CSCI 241
Lecture 14b
AVL rebalancing
Goals

• Understand how rebalance decides to what rotations to perform.

• Be prepared implement rebalance.
AVL Insertion

\[
\text{insert}(\text{Node } n, \text{ int } v):
\]

// ...(other case, irrelevant here)
else:  // v > n.value
    if n has right:
        \text{insert}(n.\text{right}, v)
    else:
        // attach new node w/ value
        // v to n.\text{right}
        \text{rebalance}(n);

How did we know what rotation to do?
Reminder: Tree Rotations

\[
\text{LEFT-ROTATE}(T, x) \\
\text{RIGHT-ROTATE}(T, y)
\]
Reminder: Tree Rotations

subtrees (could be null, leaf, or tree with many nodes)

\[
\text{LEFT-ROTATE}(T, x) \quad \text{RIGHT-ROTATE}(T, y)
\]
AVL Rebalance

Before an insertion that unbalances n, the tree must look like one of these:
AVL Rebalance

Before an insertion that unbalances \( n \), the tree must look like one of these:
AVL Rebalance

Before an insertion that unbalances N, the tree must look like one of these:

An insertion that *unbalances* N could go one of four places.
AVL Rebalance

Before an insertion that unbalances $n$, the tree must look like one of these:

Case 1

Case 2

Case 3

Case 4

An insertion that unbalances $n$ could go one of four places.
AVL Rebalance

Case 1: After BST insertion step, the tree looks like this.

```
   N
  /   \
 C     h \\
/     /  \h
T     V   \\
/  h+1/   \U
```
AVL Rebalance

Case 1: After BST insertion step, the tree looks like this.
Case 1: After BST insertion step, the tree looks like this.

Solution: right rotate on N.
Case 1: After BST insertion step, the tree looks like this.
Solution: right rotate on N.
AVL Rebalance

Case 1: After BST insertion step, the tree looks like this.

Solution: right rotate on N.

N is now AVL balanced.
AVL Rebalance

Before an insertion that unbalances n, the tree must look like one of these:

An insertion that unbalances n could gone one of four places.
Case 2: After BST insertion step, the tree looks like this.
Case 2: After BST insertion step, the tree looks like this.
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. Left rotate C
2. Right rotate N
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. **Left rotate C**
2. **Right rotate N**
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. Left rotate C
2. Right rotate N
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. **Left rotate C**
2. **Right rotate N**
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. Left rotate C
2. Right rotate N
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. Left rotate C
2. Right rotate N
AVL Rebalance

Case 2: After BST insertion step, the tree looks like this.

Solution - two rotations:
1. Left rotate C
2. Right rotate N

Tree is now AVL balanced.
AVL Rebalance

Before an insertion that unbalances $n$, the tree must look like one of these:

- **Case 1**
  - $N_{-1}$
  - $C_{h+1}$
  - $T_h$
  - $U_h$

- **Case 2**
  - $N_{-1}$
  - $C_{h+1}$
  - $V_h$

- **Case 3**
  - $N_1$
  - $C_{h+1}$
  - $V_h$

- **Case 4**
  - $N_1$
  - $C_{h+1}$

An insertion that unbalances $n$ could have gone one of four places.
AVL Rebalance

Before an insertion that unbalances $n$, the tree must look like one of these:

Case 1

Case 2

Case 3

Case 4

An insertion that unbalances $n$ could go to one of four places.
AVL Rebalance

Before an insertion that unbalances n, the tree must look like one of these:

Case 1

Case 2

Case 3

Case 4

An insertion that unbalances n could gone one of four places.
Before an insertion that unbalances $n$, the tree must look like one of these:

Case 1

Case 2

Case 3

Case 4

An insertion that unbalances $n$ could go to one of four places.
void rebalance(n):
  if balance(n) < -1:
    if balance(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if balance(n) > 1:
    if balance(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case 4:
      // leftRot(n)
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case 4:
      // leftRot(n)
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)

Cases 3 and 4 are symmetric with 2 and 1
Implementation

void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)

Cases 3 and 4 are symmetric with 2 and 1.
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)

Cases 3 and 4 are symmetric with 2 and 1.

Details

• Implementing bal:
  • calculating height as in lab 4 is O(n)! Not good enough.
  • Nodes track their height and update when the tree changes
  • Update each node’s height on the way up the tree, calculating height using only its children’s heights.
How did we know what rotation to do?

Insertion with Rebalance

```
insert(Node n, int v):
    // ...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            //   v to n.right
            rebalance(n);

insert(a, 16)
=>insert(c, 16)
    =>insert(f, 16)
        =>attach new node
            rebalance(f)
            rebalance(c)
                perform rotation
            rebalance(a) already balanced
```