Balanced Search Trees

Intro

- Operations on a *binary search tree* (**BST**) are $O(\log n)$ if the **tree is balanced**.

- Unfortunately, the ***add*** and ***remove*** operations do not ensure that a binary search tree remains balanced.

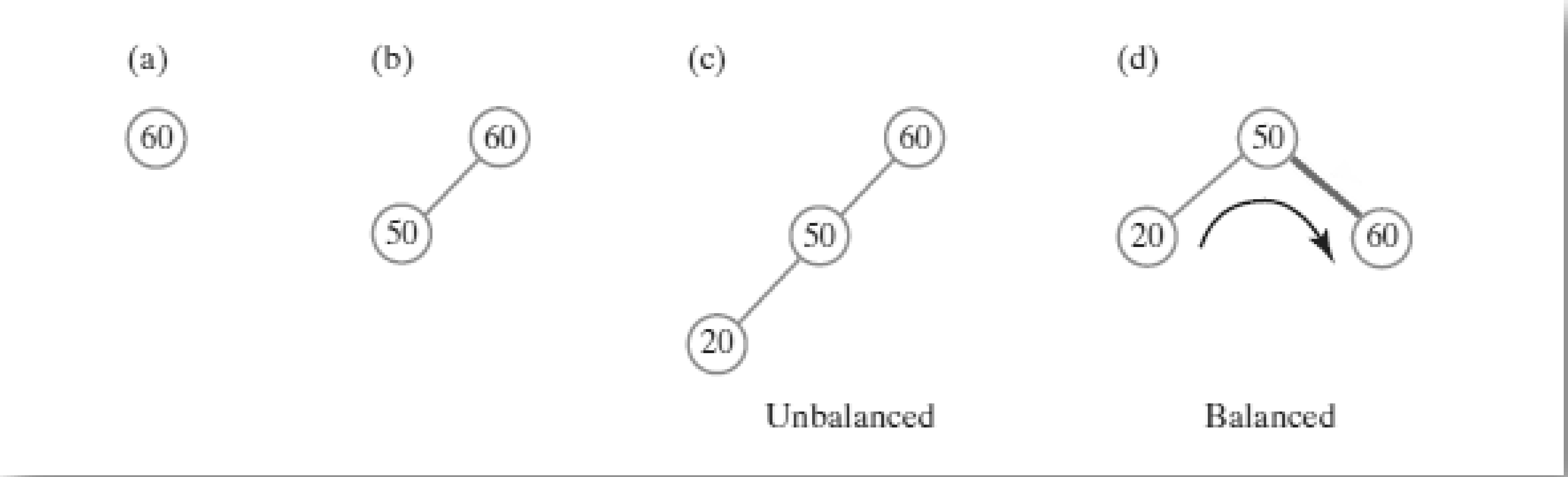
- You could take an unbalanced search tree and rearrange its nodes to **get a balanced BST**. Recall that every node in a balanced binary tree has subtrees whose height differ by no more than 1.

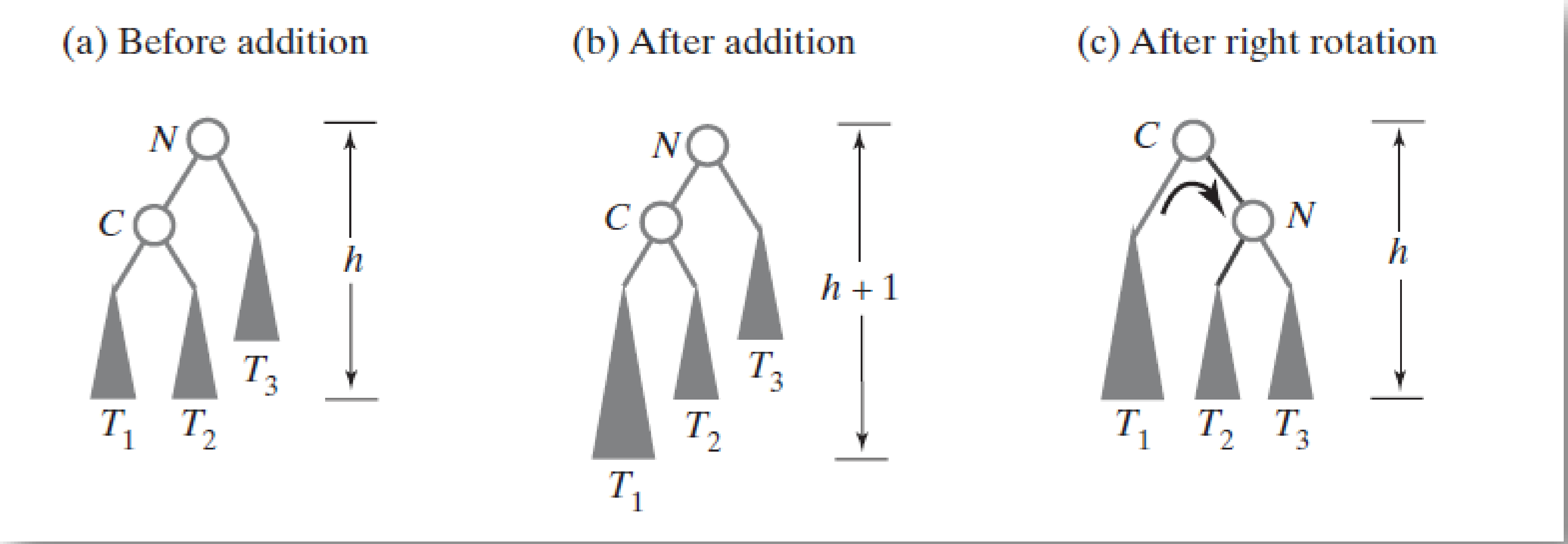
AVL Trees

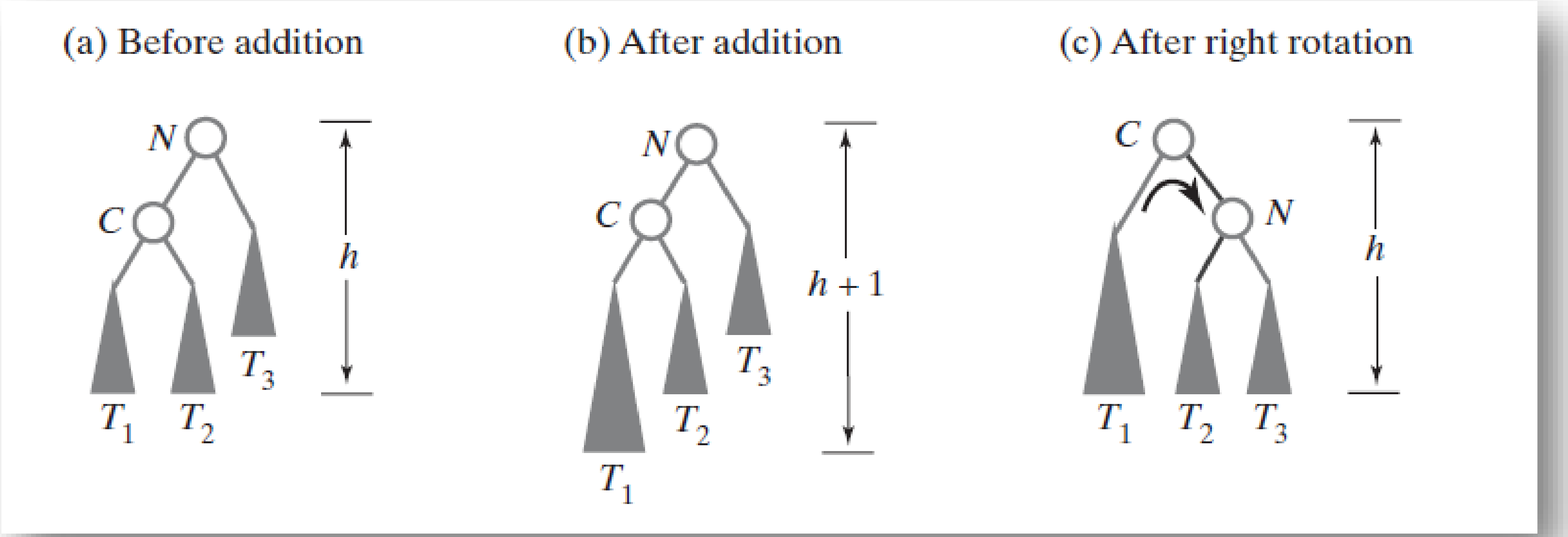
- The idea of rearranging nodes to balance a tree was first developed in 1962 by two Russian (USSR) mathematicians, Adel’son-Vel’skii and Landis. Named after them, the **AVL tree** is a *BST* that rearranges its nodes whenever it becomes unbalanced.

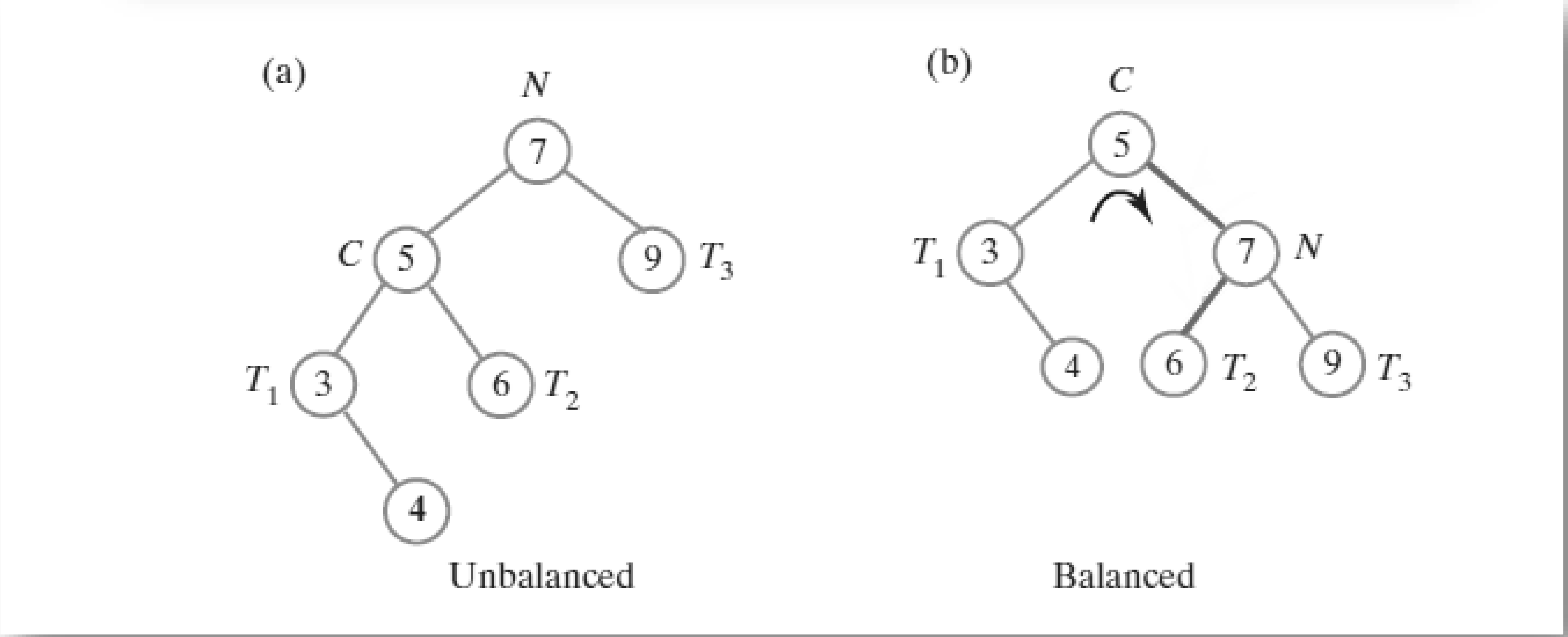
- The balance of a binary search tree is upset only when you *add* or *remove* a node. Thus, during these operations, the AVL tree rearranges nodes as necessary to maintain its balance.

Single Rotations: **Right** rotations



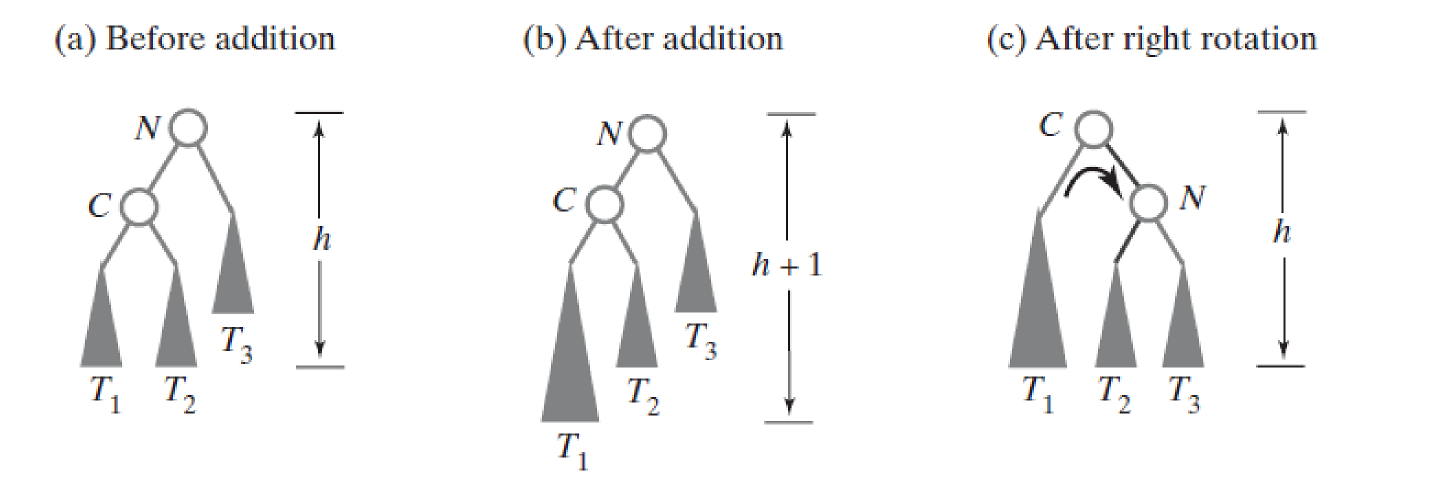

(a) Before addition
N
C
T_1
T_2
T_3
h
(b) After addition
N
C
T_1
T_2
T_3
$h + 1$
(c) After right rotation
C
N
T_1
T_2
T_3
h


(a) Before addition
N
C
h
T_1
T_2
T_3
(b) After addition
N
C
$h + 1$
T_1
T_2
T_3
(c) After right rotation
C
N
h
T_1
T_2
T_3

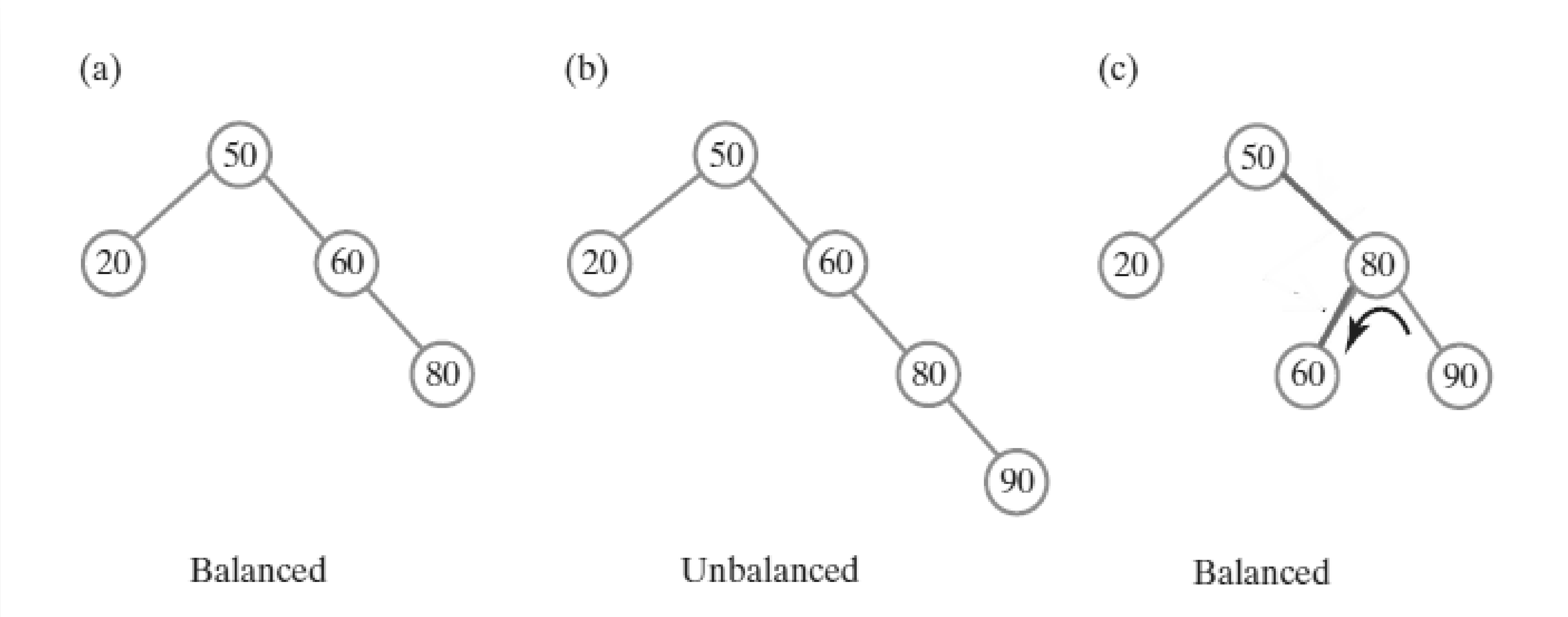

(a)
N
7
C
5
9
T_3
T_1
3
6
T_2
4
Unbalanced
(b)
C
5
T_1
3
7
N
4
6
T_2
9
T_3
Balanced

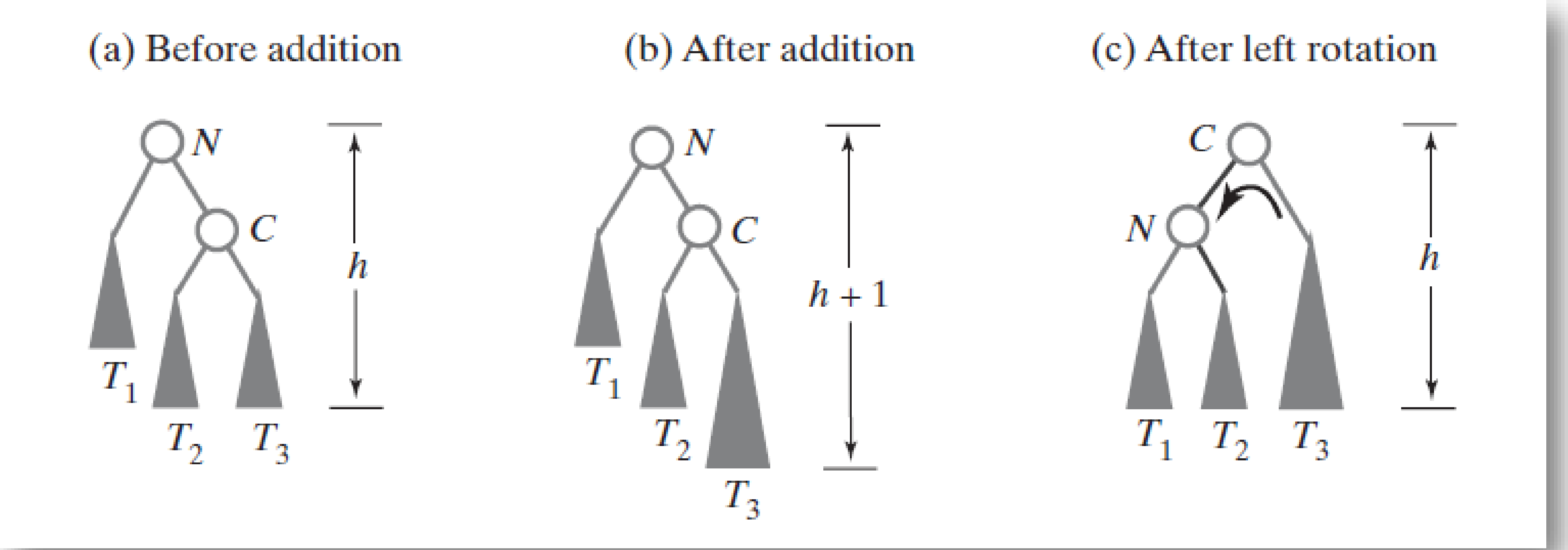
Algorithm `rotateRight(nodeN)`
// Corrects an imbalance at a given node nodeN *due to an addition*
// in the left subtree of nodeN*'s left child.*

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



Single Rotations: **Left** Rotations

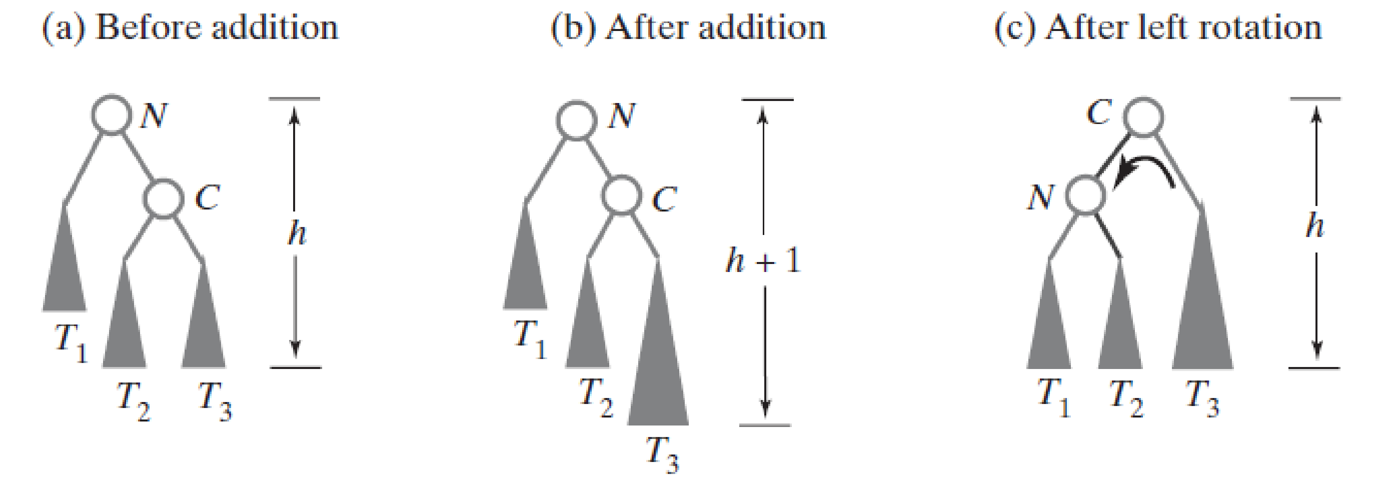




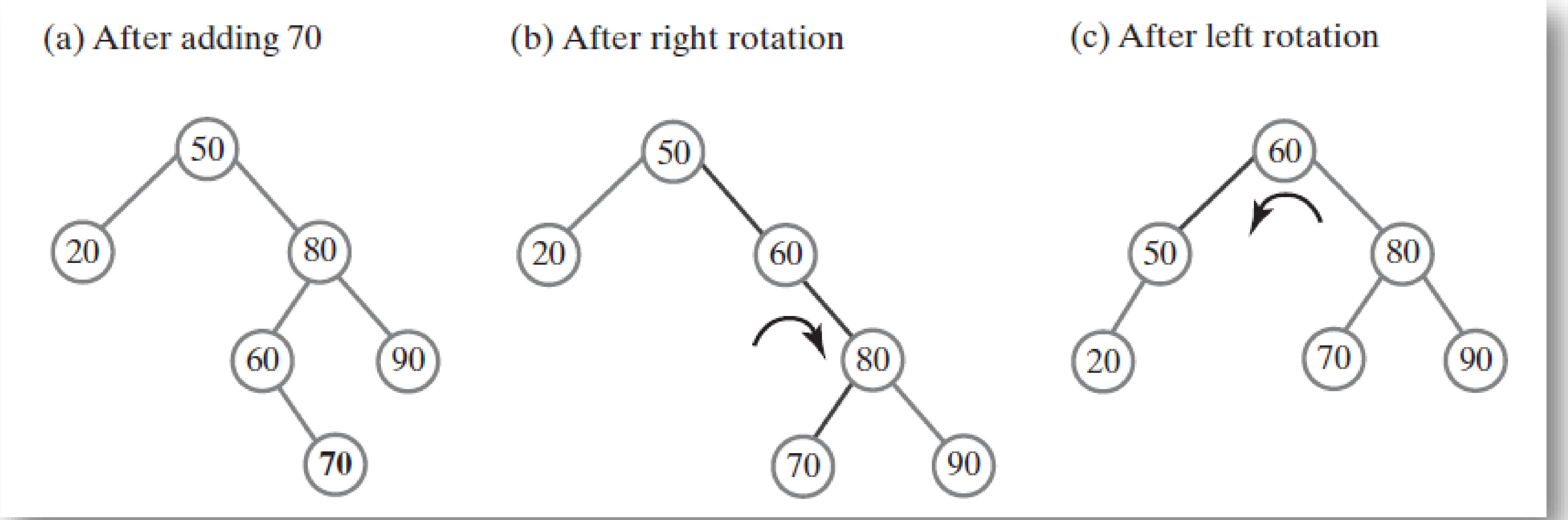
(a) Before addition
N
C
h
T_1
T_2
T_3
(b) After addition
N
C
h + 1
T_1
T_2
T_3
(c) After left rotation
C
N
h
T_1
T_2
T_3

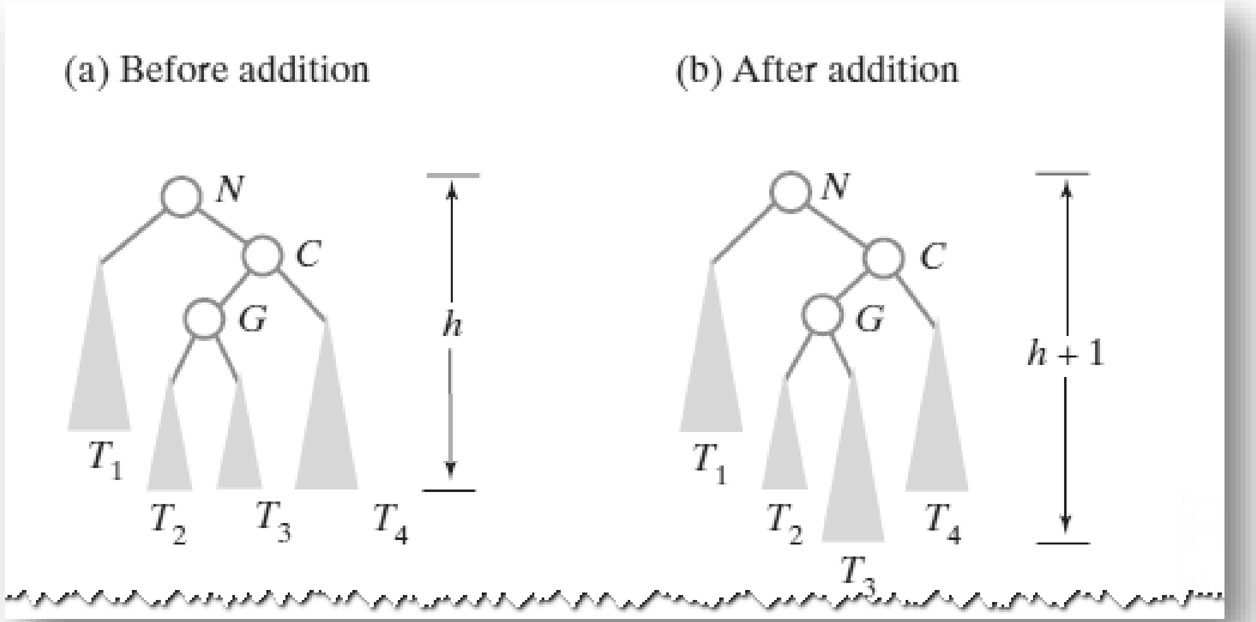
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

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

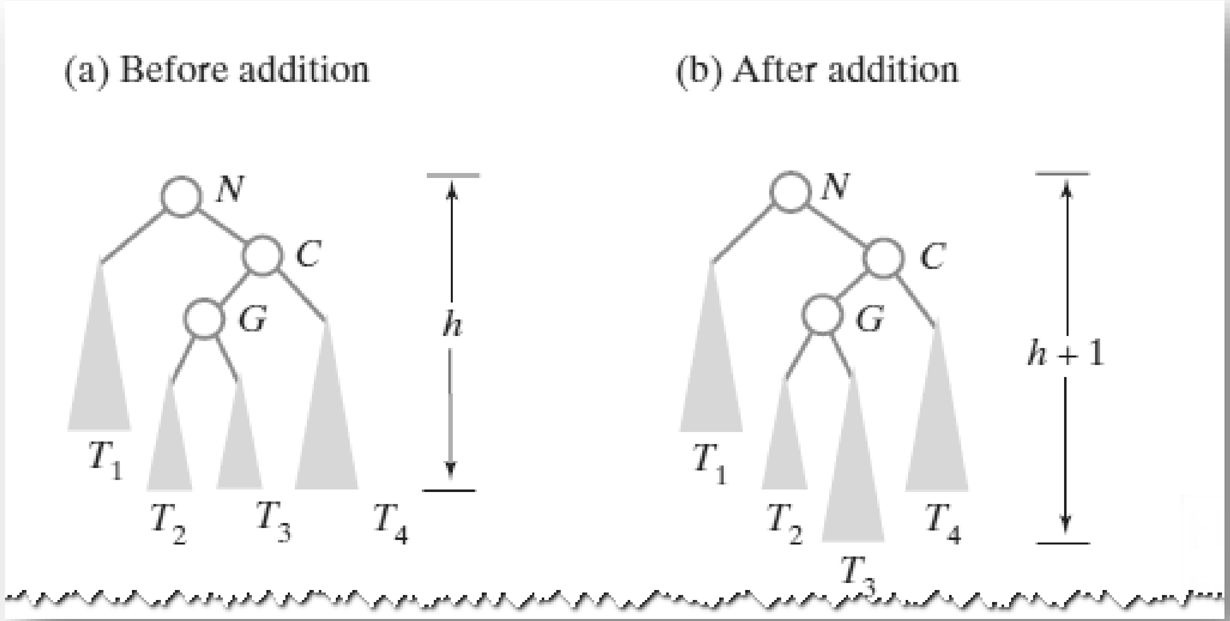


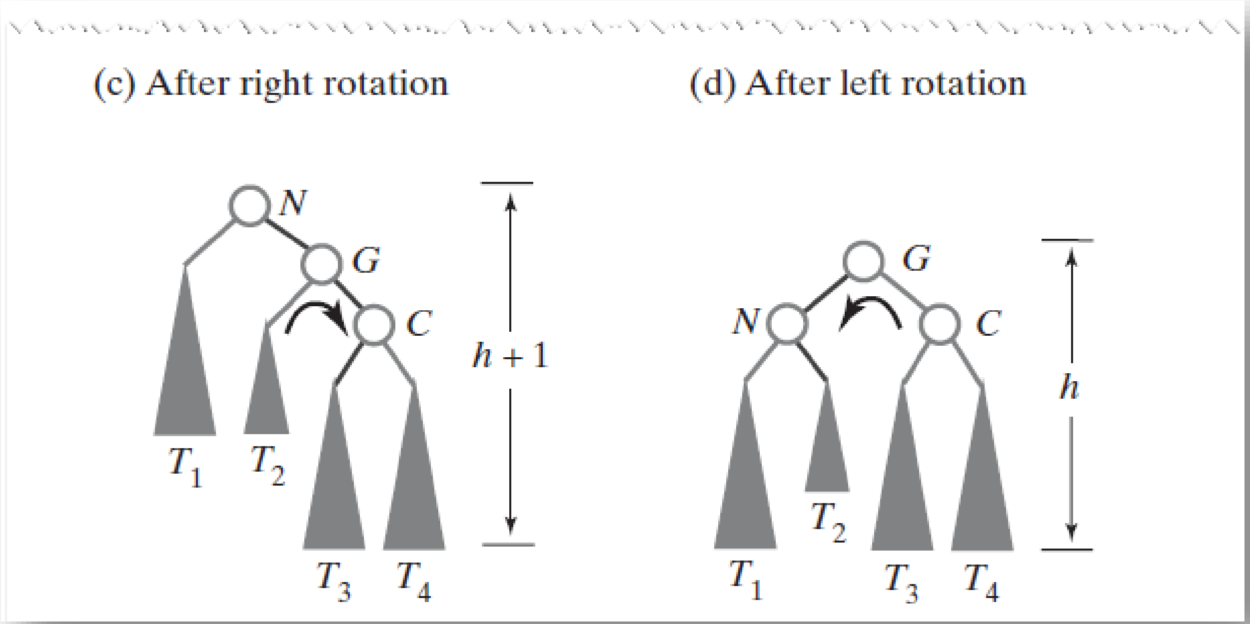
Double Rotations: **Right-Left** double rotations

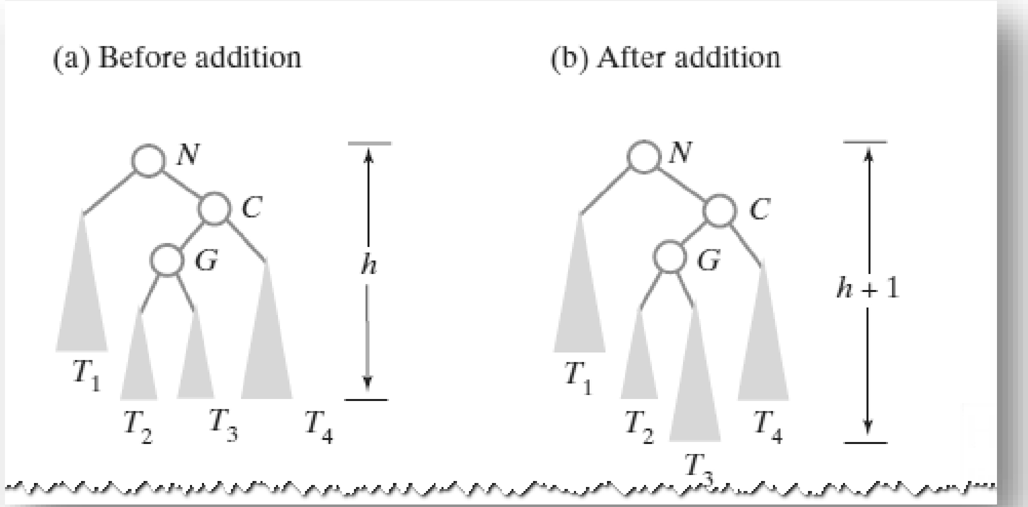


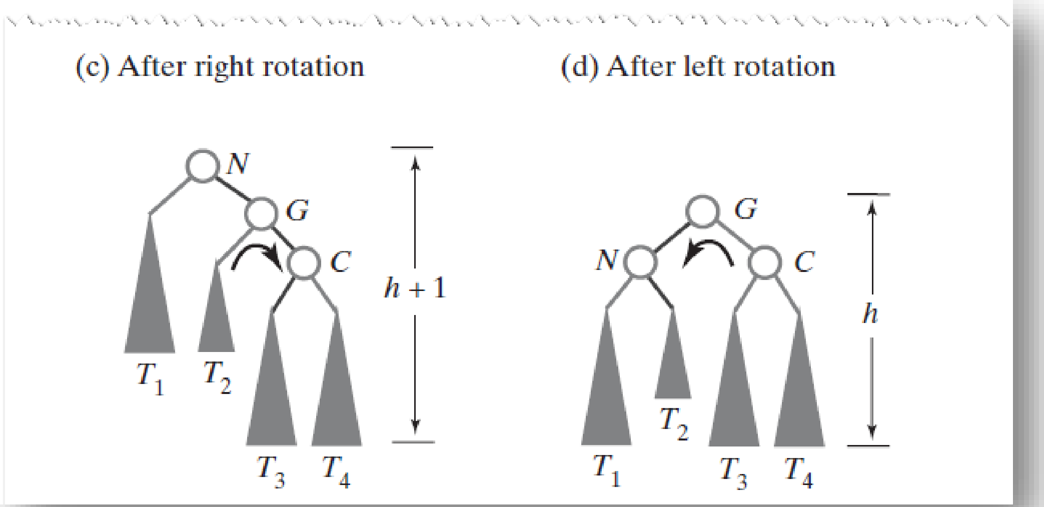


(a) Before addition
(b) After addition
N
C
G
h
T1
T2
T3
T4
h + 1


(a) Before addition
(b) After addition
N
C
G
h
h + 1
T1
T2
T3
T4


(c) After right rotation
(d) After left rotation
N
G
C
h + 1
h
T1
T2
T3
T4

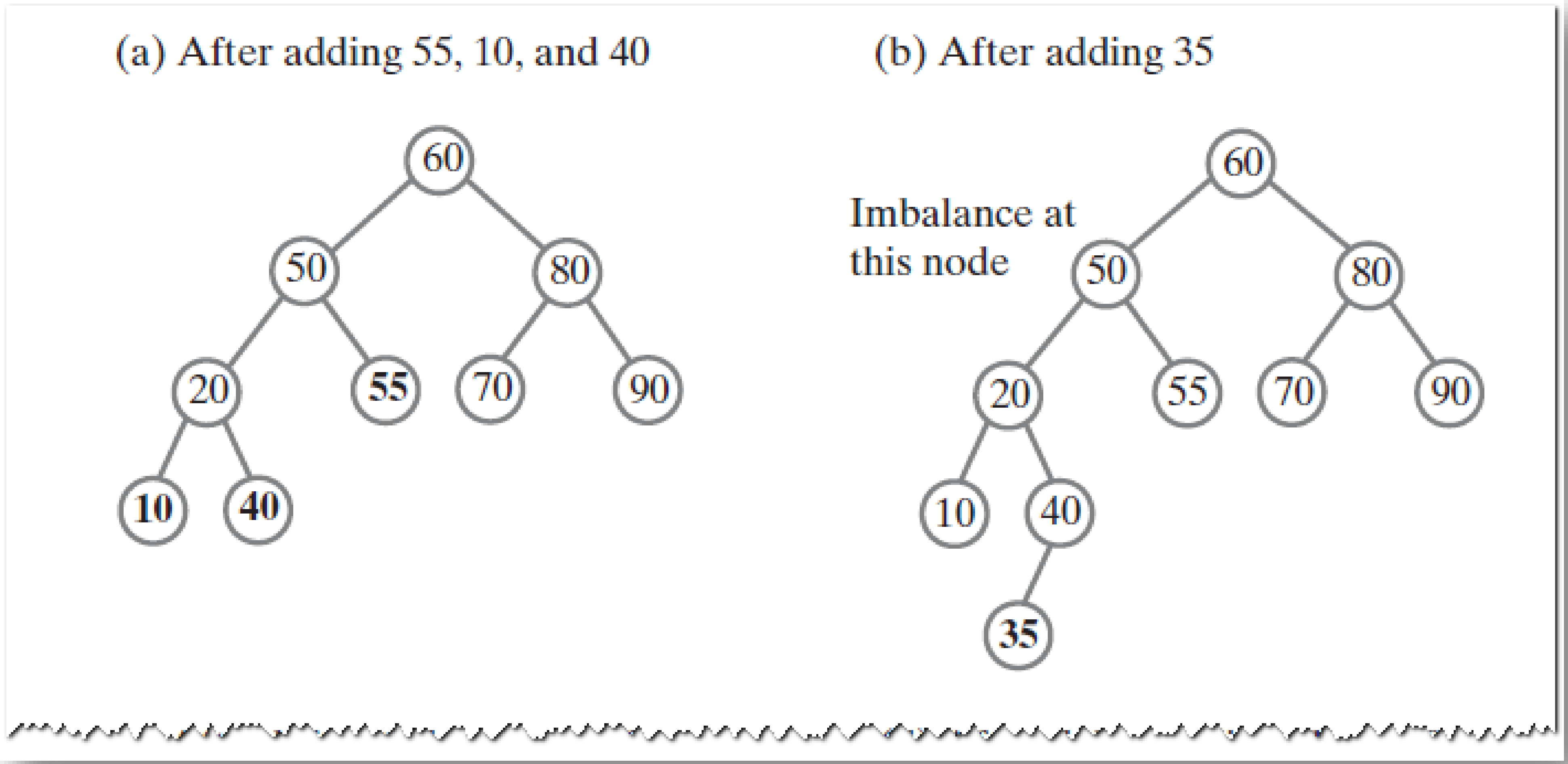




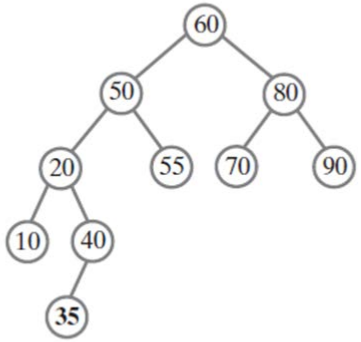
Algorithm `rotateRightLeft(nodeN)`
// Corrects an imbalance at a given node `nodeN` *due to an addition*
// in the left subtree of `nodeN`*'s right child.*

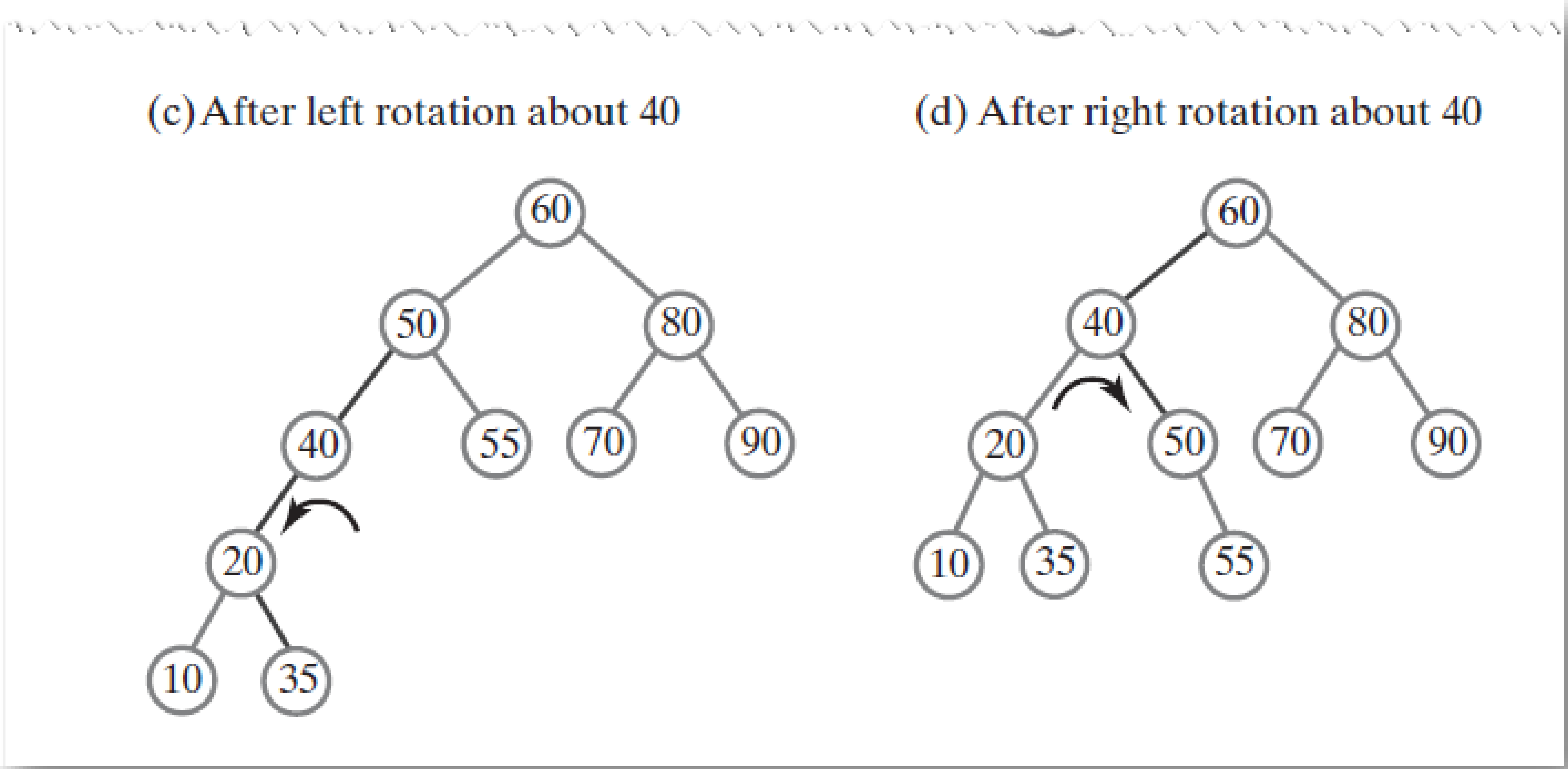
`nodeC =` *right child of* `nodeN`
Set `nodeN`*'s right child to the node returned by* `rotateRight(nodeC)`
`return` `rotateLeft(nodeN)`

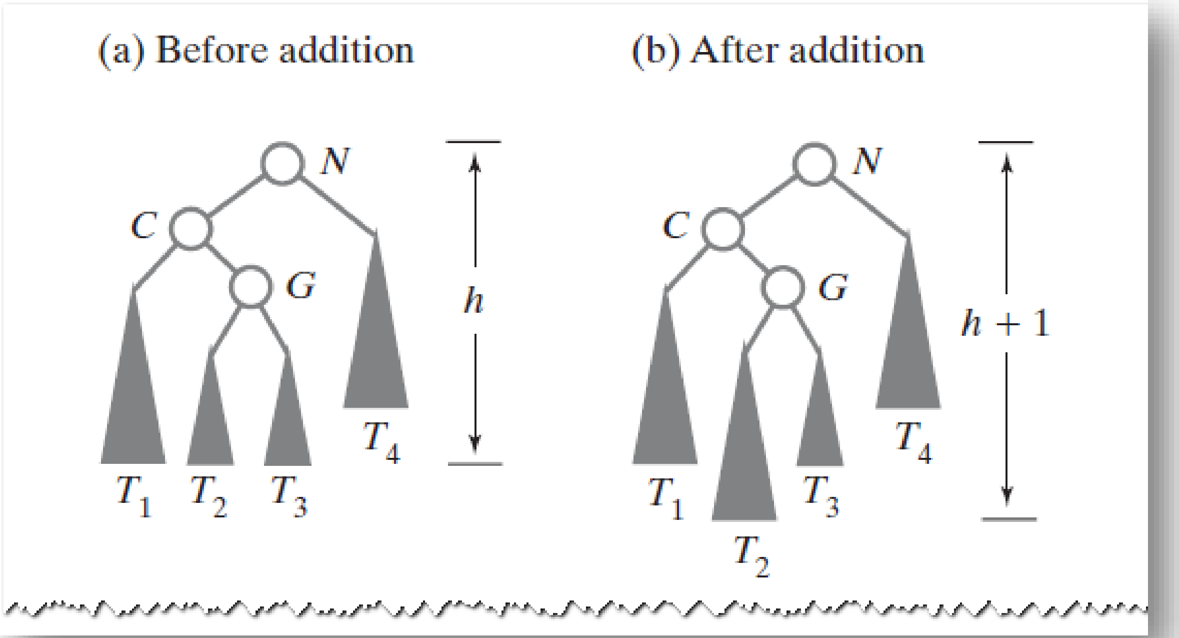
Double Rotations: **Left-Right** double rotations



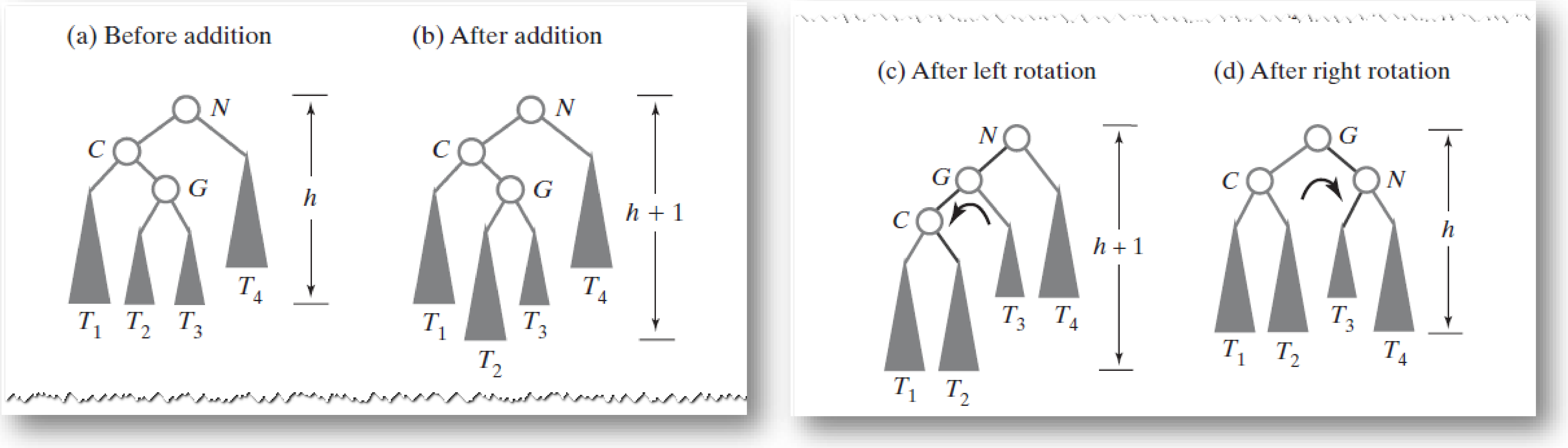
Double Rotations:
Left-Right double rotations

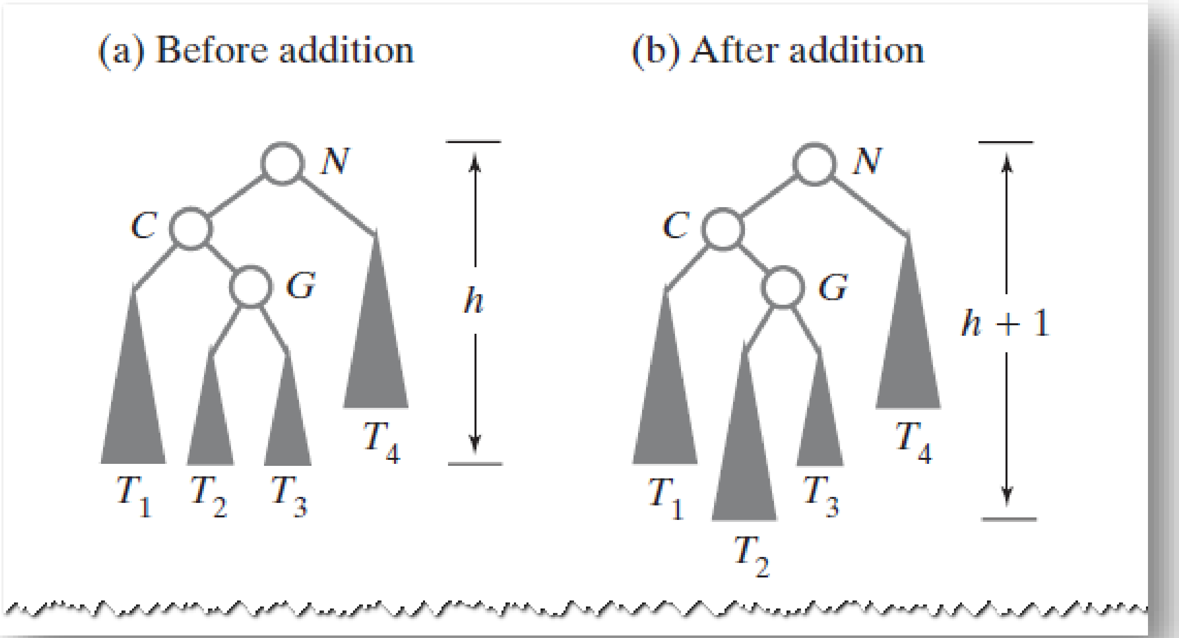


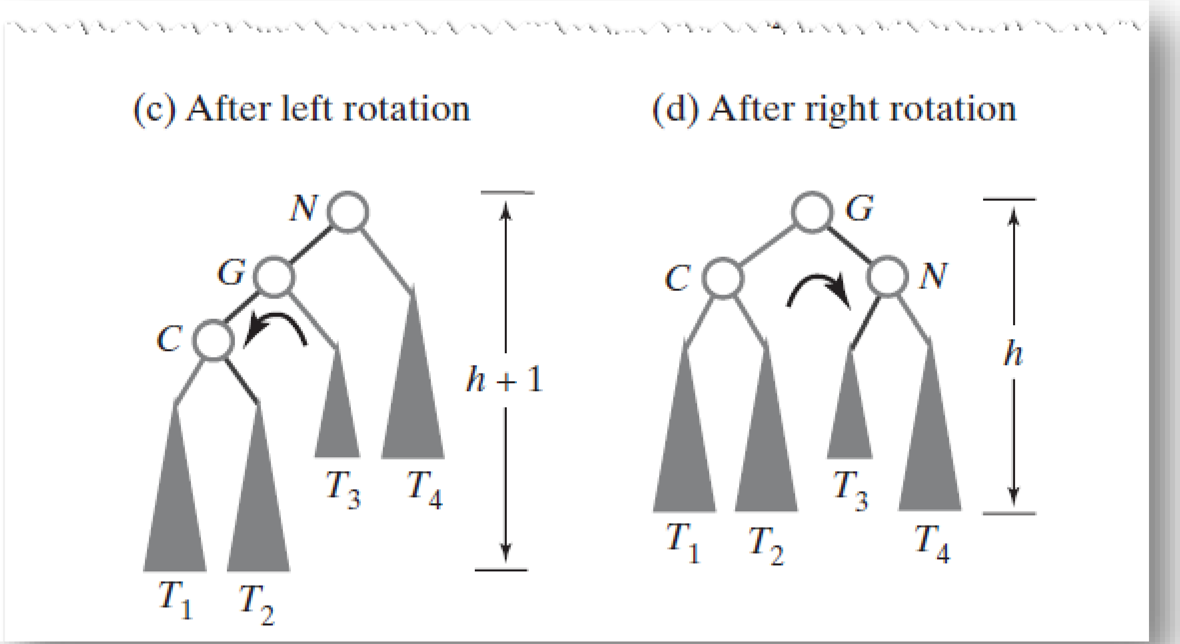




(a) Before addition
(b) After addition
N
C
G
T_1
T_2
T_3
T_4
h
N
C
G
T_1
T_2
T_3
T_4
$h + 1$


(a) Before addition
N
C
G
h
T1
T2
T3
T4
(b) After addition
N
C
G
h + 1
T1
T2
T3
T4
(c) After left rotation
N
G
C
h + 1
T1
T2
T3
T4
(d) After right rotation
G
C
N
h
T1
T2
T3
T4





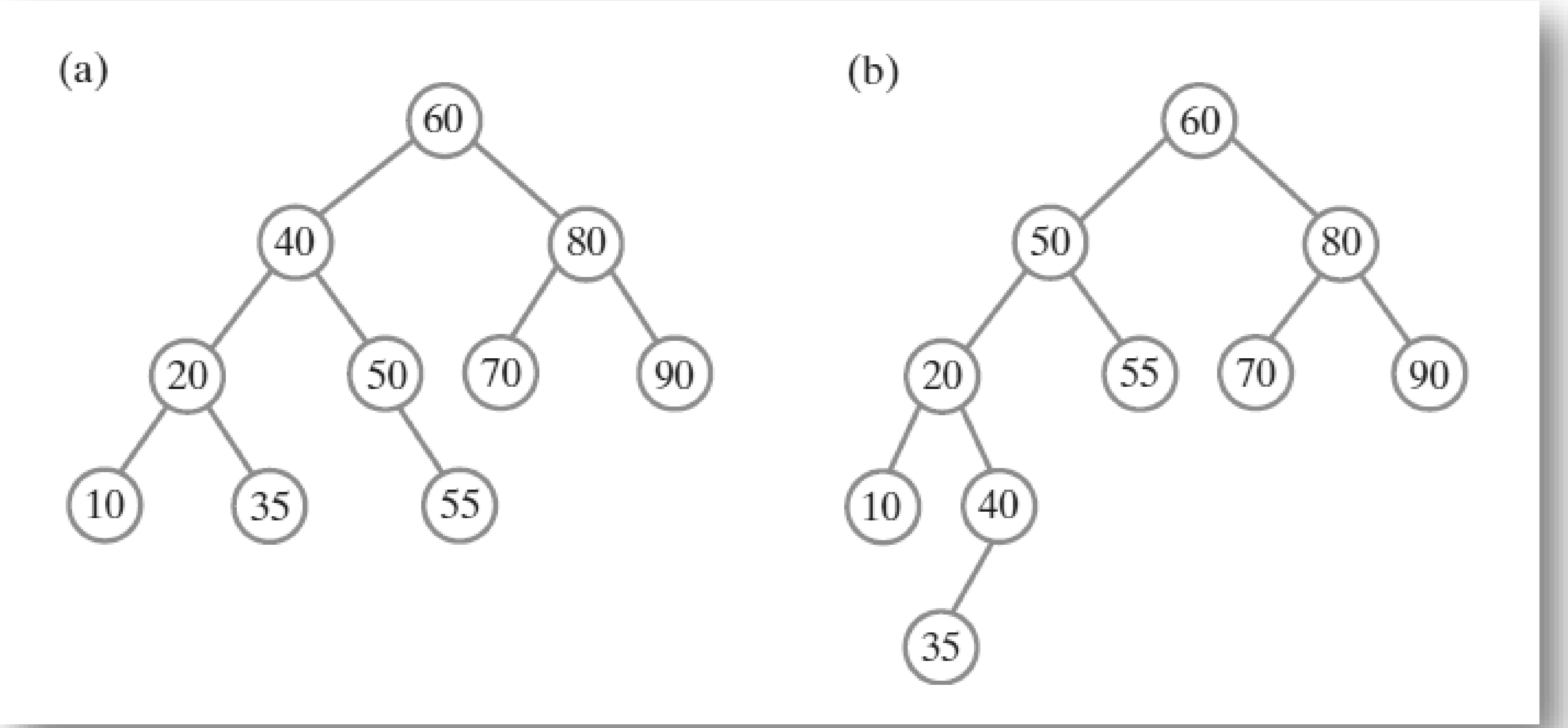
Algorithm **`rotateLeftRight(nodeN)`**
// Corrects an imbalance at a given node `nodeN` *due to an addition*
// in the right subtree of `nodeN`*'s left child.*

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

Summary

- Following an addition to an AVL tree, a temporary imbalance might occur. Let N be an unbalanced node that is closest to the new leaf. Either a single or double rotation will restore the tree's balance. No other rotations are necessary.
- The four rotations cover the only four possibilities for the cause of the imbalance at node N:
    1. The addition occurred in the left subtree of N's left child (right rotation)
    2. The addition occurred in the right subtree of N's left child (left-right rotation)
    3. The addition occurred in the left subtree of N's right child (right-left rotation)
    4. The addition occurred in the right subtree of N's right child (left rotation)

An AVL tree versus a binary search tree



The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) AVL tree; (b) binary search tree.

References

- F. M. Carrano & T. M. Henry, “Data Structures and Abstractions with Java”, 4th ed., 2015. Pearson Education, Inc.