CSE 250
Lecture 26-27
AVL Trees & RB Trees
BST Operation Costs

<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>
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.

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

balance(n) = \begin{cases} 
    -1 & \text{if } n._isLeftHeavy = T \\
    +1 & \text{if } n._isRightHeavy = T \\
    0 & \text{otherwise} 
\end{cases}
Fixing Unbalanced Trees

- Assumptions:
  - There is one subtree with exactly one unbalanced node
  - It has a balance factor of $\pm 2$
Fixing Unbalanced Trees

A

balance = +2

X
height = h-1

Y
height = h+1
Fixing Unbalanced Trees

balance = -1, 0, or +1

height = h+1

height = h_y

height = h_z

\[\begin{array}{c|c|c}
\text{bal} & -1 & 0 \\
\text{h_y} & h & h \\
\text{h_z} & h-1 & h-1 \\
\end{array}\]
Fixing Unbalanced Trees

Case 1:

- **A**: balance = +2
- **B**: balance = +1
- **X**: height = h-1
- **Y**: height = h-1
- **Z**: height = h
Fixing Unbalanced Trees

Case 1:

balance = 0
height = h-1

balance = 0
height = h

balance = 0
height = h-1
Fixing Unbalanced Trees

Case 2:

- A
  - balance = +2
  - X
    - height = h - 1
  - B
    - balance = 0
    - Y
      - height = h
    - Z
      - height = h
Case 2:

- **Balance:** $-1$
- **Height:** $h - 1$

**Subtrees:**
- **X:** 
  - **Balance:** $+1$
  - **Height:** $h$
- **Y:** 
  - **Height:** $h$
- **Z:** 
  - **Height:** $h$
Fixing Unbalanced Trees

Case 3:

- A: balance = +2
  - X: height = h-1
  - B: balance = -1
    - Y: height = h
    - Z: height = h-1
Fixing Unbalanced Trees

Case 3:

balance = -2

balance = +1

height = h-1

height = h

height = h-1
Fixing Unbalanced Trees

Y

height = h

Y

height = h

W

height = h

balance = -1, 0, or +1

\[ \text{bal} = \begin{cases} -1 & \text{if } h_y = h-1 \text{ and } h_w = h-2 \\ 0 & \text{if } h_y = h-1 \text{ and } h_w = h-1 \\ +1 & \text{if } h_y = h-1 \text{ and } h_w = h-1 \end{cases} \]
Fixing Unbalanced Trees

Case 3.1:

- **A**: balance = +2
- **B**: balance = -1
- **C**: balance = +1

- **X**: height = h-1
- **Y**: height = h-2
- **W**: height = h-1
- **Z**: height = h-1
Fixing Unbalanced Trees

Case 3.1:

- **A**: balance = +2
- **C**: balance = +2
- **B**: balance = 0

- **X**: height = h-1
- **Y**: height = h-2
- **W**: height = h-1
- **Z**: height = h-1
Fixing Unbalanced Trees

Case 3.1:

balance = -1
height = h-1

balance = 0
height = h-1

balance = 0
height = h-2

balance = 0
height = h-1
Fixing Unbalanced Trees

Case 3.2:

\[
\text{balance} = 0 \\
\text{height} = h - 1 \\
\text{height} = h - 1 \\
\text{height} = h - 1
\]
Fixing Unbalanced Trees

Case 3.3:

balance = 0
height = h-1
A

balance = 0
height = h-1
X

balance = +1
height = h-2
W

balance = 0
height = h-1
Y

balance = +1
height = h-1
Z

balance = 0
height = h-1
B
Enforcing the AVL Constraint

- Left Rotation
  - Before
    - (A) root; \(\text{balance}(A) = +2\) (too right heavy)
    - (B) root.right; \(\text{balance}(B) = +1\) (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

- 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

- 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
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){ /* Pick rotation */
        } else { node._parent.rotateLeftRight() 
        } return
      } else {
        node._parent.isLeftHeavy = true
      }
    } else {
      /* symmetric to above */
      node = node._parent
    }
  }
}
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)
Maintaining Balance

- Enforcing height-balance is too strict
  - May require “unnecessary” rotations
- Weaker restriction:
  - Balance the depth of EmptyTree nodes
  - If a, b are EmptyTree nodes:
    - $\text{depth}(a) \geq \left( \text{depth}(b) \div 2 \right)$
    - or
    - $\text{depth}(b) \geq \left( \text{depth}(a) \div 2 \right)$
Balancing Empty Node Depth

```
          A
         / \
        B   C
       /   /  \
      D   E   F  G
     /   /   /   \
    H   I   F   G
   /   /   /   \
  J   I   F   G
```

OK!
Balancing Empty Node Depth

Not OK!
Balancing Empty Node Depth

Must be full \((2^{\lceil d/2 \rceil} \text{ nodes})\)

\[
d/2 = \log(n) \\
d = 2\log(n) = O(\log(n))
\]
Red-Black Trees

• Color each node red or black
  1) # of black nodes from each empty to root must be identical
  2) Parent of a red node must be black

• On Insertion (or deletion)
  – Inserted node is red (won’t change # of black nodes)
  – “Repair” violations of rule 2 by rotating or recoloring
    • Repairs guarantee rule 1 is preserved
Red-Black Trees
Red-Black Trees

Repair A

All Valid R-B Tree Fragments
Red-Black Trees

Case 1: All Good!
Red-Black Trees

Case 1b: All Good!
Red-Black Trees

Case 1b: All Good!
Problem!
Case 2: Split Black Node
Red-Black Trees

Case 2: Split Black Node

C’s parent may be red
(repeat the repair process)

# of black nodes on each path didn’t change
Red-Black Trees

Case 2: Split Black Node

Also works if A is right-child of B (or B is right-child of C)
Red-Black Trees

Case 3: Rotate B, C
Case 3: Rotate B, C

-1 black node to the root

Same # of black nodes to the root
Red-Black Trees

Case 3: Rotate B, C

-1 black node to the root

Root of subtree under consideration is black (repair is all done)

Same # of black nodes to the root
Case 4: Rotate A, B → B, C
Red-Black Trees

Case 4: Rotate A, B → B, C

Now identical to case 3
Red-Black Trees

- Each insertion creates at most one red-red parent-child conflict
  - \(O(1)\) time to recolor/rotate to repair color
  - May create a red-red conflict in grandparent
    - Up to \(d/2 = O(\log(n))\) repairs required
- Each deletion removes at most one black node
  - \(O(1)\) time to recolor/rotate to preserve black-depth
  - May require recoloring (grand-)parent from black to red
    - Up to \(d = O(\log(n))\) repairs required