text{internal nodes of a black child}) \\
\ge & \quad \langle \text{inductive hypothesis} \rangle \\
& 1 + 2 \cdot (2^{bh(x)} - 1 - 1) \\
= & \quad \langle \text{math} \rangle \\
& 2^{bh(x)} - 1 //
\end{align*}
\]
Theorem

A red-black tree with n internal nodes has height
\[ h \leq 2 \lg(n + 1) \]
\[ n \geq <\text{Lemma 2}> \]
\[ 2^{bh(x)} - 1 \]
\[ \geq <\text{Lemma 1}> \]
\[ 2^{h/2} - 1 \]
\[ n \geq 2^{h/2} - 1 \]
\[ = 2^{h/2} \leq n + 1 \]
\[ = \text{Take } \log \text{ of both sides} \]
\[ h/2 \leq \log(n + 1) \]
\[ = \text{math} \]
\[ h \leq 2 \log(n + 1) \]

Note: For AVL tree, \( h \leq 1.44 \log(n + 2) - 1 \)
The Singleton Design Pattern

In the RBTree project, apply the singleton pattern to the leaves. There is only one leaf.
Cormen Figure 13.1
Advantages of the singleton pattern:

* Allows an empty child to have a color.

* Saves storage.

* Allows a node to be compared to the NIL node.
enum ColorType {
    eRED, eBLACK
};

template<class T>
class Node {

private:
    ColorType _color;
    T _key;
    Node * _left;
    Node * _parent;
    Node * _right;
    static Node * _nilT;  // Singleton.
template<class T> class RBTree {
private:
    Node<T> * _root;

    void leftRotate(Node<T> * x);
    // Post: x is rotated left.

    void rightRotate(Node<T> * x);
    // Post: x is rotated right.

    void insertFixup(Node<T> * z);
    // Pre: z is inserted in order as a leaf without regard to balance.
    // Post: This red-black tree is balanced.

public:
    RBTree(); // Constructor
    // Post: This red-black tree is initialized to be empty.

    ~RBTree(); // Destructor
    // Post: This red-black tree is deallocated.

    void insert(T val);
    // Post: val is stored in this red-black tree in order.

    void toStream(ostream & os) const;
    // Post: A string representation of this tree is sent to os.
};
Cormen Figure 13.1
\text{LEFT-ROTATE}(T, x)$

Cormen Figure 13.3
template<class T> 
void RBTree<T>::leftRotate(Node<T> *x) { 
    Node<T>* y = x->_right; 
    // Step 1, Exercise for the student. 
   if (y->_left != Node<T>::_nilT) { 
        // Step 2, Exercise for the student. 
    } 
    // Step 3, Exercise for the student. 
    if (x->_parent == Node<T>::_nilT) { 
        _root = y;
    } else if (x == x->_parent->_left) { 
        // Step 4a, Exercise for the student. 
    } else { 
        // Step 4b, Exercise for the student. 
    } 
    // Step 5, Exercise for the student. 
    // Step 6, Exercise for the student. 
}
leftRotate(x)

(a) Initial state before rotation.

(b) The state after steps 1, 2, 3, and 4b.
leftRotate(x)

(b) The state after steps 1, 2, 3, and 4b.

(c) The state of part (b) with the nodes rearranged.
leftRotate(x)

(a) Initial state before rotation.

(b) The state after steps 1, 2, 3, and 4b.

(c) The state of part (b) with the nodes rearranged.

(d) The state after steps 5 and 6.

(e) The state of part (b) with the nodes rearranged.
rightRotate(x)

The code is identical to the code for leftRotate(x) with “left” everywhere exchanged with “right”.
Insertion into a red-black tree

* Insert a new node in place of a leaf.
* Color it red.
* Which red-black property is violated?

1. Every node is either red or black.
2. The root is black.
3. Every leaf \((T.nil)\) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
Property 4 only!
(unless insertion is into the empty tree)

Furthermore, only if the parent of the node inserted is red.

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
template<class T>
RBTree<T>::RBTree() {
    T junk;
    Node<T>::_nilT = new Node<T>(junk);  // Allocate singleton.
    Node<T>::_nilT->_color = eBLACK;
    _root = Node<T>::_nilT;  // The empty tree.
}

template<class T>
Node<T>::Node(T val) :
    _color(eRED),
    _key(val),
    _left(Node<T>::_nilT),
    _parent(Node<T>::_nilT),
    _right(Node<T>::_nilT) { }
template<class T>
void RBTree<T>::insert(T val) {
    Node<T> *z = new Node<T>(val);
    Node<T> *y = Node<T>::_nilT;
    Node<T> *x = _root;
    while (x != Node<T>::_nilT) {
        y = x;
        if (z->_key < x->_key) {
            x = x->_left;
        } else {
            x = x->_right;
        }
    }
}

At this point, \textit{y} is the leaf that will become the parent of \textit{z}.
z->_parent = y;
if (y == Node<T>::_nilT) {
    _root = z;
} else if (z->_key < y->_key) {
    y->_left = z;
} else {
    y->_right = z;
} insertFixup(z);

Make y the parent of z and get fixed up.
template<class T>
void RBTree<T>::insertFixup(Node<T> *z) {
    cerr << "insertFixup: Exercise for the student." << endl;
    throw -1;
    Node<T>* y;
    while (z->_parent->_color == eRED) {
        if (z->_parent == z->_parent->_parent->_left) {
            y = z->_parent->_parent->_right;
            if (y->_color == eRED) {
                // Case 1, Exercise for the student.
                // Four assignment statements.
            }
        } else {
            if (z == z->_parent->_right) {
                // Case 2, Exercise for the student.
                // One assignment statement, one rotate.
            }
        }
    // Case 3, Exercise for the student.
    // Two assignment statements, one rotate.
    }
}
else {
    y = z->_parent->_parent->_left;
    if (y->_color == eRED) {
        // Case 4, Exercise for the student.
        // Four assignment statements.
    }
    else {
        if (z == z->_parent->_left) {
            // Case 5, Exercise for the student.
            // One assignment statement, one rotate.
        }
        // Case 6, Exercise for the student.
        // Two assignment statements, one rotate.
    }
}
_root->_color = eBLACK;

The code for cases 4, 5, and 6 is identical to the code for cases 1, 2, and 3 with “left” everywhere exchanged with “right”.
Loop condition

z points to a red node with a red parent being the only violation

⇒ The grandparent of z is black.
Possible Case executions

* Case 1 only — the loop repeats

* Case 2 followed by Case 3 — the loop terminates

* Case 3 only — the loop terminates
Case conditions

* Case 1: z’s uncle y is red

* Case 2: z’s uncle y is black, and z is a right child

* Case 3: z’s uncle y is black, and z is a left child
Case 1: z’s uncle y is red
Change colors to make z’s uncle y black, and move z up two generations, keeping z red. Keep looping.
Case 2: z’s uncle y is black and z is a right child. Make z a left child in preparation for Case 3. If z were already a left child, this step would be skipped.
Case 3: z’s uncle y is black and z is a left child
Change the colors of z’s parent and grandparent, and rotate
at z’s grandparent. The tree is now a legal RB tree.

Cormen Figure 13.4
Case 1: z’s uncle y is red
Change colors to make z’s uncle y black, and move z up two generations, keeping z red. Keep looping.

Cormen Figure 13.5
Case 2: z’s uncle y is black and z is a right child

Make z a left child in preparation for Case3.

If z were already a left child, this step would be skipped.

Cormen Figure 13.6
Case 3: z’s uncle y is black and z is a left child
Change the colors of z’s parent and grandparent, and rotate
at z’s grandparent. The tree is now a legal RB tree.
// ========= Destructors =========
template<class T>
RBTree<T>::~RBTree() {
    if (_root != Node<T>::_nilT) {
        delete _root;
        _root = NULL;
    }
    delete Node<T>::_nilT; // Deallocate singleton
}

template<class T>
Node<T>::~Node() {
    if (_left != Node<T>::_nilT) {
        delete _left; // Delete left subtree.
        _left = NULL;
    }
    if (_right != Node<T>::_nilT) {
        delete _right; // Delete right subtree.
        _right = NULL;
    }
}