Introduction to Weight-Balanced BSTs

Albert Y. C. Lai

I assume that you already know the basic ideas of binary search trees, how to look up a key, how to insert a key, and how to delete a key.

Recall that if you stick to the basic insertion algorithm, there are ways to insert $n$ keys in pathological orders (e.g., ascending order), resulting in tree heights of $\Theta(n)$ as opposed to the ideal $\Theta(\lg n)$. Tree heights determine the time costs of all operations on binary search trees.

Weight-Balanced BSTs: Definition and Promise

One way to fix this is weight-balanced binary search trees. It means binary search trees satisfying an extra condition. First we define:

- The size of a subtree is the number of keys in the subtree.
- The size at a node is the size of the subtree rooted at that node. I emphasize that I carefully choose the preposition “at” rather than “of” to avoid a subtle ambiguity. I write size($u$) in formulas (if the node is $u$).
- size($null$) = 0 for the empty tree, represented by the null node.

The extra condition: For every node $v$, “$v$ is balanced”, meaning

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

Equivalently, we write the two inequalities

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

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

The main idea is that the left and the right have similar number of keys, i.e., the keys are distributed somewhat evenly. This forbids pathologically biased trees and ensures $\Theta(\lg n)$ tree heights. To see this, we prove:

For all natural $n$, the height of every weight-balanced BST of size $n$ is at most $c \lg(n + 1)$ where $c = 1/\lg(4/3) \approx 2.4$.

Proof:
• Base case: For a tree $T$ of size 0,

\[
\begin{align*}
\text{height}(T) &= 0 \\
c \log(n + 1) &= c \log(0 + 1) \\
&= 0
\end{align*}
\]

Therefore $\text{height}(T) \leq c \log(n + 1)$.

• Induction step: Let natural $n \geq 1$ be given. Let $T$ be a weight-balanced BST of size $n$.

Induction hypothesis: Suppose for all natural $k$ where $0 \leq k < n$, the height of every weight-balanced BST of size $k$ is at most $c \log(k + 1)$.

Define the abbreviations $n_l = \text{size}(T, \text{left})$ and $n_r = \text{size}(T, \text{right})$. So $n = n_l + n_r + 1$. We deduce:

\[
\begin{align*}
& n + 1 = n_l + n_r + 1 + 1 \\
& \geq (n_r + 1)/3 + n_r + 1 \quad \text{T is balanced} \\
& = (4n_r + 4)/3 \\
& = 4(n_r + 1)/3 \\
& n_r + 1 \leq (n + 1) \times 3/4 \\
\end{align*}
\]

Without loss of generality, assume $\text{height}(T, \text{left}) \leq \text{height}(T, \text{right})$, so $\text{height}(T) = 1 + \text{height}(T, \text{right})$. We calculate:

\[
\begin{align*}
\text{height}(T) &= 1 + \text{height}(T, \text{right}) \\
&\leq 1 + c \log(n_r + 1) \quad \text{I.H., } k = n_r \\
&\leq 1 + c \log((n + 1) \times 3/4) \quad \text{deduced above} \\
&= 1 + c \log(3/4) + c \log(n + 1) \\
&= 1 + \log(3/4) / \log(4/3) + c \log(n + 1) \\
&= 1 - \log(4/3) / \log(4/3) + c \log(n + 1) \\
&= c \log(n + 1) \\
&= c \log(\text{size}(T) + 1)
\end{align*}
\]

End proof.

The choice of the magic number 3 is an advanced topic mentioned in the final section.

**Insertion**

Suppose we already have a weight-balanced BST. Insertion begins as in basic BSTs: attempt a lookup, if it fails and stops at node $u$ (say), then add a new node as a new child of $u$. Now each ancestor of $u$ may become unbalanced.

To fix this, for each node $v$ from $u$ to the root, in bottom-up order, we will check the balancing condition; where it is broken, we will perform a “rotation” to restore balance.
(Exercise: we do not have to check \( u \) itself; it is still balanced after getting the new child node. Why?)

There are two cases \( v \) can be unbalanced, and they are mutually exclusive: either

\[
size(v.left) + 1 > (size(v.right) + 1) \times 3
\]

or

\[
(size(v.left) + 1) \times 3 < size(v.right) + 1
\]

I will discuss the second case, where intuitively the right subtree is "too big"; the first case is symmetric, where intuitively the left subtree is "too big".

\( v.right \) cannot be null (exercise: why?), so let \( x = v.right \), and further examine \( x \)'s two child subtrees. There are two subcases:

**Subcase 1 “single-rotation”**  This happens when

\[
size(x.left) + 1 < (size(x.right) + 1) \times 2
\]

Intuitively, \( x \)'s right subtree—\( T \) in the picture—is too big, so we re-arrange \( R \) and \( S \) to be on the same side in the hope that they together can counteract \( T \) on the other side. This is commonly known as a “single-rotation”:

![Diagram of single-rotation](image)

(Exercise: Assume that the tree is a binary search tree before the rotation. Prove that the tree after the rotation is still a binary search tree.)

We have to prove that after the rotation, the new \( x \) and the new \( v \) are balanced. This is left as an assignment question. (It is easier than the subcase below.)

**Subcase 2 “double-rotation”**  This happens when

\[
size(x.left) + 1 \geq (size(x.right) + 1) \times 2
\]

Intuitively, this time \( x \)'s left subtree is too big, so we split it up. \( x.left \) cannot be null (why?), so let \( w = x.left \), and perform a “double-rotation”:

![Diagram of double-rotation](image)
We have to prove that after the rotation, the new \( v \), \( w \), and \( x \) are balanced. First define these short-hands:

\[
\begin{align*}
    a &= \text{size}(R) \\
    b_1 &= \text{size}(S_1) \\
    b_2 &= \text{size}(S_2) \\
    c &= \text{size}(T)
\end{align*}
\]

In the proof, we will use these facts known before the rotation:

1. We are in the second case of unbalance and its second subcase:
   
   \[
   \begin{align*}
   (a + 1) \times 3 &< b_1 + b_2 + c + 3 & (1.\text{i}) \\
   b_1 + b_2 + 2 &\geq (c + 1) \times 2 & (1.\text{ii})
   \end{align*}
   \]

2. The insertion adds the new node somewhere under \( x \) (because the size at \( x \) becomes too big after the insertion). Therefore, the size at \( x \) before the insertion was

   \[
   b_1 + b_2 + c + 2 - 1 = b_1 + b_2 + c + 1
   \]

   And \( v \) was balanced before the insertion, so we have this inequality:

   \[
   (a + 1) \times 3 \geq b_1 + b_2 + c + 2
   \]  
   
   (We won’t need the other inequality.)

3. \( x \) is already balanced before the rotation (because we fix balancing bottom-up):

   \[
   b_1 + b_2 + 2 \leq (c + 1) \times 3
   \]  
   
   (We won’t need the other inequality.)

4. \( w \) is already balanced before the rotation (because we fix balancing bottom-up):

   \[
   \begin{align*}
   b_1 + 1 &\leq (b_2 + 1) \times 3 & (4.\text{i}) \\
   (b_1 + 1) \times 3 &\geq b_2 + 1 & (4.\text{ii})
   \end{align*}
   \]

We now prove:
• The new \( v \) is balanced:

Proof of \( a + 1 \leq (b_1 + 1) \times 3 \):

\[
(a + 1) < (b_1 + b_2 + c + 3)/3 \quad \text{by (1.i)}
\]
\[
\leq (b_1 + b_2 + (b_1 + b_2 + 2)/2 + 2)/3 \quad \text{by (1.ii)}
\]
\[
= ((3b_1 + 3b_2 + 6)/2)/3
\]
\[
= (b_1 + b_2 + 2)/2
\]
\[
= (b_1 + 1)/2 + (b_2 + 1)/2
\]
\[
\leq (b_1 + 1)/2 + (b_1 + 1) \times 3/2 \quad \text{by (4.ii)}
\]
\[
= (b_1 + 1) \times 2
\]
\[
\leq (b_1 + 1) \times 3
\]

Proof of \((a + 1) \times 3 \geq b_1 + 1\):

\[
(a + 1) \times 3 \geq b_1 + b_2 + c + 2 \quad \text{by (2)}
\]
\[
\geq b_1 + 1
\]

• The new \( x \) is balanced:

Proof of \( b_2 + 1 \leq (c + 1) \times 3 \):

\[
(c + 1) \times 3 \geq b_1 + b_2 + 2 \quad \text{by (3)}
\]
\[
\geq b_2 + 1
\]

Proof of \((b_2 + 1) \times 3 \geq c + 1\):

\[
c + 1 \leq (b_1 + b_2 + 2)/2 \quad \text{by (1.ii)}
\]
\[
\leq ((b_2 + 1) \times 3 + b_2 + 1)/2 \quad \text{by (4.i)}
\]
\[
= (4b_2 + 4)/2
\]
\[
= (b_2 + 1) \times 2
\]
\[
\leq (b_2 + 1) \times 3
\]

• The new \( w \) is balanced:

Proof of \( a + b_1 + 2 \leq (b_2 + c + 2) \times 3 \):

\[
a + b_1 + 2 < (b_1 + b_2 + c + 3)/3 + b_1 + 1 \quad \text{by (1.i)}
\]
\[
\leq ((c + 1) \times 3 + c + 1)/3 + b_1 + 1 \quad \text{by (3)}
\]
\[
= (c + 1) \times 4/3 + b_1 + 1
\]
\[
\leq (c + 1) \times 4/3 + (b_2 + 1) \times 3 \quad \text{by (4.i)}
\]
\[
\leq (c + 1) \times 3 + (b_2 + 1) \times 3
\]
\[
= (b_2 + c + 2) \times 3
\]

Proof of \((a + b_1 + 2) \times 3 \geq b_2 + c + 2\):

\[
b_2 + c + 2 = (b_1 + b_2 + c + 2) - b_1
\]
\[
\leq (a + 1) \times 3 - b_1 \quad \text{by (2)}
\]
\[
\leq (a + b_1 + 2) \times 3
\]
The choice of the magic number 2 is an advanced topic mentioned in the final section.

**Deletion**

Suppose we already have a weight-balanced BST. Deletion begins as in basic BSTs: perform a lookup, find the key at node \( v \) (say), then there are two cases:

- \( v \) has at most one child, so prune \( v \) and promote \( v \)'s child (if any), but now \( v \)'s ancestors may become unbalanced.
- \( v \) has two children, so replace its key by \( v \)'s successor \( w \) (say), prune \( w \) instead and promote \( w \)'s child, but now \( w \)'s ancestors may become unbalanced.

To restore balance, we check the balancing condition bottom-up and apply rotations, starting from the pruned node’s parent, ending at the root. We use the same case analysis and rotation scheme from the previous section.

The proofs that the rotations really restore balance are similar to those in the previous section, but there is a difference: this time it is the smaller subtree that had one more node before deleting, rather than the bigger subtree that had one fewer node before inserting.

**Tracking Sizes**

We extend each node with a field \( \text{num} \) to store the size at that node. Then querying \( \text{size}(u) \) takes constant time by just fetching \( u.\text{num} \).

This field needs updating when there is an insertion or deletion. But only the following nodes need this update:

- the nodes along the path from the root to the site of insert/delete
- for each node in the above, possibly one child node and one grandchild node (during a rotation)

So at most 3 \( \text{height}(T) \) nodes need this update. Each update takes constant time because:

We can update bottom-up, as we perform rebalancing rotations bottom-up. Then when updating a node \( v \), we can assume that \( v.\text{left} \) and \( v.\text{right} \) have their \( \text{num} \) fields already updated, so we simply set \( v.\text{num} \) to \( 1 + \text{size}(v.\text{left}) + \text{size}(v.\text{right}) \), which takes constant time too.

**Bibliography**

Weight-balanced BSTs were originally called bounded balance BSTs, coming from

The original magic numbers from that paper were $1 + \sqrt{2}$ instead of 3 in the balancing condition, and $\sqrt{2}$ instead of 2 when deciding whether to use a single-rotation or a double-rotation.

The name “weight-balanced BST” was used in Knuth’s *The Art of Computer Programming*, volume 3. Since then, it has become the preferred name. “Weight” refers to size+1.

The magic numbers must be chosen carefully to ensure that the rotations always restore balance. Impressively thorough analysis of valid magic numbers is reported in Yoichi Hirai and Kazuhiko Yamamoto, Balancing weight-balanced trees. *Journal of Functional Programming*, 21(3), 2011.

It turns out that the two magic numbers—the multiplier in the balancing condition, and the multiplier when deciding which rotation to use—must be chosen together (i.e., not independently chosen), and the region of valid pairs on the 2D plane is small and irregular. The pair $(1 + \sqrt{2}, \sqrt{2})$ by Nievergelt and Reingold is at the optimal corner, but comparisons involving irrational numbers is slow or annoying to code. The pair $(3, 2)$ is the only integer pair in the valid region. Rational pairs are more plentiful, such as $(2.5, 1.5)$. 