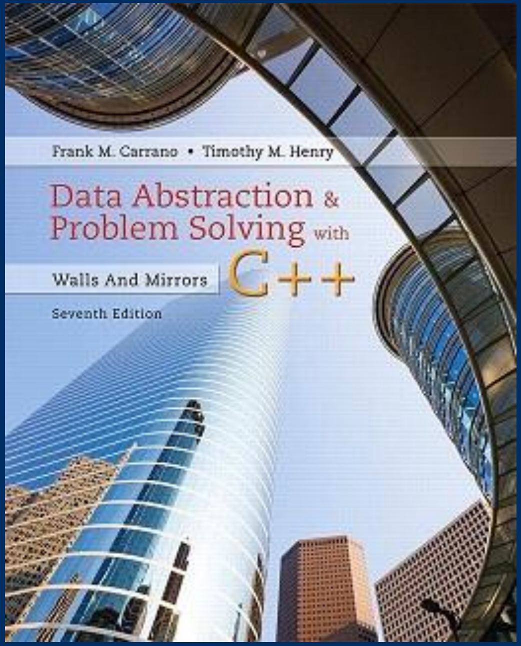
Chapter 19

Balanced Search Trees

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

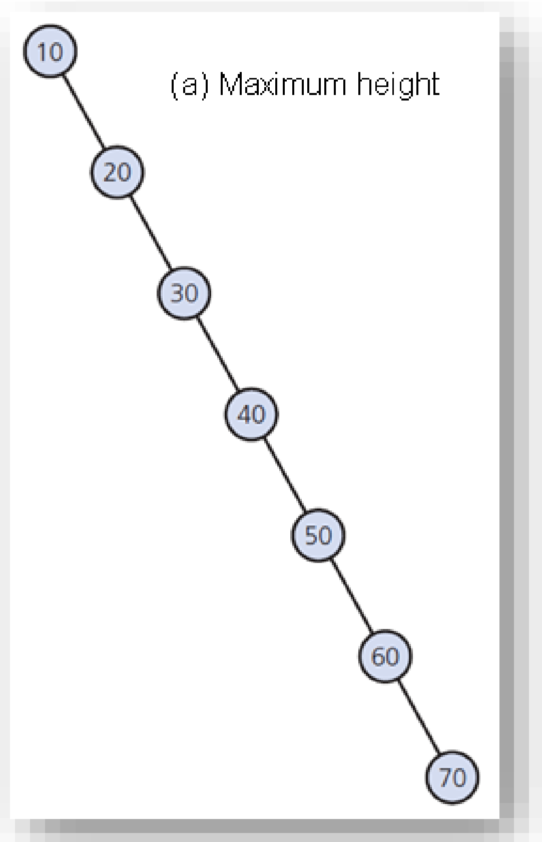
Reminders

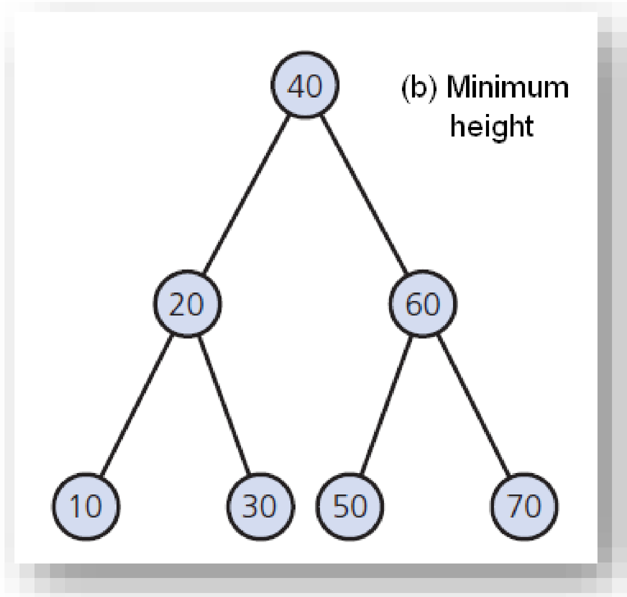
- Assignment 6 is available
  - Due Monday April 23rd at 2pm
- TA
  - Shehryar Khattak,
    Email: *shehryar [at] nevada {dot} unr {dot} edu*,
    Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116
- Quiz 10 is available
  - Today between 4pm to 11:59pm

Balanced Search Trees

- Height of binary search tree
  – Sensitive to order of additions and removals
- Various search trees can retain balance
  – Despite additions and removals

Balanced Search Trees



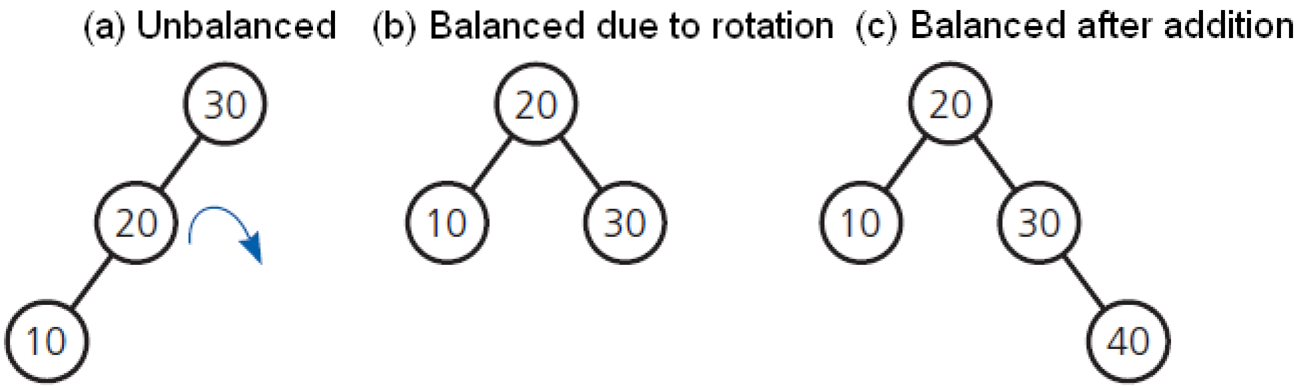


- The tallest and shortest binary search trees containing the same data

AVL Trees

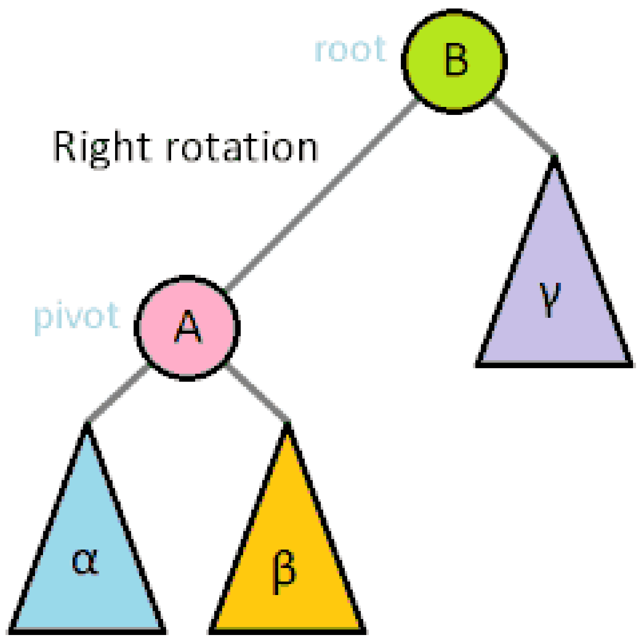
- An AVL tree
  - A balanced binary search tree
- Maintains its height close to the minimum
- Rotations restore the balance

AVL Trees

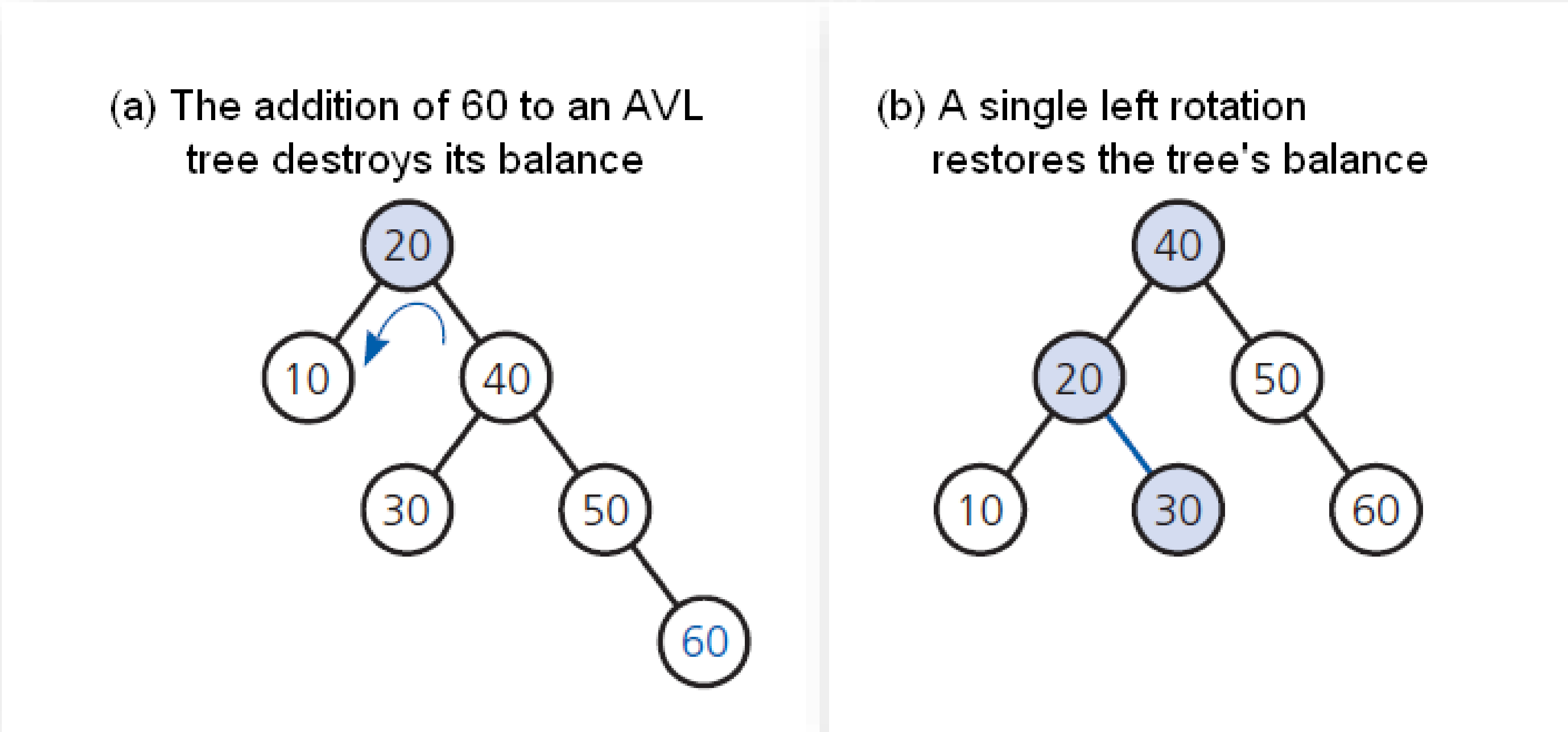


- An unbalanced binary search tree

Rotations

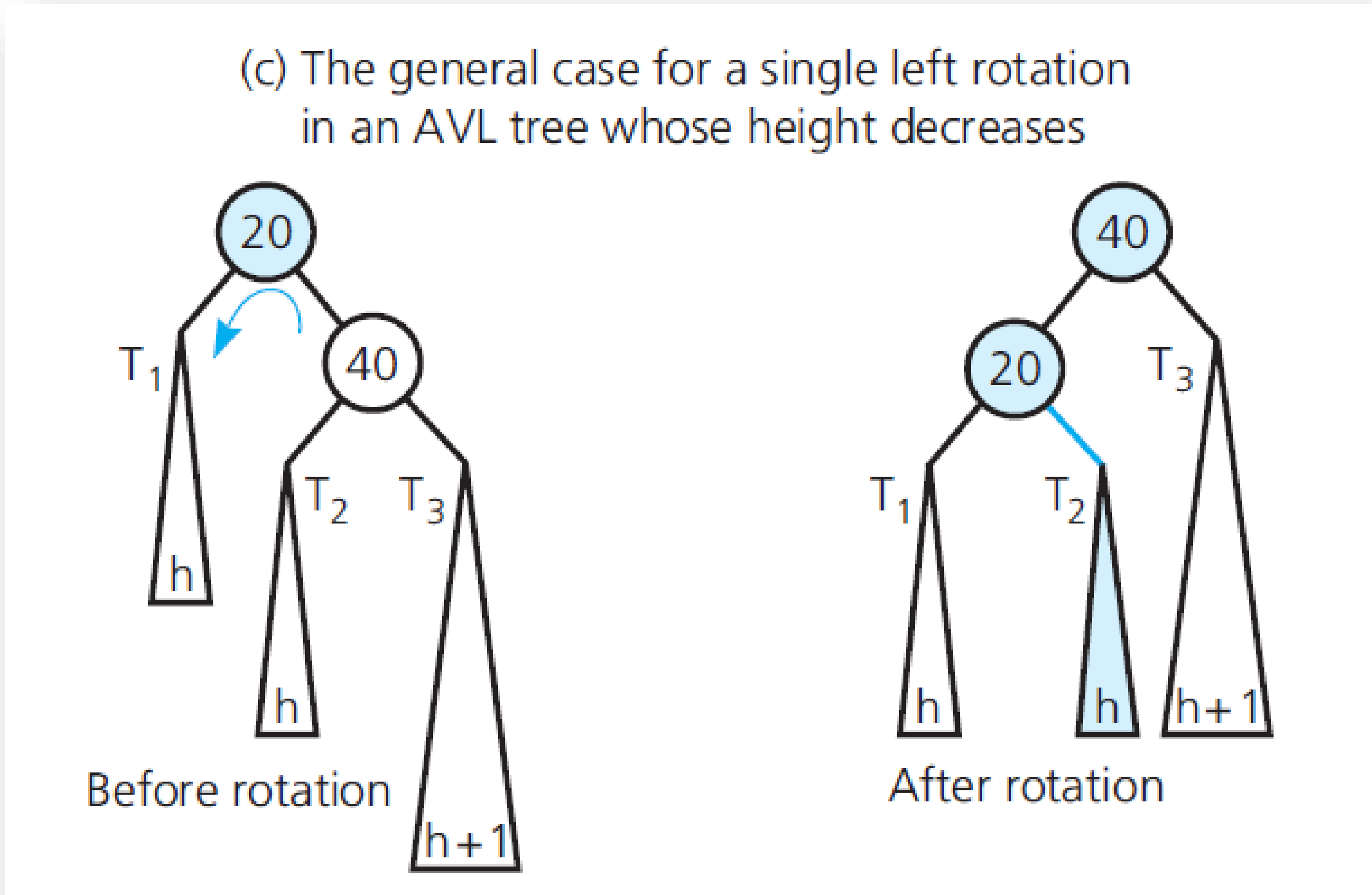


AVL Trees



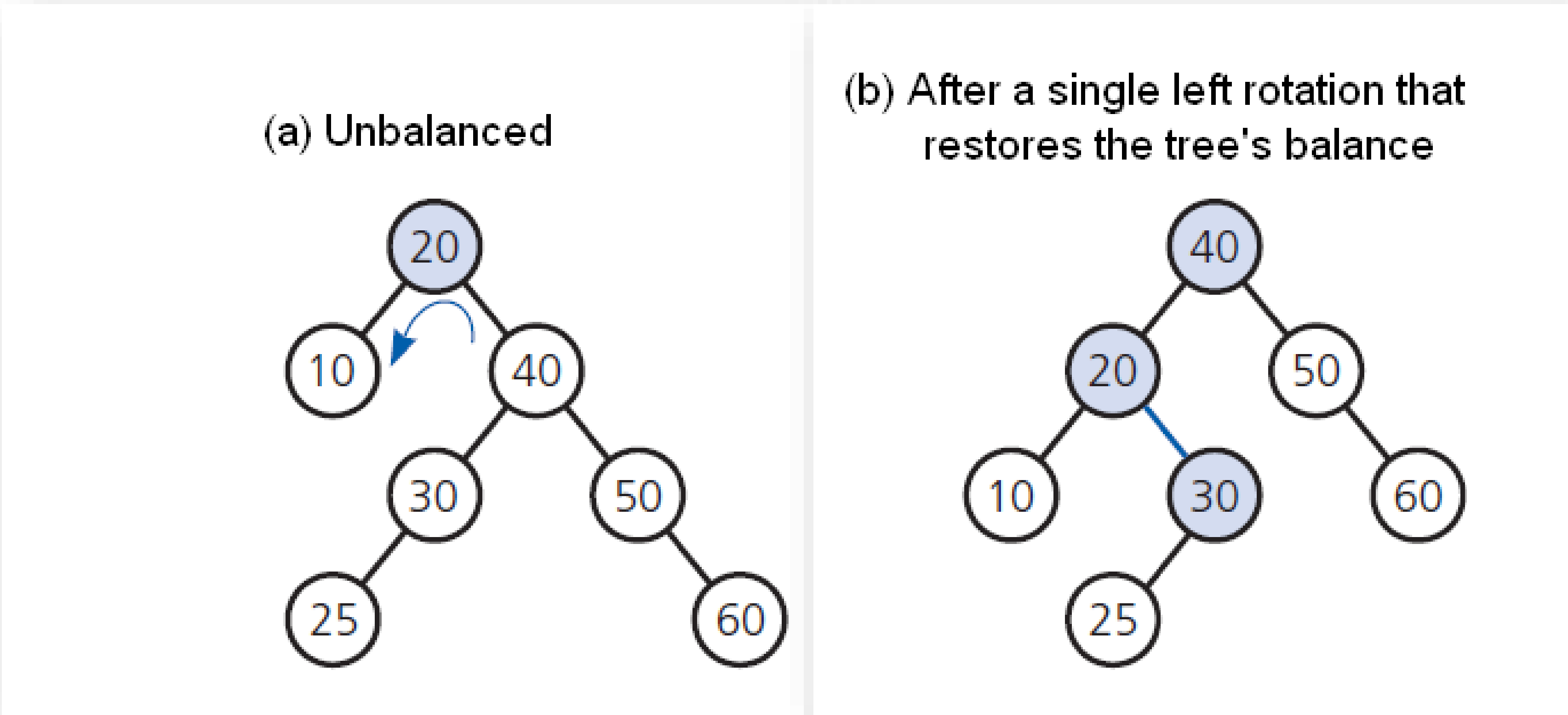
- Correcting an imbalance in an AVL tree due to an addition by using a single rotation to the left

AVL Trees



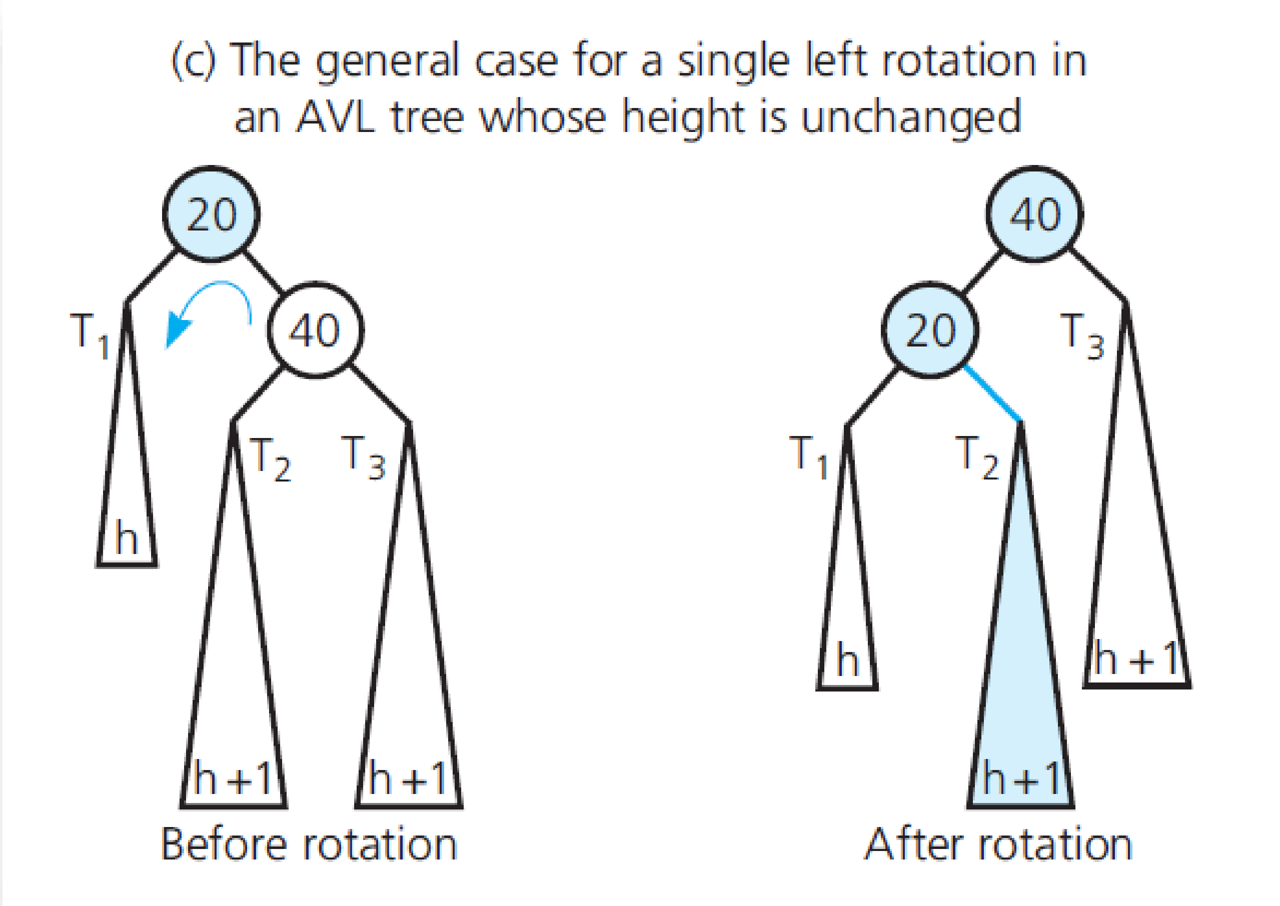
- Correcting an imbalance in an AVL tree due to an addition by using a single rotation to the left

AVL Trees



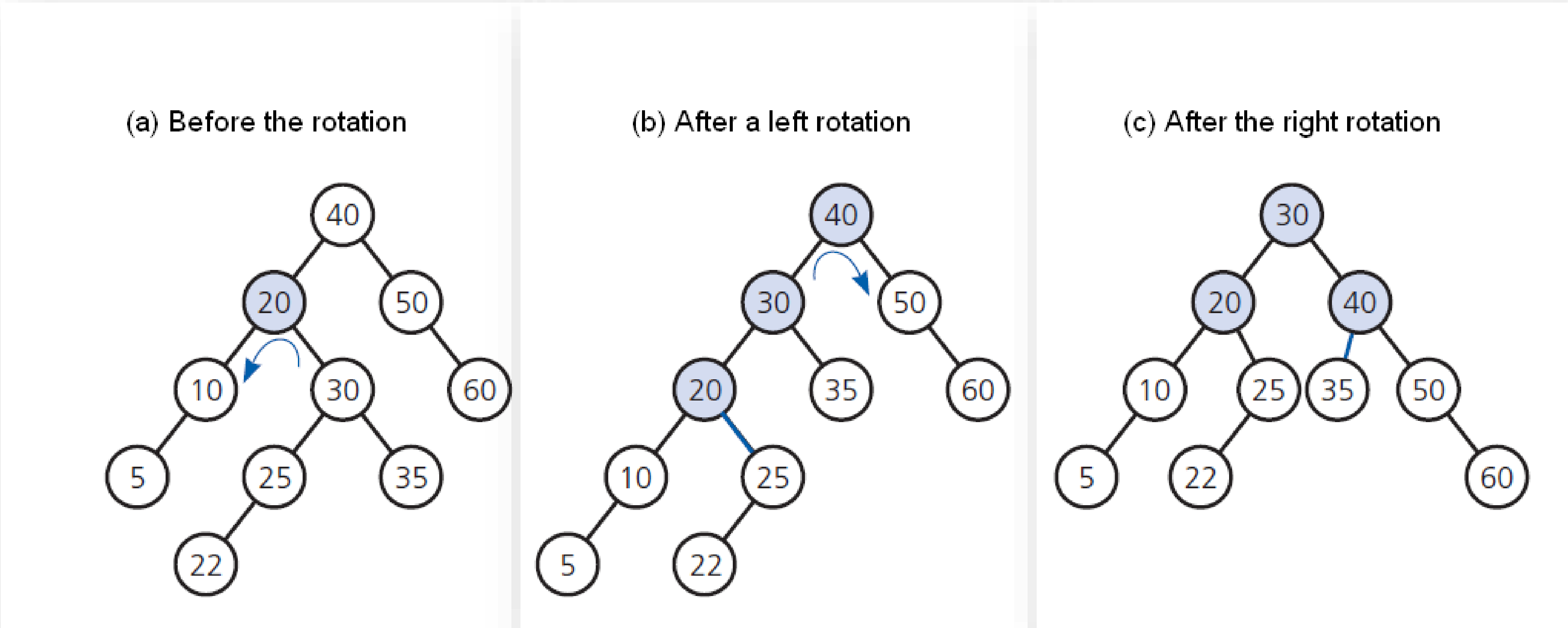
- A single rotation to the left that does not affect the height of an AVL tree

AVL Trees



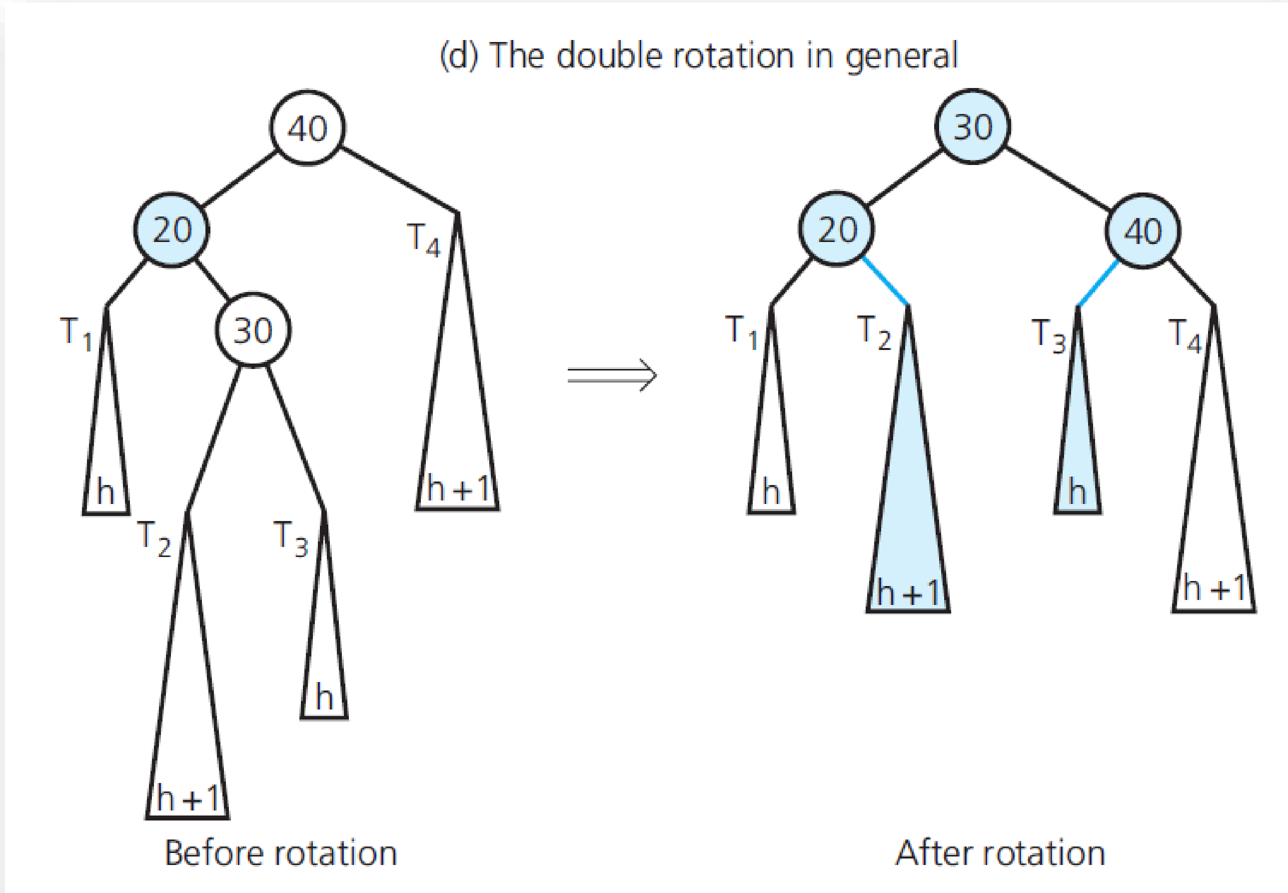
- A single rotation to the left that does not affect the height of an AVL tree

AVL Trees



- A double rotation that decreases the height of an AVL tree

AVL Trees

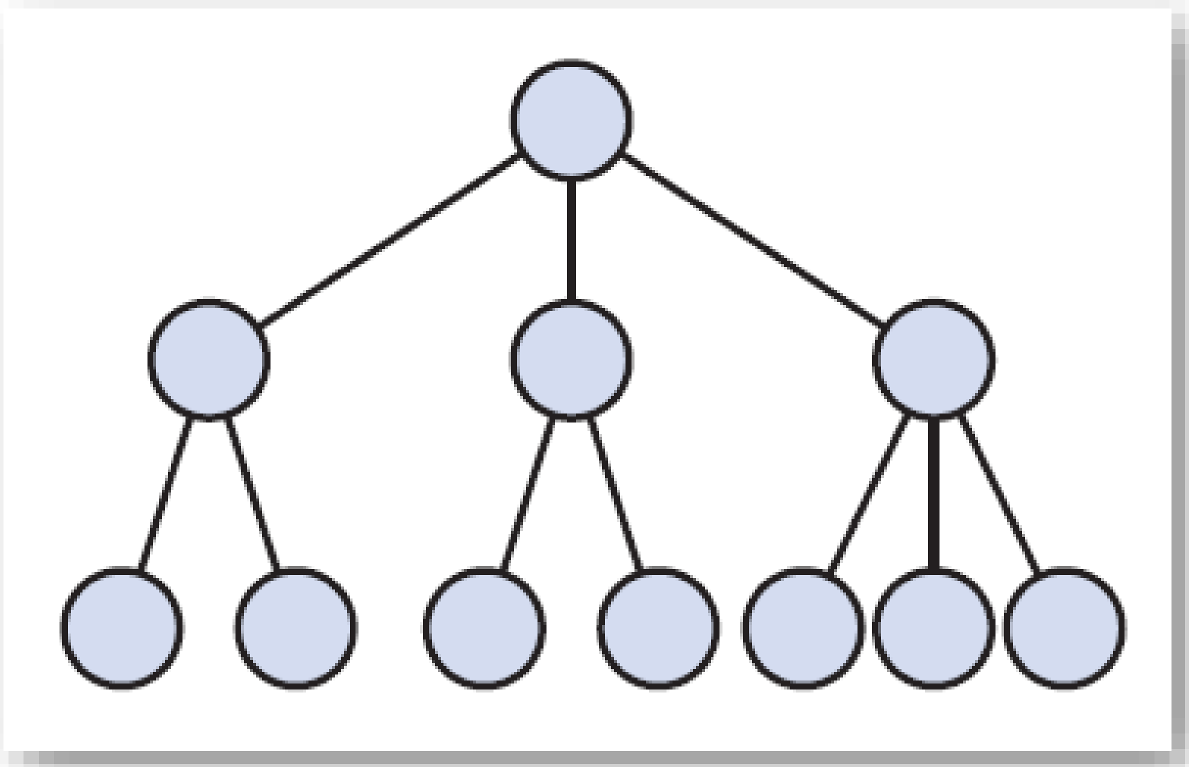


- A double rotation that decreases the height of an AVL tree

2-3 Trees

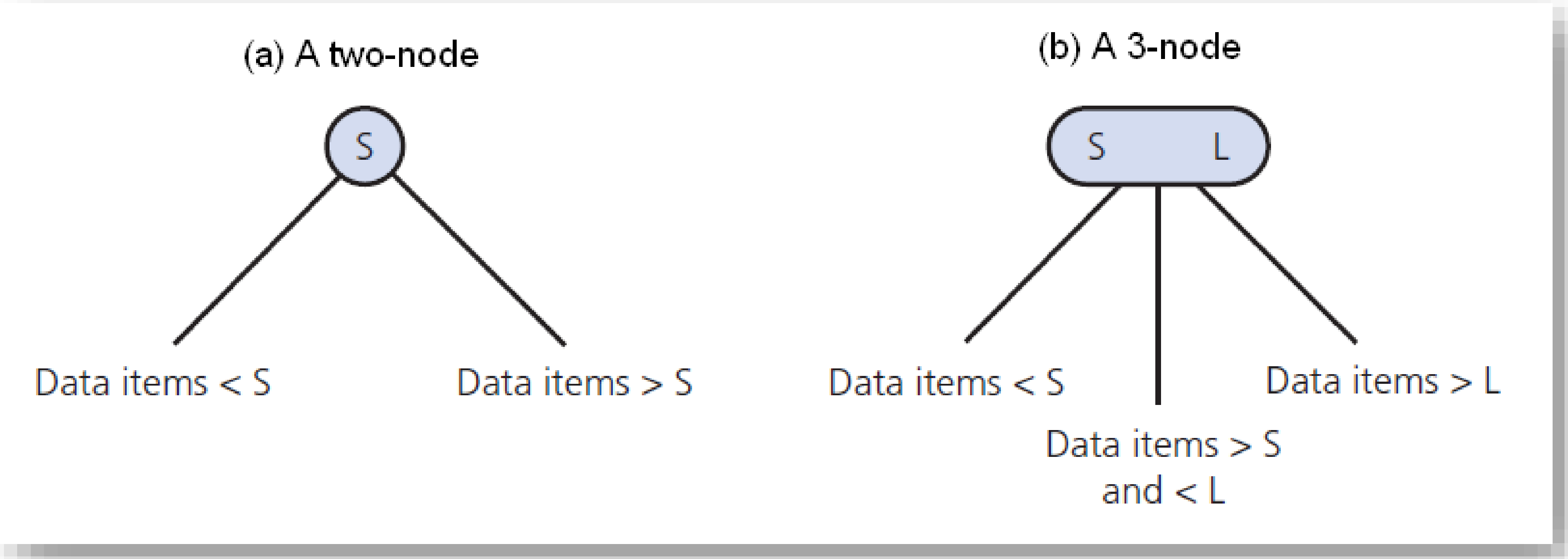
2-3 Trees

- A 2-3 tree of height 3



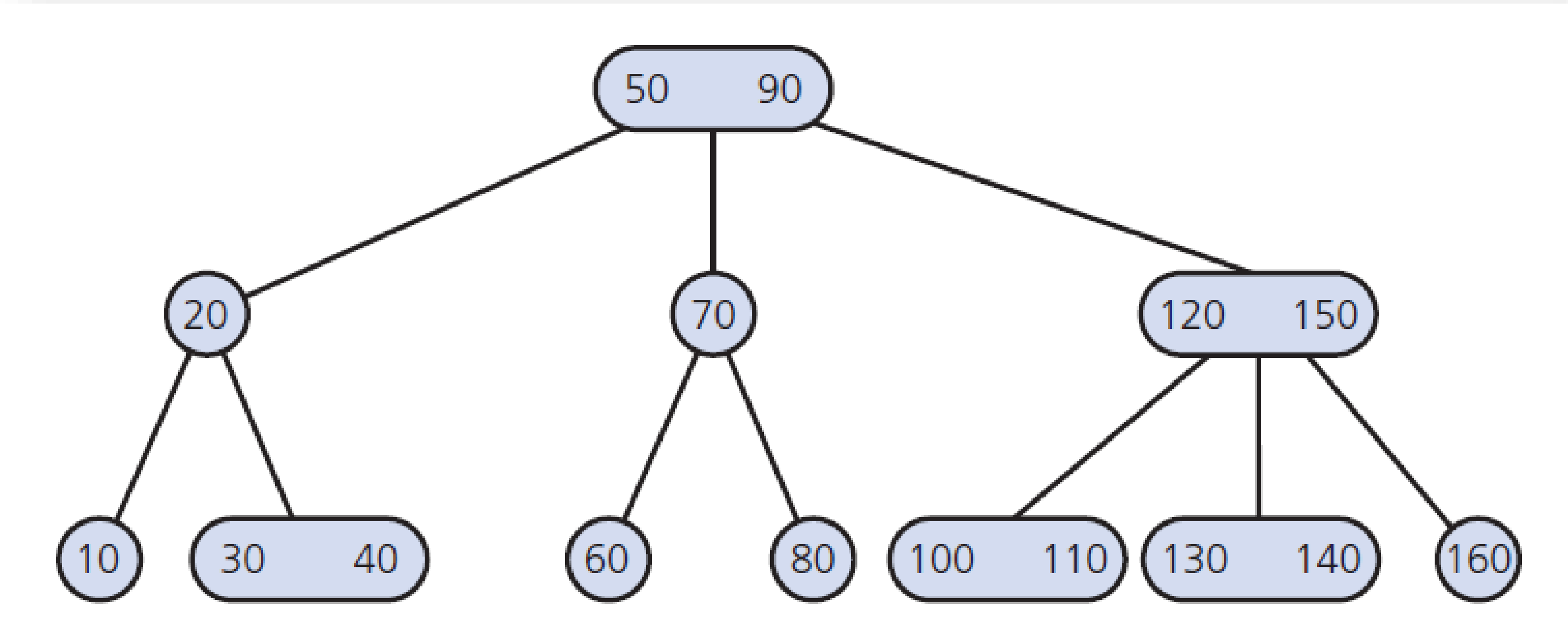
2-3 Trees

- Nodes in a 2-3 tree



2-3 Trees

- A 2-3 tree



2-3 Trees

- A header file for a class of nodes for a 2-3 tree

```
/** A class of nodes for a link-based 2-3 tree.
 @file TriNode.h */

#ifndef TRI_NODE_
#define TRI_NODE_

template<class ItemType>
class TriNode
{
private:
   ItemType smallItem;                                  // Data portion
   ItemType largeItem;                                  // Data portion
   std::shared_ptr<TriNode<ItemType>> leftChildPtr;     // Left-child pointer
   std::shared_ptr<TriNode<ItemType>> midChildPtr;      // Middle-child pointer
   std::shared_ptr<TriNode<ItemType>> rightChildPtr;    // Right-child pointer

public:
   TriNode();
```

2-3 Trees

```

    bool isLeaf() const;
    bool isTwoNode() const;
    bool isThreeNode() const;

    ItemType getSmallItem() const;
    ItemType getLargeItem() const;

    void setSmallItem(const ItemType& anItem);
    void setLargeItem(const ItemType& anItem);
    auto getLeftChildPtr() const;
    auto getMidChildPtr() const;
    auto getRightChildPtr() const;

    void setLeftChildPtr(std::shared_ptr<TriNode<ItemType>> leftPtr);
    void setMidChildPtr(std::shared_ptr<TriNode<ItemType>> midPtr);
    void setRightChildPtr(std::shared_ptr<TriNode<ItemType>> rightPtr);
}; // end TriNode
#include “TriNode.cpp”
#endif
```

- A header file for a class of nodes for a 2-3 tree

Traversing a 2-3 Tree

```
// Traverses a nonempty 2-3 tree in sorted order.
inorder(23Tree: TwoThreeTree): void
{
   if (23Tree's root node r is a leaf)
      Visit the data item(s)
   else if (r has two data items)
   {
      inorder(left subtree of 23Tree's root)
      Visit the first data item
      inorder(middle subtree of 23Tree's root)
      Visit the second data item
      inorder(right subtree of 23Tree's root)
   }
   else // r has one data item
   {
      inorder(left subtree of 23Tree's root)
      Visit the data item
      inorder(right subtree of 23Tree's root)
   }
}
```

- Performing the analogue of an inorder traversal on a binary tree:

Searching a 2-3 Tree

```
// Locates the value target in a nonempty 2-3 tree. Returns either the located
// entry or throws an exception if such a node is not found.
findItem(23Tree: TwoThreeTree, target: ItemType): ItemType
{
   if (target is in 23Tree's root node r)
   {  // The data item has been found
      treeItem = the data portion of r
      return treeItem // Success
   }
   else if (r is a leaf)
      throw NotFoundException   // Failure

   // Else search the appropriate subtree
   else if (r has two data items)
   {
      if (target < smaller data item in r)
```

- Retrieval operation for a 2-3 tree

Searching a 2-3 Tree

```
    else if (r has two data items)
    {
        if (target < smaller data item in r)
            return findItem(r's left subtree, target)
        else if (target < larger data item in r)
            return findItem(r's middle subtree, target)
        else
            return findItem(r's right subtree, target)
    }
    else // r has one data item
    {
        if (target < r's data item)
            return findItem(r's left subtree, target)
        else
            return findItem(r's right subtree, target)
    }
}
```

- Retrieval operation for a 2-3 tree

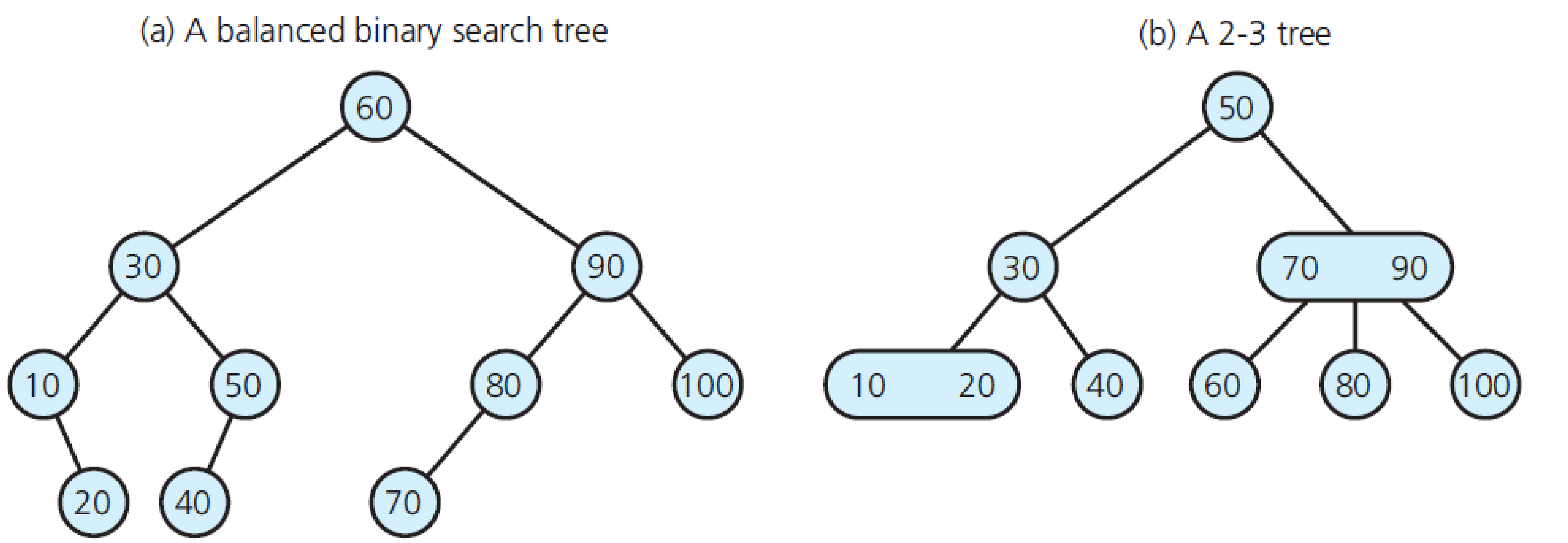
Searching a 2-3 Tree

- Search of a 2-3 and shortest binary search tree approximately same efficiency
  - A binary search tree with *n* nodes cannot be shorter than $\log_2(n + 1)$
  - A 2-3 tree with n nodes cannot be taller than $\log_2(n + 1)$
  - Node in a 2-3 tree has at most two data items

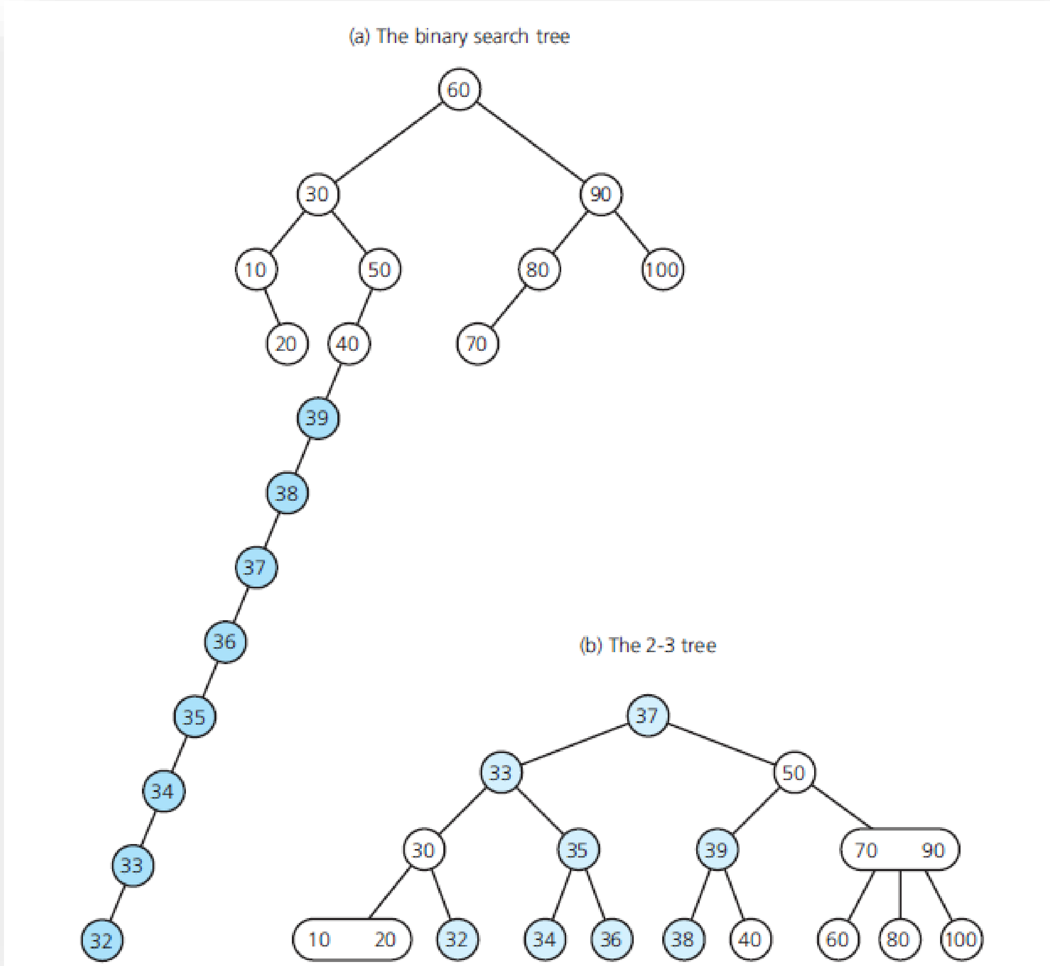
- Searching 2-3 tree is $O(\log n)$

Searching a 2-3 Tree

- A balanced binary search tree



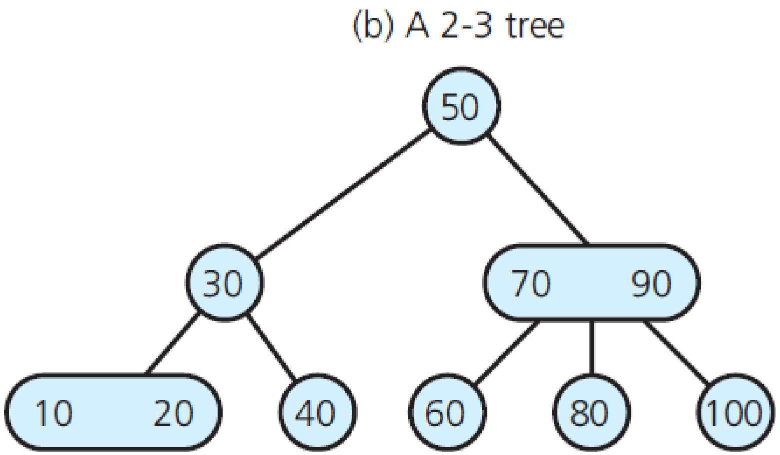
Searching a 2-3 Tree

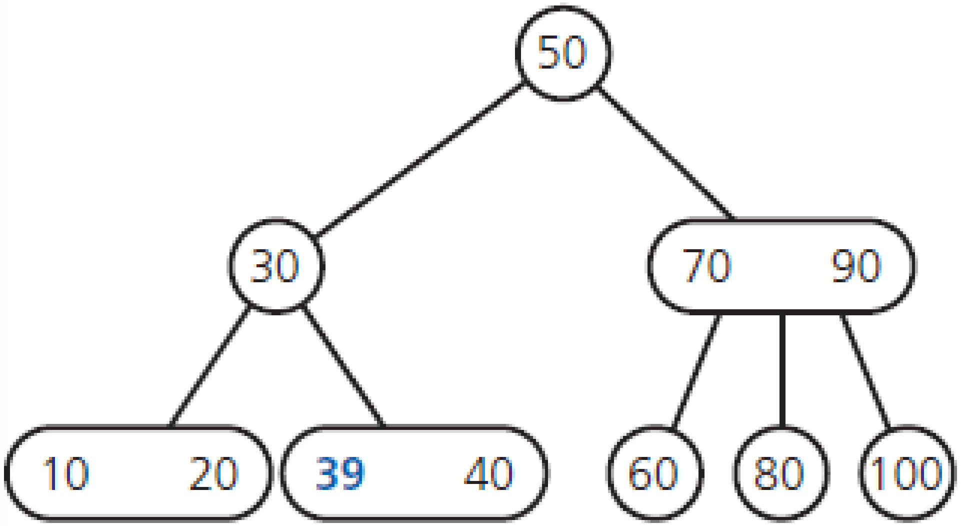


- The trees after adding the values 39 down to 32

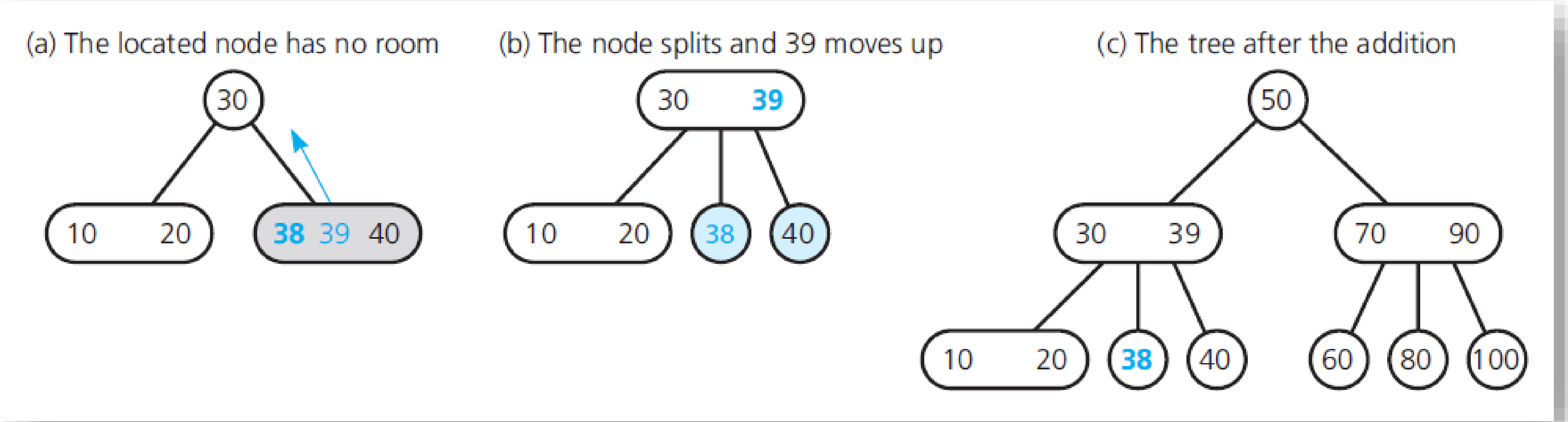
Adding Data to a 2-3 Tree

- After inserting 39 into the tree



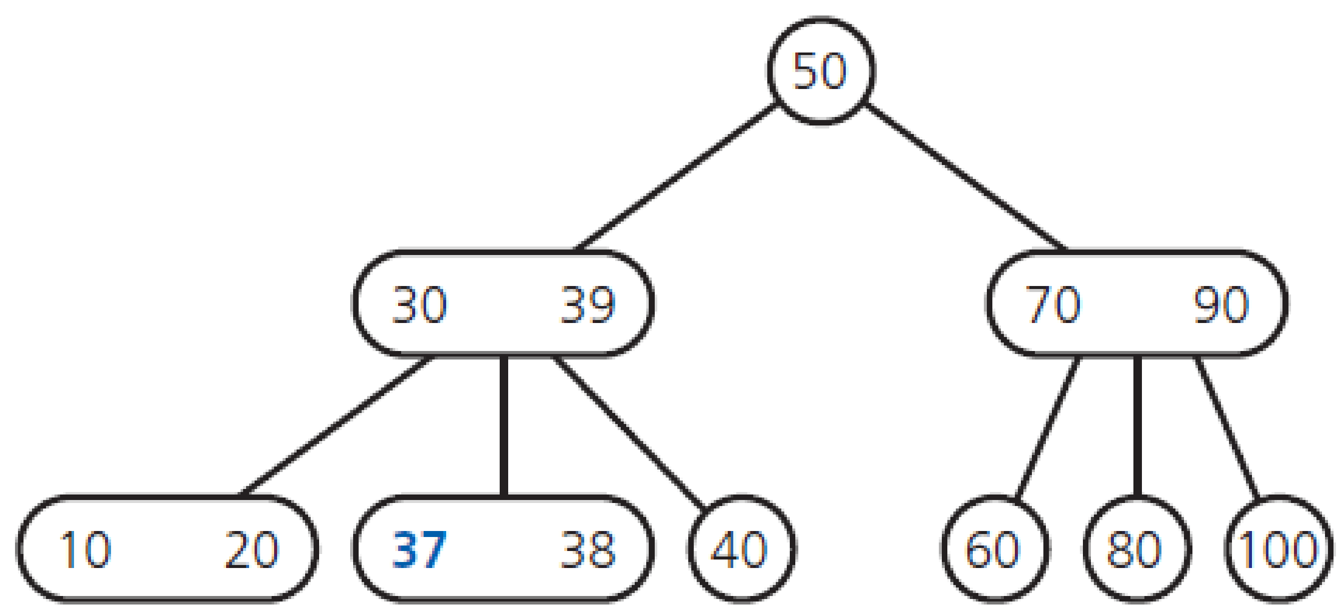


Adding Data to a 2-3 Tree



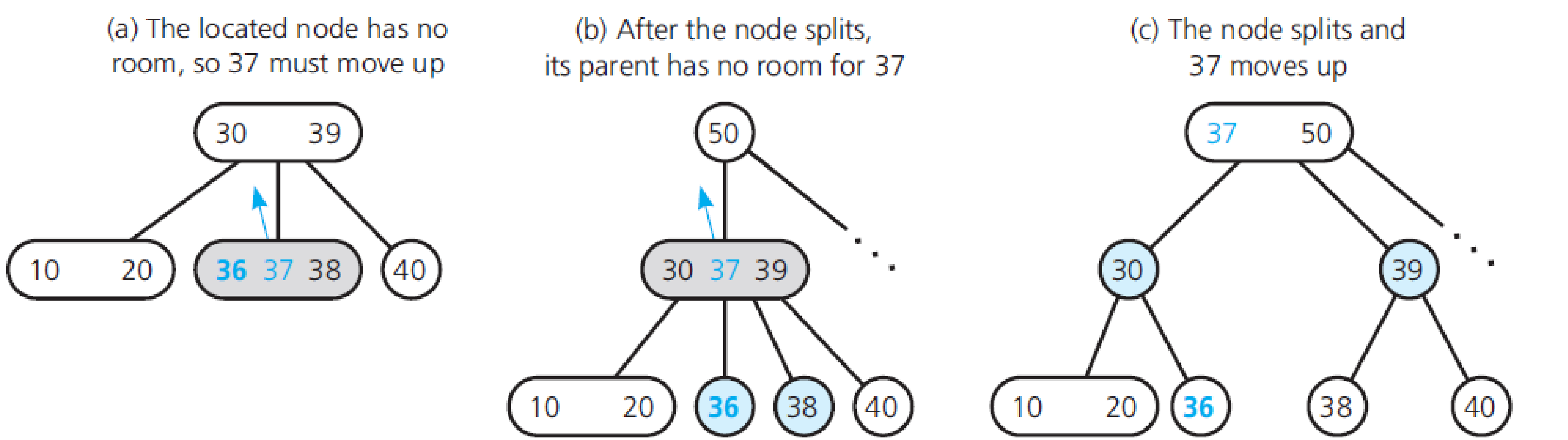
- The steps for adding 38 to the tree

Adding Data to a 2-3 Tree



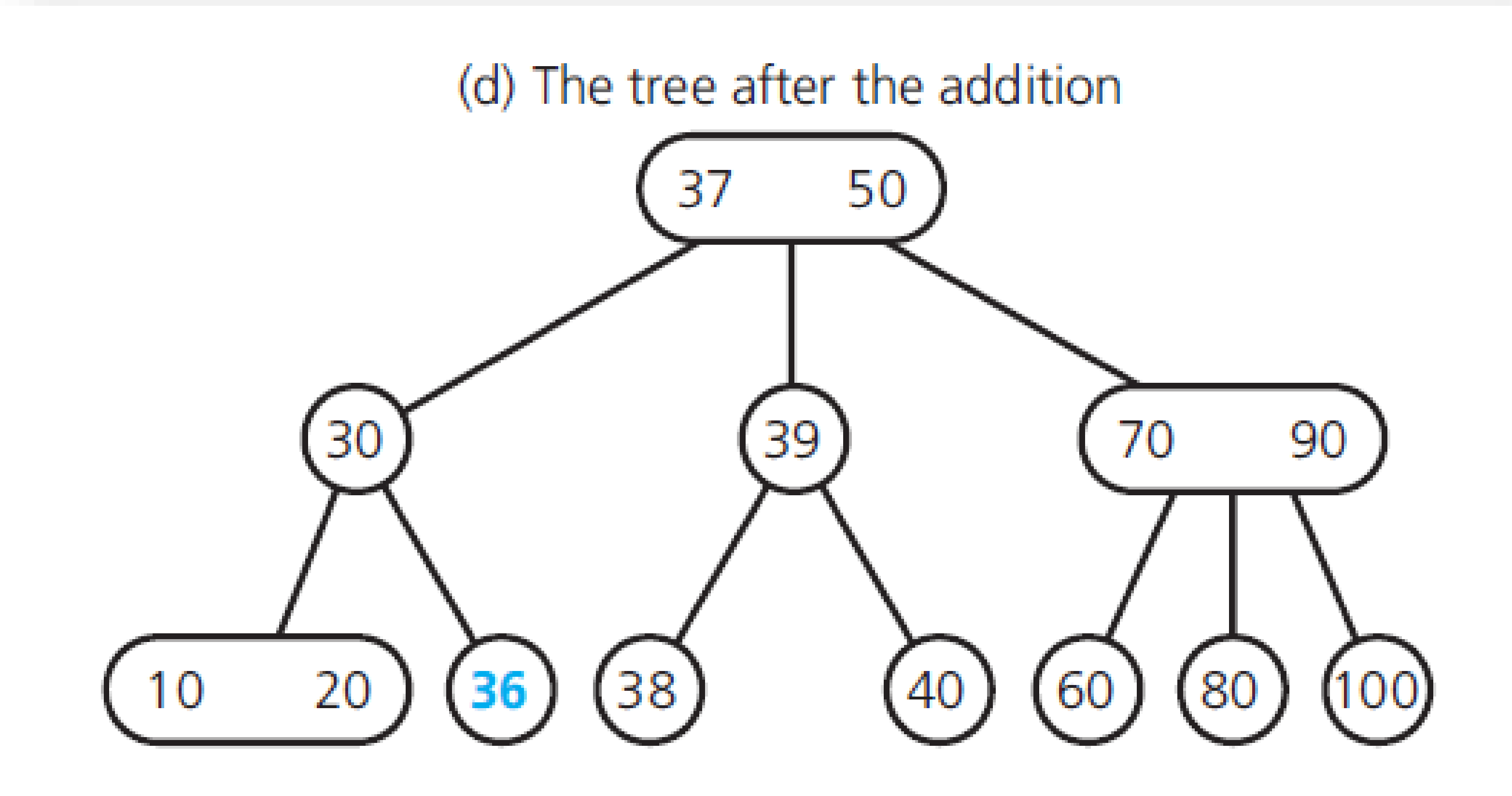
- After adding 37 to the tree

Adding Data to a 2-3 Tree



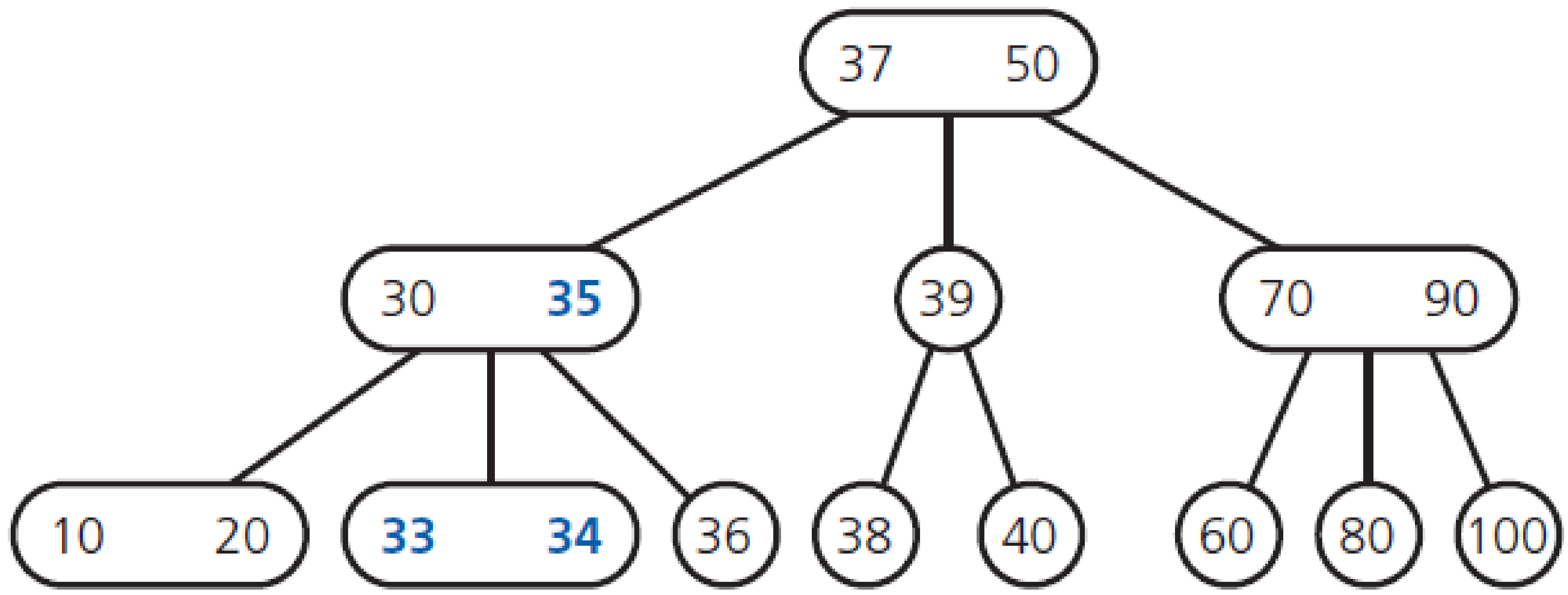
- The steps for adding 36 to the tree

Adding Data to a 2-3 Tree



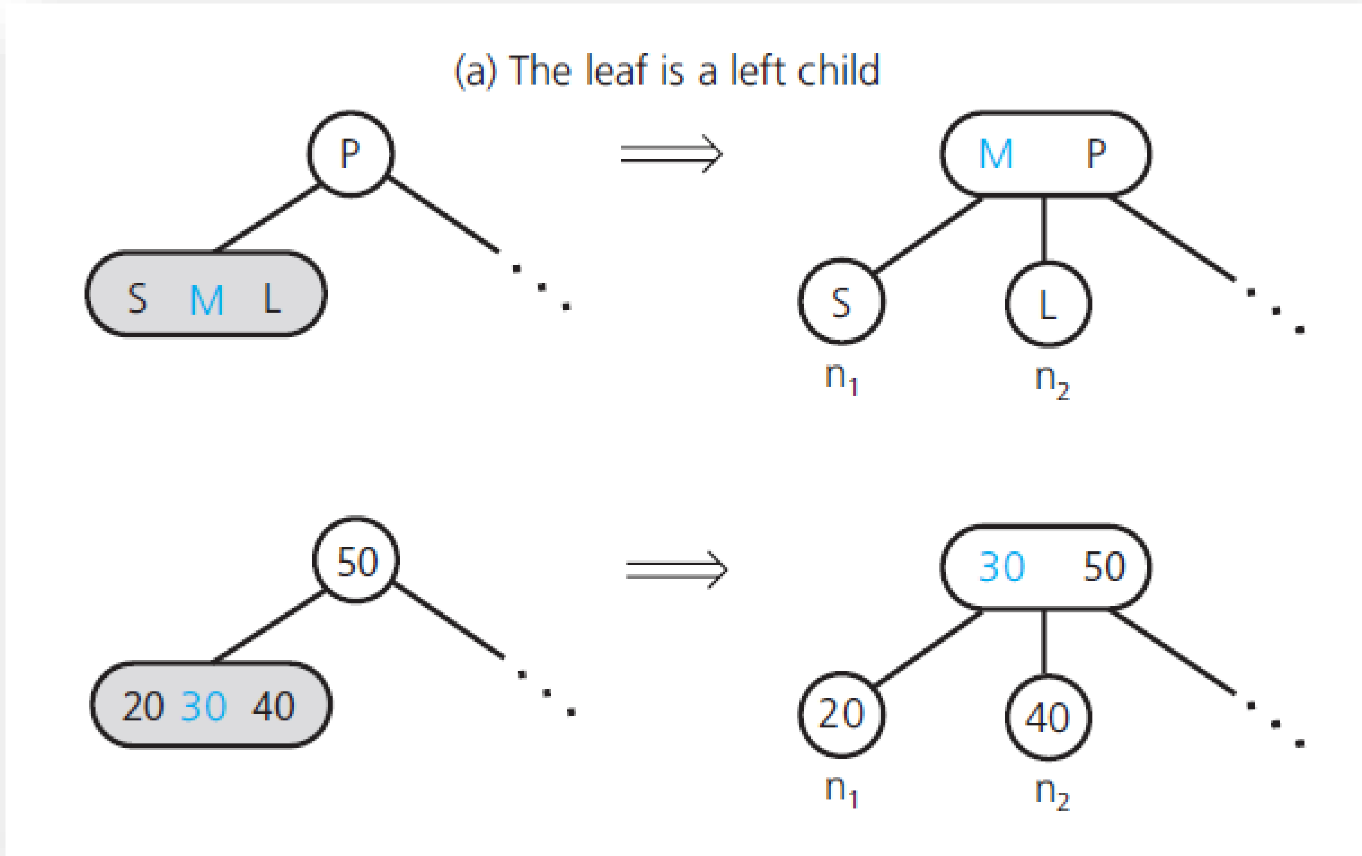
- The steps for adding 36 to the tree

Adding Data to a 2-3 Tree



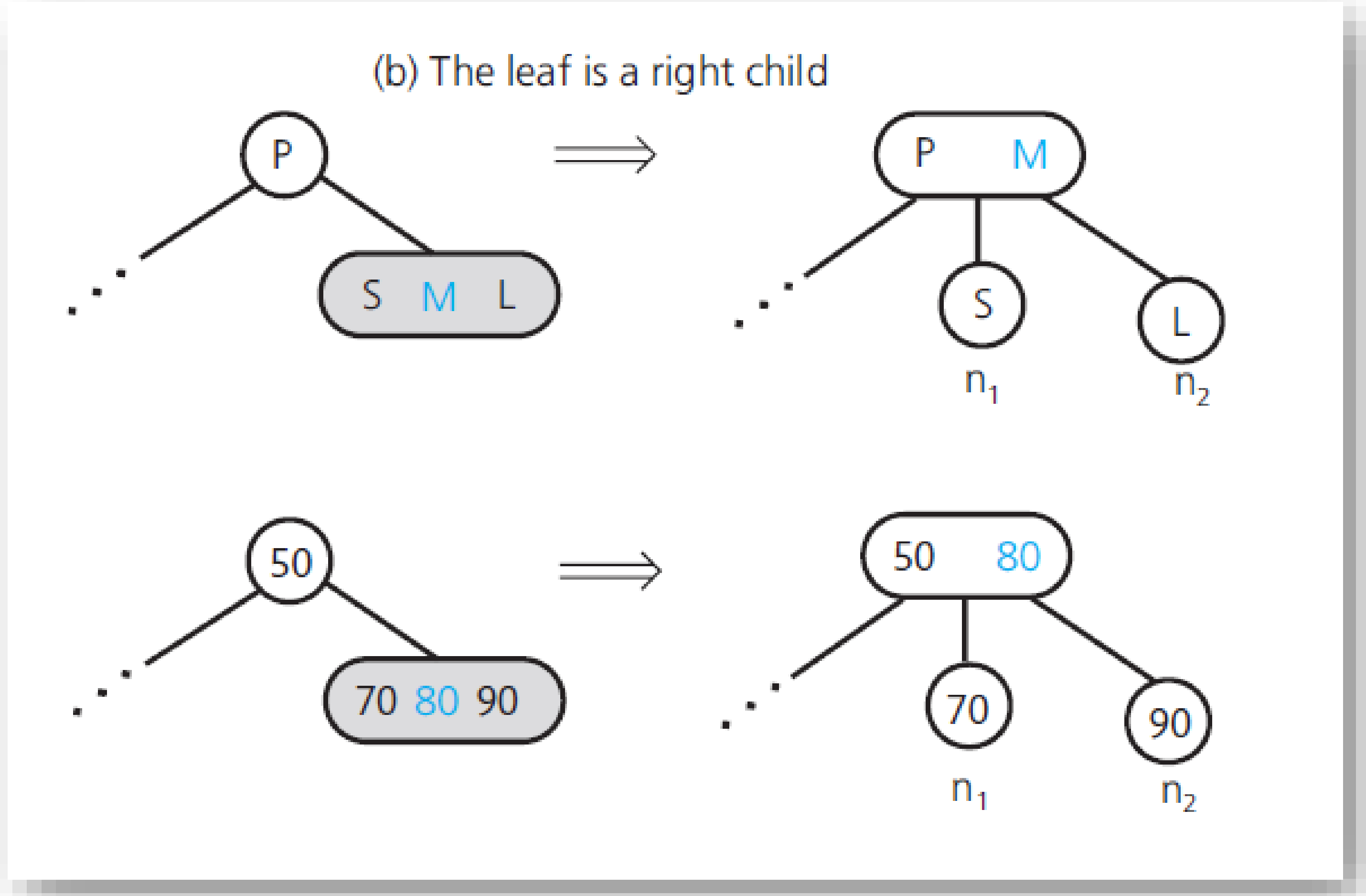
- The tree after the adding 35, 34, and 33 to the tree

Adding Data to a 2-3 Tree



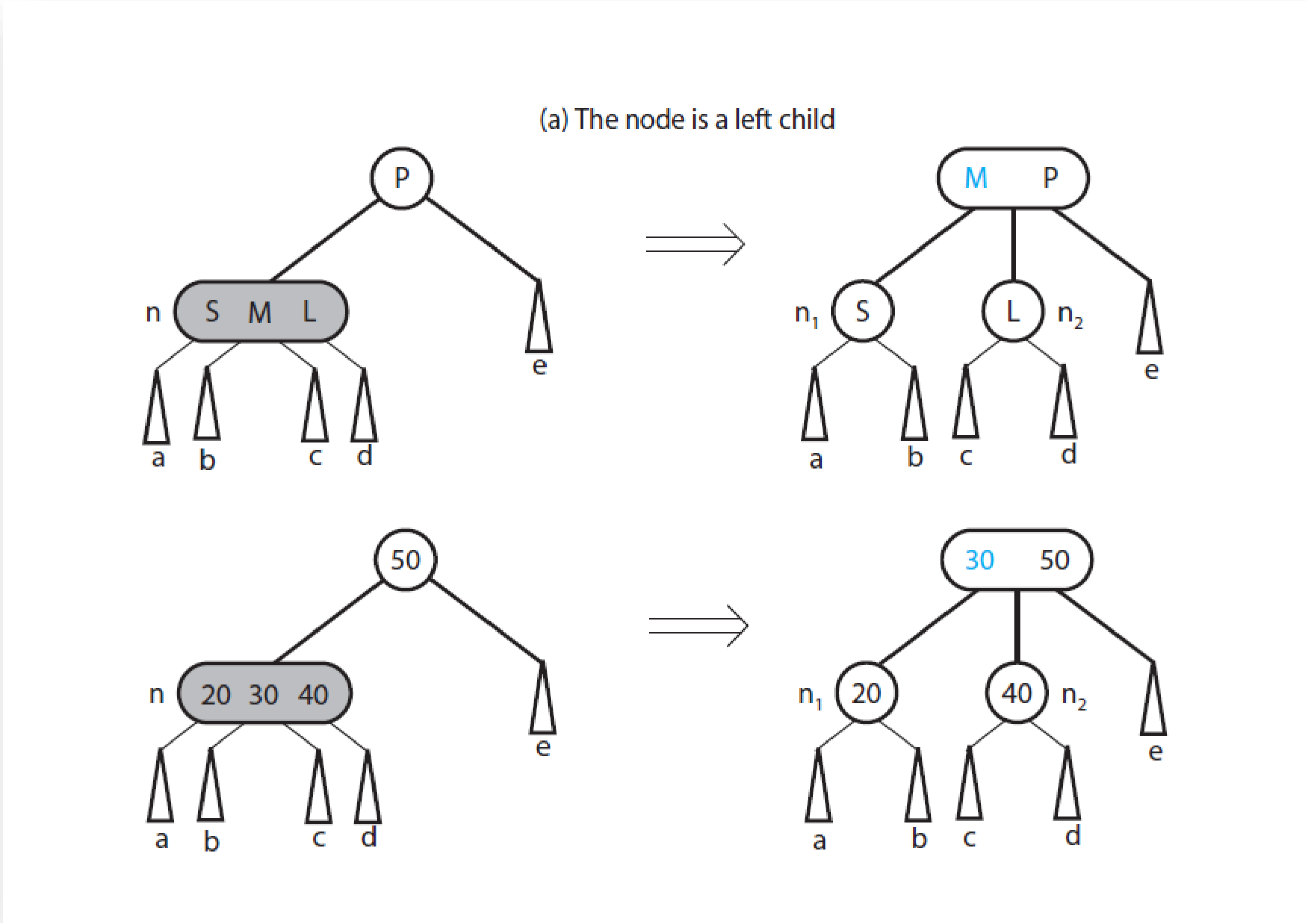
- Splitting a leaf in a 2-3 tree in general and in a specific example

Adding Data to a 2-3 Tree



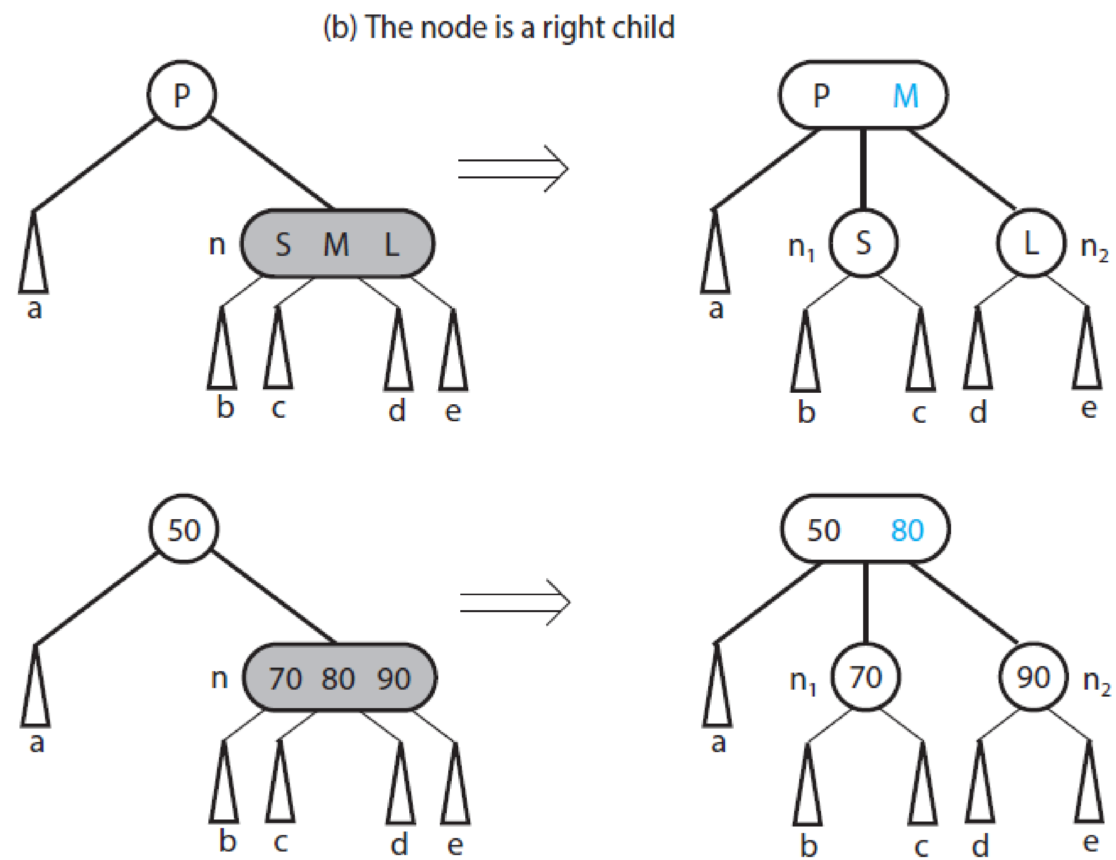
- Splitting a leaf in a 2-3 tree in general and in a specific example

Adding Data to a 2-3 Tree



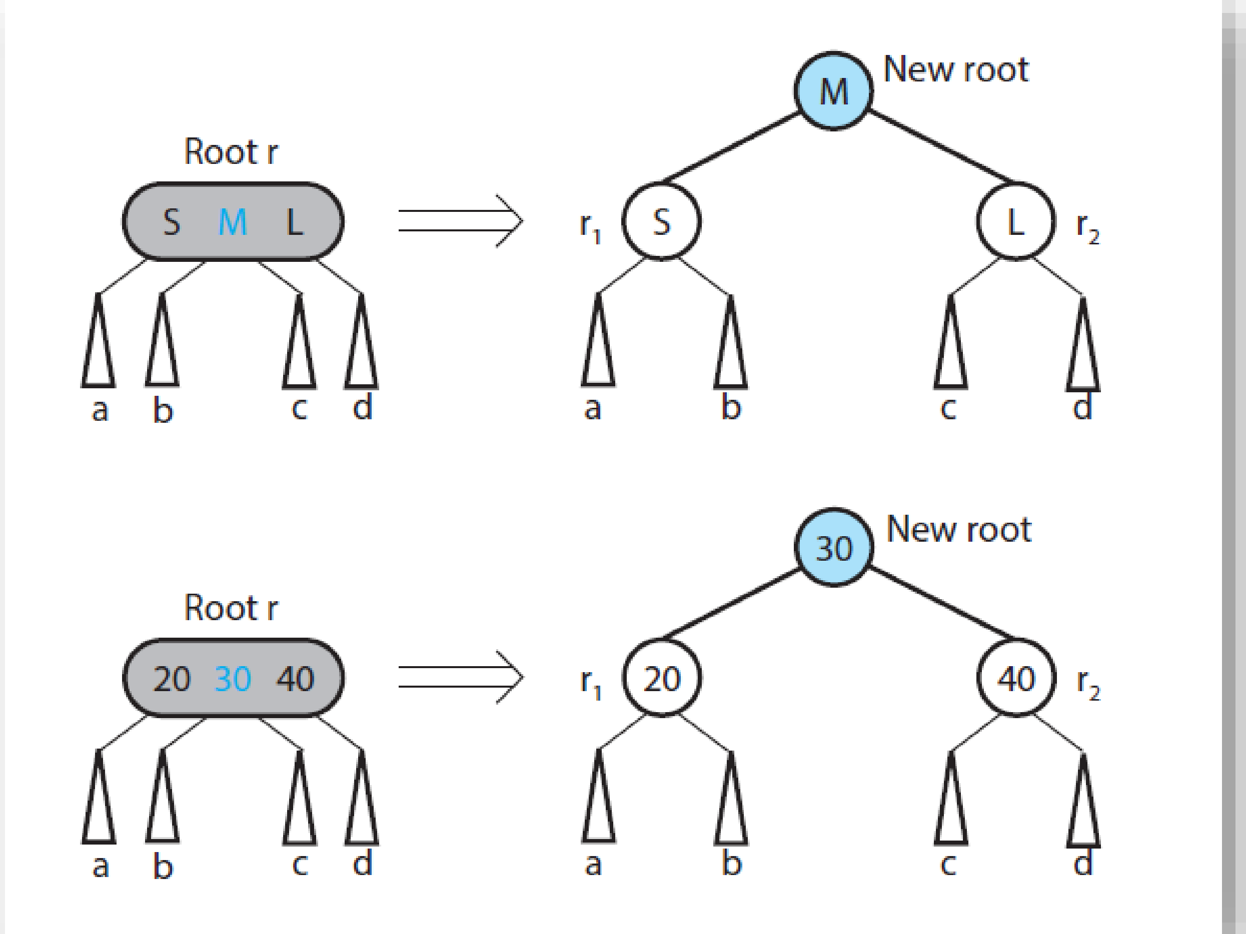
- Splitting an internal node in a 2-3 tree in general and in a specific example

Adding Data to a 2-3 Tree



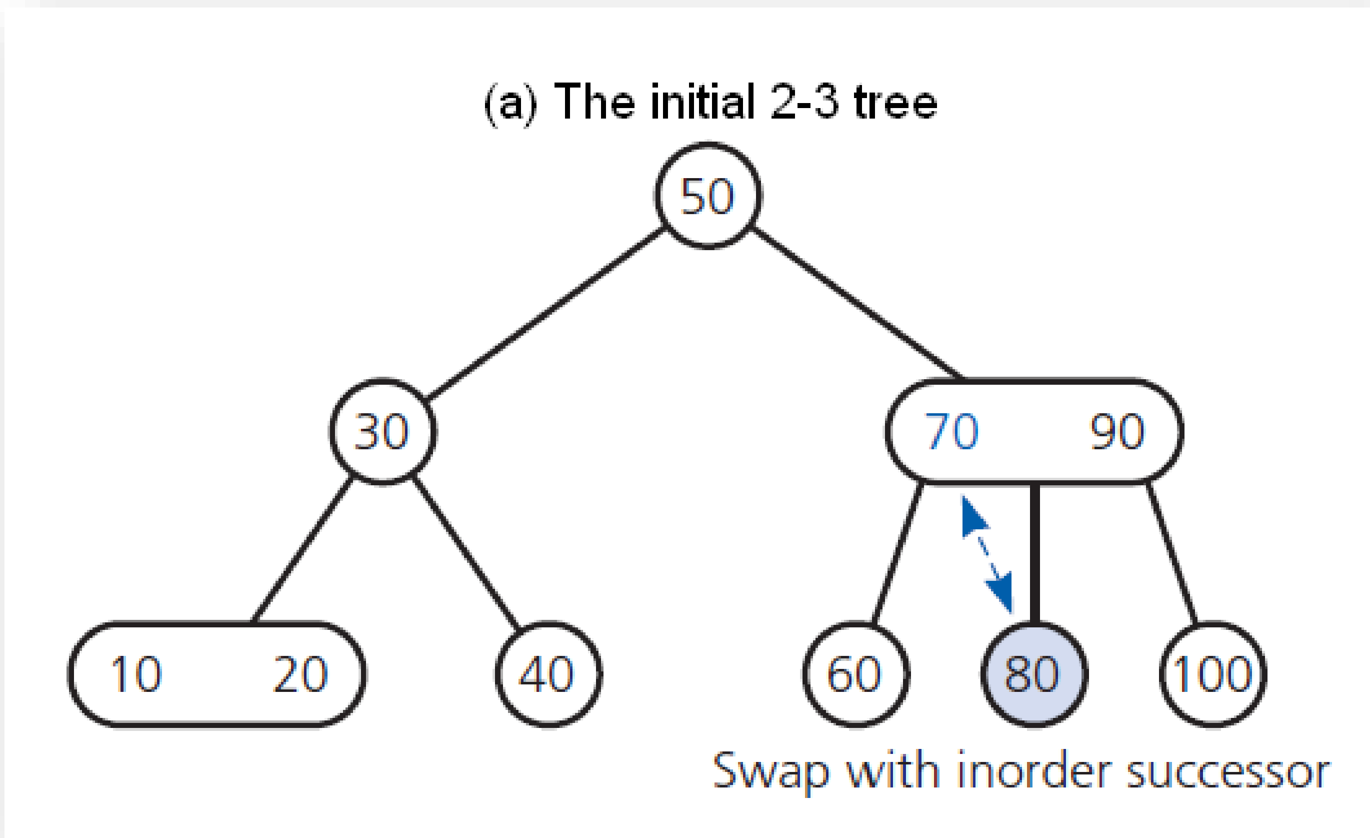
- Splitting an internal node in a 2-3 tree in general and in a specific example

Adding Data to a 2-3 Tree



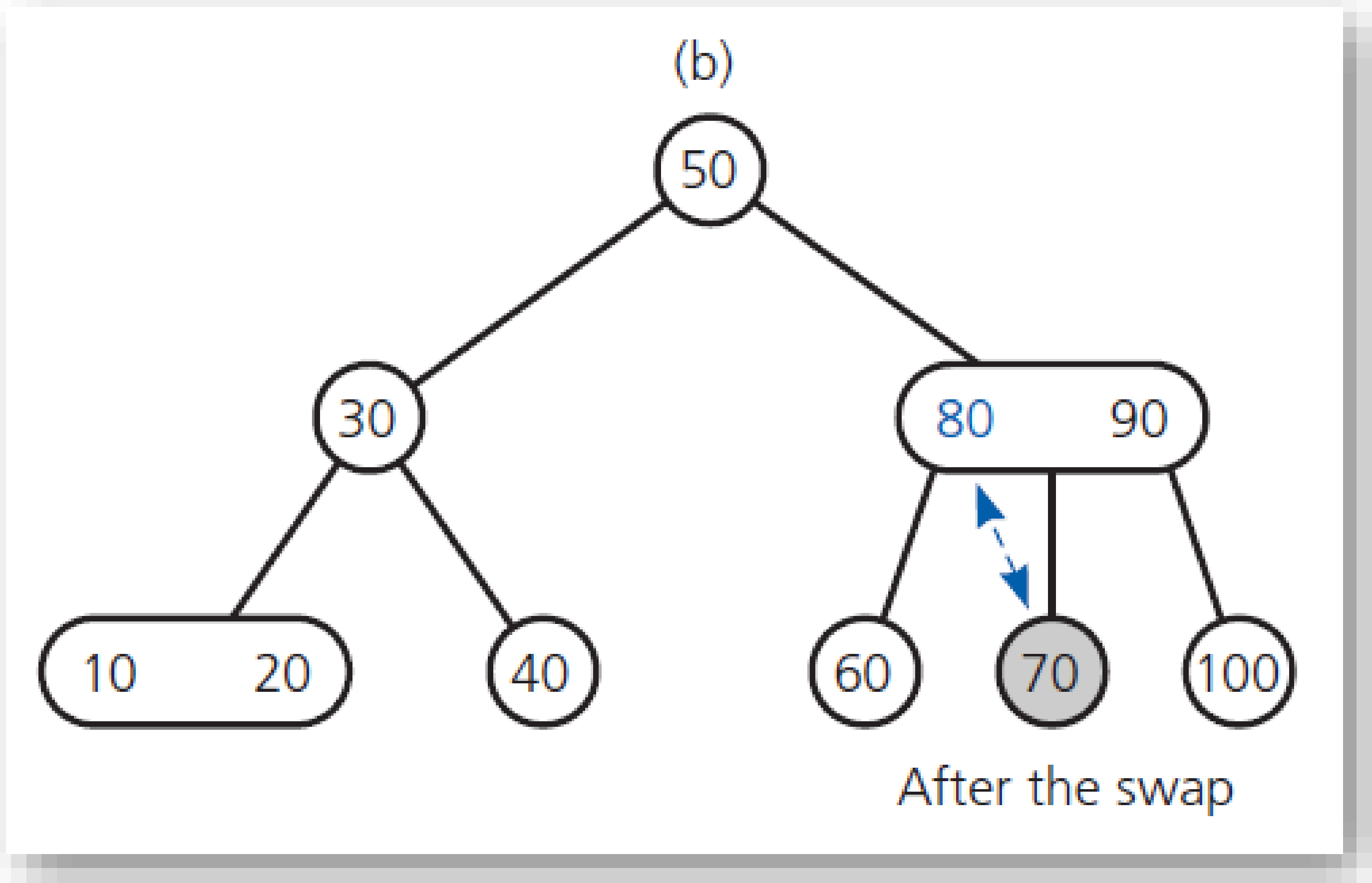
- Splitting the root of a 2-3 tree general and in a specific example

Removing Data from a 2-3 Tree



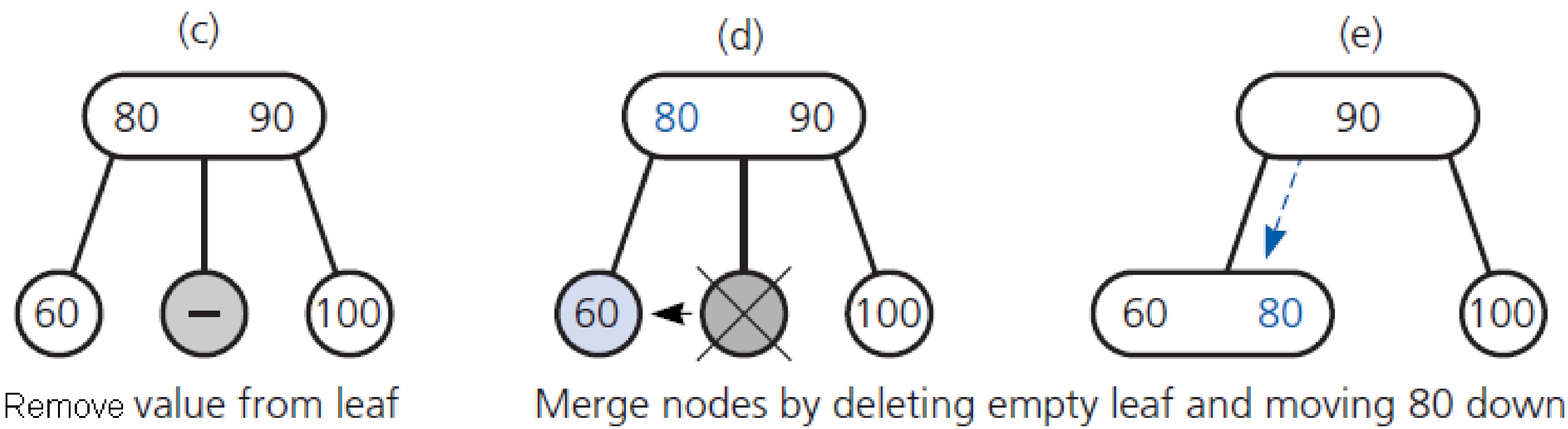
- The steps for removing 70 from the 2-3 tree

Removing Data from a 2-3 Tree



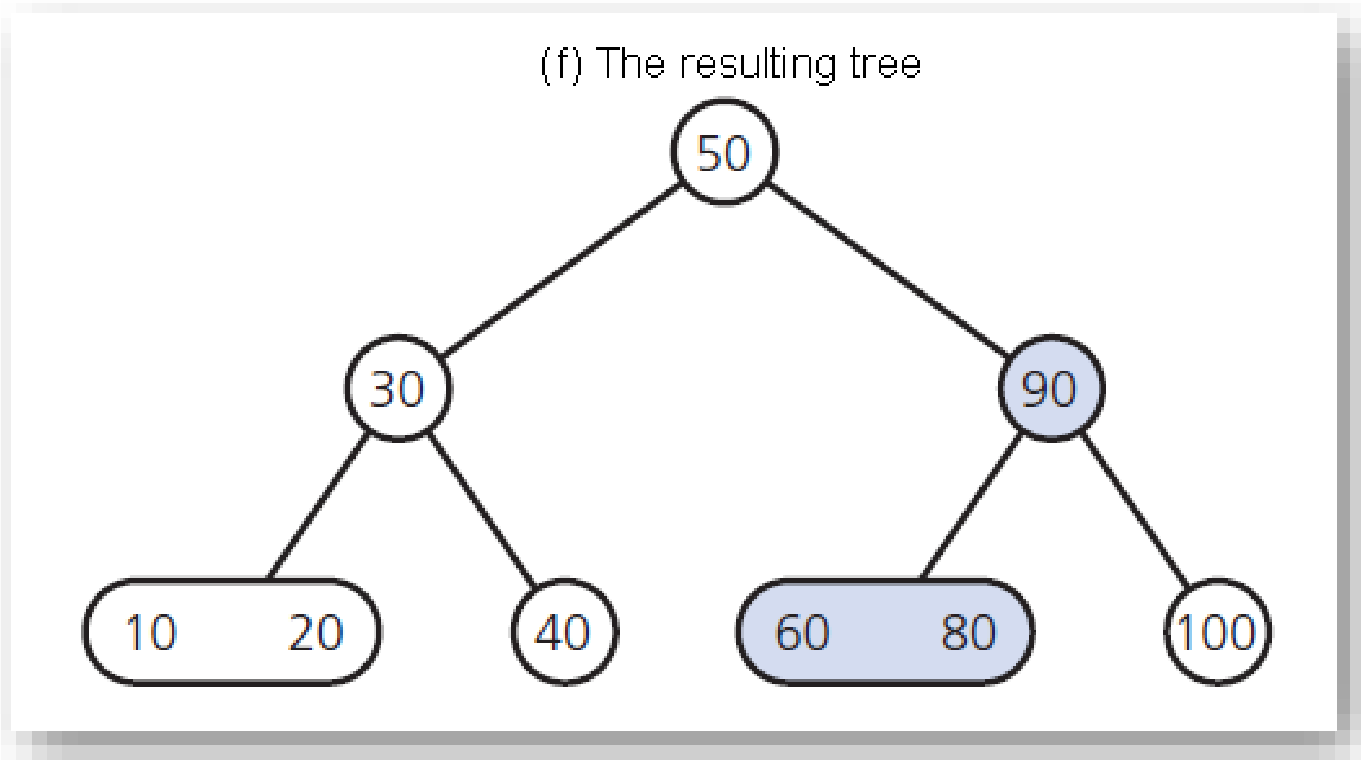
- The steps for removing 70 from the 2-3 tree

Removing Data from a 2-3 Tree



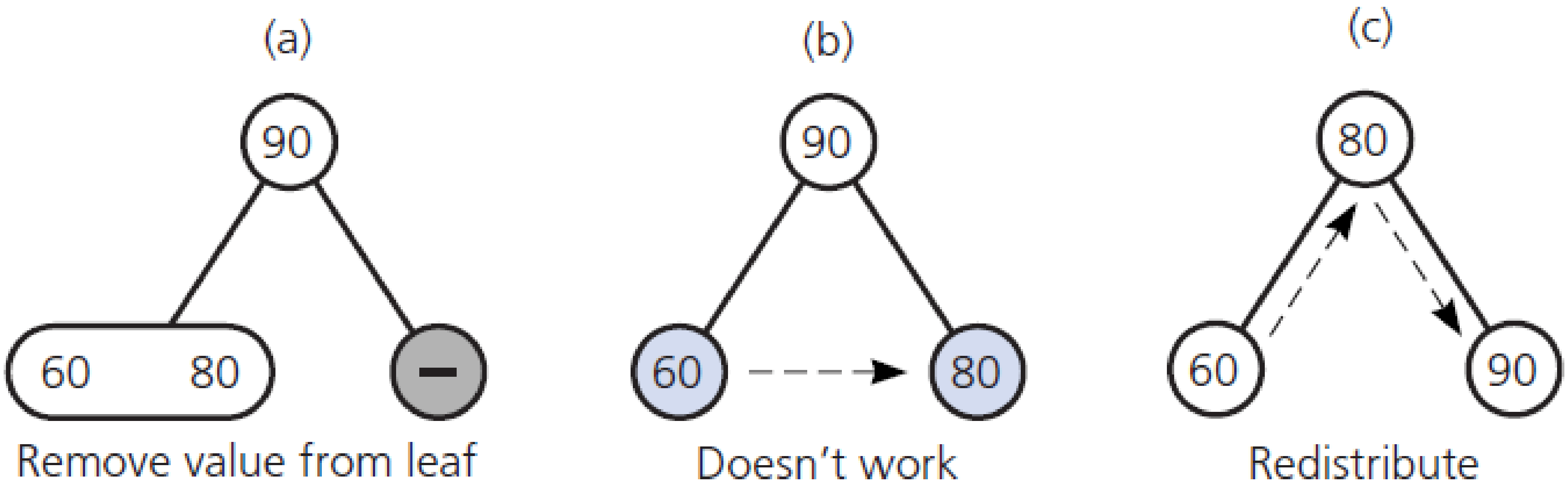
- The steps for removing 70 from the 2-3 tree

Removing Data from a 2-3 Tree



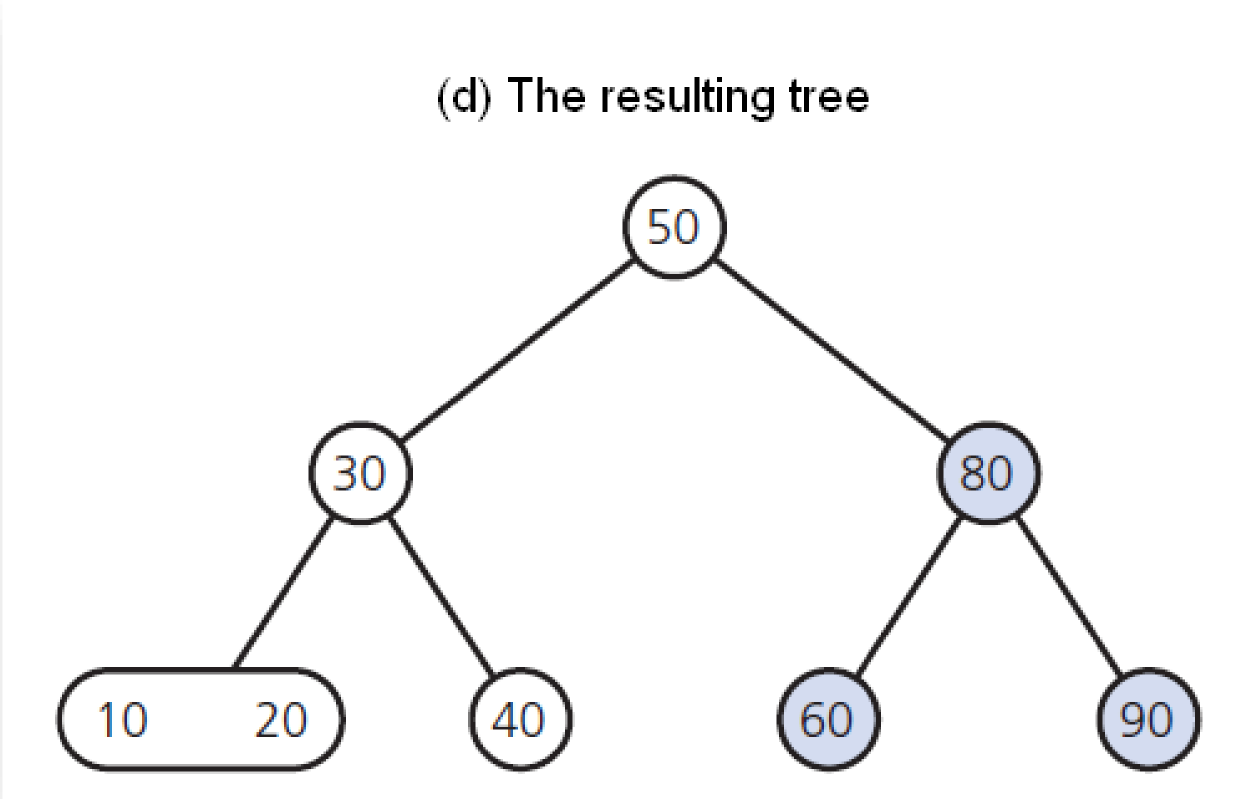
- The steps for removing 70 from the 2-3 tree

Removing Data from a 2-3 Tree



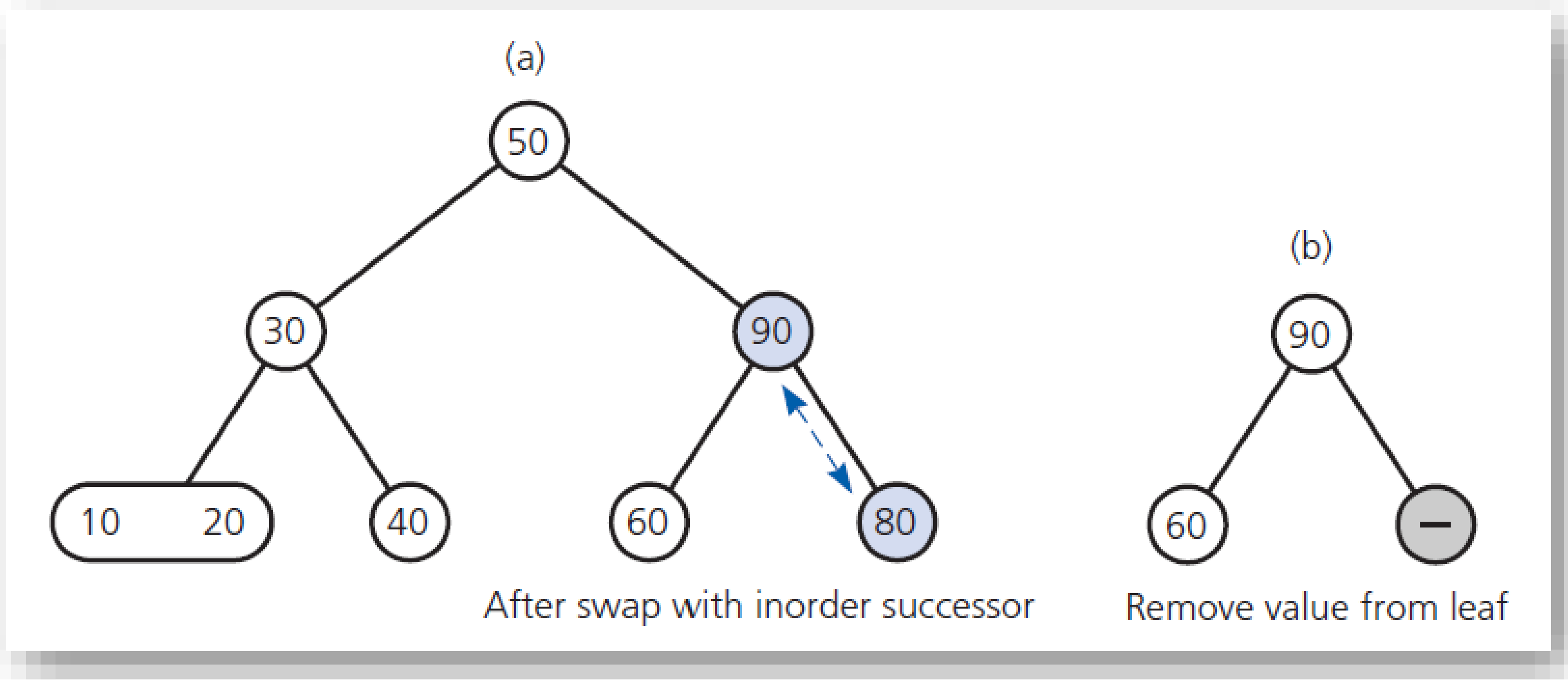
- The steps for removing 100 from the tree

Removing Data from a 2-3 Tree



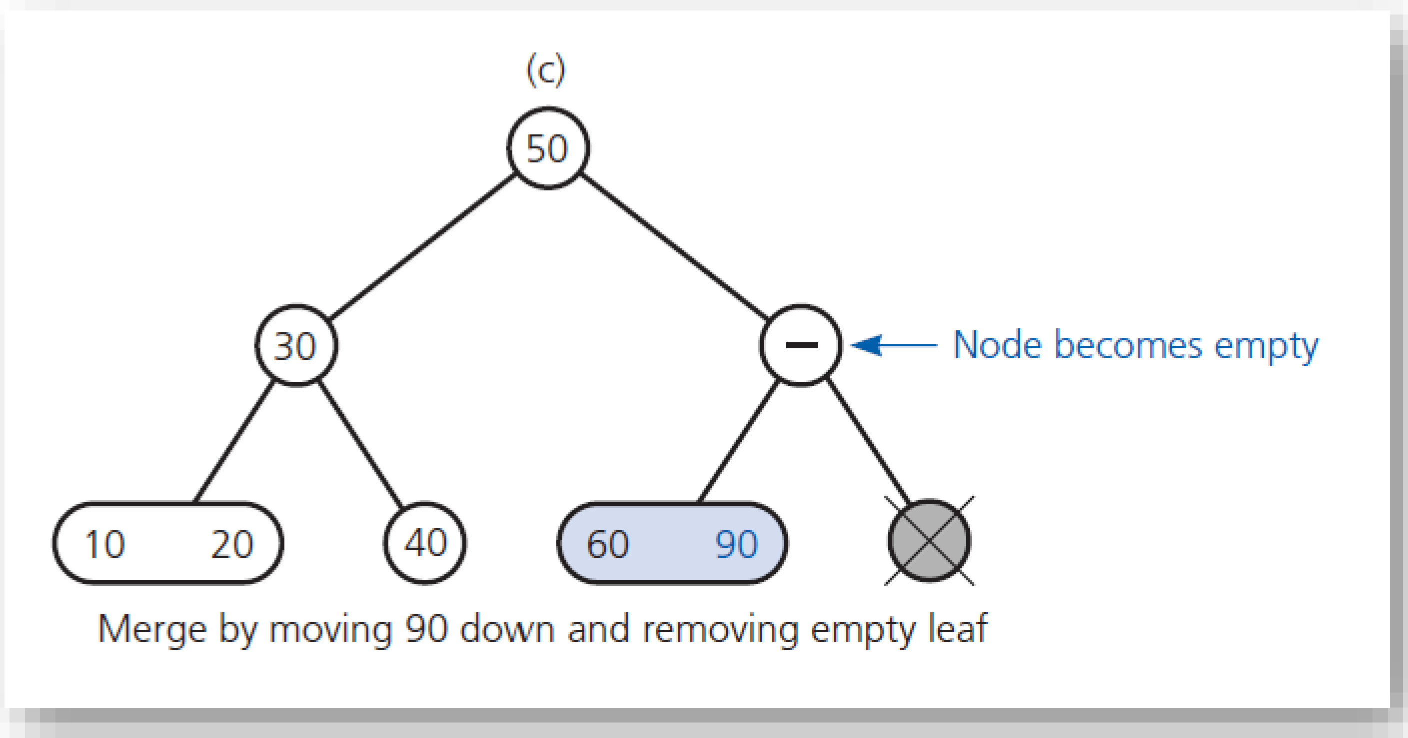
- The steps for removing 100 from the tree

Removing Data from a 2-3 Tree



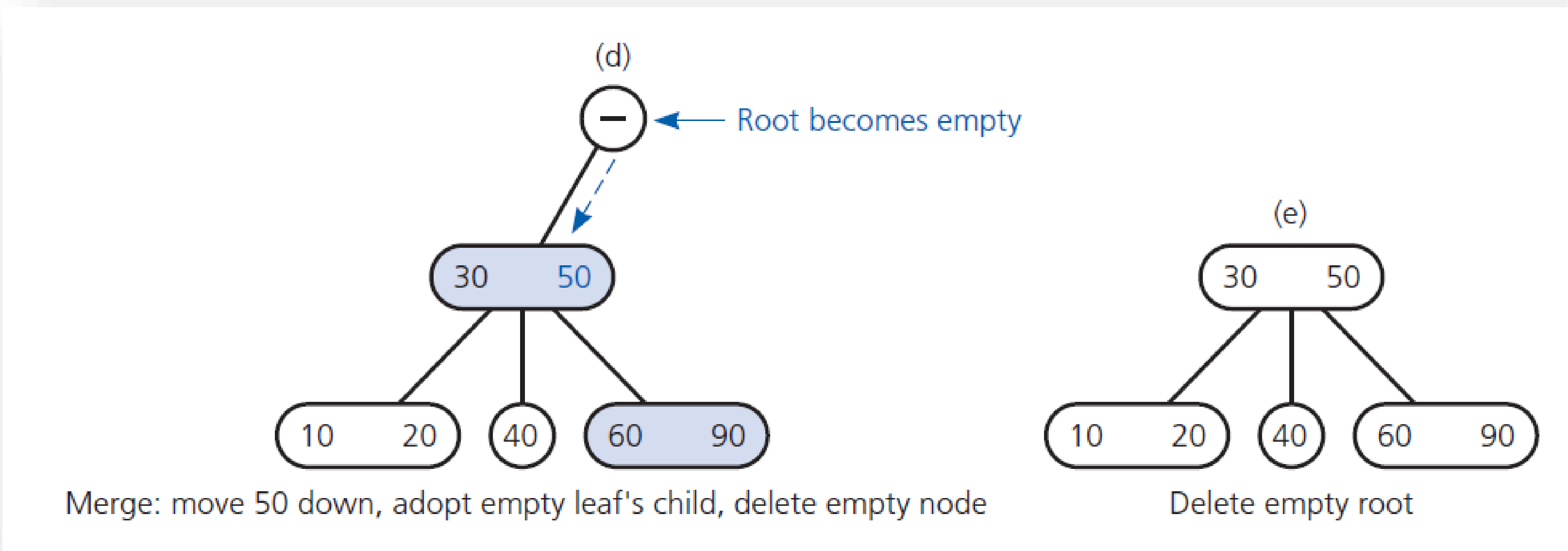
- The steps for removing 80 from the 2-3 tree

Removing Data from a 2-3 Tree



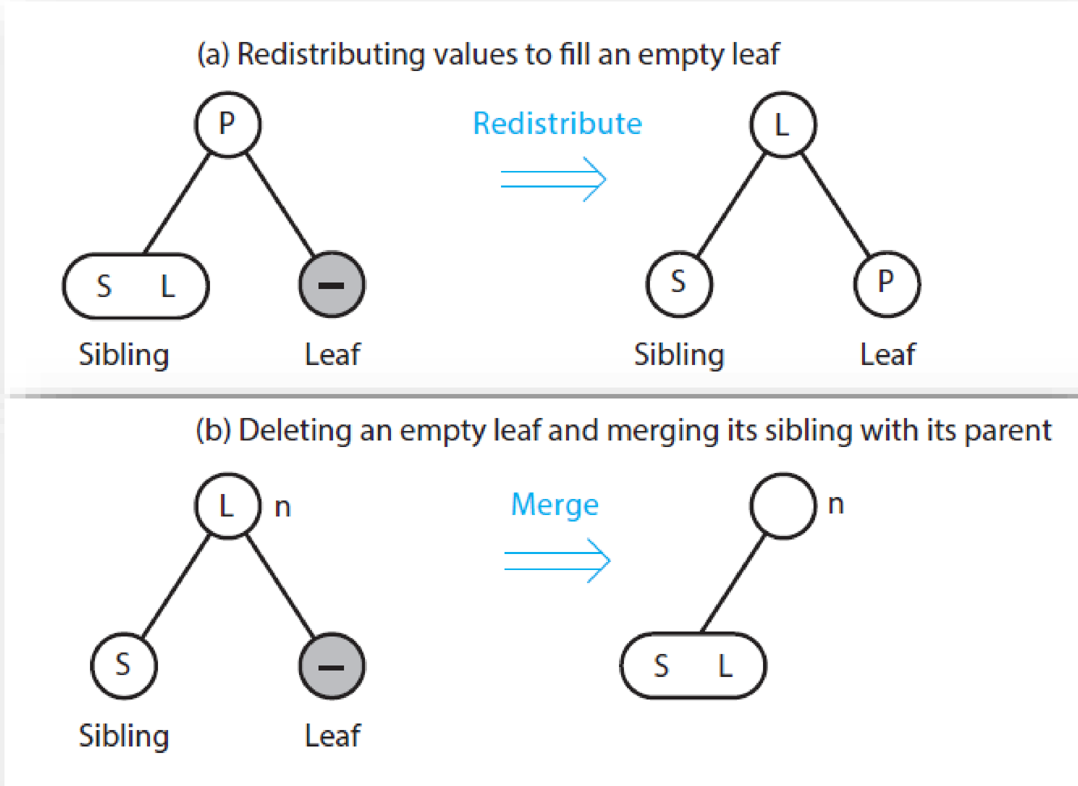
- The steps for removing 80 from the 2-3 tree

Removing Data from a 2-3 Tree



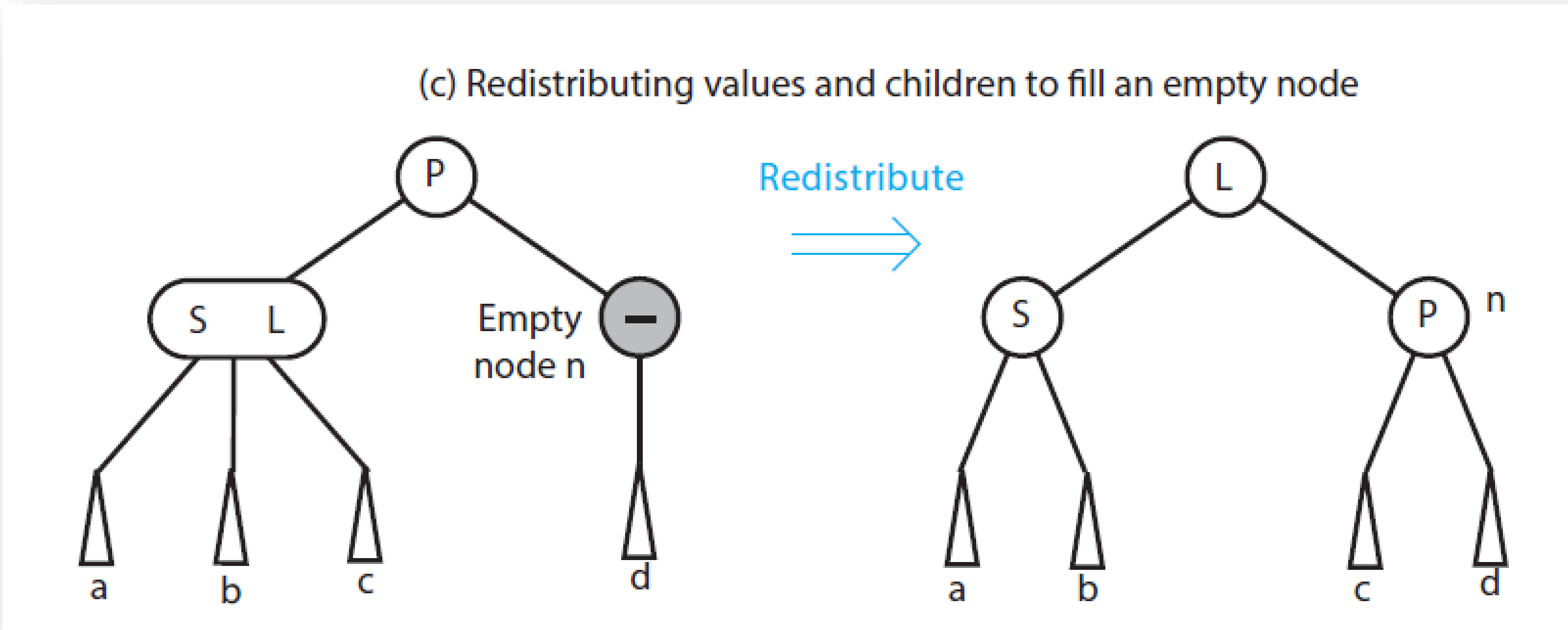
- The steps for removing 80 from the 2-3 tree

Removing Data from a 2-3 Tree



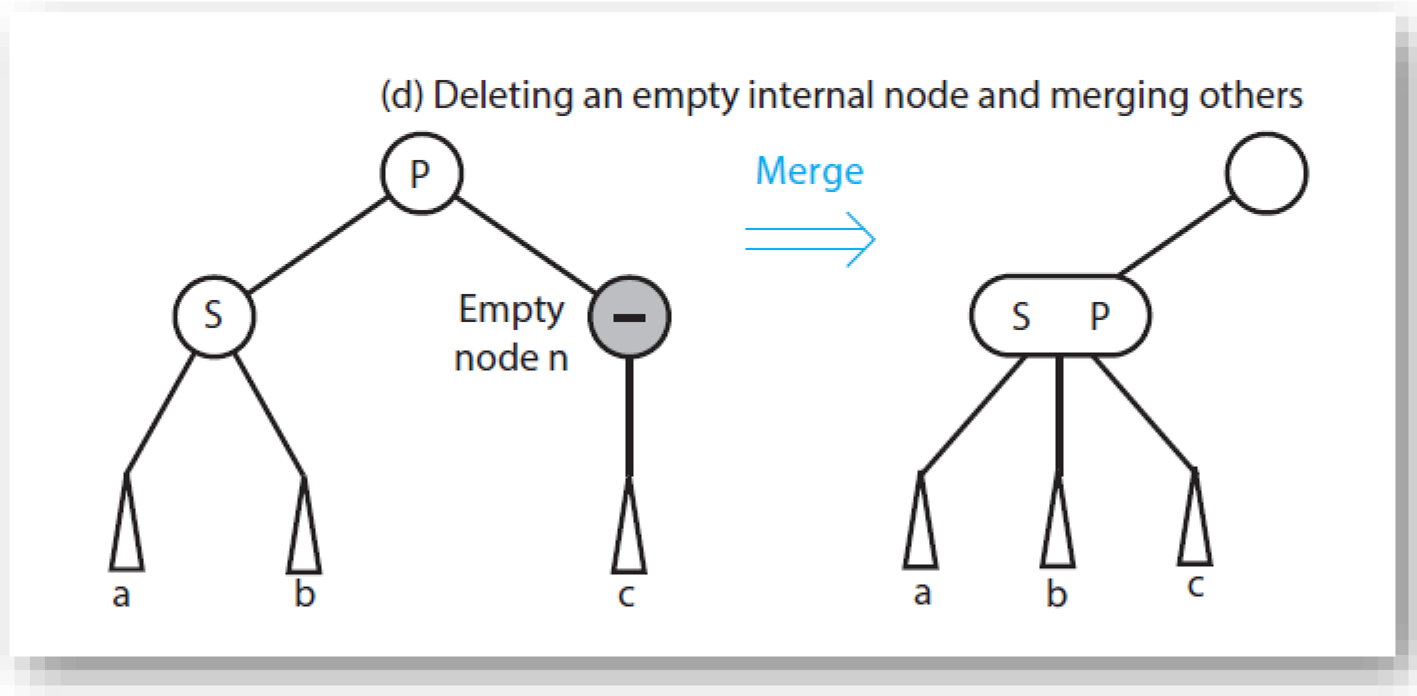
- Possible situations during the removal of a data item

Removing Data from a 2-3 Tree



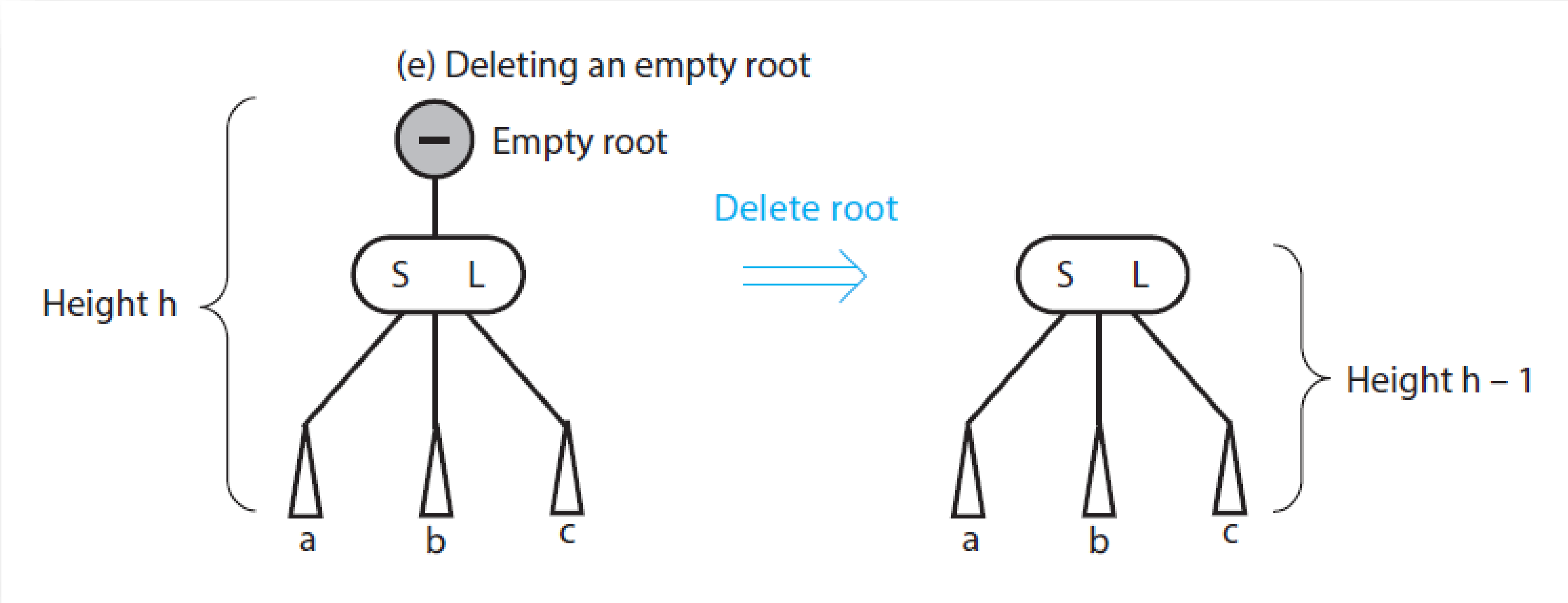
- Possible situations during the removal of a data item

Removing Data from a 2-3 Tree



- Possible situations during the removal of a data item

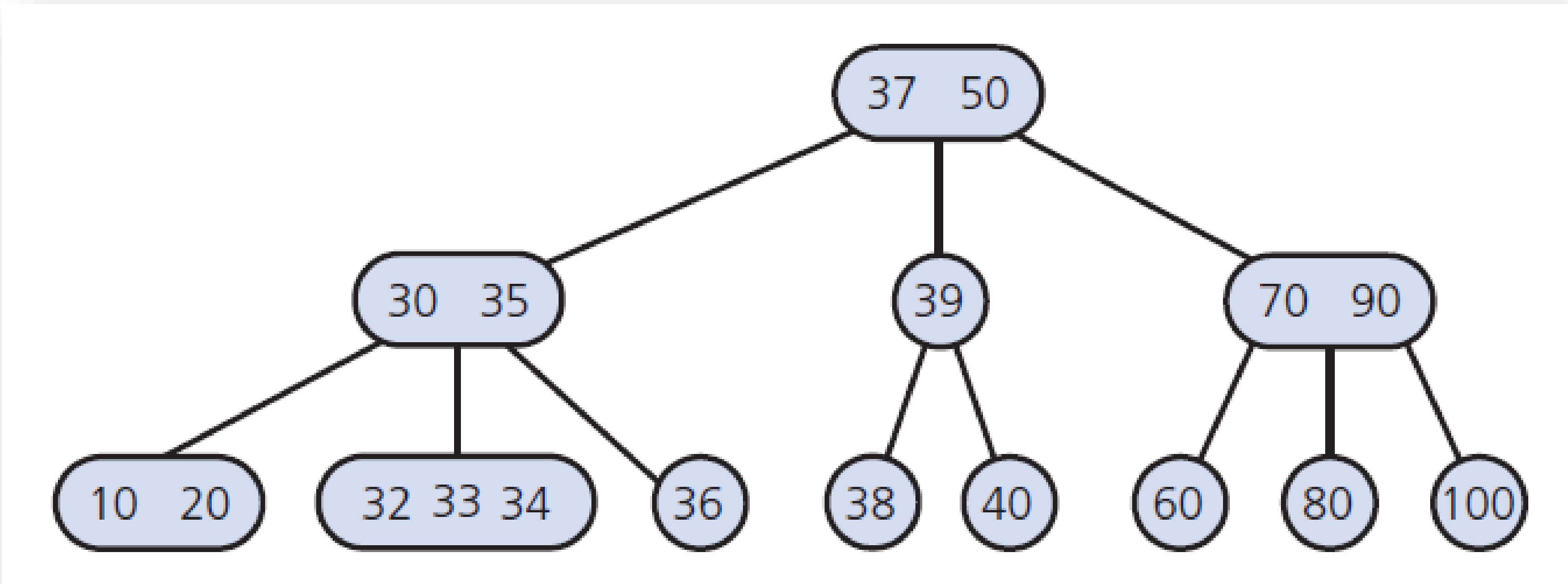
Removing Data from a 2-3 Tree



- Possible situations during the removal of a data item

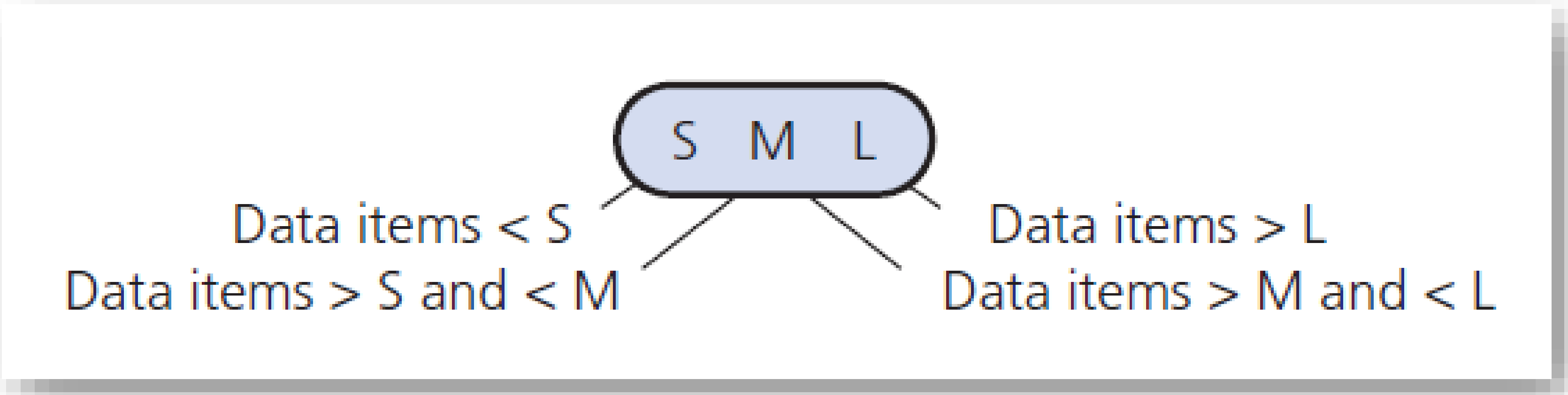
2-3-4 Trees

2-3-4 Trees



- A 2-3-4 tree with the same data items as the 2-3 tree

2-3-4 Trees



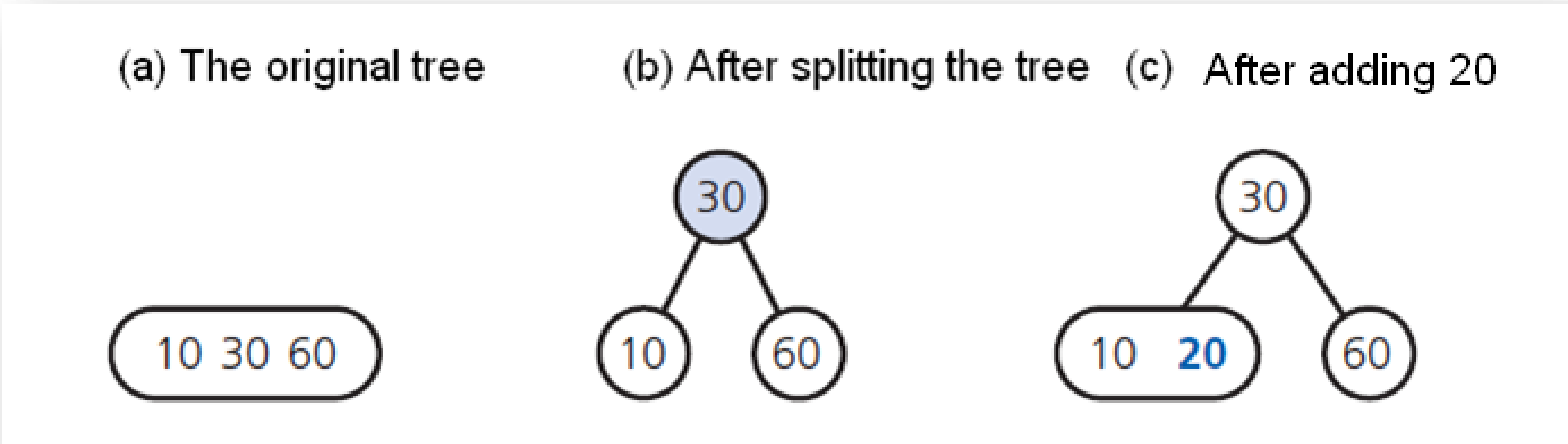
- A 4-node in a 2-3-4 tree

2-3-4 Trees

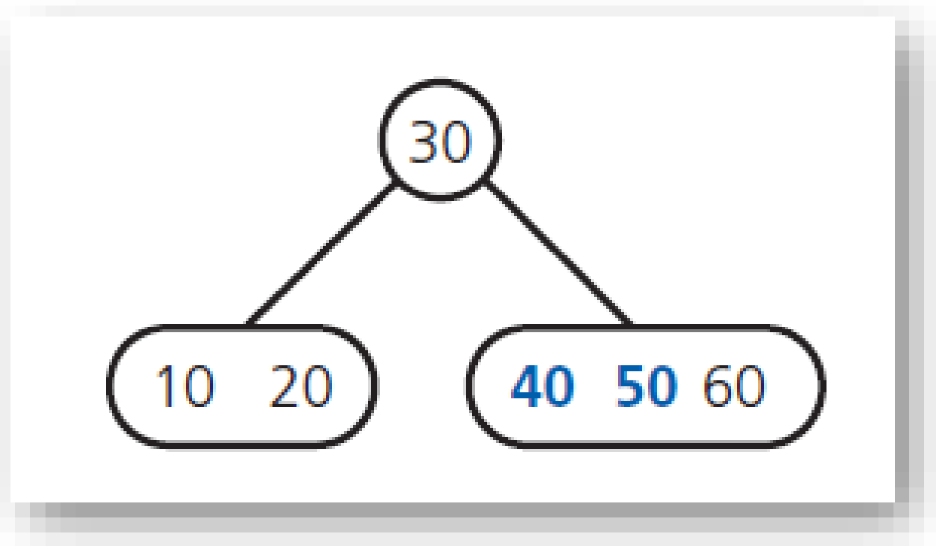
- Searching and traversing
  - Simple extensions of corresponding algorithms for a 2-3 tree
- Adding data
  - Like addition algorithm for 2-3 tree
  - Splits node by moving one data item up to parent node

Adding Data to 2-3-4 Trees

- Adding 20 to a one-node 2-3-4 tree

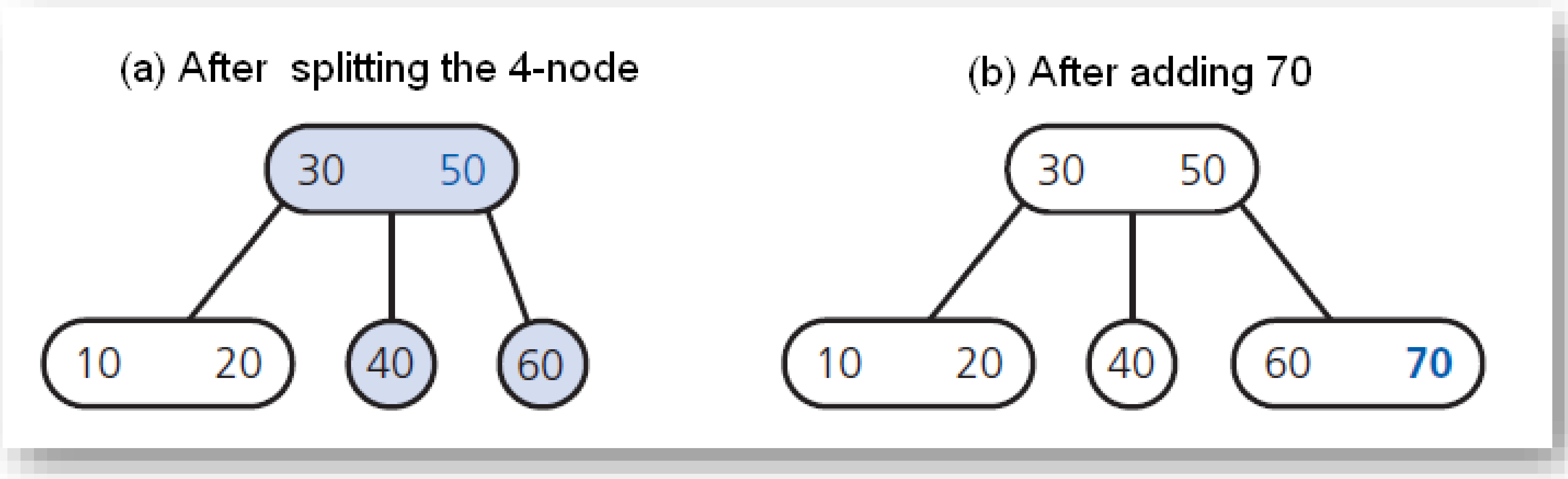


Adding Data to 2-3-4 Trees



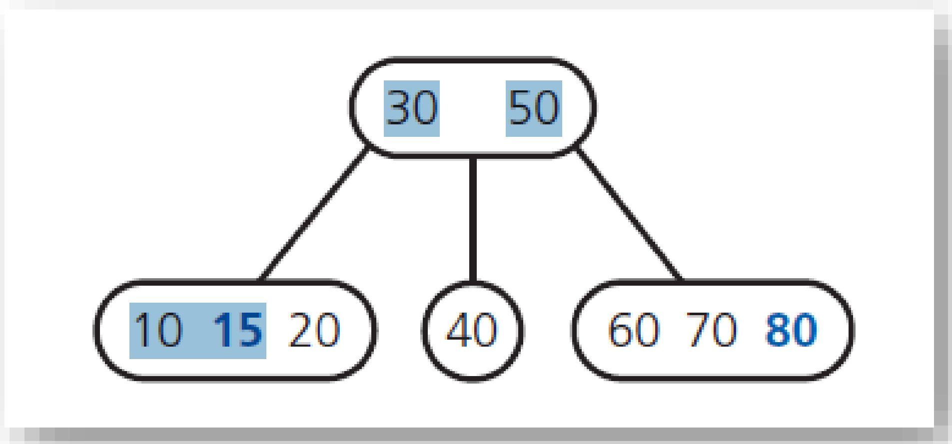
- After adding 50 and 40 to the tree

Adding Data to 2-3-4 Trees



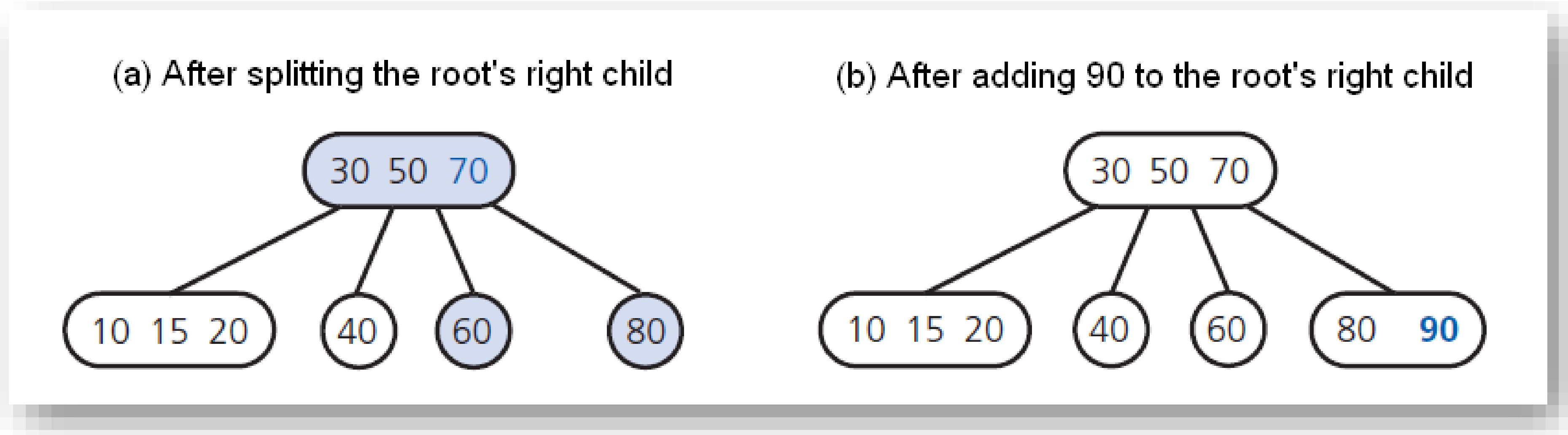
- The steps for adding 70 to the tree

Adding Data to 2-3-4 Trees



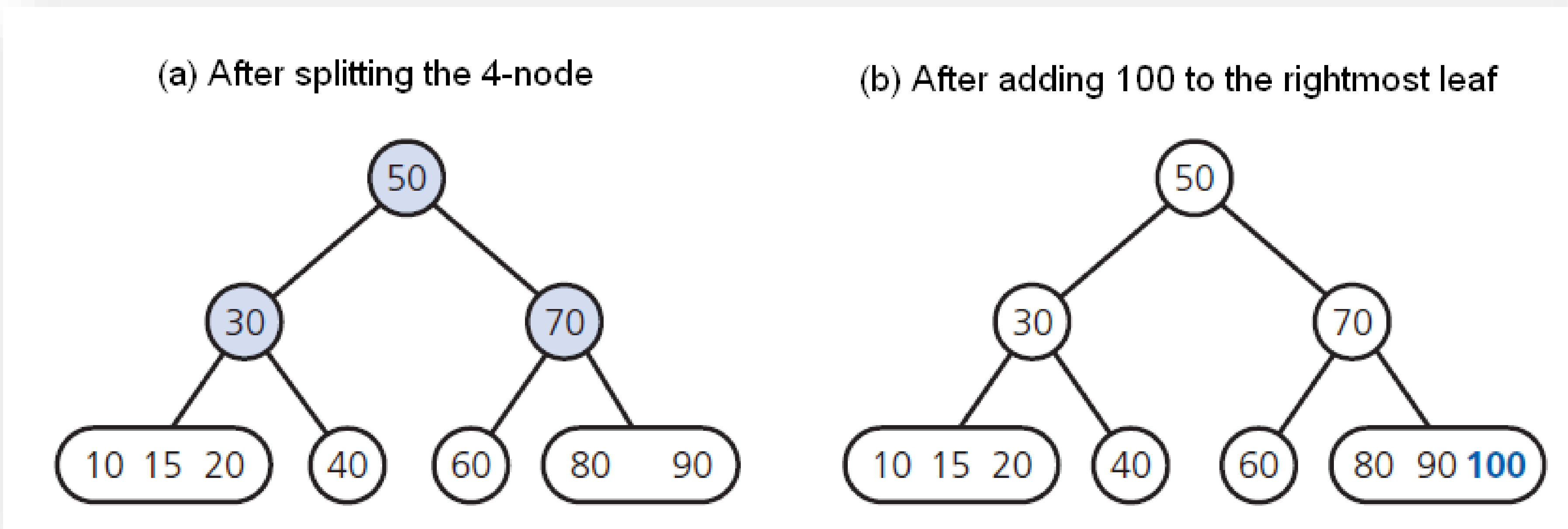
- After adding 80 and 15 to the tree

Adding Data to 2-3-4 Trees



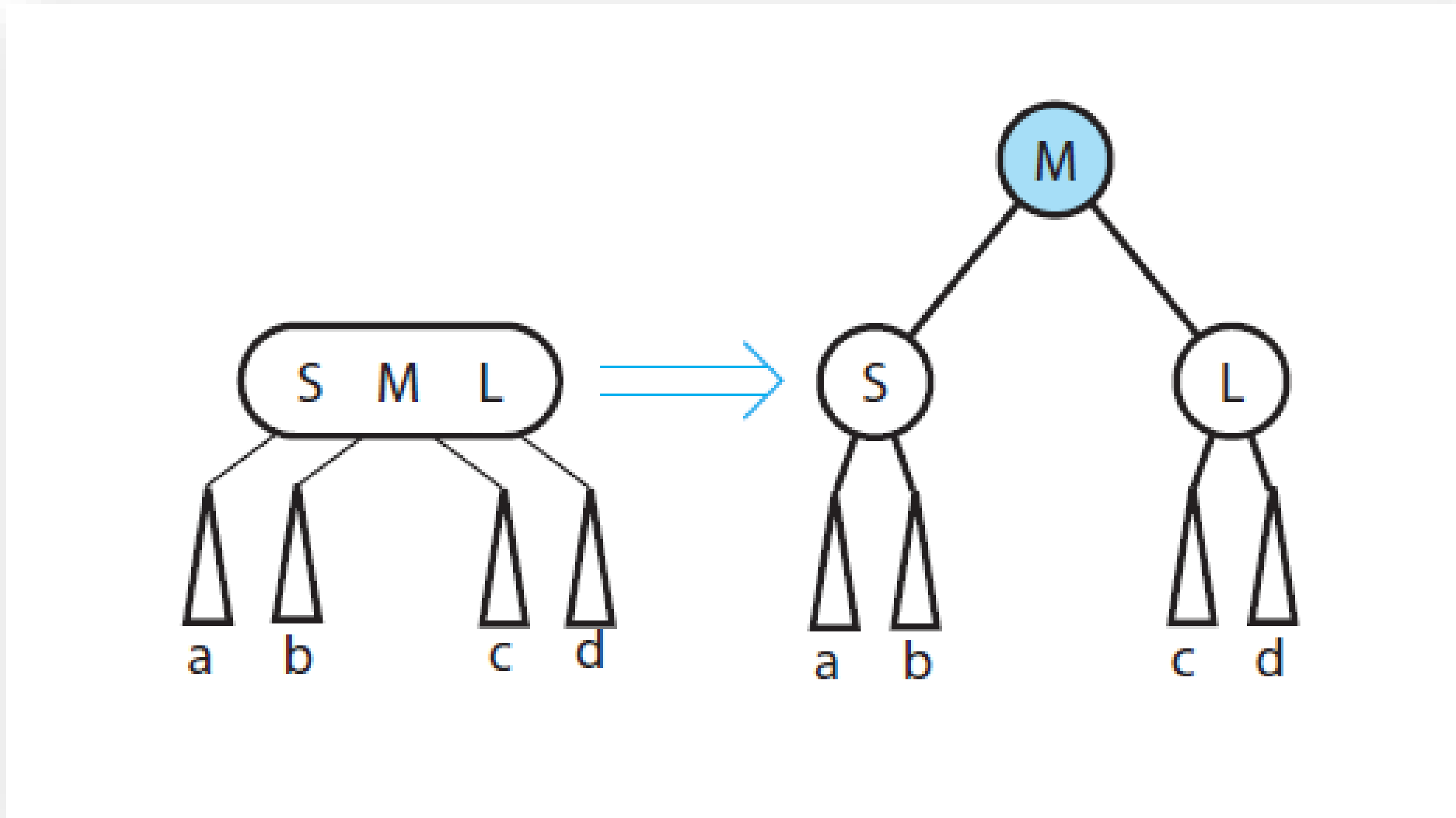
- The steps for adding 90 to the tree

Adding Data to 2-3-4 Trees



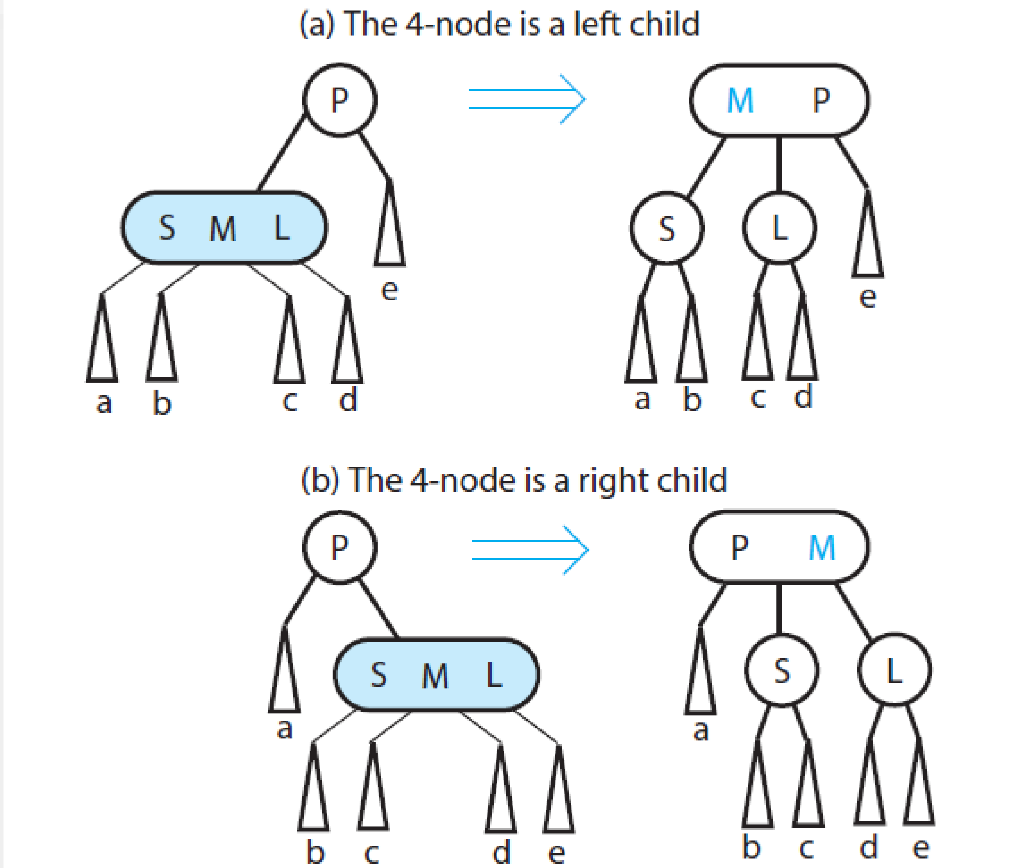
- The steps for adding 100 to the tree

Adding Data to 2-3-4 Trees



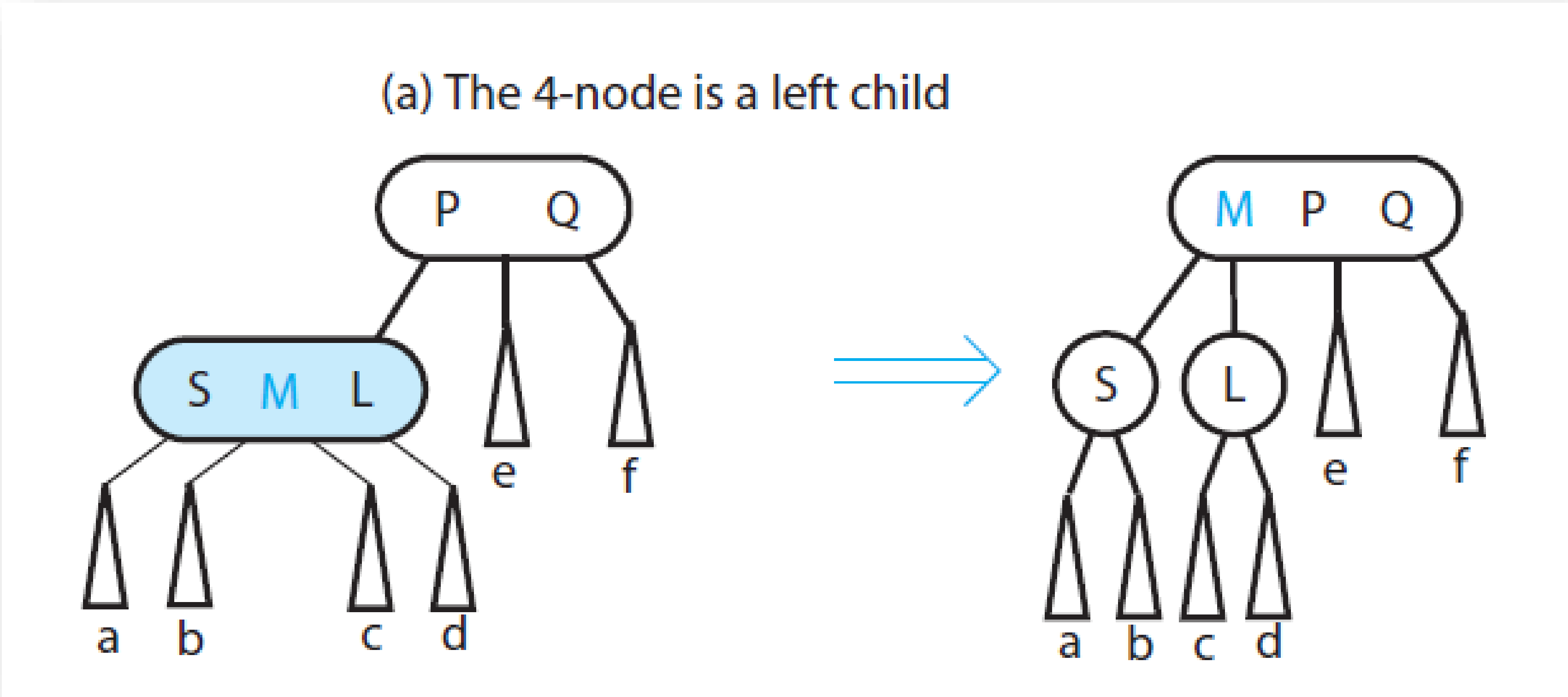
- Splitting a 4-node root when adding data to a 2-3-4 tree

Adding Data to 2-3-4 Trees



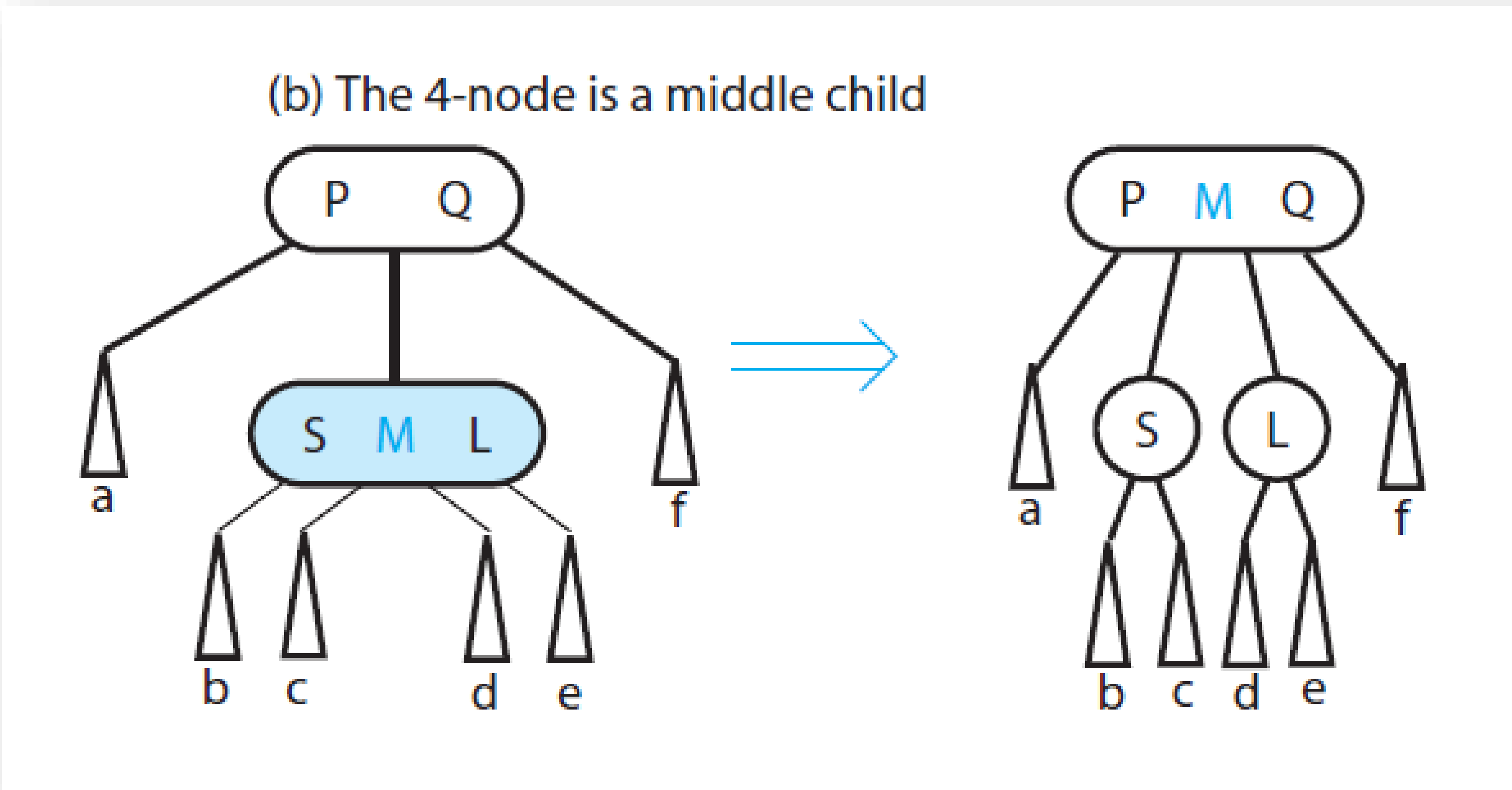
- Splitting a 4-node whose parent is a 2-node when adding data to a 2-3-4 tree

Adding Data to 2-3-4 Trees



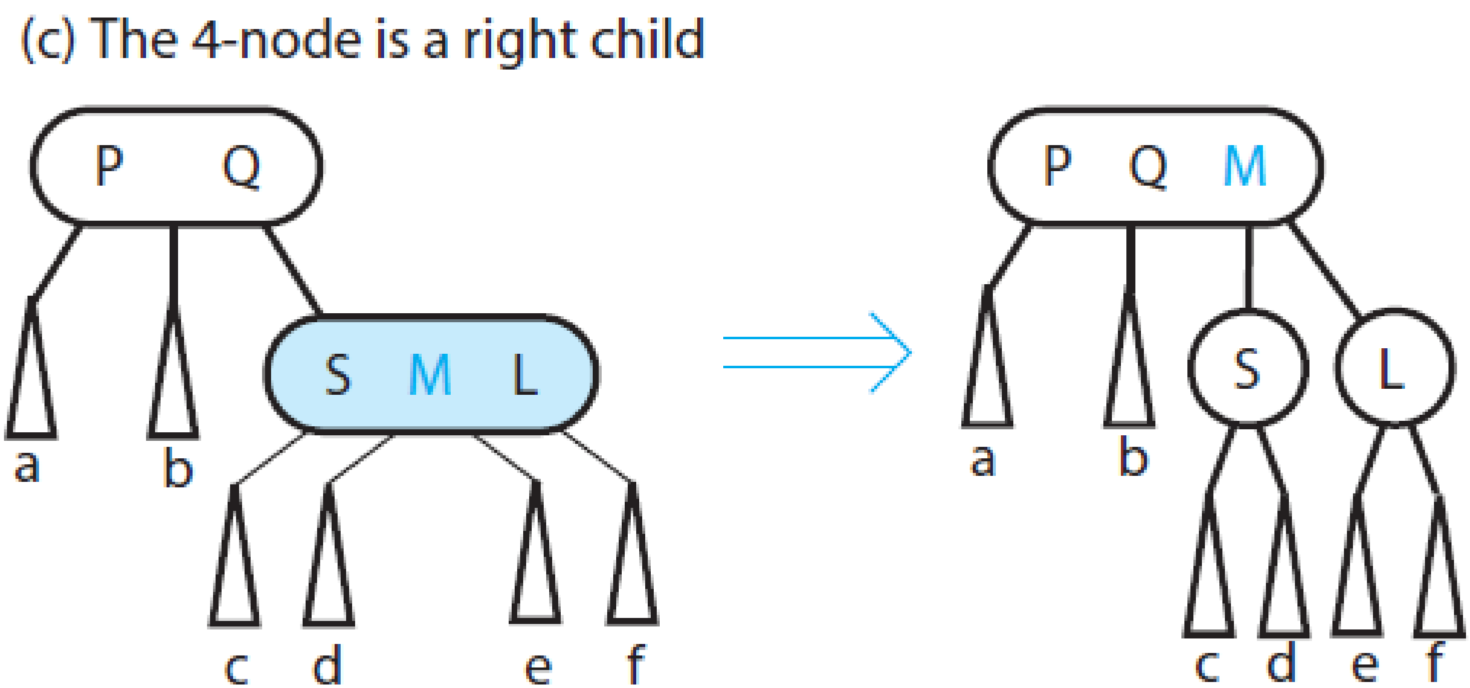
- Splitting a 4-node whose parent is a 3-node when adding data to a 2-3-4 tree

Adding Data to 2-3-4 Trees



- Splitting a 4-node whose parent is a 3-node when adding data to a 2-3-4 tree

Adding Data to 2-3-4 Trees



- Splitting a 4-node whose parent is a 3-node when adding data to a 2-3-4 tree

Removing Data from a 2-3-4 Tree

- Has same beginning as removal algorithm for a 2-3 tree
- Transform each 2-node into a 3-node or a 4-node
- Insertion and removal algorithms for 2-3-4 tree require fewer steps than for 2-3 tree

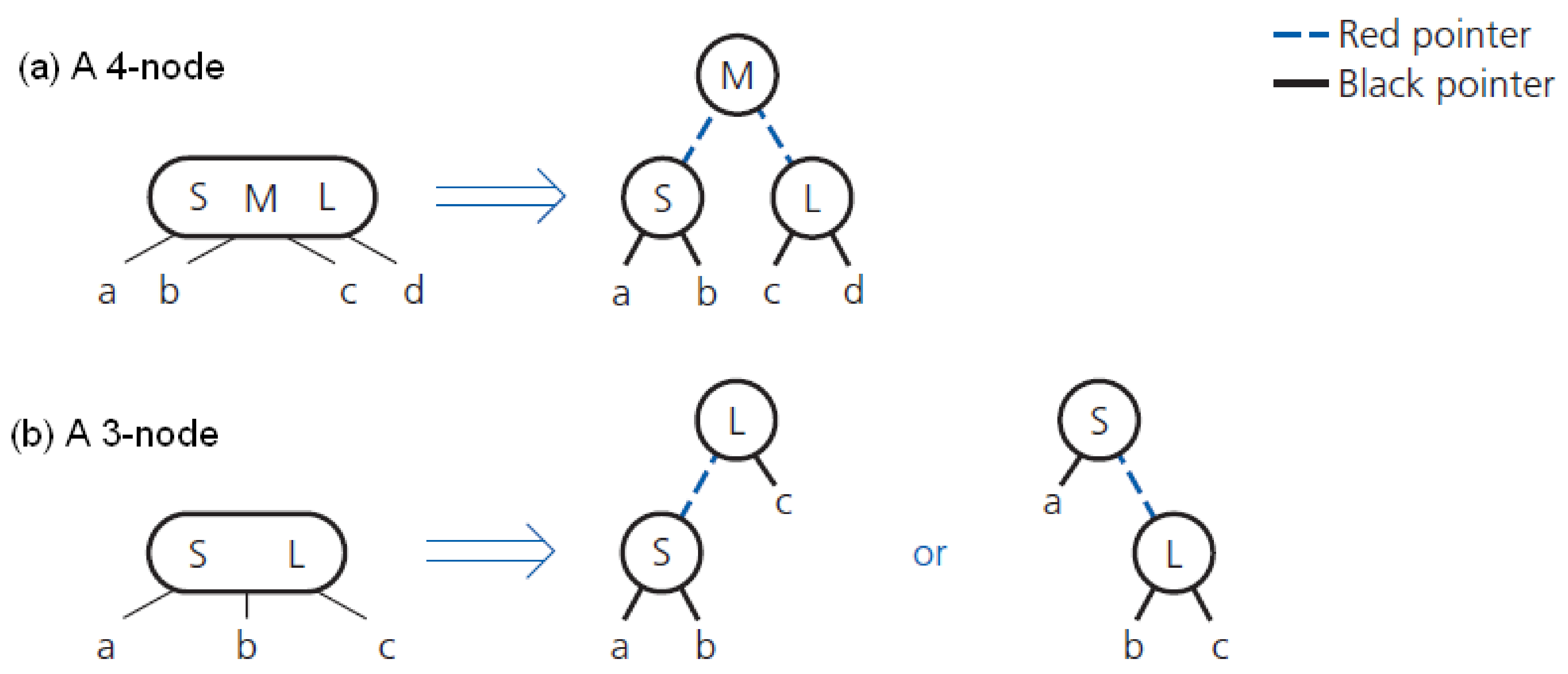
Red-Black Trees

Modified from Dr Monica Nicolescu

Red-Black Trees

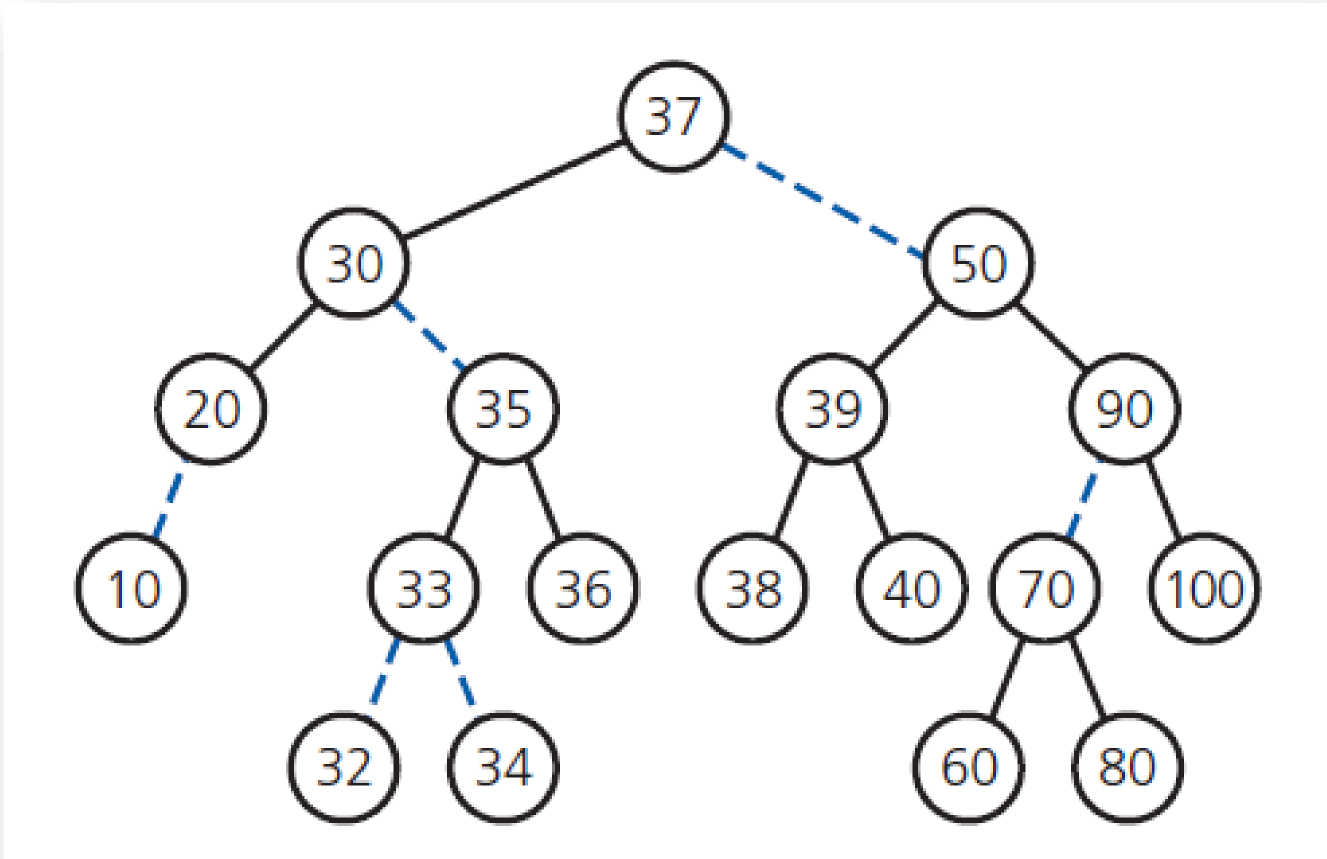
- A 2-3-4 tree requires more storage than binary search tree
- Red-black tree has advantages of a 2-3-4 tree but requires less storage
- In a red-black tree,
  - Red pointers link 2-nodes that now contain values that were in a 3-node or a 4-node

Red-Black Trees



- Red-black representation s of a 4-node and a 3-node

Red-Black Trees



- A red-black tree that represents the 2-3-4 tree

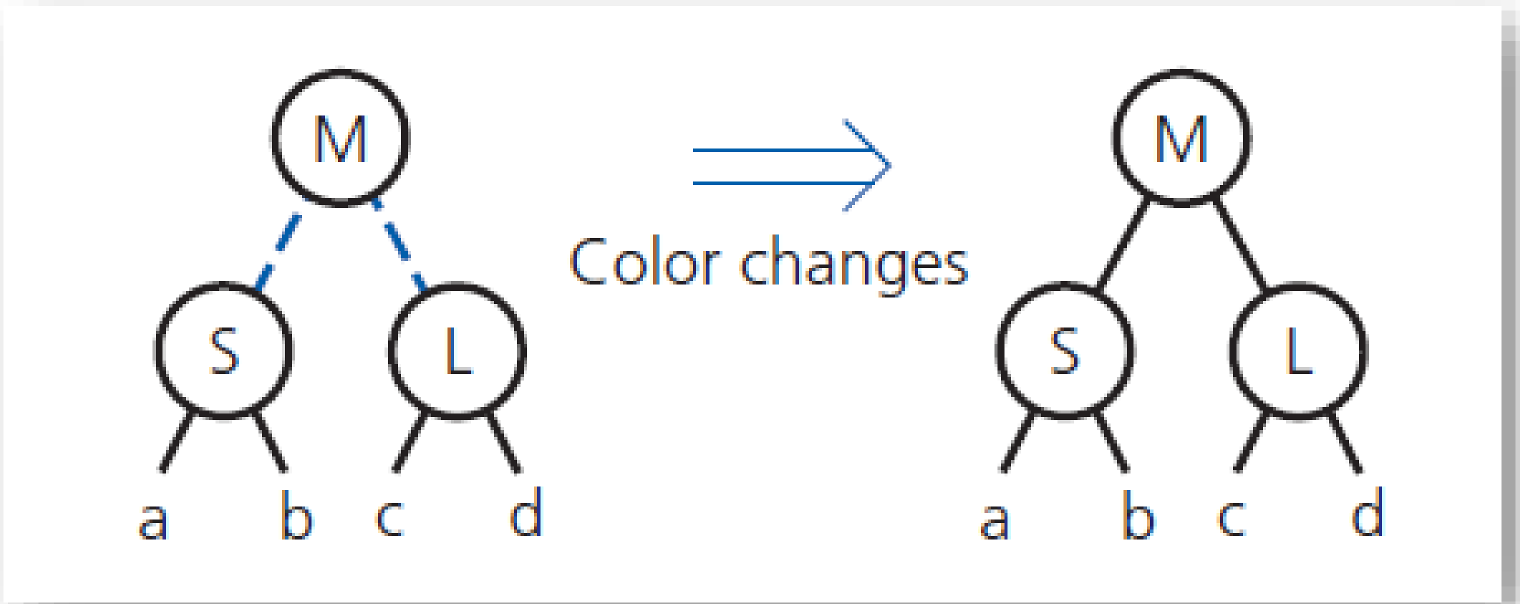
Searching and Traversing a Red-Black Tree

- A red-black tree is a binary search tree
- Thus, search and traversal
  - Use algorithms for binary search tree
  - Simply ignore color of pointers

Adding to and Removing from a Red-Black Tree

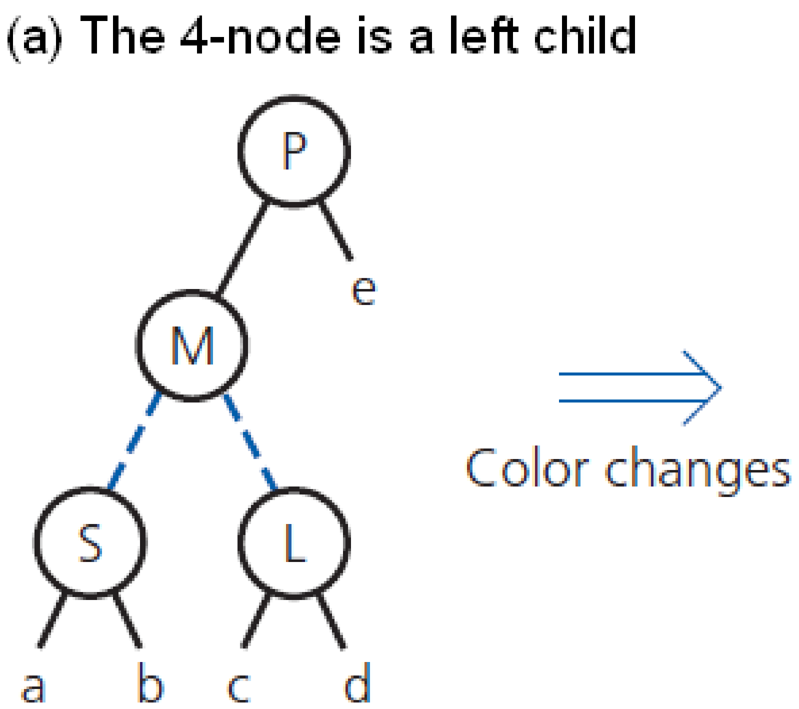
- Red-black tree represents a 2-3-4 tree
  - Simply adjust 2-3-4 addition algorithms
  - Accommodate red-black representation
- Splitting equivalent of a 4-node requires simple color changes
  - Pointer changes called rotations result in a shorter tree

Adding to and Removing from a Red-Black Tree



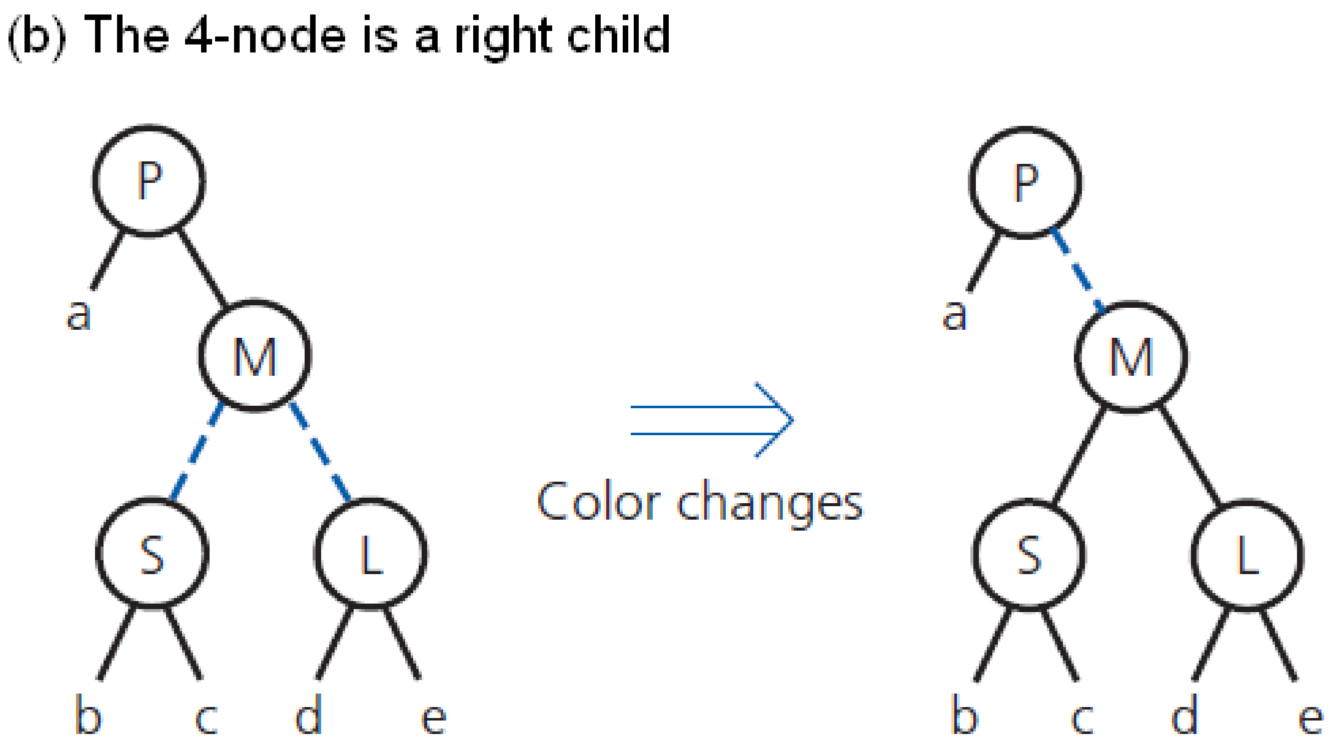
- Splitting a red-black representation of a 4-node root

Adding to and Removing from a Red-Black Tree



- Splitting a red-black representation of a 4-node whose parent is a 2-node

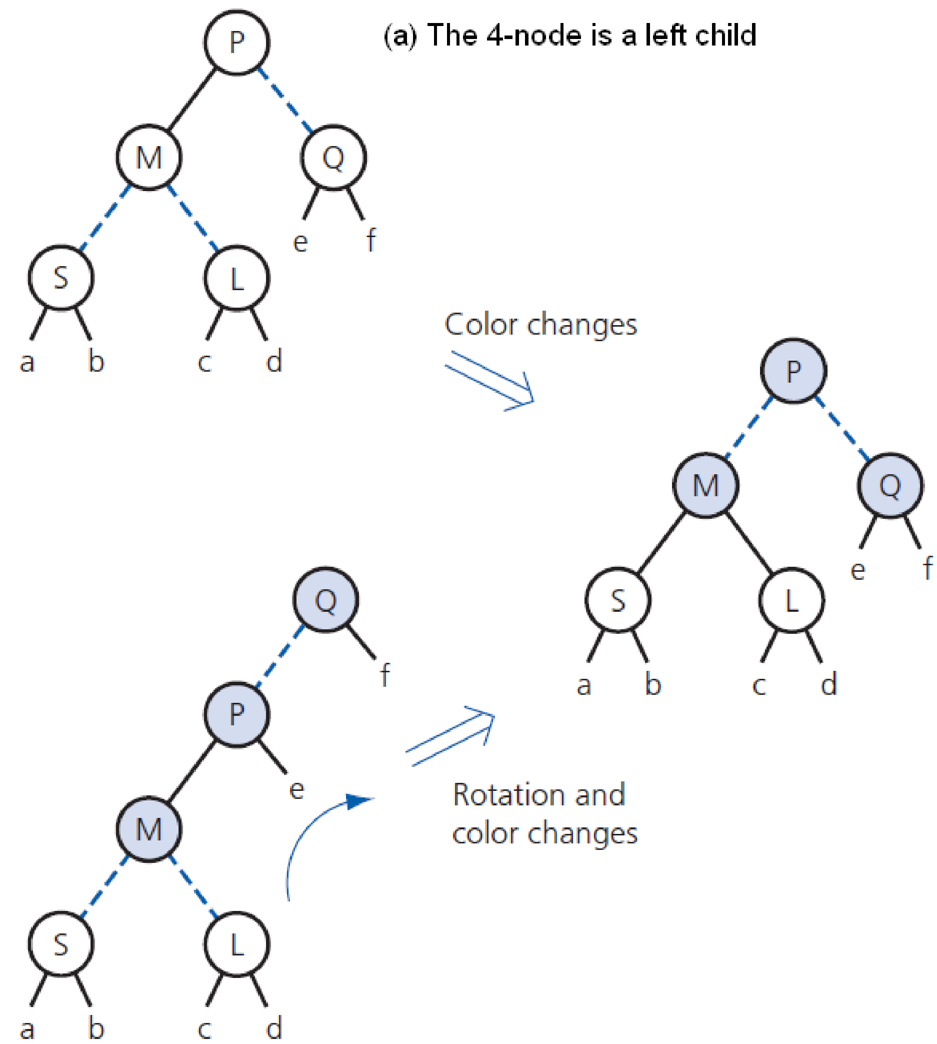
Adding to and Removing from a Red-Black Tree



- Splitting a red-black representation of a 4-node whose parent is a 2-node

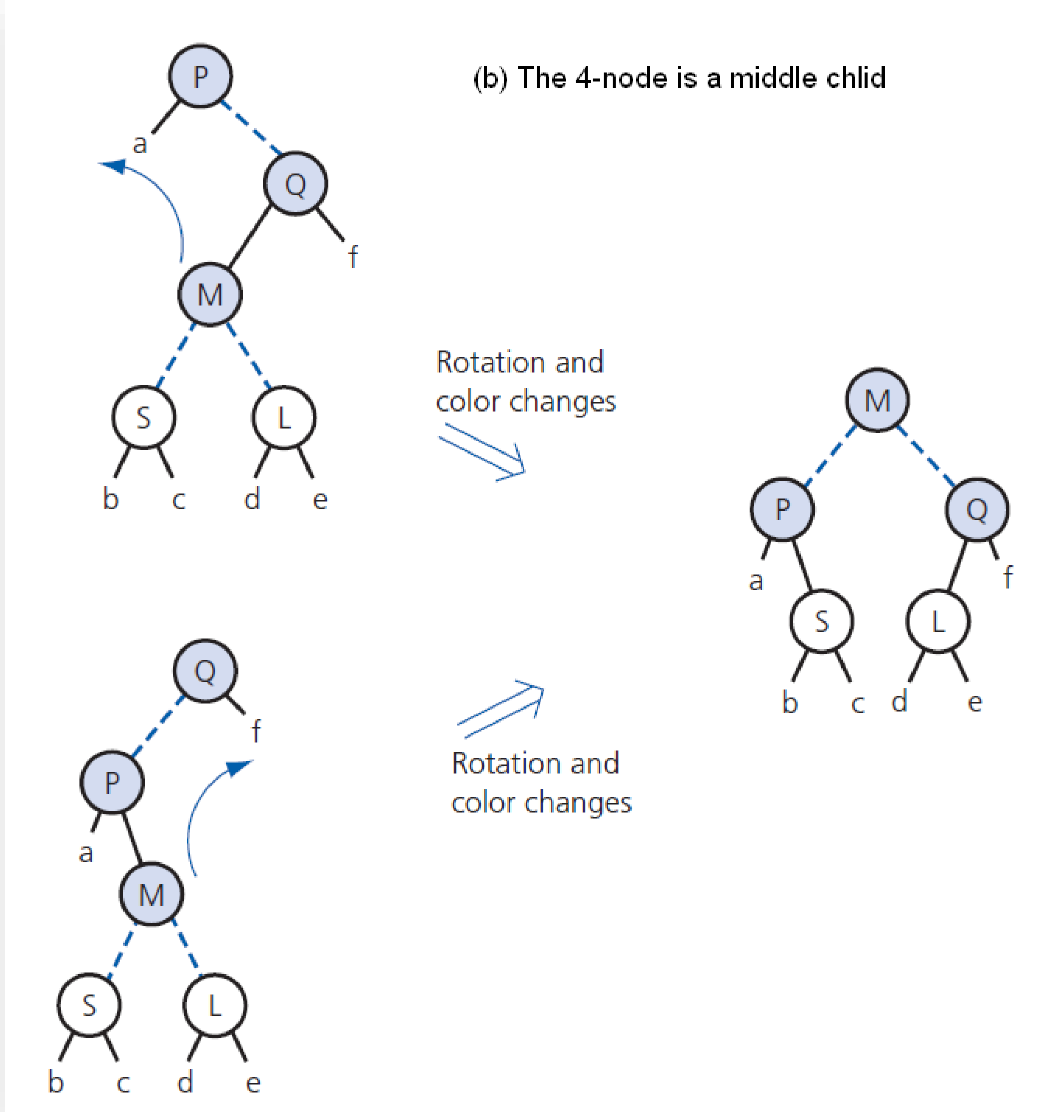
Adding to and Removing from a Red-Black Tree

- Splitting a red-black representation of a 4-node whose parent is a 3-node



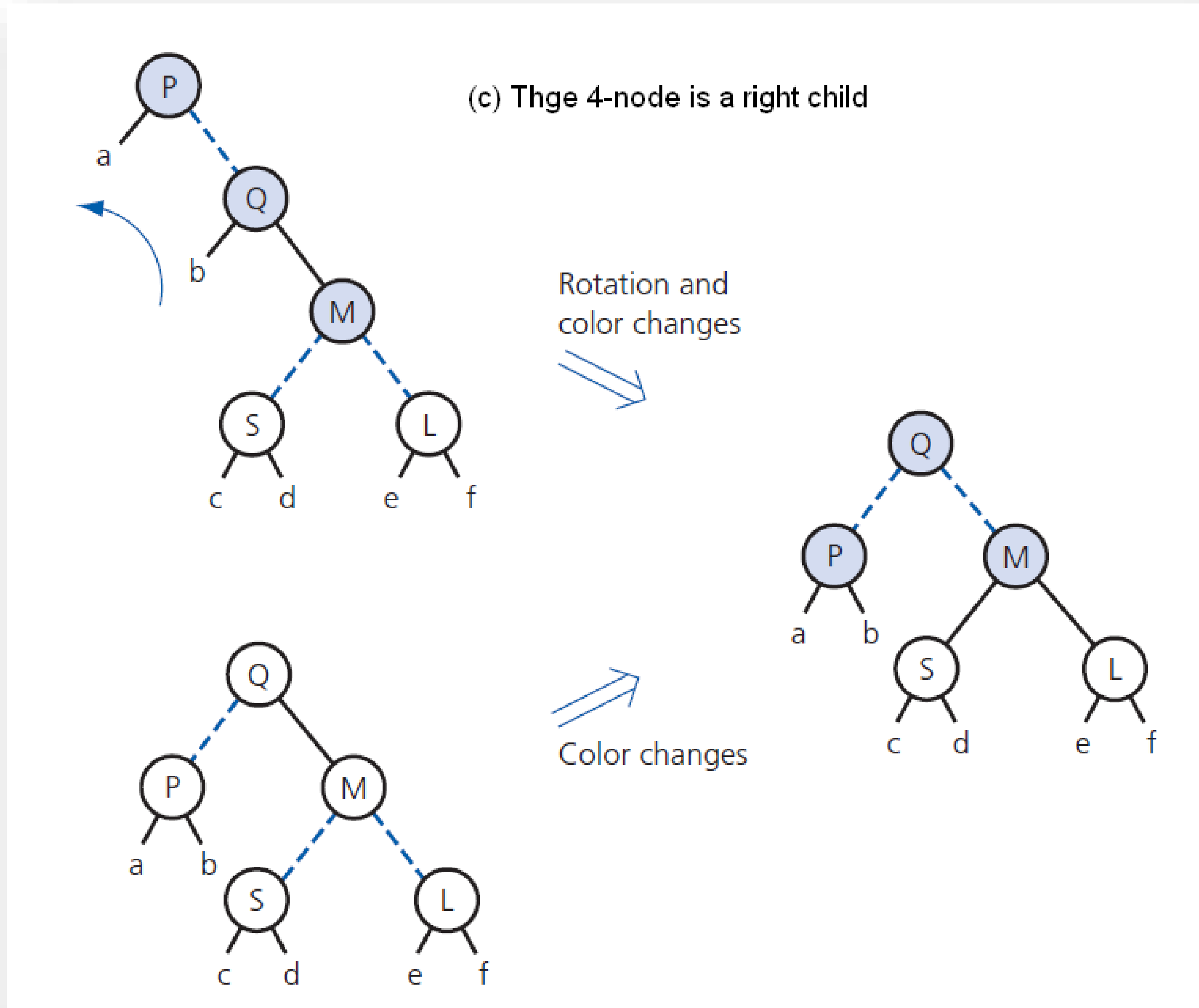
Adding to and Removing from a Red-Black Tree

- Splitting a red-black representation of a 4-node whose parent is a 3-node



Adding to and Removing from a Red-Black Tree

- Splitting a red-black representation of a 4-node whose parent is a 3-node

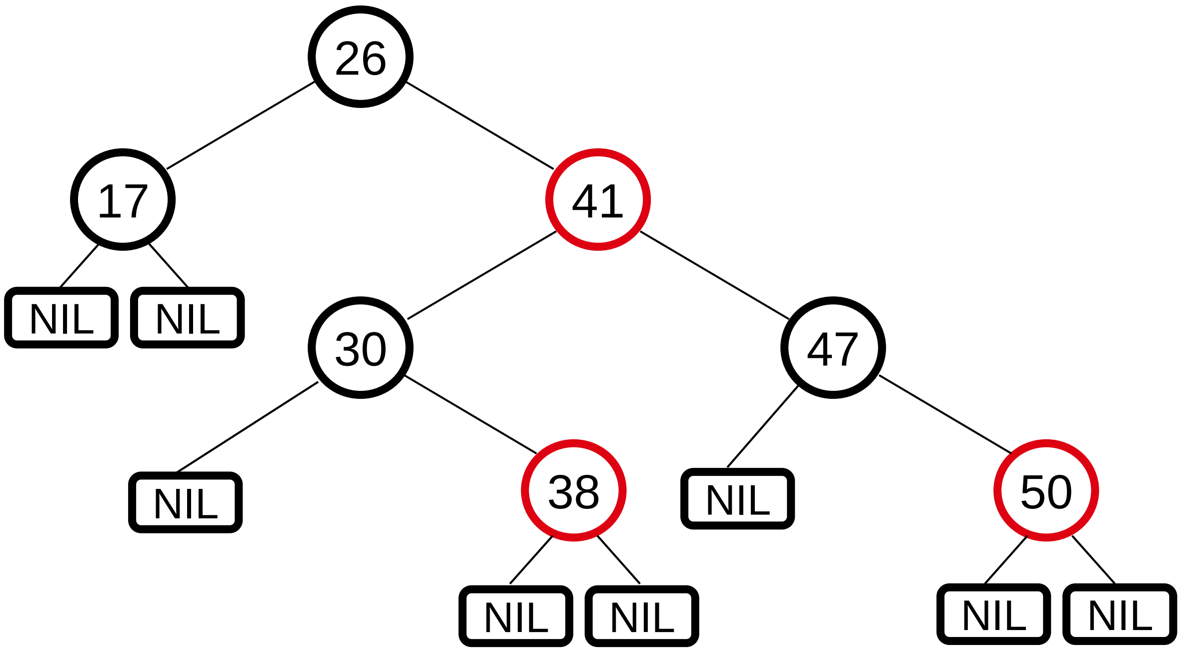


Red-Black-Trees Properties

(**Binary search tree property is satisfied**)

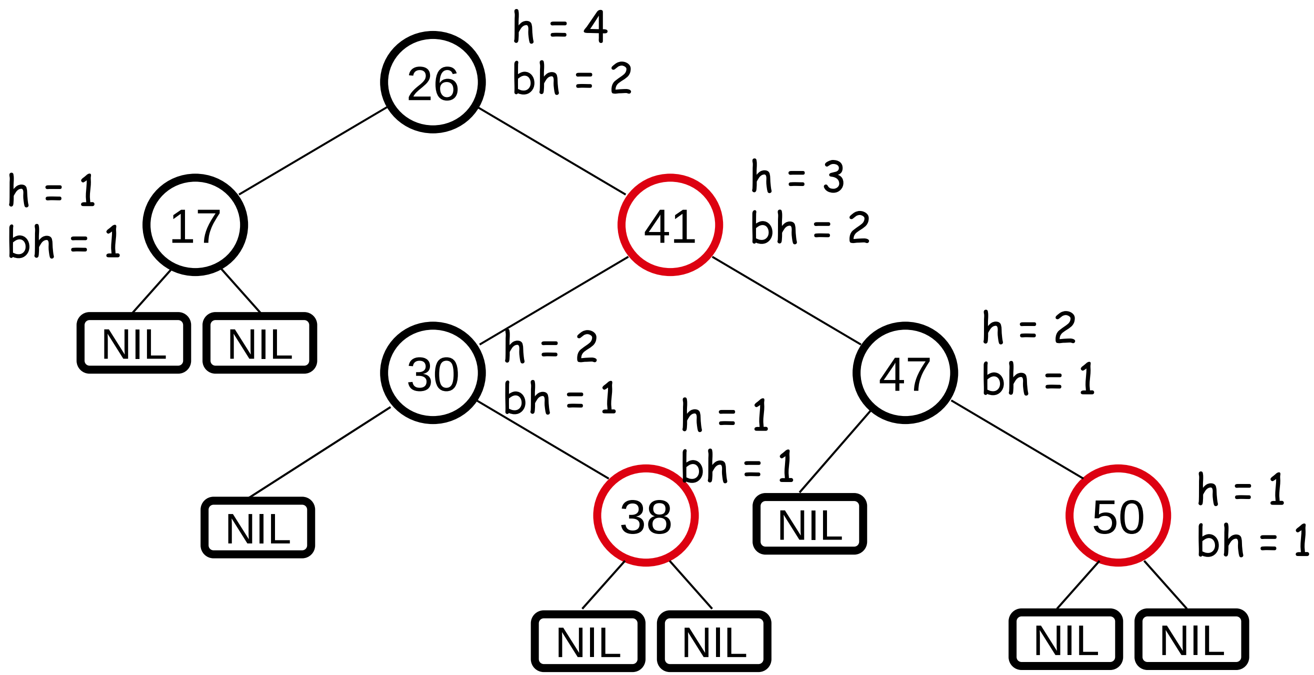
1. Every node is either red or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is red, then both its children are **black**
    - No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from that node to a leaf contain the same number of **black** nodes

Example: RED-BLACK-TREE



- For convenience, we add NIL nodes and refer to them as the leaves of the tree.
  - Color[NIL] = BLACK

Definitions

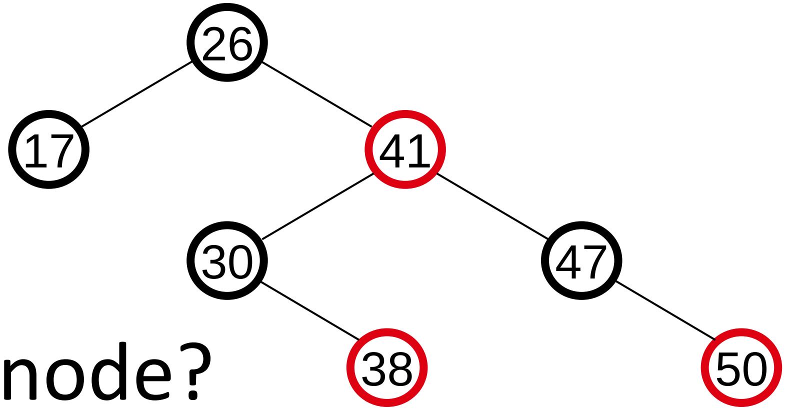


- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** bh(x) of a node x: the number of black nodes (including NIL) on the path from x to a leaf, not counting x

Height of Red-Black-Trees

A red-black tree with n internal nodes has height at most $2log(N+1)$

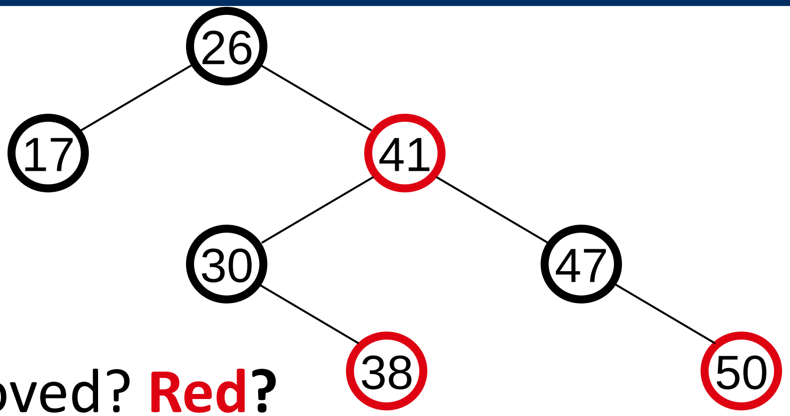
Insert Item



What color to make the new node?

- Red?
  - Let's insert 35!
    - Property 4 is violated: if a node is red, then both children are black
- Black?
  - Let's insert 14!
    - Property 5 is violated: all paths from a node to its leaves contain the same number of black nodes

Delete Item

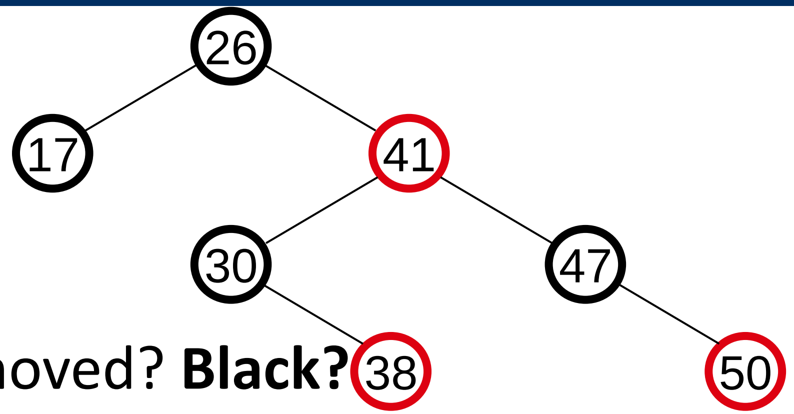


What color was the node that was removed? **Red?**

1. Every node is either **red** or **black** OK!
2. The root is **black** OK!
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black OK!

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes OK!

Delete Item



What color was the node that was removed? **Black?**

1. Every node is either **red** or **black** OK!
2. The root is **black** Not OK! If removing the root and the child that replaces it is **red**
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

Not OK! Could change the black heights of some nodes

Not OK! Could create two red nodes in a row

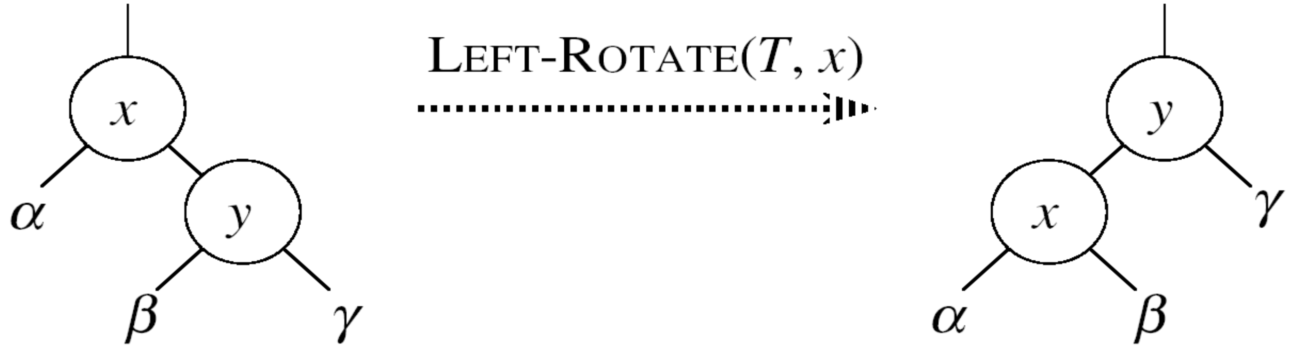
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for re-structuring the tree after insert and delete operations
  - Together with some node <u>re-coloring,</u> they help restore the red-black-tree property
  - Change some of the pointer structure
  - **Preserve** the binary-search tree property
- Two types of rotations:
  - Left & right rotations

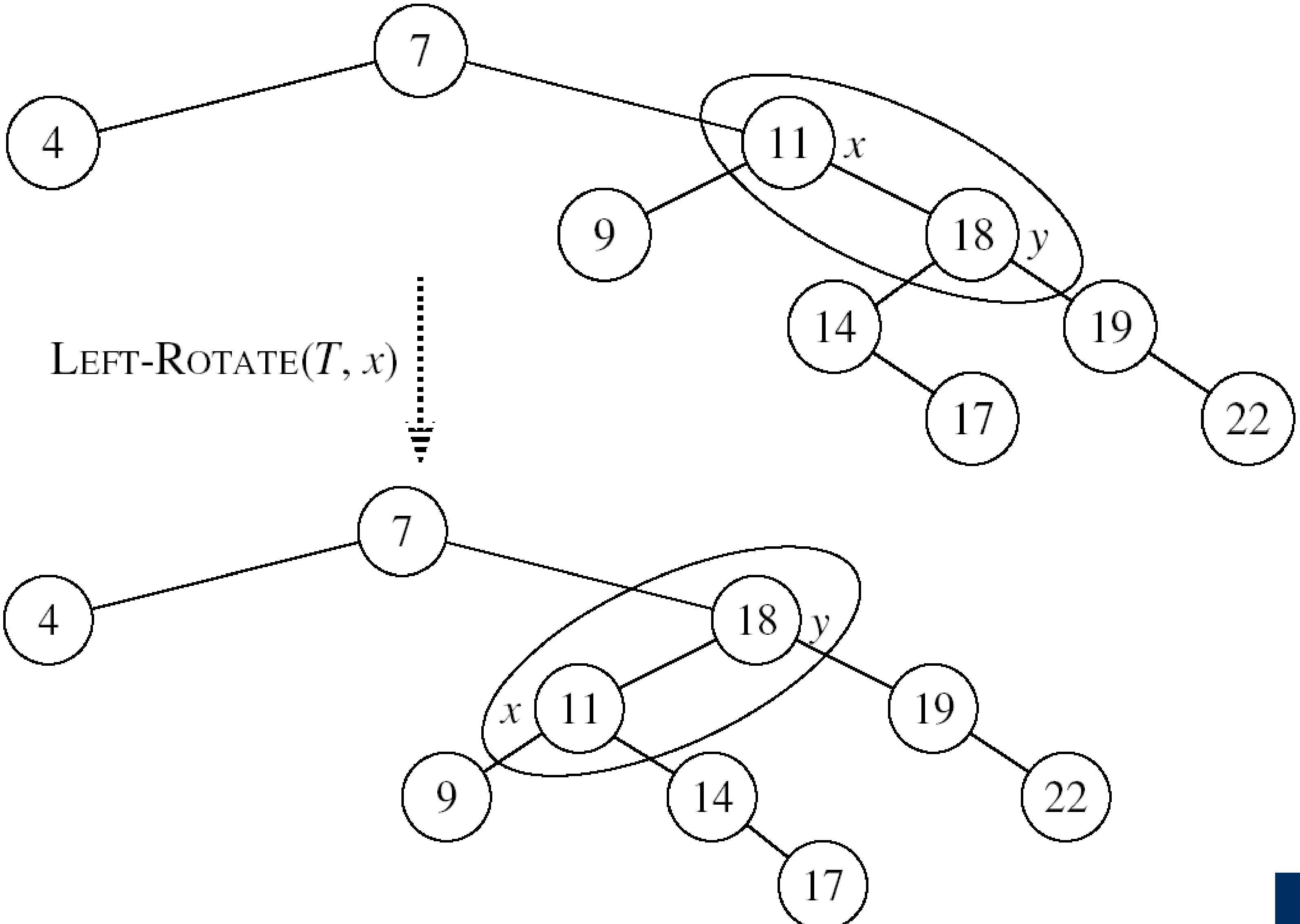
Left Rotations

- Assumptions for a left rotation on a node x:
  - The right child y of x is not NIL



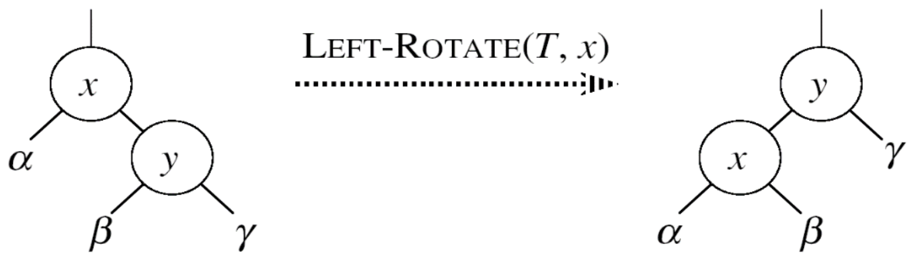
- Idea:
  - Pivots around the link from x to y
  - Makes y the new root of the subtree
  - x becomes y's left child
  - y's left child becomes x's right child

Example: LEFT-ROTATE



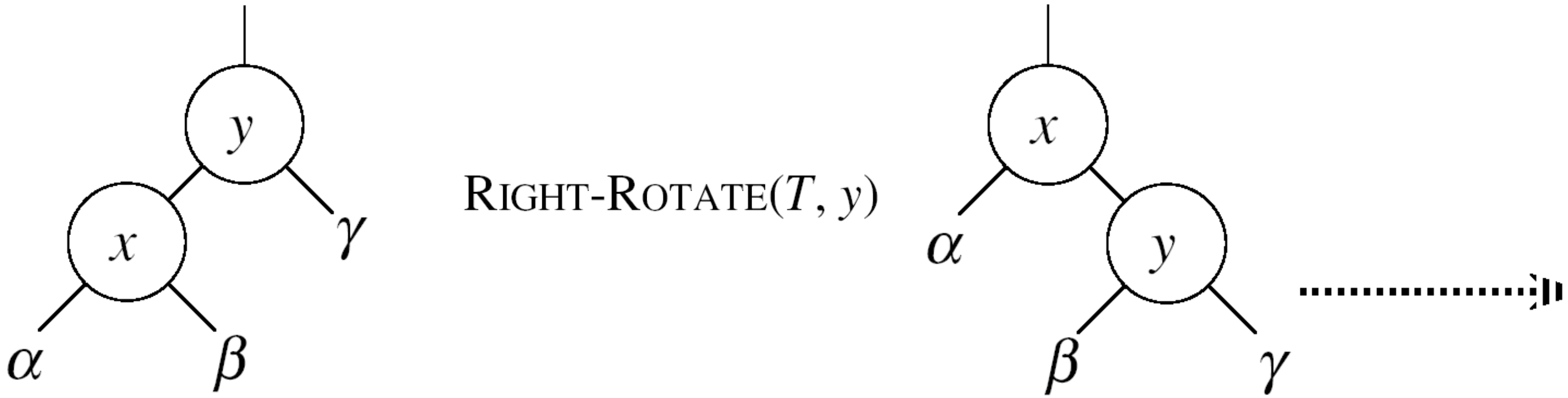
LEFT-ROTATE(T, x)

```
y ← right[x]                       ► Set y
right[x] ← left[y]                 ► y' s left subtree becomes x' s right subtree
if left[y] ≠ NIL
   then p[left[y]] ← x             ► Set the parent relation from left[y] to x
p[y] ← p[x]                        ► The parent of x becomes the parent of y
if p[x] = NIL
   then root[T] ← y
   else if x = left[p[x]]
           then left[p[x]] ← y
           else right[p[x]] ← y
left[y] ← x                        ► Put x on y' s left
p[x] ← y                           ► y becomes x' s parent
```



Right Rotations

- Assumptions for a right rotation on a node x:
  - The left child x of y is not NIL



- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Insert Item

- Goal:
  - Insert a new node z into a red-black tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black tree properties

RB-INSERT(T, z)

1. $y \leftarrow NIL$
2. $x \leftarrow root[T]$

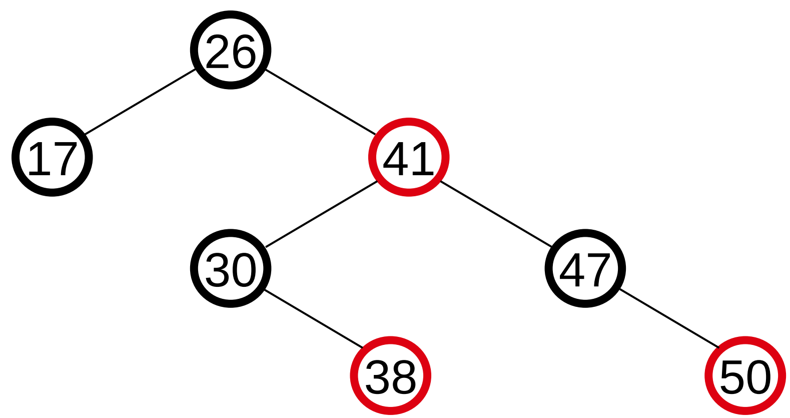
- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

3. **while** $x \neq NIL$
4. **do** $y \leftarrow x$
5. **if** $key[z] < key[x]$
6. **then** $x \leftarrow left[x]$
7. **else** $x \leftarrow right[x]$

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

8. $p[z] \leftarrow y$

- Sets the parent of z to be y



RB-INSERT(T, z)

9. **if** y = NIL
10. **then** root[T] ← z

The tree was empty: set the new node to be the root

11. **else if** key[z] < key[y]
12. **then** left[y] ← z
13. **else** right[y] ← z

Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y' s key

14. left[z] ← NIL
15. right[z] ← NIL
16. color[z] ← RED

Set the fields of the newly added node

17. RB-INSERT-FIXUP(T, z)

Fix any inconsistencies that could have been introduced by adding this new red node

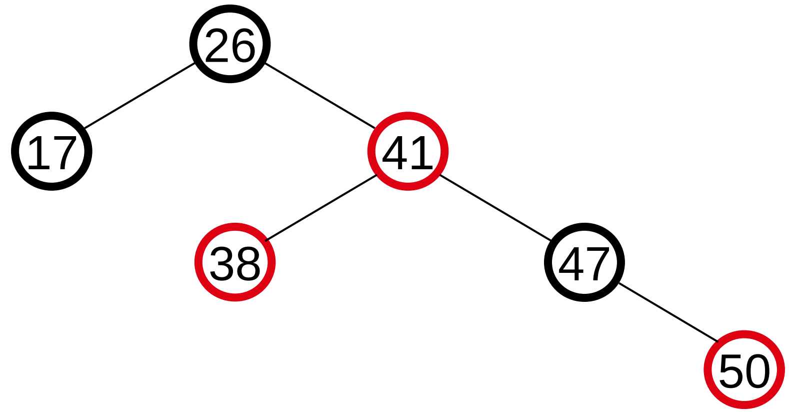
RB Properties Affected by Insert

1. Every node is either **red** or **black** — OK!
2. The root is **black** — If z is the root $\Rightarrow$ not OK
3. Every leaf (NIL) is **black** — OK!
4. If a node is red, then both its children are black

If p(z) is red $\Rightarrow$ not OK
z and p(z) are both red

OK!

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes



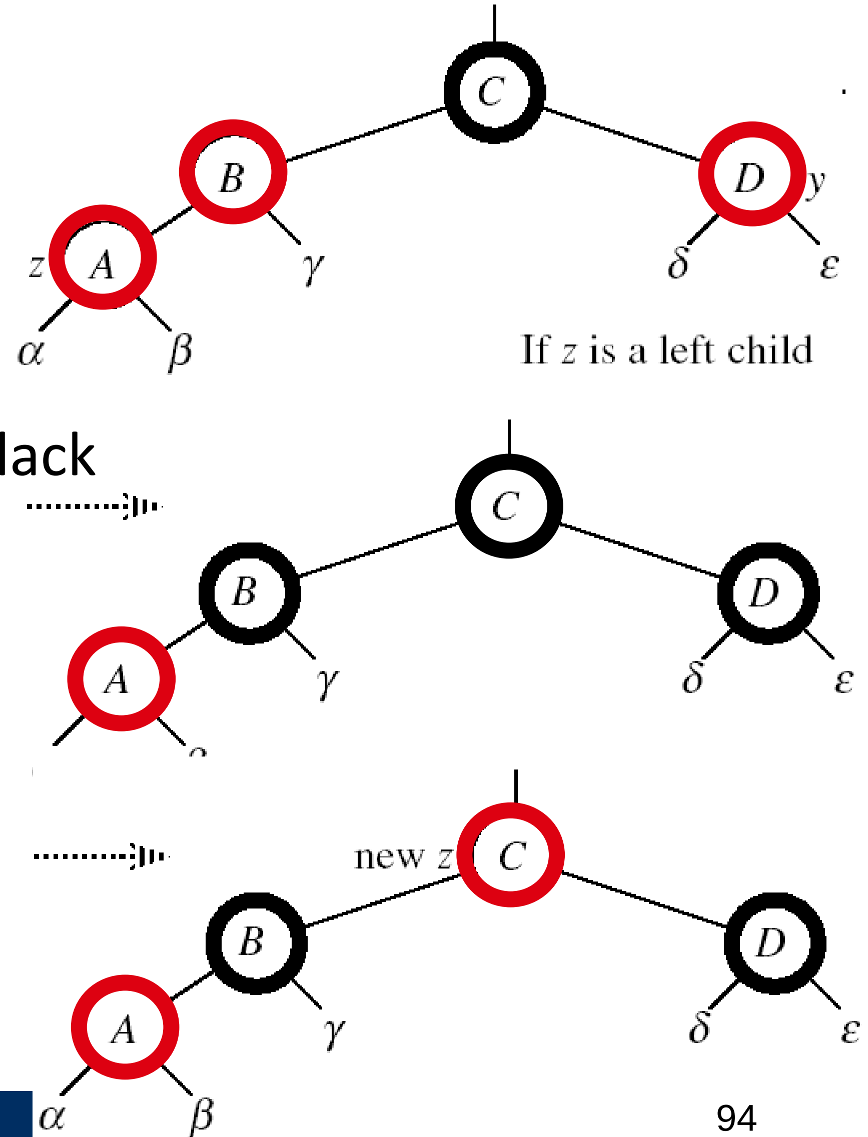
RB-INSERT-FIXUP

<u>Case 1:</u> z' s "uncle" (y) is **red**

(z could be either left or right child)

Idea:

- p[p[z]] (z' s grandparent) must be black
- color p[z] $\leftarrow$ **black**
- color y $\leftarrow$ **black**
- color p[p[z]] $\leftarrow$ **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree



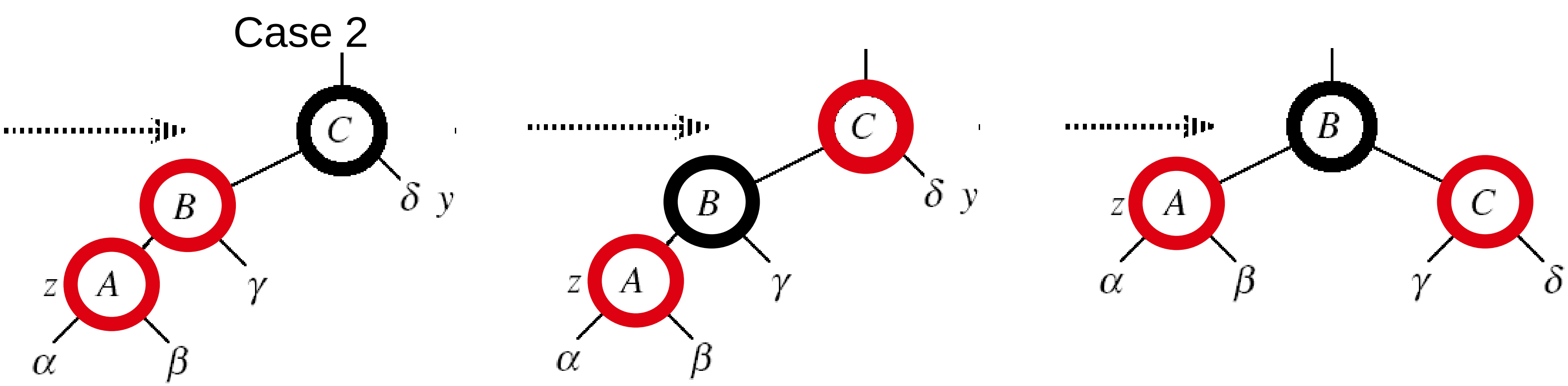
RB-INSERT-FIXUP

Case 2:

- z's "uncle" (y) is **black**
- z is a left child

Idea:

- color $p[z] \leftarrow$ **black**
- color $p[p[z]] \leftarrow$ **red**
- RIGHT-ROTATE(T, $p[p[z]]$)
- No longer have 2 reds in a row
- $p[z]$ is now black



RB-INSERT-FIXUP

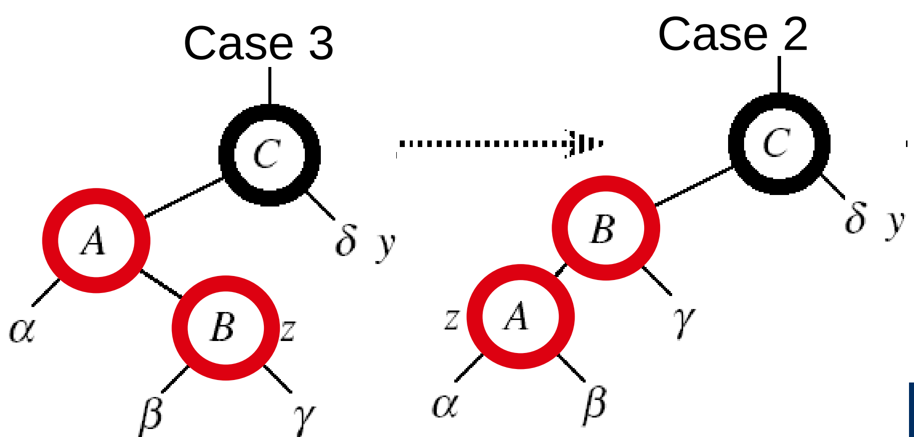
Case 3:

- z's "uncle" (y) is **black**
- z is a right child

Idea:

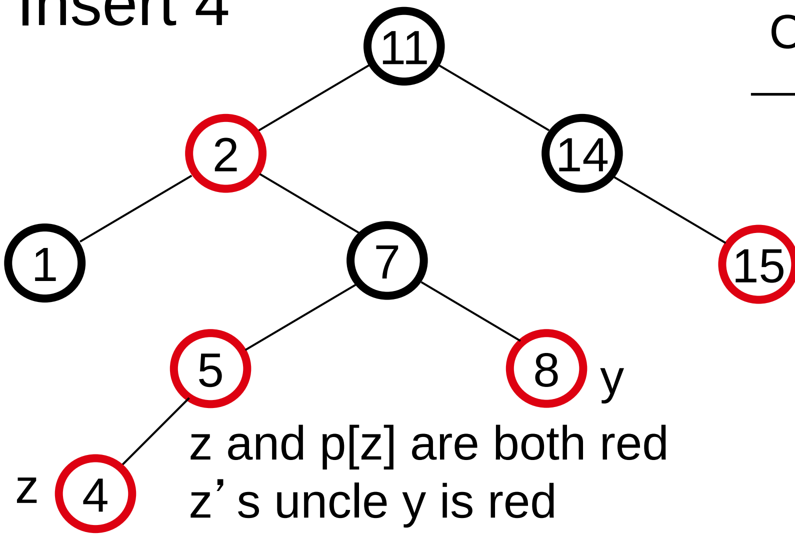
- z $\leftarrow$p[z]
- LEFT-ROTATE(T, z)

$\Rightarrow$ now z is a left child, and both z and p[z] are red $\Rightarrow$ case 2



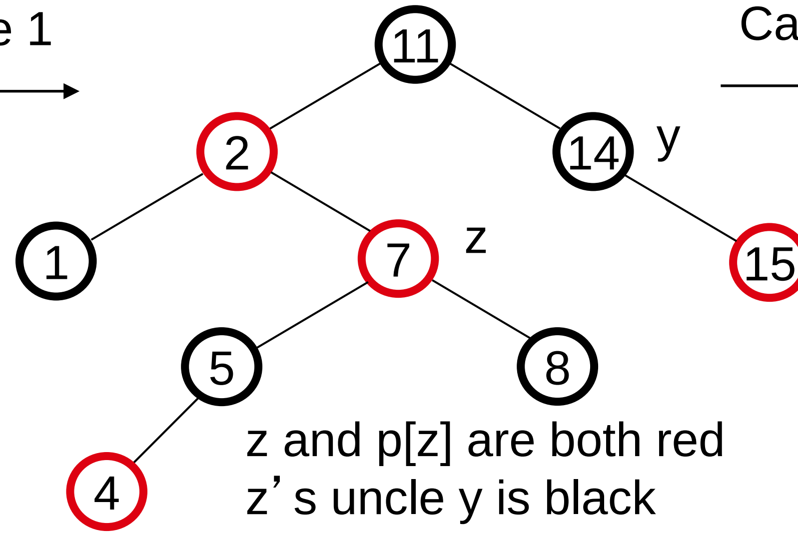
Example

Insert 4



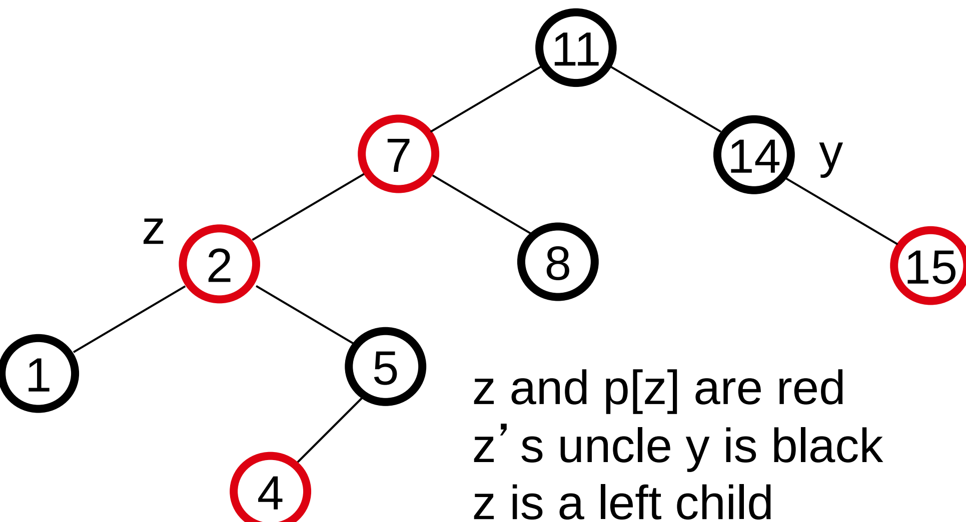
z and p[z] are both red
z’s uncle y is red

Case 1 →



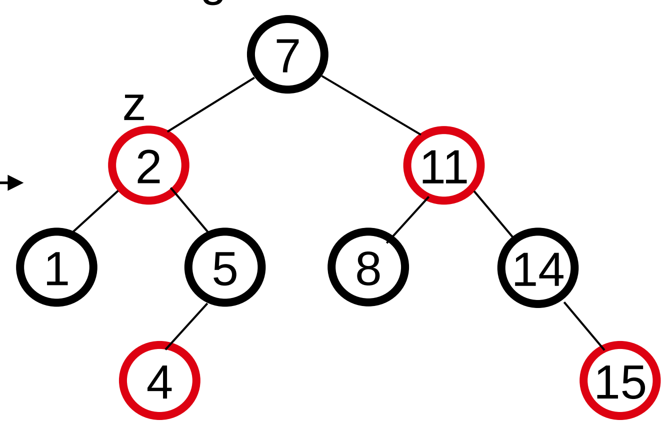
z and p[z] are both red
z’s uncle y is black
z is a right child

Case 3 →



z and p[z] are red
z’s uncle y is black
z is a left child

Case 2 →



RB-INSERT-FIXUP(T, z)

1. **while** color[p[z]] = RED — The while loop repeats only when case1 is executed: $O(\log N)$ times
2. **if** p[z] = left[p[p[z]]]
3. **then** y ← right[p[p[z]]] — Set the value of x' s "uncle"
4. **if** color[y] = RED
5. **then** ***Case1***
6. **else if** z = right[p[z]]
7. **then** ***Case3***
8. ***Case2***
9. **else** (same as **then** clause with "right" and "left" exchanged for lines 3-4)
10. color[root[T]] ← BLACK — We just inserted the root, or The red violation reached the root

Analysis of InsertItem

- Inserting the new element into the tree $O(\log N)$

- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is $O(\log N)$

- Total running time of Insert Item: $O(\log N)$

Delete Item

- Delete as usually, then re-color/rotate

- A bit more complicated though …

- Demo
  - http://gauss.ececs.uc.edu/RedBlack/redblack.html

Discussion Problems

- What is the ratio between the longest path and the shortest path in a red-black tree?

  - The shortest path is at least bh(root)

  - The longest path is equal to h(root)

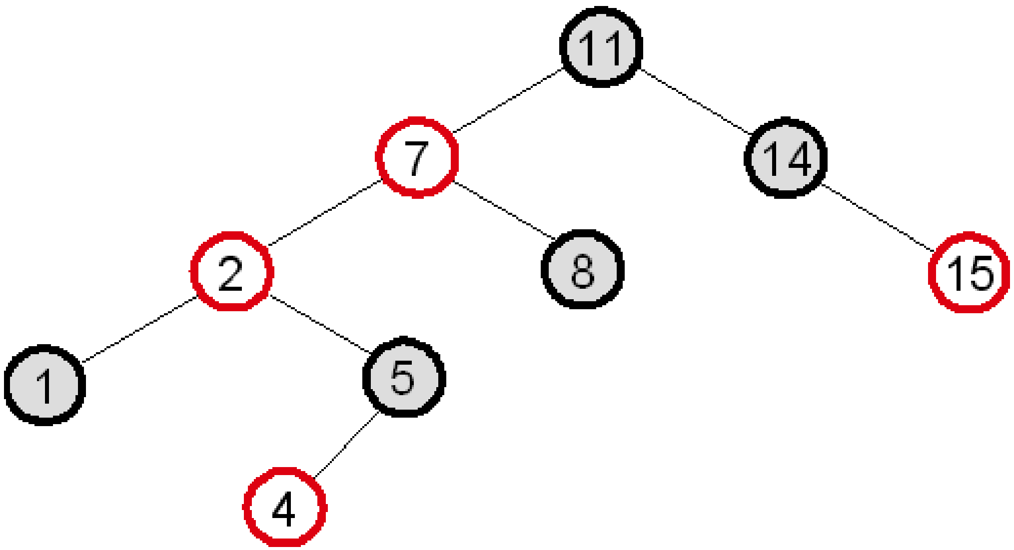
  - From Claim 1, bh(root) $\geq$ h(root)/2

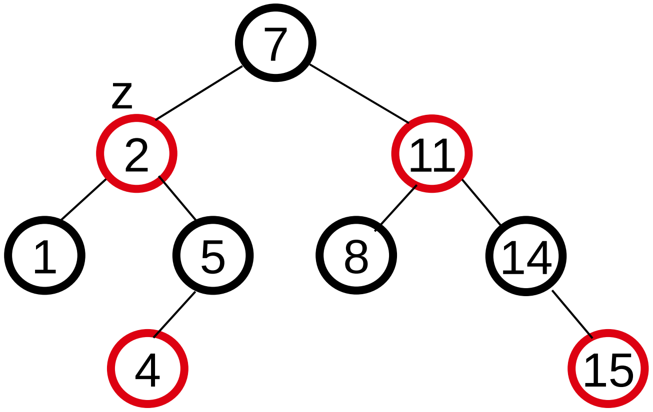
    or h(root) $\leq$2 bh(root)

  - Therefore, the ratio is ≤ 2

Discussion Problems

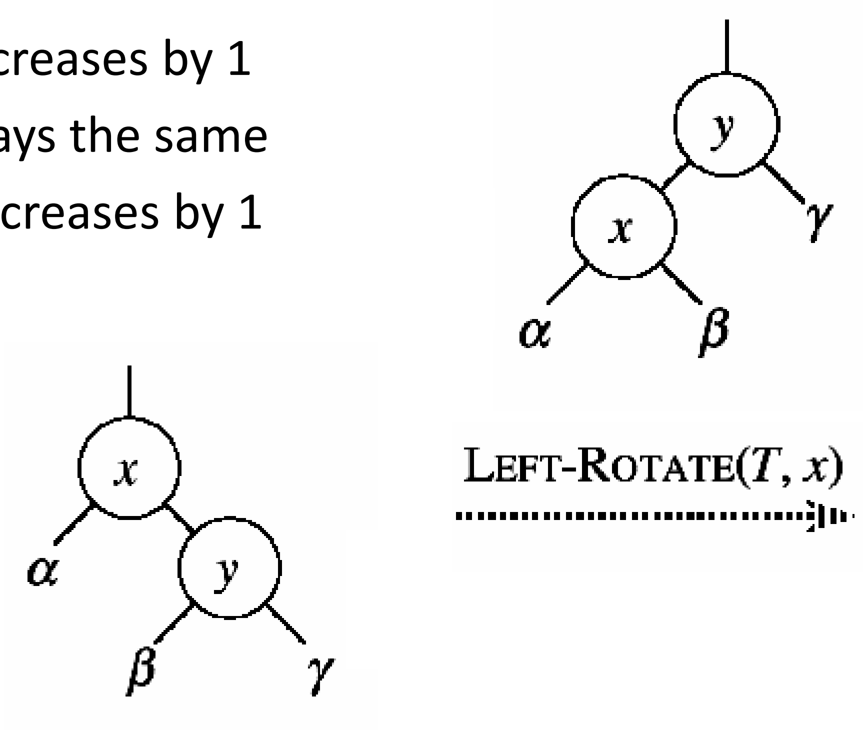
- What red-black tree property is violated in the tree below? How would you restore the red-black tree property in this case?
  - **Property violated**: if a node is red, both its children are black
  - **Fixup**: color 7 black, 11 red, then right-rotate around 11





Discussion Problems

- Let a, b, c be arbitrary nodes in subtrees α, β, γ in the tree below.
- How do the depths of a, b, c change when a left rotation is performed on node x?
  - a: increases by 1
  - b: stays the same
  - c: decreases by 1



Discussion Problems

- When we insert a node into a red-black tree, we initially set the color of the new node to red.

  Why didn't we choose to set the color to black?

- Would inserting a new node to a red-black tree and then immediately deleting it, change the tree?