Balance factor: \( \text{bal}(v) = \text{hgt}(v.\text{right}) - \text{hgt}(v.\text{left}) \)

AVL Height Balance
- for each node \( v \), the heights of its subtrees differ by \( \leq 1 \).

AVL tree: A binary search tree that satisfies this condition.

AVL Trees I
- Basic defs
- Height props
- Rotations

Does this imply \( O(\log n) \) height?
Worst cases:
- height: \( h = 0 \ 1 \ 2 \ldots \ h \)
- nodes: \( n = 1 \ 2 \ 4 \ 7 \ 12 \ 20 \ldots \)

Recall: \( F_0 = 0, F_1 = 1, F_h = F_{h-1} + F_{h-2} \)

Conjecture: Min no. of nodes in AVL tree of height \( h \) is \( F_{h+3} - 1 \)

Theorem: An AVL tree of height \( h \) has at least \( F_{h+3} - 1 \) nodes.

Proof: (Induct. on \( h \))
- \( h = 0 \): \( n(h) = 1 = F_3 - 1 \)
- \( h = 1 \): \( n(h) = 2 = F_4 - 1 \)
- \( h \geq 2 \):
  - \( n(h) = 1 + n(h-1) + n(h-2) \)
  - \( = 1 + (F_{h-2} + 1) + (F_{h-3} + 1) \)
  - \( = (F_{h+1} + F_{h+1}) - 1 = F_{h+3} - 1 \)

Corollary: An AVL tree with \( n \) nodes has height \( O(\log n) \).

Proof: Fact: \( F_n \approx \phi^n / \sqrt{5} \) where \( \phi = (1 + \sqrt{5}) / 2 \) "Golden ratio"
- \( n \geq \phi^n = c \cdot \phi^n \Rightarrow h \leq \log \phi n + c' \)
- \( h \leq \log_\phi n / \log_\phi \phi = O(\log n) \)
**Double rotations:**

- **left-right (LR):**
  - If the balance factor is less than -1, perform a right rotation on the left child of the node.
  - If the balance factor is greater than 1, perform a left rotation on the right child of the node.

- **right-left (RL):**
  - If the balance factor is greater than 1, perform a left rotation on the left child of the node.
  - If the balance factor is less than -1, perform a right rotation on the right child of the node.

**AVL Trees II**

- Simplified than bal factor.

**AVL Tree:**

- **AVL Node:** Same as BSTNode (from Lect 4) but add:
  - `int height` Utility:
    - `int height(AVLNode p) { return p == null ? -1 : p.height; }
    - `void updateHeight(AVLNode p) { p.height = 1 + max(height(p.left), height(p.right)); }
    - `int balanceFactor(AVLNode p) { return height(p.right) - height(p.left); }

- **Find:** Same as BST.
- **Insert:** Same as BST but as we "back out" rebalance.

**How to rebalance?** Bal = -2

- **Left-right heavy:**
  - **Left-left heavy:**
  - **Right-right heavy:**
  - **Right-left heavy:**

**AVL Node rebalance (AVLNode p):**

- `int balanceFactor(AVLNode p) { return height(p.right) - height(p.left); }
- if (p == null) return p
- if (balanceFactor(p) < -1) {
  - if (height(p.left.left) >= height(p.left.right)) {
    - p = rotateRight(p)
  - else p = rotateLeftRight(p)
  - single rotation
  - updateHeight(p); return p
- } else if (balanceFactor(p) > 1) {
  - if (height(p.left.left) <= height(p.left.right)) {
    - p = rotateLeft(p)
  - else if (balanceFact(p) < -1) {
    - p = rotateRightRight(p)
  - else if (balanceFact(p) > 1) {
    - p = rotateRightRight(p)
  - else throw Error - Duplicate!
  - hole = null
  - return rebalance(p)
- } else return p

**AVLNode insert(Key x, Value v, AVLNode p);**

- if (p == null) p = new AVLNode(x, v)
- else if (x < p.key) p.left = insert(x, v, p.left)
- else if (x > p.key) p.right = insert(x, v, p.right)
- else throw Error - Duplicate!
- return rebalance(p)
Deletion: Basic plan
- Apply standard BST deletion
  - find key to delete
  - find replacement node
  - copy contents
  - delete replacement
  - rebalance

AVL Trees III
- Deletion
- Examples

AVL-Node delete (Key x, AVLNode p)
  - same as BST delete
  - return rebalance(p)

Examples:

Insert(8):

Example 2:

Example 3:
delete(7)

Example 4:
delete(7)
Rebalancing after insertion:

Left-left heavy:

Left-right heavy:

All $O(\log n)$

Rebalancing after deletion:

Left-left heavy:

Left-right heavy:

Further rot. may be needed

Further not needed

one of $? \exists$ exists

 deletes

delete
Quake Heaps

- Heap-ordered tree:
  - Parent key ≤ child keys

- Quake Heap:
  - Leaves all at same depth (level 0)
  - Every key appears in one leaf
  - For any internal u:
    - either: right == null
    - left key ≤ right key
  - u's key = left key

- Quake-Heap can have multiple trees
  - Eg. use Java linked list

```
Level
3 -
2 -
1 -
0 -
```

```
keys: 1, 6, 4, 9, 20
```
To prevent trees from proliferating - merge trees

for $k = 0$ to $n$ levels - 2

- sort roots on level $k$ [not needed, just for consistency]
- while (roots[$k$] has $\geq 2$ trees)
  - remove smallest two - $u, v$
  - merge them & add new root to level $k+1$
Merge Trees - Cascading merge

1. Merge the root nodes:
   - Merge 7 and 21 to get 28.

2. Sort the merged nodes:
   - Sort the resulting nodes 5, 8, 28, 10, 11, 14.

3. Merge the sorted nodes:
   - Merge 7 and 28 to get 35.

4. Sort the merged nodes again:
   - Sort the resulting nodes 5, 8, 35, 10, 11, 14.

5. Merge the sorted nodes:
   - Merge 5 and 7 to get 12.

6. Sort the merged nodes one last time:
   - Sort the resulting nodes 5, 8, 12, 10, 11, 14.

7. The tree is now fully merged and sorted.

Done.