Weight-balanced Binary Search Trees

Weight-balanced BSTs are one way to achieve $\Theta(\log(n))$ tree height.

A weight-balanced BST:

- is a binary search tree
- at every node $v$:

$$\frac{1}{3} \leq \frac{\text{size}(v.left) + 1}{\text{size}(v.right) + 1} \leq 3$$

Equivalently:

$$\text{size}(v.left) + 1 \leq (\text{size}(v.right) + 1) \times 3$$

$$(\text{size}(v.left) + 1) \times 3 \geq \text{size}(v.right) + 1$$

(“weight” means size+1)
Example & Counterexample

balanced

unbalanced
Rebalancing: Overview

If a node $v$ is unbalanced, the right or the left is too big:

if $(\text{size}(v.\text{left}) + 1) \times 3 < \text{size}(v.\text{right}) + 1$
  (the right is too big)
  re-balance

else if $\text{size}(v.\text{left}) + 1 > (\text{size}(v.\text{right}) + 1) \times 3$
  (the left is too big)
  re-balance

else
  nothing to fix
Rebalancing: Overview

If a node \( v \) is unbalanced, the right or the left is too big:

\[
\text{if } (\text{size}(v\.left) + 1) \times 3 < \text{size}(v\.right) + 1 \\
\text{(the right is too big)} \\
\text{re-balance} \\
\text{two further subcases here}
\]

\[
\text{else if } \text{size}(v\.left) + 1 > (\text{size}(v\.right) + 1) \times 3 \\
\text{(the left is too big)} \\
\text{re-balance} \\
\text{two further subcases here}
\]

\[
\text{else} \\
\text{nothing to fix}
\]
Rebalancing: Overview

If a node $v$ is unbalanced, the right or the left is too big:

if $(size(v.left) + 1) \times 3 < size(v.right) + 1$
  (the right is too big)
  re-balance
  two further subcases here
else if $size(v.left) + 1 > (size(v.right) + 1) \times 3$
  (the left is too big)
  re-balance
  two further subcases here
else
  nothing to fix

We do this check and fix for each node from bottom to top.
⇒ When fixing $v$, assume descendents are balanced.
This goes by the name *single-rotation*.
Single-Rotation on The Right, Generally

If \((\text{size}(v\.left) + 1) \times 3 < \text{size}(v\.right) + 1\):

Let \(x = v\.right\)

If \(\text{size}(x\.left) + 1 < (\text{size}(x\.right) + 1) \times 2\):

\[
\begin{array}{c}
\text{v} \\
R \\
S \\
\end{array}
\xrightarrow{\text{ }}
\begin{array}{c}
x \\
R \\
S \\
\end{array}
\xrightarrow{\text{ }}
\begin{array}{c}
v \\
T \\
\end{array}
\]

Idea: \(T\) is big enough to be on par with \(R\), \(v\), and \(S\) combined.
Rebalance: Subcase 2 of 2

This goes by the name *double-rotation*. 
Double Rotation on The Right, Generally

\[
\text{If } (\text{size}(v.\text{left}) + 1) \times 3 < \text{size}(v.\text{right}) + 1: \\
\text{Let } x = v.\text{right} \\
\text{If } \text{size}(x.\text{left}) + 1 \geq (\text{size}(x.\text{right}) + 1) \times 2: \\
\text{Let } w = x.\text{left}:
\]

\[
\begin{array}{ccc}
\text{v} & \rightarrow & \text{w} \\
\downarrow & & \downarrow \\
R & & T \\
\downarrow & & \downarrow \\
S_1 & & S_2 \\
\end{array} \\
\begin{array}{ccc}
\text{w} & \rightarrow & \text{v} \\
\downarrow & & \downarrow \\
R & & S_1 \\
\downarrow & & \downarrow \\
S_2 & & T \\
\end{array}
\]

Idea: $S$ in the middle is too big. Split.
Single-Rotation on The Left, Generally

If \( \text{size}(v.left) + 1 > (\text{size}(v.right) + 1) \times 3 \):

Let \( x = v.left \)

If \( (\text{size}(x.left) + 1) \times 2 > \text{size}(x.right) + 1 \):
Double Rotation on The Left, Generally

If \(\text{size}(v.left) + 1 > (\text{size}(v.right) + 1) \times 3\):

Let \(x = v.left\)

If \((\text{size}(x.left) + 1) \times 2 \leq \text{size}(x.right) + 1\):

Let \(w = x.right\):
Rebalancing: Summary

For each node $v$ from bottom to top:

if $(\text{size}(v.\text{left}) + 1) \times 3 < \text{size}(v.\text{right}) + 1$
  let $x = v.\text{right}$
  if $\text{size}(x.\text{left}) + 1 < (\text{size}(x.\text{right}) + 1) \times 2$
    single-rotation on the right
  else
    double-rotation on the right
else if $\text{size}(v.\text{left}) + 1 > (\text{size}(v.\text{right}) + 1) \times 3$
  let $x = v.\text{left}$
  if $(\text{size}(x.\text{left}) + 1) \times 2 > \text{size}(x.\text{right}) + 1$
    single-rotation on the left
  else
    double-rotation on the left
else
  no rotation
Size Query And Update

Remember sizes at all nodes so querying is trivial:

class Node<K> {
    public K key;
    public Node<K> left, right, parent;
    public int num;
...
    public static int size(Node<K> u) {
        return (u == null ? 0 : u.num);
    }
}

To set or update, do it bottom-up, so we can simply:

v.num = 1 + size(v.left) + size(v.right);
Summary of WBT Insertion

Later we will see why WBT height is in $\Theta(\log(n))$.

With that in mind, WBT insertion:

1. find which node to become parent of new node [$\Theta(\log(n))$ time]
2. put new node there [\(\Theta(1)\) time]
3. from that parent to root (bottom-up): check and fix balance, update size [$\Theta(\log(n))$ nodes, $\Theta(1)$ time per node]
Delete: Easy Case

Delete 32, or 48, or 62, or 88.

If the node has no children, just unlink from parent.
(Then update sizes of ancestors, rebalance...)
Delete 17. (Note that 32 is a good replacement.)

If the node has at most one child, just link parent to that child. (Then update sizes of ancestors, rebalance...)

This generalizes the easy case.
Delete: Slightly Harder Case, Generally

Prune \(w\). \(T_0\) may be empty. \(p\) and ancestors need size updates and rebalancing.

There are two more mirror images.
Delete: Hard Case

Delete 5. Call the node $w$. Both children non-empty.

Find successor: Go right once, go left all the way, call it $x$. Replace $w.key$ by $w.key$. $x$’s parent adopts $x$’s right child $T$. 
Delete: Hard Case, Degenerate

Delete 5. Call the node $w$. Both children non-empty.

Go right. Go left all the way—can’t! That’s already $x$. $x$’s parent is $w$.

Still, replace $w.key$ by $x.key$. $w$ also adopts $x$’s right child $T$. 
Summary of WBT Deletion

Next we will see why WBT height is $\Theta(\log(n))$.

1. find which node has the key, call it $w$ [$\Theta(\log(n))$ time]
2. if at most one child, $w.parent$ adopts that child [$\Theta(1)$ time]
3. else:
   3.1 go to successor $x$ [$\Theta(\log(n))$ time]
   3.2 $w.key := x.key$ [$\Theta(1)$ time]
   3.3 $x.parent$ adopts $x.right$ [$\Theta(1)$ time]
4. from adopter to root (bottom-up): check and fix balance, update size [$\Theta(\log(n))$ time]
WBT Height

\[ \text{height}(T) \leq c \times \log(\text{size}(T) + 1) \text{ where } c = \frac{1}{\log(4/3)} \approx 2.4. \]

Induction on size:

- Basis: \( \text{height(\text{empty})} = 0; \ c \log(\text{size(\text{empty})} + 1) = 0. \)
- Induction step (sketch): W.l.o.g. assume the right subtree is higher than the left subtree.

\[
\text{size}(T.\text{right}) + 1 \leq (\text{size}(T) + 1) \times \frac{3}{4}
\]

\[
\text{height}(T) = 1 + \text{height}(T.\text{right})
\]

\[
\leq 1 + c \log(\text{size}(T.\text{right}) + 1)
\]

\[
\leq 1 + c \log((\text{size}(T) + 1) \times \frac{3}{4})
\]

\[
= 1 + c \log(\frac{3}{4}) + c \log((\text{size}(T) + 1)
\]

\[
= c \log((\text{size}(T) + 1)
\]
Dictionary

Stores a mapping (function) from finitely many keys to values.

Example: Store student names and marks. Find the mark of a student.

- $d$.insert($k$, $v$): $d$ now maps $k$ to $v$
- $d$.lookup($k$): return what $k$ is mapped to (throw an exception if none)
- $d$.delete($k$): forget what $k$ is mapped to (some versions also return what $k$ was mapped to)

Binary search trees are good for dictionaries too. Time costs are still $\Theta(\log(n))$. 
Binary Search Trees for Dictionaries

- Tree nodes: Add a field for the value.

```java
class Node<K,V> {
    public K key;
    public V value;
    public Node<K,V> left, right, parent;
    ...
}
```

- `d.insert(k, v)`: Insert new node into tree (and rebalance). Just remember to store the value in the new node!

- `d.delete(k)`: Find and remove node from tree (and rebalance).

- `d.lookup(k)`: Search for node in tree. Return the value in the node found. (Throw exception if no node has key k.)
Ranking

Rank of key $k$ in set $s$: how many keys are smaller than $k$.

Example: if $s$ stores these keys 42, 55, 61, then:

- $s\text{.rank}(42)$ returns 0
- $s\text{.rank}(55)$ returns 1
- $s\text{.rank}(61)$ returns 2
- $s\text{.rank}(k)$ throws exception if $k$ not found

Similarly for dictionaries.

How to extend WBT for this? All operations $\Theta(\log(n))$ time.
Ranking

Rank of key $k$ in set $s$: how many keys are smaller than $k$.

Example: if $s$ stores these keys 42, 55, 61, then:

- $s$.rank(42) returns 0
- $s$.rank(55) returns 1
- $s$.rank(61) returns 2
- $s$.rank($k$) throws exception if $k$ not found

Similarly for dictionaries.

How to extend WBT for this? All operations $\Theta(\log(n))$ time.

(Not good enough: at each node, store its rank.)

Cliff hanger! Come next time for a solution!