CSCI 241

Scott Wehrwein

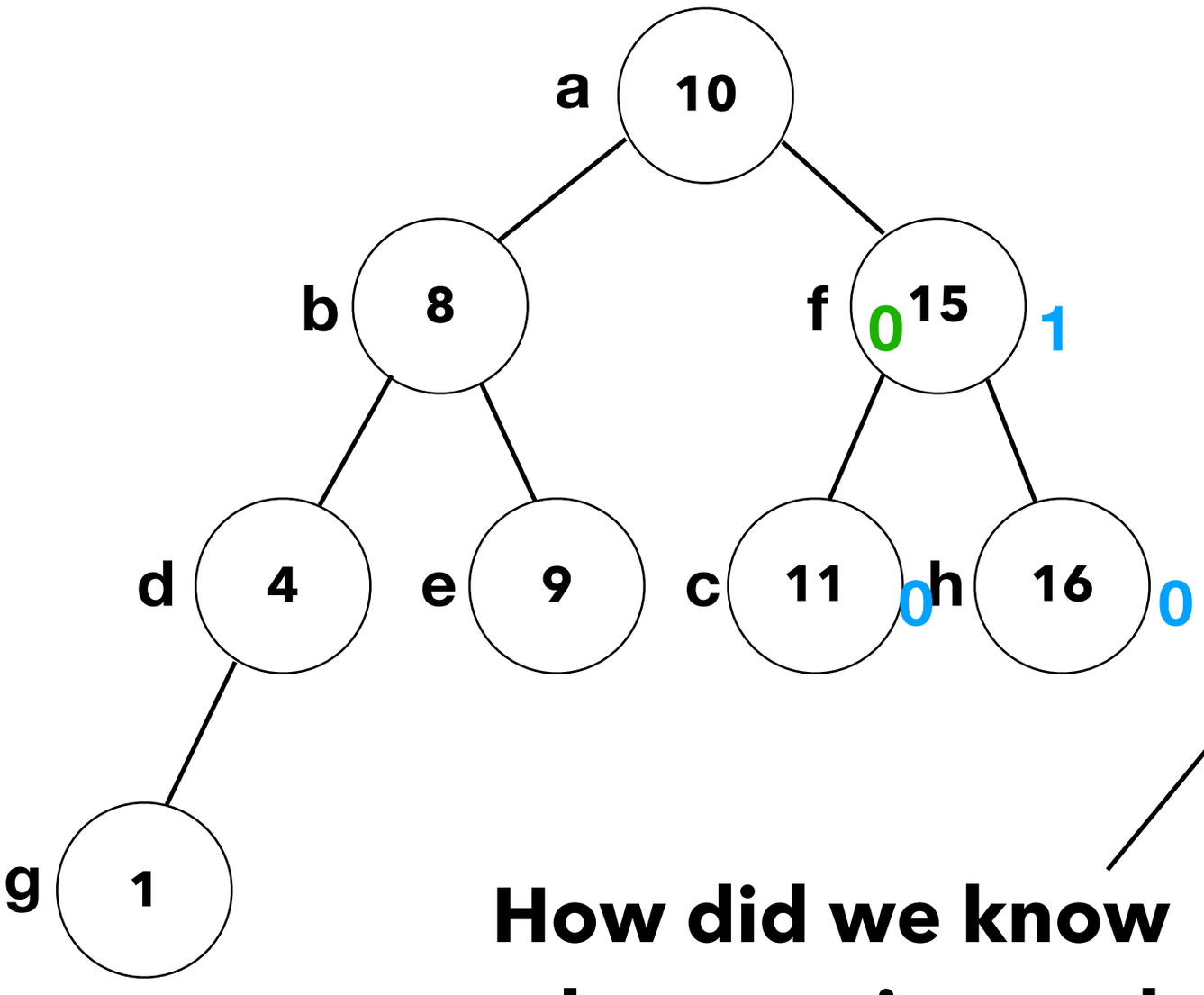
AVL Trees: Rebalancing

Goals

Understand how rebalance decides what rotations to perform.

Be prepared implement rebalance.

AVL Insertion




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

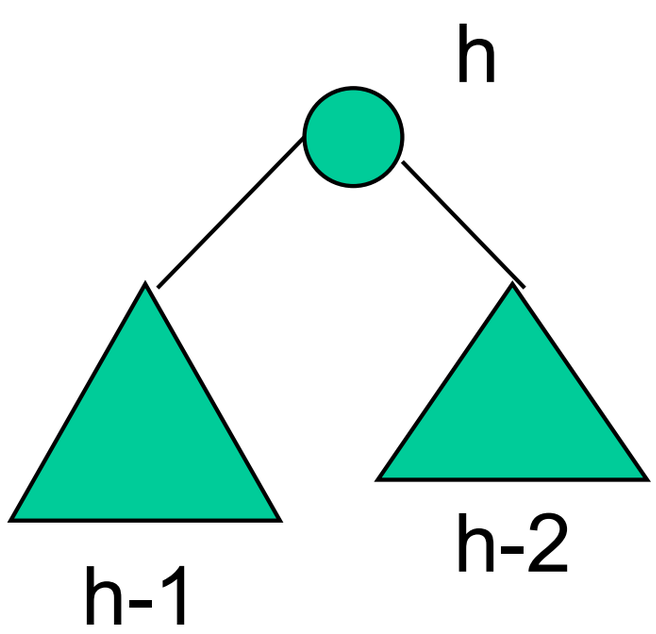
How did we know what rotation to do?

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

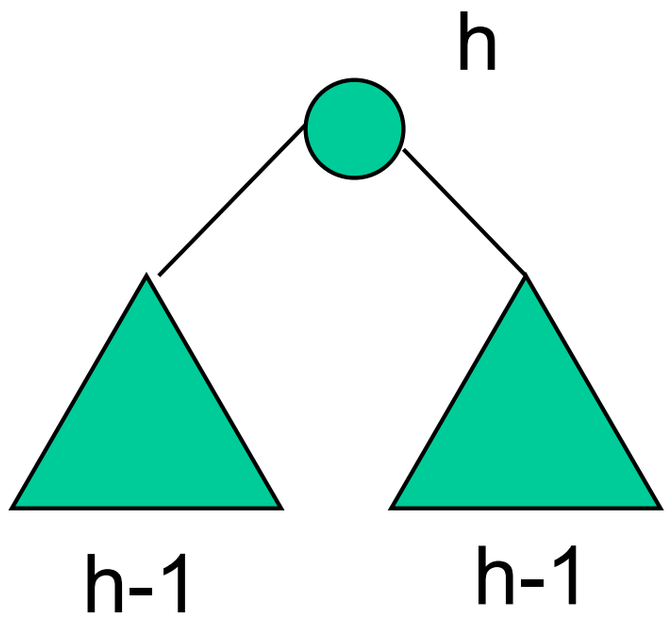
Reminder: AVL Property

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

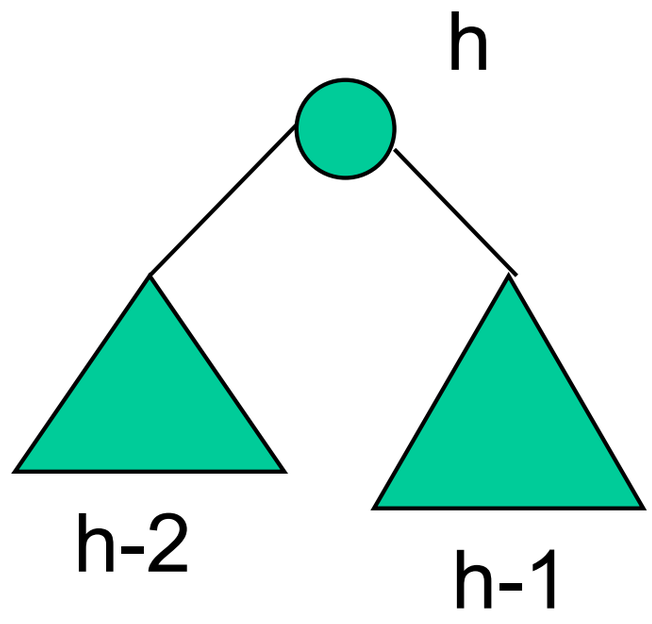
Every subtree in an AVL tree looks like one of these three trees:



(a) Balance factor:-1

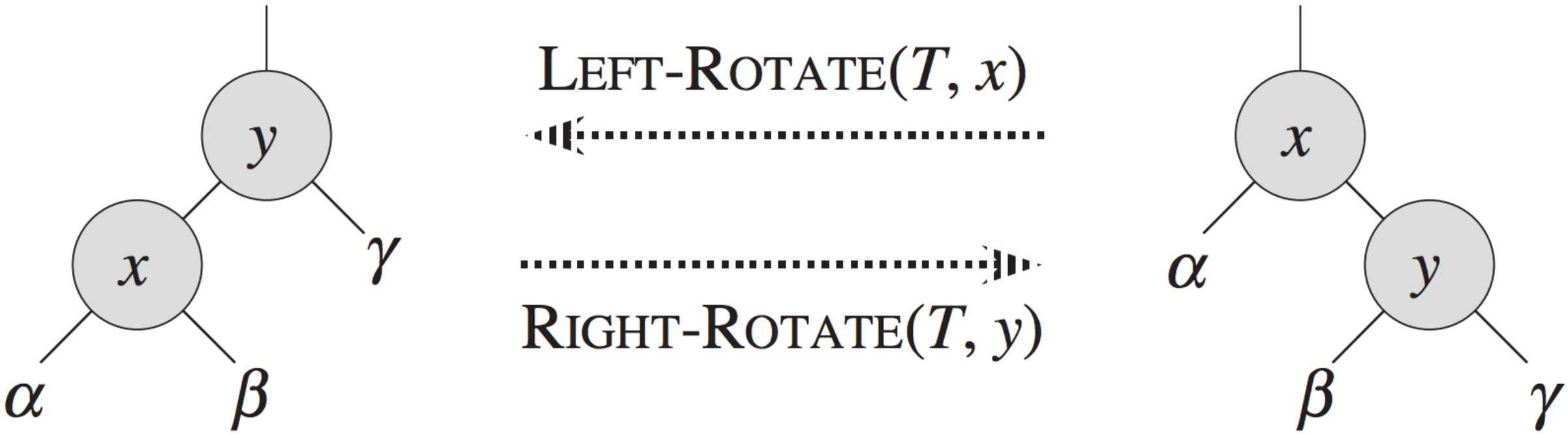


(b) Balance factor: 0

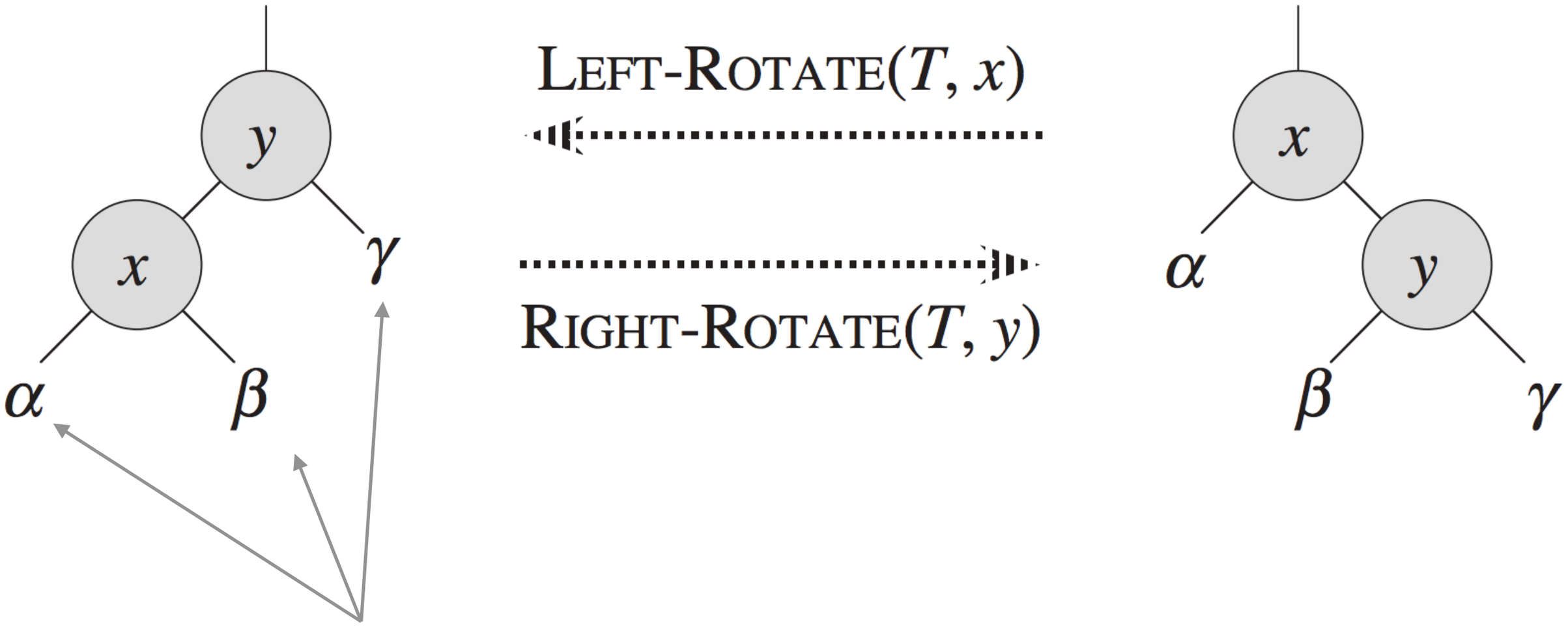


(c) Balance factor: +1

Reminder: Tree Rotations



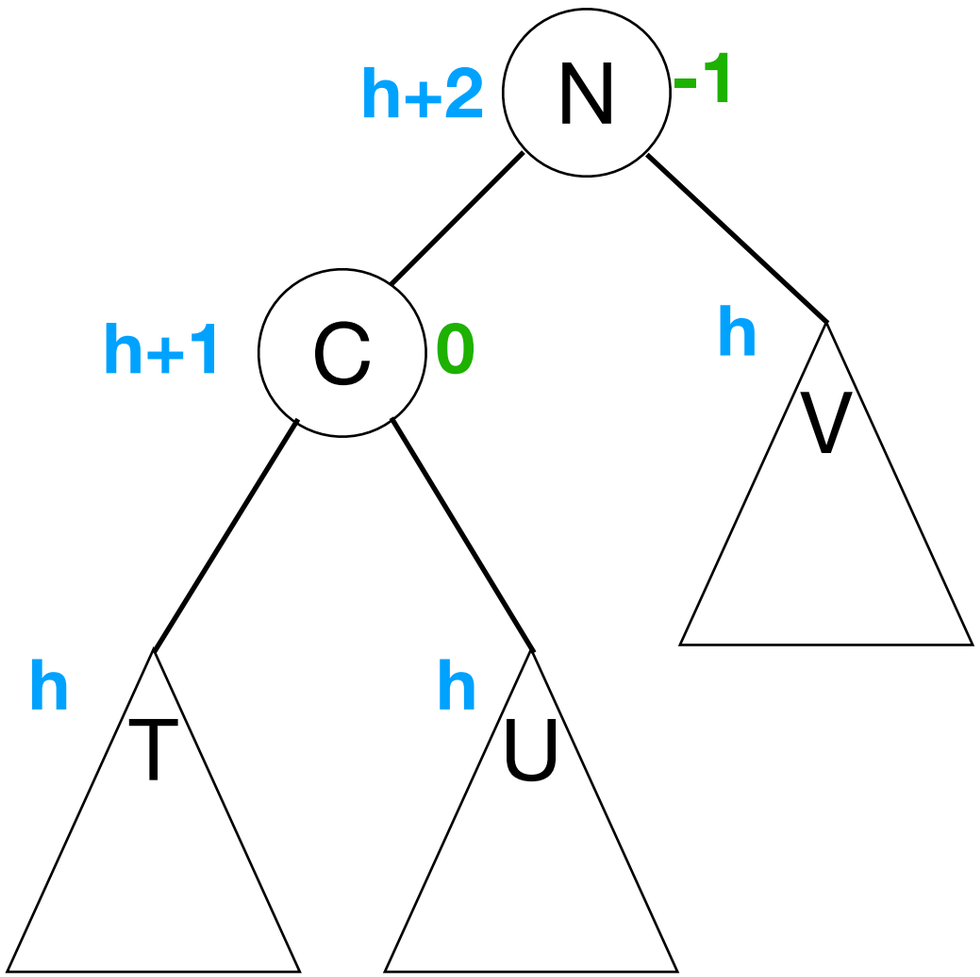
Reminder: Tree Rotations

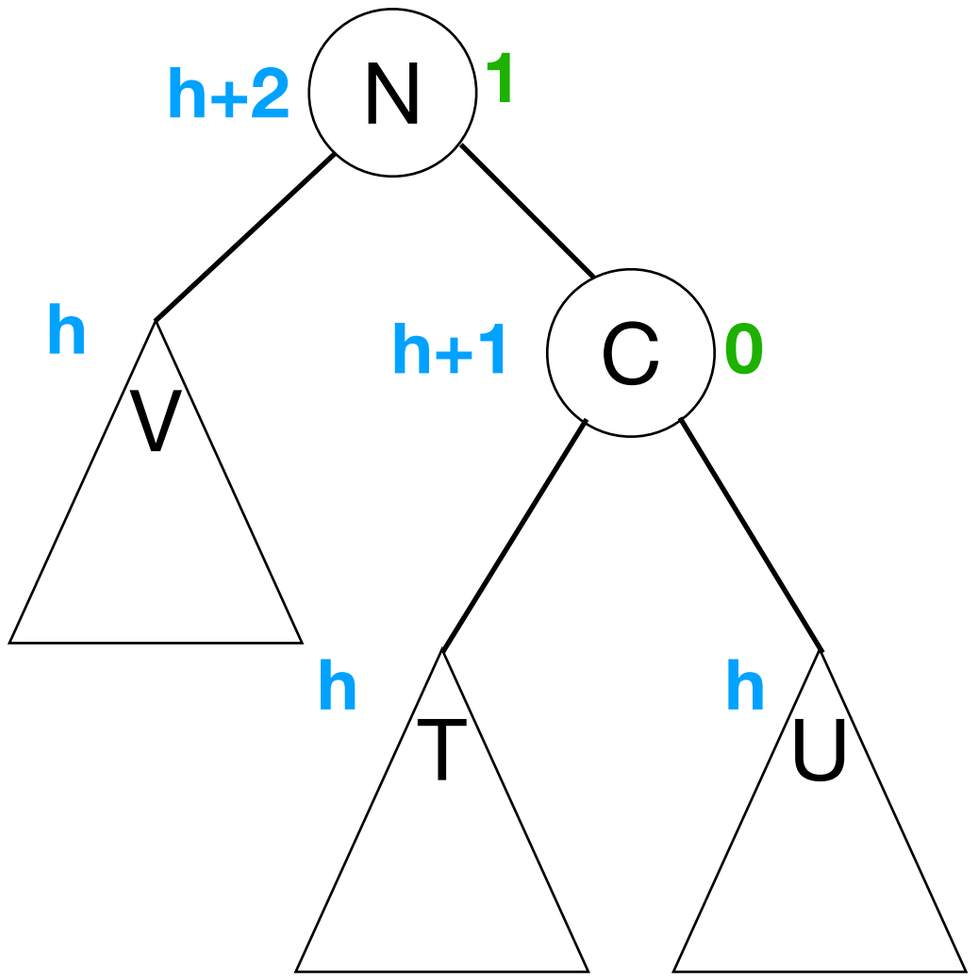


subtrees (could be null, leaf, or tree with many nodes)

AVL Rebalance

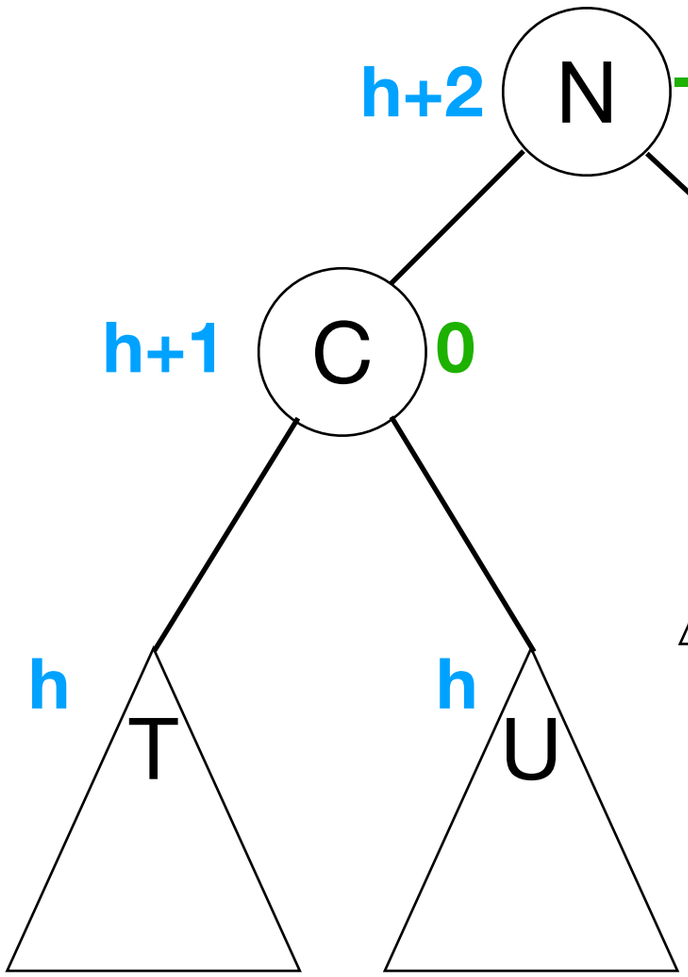
Before an insertion that unbalances N,
the tree must look like one of these:

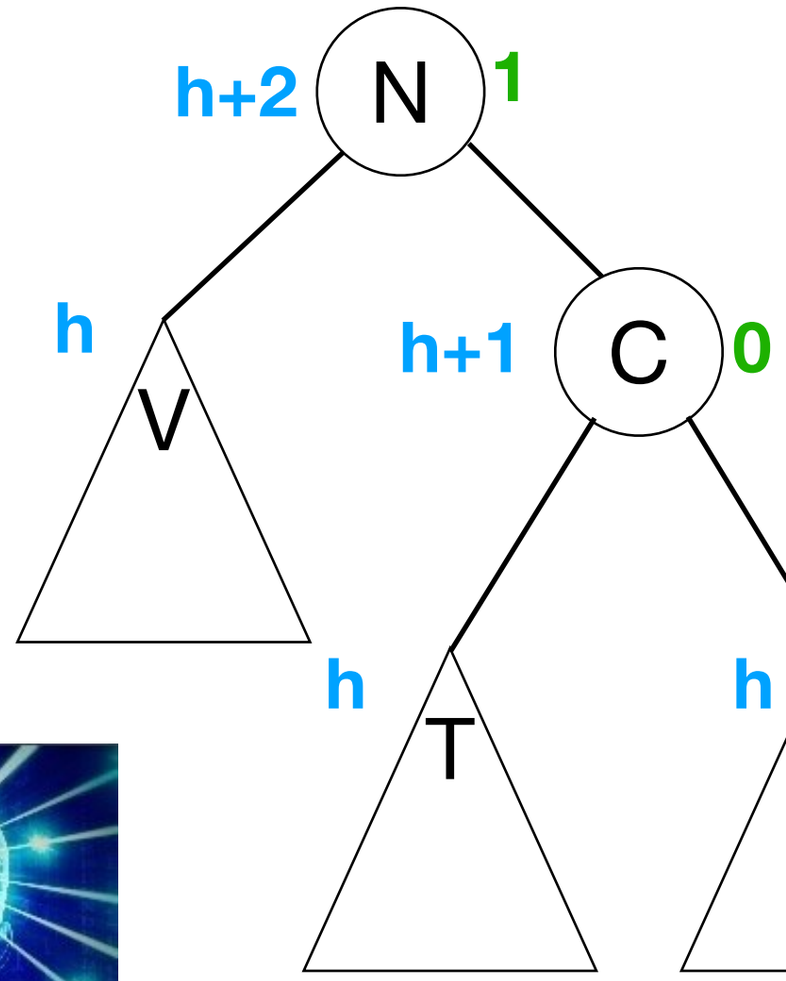




AVL Rebalance

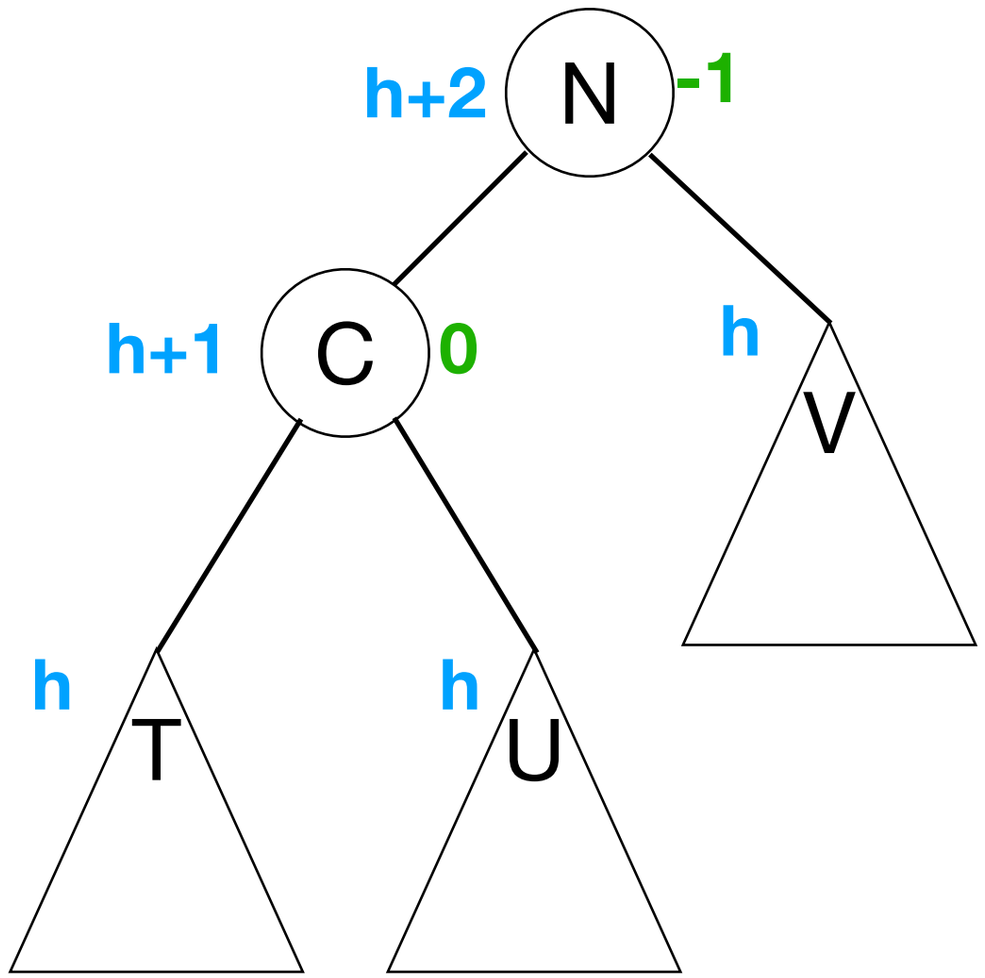
Before an insertion that unbalances N,
the tree must look like one of these:

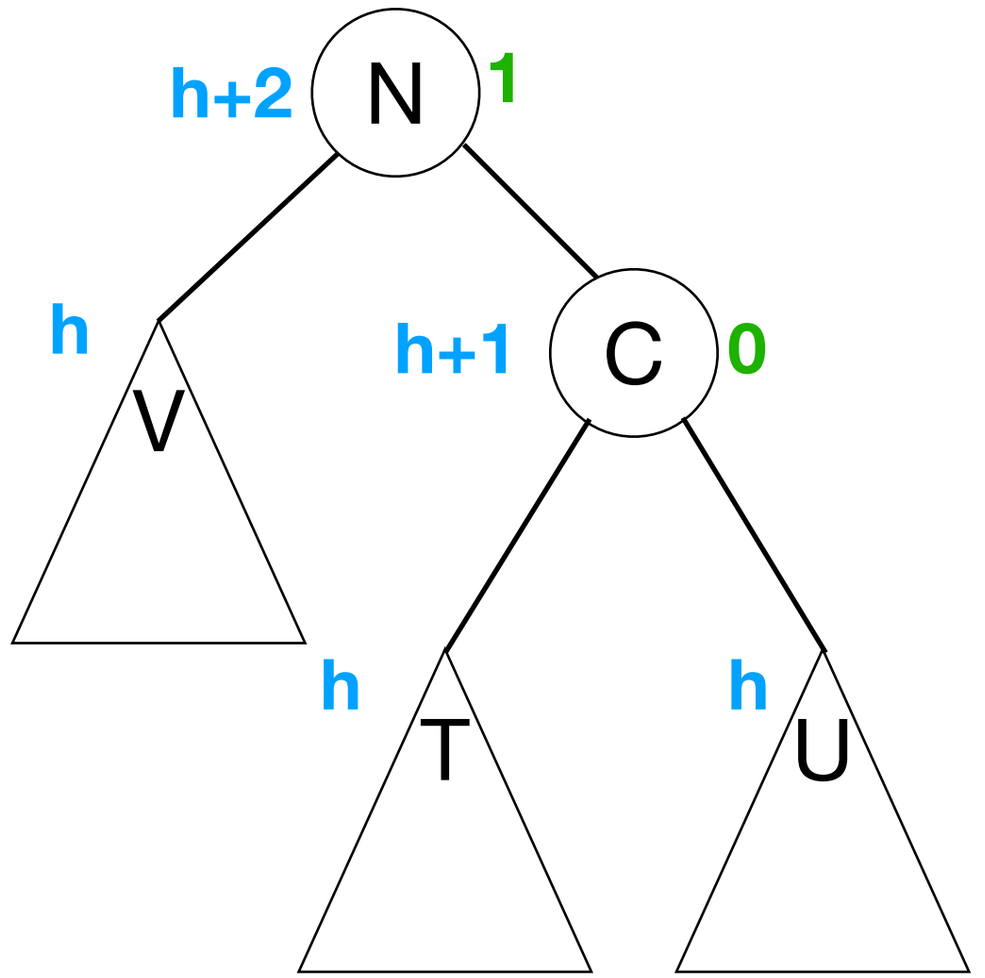




AVL Rebalance

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

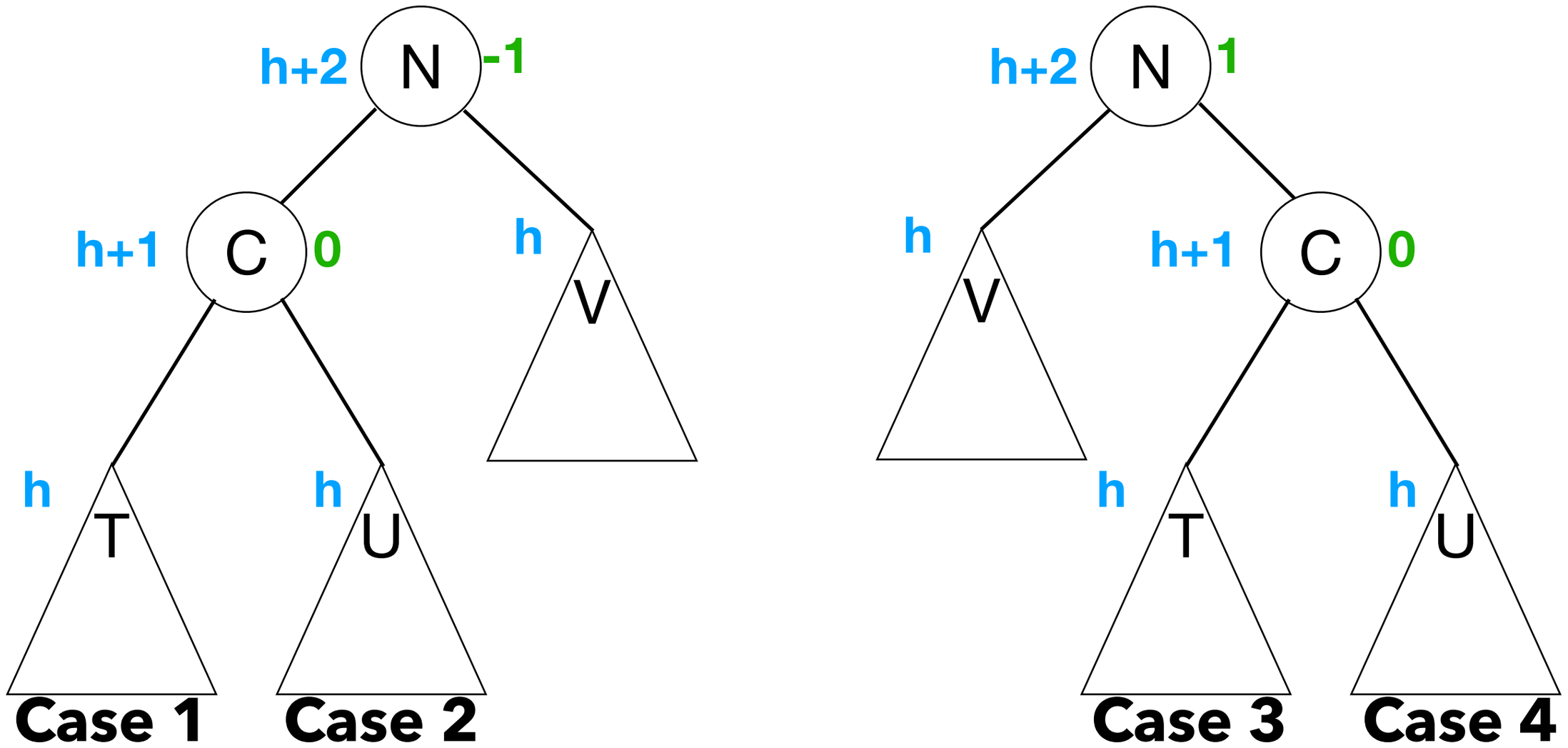




An insertion that *unbalances* N could go one of four places.

AVL Rebalance

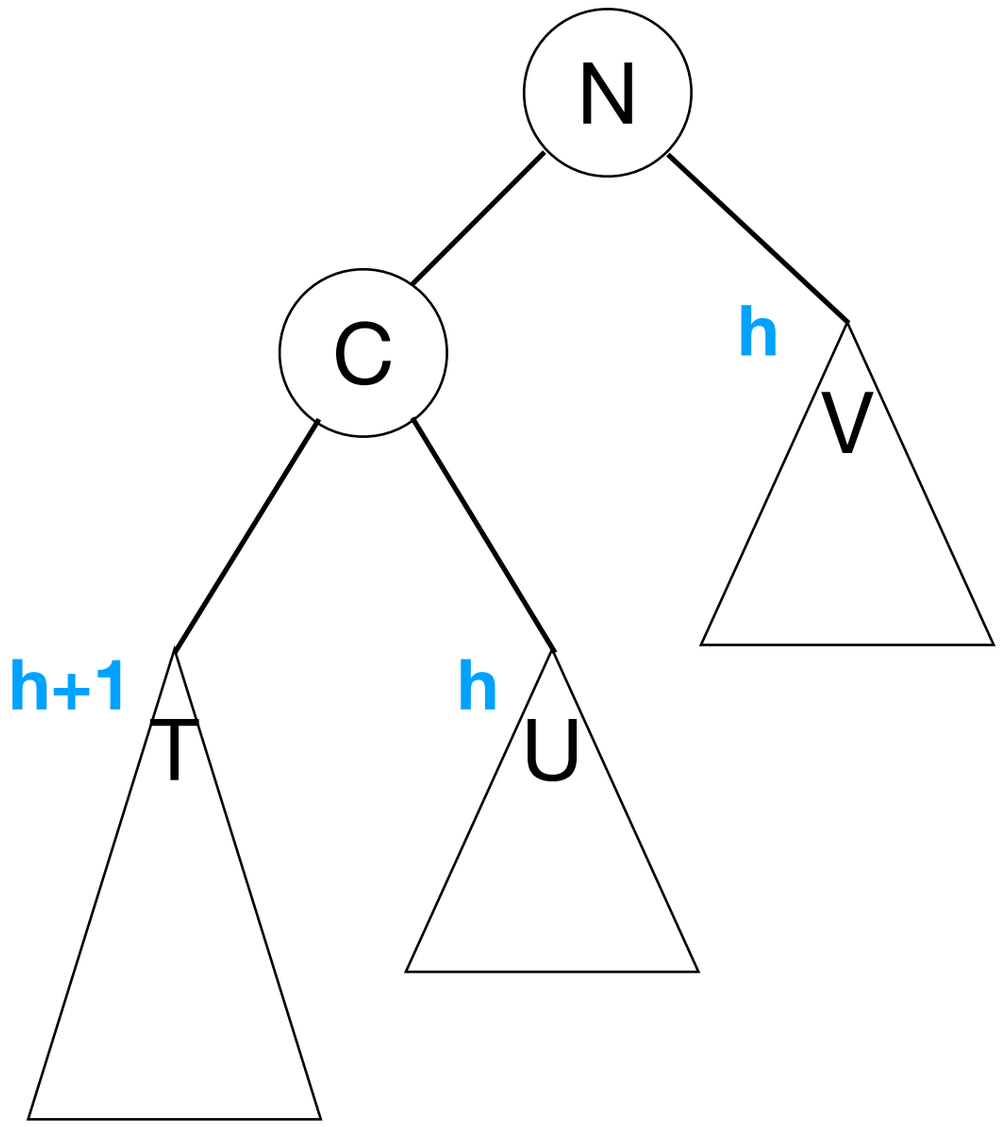
Before an insertion that unbalances N,
the tree must look like one of these:



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

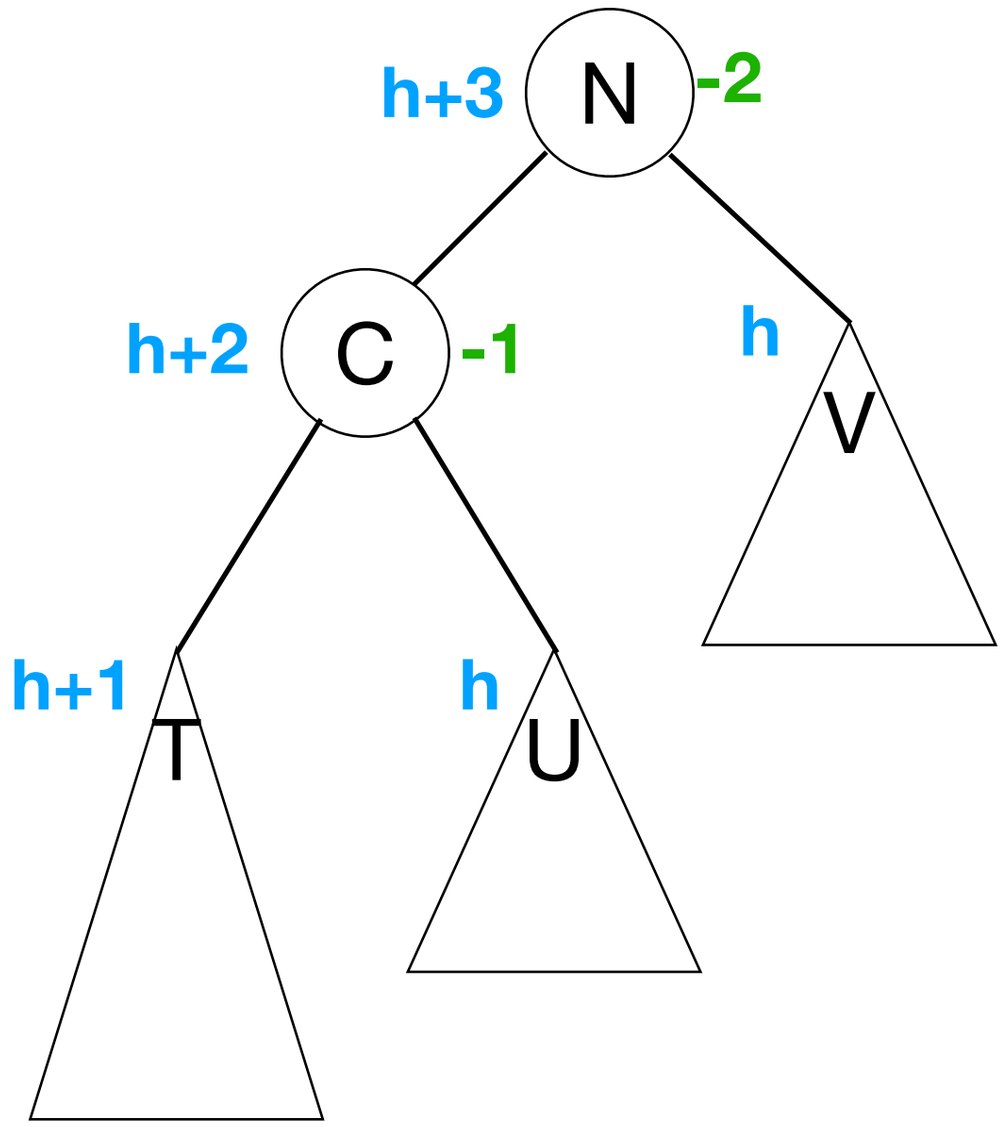
AVL Rebalance

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



AVL Rebalance

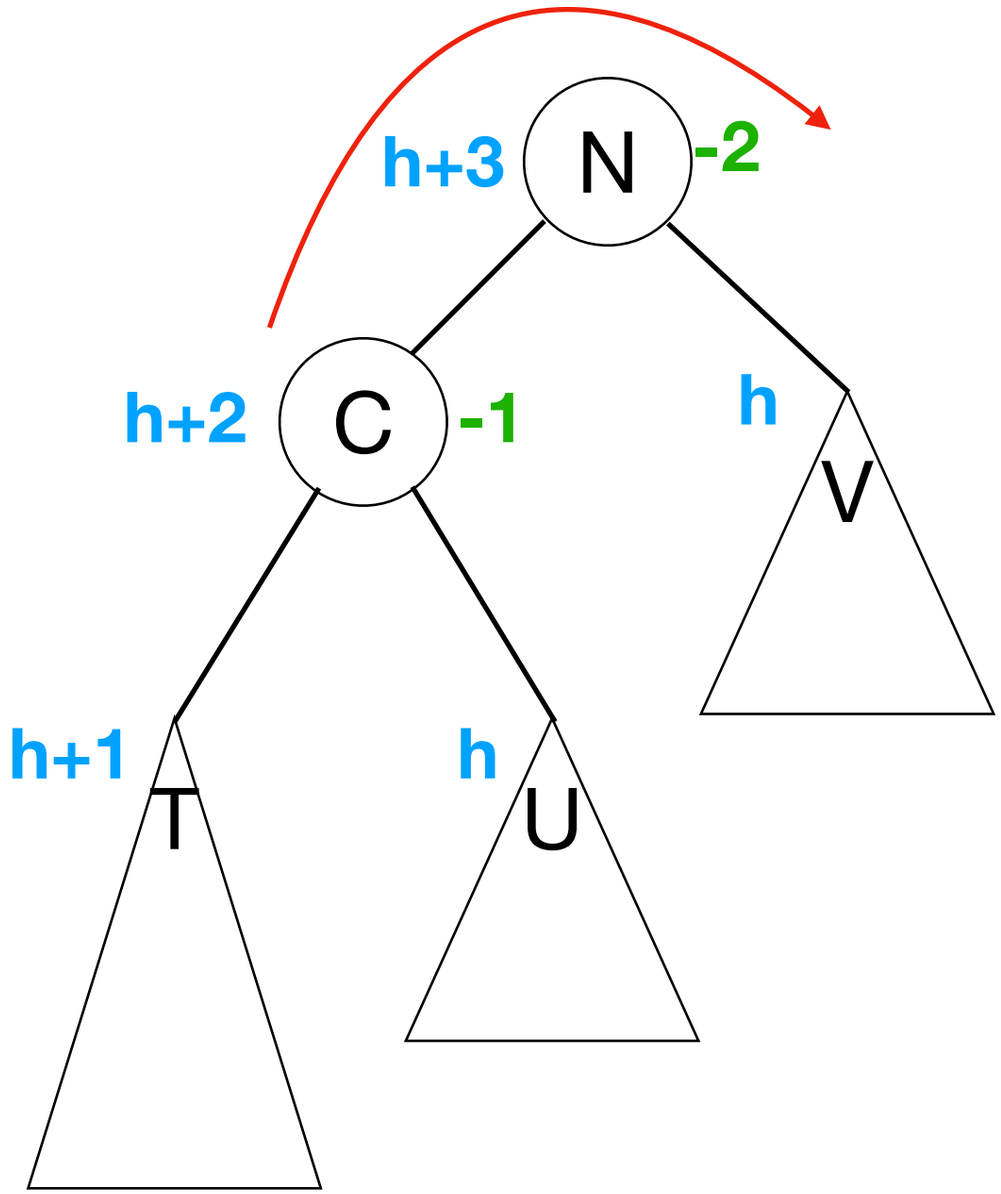
Case 1: After BST insertion step, the tree looks like this.



AVL Rebalance

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

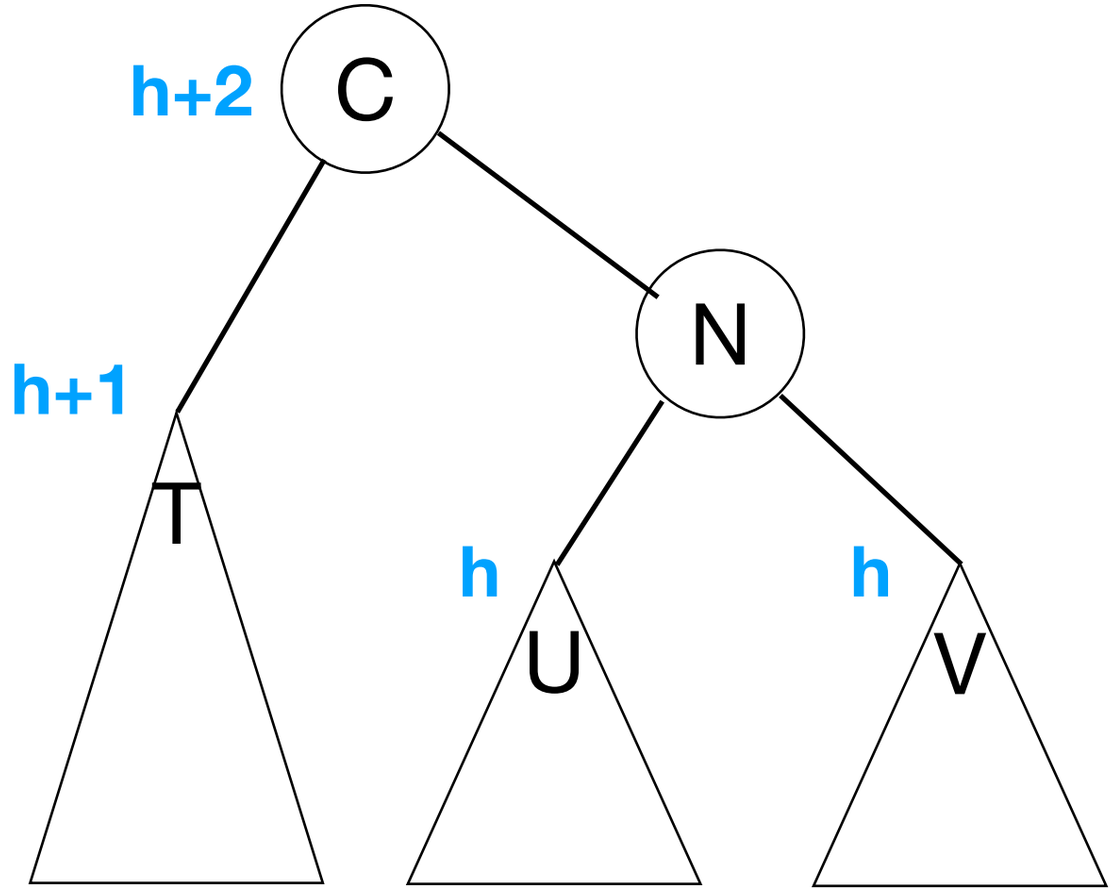
Solution: right rotate on N.



AVL Rebalance

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

Solution: right rotate on N.

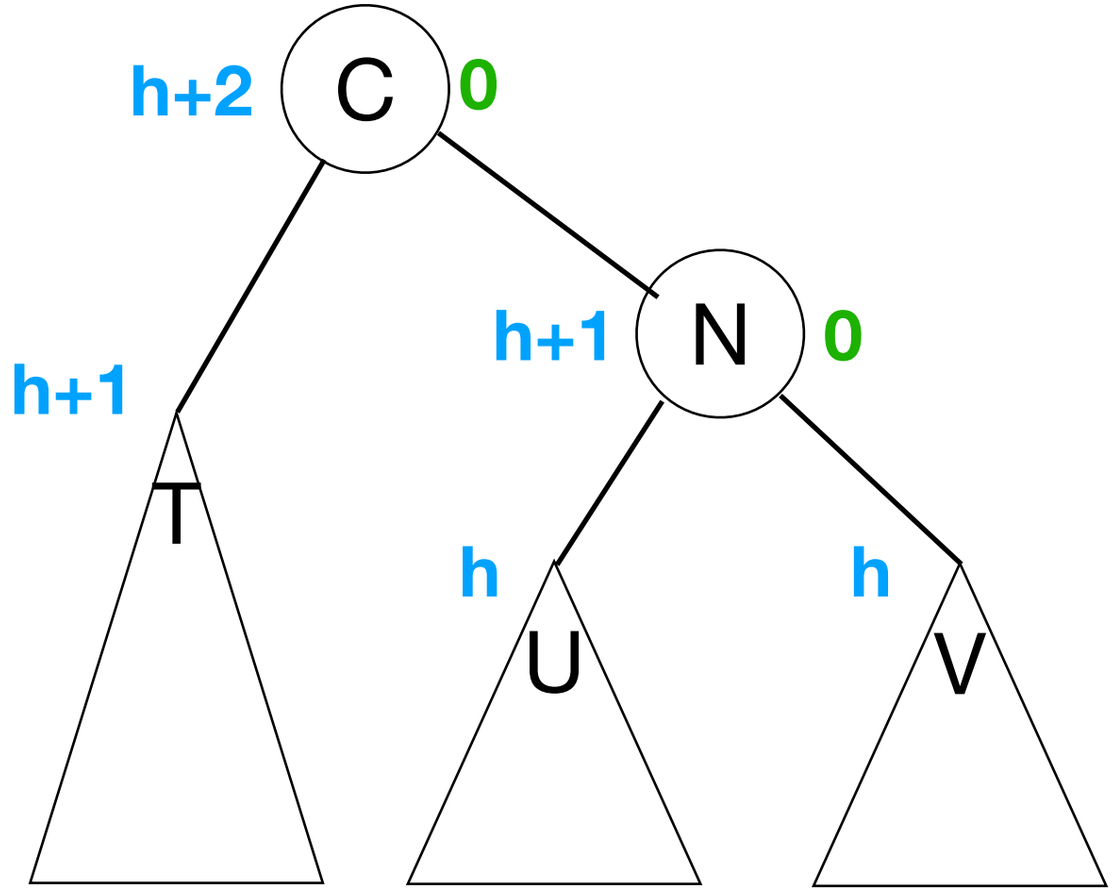


AVL Rebalance

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

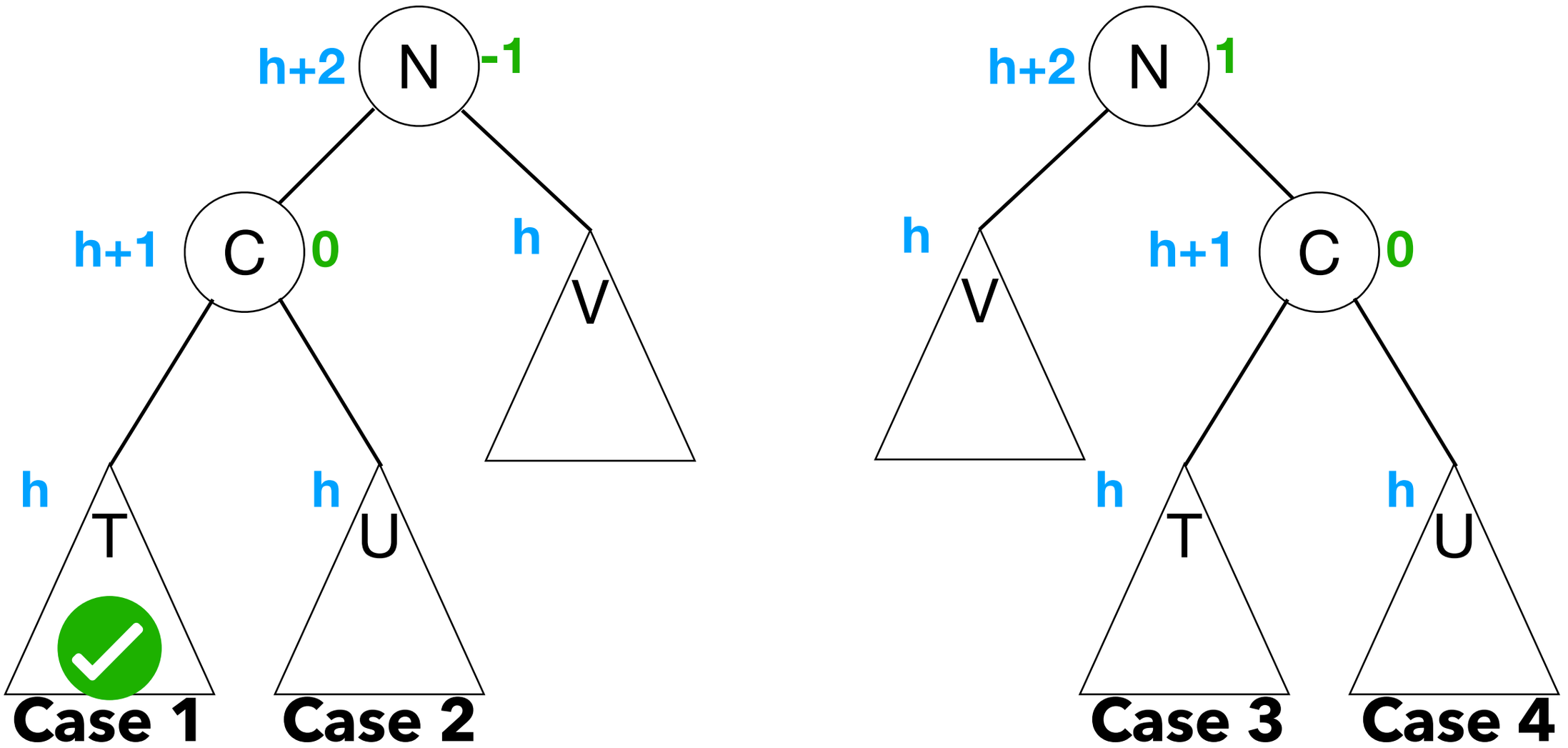
Solution: right rotate on N.

N is now AVL balanced.



AVL Rebalance

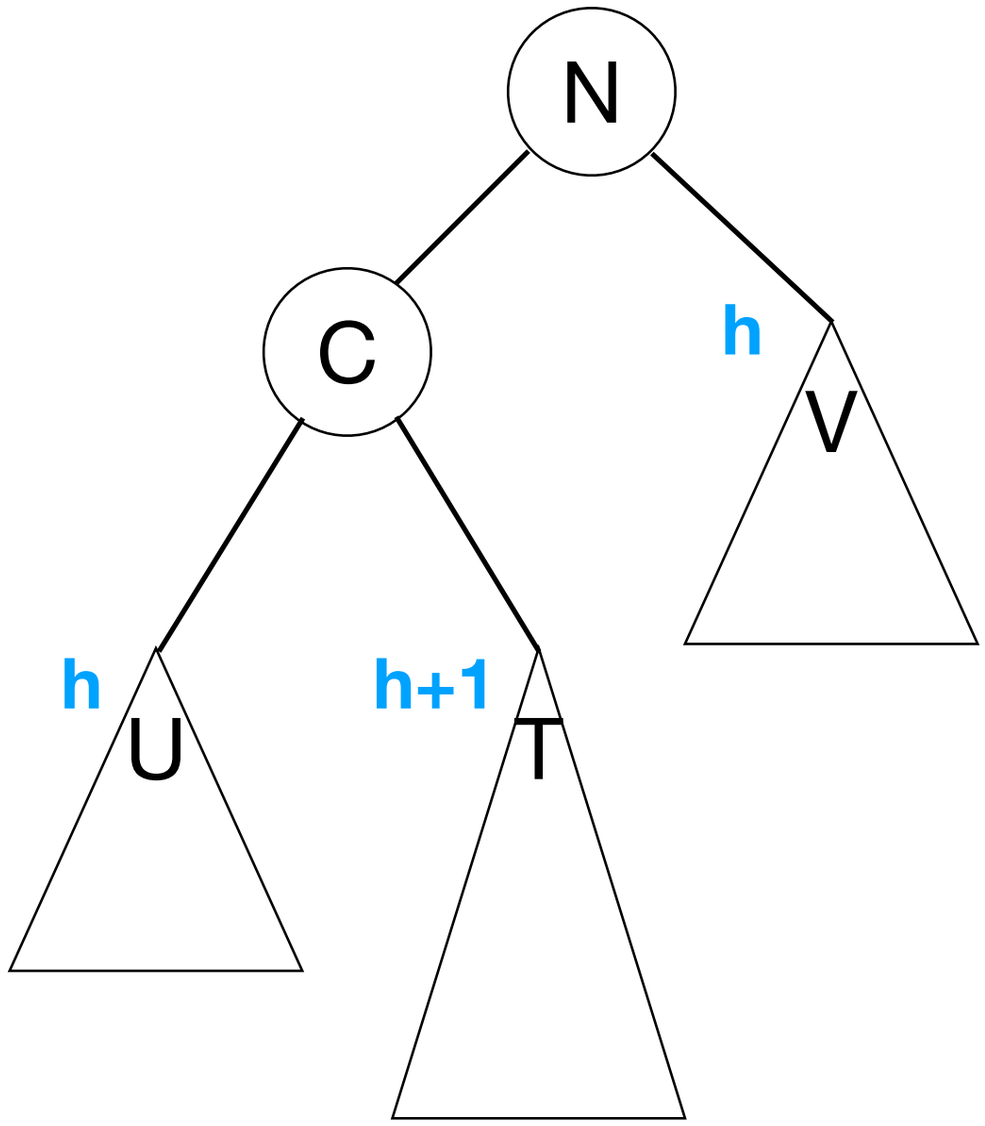
Before an insertion that unbalances n,
the tree must look like one of these:



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

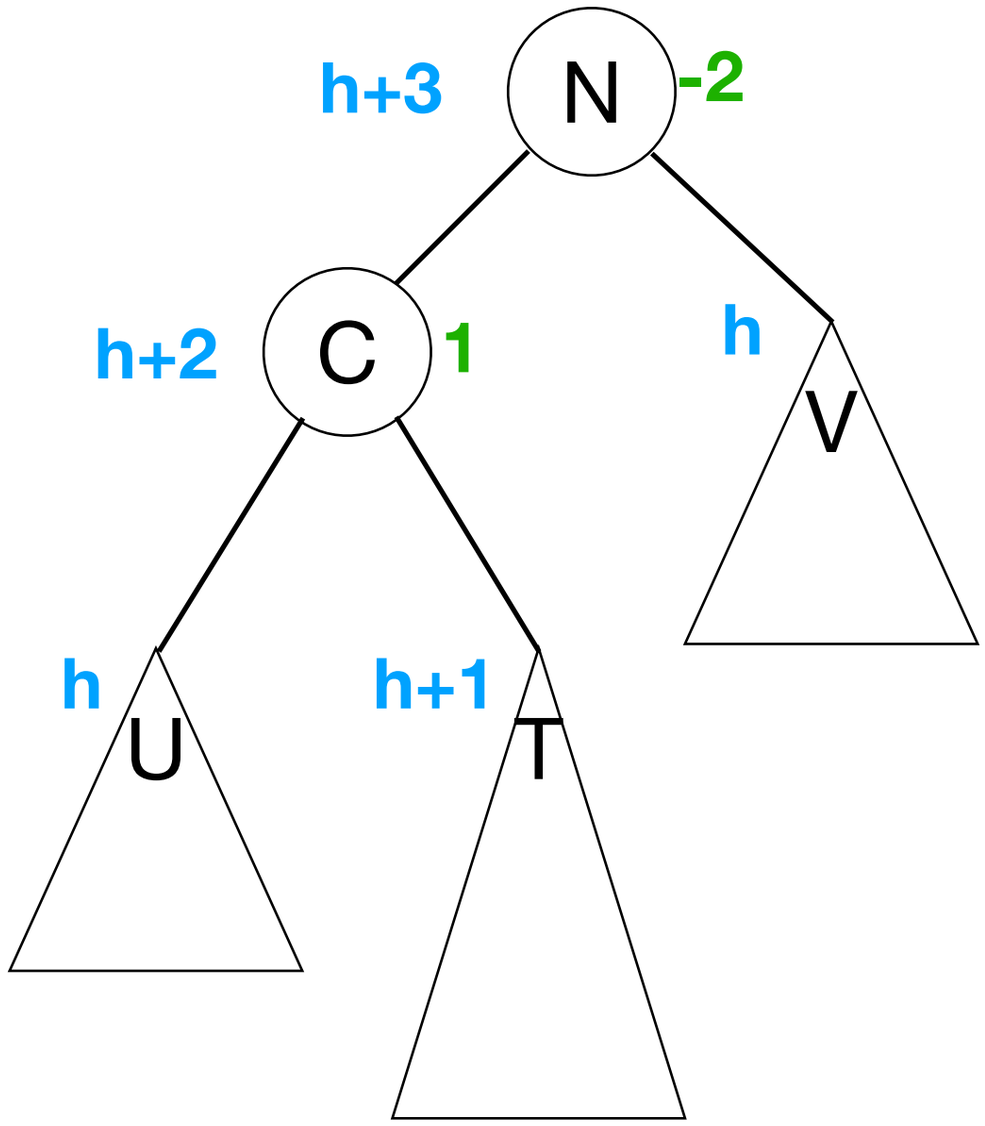
AVL Rebalance

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



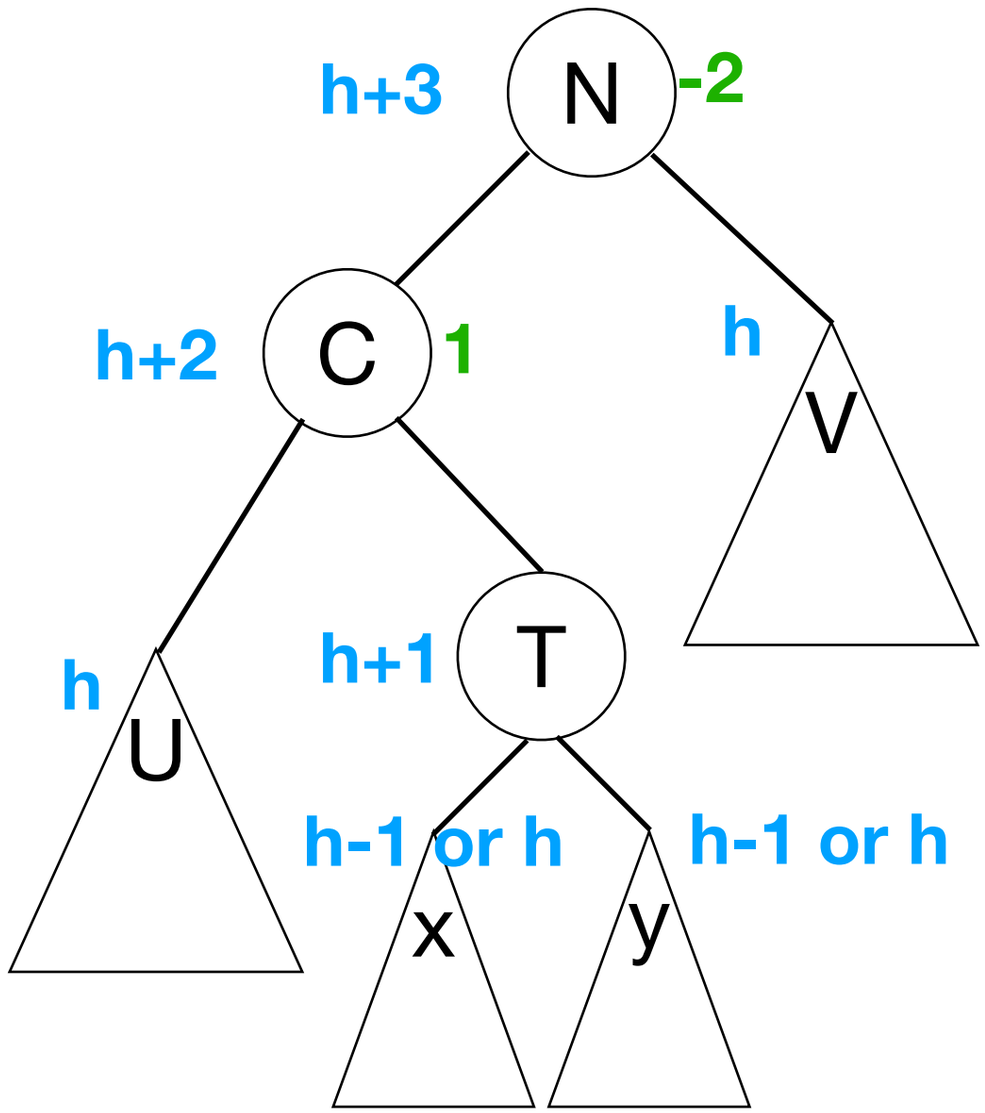
AVL Rebalance

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



AVL Rebalance

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

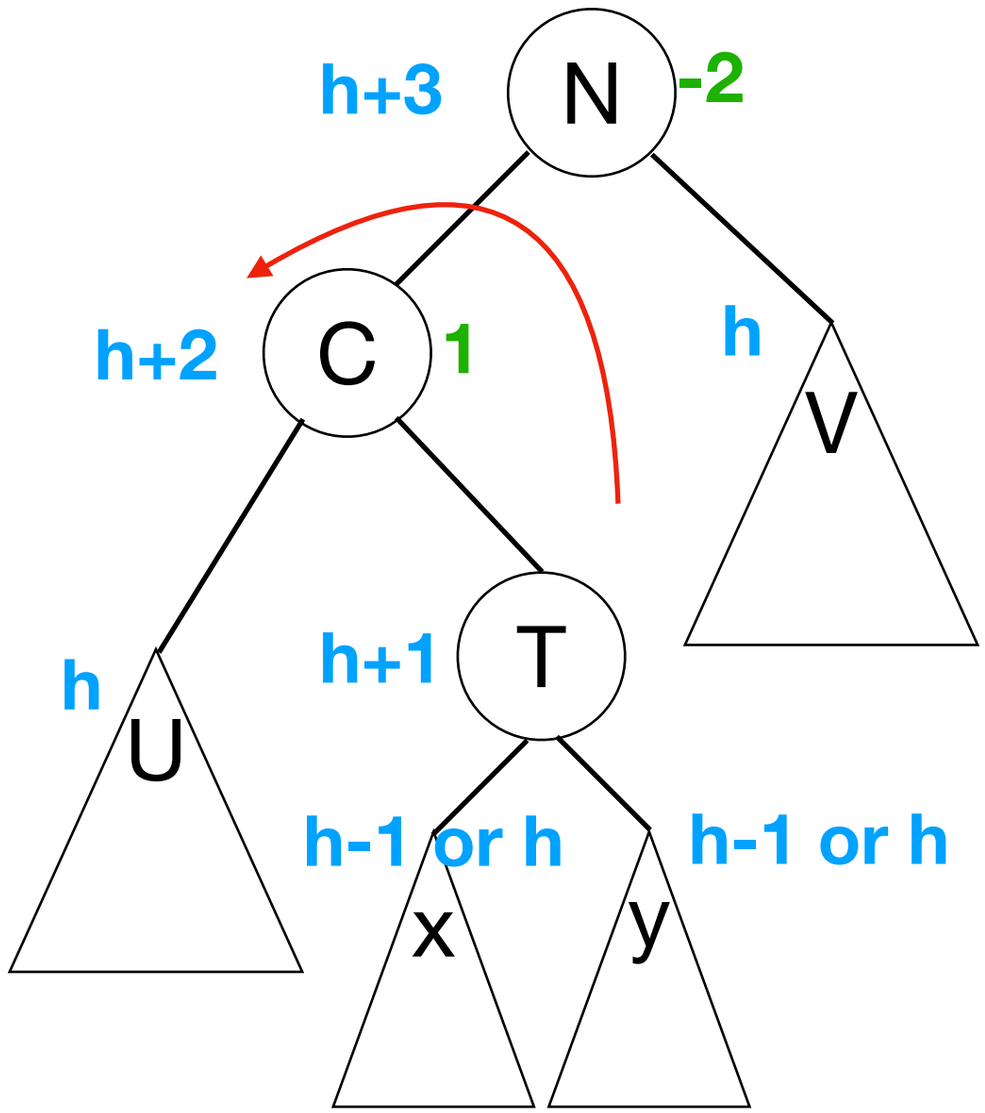


Solution - two rotations:

1. Left rotate C
2. Right rotate N

AVL Rebalance

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

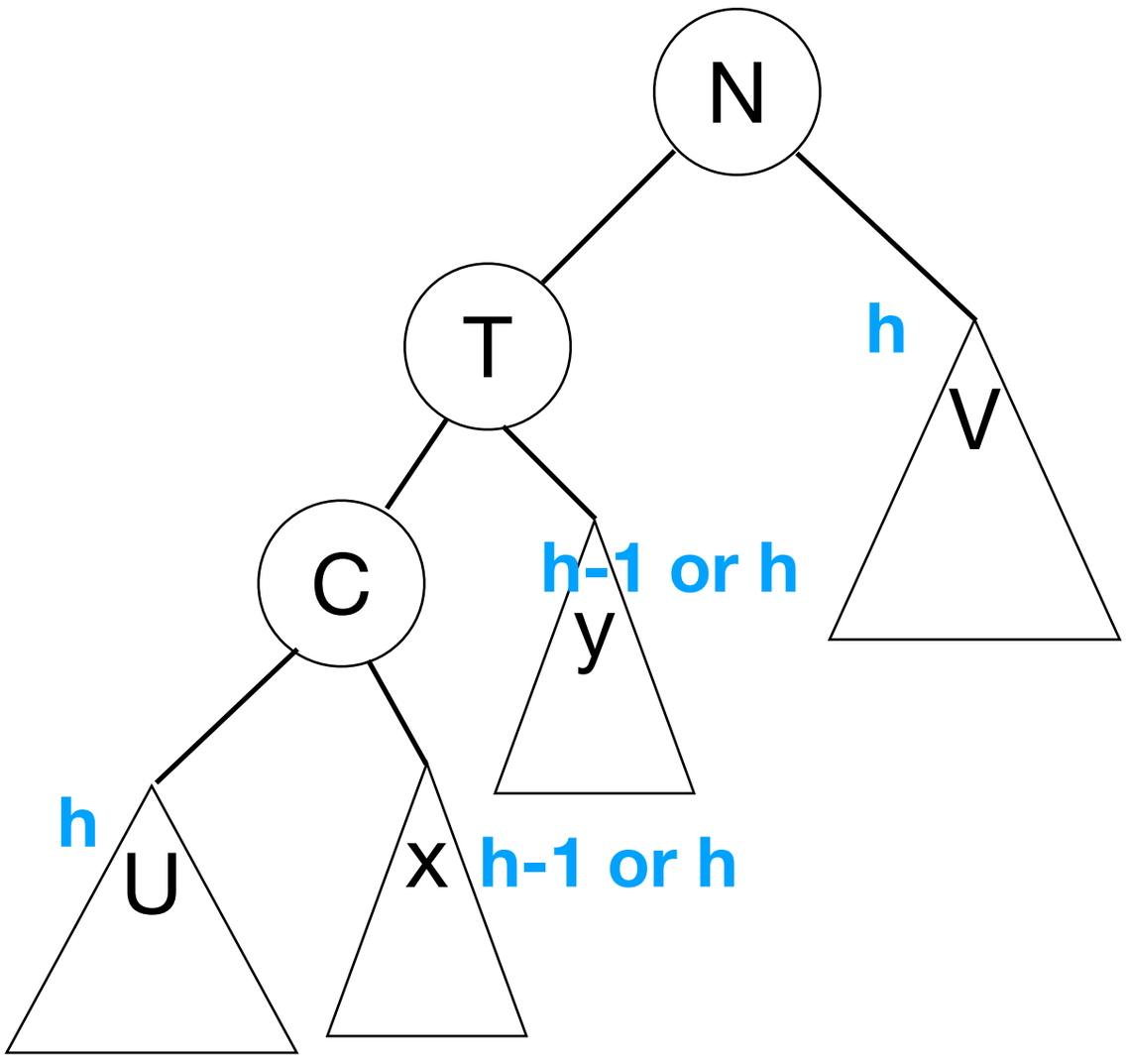


Solution - two rotations:

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

AVL Rebalance

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

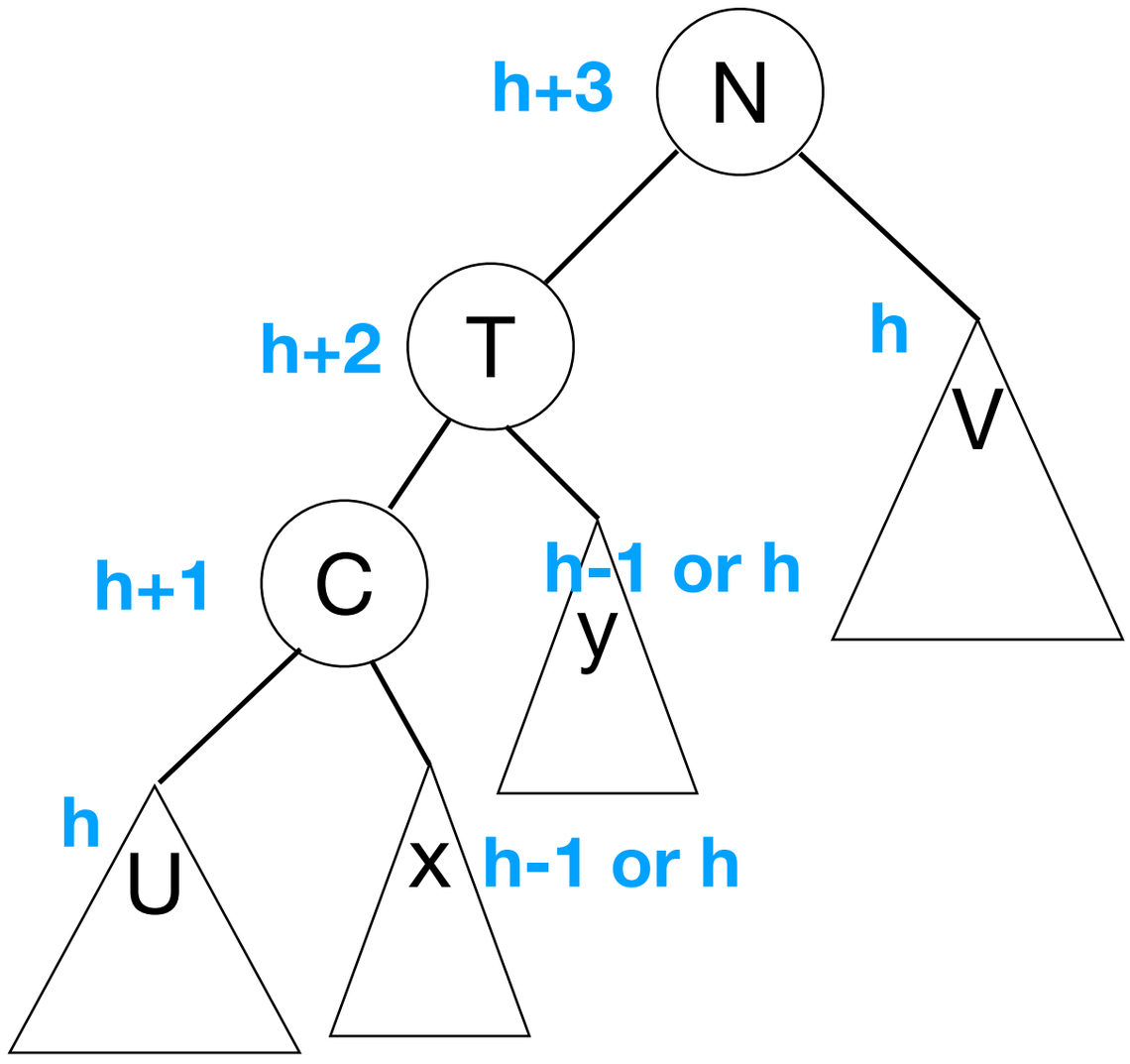


Solution - two rotations:

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

AVL Rebalance

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

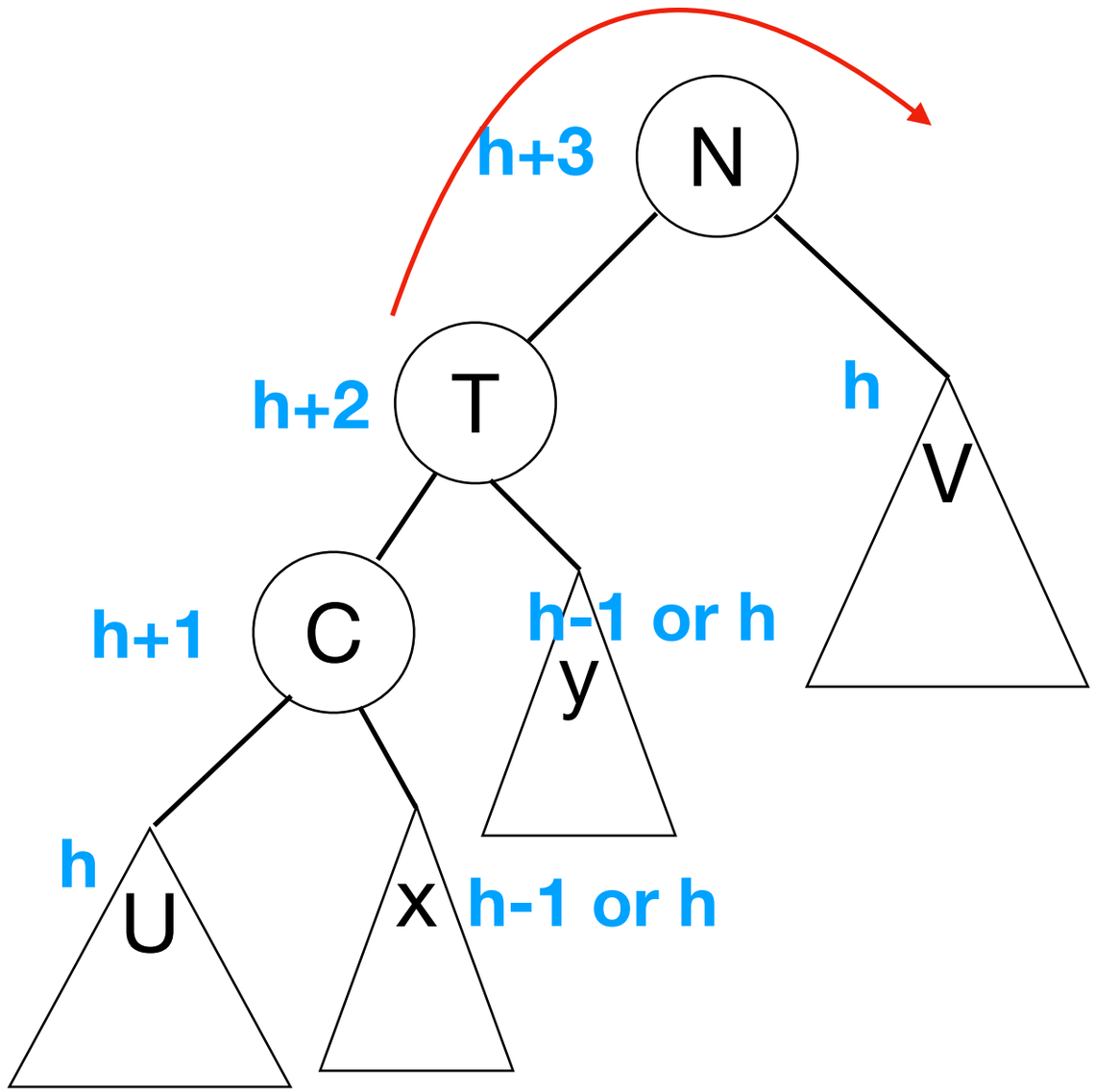


Solution - two rotations:

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

AVL Rebalance

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

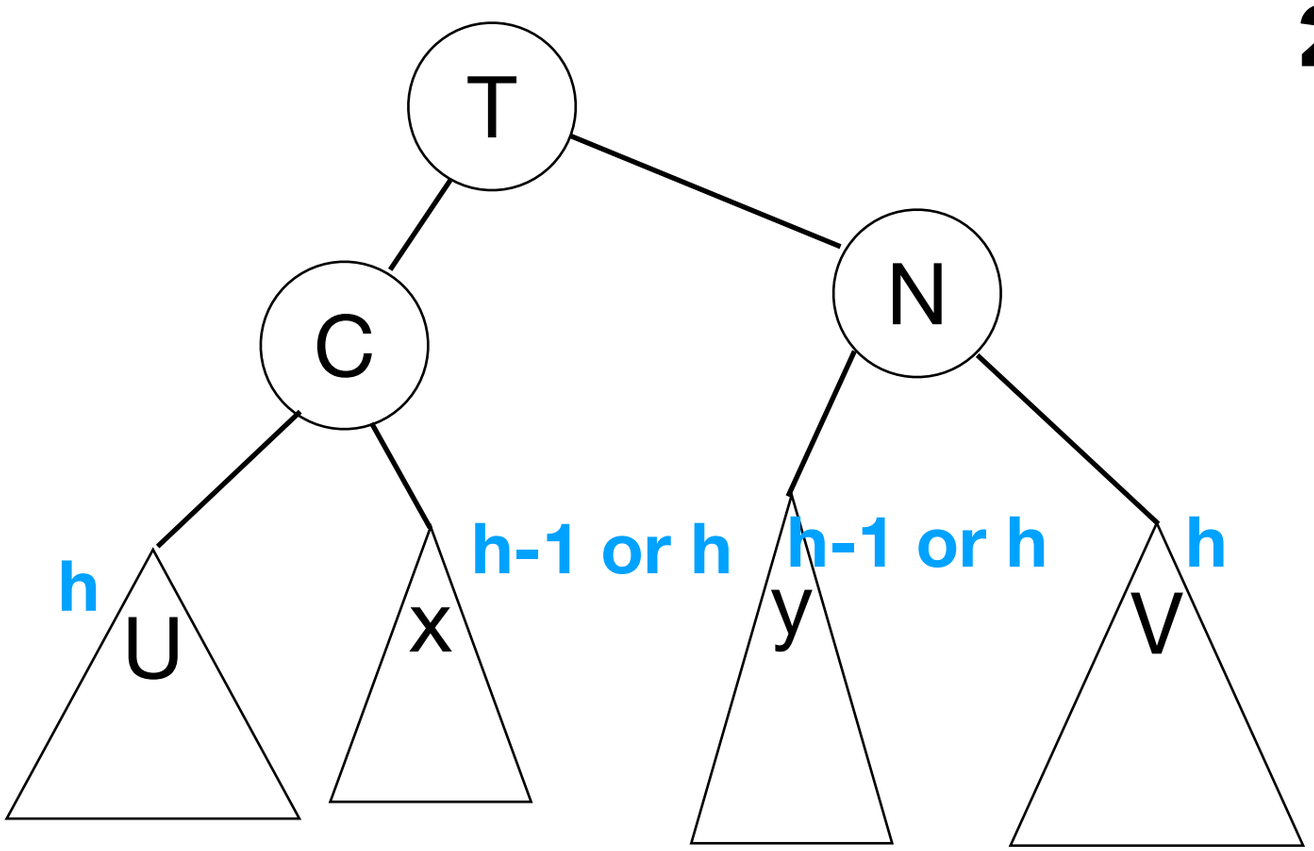


Solution - two rotations:

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

AVL Rebalance

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

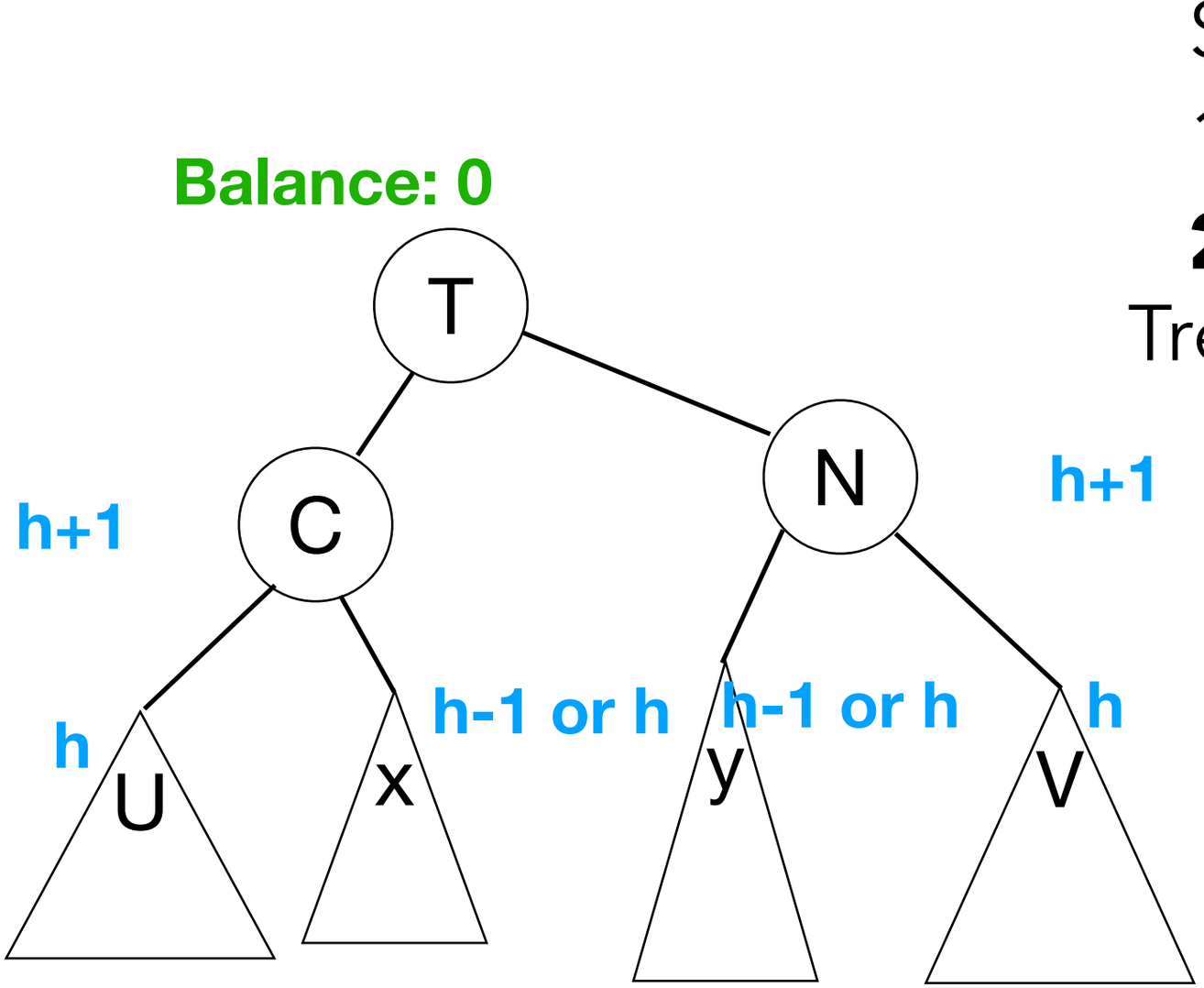


Solution - two rotations:

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

AVL Rebalance

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



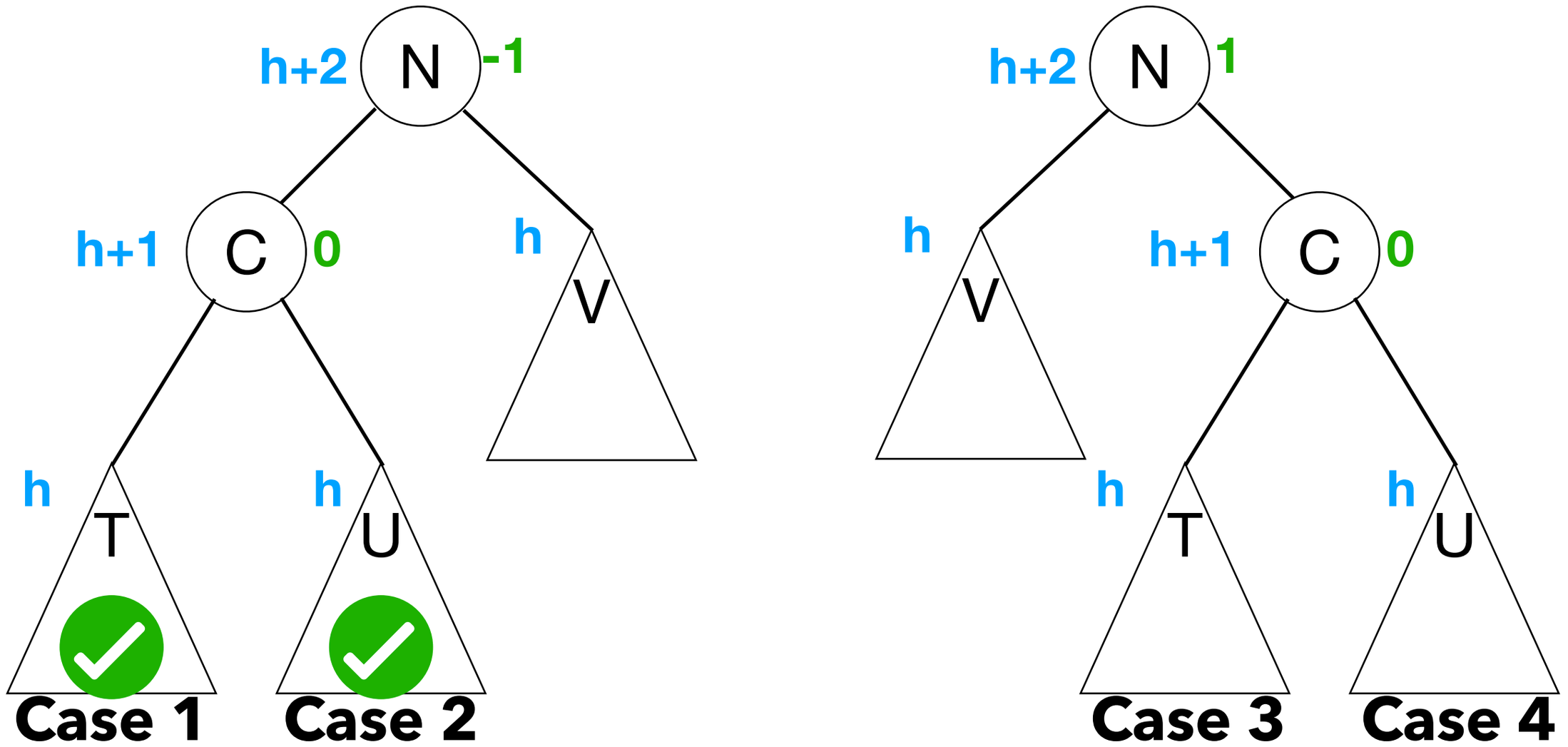
Solution - two rotations:

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

Tree is now AVL balanced.

AVL Rebalance

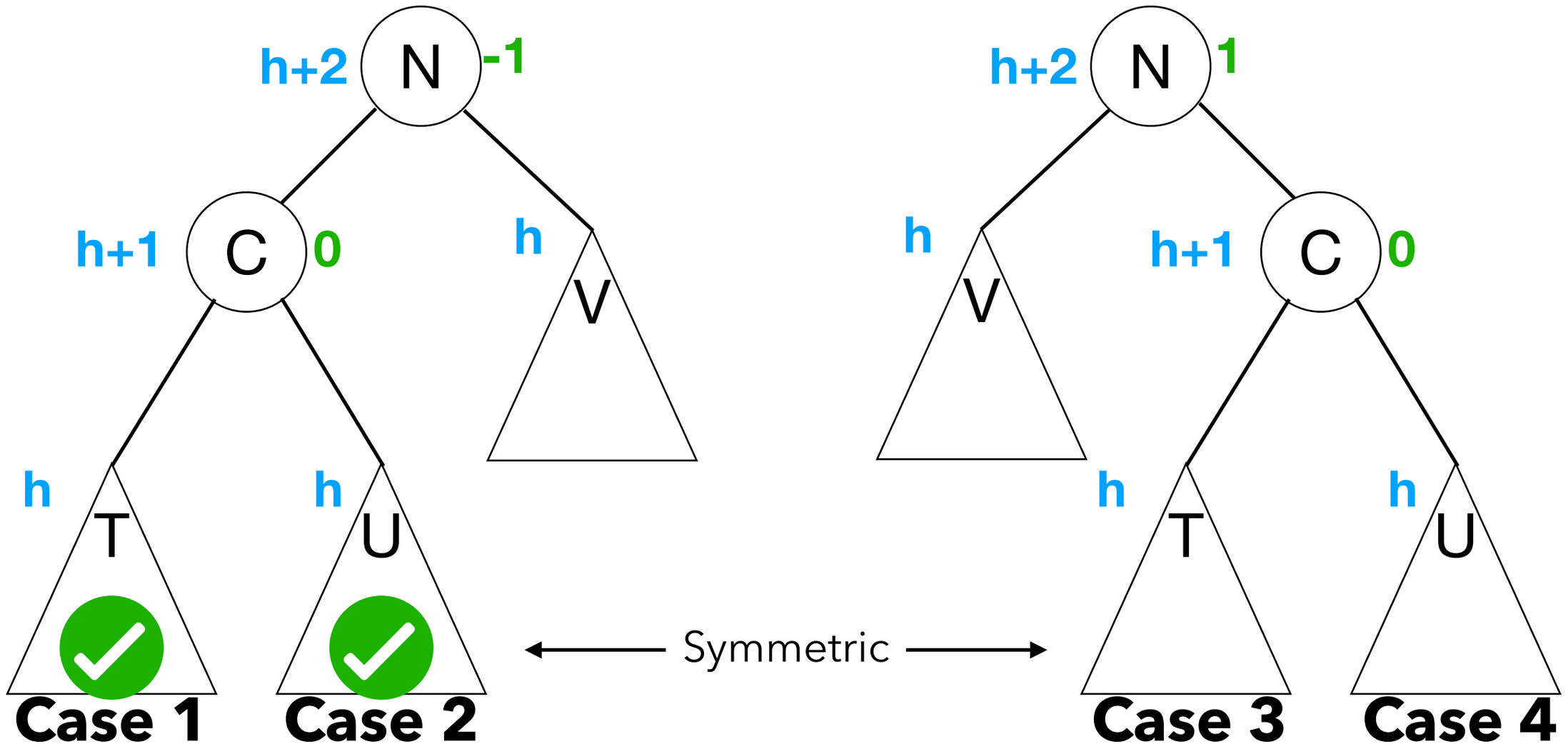
Before an insertion that unbalances n,
the tree must look like one of these:



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

AVL Rebalance

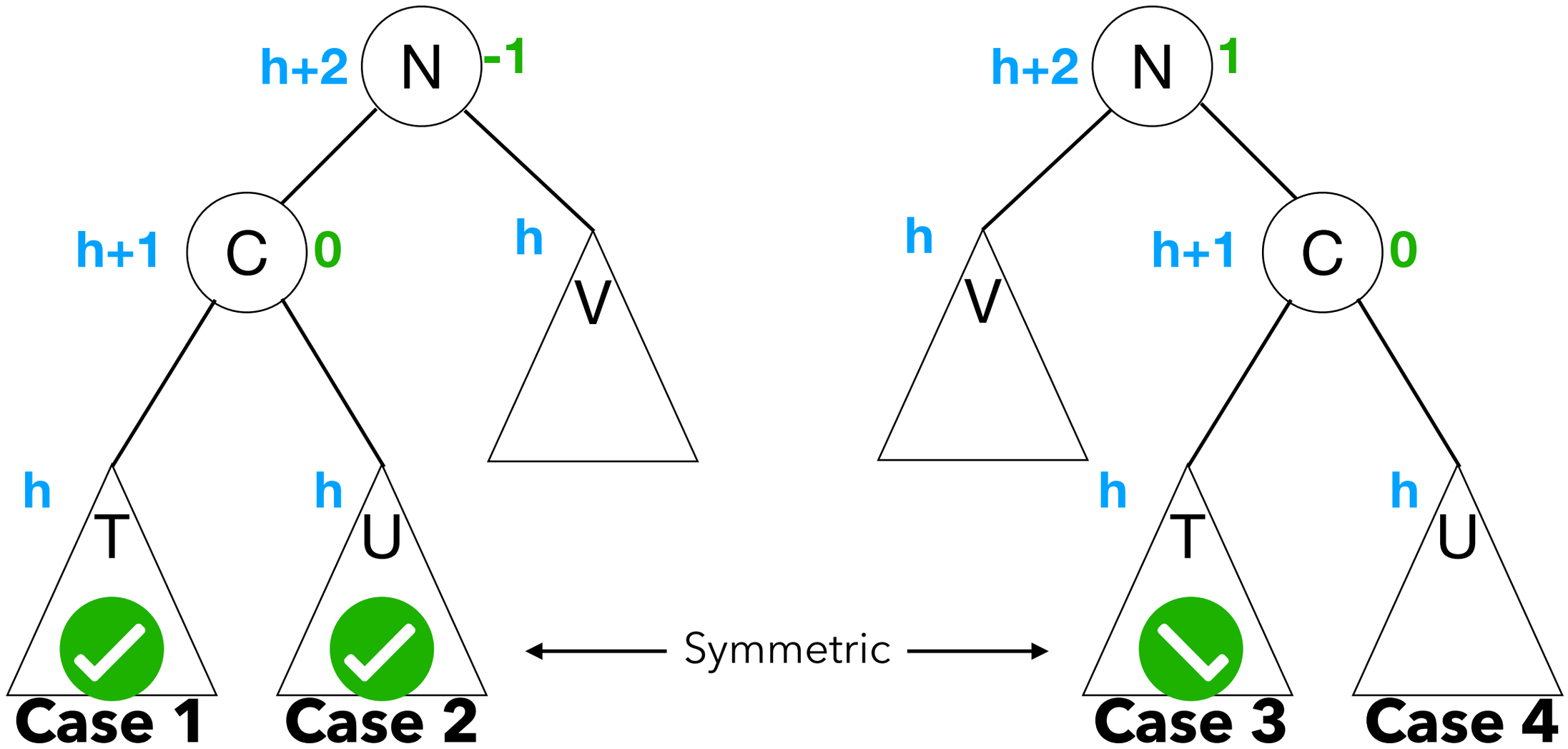
Before an insertion that unbalances n,
the tree must look like one of these:



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

AVL Rebalance

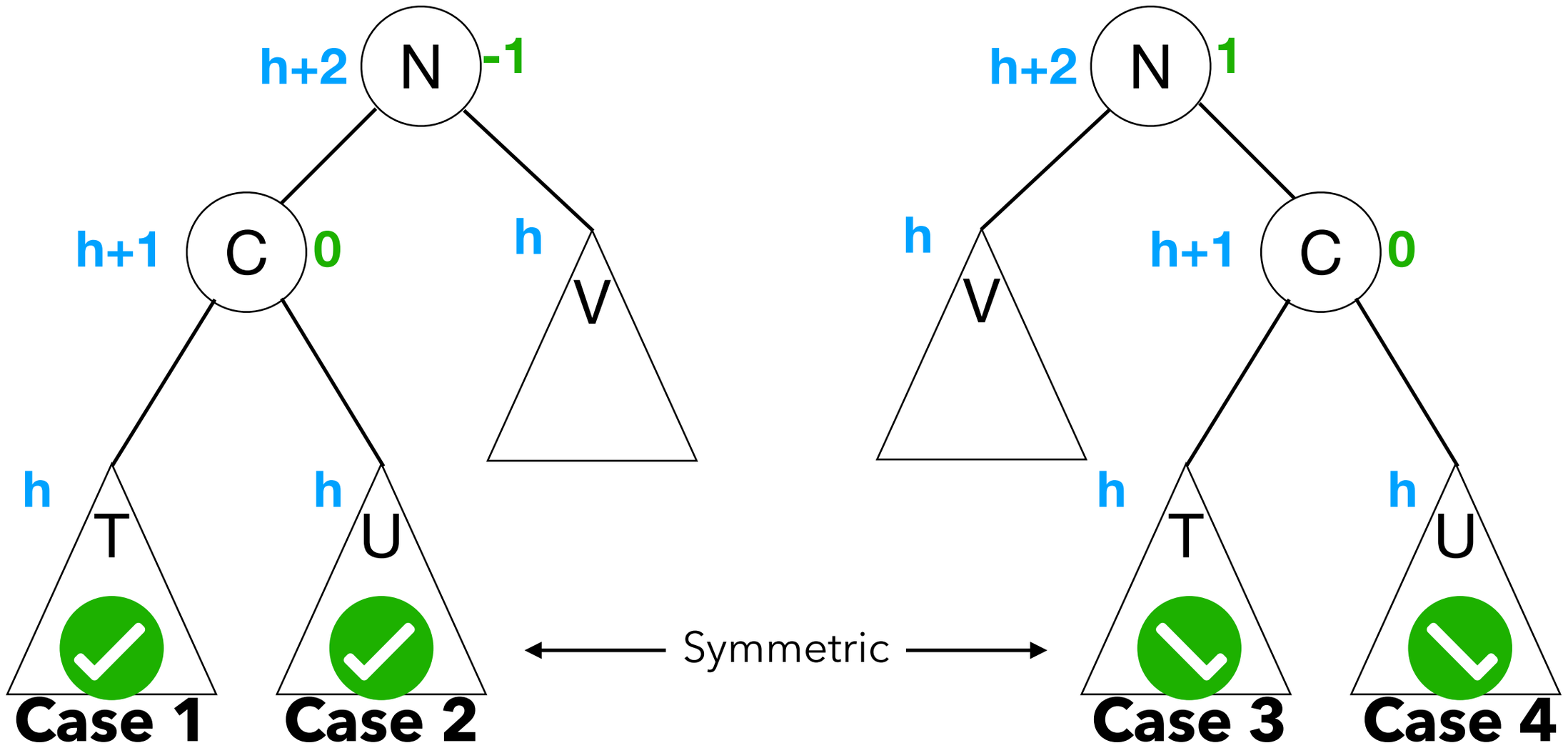
Before an insertion that unbalances n,
the tree must look like one of these:



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

AVL Rebalance

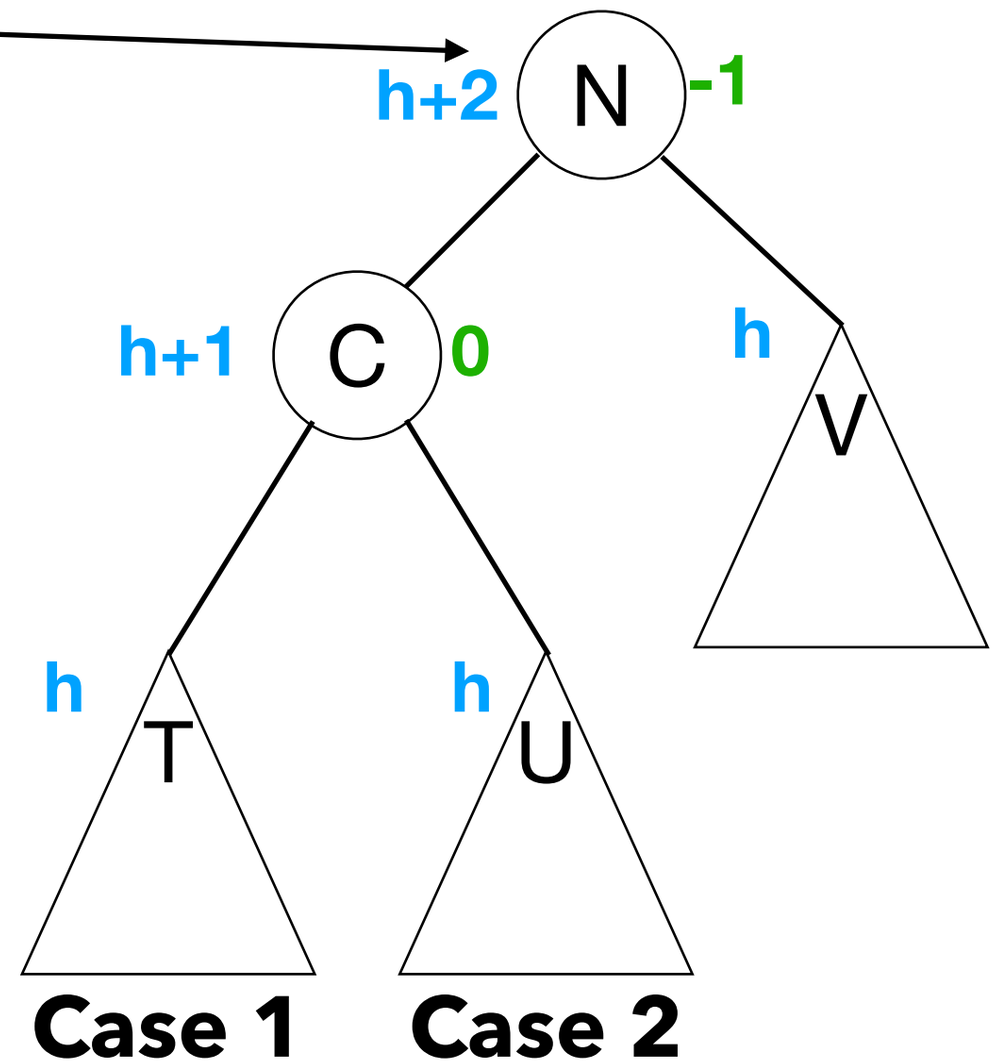
Before an insertion that unbalances n,
the tree must look like one of these:



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

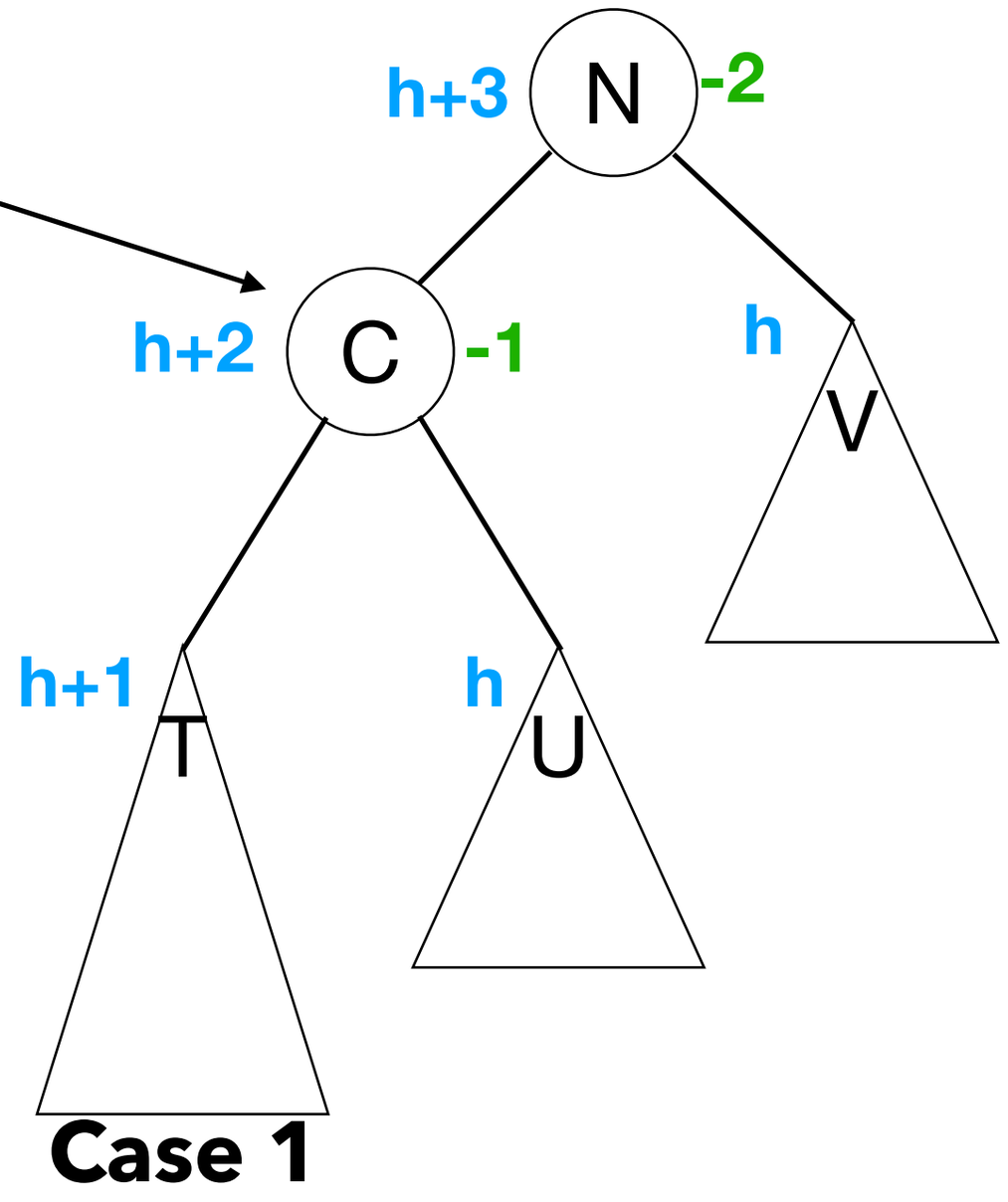
Implementation

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



Implementation

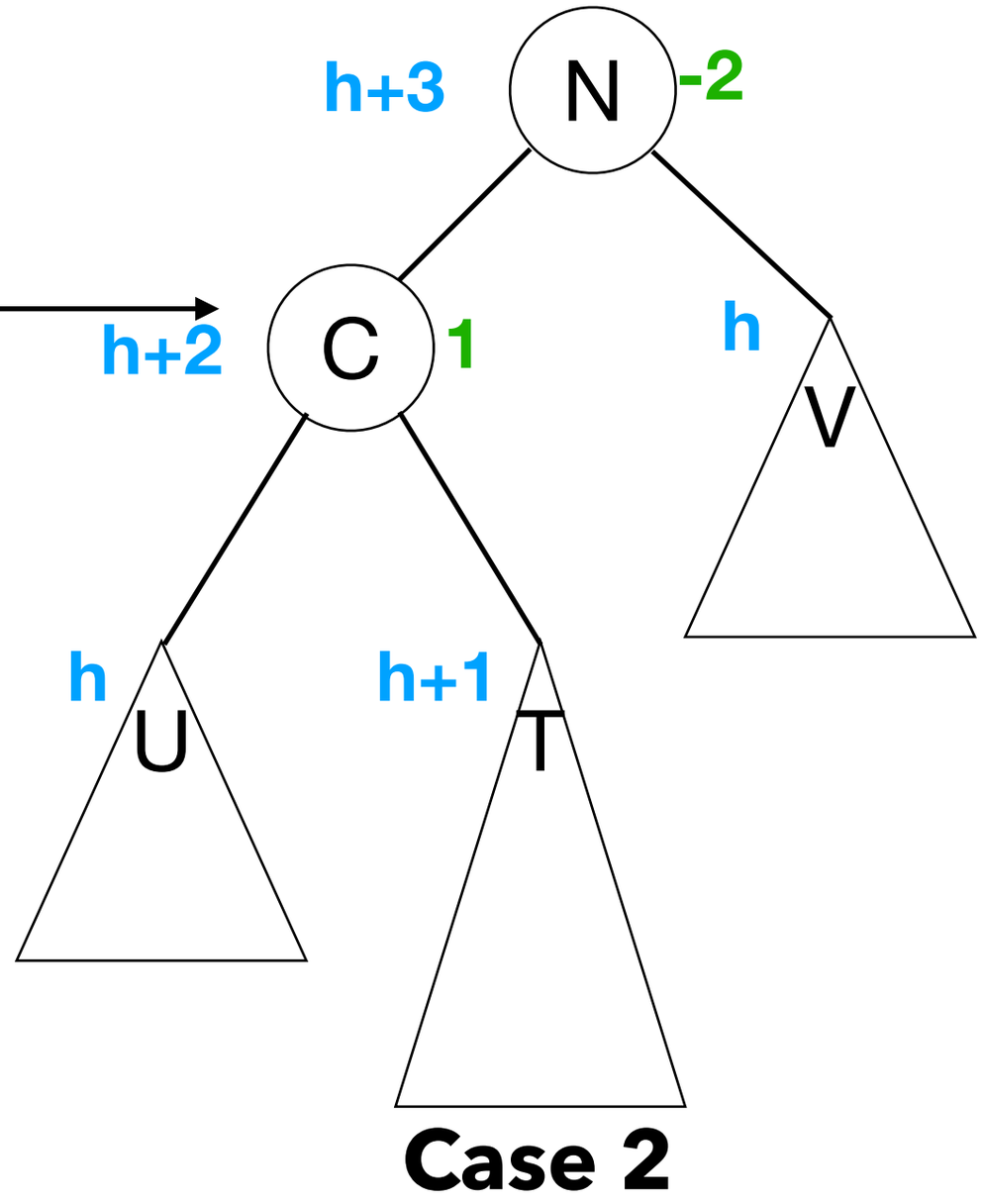
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1
    else:
      // case 2
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3
    else:
      // case 4
```



Case 1

Implementation

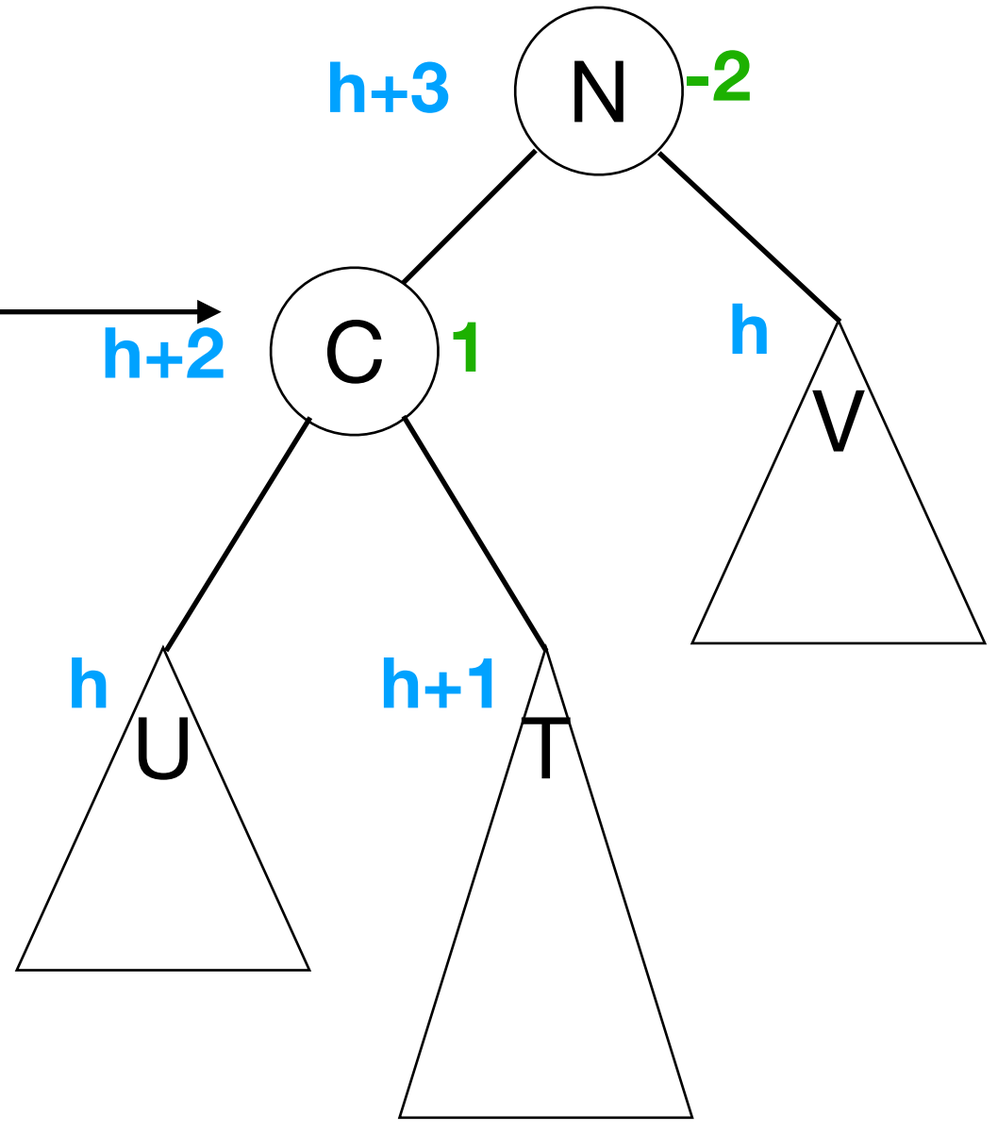
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1
    else:
      // case 2
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3
    else:
      // case 4
```



Case 2

Implementation

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



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
    else:
      // case 2
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3
    else:
      // case 4
```

Details - implementing `bal`:

- calculating height as in Lab 3 is O(n) - not good enough!
- Instead, nodes store their height. Need to update when the tree changes.
- Update each node's height **on the way up the tree**, calculating height using only the children's heights.

Removing from AVL Tree

- Similar to 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.