Data structures and Algorithms

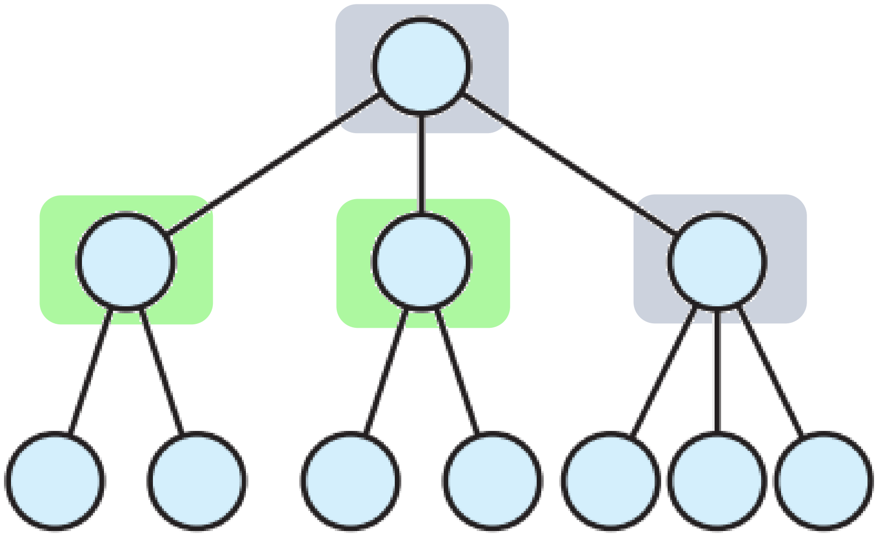
2-3 AND 2-3-4 TREES

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

2-3 Trees

2-3 trees: A definition

- A 2-3 tree has every internal node of either two or three children and all leaves at the same level.



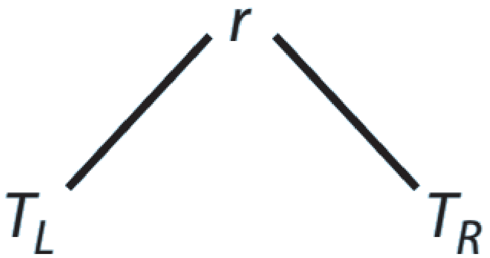
- A node with two children is called a 2-node.
    - The nodes in a binary tree are all 2-nodes.
- A node with three children is called a 3-node.

2-3 trees: A definition

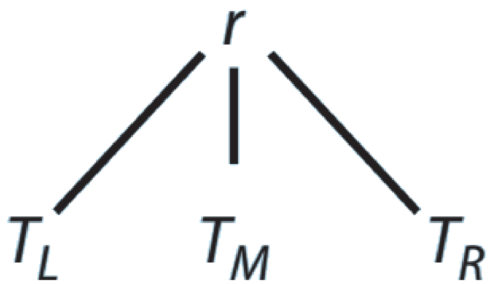
- Not a binary tree, yet resemble a full binary tree
- A 2-3 tree of height h has at least as many nodes as a full binary tree of the same height: at least $2^h - 1$ nodes.
- A 2-3 tree of n nodes has height at most $\lceil \log_2(n+1) \rceil$.

2-3 trees: A definition

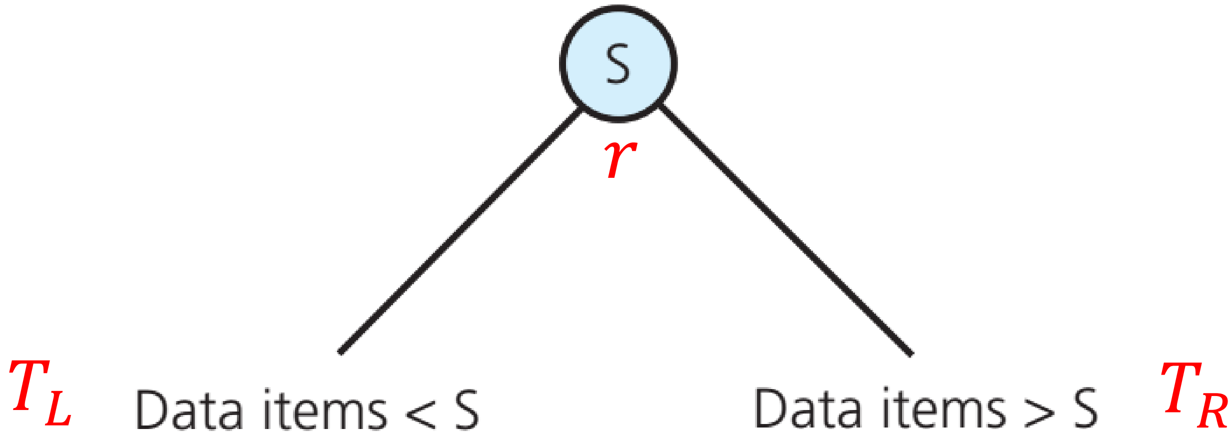
- T is a 2-3 tree of height h if one of the following holds.
- T is empty, in which case h is 0.
- T is of the form



- T is of the form

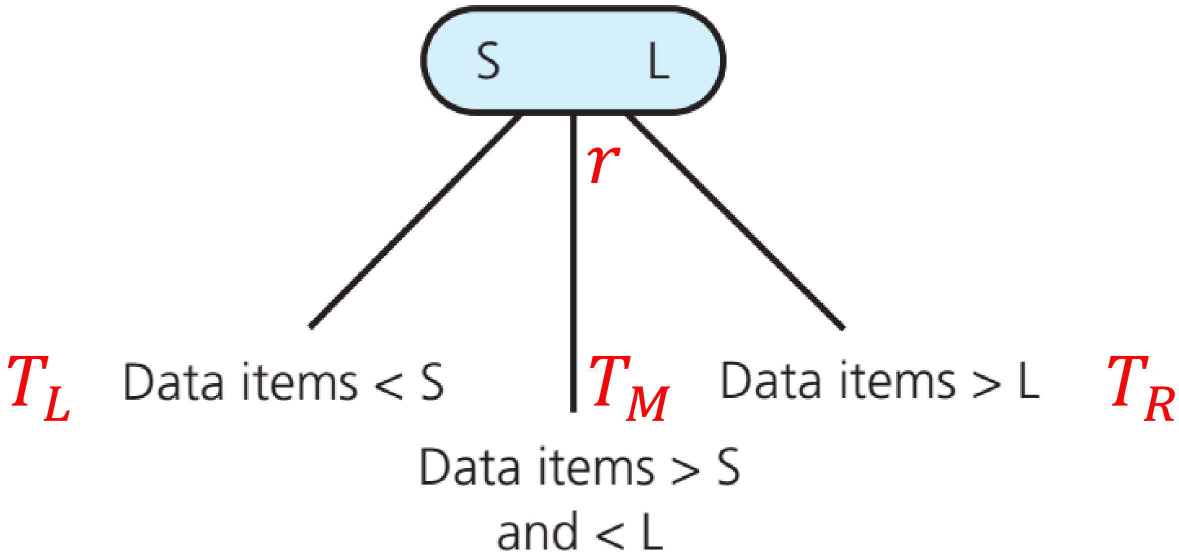


The node contains one data item



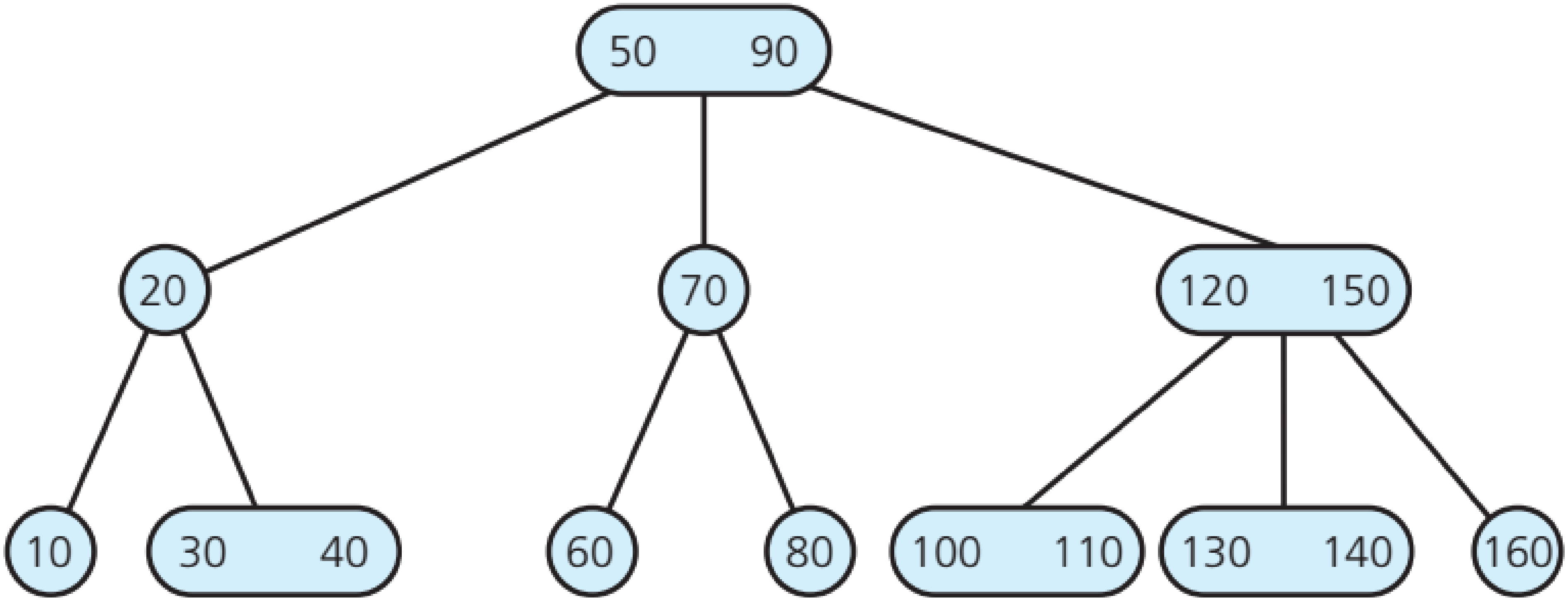
- The node S has one data item, which must be greater than each item in T_L and smaller than each item in T_R.
- T_L and T_R are both 2-3 trees of height $h - 1$.
- A leaf may contain either one or two data items.

The node contains two data items



- The node r contains two ordered data items.
  - The smaller item S must be greater than each item in T_L and smaller than each item in T_M.
  - The larger item L must be greater than each item in T_M and smaller than each item in T_R.
- T_L, T_M and T_R are 2-3 trees of height $h - 1$.

Example: An example of 2-3 tree



2-3 trees: Implementation

```
class TriNode{
  private:
    ItemType smallItem, largeItem;   // Data portion
   // Pointers for the left-child, mid-child and right-child
    TriNode* leftChildPtr, *midChildPtr, * rightChildPtr;

  public:
    bool isTwoNode() const;
    bool isThreeNode() const;
    ItemType getSmallItem() const;
    ItemType getLargeItem() const;
    void setSmallItem(const ItemType& anItem)
    …
} // end TriNode
```

Traversing a 2-3 tree

```
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)
    }
```

Searching a 2-3 tree

- Quite similar to the retrieval operation for a BST

```
// Locate the value target in a nonempty 2-3 tree. Return either
// the 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){ // Item found
        treeItem = the data portion of r
        return treeItem // Success
    }
    else if (r is a leaf)
        throw NotFoundException // Failure
    .........
```

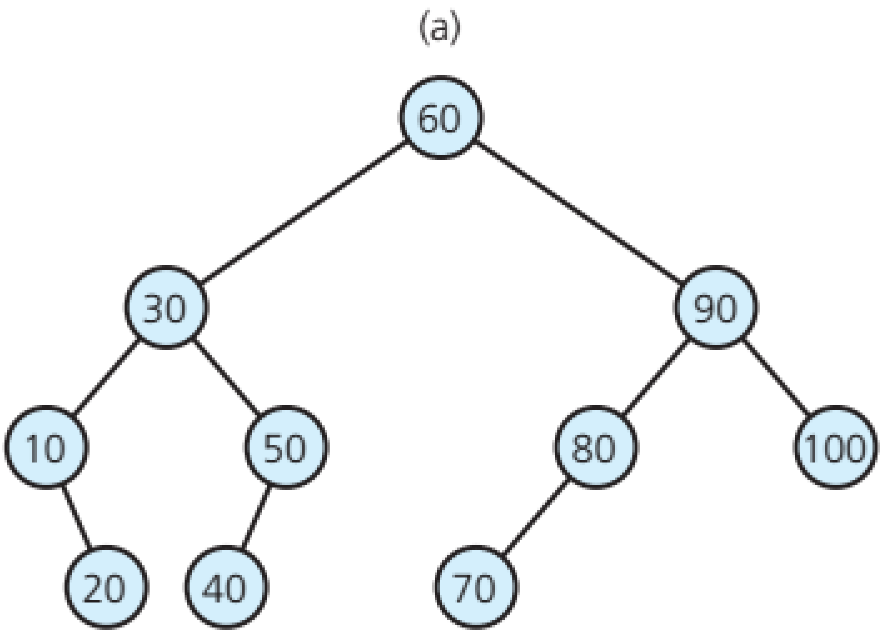
Searching a 2-3 tree

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

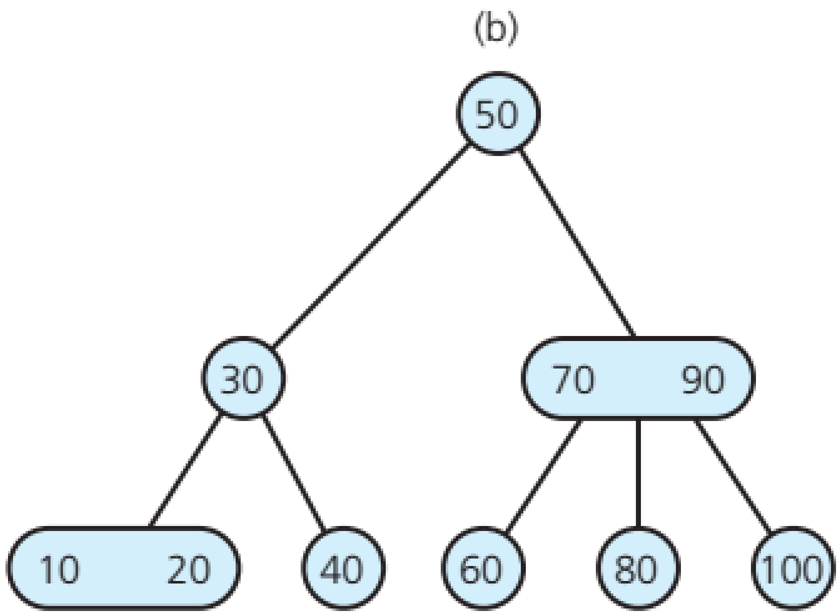
Searching a 2-3 tree

- Searching a 2-3 tree and a balanced (shortest) BST are approximately of the same efficiency.
  - A BST with n nodes is not shorter than $\lceil \log_2(n+1) \rceil$
  - A 2-3 tree with n nodes is not taller than $\lceil \log_2(n+1) \rceil$
  - A node in a 2-3 tree has at most two items
- *Then why should use a 2-3 tree?*
  - Maintaining the shape of a 2-3 tree is relatively simple, while those for a BST is difficult due to insertion and removal operations.

Example: 2-3 tree vs. Balance BST

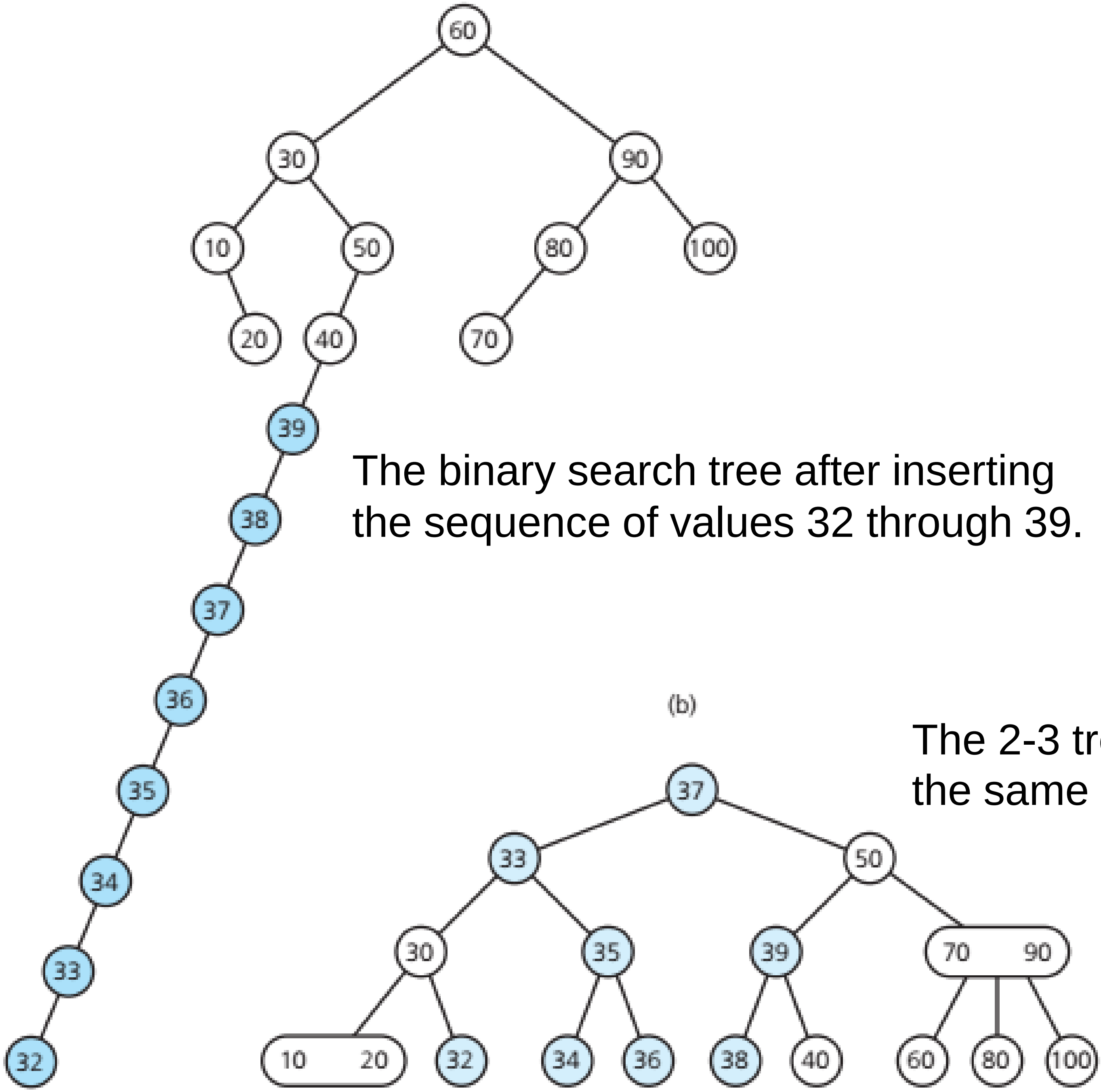


A balanced binary search tree



A 2-3 tree with the same entries

(a)



The binary search tree after inserting the sequence of values 32 through 39.

(b)

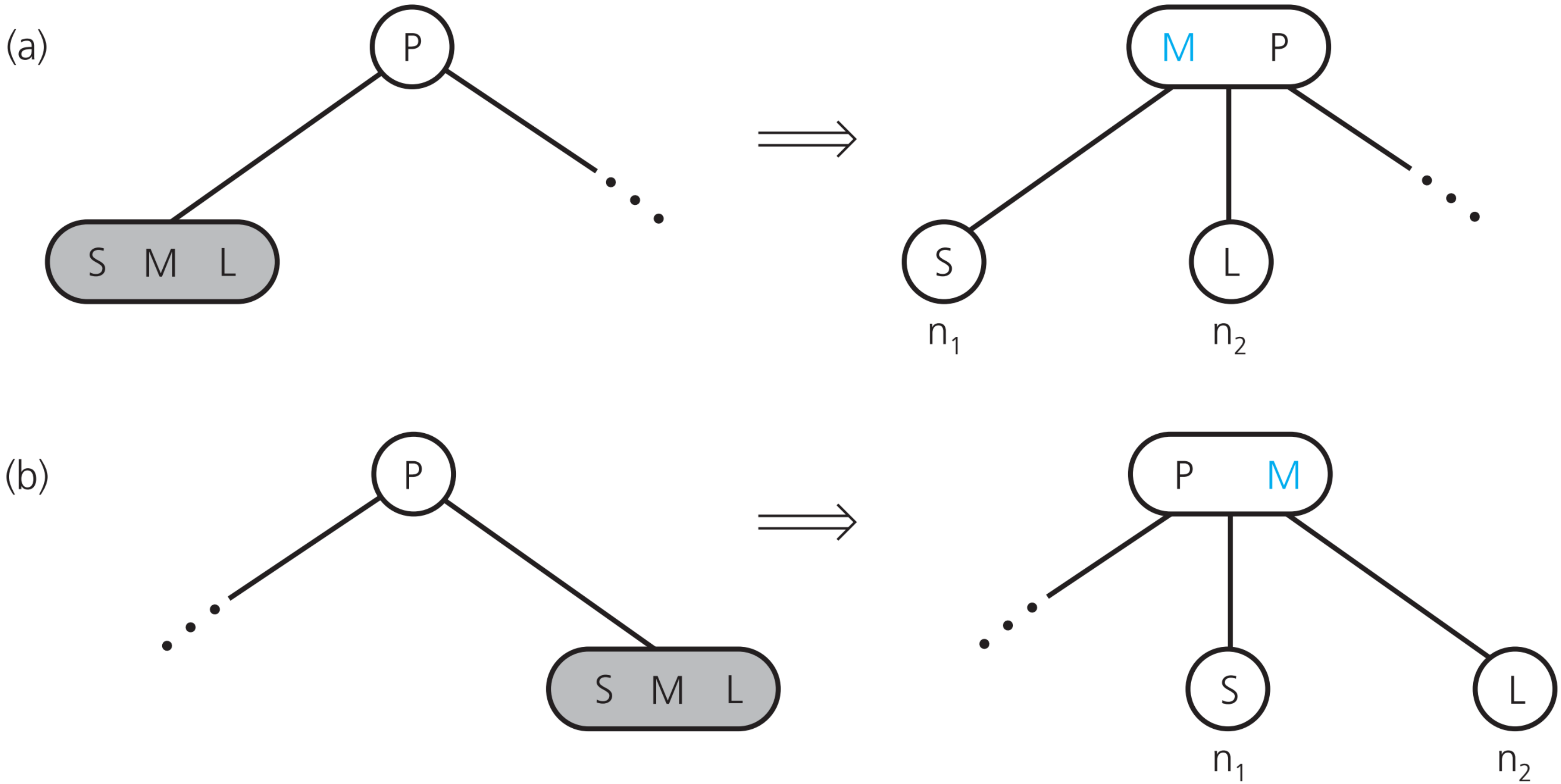
The 2-3 tree after the same insertion

Inserting data into a 2-3 tree

- Locate the leaf node r at which the search for the new item would terminate
- If r contains one items, insert the new item into the leaf.
- If r contains two items, split the leaf node and move the middle-valued item up to its parent.
  - If the parent cannot accommodate the item moving up, further split this internal node.
  - The process continues recursively until reaching a node that had only one item before insertion.

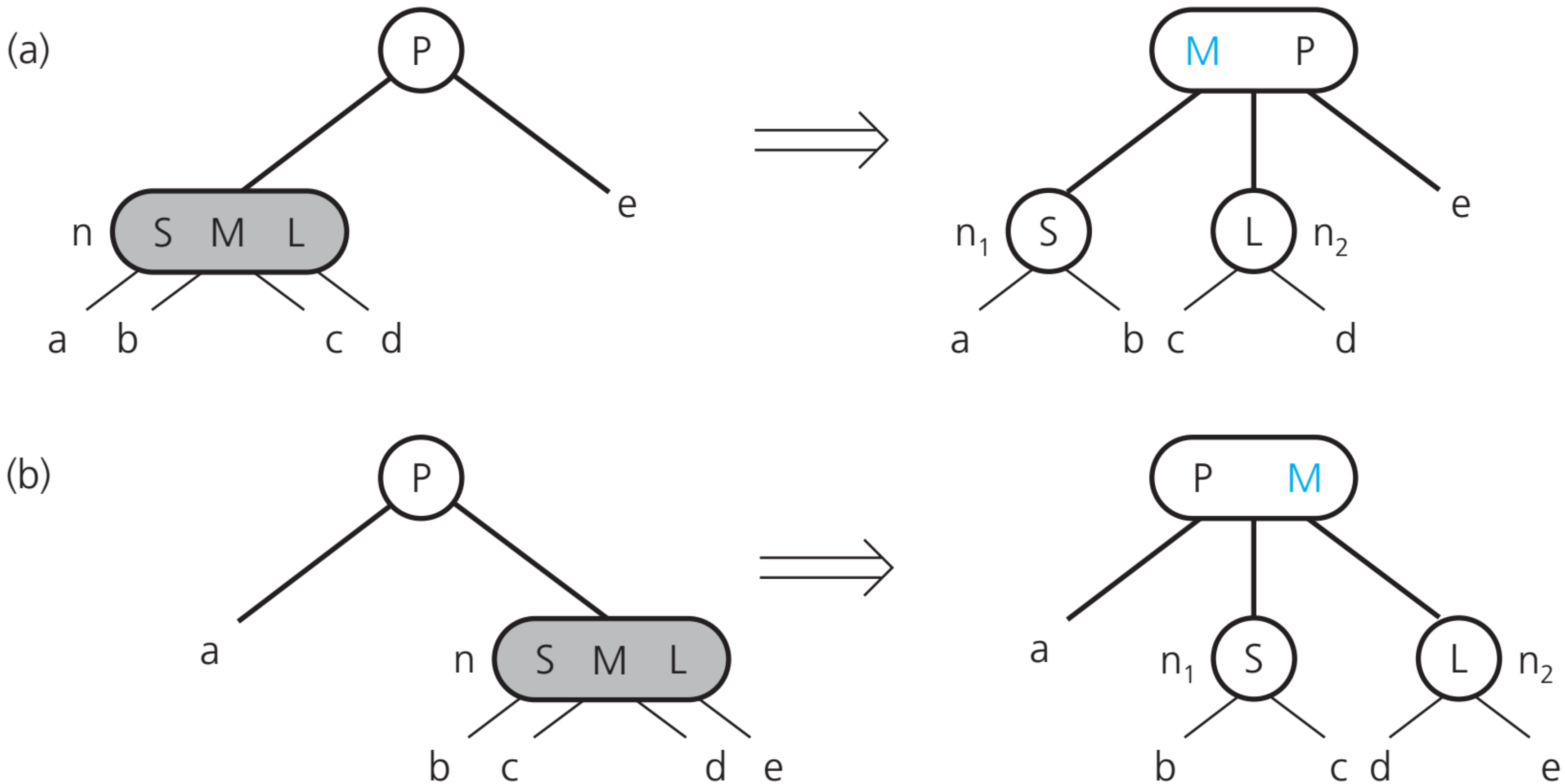
Inserting data into a 2-3 tree

- Splitting a leaf in a 2-3 tree



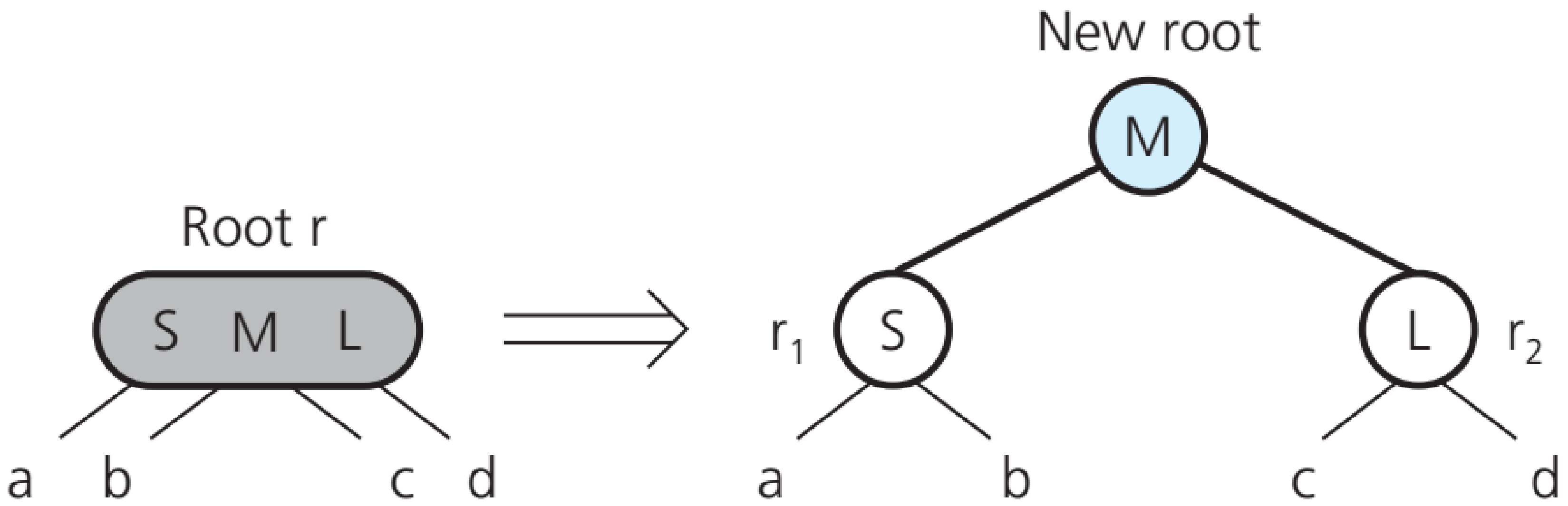
Inserting data into a 2-3 tree

- Splitting an internal node



Inserting data into a 2-3 tree

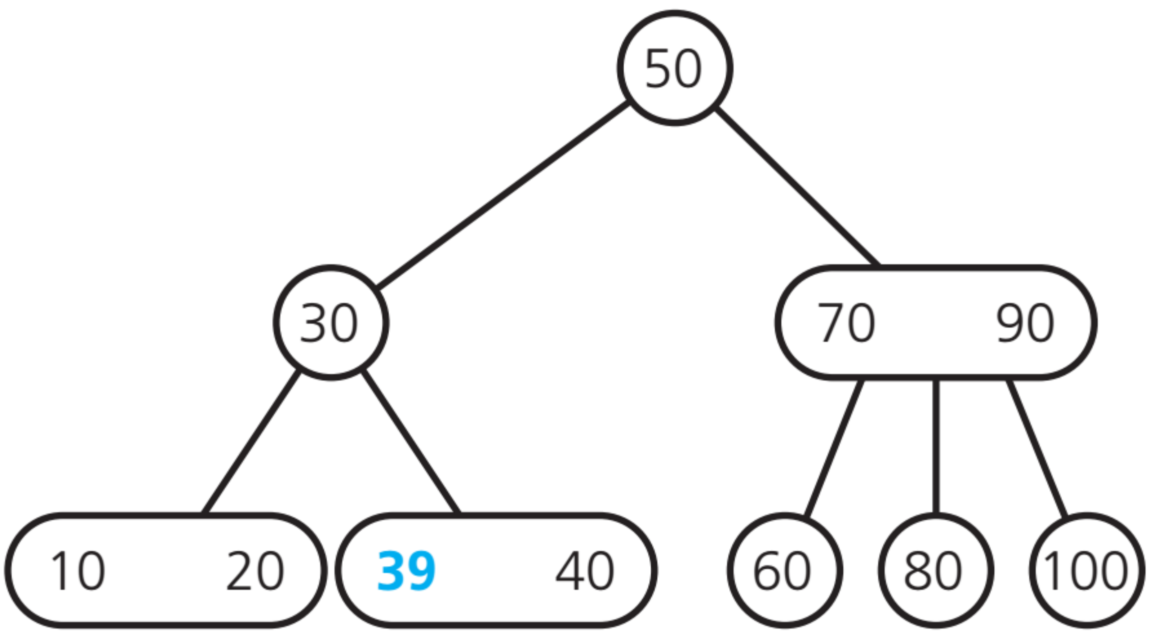
- Splitting a root



- A 2-3 tree postpones the growth of the tree's height much more effectively than a BST.

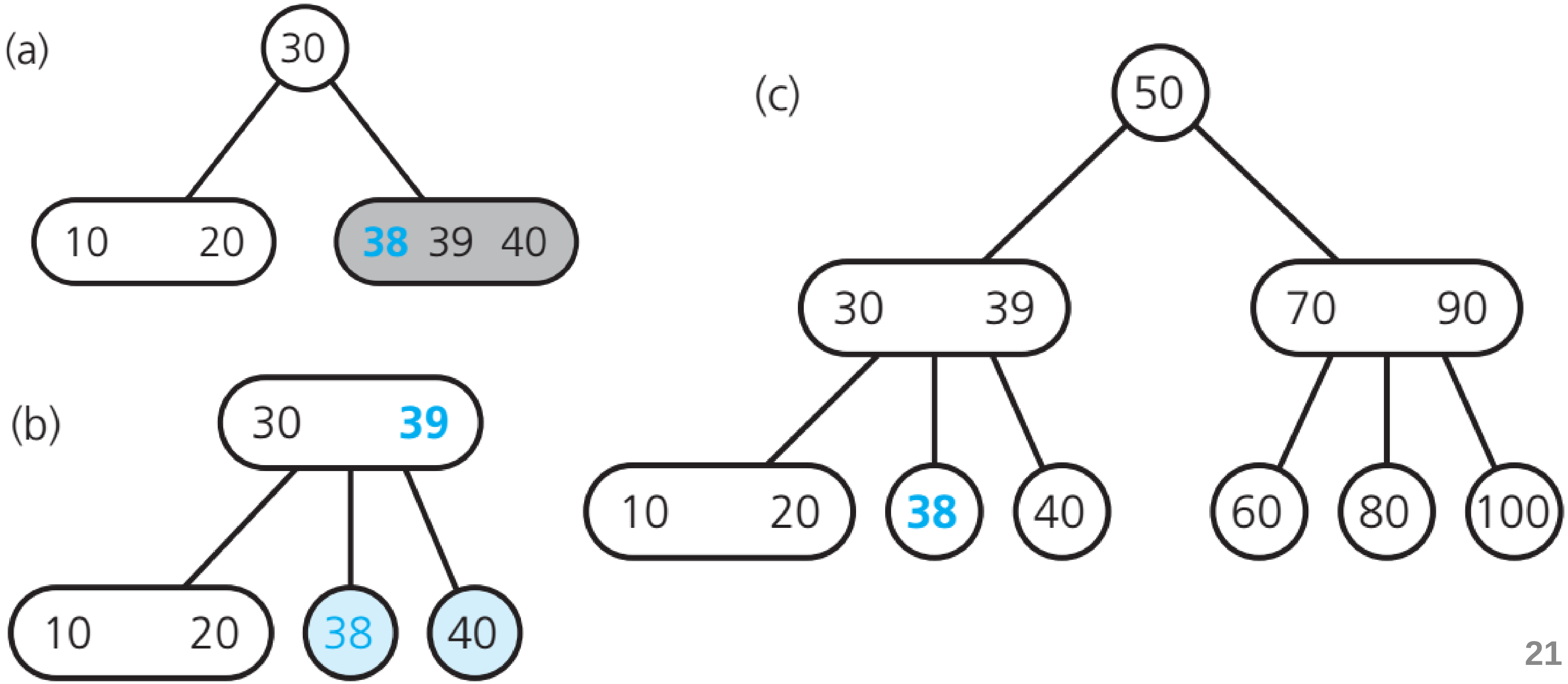
Example: 2-3 tree insersion

- Insert data item 39
  - The `findItem` always terminates at a leaf, i.e., node ⟨40⟩.
  - This node contains 1 item → simply insert the new item into the node



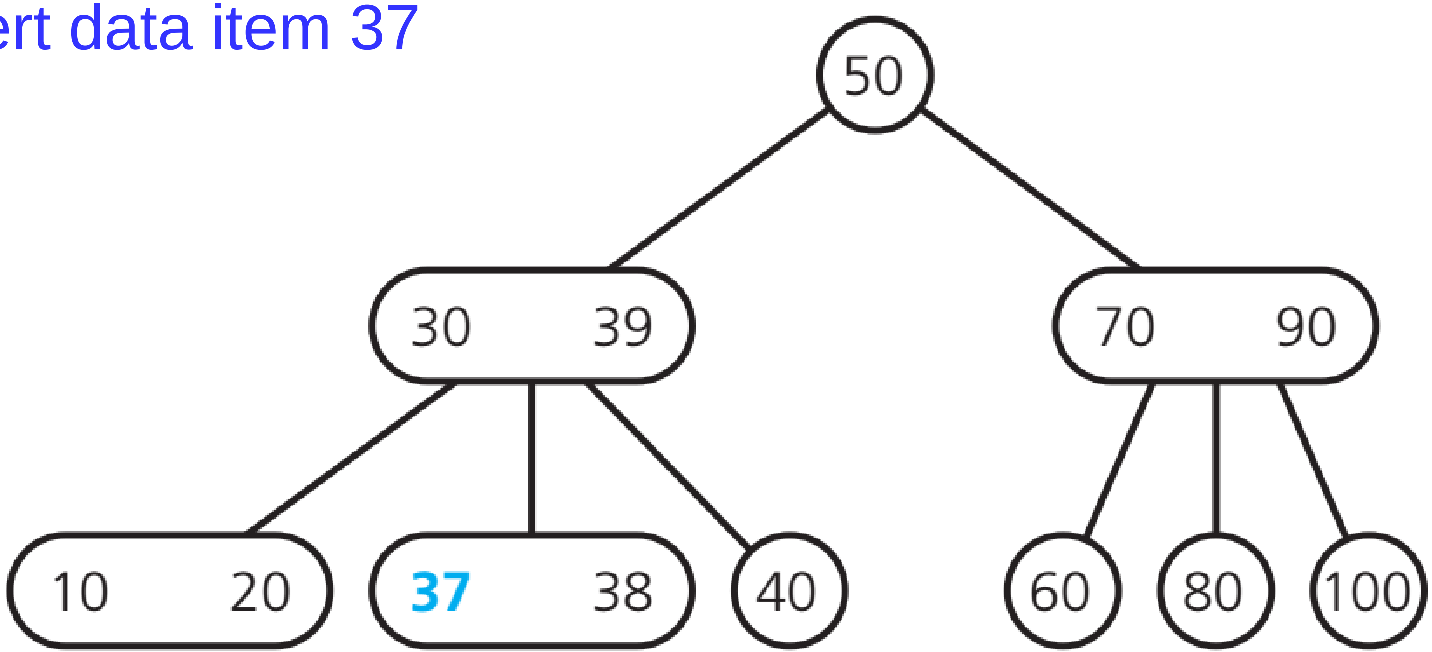
Example: 2-3 tree insersion

- Insert data item 38
  - The search stops at the leaf $\langle 39, 40 \rangle \rightarrow$ 38 **cannot** be in this node.
  - Move the middle value (39) up to the node's parent p and separate remaining values, 38 and 40, into two nodes that are attached to p.



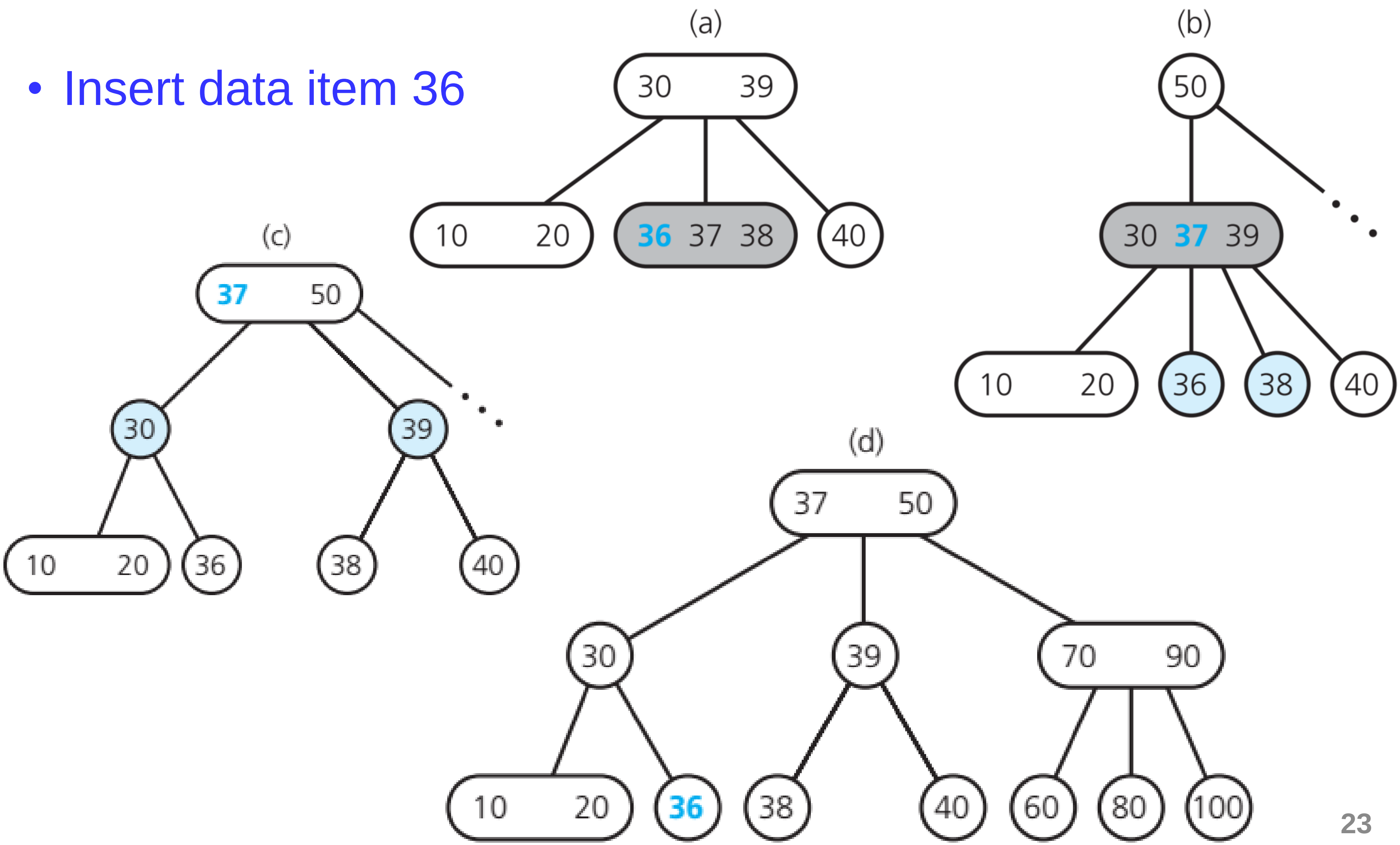
Example: 2-3 tree insersion

- Insert data item 37



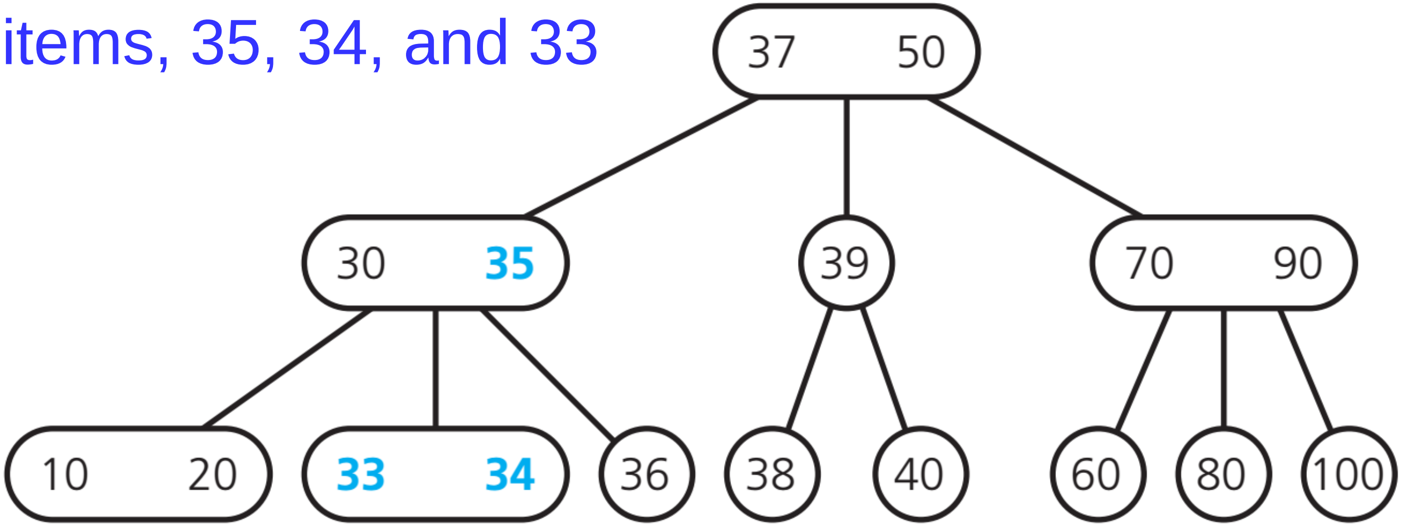
Example: 2-3 tree insersion

- Insert data item 36



Example: 2-3 tree insersion

- Insert data items, 35, 34, and 33



- Insert data item 32?

2-3 tree insertion: Implementation

```
// Inserts a new item into a 2-3 tree whose items are distinct and
// differ from the new item.
insertItem(23Tree: TwoThreeTree, newItem: ItemType)
    Locate the leaf, leafNode, in which newItem belongs
    Add newItem to leafNode
    if (leafNode has three items)
        split(leafNode)
```

Inserting data into a 2-3 tree

```
// Split node n, which contains three items.
// If n is not a leaf, it has four children.
split(n: TwoThreeNode)
    if (n is the root)
        Create a new node p
    else
        Let p be the parent of n
    Replace node n with two nodes, n1 and n2; p is their parent
    Give n1 the item in n with the smallest value
    Give n2 the item in n with the largest value
    if (n is not a leaf){
        n1 becomes the parent of n's two leftmost children
        n2 becomes the parent of n's two rightmost children
    }
    Move the item in n that has the middle value up to p
    if (p now has three items)
        split(p)
```

Checkpoint 01a: Insertion on a 2-3 tree

What is the result of inserting 5, 40, 10, 20, 15, and 30, in the order given, into an initially empty 2-3 tree?

Note that insertion of one item into an empty 2-3 tree will create a single node that contains the inserted item.

Checkpoint 01b: Insertion on a 2-3 tree

What is the result of inserting 3 and 4 into the 2-3 tree that you created in the previous question?

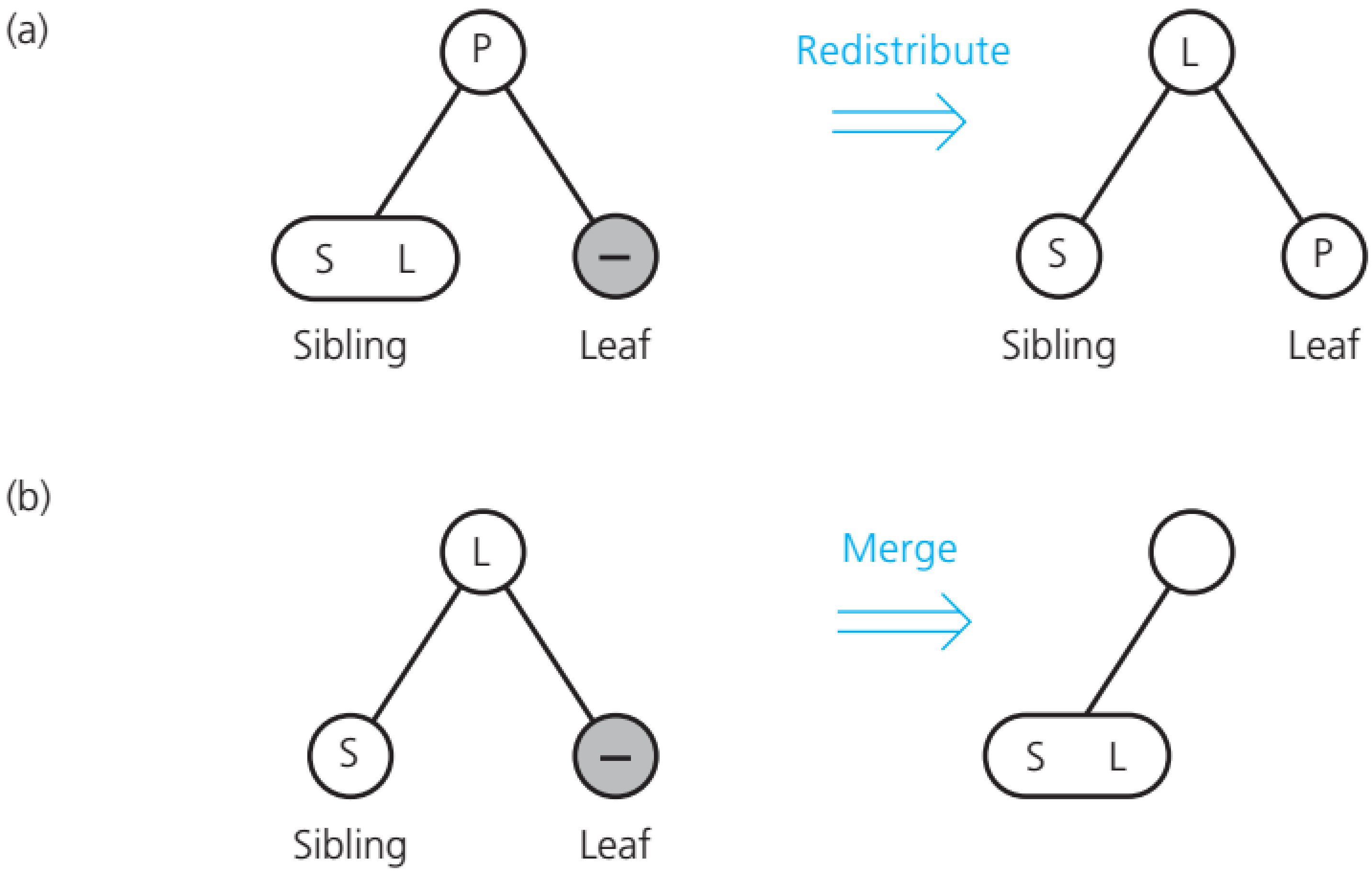
Removing data from a 2-3 tree

- The removal strategy is the inverse of the insertion strategy.
  - Insertions are spreaded throughout the tree by splitting nodes when they would become too full.
  - Removals are by merging nodes when they become empty.
- Note that the removal process usually begins at a leaf.

Removing data from a 2-3 tree

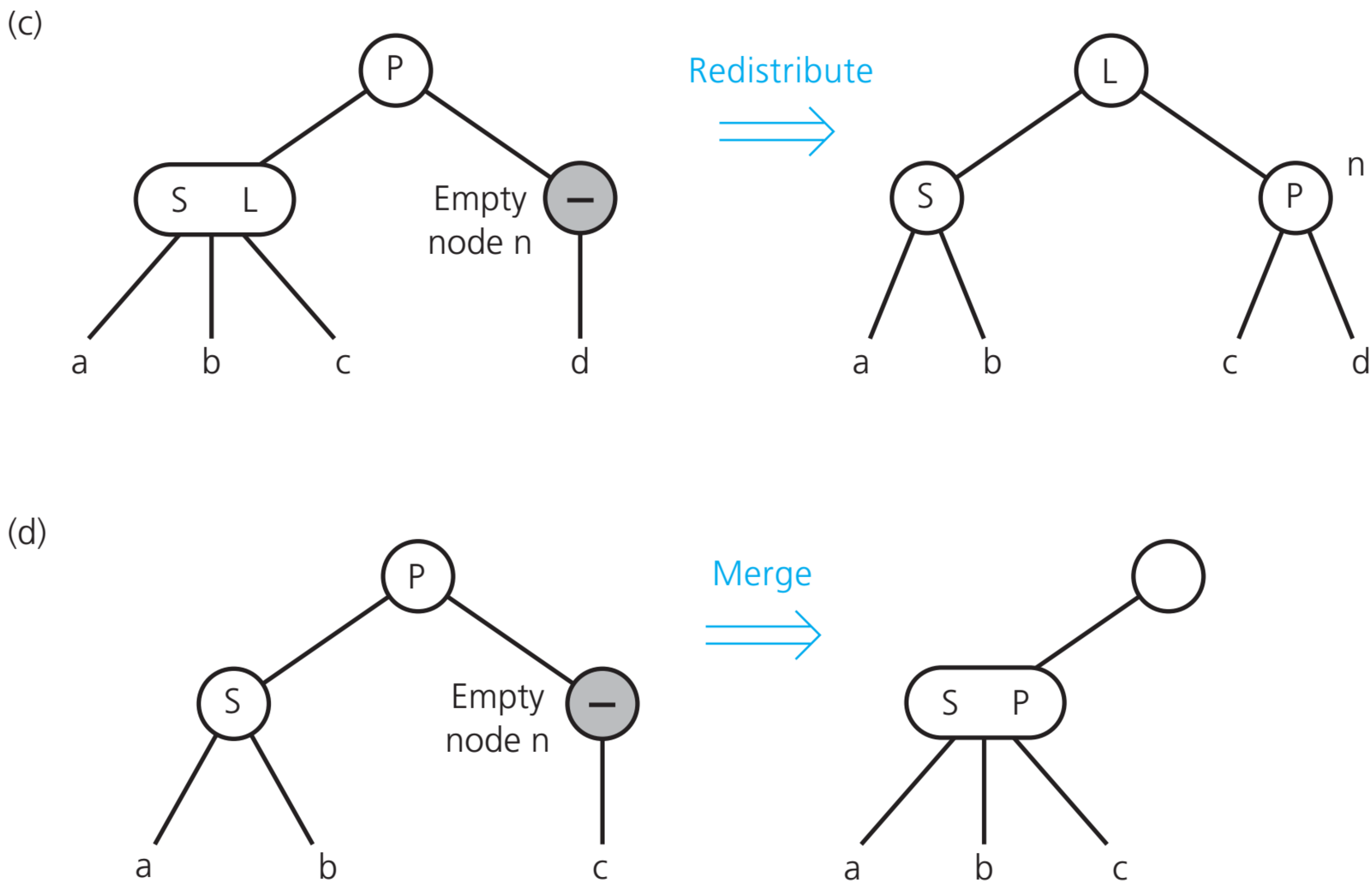
- Let I be the item to be removed from a 2-3 tree
- First locate the node n that contains I.
- If n is not a leaf, find I's inorder successor and swap it with I.
  - A node's inorder successor is node with least value in its right subtree i.e., its right subtree's left-most child.
- If the leaf contains an item in addition to I, simply remove I.
- If the leaf contains only I, redistributing and merging.

Redistributing and merging



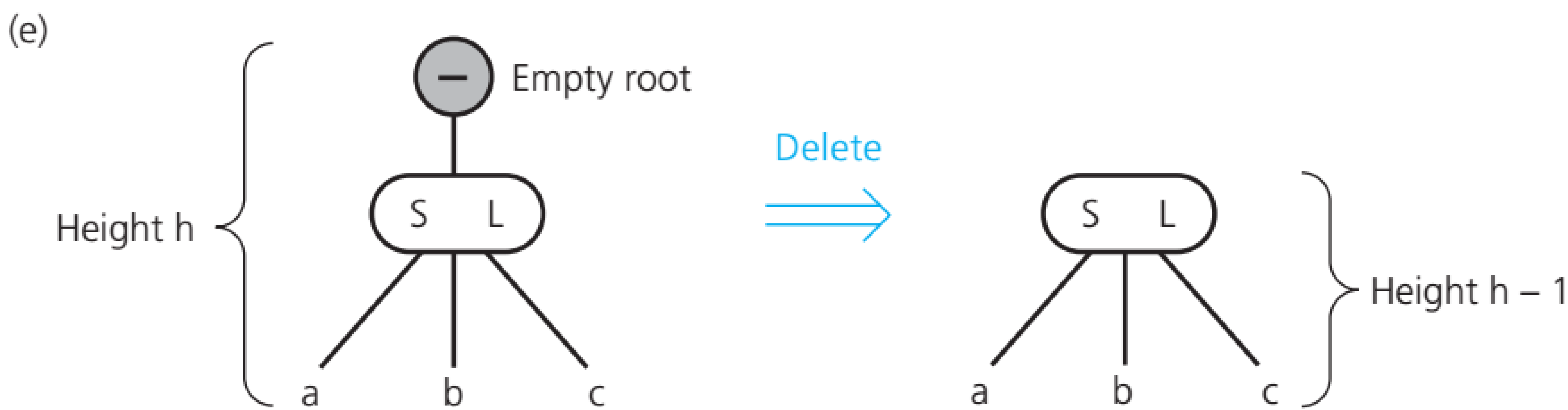
(a) Redistributing values; (b) Merging a leaf

Redistributing and merging



(c) Redistributing values and children; (d) merging an internal node

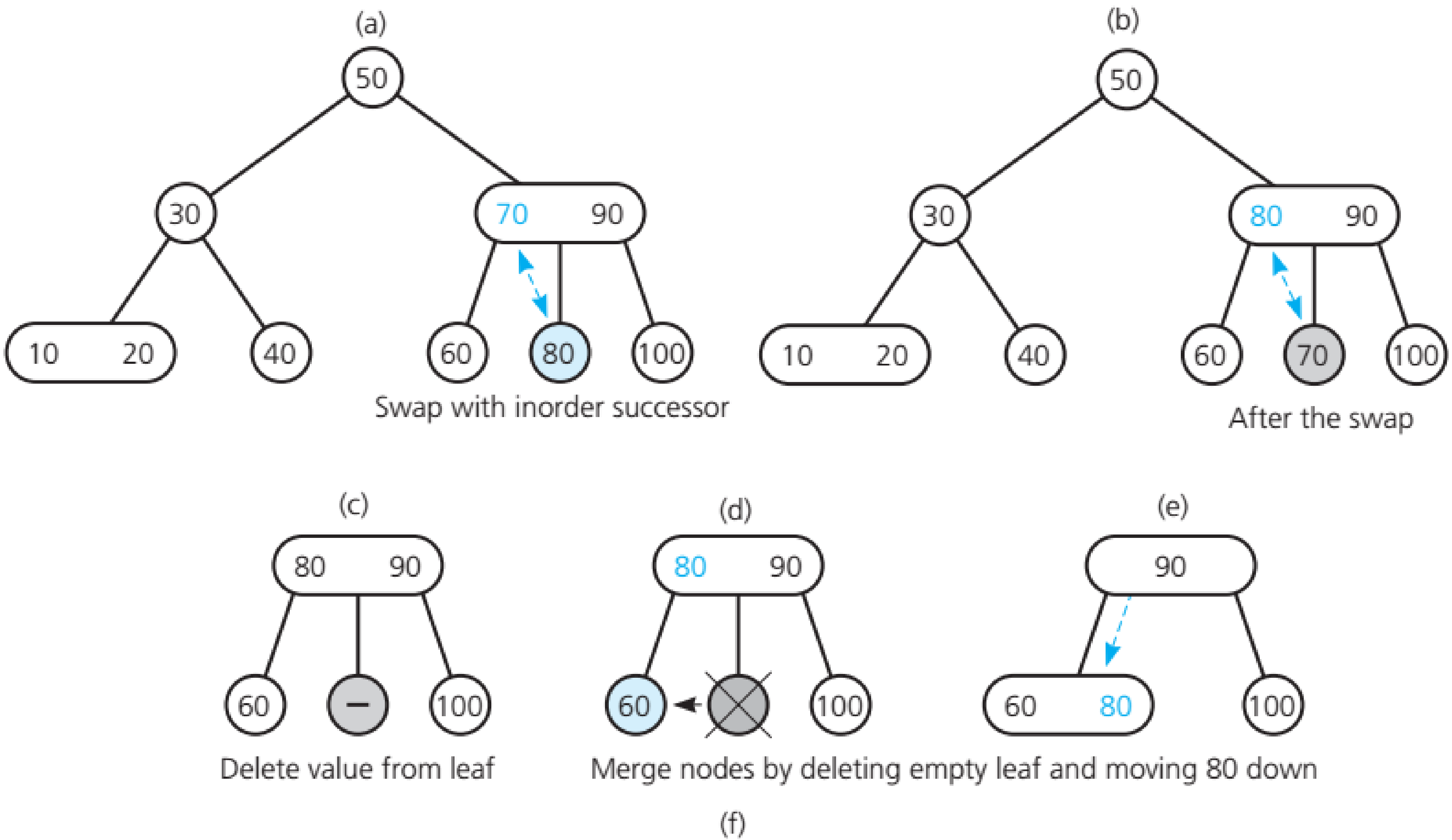
Redistributing and merging



(e) deleting the root

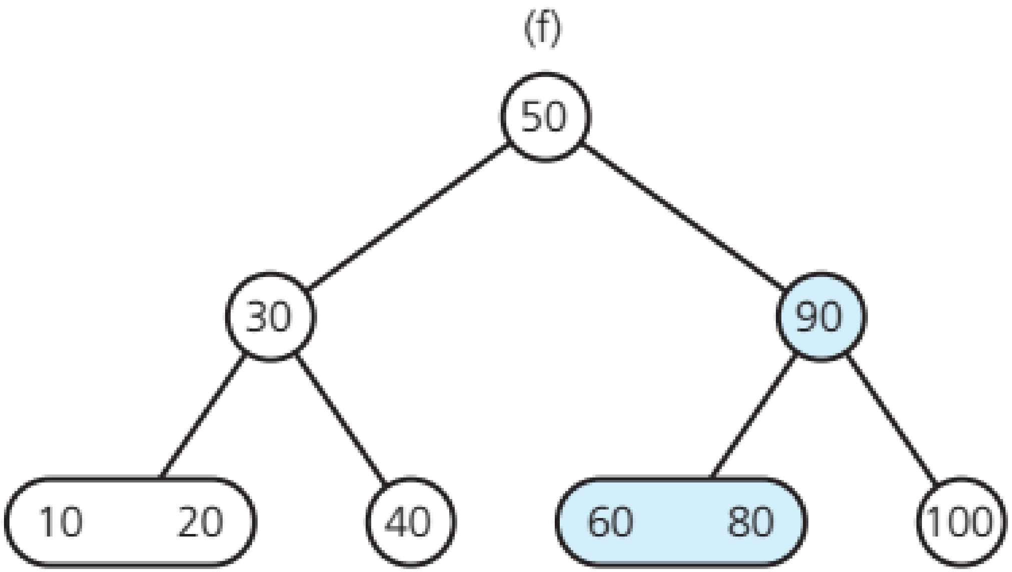
Example: 2-3 tree removal

- Remove data item 70



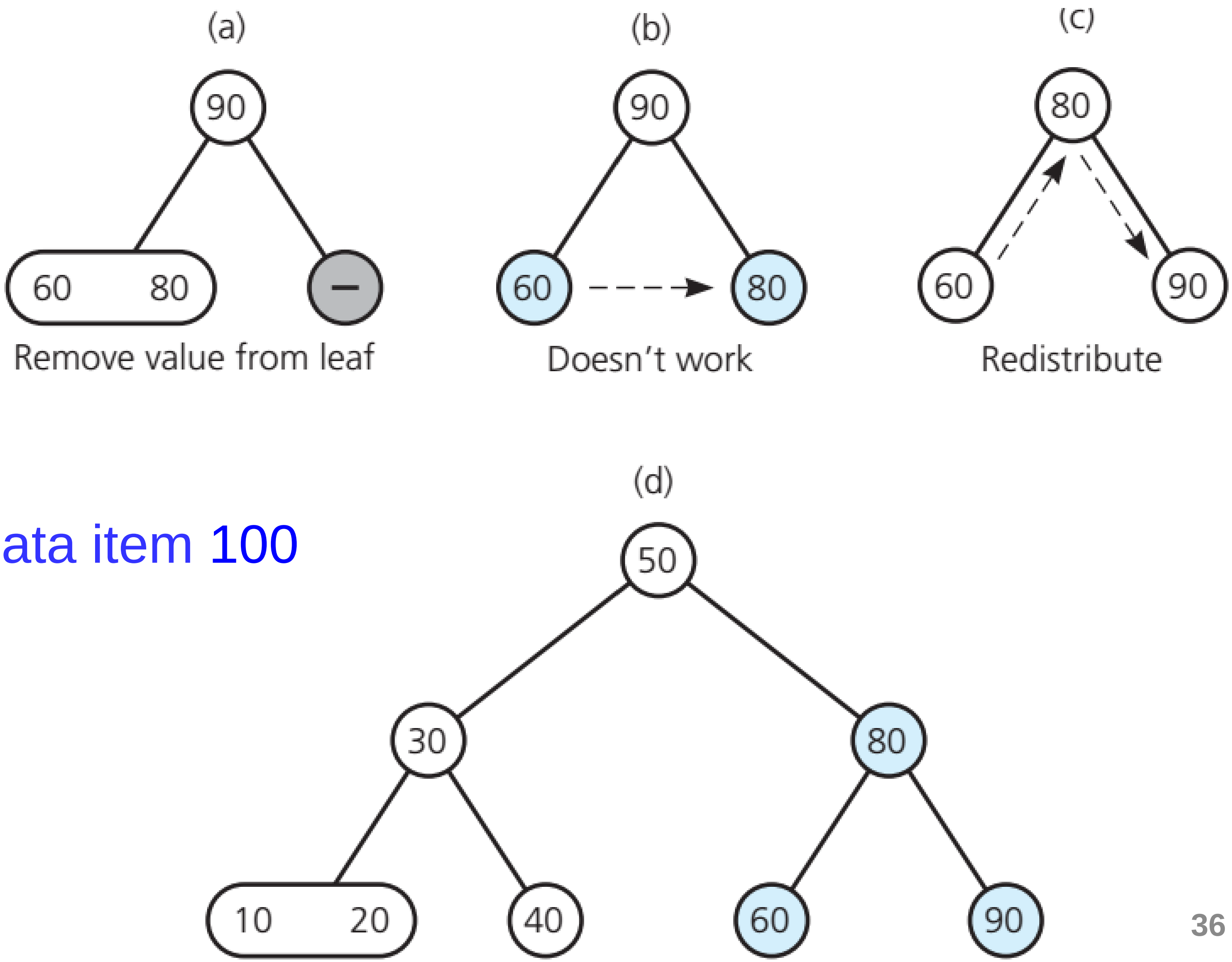
Example: 2-3 tree removal

- Remove data item 70

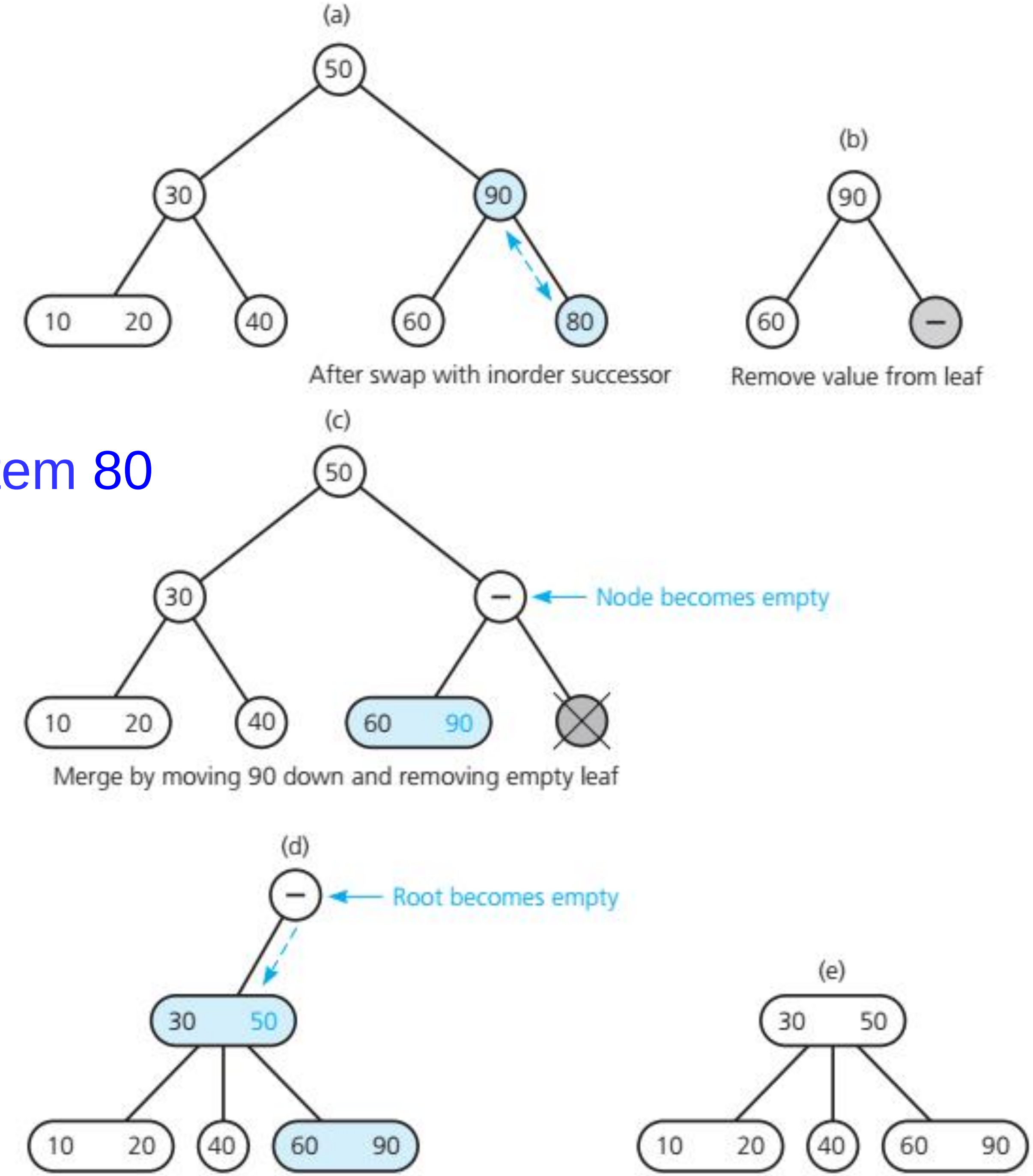


Example: 2-3 tree removal

- Remove data item 100



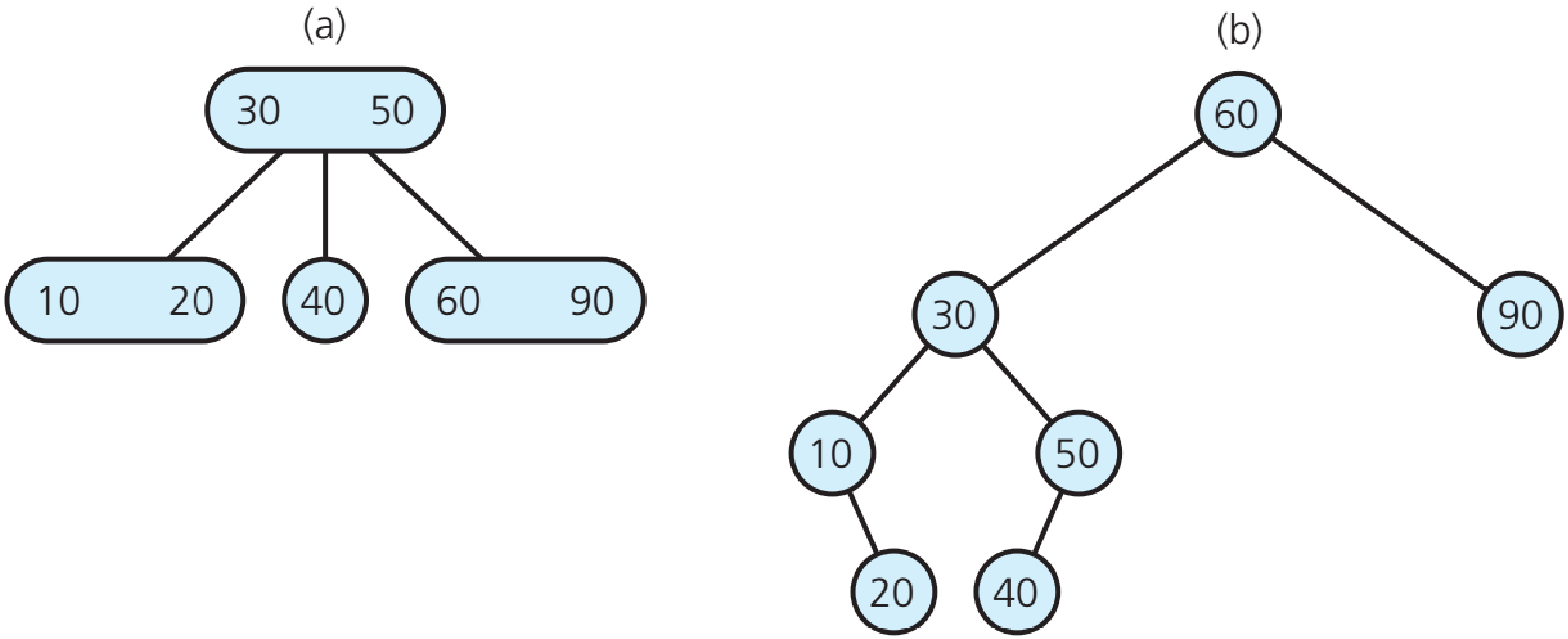
- Remove data item 80



Merge: move 50 down, adopt empty leaf's child, delete empty node

Delete empty root

Example: 2-3 tree vs. Balance BST



Results of removing 70, 100, and 80 from (a) the 2-3 Tree and (b) the BST.

2-3 tree removal implementation

```
// Remove the given data item from a 2-3 tree.
// Return true if successful or false if no such item exists.
removeItem(23Tree: TwoThreeTree, dataItem: ItemType): boolean
   Attempt to locate dataItem
   if (dataItem is found){
      if (dataItem is not in a leaf)
         Swap dataItem with its inorder successor,
         which will be in a leaf leafNode
      // The removal always begins at a leaf
      Remove dataItem from leaf leafNode
      if (leafNode now has no items)
         fixTree(leafNode)
      return true
   }
   return false
```

2-3 tree removal implementation

```
// Complete the removal when node n is empty by either deleting
the root, redistributing values, or merging nodes.
// Note: If n is internal, it has one child.
fixTree(n: TwoThreeNode)
    if (n is the root)
        Delete the root
    else{
        Let p be the parent of n
        if (some sibling of n has two items){
            Distribute items appropriately among n, sibling,
            and p
            if (n is internal)
                Move the appropriate child from sibling to n
        }
```

2-3 tree removal implementation

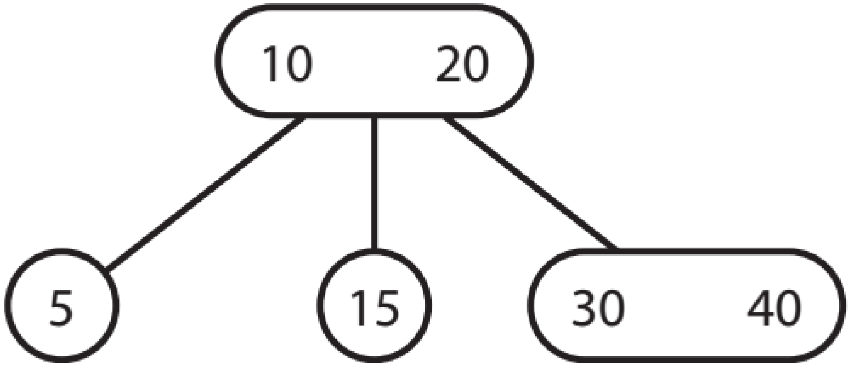
```
else{ // Merge the node
    Choose an adjacent sibling s of n
    Bring the appropriate item down from p into s
    if (n is internal)
        Move n's child to s
    Remove node n
    if (p is now empty)
        fixTree(p)
    }
}
```

Some properties of 2-3 trees

- The extra work, e.g., splitting and merging nodes, required to maintain the structure of a 2-3 tree is not significant.
  - It is sufficient to consider only the time required to locate the item when analyzing the efficiency of insertion and removal.
- A 2-3 tree is always balanced $\rightarrow$ search with the logarithmic efficiency of a binary search in all situations.
- Searching a 2-3 tree may not be quite as efficient as searching a BST of minimum height
- It is relatively simple to maintain the tree.

Checkpoint 02: Removal in a 2-3 tree

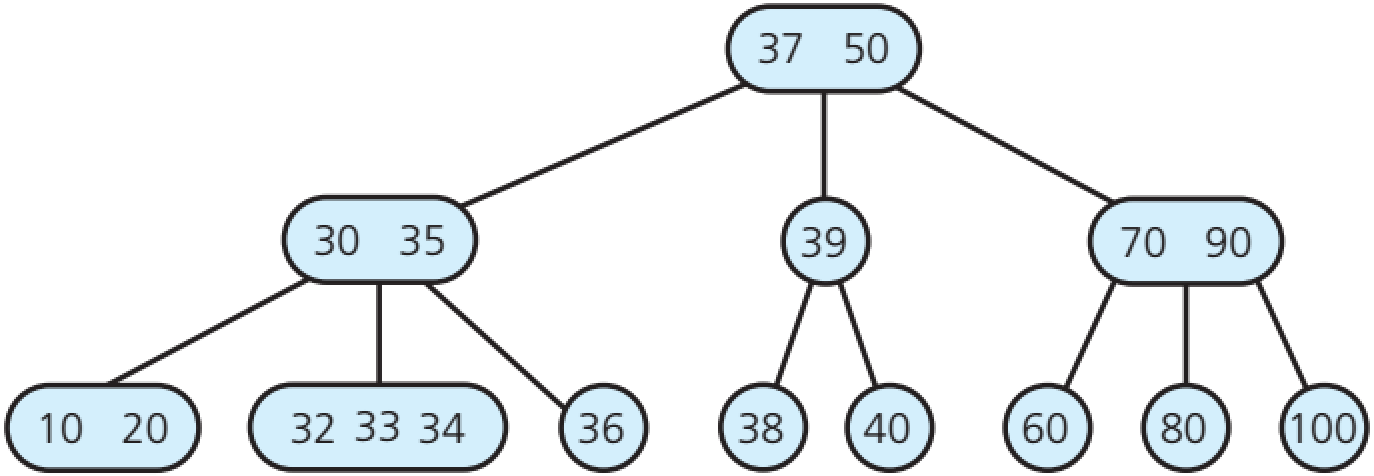
What is the result of removing 10 from the 2-3 tree shown aside?

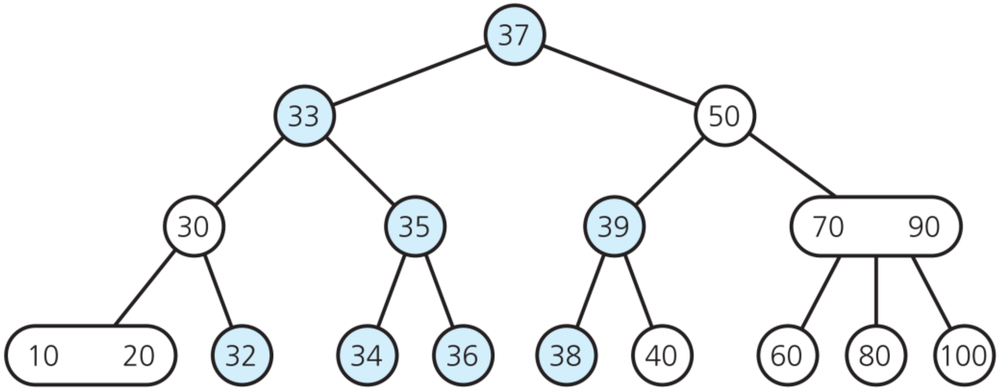


2-3-4 Trees

2-3-4 trees: A definition

- A 2-3-4 tree is like a 2-3 tree, but it also allows 4-nodes, which are nodes with four children and three data items.

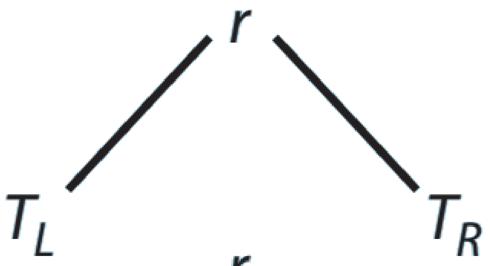




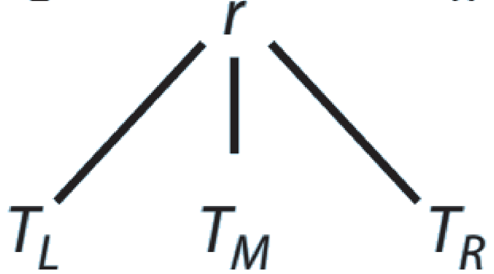
A 2-3-4 tree and a 2-3 tree with the same data items

2-3-4 trees: A definition

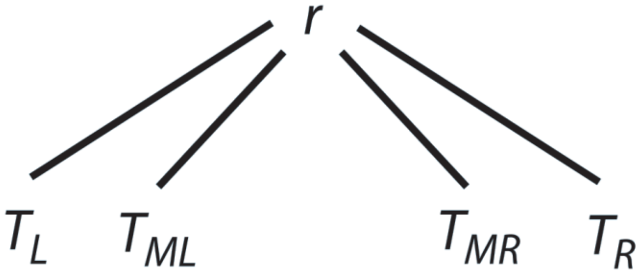
- T is a 2-3-4 tree of height h if one of the following is true
- T is empty, in which case h is 0.
- T is of the form



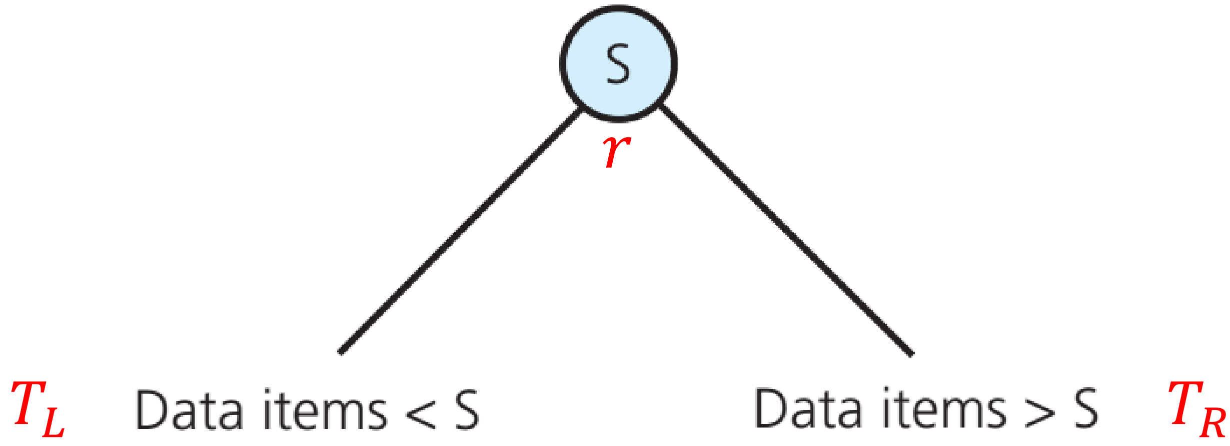
- T is of the form



- T is of the form

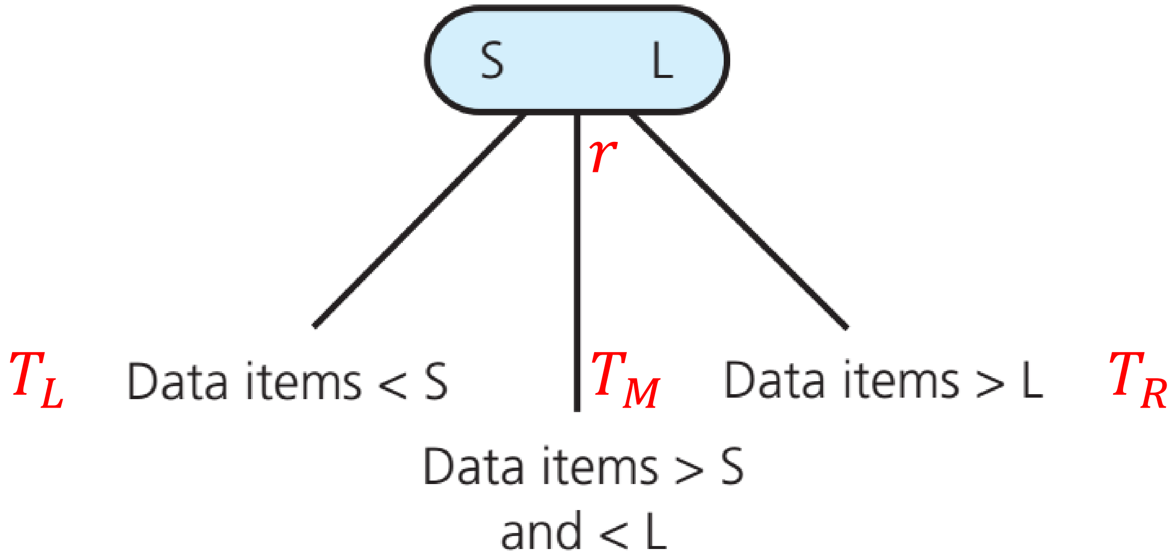


The node contains one data item



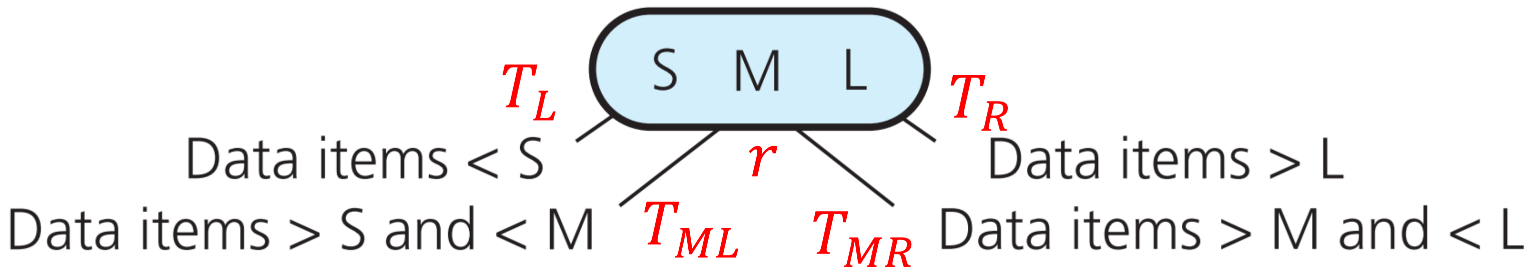
- The node S has one data item, which must be greater than each item in T_L and smaller than each item in T_R.
- T_L and T_R are both 2-3-4 trees of height $h - 1$.
- A leaf may contain one, two or three data items.

The node contains two data items



- The node r contains two ordered data item.
  - The smaller item S must be greater than each item in T_L and smaller than each item in T_M.
  - The larger item L must be greater than each item in T_M and smaller than each item in T_R.
- T_L, T_M and T_R are 2-3-4 trees of height $h - 1$.

The node contains three data items



- The node r contains three ordered data item.
  - The smallest item S must be greater than each item in T_L and smaller than each item in T_{ML}.
  - The middle item M must be greater than each item in T_{ML} and smaller than each item in T_{MR}.
  - The largest item L must be greater than each item in T_{MR} and smaller than each item in T_R.
- T_L, T_{ML}, T_{MR} and T_R are 2-3-4 trees of height $h-1$.

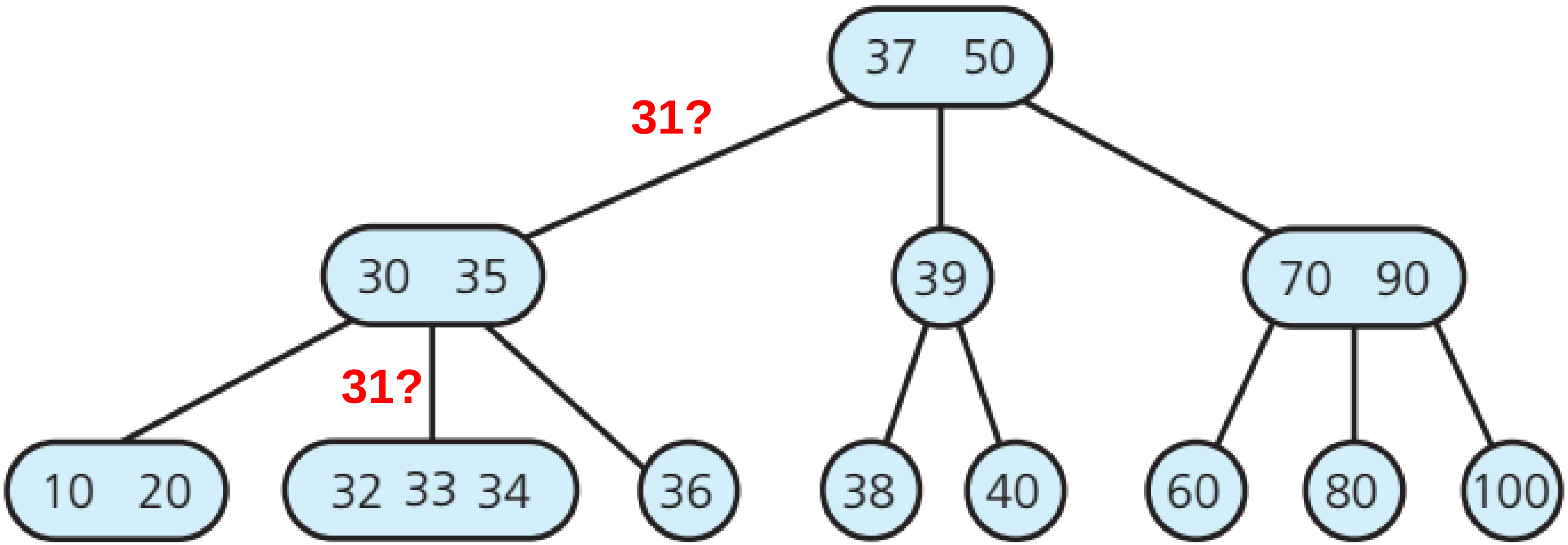
2-3-4 trees implementation

- Greater storage requirements due to more data members

```
class QuadNode{
   private:
      // Data portion
      ItemType smallItem, middleItem, largeItem;
      // Pointers for left child and right child
      QuadNode*leftChildPtr, * rightChildPtr;
      // Pointers for middle-left child and middle-right child
      QuadNode* leftMidChildPtr, * rightMidChildPtr;
public:
      // Constructors, accessor, and mutator methods are here.
      ……
}
```

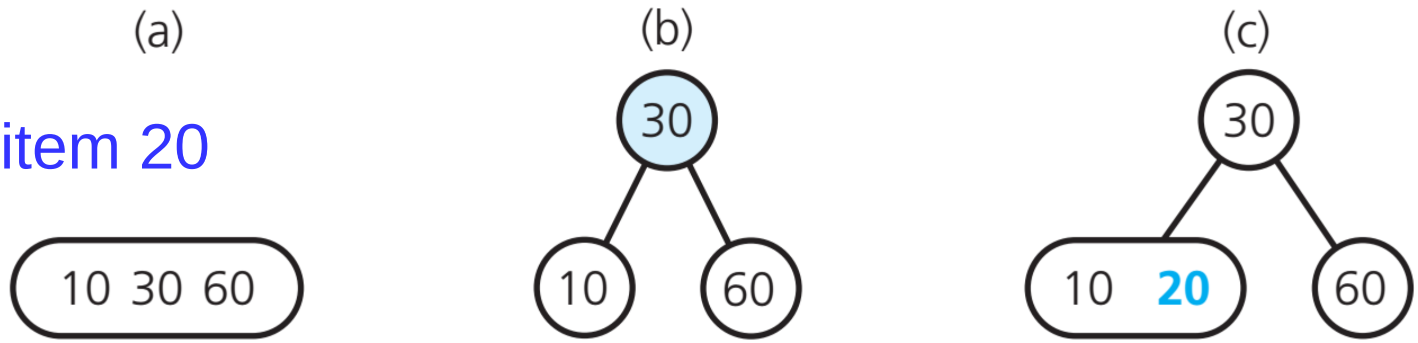
Searching and traversing a 2-3-4 tree

- Extend the corresponding algorithms for a 2-3 tree

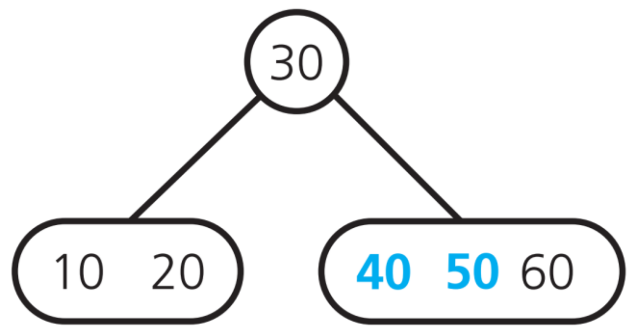


Inserting data into a 2-3-4 tree

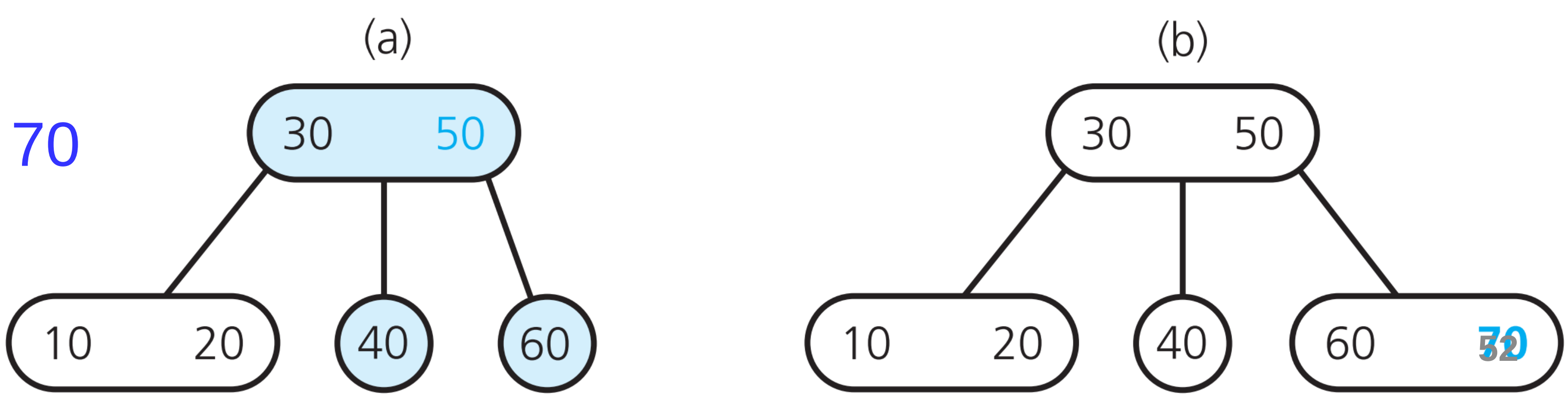
- Insert data item 20



- Insert data items, 50 and 40

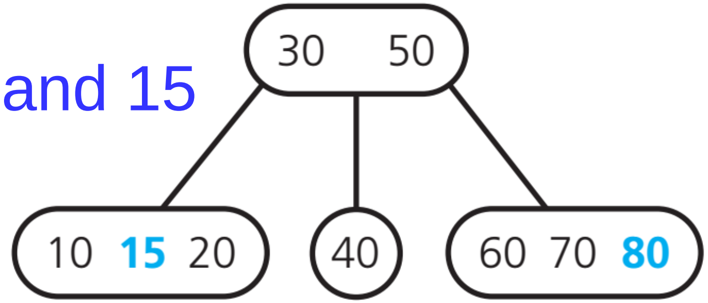


- Insert 70



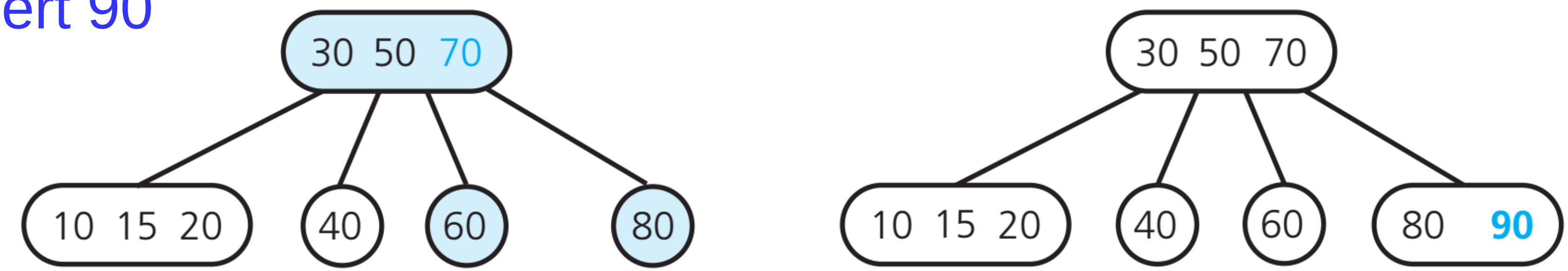
Inserting data into a 2-3-4 tree

- Insert data items, 80 and 15



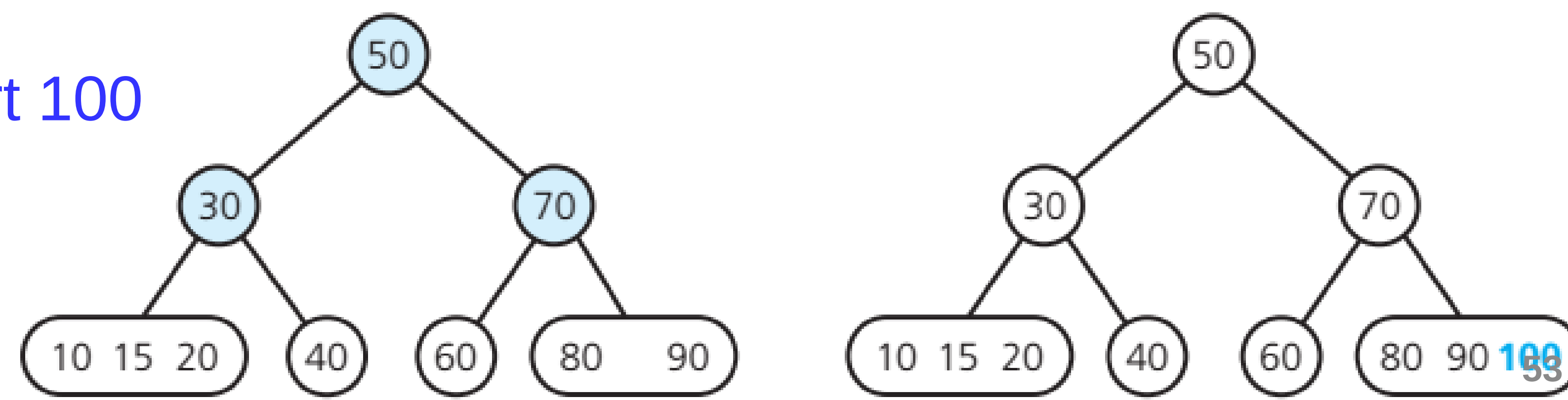
- Insert 90

(a) (b)



- Insert 100

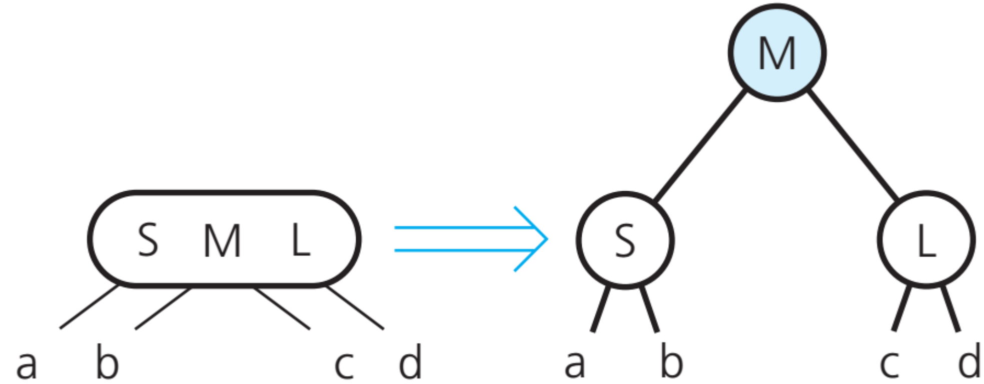
(a) (b)



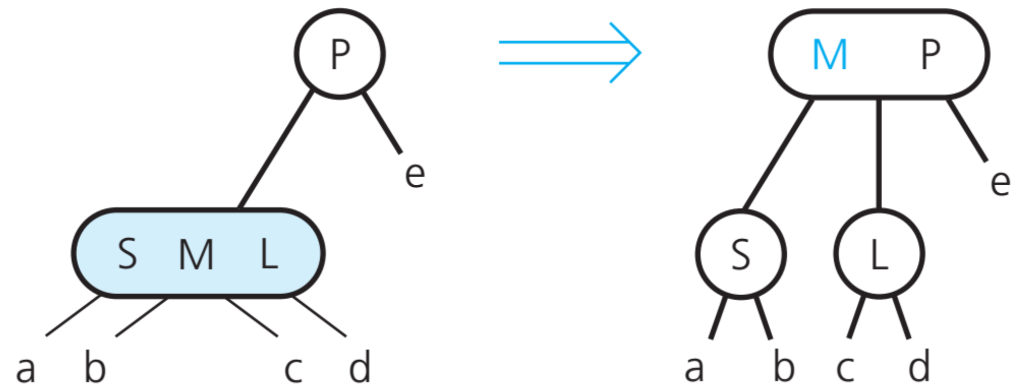
Splitting 4-nodes during insertion

- Split each 4-node as soon as it is encountered during the search from the root to the leaf that will accommodate the new item to be inserted.
- As a result, each 4-node either will
  - Be the root,
  - Have a 2-node parent, or
  - Have a 3-node parent

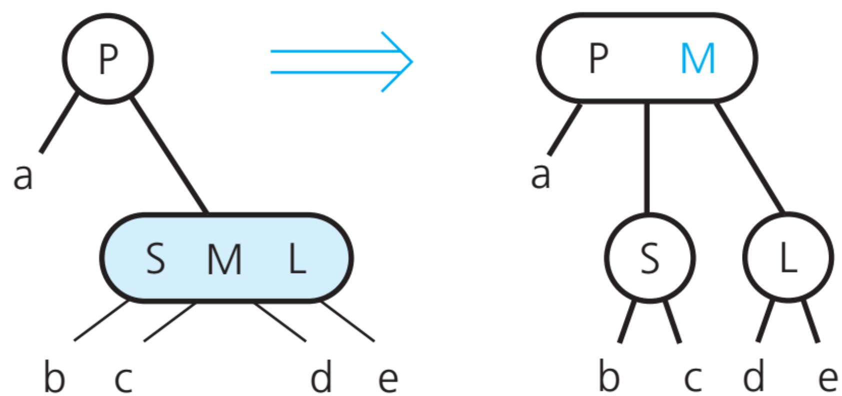
Splitting a 4-node root



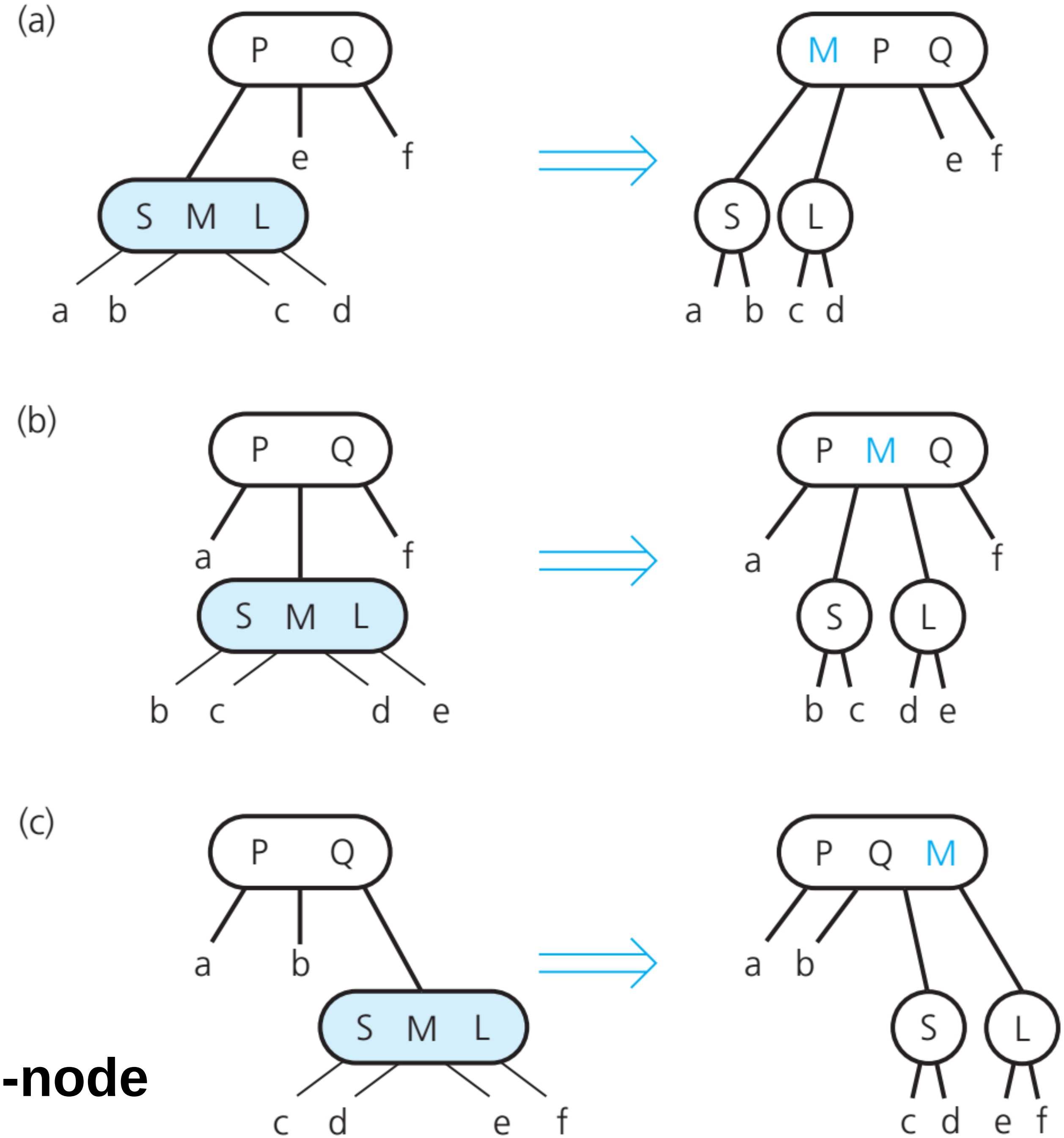
(a)



(b)



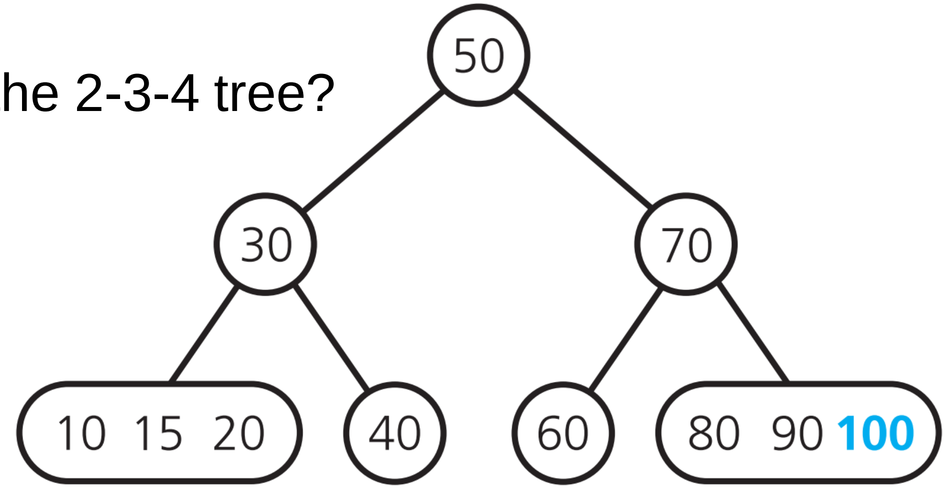
Splitting a 4-node whose parent is 2-node



Splitting a 4-node whose parent is 3-node

Checkpoint 03a: Insertion in a 2-3-4 tree

What is the result of inserting 25 into the 2-3-4 tree?



Checkpoint 03b: Insertion in a 2-3-4 tree

What is the result of inserting 3 and 4 into the 2-3-4 tree that you created in the previous question?

Removing data from a 2-3-4 tree

- The removal for a 2-3-4 tree has the same beginning as the removal for a 2-3 tree.
  - Let I the item to be remove from a 2-3-4 tree
  - First locate the node n that contains I.
  - If n is not a leaf, find I's inorder successor and swap it with I.
  - If the leaf contains an item in addition to I, simply remove I
- If I is ensured to not occur in a 2-node, the removal performs in one pass through the tree from root to leaf.
  - You will not have to back away from the leaf and restructure the tree like in the case of a 2-3 tree.

Removing data from a 2-3-4 tree

- It can be guaranteed that I does not occur in a 2-node by transforming each 2-node encountered during the search for I into either a 3-node or a 4-node.
- Several cases are possible, depending on the configuration of the 2-node's parent and its nearest sibling.
  - If both the parent and nearest sibling are 2-nodes, reversely apply the transformation 
  - If the parent is a 3-node, reversely apply the transformation 
  - If the parent is a 4-node, reversely apply 

Acknowledgements

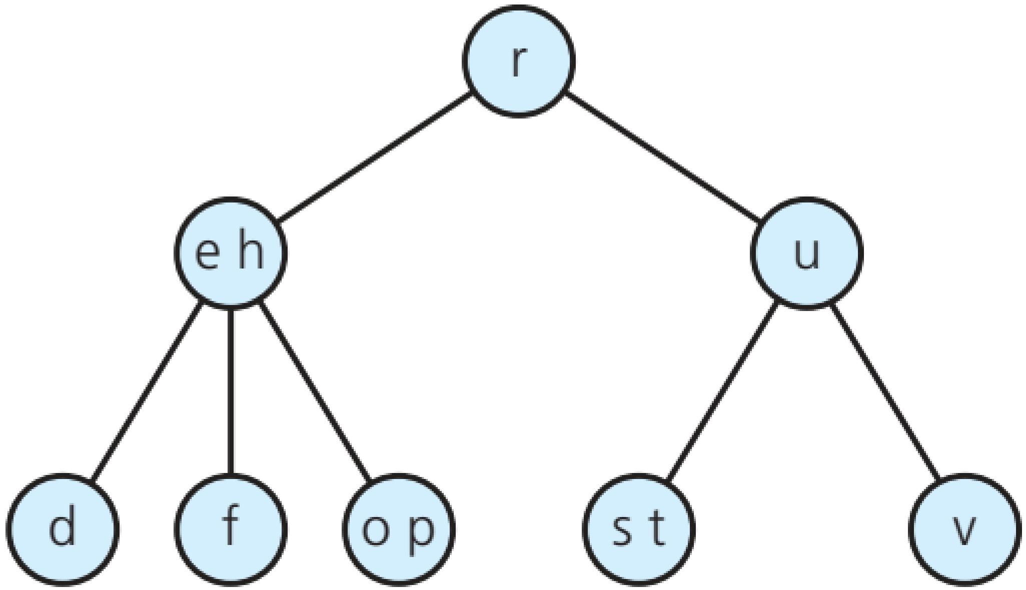
- This part of the lecture is adapted from

[1] Frank M. Carrano, Robert Veroff, Paul Helman (2014) "*Data Abstraction and Problem Solving with C++: Walls and Mirrors*" Sixth Edition, Addion-Wesley. **Chapter 19, section 19.2 – 19.3.**

Exercises

01. Insertion in a 2-3 tree

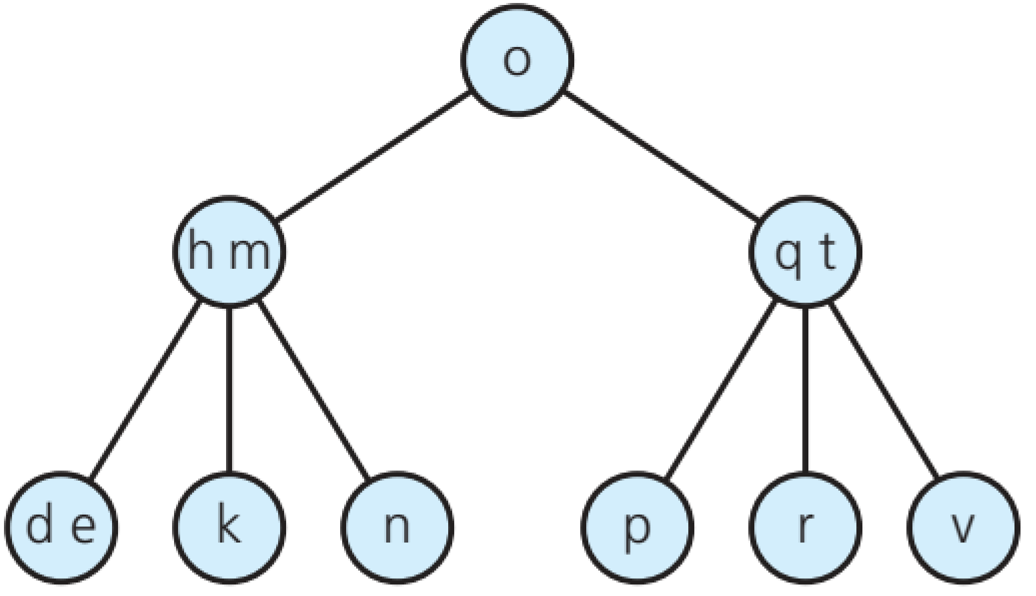
- Consider the following 2-3 tree.



- Draw the tree that results after inserting k , b , c , y , and w into the tree.

02. Removal in a 2-3 tree

- Consider the following 2-3 tree.



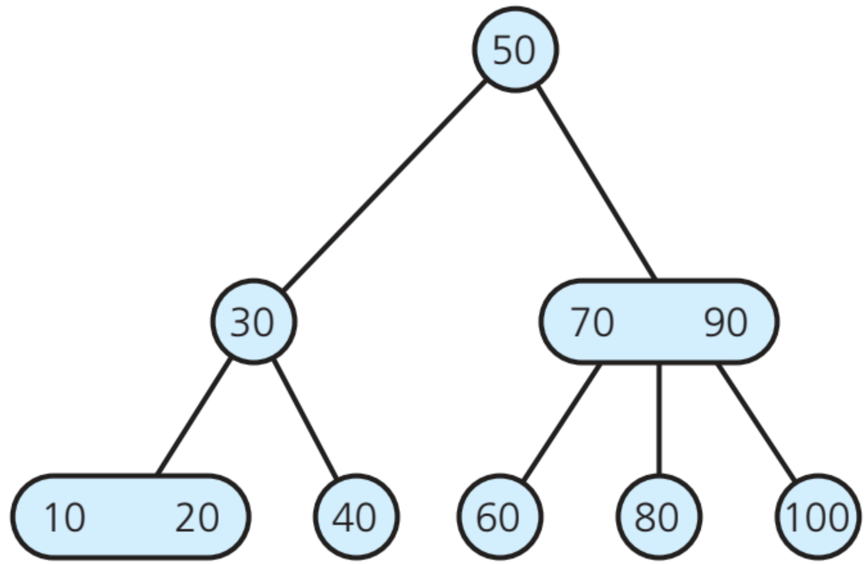
- Draw the tree that results after removing t , e , k , and d from the tree.

03. Insertion in a 2-3-4 tree

- Draw the 2-3-4 tree that results from inserting o , d , j , h , s , g , and a , in the order given, into a 2-3-4 tree that contains a single node of value n.

04. Insertion in a 2-3-4 tree

- Consider the following 2-3-4 tree.



- Draw the tree that results from inserting 39, 38, 37, 36, 35, 34, 33, and 32 into the tree.

05. 2-3 tree insertion / removal

- Consider the following sequence of operations on an initially empty search tree
    1. Insert 10
    2. Insert 100
    3. Insert 30
    4. Insert 80
    5. Insert 50
    6. Remove 10
    7. Insert 60
    8. Insert 70
    9. Insert 40
    10. Remove 80
    11. Insert 90
    12. Insert 20
    13. Remove 30
    14. Remove 70
- What does the tree look like after these operations execute if the tree is a 2-3 tree?

06. 2-3-4 tree insertion / removal

- Consider the following sequence of operations on an initially empty search tree
  1. Insert 10
  2. Insert 100
  3. Insert 30
  4. Insert 80
  5. Insert 50
  6. Remove 10
  7. Insert 60
  8. Insert 70
  9. Insert 40
  10. Remove 80
  11. Insert 90
  12. Insert 20
  13. Remove 30
  14. Remove 70
- What does the tree look like after these operations execute if the tree is a 2-3-4 tree?


...the end.