02110

Inge Li Gørtz

Thank you to Kevin Wayne for inspiration to slides

Balanced Search Trees

2-3-4 trees
red-black trees

Balanced search trees

Dynamic sets

- Search
- Insert
- Delete
- Maximum
- Minimum
- Successor
- Predecessor

This lecture: 2-3-4 trees, red-black trees

Next time: Splay trees

Dynamic set implementations

Worst case running times

Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
ordered array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(log n)	O(log n)
BST	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)

In worst case h=n.

In best case h= log n (fully balanced binary tree)

Today: How to keep the trees balanced.

2-3-4 trees

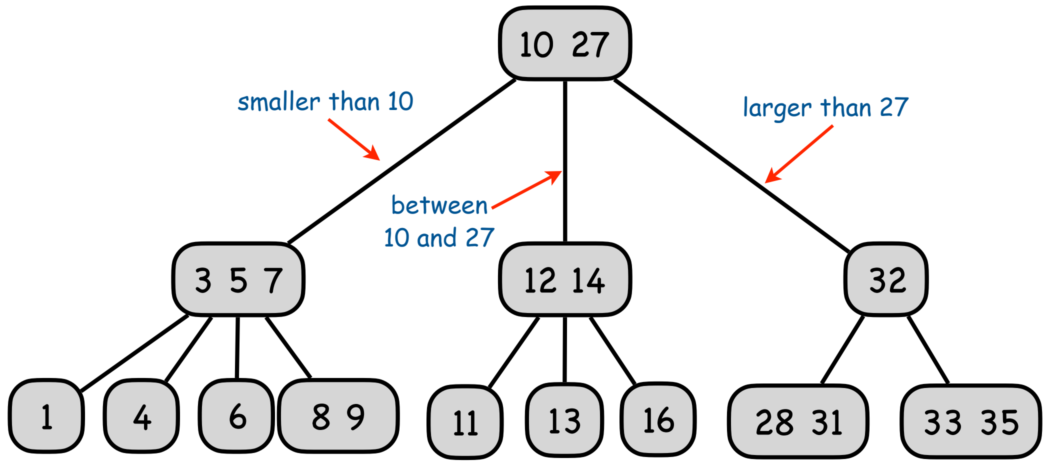
2-3-4 trees

2-3-4 trees. Allow nodes to have multiple keys.

Perfect balance. Every path from root to leaf has same length.

Allow 1, 2, or 3 keys per node

- 2-node: one key, 2 children
- 3-node: 2 keys, 3 children
- 4-node: 3 keys, 4 children

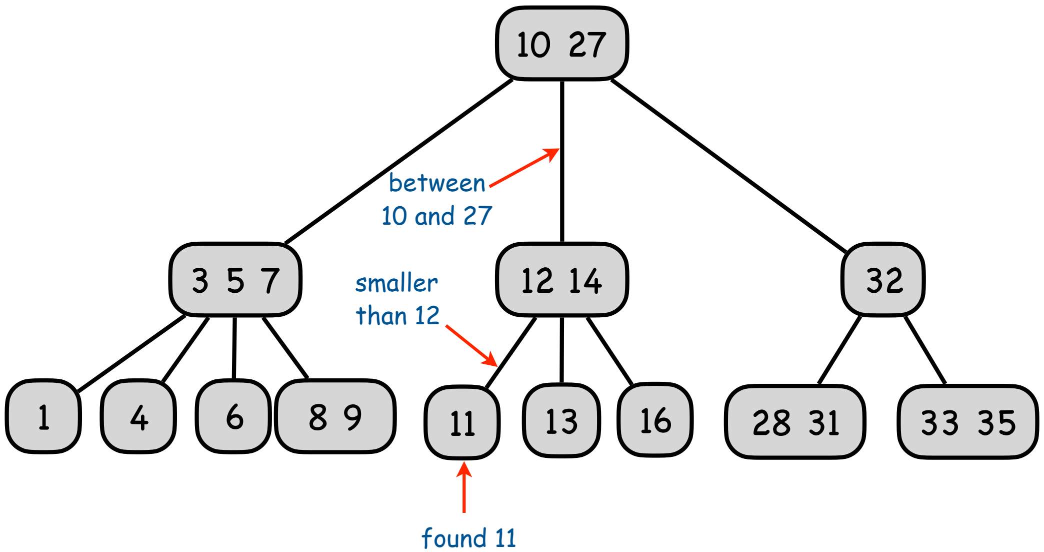


Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for 11

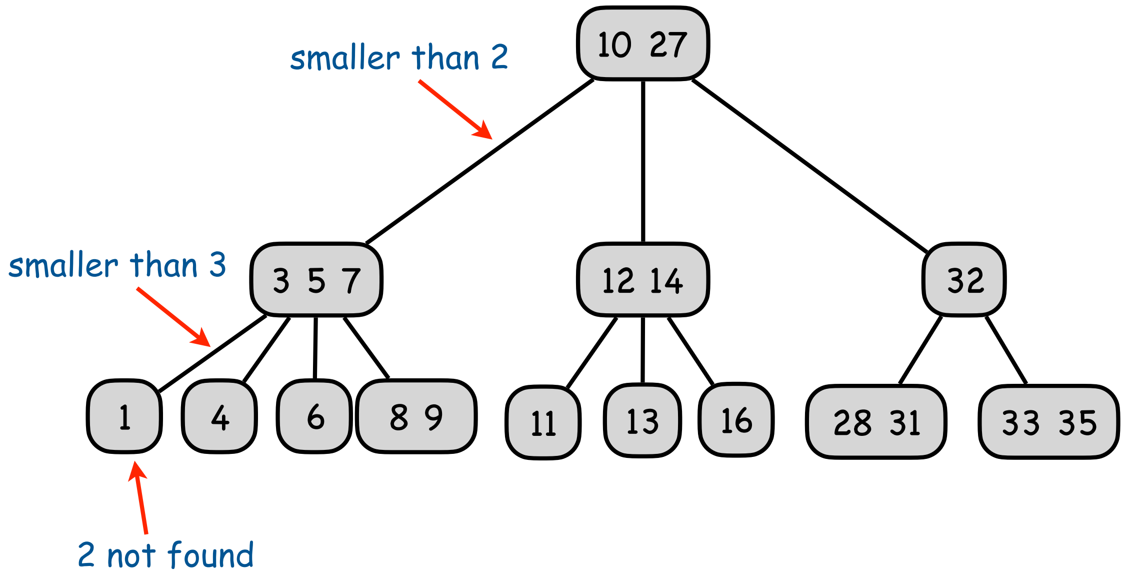


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.

Ex. Insert 2

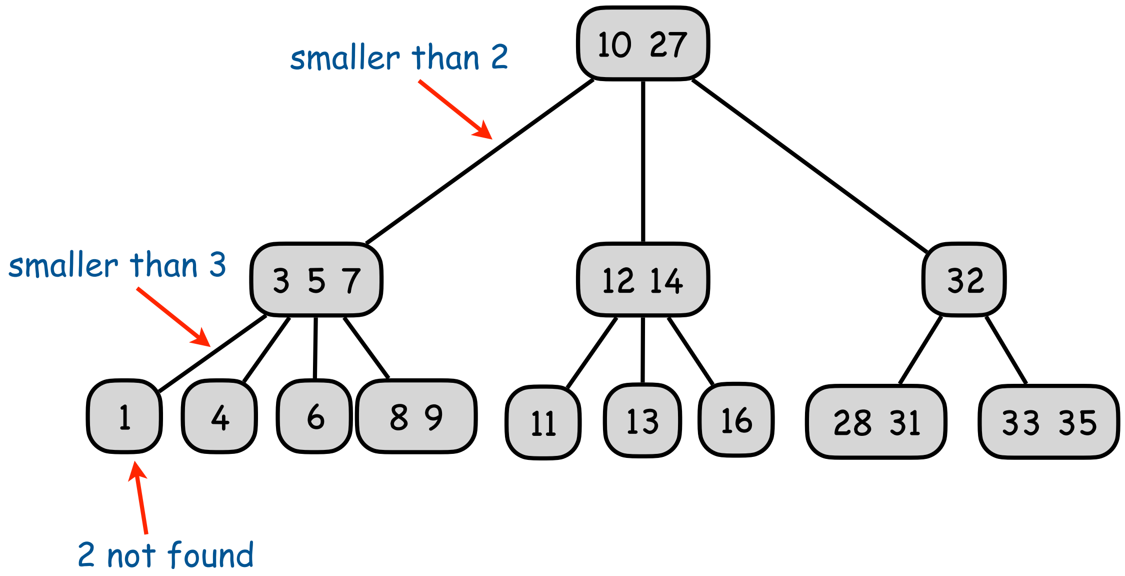


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert 2

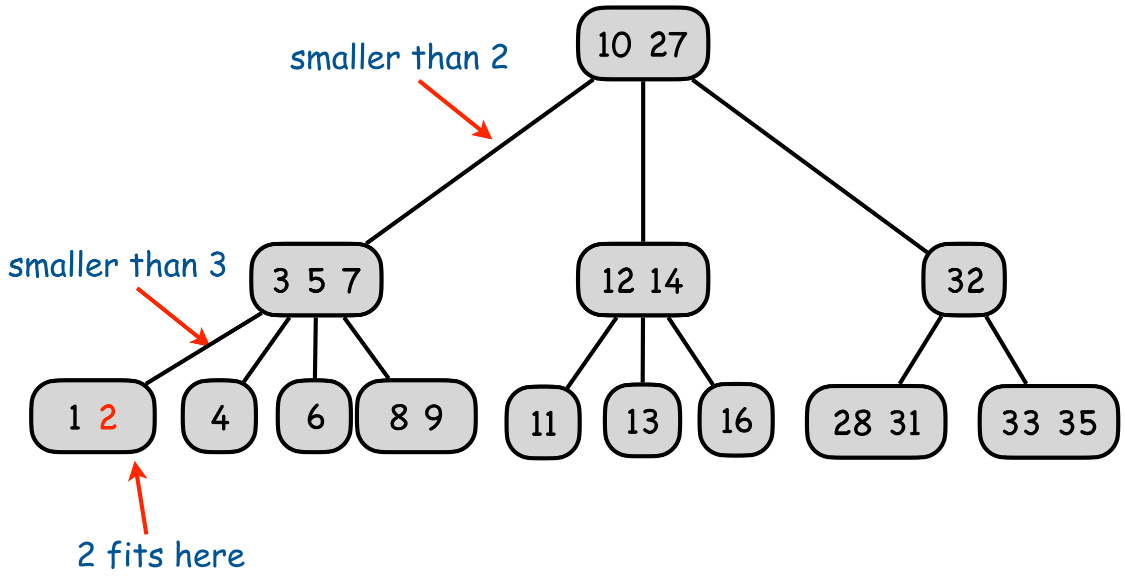


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert 2

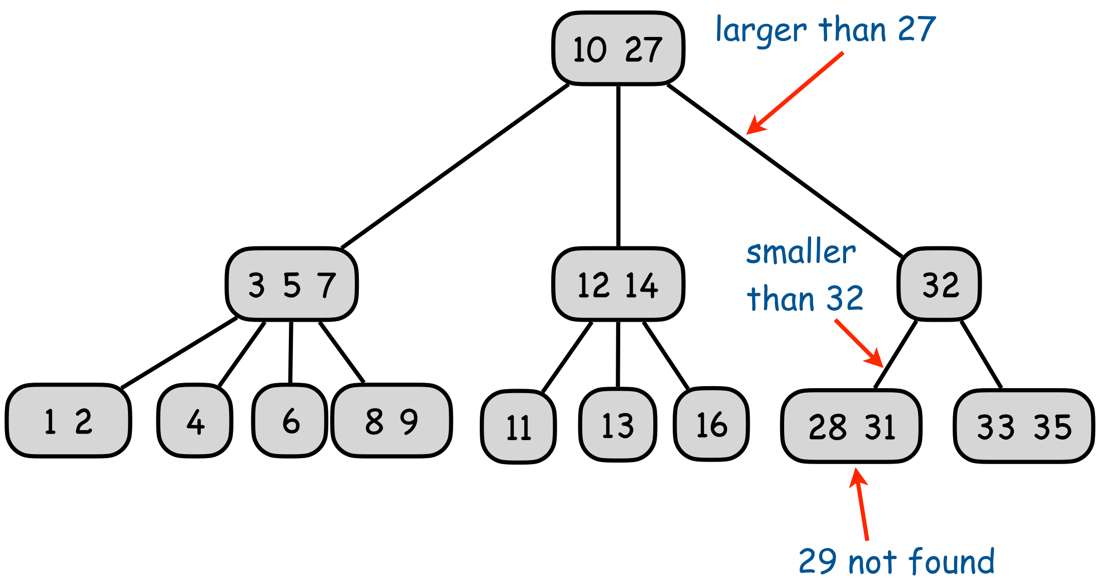


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert 29

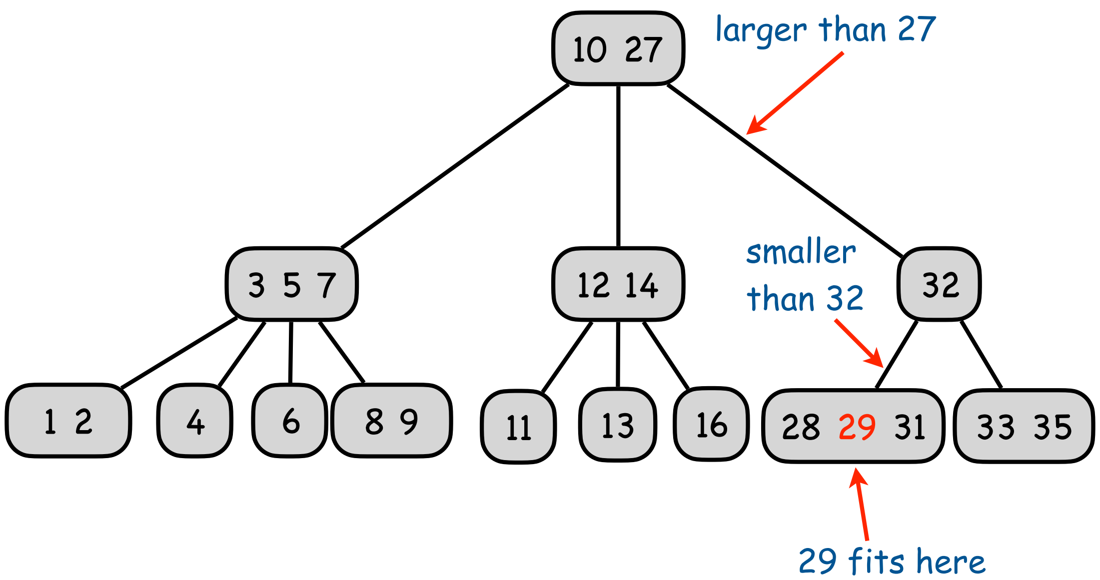


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node

Ex. Insert 29

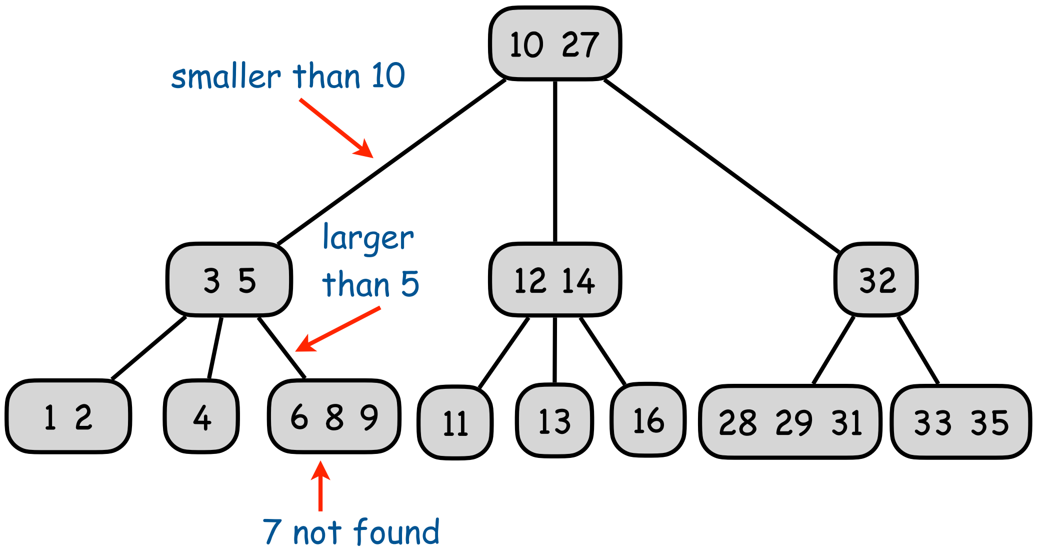


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node

Ex. Insert 7

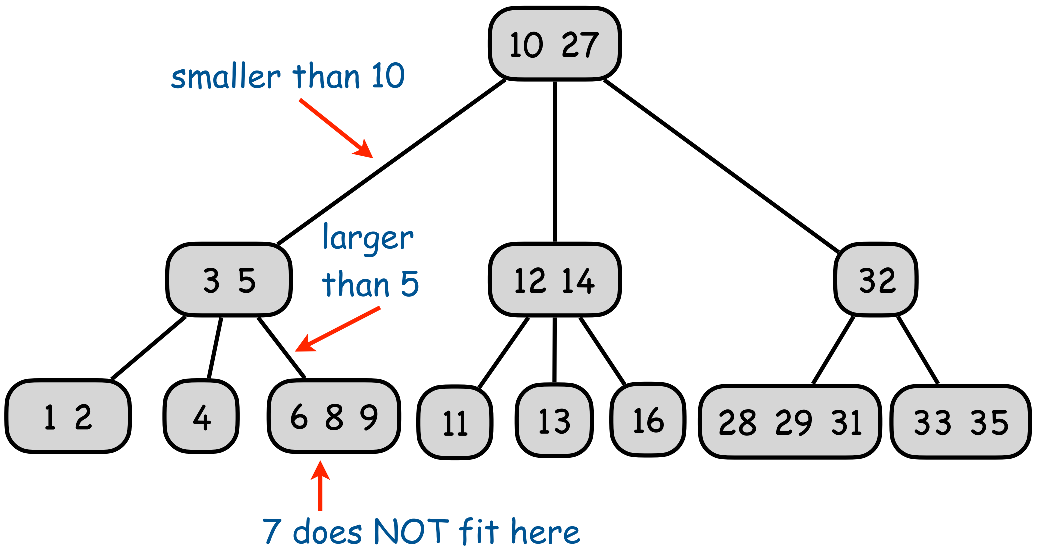


Insertion in a 2-3-4 tree

Insert.

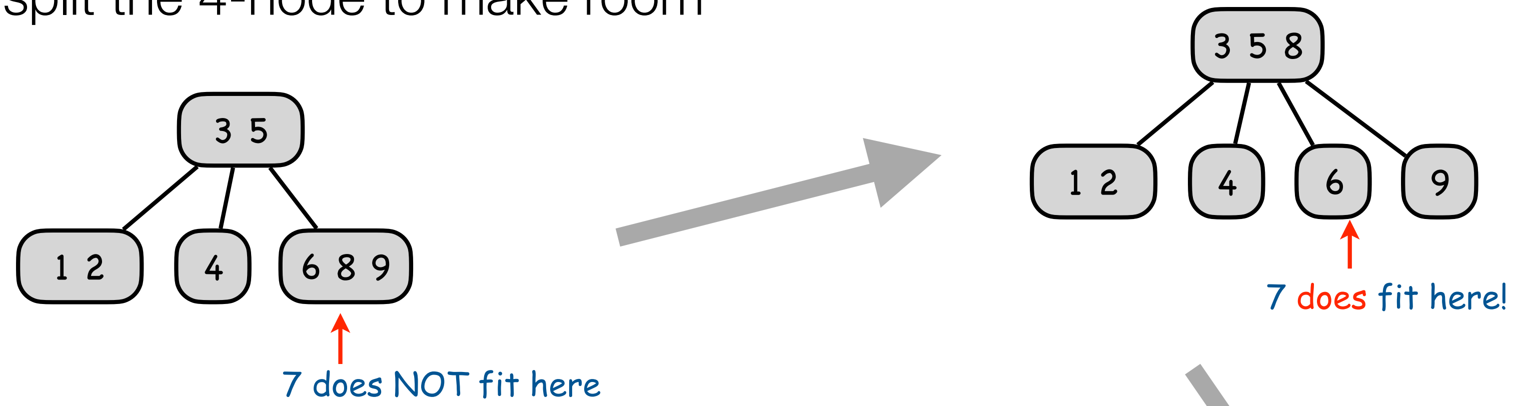
- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

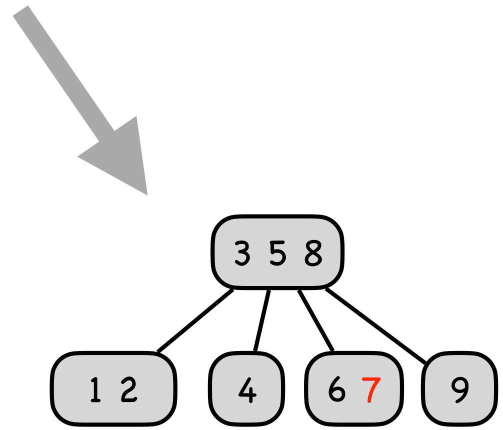
Ex. Insert 7



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

Idea: split the 4-node to make room





Problem: Doesn't work if parent is a 4-node

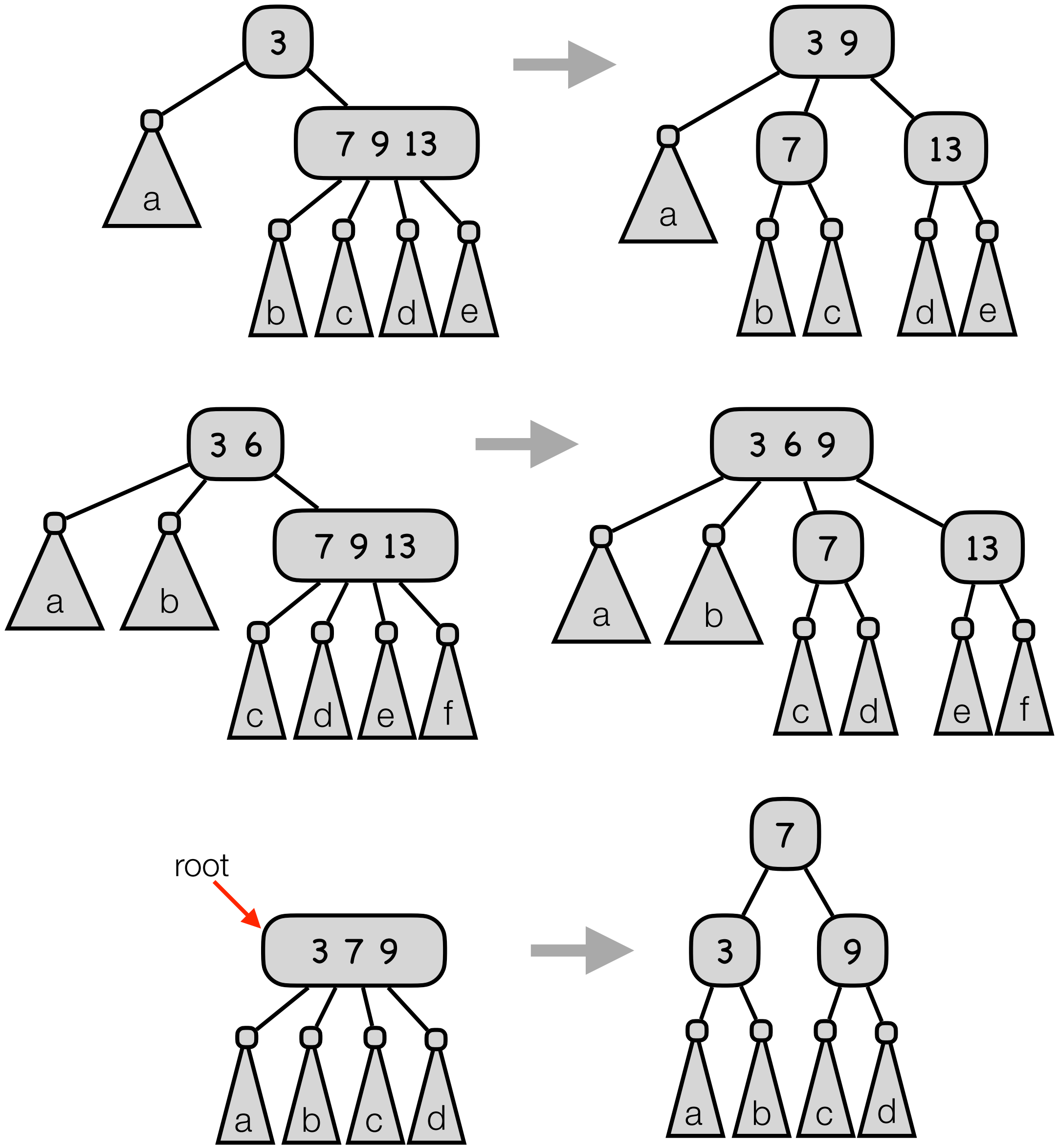
Solution 1: Split the parent (and continue splitting while necessary).

Solution 2: Split 4-nodes on the way down.

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

Idea: split 4-nodes on the way down the tree.

- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:



Invariant. Current node is not a 4-node.

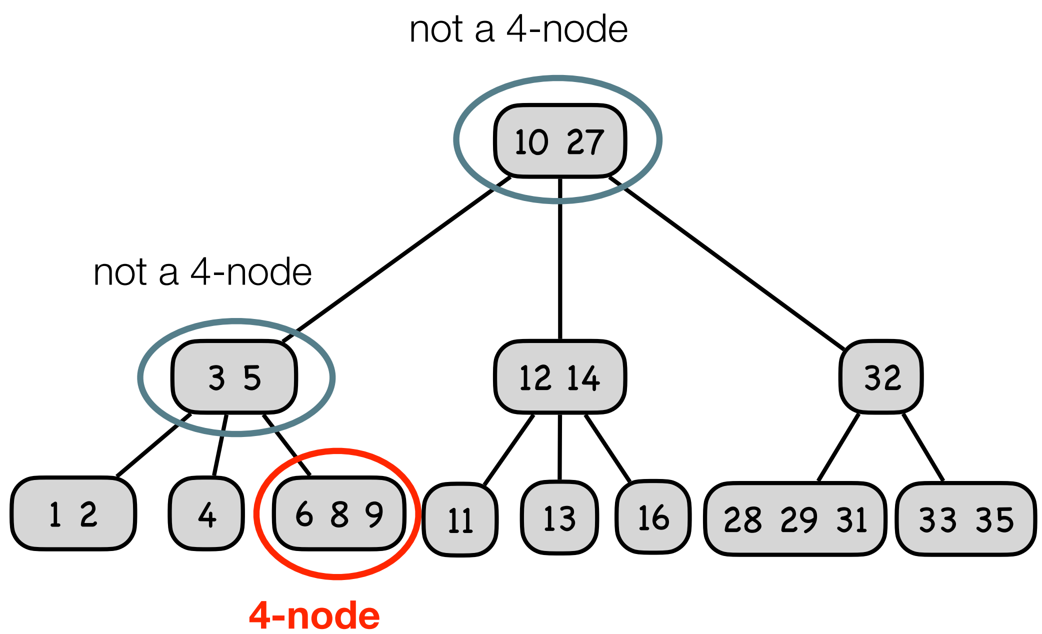
Consequence. Insertion at bottom is easy since it's not a 4-node.

Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert 7

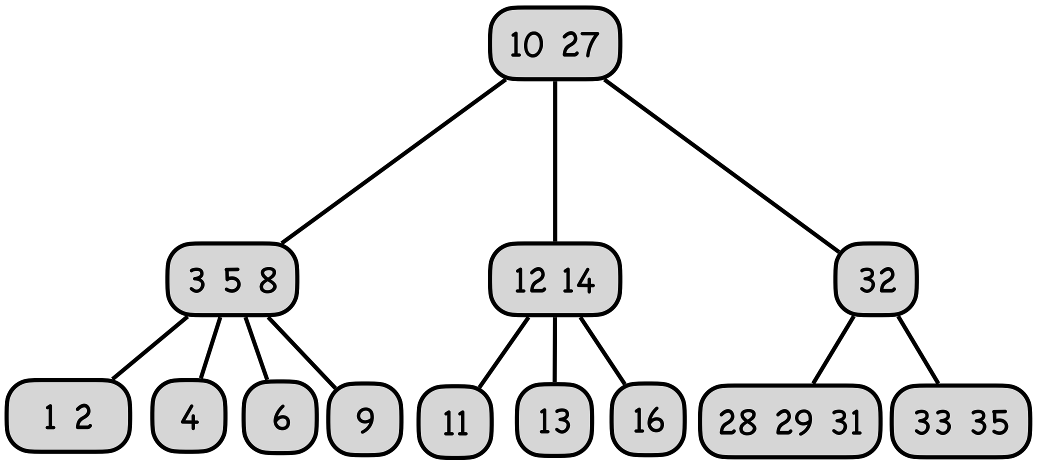


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert 7

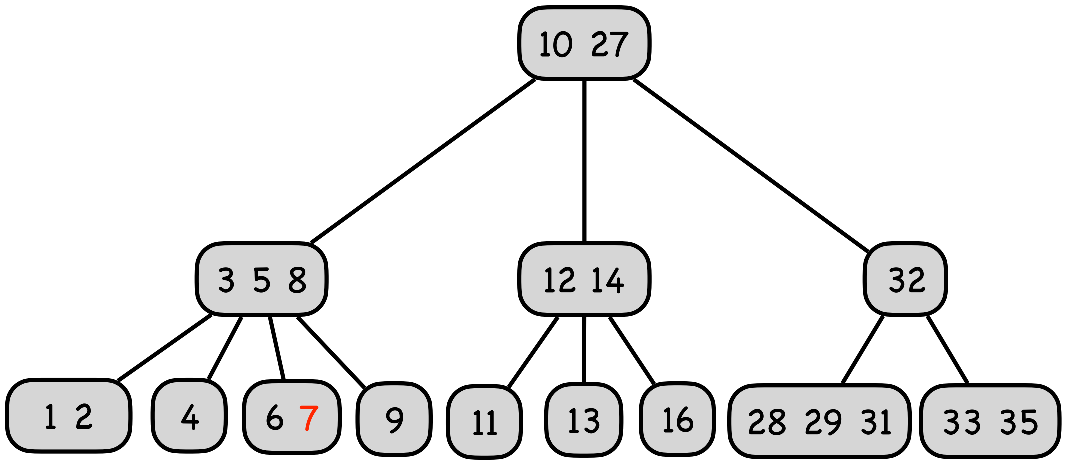


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

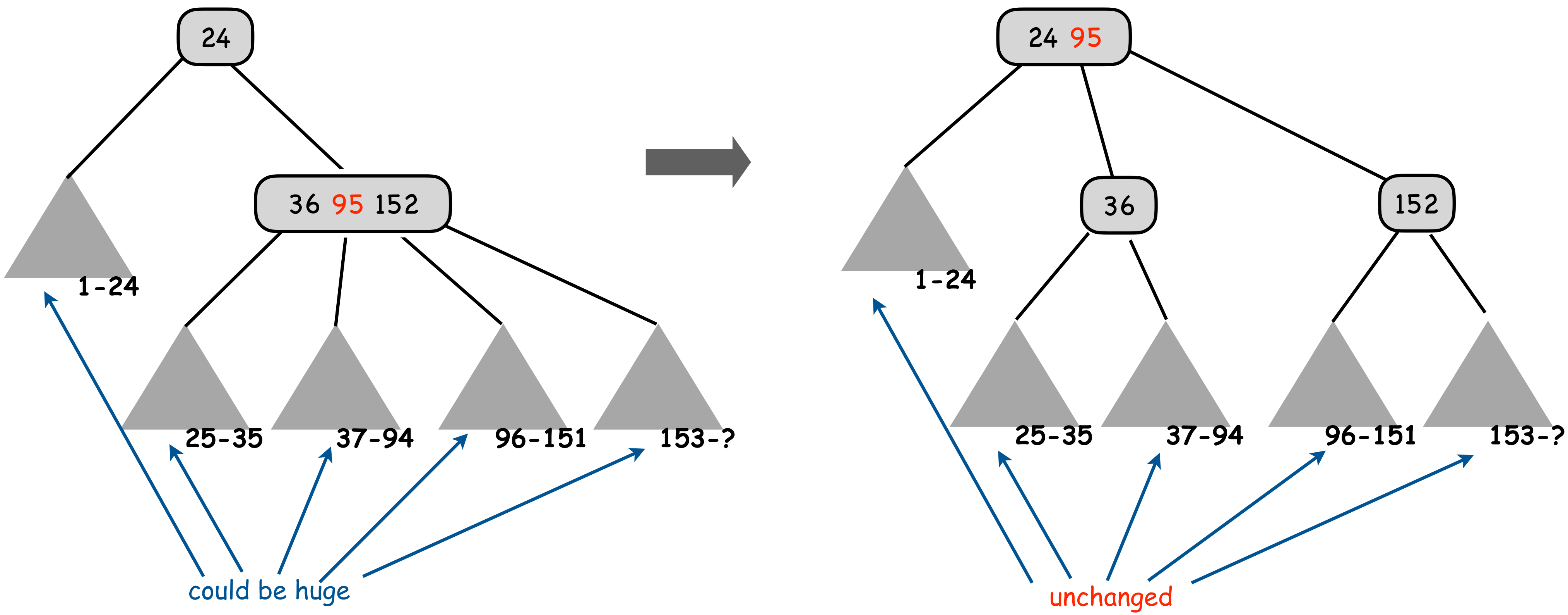
Ex. Insert 7



Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree.

Ex. Splitting a 4-node attached to a 2-node



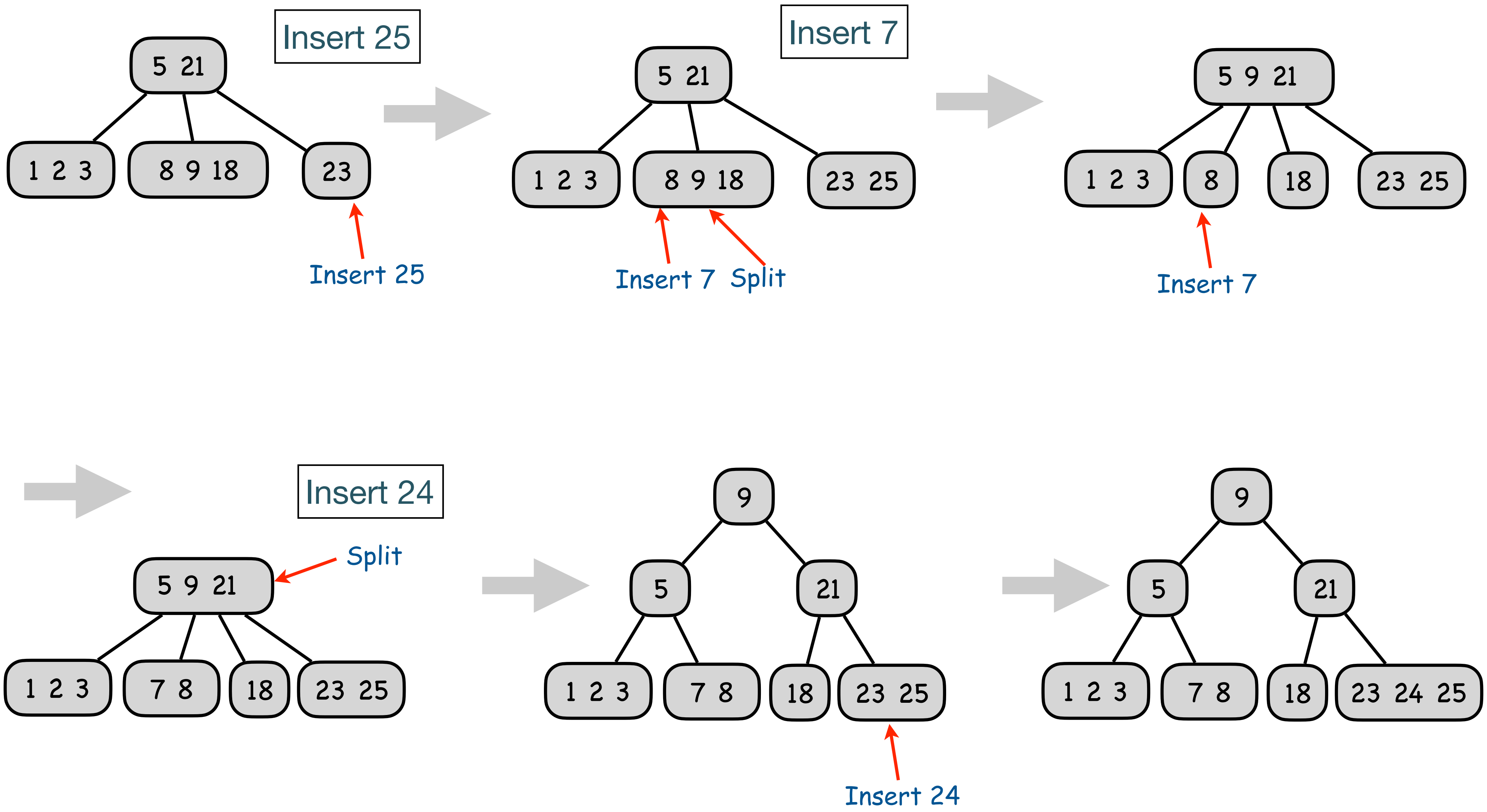
Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree.

Splitting a 4-node attached to a 4-node never happens when we split nodes on the way down the tree.

Invariant. Current node is not a 4-node.

Insertion 2-3-4 trees

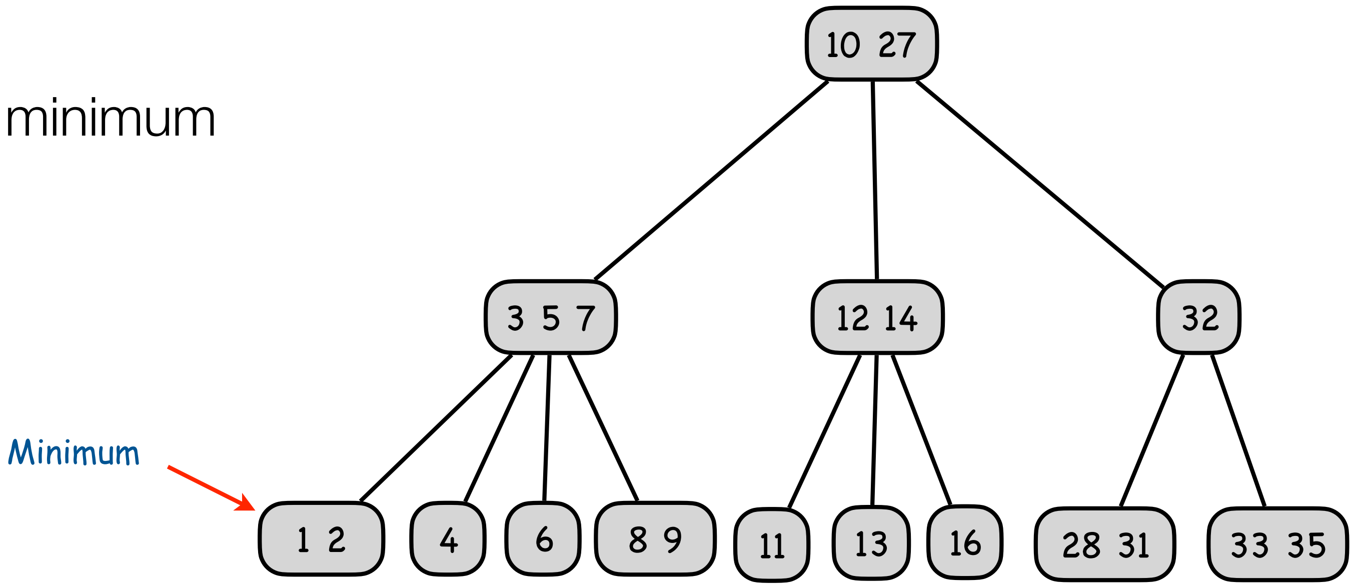


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum

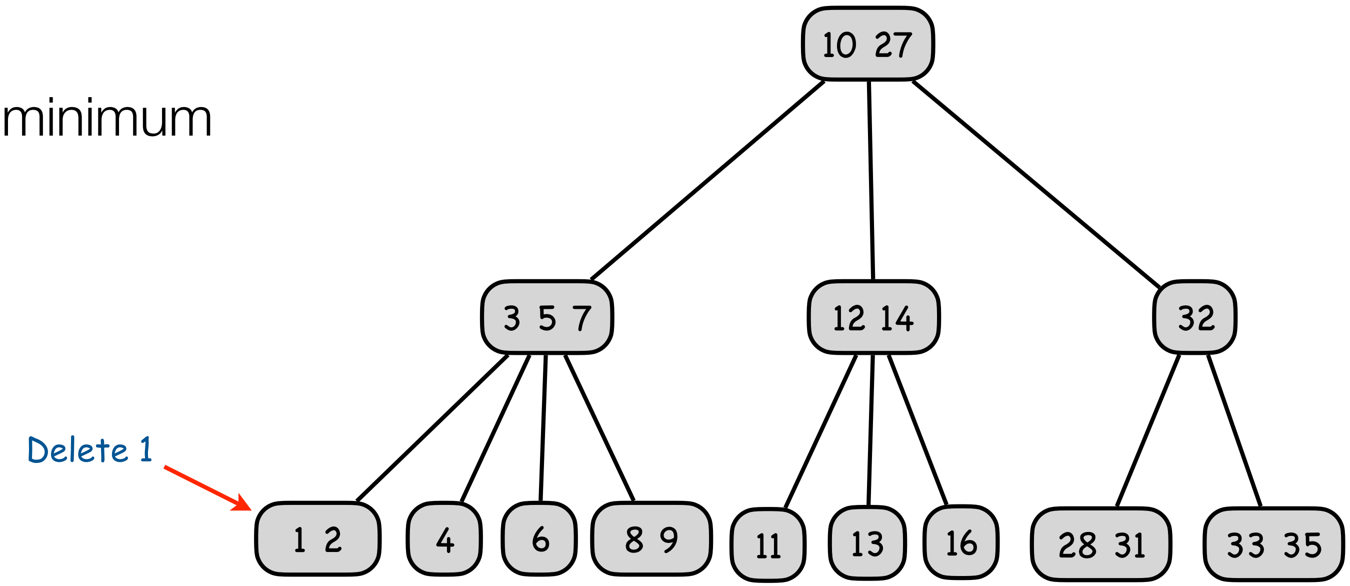


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum

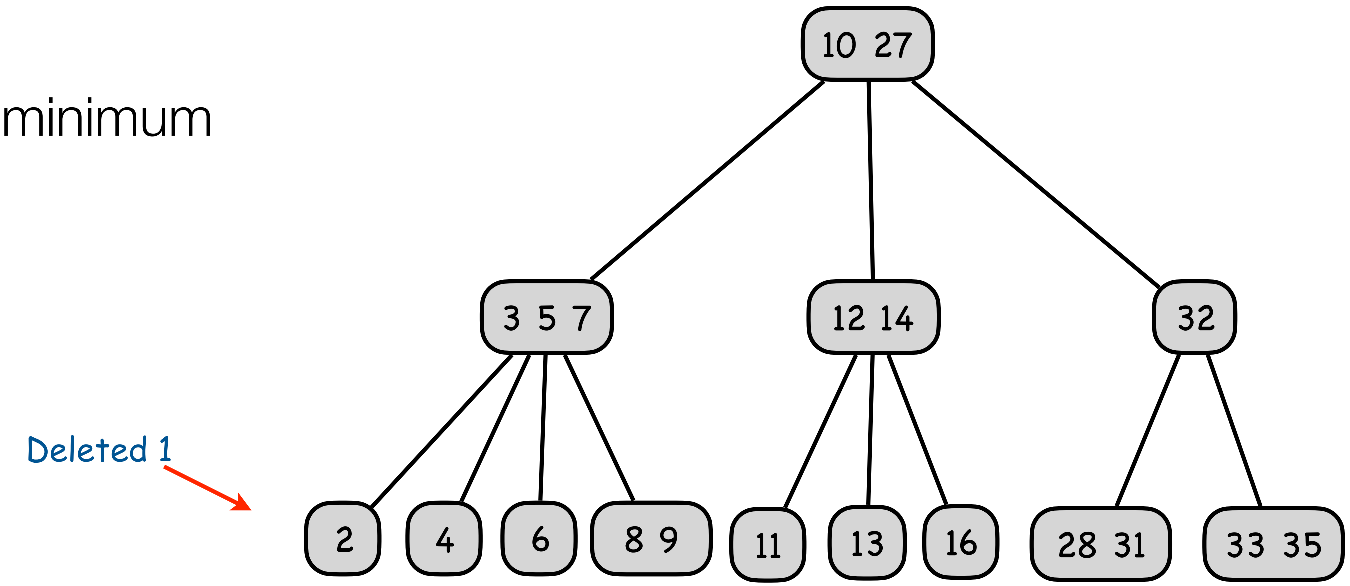


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum

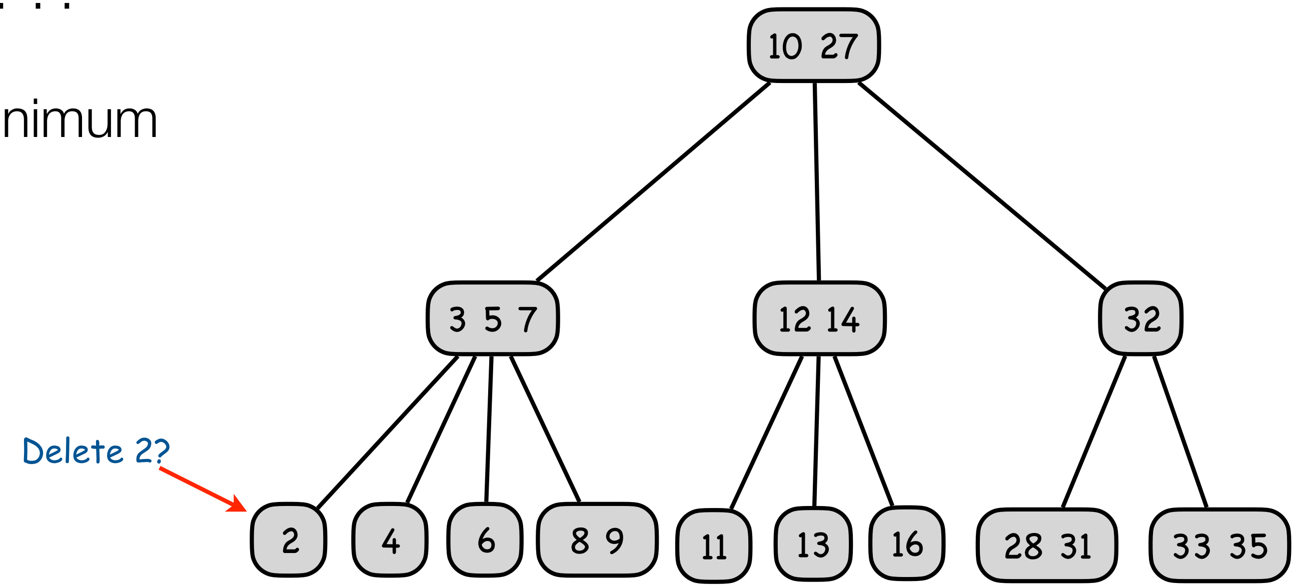


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: ??

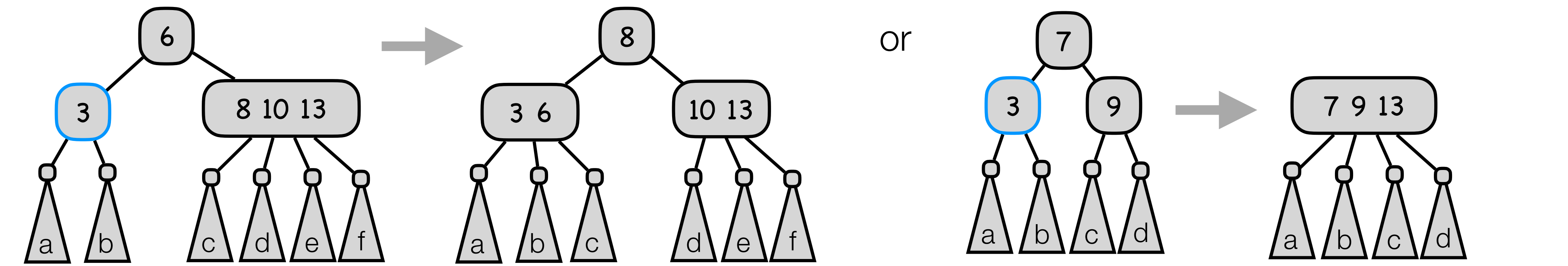
Ex. Delete minimum



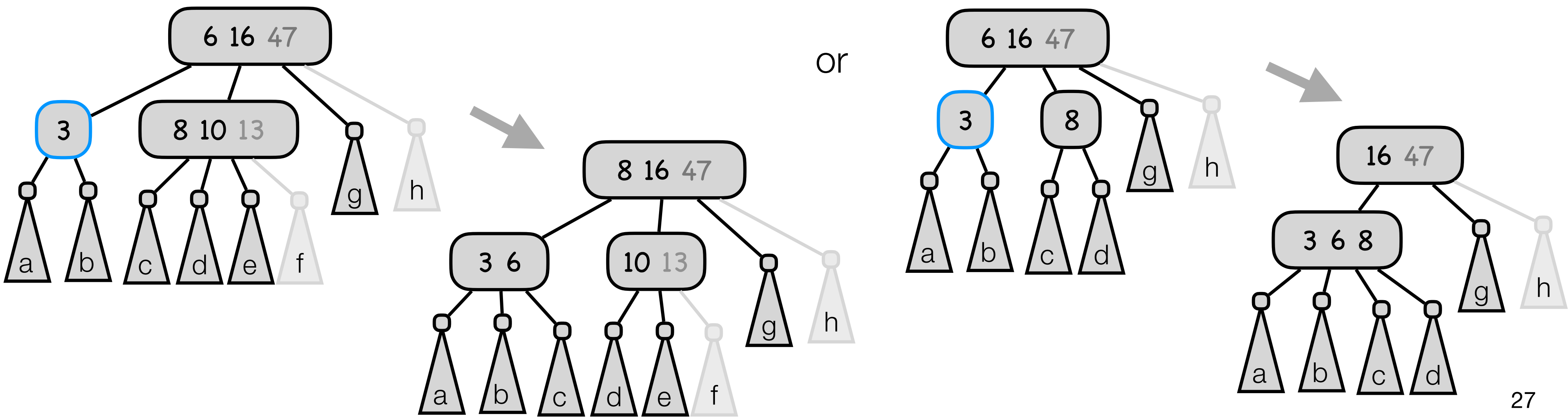
Deletions in 2-3-4 trees

Idea: On the way down maintain the invariant that current node (except root) is not a 2-node.

- Child of root and root is a 2-node:



- on the way down:

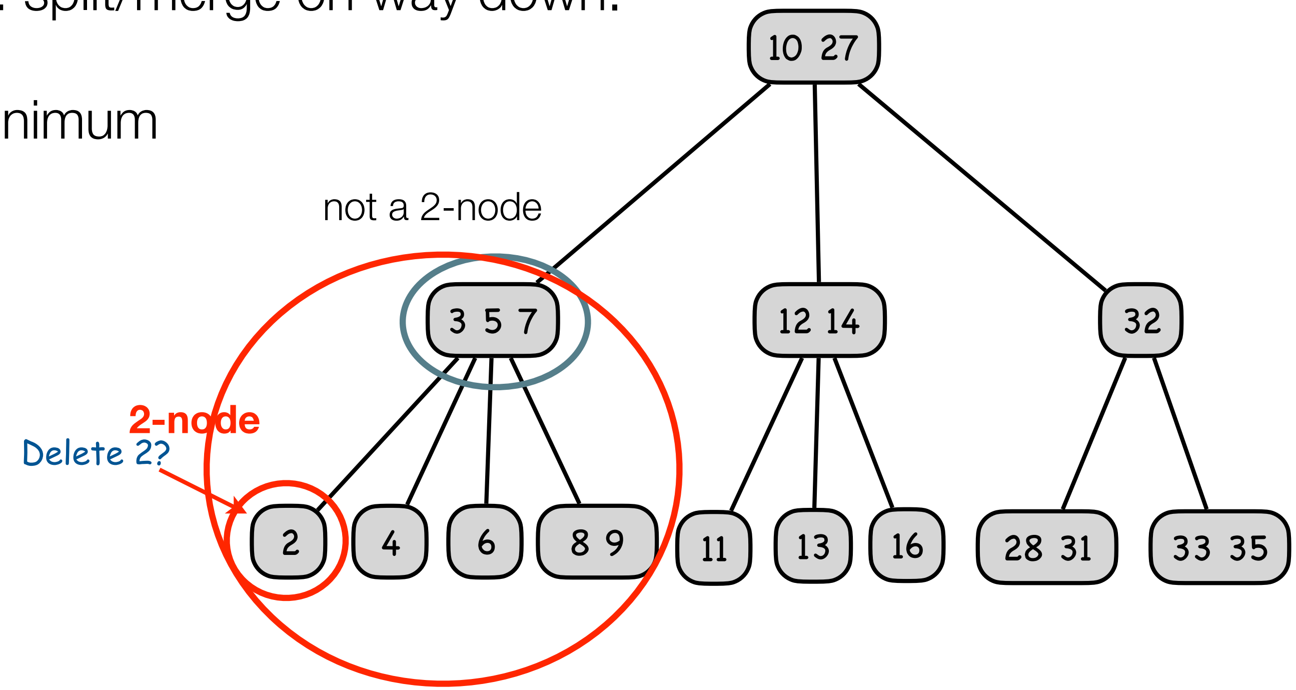


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

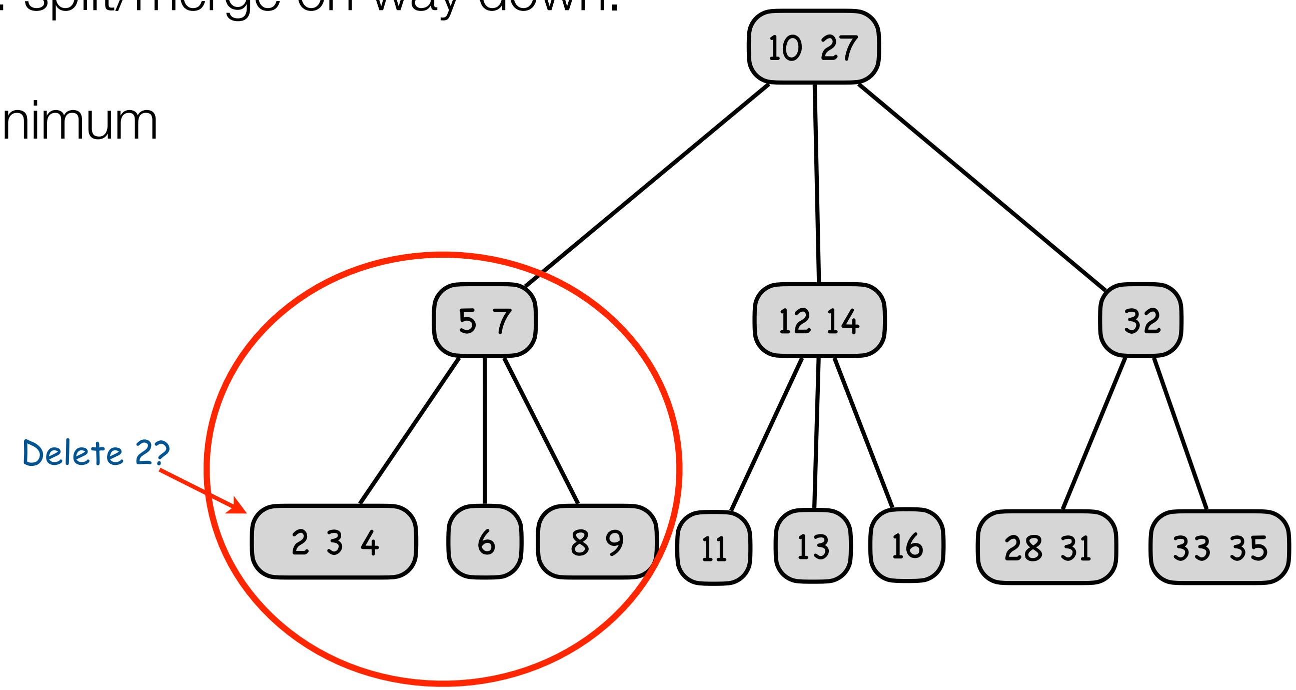


Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

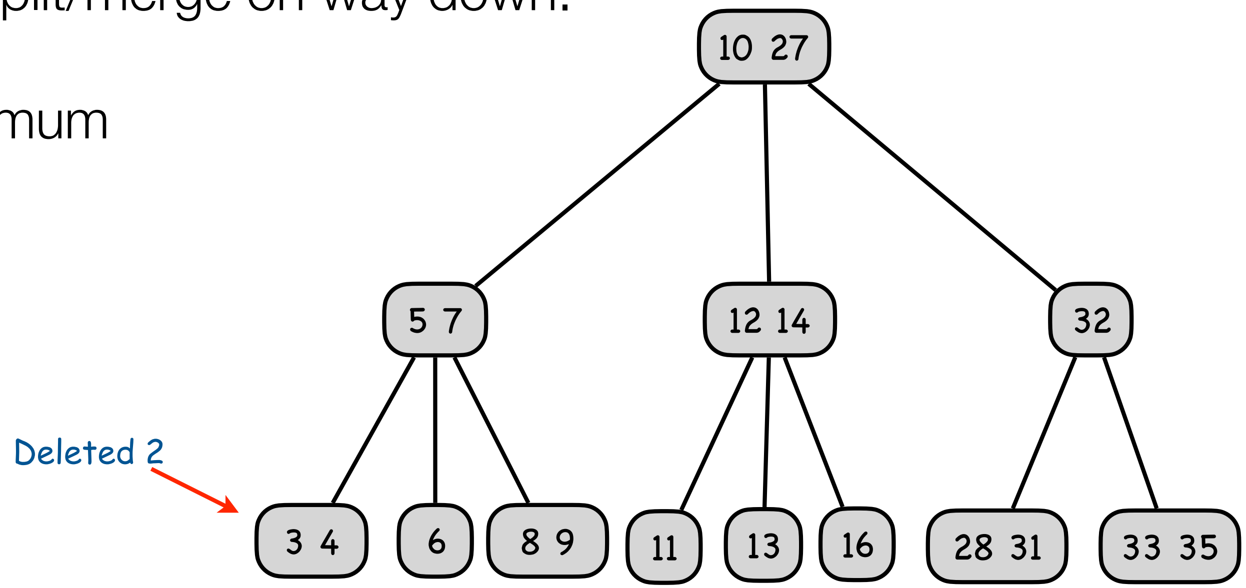


Deletions in 2-3-4 trees

Delete minimum:

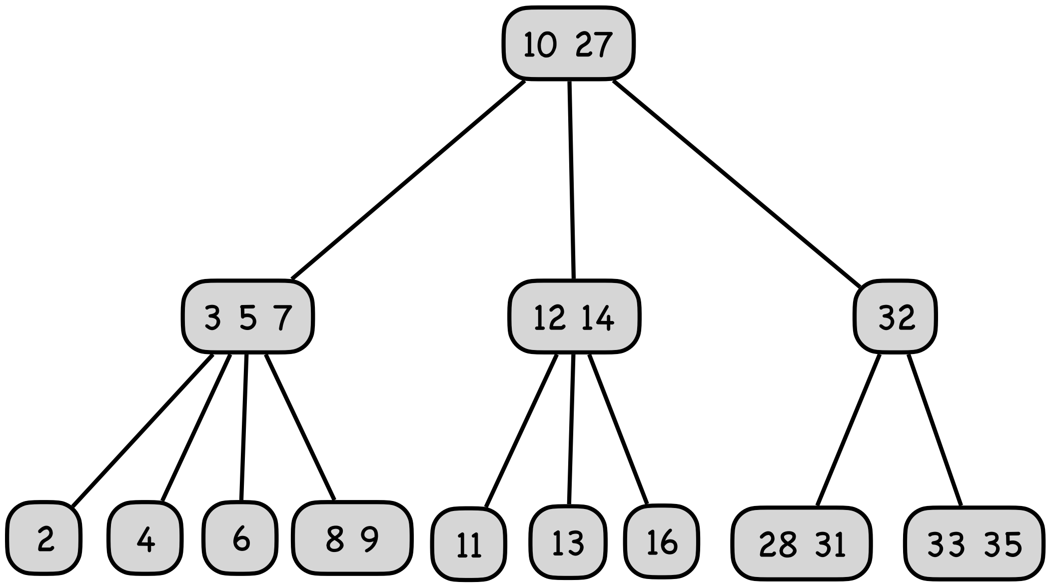
- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum



Deletions in 2-3-4 trees

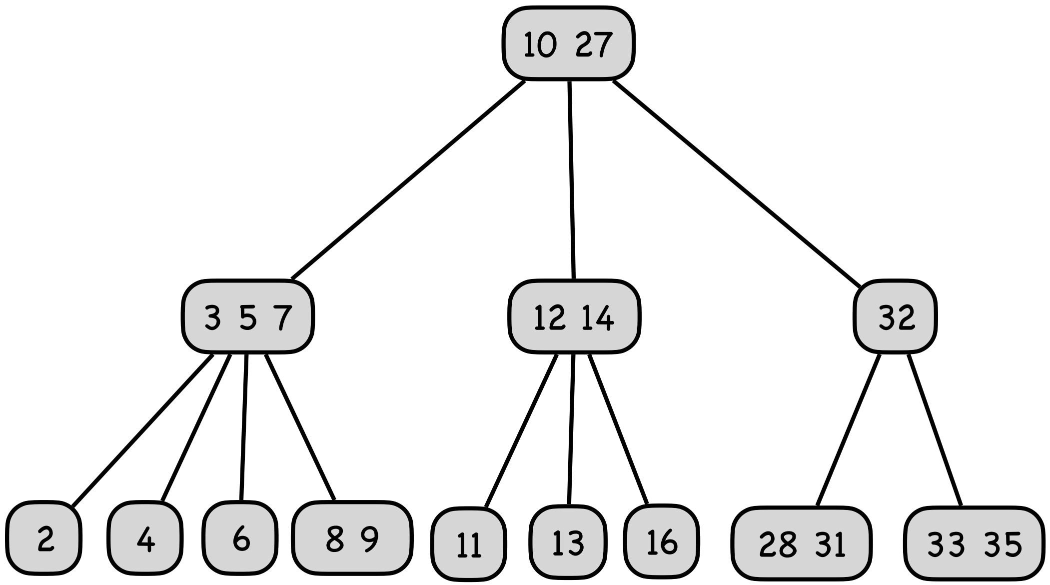
Delete:



Deletions in 2-3-4 trees

Delete:

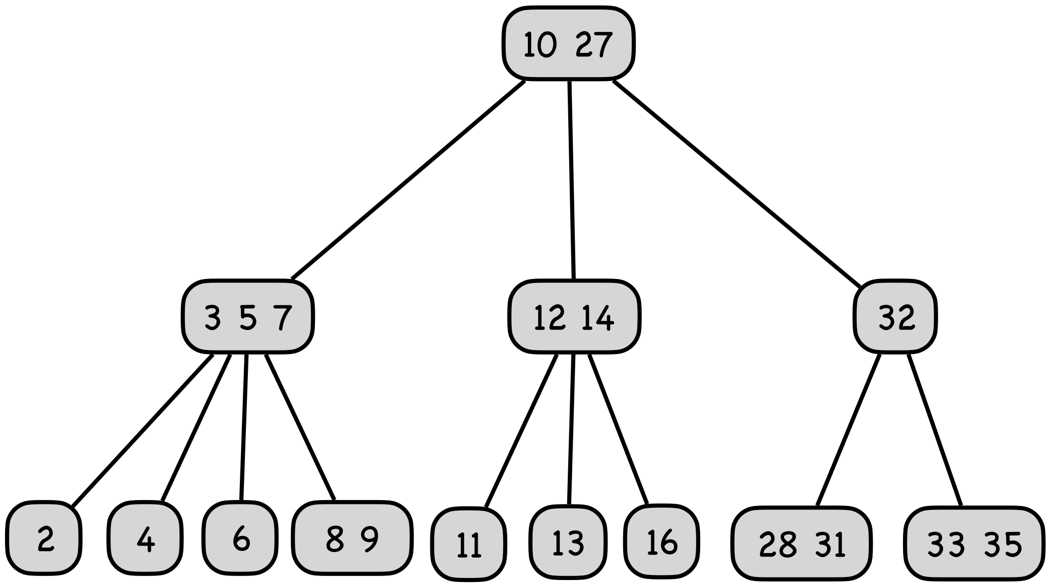
- During search maintain invariant that current node is not a 2-node



Deletions in 2-3-4 trees

Delete:

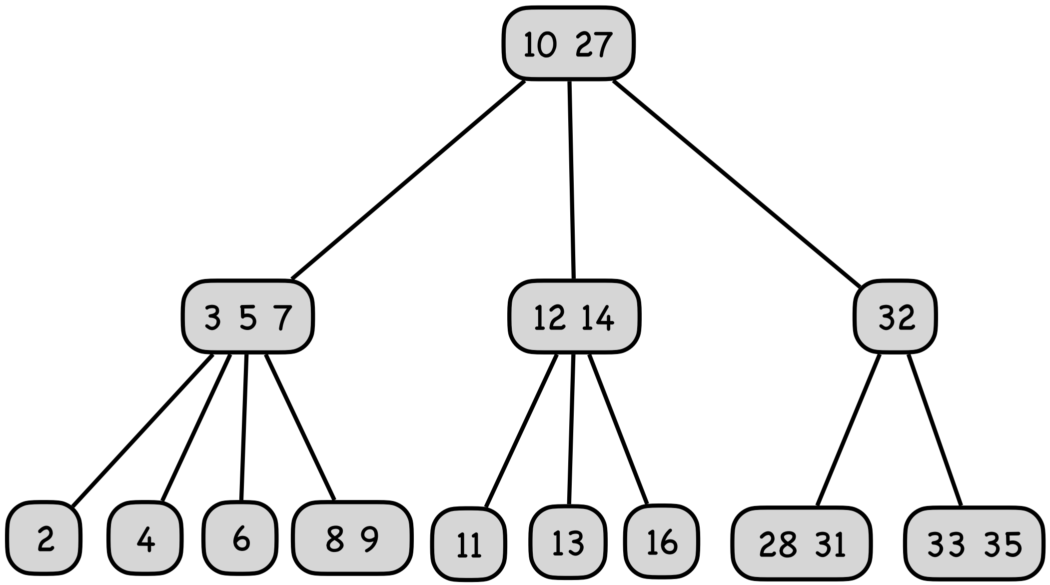
- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key



Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

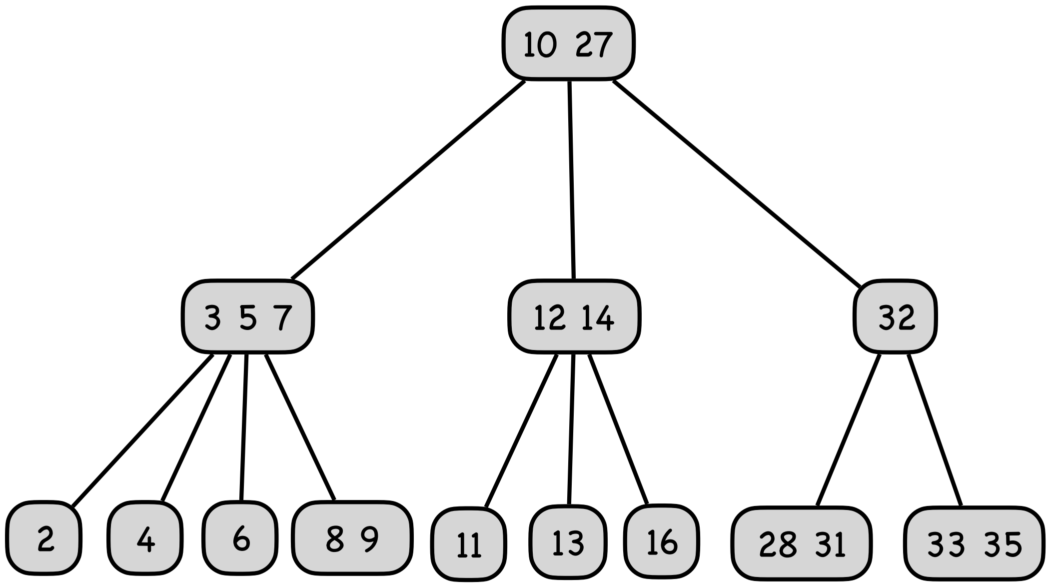


Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10



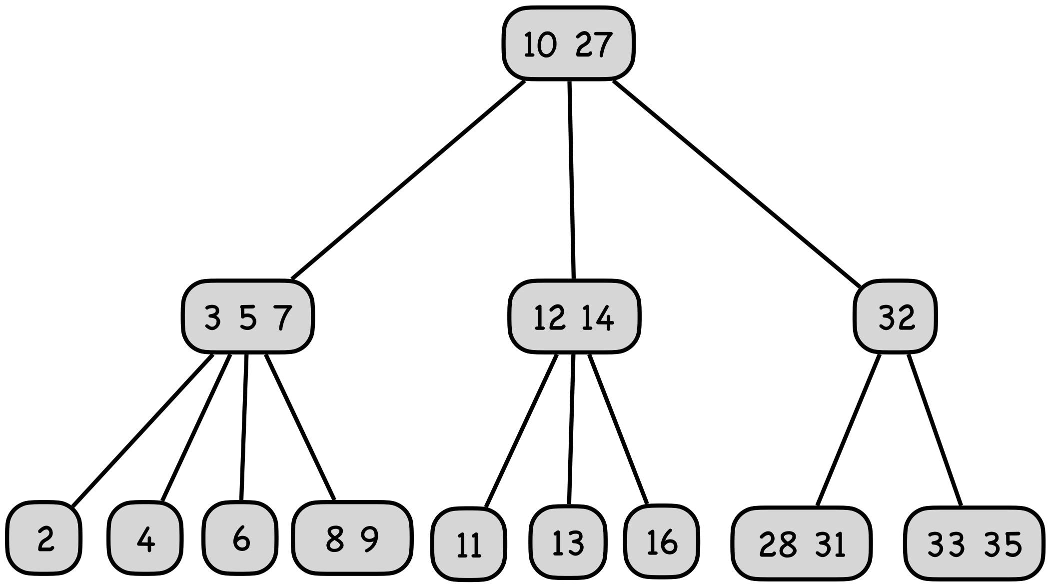
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor



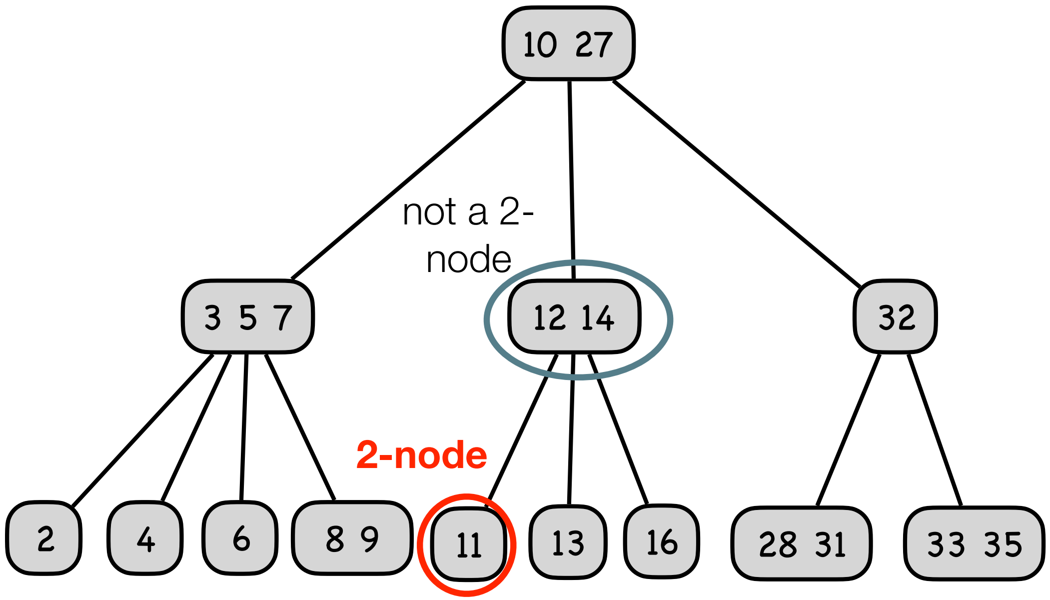
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor



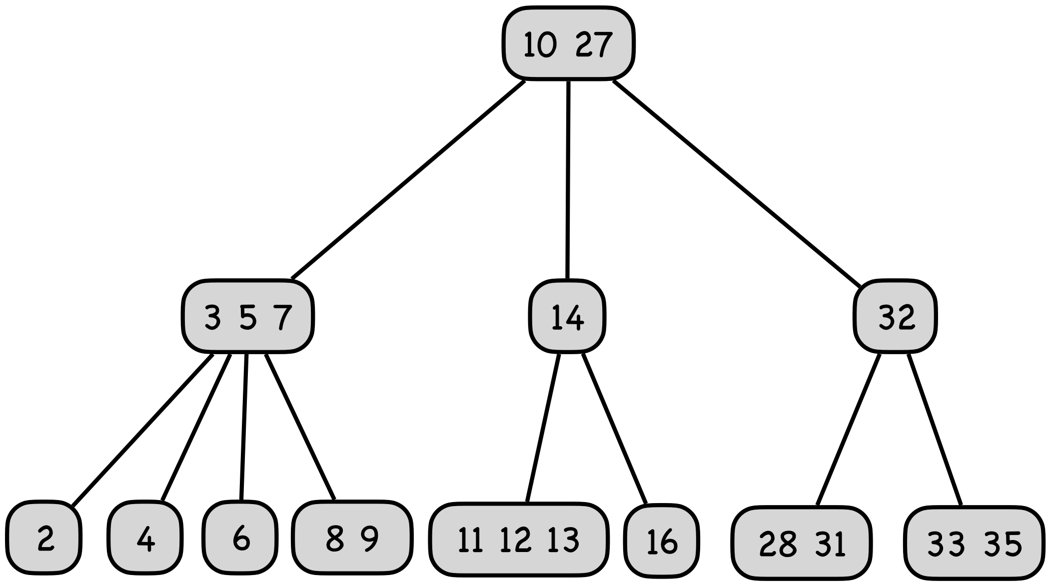
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor



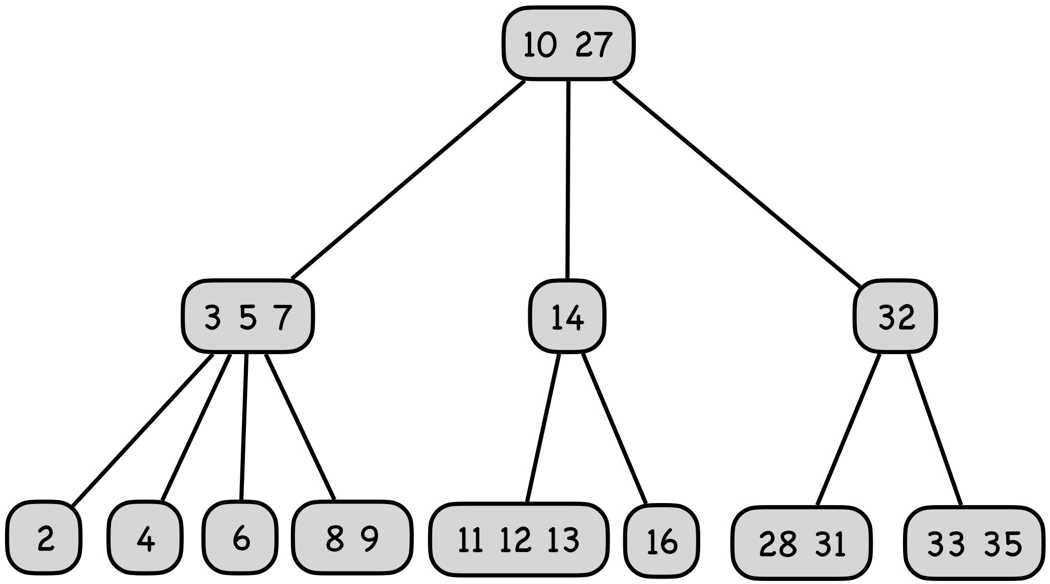
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor
- Delete 11 from leaf



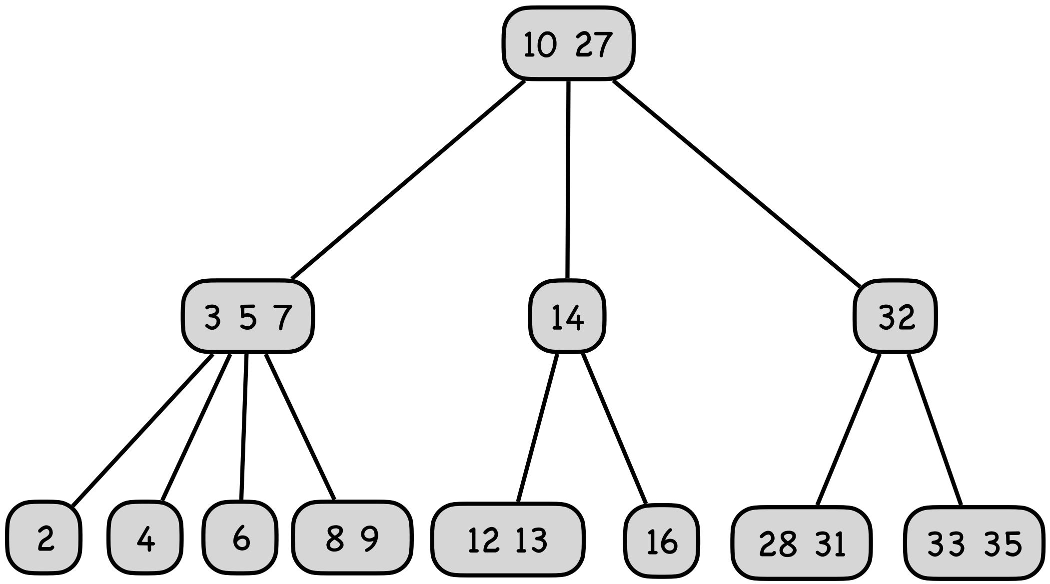
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor
- Delete 11 from leaf



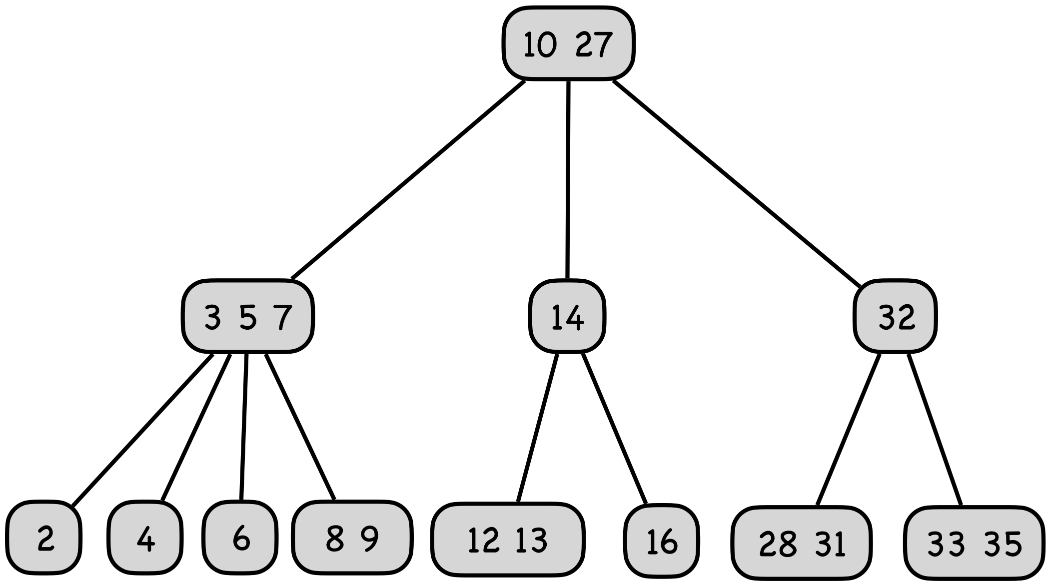
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete 10

- Find successor
- Delete 11 from leaf
- Replace 10 with 11



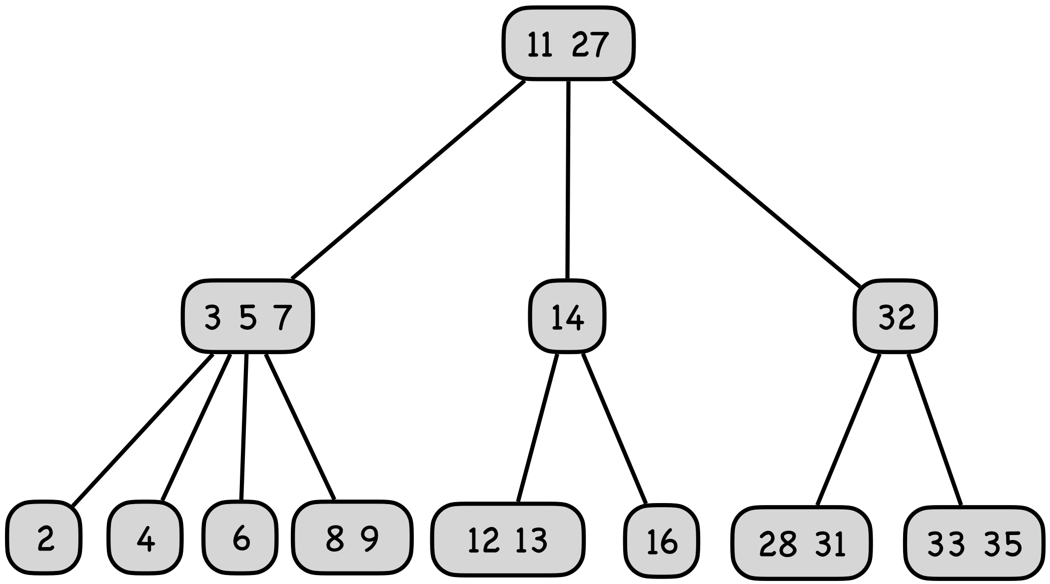
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

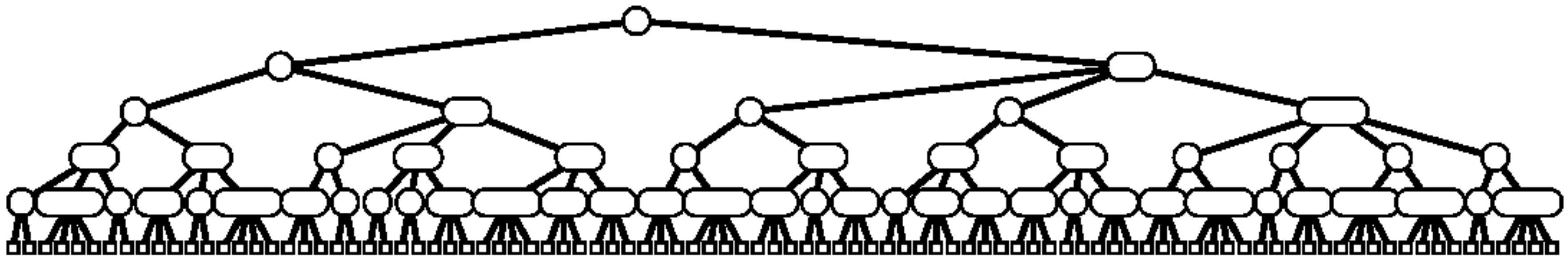
Ex. Delete 10

- Find successor
- Delete 11 from leaf
- Replace 10 with 11



2-3-4 Tree: Balance

Property. All paths from root to leaf have same length.



Tree height.

Worst case: lg N [all 2-nodes]

Best case: $\log_4 N = 1/2 \lg N$ [all 4-nodes]

Between 10 and 20 for a million nodes.

Between 15 and 30 for a billion nodes.

Dynamic set implementations

Worst case running times

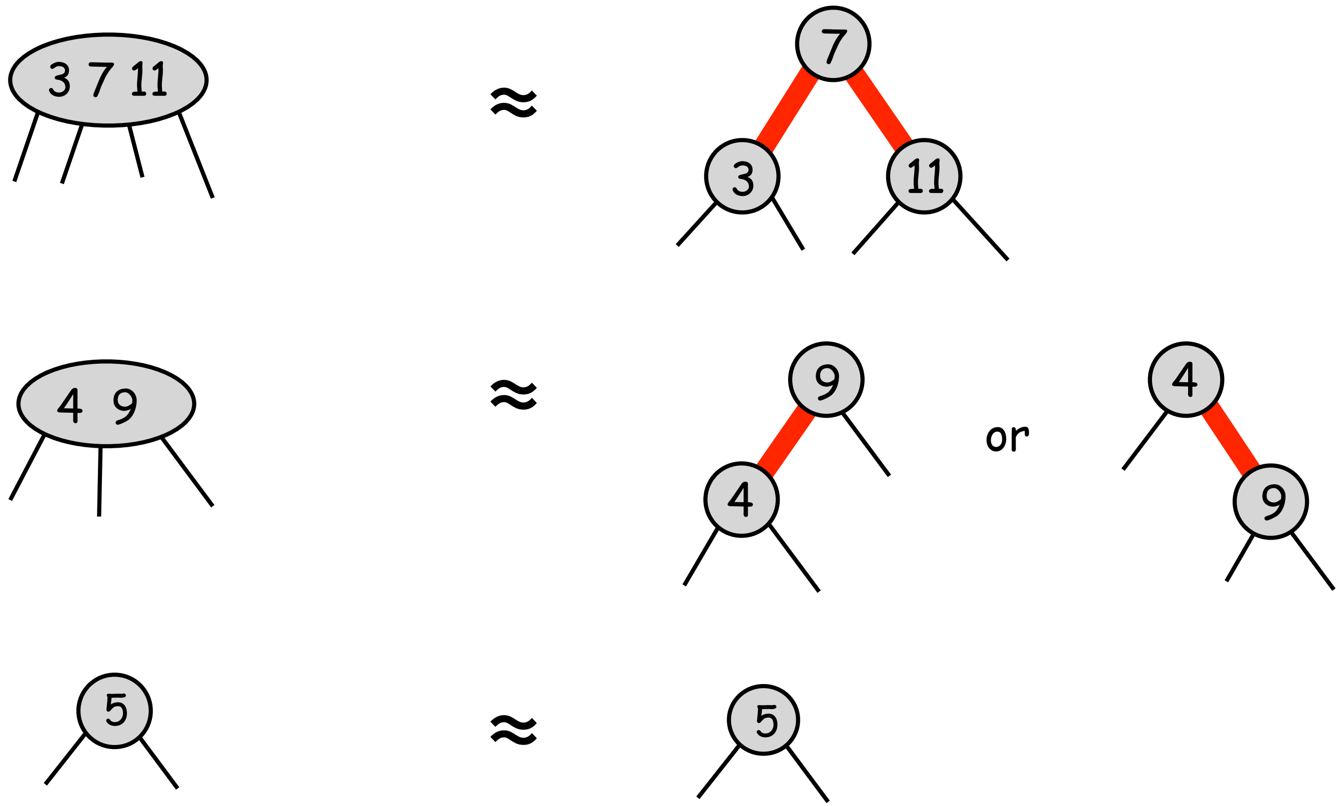
Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
ordered array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(log n)	O(log n)
BST	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)
2-3-4 tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Red-black trees

Red-black tree

Represent 2-3-4 tree as a binary search tree

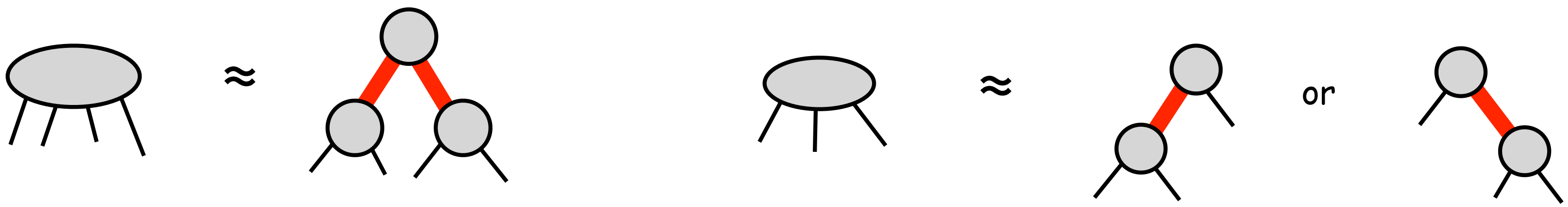
- Use colors on edges to represent 3- and 4-nodes (red edges glues nodes together).



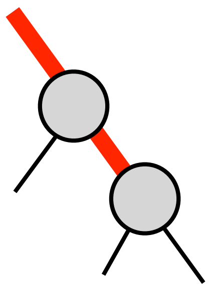
Red-black tree

Represent 2-3-4 tree as a binary search tree

- Use colors on edges to represent 3- and 4-nodes.



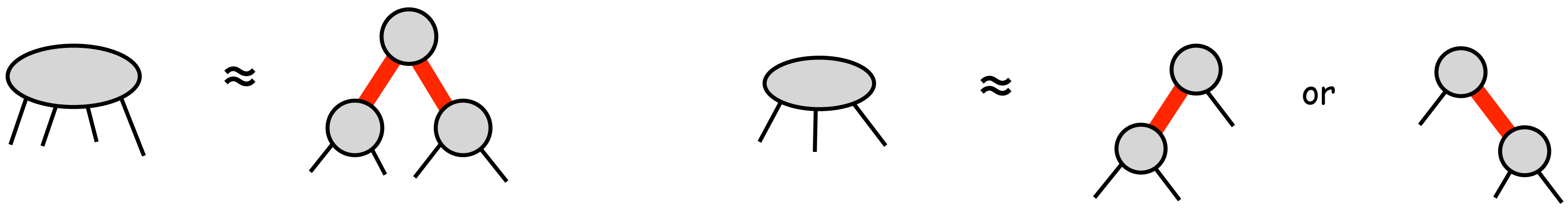
- *Disallowed*: 2 red nodes in-a-row.



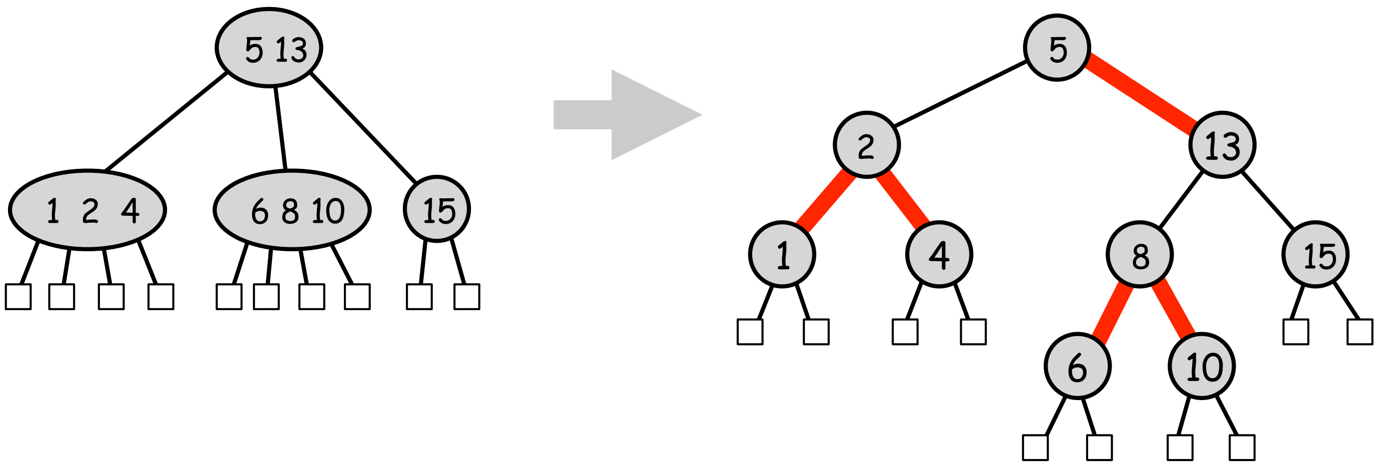
Red-black tree

Represent 2-3-4 tree as a binary search tree

- Use colors on edges to represent 3- and 4-nodes.



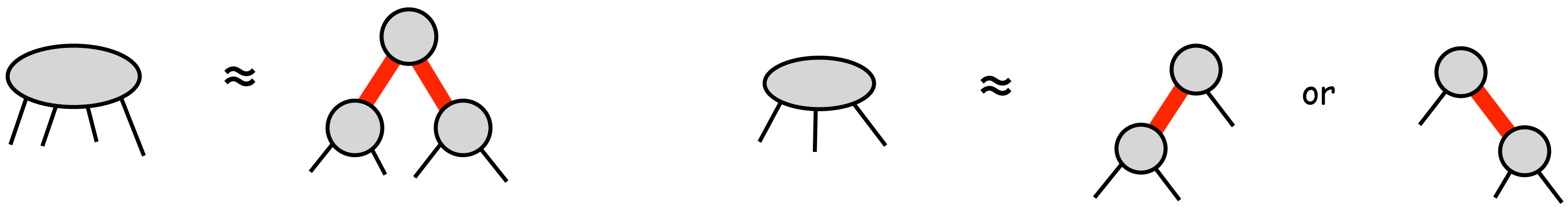
- Connection between 2-3-4 trees and red-black trees:



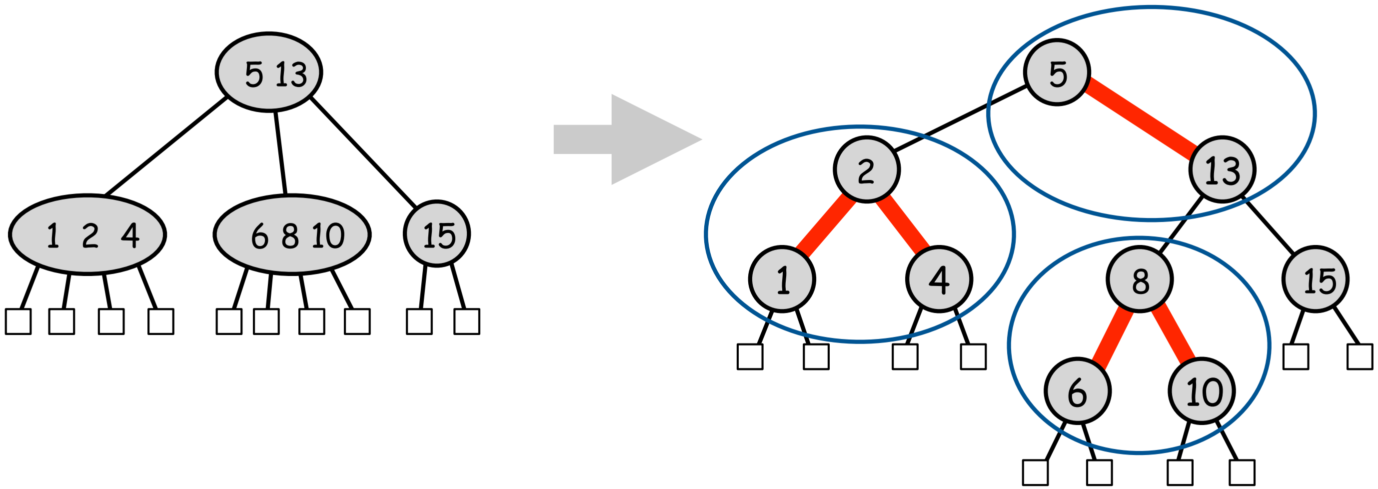
Red-black tree

Represent 2-3-4 tree as a binary search tree

- Use colors on edges to represent 3- and 4-nodes.



- Connection between 2-3-4 trees and red-black trees:



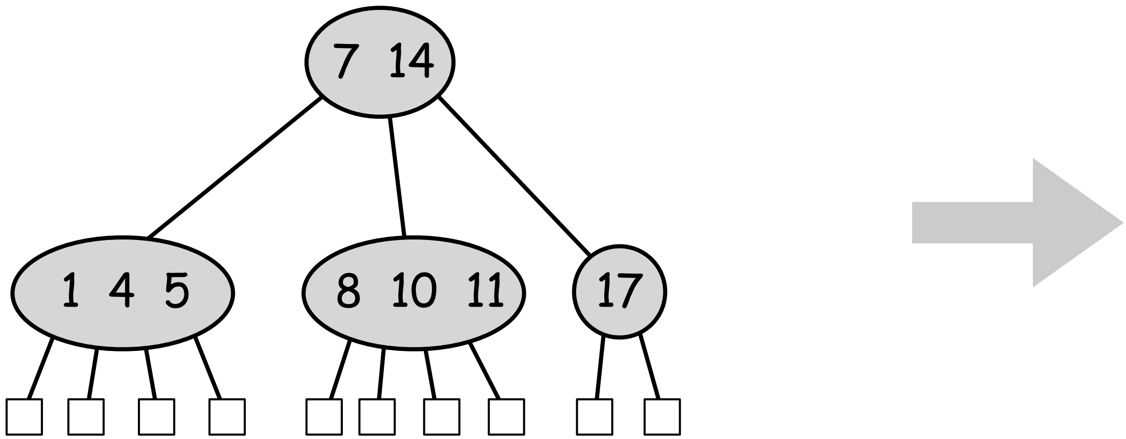
Red-black tree

Properties of red-black trees:

- All root-to-leaf paths have the same number of black edges.
- No root-to-leaf path has two red edges in a row.

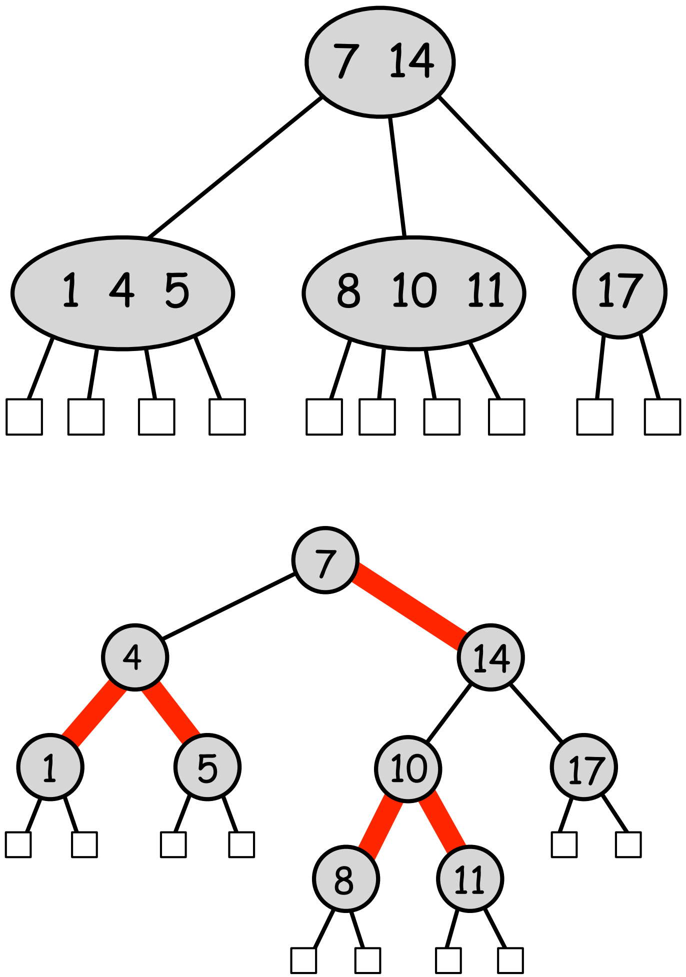
Red-black tree

Connection between 2-3-4 trees and red-black trees:



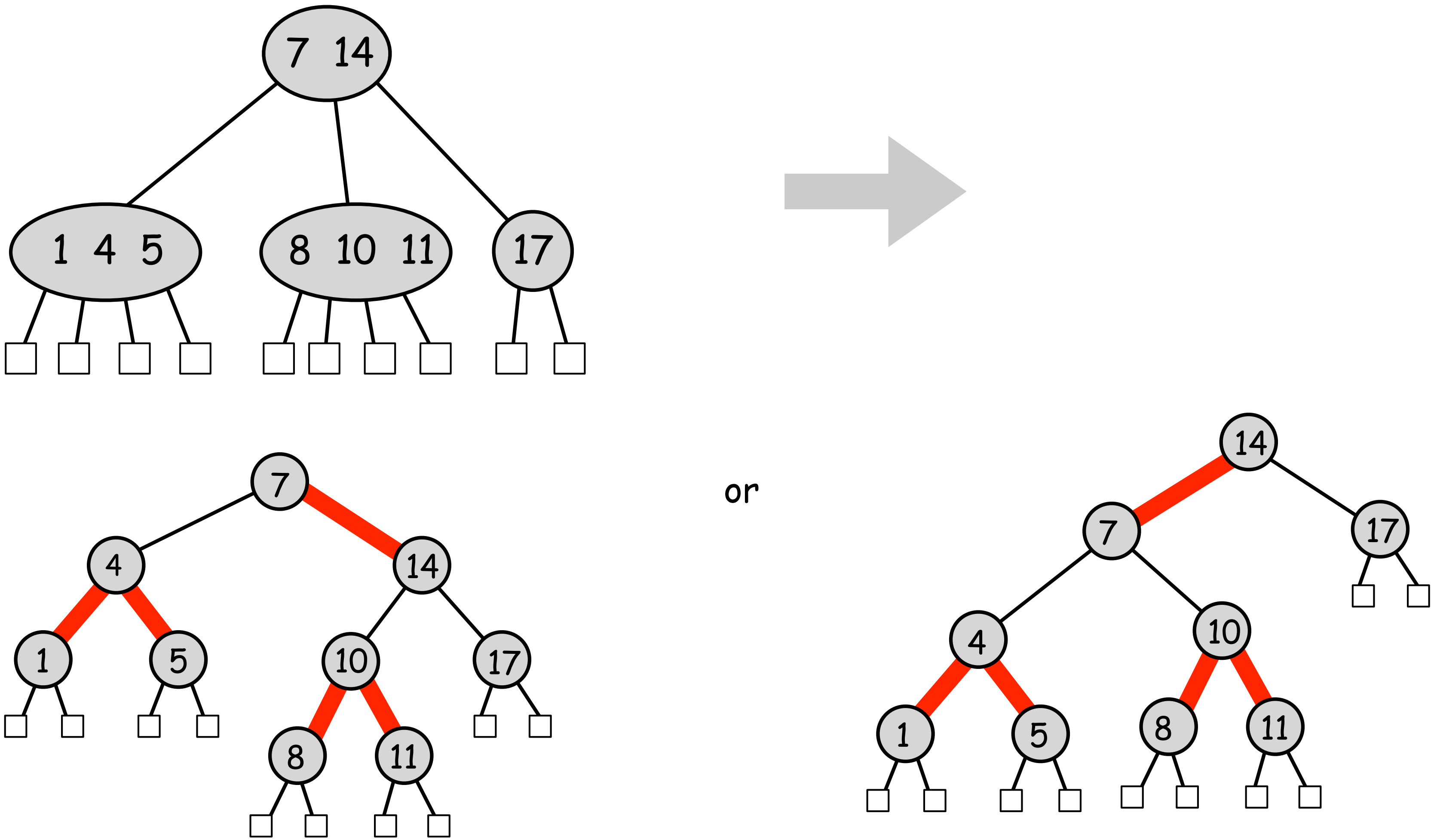
Red-black tree

Connection between 2-3-4 trees and red-black trees:



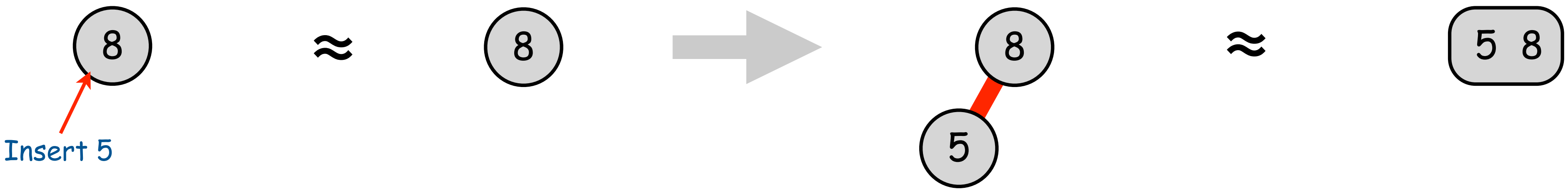
Red-black tree

Connection between 2-3-4 trees and red-black trees:

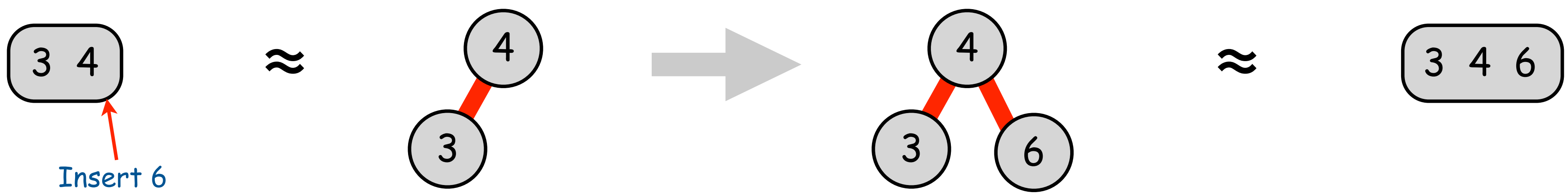


Insertion in red-black trees

Insertion in 2-node:

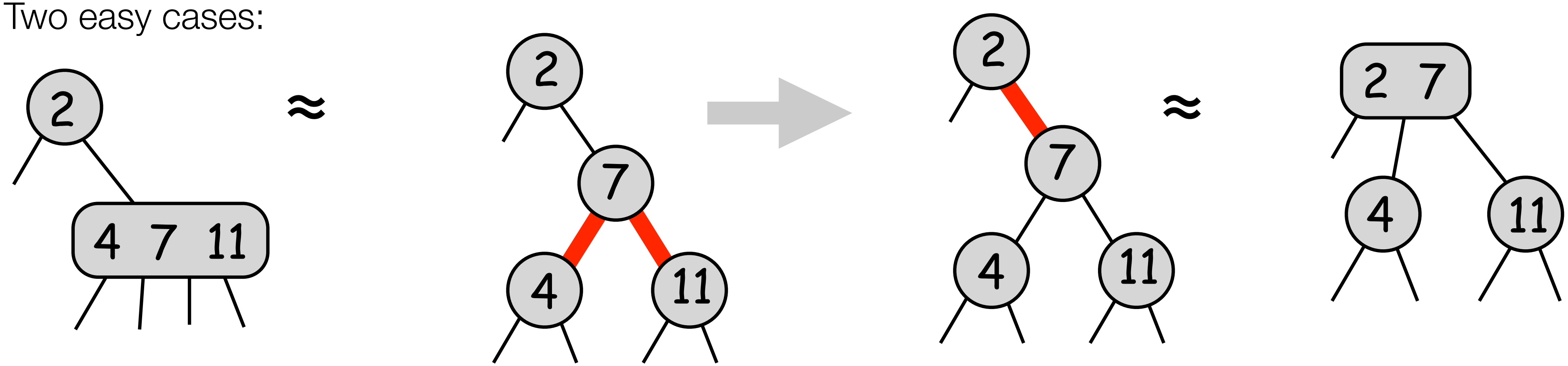


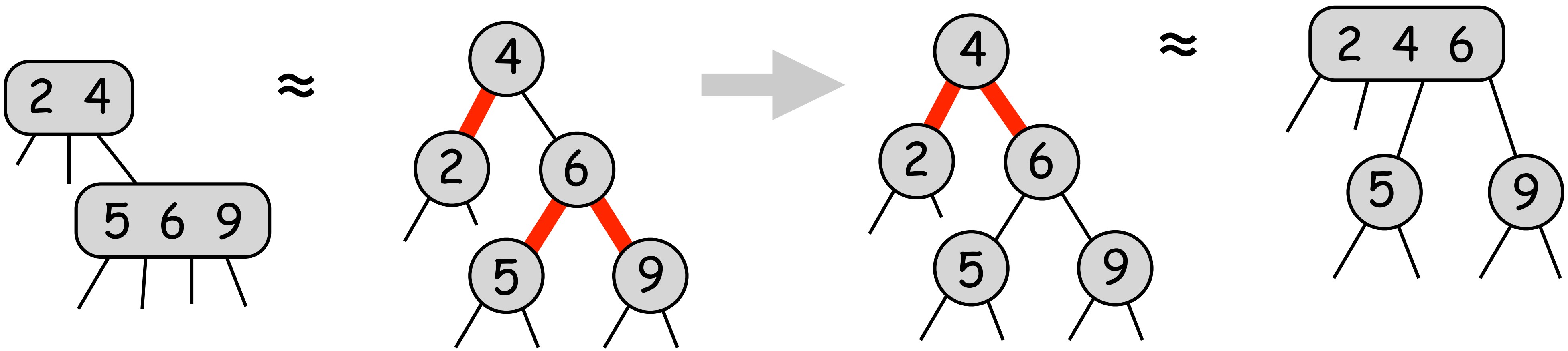
Insertion in 3-node:



Red-black tree: Splitting 4-nodes

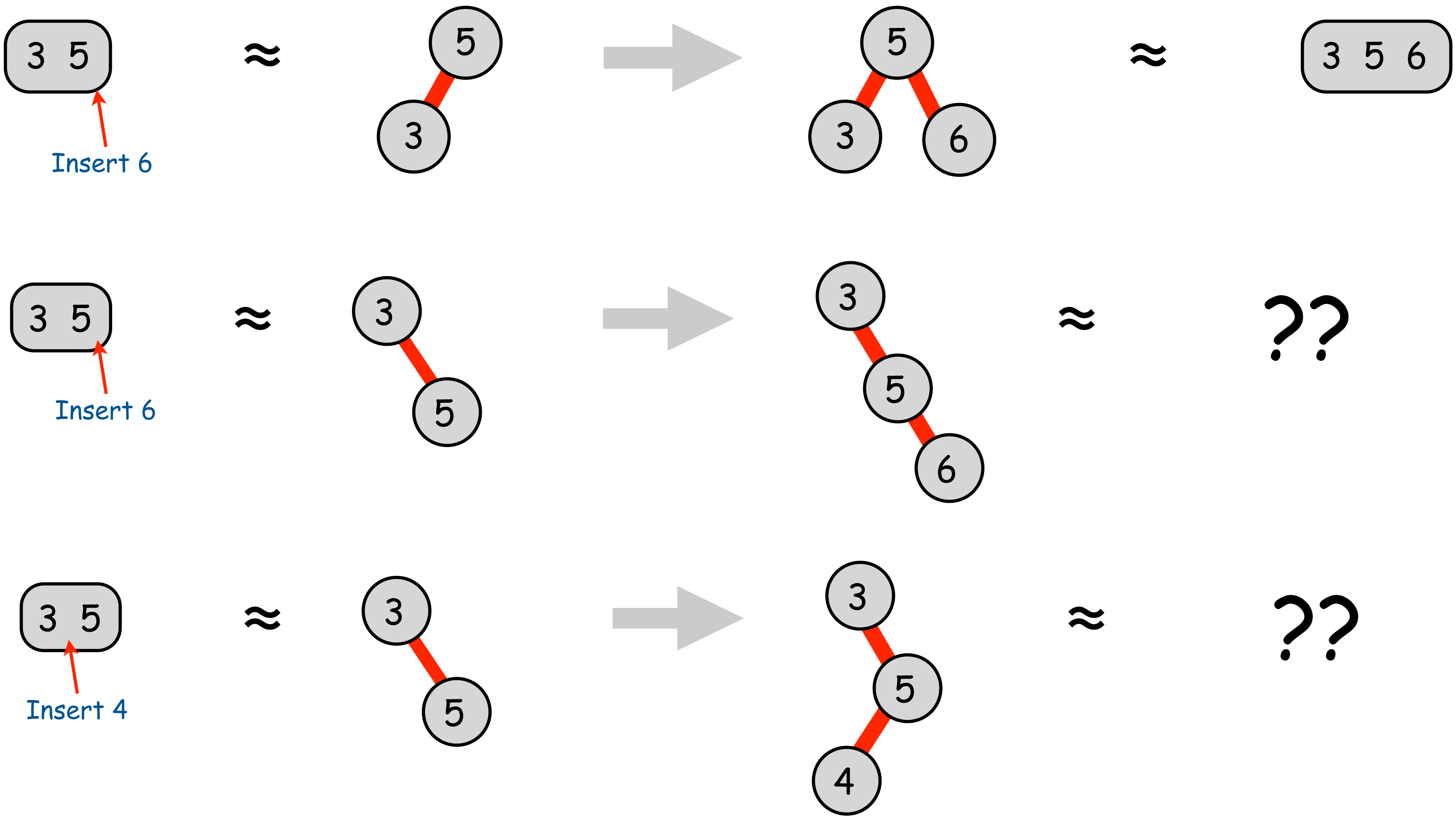
Two easy cases:





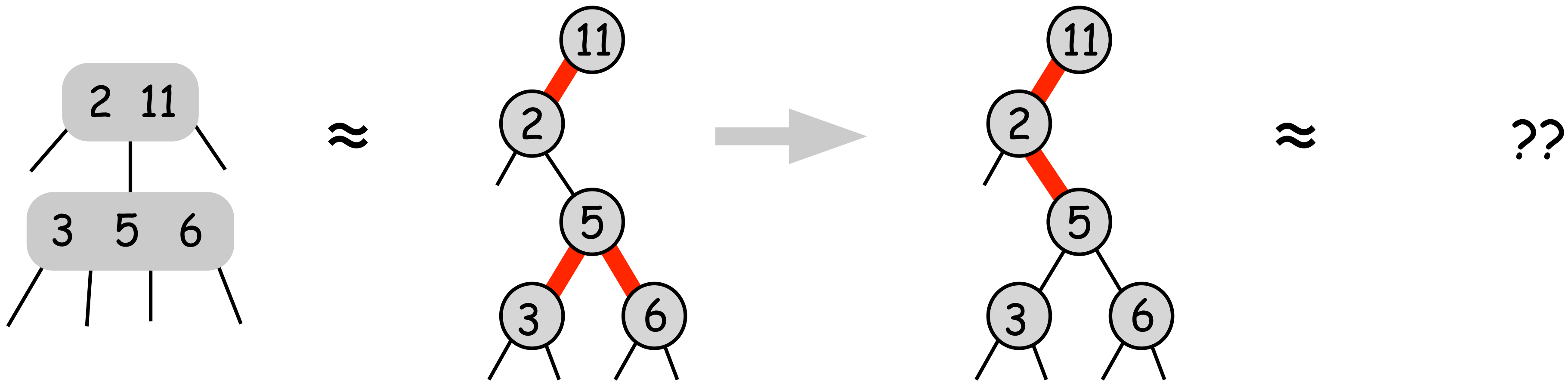
Insertion in red-black trees

Insertion in 3-node (continued):



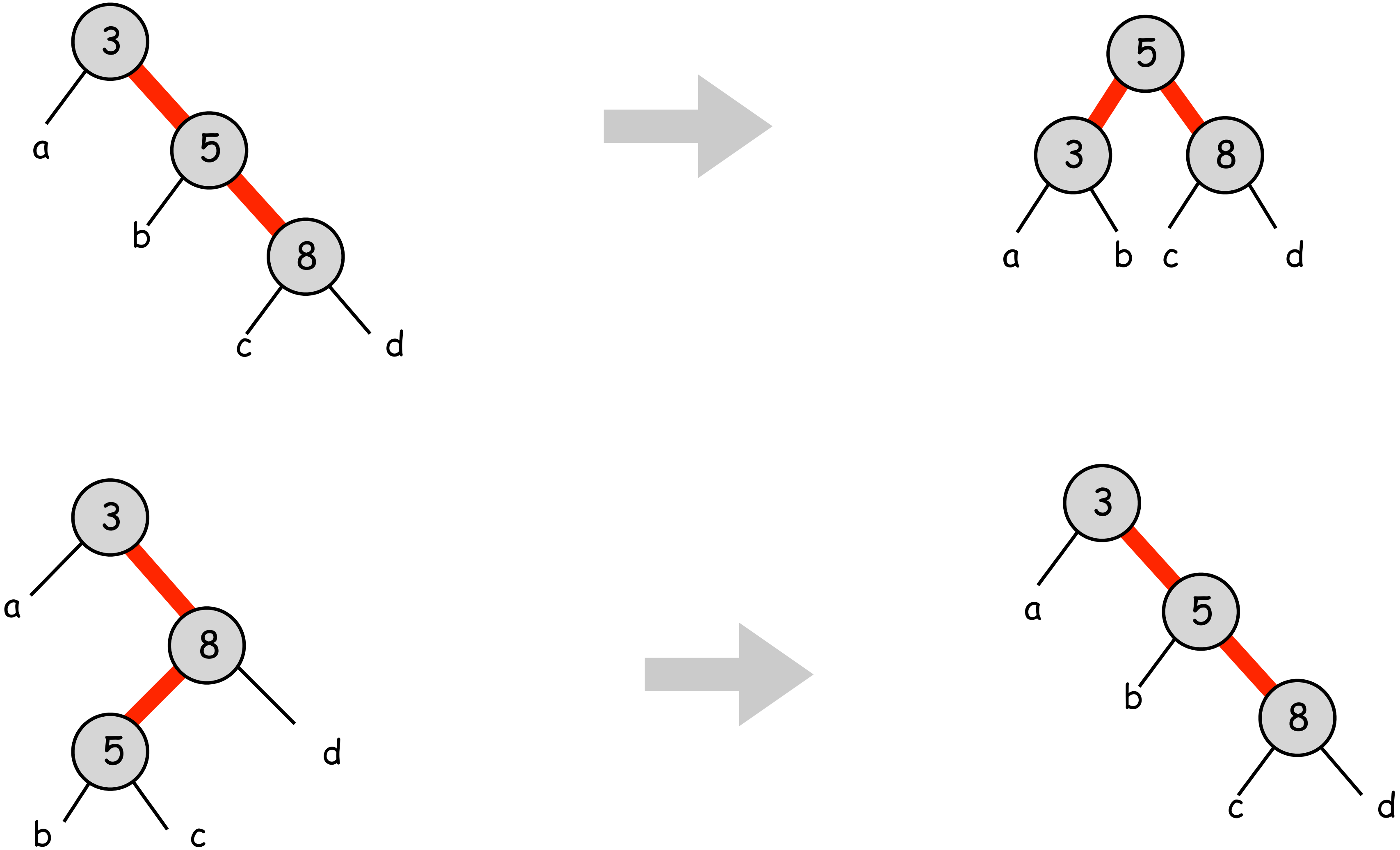
Red-black trees: Splitting of 4-nodes

Example of hard case:



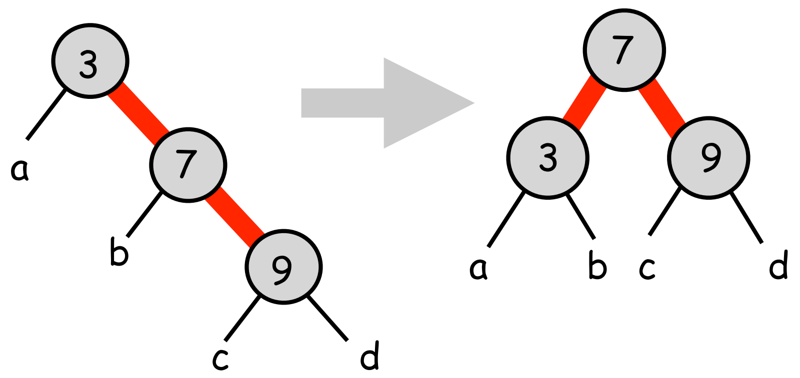
Solution: Rotations!

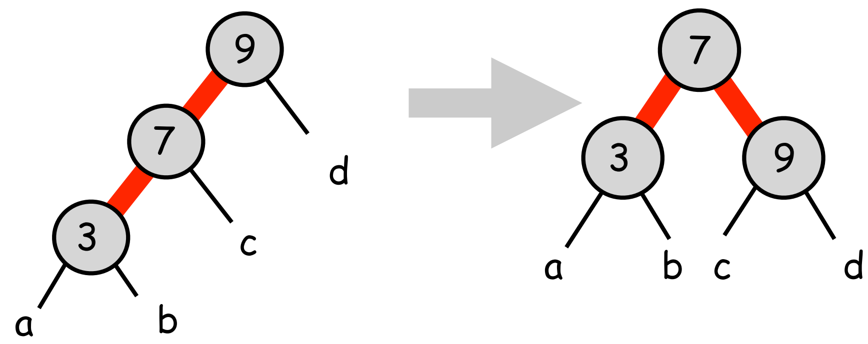
Rotations in red-black trees

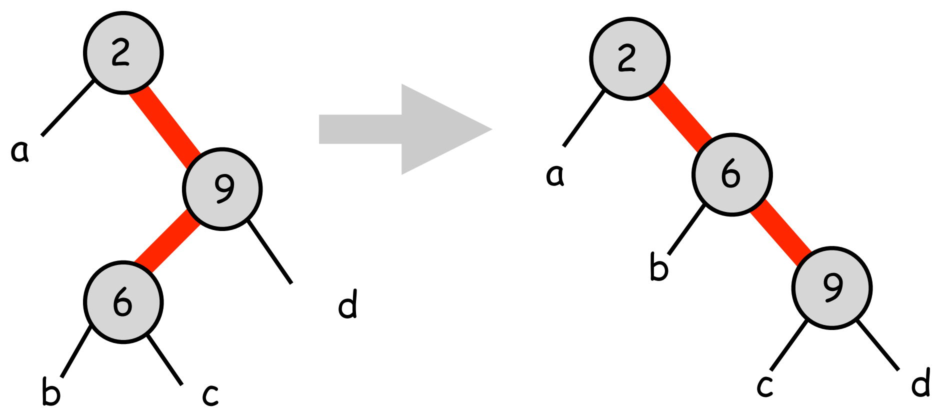


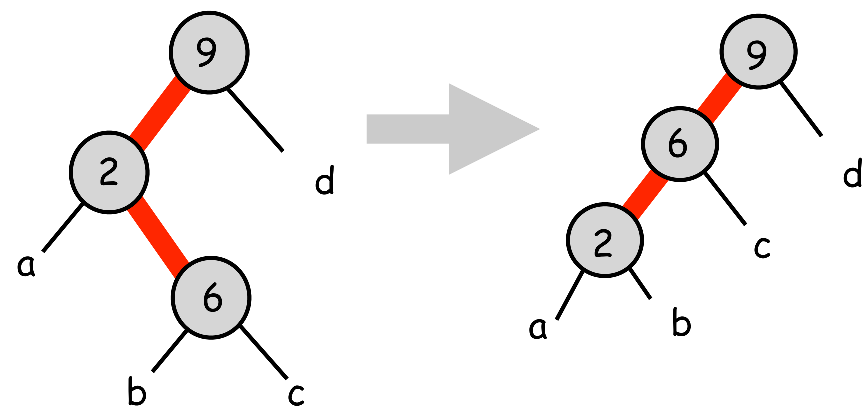
Rotations in red-black trees

Two types of rotations:



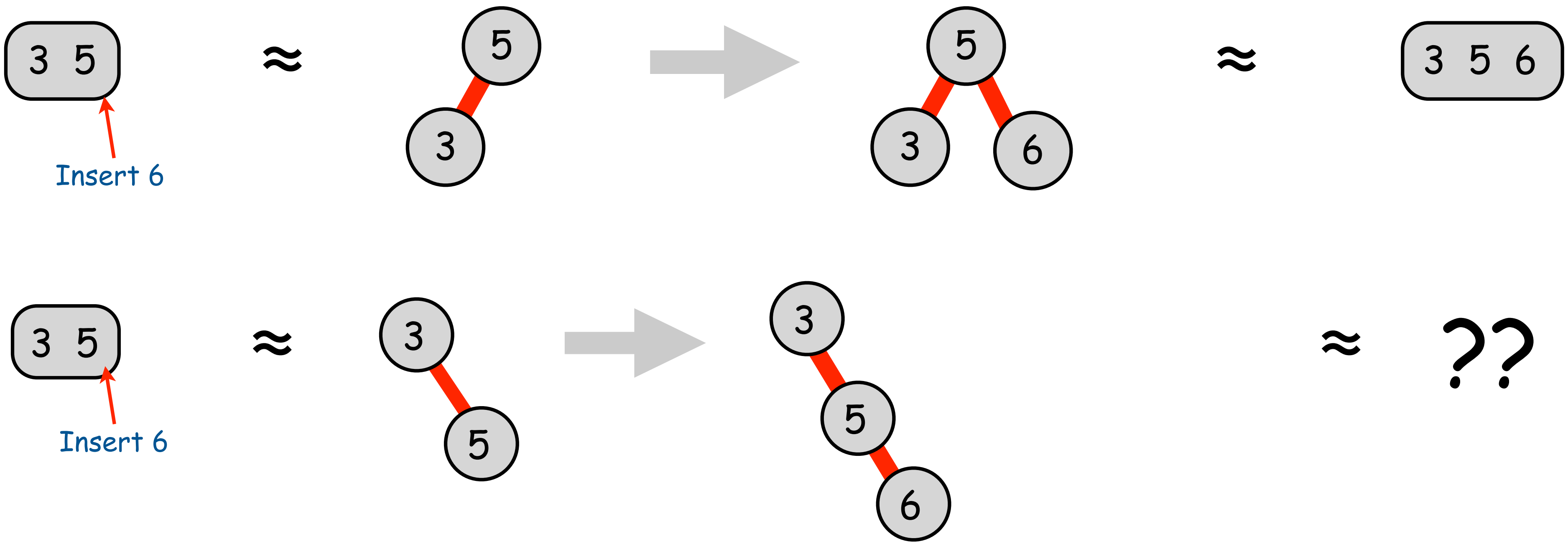






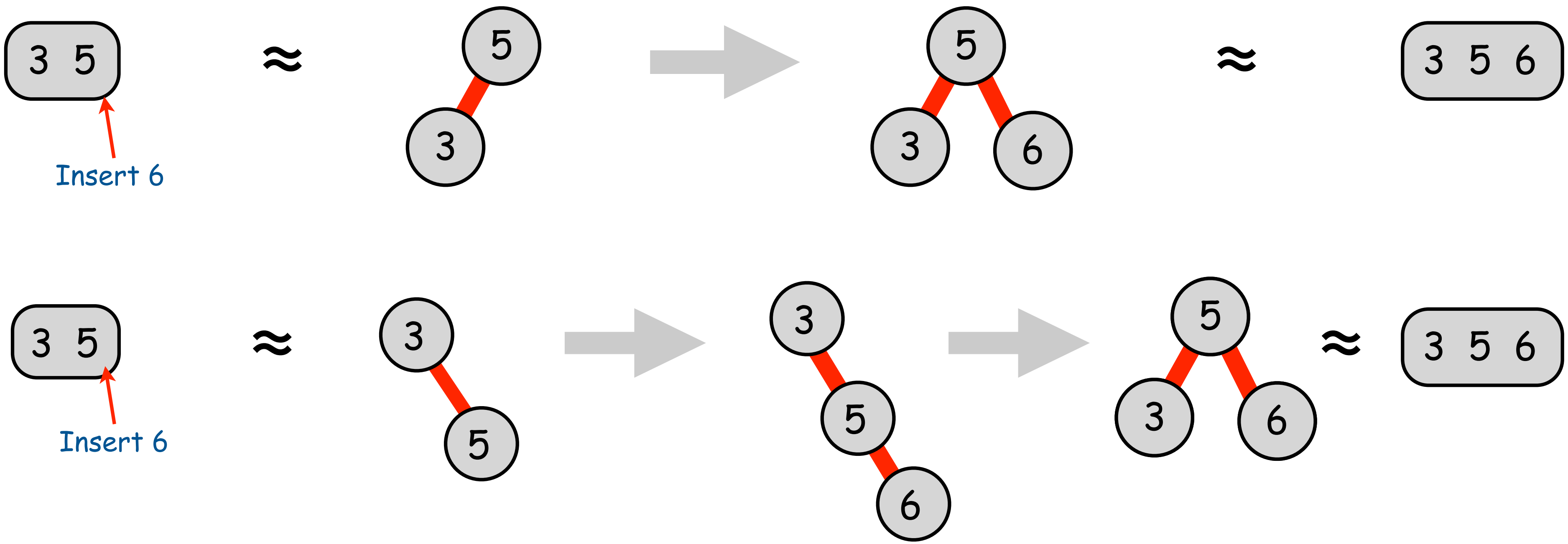
Insertion in red-black trees

Insertion in 3-node (continued):



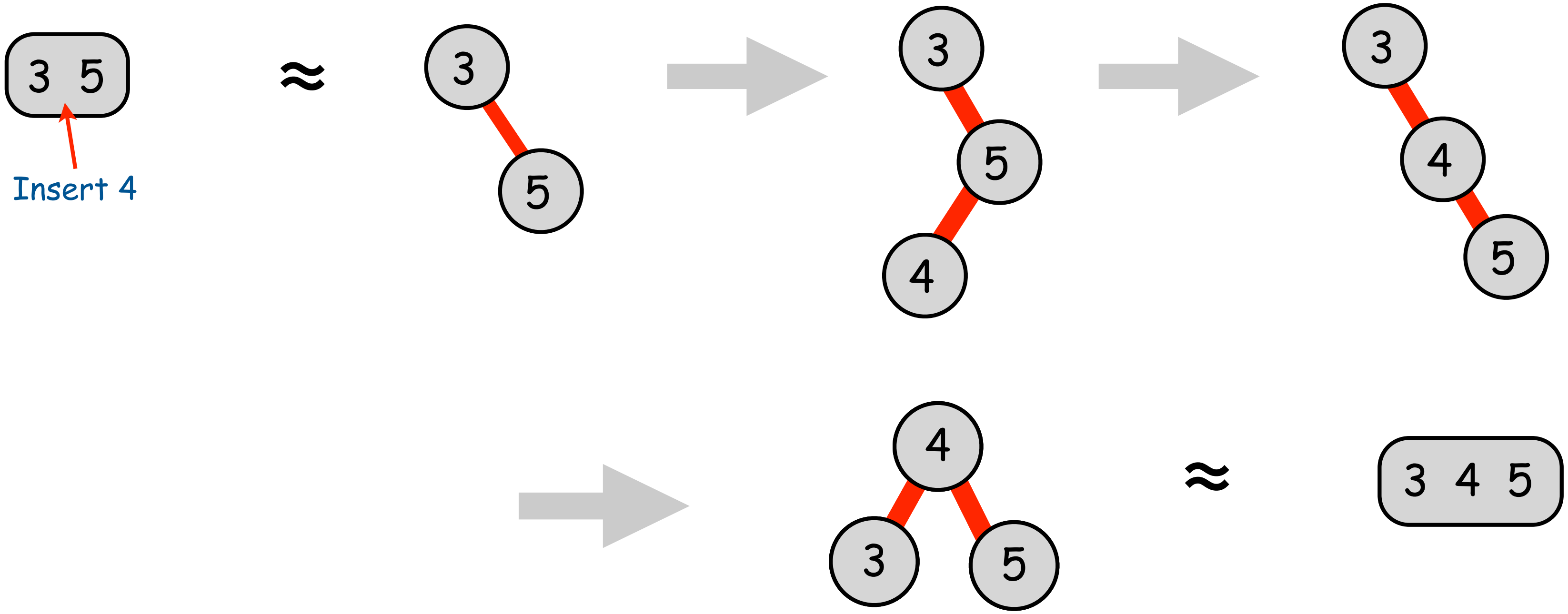
Insertion in red-black trees

Insertion in 3-node (continued):



Insertion in red-black trees

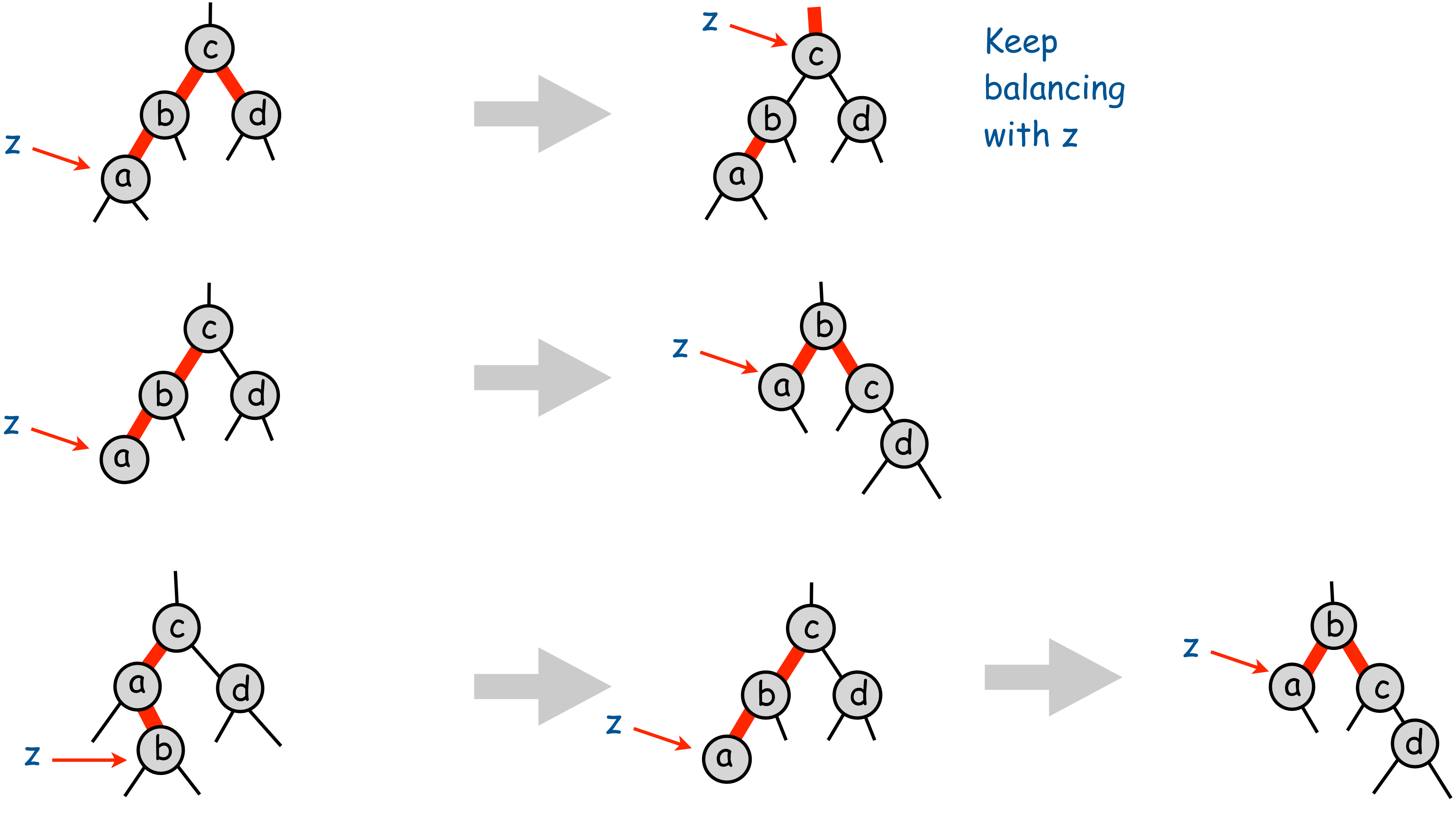
Insertion in 3-node:



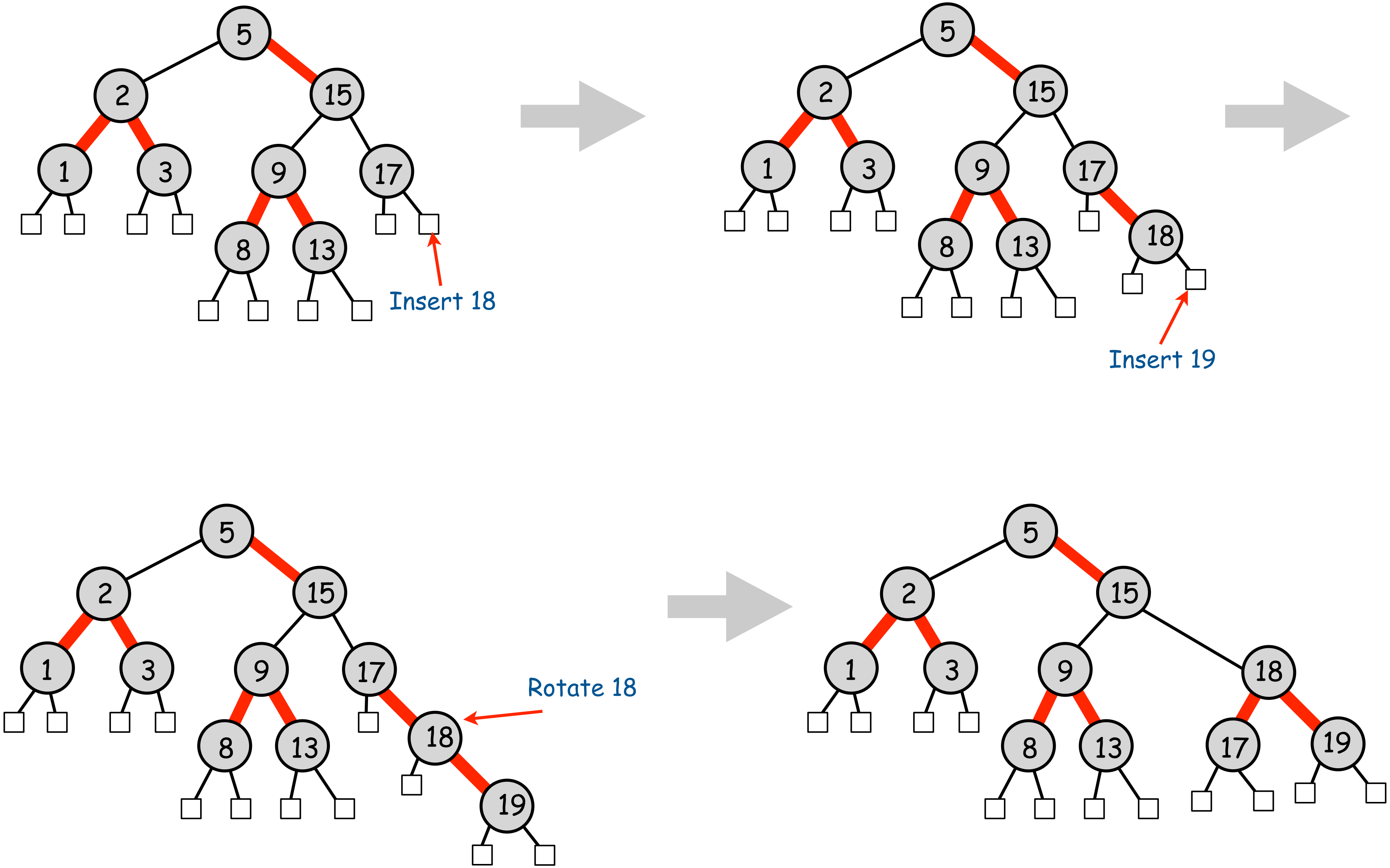
Insertion in red-black tree

Insert x:

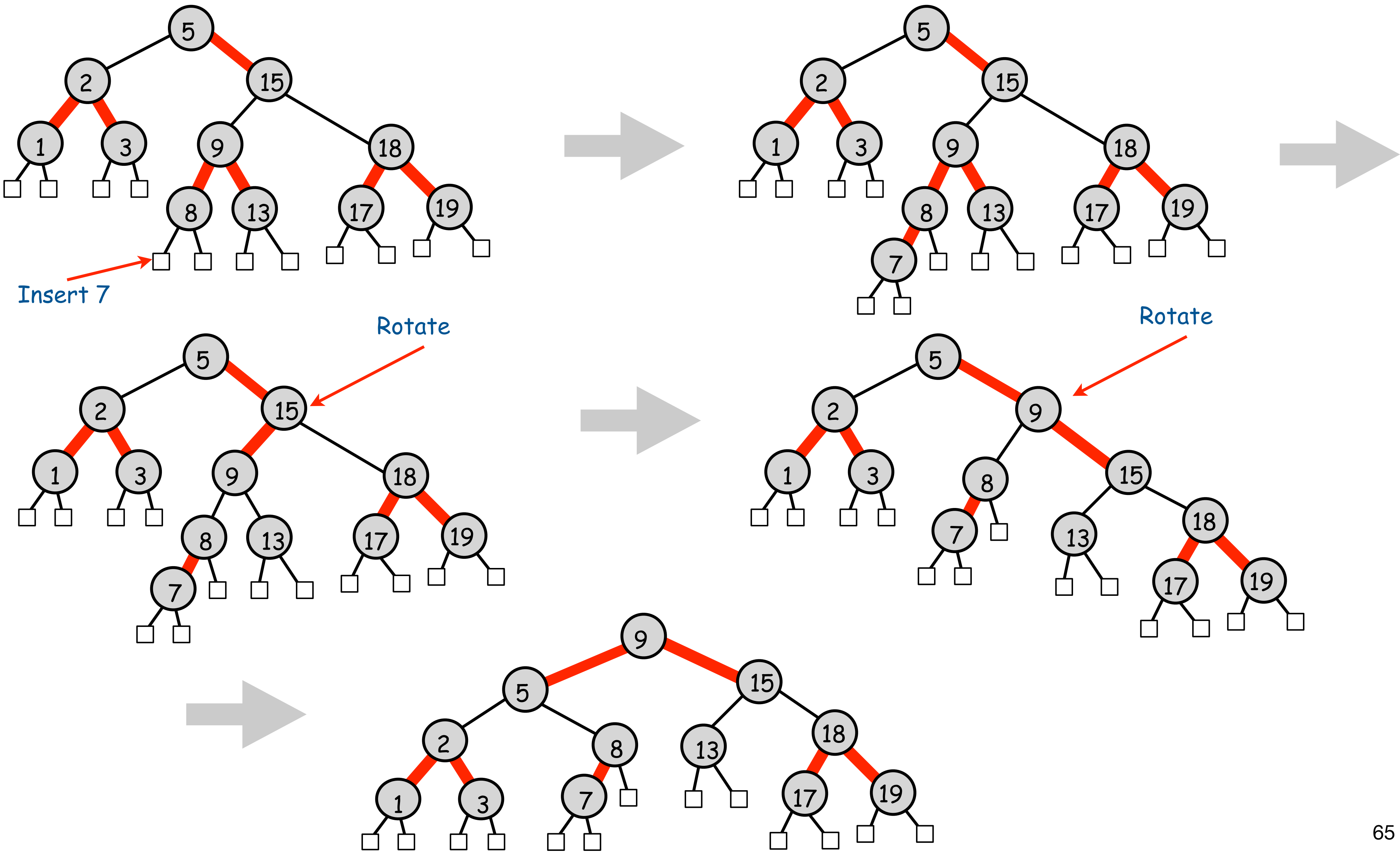
- Search to bottom after key (x)
- Insert leaf with red edge to parent
- Balance: 3 cases (+ symmetric)



Eksempel



Example



Running times in red-black trees

- Time for insertion:
  - Search to bottom after key: O(h)
  - Insert leaf with red edge: O(1)
  - Perform recoloring and rotations on way up: O(h)
    - Can recolor many times (but at most h)
    - At most 2 rotations.
- Total O(h).

- Time for search
  - Same as BST: O(h)

- Height: O(log n)

Dynamic set implementations

Worst case running times

Implementation	search	insert	delete	minimum	maximum	successor	predecessor
linked lists	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
ordered array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(log n)	O(log n)
BST	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)	O(h)
2-3-4 tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
red-black tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Balanced trees: implementations

Redblack trees:

Java: **`java.util.TreeMap, java.util.TreeSet.`**

C++ STL: map, multimap, multiset.

Linux kernel: **`linux/rbtree.h.`**