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 $v$:

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

Equivalently:

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

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

("weight" means size+1)

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
  two further subcases here
else if $\text{size}(v.\text{left}) + 1 > (\text{size}(v.\text{right}) + 1) \times 3$
  (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.

Example & Counterexample

Rebalance: Subcase 1 of 2

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

Let \(x = v.right\)

\[
\begin{tabular}{c}
R \\
\hline
S \\
\hline
T \\
\hline
\end{tabular}
\]

\[
\begin{tabular}{c}
R \\
\hline
S \\
\hline
T \\
\hline
\end{tabular}
\]

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

Double Rotation on The Right, Generally

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

Let \(x = v.right\)

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

Let \(w = x.left\):

\[
\begin{tabular}{c}
R \\
\hline
w \\
\hline
S_1 \\
\hline
S_2 \\
\hline
T \\
\hline
\end{tabular}
\]

\[
\begin{tabular}{c}
R \\
\hline
x \\
\hline
w \\
\hline
T \\
\hline
\end{tabular}
\]

\[
\begin{tabular}{c}
R \\
\hline
x \\
\hline
v \\
\hline
T \\
\hline
\end{tabular}
\]

\[
\begin{tabular}{c}
R \\
\hline
x \\
\hline
v \\
\hline
S_1 \\
\hline
S_2 \\
\hline
T \\
\hline
\end{tabular}
\]

This goes by the name double-rotation.

Rebalance: Subcase 2 of 2

Single-Rotation on The Left, Generally

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

Let \(x = v.right\)

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

Let \(x = v.right\)

\[
\begin{tabular}{c}
R \\
\hline
T \\
\hline
S \\
\hline
v \\
\hline
\end{tabular}
\]

\[
\begin{tabular}{c}
R \\
\hline
T \\
\hline
S \\
\hline
v \\
\hline
\end{tabular}
\]

Idea: \(S\) in the middle is too big. Split.
Double Rotation on The Left, Generally

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

Let $x = \text{v.left}$

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

Let $w = \text{x.right}$:

```
    v
   /|
  x  w
 /|
T  R
```

Rebalancing: Summary

For each node $v$ from bottom to top:

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

let $x = \text{v.right}$

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

single-rotation on the right

else

double-rotation on the right

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

let $x = \text{v.left}$

if $(\text{size}(\text{x.left}) + 1) \times 2 > \text{size}(\text{x.right}) + 1$

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:

```java
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: Slightly Harder Case

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. T₀ 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.

\[
\begin{array}{ccc}
\text{5} & \rightarrow & \text{80} \\
\text{S} & & \text{S} \\
T & & T
\end{array}
\]

Go right. Go left all the way—can't! That's already \( x \)'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 \( w \) \( [\Theta(\log(n)) \text{ time}] \)
2. if at most one child, \( w.parent \) adopts that child \([\Theta(1) \text{ time}] \)
3. else:
   3.1 go to successor \( x \) \( [\Theta(\log(n)) \text{ time}] \)
   3.2 \( w.key := x.key \) \( [\Theta(1) \text{ time}] \)
   3.3 \( x.parent \) adopts \( x.right \) \( [\Theta(1) \text{ time}] \)
4. from adopter to root (bottom-up): check and fix balance, update size \([\Theta(\log(n)) \text{ time}] \)

WBT Height

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

Induction on size:

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

\[
\begin{align*}
\text{balance} & \quad \text{size}(T.right) + 1 \leq (\text{size}(T) + 1) \times 3/4 \\
height(T) &= 1 + \text{height}(T.right) \\
&\leq 1 + c \lg(\text{size}(T.right) + 1) \\
&\leq 1 + c \lg((\text{size}(T) + 1) \times 3/4) \\
&= 1 + c \lg(3/4) + c \lg((\text{size}(T) + 1) \\
&= c \lg((\text{size}(T) + 1)
\end{align*}
\]

Dictionary

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

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

\begin{itemize}
\item \( d.insert(k, v) \): \( d \) now maps \( k \) to \( v \)
\item \( d.lookup(k) \): return what \( k \) is mapped to (throw an exception if none)
\item \( d.delete(k) \): forget what \( k \) is mapped to (some versions also return what \( k \) was mapped to)
\end{itemize}

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.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!