CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall
Announcements

- PA3 Tests due tonight @11:59PM
- PA3 Implementation due Sunday @ 11:59PM
  - Recitation this week will have some PA3 related content again
## 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 $n$? $O(n)$

$log(n) \leq d \leq n$
Tree Depth vs Size

If $\text{height}(\text{left}) \approx \text{height}(\text{right})$

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

If $\text{height}(\text{left}) \ll \text{height}(\text{right})$

$d = O(n)$
Tree Depth vs Size

If height(left) \approx height(right)

\[ d = O(\log(n)) \]

If height(left) \ll height(right)

\[ d = O(n) \]

We want this, not this
Keeping Depth Small - Two Approaches

Option 1

Keep tree **balanced**: subtrees $+/-1$ of each other in height

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

Option 2

Keep leaves at some minimum depth ($d/2$)

(Add a color to each node marking it as "red" or "black")
Balanced Trees are good: Faster find, insert, remove
Balanced Trees

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? |height(left) - height(right)| ≤ 1

How do we keep a tree balanced?
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!

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?*

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)

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? $O(1)$

This is called a left rotation

(right rotation is the opposite)

Rotate(A, B)
Rebalancing Trees (rotations)

Before Rotation:

```
Before Rotation:
A
  /
 /  
B /    
  
X        Y        Z
```

Rotate(A, B)
Rebalancing Trees (rotations)

Before Rotation:
\[ h(A) = 1 + \max(h(X), 1 + \max(h(Y), h(Z))) \]
Rebalancing Trees (rotations)

Before Rotation:
$h(A) = 1 + \max(h(X), 1 + \max(h(Y), h(Z)))$

After Rotation:
Rebalancing Trees (rotations)

Before Rotation:
\[ h(A) = 1 + \max(h(X), 1 + \max(h(Y), h(Z))) \]

After Rotation:
\[ h(B) = 1 + \max(1 + \max(h(X), h(Y)), h(Z)) \]
Rebalancing Trees (rotations)

Before Rotation:
\[ h(A) = 1 + \max(h(X), 1 + \max(h(Y), h(Z))) \]

After Rotation:
\[ h(B) = 1 + \max(1 + \max(h(X), h(Y)), h(Z)) \]

- If \( X \) was the tallest of \( X, Y, Z \) our total height increased by 1.
- If \( Z \) was the tallest our total height decreased by 1.
- If \( X, Z \) same height, or \( Y \) is the tallest then total is unchanged
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 \( \text{balance}(v) \in \{-1, 0, 1\} \)

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

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

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

*What does enforcing this gain us?*
Question: Does the AVL property result in any guarantees about depth?
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
Let $\text{minNodes}(d)$ be the min number of nodes an in AVL tree of depth $d$

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

For any tree of depth $d$: 

![Diagram showing AVL tree structure with depth bounds](image-url)
AVL Trees - Depth Bounds

For any tree of depth $d$:

At least one subtree must have depth of $d - 1$ (because total depth is $d$)
For any tree of depth $d$:

The other subtree must have a depth of at least $d - 2$ because the AVL constraint does not allow it to differ by more than 1.

At least one subtree must have depth of $d - 1$ (because total depth is $d$).
AVL Tree - Depth Bounds

For $d < 1$:

$$\text{minNodes}(d) = 1 + \text{minNodes}(d - 1) + \text{minNodes}(d - 2)$$
For $d < 1$:

$$\text{minNodes}(d) = 1 + \text{minNodes}(d - 1) + \text{minNodes}(d - 2)$$

This is the Fibonacci Sequence!
For $d < 1$:

$$\text{minNodes}(d) = 1 + \text{minNodes}(d - 1) + \text{minNodes}(d - 2)$$

This is the Fibonacci Sequence!

What is the $d^{\text{th}}$ term of the Fibonacci sequence?

Coarse approximation: $\text{minNodes}(d) = \Omega(1.5^d)$

https://en.wikipedia.org/wiki/Fibonacci_sequence
AVL Tree - Depth Bounds

\[ \text{minNodes}(d) = \Omega(1.5^d) \]
AVL Tree - Depth Bounds

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

\[ n \geq c1.5^d \]
AVL Tree - Depth Bounds

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

\[ n \geq c1.5^d \]

\[ \log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5^d) \]
AVL Tree - Depth Bounds

\[ \min\text{Nodes}(d) = \Omega(1.5^d) \]

\[ n \geq c1.5^d \]

\[ \log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5^d) \]

\[ \log_2 \left( \frac{n}{c} \right) \geq d \log_2(1.5) \]
AVL Tree - Depth Bounds

\[
\min \text{Nodes}(d) = \Omega(1.5^d)
\]

\[
n \geq c1.5^d
\]

\[
\frac{\log_2 \left( \frac{n}{c} \right)}{\log_2(1.5)} \geq d
\]

\[
\log_2 \left( \frac{n}{c} \right) \geq d \log_2(1.5)
\]
AVL Tree - Depth Bounds

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

\[ n \geq c1.5^d \]

\[ \log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5^d) \]

\[ \log_2 \left( \frac{n}{c} \right) \geq d \log_2(1.5) \]

\[ \frac{\log_2 \left( \frac{n}{c} \right)}{\log_2(1.5)} \geq d \]

\[ \frac{\log_2(n)}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d \]
AVL Tree - Depth Bounds

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

$n \geq c1.5^d$

$\log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5^d)$

$\log_2 \left( \frac{n}{c} \right) \geq d \log_2(1.5)$

$\frac{\log_2 \left( \frac{n}{c} \right)}{\log_2(1.5)} \geq d$

$\frac{\log_2(n)}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$

All constants
AVL Tree - Depth Bounds

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

\[ n \geq c1.5^d \]

\[ \log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5^d) \]

\[ \log_2 \left( \frac{n}{c} \right) \geq d \log_2(1.5) \]

\[ \frac{\log_2 \left( \frac{n}{c} \right)}{\log_2(1.5)} \geq d \]

\[ \frac{\log_2(n)}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d \]

\[ d \in O(\log_2(n)) \]
AVL Tree - Depth Bounds

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

Therefore if we enforce the AVL constraint, then a tree with \( n \) nodes will have logarithmic depth

\[ \log_2 \left( \frac{n}{c} \right) \geq \log_2(1.5)^d \]

\[ \frac{\log_2 \left( \frac{n}{c} \right)}{\log_2(1.5)} \geq d \]

\[ d \in O(\log_2(n)) \]
AVL Tree - Depth Bounds

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

Therefore if we enforce the AVL constraint, then a tree with \( n \) nodes will have logarithmic depth.

So how do we enforce the constraint?
Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node
Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node

**Idea:** Store height of each node at the node
Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node

  **Idea:** Store height of each node at the node

  **Better Idea:** Just store the balance factor (only needs 2 bits)
Enforcing the AVL Constraint

class AVLNode[K,V](
    var key: K,
    var value: V,
    var parent: Option[AVLNode[K,V]],
    var left: Tree[K,V],
    var right: Tree[K,V],
    var isLeftHeavy : Boolean, // true if height(right) - height(left) == -1
    var isRightHeavy : Boolean, // true if height(right) - height(left) == 1
)
```scala
class AVLNode[K,V](
    var key: K,
    var value: V,
    var parent: Option[AVLNode[K,V]],
    var left: Tree[K,V],
    var right: Tree[K,V],
    var isLeftHeavy: Boolean, // true if height(right) - height(left) == -1
    var isRightHeavy: Boolean, // true if height(right) - height(left) == 1
)
```

Add fields to track balance, and update them during insertion/removal.
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of insert?
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of insert? Increases by at most 1
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of insert? Increases by at most 1
- What is the effect on the height of remove?
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of insert? Increases by at most 1
- What is the effect on the height of remove? Decreases by at most 1
Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of insert? Increases by at most 1
- What is the effect on the height of remove? Decreases by at most 1

Therefore after an operation that modifies an AVL tree, the difference in heights can be at most 2.

What are the exact ways this broken constraint might show up?
Enforcing the AVL Constraint: Case 1

balance = +2 (too right heavy)  
How can we fix this?

balance = +1 (right heavy)

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

How can we fix this? $\text{rotate}(A, B)$

$\text{B}$

balance = ?

$\text{A}$

balance = ?

$\text{X}$

height = h - 1

$\text{Y}$

height = h - 1

$\text{Z}$

height = h
Enforcing the AVL Constraint: Case 1

How can we fix this? \( \text{rotate}(A, B) \)

Balance = 0 ✓

Height = \( h - 1 \)  Height = \( h - 1 \)  Height = \( h \)
Enforcing the AVL Constraint: Case 2

balance = +2 (too right heavy)  How can we fix this?

balance = 0 (balanced)

case 2 diagram:

A

B

X

Y

Z

height = h - 1  height = h  height = h
Enforcing the AVL Constraint: Case 2

How can we fix this? rotate(A, B)
Enforcing the AVL Constraint: Case 2

How can we fix this? \texttt{rotate}(A,B)

\begin{itemize}
  \item \texttt{balance} = 1 $\checkmark$
  \item \texttt{balance} = -1 $\checkmark$
  \item \texttt{height} = h - 1
  \item \texttt{height} = h
  \item \texttt{height} = h
\end{itemize}
Enforcing the AVL Constraint: Case 3

How can we fix this?

balance = +2 (too right heavy)
balance = -1 (left heavy)

height = h - 1
height = h
height = h - 1
Enforcing the AVL Constraint: Case 3

balance = +2 (too right heavy)

balance = -1 (left heavy)

How can we fix this?
Will just a single left rotation work?

height = \( h - 1 \)
height = \( h \)
height = \( h - 1 \)
Enforcing the AVL Constraint: Case 3

How can we fix this?
Will just a single left rotation work? No

balance = 2 ✗

balance = -1 ✓
Enforcing the AVL Constraint: Case 3

balance = +2 (too right heavy)  
How can we fix this?

balance = -1 (left heavy)

height = h - 1  
height = h  
height = h - 1
Enforcing the AVL Constraint: Case 3

- Balance of A: +2 (too right heavy)
- Balance of B: -1 (left heavy)

How can we fix this?

Height of C we know must be $h$

Therefore at least one of $h_x$ or $h_y$ must be $h - 1$

The other can also be $h - 2$, or $h - 1$
Enforcing the AVL Constraint: Case 3

How can we fix this?
Rotate right first: rotate(B,C)

Height of C we know must be $h$
Therefore at least one of $h_x$ or $h_y$ must be $h - 1$
The other can also be $h - 2$, or $h - 1$
Enforcing the AVL Constraint: Case 3

How can we fix this?
Rotate right first: \texttt{rotate}(B, C)
Then right left: \texttt{rotate}(A, C)

Height of C we know must be $h$
Therefore At least one of $h_x$ or $h_y$ must be $h - 1$
The other can also be $h - 2$, or $h - 1$
Enforcing the AVL Constraint: Case 3

How can we fix this?
Rotate right first: `rotate(B, C)`
Then right left: `rotate(A, C)`

Height of C we know must be $h$
Therefore At least one of $h_x$ or $h_y$ must be $h - 1$
The other can also be $h - 2$, or $h - 1$
Enforcing the AVL Constraint

- If too right heavy (balance == +2)
  - If right child is right heavy (balance == +1) or balanced (balance == 0)
    - rotate left around the root
  - If right child is left heavy (balance == -1)
    - rotate right around root of right child, then rotate left around root
- If too left heavy (balance == -2)
  - Same as above but flipped

Therefore if we have a balance factor that is off, but all children are AVL trees, we can fix the balance factor in at most 2 rotations
To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST)
2. Insert the new leaf and set balance factor to 0
3. Trace path back up to root and update balance factors
   a. If a balance factor becomes +/-2 then rotate to fix
To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST) \( O(d) = O(\log n) \)
2. Insert the new leaf and set balance factor to 0 \( O(1) \)
3. Trace path back up to root and update balance factors \( O(d) = O(\log n) \)
   a. If a balance factor becomes +/-2 then rotate to fix \( O(1) \)
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
}
Inserting Records

```python
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
}
```

Find insertion point and create the new leaf $O(d) = O(\log n)$
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
  }
  Find insertion point and create the new leaf $O(d) = O(\log n)$
  $O(d) = O(\log n)$ iterations
}
Inserting Records

```python
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
    }
}
```

Find insertion point and create the new leaf $O(d) = O(\log n)$

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

$O(1)$ per iteration
Removing Records

- Removal follows essentially the same process as insertion