AVL Trees

Version of September 6, 2016
Binary Search Trees

Binary-search-tree property

For every node $x$:
- All keys in its left subtree are smaller than the key value in $x$.
- All keys in its right subtree are larger than the key value in $x$. 
Binary-search-tree property

For every node $x$
  - All keys in its left subtree are smaller than the key value in $x$
  - All keys in its right subtree are larger than the key value in $x$
**Binary-search-tree property**

For every node $x$
- All keys in its left subtree are smaller than the key value in $x$
- All keys in its right subtree are larger than the key value in $x$
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf.
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf.

- **Node height**
  
  $= \max(\text{children height}) + 1$
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- **Node height**
  \[ \text{node height} = \text{max(children height)} + 1 \]
- **Leaves**: height = 0
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[= \text{max}(\text{children height}) + 1\]
- Leaves: \(\text{height} = 0\)
- Tree height \(=\) root height
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[= \max(\text{children height}) + 1\]
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = \(-1\)
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[ = \max(\text{children height}) + 1 \]
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

- Tree operations typically take \( O(\text{height}) \) time
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[ = \max(\text{children height}) + 1 \]
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1
- Tree operations typically take \( O(\text{height}) \) time
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[ = \max(\text{children height}) + 1 \]
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = \(-1\)

- Tree operations typically take \(O(\text{height})\) time
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf.

- **Node height**
  \[ \text{height} = \max(\text{children height}) + 1 \]
- **Leaves**: height = 0
- **Tree height** = root height
- **Empty tree**: height = $-1$

- Tree operations typically take $O(\text{height})$ time.
The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  \[ = \max(\text{children height}) + 1 \]
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

- Tree operations typically take \(O(\text{height})\) time

---

**Question**

Let \(n\) be the size of a binary search tree. How can we keep its height \(O(\log n)\) under insertion and deletion?
An AVL [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

- for every node in the tree, heights of its left and right subtrees differ by at most 1.
An **AVL** [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

- for every node in the tree, heights of its left and right subtrees differ by at most 1.

![AVL Tree Diagram](image)

- The *balance factor* of a node is the height of its right subtree minus the height of its left subtree.
An AVL [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

- for every node in the tree, heights of its left and right subtrees differ by at most 1.

- The balance factor of a node is the height of its right subtree minus the height of its left subtree.

- A node with balance factor 1, 0 or −1 is considered balanced.
Let $n_h$ be the minimum number of nodes in an AVL tree of height $h$.

\[ n_0 = 1 \]
Let $n_h$ be the minimum number of nodes in an AVL tree of height $h$.

\[ n_0 = 1 \quad n_1 = 2 \]
Let \( n_h \) be the minimum number of nodes in an AVL tree of height \( h \)

\[
\begin{align*}
n_0 &= 1 \\
n_1 &= 2 \\
n_2 &= n_1 + n_0 + 1 = 4
\end{align*}
\]
Let $n_h$ be the minimum number of nodes in an AVL tree of height $h$

$n_0 = 1 \quad n_1 = 2 \quad n_2 = n_1 + n_0 + 1 = 4 \quad n_3 = n_2 + n_1 + 1 = 7$
Let $n_h$ denote the minimum number of nodes in an AVL tree of height $h$

- $n_0 = 1$, $n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
Let $n_h$ denote the minimum number of nodes in an AVL tree of height $h$

- $n_0 = 1$, $n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$

For any AVL tree of height $h$ and size $n$,

$$n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}$$
Let \( n_h \) denote the minimum number of nodes in an AVL tree of height \( h \)

- \( n_0 = 1, n_1 = 2 \) (base)
- \( n_h = n_{h-1} + n_{h-2} + 1 \)
- For any AVL tree of height \( h \) and size \( n \),

\[
    n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}
\]

- If \( h \) is even, let \( i = h/2 \).
  \( \Rightarrow n > 2^{h/2} n_0 = 2^{h/2} \cdot 1 \Rightarrow h = O(\log n) \)
Height of AVL Tree

- Let $n_h$ denote the minimum number of nodes in an AVL tree of height $h$
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
- For any AVL tree of height $h$ and size $n$,
  \[ n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i} \]

- If $h$ is even, let $i = h/2$. \(\Rightarrow\) $n > 2^{h/2}n_0 = 2^{h/2} \cdot 1 \Rightarrow\)
  \(h = O(\log n)\)
- If $h$ is odd, let $i = (h - 1)/2$. \(\Rightarrow\) $n > 2^{(h-1)/2}n_1 = 2^{(h-1)/2} \cdot 2 \Rightarrow\)
  \(h = O(\log n)\)
Let $n_h$ denote the minimum number of nodes in an AVL tree of height $h$

- $n_0 = 1$, $n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$

For any AVL tree of height $h$ and size $n$,

\[ n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i} \]

If $h$ is even, let $i = h/2$. \( \Rightarrow \) $n > 2^{h/2} n_0 = 2^{h/2} \cdot 1 \Rightarrow h = O(\log n)$

If $h$ is odd, let $i = (h-1)/2$. \( \Rightarrow \) $n > 2^{(h-1)/2} n_1 = 2^{(h-1)/2} \cdot 2 \Rightarrow h = O(\log n)$

Thus, many operations (e.g., insertion, deletion, and search) on an AVL tree will take $O(\log n)$ time.
We saw that \( n_h = n_{h-1} + n_{h-2} + 1 \).
Recall Fibonacci numbers satisfy \( f_h = f_{h-1} + f_{h-2} \). Now compare...
We saw that $n_h = n_{h-1} + n_{h-2} + 1$.
Recall Fibonacci numbers satisfy $f_h = f_{h-1} + f_{h-2}$. Now compare

<table>
<thead>
<tr>
<th>$h$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_h$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>20</td>
<td>33</td>
<td>54</td>
<td>88</td>
</tr>
<tr>
<td>$f_h$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>
We saw that \( n_h = n_{h-1} + n_{h-2} + 1 \).
Recall Fibonacci numbers satisfy \( f_h = f_{h-1} + f_{h-2} \). Now compare

\[
\begin{array}{cccccccc}
  h & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  n_h & 1 & 2 & 4 & 7 & 12 & 20 & 33 & 54 & 88 \\
  f_h & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 \\
\end{array}
\]

Lemma:

\[ n_h = f_{h+2} - 1 \]
We saw that \( n_h = n_{h-1} + n_{h-2} + 1 \).

Recall Fibonacci numbers satisfy \( f_h = f_{h-1} + f_{h-2} \). Now compare

<table>
<thead>
<tr>
<th>( h )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_h )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>20</td>
<td>33</td>
<td>54</td>
<td>88</td>
</tr>
<tr>
<td>( f_h )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>

**Lemma:**

\( n_h = f_{h+2} - 1 \)

**Proof: by induction**

\[
\begin{align*}
n_{h+1} &= 1 + n_h + n_{h-1} = 1 + f_{h+2} - 1 + n_{h+1} - 1 = f_{h+3} - 1
\end{align*}
\]

Since \( f_h \sim c\phi^h \) for golden ratio \( \phi = \frac{1+\sqrt{5}}{2} \), this also immediately provides alternative derivation that \( h = O(\log n) \).
Insertion

- Basically follows insertion strategy of binary search tree
• Basically follows insertion strategy of binary search tree
  • But may cause violation of AVL tree property
Insertion

- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed height balance if needed
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed height balance if needed
After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered.
Insertion: Observation

After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered.

Because only those nodes have their subtrees altered.
After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered.

- Because only those nodes have their subtrees altered.

Insert 1
Insertion: Four Cases

● Let $A$ denote the \textit{lowest node} violating AVL tree property
Let $A$ denote the \textit{lowest node} violating AVL tree property.

- **Case 1 (Left-Left case)**
  - insert into the left subtree of the left child of $A$
Let $A$ denote the *lowest node* violating AVL tree property

- **Case 1 (Left-Left case)**
  - insert into the left subtree of the left child of $A$

- **Case 2 (Left-Right case)**
  - insert into the right subtree of the left child of $A$
Let $A$ denote the *lowest node* violating AVL tree property

- **Case 1** (Left-Left case)
  - insert into the left subtree of the left child of $A$

- **Case 2** (Left-Right case)
  - insert into the right subtree of the left child of $A$

- **Case 3** (Right-Left case)
  - insert into the left subtree of the right child of $A$

- **Case 4** (Right-Right case)
  - insert into the right subtree of the right child of $A$
Let $A$ denote the *lowest node* violating AVL tree property

- **Case 1 (Left-Left case)**
  - insert into the left subtree of the left child of $A$

- **Case 2 (Left-Right case)**
  - insert into the right subtree of the left child of $A$

- **Case 3 (Right-Left case)**
  - insert into the left subtree of the right child of $A$

- **Case 4 (Right-Right case)**
  - insert into the right subtree of the right child of $A$
Let $A$ denote the \textit{lowest node} violating AVL tree property

- **Case 1 (Left-Left case)**  
  - insert into the left subtree of the left child of $A$

- **Case 2 (Left-Right case)**  
  - insert into the right subtree of the left child of $A$

- **Case 3 (Right-Left case)**  
  - insert into the left subtree of the right child of $A$

- **Case 4 (Right-Right case)**  
  - insert into the right subtree of the right child of $A$

Cases 1 and 4 are mirror image symmetries with respect to $A$, as are cases 2 and 3
- Right rotation with B as the pivot

- The new subtree rooted at B has height $k + 2$, exactly the same height before the insertion

- The rest of the tree (if any) that was originally above node A always remains balanced
Insertion: Right-Right Case

- **Left rotation** with \( B \) as the pivot

- The new subtree rooted at \( B \) has height \( k + 2 \), *exactly the same* height before the insertion

- The rest of the tree (if any) that was originally above node \( A \) always remains balanced
Insertion: Left-Right Case

- Single rotation fails to fix it
- Subtree $\beta$ is too tall
Left-Right Case: Special Case

When subtree $\alpha$, $\beta$ and $\gamma$ are empty, $k = -1$. Insert $C$:

Left rotation and then right rotation with $C$ as the pivot. Done!
When subtree $\alpha$, $\beta$ and $\gamma$ are empty, $k = -1$. Insert $C$:

- Left rotation and then right rotation with $C$ as the pivot.
  Done!
Left-Right Case: General Case 1

\begin{align*}
A & \quad k + 2 \\
B & \quad -1 \\
\downarrow & \\
X & \quad 0 \\
\downarrow & \\
k & \\
\downarrow & \\
\alpha & \\
\downarrow & \\
k - 1 & \\
\downarrow & \\
\beta & \\
\downarrow & \\
k & \\
\downarrow & \\
\gamma & \\
\downarrow & \\
k - 1 & \\
\downarrow & \\
\delta &
\end{align*}

\begin{align*}
A & \quad k + 3 \\
B & \quad -2 \\
\downarrow & \\
X & \quad k \\
\downarrow & \\
k & \\
\downarrow & \\
\alpha & \\
\downarrow & \\
\gamma & \\
\downarrow & \\
k - 1 & \\
\downarrow & \\
\delta &
\end{align*}

\begin{align*}
A & \quad k + 2 \\
B & \quad 0 \\
\downarrow & \\
X & \quad 0 \\
\downarrow & \\
k & \\
\downarrow & \\
\alpha & \\
\downarrow & \\
k & \\
\downarrow & \\
\beta & \\
\downarrow & \\
k & \\
\downarrow & \\
\gamma & \\
\downarrow & \\
k - 1 & \\
\downarrow & \\
\delta &
\end{align*}

\begin{align*}
A & \quad k + 3 \\
B & \quad 1 \\
\downarrow & \\
X & \quad k \\
\downarrow & \\
k & \\
\downarrow & \\
\beta & \\
\downarrow & \\
k & \\
\downarrow & \\
\gamma & \\
\downarrow & \\
k - 1 & \\
\downarrow & \\
\delta &
\end{align*}
Left-Right Case: General Case 2

\[
A^{-1}B^{-1}X_0\delta_k k + 2 X_0\beta_0^k \gamma_{k-1} B_0 A_{k+2}
\]

AVL Trees Version of September 6, 2016
Right-Left Case: Special Case

When subtree $\alpha$, $\beta$ and $\gamma$ are empty, $k = -1$. Insert $C$:

Right rotation and then left rotation with $C$ as the pivot.

Done!
Right-Left Case: Special Case

When subtree $\alpha$, $\beta$ and $\gamma$ are empty, $k = -1$. Insert $C$:

- Right rotation and then left rotation with $C$ as the pivot.
  Done!
Insertion: Summary

- **Left-Left case:** single rotation
  - Right rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.

Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.
Insertion: Summary

- **Left-Left case**: single rotation
  - Right rotation

- **Left-Right case**: double rotation
  - Left rotation and then right rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree. Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.
Insertion: Summary

- **Left-Left case**: single rotation
  - Right rotation

- **Left-Right case**: double rotation
  - Left rotation and then right rotation

- **Right-Left case**: double rotation
  - Right rotation and then left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.

Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.
Insertion: Summary

- **Left-Left case:** single rotation
  - Right rotation

- **Left-Right case:** double rotation
  - Left rotation and then right rotation

- **Right-Left case:** double rotation
  - Right rotation and then left rotation

- **Right-Right case:** single rotation
  - Left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree. Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.
Insertion: Summary

- **Left-Left case:** single rotation
  - Right rotation

- **Left-Right case:** double rotation
  - Left rotation and then right rotation

- **Right-Left case:** double rotation
  - Right rotation and then left rotation

- **Right-Right case:** single rotation
  - Left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.
Insertion: Summary

- **Left-Left case:** single rotation
  — Right rotation

- **Left-Right case:** double rotation
  — Left rotation and then right rotation

- **Right-Left case:** double rotation
  — Right rotation and then left rotation

- **Right-Right case:** single rotation
  — Left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.

Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.
Delete a node as in ordinary binary search tree
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node

Note: The replacement node has at most one subtree
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root
- For each node along path, restore the height balance if needed by doing single or double rotations
Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root
- For each node along path, restore the height balance if needed by doing single or double rotations
For each node along the path, restore the height balance if needed by doing single or double rotations.
For each node along the path, restore the height balance if needed by doing single or double rotations
- Very similar to insertion with one major caveat
For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance and kept height of rebalanced subtree unchanged. ⇒ Only one rotation needed.
  - In deletion, rotation restores balance but final height of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.
For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance \textit{and} kept height of rebalanced subtree unchanged.  
    \[\Rightarrow\text{ Only one rotation needed.}\]
  - In deletion, rotation restores \textit{balance} but final \textit{height} of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.
  - Since path has length \(O(h)\) this might require \(O(h) = O(\log n)\) rotations.
For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance \textit{and} kept height of rebalanced subtree unchanged. ⇒ Only \textit{one} rotation needed.
  - In deletion, rotation restores \textit{balance} but final \textit{height} of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.
  - Since path has length $O(h)$ this might require $O(h) = O(\log n)$ rotations.

⇒ Deletion can also be done in $O(\log n)$ time.
Diagram below illustrates example in which subtree rooted at $A$ has height $k + 3$. An item is deleted from subtree $\gamma$, reducing its height from $k + 1$ to $k$, leading to an imbalance.

After a single rotation, the subtree is now rooted at $B$ with no imbalance. But, $B$ has height $k + 2$. This might cause an imbalance further up the tree, so the algorithm might need to continue walking upwards, correcting that imbalance.
AVL trees are one particular type of *Balanced* Search trees, yielding $O(\log n)$ behavior for dictionary operations.

There are many other types of Balanced Search Trees, e.g.

- red-black trees
- $B$-trees
- $(a, b)$ trees
  - $(2, 3)$ and $(2, 3, 4)$ trees are special cases
- treaps (randomized BSTs)
- splay Trees (only $O(\log n)$ in amortized sense)