Topic 7
Lecture 7b
AVL Trees

CSCI 240

Data Structures and Algorithms

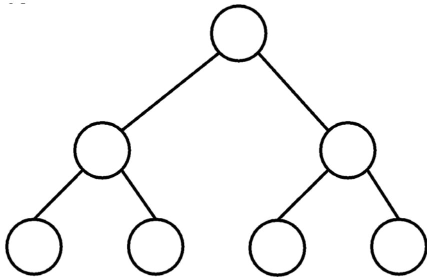
Prof. Dominick Atanasio

Today

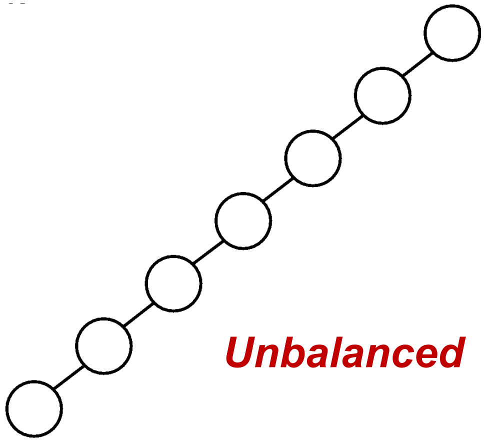
- Agenda
  - AVL Trees
    - Definition
    - Operations in AVL
    - Implementations
    - Efficiency of operations

Problem in an Ordinary BST

- The problem in an ordinary BST
  - Possible to form several differently shaped binary search trees (from the same collection of data)



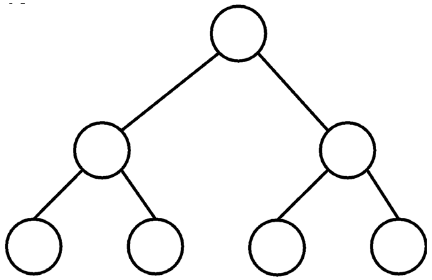
Height Balanced



Unbalanced

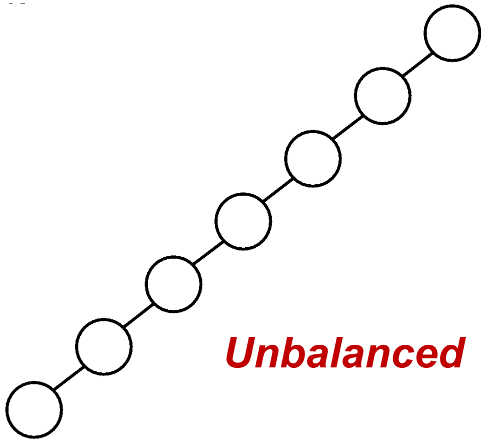
- The problem in an ordinary BST
  - Possible to form several differently shaped binary search trees (from the same collection of data)

- Most operations on a BST take time proportional to the height of the tree, so it is desirable to keep the height small.



Height Balanced

Algorithm	Average	Worst Case
Space	$\Theta(n)$	$O(n)$
Search	$\Theta(\log n)$	$O(n)$
Insert	$\Theta(\log n)$	$O(n)$
Delete	$\Theta(\log n)$	$O(n)$



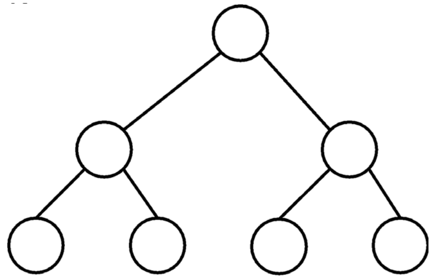
Unbalanced

Avoid the Worst Case

- Self-balancing search trees solve this problem by performing transformations on the tree at key times, in order to reduce the height.

- Although a certain overhead is involved, it is justified in the long run by ensuring fast execution of later operations.

- The height must always be at most the ceiling of $\log_2 n$.



- Types of Balanced Search Trees
  - AVL trees
  - Red-black trees
  - B-trees
    - 2-3 Trees
    - 2-4 Trees
    - Balanced trees of order m

- Balanced Search Trees are not always so precisely balanced, since it can be expensive to keep a tree at minimum height at all times;
  - Most algorithms keep the height within a constant factor of this lower bound.

Balanced Search Trees

- Types of Balanced Search Trees
  - AVL trees
  - Red-black trees
  - B-trees
    - 2-3 Trees
    - 2-4 Trees
    - Balanced trees of order m

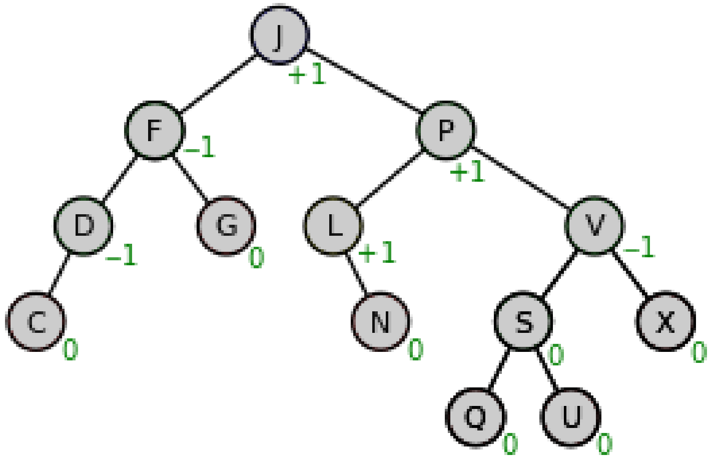
Binary Search Trees

Multiway Search Trees

- Balanced Search Trees are not always so precisely balanced, since it can be expensive to keep a tree at minimum height at all times;
  - Most algorithms keep the height within a constant factor of this lower bound.

AVL Trees

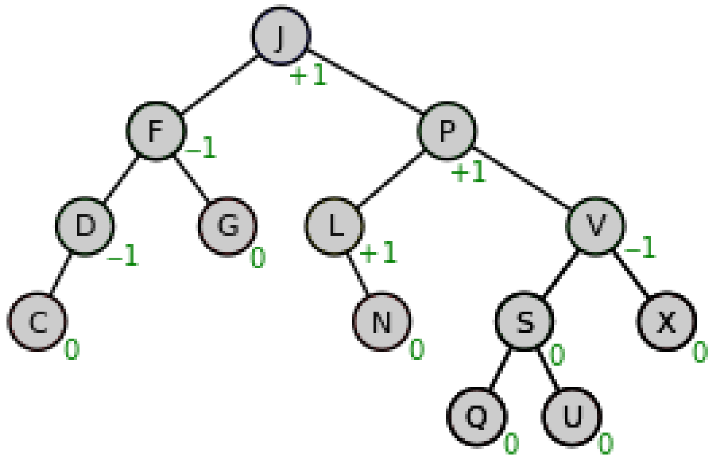
- AVL trees are self-balancing binary search trees.
  - Named after Georgy **A**delson-**V**elskii and Evgenii **L**andis.
- The idea of AVL
  - Rearranging its nodes whenever it becomes unbalanced.
  - The balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with a balance factor 1, 0, or -1 is considered balanced.



BalanceFactor(N) = |N.rightChild.height - N.leftChild.height|

AVL Trees

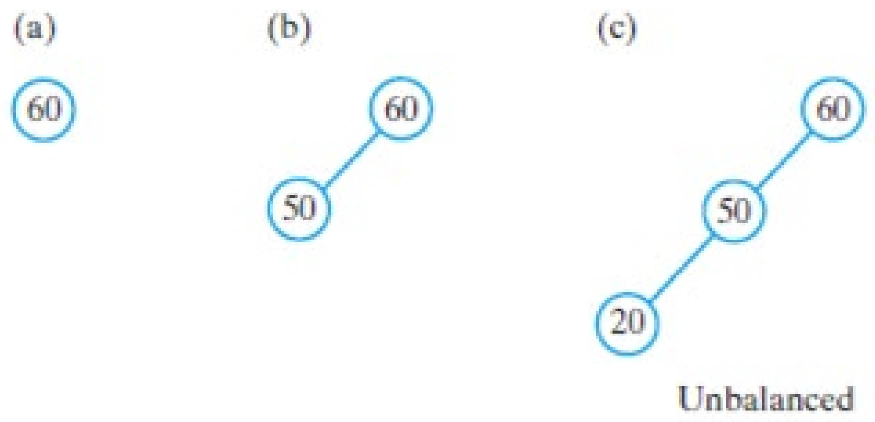
- Properties of an AVL tree:
  - In an AVL tree, the heights of the two child subtrees of any node differ by at most one; (height-balanced)
  - Add, remove, and lookup all take O(log2n) time in both the average and worst cases, where n is the number of nodes in the tree.
  - Add and remove may require the tree to be rebalanced by one or more tree rotations.



AVL Trees

- Add or remove a node may cause balance factor to become 2 for some node

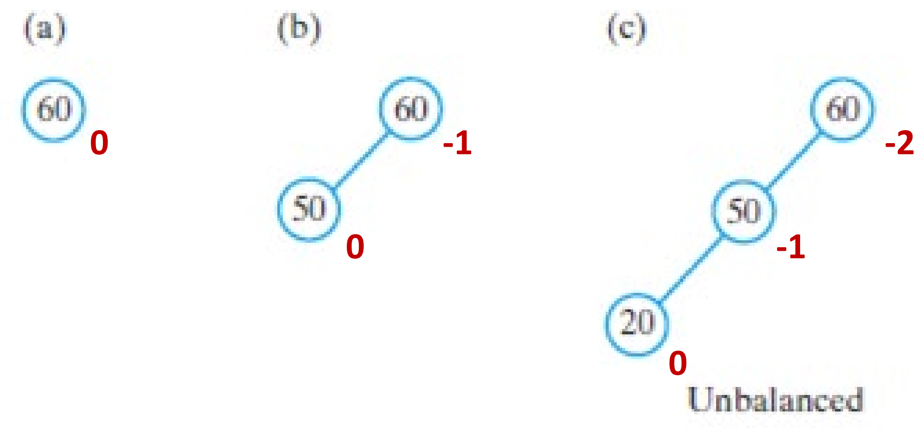
- Example in adding a node



AVL Trees

- Add or remove a node may cause balance factor to become 2 for some node

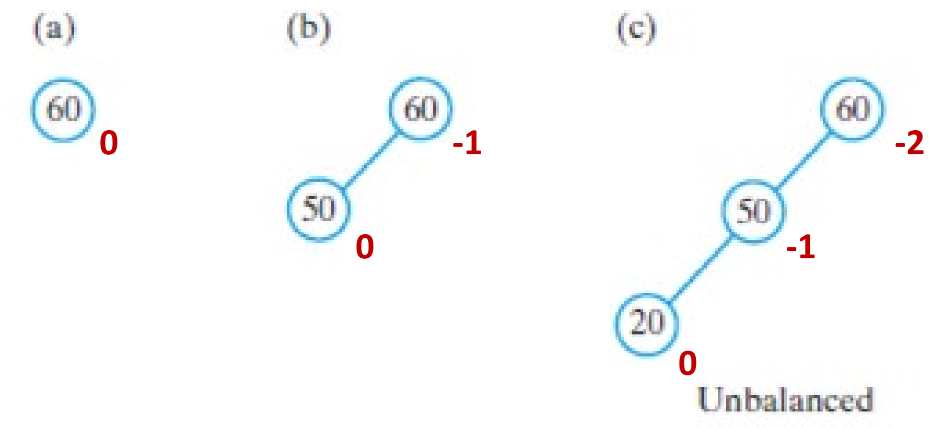
- Example in adding a node



- Add or remove a node may cause balance factor to become 2 for some node

- Example in adding a node

In a AVL tree, rotations will be carried out to arrange its nodes to restore balance.

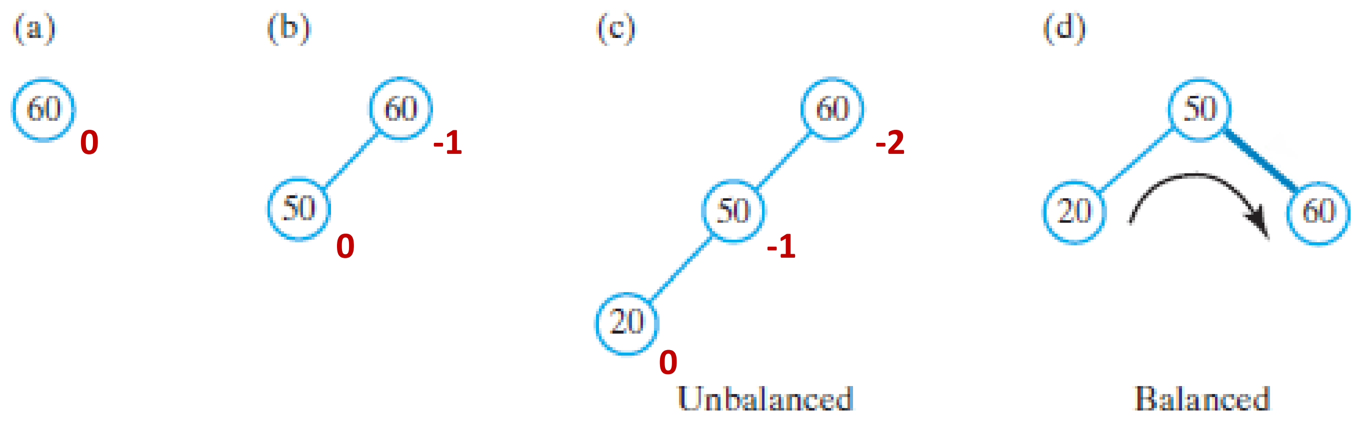


AVL Trees

- Add or remove a node may cause balance factor to become 2 for some node

- Example in adding a node

In a AVL tree, rotations will be carried out to arrange its nodes to restore balance.

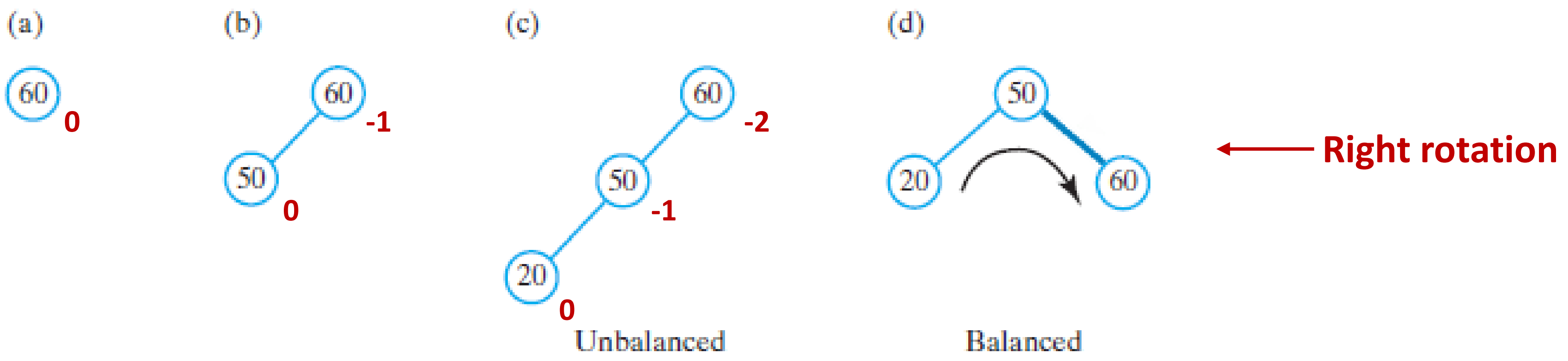


AVL Trees

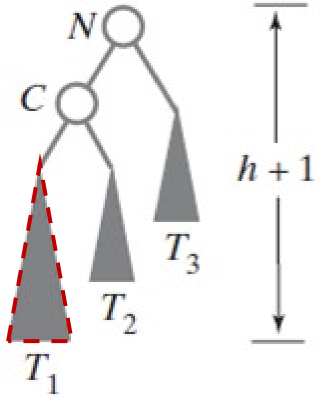
- Add or remove a node may cause balance factor to become 2 for some node

- Example in adding a node

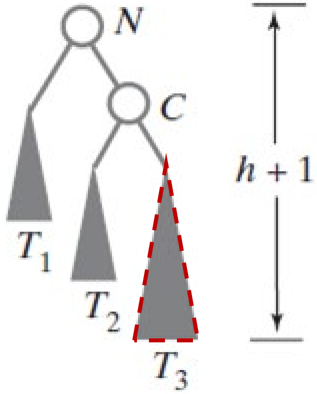
In a AVL tree, rotations will be carried out to arrange its nodes to restore balance.



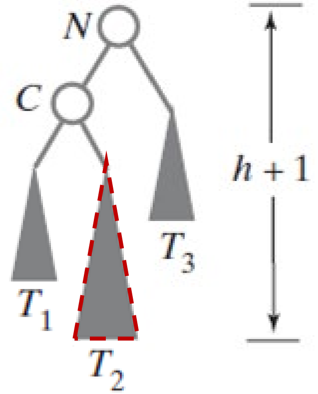
- There are four cases for the cause of the imbalance at node *N*:
  - Outside Branches which require single rotation
    Case 1: The left subtree of *N*'s left child (right rotation)
    Case 2: The right subtree of *N*'s right child (left rotation)
  - Inside Branches which require double rotation
    Case 3: The right subtree of *N*'s left child (left-right rotation)
    Case 4: The left subtree of *N*'s right child (right-left rotation)



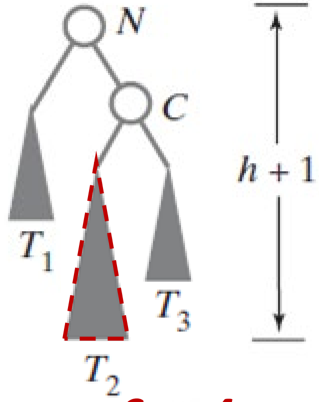
Case 1



Case 2



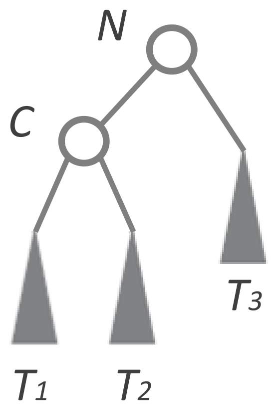
Case 3



Case 4

Adding a Node in AVL Trees

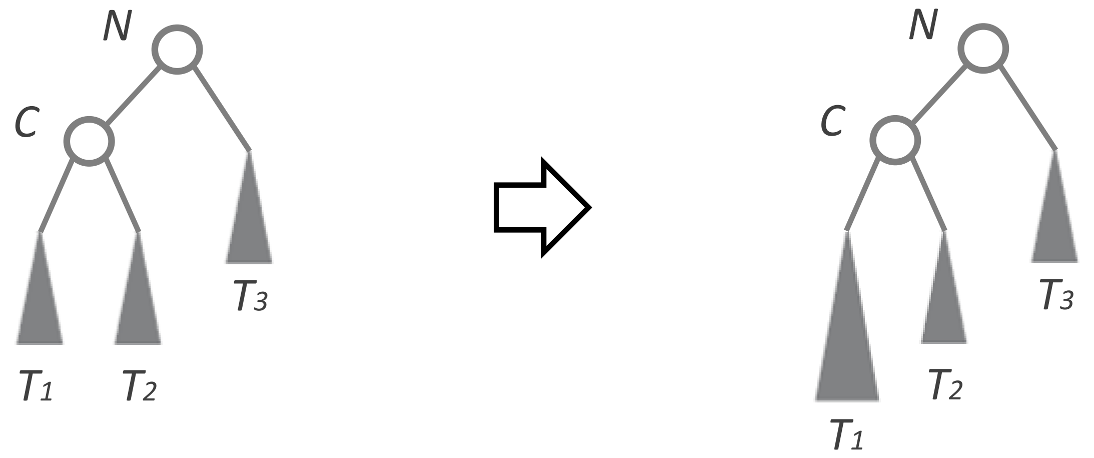
- Outside Branches (which require single rotation) :
  - **Case 1:** The left subtree of N's left child (right rotation)



Consider a valid AVL tree

Adding a Node in AVL Trees

- Outside Branches (which require single rotation) :
  - **Case 1:** The left subtree of N's left child (right rotation)

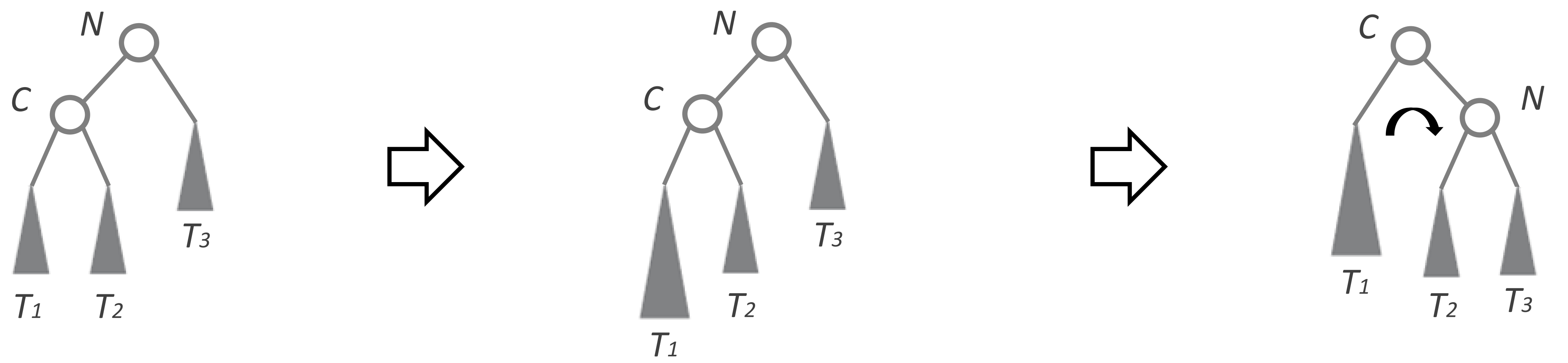


Consider a valid AVL tree

Add a node in T_1
(which changes the balance factor of node N)

Adding a Node in AVL Trees

- Outside Branches (which require single rotation) :
  - **Case 1:** The left subtree of N's left child (**right rotation**)



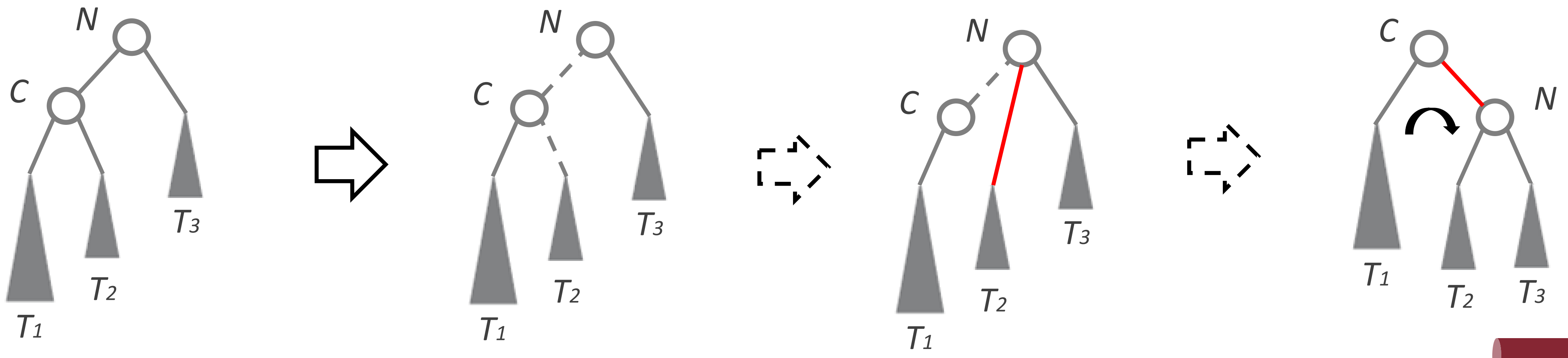
Consider a valid AVL tree	***Add a node in T_1 (which changes the balance factor of node N)***	***Perform right rotation about C (to restore the balance of the tree)***

The Algorithm Performs Right Rotation

- Outside Branches (which require single rotation) :
  - **Case 1:** The left subtree of N's left child (**right rotation**)

Algorithm **rotateRight(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
// in the left subtree of nodeN*'s left child.*

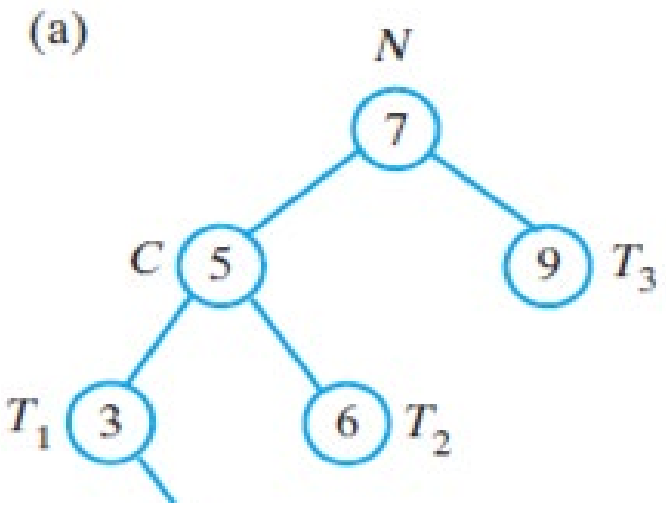
nodeC = *left child of* nodeN
Set nodeN*'s left child to* nodeC*'s right child*
Set nodeC*'s right child to* nodeN
return nodeC



```
/** corrects  for an imbalance in N due to
     an addition to the left subtree of N’s left child
*/
Algorithm rotateRight(N){
    C = N.leftChild
    N.leftChild = C.rightChild
    C.rightChild = N
    return N
}
```

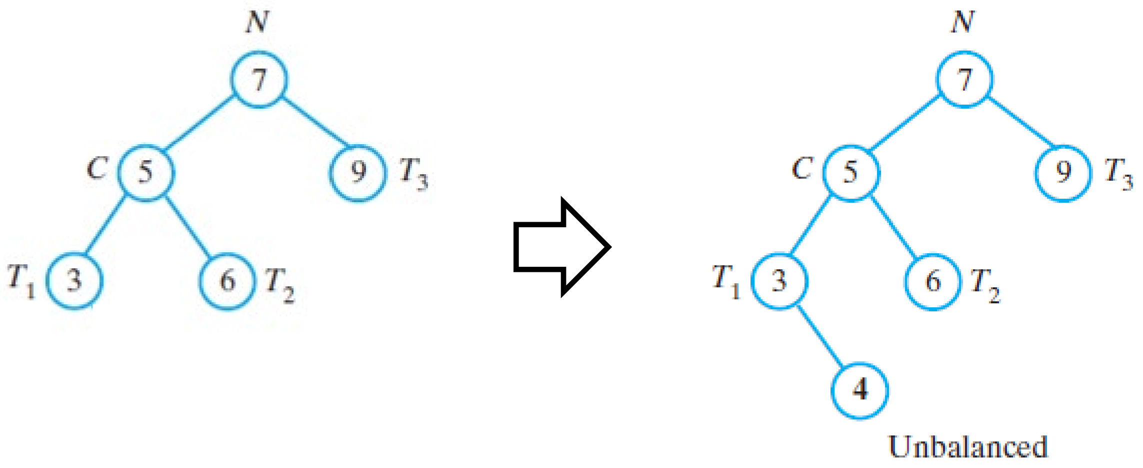
```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN’s left child.

nodeC = left child of nodeN
Set nodeN’s left child to nodeC’s right child
Set nodeC’s right child to nodeN
return nodeC
```

- Adding 4 into the AVL change the balance factor of node N



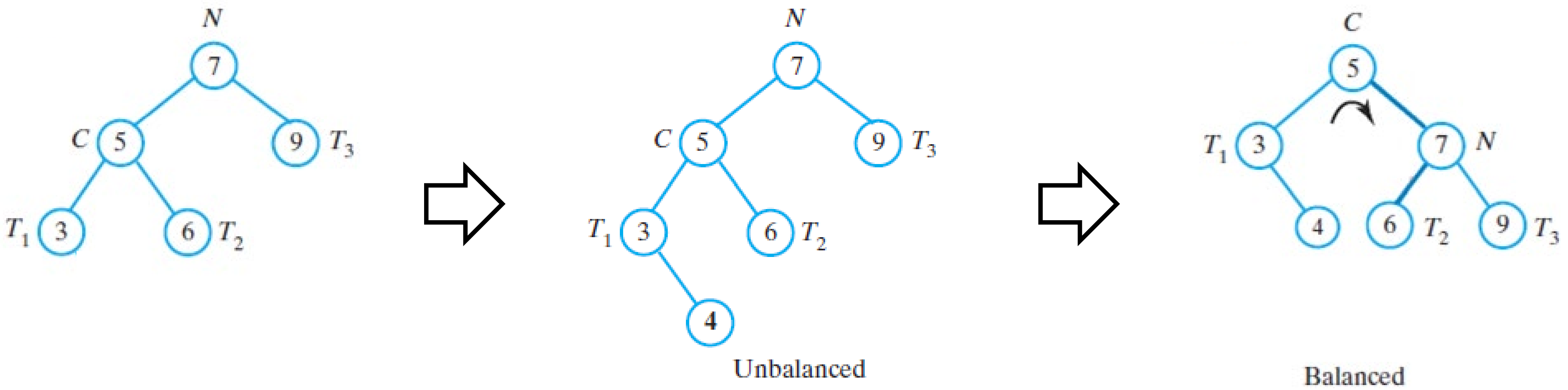
In-Class Exercise

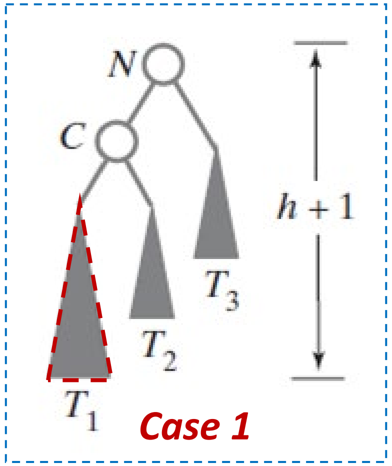
- Adding 4 into the AVL change the balance factor of node N



In-Class Exercise

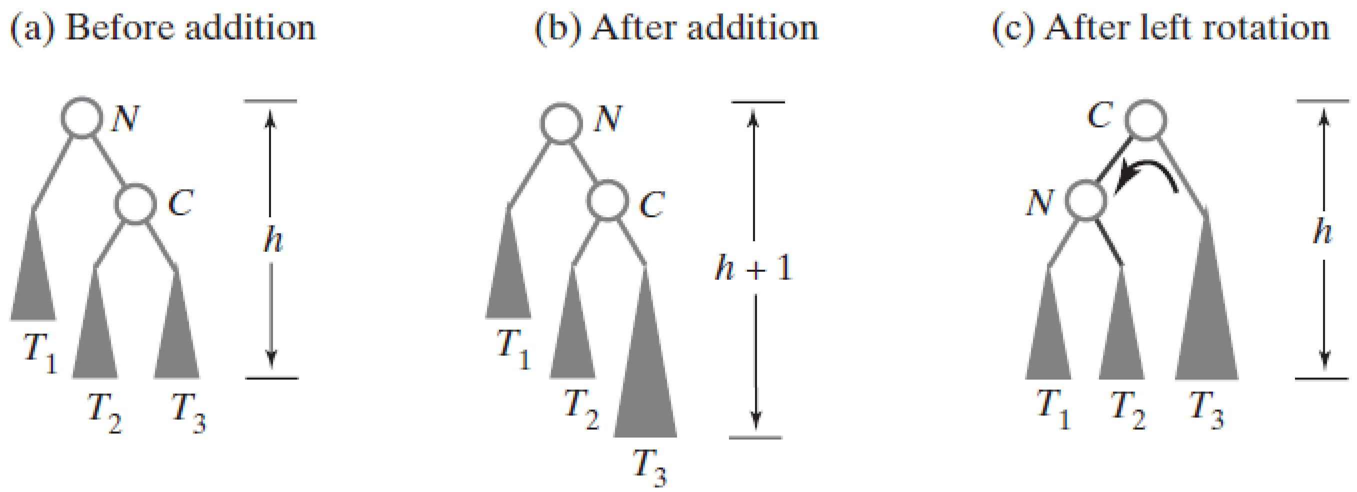
- Adding 4 into the AVL change the balance factor of node N





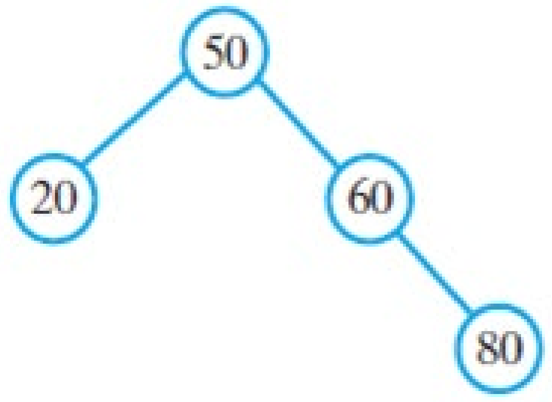
Adding a Node in AVL Trees

- Outside Branches (which require single rotation) :
  - **Case 2:** The right subtree of N's right child (left rotation)



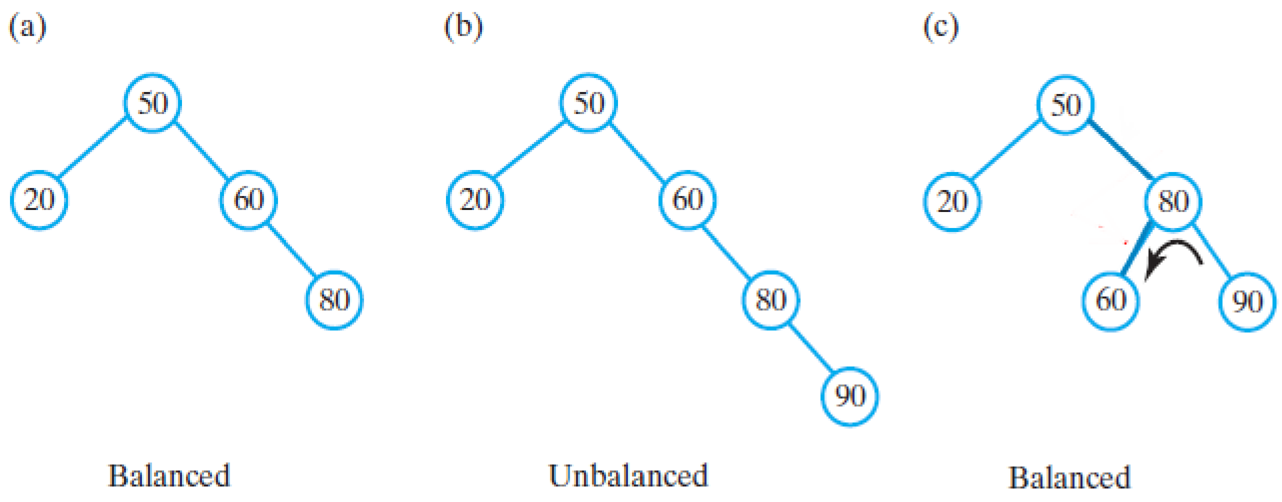
In-Class Exercise

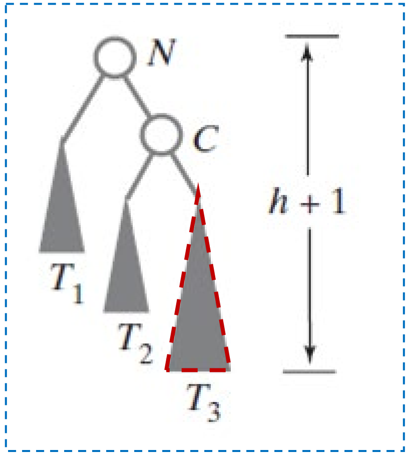
- Adding 90 into the AVL



In-Class Exercise

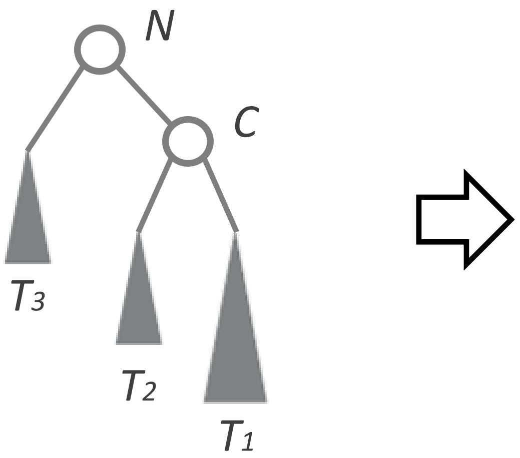
- Adding 90 into the AVL

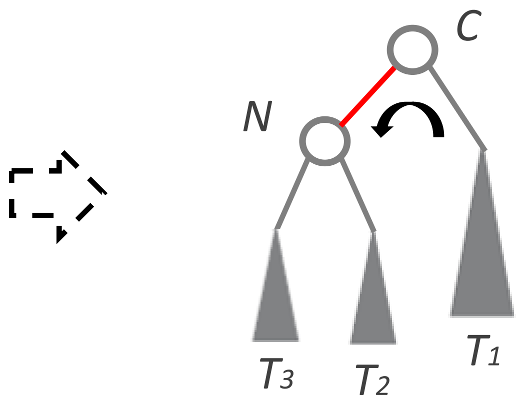




The Algorithm Performs Right Rotation

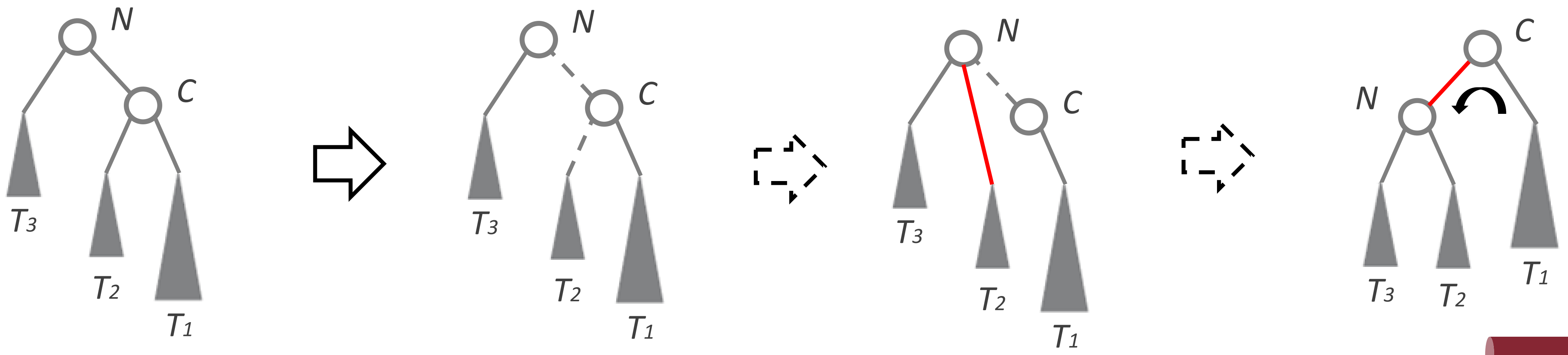
- Outside Branches (which require single rotation) :
  - **Case 2:** The right subtree of N's right child (**left rotation**)





- Outside Branches (which require single rotation) :
  - **Case 2:** The right subtree of N's right child (**left rotation**)

```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

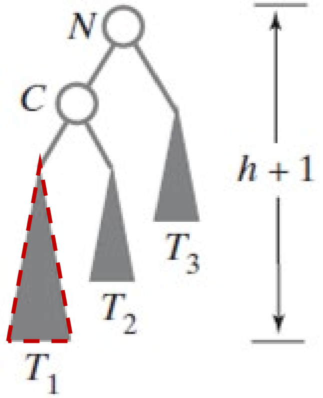


```
/** corrects for an imbalance in N due to
     an addition to the right subtree of N's right child
*/
Algorithm rotateLeft(N){
    C = N.rightChild
    N.rightChild = C.leftChild
    C.leftChild = N
    return N
}
```

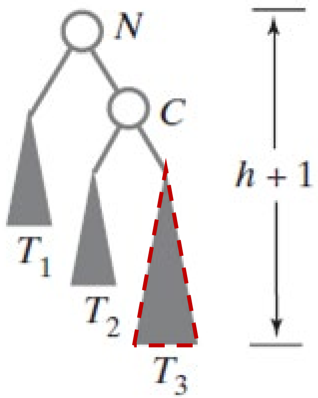
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

Adding a Node in AVL Trees

- There are four cases for the cause of the imbalance at node *N:*
  - Outside Branches which require single rotation
    Case 1: The left subtree of *N*'s left child (right rotation)
    Case 2: The right subtree of *N*'s right child (left rotation)



Case 1



Case 2

```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

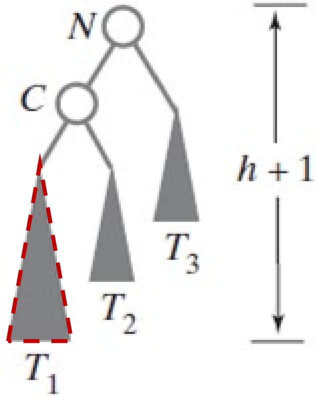
```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN du
// in the left subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to nodeC's right child
Set nodeC's right child to nodeN
return nodeC
```

Adding a Node in AVL Trees

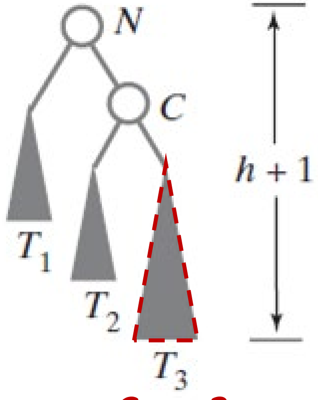
- There are four cases for the cause of the imbalance at node *N:*

  - Outside Branches which require single rotation
    Case 1: The left subtree of N's left child
    (right rotation)
    Case 2: The right subtree of N's right child
    (left rotation)

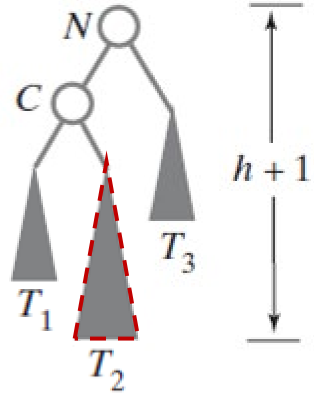
  - Inside Branches which require double rotation
    Case 3: The right subtree of N's left child
    (left-right rotation)
    Case 4: The left subtree of N's right child
    (right-left rotation)



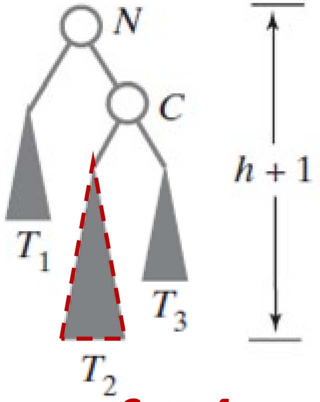
Case 1



Case 2

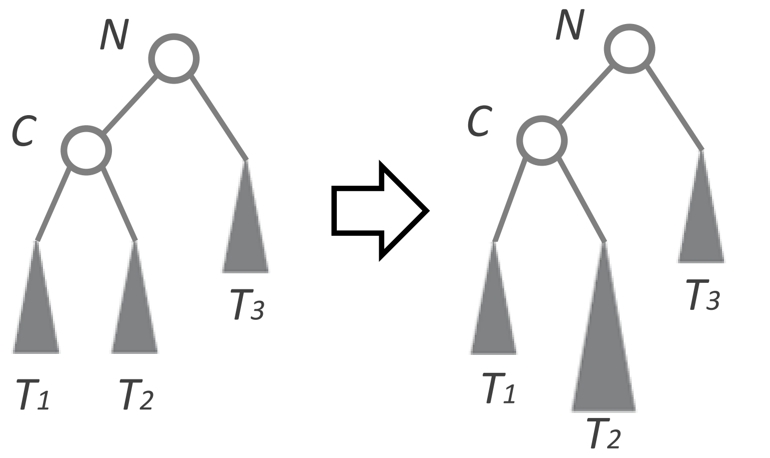


Case 3



Case 4

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



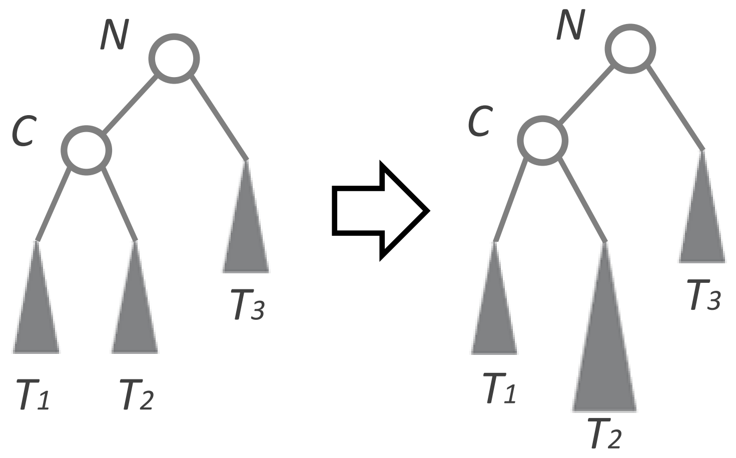
Consider a valid AVL tree

Add a node in T_2 (which changes the balance factor of node N)

How to make it balanced?

Adding a Node in AVL Trees

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



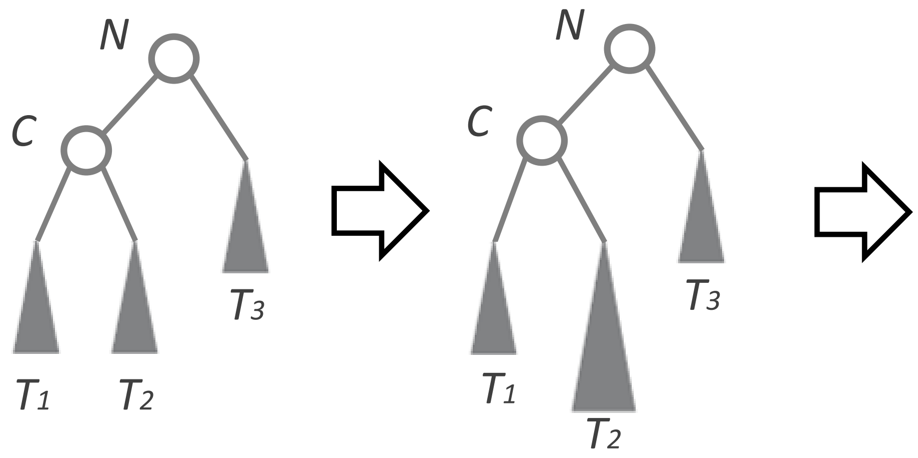
Consider a valid AVL tree

Add a node in T_2 (which changes the balance factor of node N)

How to make it balanced?
(we have learned right rotation and left rotation)

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)

Let's try right rotation first

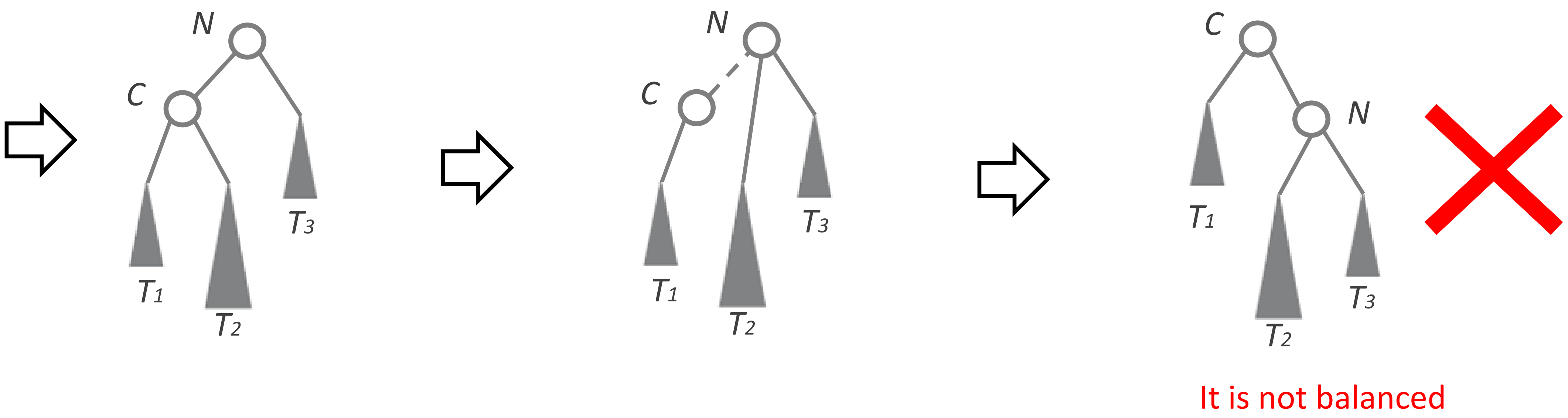


Consider a valid AVL tree

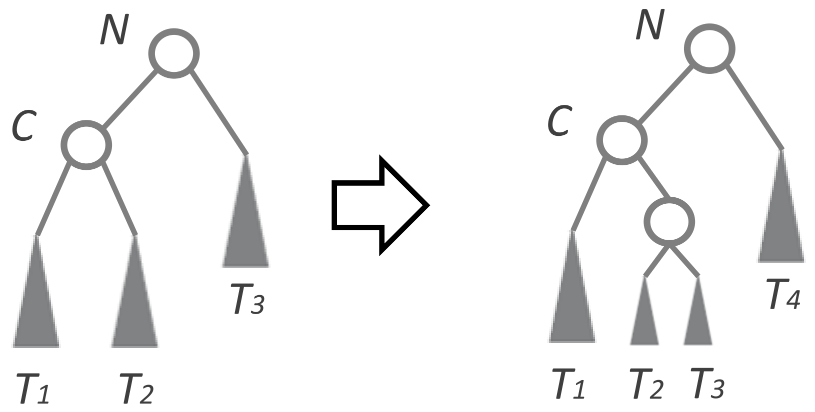
Add a node in T_2 (which changes the balance factor of node N)

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)

Let's try right rotation first



- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



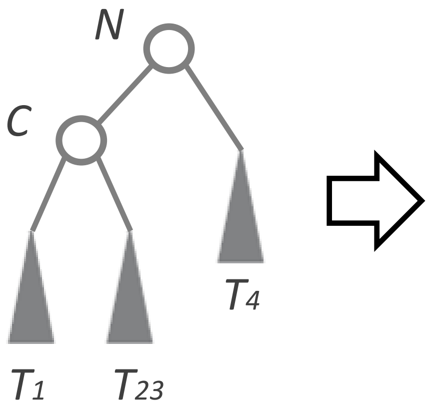
Consider a valid AVL tree

Since right-rotation doesn't work, let's try left-rotation.

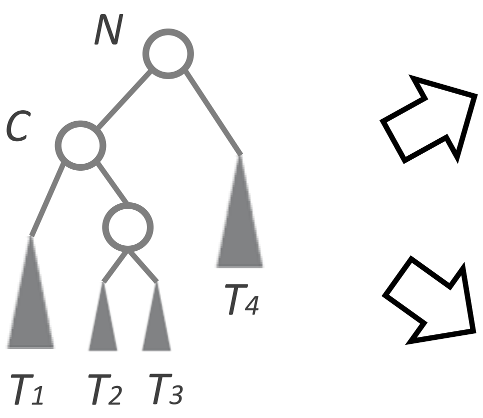
To apply left-rotation, we reformat the diagram

Adding a Node in AVL Trees

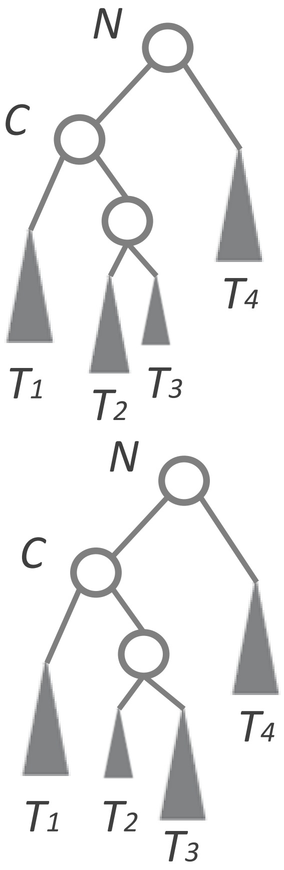
- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



Consider a valid AVL tree



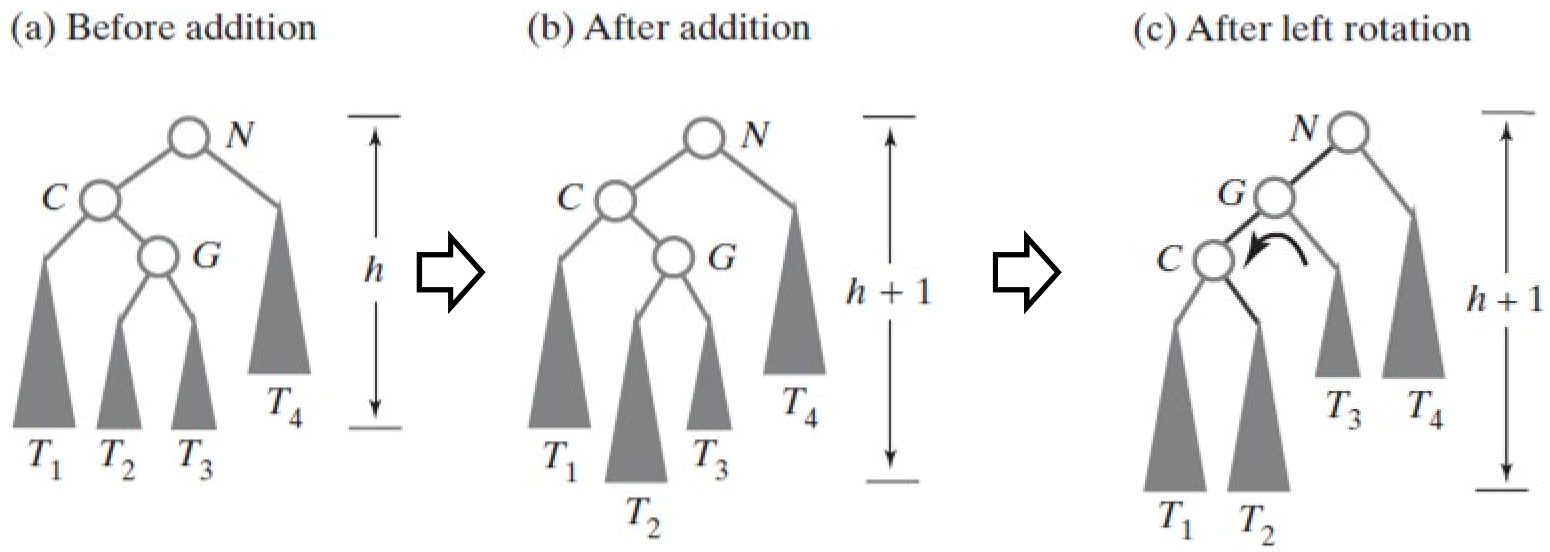
The same AVL tree



Add a node into the right subtree of N's left child (which changes the balance factor of node N)

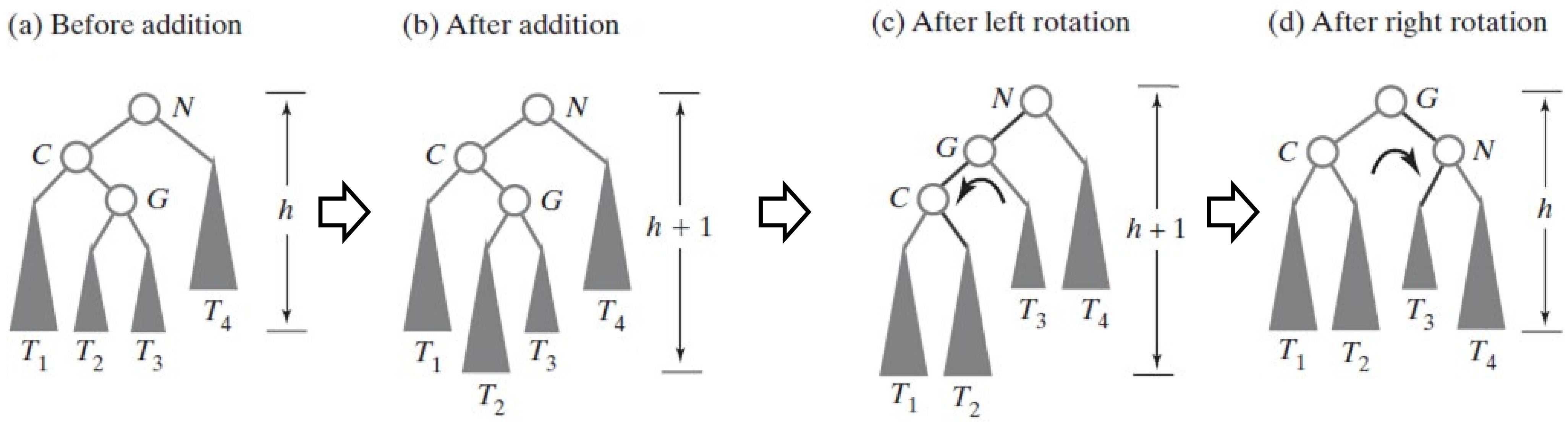
Adding a Node in AVL Trees

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



Adding a Node in AVL Trees

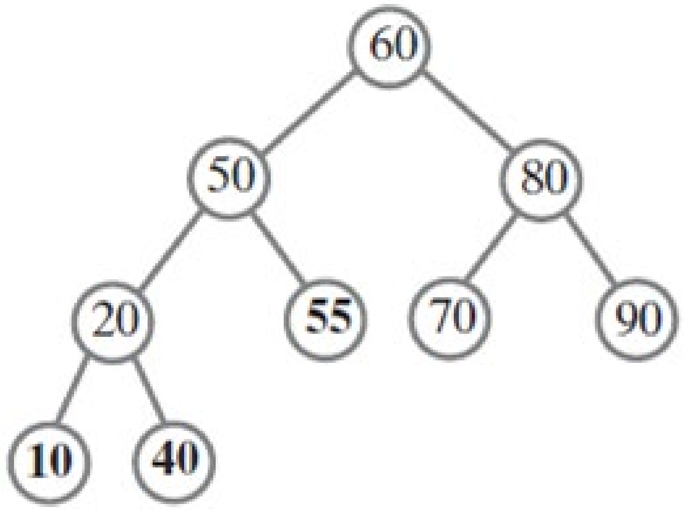
- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



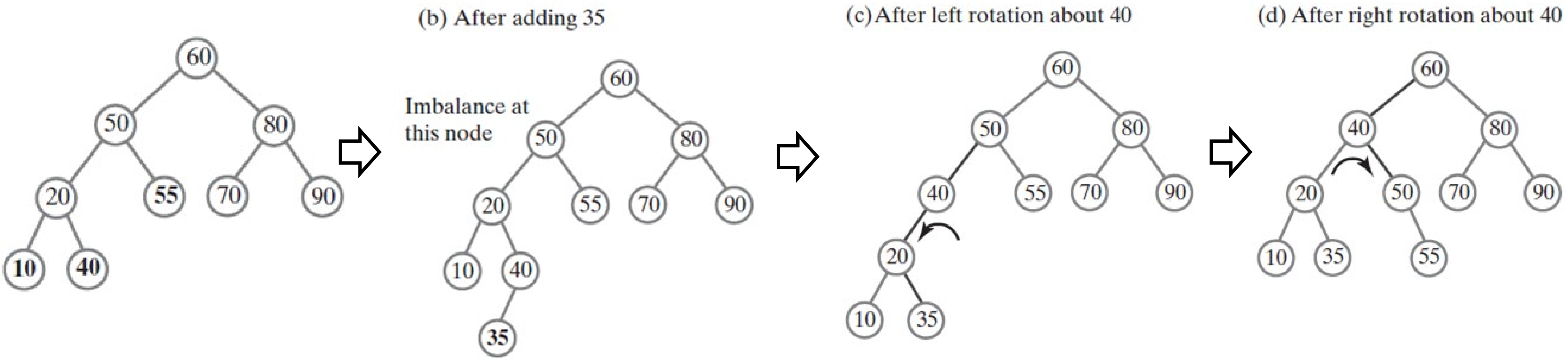
```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to the node returned by rotateLeft(nodeC)
return rotateRight(nodeN)
```

In-Class Exercise

- Adding 35 into the AVL



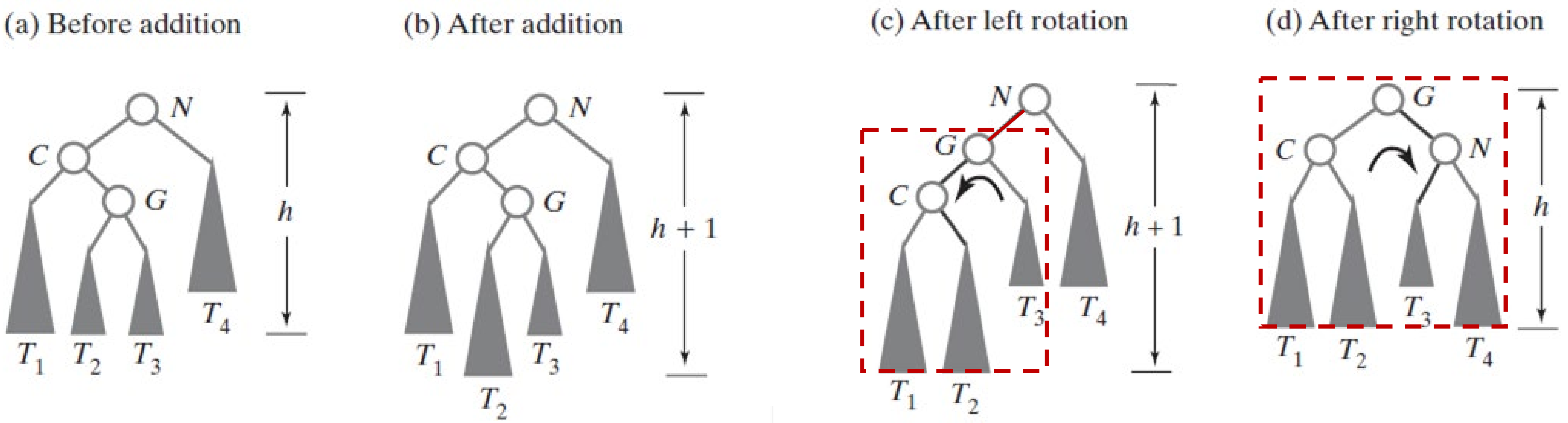
- Adding 35 into the AVL



The Algorithm Performs Right Rotation

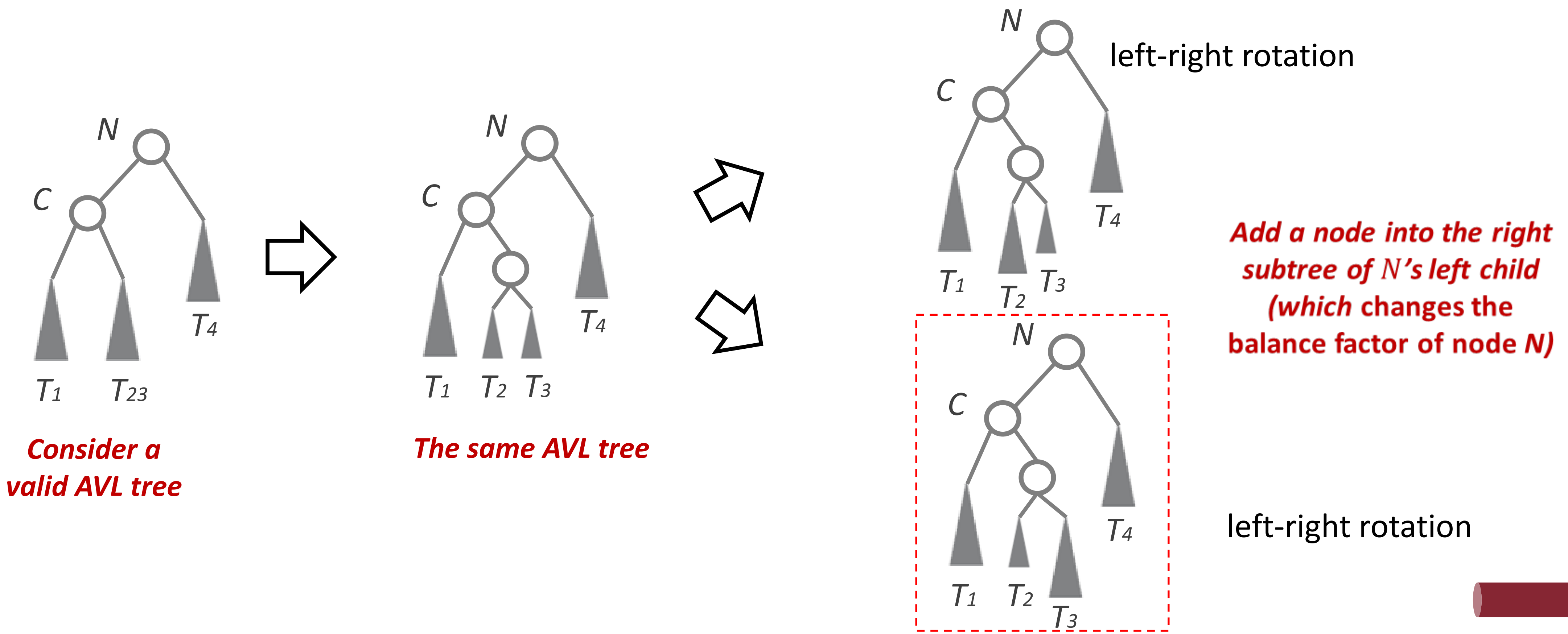
- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)

```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to the node returned by rotateLeft(nodeC)
return rotateRight(nodeN)
```



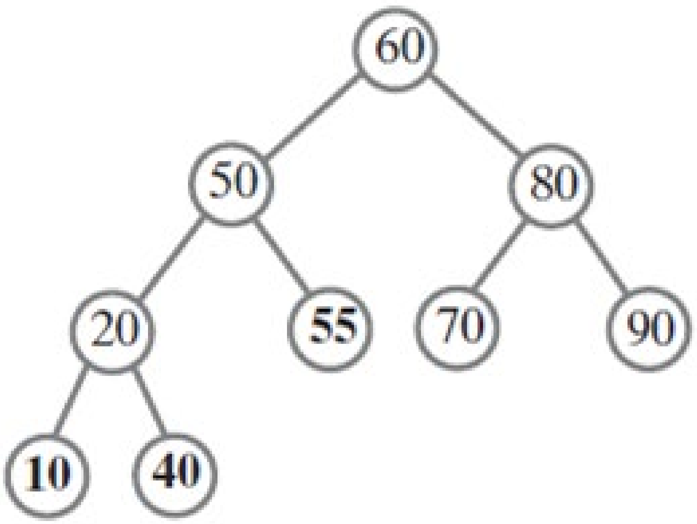
Adding a Node in AVL Trees

- Inside Branches (which require double rotations) :
  - **Case 3**: The right subtree of N's left child (left-right rotation)



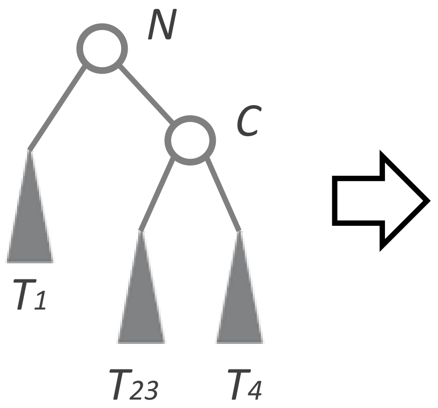
In-Class Exercise

- Adding 45 into the AVL

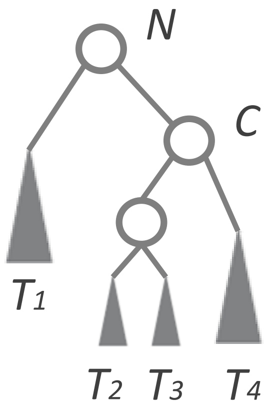


Adding a Node in AVL Trees

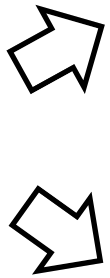
- Inside Branches (which require double rotations) :
  - **Case 4:** The left subtree of N's right child (right-left rotation)

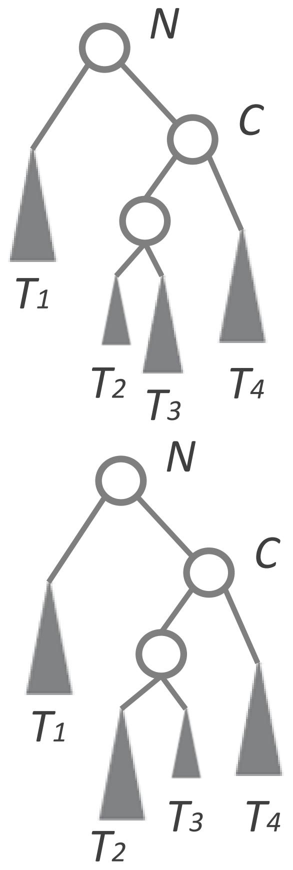


Consider a valid AVL tree



The same AVL tree

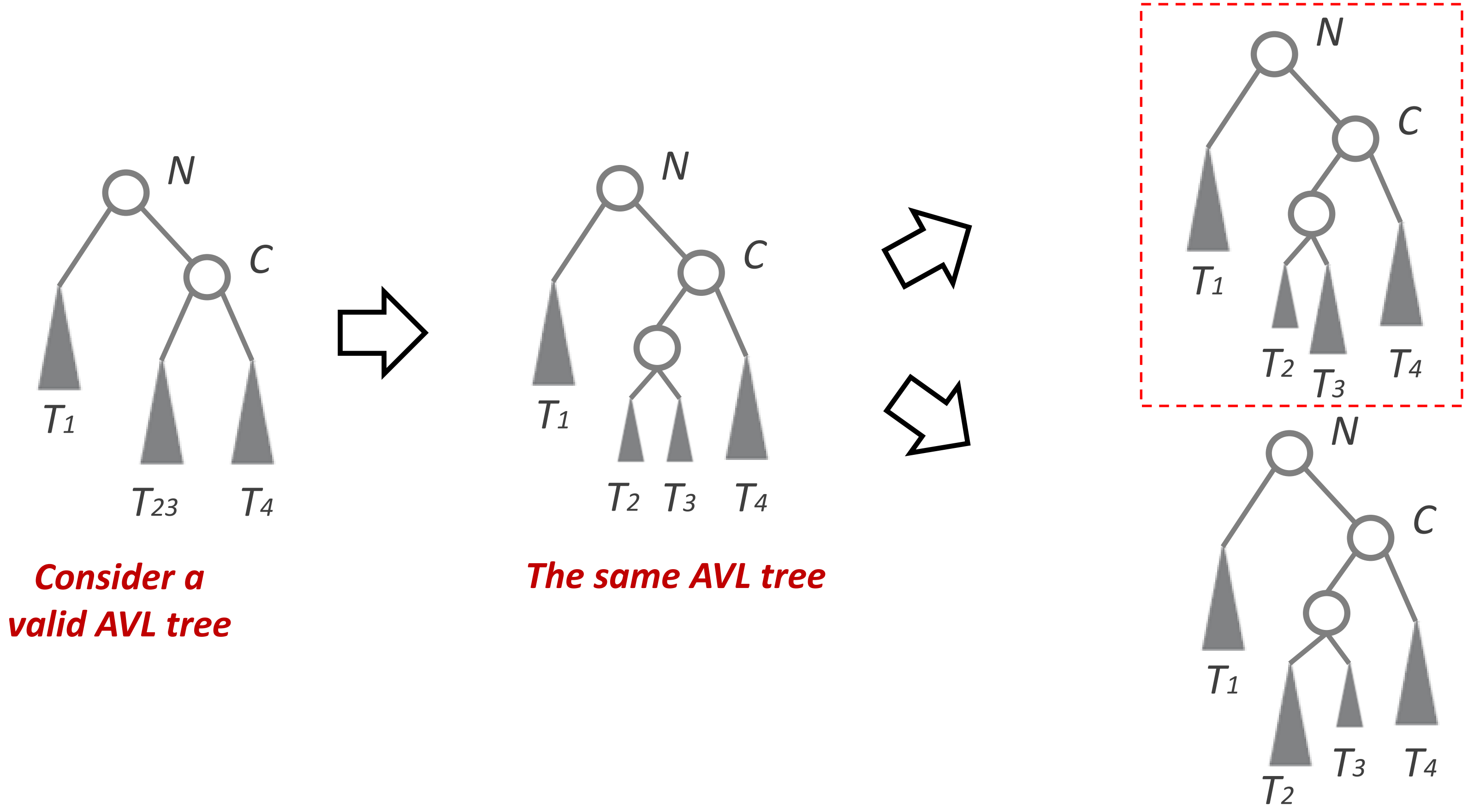




Add a new node

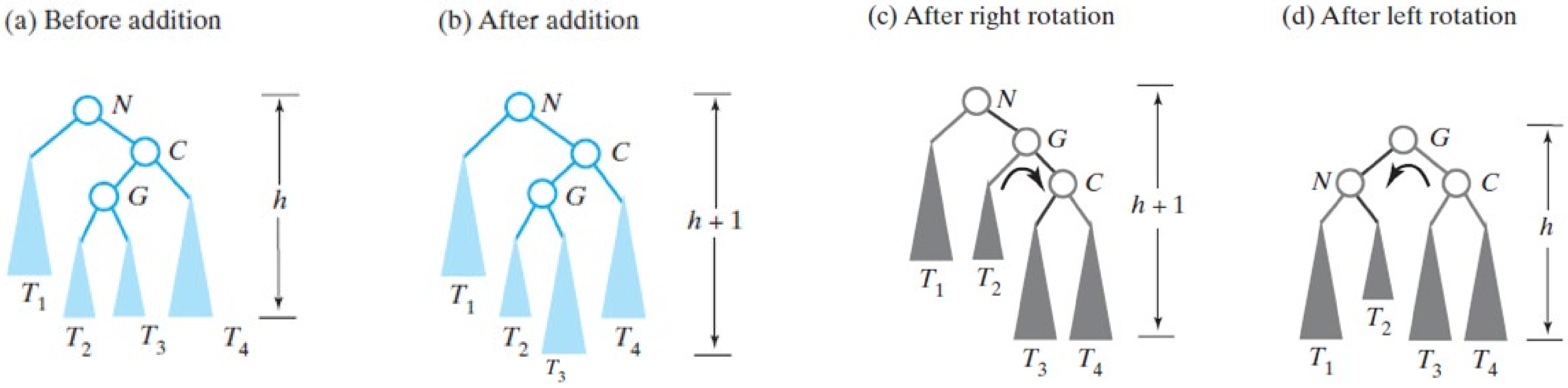
Adding a Node in AVL Trees

- Inside Branches (which require double rotations) :
  - **Case 4:** The left subtree of N's right child (right-left rotation)



Adding a Node in AVL Trees

- Inside Branches (which require double rotations) :
  - **Case 4:** The left subtree of *N*'s right child (right-left rotation)

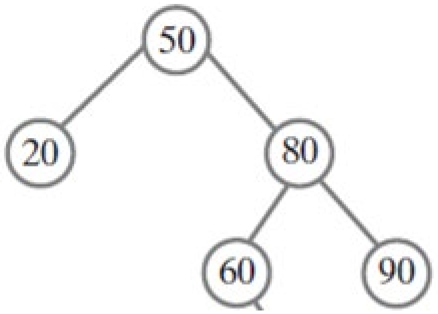


Algorithm

```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```

In-Class Exercise

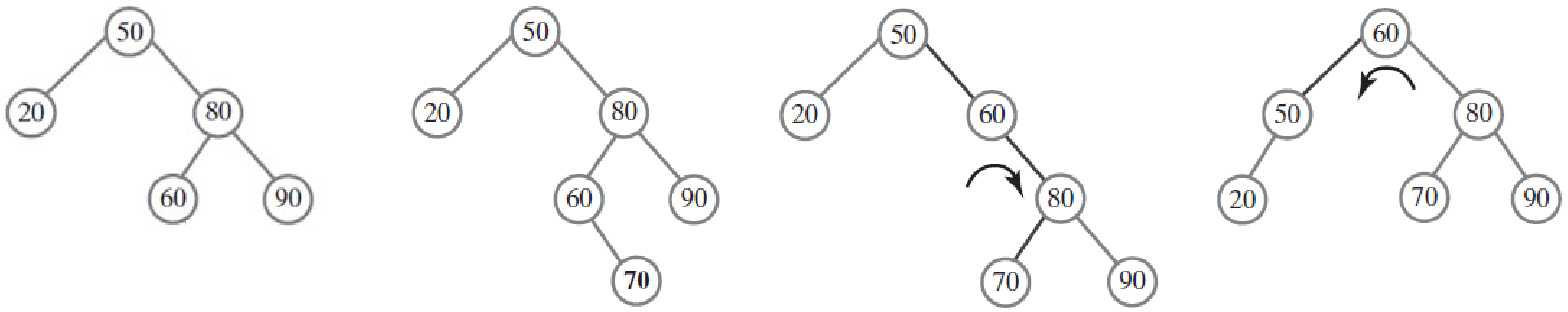
- Adding 70 into the AVL



In-Class Exercise

- Adding 70 into the AVL

(a) After adding 70 (b) After right rotation (c) After left rotation

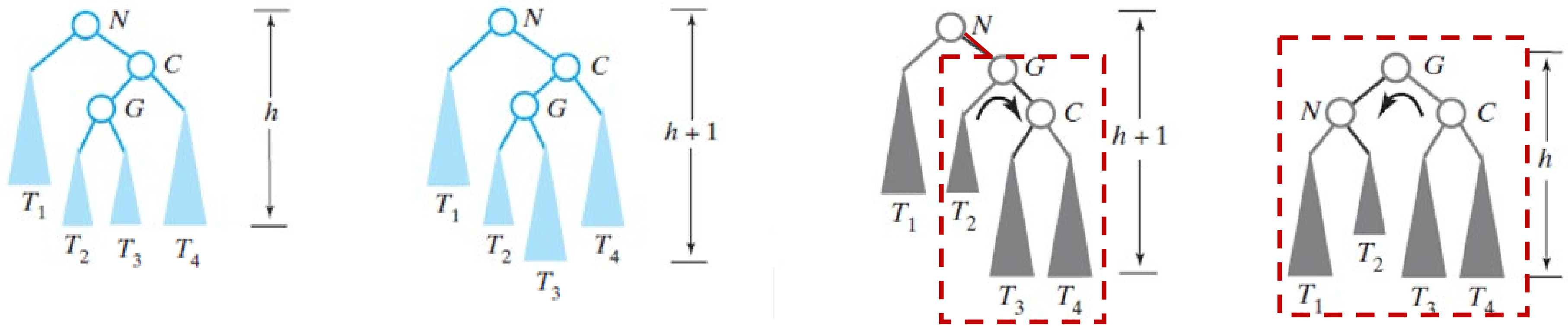


The Algorithm Performs Right Rotation

- Inside Branches (which require double rotations) :
  - **Case 4:** The left subtree of N's right child (**right-left rotation**)

```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```

(a) Before addition (b) After addition (c) After right rotation (d) After left rotation

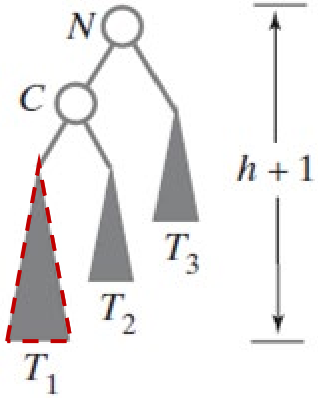


Adding a Node in AVL Trees

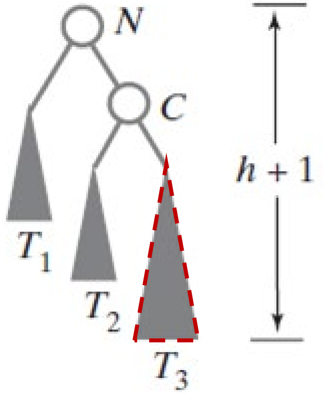
- There are four cases for the cause of the imbalance at node *N*:

  - Outside Branches which require single rotation
    - Case 1: The left subtree of N's left child (right rotation)
    - Case 2: The right subtree of N's right child (left rotation)

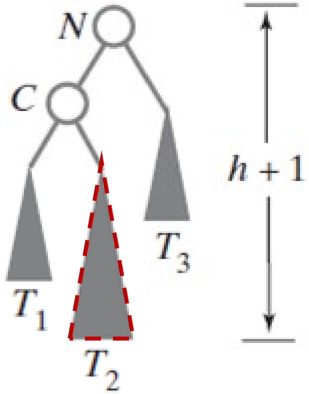
  - Inside Branches which require double rotation
    - **Case 3: The right subtree of N's left child (left-right rotation)**
    - **Case 4: The left subtree of N's right child (right-left rotation)**



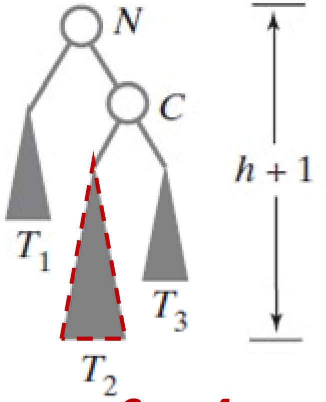
Case 1



Case 2



Case 3



Case 4

In-Class Exercise

- Adding 14, 17, 11, 7, 53, 4, 13, 12, and 8 to an initially empty AVL tree

In-Class Exercise

- Adding 41, 20, 65, 11, 29, 50, 26, 23, and 55 into an initially empty AVL tree

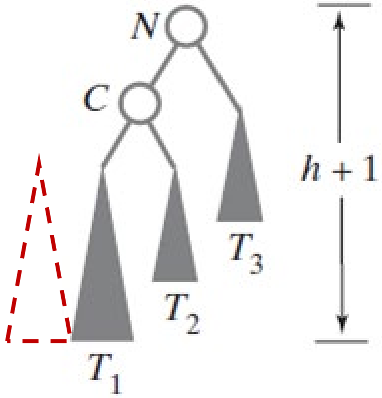
In-Class Exercise

- Adding 1, 2, 3, 4, 5, and 6 to an initially empty
  - (1) AVL Tree
  - (2) Binary Search Tree
  - (3) Compare the height of the resulting AVL tree and the resulting Binary Search Tree

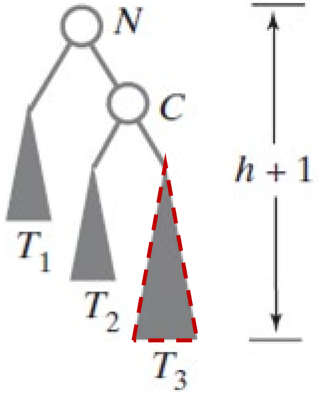
- Adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty
  - (1) AVL Tree
  - (2) Binary Search Tree
  - (3) Compare the height of the resulting AVL tree and the resulting Binary Search Tree

Adding a Node in AVL Trees

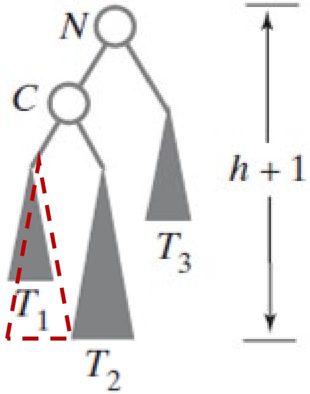
```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{ // addition was in nodeN's left subtree
    if (the left child of nodeN has a left subtree that is taller than its right subtree)
        rotateRight(nodeN)        // addition was in left subtree of left child
    else
        rotateLeftRight(nodeN)   // addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{ // addition was in nodeN's right subtree
    if (the right child of nodeN has a right subtree that is taller than its left subtree)
        rotateLeft(nodeN)         // addition was in right subtree of right child
    else
        rotateRightLeft(nodeN)   // addition was in left subtree of right child
}
```



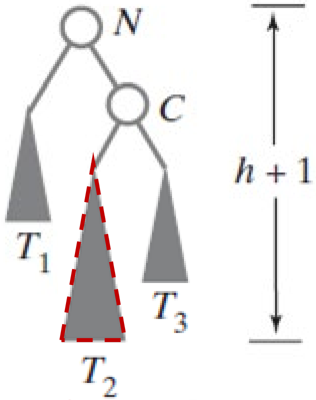
Case 1



Case 2



Case 3



Case 4

Algorithm

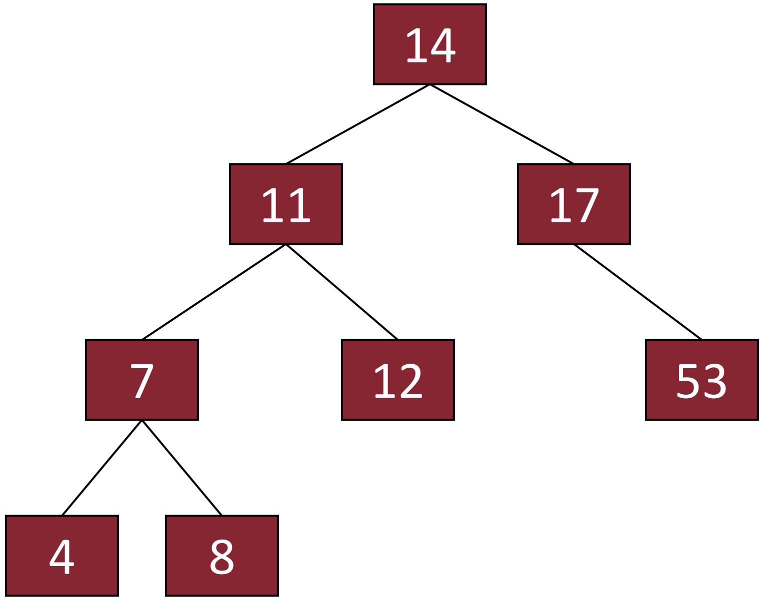
```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{ // addition was in nodeN's left subtree
   if (the left child of nodeN has a left subtree that is taller than its right subtree)
      rotateRight(nodeN)       // addition was in left subtree of left child
   else
      rotateLeftRight(nodeN) // addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{ // addition was in nodeN's right subtree
   if (the right child of nodeN has a right subtree that is taller than its left subtree)
      rotateLeft(nodeN)        // addition was in right subtree of right child
   else
      rotateRightLeft(nodeN) // addition was in left subtree of right child
}
```

Removing a Node in AVL Trees

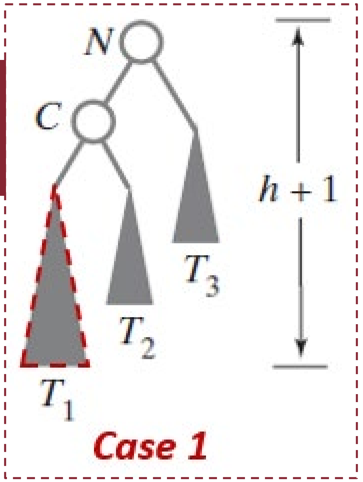
- If a node is a leaf, remove it.
- If the node is not a leaf, replace it with either the largest in its left subtree (rightmost) or the smallest in its right subtree (leftmost), and remove that node. The node that was found as replacement has at most one subtree.
- After deletion, retrace the path from parent of the replacement to the root, adjusting the balance factors as needed.
- Readings: http://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Removing a Node in AVL Trees

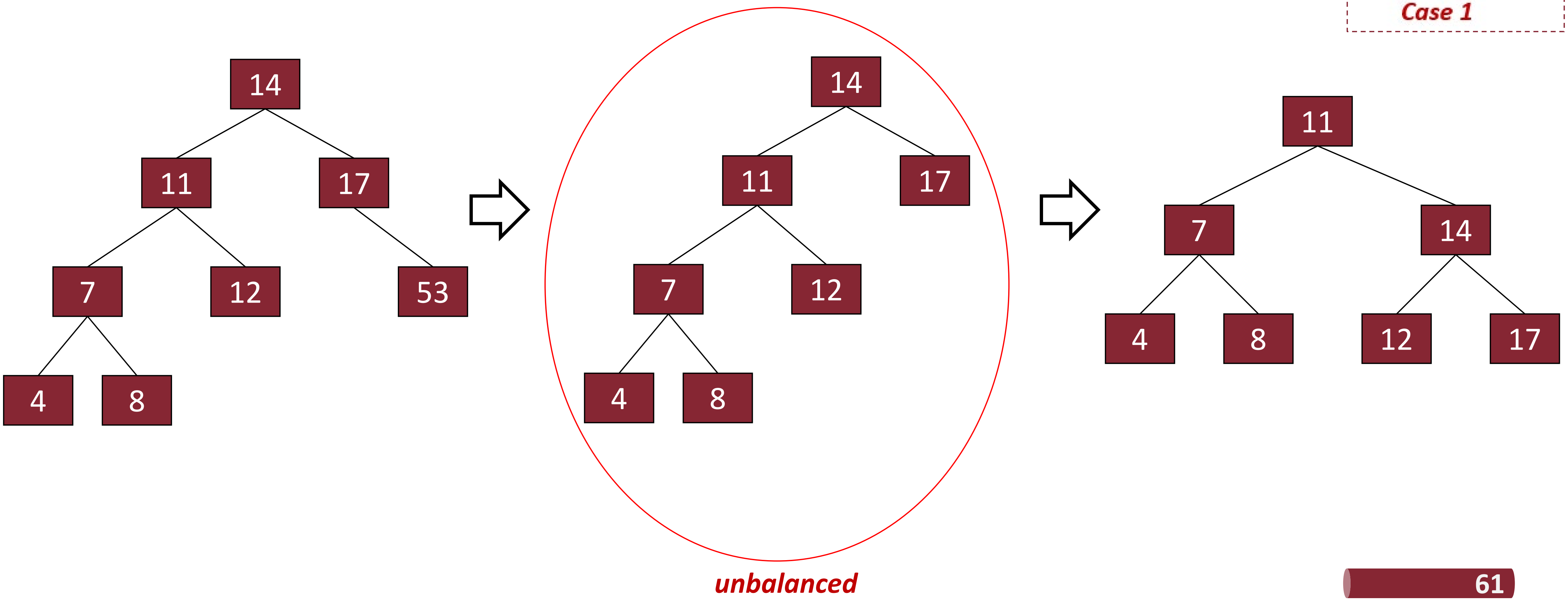
- Remove 53



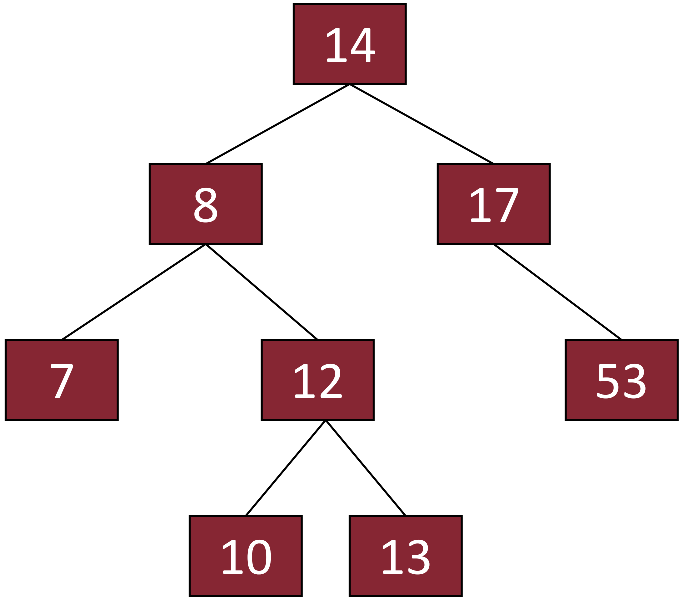
Removing a Node in AVL Trees



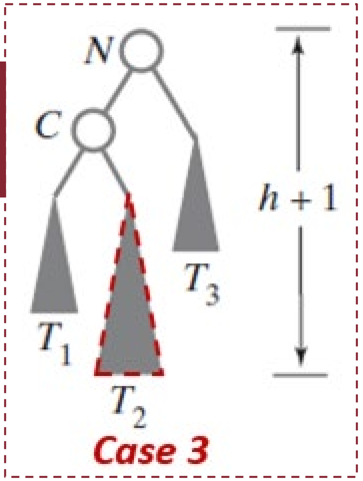
- Remove 53



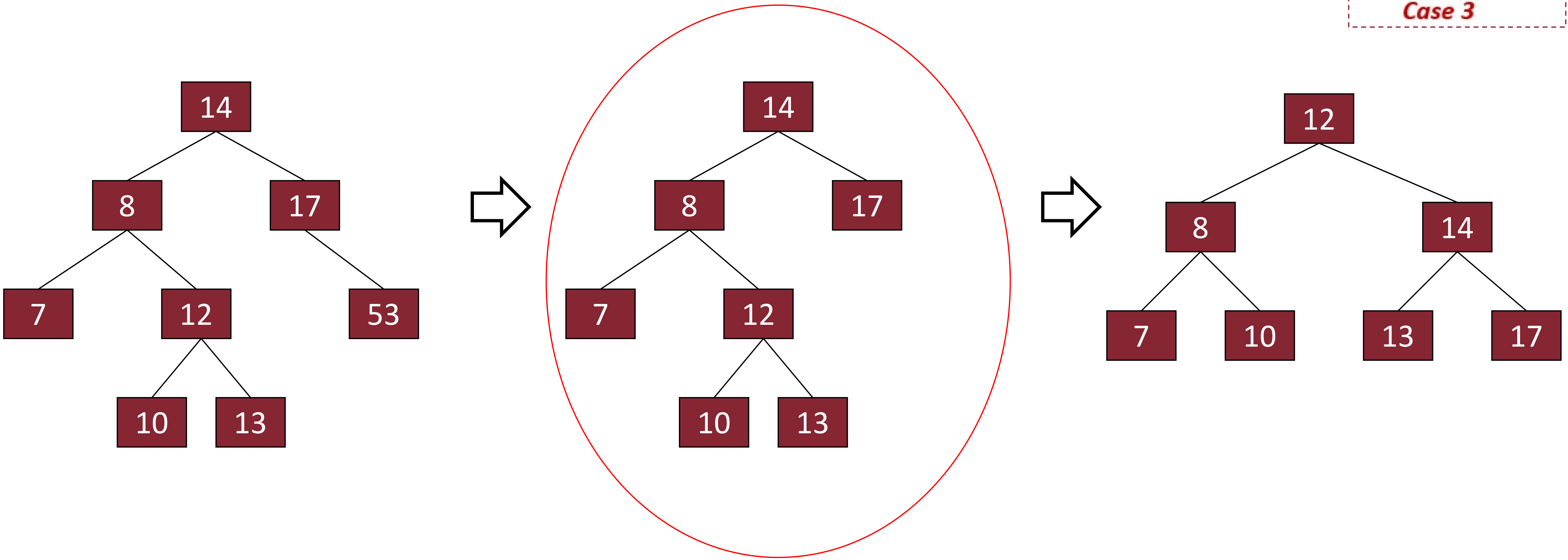
- Remove 53



Removing a Node in AVL Trees



- Remove 53



unbalanced

In-Class Exercise

- (1) Build an AVL tree with the following values:
  - 15, 20, 24, 10, 13, 7, 30, 25
- (2) Then, remove 24 and 20 from the AVL tree.

BST vs Hash Table

- Compare binary search trees with hash tables. Find pros and cons of each data structure.
  - Time complexity of operations
  - Space complexity of data structure
  - Handling varying input sizes
  - Traversal
  - Other supported operations?

- Readings: http://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/