Data Structures

October 26

The Family of B-trees

- B* - trees
- B^+ - trees
- Simple prefix B^+ - trees
- prefix B^+ - trees
- R-trees
- 2-4 B-trees, vh-trees, **red-black trees**

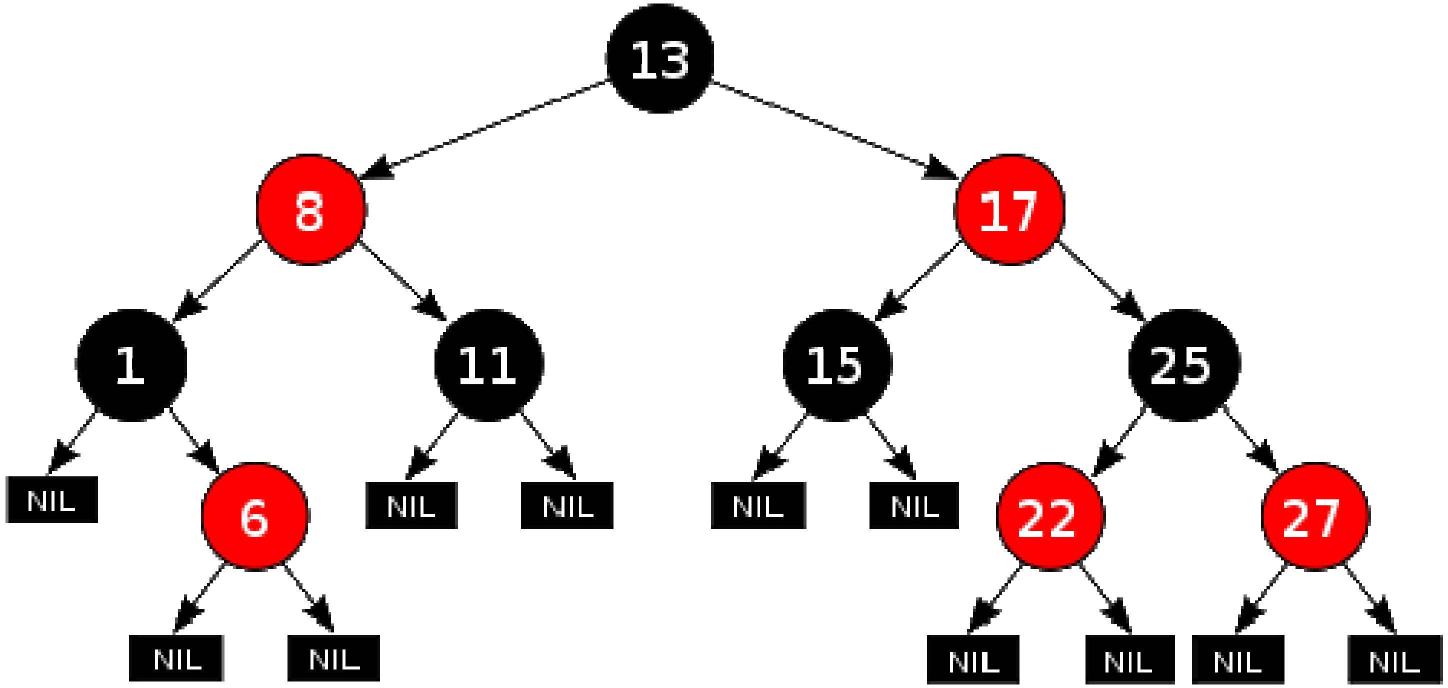
Red-Black Trees

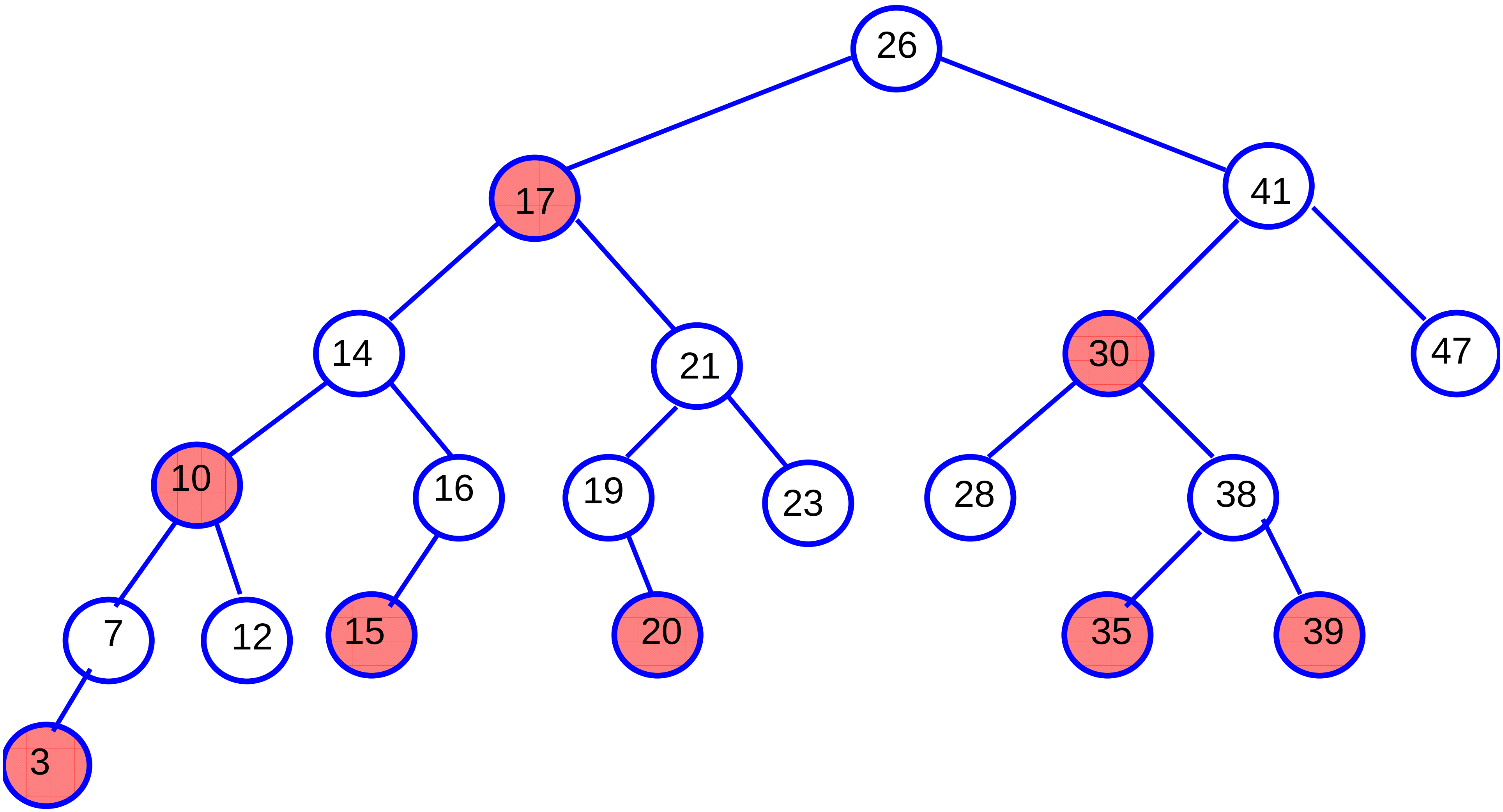
Red-Black Tree is a BST (binary search tree) with the following properties:

1. Each node is either black or red
2. The root of the tree is black
3. If a node is red, than its children are black
4. Each simple path from a given node to any of its descendant leaves contains the same number of black nodes
5. Each node is made to have two children (also the leaves); these added nodes are called *external leaves;* these external leaves are *black*

Key, parent, left, right, color

A Red-Black Tree
Nodes are made internal by attaching (black) external nil nodes

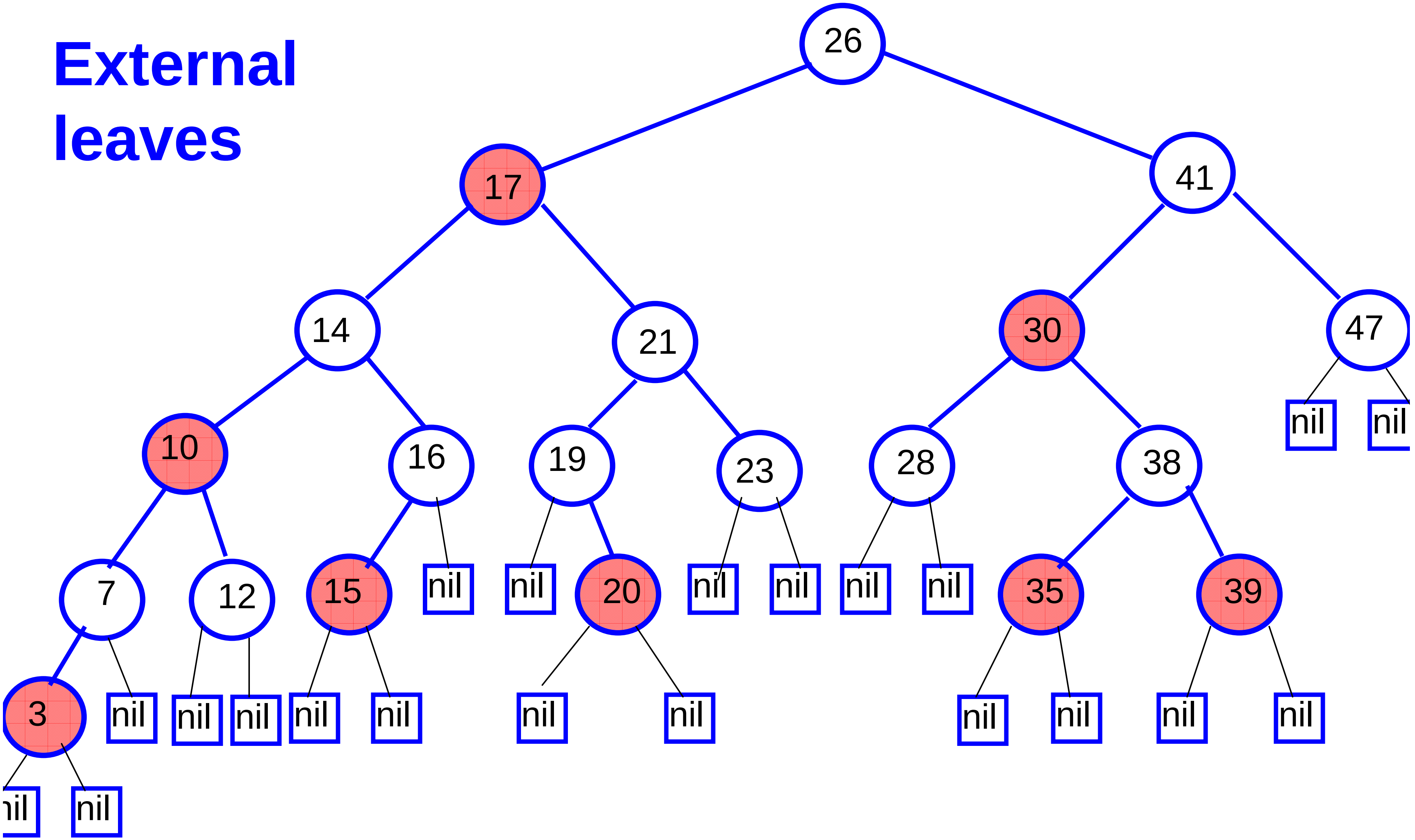




A Red-Black Tree

Black nodes are shown in *white*

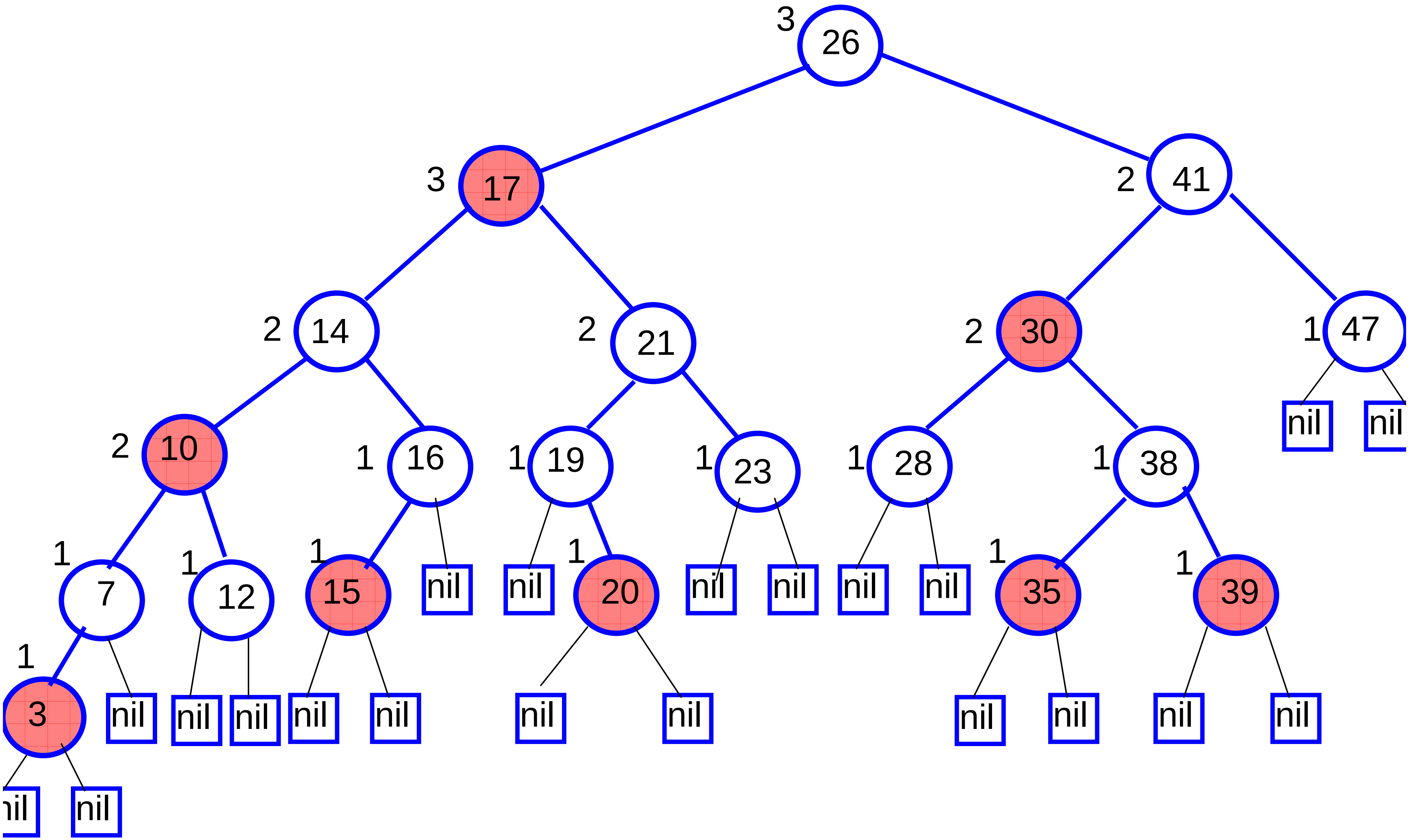
External leaves



A Red-Black Tree; external leaves are shown

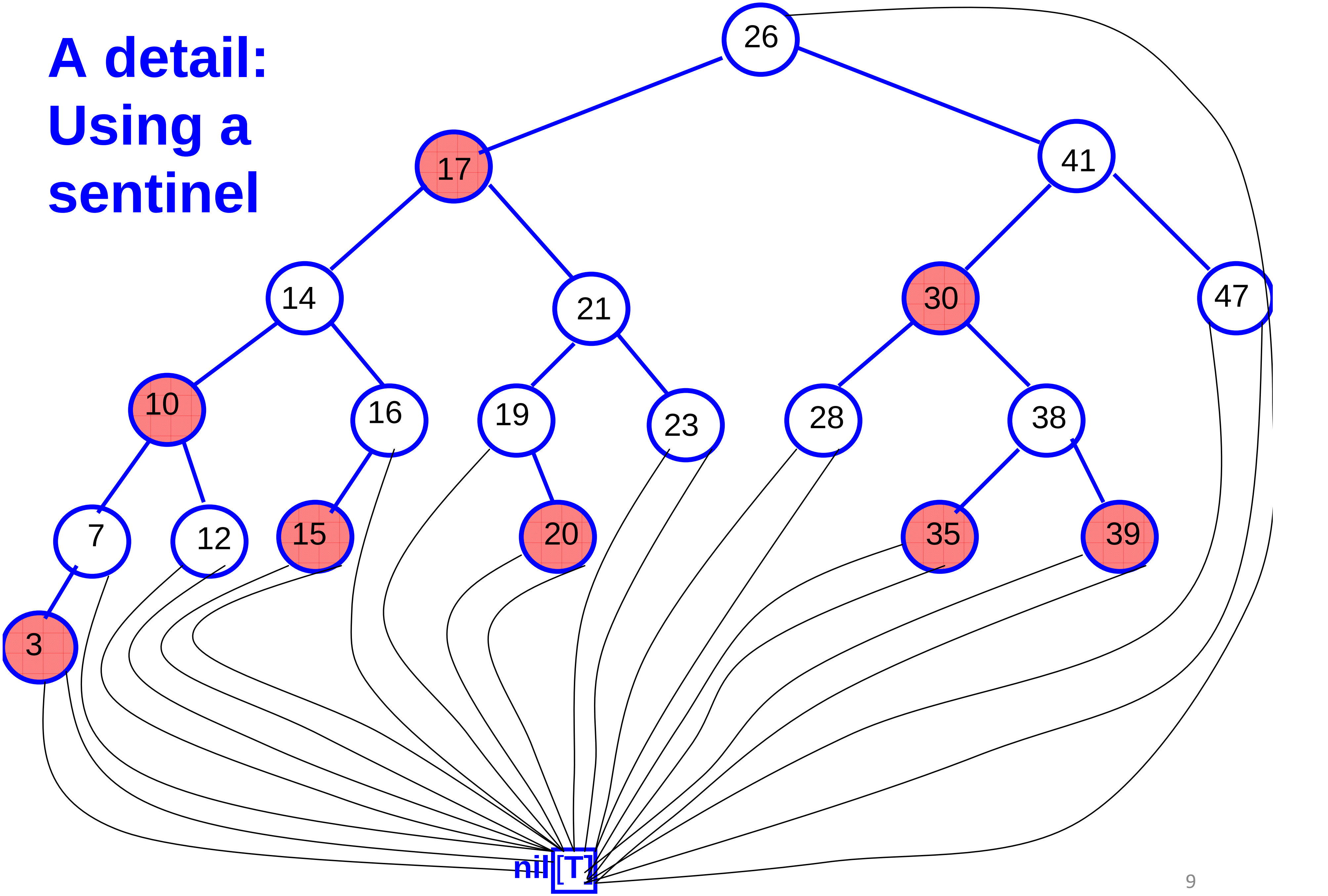
Red-Black Trees

- **bh(*v*)** = the number of black nodes (not counting *v* if it is black) from *v* to any leaf in the subtree (called the black-height).
- (As done before: we will denote by **h(*v*)** the height of the subtree rooted at node *v*)



A Red-Black Tree; external leaves and black heights (bh(x)) are shown

A detail: Using a sentinel



Red-Black Trees

- Statement: a red-black tree which contains n *internal* nodes has a height of O(log(n)). (Internal nodes are the nodes of the original tree without the nil leaves decoration.)
- This follows easily from the following assertion: a subtree in a red-black tree rooted at *v* has *at least* $2^{bh(v)} - 1$ *internal* nodes.
- Proof by induction on h(*v*), the *height* of *v.*

Base case: h(*v*) = 0; if *v* has height zero it must be nil, thus bh(*v*) = 0; and the base case follows ($2^{bh(v)} - 1 = 2^0 - 1 = 0$).

Red-Black Trees

Induction Step

Induction hypothesis: if *v* is such that h(*v*) = k, then the subtree rooted *v* has at least $2^{bh(v)} - 1$ *internal* nodes.

This implies that, if *v'* is such that it has h(*v'*) = k+1, then subtree rooted at *v'* has $2^{bh(v')} - 1$ *internal* nodes. For:

v' has h(*v'*) > 0. Thus it has two children, each of which has black-height of either bh(*v'*) or bh(*v'*) -1. By ind. hyp. each child has at least $2^{bh(v')-1} - 1$ *internal* nodes, so *v'* has at least

$2^{bh(v')-1} - 1 + 2^{bh(v')-1} - 1 + 1 = 2^{bh(v')} - 1$ internal nodes.

Done.

Red-Black Trees

Next we show that the height of a red-black tree with n internal nodes is O(log(n)) by using the previously derived assertion:

On any path of the root to a leaf at least half of the nodes are black (property 3: if a node is red than both its children are black), the black height of the root is h(root)/2. Combined with the assertion we have:

$$\mathbf{n \;\geq\; 2^{bh(root)} - 1 \;\geq\; 2^{h(root)/2} - 1}$$

Which is equivalent to

$$\mathbf{\log_2(n+1) \;\geq\; h(root) / 2}$$

And in turn to

$$\mathbf{h(root) \;\leq\; 2 \log_2(n+1)}$$

Thus the height is O(log(n)).

Red-Black Trees: Insertion

Put the new node X as in any BST- variant (binary search tree) insertion in the appropriate leaf

What color will the inserted node get? The wise decision is to color it red.

Next we have to worry about restoring properties 1-5. We consider the following cases:

a) Tree was empty before insertion: the new node is the root; color it black

Red-Black Trees: Insertion

b) The inserted node X has a father: two subcases:

1) the father is black (we are done);

2) the father is red: so the inserted node has a grandfather who is necessarily black (property 3);

b)2) has again two subcases:

i) the grandfather of X has another child (the brother of the father; the uncle of X) and it is black *or* the grandfather has only one child; in this case we do a rotation – we will describe this in full generality for any node X being red, the father is also red, no uncle or a black uncle;

ii) the uncle of X is red: flip-flag;

14

Red-Black Trees: Insertion

b) 2) ii) the uncle of X is red: flag-flip; now the grandfather is red which might violate the rules of a red-black tree, so now we let the role of X be played by the grandfather (who became red);

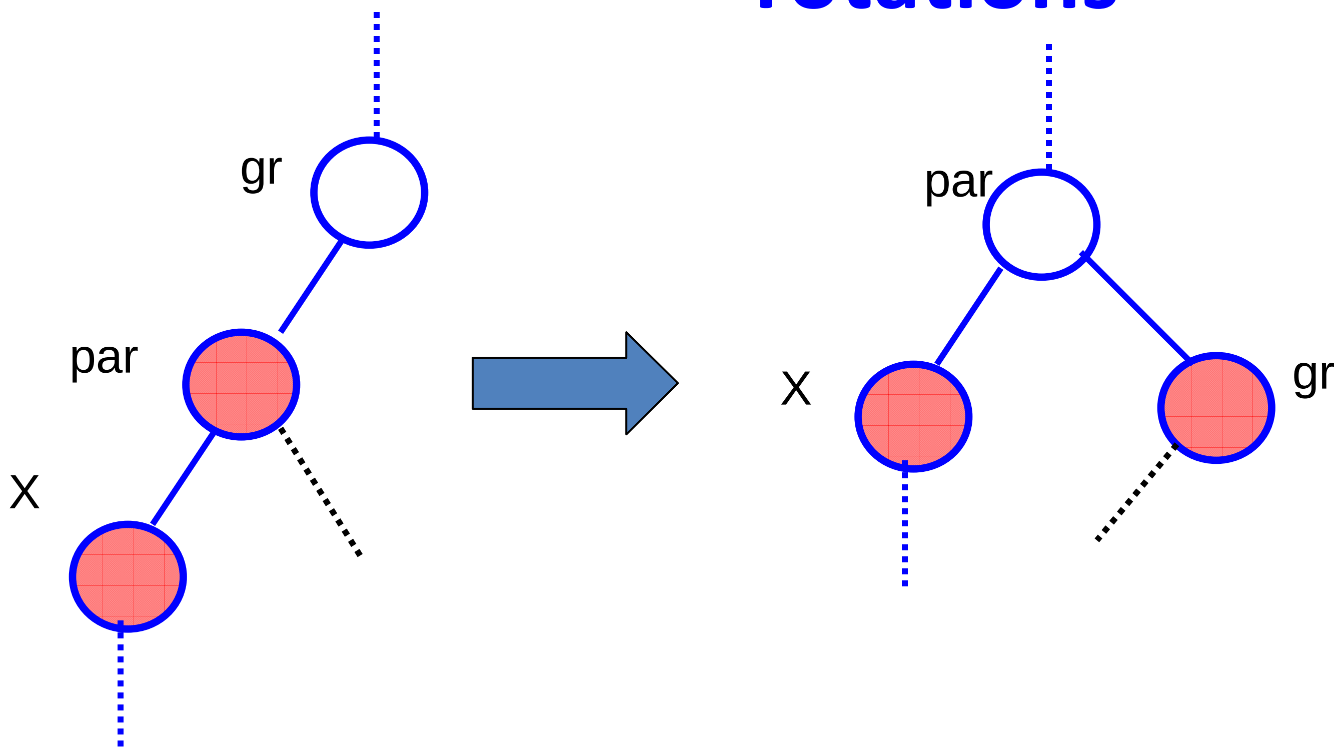
If the grandfather is the root, then we color the root black and we are done: we get a red-black tree whose black height has increased by 1.

If the grandfather is not the root, we proceed as before; this process will not go on indefinitely, of course.

Remark: rotations and flag-flip do not change the black height of the tree; the changing of the color of the root from red to black is the only case in which the black height is changed (by +1).

15

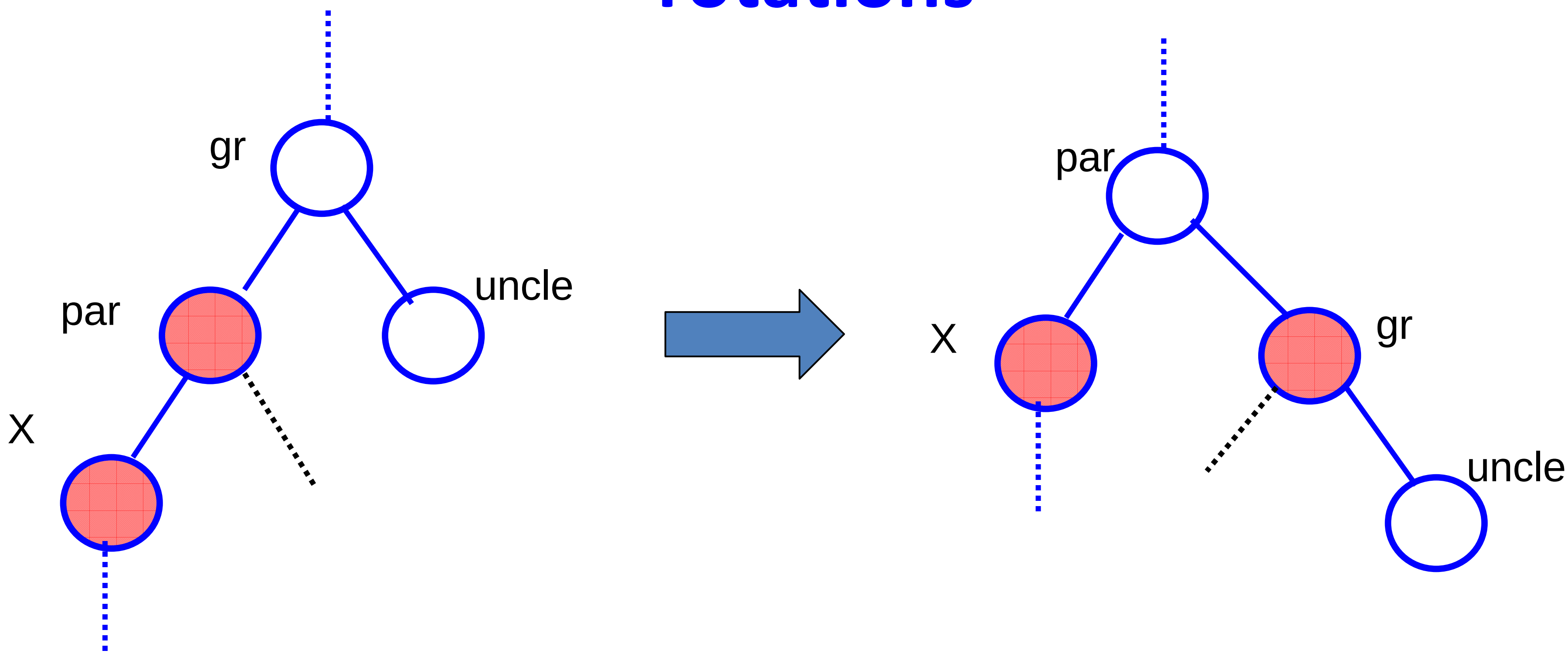
Red-Black Trees: Insertion; description of rotations



Case: “left-left”; no uncle (next page: uncle is black)

Of course: parent right child of grandfather; X right child of parent is dealt with analogously by using a left rotation

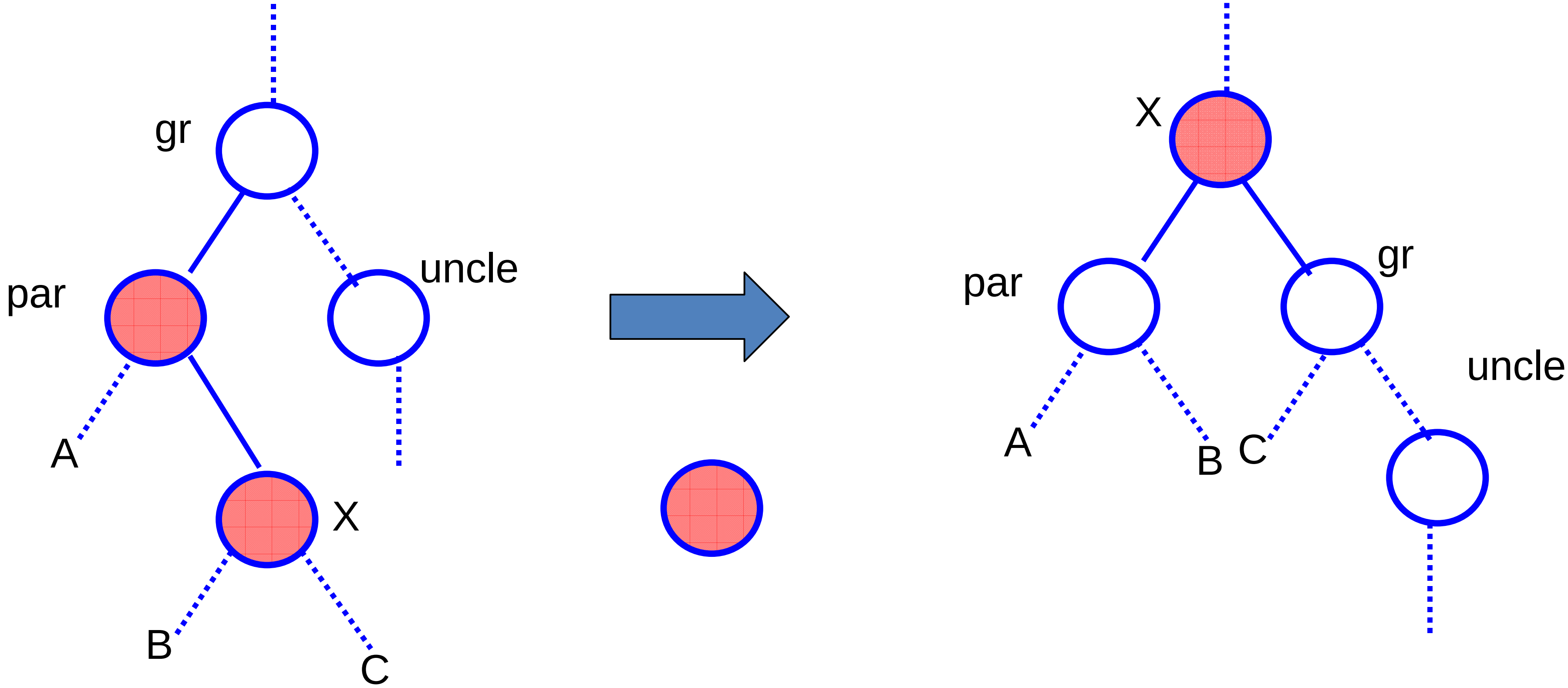
Red-Black Trees: Insertion; description of rotations



Case: “left-left” and uncle is black)

Of course: parent right child of grandfather; X right child of parent is dealt with analogously by using a left rotation

Red-Black Trees: Insertion; description of rotation

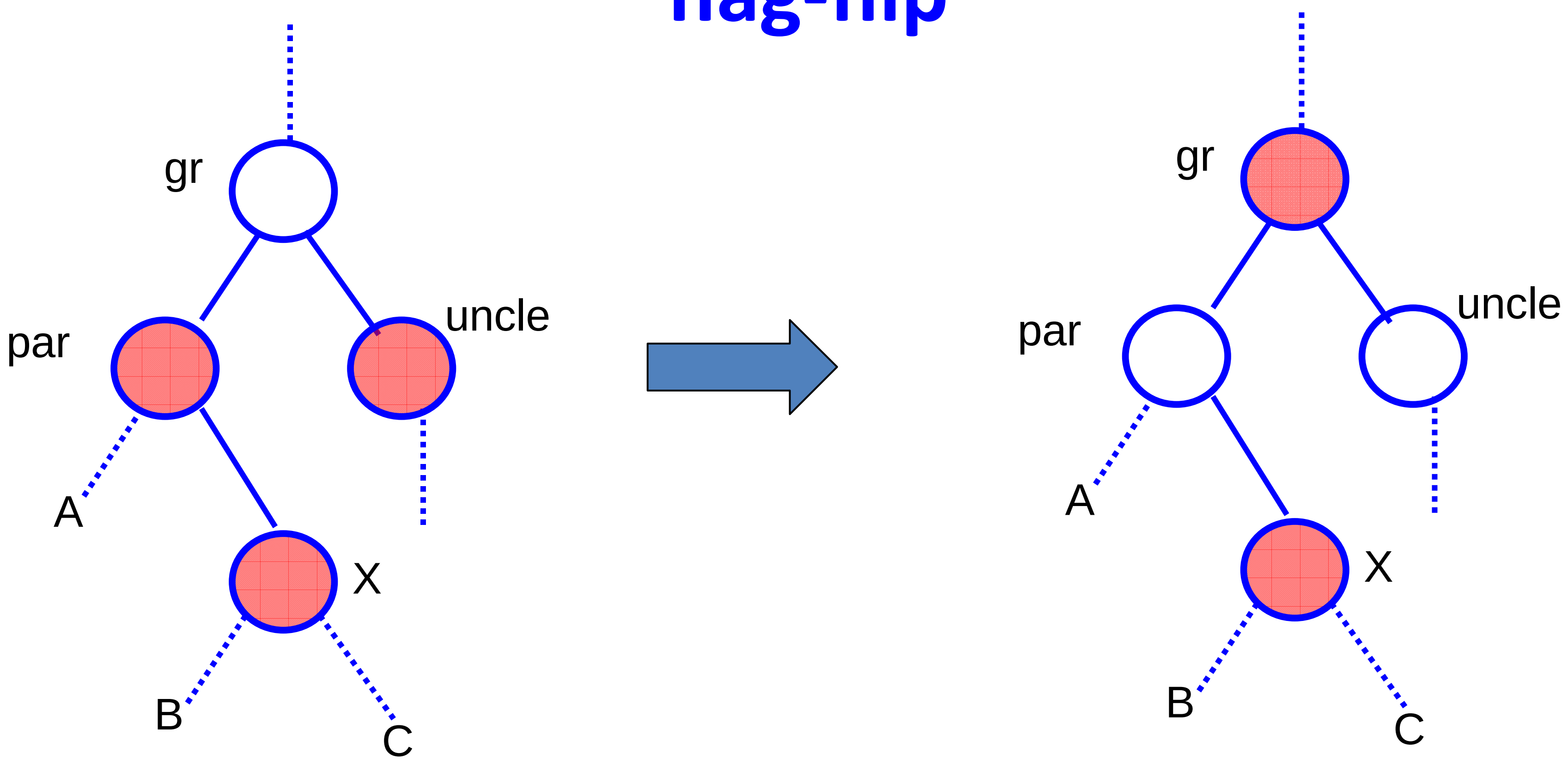


Case “right-left” and no uncle or black uncle

Double rotation   black height does not change

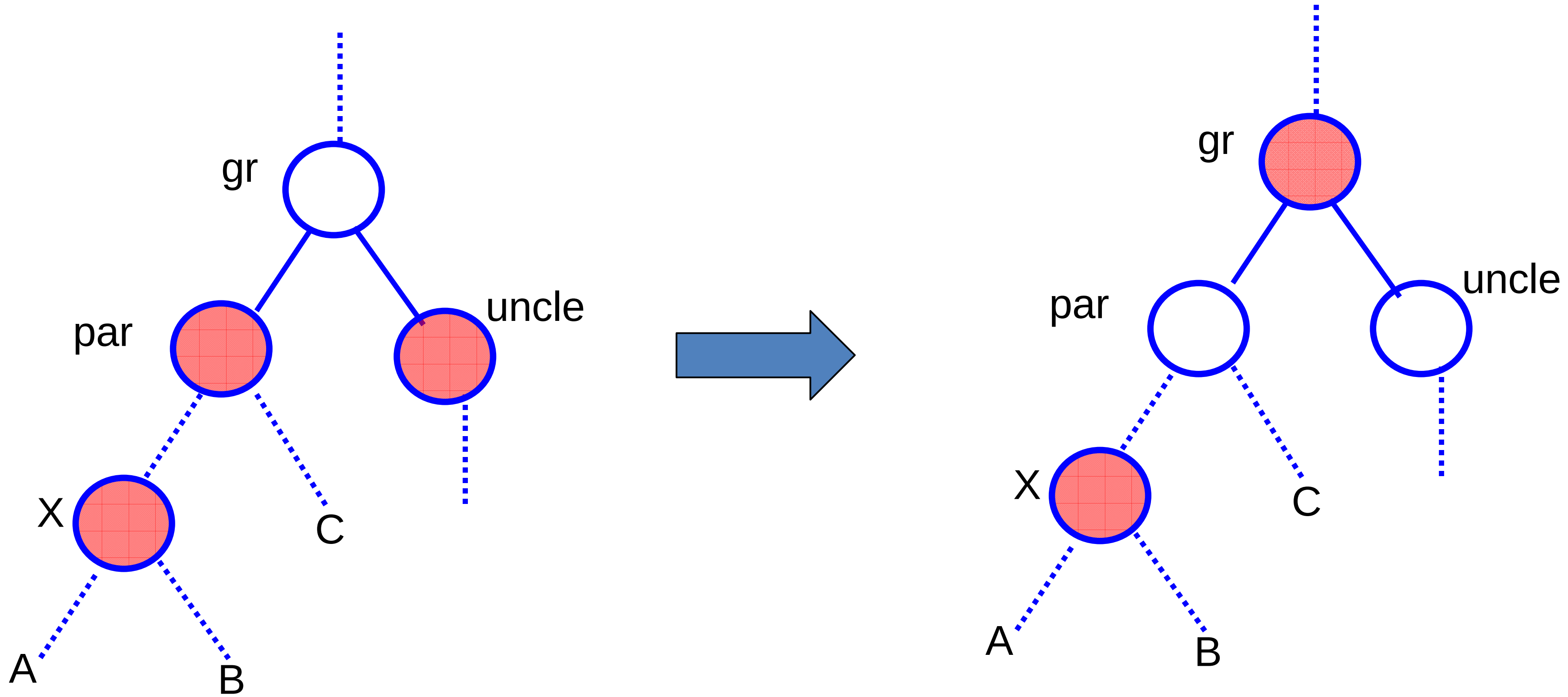
Case “left-right” is done similarly

Red-Black Trees: Insertion; description of flag-flip



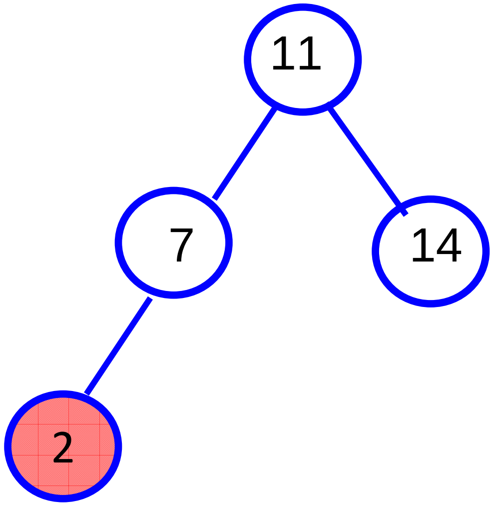
Case of red uncle: flag-flip; now grandfather (gr) can cause trouble

Red-Black Trees: ((Insertion; description of flag-flip)) this case should already be clear



Case of red uncle: flag-flip; now grandfather (gr) can cause trouble

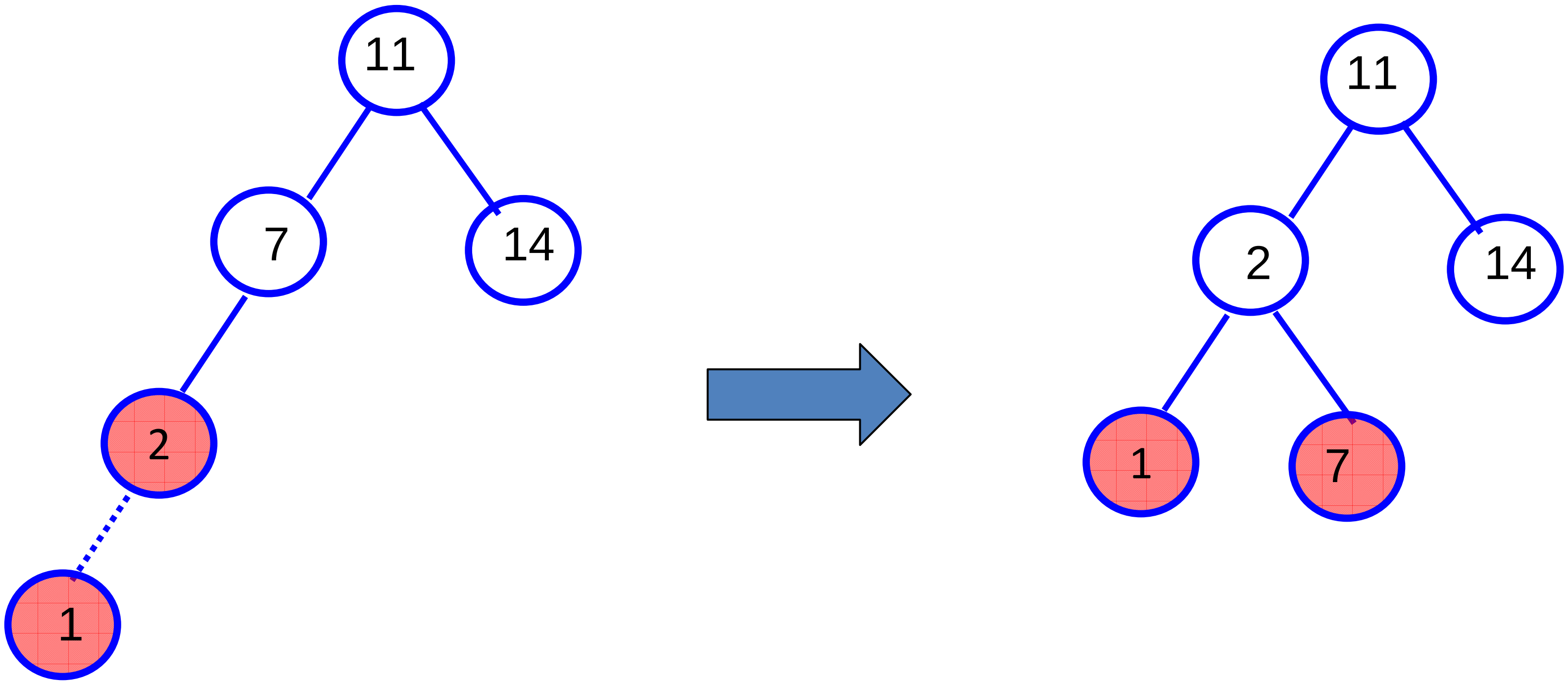
Red-Black Trees: Insertion Examples



Insert 1

R

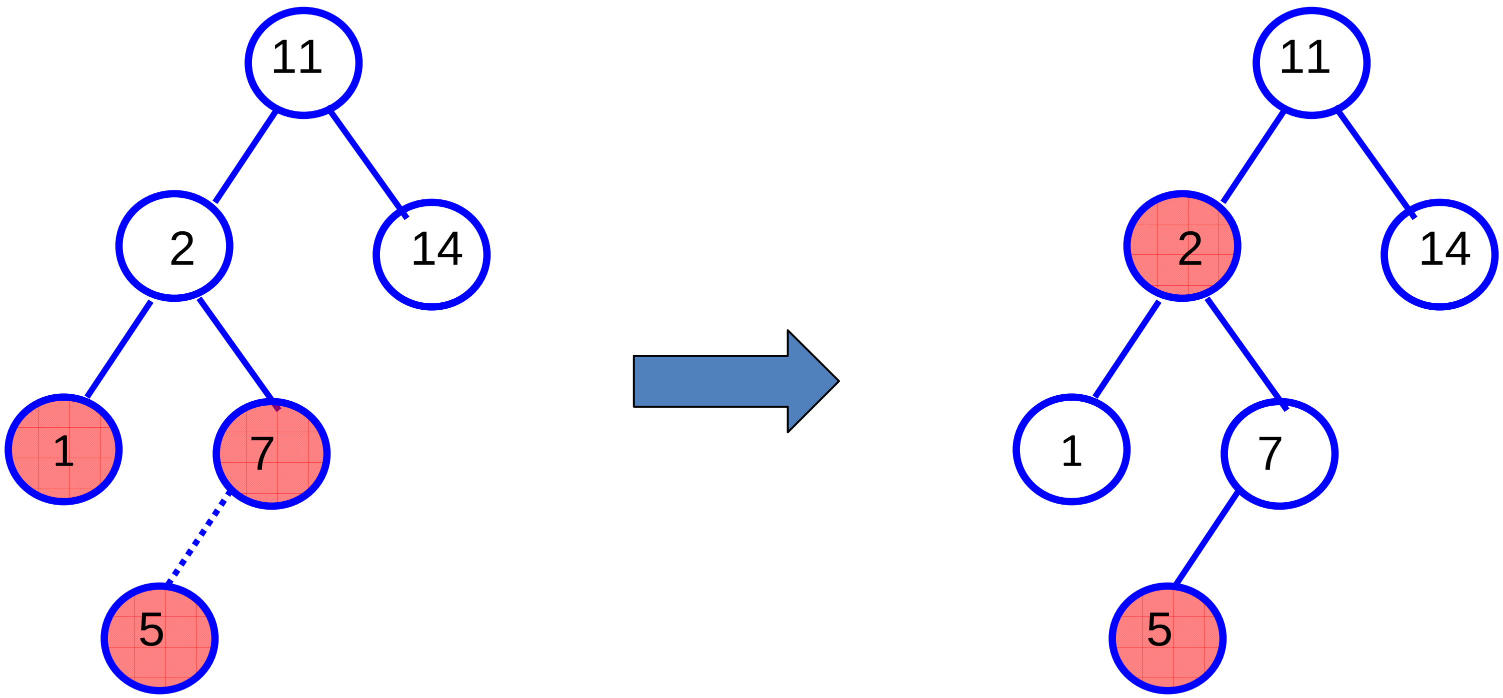
Red-Black Trees: Insertion Examples



Single right rotation

Next: Insert 5

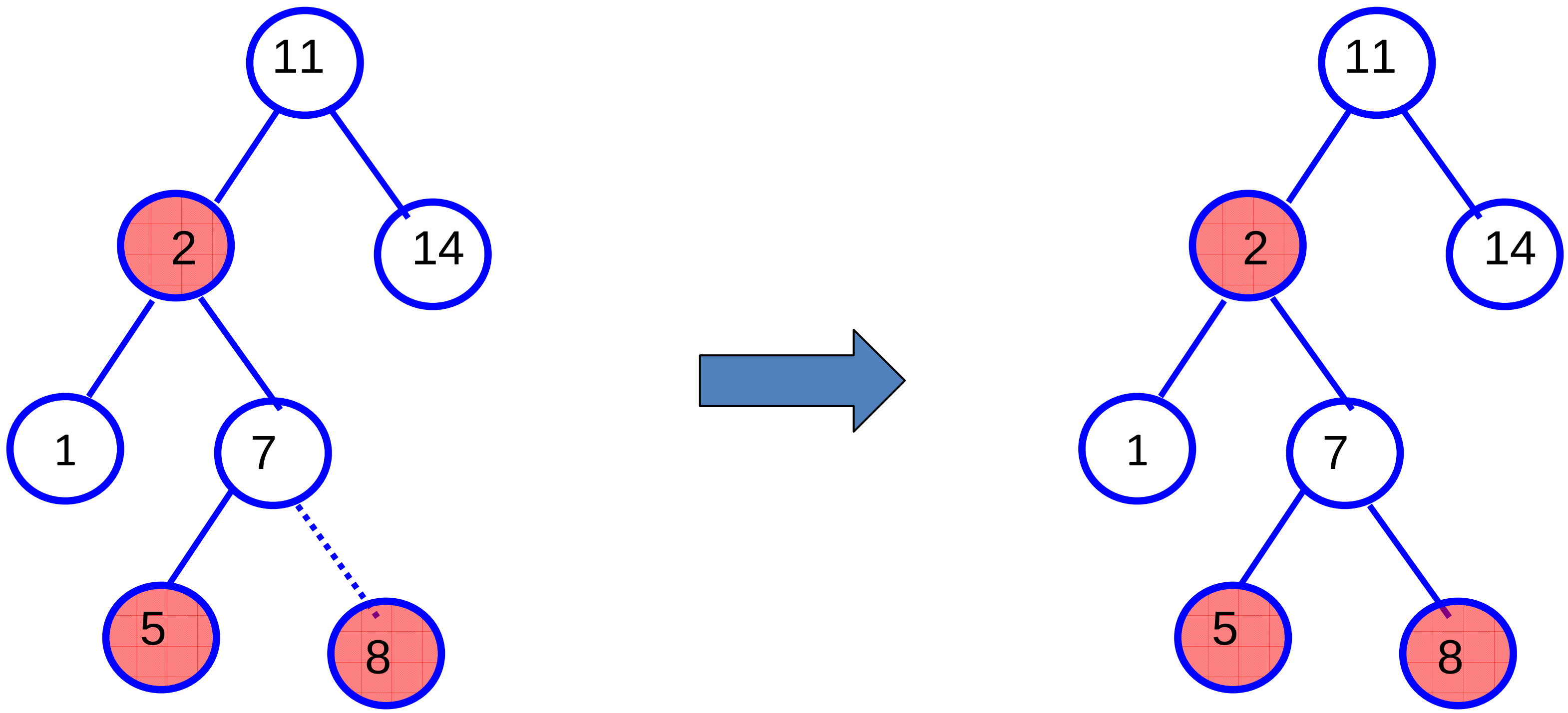
Red-Black Trees: Insertion Examples



Insert 5;   flag-flip (can stop)

Next insert   8

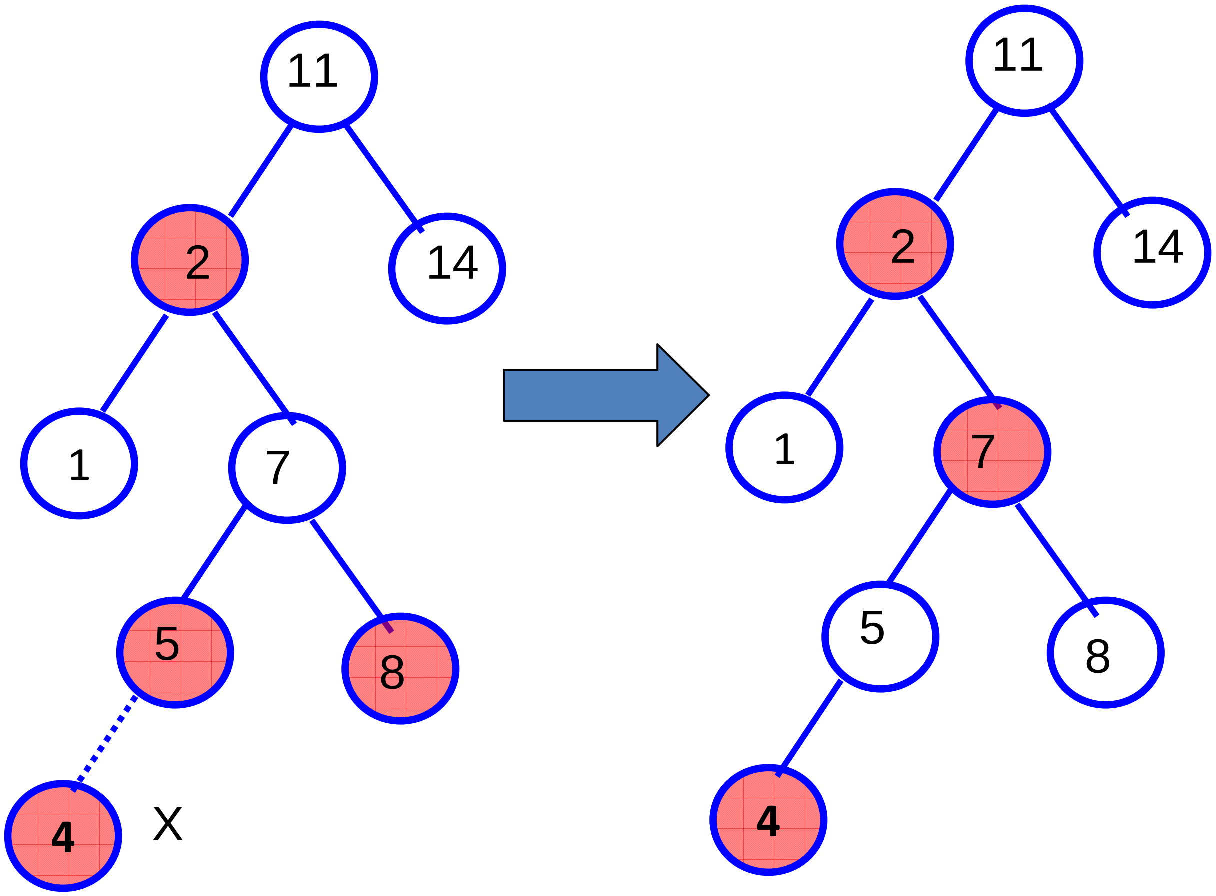
Red-Black Trees: Insertion Examples



Insert 8; nothing to be done

Next: insert 4

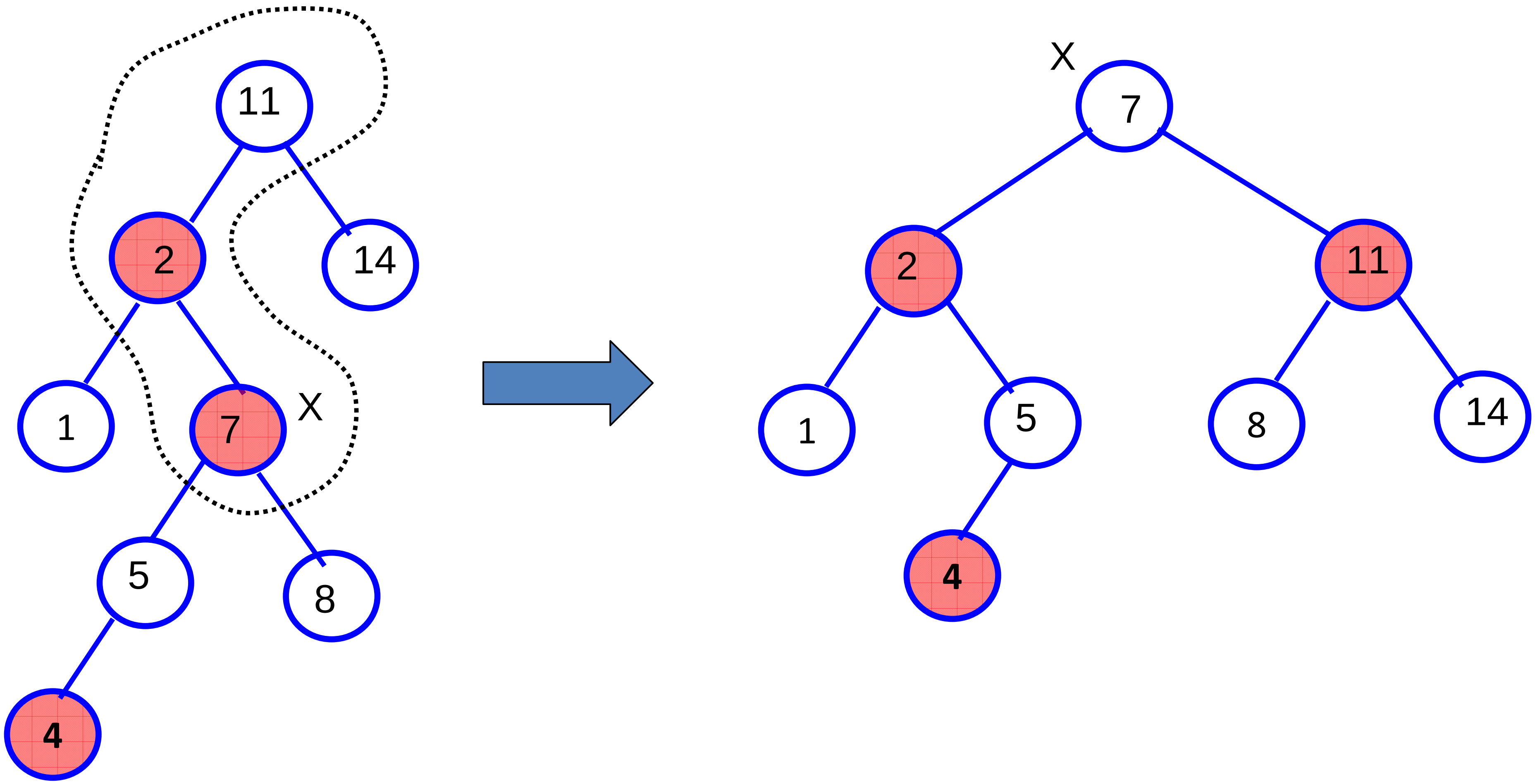
Red-Black Trees: Insertion Examples



insert 4: flag-flip;

Flag-flip brings grandfather in trouble;
Next: The role of X is played by the grandfather
(The node with key 7)

Red-Black Trees: Insertion Examples



Flag-flip brings grandfather in trouble;
Next: The role of X is played by the grandfather
(The node with key 7)

Double rotation

Red-Black Trees

- Discussion of deletion in Red-Black trees
- Discussion of relation with B-trees
- How are the operations on Red-Black trees reflected at the B-tree counterpart
- Comparison with AVL trees

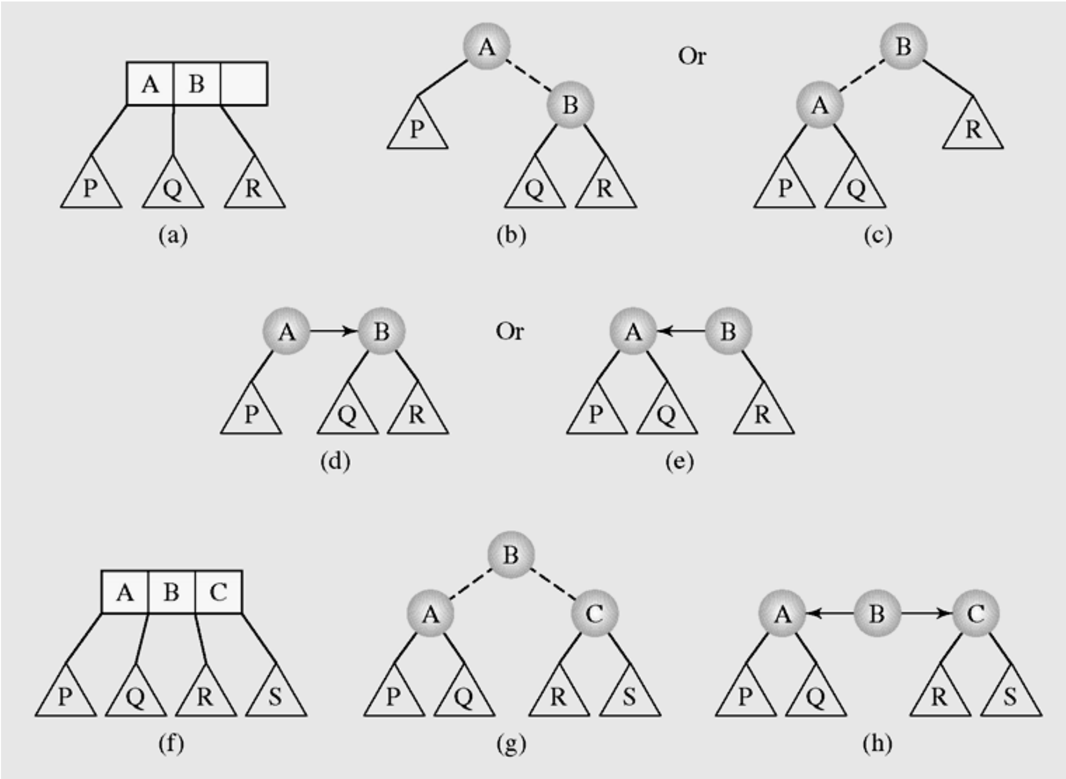
Next slides give an alternative approach to red-black trees;
These (until slide 51) can be skimmed or skipped ☺

2–4 Trees

- In 2–4 trees, only one, two, or at most three elements can be stored in one node
- To represent a 2–4 tree as a binary tree, two types of links between nodes are used:
  - One type indicates links between nodes representing keys belonging to the same node of a 2–4 tree
  - Another represents regular parent–children links

29

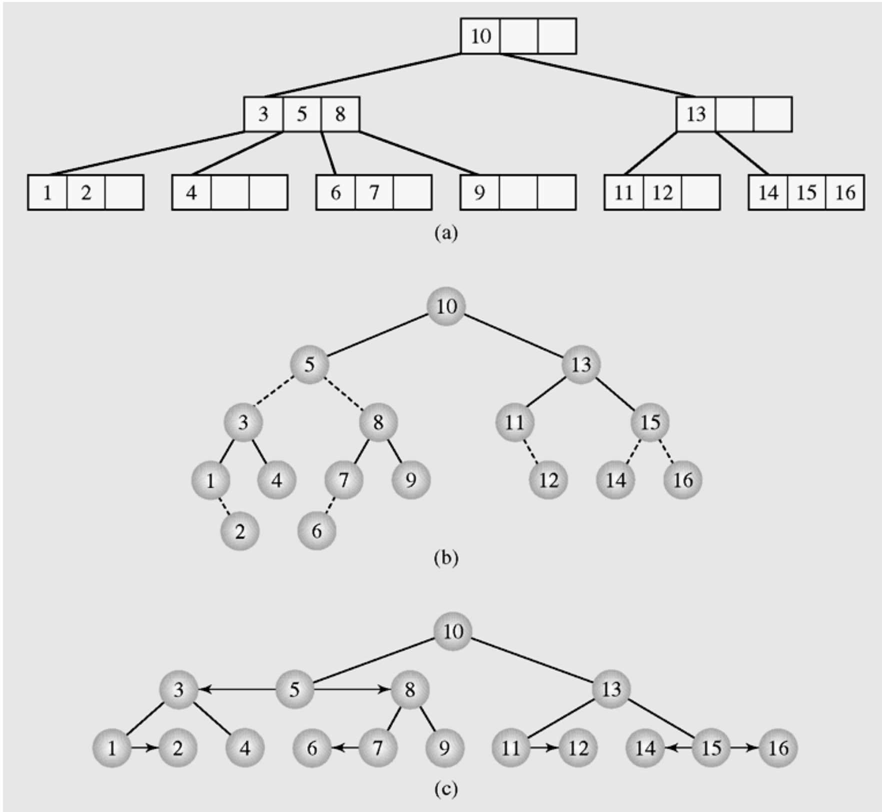
2–4 Trees (continued)



(a) A 3-node represented (b–c) in two possible ways by red-black trees and (d–e) in two possible ways by vh-trees. (f) A 4-node represented (g) by a red-black tree and (h) by a vh-tree.

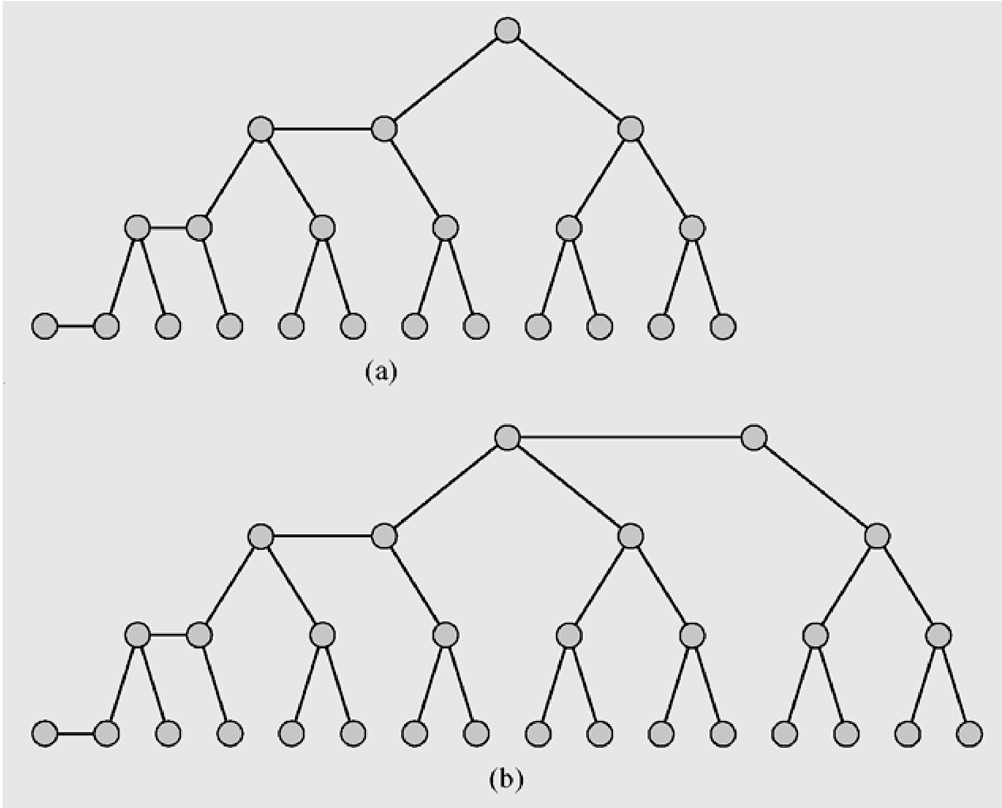
30

2–4 Trees (continued)



(a) A 2–4 tree represented (b) by a red-black tree and (c) by a binary tree with horizontal and vertical pointers

2–4 Trees (continued)

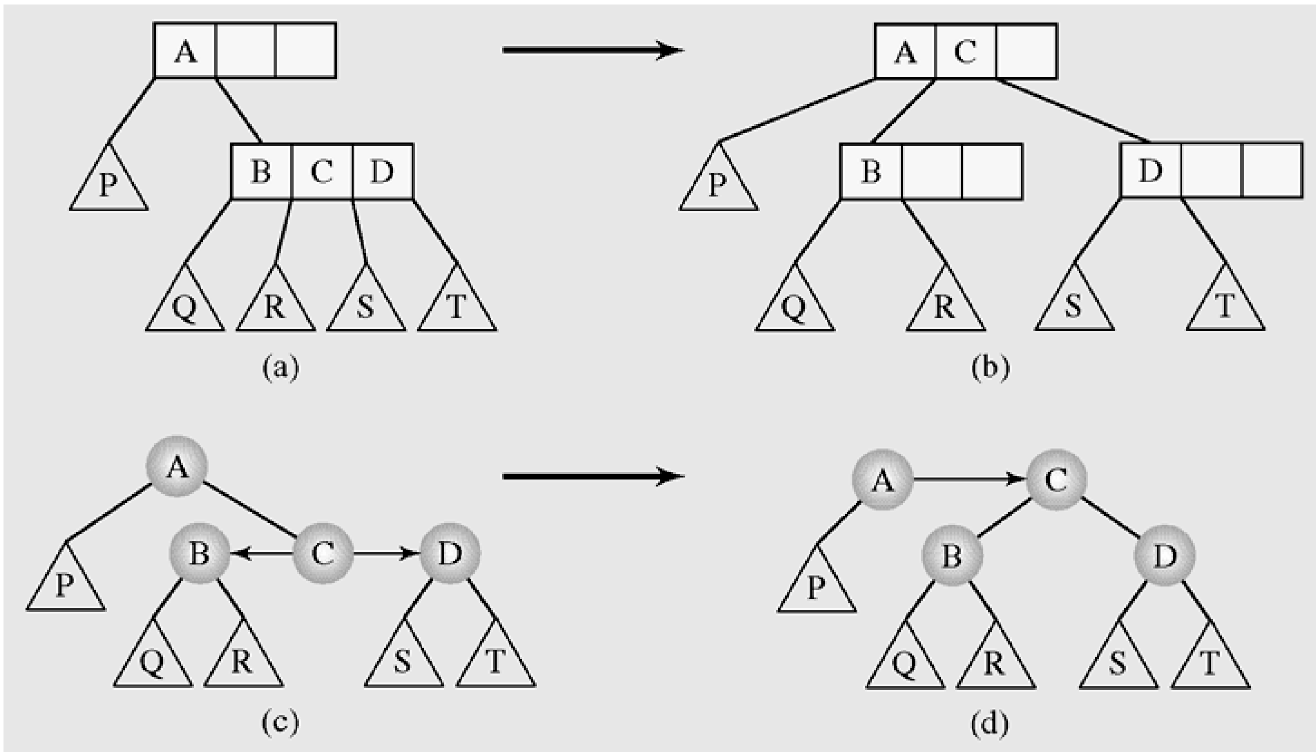


(a) A vh-tree of height 7; (b) a vh-tree of height 8

$$\lg(n+1) \leq h \leq 2\lg(n+2) - 2$$

32

2–4 Trees (continued)

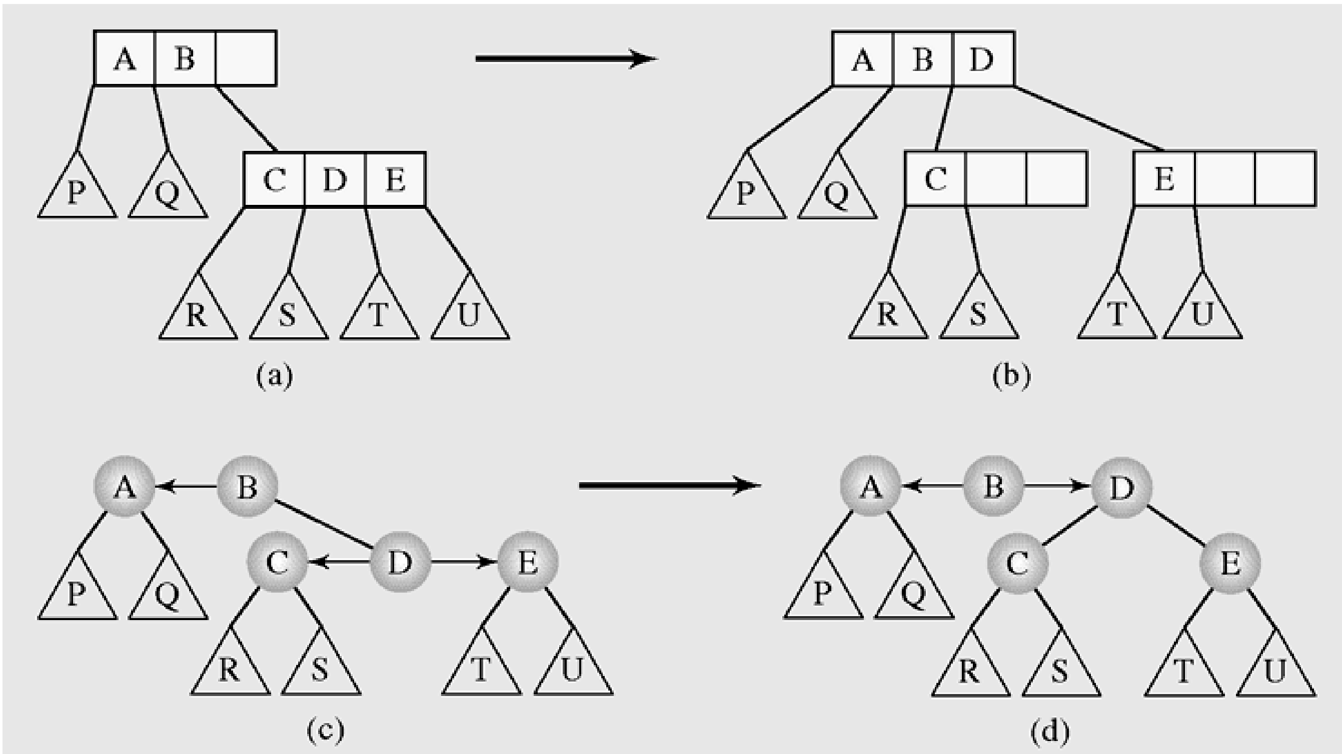


(a–b) Split of a 4-node attached to a node with one key in a 2–4 tree. (c–d) The same split in a vh-tree equivalent to these two nodes.

flagFlipping

33

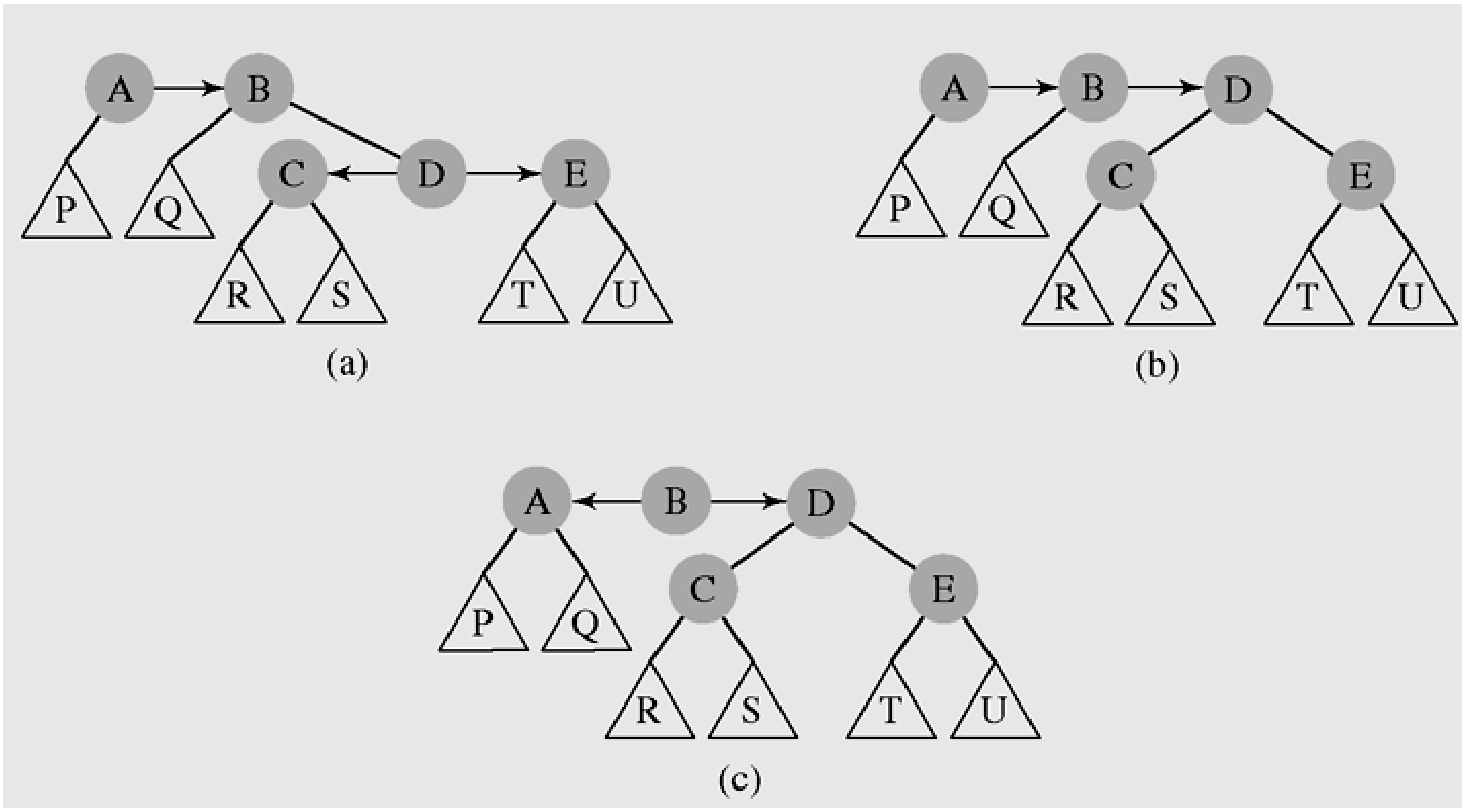
2–4 Trees (continued)



(a–b) Split of a 4-node attached to a 3-node in a 2–4 tree and (c–d) a similar operation performed on one possible vh-tree equivalent to these two nodes.

34

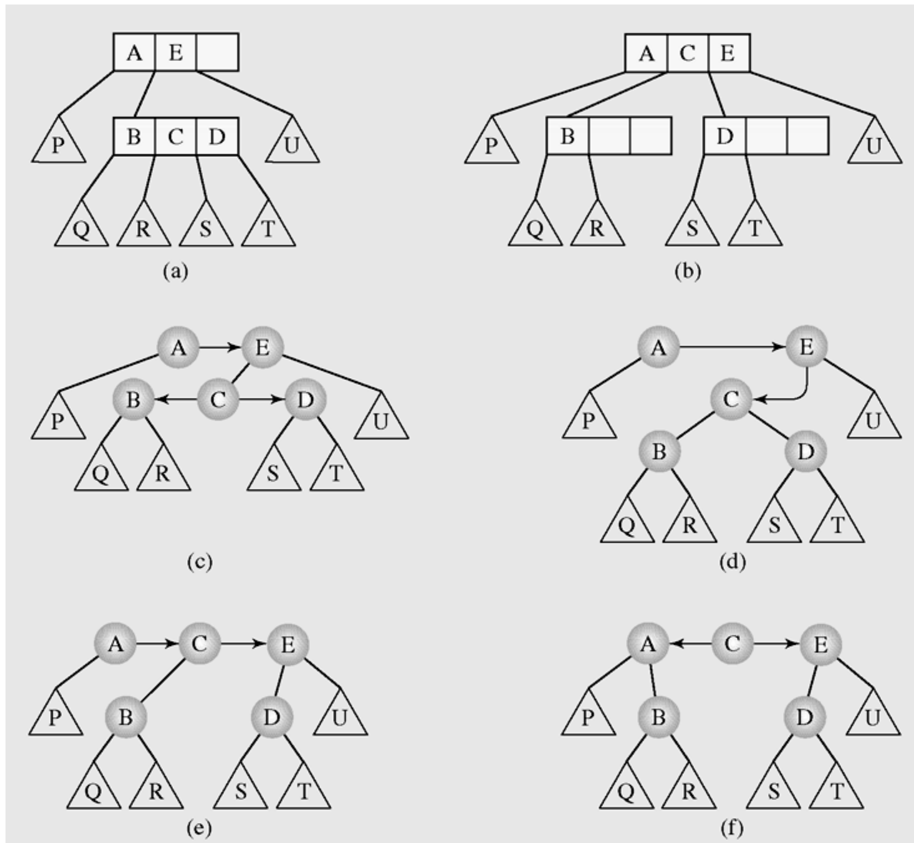
2–4 Trees (continued)



Fixing a vh-tree that has consecutive horizontal links

35

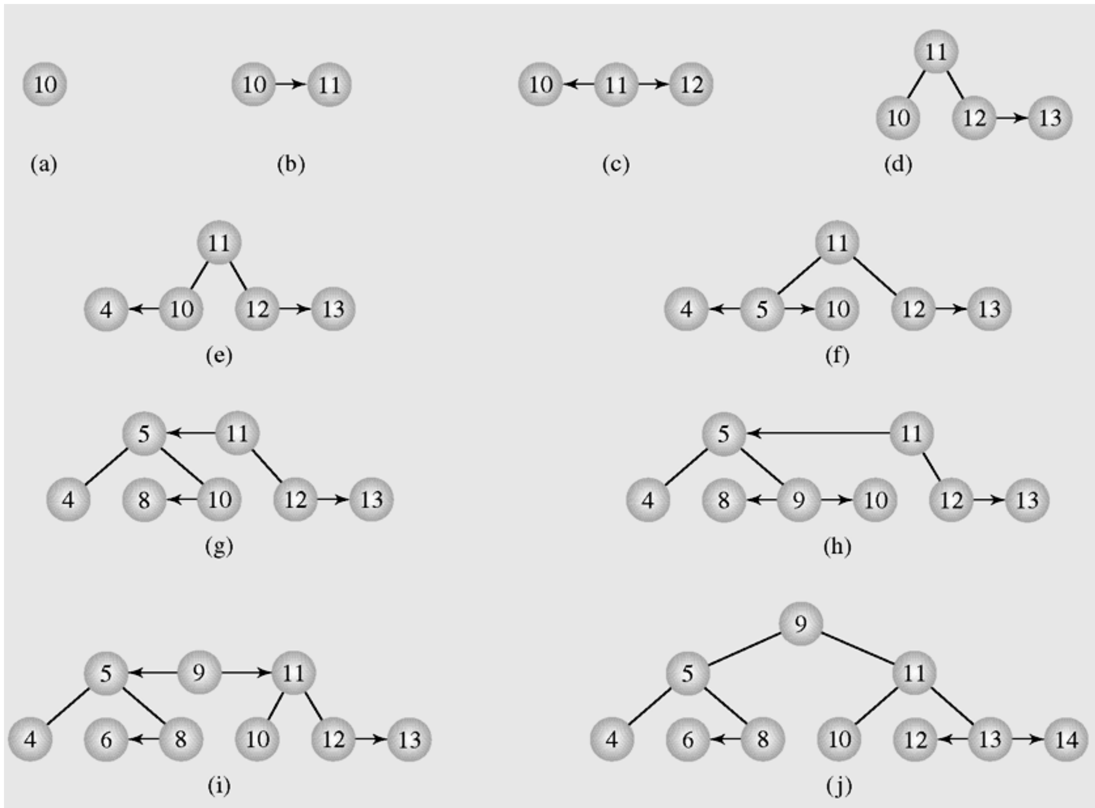
2–4 Trees (continued)



A 4-node attached to a 3-node in a 2–4 tree

36

2–4 Trees (continued)



Building a vh-tree by inserting numbers in this sequence: 10, 11, 12, 13, 4, 5, 8, 9, 6, 14

37

2–4 Trees (continued) deletion

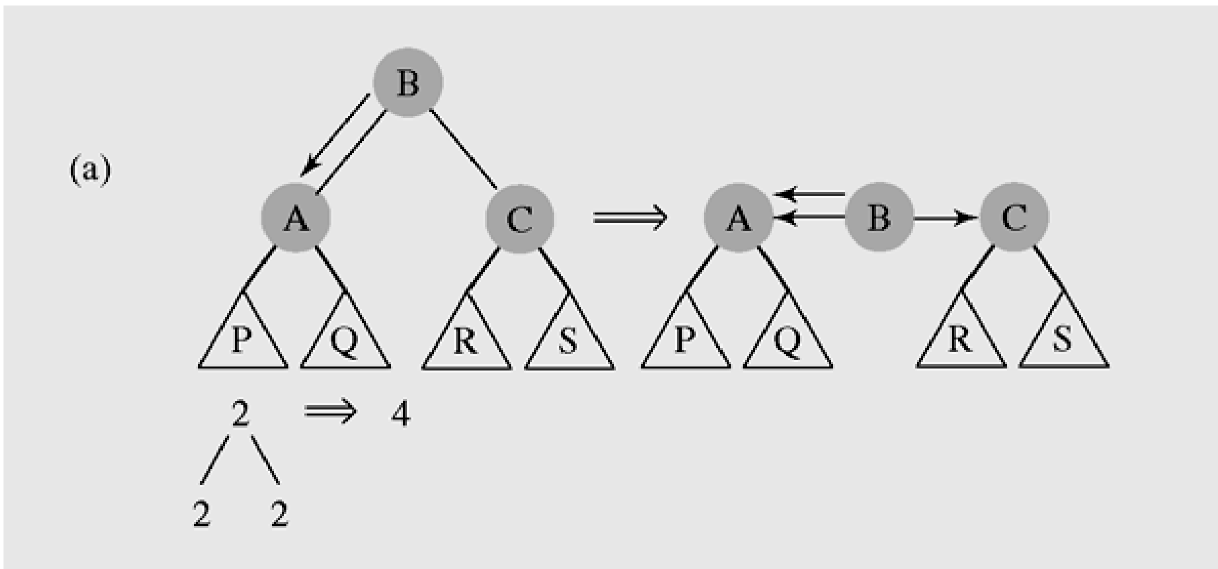
We delete a node by first interchanging it with its successor.

Bad case: successor with no descendants which is connected to its parent by a vertical link

While looking for successor transform vh tree into another vh tree such that successor without descendants is attached to its parent with a horizontal link

38

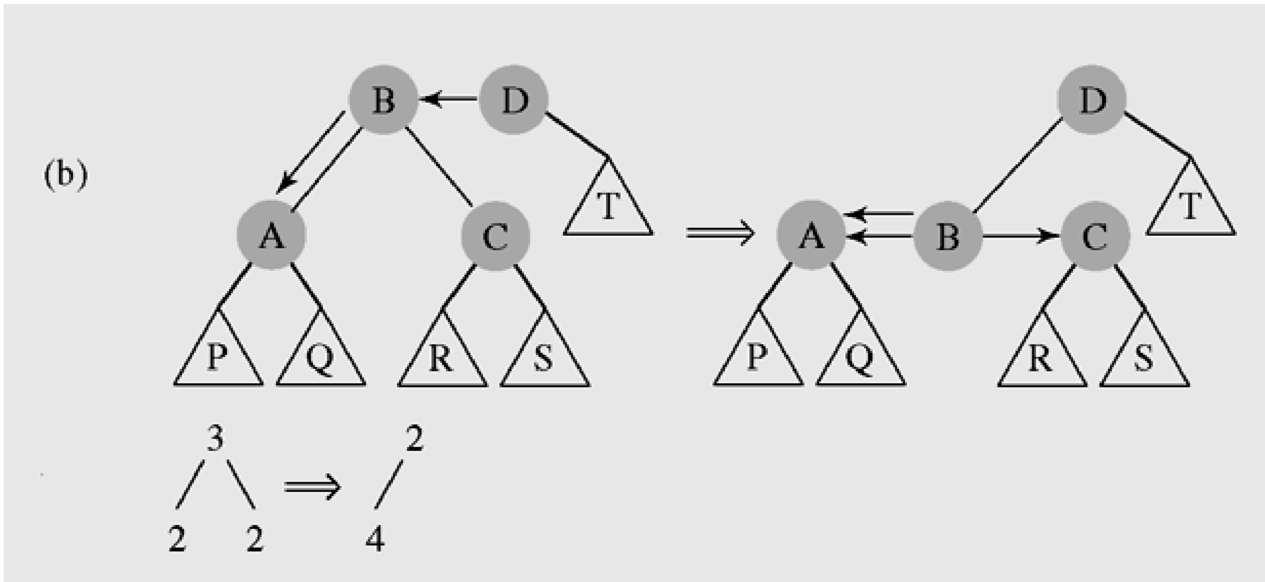
2–4 Trees (continued)



Deleting a node from a vh-tree

39

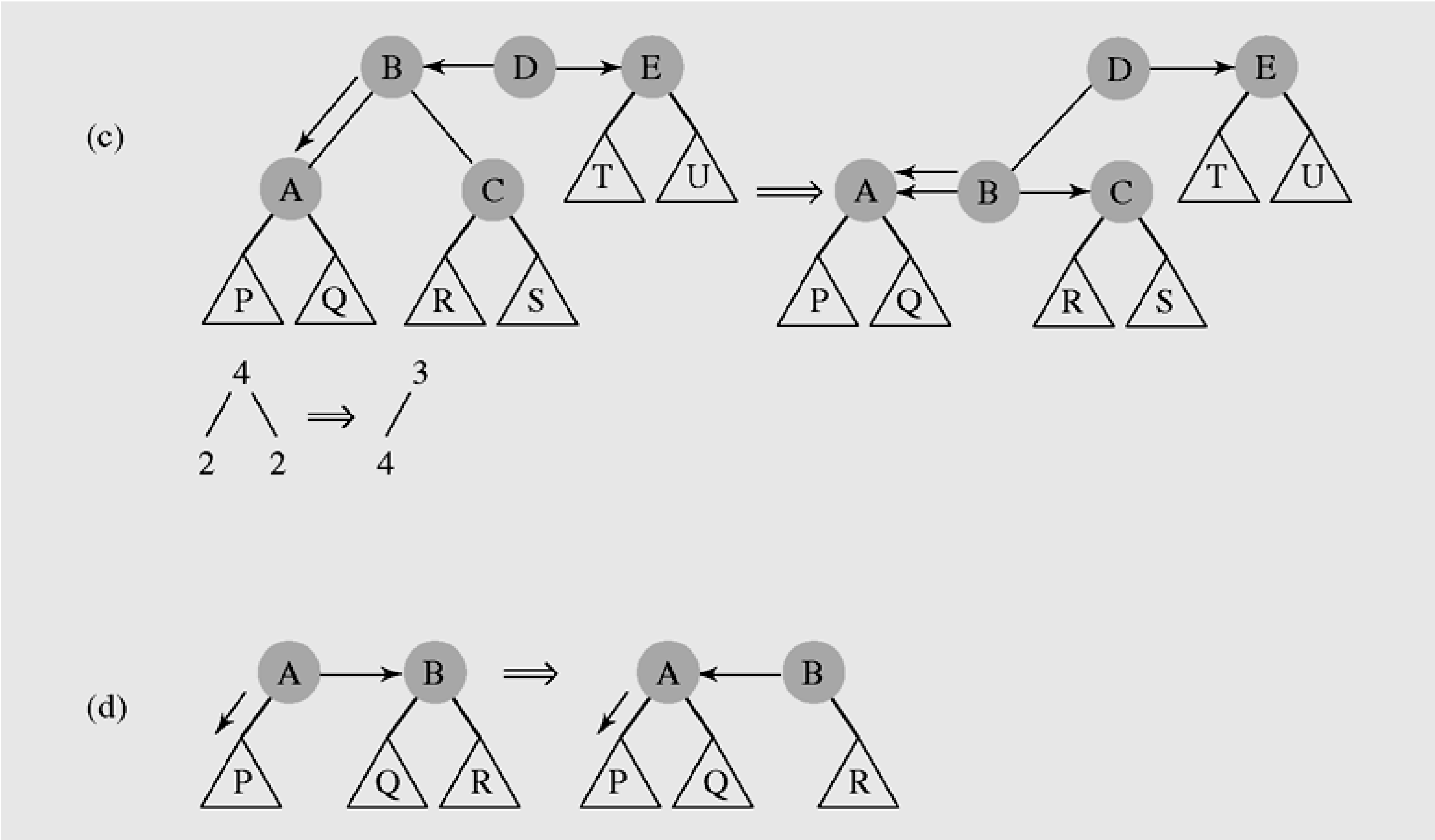
2–4 Trees (continued)



Deleting a node from a vh-tree

40

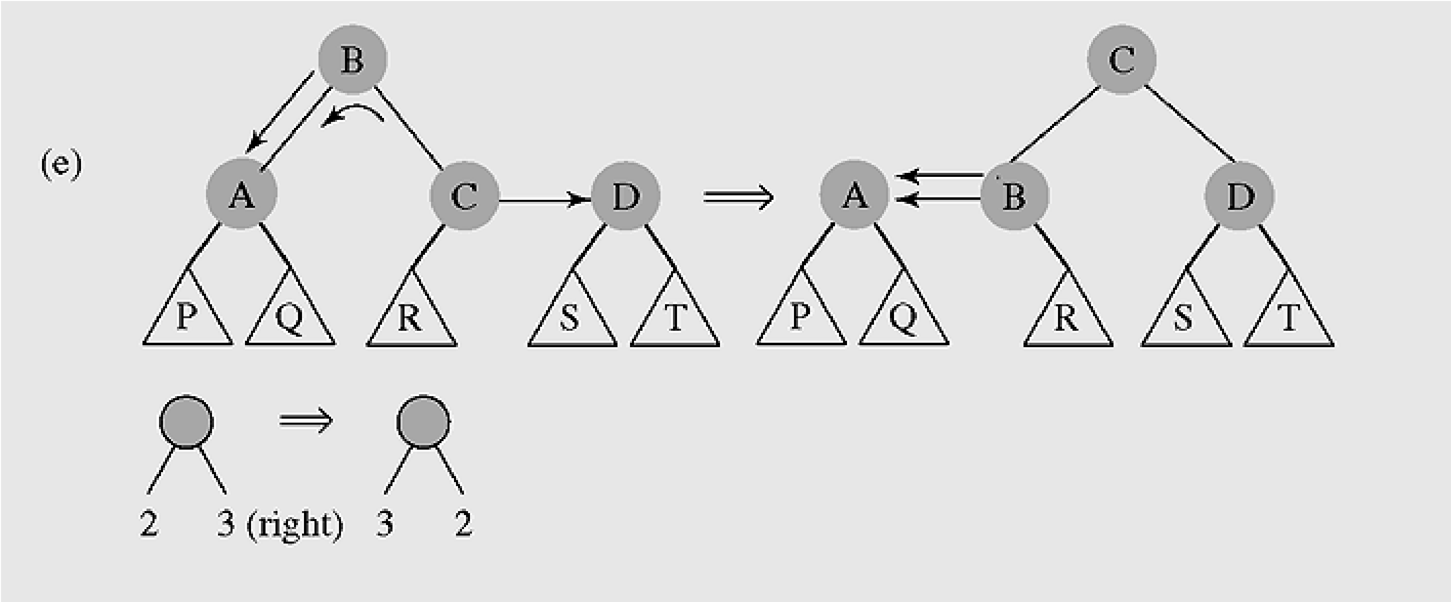
2–4 Trees (continued)



Deleting a node from a vh-tree (continued)

41

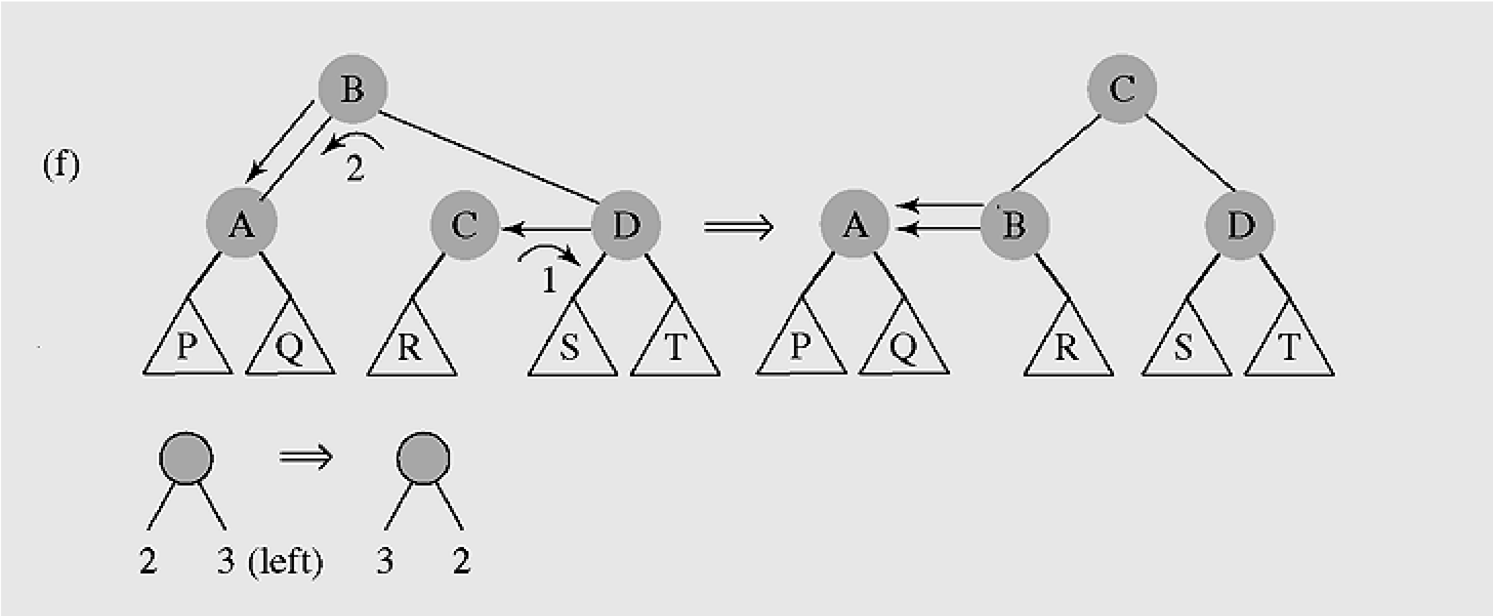
2–4 Trees (continued)



Deleting a node from a vh-tree (continued)

42

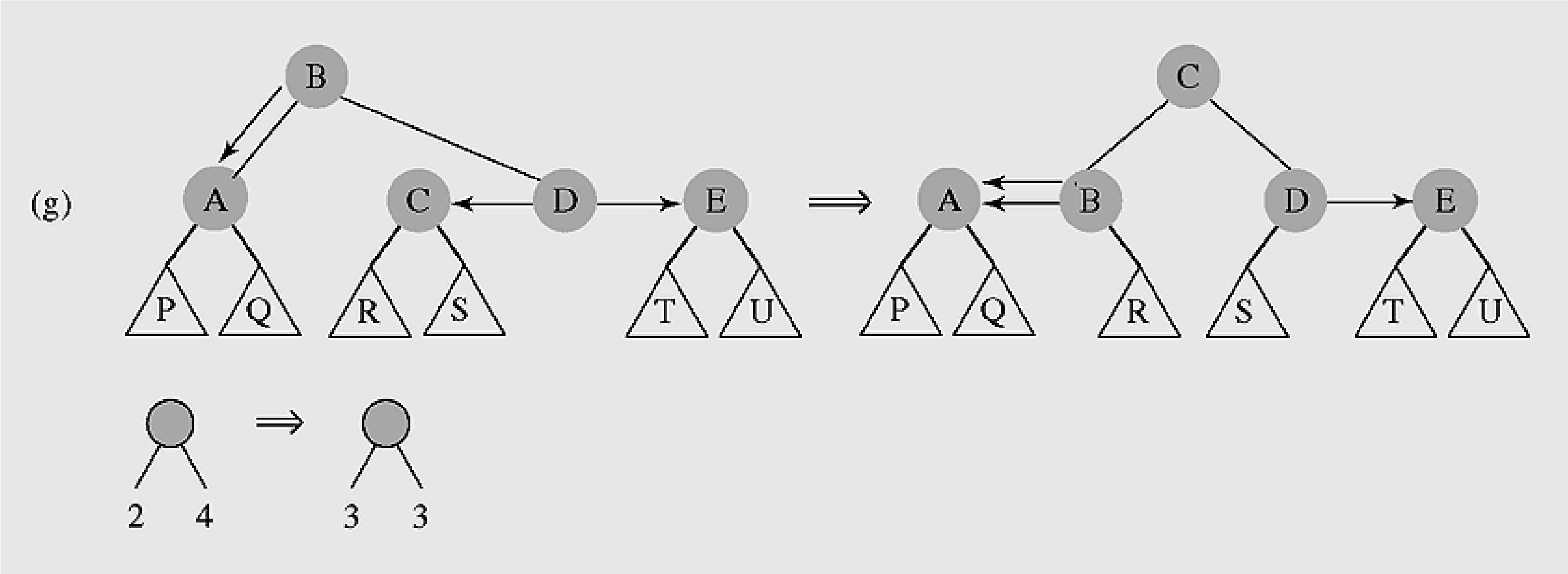
2–4 Trees (continued)



Deleting a node from a vh-tree (continued)

43

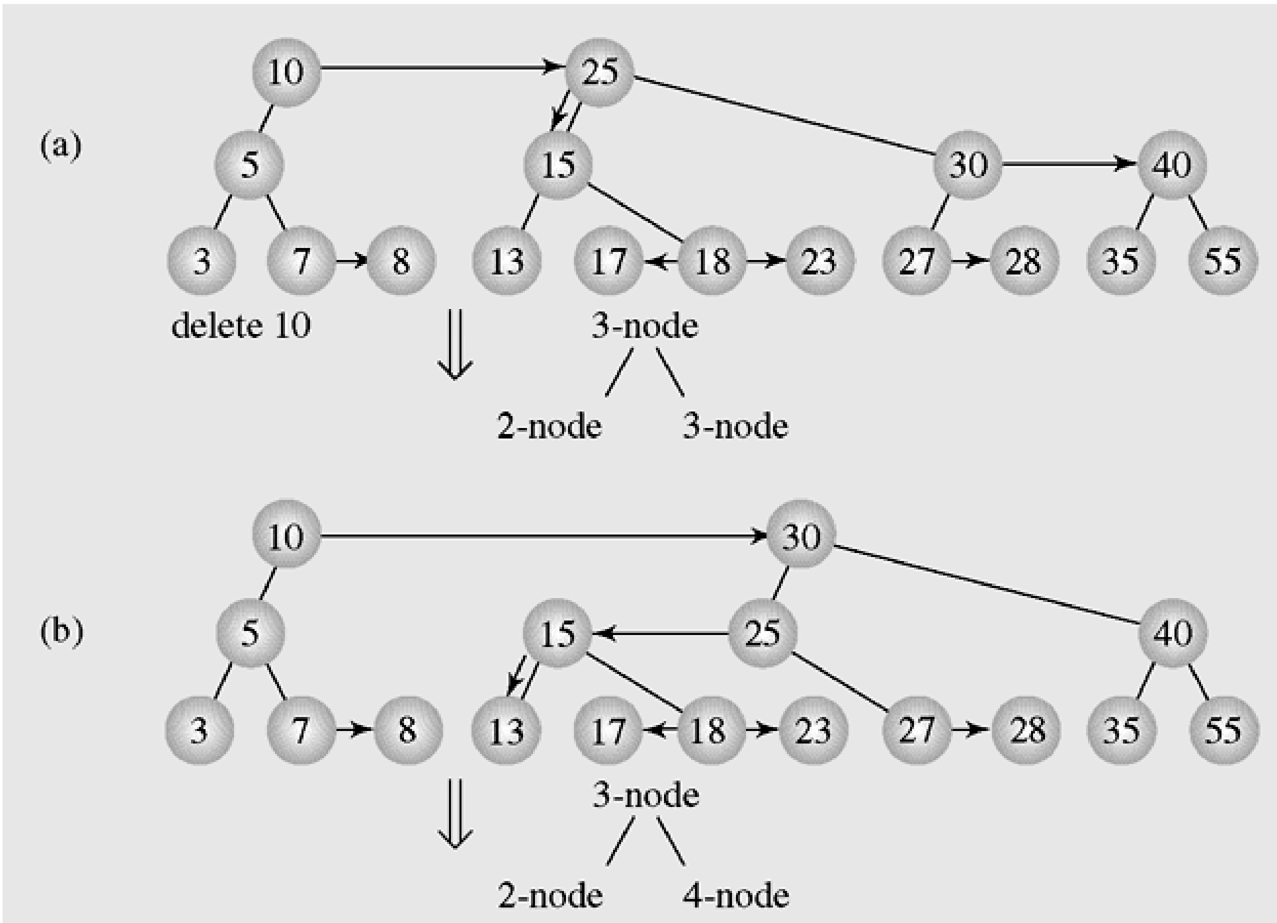
2–4 Trees (continued)



Deleting a node from a vh-tree (continued)

44

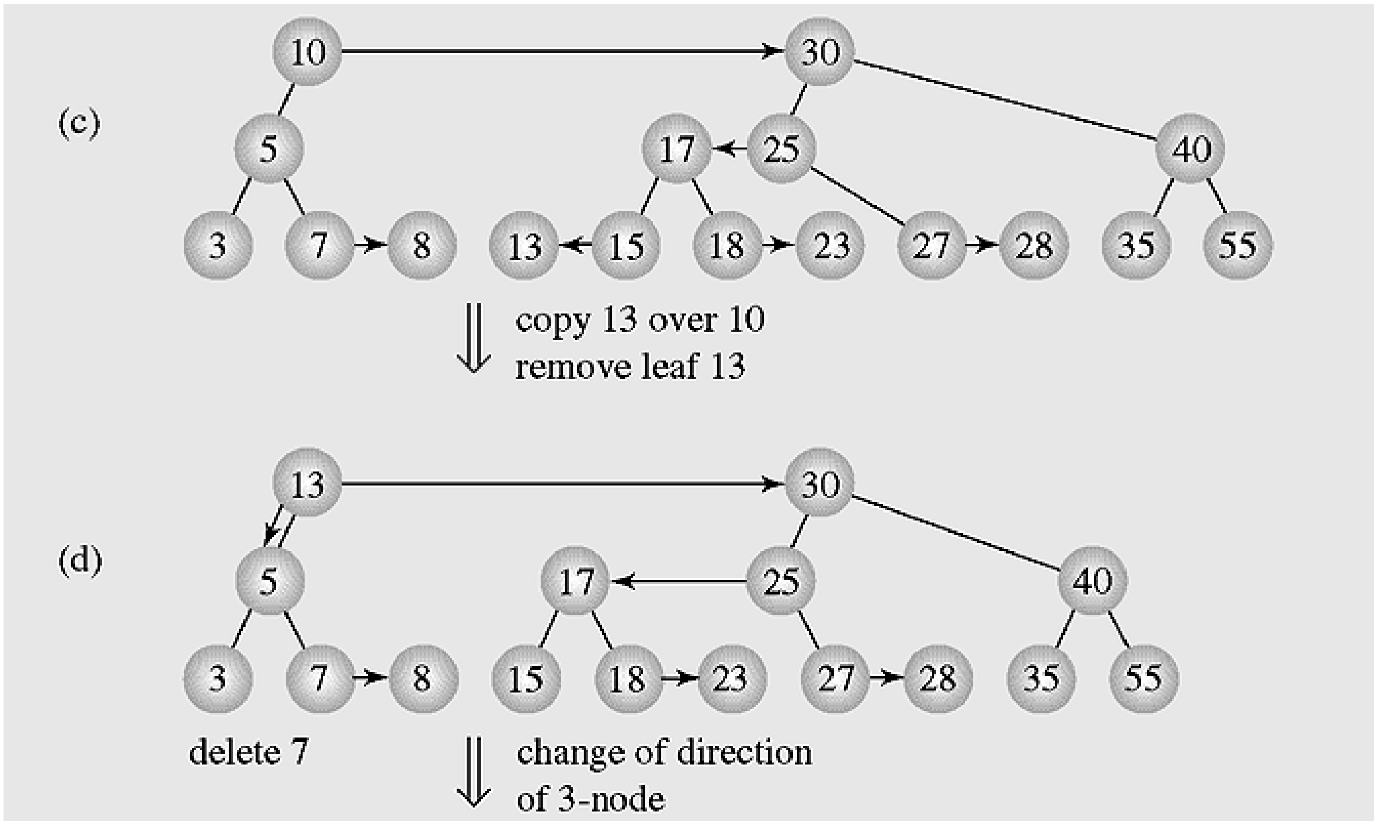
2–4 Trees (continued)



Examples of node deletions from a vh-tree

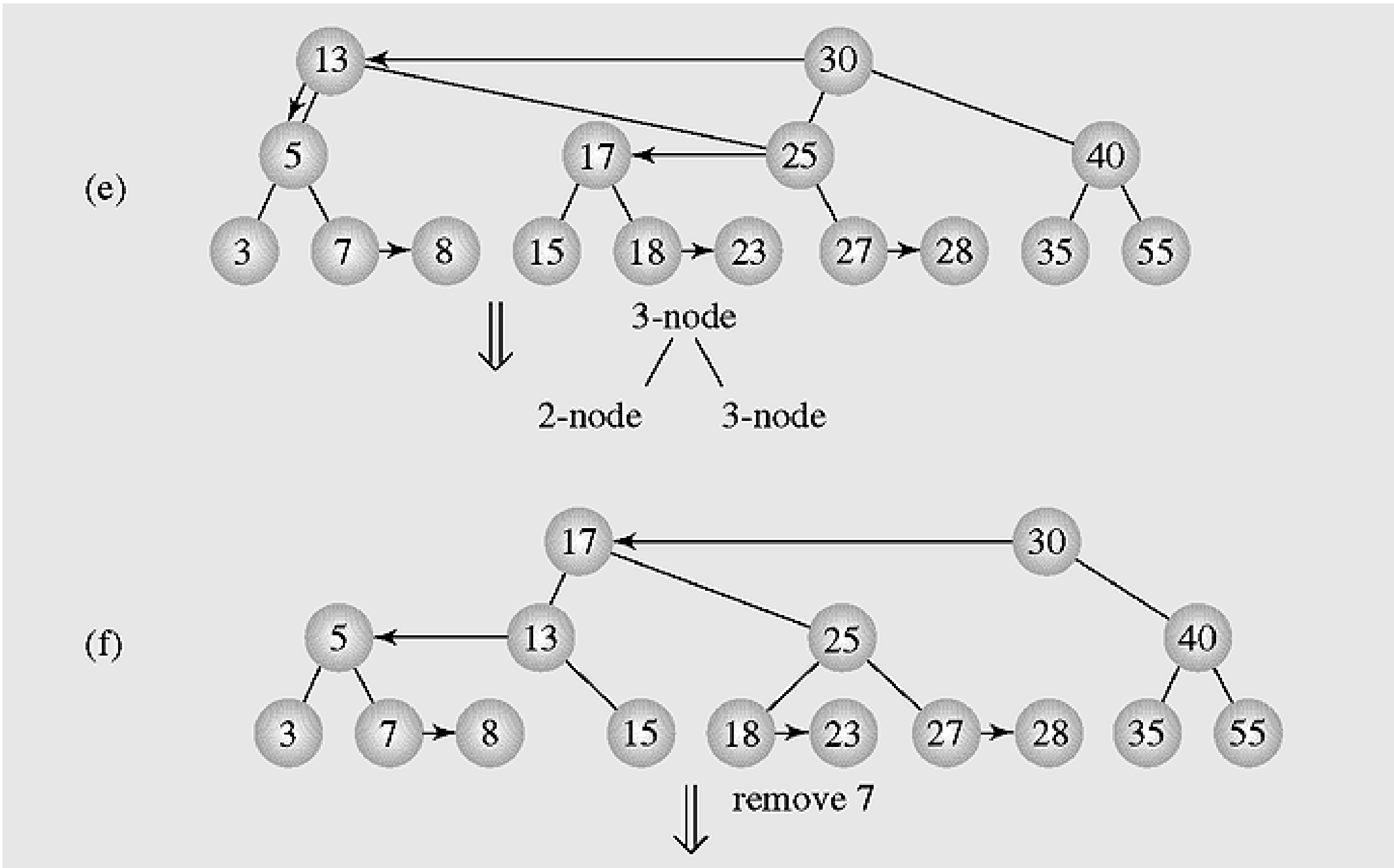
45

2–4 Trees (continued)



Examples of node deletions from a vh-tree (continued)

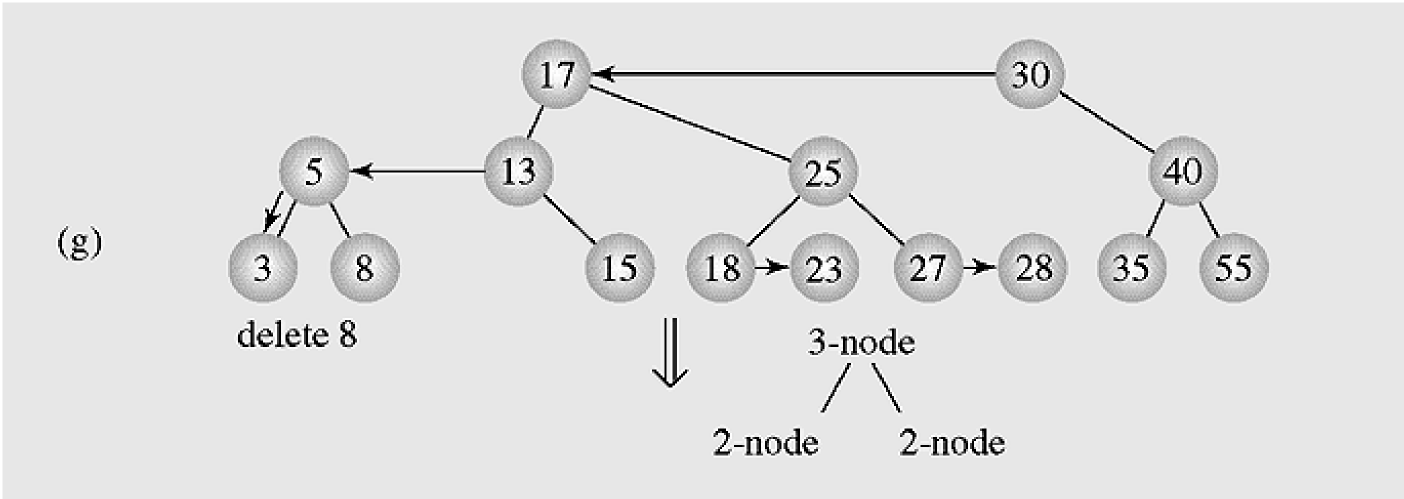
2–4 Trees (continued)



Examples of node deletions from a vh-tree (continued)

47

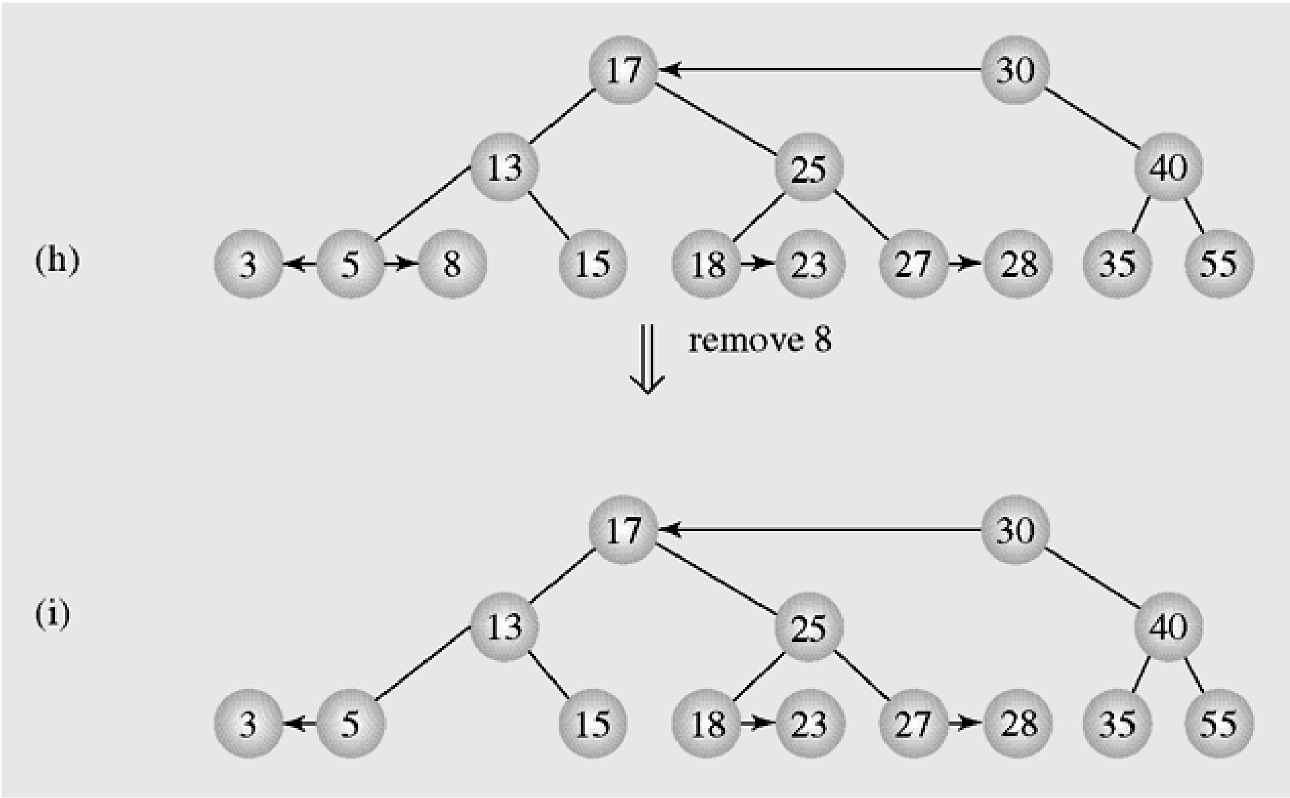
2–4 Trees (continued)



Examples of node deletions from a vh-tree (continued)

48

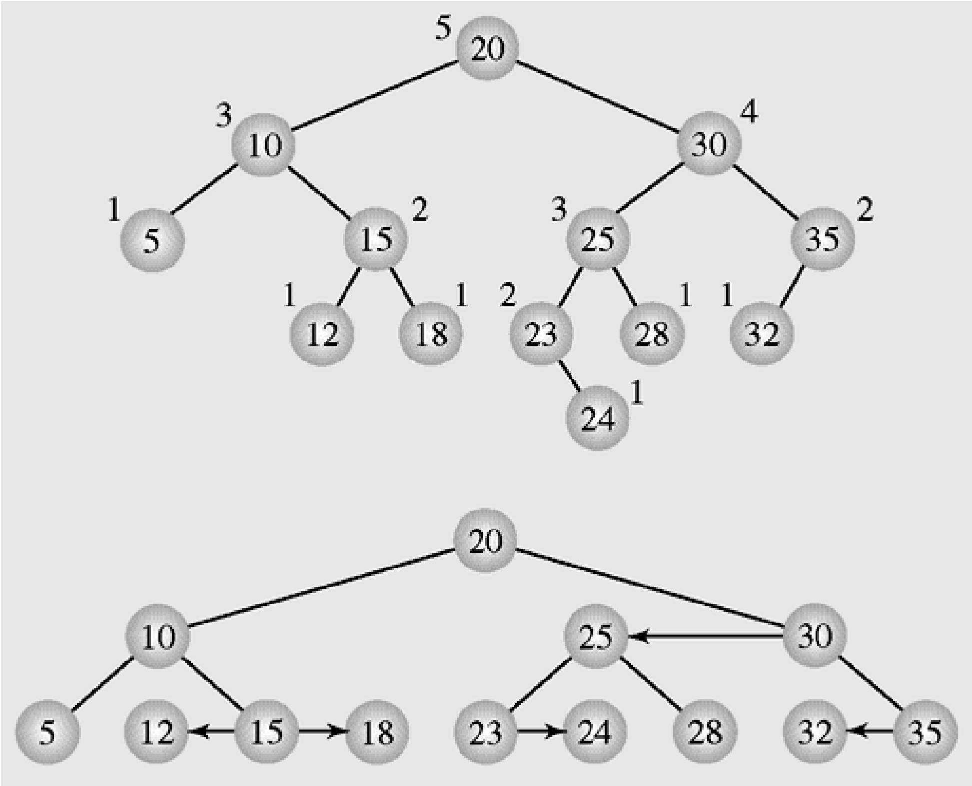
2–4 Trees (continued)



Examples of node deletions from a vh-tree (continued)

49

2–4 Trees (continued)



An example of converting (a) an AVL tree into (b) an equivalent vh-tree

50

Graphs

Chapter 8

Objectives

Discuss the following topics:

- Graphs
- Graph Representation
- Graph Traversals (breadth first, depth first)
- Connectivity
- Bipartiteness
- Topological Sort (aka topological ordering)
- Cycle Detection
- Shortest Paths

52

Graphs

- A **graph** is a collection of vertices (or nodes) and the connections between them
- A **simple graph** $G = (V, E)$ consists of a nonempty set V of **vertices** and a possibly empty set E of **edges**, each edge being a set of two vertices from V
- A **directed graph**, or a **digraph**, $G = (V, E)$ consists of a nonempty set V of vertices and a set E of edges (also called **arcs**), where each edge is a pair of vertices from V

53

Graphs (continued)

- A **multigraph** is a graph in which two vertices can be joined by multiple edges
- A **pseudograph** is a multigraph with the condition $v_i \neq v_j$ removed, which allows for loops to occur
- A graph is called a **weighted graph** if each edge has an assigned number

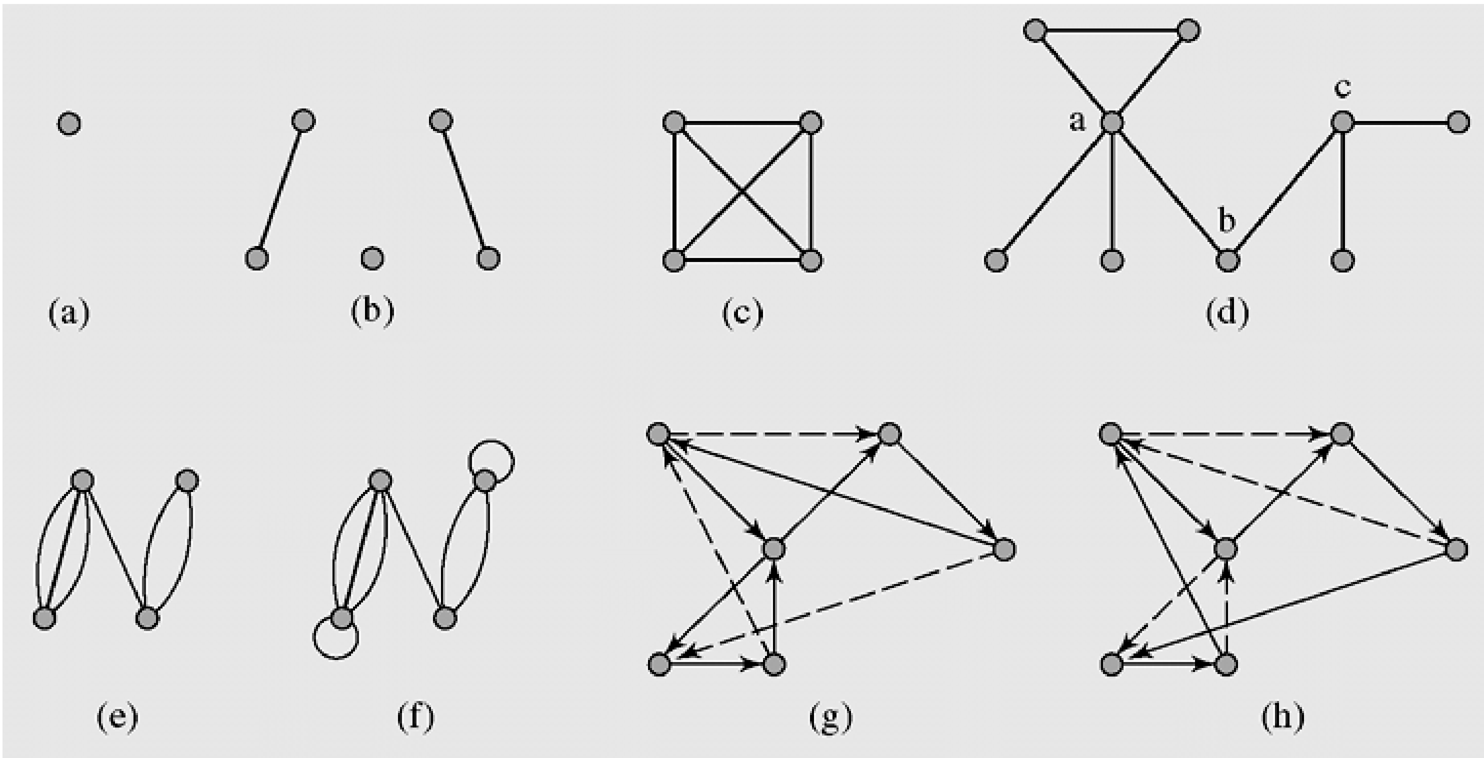
54

Graphs (continued)

- A **path** from v_1 to v_n is a sequence of edges *edge*(v_1,v_2), *edge*(v_2,v_3), …, *edge*(v_{n-1},v_n)
- If v_1=v_n, and no edge is repeated, then the path is called a **circuit**
- If all vertices in a circuit are different, then it is called a **cycle**.

55

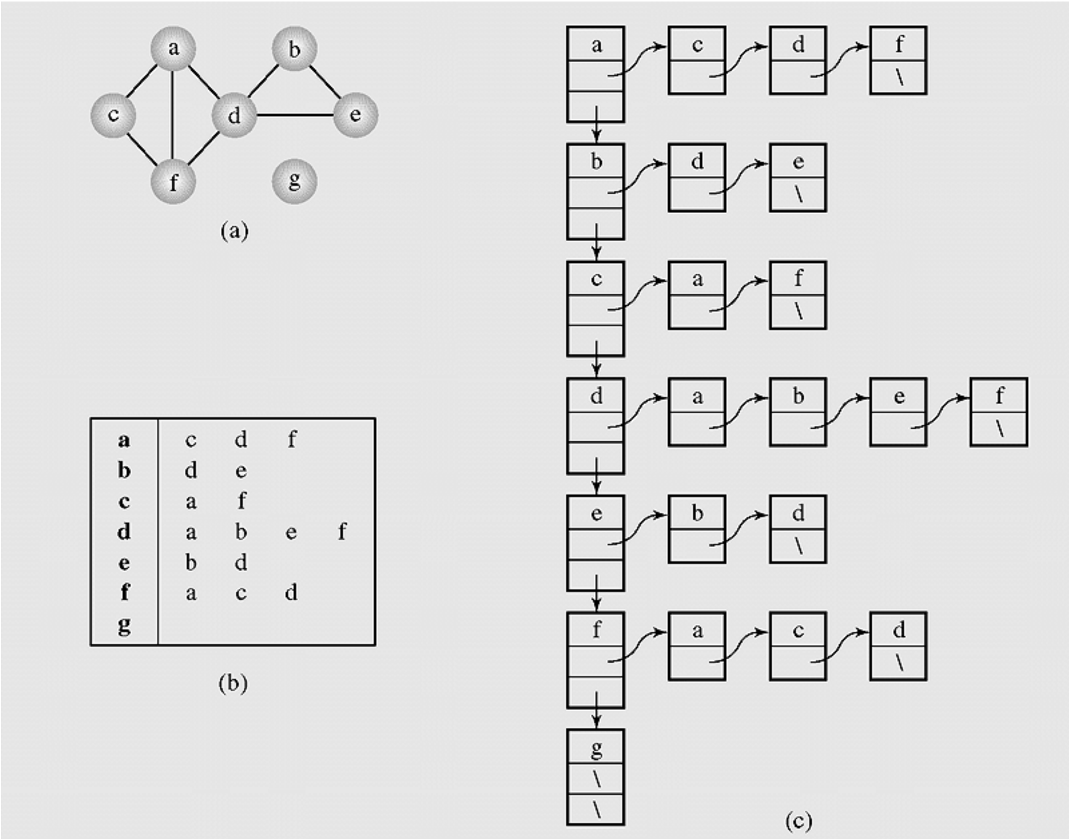
Graphs (continued)



Examples of graphs: (a–d) simple graphs; (c) a complete graph K_4; (e) a multigraph; (f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph

56

Graph Representation



Graph representations (a) A graph represented as (b–c) an adjacency list

$m \leq n(n-1)/2! \leq n^2$ (m=#edges; n=#nodes)

G connected: $n-1 \leq m \leq n(n-1)/2! \leq n^2$

G sparse: $m << n(n-1)/2!$

Adjacency matrix requires $O(n^2)$ space; process neighbors of v needs |V| steps.

Adjacency list: O(m+n) space; steps; process neighbors of v needs deg(v) steps

57

Graph Representation (continued)

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

(d)

	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

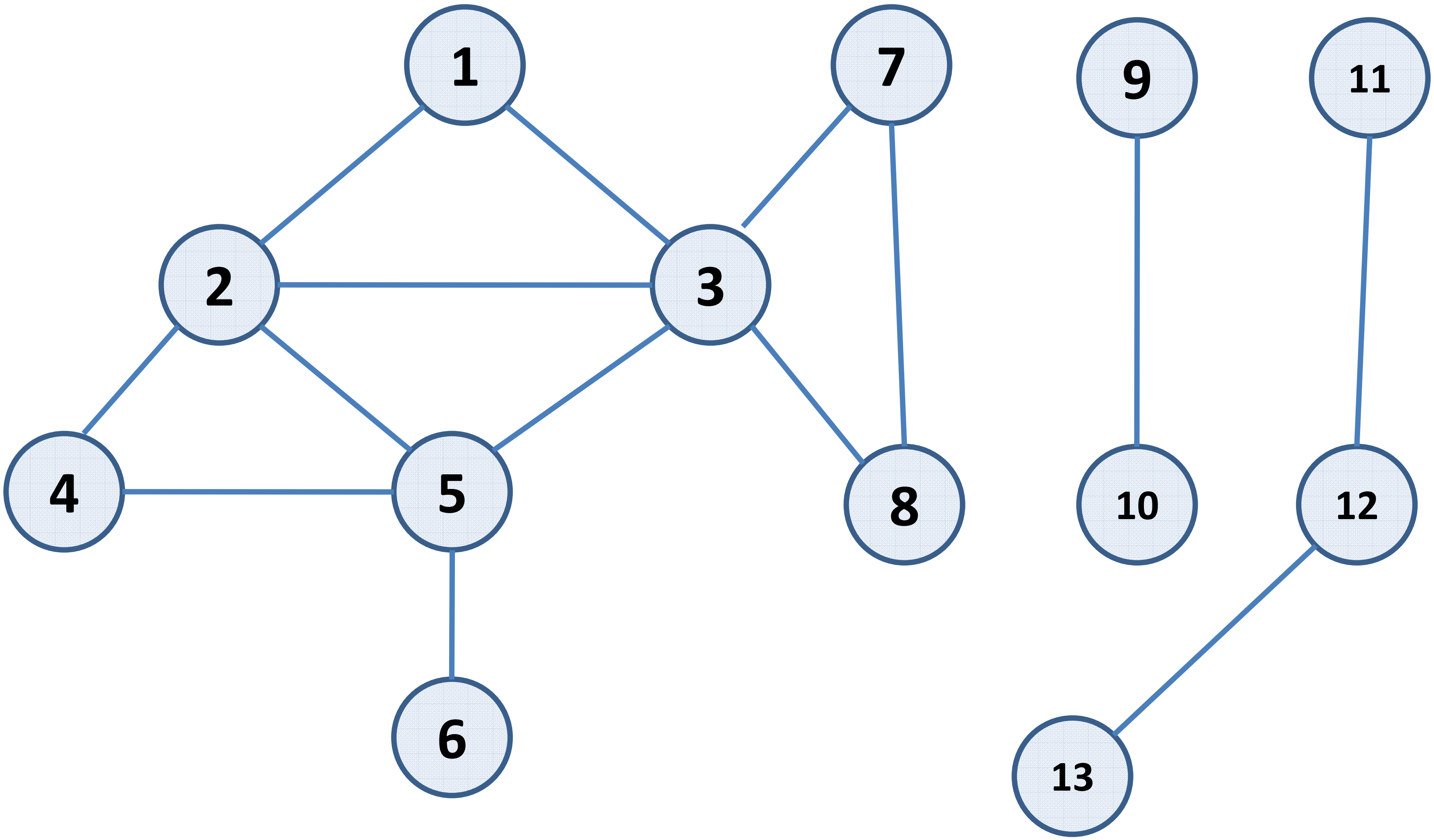
(e)

Graph representations (d) an adjacency matrix, and (e) an incidence matrix (continued)

58

Graph Traversal: breadth first search and depth first search

- Let G = (V, E) be a graph and let s and t be two particular nodes. Is there a path from s to t in G?
- Two high level solutions: breadth first search and depth first search
- *Breadth-first search*



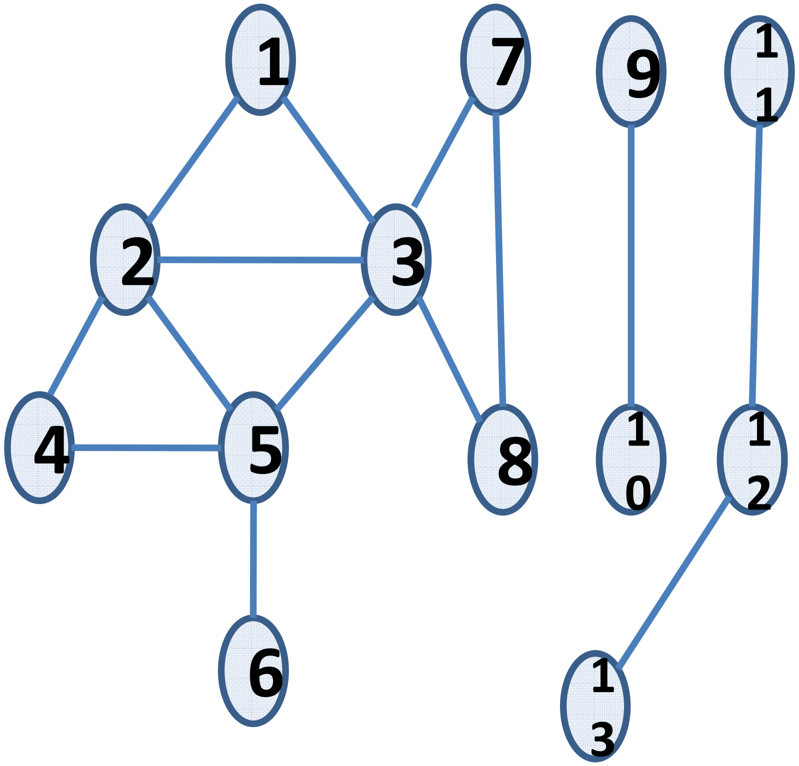
Layers, flooding; more precisely:

Graph traversal: bsf

- Define the layers L_1, L_2, L_3, … more precisely
- Layer L_1 consists of all nodes that are neighbors of node s. (Denote the set $\{s\}$ by L_0)
- Assume we have defined $L_1, \ldots, L_j$, then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.
- Distance between two nodes: minimum number of edges on a path joining them

Graph traversal: bsf

- *For each $j \geq 1$, layer $\boldsymbol{L_j}$ produced by BFS consists of all nodes at distance exactly **j** from **s**.*
- *There is a path from **s** to **t** if and only if **t** appears in some layer.*
- BFS → a tree T rooted at s on the set of nodes reachable from s. Breadth first search tree.



→Bfs tree starting from Node 1.

Graph traversal: bsf

- Let T be a breadth-first search tree, let x and y be nodes in T belonging to L_i and L_j, and let (x,y) be an edge of G. Then i and j differ by at most 1.

Use array discovered[], and for each layer L_i we have a list L[i], i=0, 1, 2, ….

```
BFS(s):
   discovered[s] ← true;
   discovered[v] ← false    // for all other nodes of G
   initialize L[0] to consist of the single element s
   set current BFS tree T to ø.
   While L[i] is not empty
          initialize empty list L[i+1]
             for each node  u ε L[i]
                 consider each edge (u,v) incident to u
                 if (discovered[v] == false ) {
                    discovered[v] ← true;
                    add edge (u,v) to T
                    add v to the list L[i+1]
                 }
               endfor
Endwhile
```

Can also use a queue; will have one list L then.