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.

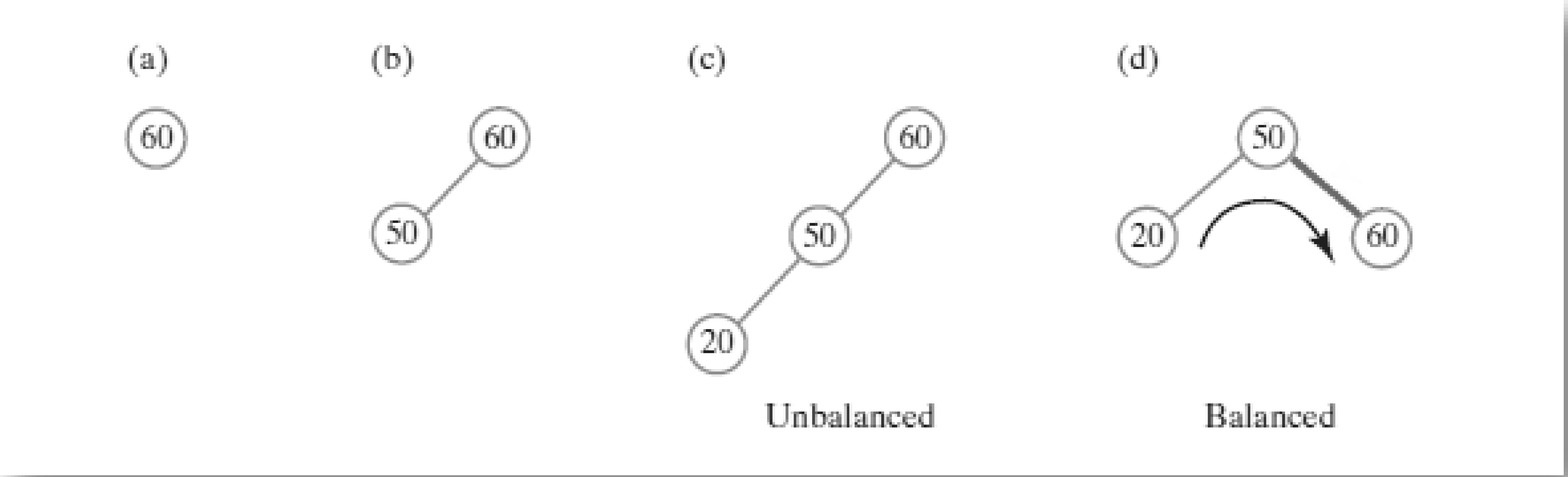
AVL Trees

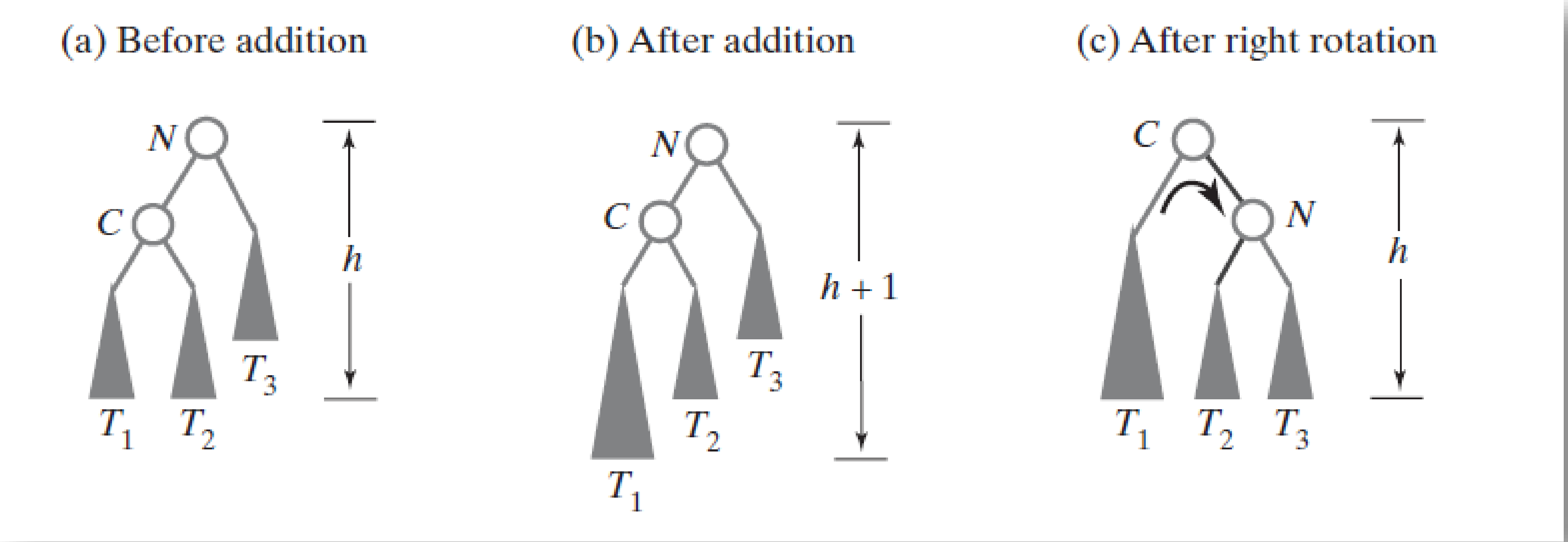
- The idea of rearranging nodes to balance a tree was first developed in 1962 by two Russian (USSR) mathematicians, Adel’son-Vel’skii and Landis. Named after them, the **AVL tree** is a *BST* that rearranges its nodes whenever it becomes unbalanced.

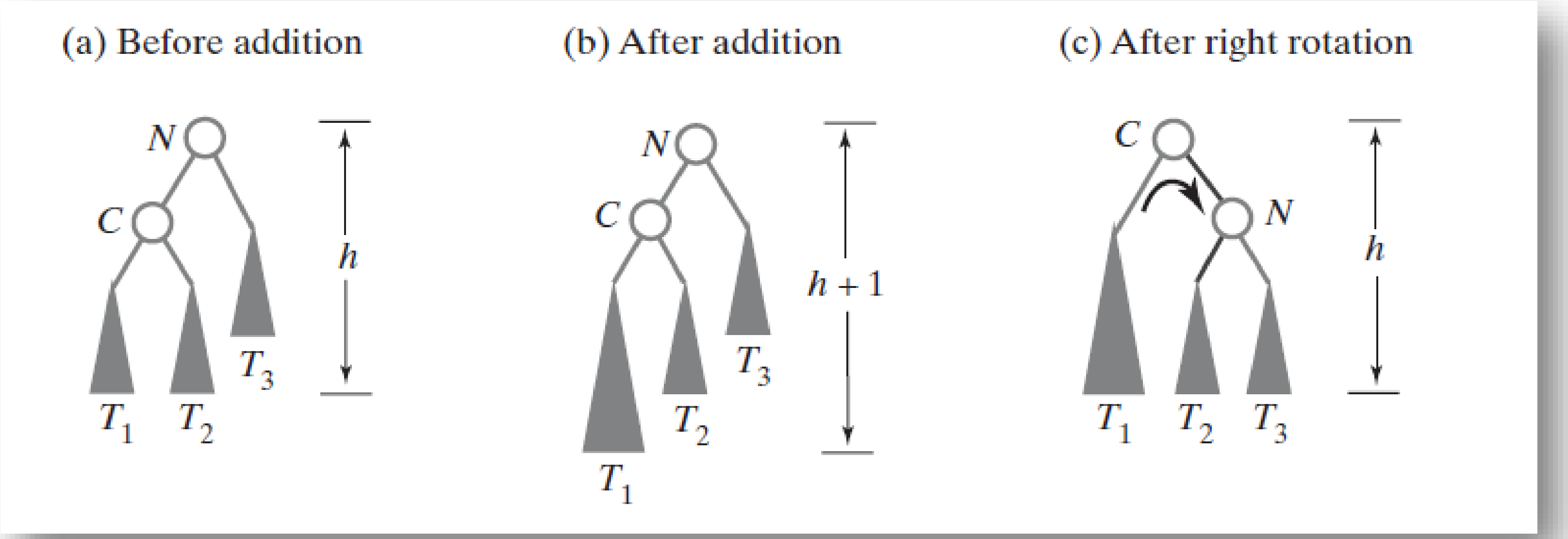
- The balance of a binary search tree is upset only when you *add* or *remove* a node. Thus, during these operations, the AVL tree rearranges nodes as necessary to maintain its balance.

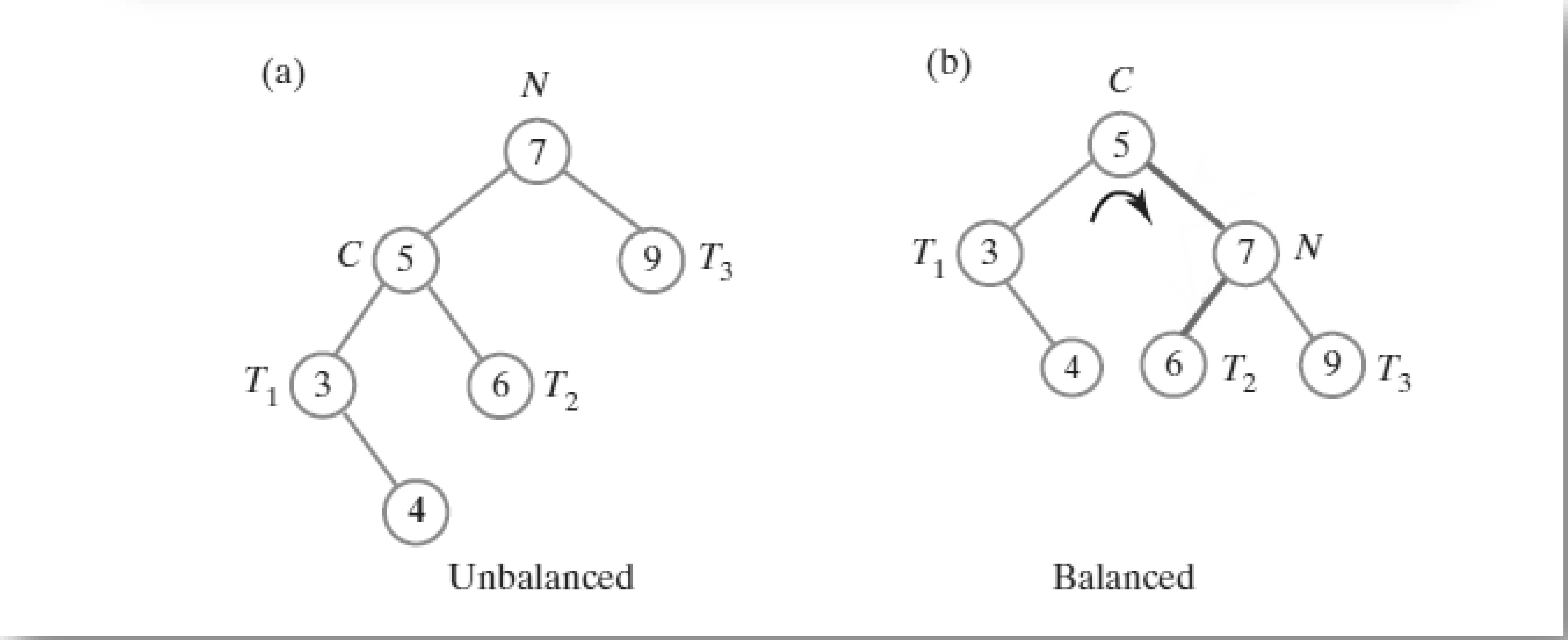
Single Rotations

Single Rotations: **Right** rotations



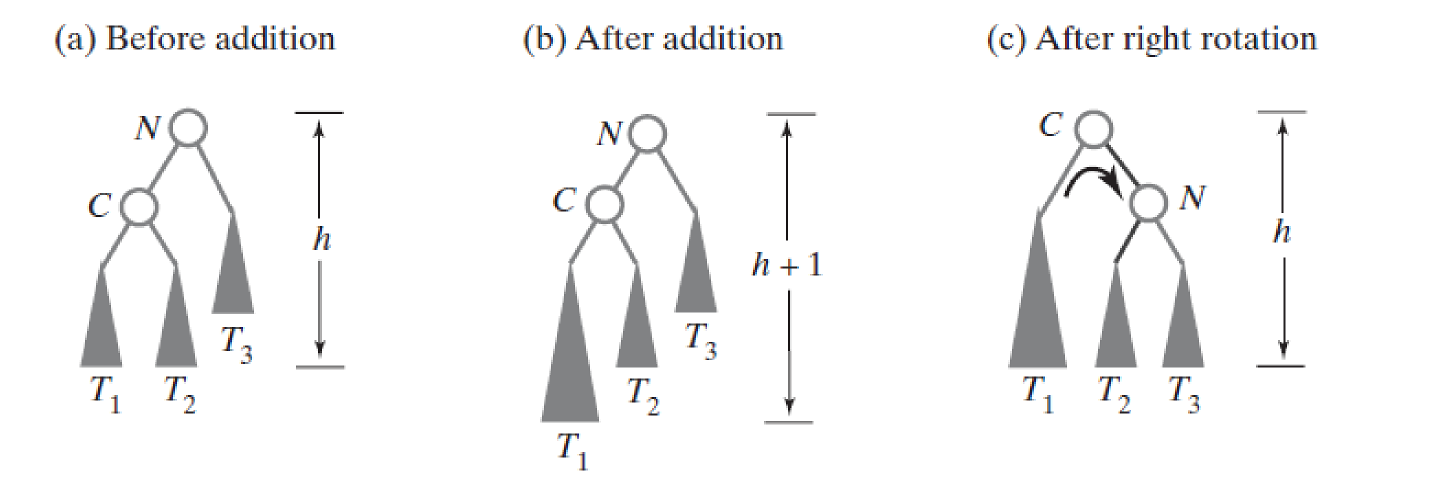

(a) Before addition
N
C
T_1
T_2
T_3
h
(b) After addition
N
C
T_1
T_2
T_3
$h + 1$
(c) After right rotation
C
N
T_1
T_2
T_3
h


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

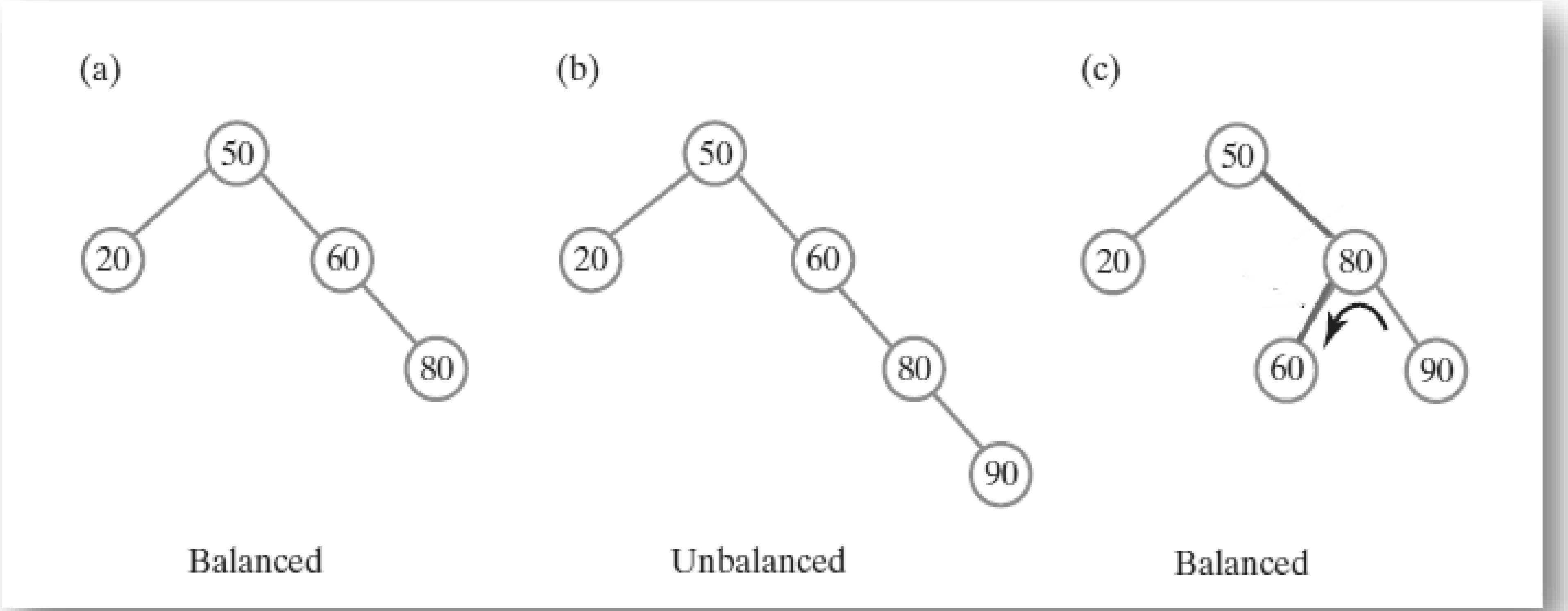

(a)
(b)
N
C
7
5
9
3
6
4
T_1
T_2
T_3
Unbalanced
Balanced

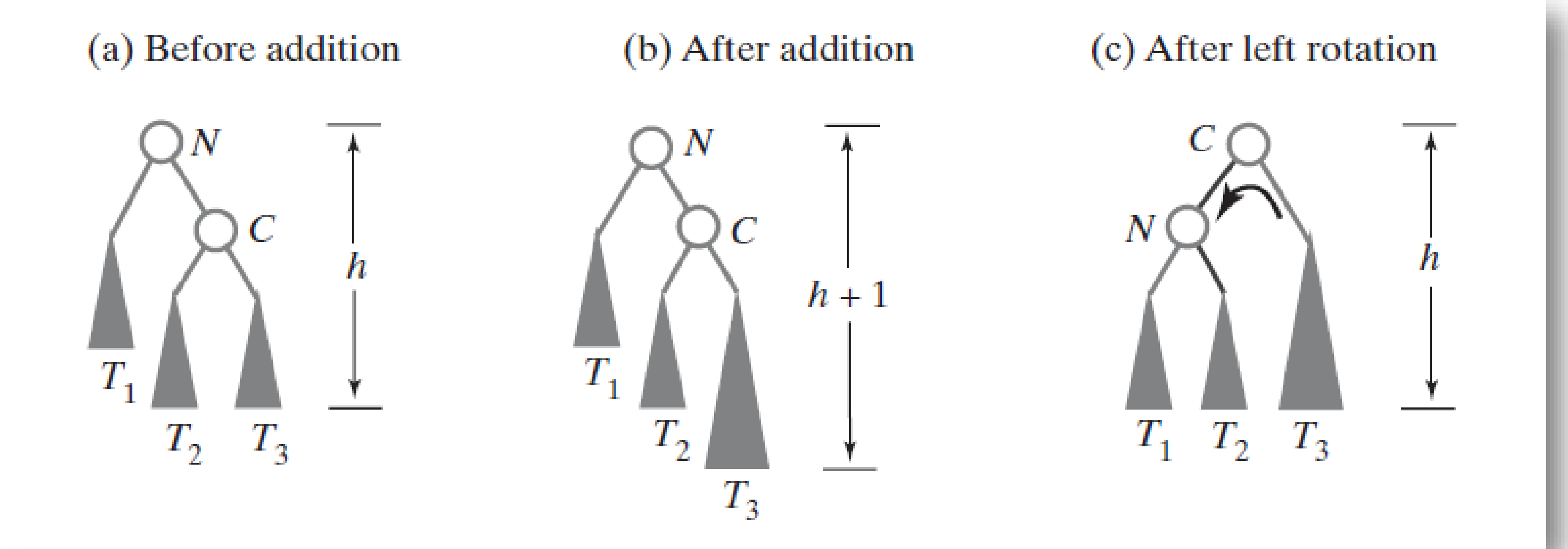
Algorithm **rotateRight(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
// in the left subtree of nodeN*'s left child.*

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

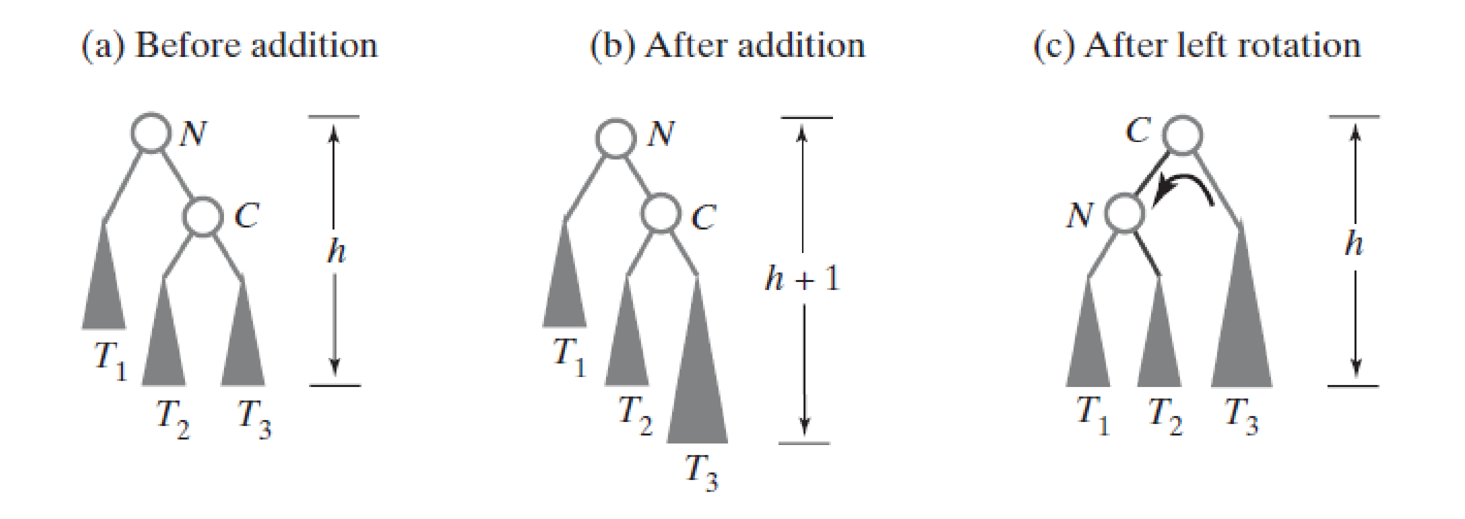


Single Rotations: **Left** Rotations



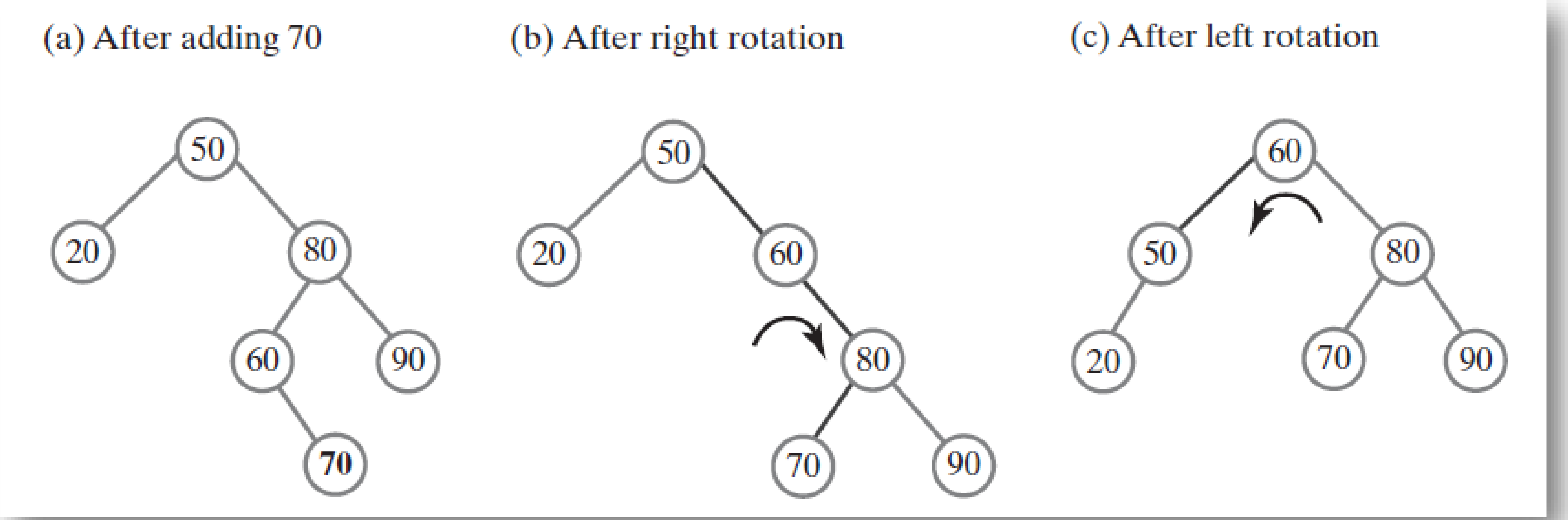

(a) Before addition
N
C
h
T_1
T_2
T_3
(b) After addition
N
C
h + 1
T_1
T_2
T_3
(c) After left rotation
C
N
h
T_1
T_2
T_3

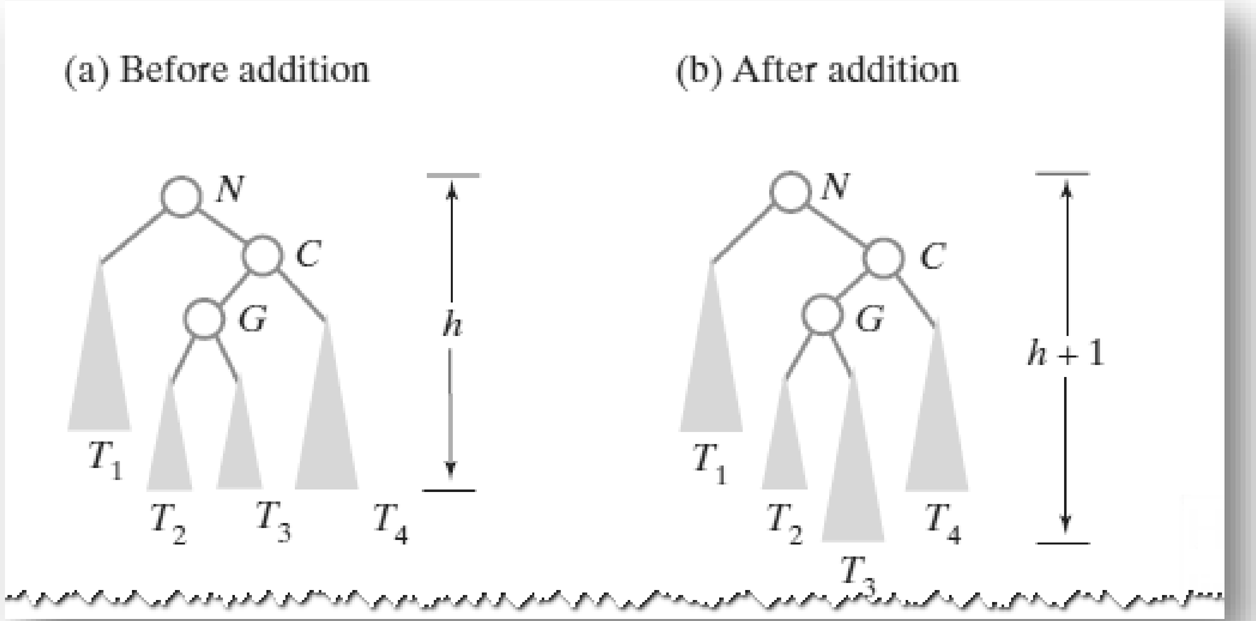
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

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

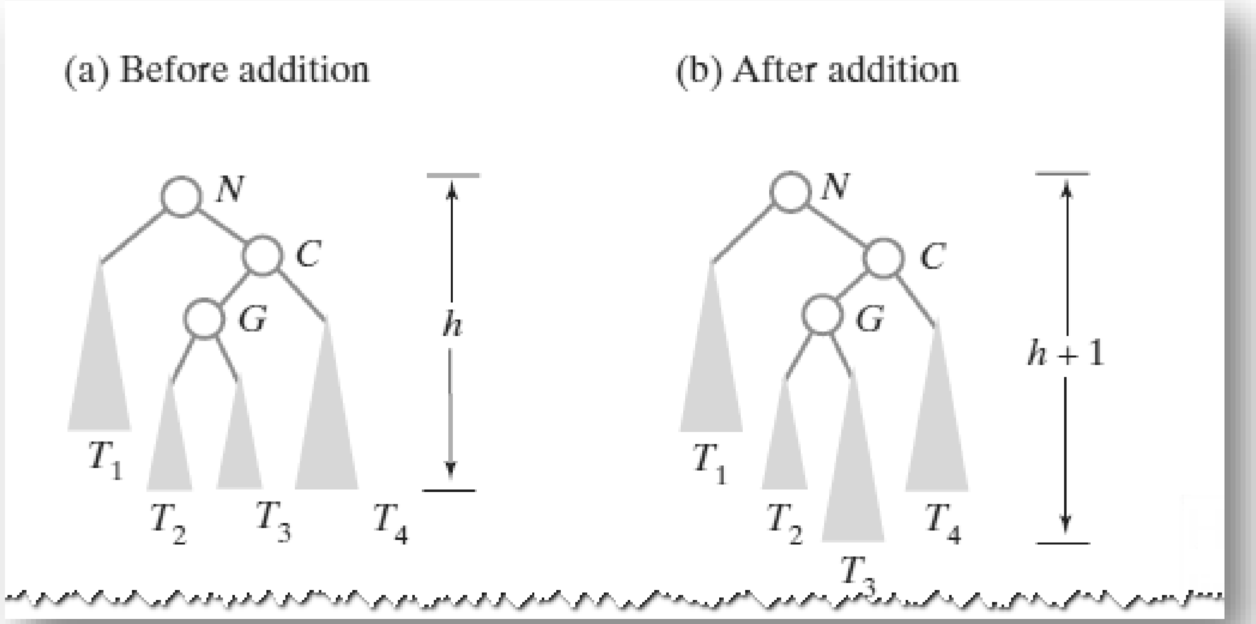


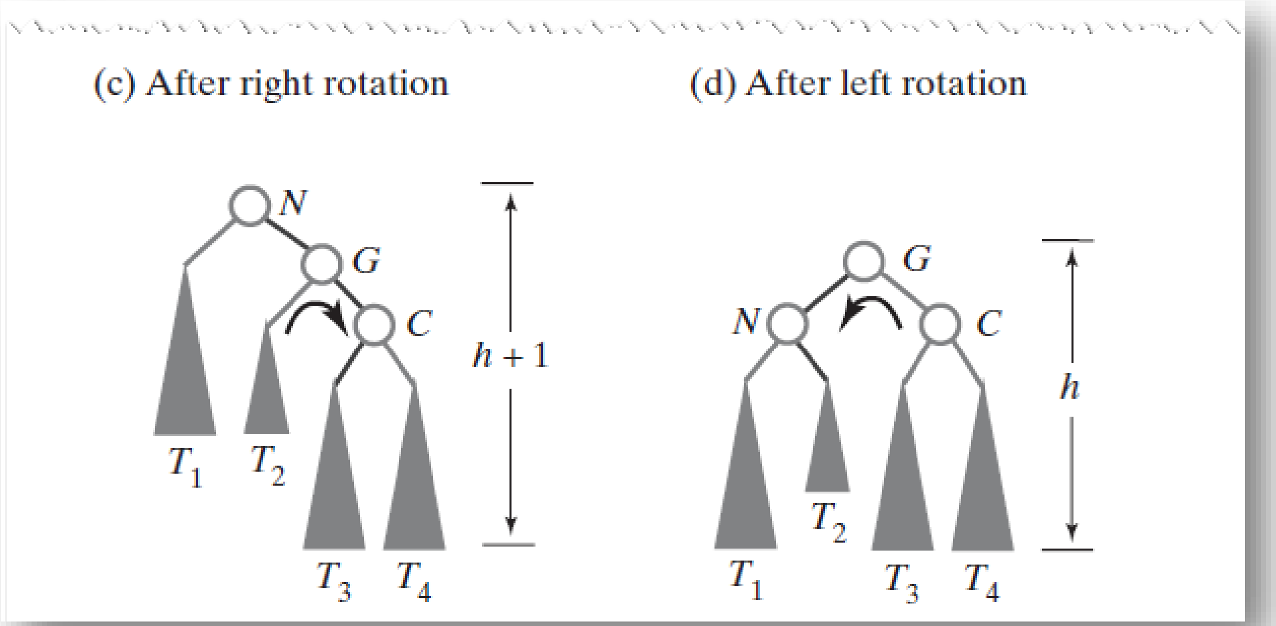
Double Rotations

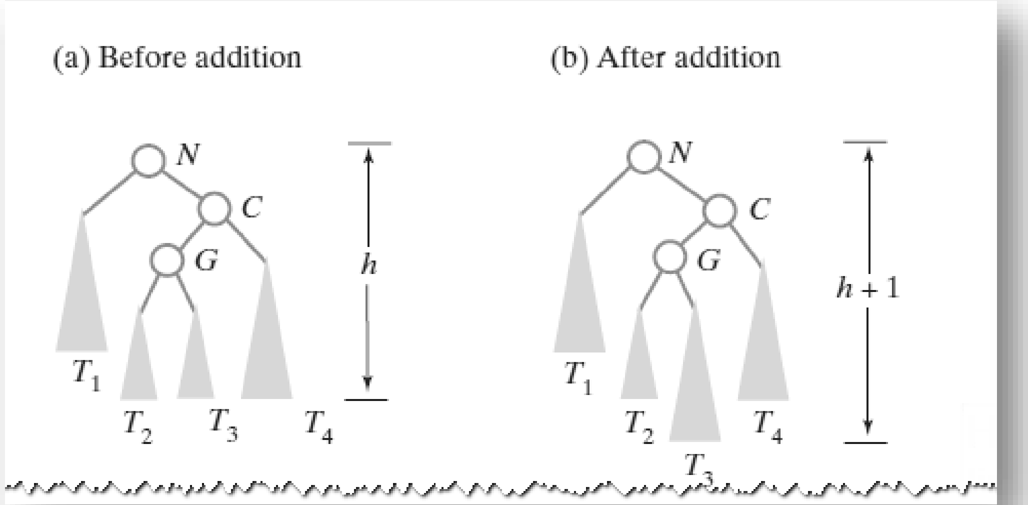
Double Rotations: **Right-Left** double rotations

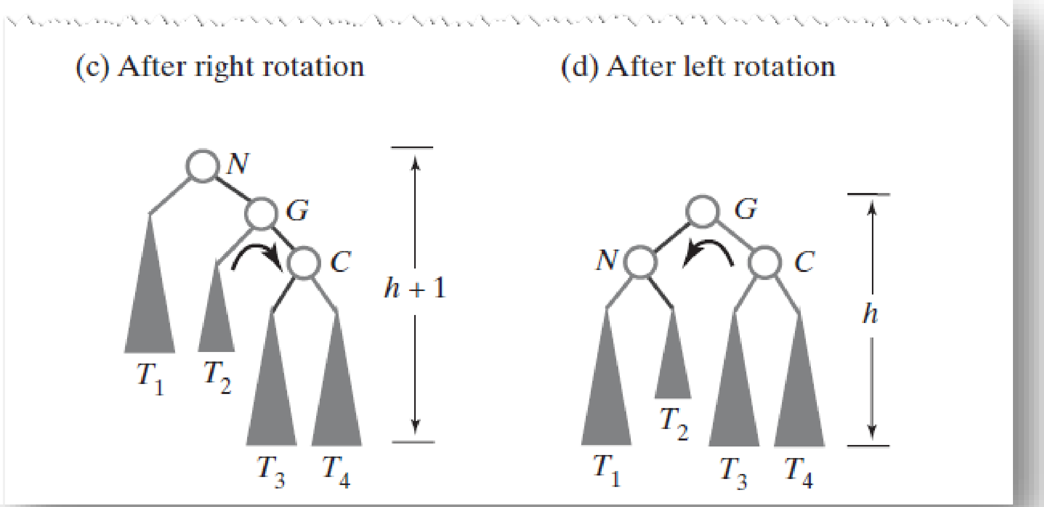



(a) Before addition
(b) After addition
N
C
G
h
h + 1
T1
T2
T3
T4


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


(c) After right rotation
(d) After left rotation
N
G
C
T_1
T_2
T_3
T_4
$h+1$
h

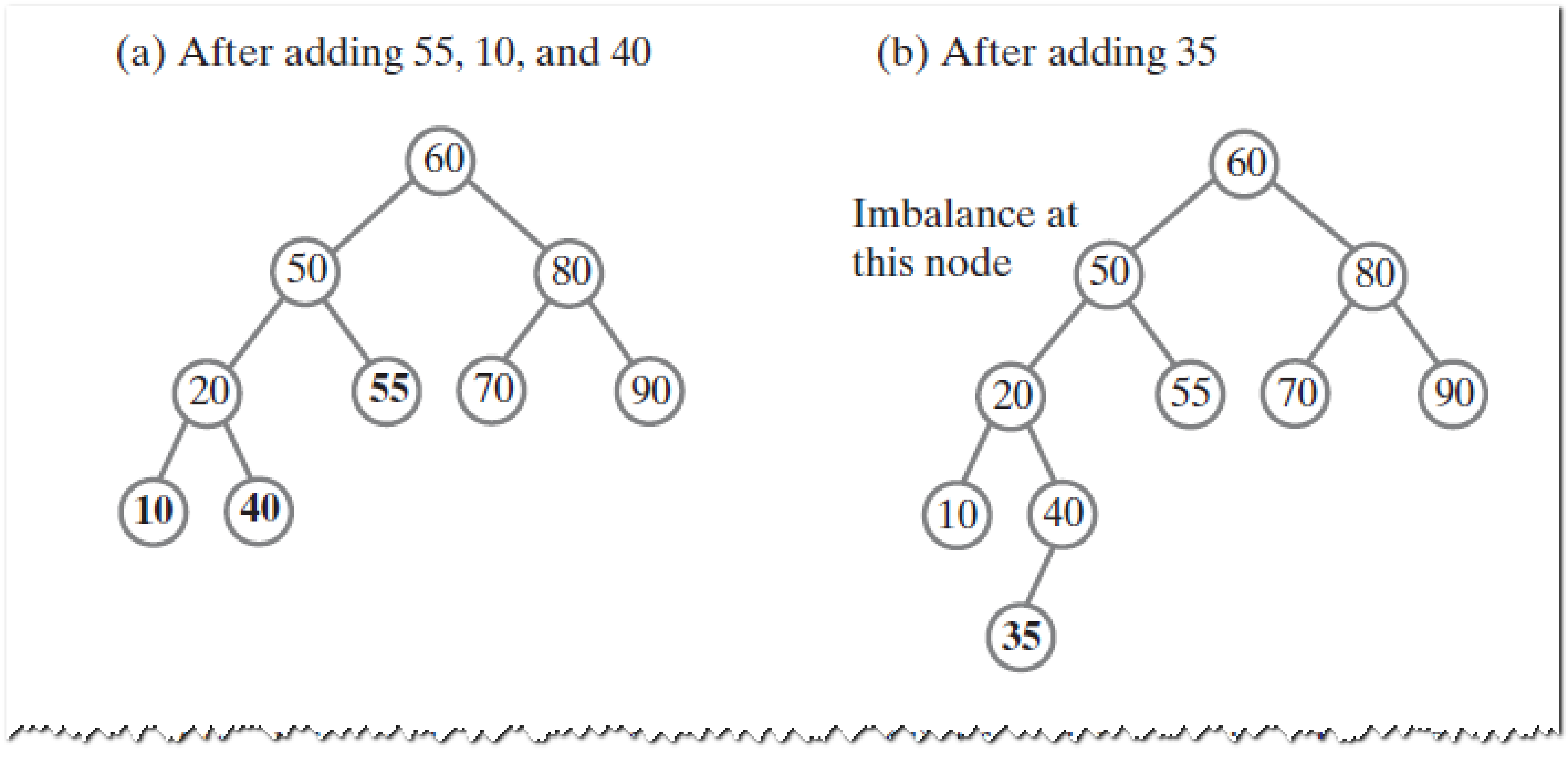




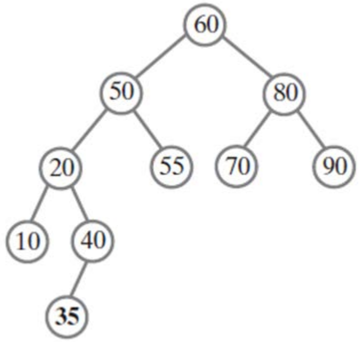
Algorithm `rotateRightLeft(nodeN)`
// Corrects an imbalance at a given node `nodeN` *due to an addition*
// in the left subtree of `nodeN`*'s right child.*

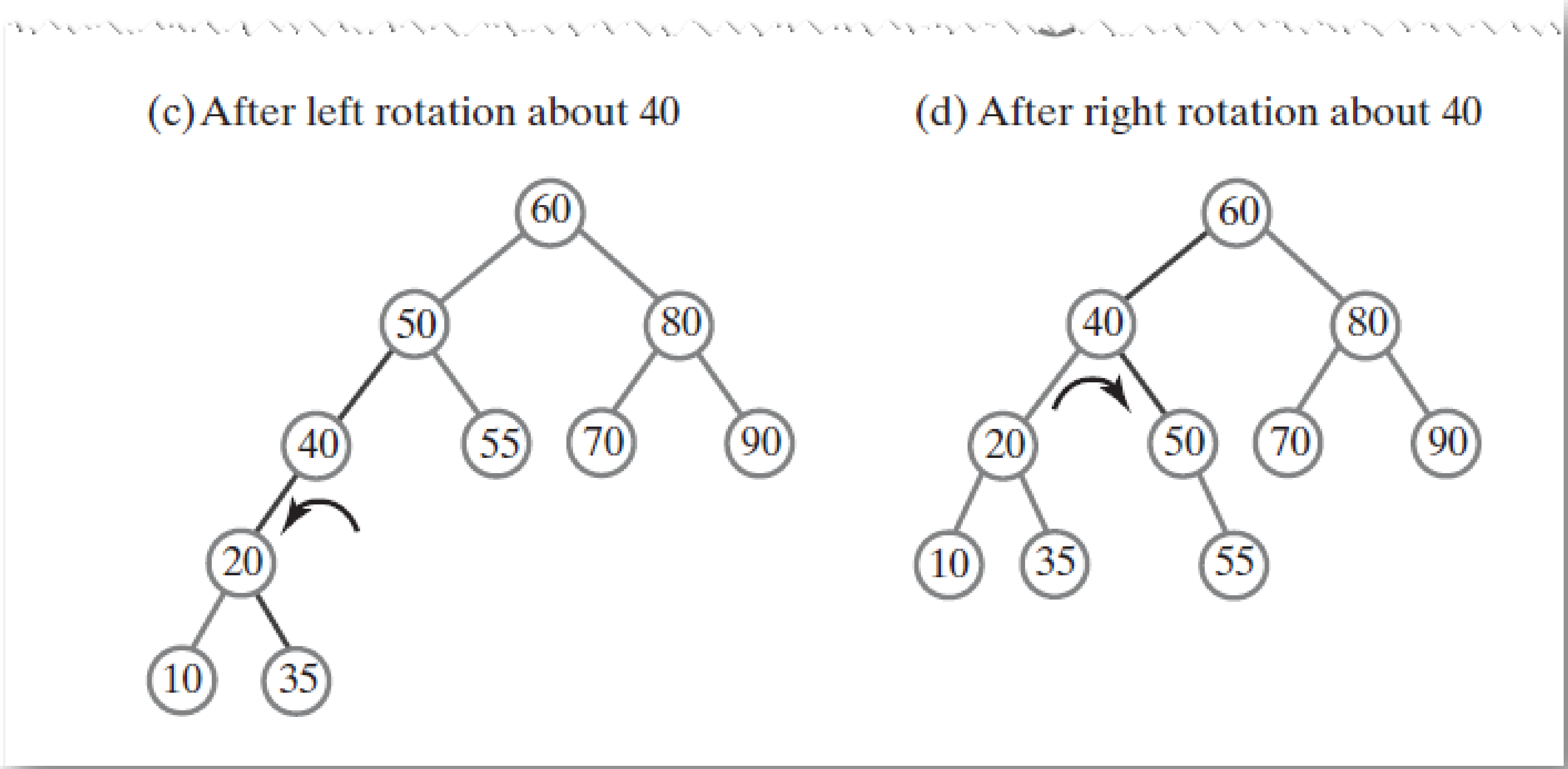
`nodeC` = *right child of* `nodeN`
Set `nodeN`*'s right child to the node returned by* `rotateRight(nodeC)`
`return` `rotateLeft(nodeN)`

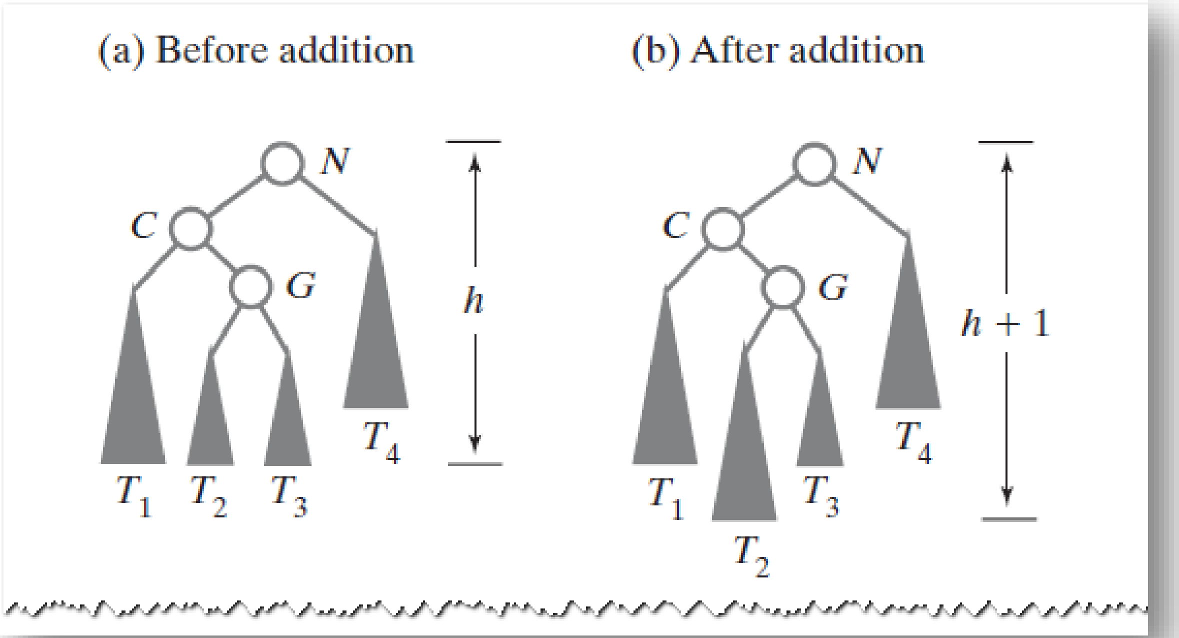
Double Rotations: **Left-Right** double rotations

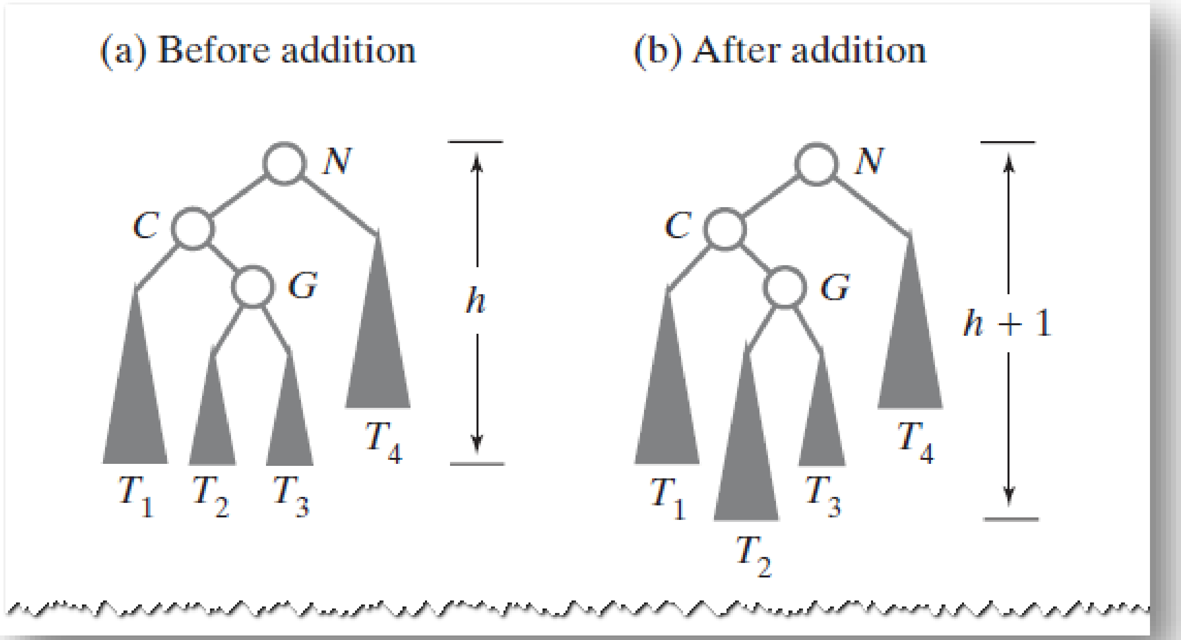


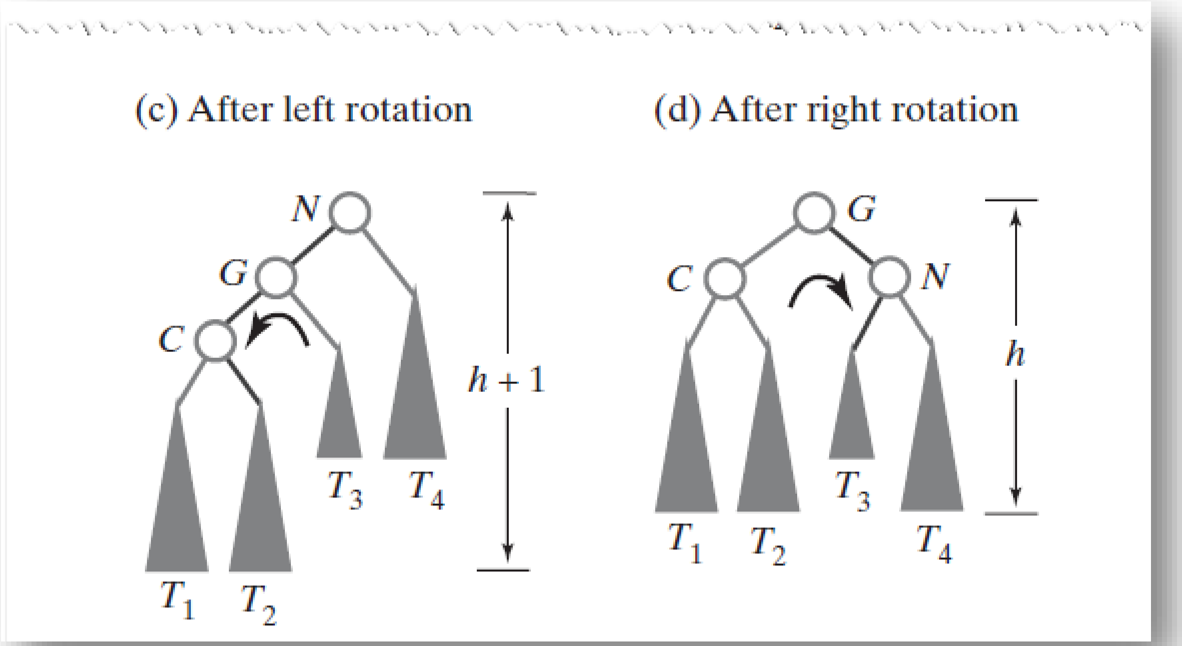
Double Rotations:
Left-Right double rotations

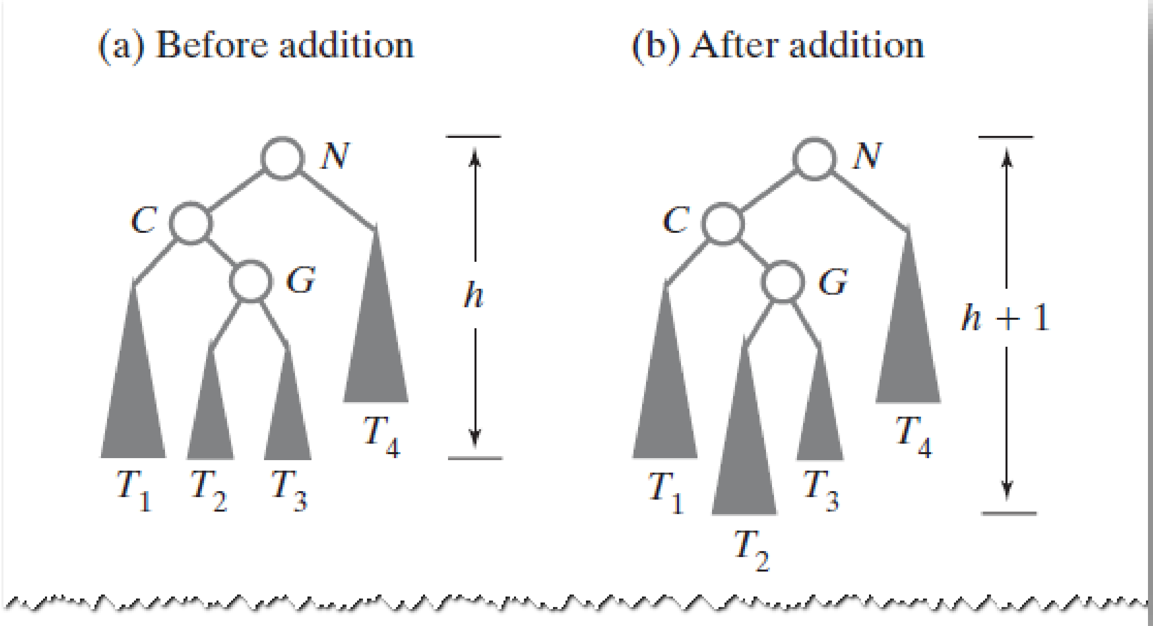


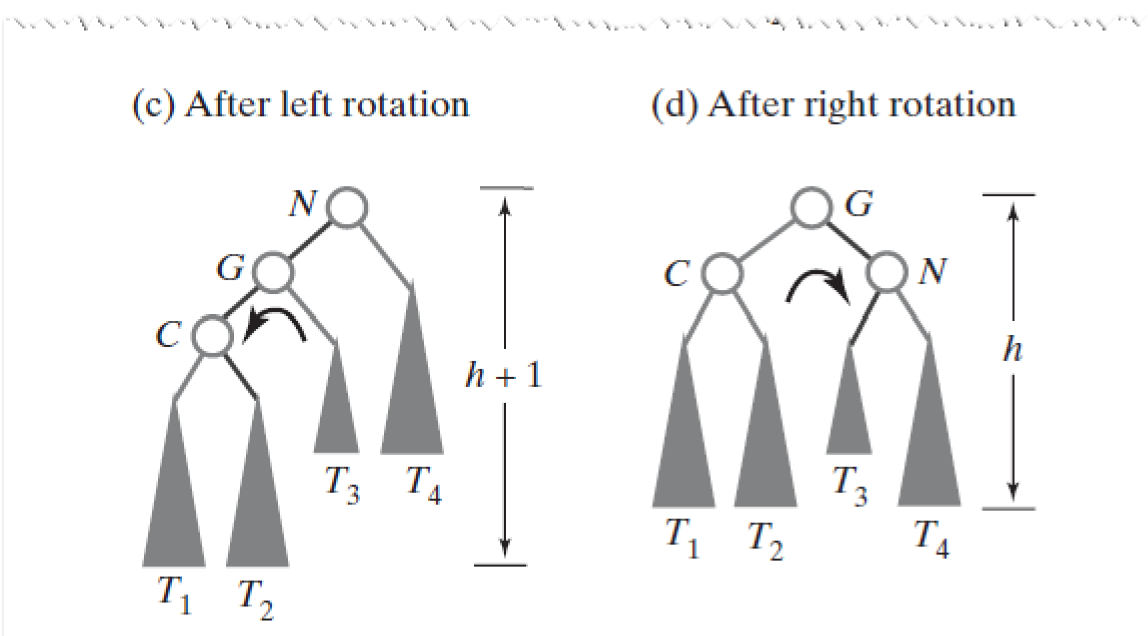



(a) Before addition
(b) After addition
N
C
G
T_1
T_2
T_3
T_4
h
N
C
G
T_1
T_2
T_3
T_4
$h + 1$


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


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





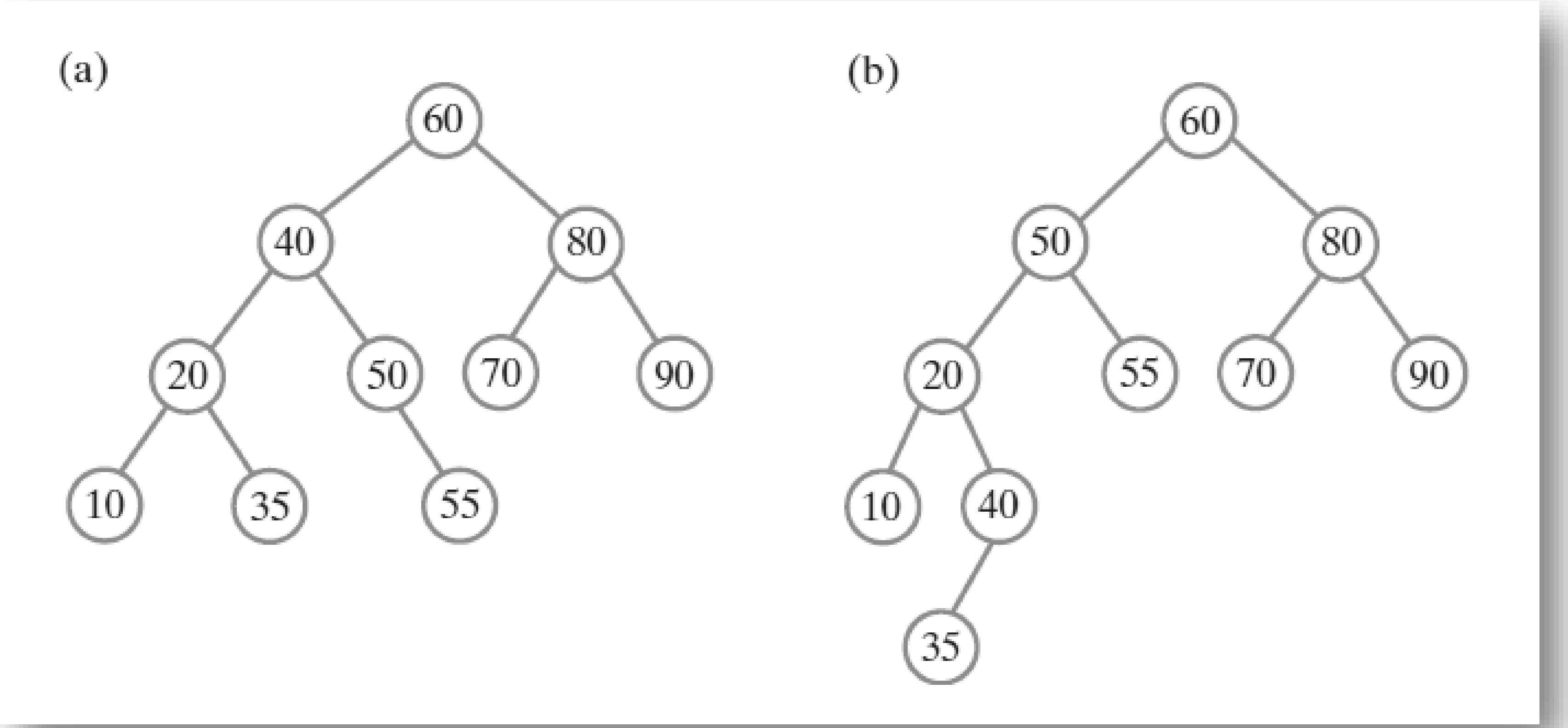
Algorithm **`rotateLeftRight(nodeN)`**
// Corrects an imbalance at a given node `nodeN` *due to an addition*
// in the right subtree of `nodeN`*'s left child.*

`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)

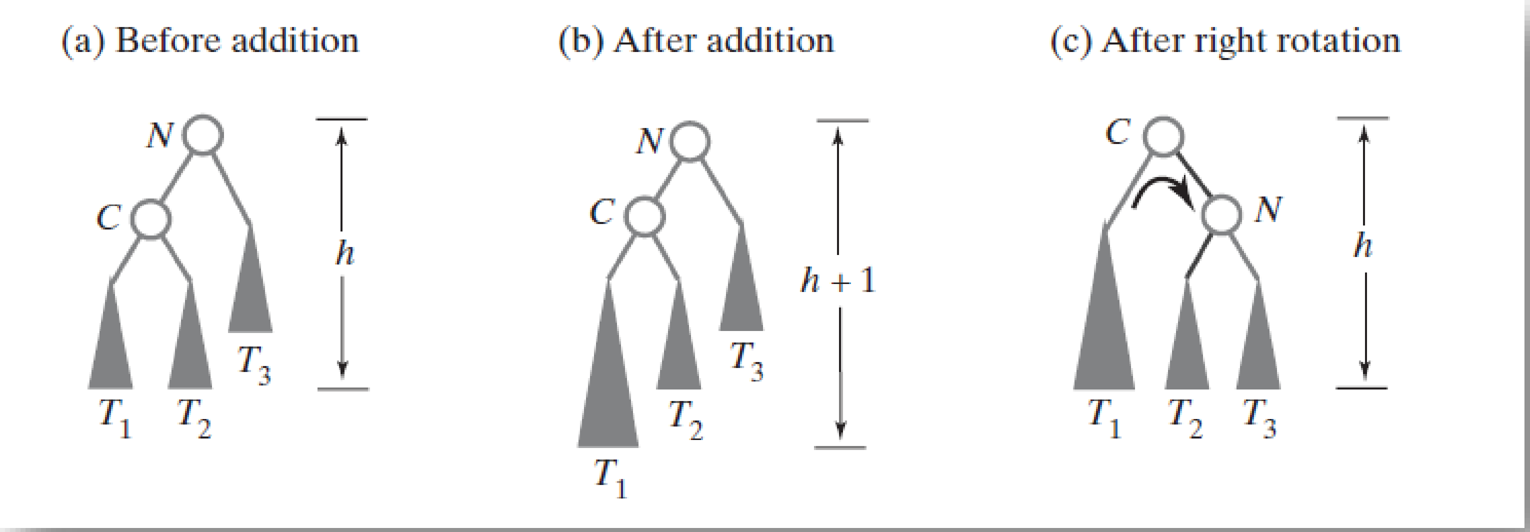
An AVL tree versus a binary search tree



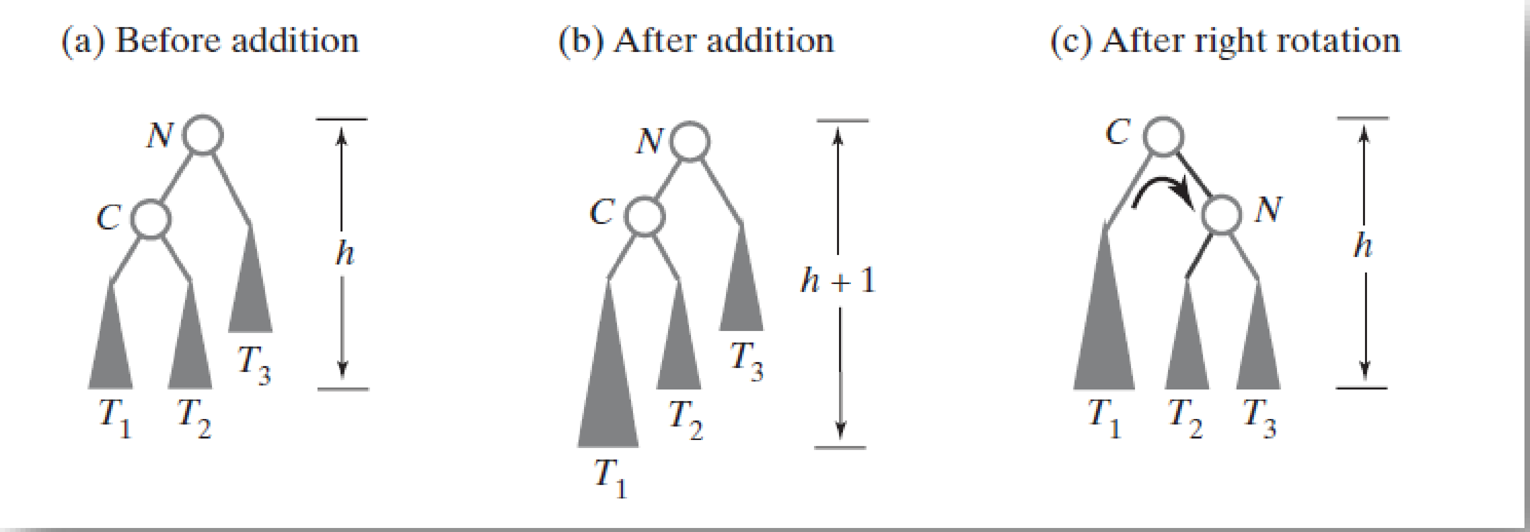
The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) AVL tree; (b) binary search tree.

```java
package TreePackage;
/**
 * A class that implements the ADT AVL tree by extending BinarySearchTree.
 * The remove operation is not supported.
 */
public class AVLTree<T extends Comparable<? super T>>
       extends BinarySearchTree<T> implements SearchTreeInterface<T>
{
 public AVLTree()
 {
  super();
 } // end default constructor

 public AVLTree(T rootEntry)
 {
  super(rootEntry);
 } // end constructor
 // other methods ...
} // end AVLTree
```

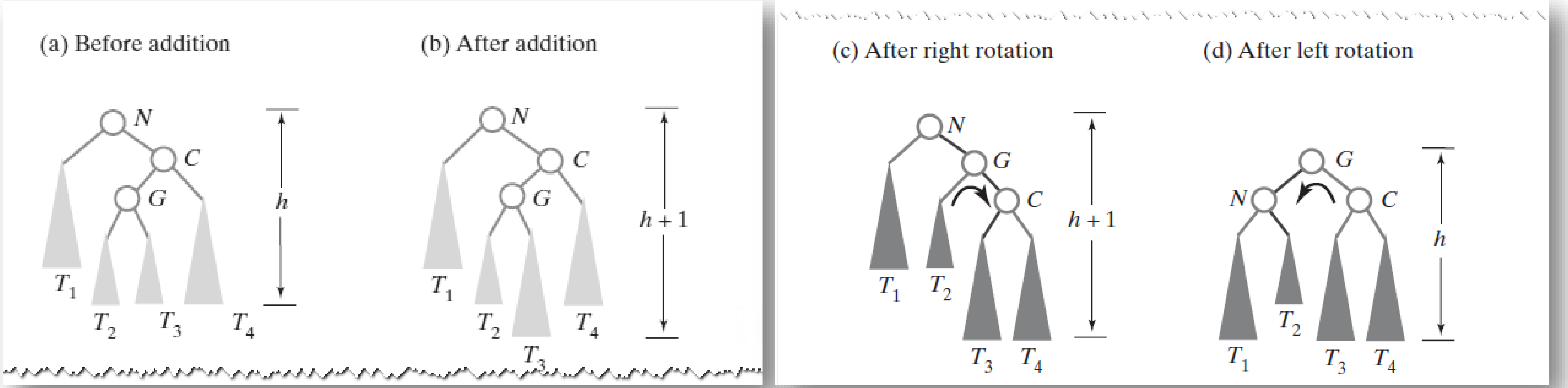
```
//Algorithm rotateRight(nodeN)
//nodeC = left child of nodeN
//Set nodeN's left child to nodeC's right child
//Set nodeC's right child to nodeN
//return nodeC
```



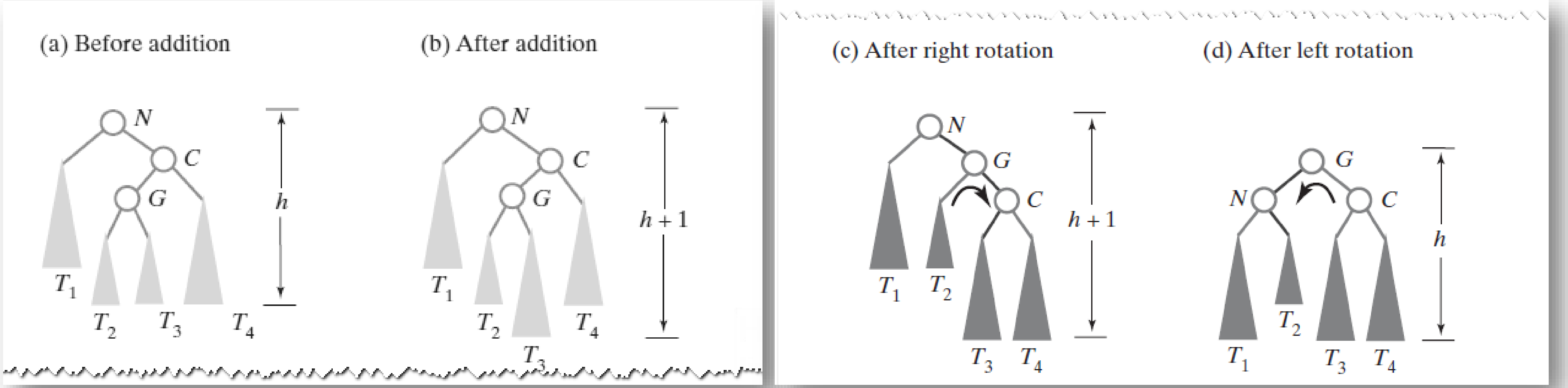
```java
//Algorithm rotateRight(nodeN)
//nodeC = left child of nodeN
//Set nodeN's left child to nodeC's right child
//Set nodeC's right child to nodeN
//return nodeC

private BinaryNode<T> rotateRight(BinaryNode<T> nodeN)
{
 BinaryNode<T> nodeC = nodeN.getLeftChild();
 nodeN.setLeftChild(nodeC.getRightChild());
 nodeC.setRightChild(nodeN);
 return nodeC;
} // end rotateRight
```



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



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

private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN)
{
 BinaryNode<T> nodeC = nodeN.getRightChild();
 nodeN.setRightChild(rotateRight(nodeC));
 return rotateLeft(nodeN);
} // end rotateRightLeft
```



Rebalancing: The method **rebalance**

```
private BinaryNode<T> rebalance(BinaryNode<T> nodeN)
{
 int heightDifference = getHeightDifference(nodeN);

 if (heightDifference > 1)
 {
  if (getHeightDifference(nodeN.getLeftChild()) > 0)
   nodeN = rotateRight(nodeN);
  else
   nodeN = rotateLeftRight(nodeN);
 }
 else if (heightDifference < -1)
 {
  if (getHeightDifference(nodeN.getRightChild()) < 0)
   nodeN = rotateLeft(nodeN);
  else
   nodeN = rotateRightLeft(nodeN);
 } // end if

 return nodeN;
} // end rebalance
```

Rebalancing: The method **add**

```java
public T add(T newEntry)
{
 T result = null;

 if (isEmpty())
  setRootNode(new BinaryNode<>(newEntry));
 else
 {
  BinaryNode<T> rootNode = getRootNode();
  result = addEntry(rootNode, newEntry);
  setRootNode(rebalance(rootNode));
 } // end if

 return result;
} // end add
```

```java
private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
 assert rootNode != null;
 T result = null;
 int comparison = newEntry.compareTo(rootNode.getData());

 if (comparison == 0)
 {
  result = rootNode.getData();
  rootNode.setData(newEntry);
 }
 else if (comparison < 0)
 {
  if (rootNode.hasLeftChild())
  {
   BinaryNode<T> leftChild = rootNode.getLeftChild();
   result = addEntry(leftChild, newEntry);
   rootNode.setLeftChild(rebalance(leftChild));
  }
  else
   rootNode.setLeftChild(new BinaryNode<>(newEntry));
 }
```

```
 else
 {
  assert comparison > 0;

  if (rootNode.hasRightChild())
  {
   BinaryNode<T> rightChild = rootNode.getRightChild();
   result = addEntry(rightChild, newEntry);
   rootNode.setRightChild(rebalance(rightChild));
  }
  else
   rootNode.setRightChild(new BinaryNode<>(newEntry));
 } // end if

 return result;
} // end addEntry
```

Question

- What AVL tree results when you make the following additions to an initially empty AVL tree?

7, 8, 9, 2, 1, 5, 6, 4, 3

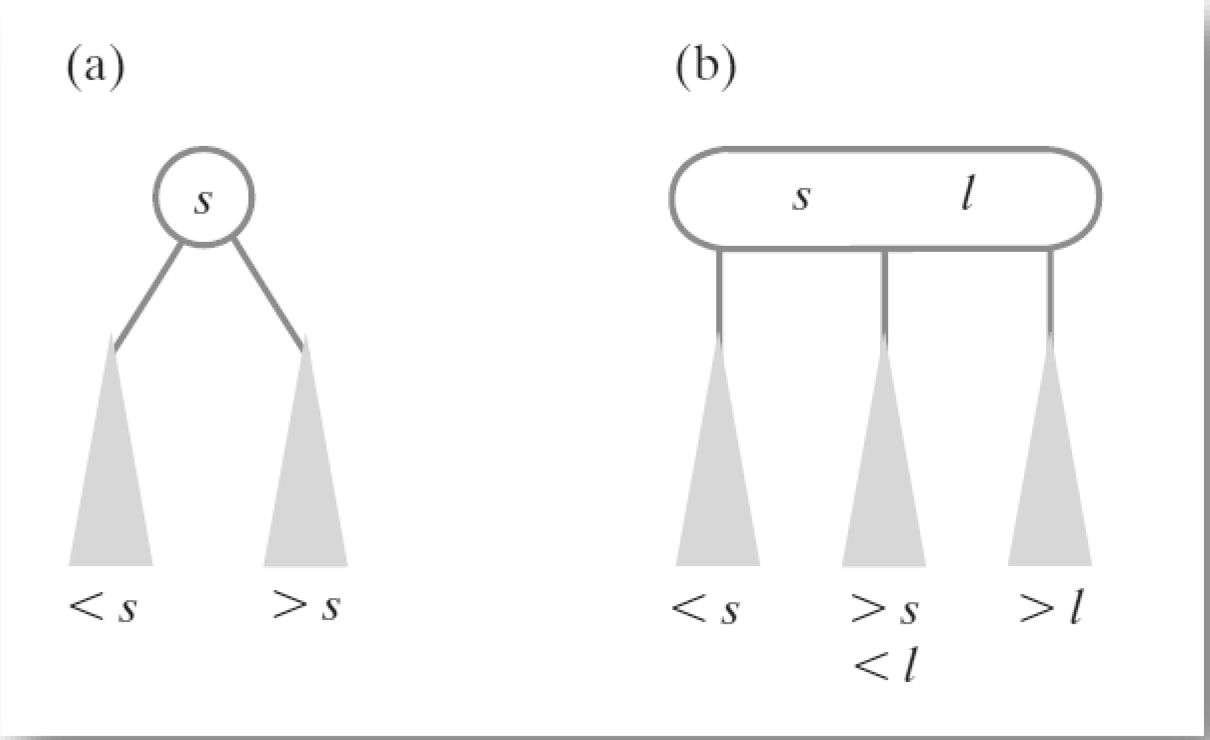
2-3 Trees

2-3 Trees

- A **2-3 Tree** is a general search tree whose interior nodes must have either two or three children.
- A **2-node** contains one data item s and has two children, like the nodes in a binary search tree, as follows:
  - This data s is greater than any data in the node's left subtree and less than any data in the right subtree. That is, the data in the node's left subtree is less than s, and any data in the right subtree is greater than s.
- A **3-node** contains two data items, s and l, and has three children, as follows:
  - Data that is less than the smaller data item s occurs in the node's left subtree.
  - Data that is greater than the larger data item l occurs in the node's right subtree.
  - Data that is between s and a l occurs in the node's middle subtree.

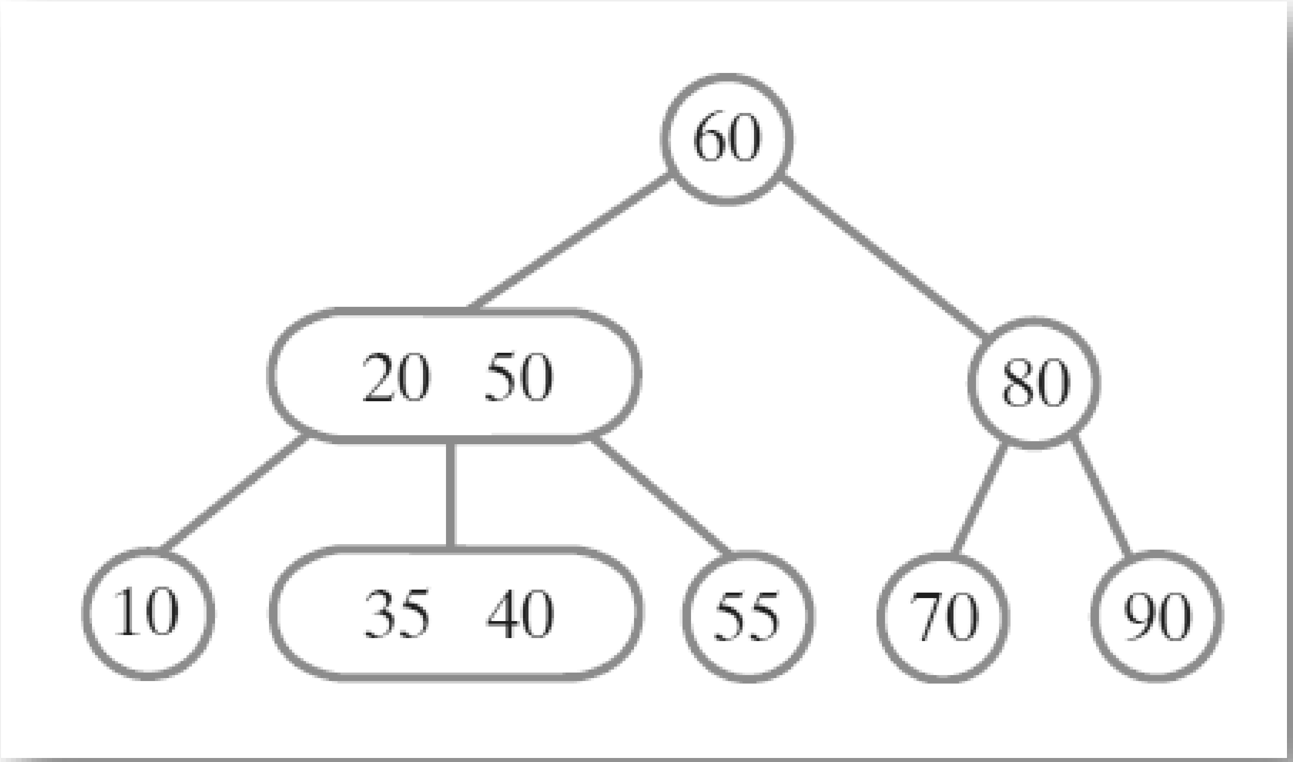
2-3 Trees (cont.)

- Because it can contain 3-nodes, a 2-3 tree tends to be shorter than a BST. To make the 2-3 tree balanced, we require that all leaves occur on the same level. Thus, a 2-3 tree is completely balanced.

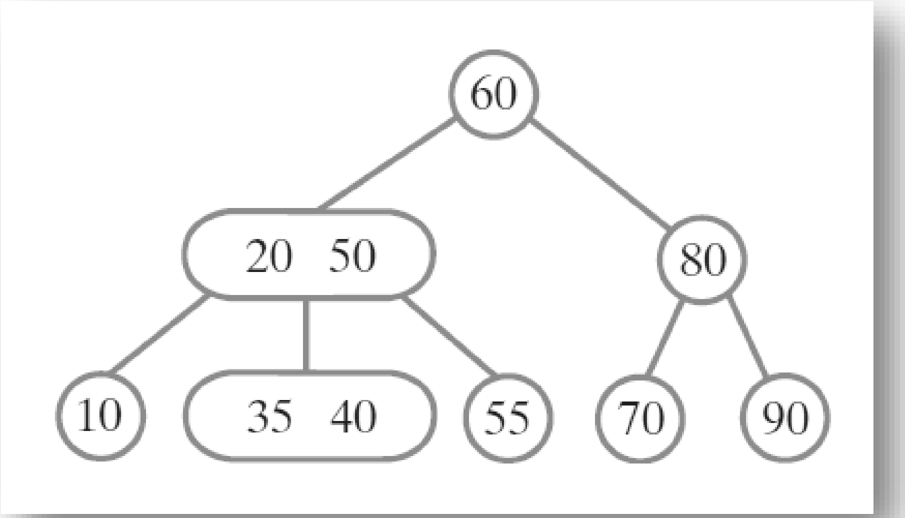


Nodes in a 2-3 tree
(a) A 2-node;
(b) A 3-node.

Searching a 2-3 Tree

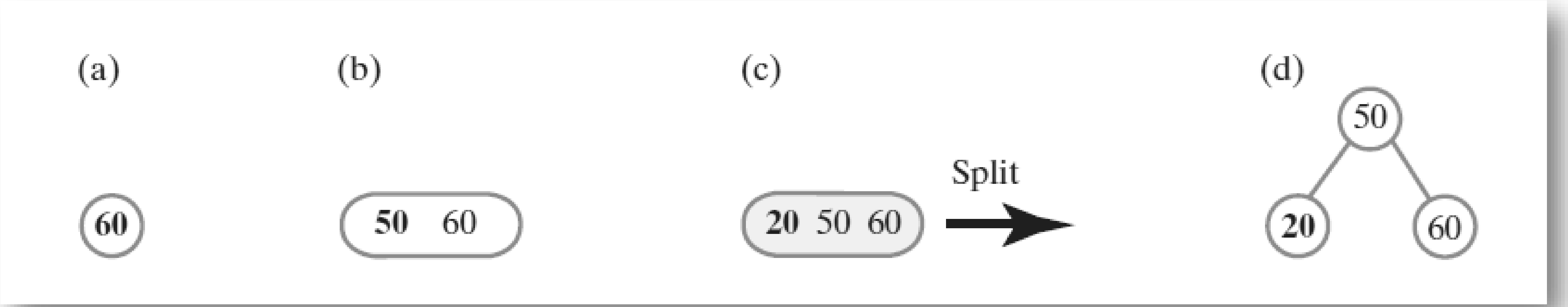


```
Algorithm searchA23Tree(A23Tree, desiredObject)

if(A23Tree is empty)
    return false
else if(desiredObject is in the root of A23Tree)
    return true
else if(the root of A23Tree contains two entries)
{
    if(desiredObject < smaller object in the root)
        return searchA23Tree(left subtree of A23Tree, desiredObject)
    else if(desiredObject > larger object in the root)
        return searchA23Tree(right subtree of A23Tree, desiredObject)
    else
        return searchA23Tree(middle subtree of A23Tree, desiredObject)
}
else if(desiredObject < object in the root)
    return searchA23Tree(left subtree of A23Tree, desiredObject)
else
    return searchA23Tree(right subtree of A23Tree, desiredObject)
```

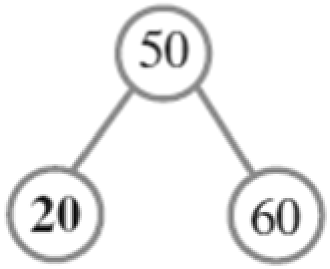


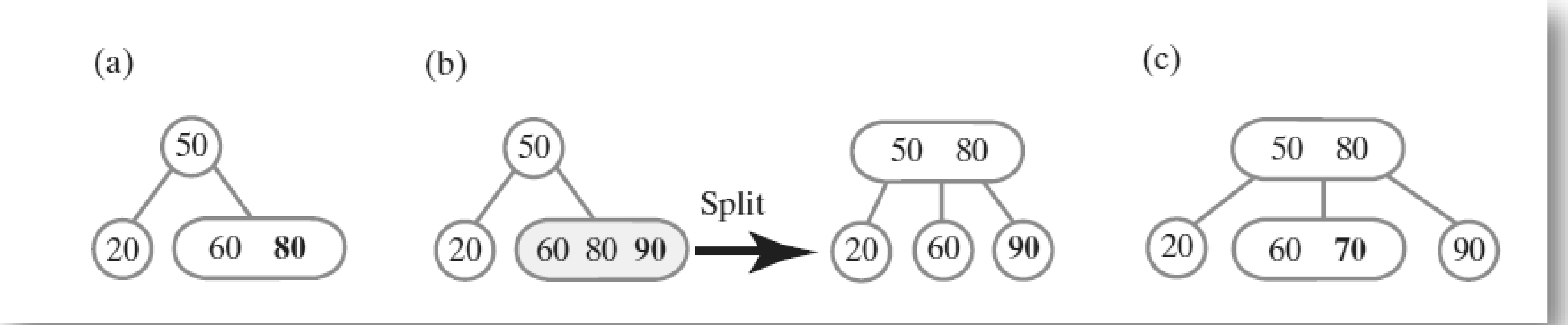
Example: Adding Entries to a 2-3 Tree

Adding Entries to a 2-3 Tree: Adding 60, 50, and 20

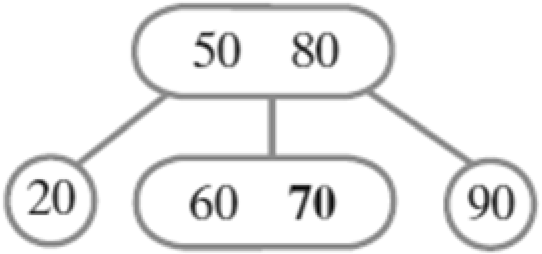


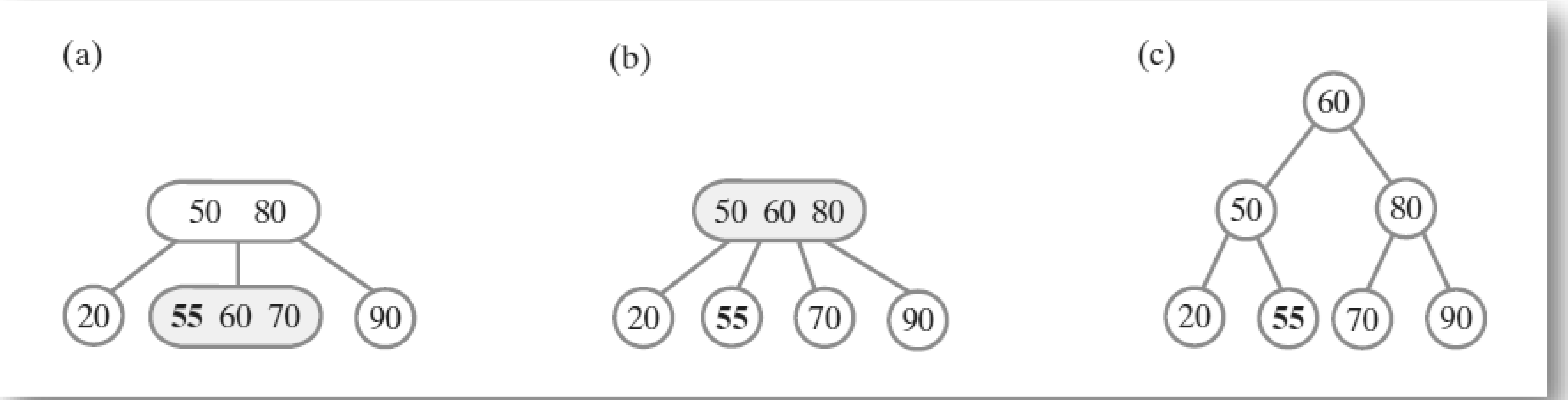
Adding Entries to a 2-3 Tree: Adding 80, 90, and 70

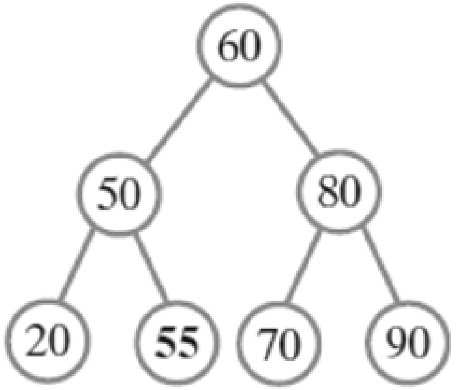




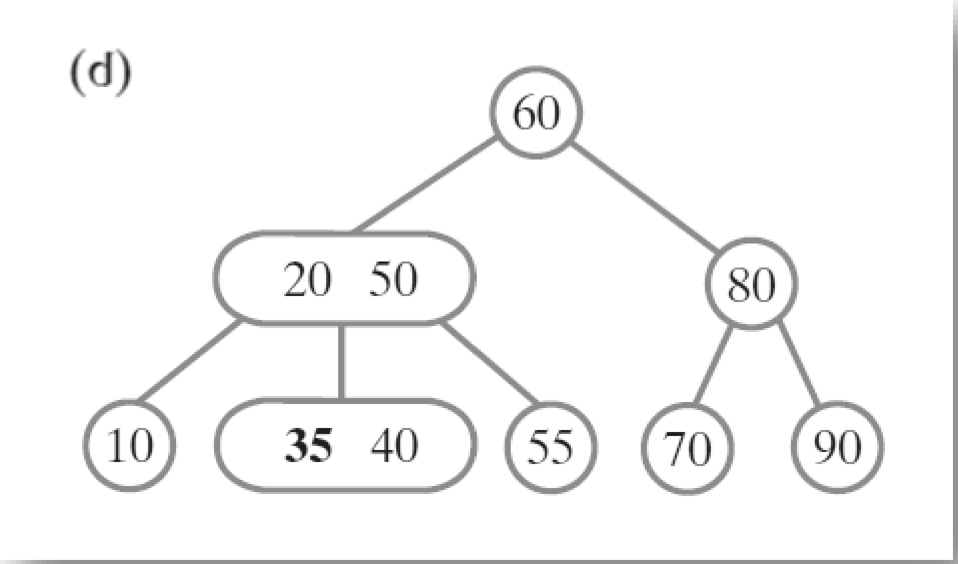
Adding Entries to a 2-3 Tree: Adding 55

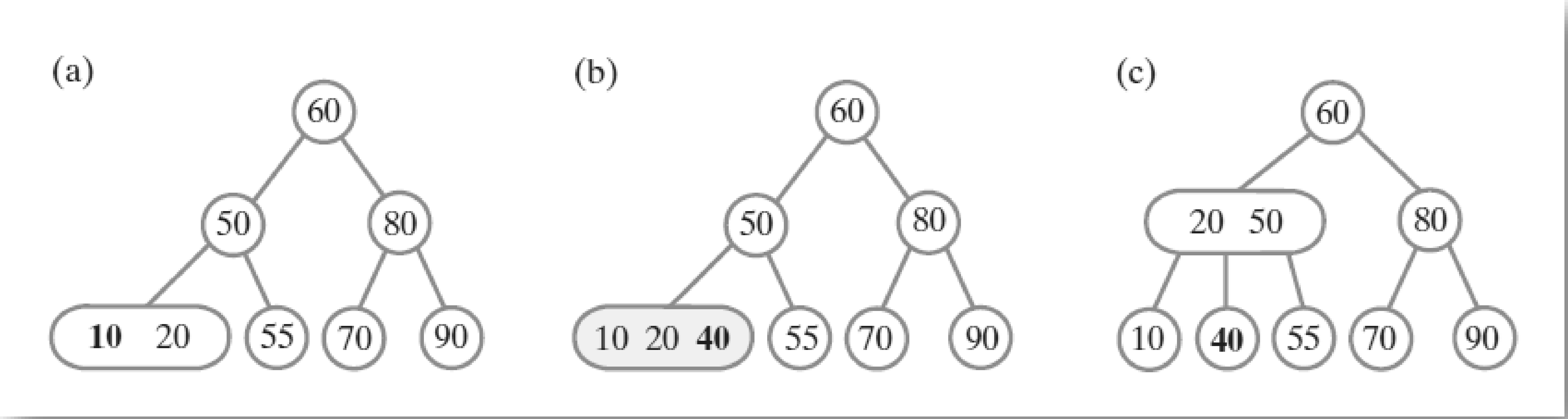






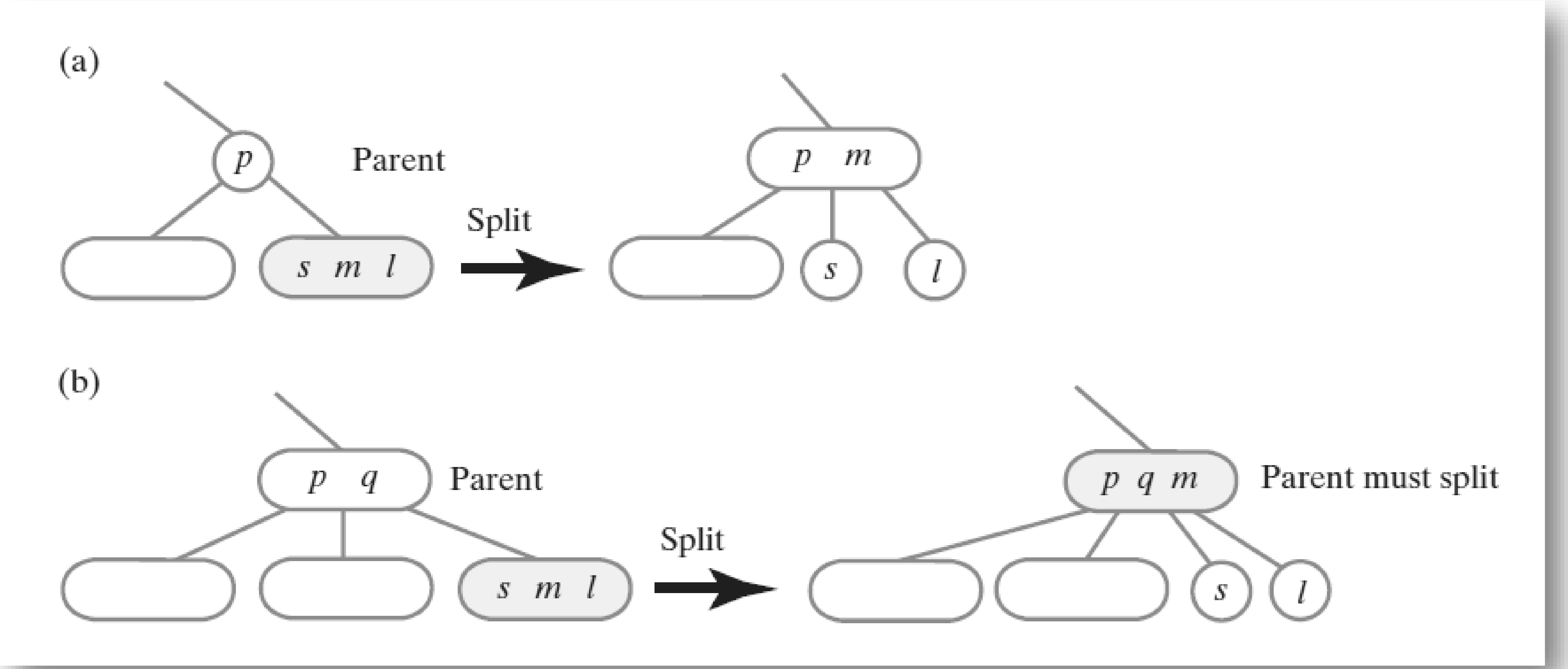
Adding Entries to a 2-3 Tree: Adding 10, 40, and 35



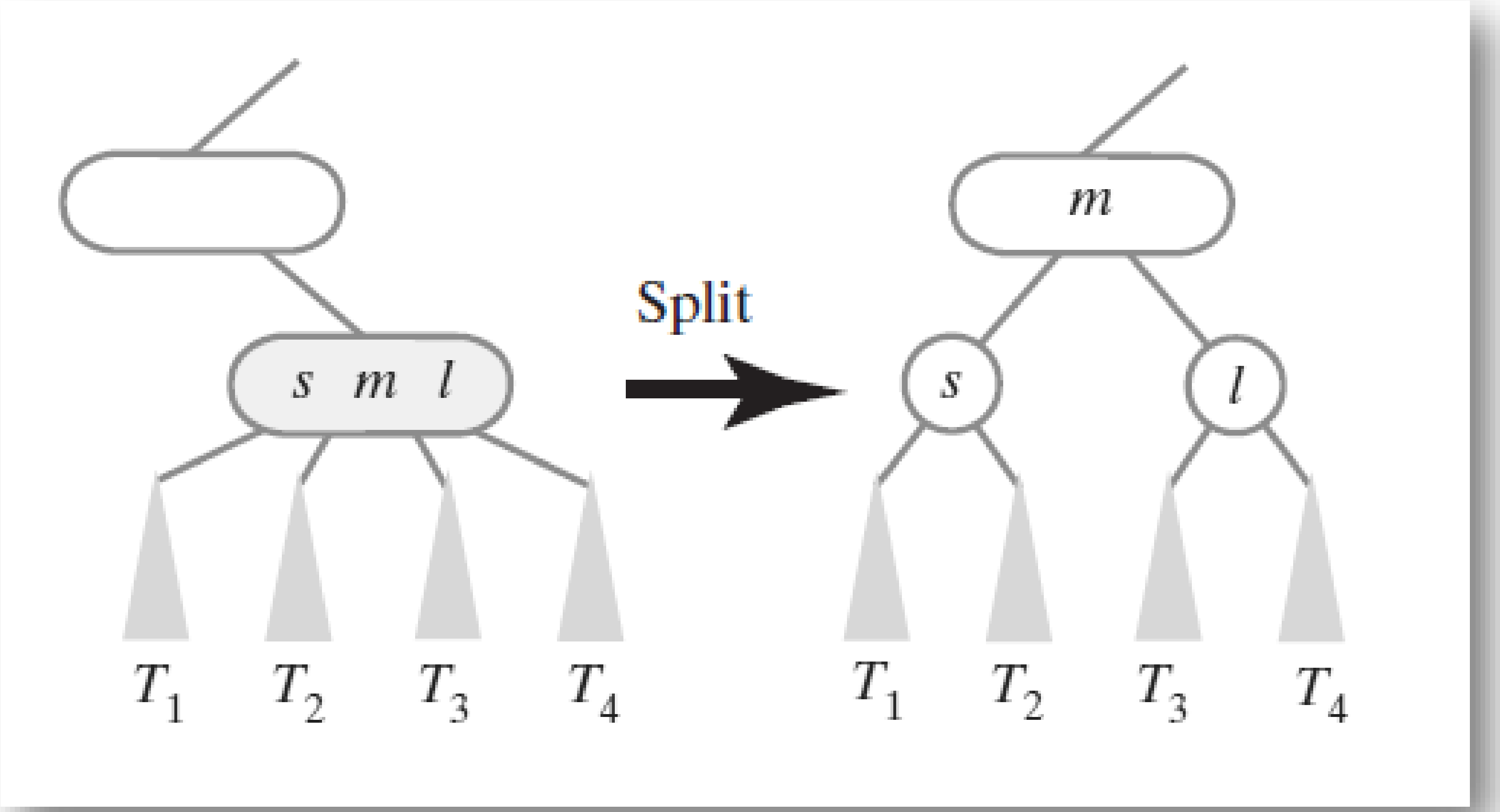


Splitting nodes in 2-3 tree during addition

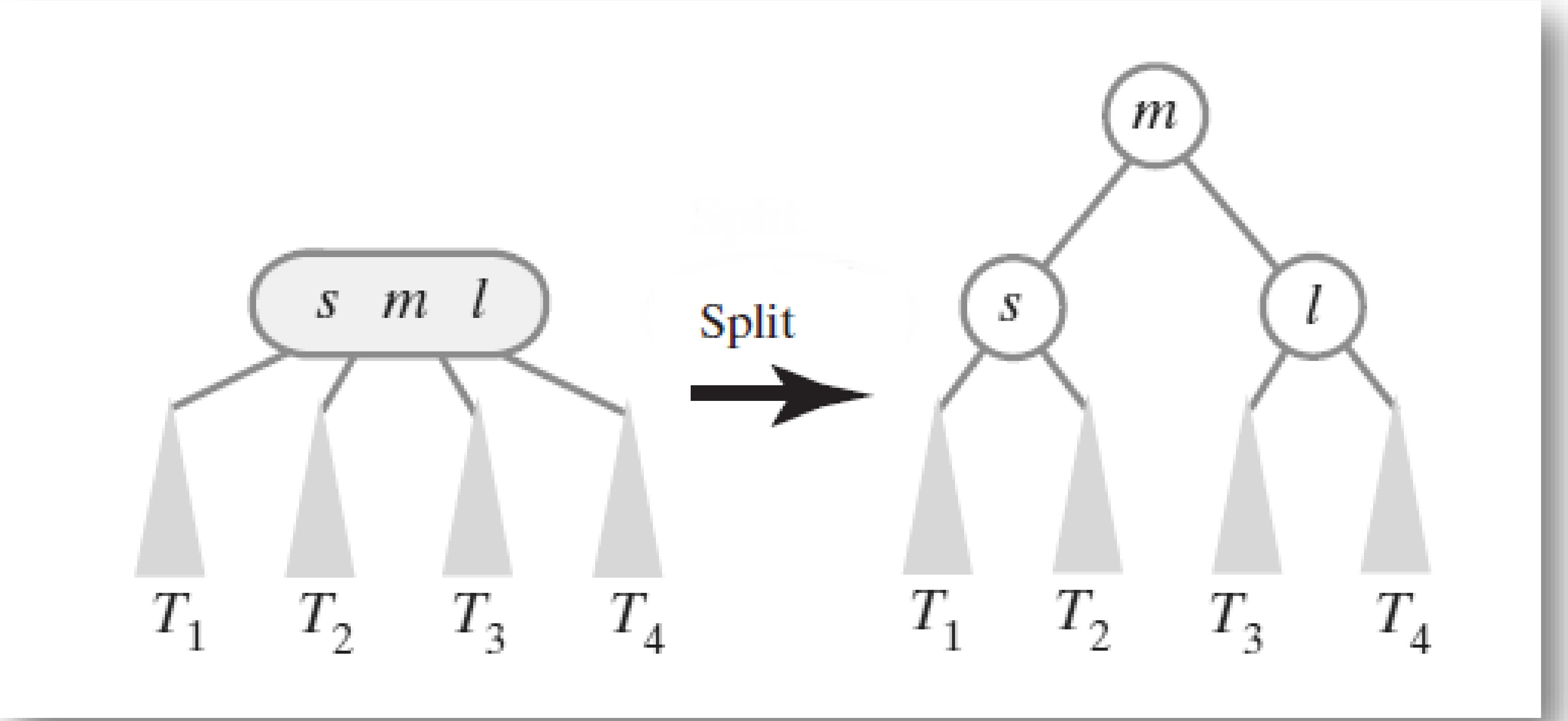
Splitting Nodes During Addition: Splitting a Leaf



Splitting Nodes During Addition: Splitting an internal node



Splitting Nodes During Addition: Splitting the root



Question

- What 2-3 tree results when you make the following additions to an initially empty 2-3 tree?

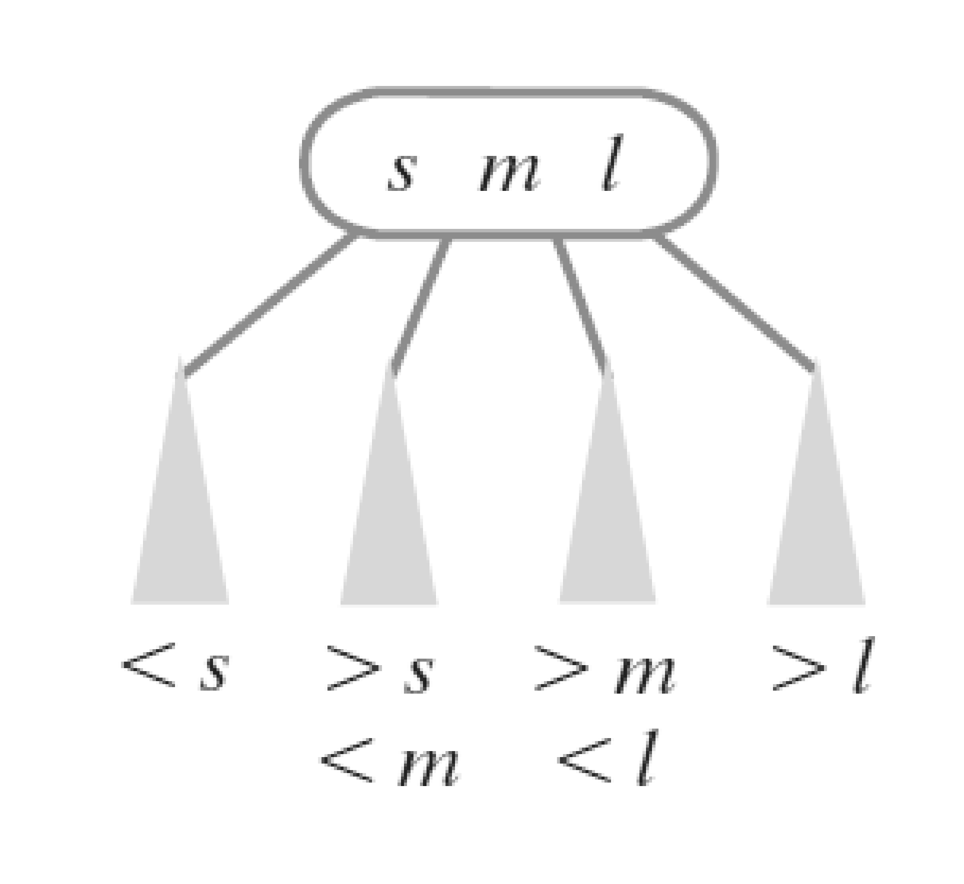
7, 8, 9, 2, 1, 5, 6, 4, 3

2-4 Trees

2-4 Trees

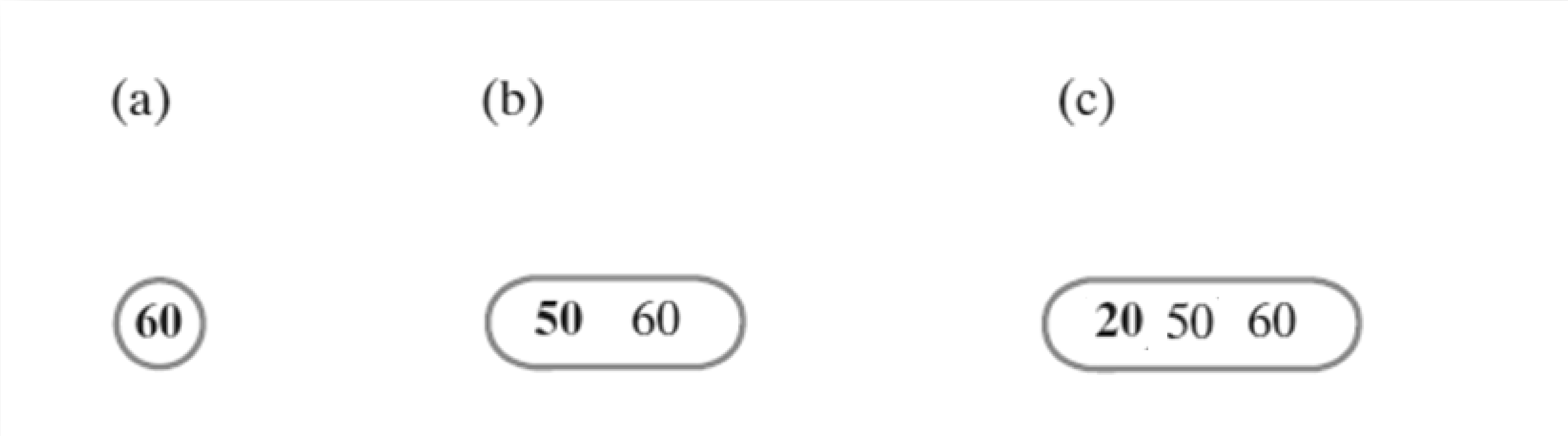
- A **2-4 tree**, sometimes called **2-3-4 tree**, is a general search tree whose interior nodes must have either two, three, or four children and whose leaves occur on the same level.
- In addition to 2-nodes and 3-nodes, as were described for 2-3 trees, this tree also contains 4-nodes.
- A **4-node** contains three data items s, m, and l and has four children.
    - Data that is less than the smallest data item s occurs in the node's left subtree.
    - Data that is greater than the largest data item l occurs in the node's right subtree.
    - Data that is between s and the middle data item m or between m and l occurs in the node's middle subtrees.

A 4-node

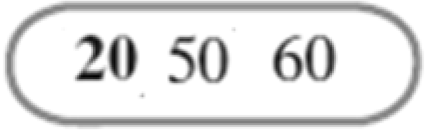


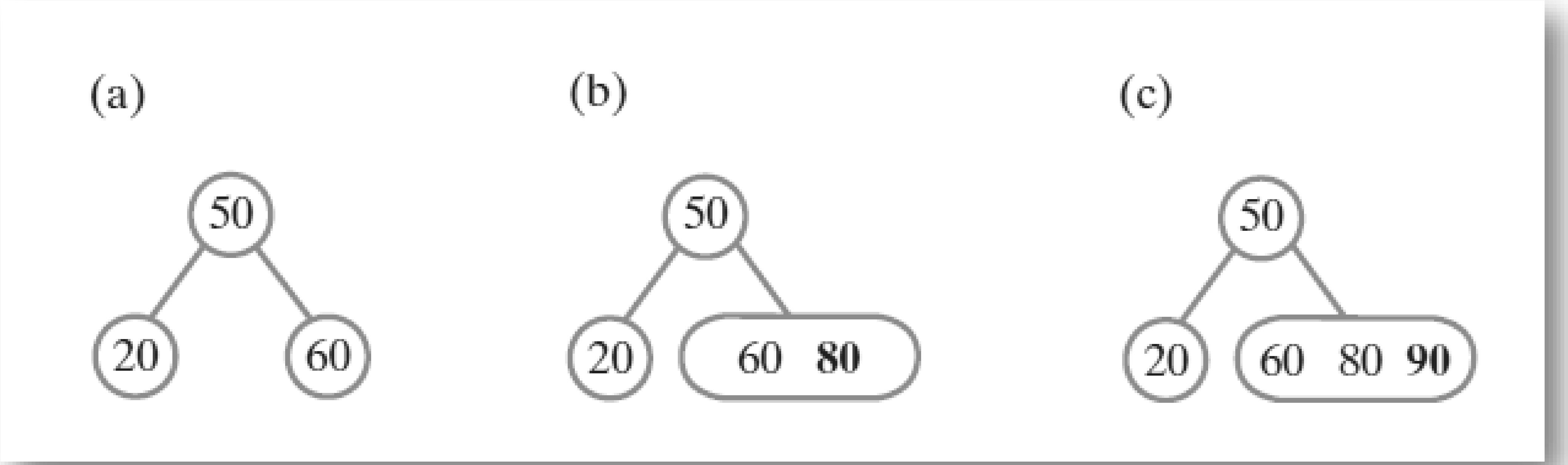
Example: Adding entries to a 2-4 Tree

Adding Entries to a 2-4 Tree: Adding 60, 50, and 20

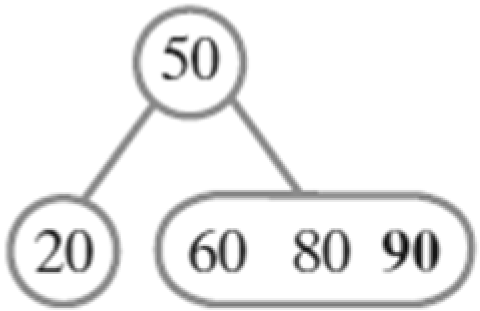


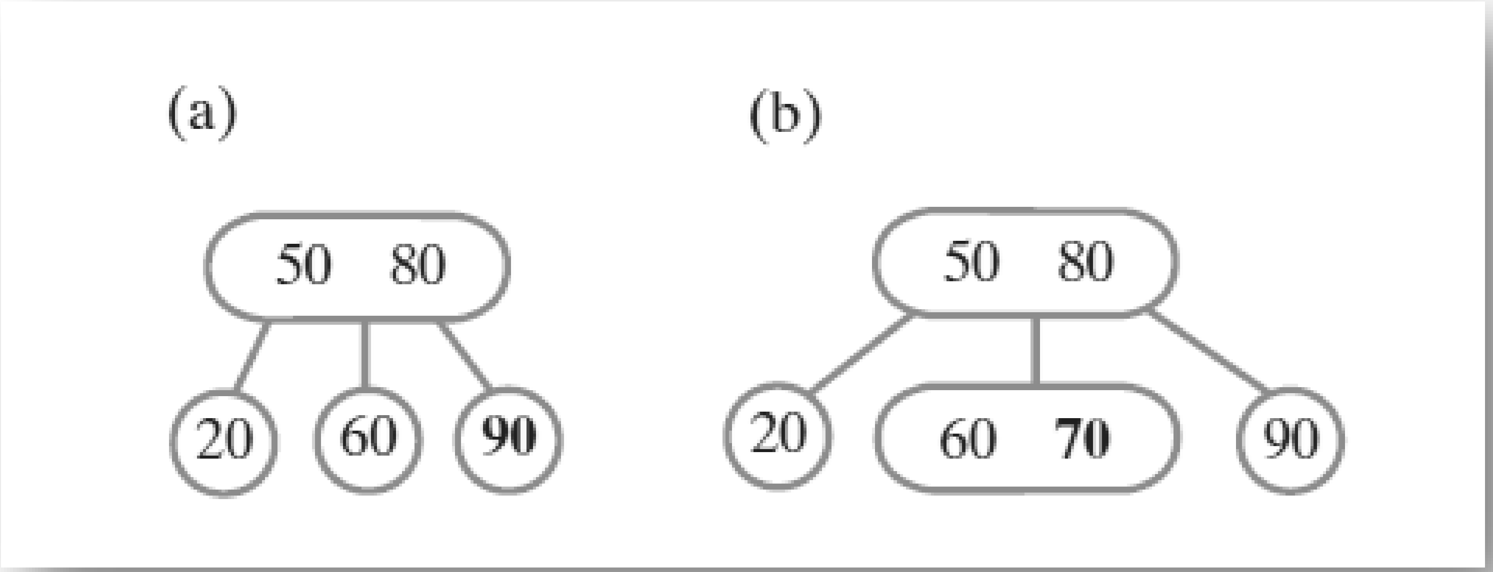
Adding Entries to a 2-4 Tree: Adding 80 and 90



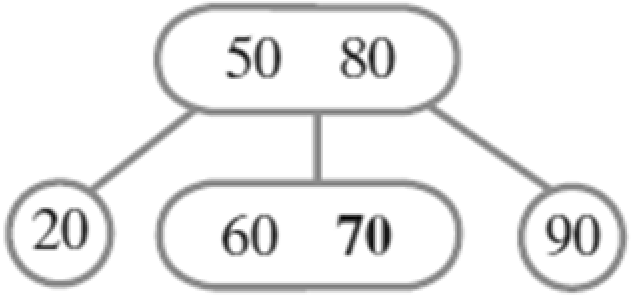


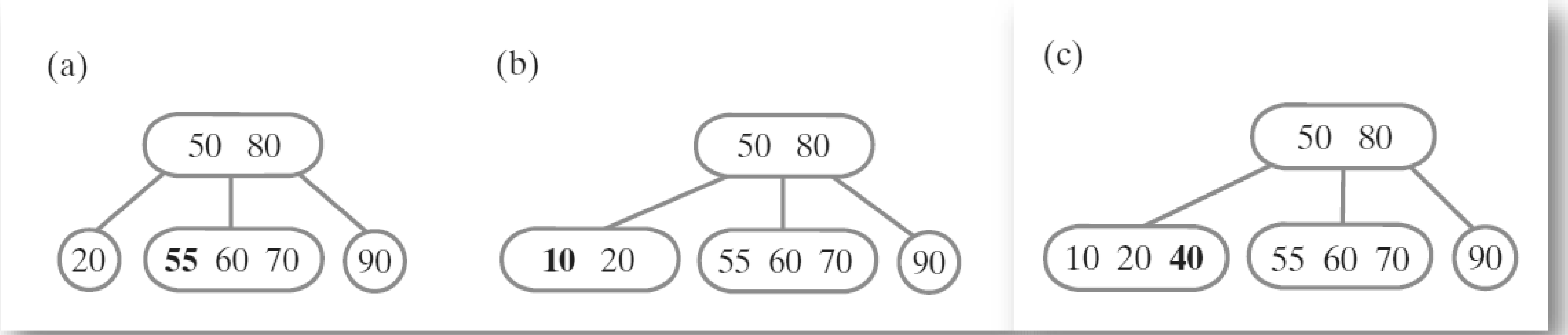
Adding Entries to a 2-4 Tree: Adding 70



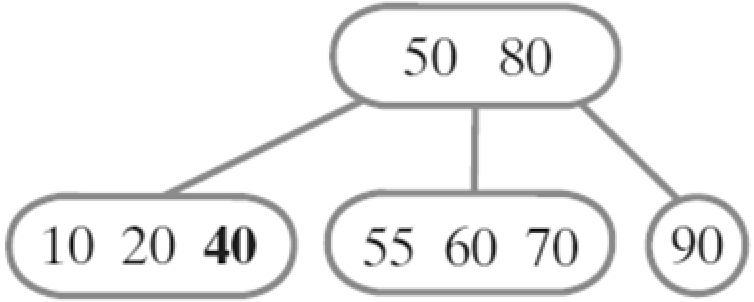


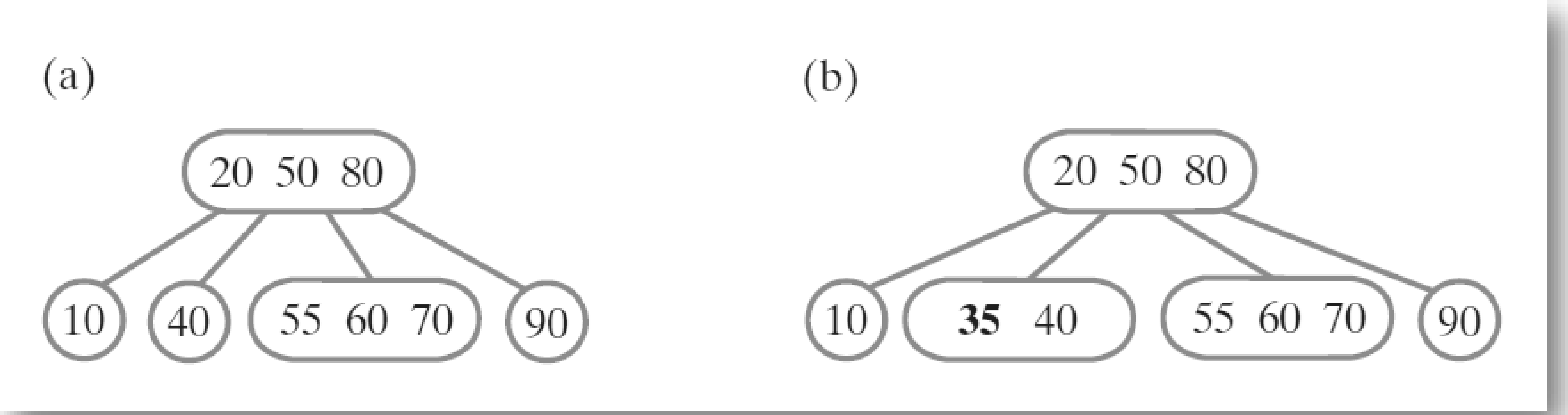
Adding Entries to a 2-4 Tree: Adding 55, 10, and 40



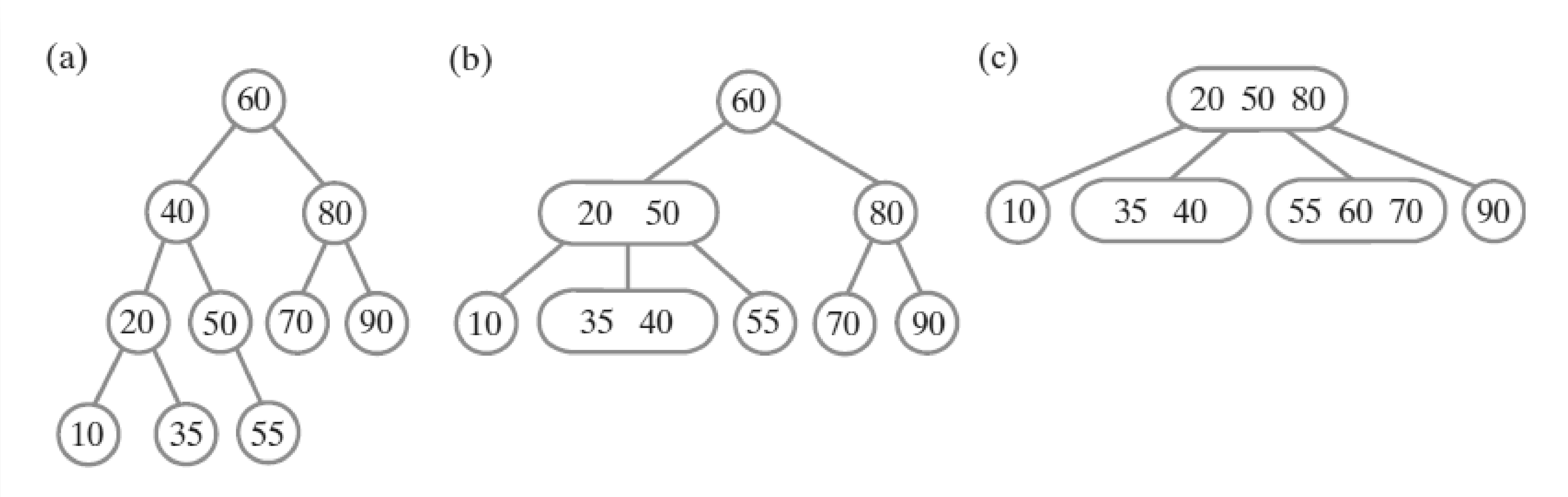


Adding Entries to a 2-4 Tree: Adding 35





Comparing AVL, 2-3, and 2-4 Trees



Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35: (a) AVL tree; (b) 2-3 tree; (c) 2-4 tree.

Question

- What 2-4 tree results when you make the following additions to an initially empty 2-4 tree?

7, 8, 9, 2, 1, 5, 6, 4, 3

References

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