CSCI 241

Lecture 12

AVL Trees, Continued

Happenings

Monday, 2/11 – CSCI Candidate, Jeff Caley (Research) – 4PM, CF 316

Tuesday, 2/12 – CSCI Candidate, Jeff Caley (Teaching)– 4PM, CF 316

Tuesday, 2/12 – Hackathon Info meeting – 5PM, CF 316

Wednesday, 2/13 – Peer Lecture Series: C Language – 5PM, CF 420

Thursday, 2/14 – CSCI Candidate, Sayeed Sajal (Research)– 4PM, CF 316

Friday, 2/15 – CSCI Candidate, Sayeed Sajal (Teaching)– 4PM, *<u>**CF 226***</u>

Saturday 2/16 & Sunday 2/17 – Hackathon – 10AM start time, 24hrs, downstairs labs

Goals

- Understand why we're doing all this tree stuff.

- Be prepared to implement AVL rebalancing.

Announcements

What are we even doing

- Trees
  - Binary Trees
    - Binary Search Trees
      - Balanced BSTs
        - AVL trees - a scheme for maintaining balance
        - Red-black trees - a different scheme for the same thing

What are we even doing

- Trees what are they? nodes with 0 or more children (subtrees)
- Binary Trees nodes with 0,1, or 2 children (subtrees)
- Binary Search Trees | **search** / **insert** / **remove** | and their **runtimes**

  BST property
- Balanced BSTs why do we want this?

  balanced trees give good runtime
- AVL trees

  how do we measure balance?

  balance factor

  how do we achieve balance?

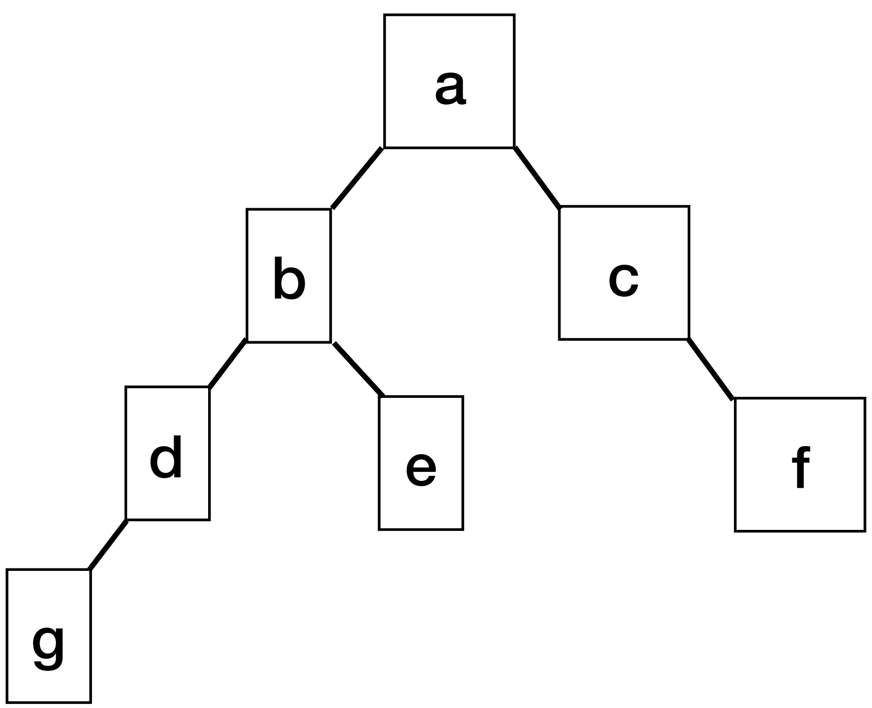
  rotations

  how do we know what/when to rotate?

Why are we even doing this

- Balanced trees have height O(log n)
- Searching, inserting, removing all have runtime O(h).
- log(n) is vastly better than n:
  - log(n) ~= number of digits in n
- When would you want this:
  - **Sets** with no duplicates (counting unique items, as in A2)
  - **Maps**, that store key-value pairs
    (e.g., dictionaries that store word: definition)
  - Especially when you want to traverse set elements (or keys) in sorted order
    - in-order traversal!

Heights and Balance Factors



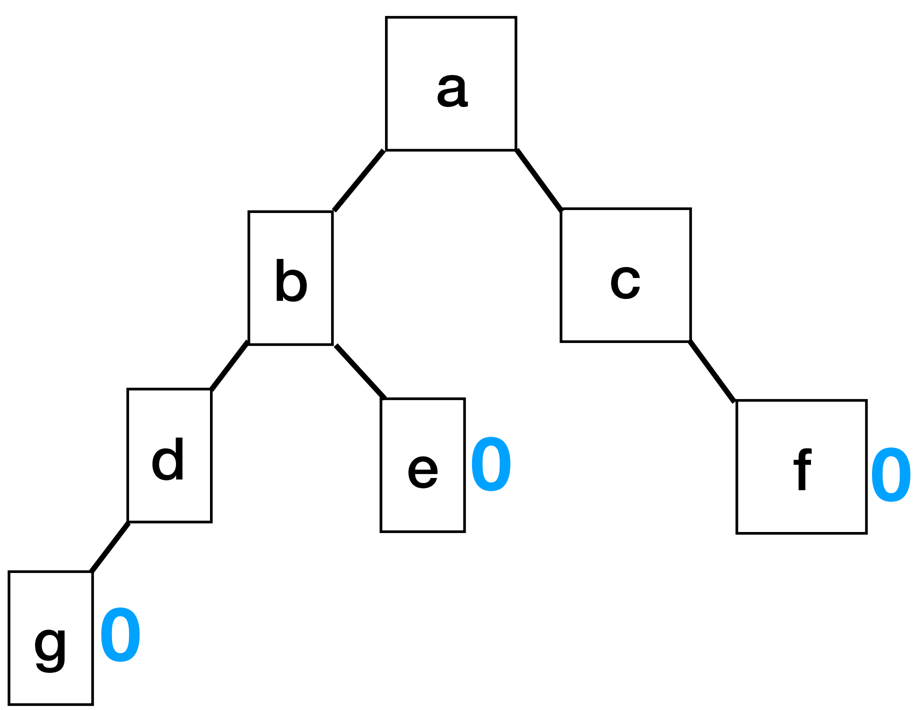
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



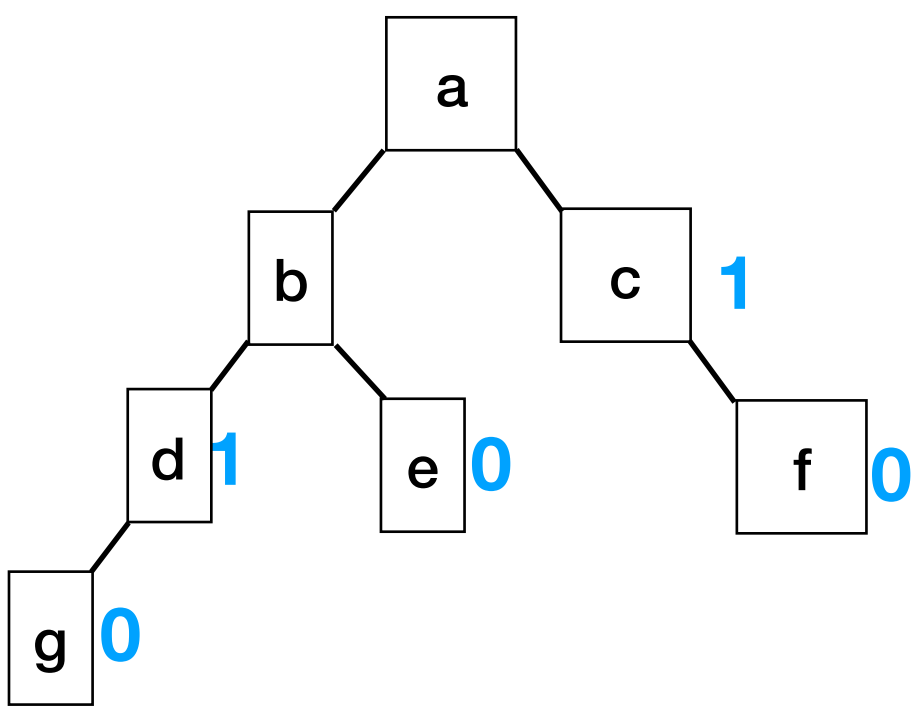
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



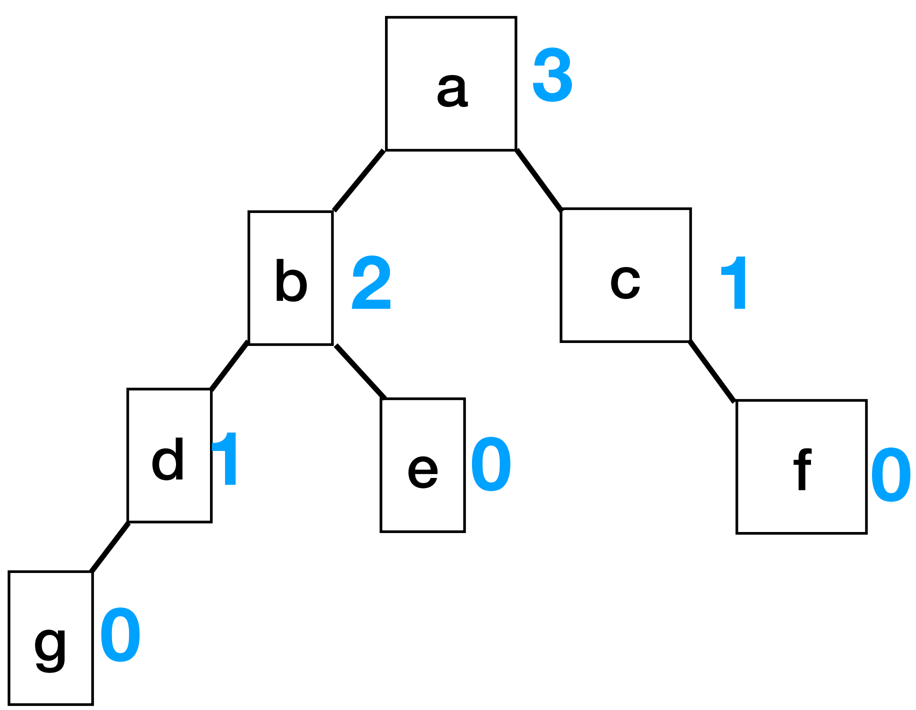
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



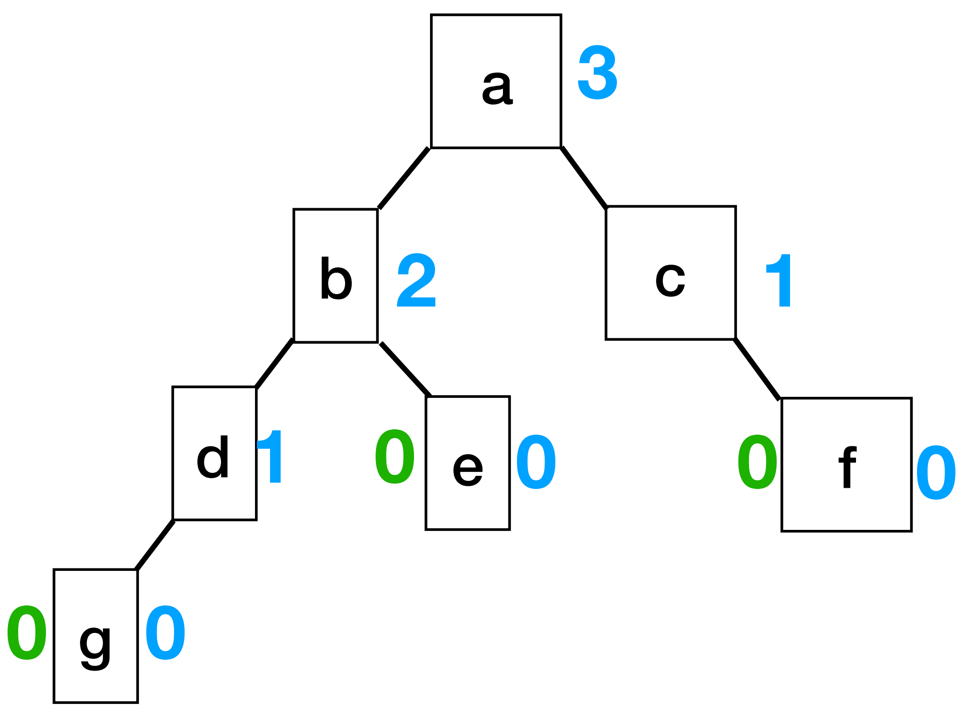
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



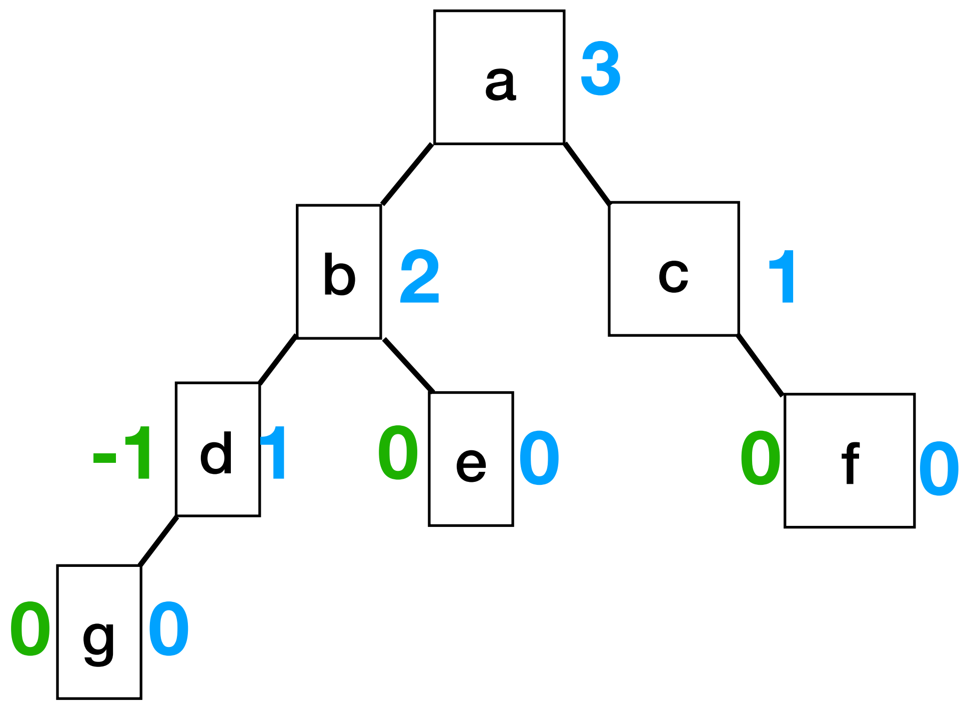
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



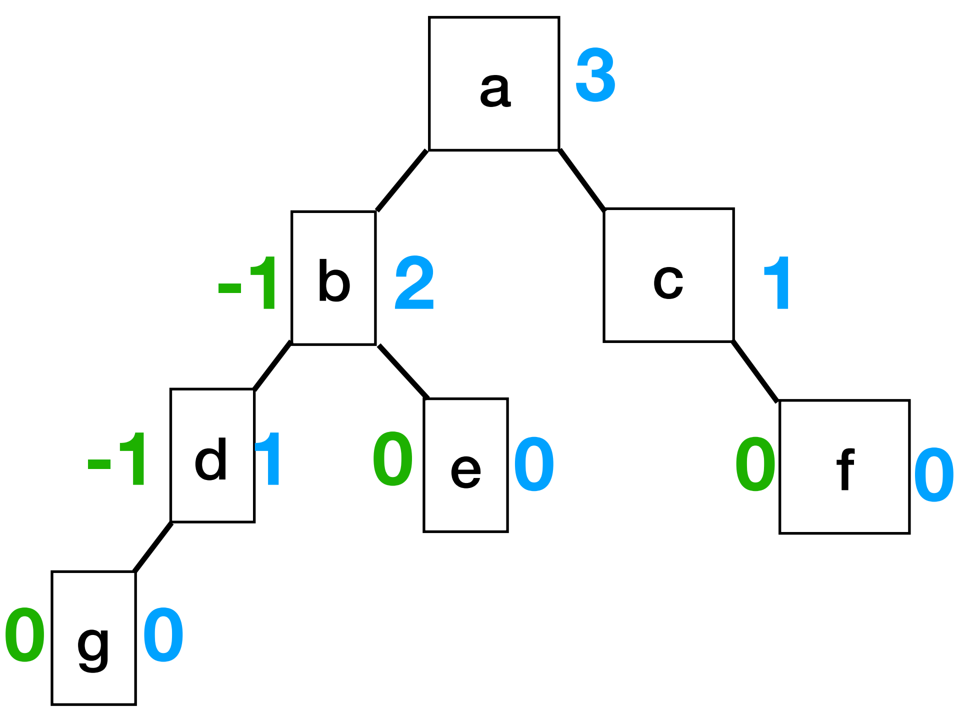
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right)
```

Heights and Balance Factors



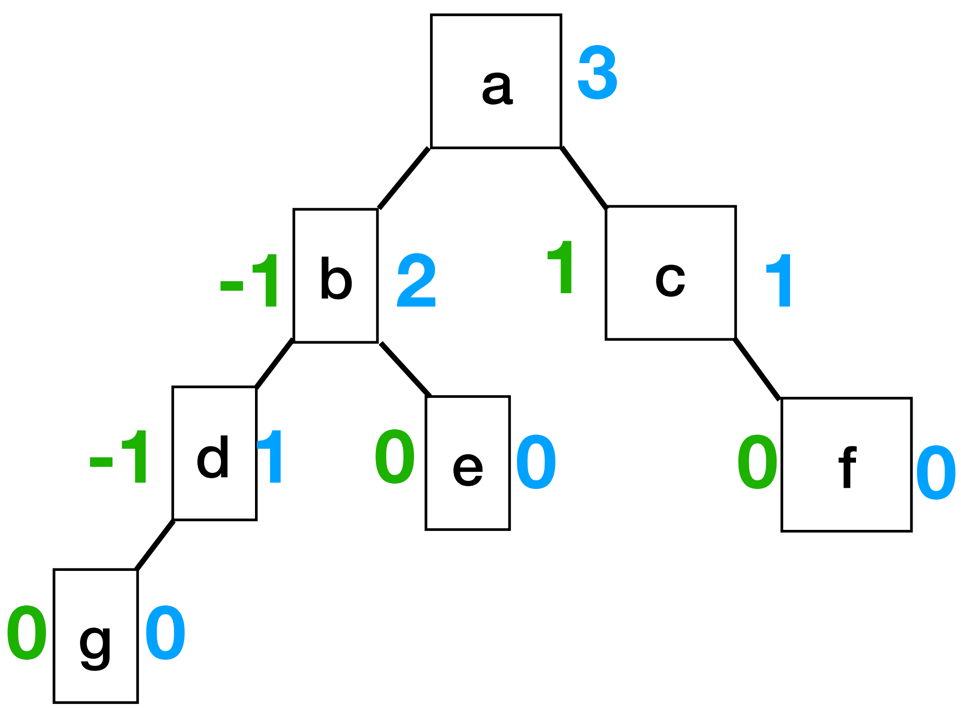
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



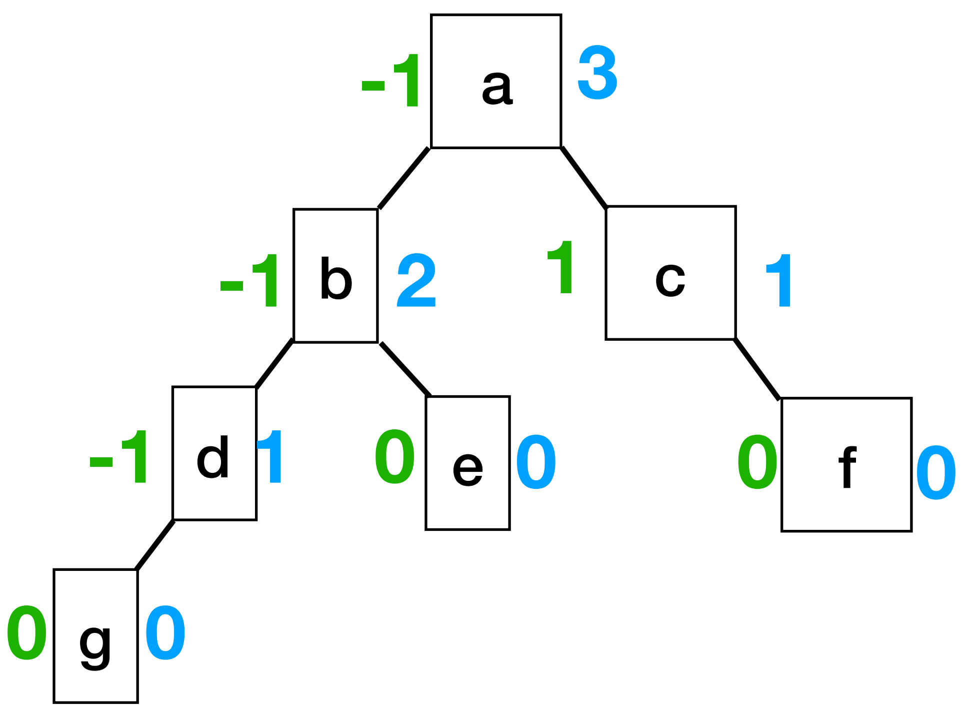
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



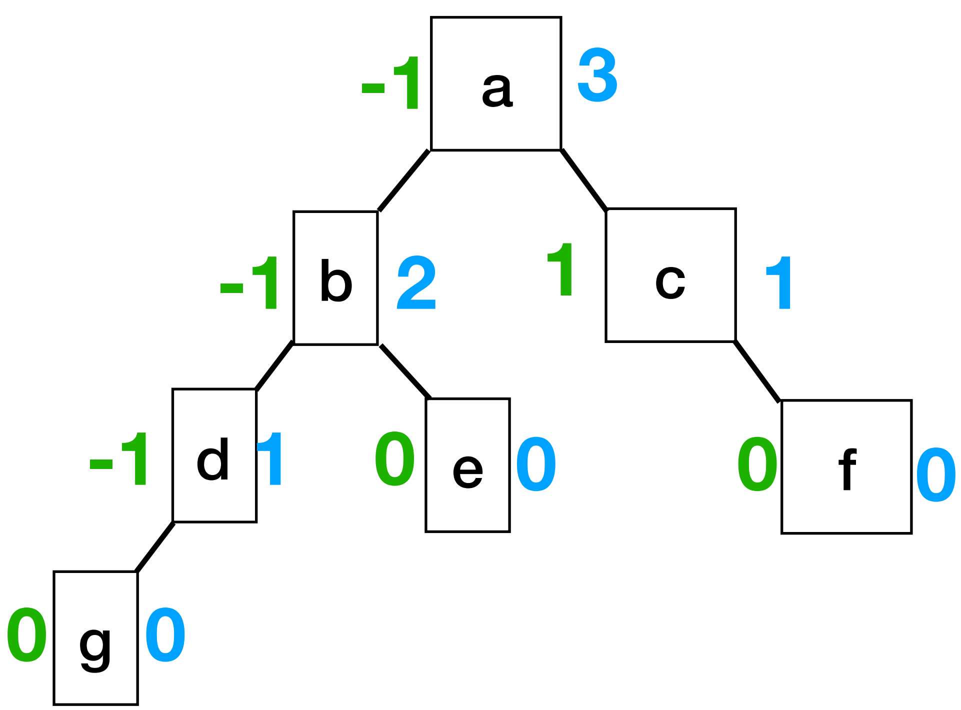
Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Heights and Balance Factors



Height(t): path length from t's deepest descendant (leaf) to t's root.

Height(n): height of the subtree rooted at n

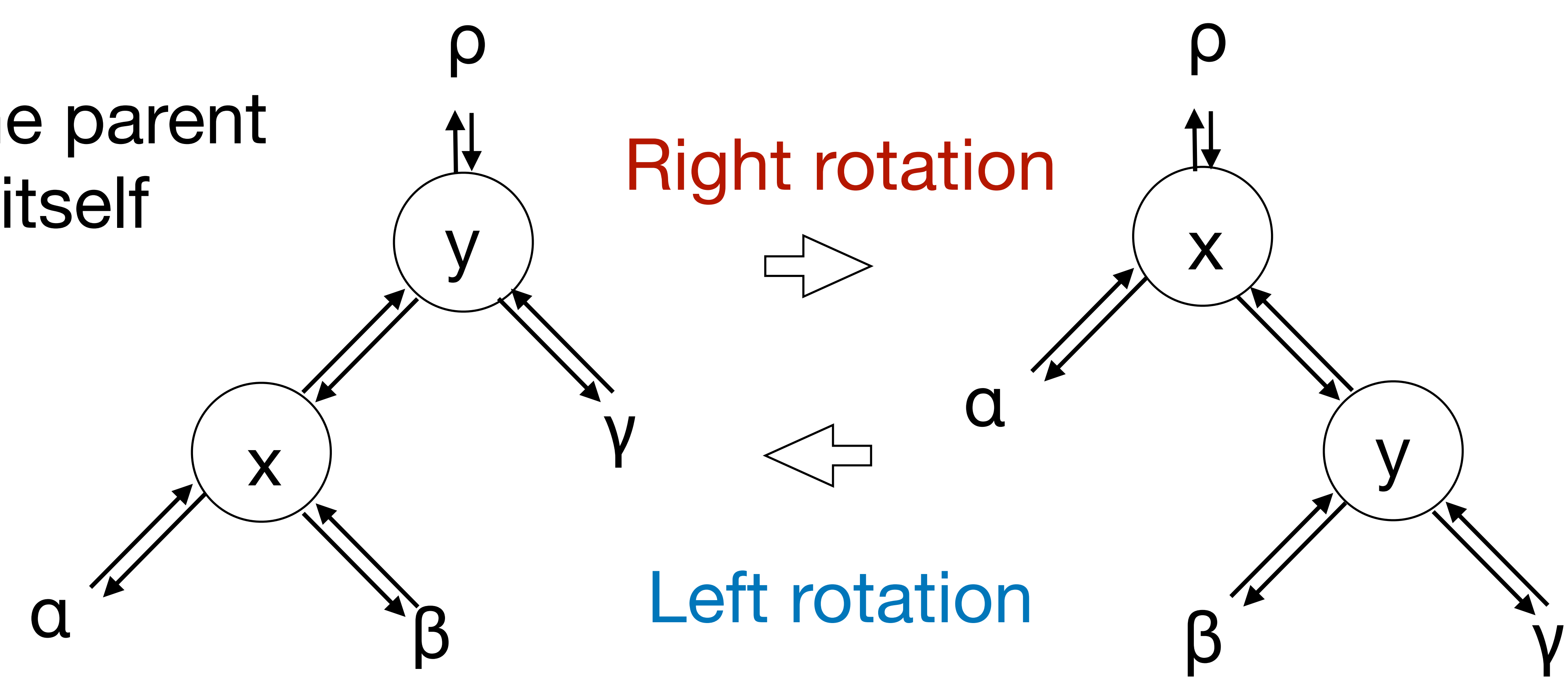
Balance(n): height(n.right) - height(n.left)

```
height(null) = -1
height(n) = 1 + max(height(n.left), height(n.right))
```

Tree Rotations

Steps in left rotation (move y up to x's position):

1. Transfer β
2. Transfer the parent
3. Transfer x itself



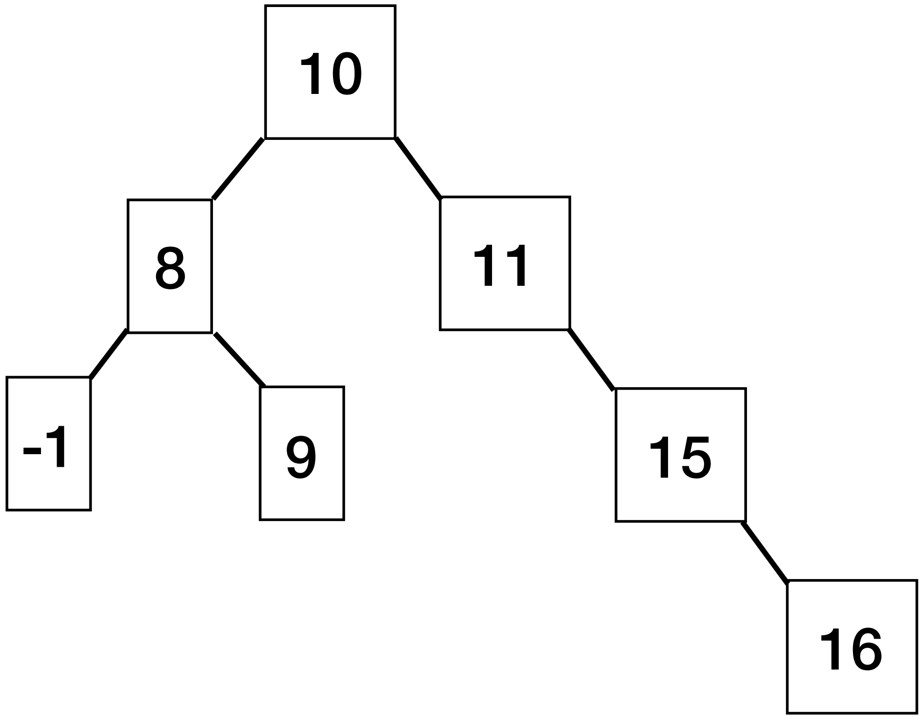
x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y

y.L gets x
x.p gets y

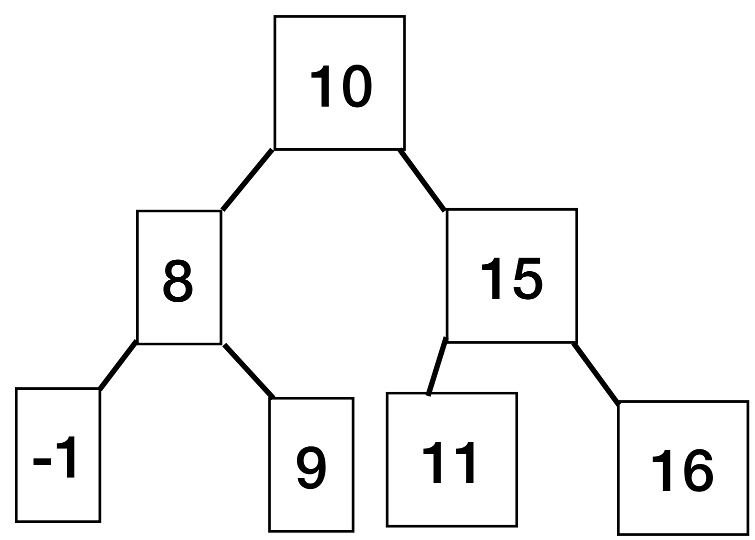
Can we improve balance?

Balance(n): height(n.right) - height(n.left)



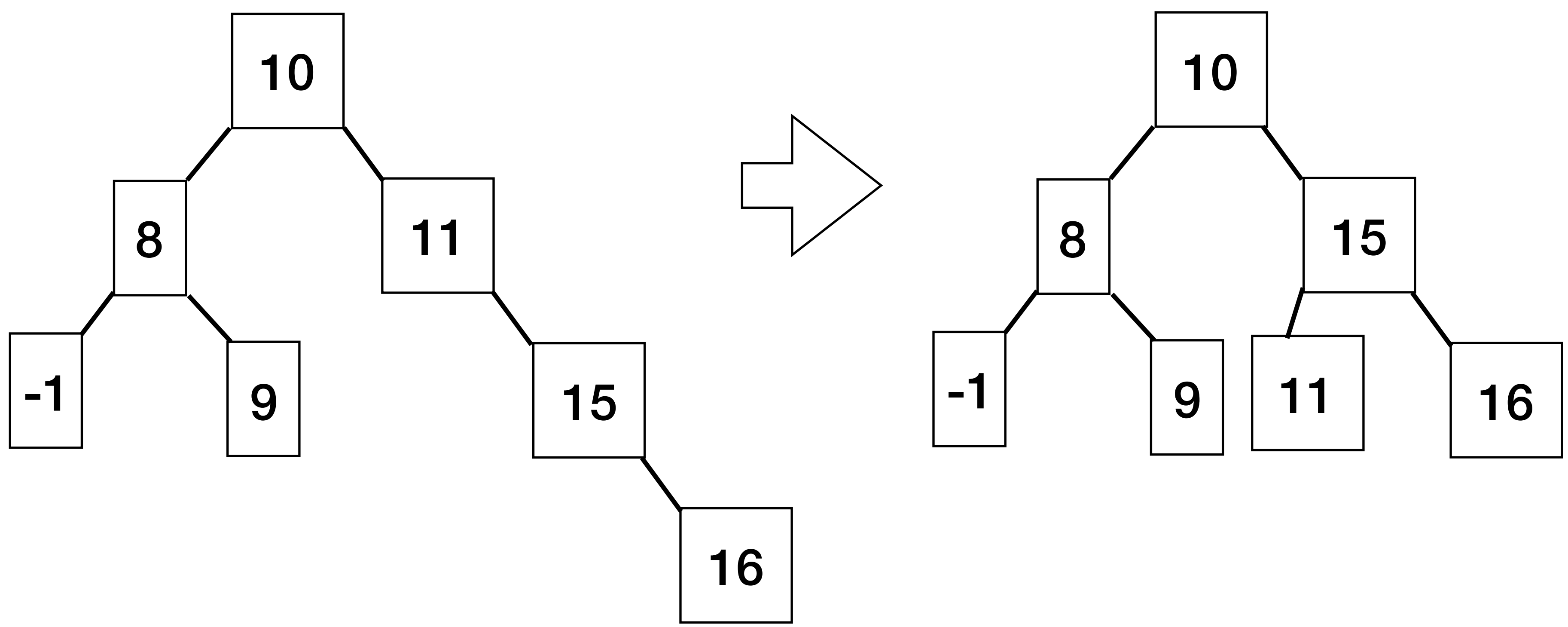
Can we improve balance?

Balance(n): height(n.right) - height(n.left)



Can we improve balance?

Balance(n): height(n.right) - height(n.left)

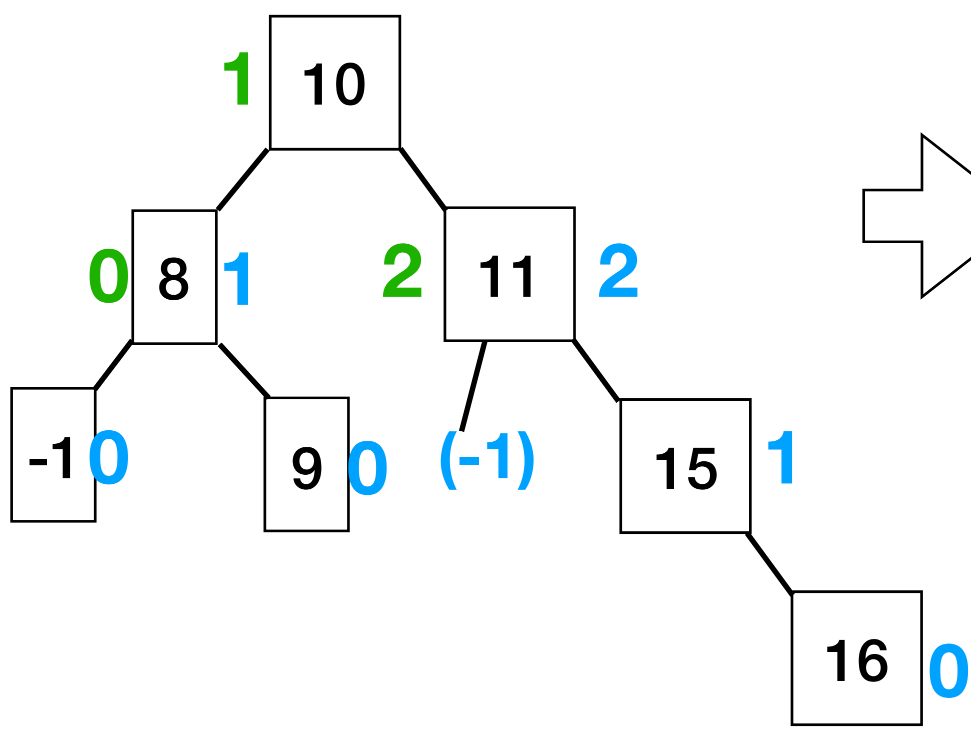


ABCD: The largest (absolute) balance factor in the tree is:

A. 0 before, 1 after rotation
B. 1 before, 0 after rotation
C. 0 before, 2 after rotation
D. 2 before, 0 after rotation

Can we improve balance?

Balance(n): height(n.right) - height(n.left)

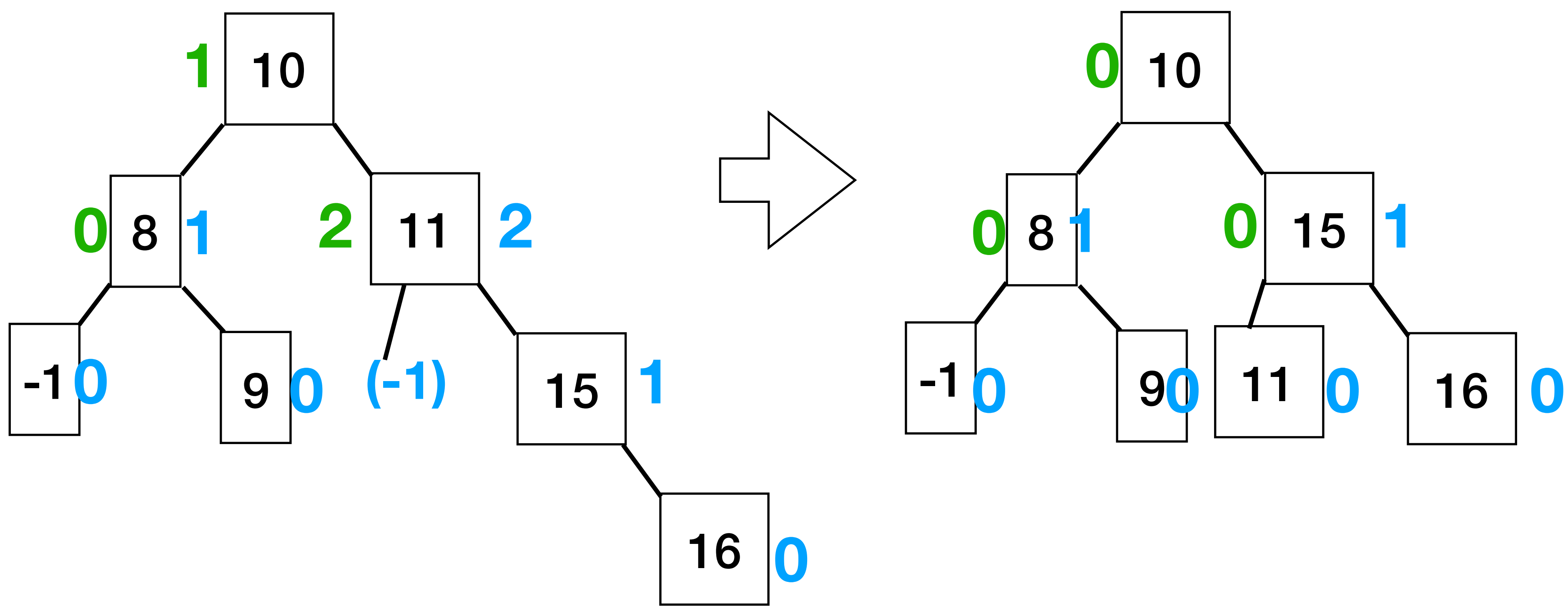


ABCD: The largest (absolute) balance factor in the tree is:

A. 0 before, 1 after rotation
B. 1 before, 0 after rotation
C. 0 before, 2 after rotation
D. 2 before, 0 after rotation

Can we improve balance?

Balance(n): height(n.right) - height(n.left)



How do we know what to rotate and when?
If the tree changes, check for imbalance and fix it if found.

AVL Trees

Balance(n): height(n.right) - height(n.left)

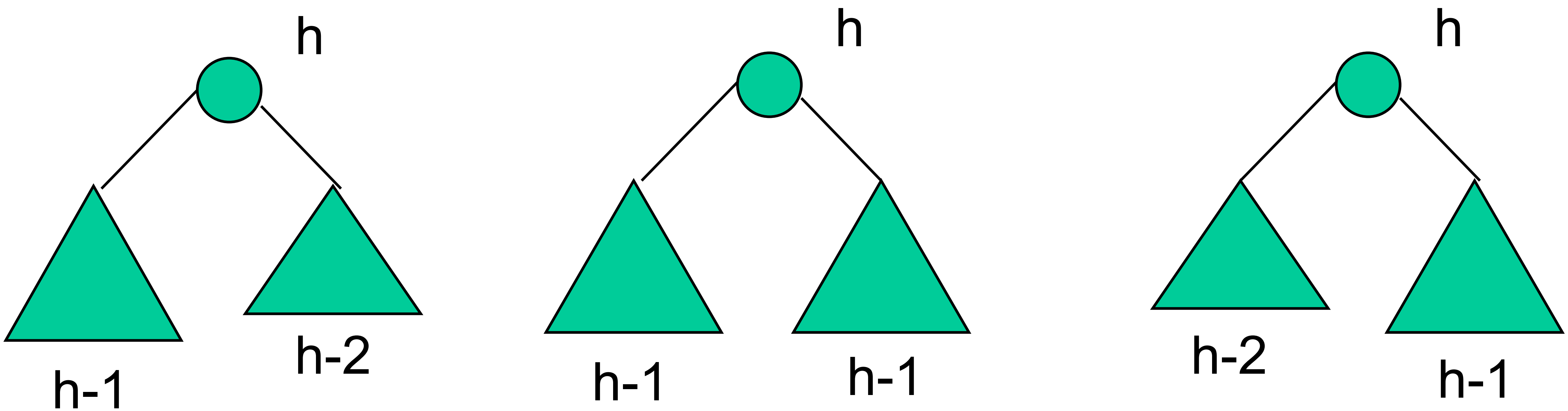
- Devised by **A**delson-**V**elsky and **L**andis
- An AVL tree is a Binary Search Tree in which the following property holds:

AVL property: -1 <= balance(n) <= 1 for all nodes n.

Balance Factor in AVL Trees

AVL property: -1 <= balance(n) <= 1 for all nodes n.

Every subtree in an AVL tree looks like one of these three trees:



(a) Balance factor: 1

(b) Balance factor: 0

(c) Balance factor: -1

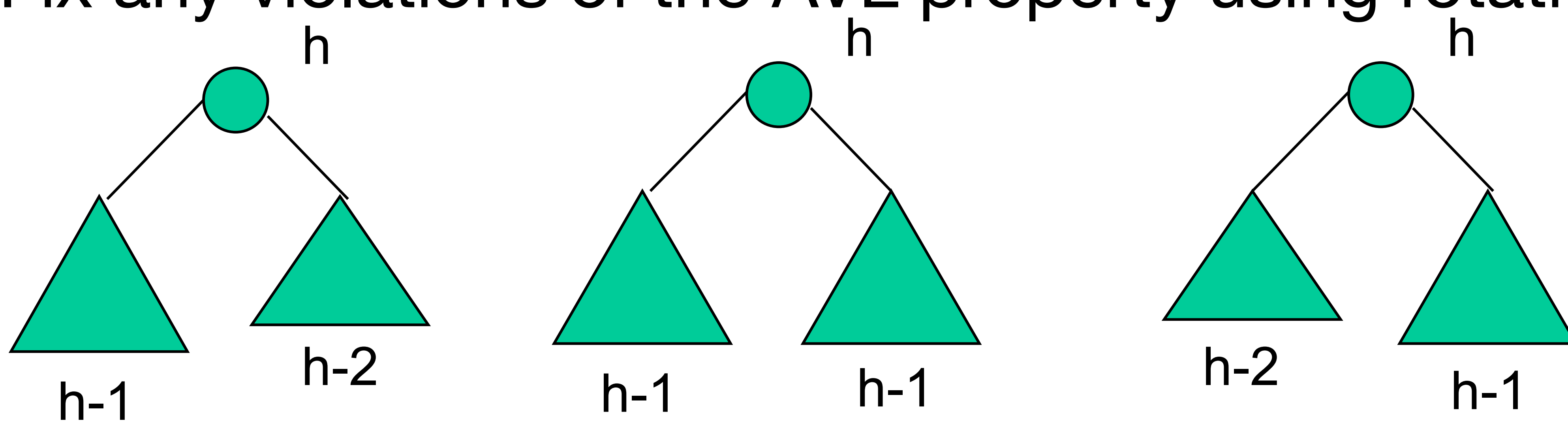
AVL Trees: Insertion

- An AVL tree is a Binary Search Tree in which the following property holds:

AVL property: -1 <= b(n) <= 1 for all nodes n.

To insert into an AVL tree:

1. Do a normal BST insertion
2. Fix any violations of the AVL property using rotations.



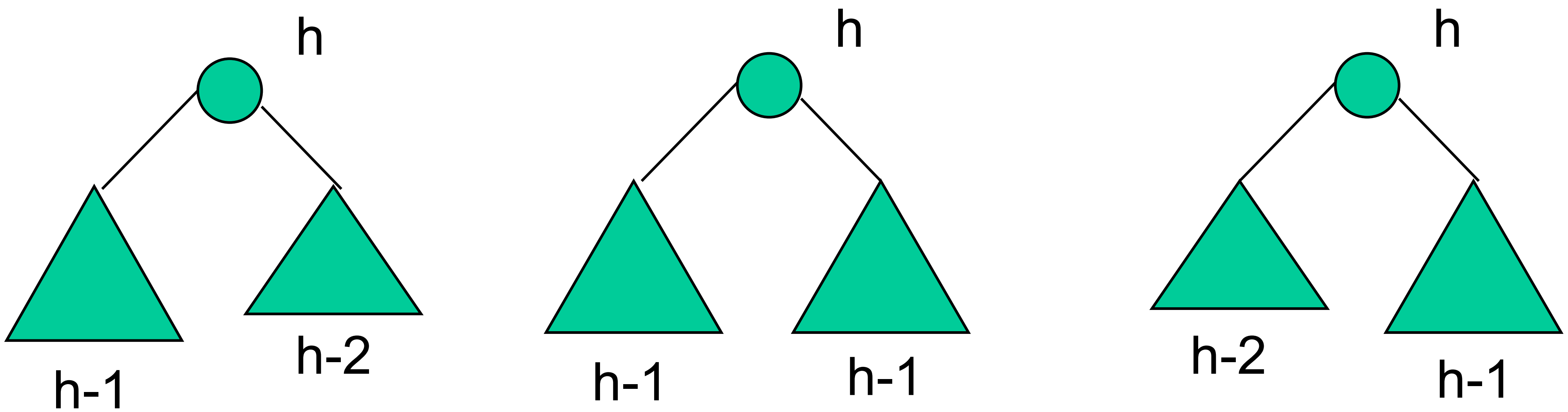
(a) Balance factor: 1 (b) Balance factor: 0 (c) Balance factor: -1

AVL Trees: Insertion

AVL property: -1 <= b(n) <= 1 for all nodes n.

To insert into an AVL tree:

1. Do a normal BST insertion
2. Fix any violations of the AVL property using rotations.



(a) Balance factor: 1 (b) Balance factor: 0 (c) Balance factor: -1

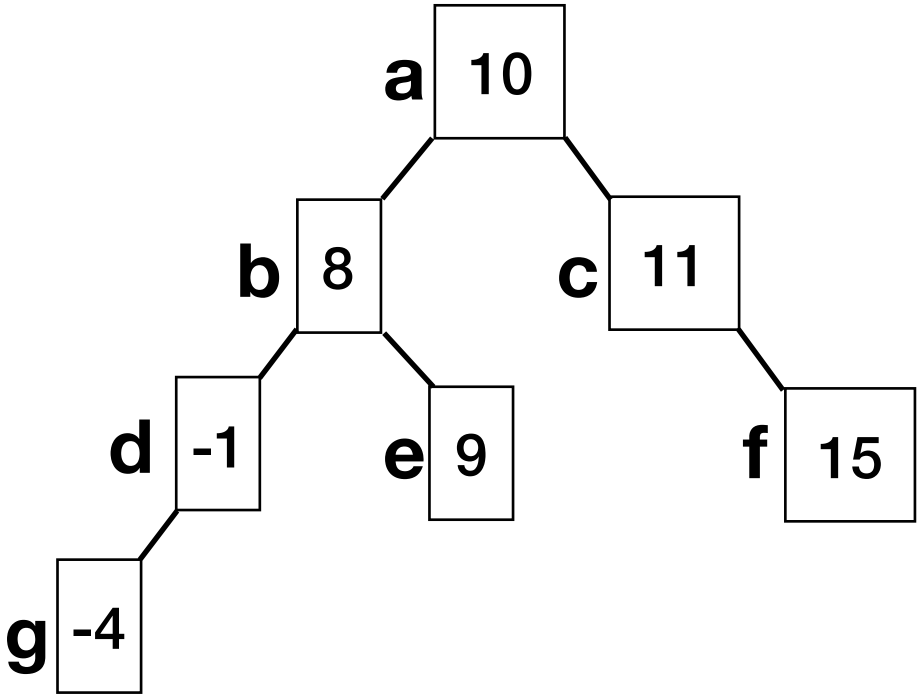
Refresher: BST Insertion

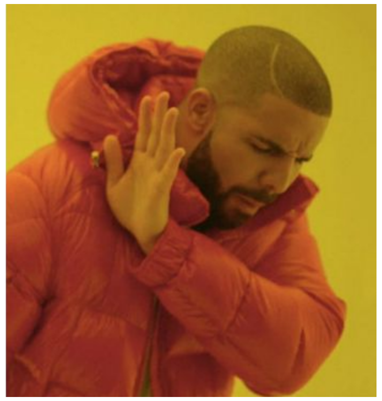
```
/* insert a node with value v into the
 * tree rooted at n. pre: n is not null. */
insert(Node n, int v):
  if n.value == v: return // (duplicate)
  if v < n.value:
    if n has left:
      insert(n.left, v)
    else:
      // attach new node w/ value v to n.left
  else: // v > n.value
    if n has right:
      insert(n.right, v)
    else:
      // attach new node w/ value v to n.right
```

AVL Insertion

```
/* insert a node with value v into the
 * tree rooted at n. pre: n is not null. */
insert(Node n, int v):
  if n.value == v: return // (duplicate)
  if v < n.value:
    if n has left:
      insert(n.left, v)
    else:
      // attach new node w/ value v to n.left
  else: // v > n.value
    if n has right:
      insert(n.right, v)
    else:
      // attach new node w/ value v to n.right
  rebalance(n);
```

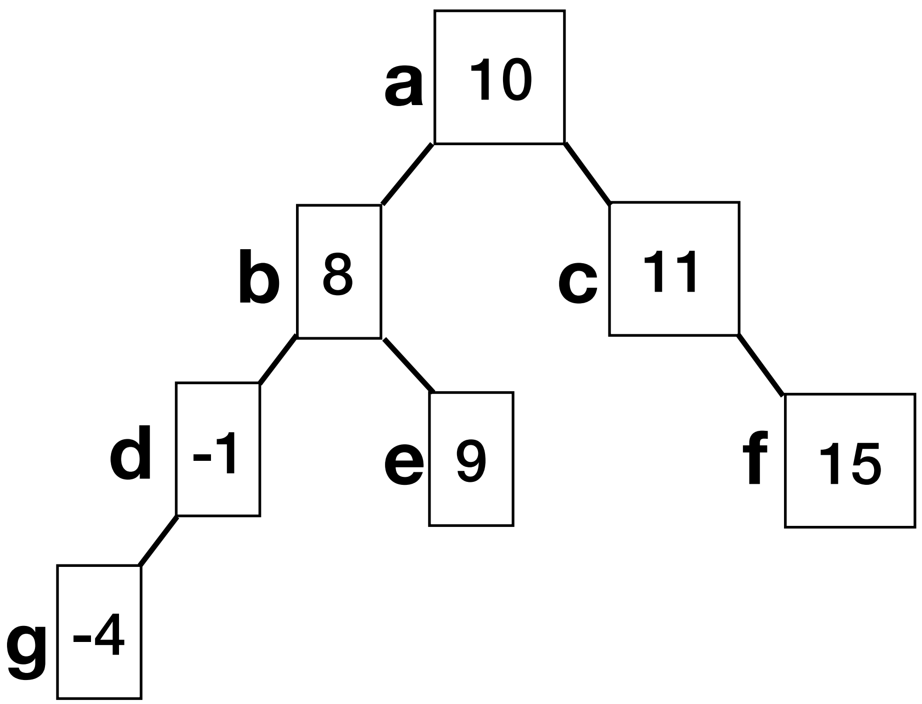
AVL Insertion



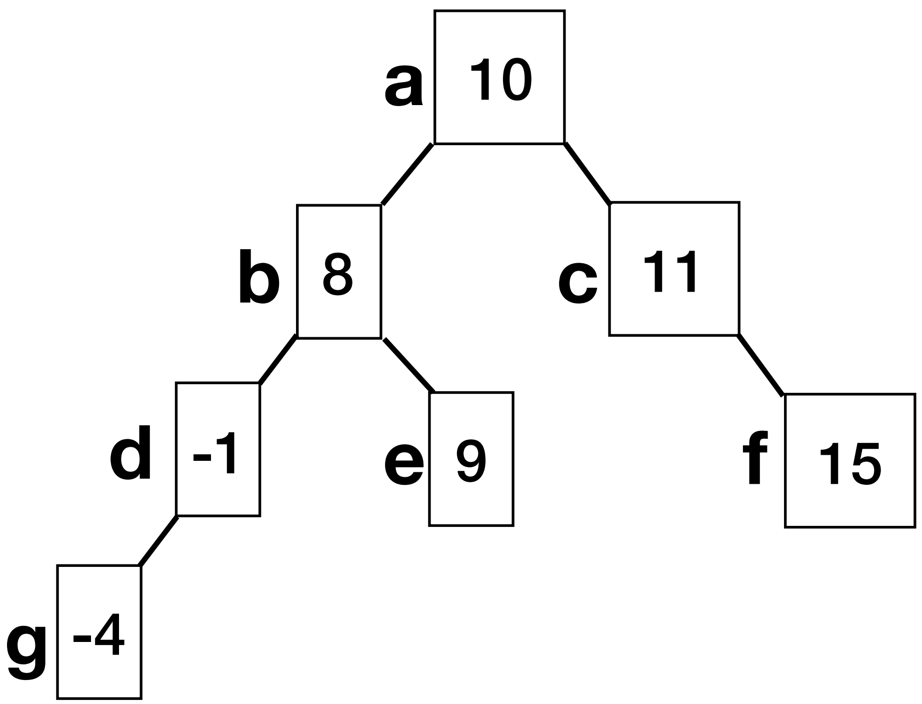


AVL Insertion

First: is this an AVL tree?



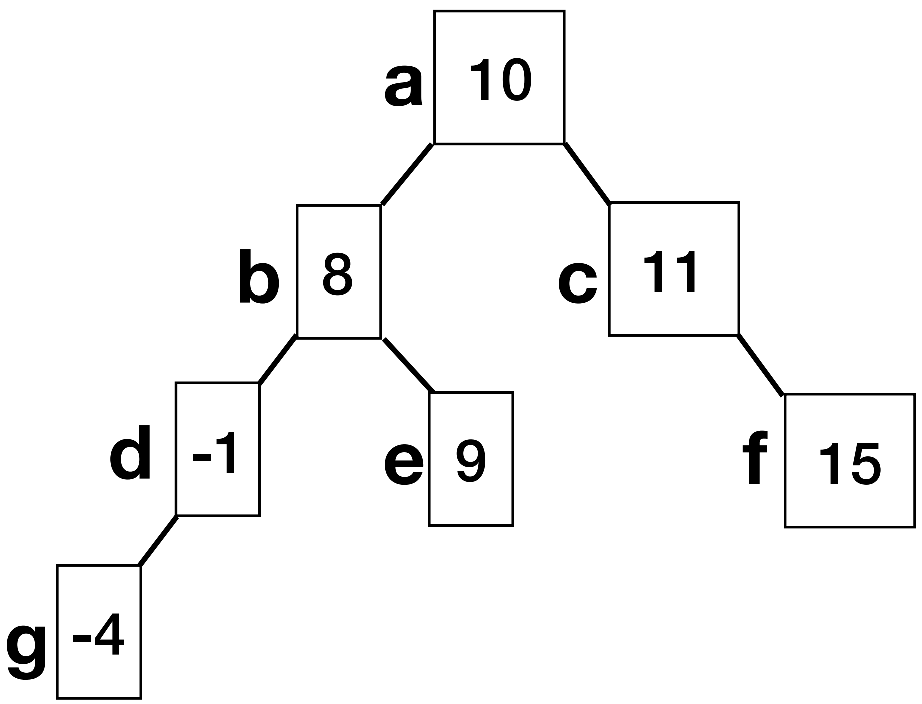
AVL Insertion




```
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)
```

AVL Insertion



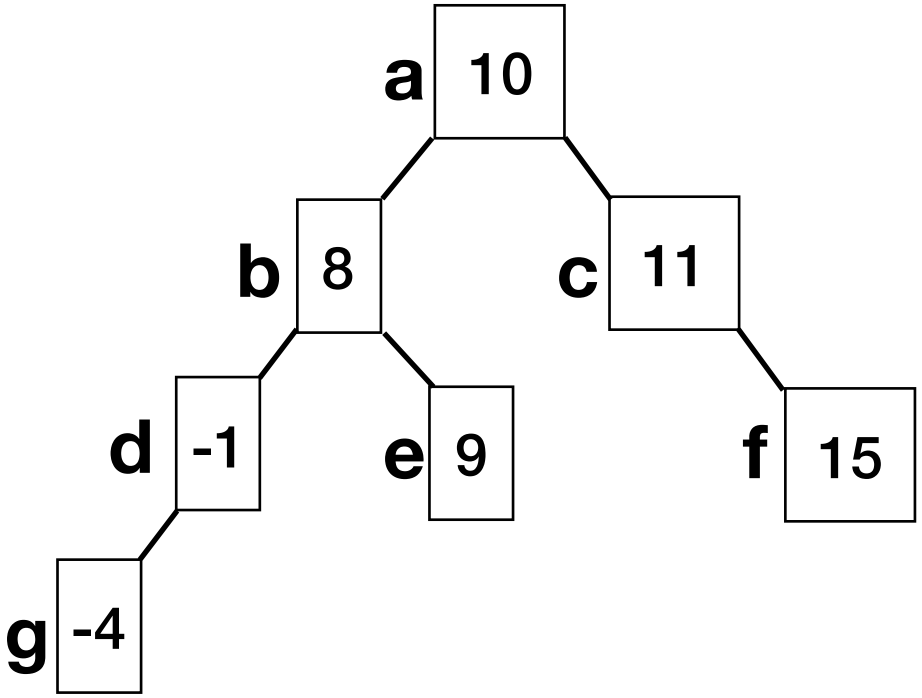

```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)




  rebalance(a)
```

AVL Insertion



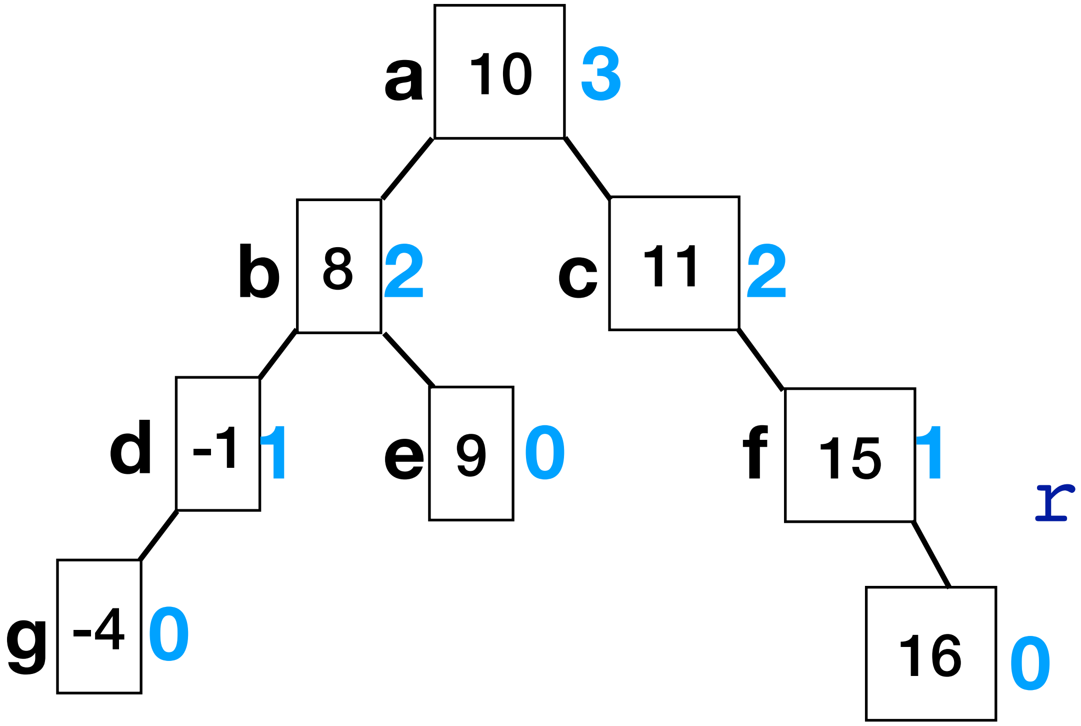

```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)



    rebalance(c)
  rebalance(a)
```

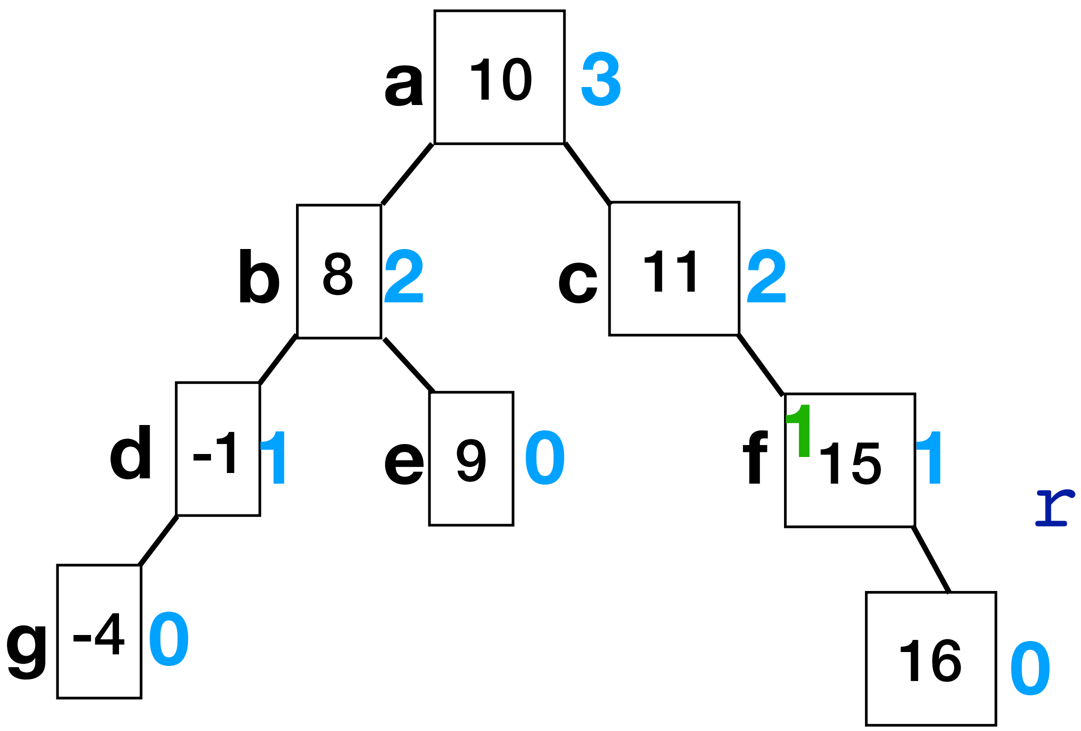
AVL Insertion




```
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)
  rebalance(a)
```

AVL Insertion

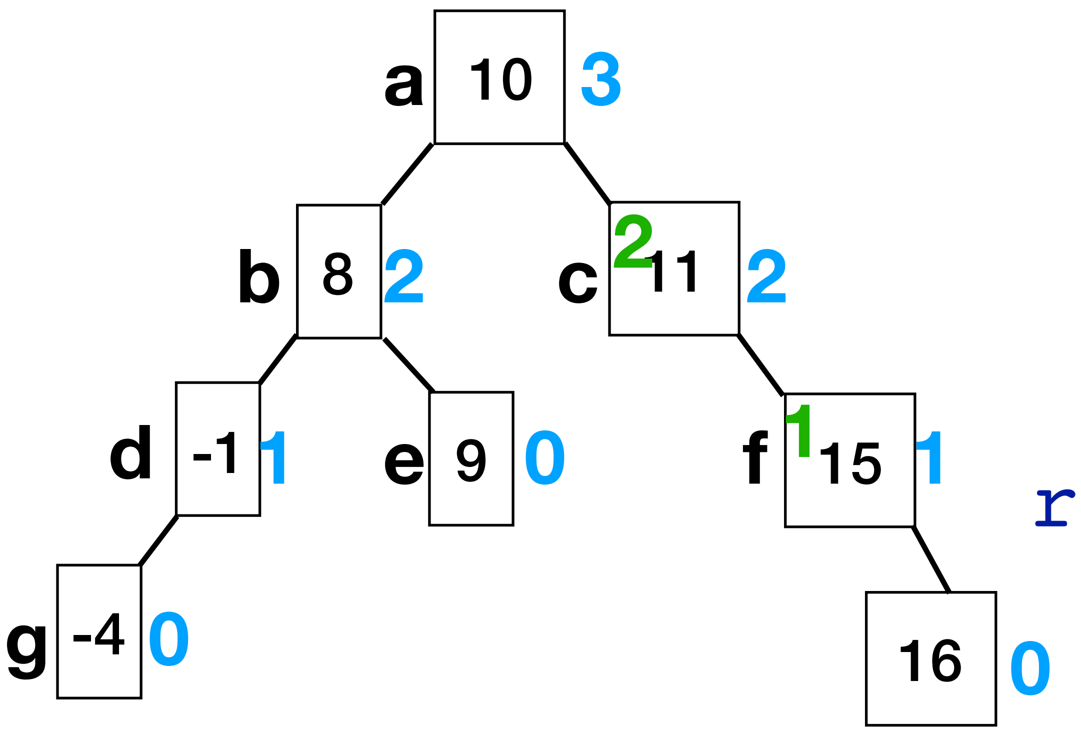



```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

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

AVL Insertion

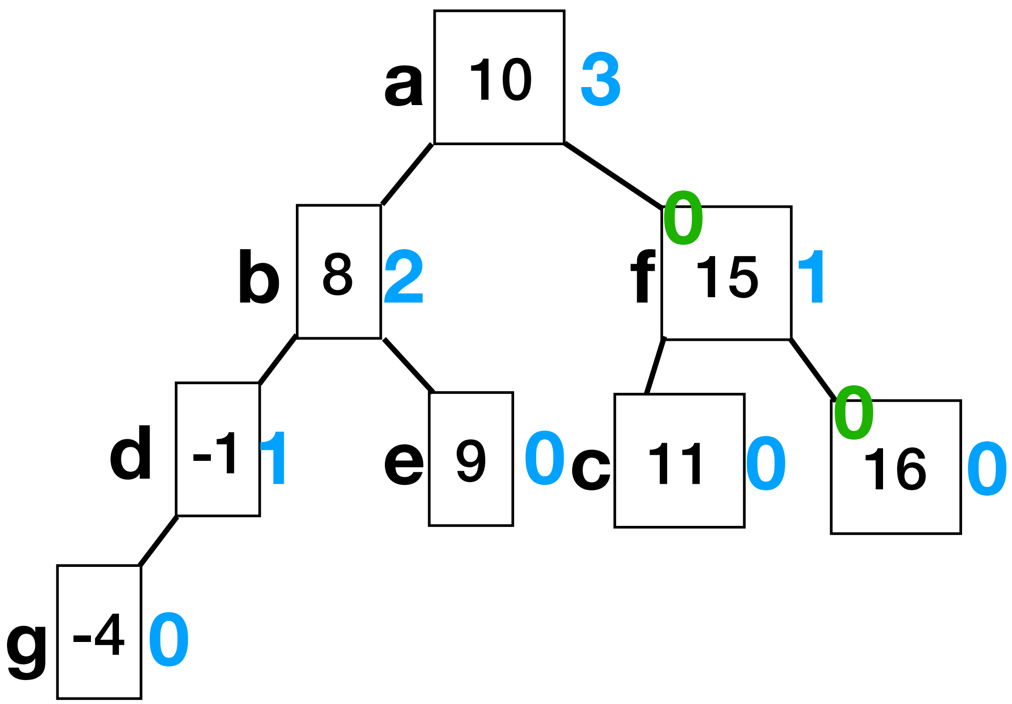
```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```




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

AVL Insertion

```
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);
```



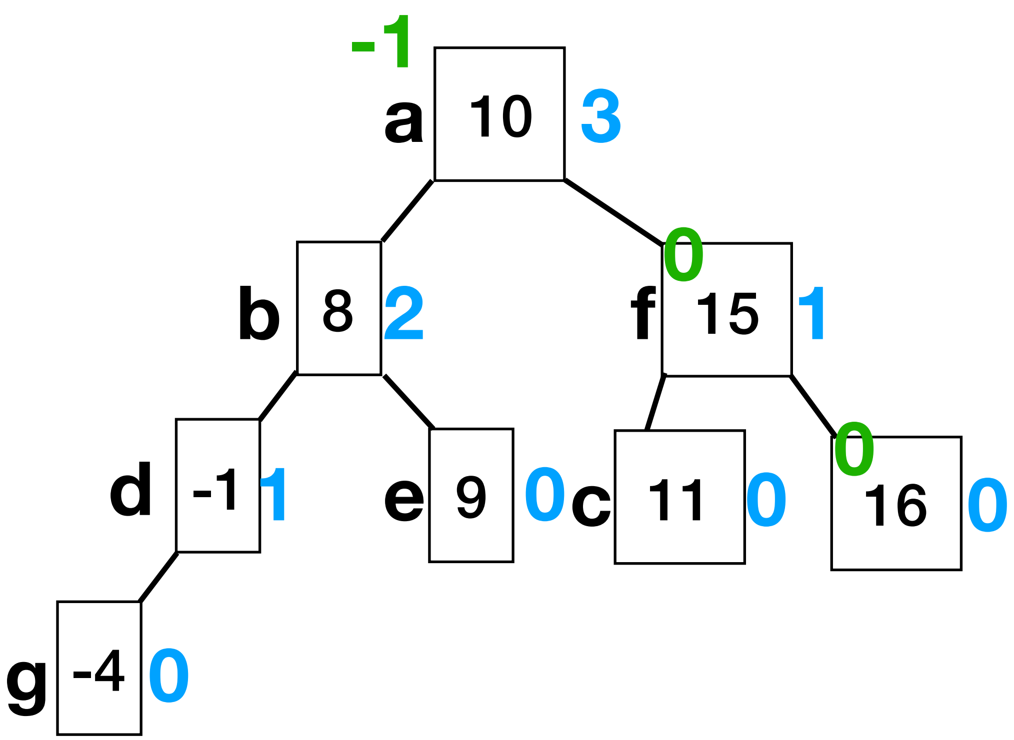

```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f)
    rebalance(c)
  rebalance(a)
```

already balanced

perform rotation

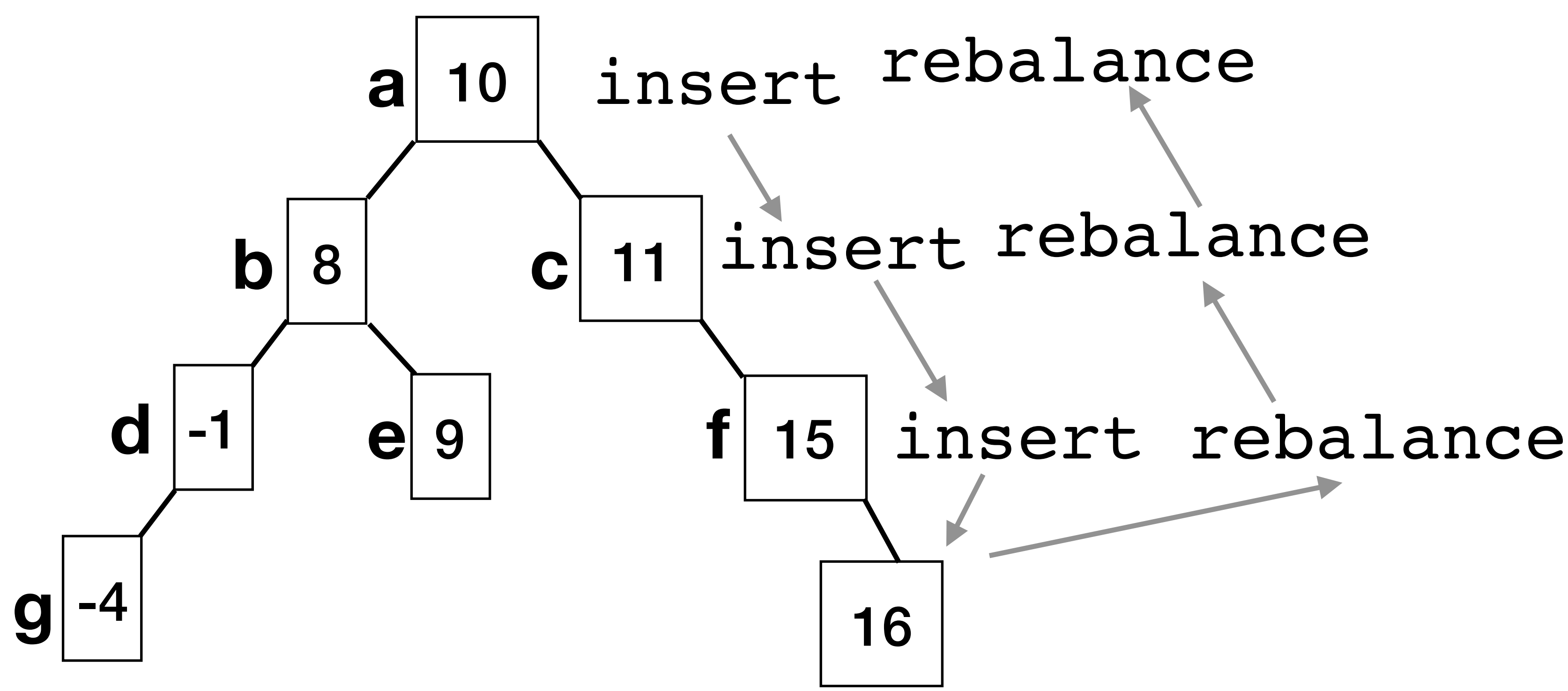
AVL Insertion

```
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);
```

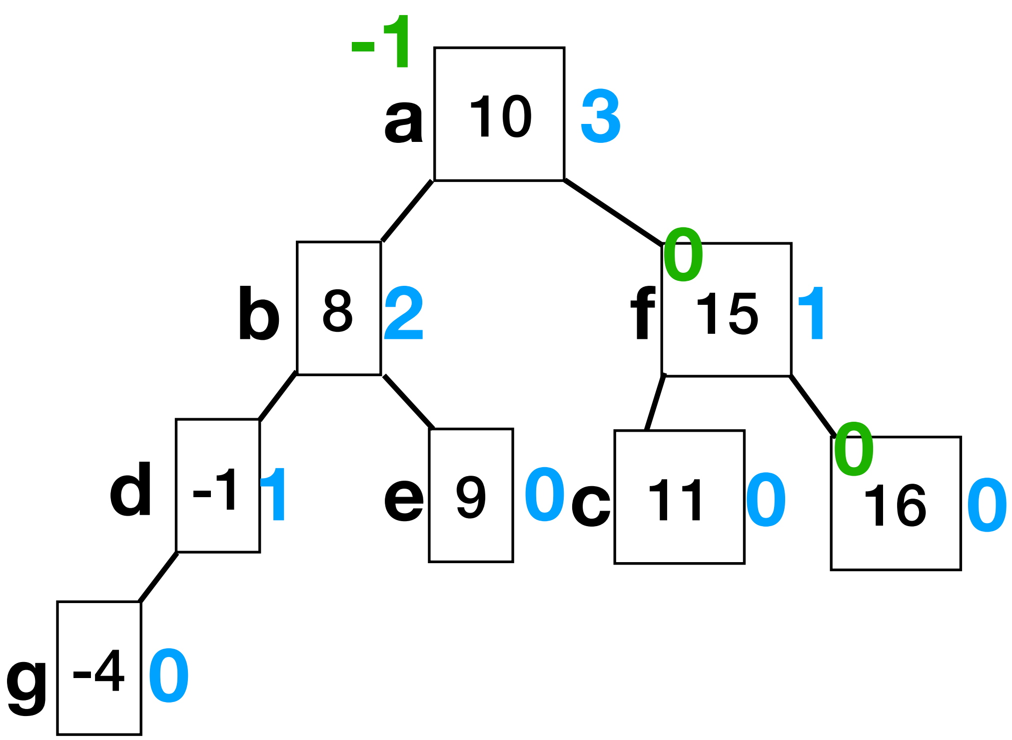



```
insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
      rebalance(f) already balanced
    rebalance(c) perform rotation
  rebalance(a) already balanced
```

Order of actual execution



AVL Insertion

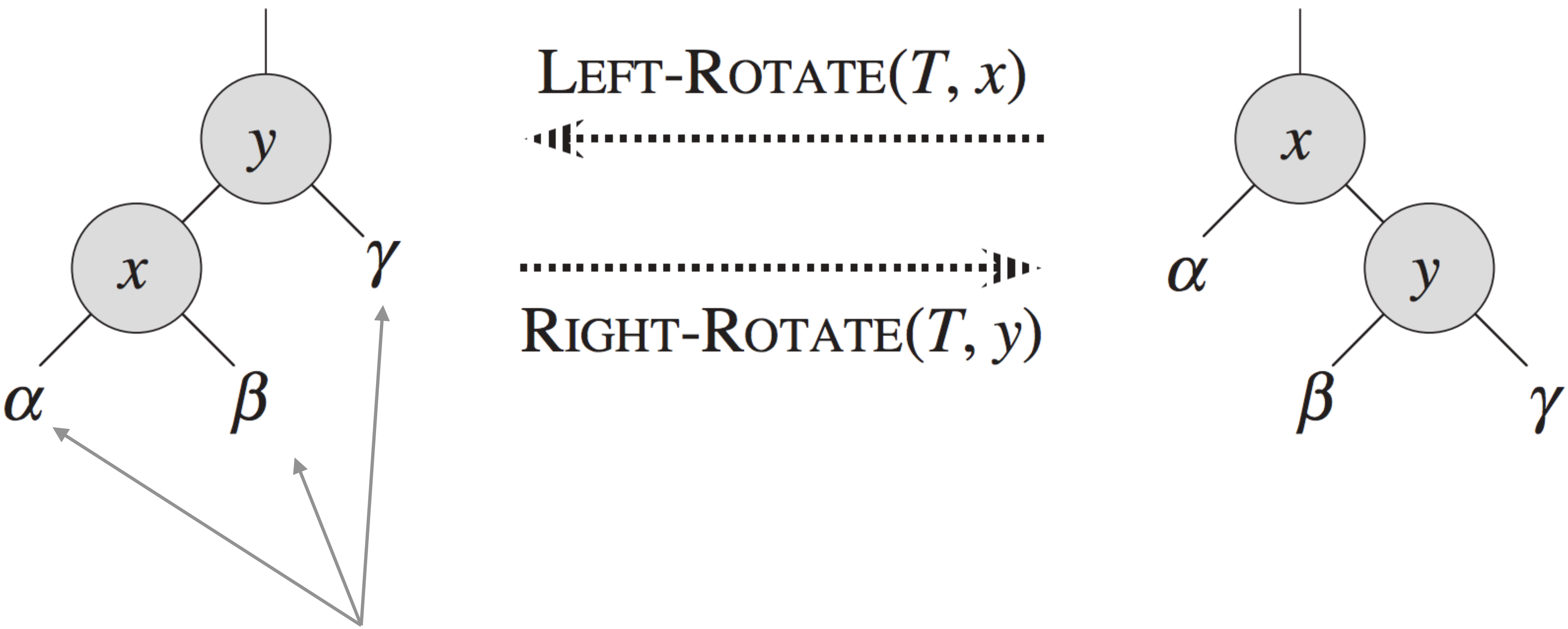



```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
rebalance(n);
```

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

How did we know what rotation to do?

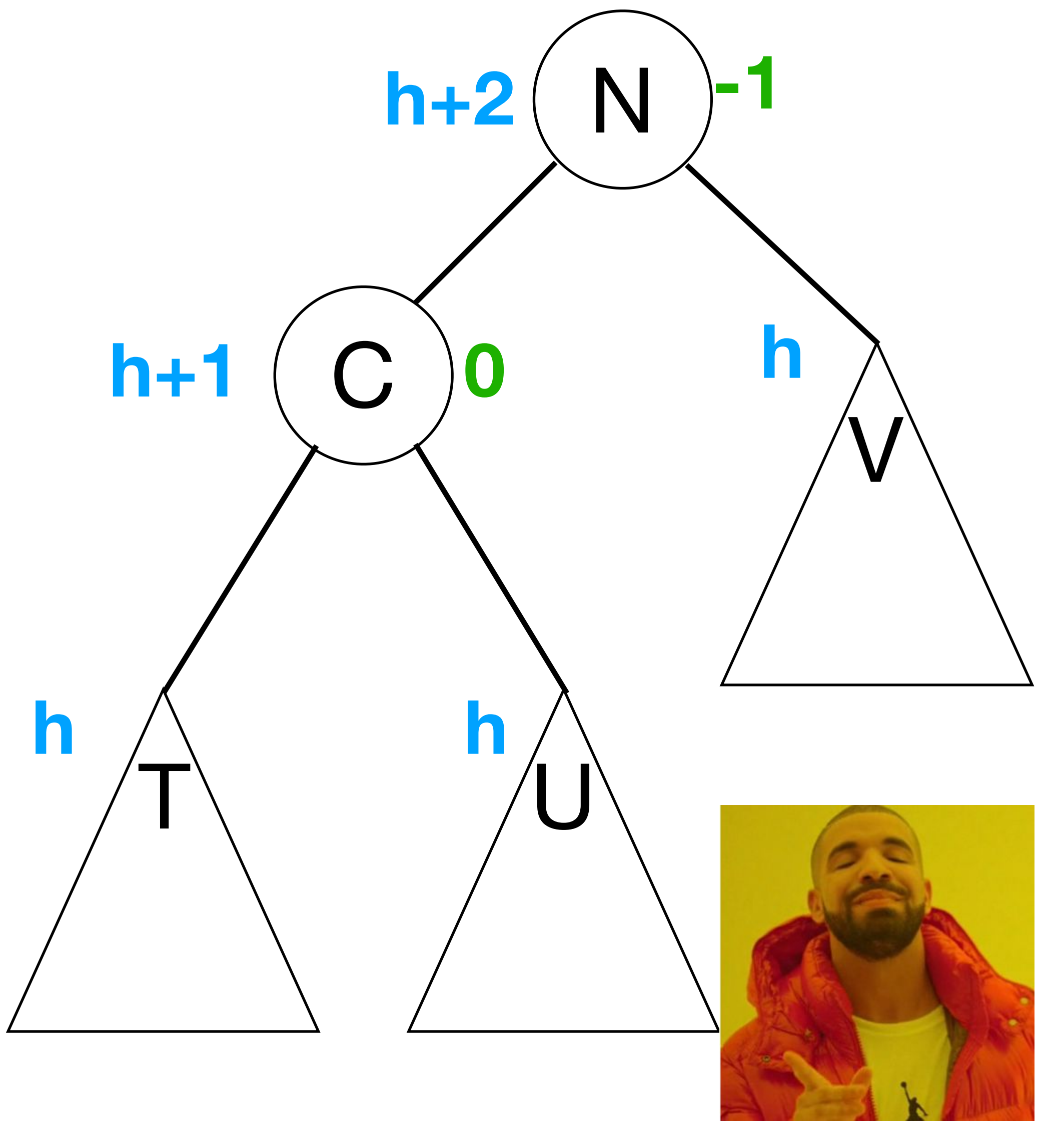
Reminder: Tree Rotations

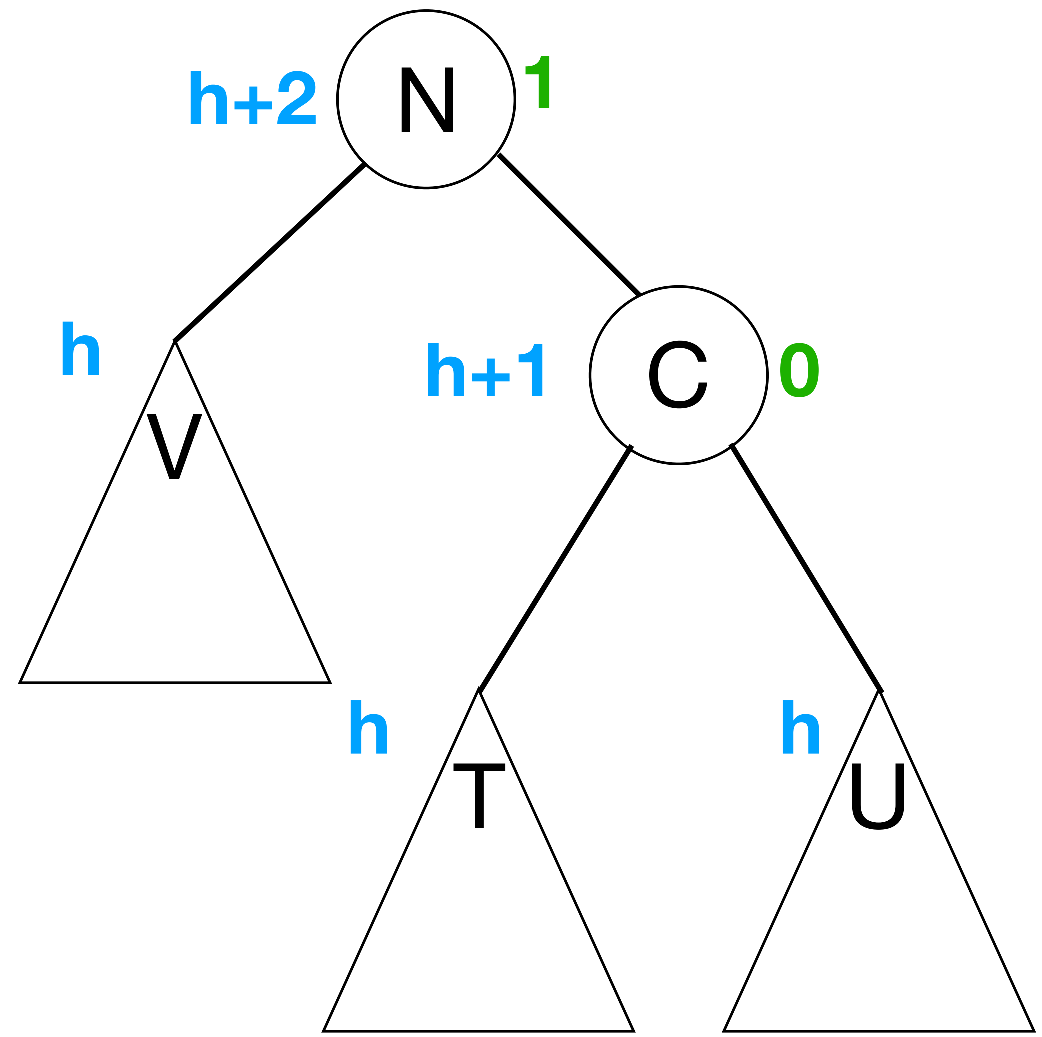


subtrees (could be null, leaf, or tree with many nodes)

AVL Rebalance

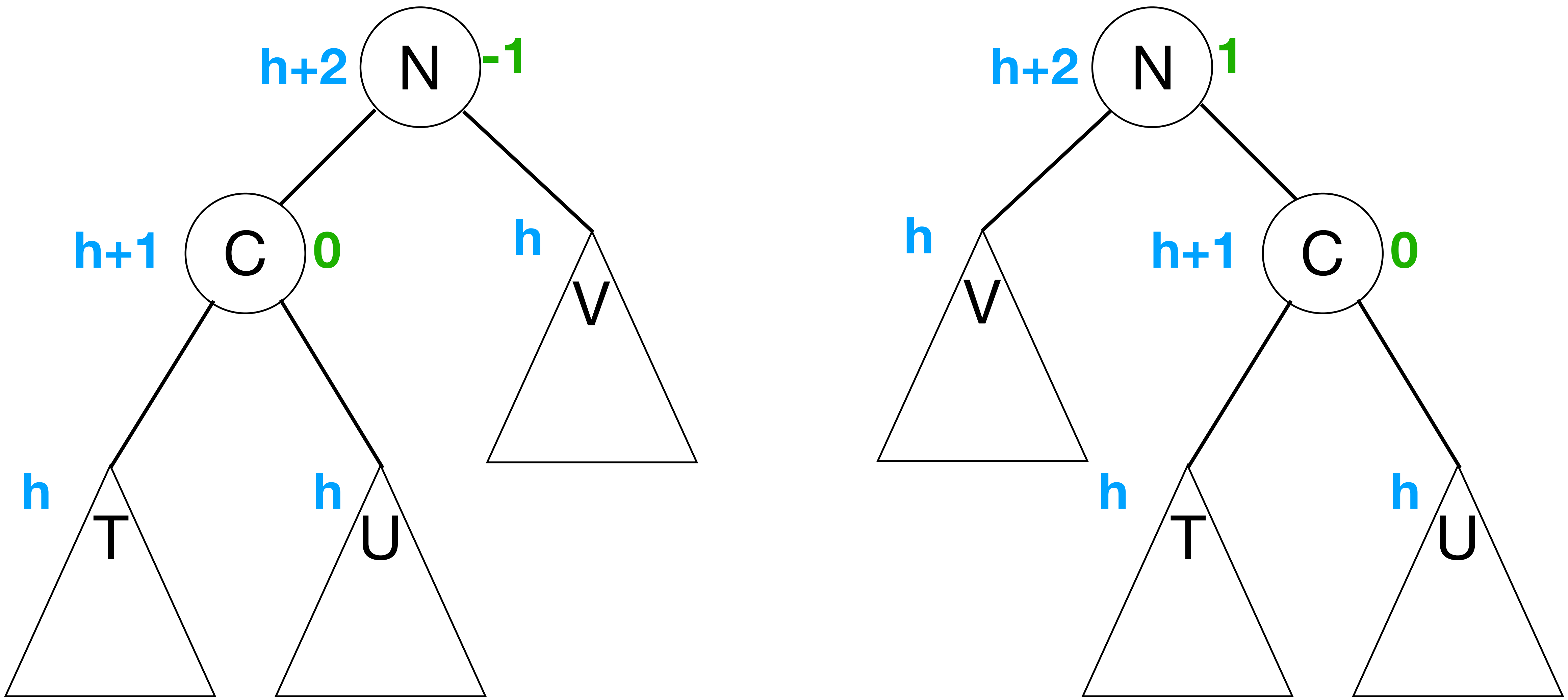
Before an insertion that unbalances n,
the tree must look like one of these:





AVL Rebalance

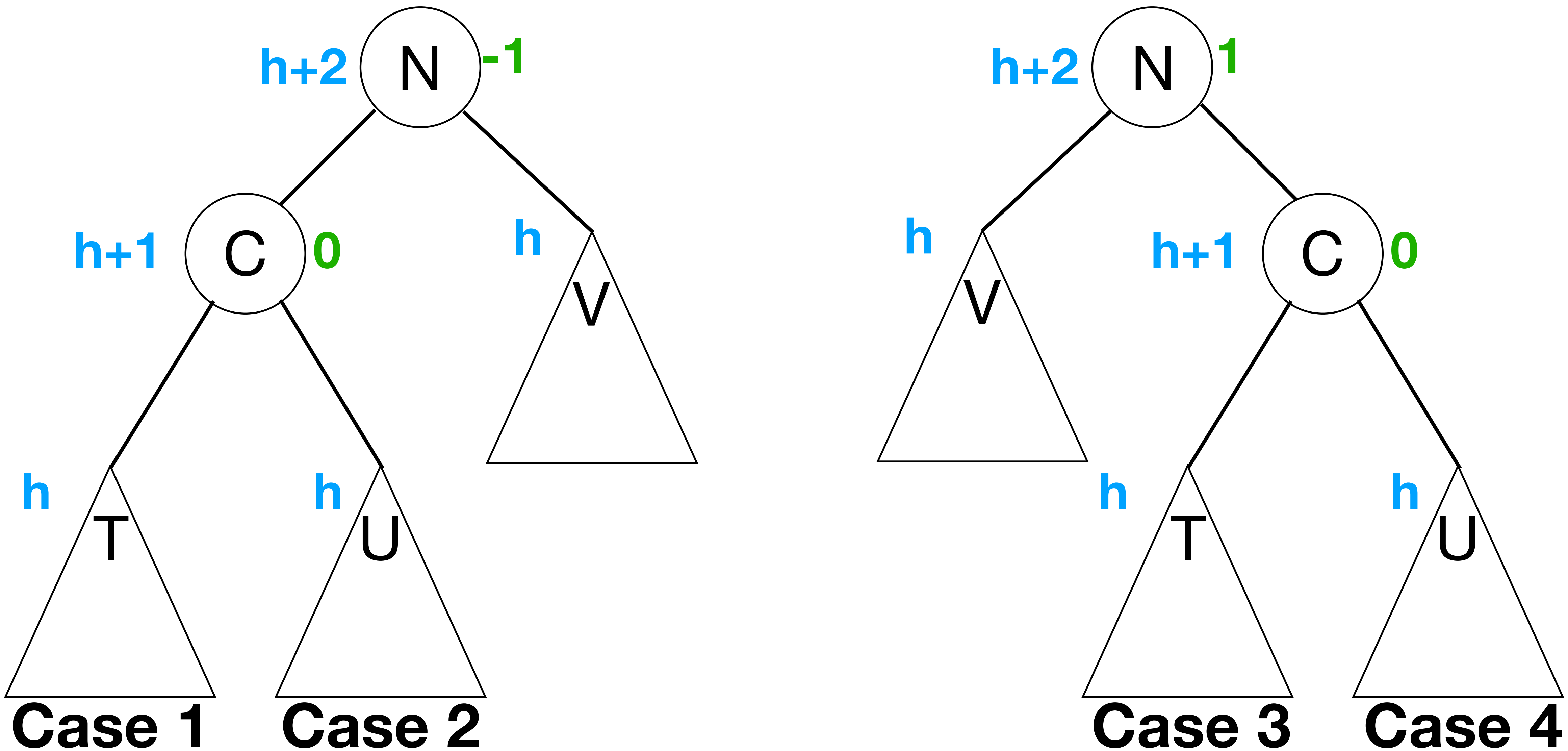
Before an insertion that unbalances n,
the tree must look like one of these:



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

AVL Rebalance

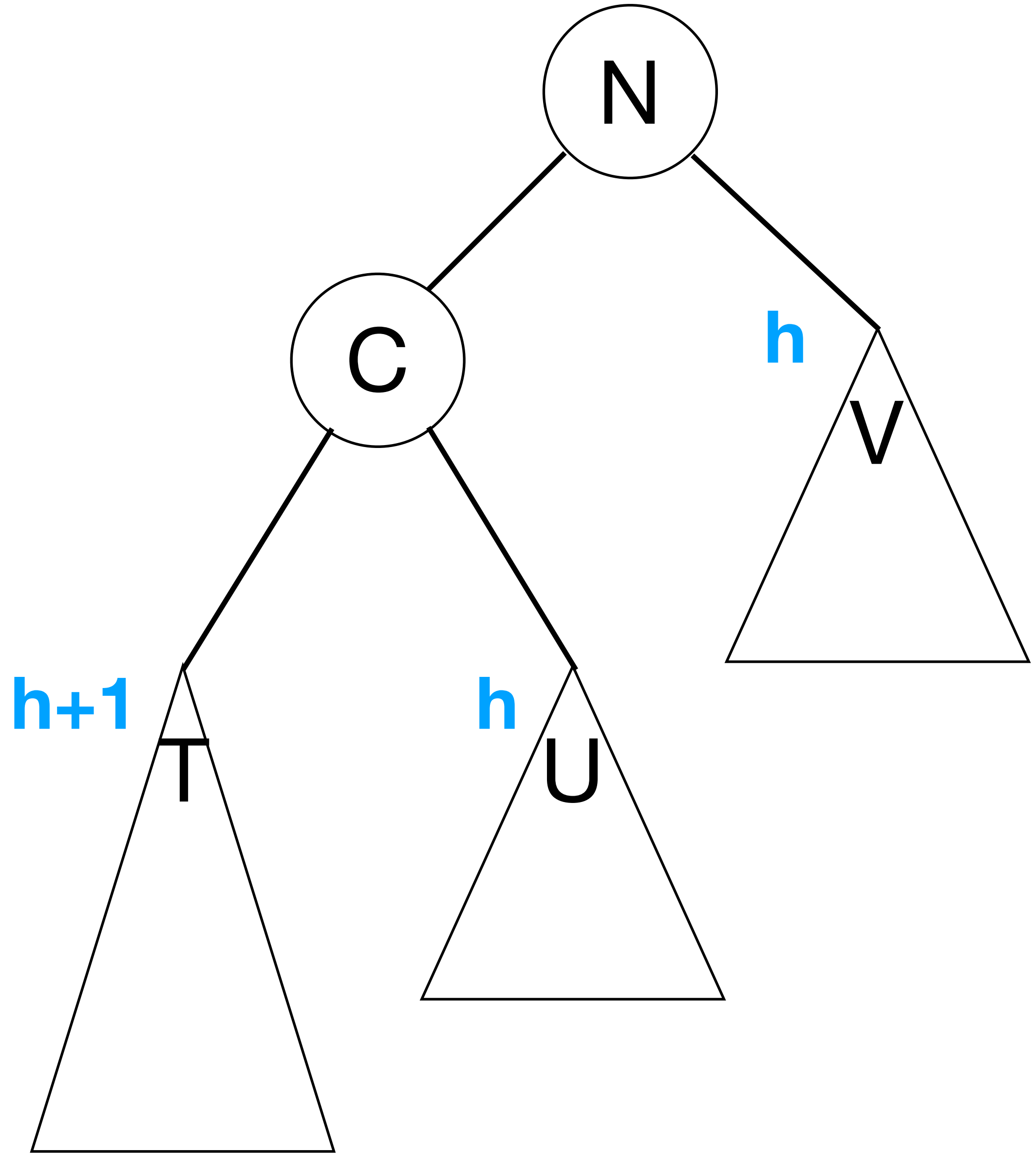
Before an insertion that unbalances n,
the tree must look like one of these:



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

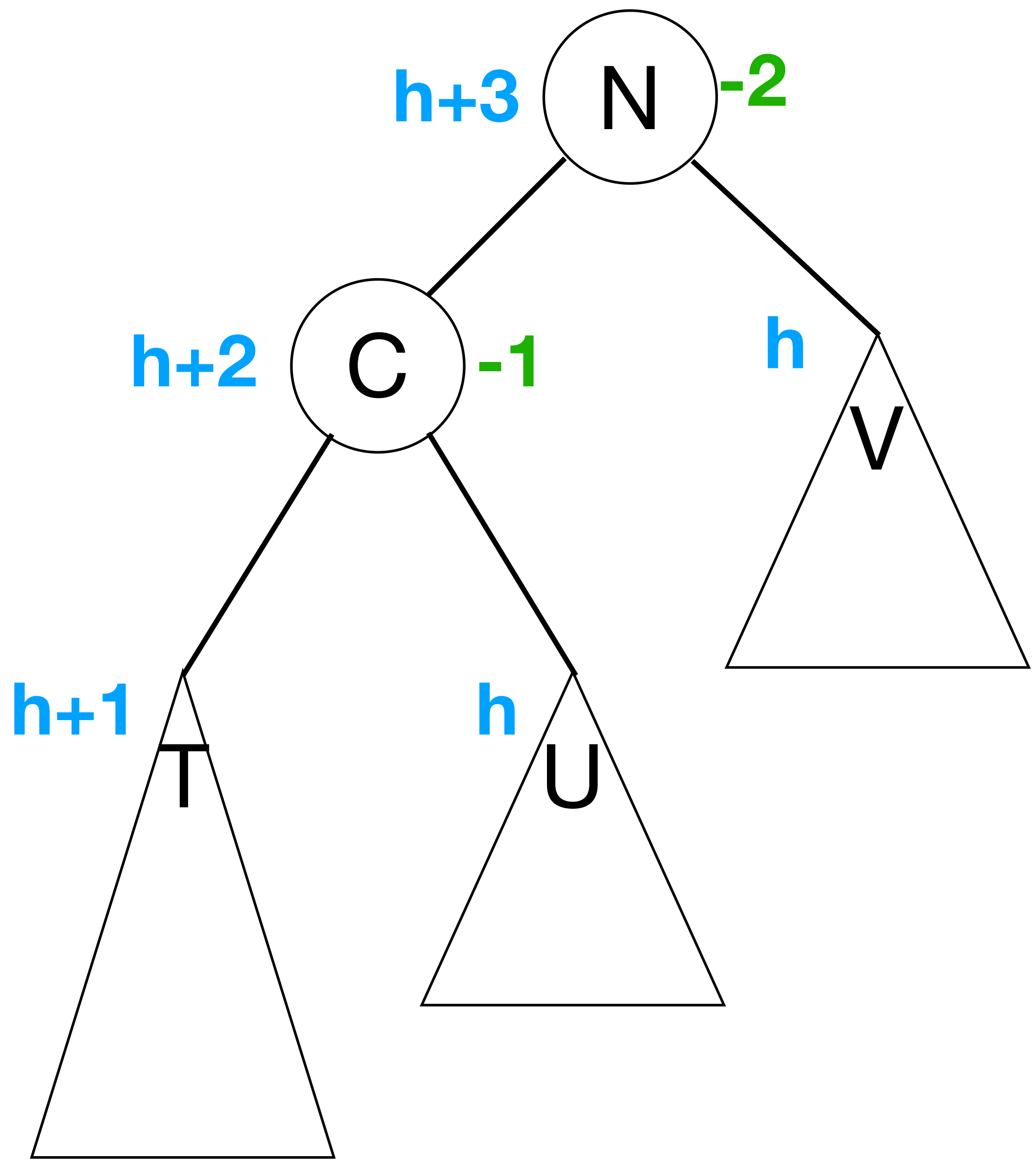
AVL Rebalance

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



AVL Rebalance

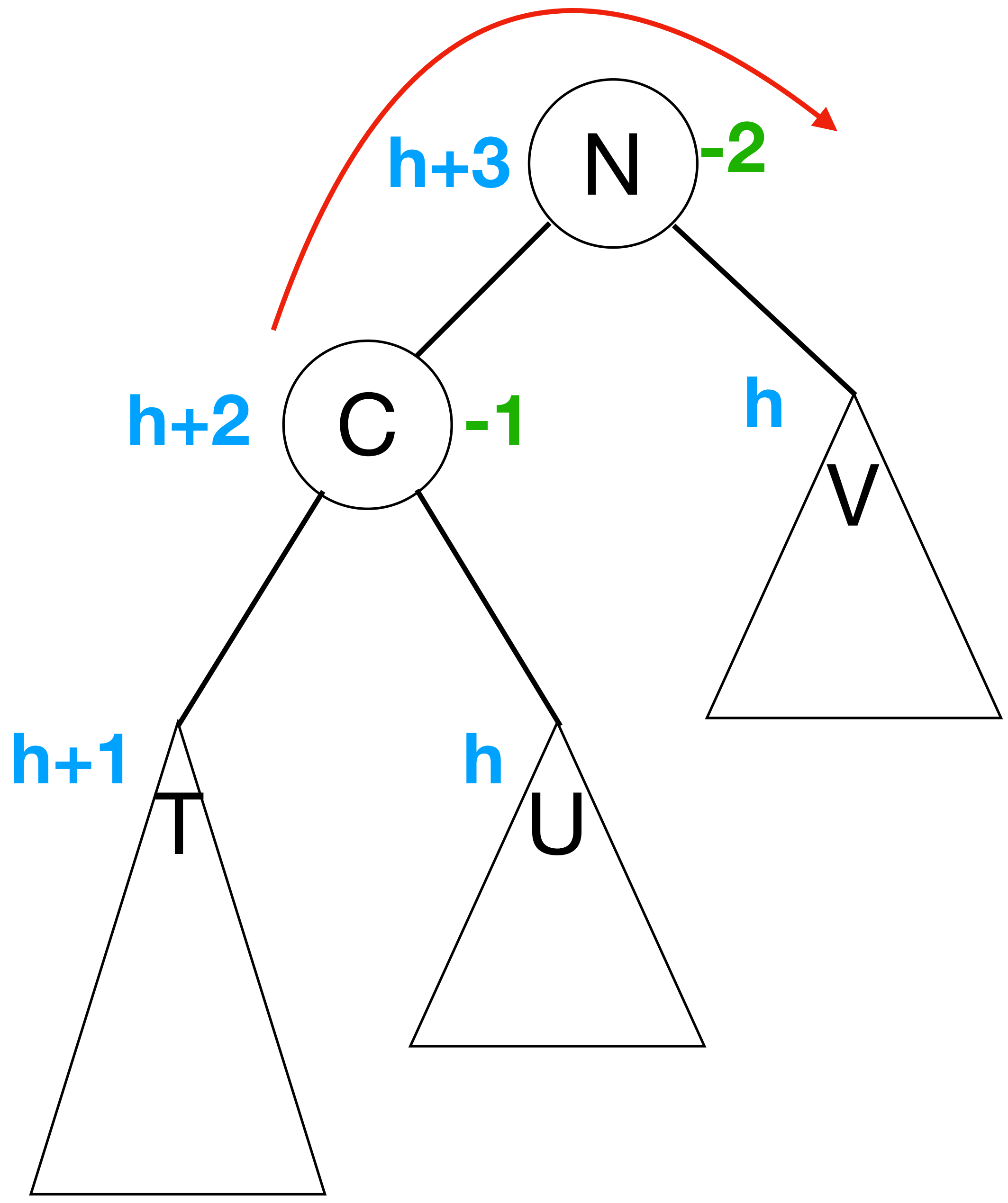
Case 1: After BST insertion step, the tree looks like this.



AVL Rebalance

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

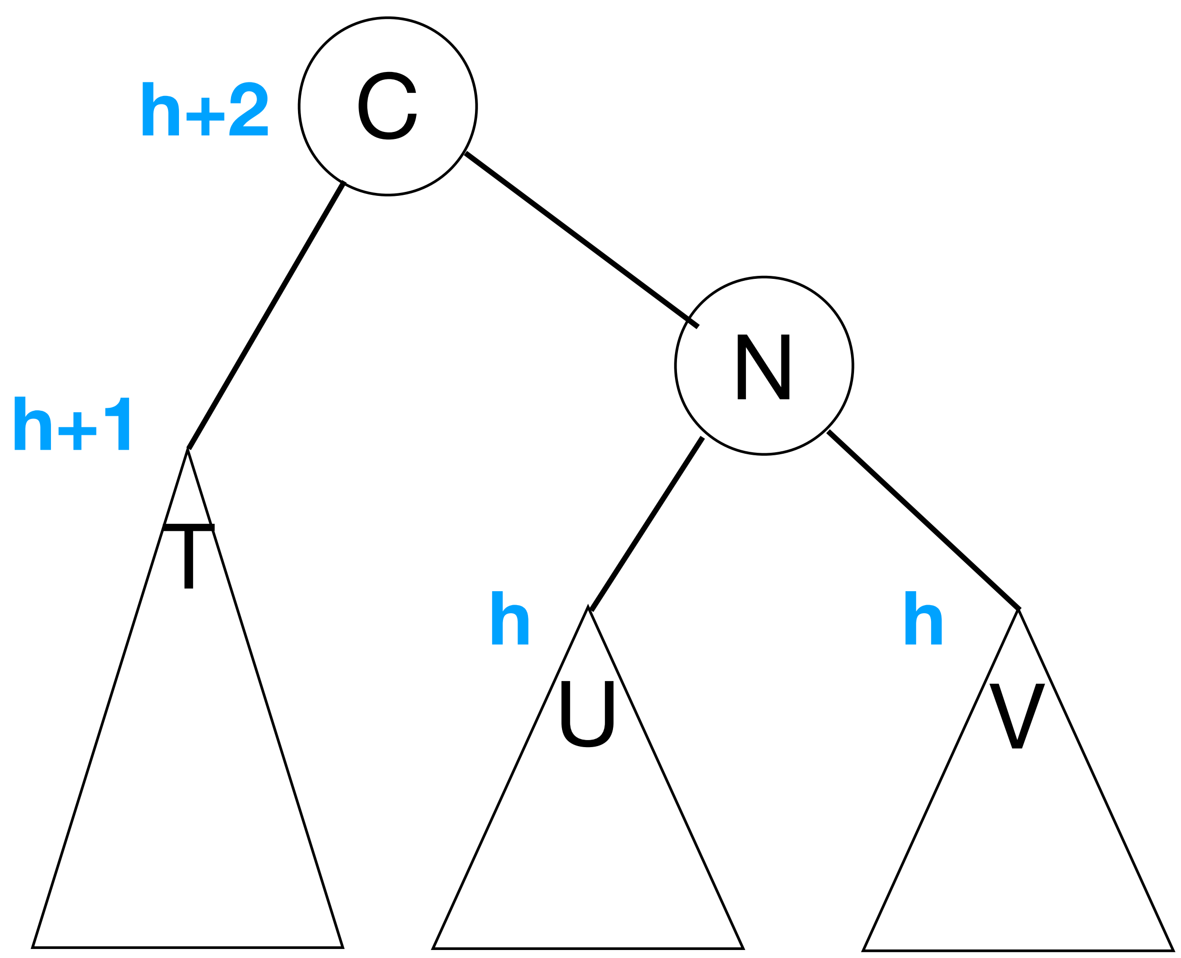
Solution: right rotate on N.



AVL Rebalance

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

Solution: right rotate on N.

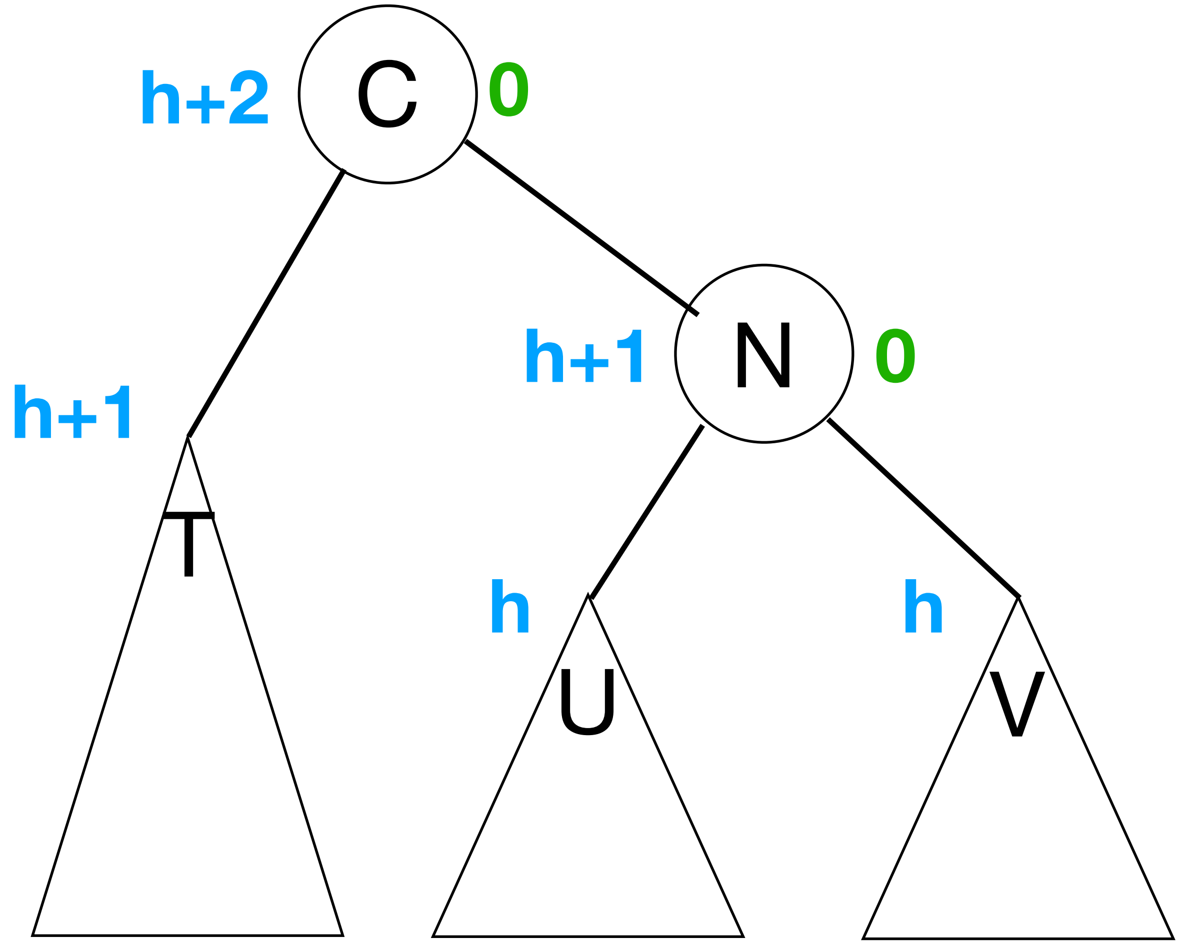


AVL Rebalance

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

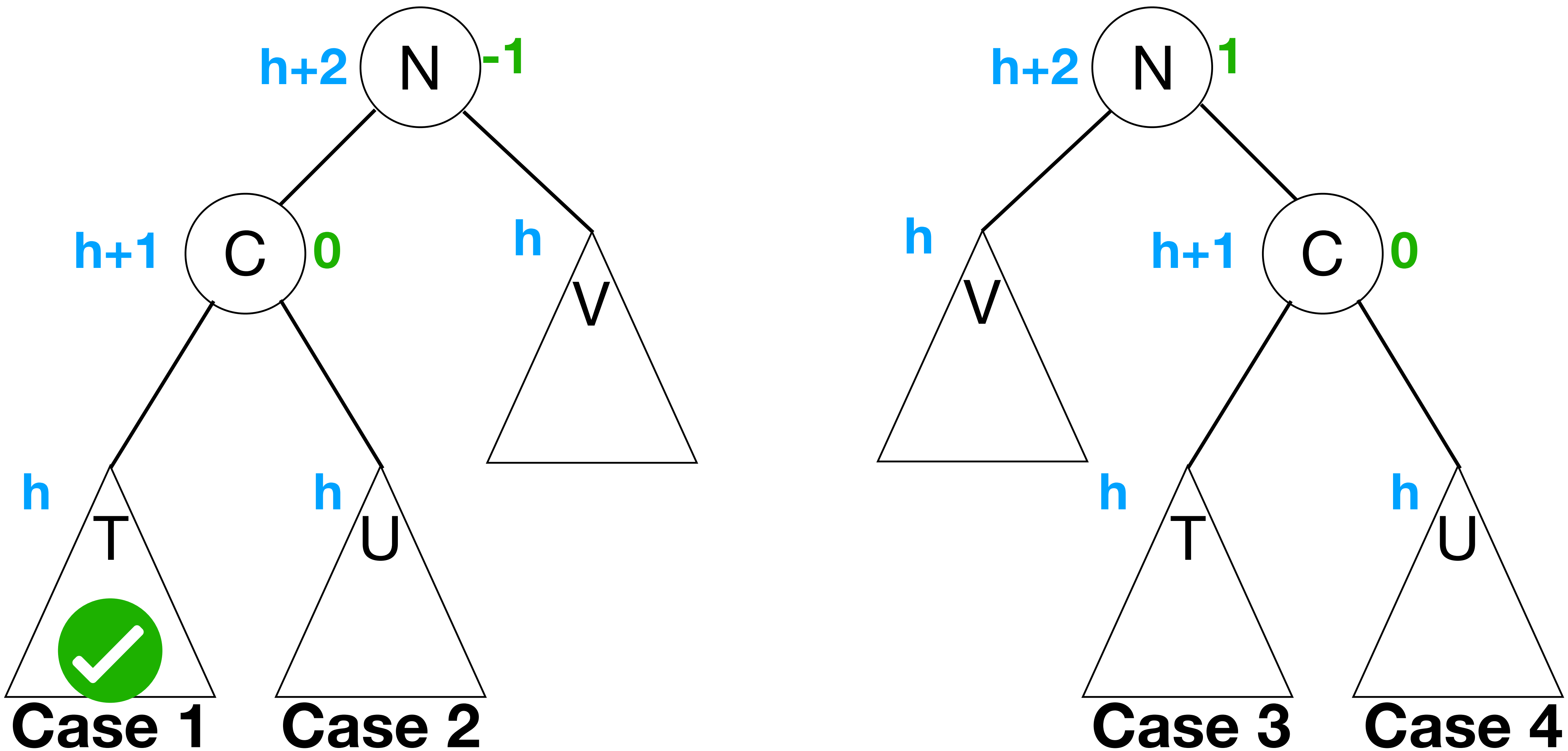
Solution: right rotate on N.

N is now AVL balanced.



AVL Rebalance

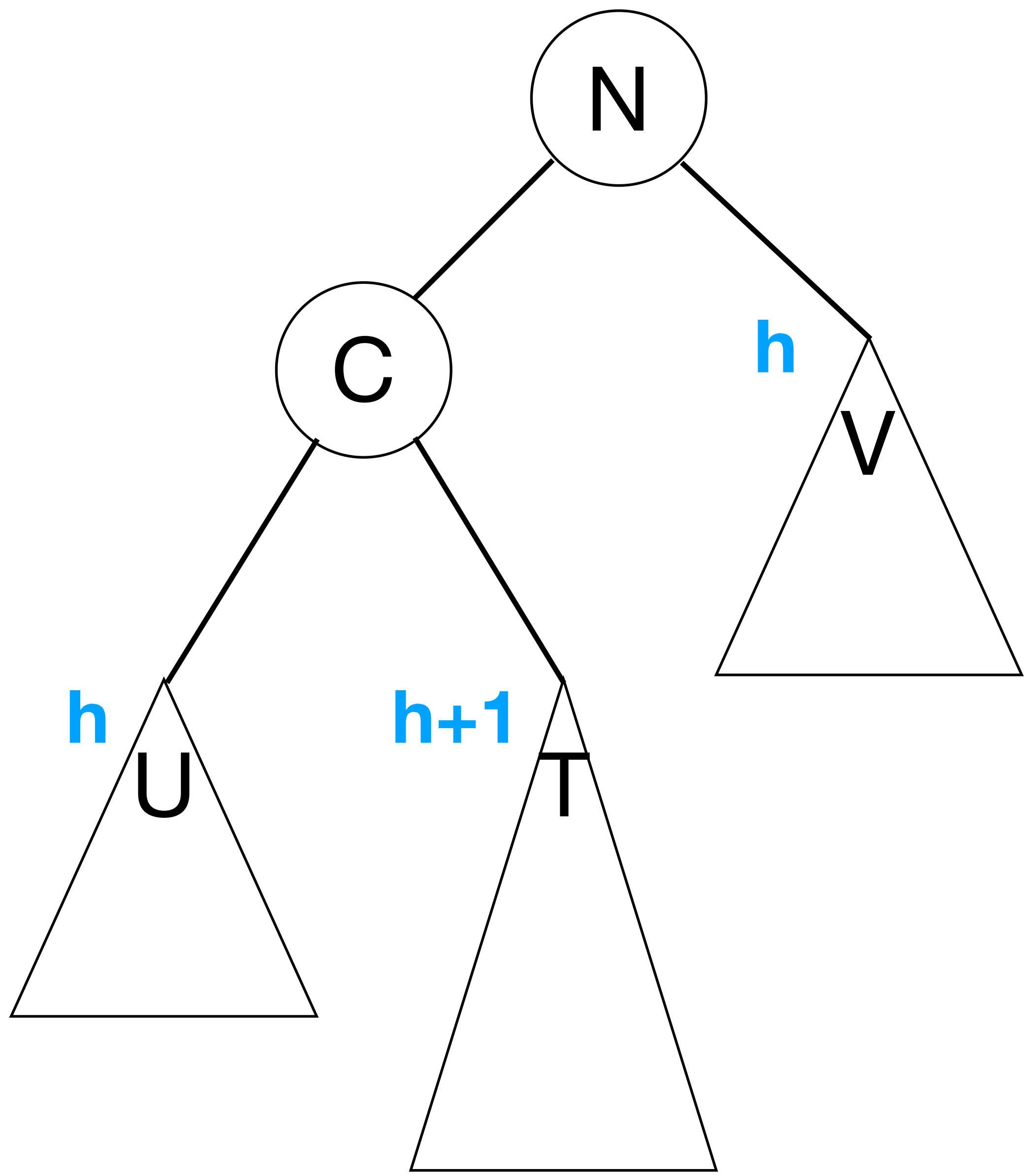
Before an insertion that unbalances n,
the tree must look like one of these:



An insertion that unbalances n could gone one of four places.

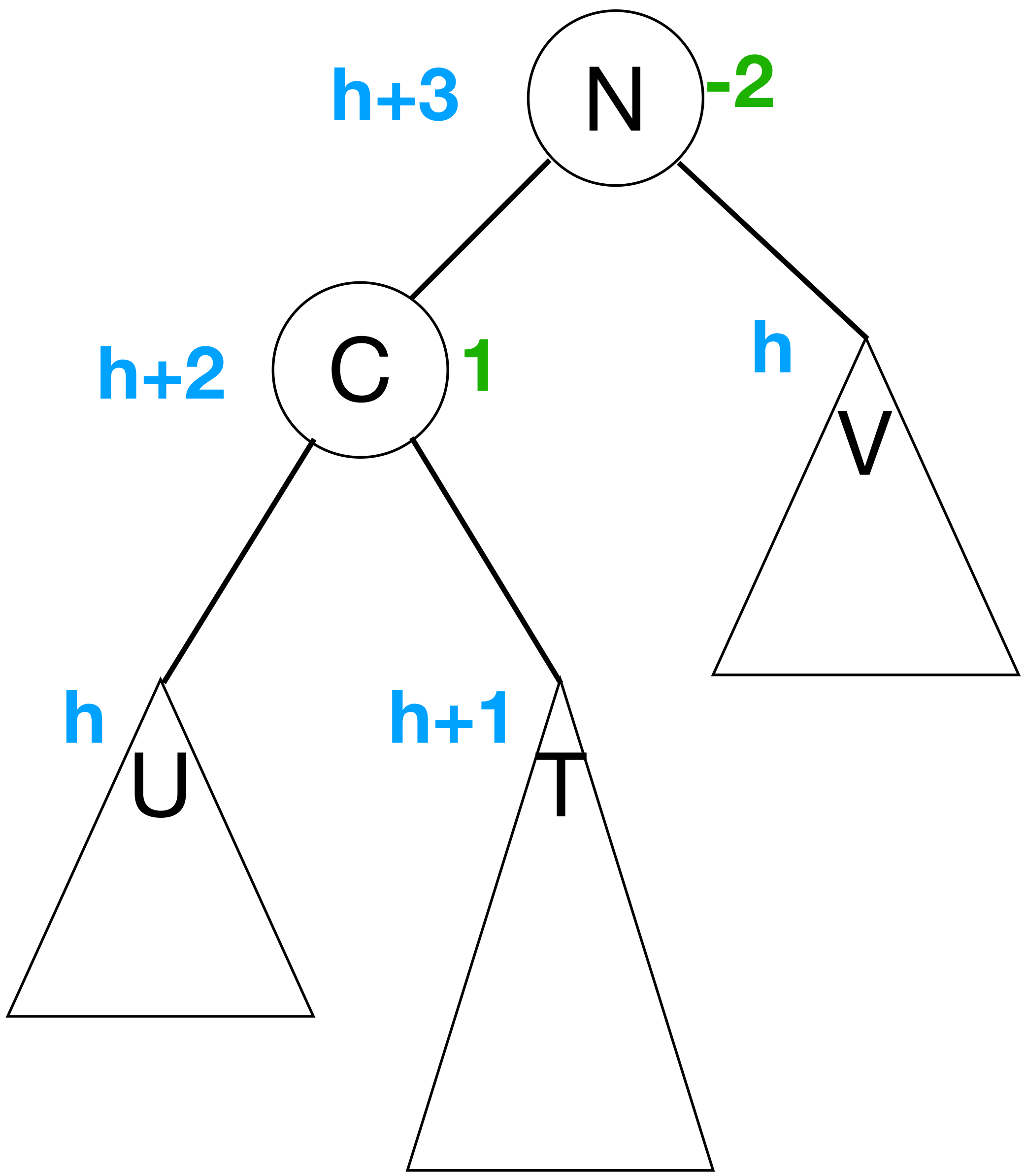
AVL Rebalance

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



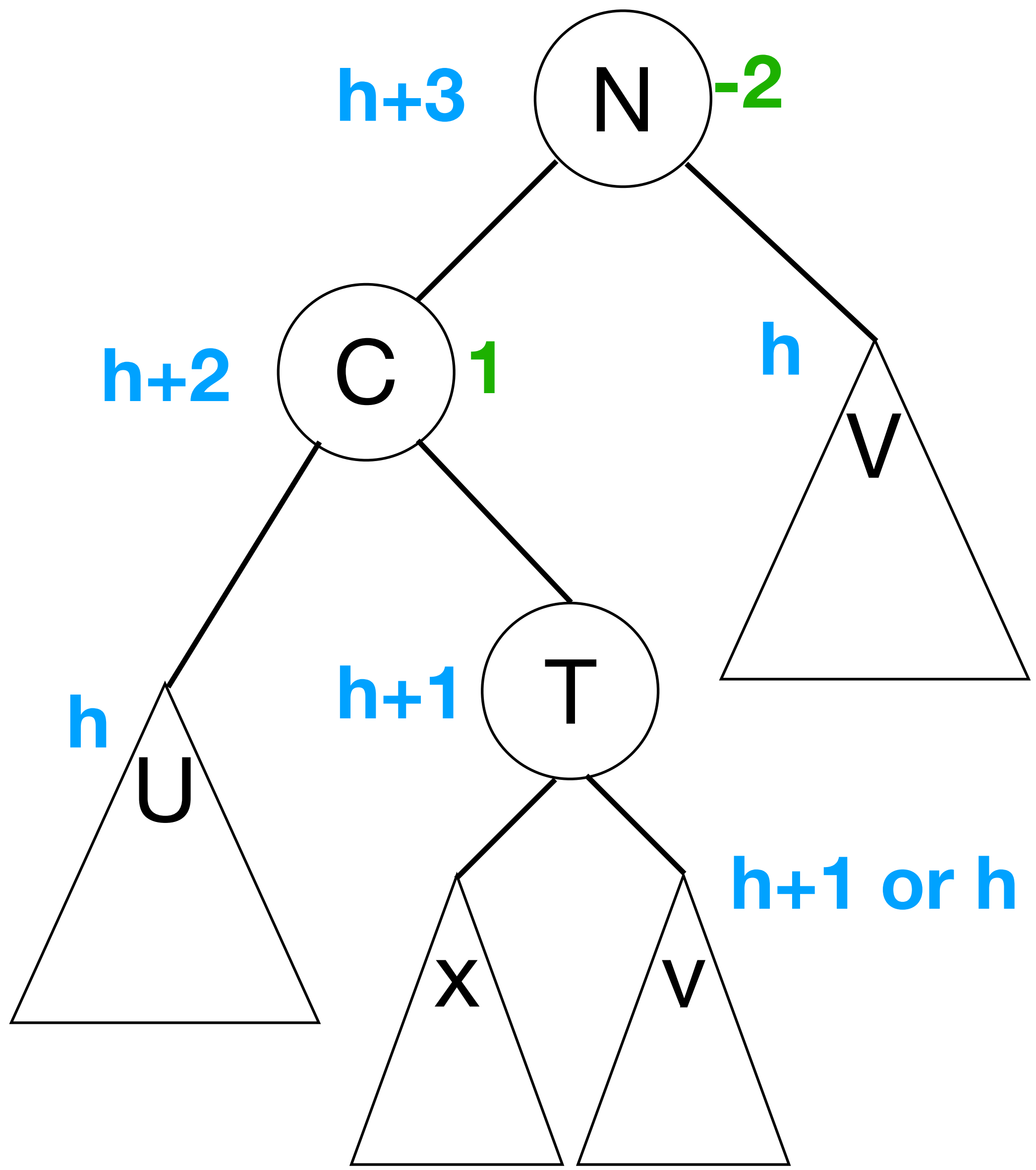
AVL Rebalance

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



AVL Rebalance

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

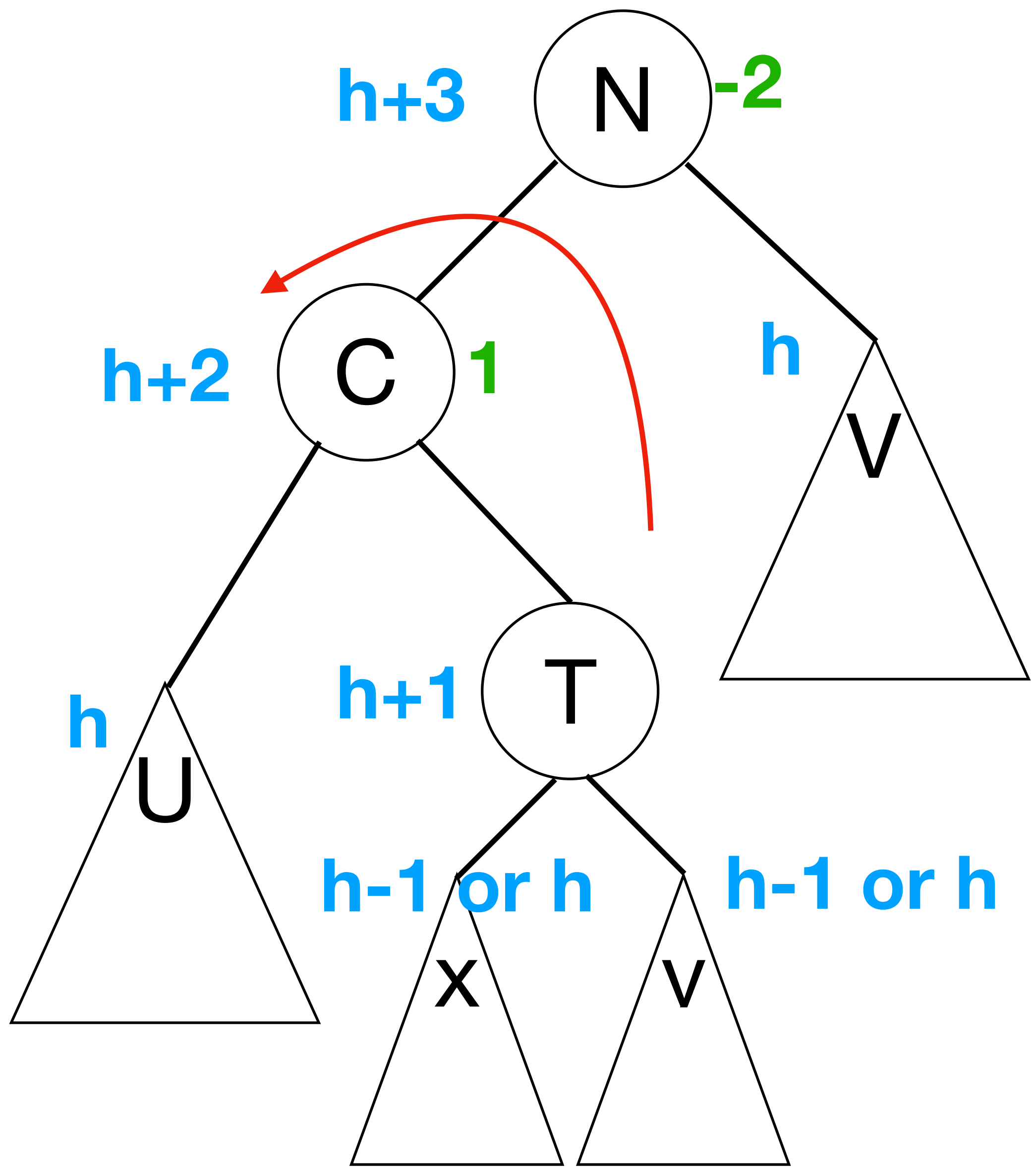


Solution - two rotations:

1. Left rotate C
2. Right rotate N

AVL Rebalance

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

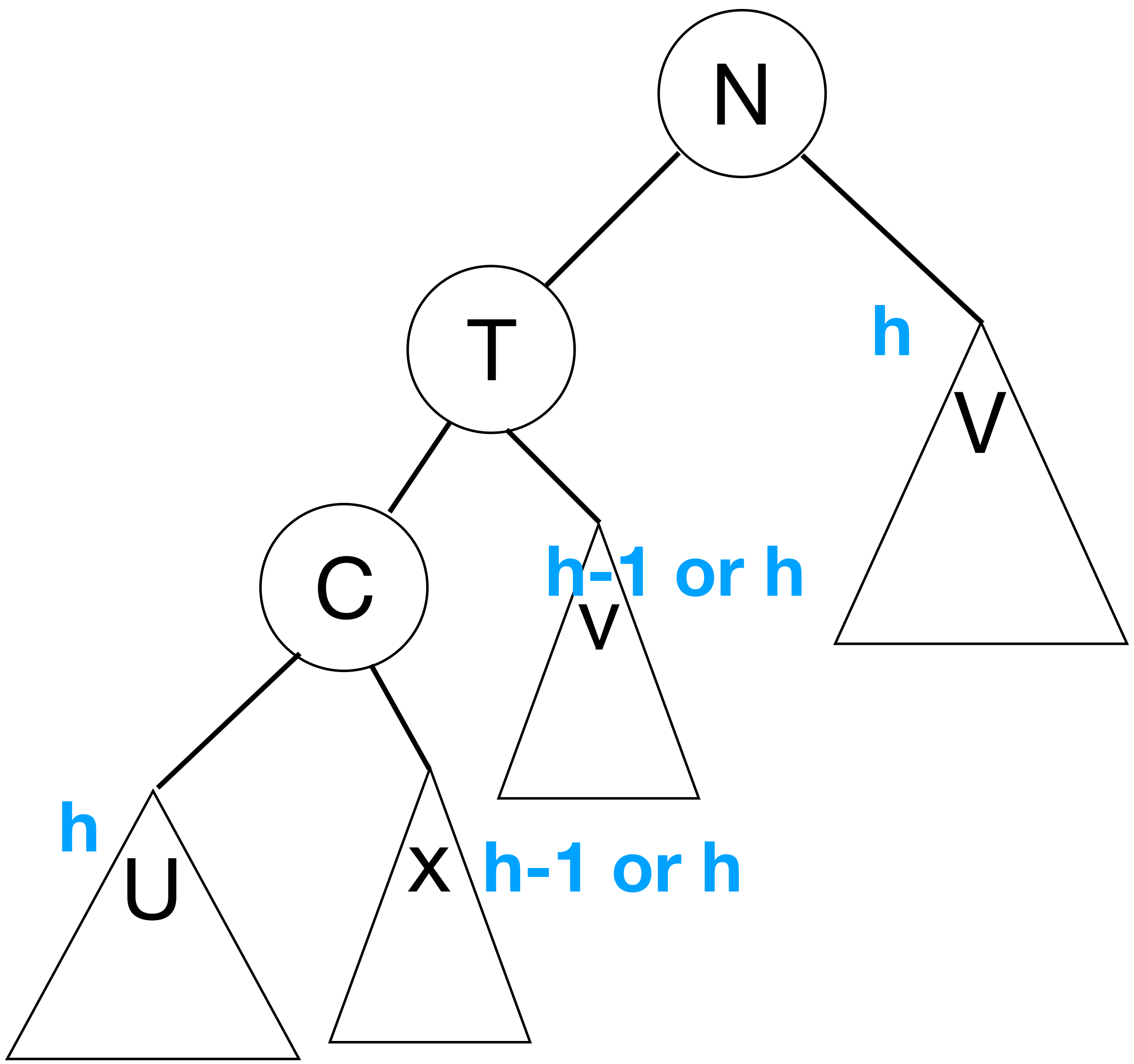


Solution - two rotations:

1. **Left rotate C**
2. Right rotate N

AVL Rebalance

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

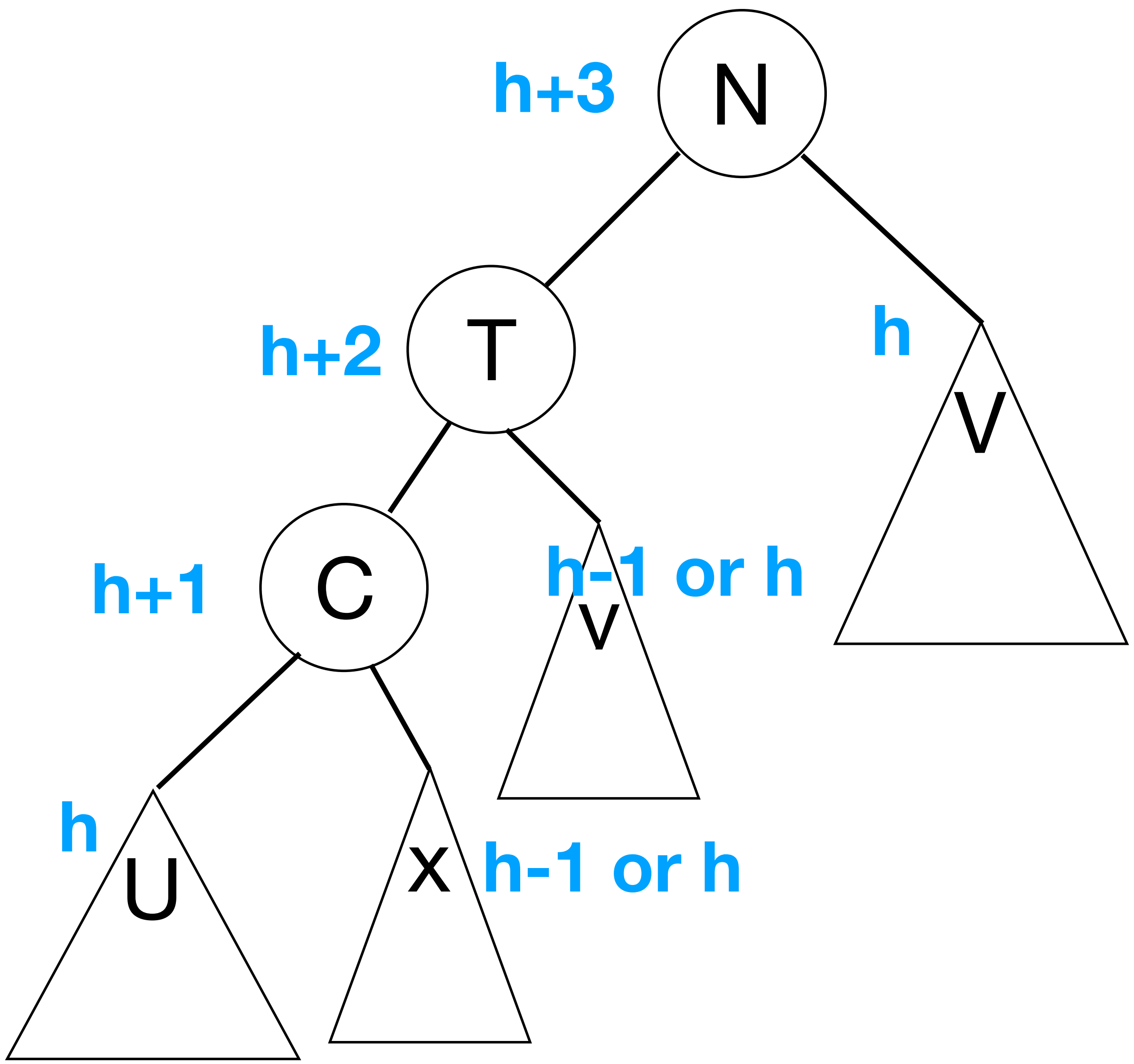


Solution - two rotations:

1. **Left rotate C**
2. Right rotate N

AVL Rebalance

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

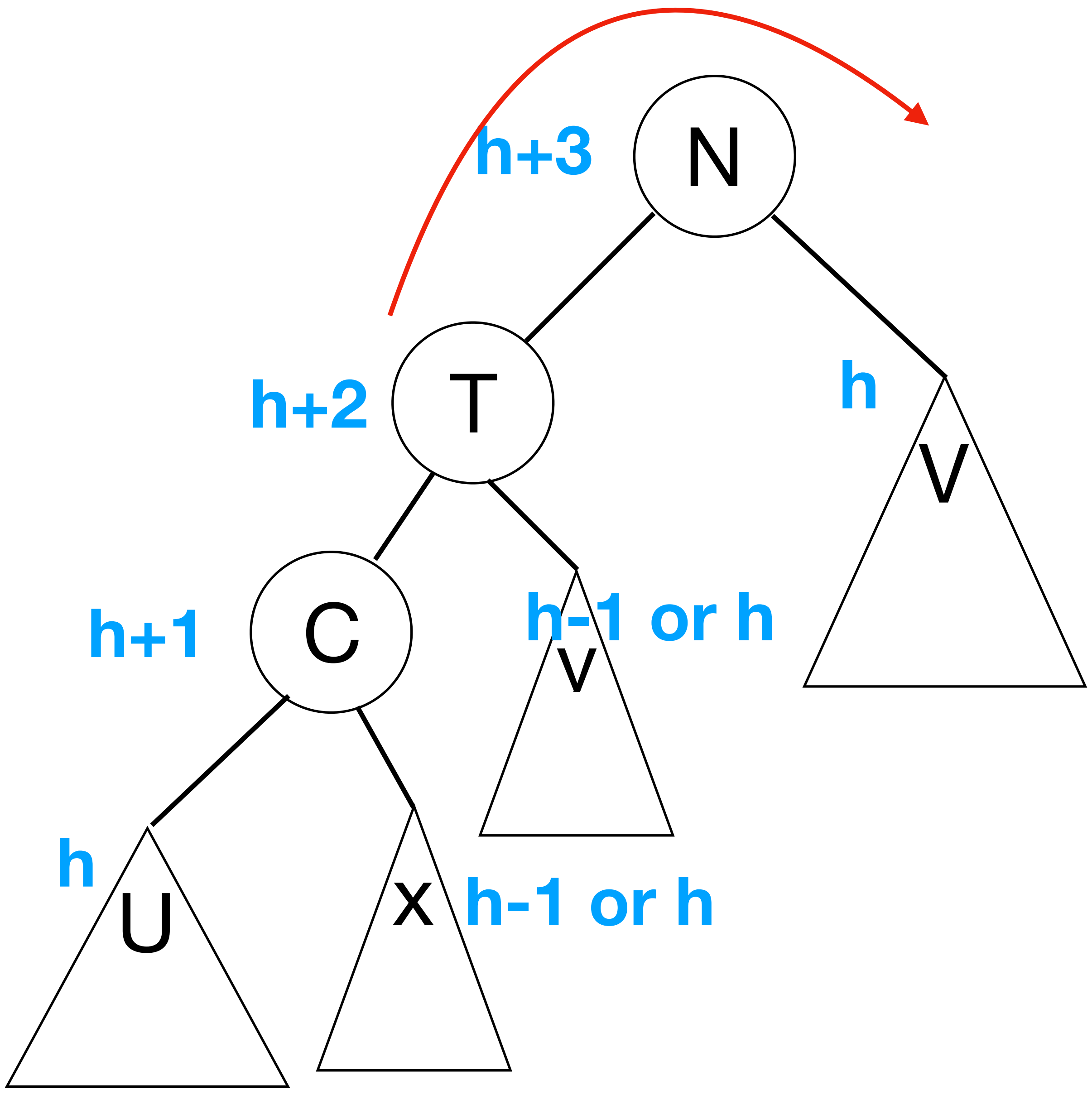


Solution - two rotations:

1. **Left rotate C**
2. Right rotate N

AVL Rebalance

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



Solution - two rotations:

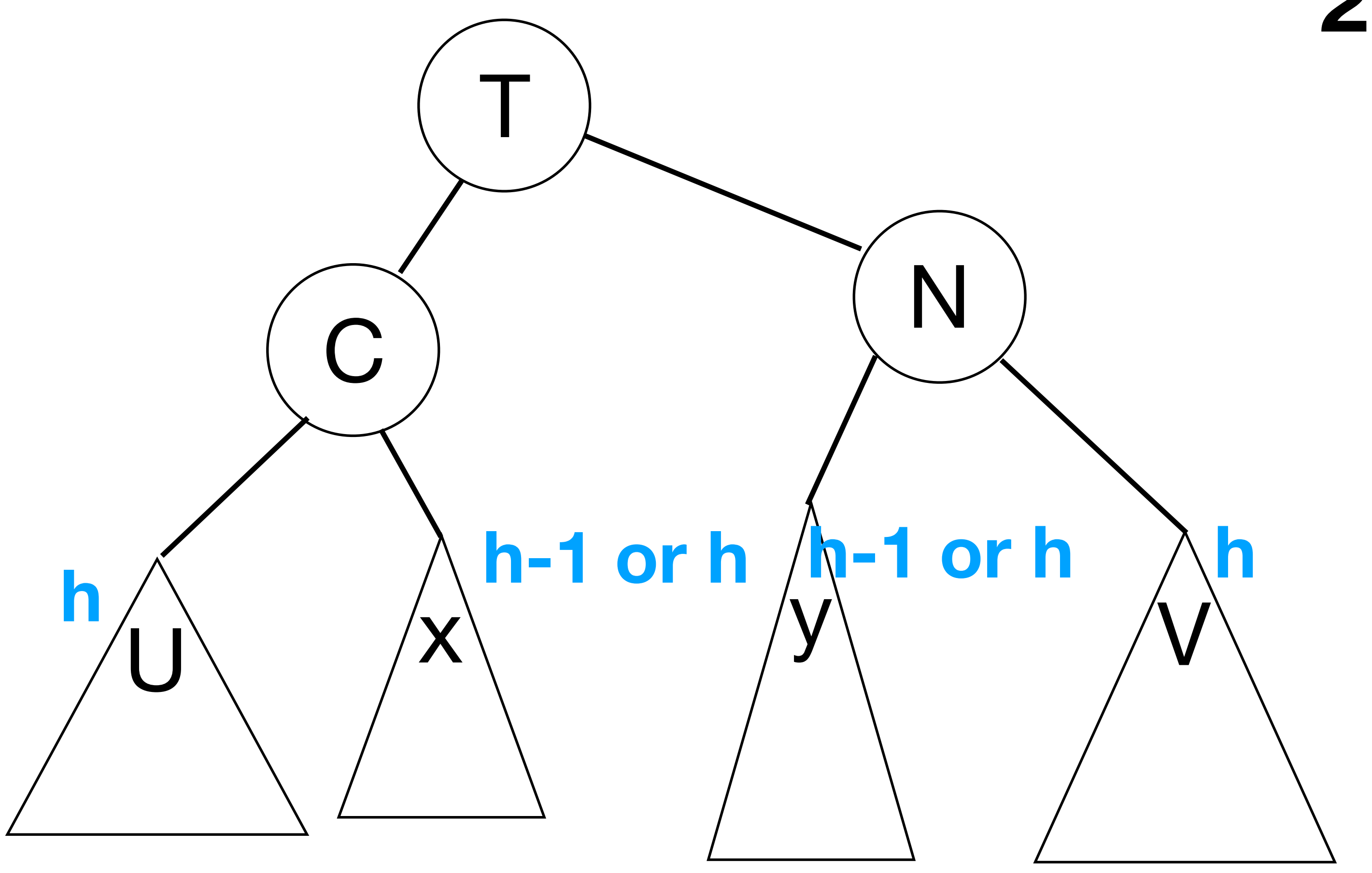
1. Left rotate C
2. **Right rotate N**

AVL Rebalance

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

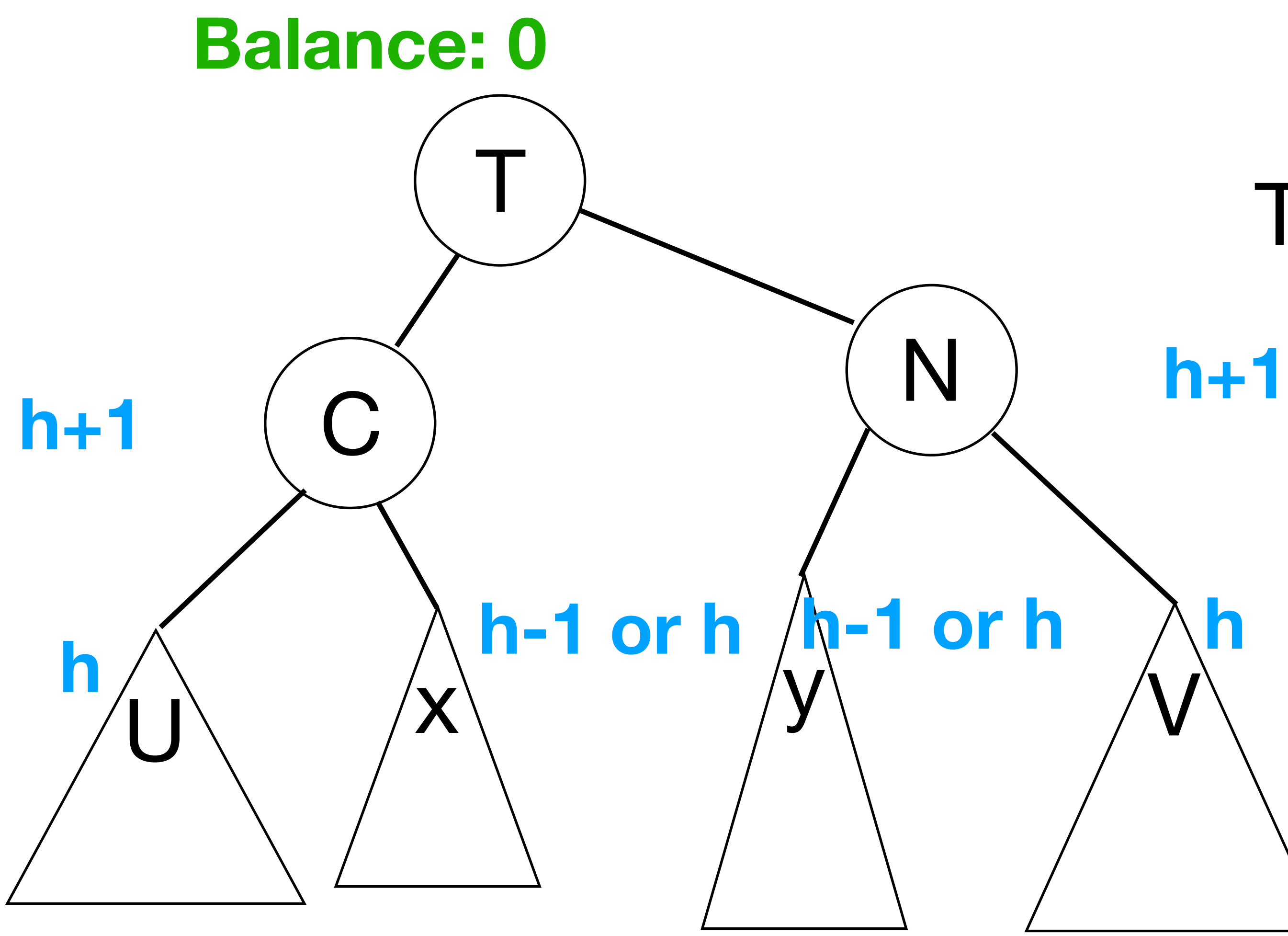
Solution - two rotations:

1. Left rotate C
2. **Right rotate N**



AVL Rebalance

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



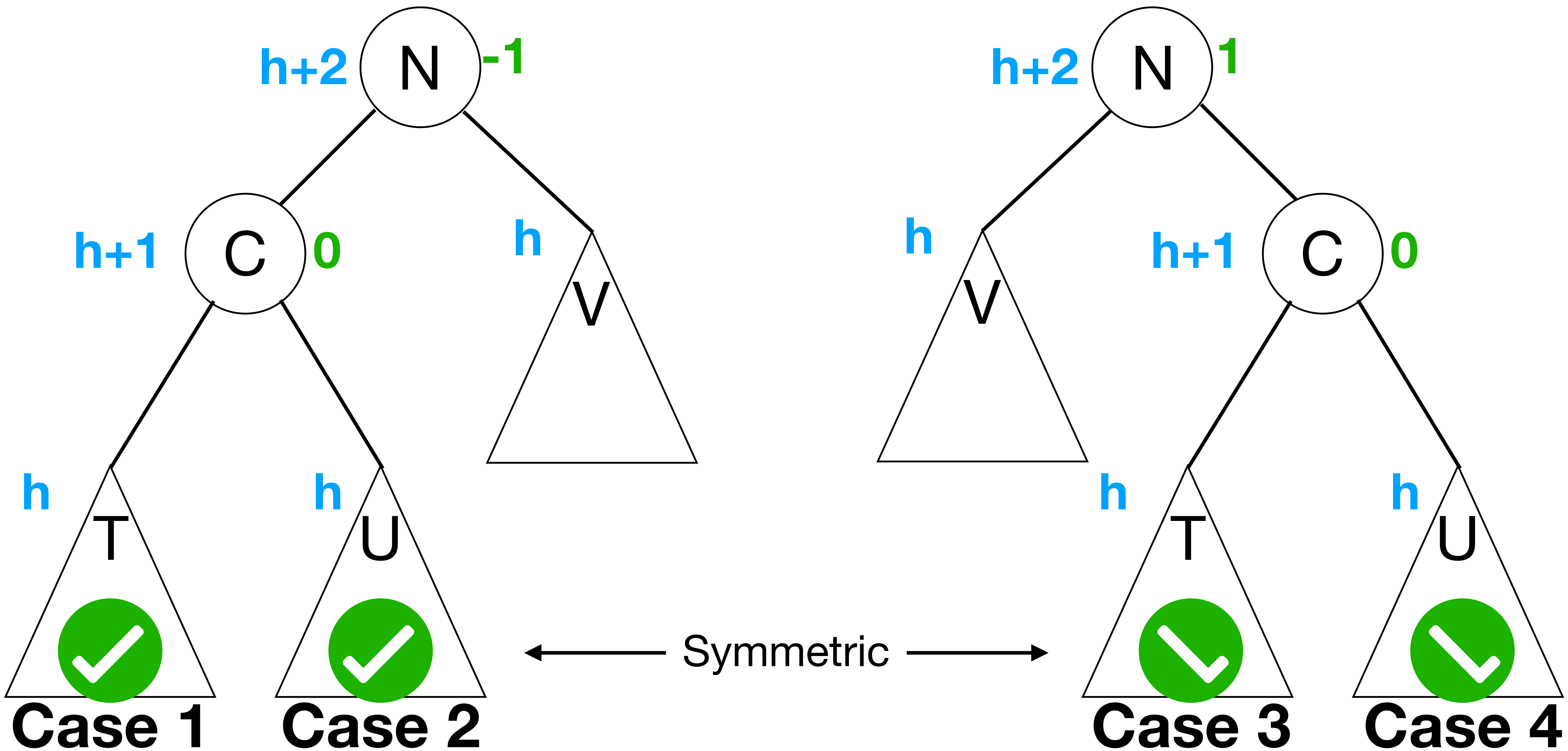
Solution - two rotations:

1. Left rotate C
2. **Right rotate N**

Tree is now AVL balanced.

AVL Rebalance

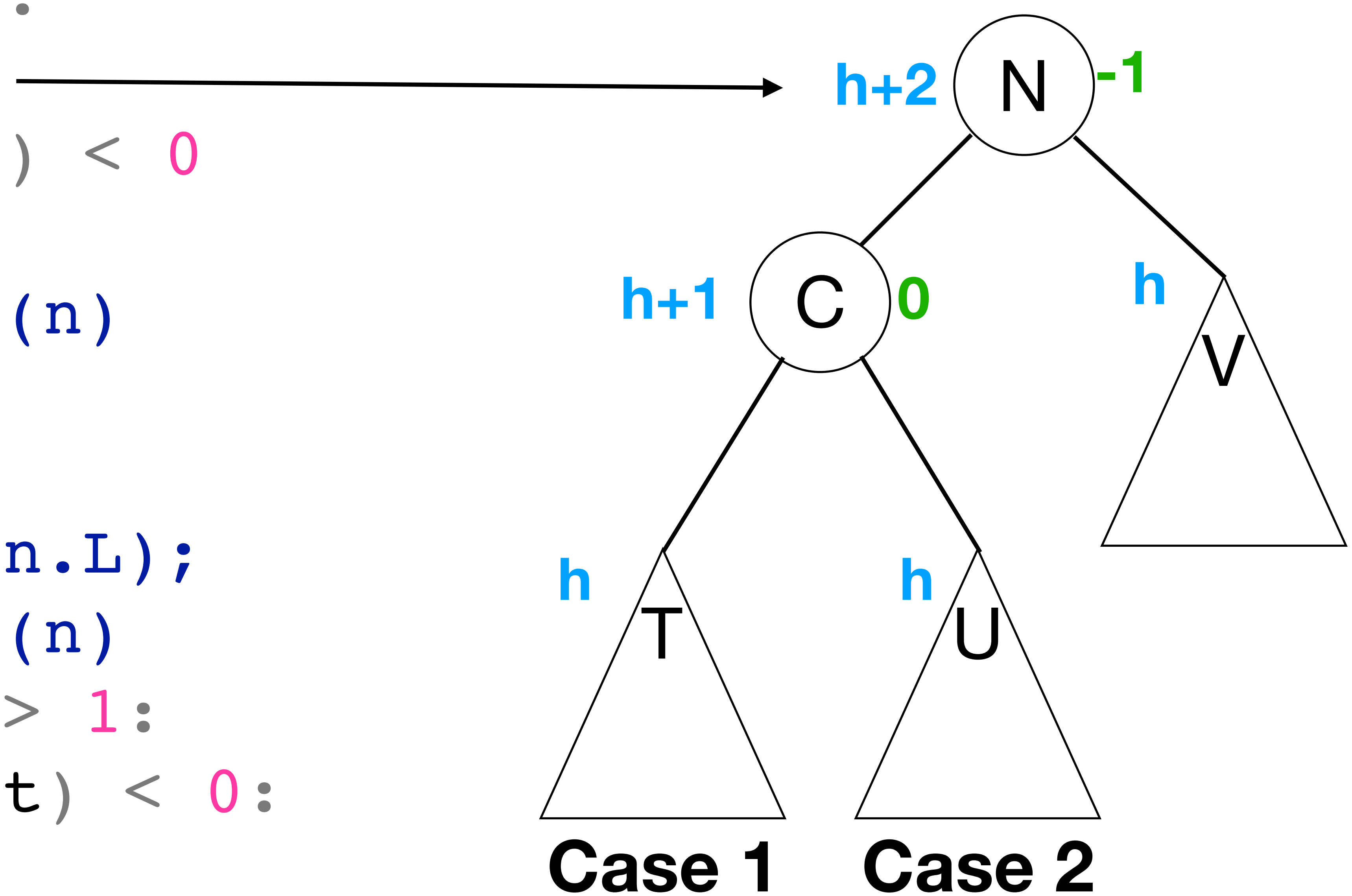
Before an insertion that unbalances n,
the tree must look like one of these:



An insertion that unbalances n could gone one of four places.

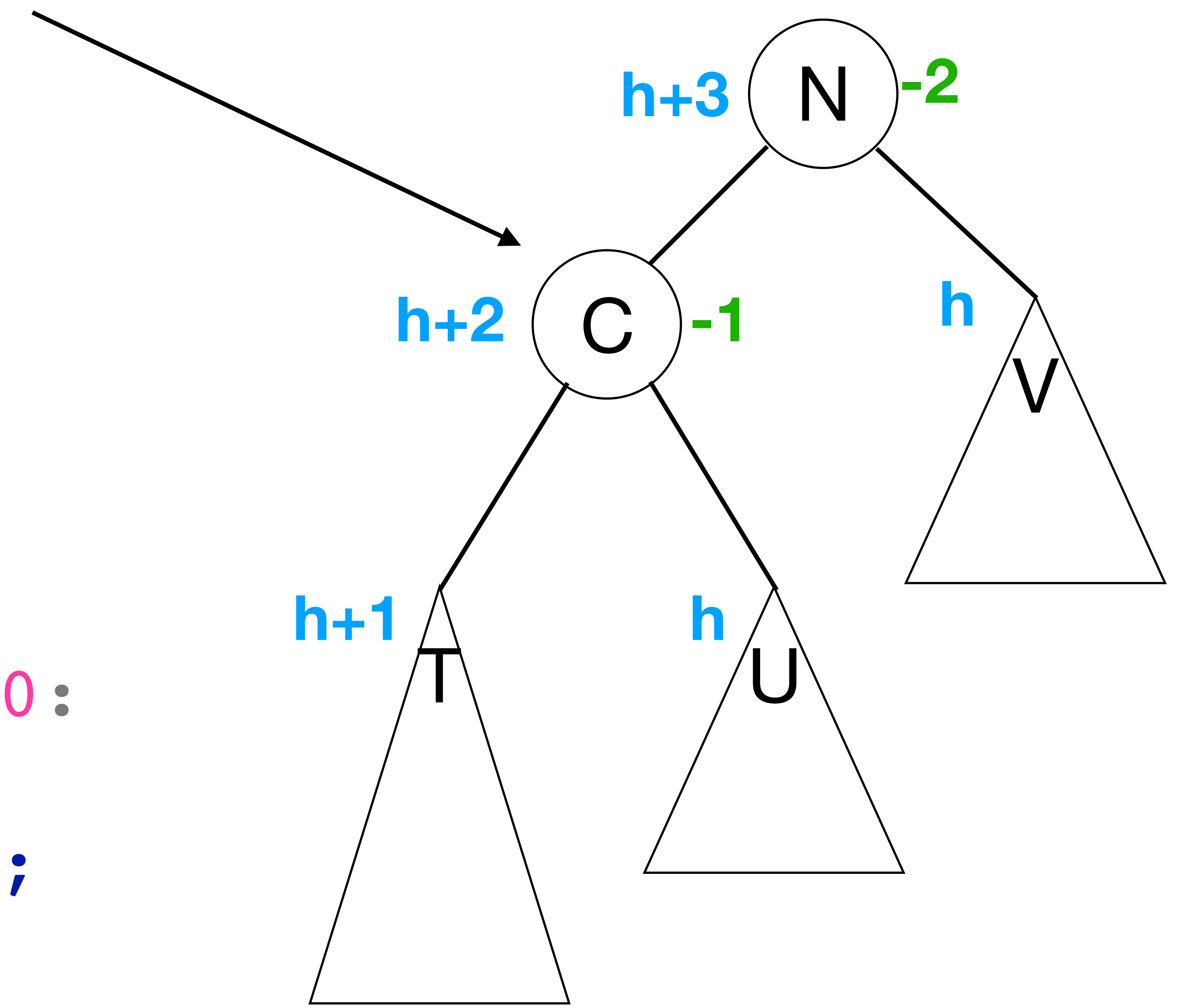
Implementation

```
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)
```



Implementation

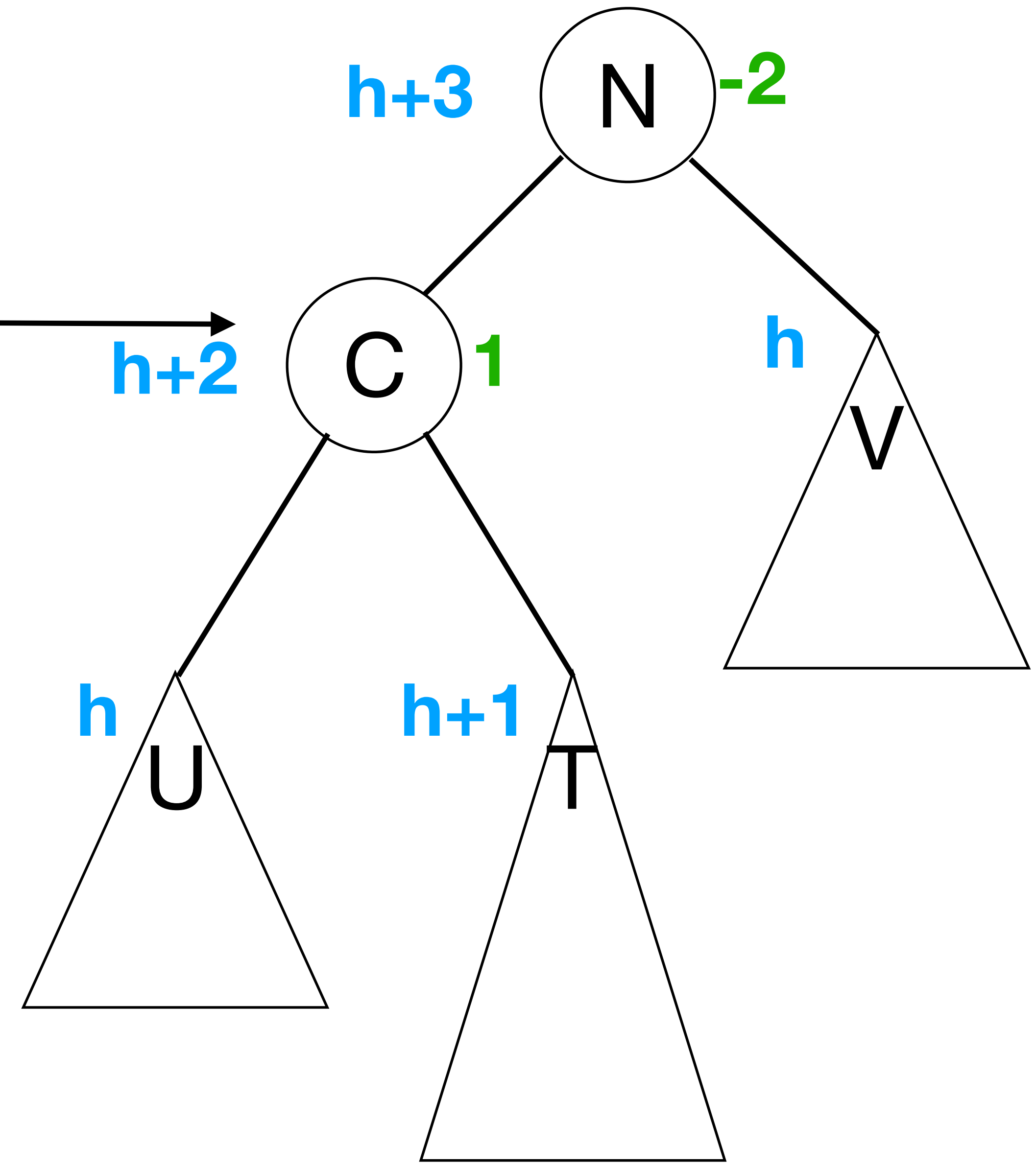
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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



Case 1

Implementation

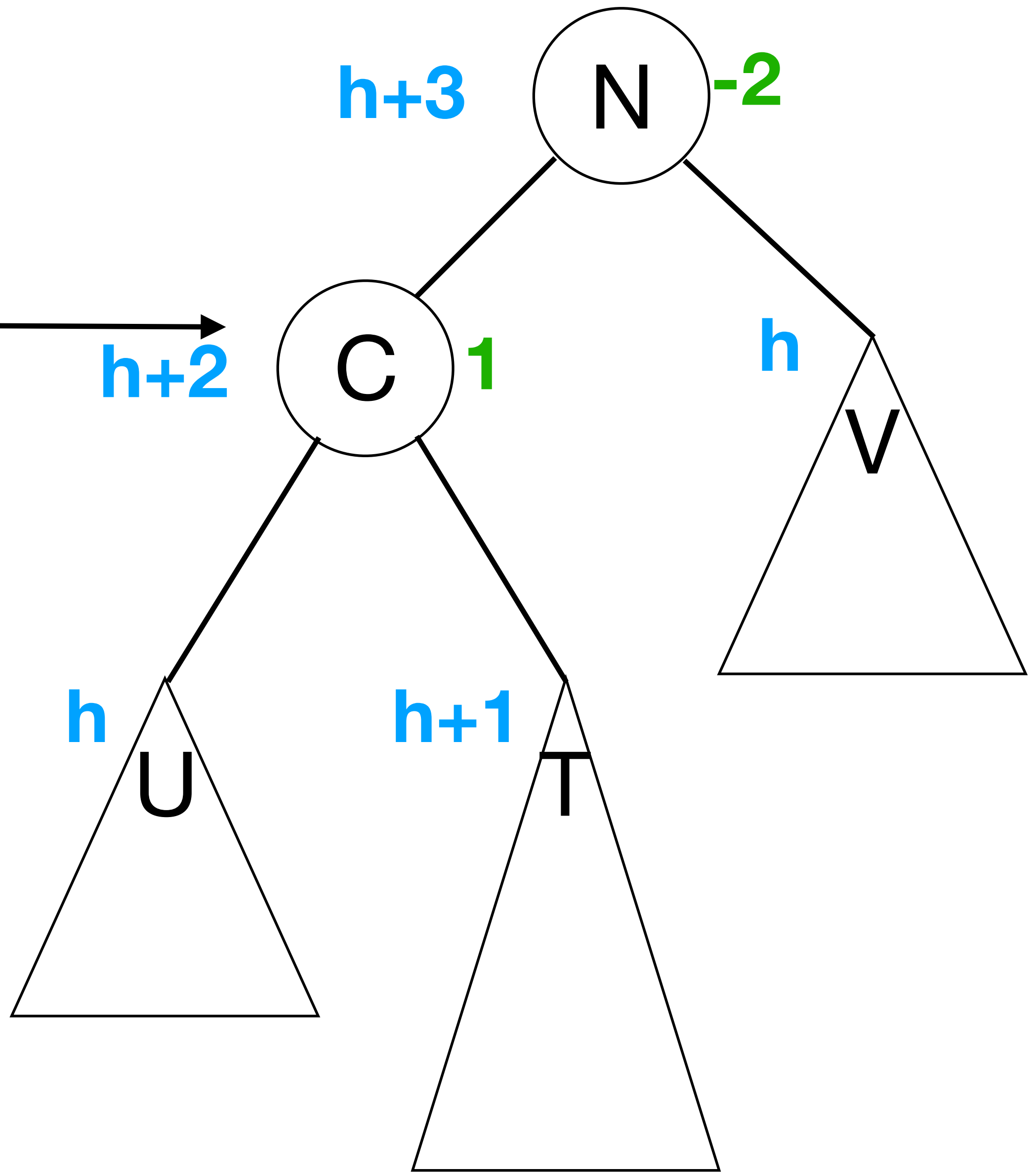
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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



Case 2

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)
```



Case 2

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.

Implementation

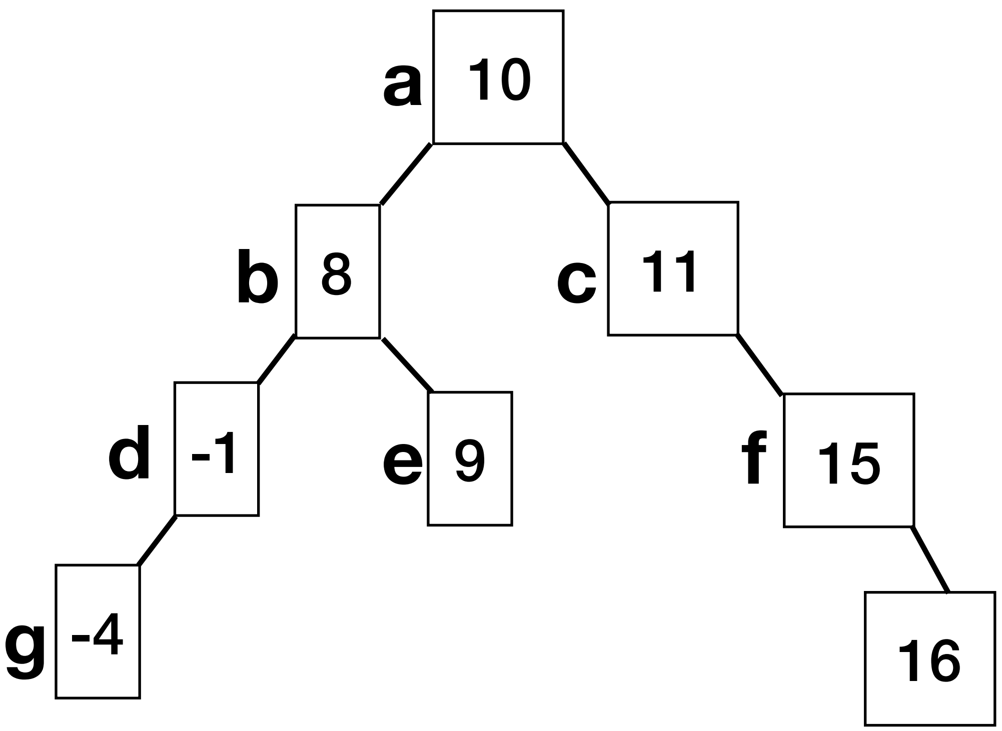
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

Cases 3 and 4 are symmetric with 2 and 1.

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.

Insertion with Rebalance



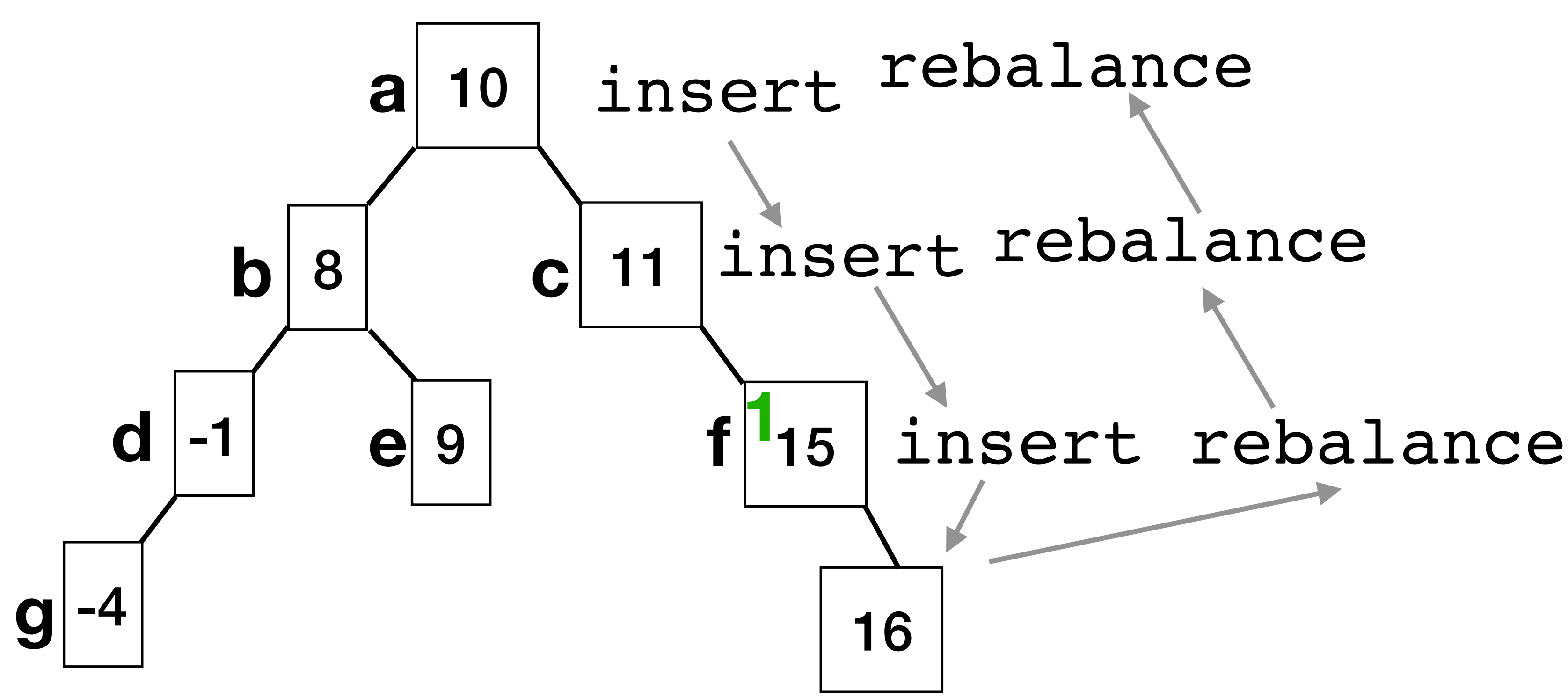

```
insert(Node n, int v):
  //…(other case, irrelevant here)
  else: // v > n.value
    if n has right:
      insert(n.right, v)
   else:
      // attach new node w/ value
      //   v to n.right
  rebalance(n);
```

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

How did we know what rotation to do?

Height of AVL Trees

- As usual, runtime of search, insert, and remove are all O(height).
  - A rotation is O(1), so even if we have to rebalance every node on the path to the root, it's still only h*O(1) rebalances.



Height of AVL Trees

- As usual, runtime of search, insert, and remove are all O(height)
- How many nodes in an AVL tree of height h?
- or, what's the tallest tree you can get with n nodes?
  - Exact proof involves fibonacci sequence(!)
- To add to root's height, you have to add to height of every subtree in one of root's subtrees.

Removing from AVL Tree

- Much like insertion: remove as usual, rebalance as necessary at each level up to the root.
- Whereas insertion only ever requires only one rebalance, deletion can require many
    - but still no more than the tree’s height.