CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 26
AVL Trees
Announcements

- WA2 due tonight @ 11:59pm
## BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

What is the runtime in terms of $n$? $O(n)$

$log(n) \leq d \leq n$
If height(left) ≈ height(right)

$d = O(\log(n))$

If height(left) ≪ height(right)

$d = O(n)$
Balanced Trees are good: Faster find, insert, remove
Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced?
Balanced Trees

Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced? $|\text{height(left)} - \text{height(right)}| \leq 1$
Balanced Trees are good: Faster find, insert, remove

What do we mean by balanced? $|\text{height(left)} - \text{height(right)}| \leq 1$

How do we keep a tree balanced?
Balanced Trees - Two Approaches

Option 1

Keep left/right subtrees within\n\[\pm 1\]\ of each other in height

(Add a field to track amount of "imbalance")

Option 2

Keep leaves at some minimum\ndepth \[(d/2)\]

(Add a color to each node marking it as "red" or "black")
Ok...but how do we enforce this...?
Rebalancing Trees (rotations)
Rebalancing Trees (rotations)

Rotate(A, B)
Rebalancing Trees (rotations)

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained?

Rotate(A, B)
A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity?

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity? O(1)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity? O(1)

This is called a left rotation
(right rotation is the opposite)
Rebalancing Trees
Rebalancing Trees

Rotate(1,2)
Rebalancing Trees

Rotate(2,3)
Rebalancing Trees

Rotate(3,4)
Rebalancing Trees

Rotate(3,2)
Rebalancing Trees

Rotate(5,6)
AVL Trees
An **AVL tree** (Adelson-Velsky and Landis) is a **BST** where every subtree is depth-balanced.

**Remember:** Tree depth = height(root)

**Balanced:** \(|\text{height(root.left)} - \text{height(root.right)}| \leq 1\)
Define $\text{balance}(v) = \text{height}(v.\text{right}) - \text{height}(v.\text{left})$

**Goal:** Maintaining $\text{balance}(v) \in \{-1, 0, 1\}$

- $\text{balance}(v) = 0 \rightarrow "v \text{ is balanced}"$
- $\text{balance}(v) = -1 \rightarrow "v \text{ is left-heavy}"$
- $\text{balance}(v) = 1 \rightarrow "v \text{ is right-heavy}"$
Define $\text{balance}(v) = \text{height}(v.right) - \text{height}(v.left)$

**Goal:** Maintaining $\text{balance}(v) \in \{-1, 0, 1\}$

- $\text{balance}(v) = 0 \rightarrow "v is balanced"
- $\text{balance}(v) = -1 \rightarrow "v is left-heavy"
- $\text{balance}(v) = 1 \rightarrow "v is right-heavy"

*What does enforcing this gain us?*
Question: Does the AVL property result in any guarantees about depth?

Answer: YES!

Depth balance forces a maximum possible depth of $\log(n)$.

Proof Idea: An AVL tree with depth $d$ has "enough" nodes.
Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of $\log(n)$
Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of $\log(n)$

Proof Idea: An AVL tree with depth $d$ has "enough" nodes
AVL Trees - Depth Bounds

Let \( \text{minNodes}(d) \) be the minimum number of nodes an in AVL tree of depth \( d \)

- \( \text{minNodes}(0) = 1 \)
- \( \text{minNodes}(1) = 2 \)
- \( \text{minNodes}(2) = 4 \)
AVL Trees

For any tree of depth $n$:

- subtrees must be balanced, so the other subtree needs to have a depth of at least $n-2$
- at least one subtree needs to have a depth of $n-1$

$$minNodes(n) = \begin{cases} \end{cases}$$
Enough Nodes?

- For $d > 1$
  - $\text{minNodes}(d) = 1 + \text{minNodes}(d-1) + \text{minNodes}(d-2)$
  - This is the Fibonacci Sequence!
    - $\text{minNodes}(d) = \text{Fib}(d+3)-1$
    - $\text{Fib}(0), \text{Fib}(1), \text{Fib}(2), \ldots = 0, 1, 1, 2, 3, 5, 8, \ldots$
    - $\text{minNodes}(d) = \Omega(1.5^d)$
**Enough Nodes?**

- \( \text{minNodes}(d) = \Omega(1.5^d) \)

\[
\begin{align*}
n & \geq c \cdot 1.5^d \\
\frac{n}{c} & \geq 1.5^d \\
\log_2 \left( \frac{n}{c} \right) & \geq \log_2 (1.5^d) \\
\log_2 \left( \frac{n}{c} \right) & \geq \log_{1.5} (1.5^d) \log_2 1.5
\end{align*}
\]

\[
\begin{align*}
\frac{\log_2 \left( \frac{n}{c} \right)}{\log_2 (1.5)} & \geq d \\
\frac{\log_2 (n)}{\log_2 (1.5)} - \frac{\log_2 (c)}{\log_2 (1.5)} & \geq d
\end{align*}
\]

\[
O \left( \log_2 (n) \right) \geq d
\]

A tree with \( n \) nodes and the AVL constraint has logarithmic depth in \( n \)
Enforcing the AVL Constraint

• Computing balance() on the fly is expensive
  – balance calls height() twice
  – Computing height requires visiting every node
    • (linear in the size of the subtree)
• Idea: Store height of each node at the node
  – Better idea: Store balance factor (only requires 2 bits)
Enforcing the AVL Constraint

maintaining _parent makes it possible to traverse up the tree (helpful for rotations), but is not possible in an immutable tree.

```scala
class AVLNode[K, V](
    var _key: K,
    var _value: V,
    var _parent: Option[AVLNode[K, V]],
    var _left: AVLNode[K, V],
    var _right: AVLNode[K, V],
    var _isLeftHeavy: Boolean, // true if balance(this) == -1
    var _isRightHeavy: Boolean, // true if balance(this) == 1
)```

\[
\text{balance}(n) = \begin{cases} 
-1 & \text{if } n._\text{isLeftHeavy} = \text{T} \\
+1 & \text{if } n._\text{isRightHeavy} = \text{T} \\
0 & \text{otherwise}
\end{cases}
\]
Enforcing the AVL Constraint

• Left Rotation
  - Before
    • (A) root; balance(A) = +2 (too right heavy)
    • (B) root.right; balance(B) = +1 (right heavy)
  1) Left subtree of (B) becomes right subtree of (A).
  2) (A) becomes left subtree of (B)
  3) (B) becomes root
  - After
    • balance(A) = 0, balance(B) = 0
Enforcing the AVL Constraint

\[ \begin{align*}
&\text{balance} = +2 \\
&\text{balance} = +1 \\
&\text{height} = h - 1 \\
&\text{height} = h - 1 \\
&\text{height} = h
\end{align*} \]
Enforcing the AVL Constraint

balance = 0

balance = 0

height = h-1  height = h-1  height = h
Enforcing the AVL Constraint

- Right-Left Rotation
  - Before
    - (A) root; balance(A) = +2 (too right heavy)
    - (B) root.right; balance(B) = -1 (left heavy)
    - (C) right.left.right
      1) Left subtree of (C) becomes right subtree of (A).
      2) Right subtree of (C) becomes left subtree of (B).
      3) (A) becomes left subtree of (C)
      4) (B) becomes right subtree of (C)
      5) (C) becomes root
Enforcing the AVL Constraint

- After
  - if (C)’s BF was originally 0
    - (A) BF = 0; (B) BF = 0; (C) BF = 0
  - if (C)’s BF was originally -1
    - (A) BF = 0; (B) BF = +1; (C) BF = 0
  - if (C)’s BF was originally +1
    - (A) BF = -1; (B) BF = 0; (C) BF = 0
Enforcing the AVL Constraint

(balance = +2)

(balance = -1)

(balance = 0, +1 or -1)

\[ h_x = h_y = h \]
or  
\[ h_x = h - 1; h_y = h \]
or  
\[ h_x = h; h_y = h - 1 \]
Enforcing the AVL Constraint

balance = 0

balance = 0 or -1

balance = 0 or +1

h_x = h_y = h
or
h_x = h - 1; h_y = h
or
h_x = h; h_y = h - 1
Enforcing the AVL Constraint

- Rotate Right
  - Symmetric to rotate left
- Rotate Left-Right
  - Symmetric to rotate right-left
Inserting Records

- Inserting Records
  - Find insertion as in BST
  - Set balance factor of new leaf to 0
    - _isLeftHeavy = _isRightHeavy = false
  - Trace path up to root, updating balance factor
    - Rotate if balance factor off
Inserting Records

```scala
def insert[K, V](key: K, value: V, root: AVLNode[K, V]): Unit = {
  var node = findInsertionPoint(key, root);
  node._key = key; node._value = value
  node._isLeftHeavy = node._isRightHeavy = false
  while(node._parent.isDefined){
    if(node._parent._left == node){
      if(node._parent._isRightHeavy){
        node._parent._isRightHeavy = false; return
      }
      else if(node._parent._isLeftHeavy) {
        if(node._isLeftHeavy){ node._parent.rotateRight() }
        else { node._parent.rotateLeftRight() }
        return
      }
    } else {
      node._parent.isLeftHeavy = true
    }
  } else {
    /* symmetric to above */
  }
  node = node._parent
}
```

O(d) = O(log(n)) loops

O(d) = O(log(n))

O(1) per loop

Total Runtime = O(log(n))
Removing Records

- Removing Records
  - Remove the node
    - Find the node containing the value as in BST
      - If it doesn’t exist, return false
    - If the node is a leaf, remove it
    - If the node has one child, the child replaces the node
    - If the node has two children
      - copy smaller child value into node
      - remove smaller child node
  - Fix balance factors
    - Inverse of insertion
Maintaining Balance

- **Claim**: Only the balance factors of ancestors are impacted
  - The height of a node is only affected by its descendents
- **Claim**: Only one rotation will fix any remove/insert imbalance
  - Insert/remove change the height by at most one
- Only log(n) rotations are required for any insert/remove
  - Insert/remove are still log(n)