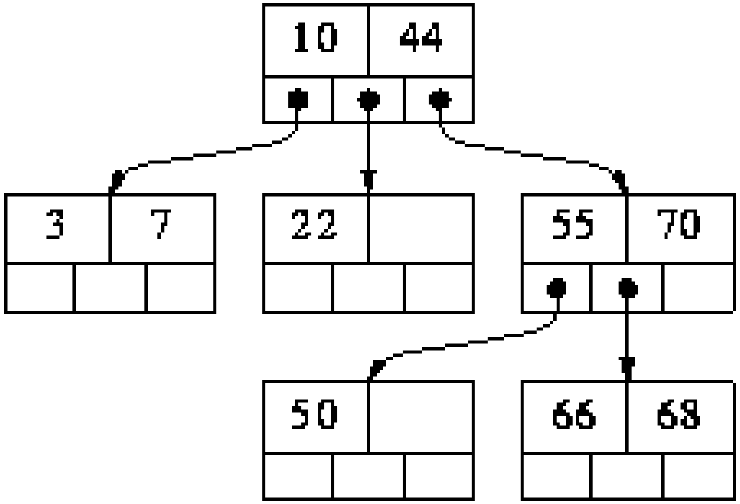
2-3 and 2-3-4 Trees

COL 106

Shweta Agrawal, Amit Kumar, Dr. Ilyas Cicekli

Multi-Way Trees

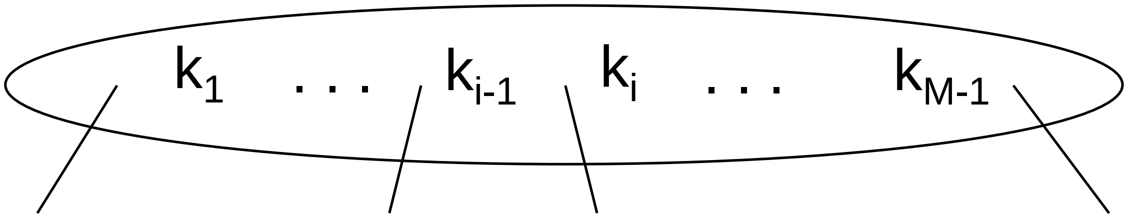
- A binary search tree:
  - *One* value in each node
  - At most 2 children
- An *M-way* search tree:
  - Between *1* to *(M-1)* values in each node
  - At most *M* children per node



M-way Search Tree Details

Each internal node of an *M-way* search has:

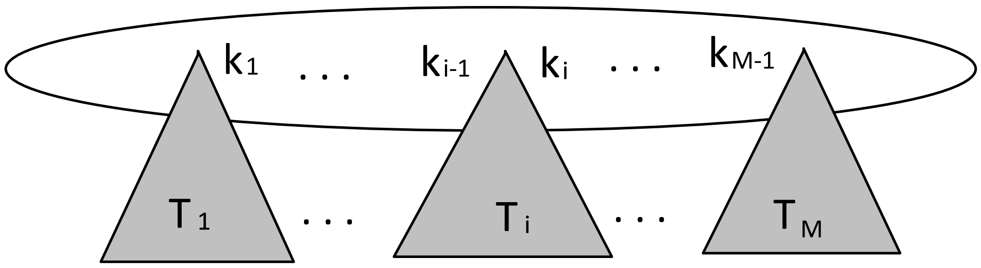
- Between *1* and *M* children
- Up to *M-1* keys $k_1, k_2, \ldots, k_{M-1}$



Keys are ordered such that:

$k_1 < k_2 < \ldots < k_{M-1}$

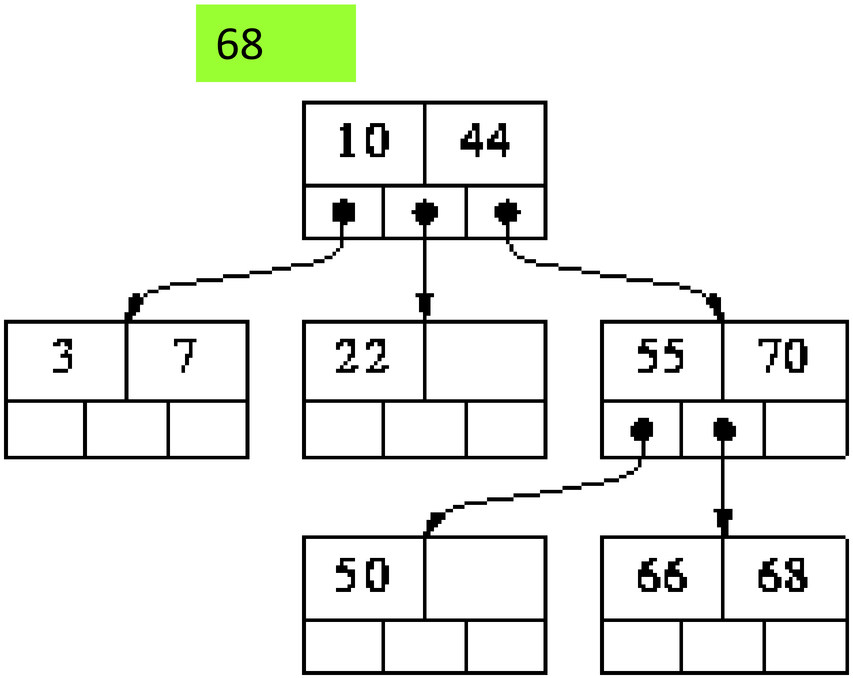
Properties of M-way Search Tree



- For a subtree T_i that is the *i-th* child of a node:
  all keys in T_i must be between keys k_{i-1} and k_i
  i.e. $k_{i-1} < keys(T_i) < k_i$
- All keys in first subtree T_1, $keys(T_1) < k_1$
- All keys in last subtree T_M, $keys(T_M) > k_{M-1}$

Example: 3-way search tree

Try: search 68



Search for X

At a node consisting of values $V_1 \ldots V_k$, there are four possible cases:

- If $X < V_1$, recursively search for X in the subtree that is left of $V1$
- If $X > V_k$, recursively search for X in the subtree that is right of V_k
- If $X = V_i$, for some i, then we are done

  (X has been found)
- Else, for some i, $V_i < X < V_{i+1}$. In this case recursively search for X in the subtree that is between V_i and V_{i+1}

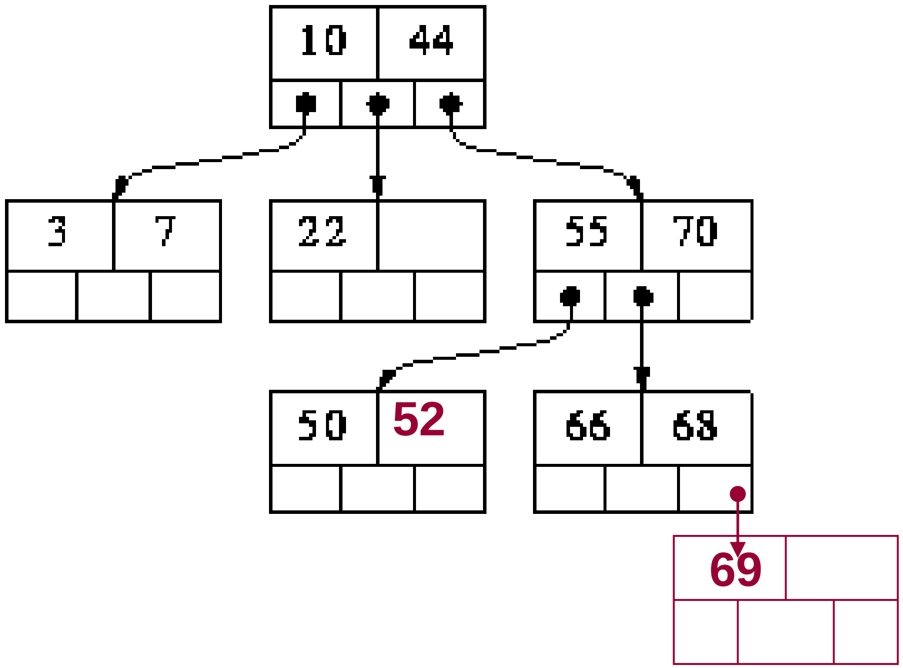
- Time Complexity: $O((M-1)*h)=O(h)$ [M is a constant]

Insert X

The algorithm for binary search tree can be generalized

- Follow the search path
  - Add new key into the last leaf, or
  - add a new leaf if the last leaf is fully occupied

Example: Add *52*,*69*



Delete X

The algorithm for binary search tree can be generalized:

- A leaf node can be easily deleted
- An internal node is replaced
  by its successor and the
  successor is deleted

Example:

- Delete *10,* Delete *44,*

Time complexity: O(Mh)=O(h), but *h* can be *O(n)*

M-way Search Tree

What we know so far:

- What is an *M-way* search tree
- How to implement *Search*, *Insert*, and *Delete*
- The time complexity of each of these operations is: $O(Mh)=O(h)$

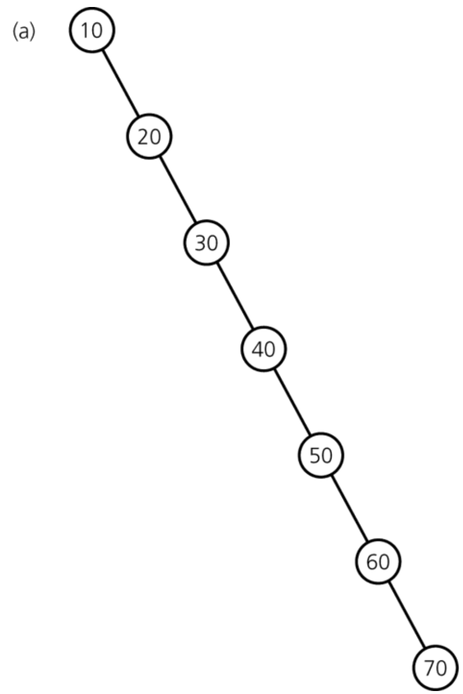
The problem (as usual): h can be $O(n)$.

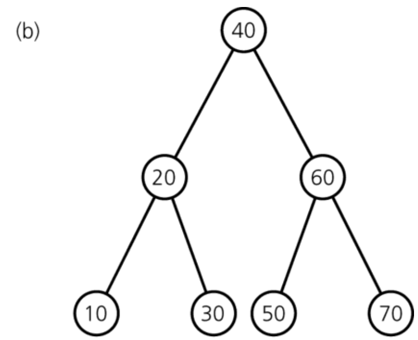
- B-tree: balanced M-way Search Tree

2-3 Tree

Why care about advanced implementations?

Same entries, different insertion sequence:



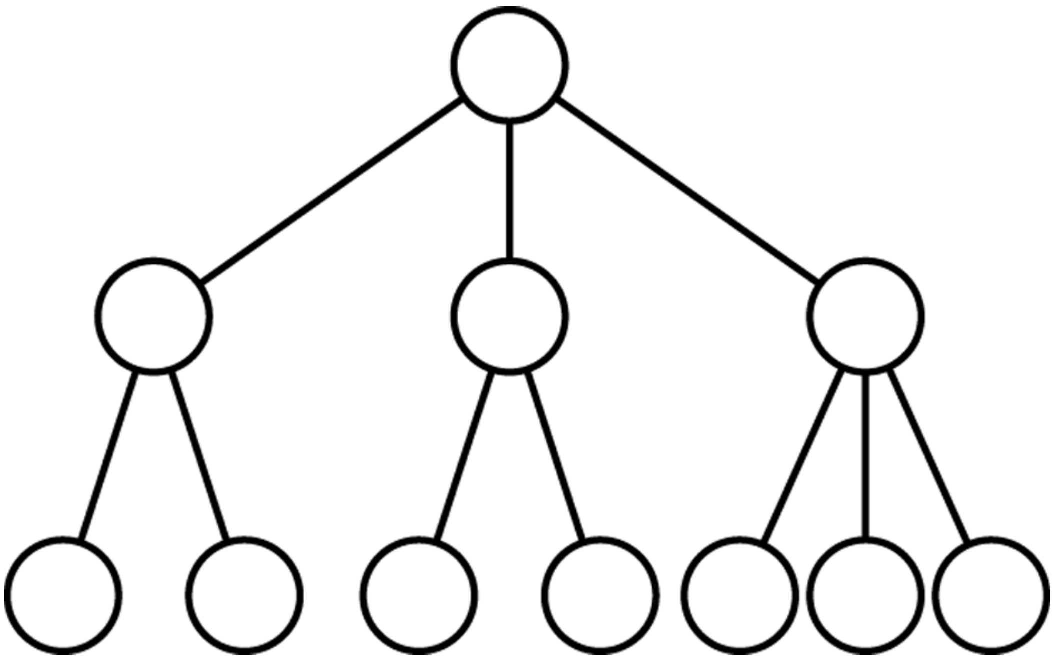


→ Not good! Would like to keep tree balanced.

2-3 Trees

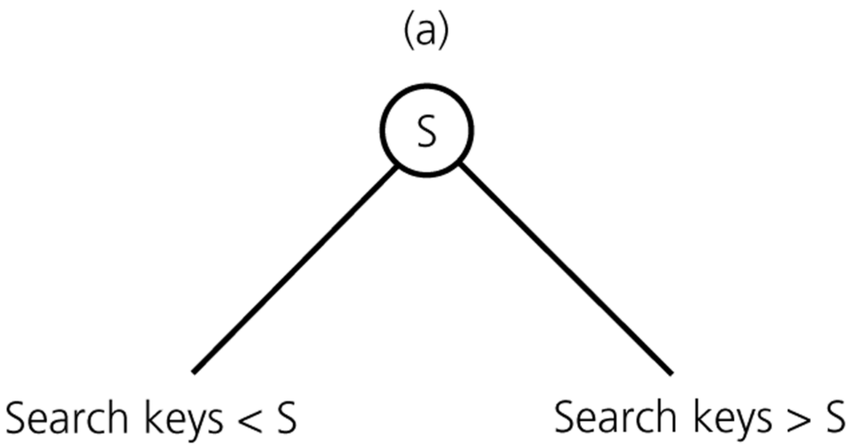
Features

- each internal node has either 2 or 3 children
- all leaves are at the same level

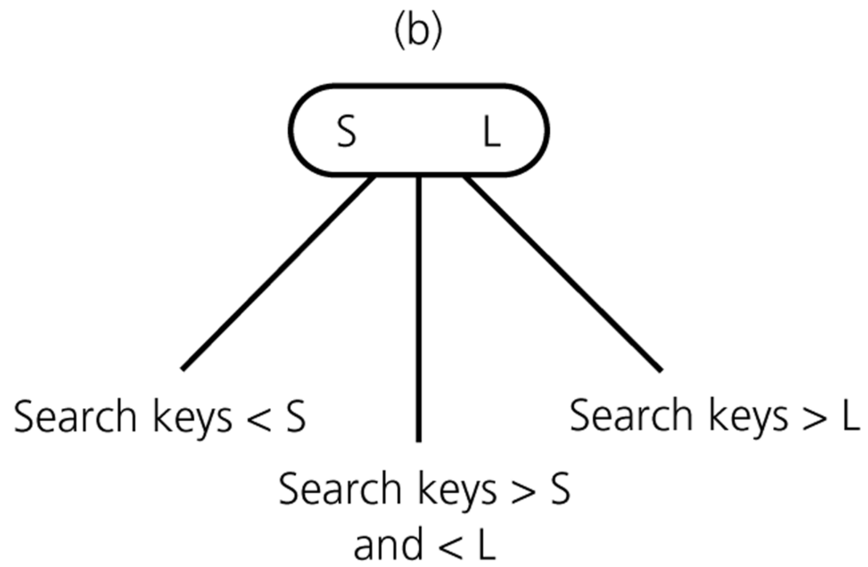


2-3 Trees with Ordered Nodes

2-node

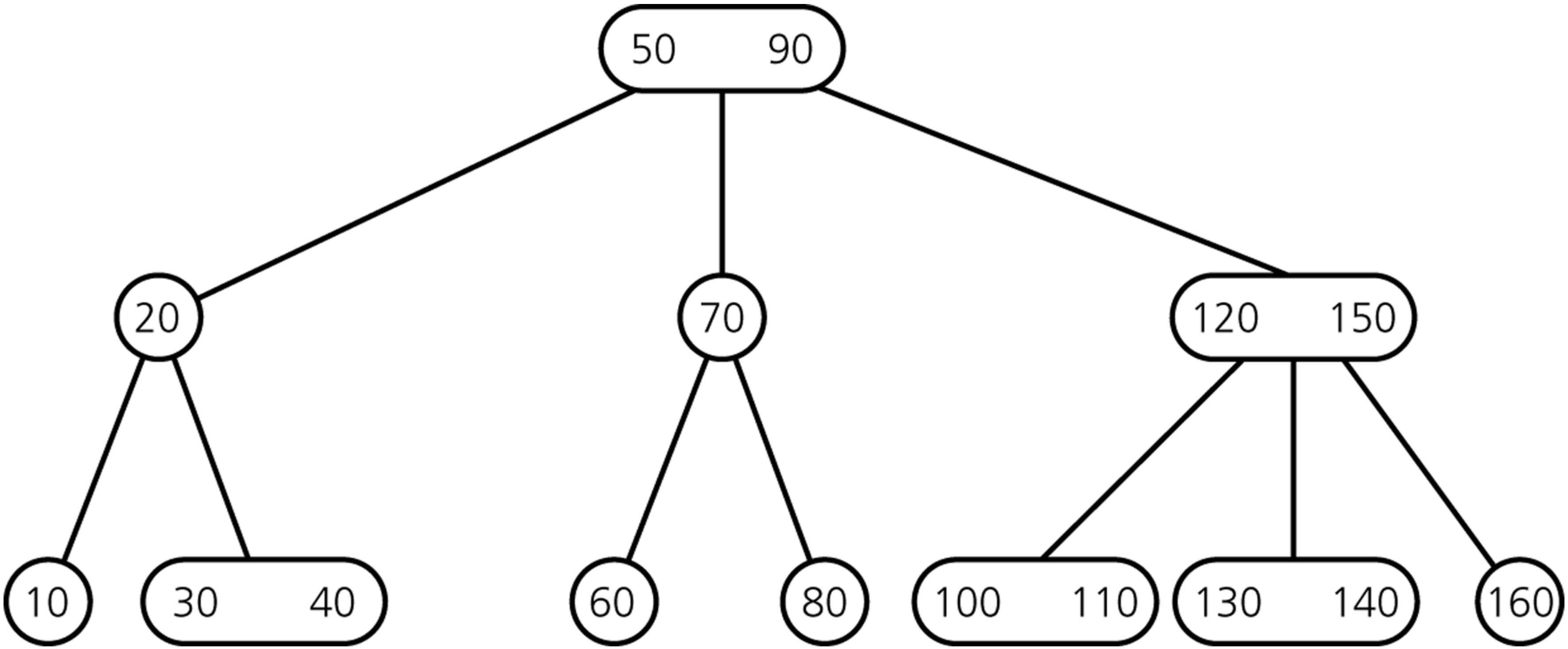


3-node

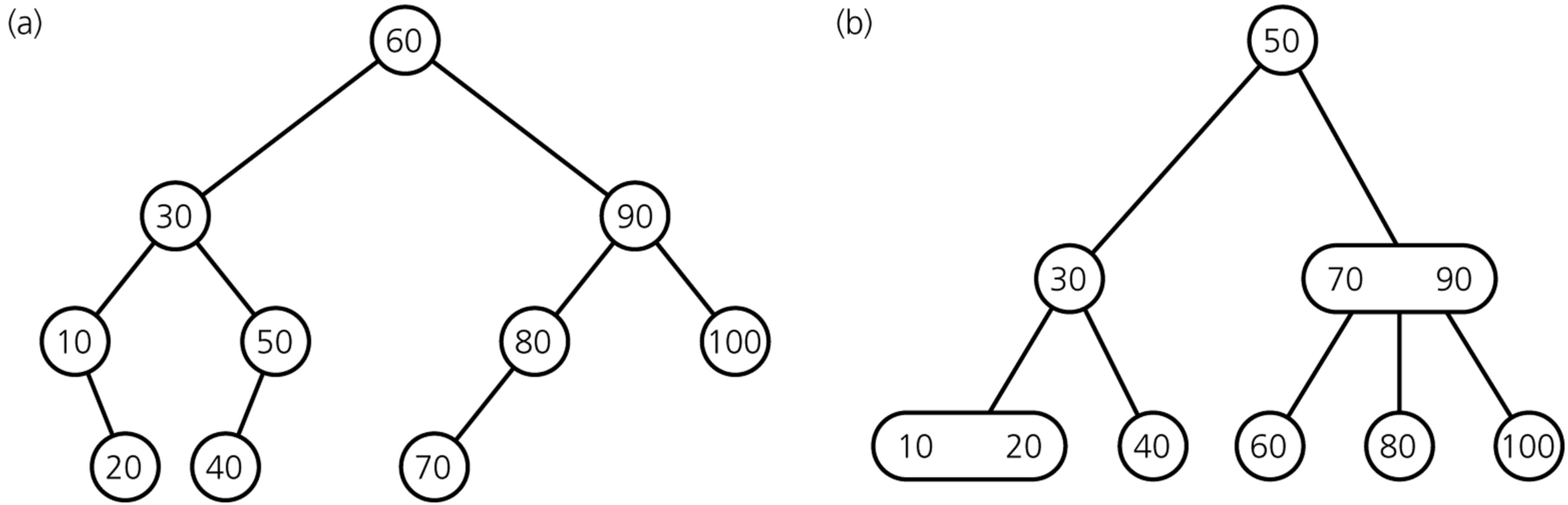


- leaf node can be either a 2-node or a 3-node

Example of 2-3 Tree



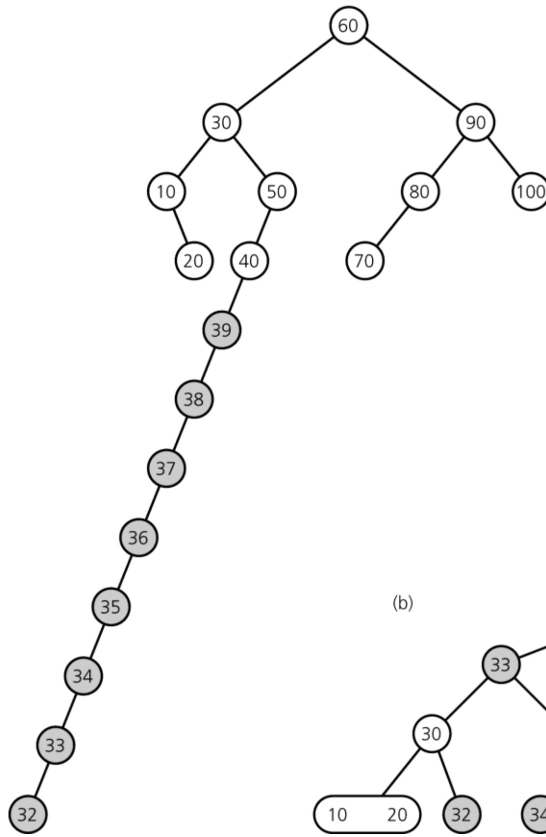
What did we gain?



What is the time efficiency of searching for an item?

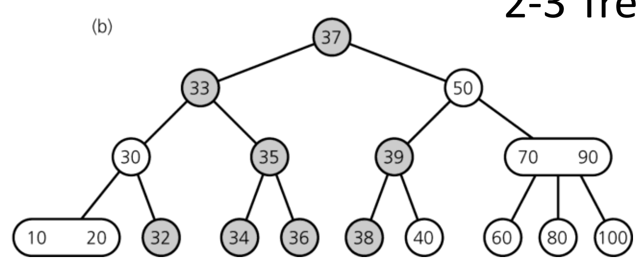
Gain: Ease of Keeping the Tree Balanced

Binary Search Tree



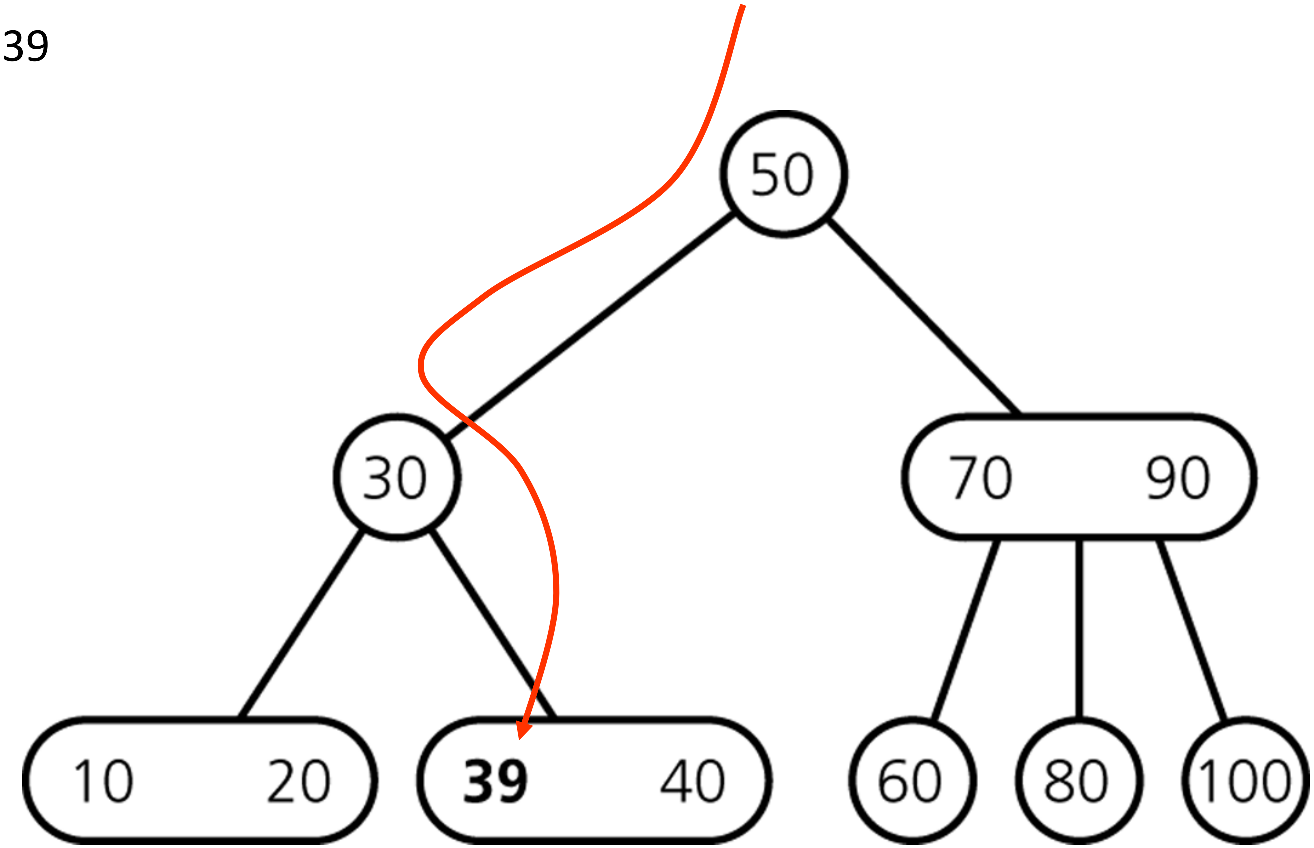
both trees after inserting items 39, 38, ... 32

2-3 Tree



Inserting Items

Insert 39



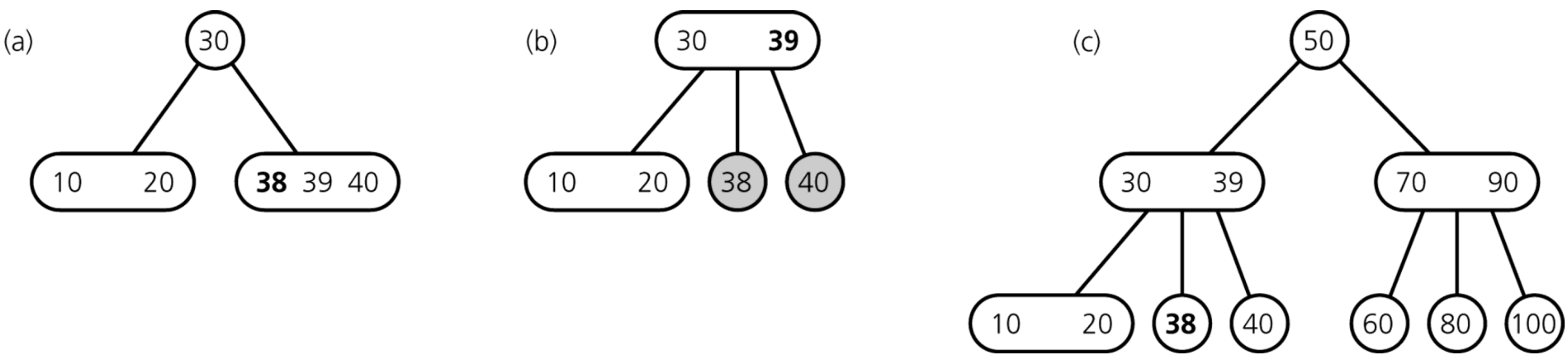
Inserting Items

Insert 38

insert in leaf

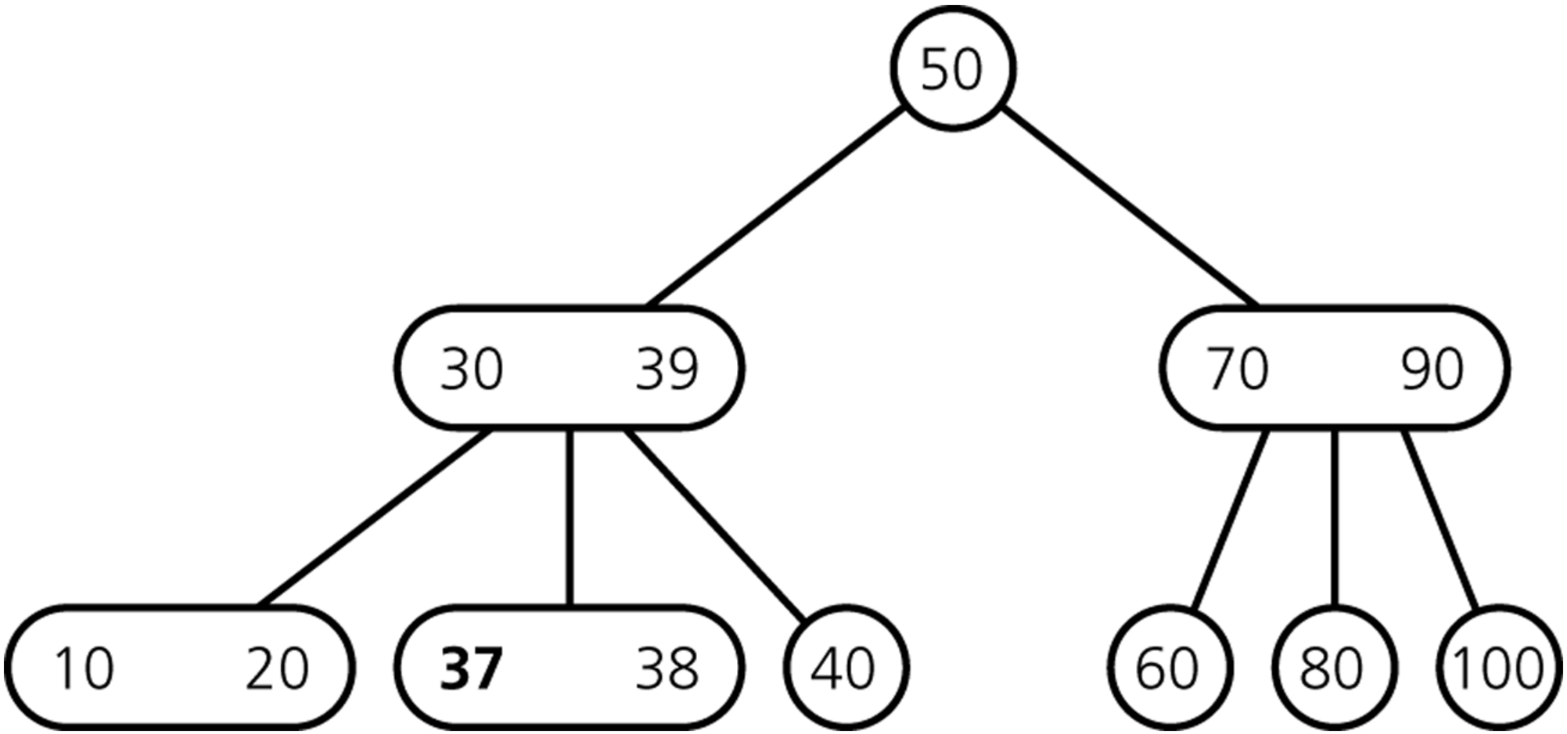
divide leaf
and move middle
value up to parent

result



Inserting Items

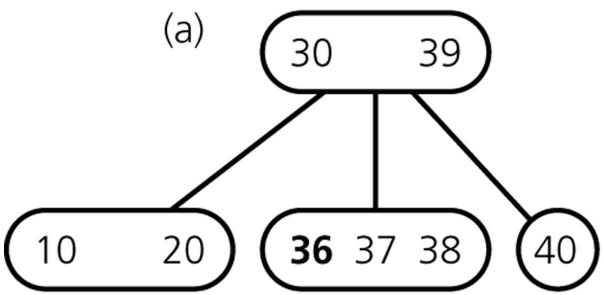
Insert 37



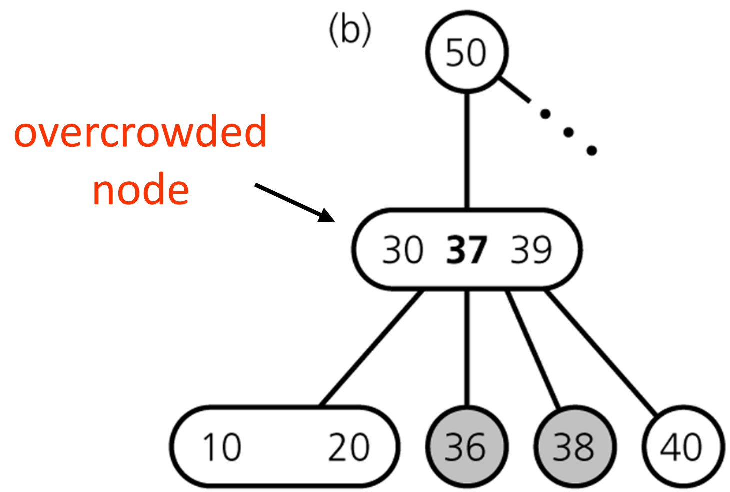
Inserting Items

Insert 36

insert in leaf



divide leaf
and move middle
value up to parent

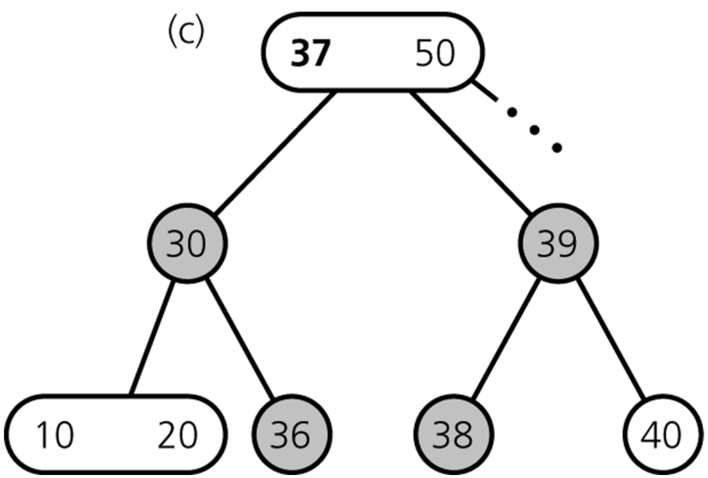


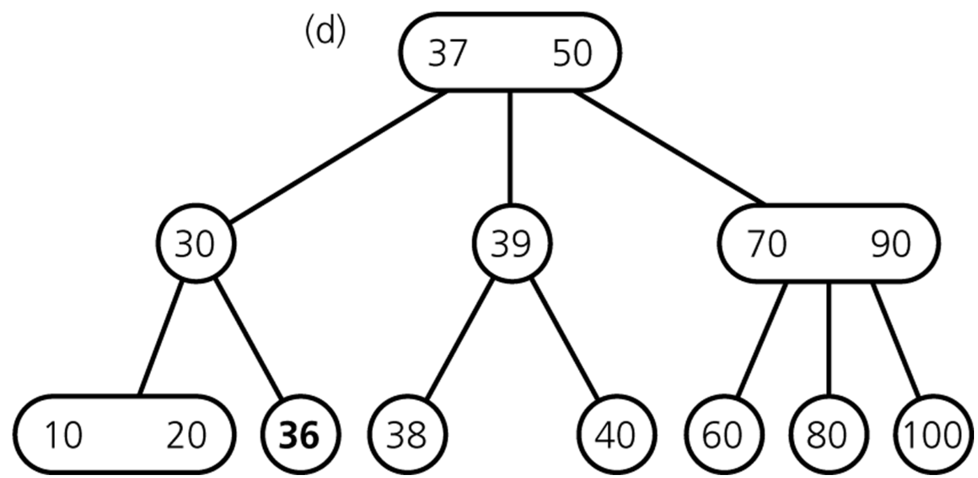
Inserting Items

... still inserting 36

divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

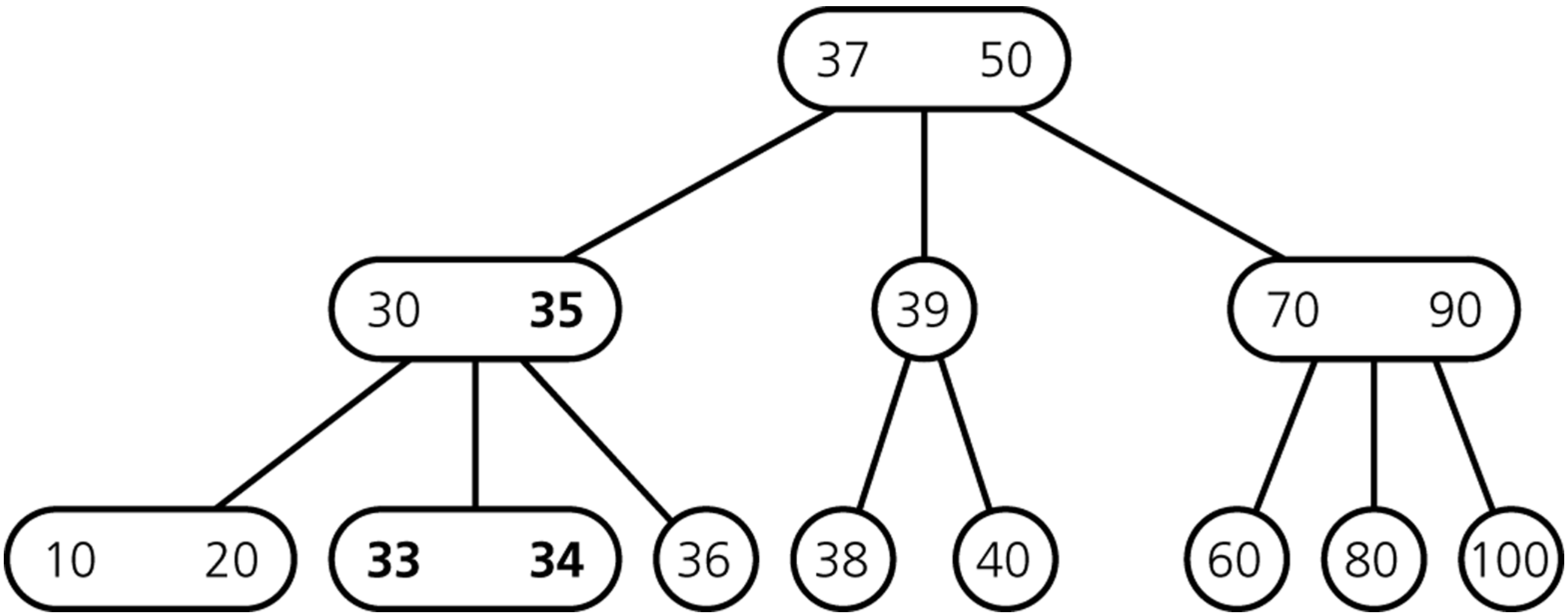
result



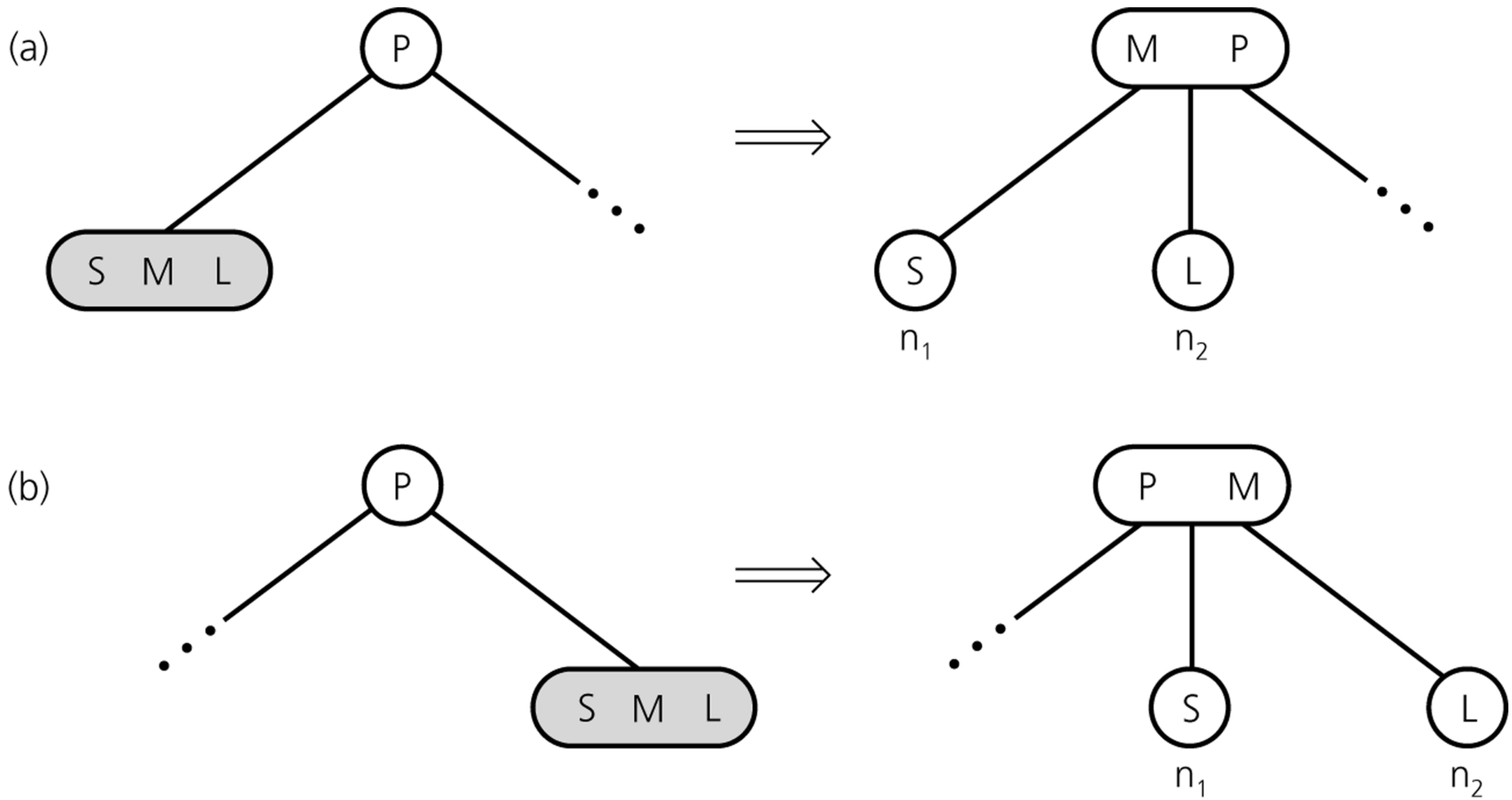


Inserting Items

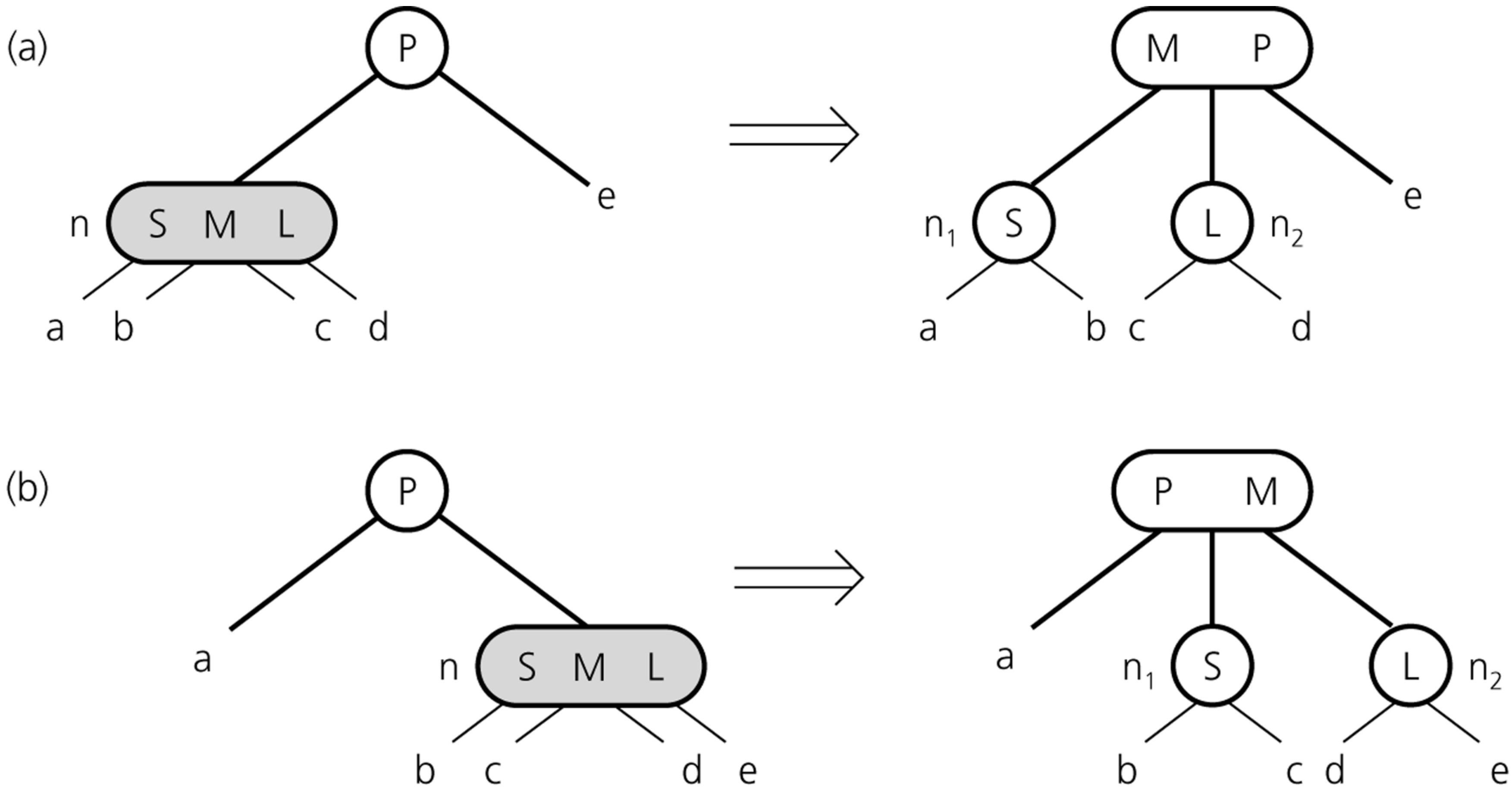
After Insertion of 35, 34, 33



Inserting so far

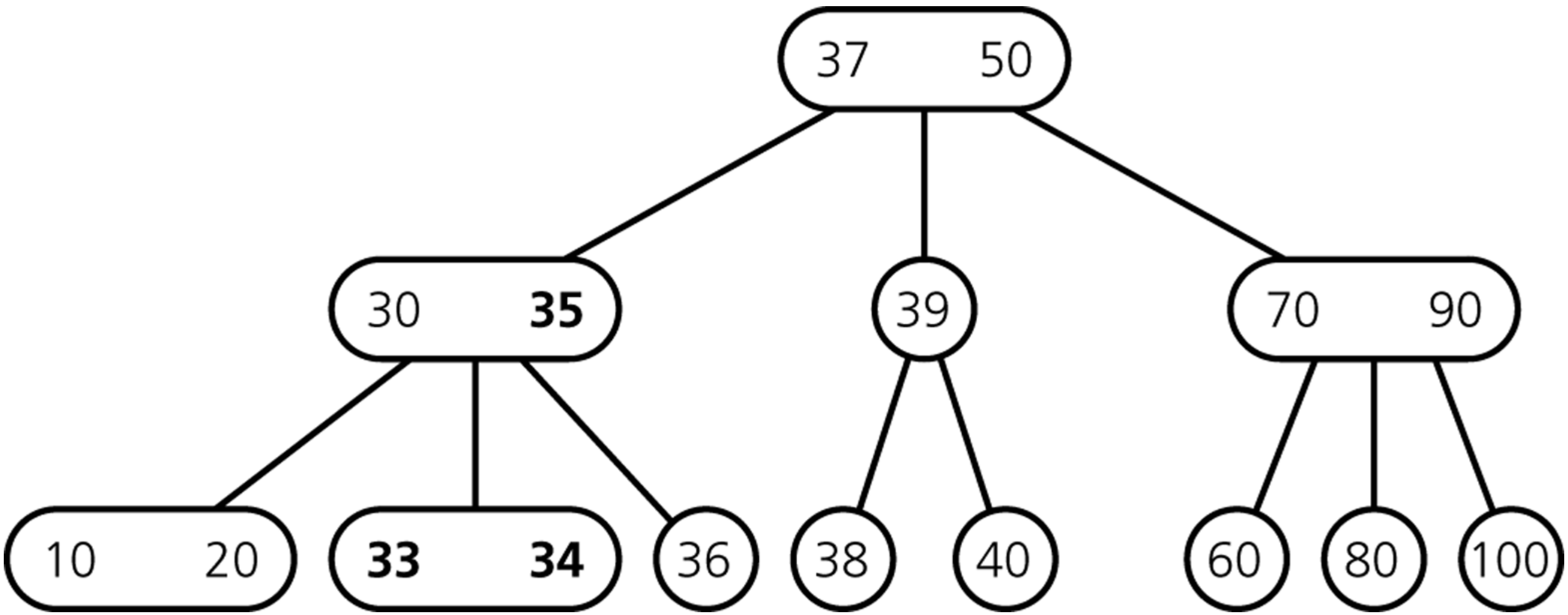


Inserting so far



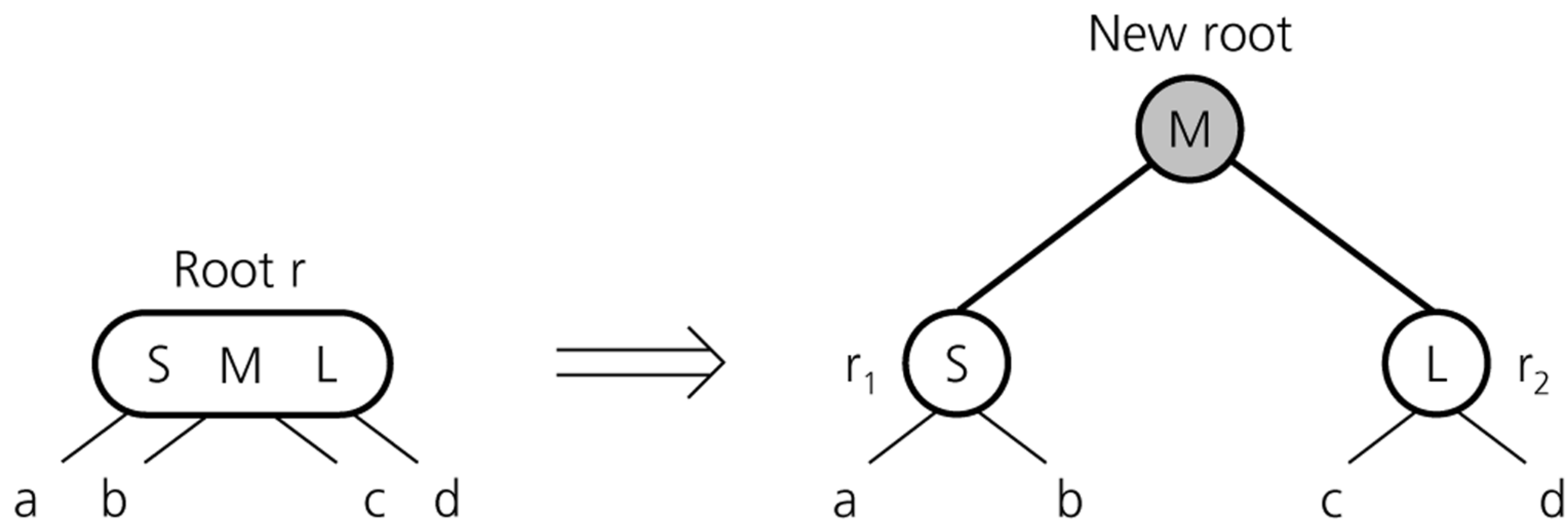
Inserting Items

How do we insert 32?



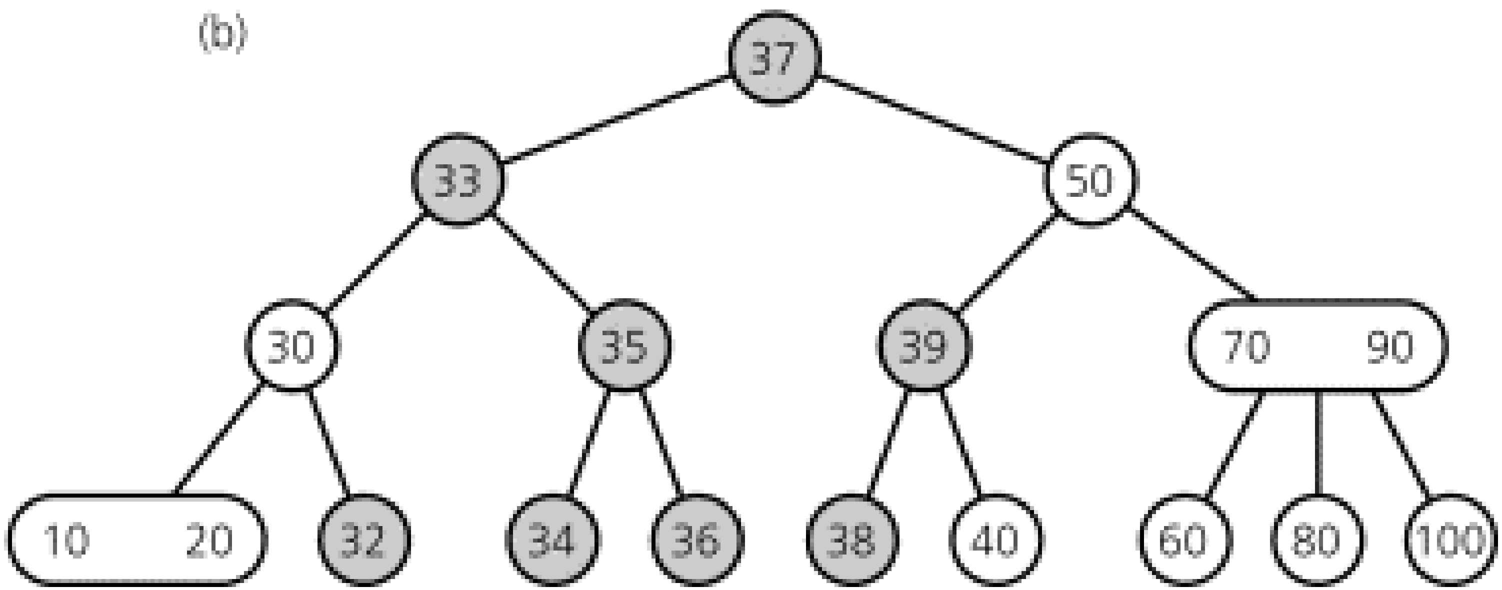
Inserting Items

→ creating a new root if necessary
→ tree grows at the root



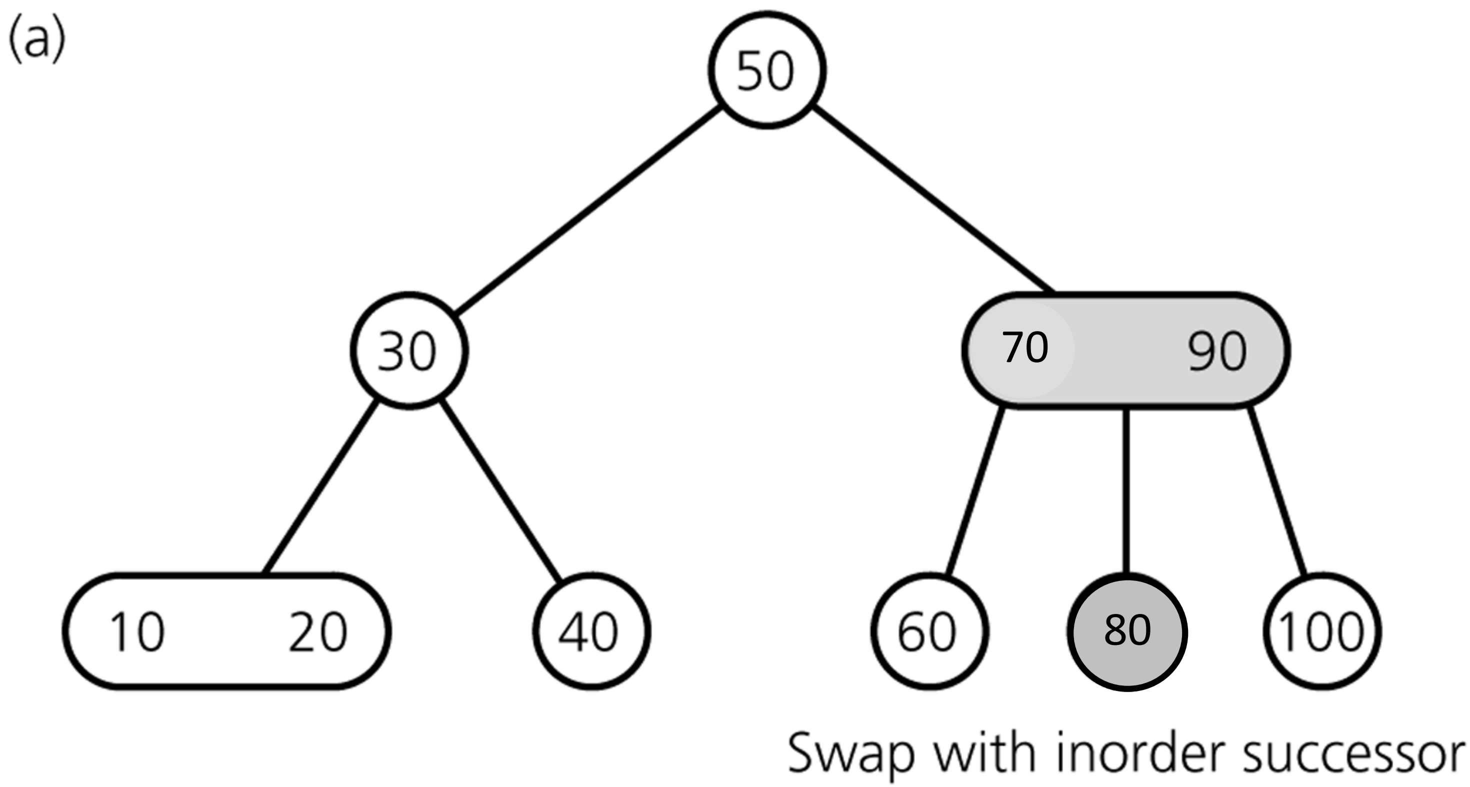
Inserting Items

Final Result



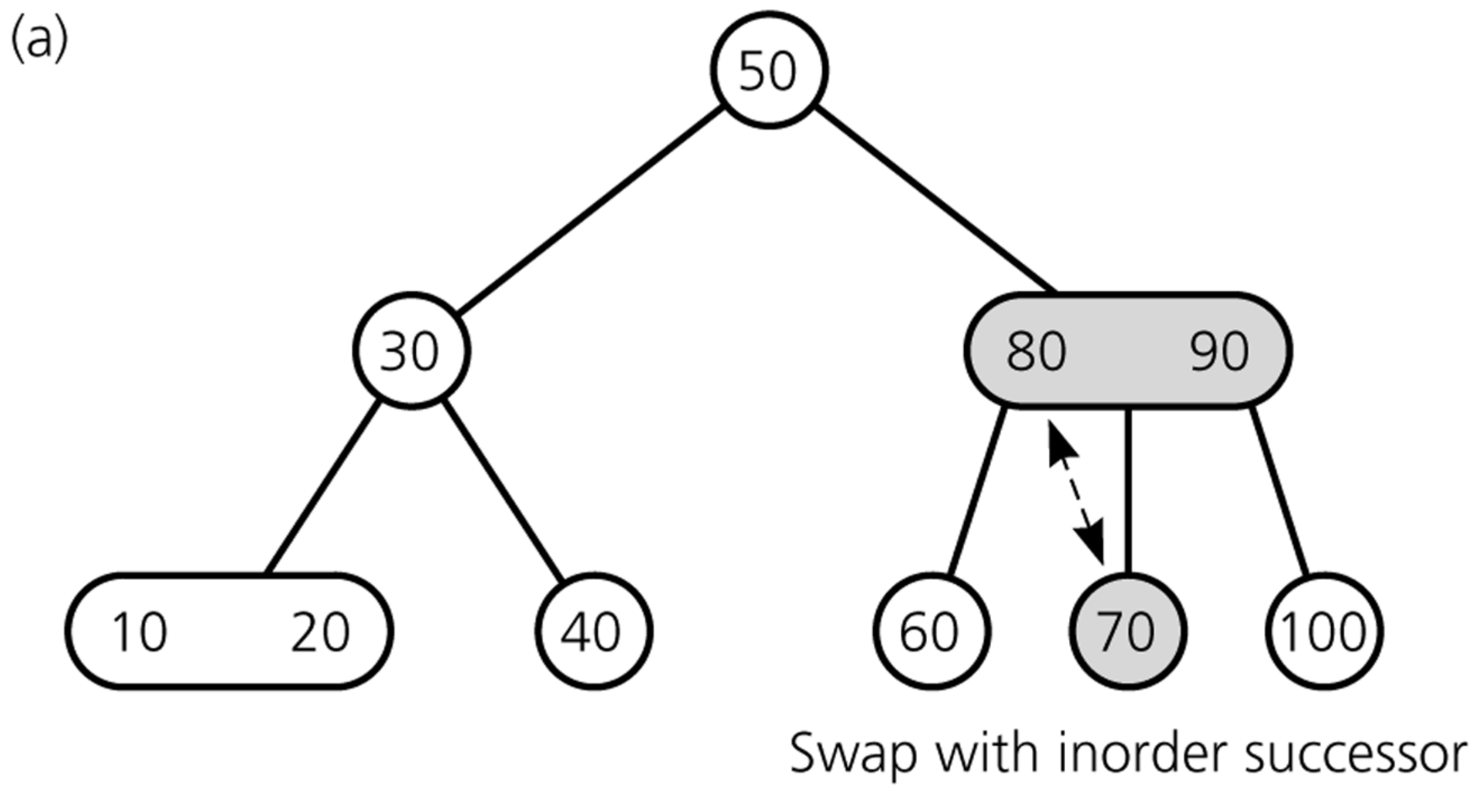
Deleting Items

Delete 70



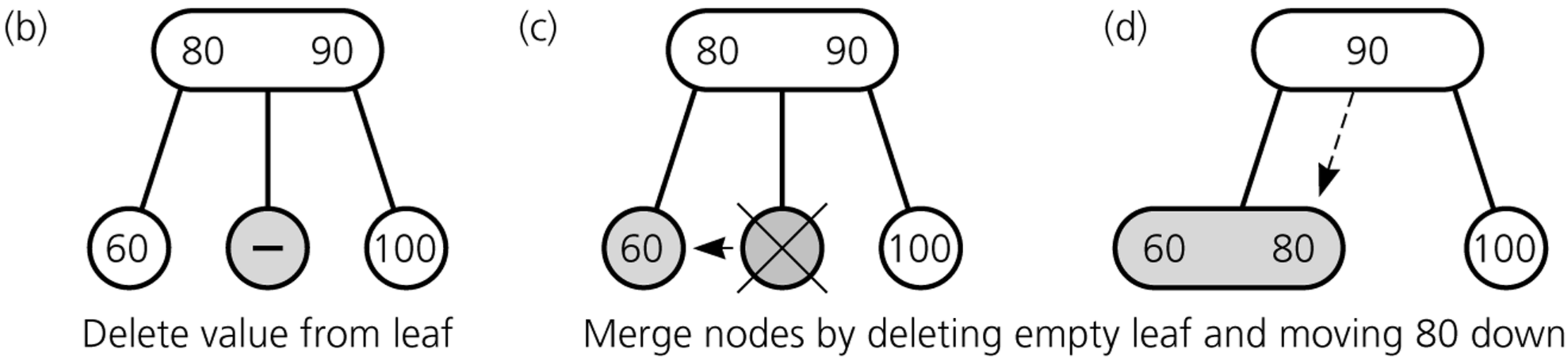
Deleting Items

Deleting 70: swap 70 with inorder successor (80)



Deleting Items

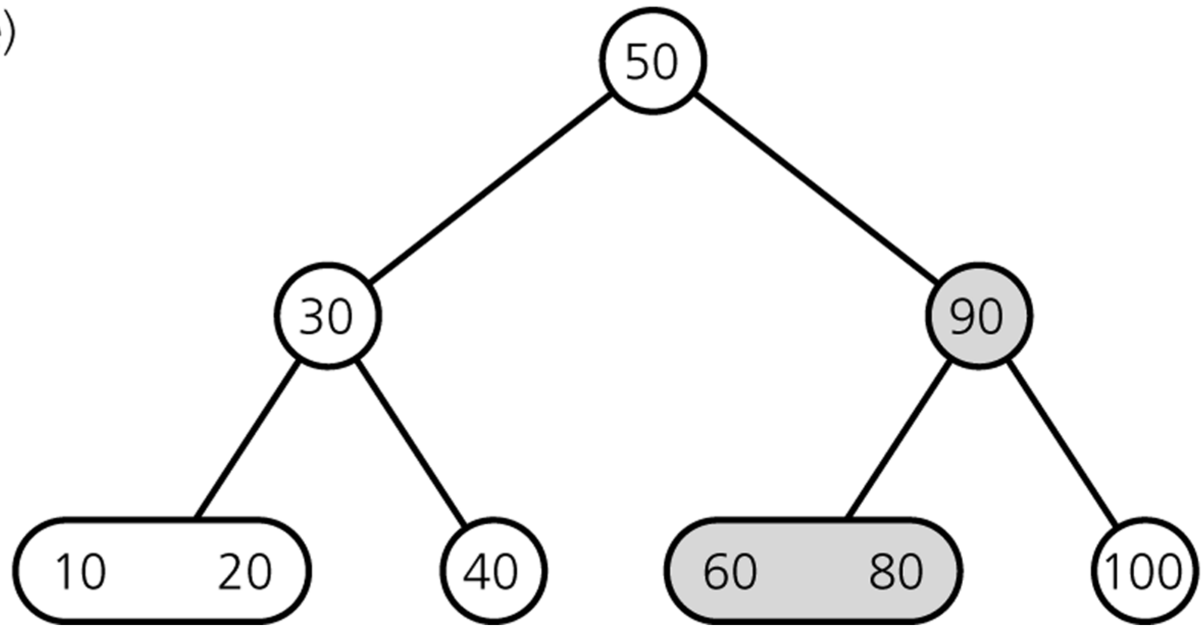
Deleting 70: ... get rid of 70



Deleting Items

Result

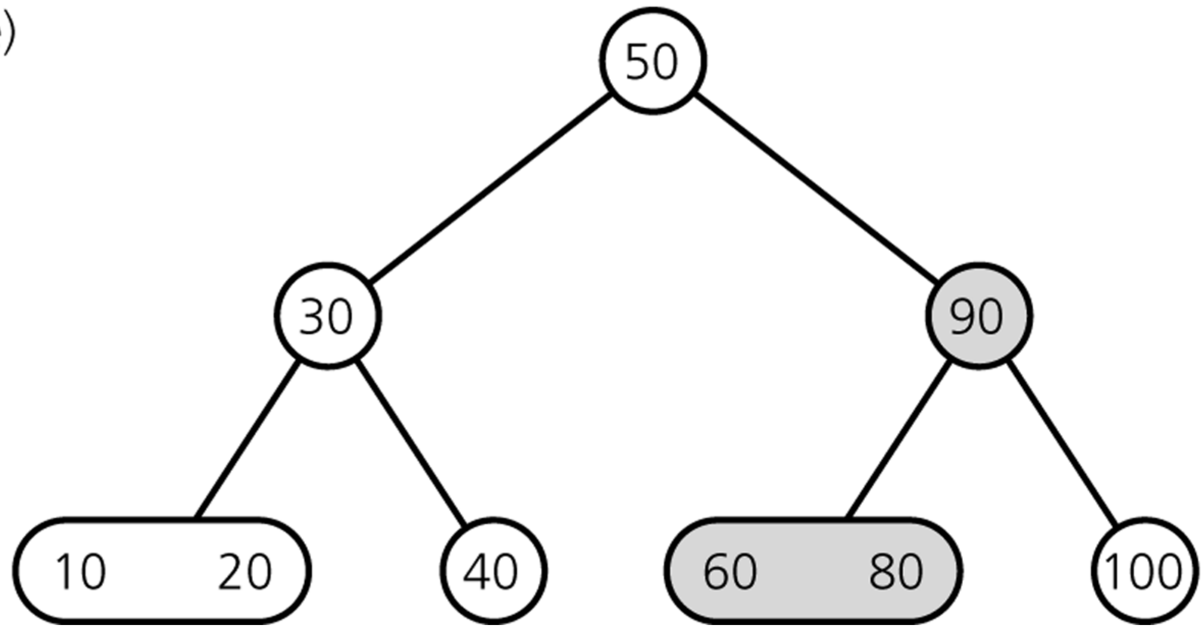
(e)



Deleting Items

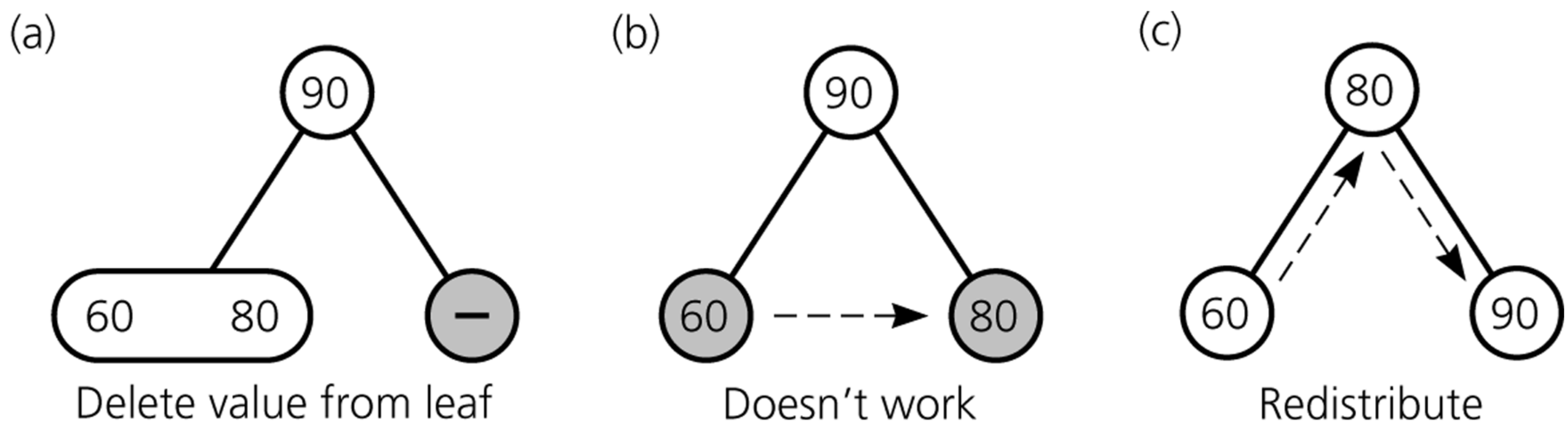
Delete 100

(e)



Deleting Items

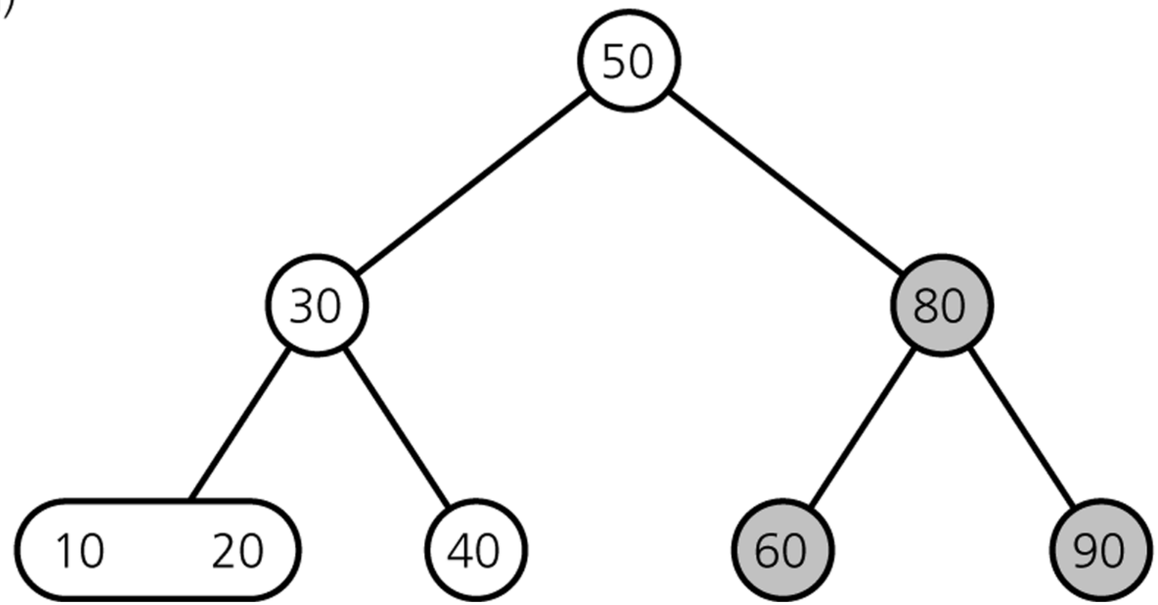
Deleting 100



Deleting Items

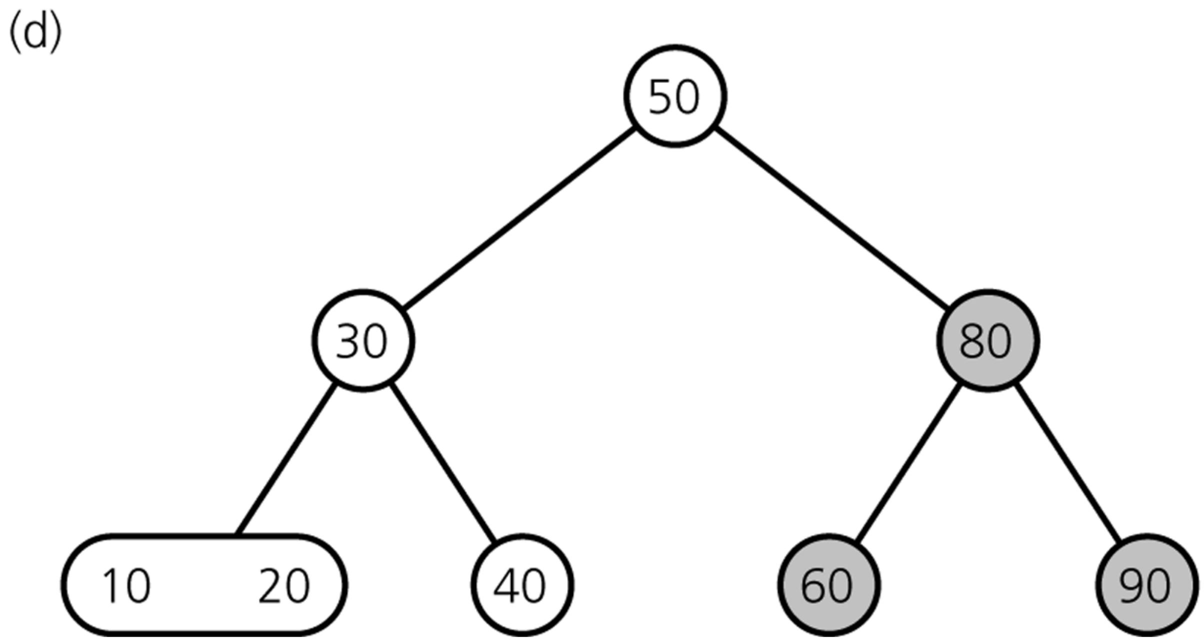
Result

(d)



Deleting Items

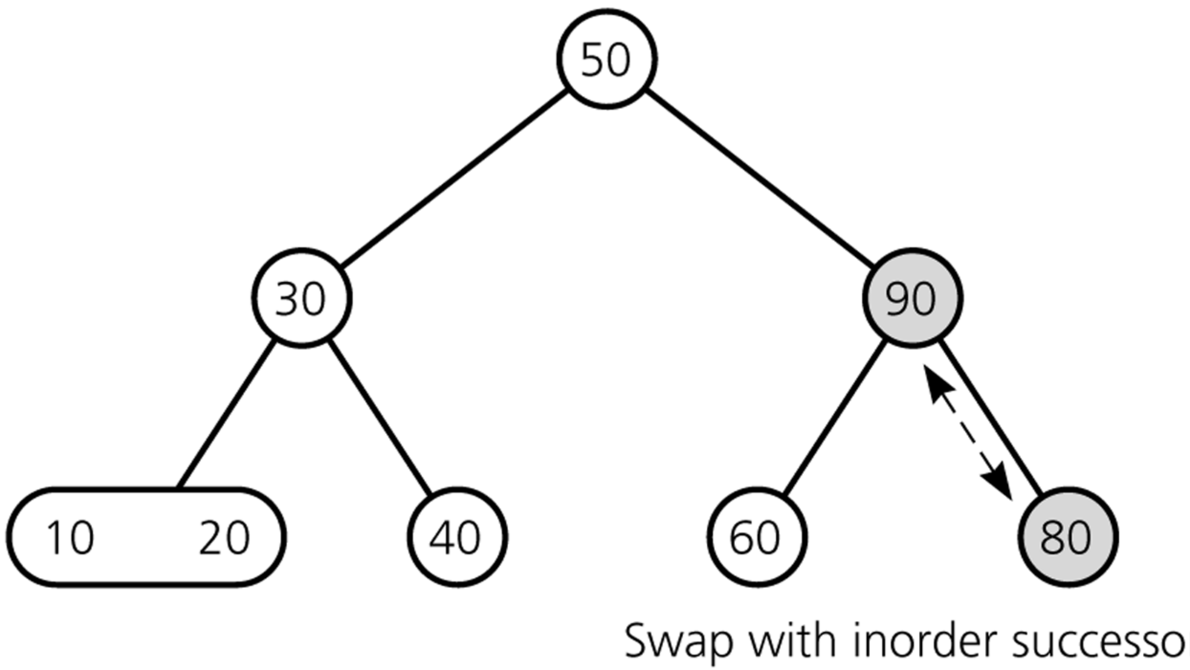
Delete 80



Deleting Items

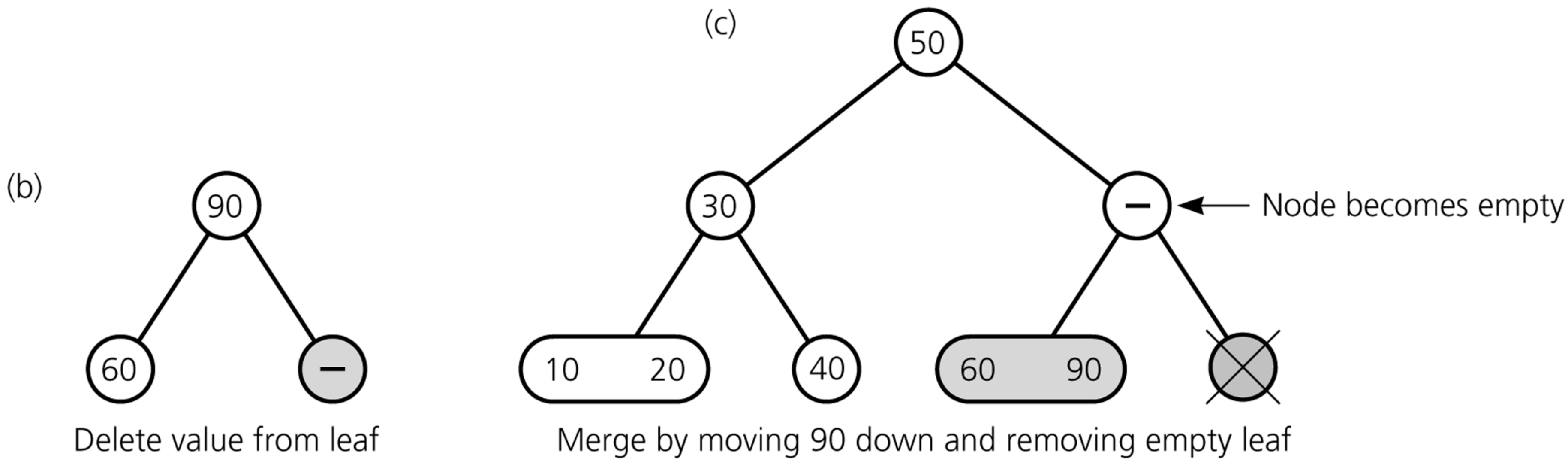
Deleting 80 ...

(a)



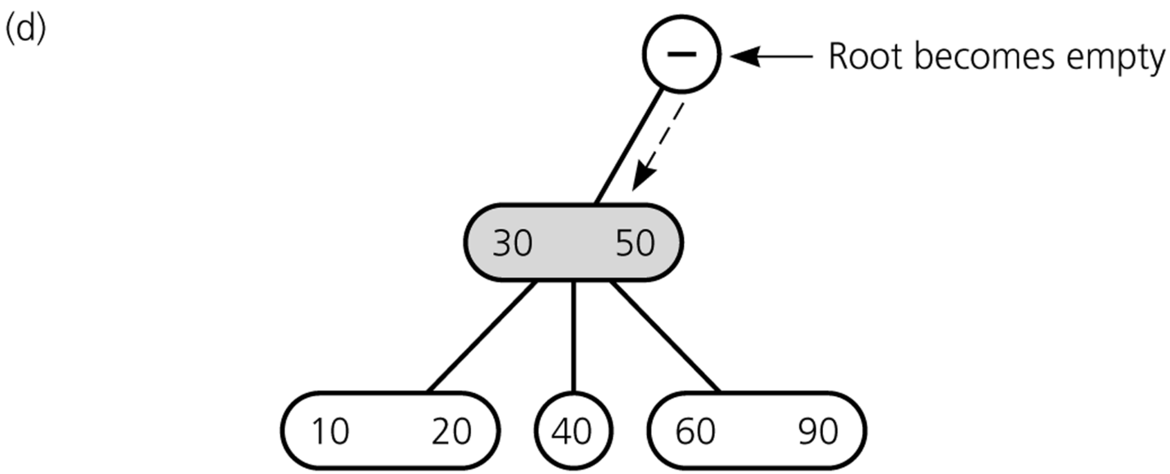
Deleting Items

Deleting 80 ...

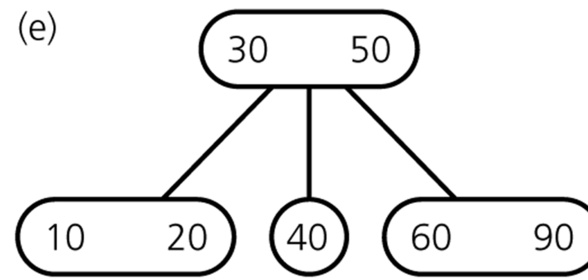


Deleting Items

Deleting 80 ...



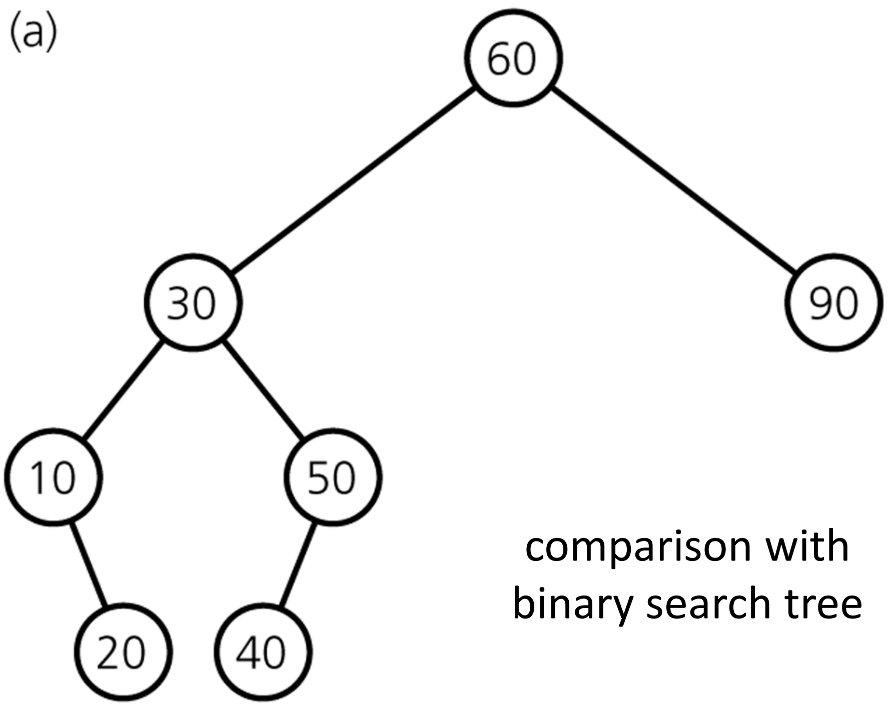
Merge: move 50 down, adopt empty leaf's child, remove empty node



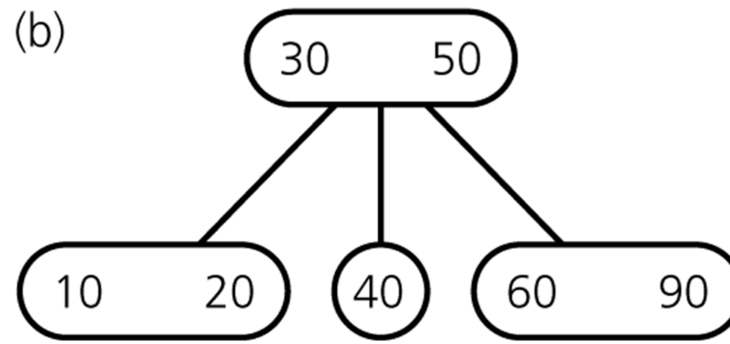
Remove empty root

Deleting Items

Final Result



comparison with
binary search tree



Deletion Algorithm I

Deleting item *I*:

1. Locate node *n*, which contains item *I*
2. If node *n* is not a leaf → swap *I* with inorder successor

→ deletion always begins at a leaf

3. If leaf node *n* contains another item, just delete item *I*
   else
   try to redistribute nodes from siblings (see next slide)
   if not possible, merge node (see next slide)

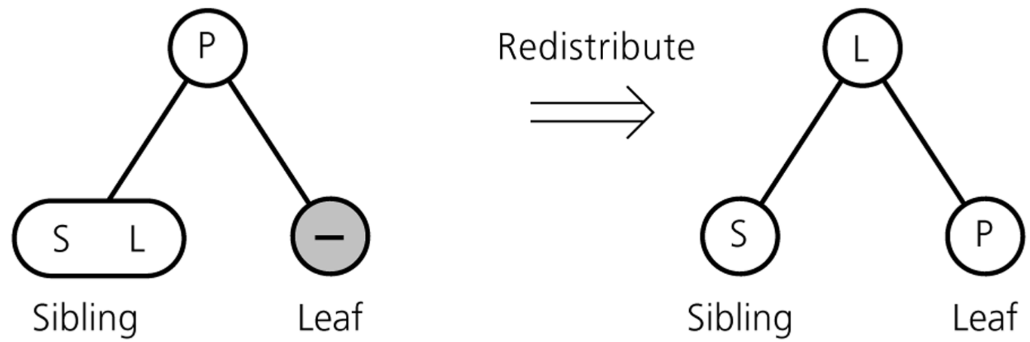
Deletion Algorithm II

Redistribution

A sibling has 2 items:

→ redistribute item between siblings and parent

(a)



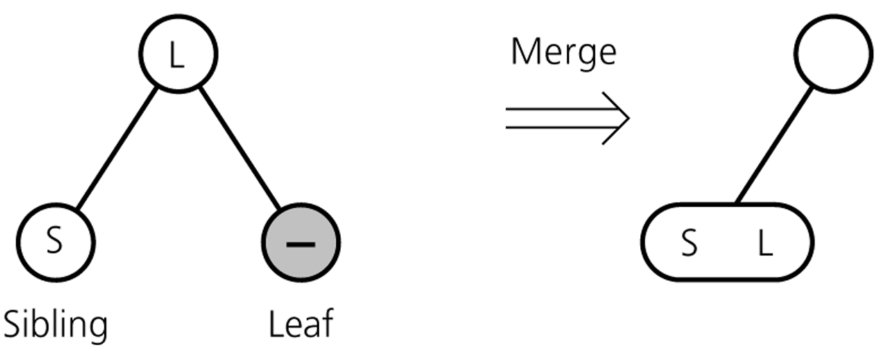
Merging

No sibling has 2 items:

→ merge node

→ move item from parent to sibling

(b)



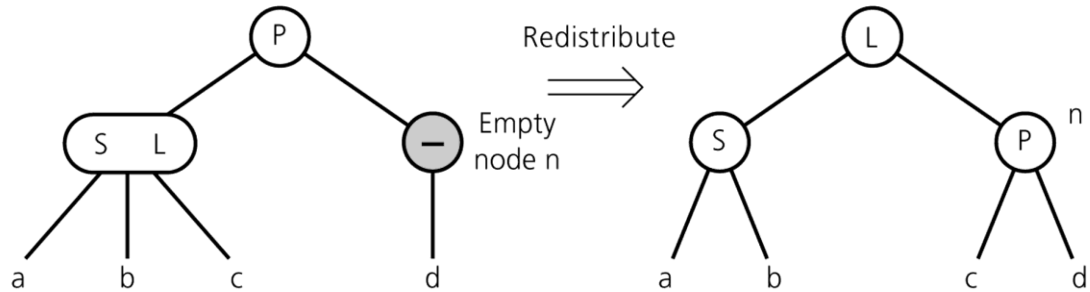
Deletion Algorithm III

Redistribution

Internal node *n* has no item left

→ redistribute

(c)

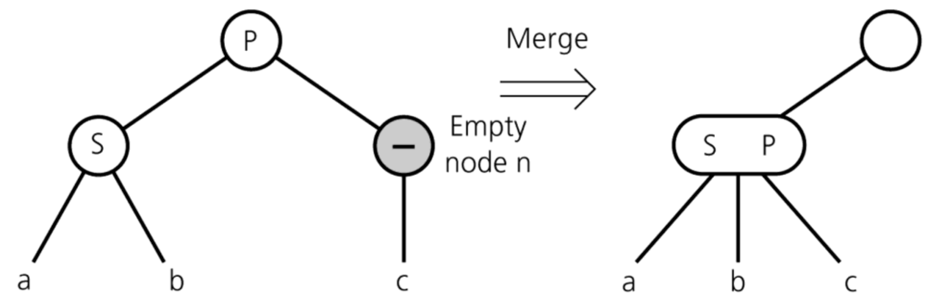


Merging

Redistribution not possible:

→ merge node
→ move item from parent to sibling
→ adopt child of *n*

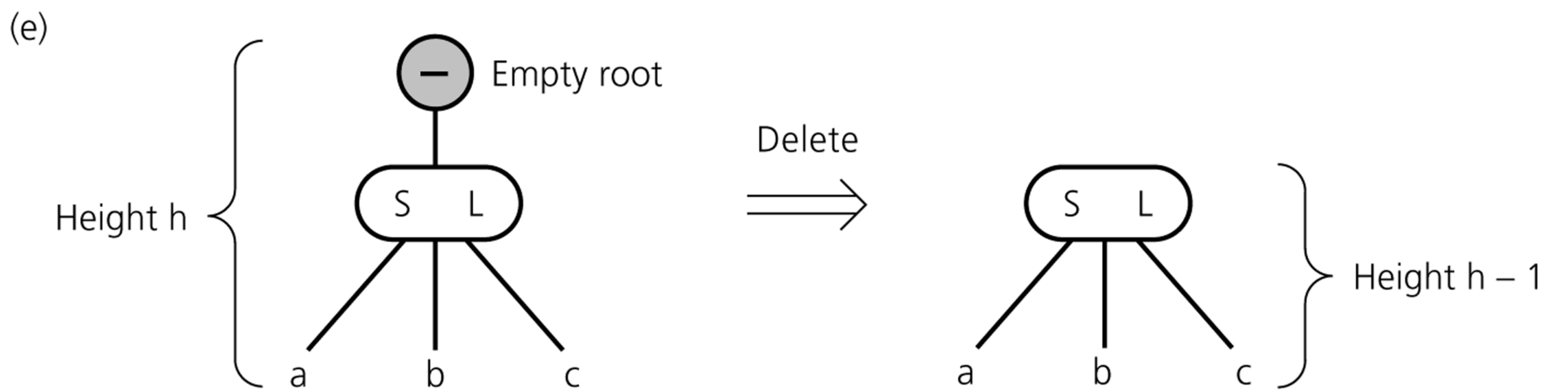
(d)



If *n*'s parent ends up without item, apply process recursively

Deletion Algorithm IV

If merging process reaches the root and root is without item
→ delete root



Operations of 2-3 Trees

all operations have time complexity of log n

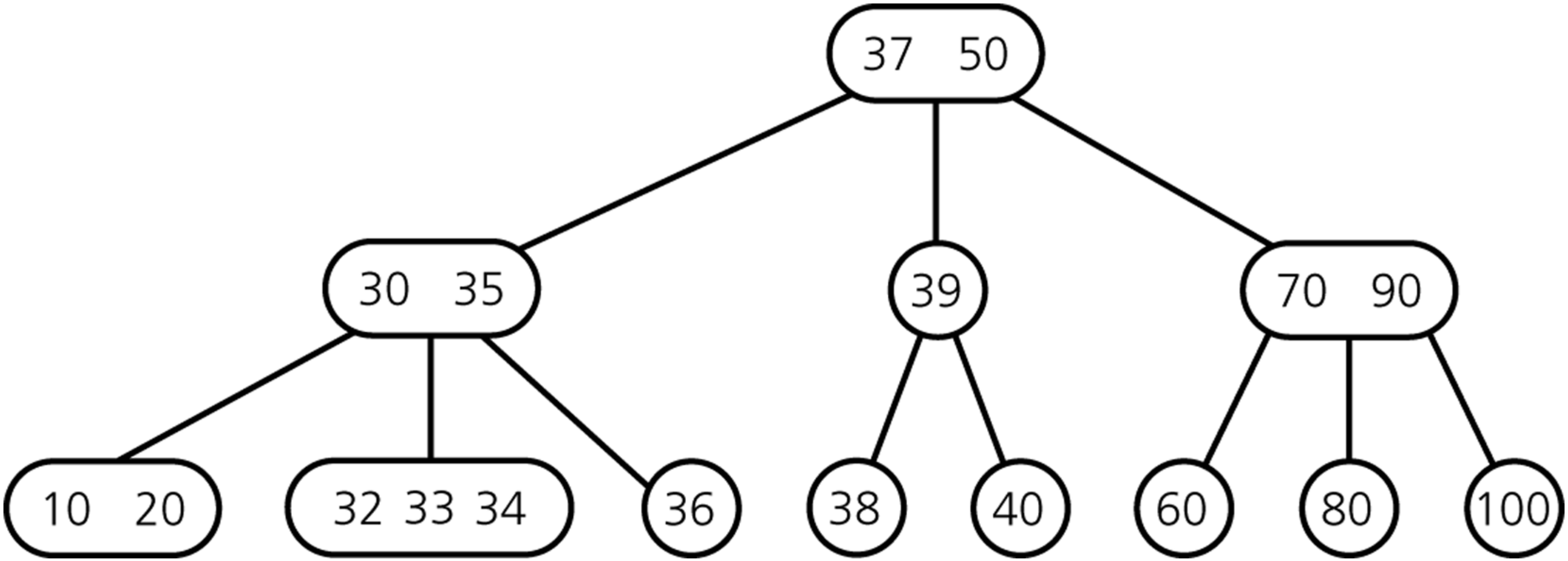
2-3-4 Trees

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

- 2-3-4 trees are also known as 2-4 trees in other books.
  - A specialization of M-way tree (M=4)
  - Sometimes also called 4th order B-trees
  - Variants of B-trees are very useful in databases and file systems
    - MySQL, Oracle, MS SQL all use B+ trees for indexing
    - Many file systems (NTFS, Ext2FS etc.) use B+ trees for indexing metadata (file size, date etc.)

- Although a 2-3-4 tree has more efficient insertion and deletion operations than a 2-3 tree, a 2-3-4 tree has greater storage requirements.

2-3-4 Trees -- Example

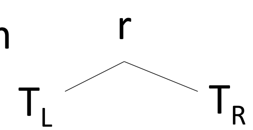


2-3-4 Trees

T is a 2-3-4 tree of height h if

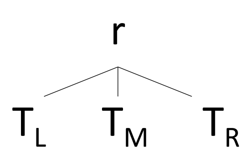
1. T is empty (a 2-3-4 tree of height 0), or

1. T is of the form



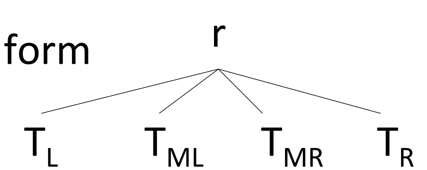
where r is a node containing one data item and T_L and T_R are both 2-3-4 trees, each of height h-1, or

3. T is of the form

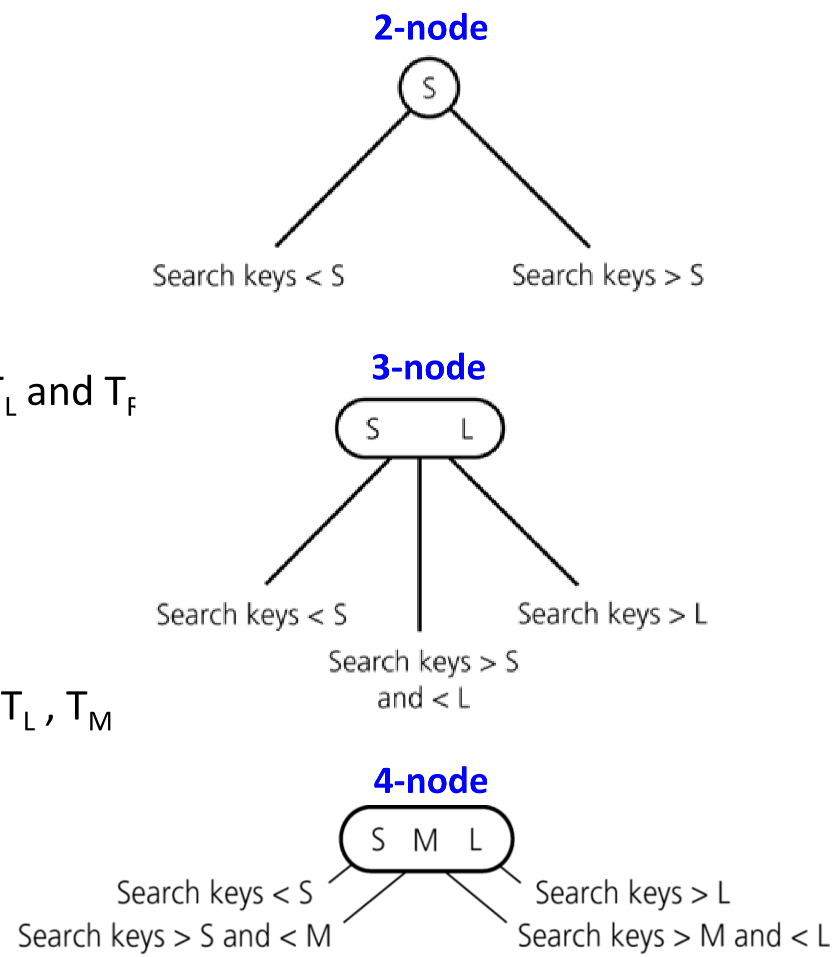


where r is a node containing two data items and T_L , T_M and T_R are 2-3-4 trees, each of height h-1, or

4. T is of the form



where r is a node containing three data items and T_L , T_{ML} , T_{MR} , and T_R are 2-3-4 trees, each of height h-1.



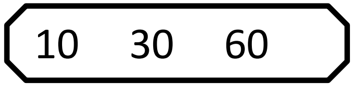
2-3-4 Trees -- Operations

- Searching and traversal algorithms for a 2-3-4 tree are similar to the 2-3 algorithms.
- For a 2-3-4 tree, insertion and deletion algorithms that are used for 2-3 trees, can similarly be used.
- But, we can also use a slightly different insertion and deletion algorithms for 2-3-4 trees to gain some efficiency.

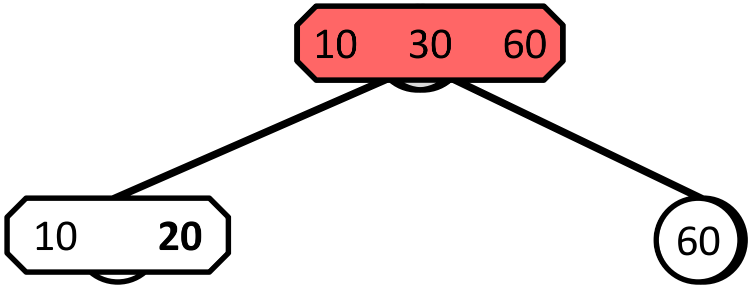
Inserting into a 2-3-4 Tree

- Splits 4-nodes by moving one of its items up to its parent node.
- For a 2-3 tree, the insertion algorithm traces a path from the root to a leaf and then backs up from the leaf as it splits nodes.
- *To avoid this return path after reaching a leaf*, the insertion algorithm for a 2-3-4 tree splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf.
  - As a result, when a 4-node is split and an item is moved up to node's parent, the parent cannot possibly be a 4-node and so can accommodate another item.

Insert[20 50 40 70 80 15 90 100] to this 2-3-4 tree



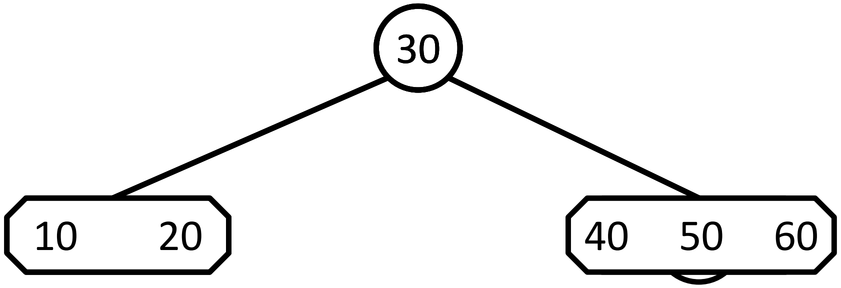
Inserting into a 2-3-4 Tree -- Example



Insert 20

- Root is a 4-node →**Split 4-nodes as they are encountered**
- So, we split it before insertion
- And, then add 20

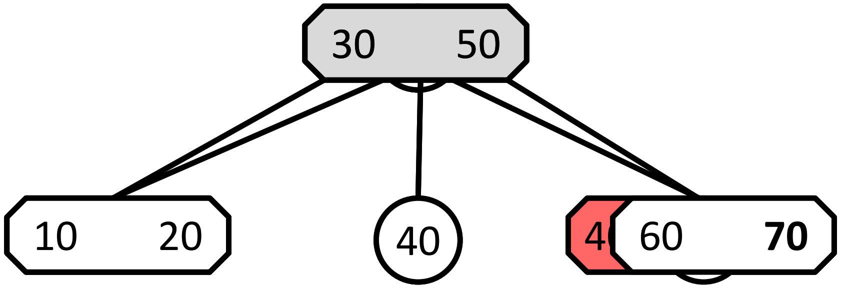
Inserting into a 2-3-4 Tree -- Example



Insert 50 and 40

- No 4-nodes have been encountered **→ No split operation** during their insertion

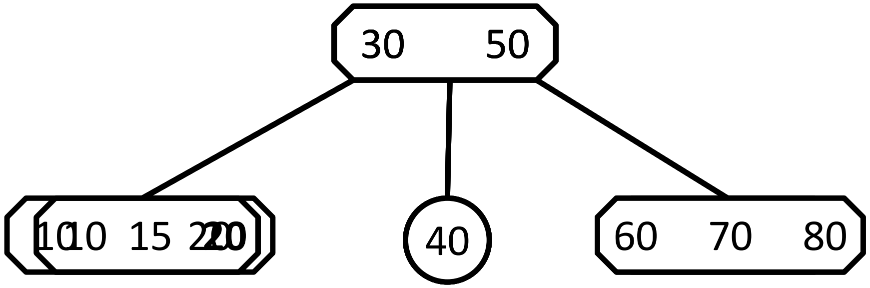
Inserting into a 2-3-4 Tree -- Example



Insert 70

- A 4-node is encountered
- So, we split it before insertion
- And, then add 70

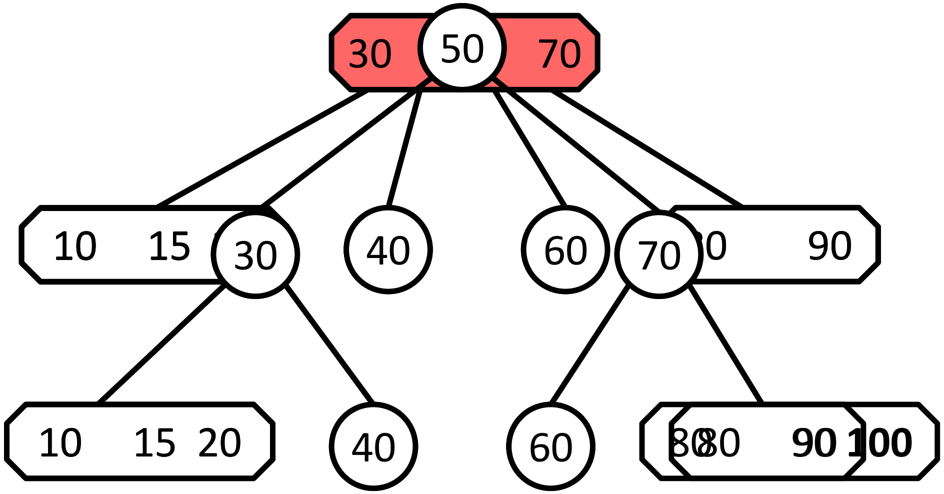
Inserting into a 2-3-4 Tree -- Example



Insert 80 and 15

- No 4-nodes have been encountered **→ No split operation** during their insertion

Inserting into a 2-3-4 Tree -- Example



Insert 100

- A 4-node is encountered
- So, we split it before insertion
- And, then add 100

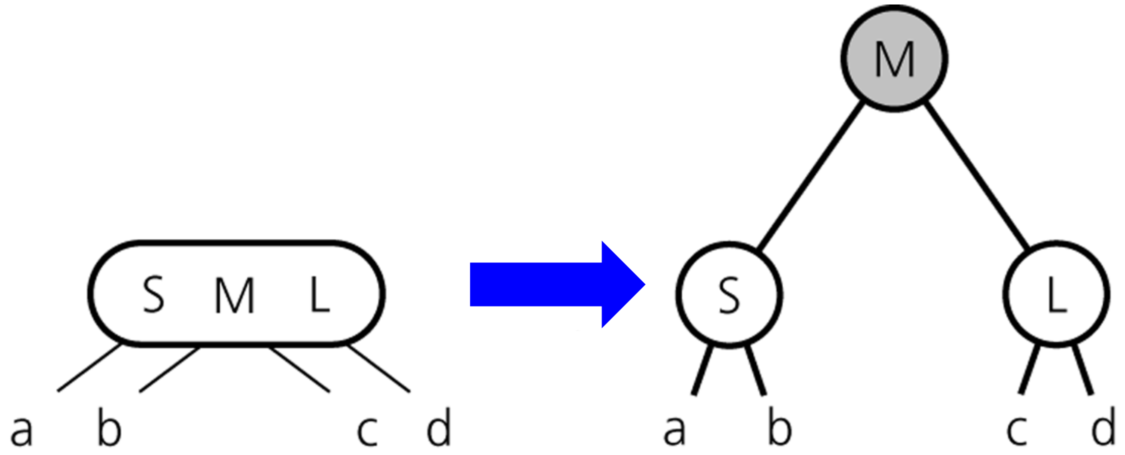
Splitting 4-nodes during insertion

- We split each 4-node as soon as we encounter it during our search from the root to a leaf that will accommodate the new item to be inserted.

- The 4-node which will be split can:
  - be the root, or
  - have a 2-node parent, or
  - have a 3-node parent.

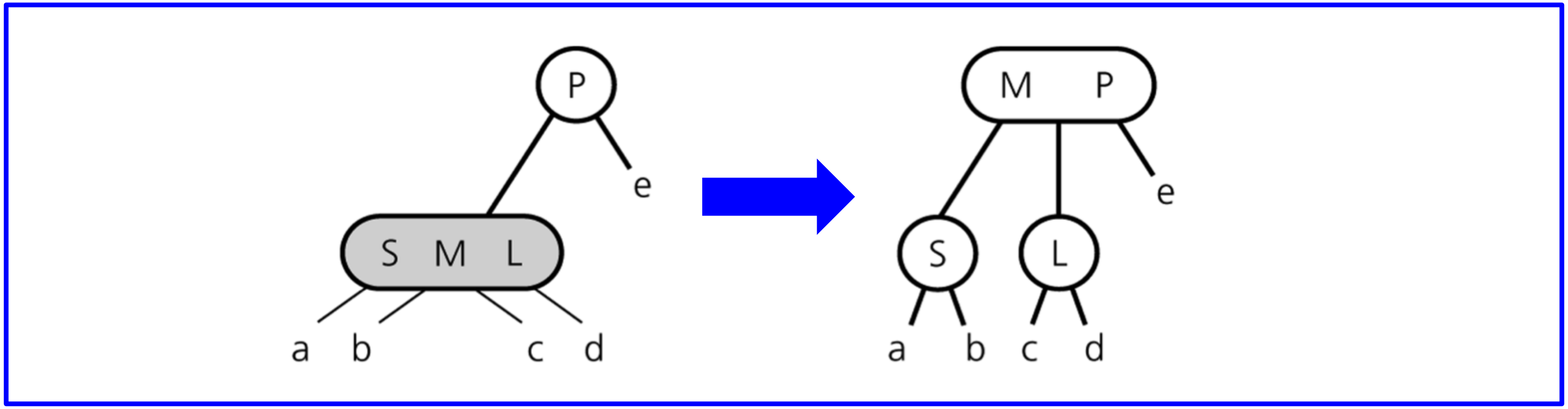
Splitting 4-nodes during insertion

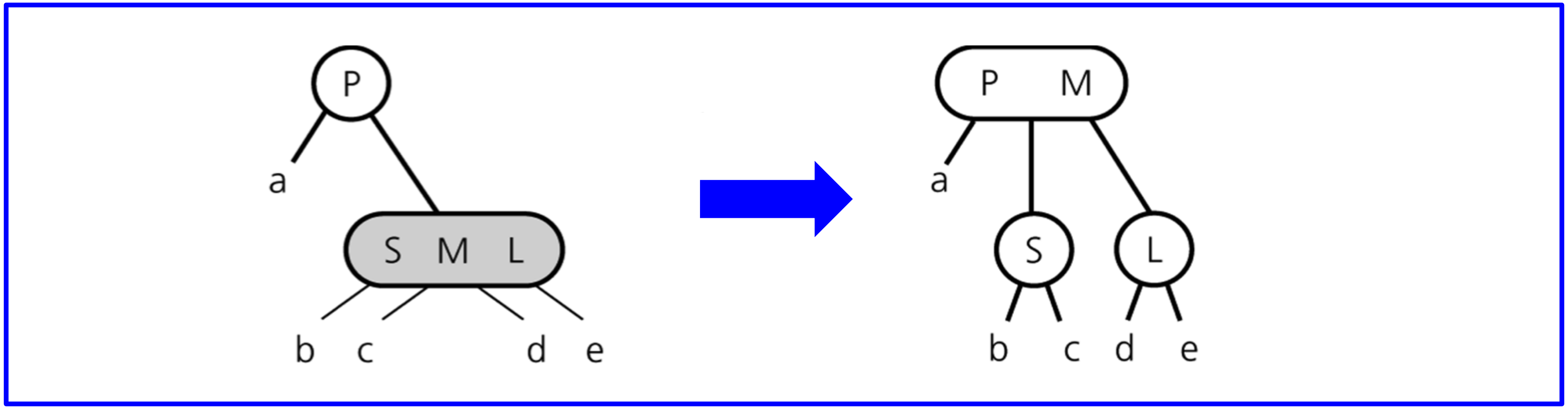
Splitting a 4-node root



Splitting 4-nodes during insertion

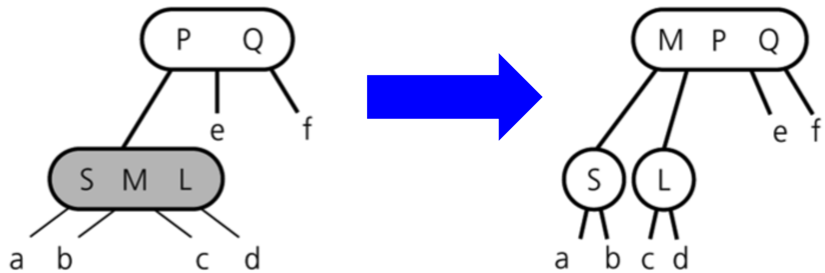
Splitting a 4-node whose parent is a 2-node

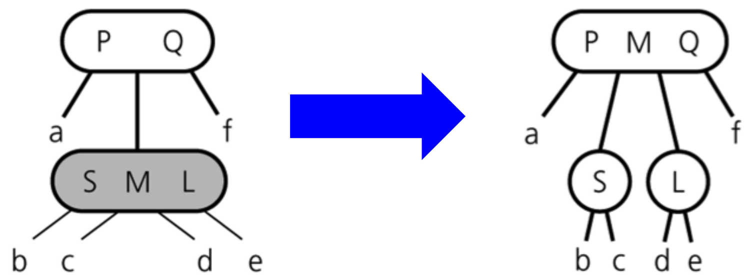


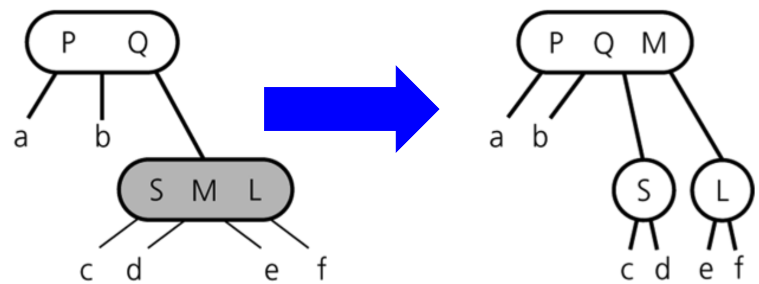


Splitting 4-nodes during insertion

Splitting a 4-node whose parent is a 3-node







Deleting from a 2-3-4 tree

- For a 2-3 tree, the deletion algorithm traces a path from the root to a leaf and then backs up from the leaf, fixing empty nodes on the path back up to root.

- *To avoid this return path after reaching a leaf*, the deletion algorithm for a 2-3-4 tree transforms each 2-node into either 3-node or 4-node as soon as it encounters them on the way down the tree from the root to a leaf.
  - If an adjacent sibling is a 3-node or 4-node, transfer an item from that sibling to our 2-node.
  - If adjacent sibling is a 2-node, merge them.