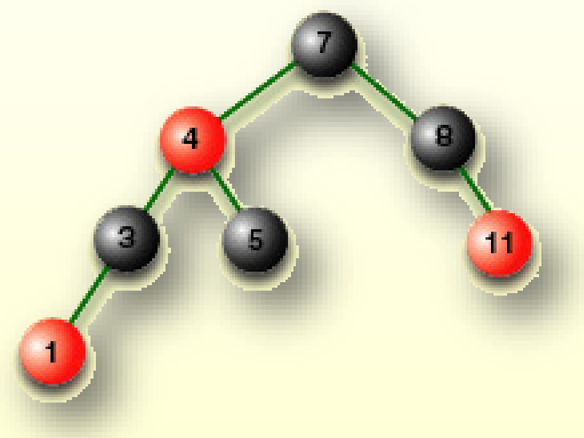
CS161: Design and Analysis of Algorithms



Lecture 10
Leonidas Guibas

Outline

- Review of last lecture: Binary Search Trees (BSTs)
- Today: Balanced BSTs
  - Red-Black Trees
    - Properties/Analysis
    - Insertion/Deletion
  - 2-3, and 2-3-4 Trees

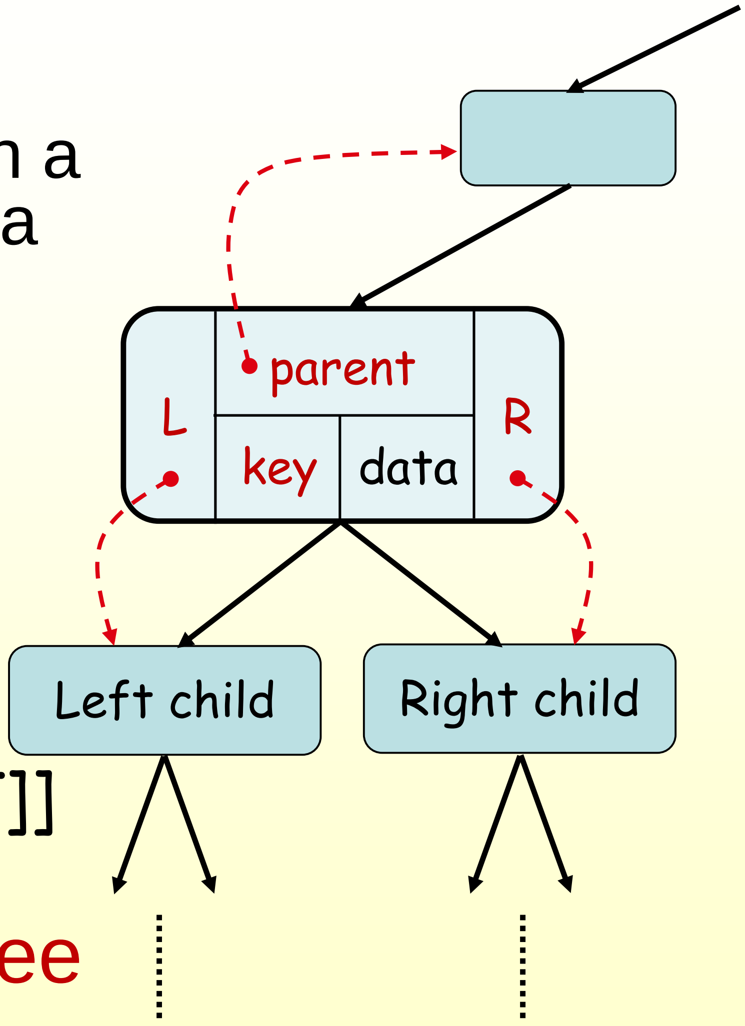
Balanced Trees

Slides modified from

- www.cse.unr.edu/~bebis/CS477/
- http://www.cs.unc.edu/~lin/COMP122-F99/
- www.dsm.fordham.edu/~agw/.../Chapter19-BalancedSearchTrees.ppt

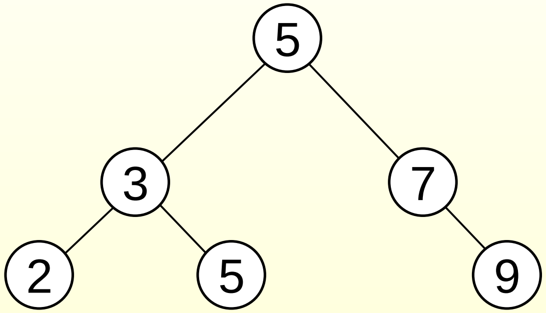
Binary Search Trees

- Tree representation:
  - A linked data structure in which a set of nodes is connected into a tree
- Node representation:
  - Key field
  - Satellite data
  - Left: pointer to left child
  - Right: pointer to right child
  - p: pointer to parent (p [root [T]] = NIL)
- Satisfies the binary-search-tree property



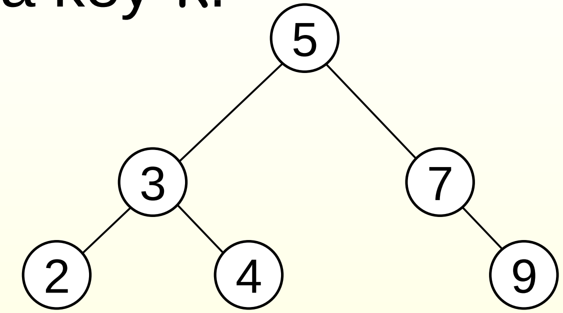
Binary Search Tree Property

- Binary search tree order property:
  - If y is in left subtree of x,
    then key [y] ≤ key [x]
  - If y is in right subtree of x,
    then key [y] ≥ key [x]



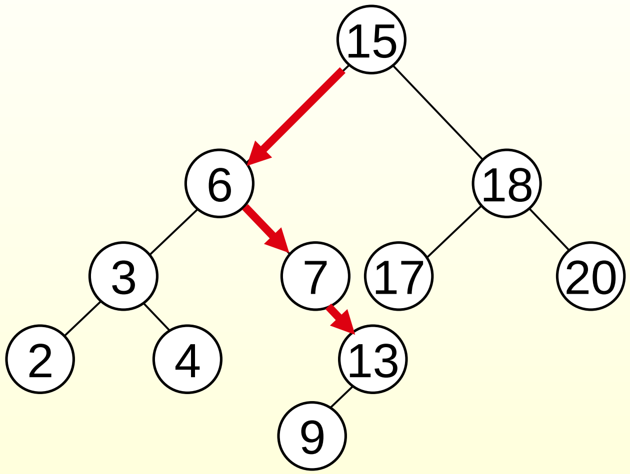
Searching for a Key

- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k (if one exists)
  - Otherwise return NIL



- Idea
  - Starting at the root: trace down a path by comparing k with the key of the current node:
    - If k = key[x], we have found the key
    - If k < key[x], search in the left subtree of x
    - If k > key[x], search in the right subtree of x

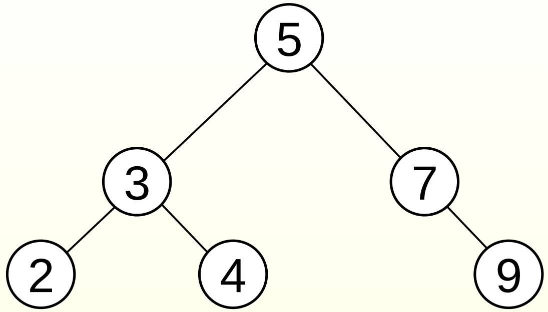
Example: TREE-SEARCH



- Search for key 13:
  - 15 $\rightarrow$ 6 $\rightarrow$ 7 $\rightarrow$ 13

Searching for a Key

Alg: TREE-SEARCH(x, k)



1. **if** x = NIL or k = key [x]
2. **then return** x
3. **if** k < key [x]
4. **then return** TREE-SEARCH(left [x], k)
5. **else return** TREE-SEARCH(right [x], k)

Running Time: *O* (h),
h – the height of the tree

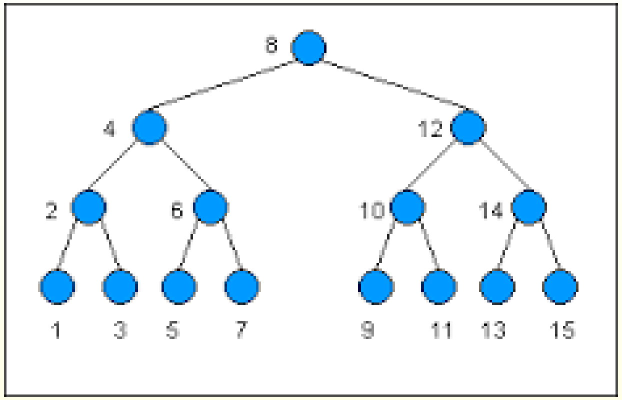
Binary Search Trees

- Support *many* dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- In particular, a Binary Search Tree (BST) can implement both the Dynamic Dictionary and the Priority Queue abstract data types

Binary Search Trees - Summary

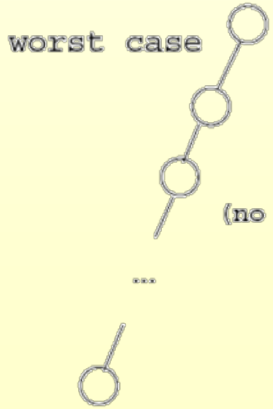
- Operations on binary search trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$
- These operations are fast if the height of the tree is *small* – otherwise their performance is similar to that of a linked list

Best/Worst Case



balanced case Height $O(\log n)$

- If the tree is very **unbalanced**, then running time will be $O(n)$.



h=n-1

(no better than a linked list)

Tree Height

- The height h of a binary search tree on n items can be as high as n-1

- However, if it is built by inserting the elements in random order, then the expected height is $O(\lg n)$.

- With red-black trees, we aim to guarantee that the height is always $O(\lg n)$.

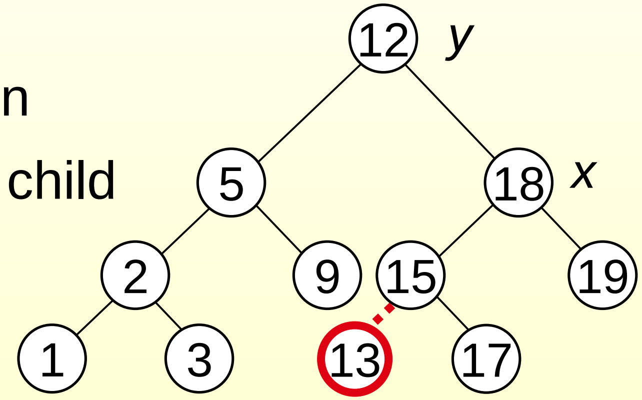
Insertion

- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If key [x] < v move to the right child of x,

    else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < key [y] insert the new node as y's left child

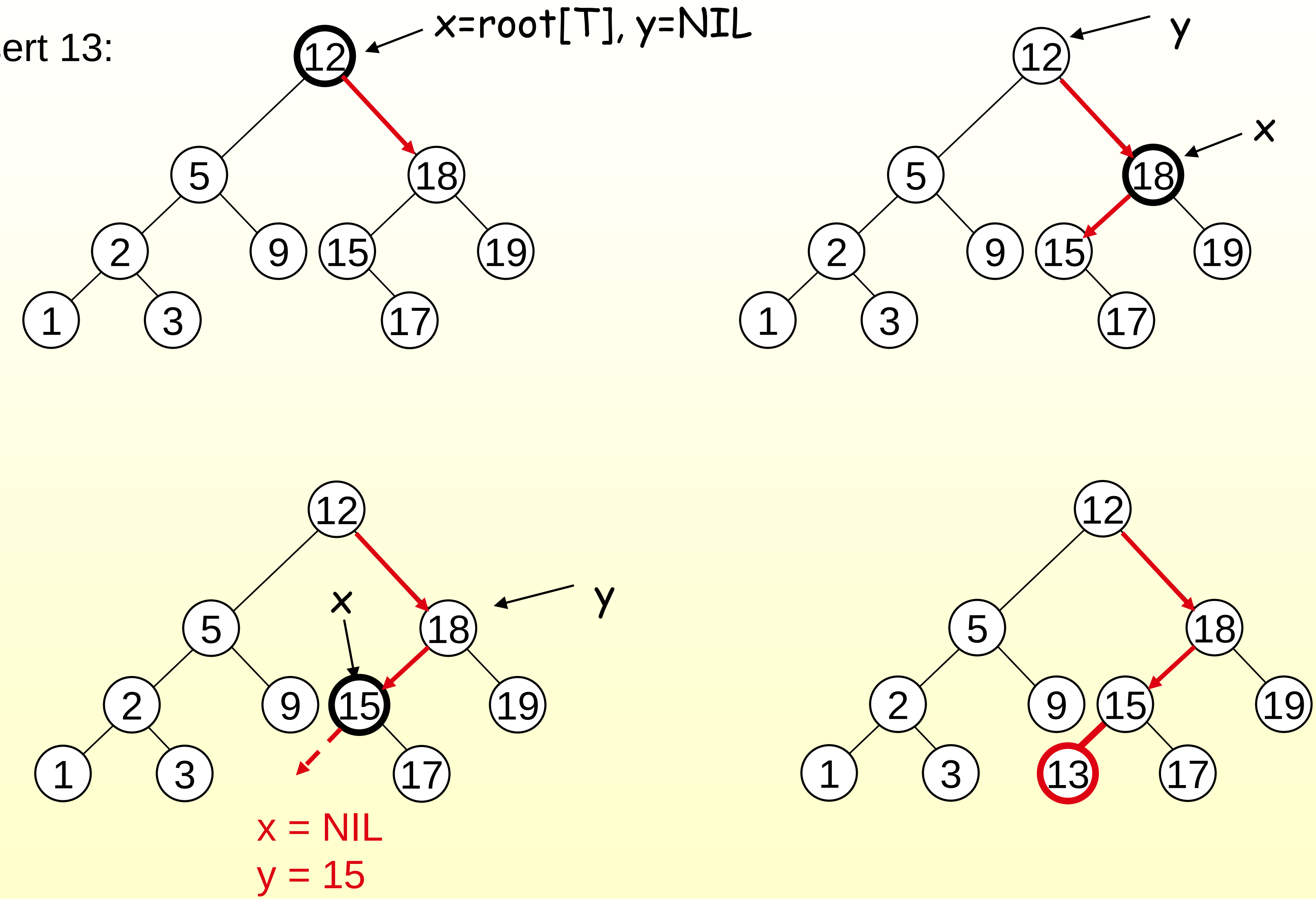
    else insert it as y's right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

Insert value 13



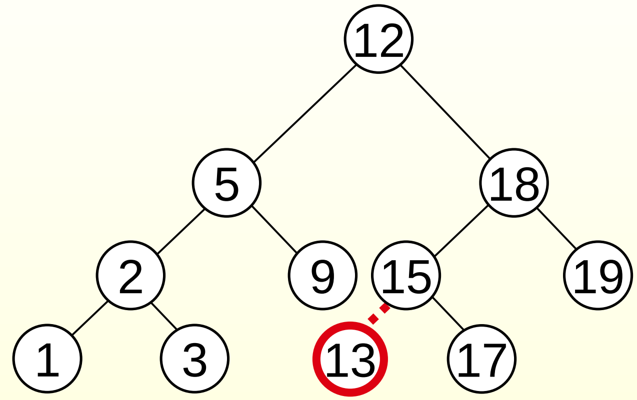
Example: TREE-INSERT

Insert 13:



Alg: TREE-INSERT(T, z)

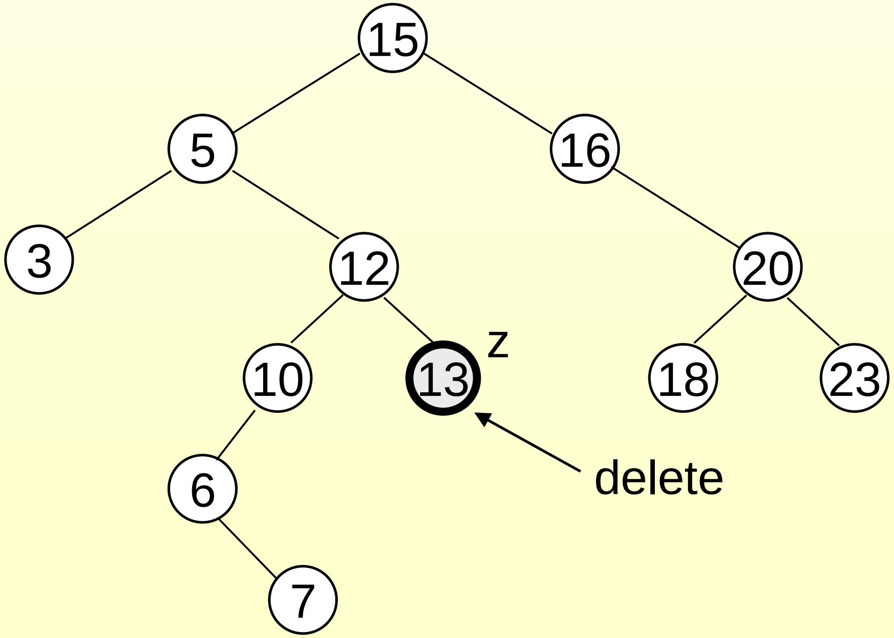
```
y ← NIL
x ← root [T]
while x ≠ NIL
    do y ← x
        if key [z] < key [x]
            then x ← left [x]
            else x ← right [x]
p[z] ← y
if y = NIL
   then root [T] ← z                    ▷ Tree T was empty
   else if key [z] < key [y]
            then left [y] ← z
            else right [y] ← z
```

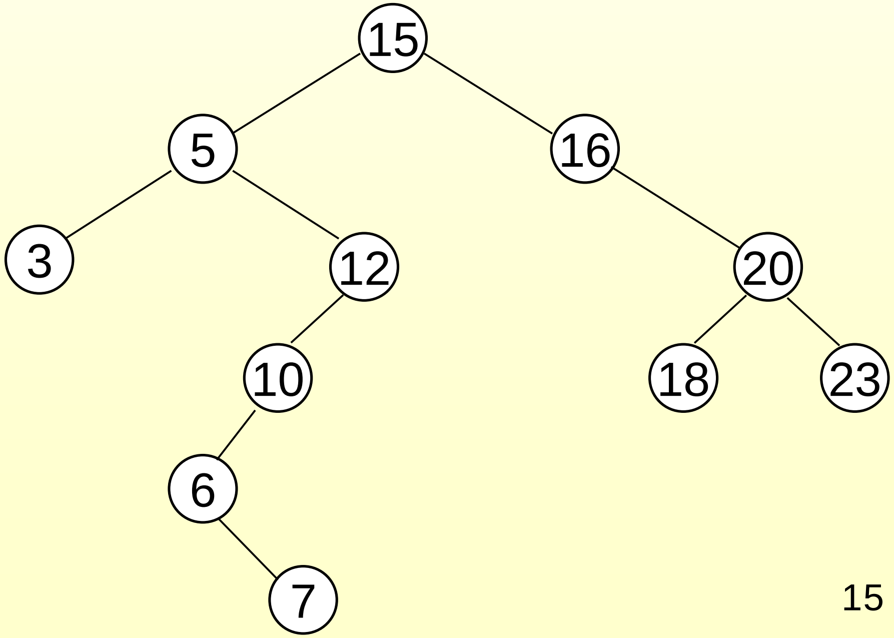


Running time: $O(h)$

Deletion

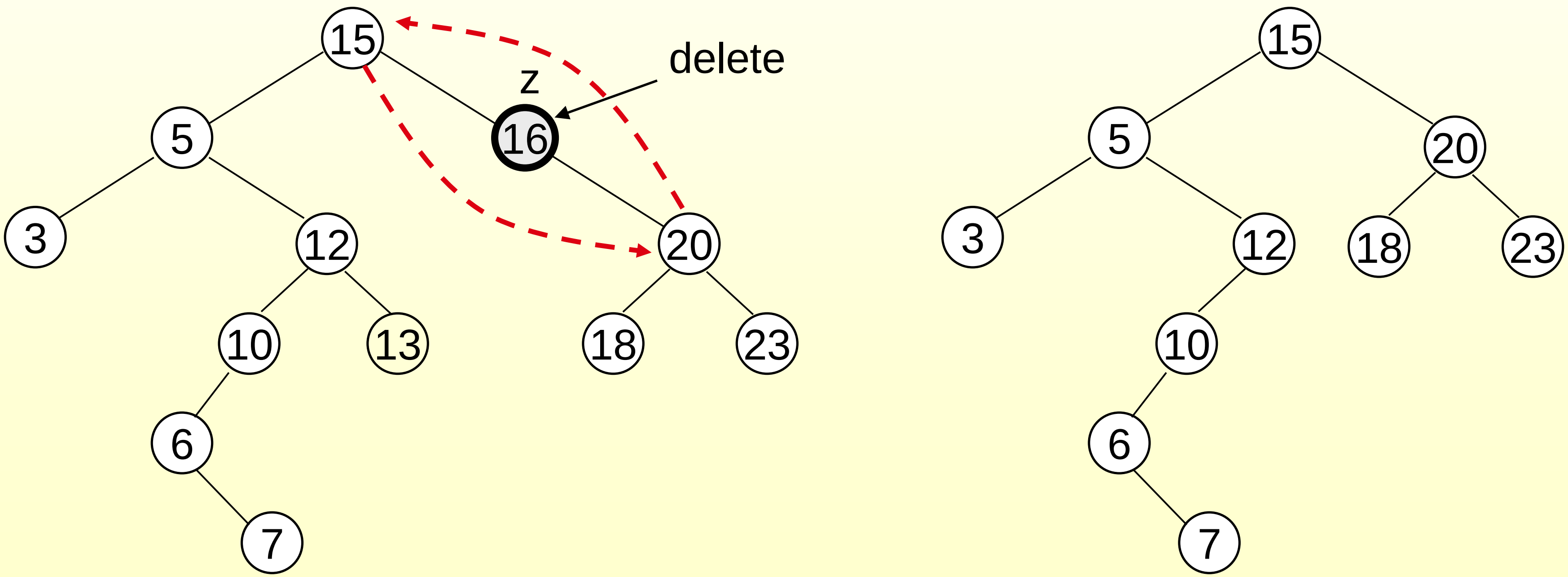
- Goal:
  - Delete a given node **z** from a binary search tree
- Idea:
  - **Case 1: z** has no children
    - Delete **z** by making the parent of **z** point to NIL





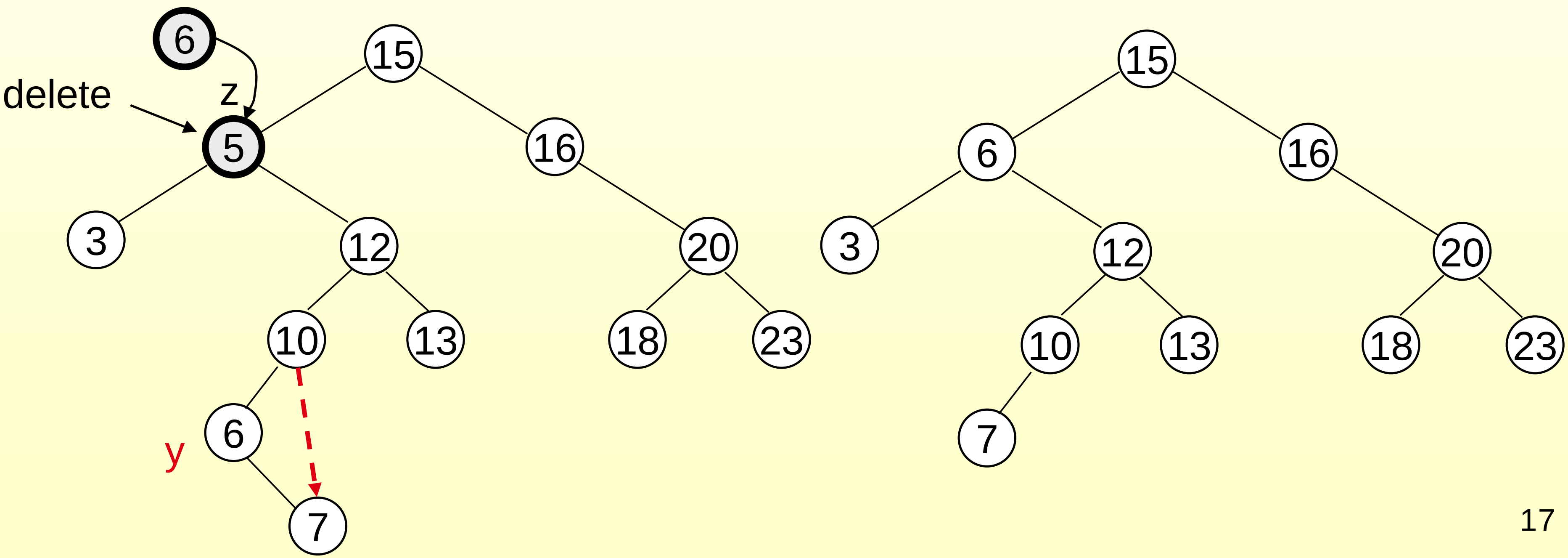
Deletion

- **Case 2: z** has one child
  - Delete **z** by making the parent of **z** point to **z**'s child, instead of to **z**



Deletion

- **Case 3: z** has two children
  - **z**'s successor (y) is the minimum node in **z**'s right subtree
  - y has either no children or one right child (but no left child)
  - Delete y from the tree (via Case 1 or 2)
  - Replace z's key and satellite data with y's.

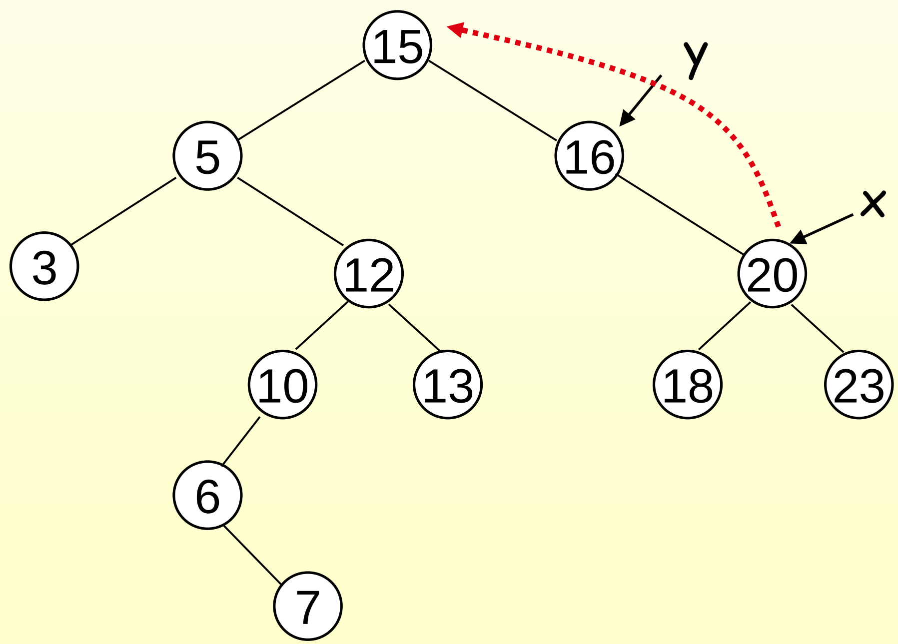


TREE-DELETE(T, z)

```
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
```

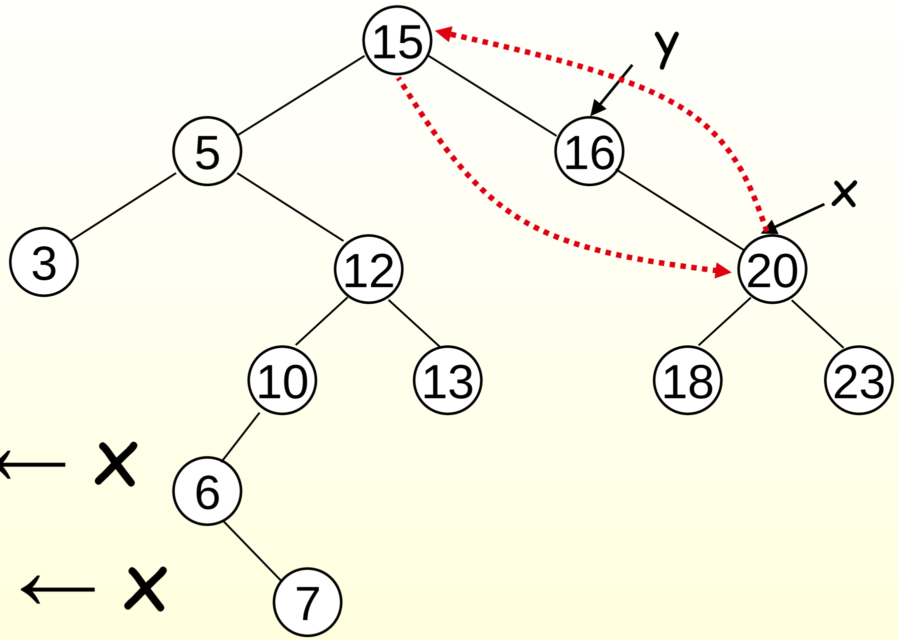
z has one child

z has 2 children



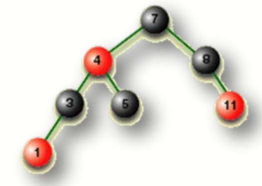
TREE-DELETE(T, z) – cont.

```
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
         copy y's satellite data into z
return y
```



Running time: $O(h)$

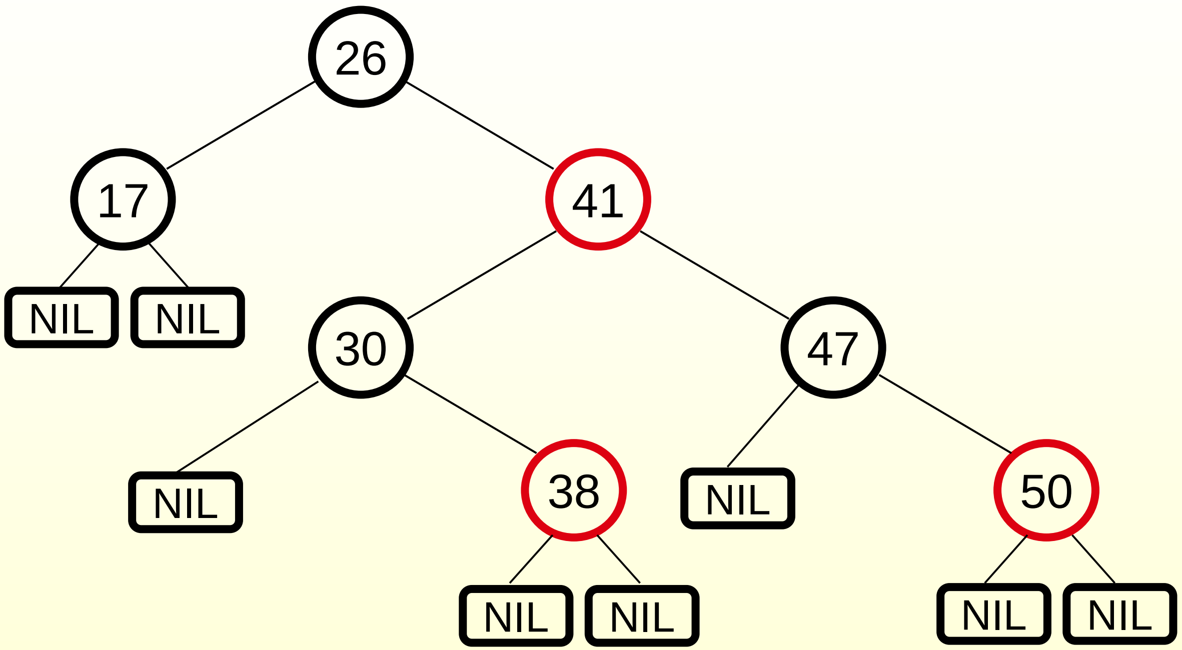
Red-Black Trees



- Balanced binary search trees that guarantee an $O(\lg n)$ running time
- Red-black-tree
  - Binary search tree with an additional binary attribute for its nodes: *color* which can be **red** or **black**
  - Constrains the way nodes can be colored on any path from the root to a leaf:

Ensures that no path is more than twice as long as any other path $\Rightarrow$ the tree is balanced

Example: RED-BLACK-TREE



- For convenience we use a sentinel NIL[T] to represent all the null nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

Red-Black Trees

- Binary search tree + 1 extra bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - *key*, *left*, *right*, and *p* (parent).

- All empty trees (leaves) are colored black.
  - We use a single sentinel, *NIL*, for all the leaves of red-black tree *T*, with *color*[*NIL*] = black.
  - The root's parent is also *NIL*[*T*].

Red-Black-Trees Properties

(**Satisfy the binary search tree property**)

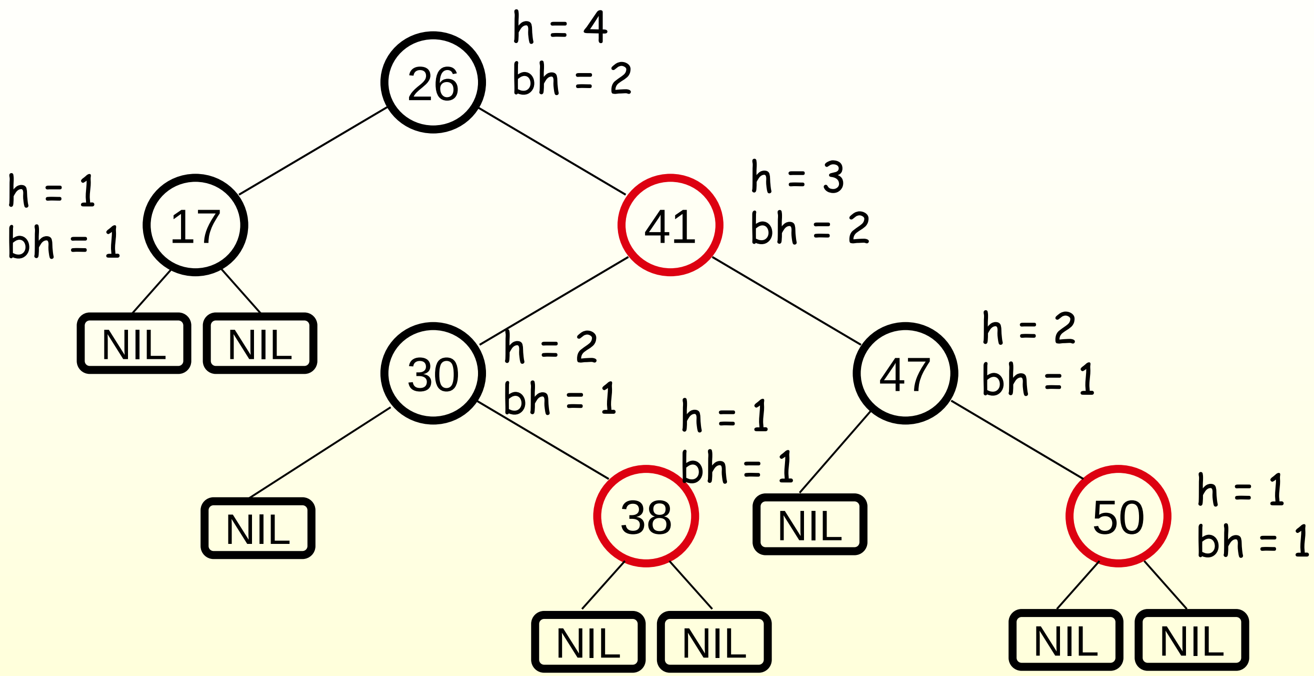
1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf

local

5. For each node, all paths from that node to descendant leaves contain <u>the same number of **black** nodes</u>

global

Black-Height of a Node



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

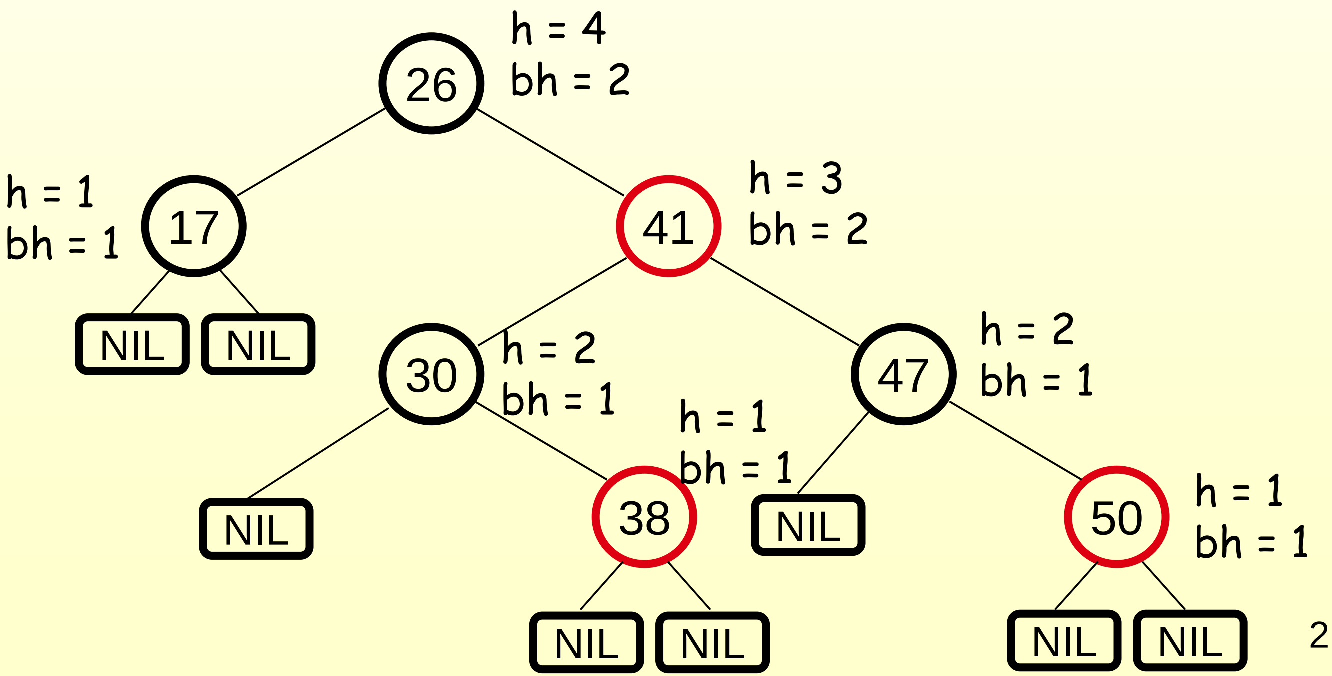
Important Property of Red-Black-Trees

$\boxed{\text{A red-black tree with } n \text{ internal nodes has height } \underline{\text{at most}}\ 2\lg(n+1)}$

- Need to prove two claims first …

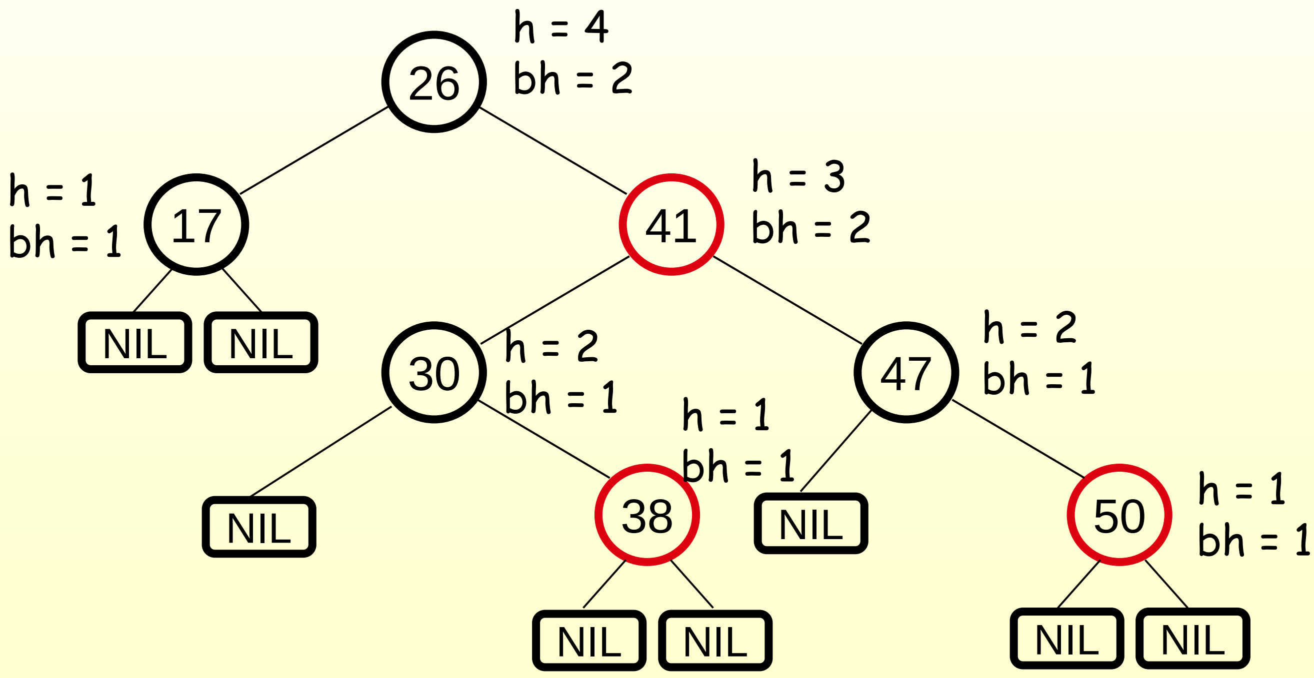
Claim 1

- Any node x with height $h(x)$ has $bh(x) \geq h(x)/2$
- **Proof**
  - By property 4, at most $h/2$ **red** nodes on the path from the node to a leaf
  - Hence at least $h/2$ are **black**



Claim 2

- The subtree rooted at any node **x** contains **at least** $2^{bh(x)} - 1$ internal nodes



Claim 2 (Cont'd)

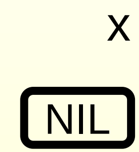
Proof: By induction on **h[x]**

Basis: $h[x] = 0 \Rightarrow$

x is a leaf (NIL[T]) $\Rightarrow$

$bh(x) = 0 \Rightarrow$

# of internal nodes: $2^0 - 1 = 0$



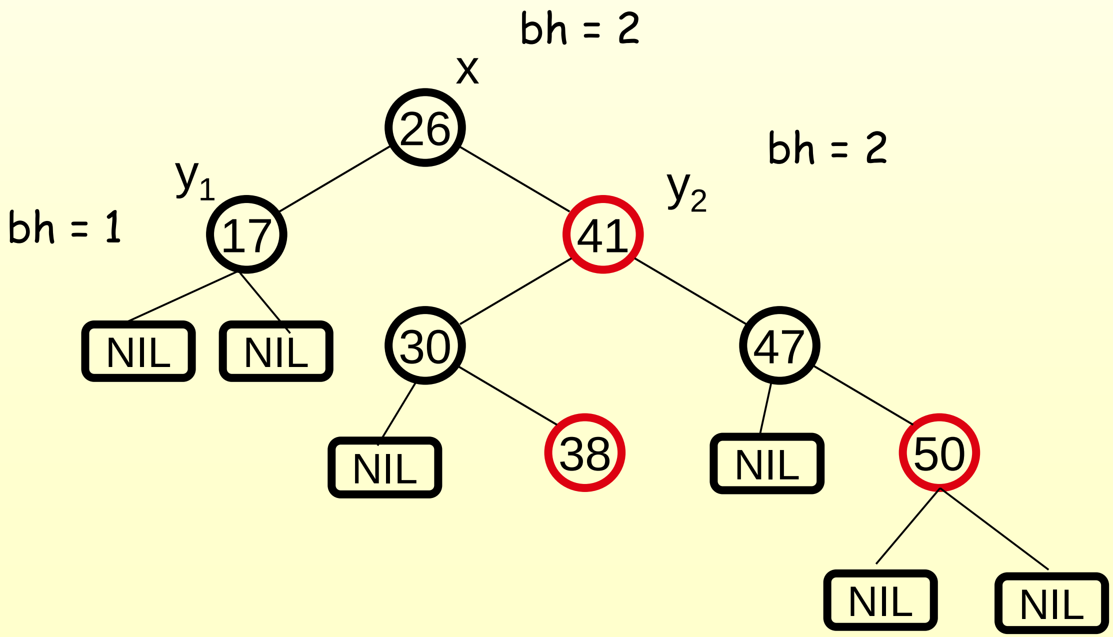
Inductive Hypothesis: assume it is true for

$h[x]=h-1$

Claim 2 (Cont'd)

Inductive step:

- Prove it for h[x]=h
- Let bh(x) = b, then any child y of x has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)



Claim 2 (Cont'd)

- Using inductive hypothesis, the number of internal nodes for each child of x is at least:

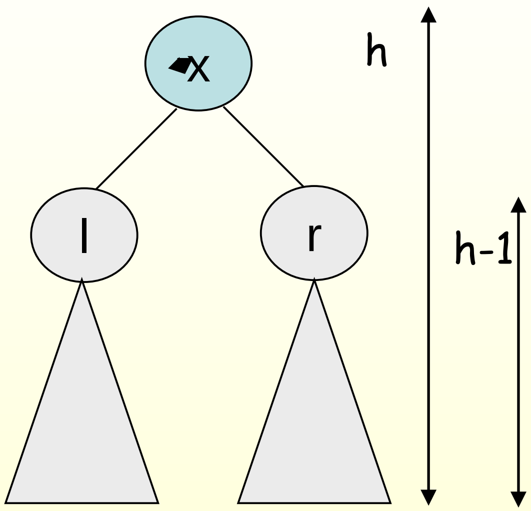
$$2^{bh(x)-1} - 1$$

- The subtree rooted at x contains at least:

$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 =$

$2 \cdot (2^{bh(x)-1} - 1) + 1 =$

$2^{bh(x)} - 1$ internal nodes



bh(l)≥bh(x)-1

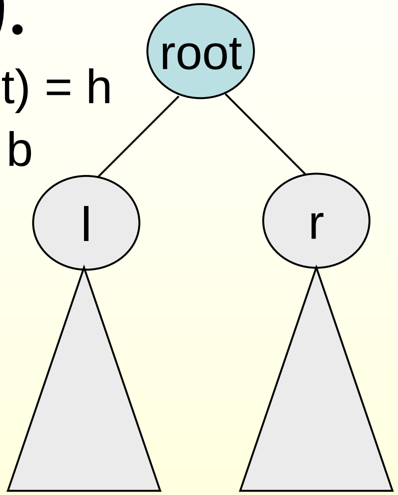
bh(r)≥bh(x)-1

Height of Red-Black-Trees (Cont'd)

Lemma: A red-black tree with n internal nodes has height at most $2\lg(n + 1)$.

Proof:

height(root) = h
bh(root) = b



$n \geq 2^b - 1 \geq 2^{h/2} - 1$

number **n** of internal nodes

since b ≥ h/2

- Add 1 to both sides and then take logs:

$$n + 1 \geq 2^b \geq 2^{h/2}$$

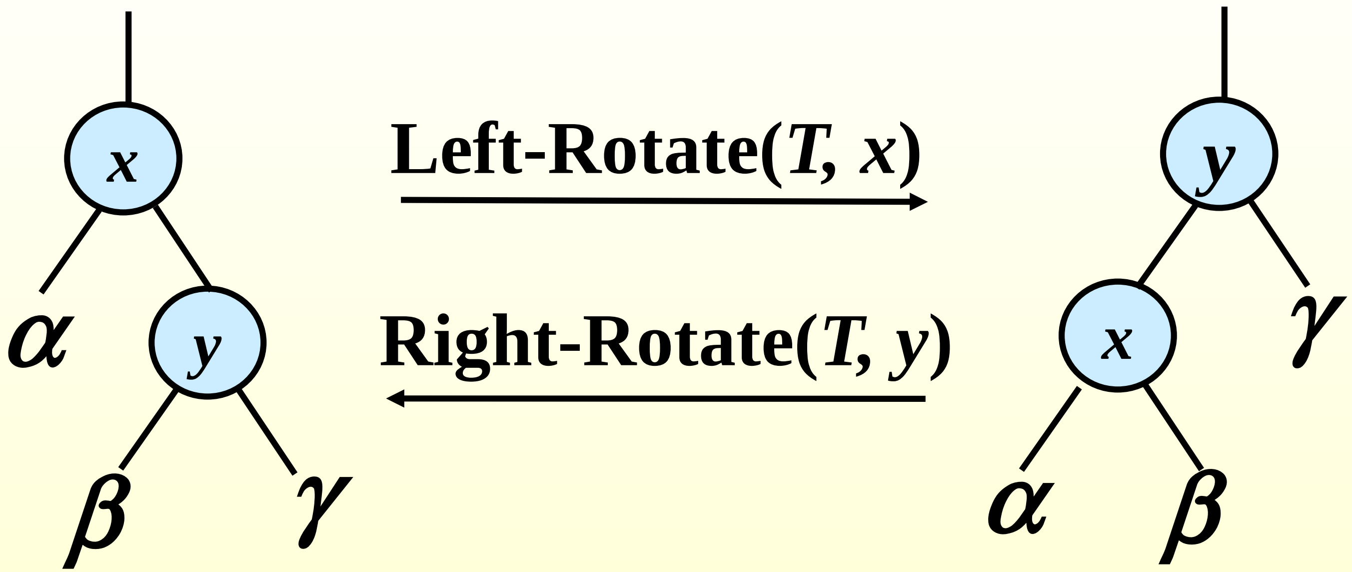
$$\lg(n + 1) \geq h/2 \Rightarrow$$

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

Operations on RB Trees

- All operations can be performed in ***O*(lg *n*)** time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in BSTs.
- Insertion and Deletion are not straightforward. Why?

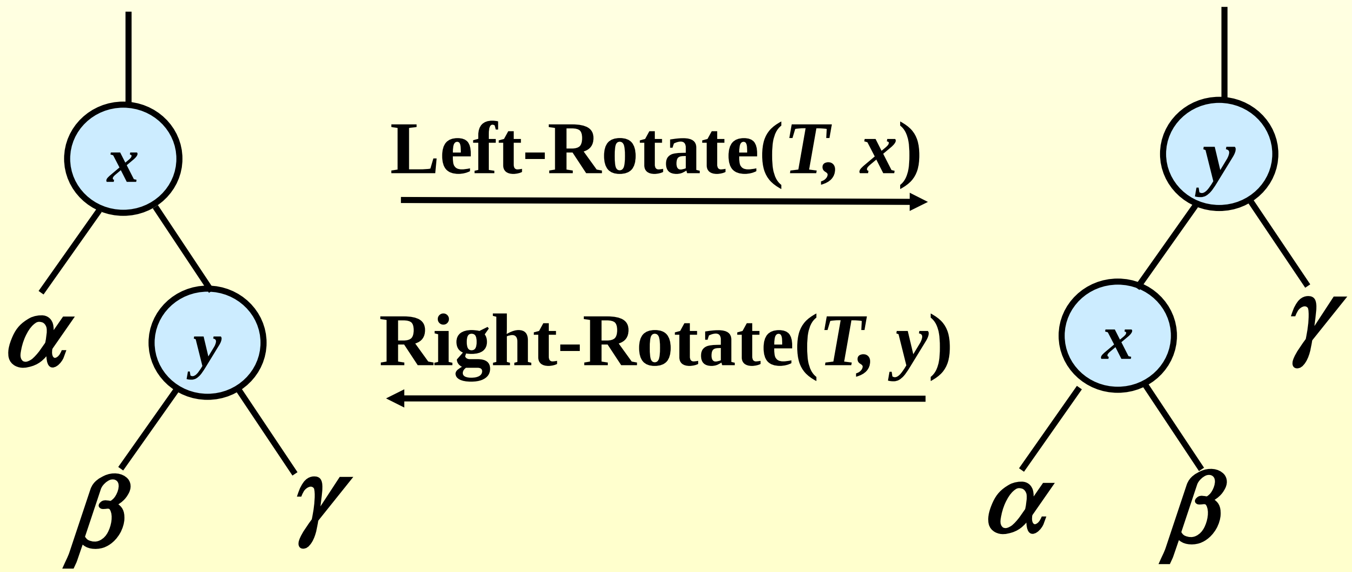
Rotations: Local Tree Rearrangements



Internal tree rebalancing operations

Rotations

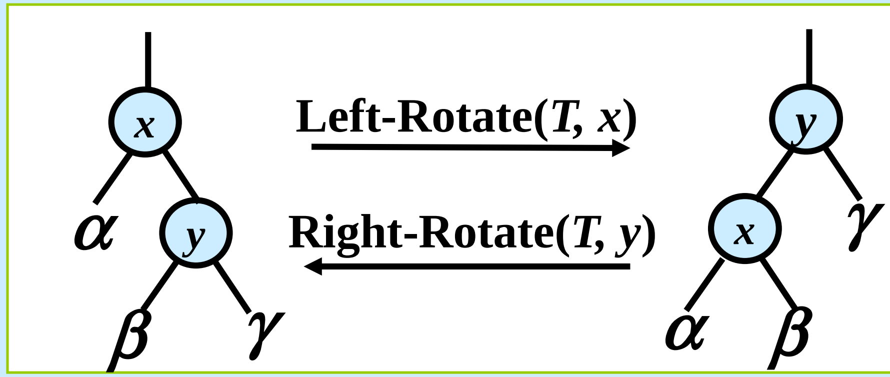
- Rotations are the basic tree-restructuring operations for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Does not violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation – Pseudocode

Left-Rotate (*T, x*)

```
y ← right[x]   // Set y.
right[x] ← left[y]   //Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T ]
      then p[left[y]] ← x
p[y] ← p[x]          // Link x's parent to y.
if p[x] = nil[T ]
      then root[T ] ← y
      else if x = left[p[x]]
              then left[p[x]] ← y
              else right[p[x]] ← y
left[y] ← x          // Put x on y's left.
p[x] ← y
```



Rotation

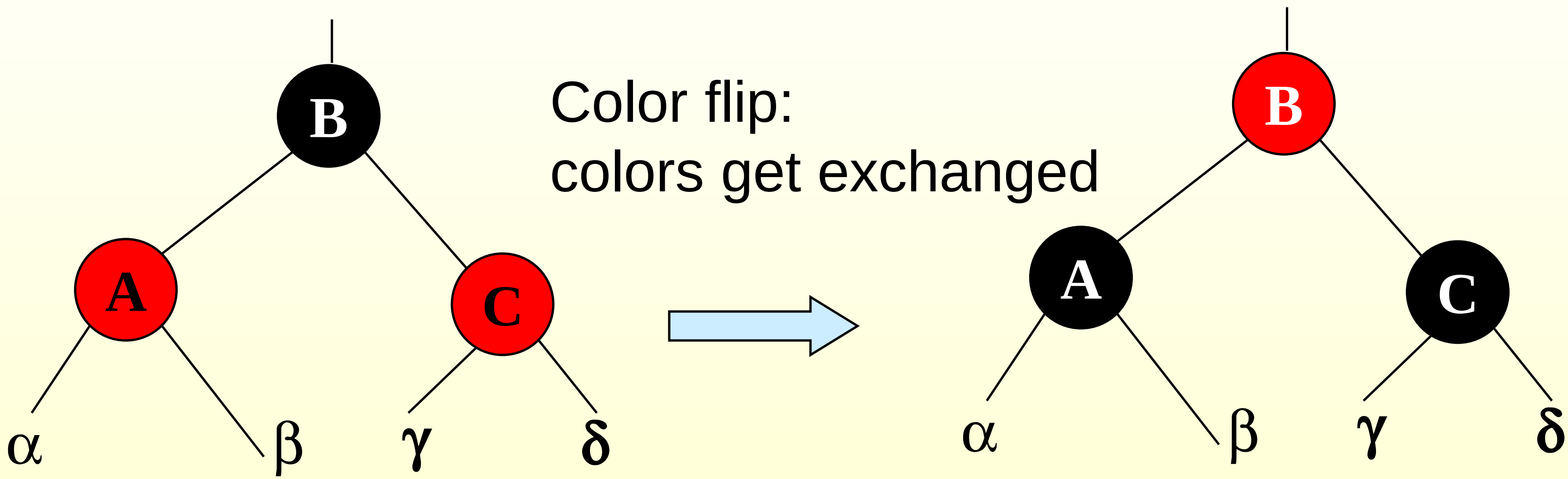
- The pseudo-code for Left-Rotate assumes that
  - *right*[*x*] $\neq$ *nil*[*T*], and
  - root's parent is *nil*[*T*].
- Left Rotation on *x*, makes *x* the left child of *y,* and the left subtree of *y* into the right subtree of *x*.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* everywhere.
- ***Time:*** *O*(1) for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Reminder: Red-Black Properties

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

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

Another Crucial Operation: Color Flip



Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
  - Use Tree-Insert from BST (slightly modified) to insert a node *x* into *T.*
    - Procedure **RB-Insert(*x*)**.
  - Color the node *x* red.
  - Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

Insertion

RB-Insert(*T*, *z*)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = nil[T]
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

RB-Insert(*T*, *z*) Contd.

```
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-Insert-Fixup (T, z)
```

How does it differ from the Tree-Insert procedure of BSTs?

Which of the RB properties might be violated?

Fix the violations by calling RB-Insert-Fixup.

Red-Black-Trees Properties

(**Satisfy the binary search tree property**)

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf

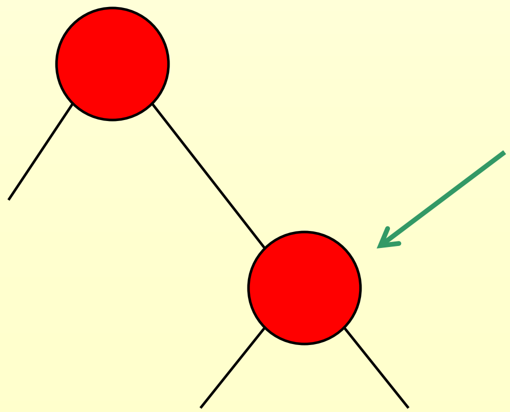
local

5. For each node, all paths from that node to descendant leaves contain <u>the same number of **black** nodes</u>

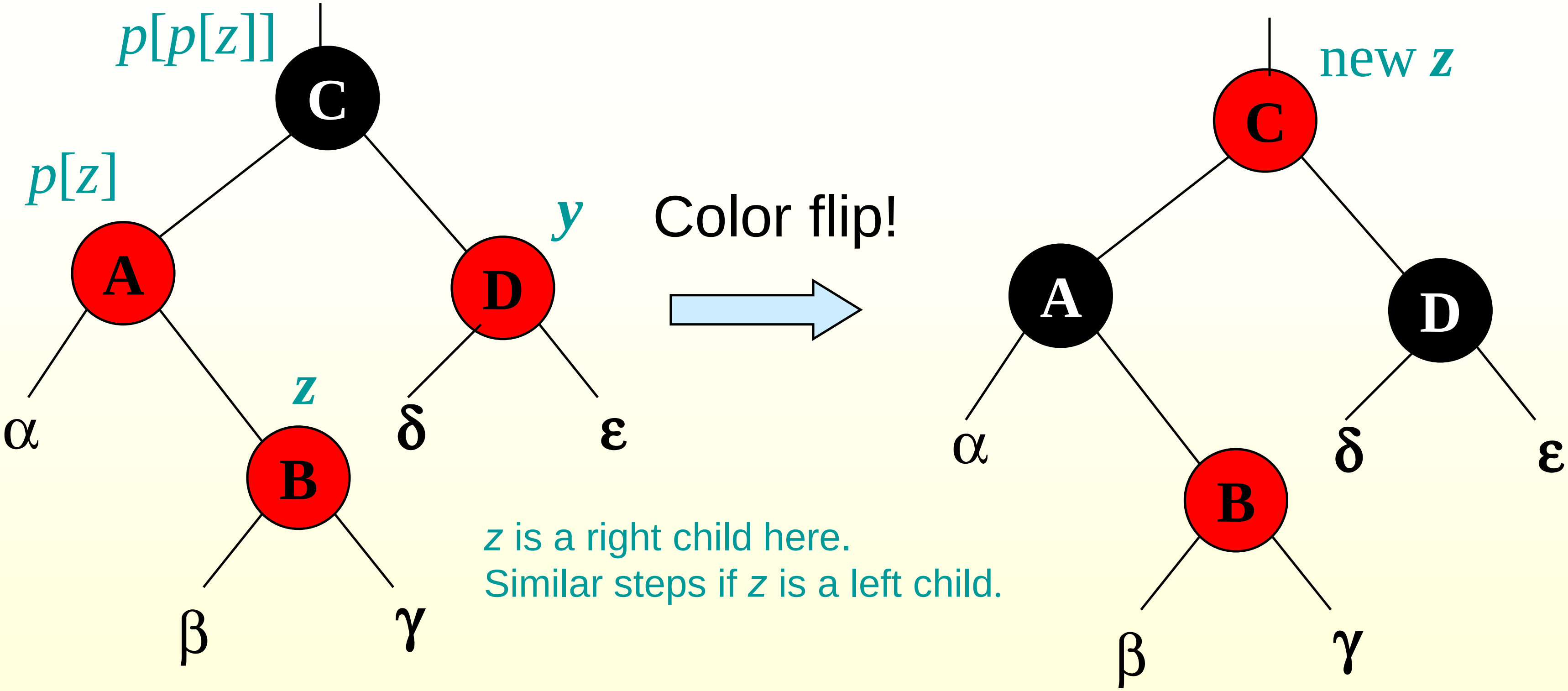
global

Insertion – Fixup

- Problem: we may have one pair of consecutive reds where we did the insertion.
- Solution: rotate it up the tree and away… Three cases have to be handled…



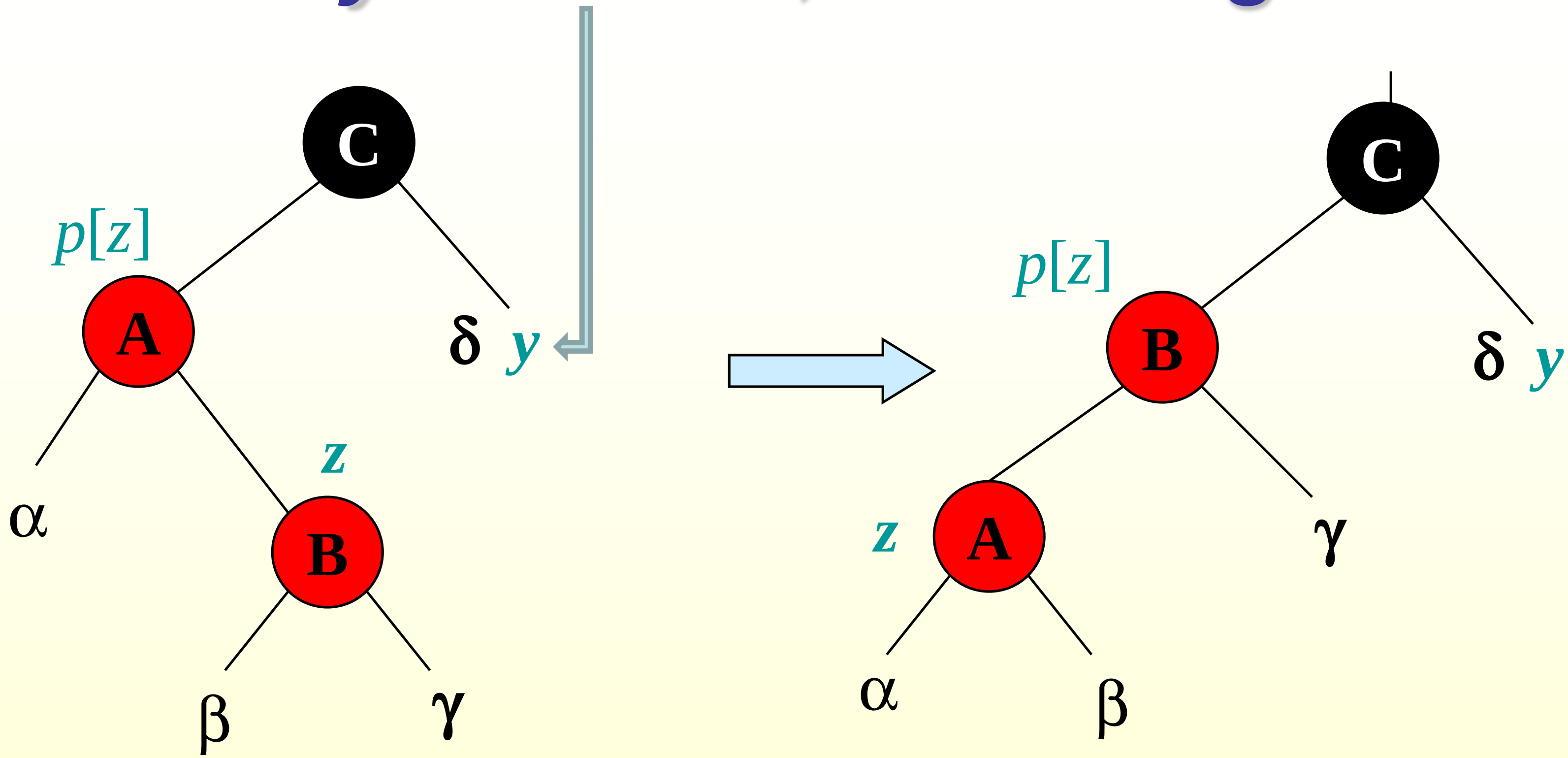
Case 1 – Uncle *y* is Red



z is a right child here.
Similar steps if *z* is a left child.

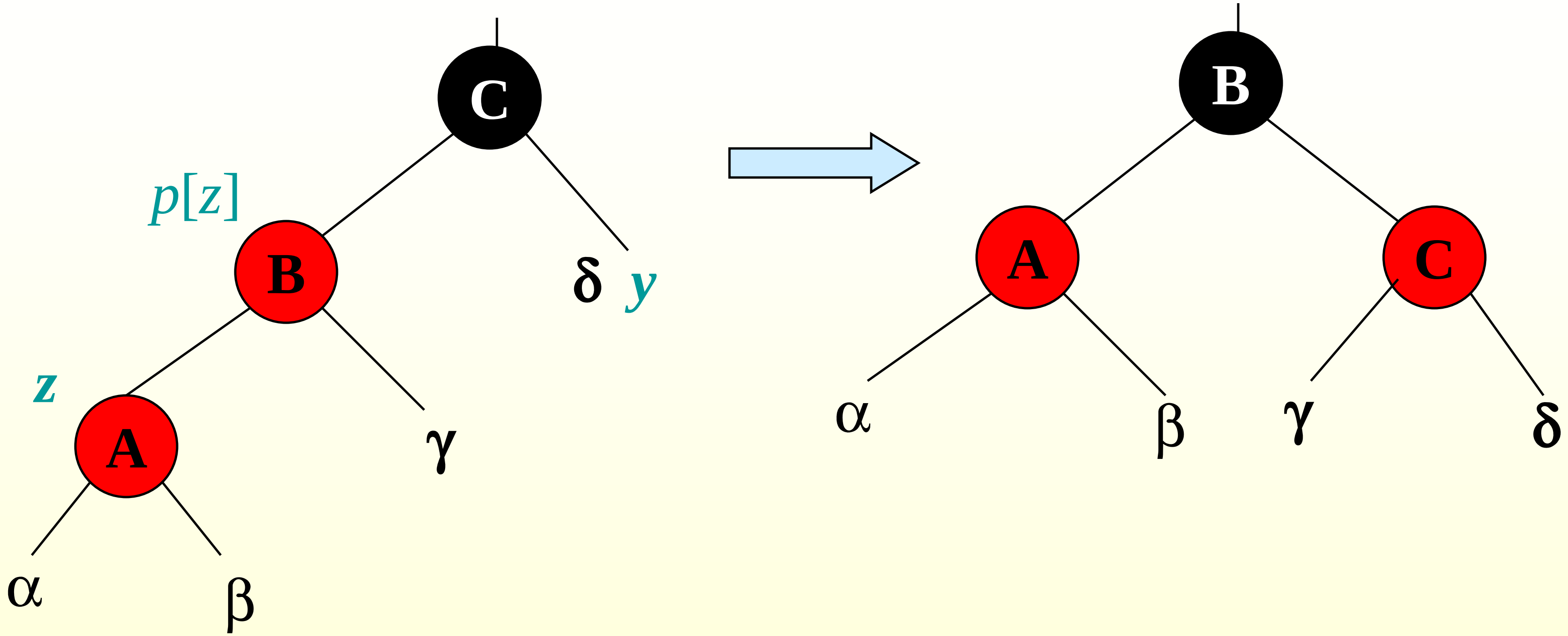
- *p*[*p*[*z*]] (*z*'s grandparent) must be black, since *z* and *p*[*z*] are both red and there are no other violations of property 4.
- Make *p*[*z*] and *y* black $\Longrightarrow$ now *z* and *p*[*z*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*z*]] red $\Longrightarrow$ restores property 5.
- The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).

Case 2 – y is Black, z is a Right Child



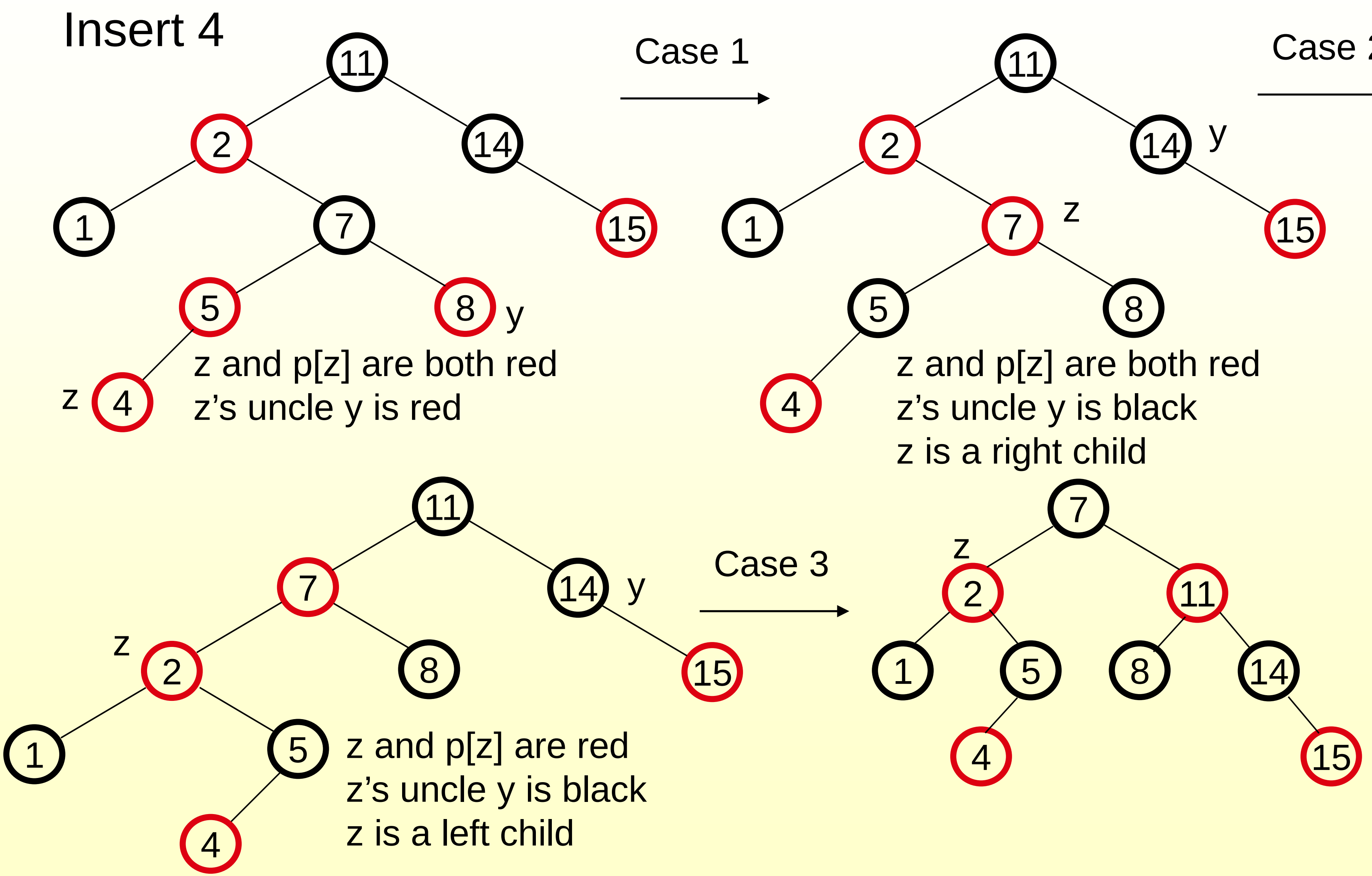
- Left rotate around $p[z]$, $p[z]$ and z switch roles $\Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 3 – *y* is Black, *z* is a Left Child



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Example Insertion



Insertion – Fixup

RB-Insert-Fixup (*T, z*)

```
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
            if color[y] = RED
                then color[p[z]] ← BLACK      // Case 1
                    color[y] ← BLACK           // Case 1
                    color[p[p[z]]] ← RED       // Case 1
                    z ← p[p[z]]                // Case 1
```

Insertion – Fixup

RB-Insert-Fixup(T, z) (Contd.)

9. **else if** $z = right[p[z]]$ // color[y] $\neq$ RED

10. **then** $z \leftarrow p[z]$ // Case 2

11. LEFT-ROTATE(T, z) // Case 2

12. $color[p[z]] \leftarrow$ BLACK // Case 3

13. $color[p[p[z]]] \leftarrow$ RED // Case 3

14. RIGHT-ROTATE(T, $p[p[z]]$) // Case 3

15. **else** (if $p[z] = right[p[p[z]]]$)(same as **10-14**

16. with “right” and “left” exchanged)

17. $color[root[T]] \leftarrow$ BLACK

Correctness

Loop invariant:

- At the start of each iteration of the **while** loop,
  - *z* is red.
  - If *p*[*z*] is the root, then *p*[*z*] is black.
  - There is at most one red-black violation:
    - Property 2: *z* is a red root, or
    - Property 4: *z* and *p*[*z*] are both red.

Correctness – Contd.

- **Initialization:** √
- **Termination:** The loop terminates only if *p*[*z*] is black. Hence, property 4 is OK.
  The last line ensures property 2 always holds.
- **Maintenance:** We drop out when *z* is the root (since then *p*[*z*] is the sentinel *nil*[*T*], which is black). When we start the loop body, the only violation is of property 4.
  - There are 6 cases, 3 of which are symmetric to the other 3. We consider cases in which *p*[*z*] is a left child.
  - Let *y* be *z*'s uncle (*p*[*z*]'s sibling).

Algorithm Analysis

- $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - Each iteration takes $O(1)$ time.
  - Each iteration but the last moves z up 2 levels.
  - $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
  - Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - Note: there are at most 2 rotations overall.

Animations

- Animation:
- https://www.youtube.com/watch?v=rcDF8IqTnyI

- Live demo:
- http://www.cs.usfca.edu/~galles/visualization/RedBlack.html

- Video explanations
- http://www.csanimated.com/animation.php?t=Red-black_tree

Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - Red – OK. Why?
  - Black?
- Steps:
  - Do regular BST deletion.
  - Fix any violations of RB properties that may result.

Deletion

```
RB-Delete(T, z)
1.  if left[z] = nil[T] or right[z] = nil[T]
2.      then y ← z
3.      else y ← TREE-SUCCESSOR(z)
4.  if left[y] = nil[T ]
5.      then x ← left[y]
6.      else x ← right[y]
7.  p[x] ← p[y]   // Do this, even if x is nil[T]
```

y is the node we really delete
x is the node that moves into *y*'s original position

Deletion

RB-Delete (*T*, *z*) (Contd.)

```
if p[y] = nil[T ]
    then root[T ] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y = z
    then key[z] ← key[y]
    copy y's satellite data into z
if color[y] = BLACK
    then RB-Delete-Fixup(T, x)
return y
```

The node passed to the fixup routine is the lone child of the spliced up node, or the sentinel.

Red-Black-Trees Properties

(**Satisfy the binary search tree property**)

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf

   local

5. For each node, all paths from that node to descendant leaves contain <u>the same number of **black** nodes</u>

   global

RB Properties Violation

- If *y* is black, we could have violations of red-black properties:
  - Prop. 1. OK.
  - Prop. 2. If *y* is the root and *x* is red, then the root has become red.
  - Prop. 3. OK.
  - Prop. 4. Violation if *p*[*y*] and *x* are both red.
  - Prop. 5. Any path containing *y* now has 1 fewer black node.

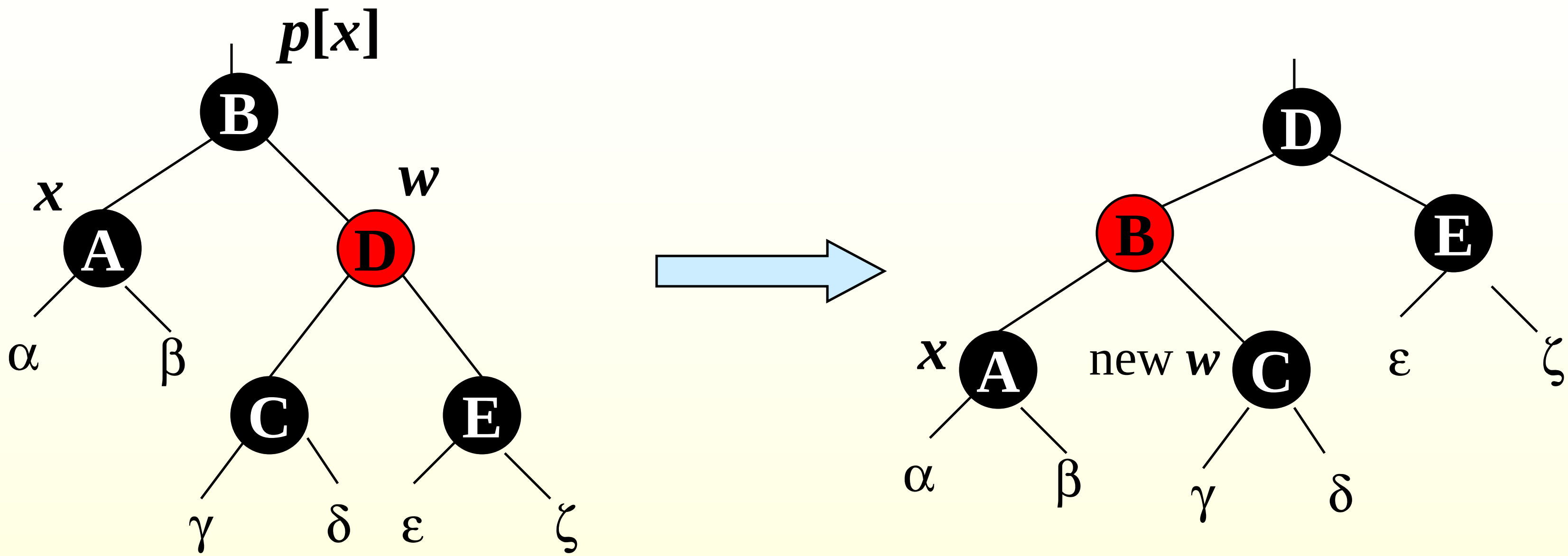
RB Properties Violation

- Prop. 5. Any path containing *y* now has 1 fewer black node.
  - Correct by giving *x* an "extra black."
  - Add 1 to count of black nodes on paths containing *x*.
  - Now property 5 is OK, but property 1 is not.
  - *x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] = RED).
  - The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.
  - In other words, the extra blackness on a node is by virtue of *x* pointing to the node.
- Remove the violations by calling RB-Delete-Fixup.

Deletion – Fixup

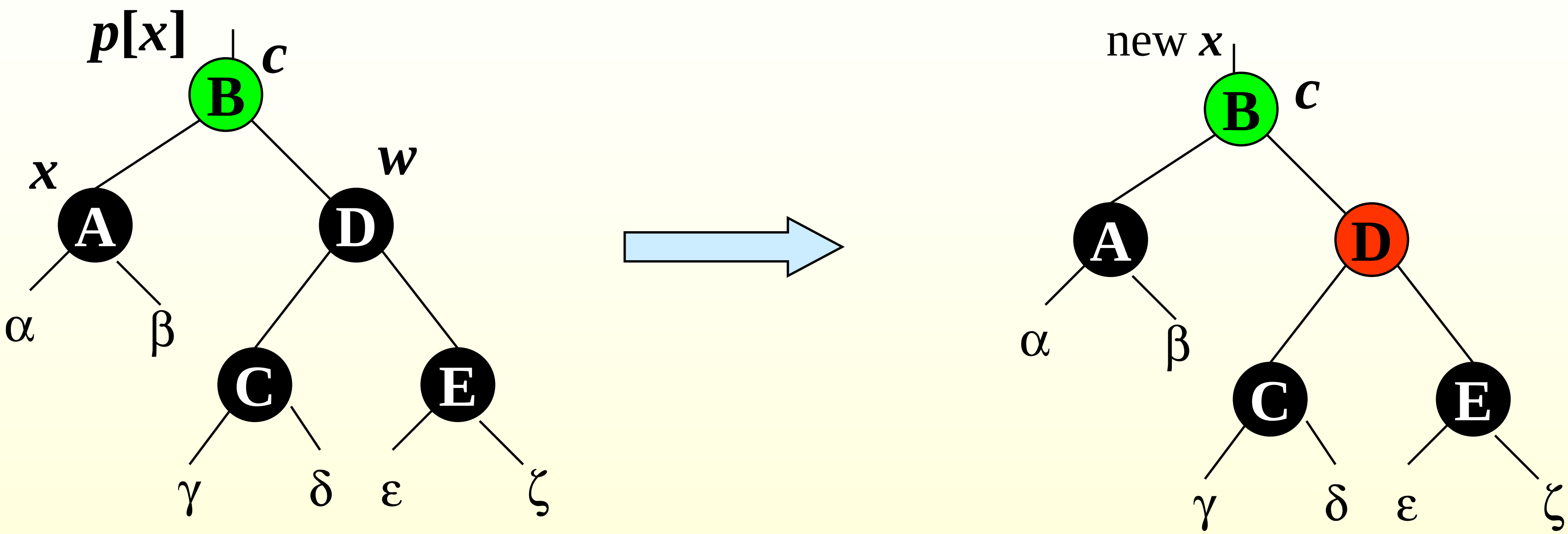
- ***Idea:*** Move the extra black up the tree until *x* points to a red & black node $\Rightarrow$ turn it into a black node,
- *x* points to the root $\Rightarrow$ just remove the extra black, or
- We can do certain rotations and re-colorings to finish.
- Within the **while** loop:
  - *x* always points to a non-root doubly black node.
  - *w* is *x*'s sibling.
  - *w* cannot be *nil*[*T*], since that would violate property 5 at *p*[*x*].
- 8 cases in all, 4 of which are symmetric to the other.

Case 1 – *w* is Red



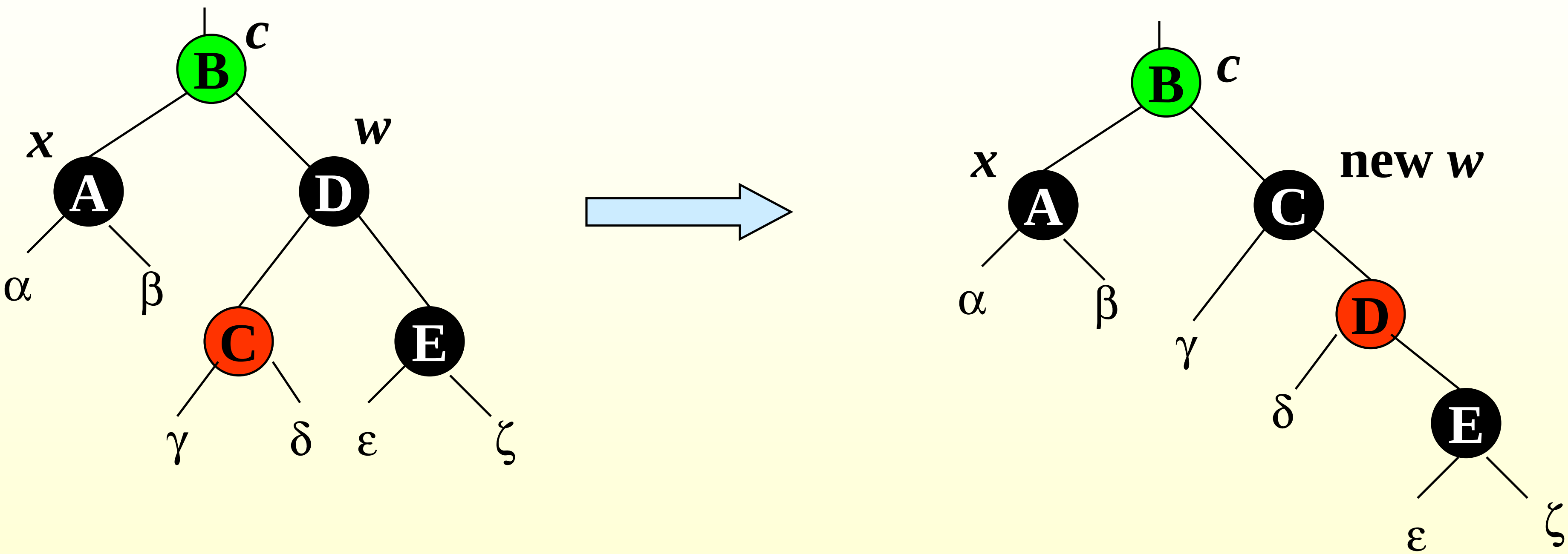
- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] couldn't have been red).
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

Case 2 – *w* is Black, Both *w*'s Children are Black



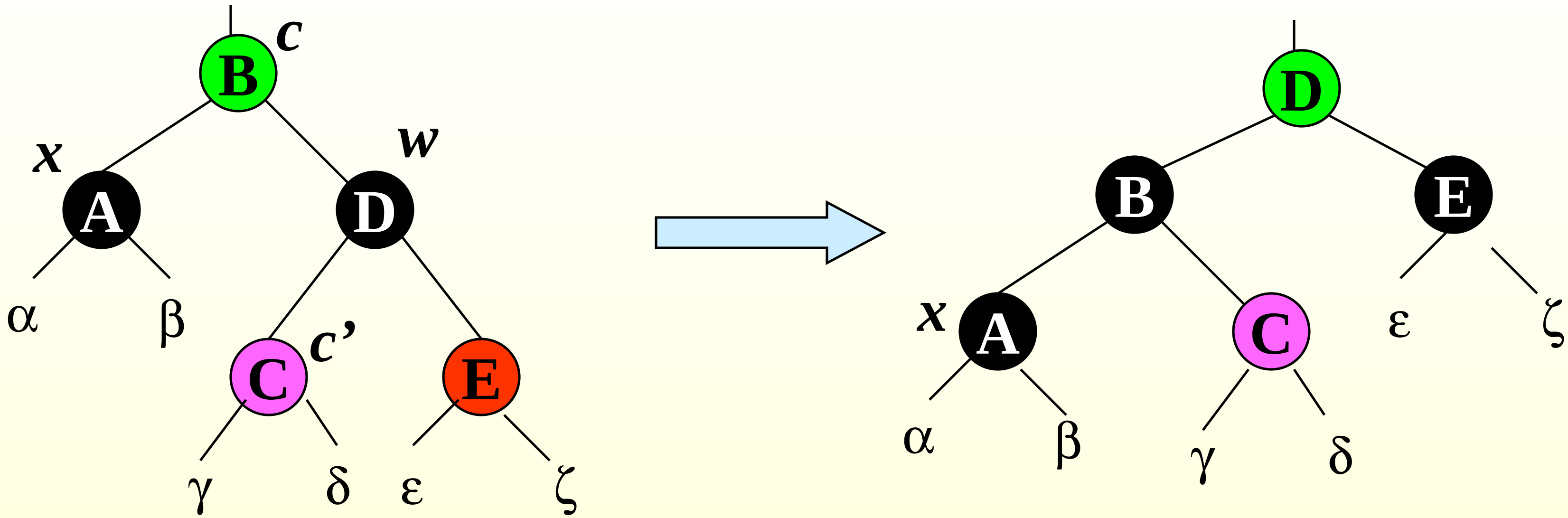
- Take 1 black off *x* (⇒ singly black) and off *w* (⇒ red).
- Move that black to *p*[*x*].
- Do the next iteration with *p*[*x*] as the new *x*.
- If entered this case from case 1, then *p*[*x*] was red ⇒ new *x* is red & black ⇒ color attribute of new *x* is RED ⇒ loop terminates. Then new *x* is made black in the last line.

Case 3 – *w* is Black, *w*'s Left Child is Red, *w*'s Right Child is Black



- Make *w* red and *w*'s left child black.
- Then right rotate on *w*.
- New sibling *w* of *x* is black with a red right child ⇒ case 4.

Case 4 – *w* is Black, *w*'s Right Child is Red



- Make *w* be *p*[*x*]'s color (*c*).
- Make *p*[*x*] black and *w*'s right child black.
- Then left rotate on *p*[*x*].
- Remove the extra black on *x* (⇒ *x* is now singly black) without violating any red-black properties.
- All done. Setting *x* to root causes the loop to terminate.

Deletion – Fixup

RB-Delete-Fixup(T, x)

```
1.  while x ≠ root[T ] and color[x] = BLACK
2.      do if x = left[p[x]]
3.            then w ← right[p[x]]
4.                 if color[w] = RED
5.                    then color[w] ← BLACK          // Case 1
6.                         color[p[x]] ← RED         // Case 1
7.                         LEFT-ROTATE(T, p[x])      // Case 1
8.                         w ← right[p[x]]           // Case 1
```

RB-Delete-Fixup(*T*, *x*) (Contd.)

```
        /* x is still left[p[x]] */
    if color[left[w]] = BLACK and color[right[w]] = BLACK
        then color[w] ← RED                              // Case 2
             x ← p[x]                                    // Case 2
        else if color[right[w]] = BLACK
                then color[left[w]] ← BLACK              // Case 3
                     color[w] ← RED                      // Case 3
                     RIGHT-ROTATE(T,w)                   // Case 3
                     w ← right[p[x]]                     // Case 3
             color[w] ← color[p[x]]                      // Case 4
             color[p[x]] ← BLACK                         // Case 4
             color[right[w]] ← BLACK                     // Case 4
             LEFT-ROTATE(T, p[x])                        // Case 4
             x ← root[T ]                                // Case 4
  else (same as then clause with "right" and "left"
exchanged)
color[x] ← BLACK
```

Analysis

- $O(\lg n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - Case 2 is the only case in which more iterations occur.
    - x moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq$ 3 rotations in all.
  - Hence, $O(\lg n)$ time.

Animations

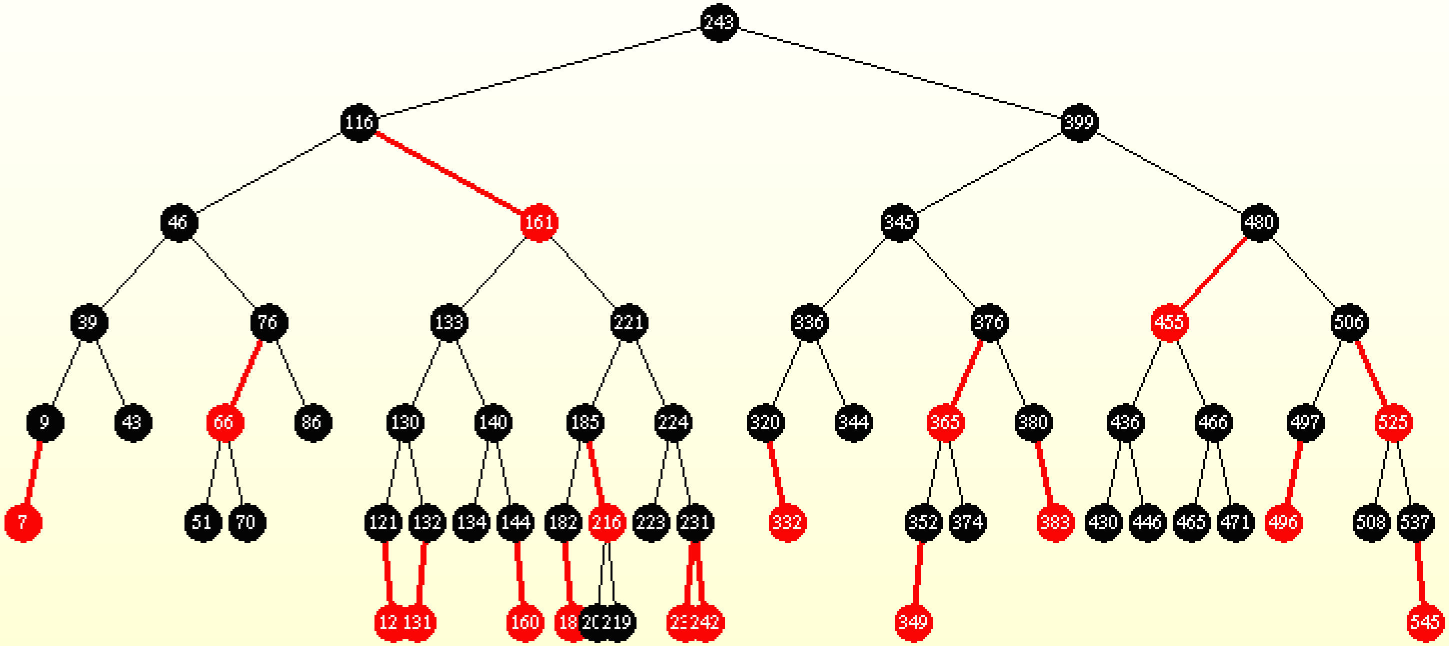
- Animation:
- https://www.youtube.com/watch?v=rcDF8IqTnyI

- Live demo:
- http://www.cs.usfca.edu/~galles/visualization/RedBlack.html

- Video explanations
- http://www.csanimated.com/animation.php?t=Red-black_tree

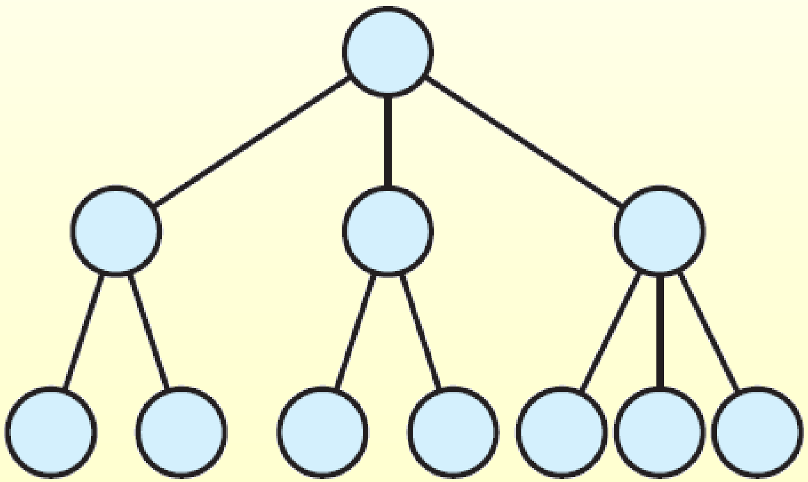
Hysteresis : or the Value of Laziness

- The red nodes give us some slack – we don't have to keep the tree perfectly balanced.
- The colors make the analysis and code much easier than some other types of balanced trees.
- Still, these aren't free – balancing costs some time on insertion and deletion.


243
116
399
46
161
345
480
39
76
133
221
336
376
455
506
9
43
66
86
130
140
185
224
320
344
365
380
436
466
497
525
7
51
70
121
132
134
144
182
216
223
231
332
352
374
383
430
446
465
471
496
508
537
131
160
219
242
349
545

2-3 Trees

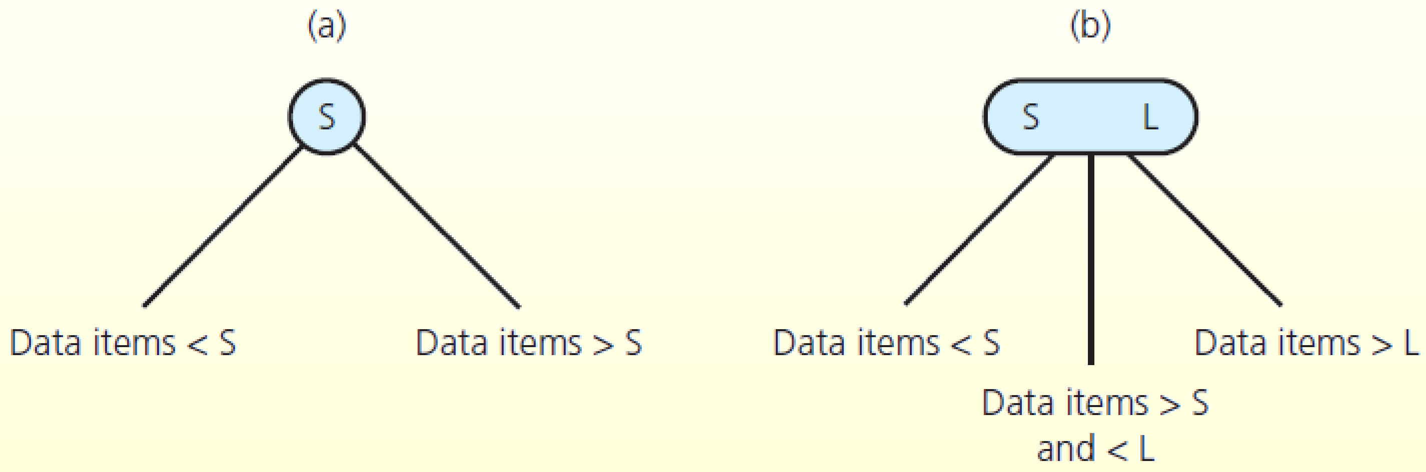
- A 2-3 tree is not a binary tree
- A 2-3 tree is always fully balanced, so height is O(lg n)



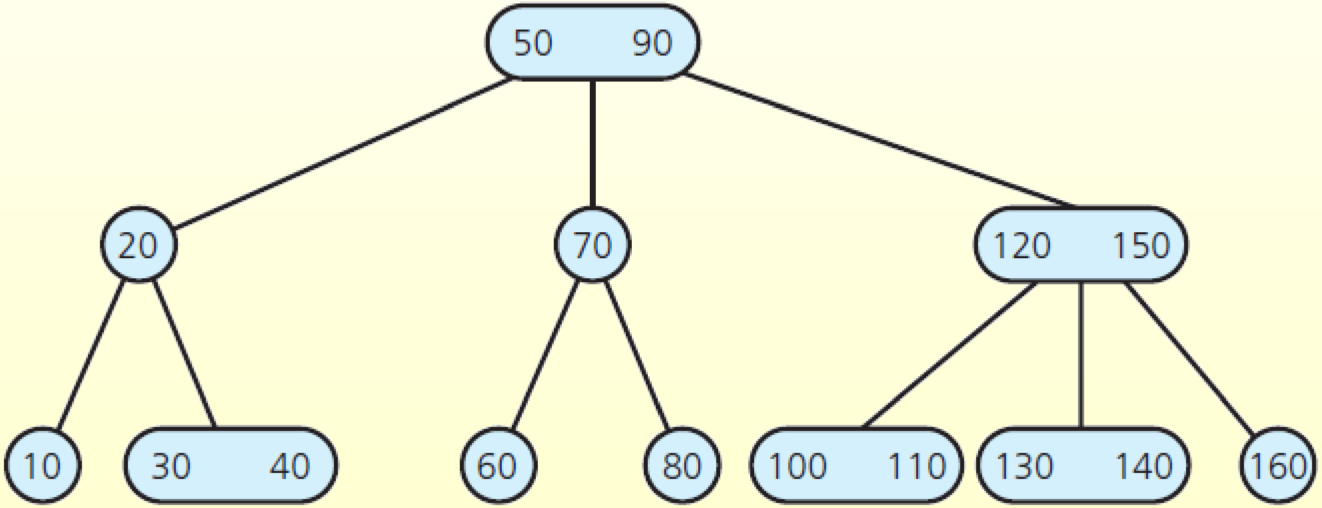
2-3 Trees

- Placing data items in nodes of a 2-3 tree
  - A 2-node (has two children) must contain single data item greater than left child’s item(s) and less than right child’s item(s)
  - A 3-node (has three children) must contain two data items, S and L , such that
    - S is greater than left child’s item(s) and less than middle child’s item(s);
    - L is greater than middle child’s item(s) and less than right child’s item(s).
  - Leaf may contain either one or two data items.

2-3 Trees


(a)
S
Data items < S
Data items > S
(b)
S
L
Data items < S
Data items > S
and < L
Data items > L

2-3 Tree Example



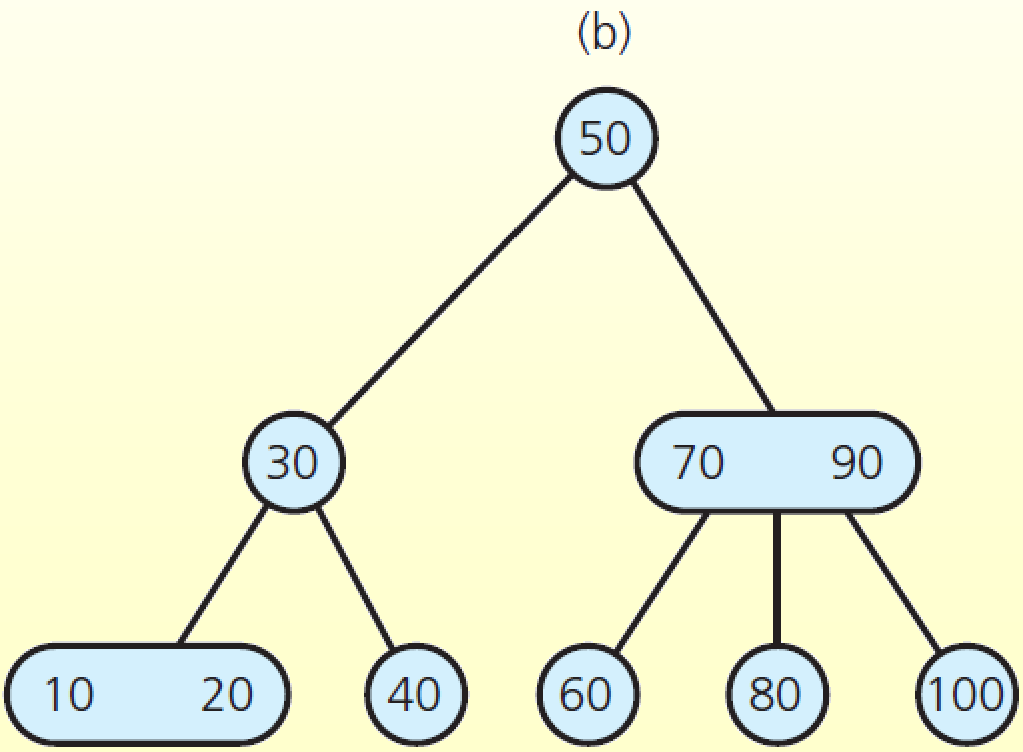
Traversing a 2-3 Tree

- Traverse 2-3 tree in sorted order by performing analogue of inorder traversal on binary tree:

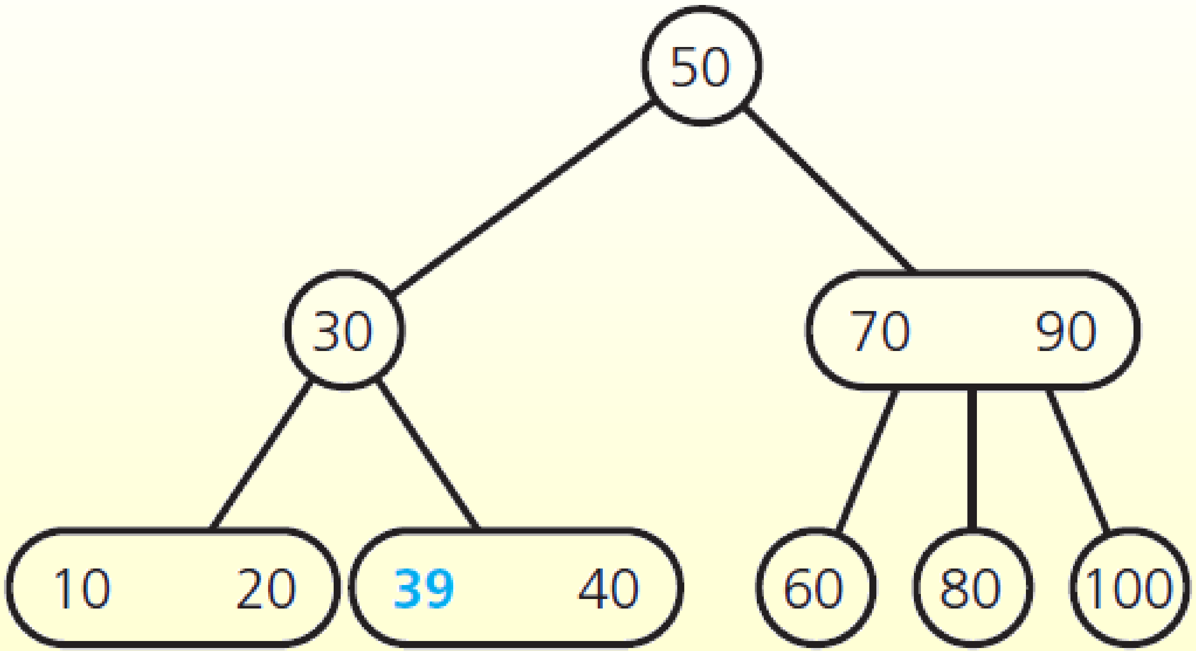
```
// Traverses a nonempty 2-3 tree in sorted order.
inorder(23Tree: TwoThreeTree): void

  if (23Tree's root node r is a leaf)
      Visit the data item(s)
  else if (r has two data items)
  {
      inorder(left subtree of 23Tree's root)
      Visit the first data item
      inorder(middle subtree of 23Tree's root)
      Visit the second data item
      inorder(right subtree of 23Tree's root)
  }
  else // r has one data item
  {
      inorder(left subtree of 23Tree's root)
      Visit the data item
      inorder(right subtree of 23Tree's root)
  }
```

Searching a 2-3 Tree

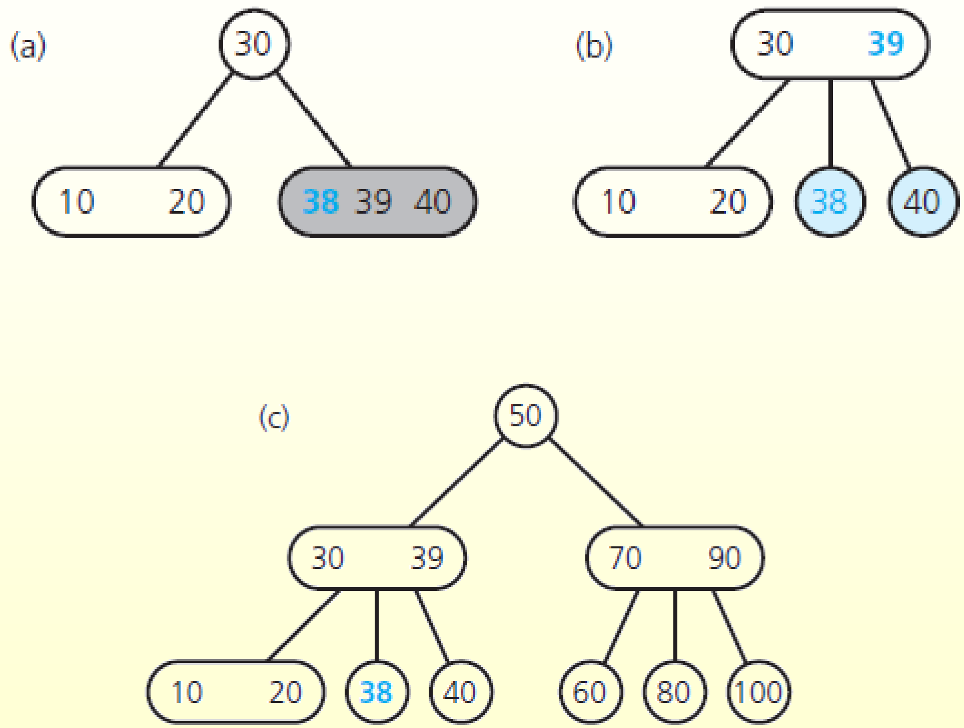
- Retrieval operation for 2-3 tree similar to retrieval operation for binary search tree



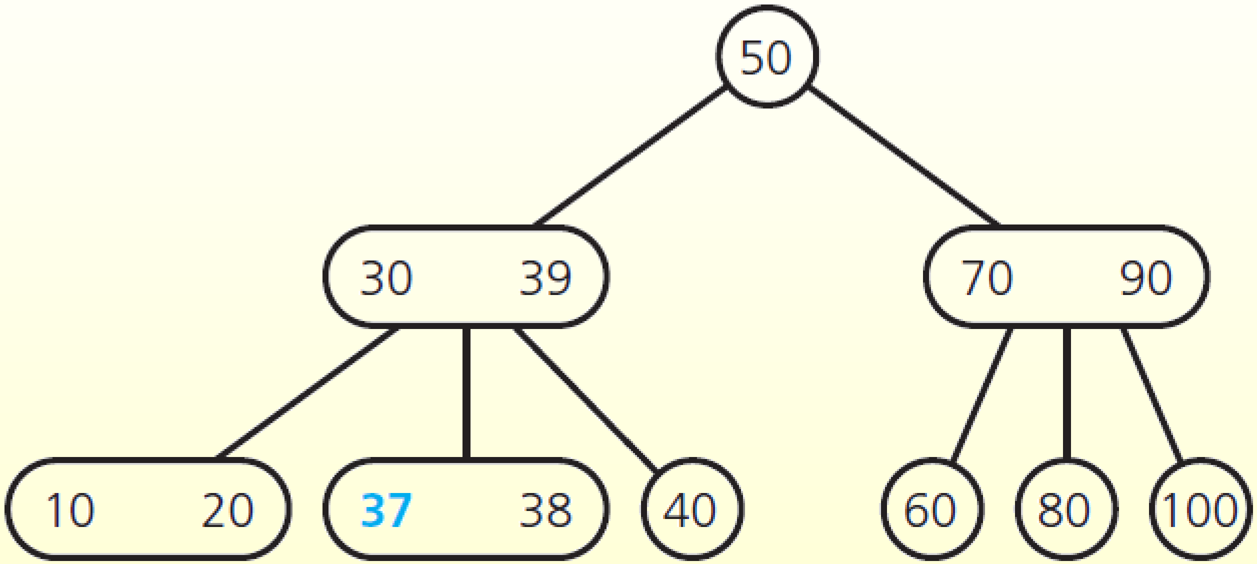
Inserting Data into a 2-3 Tree



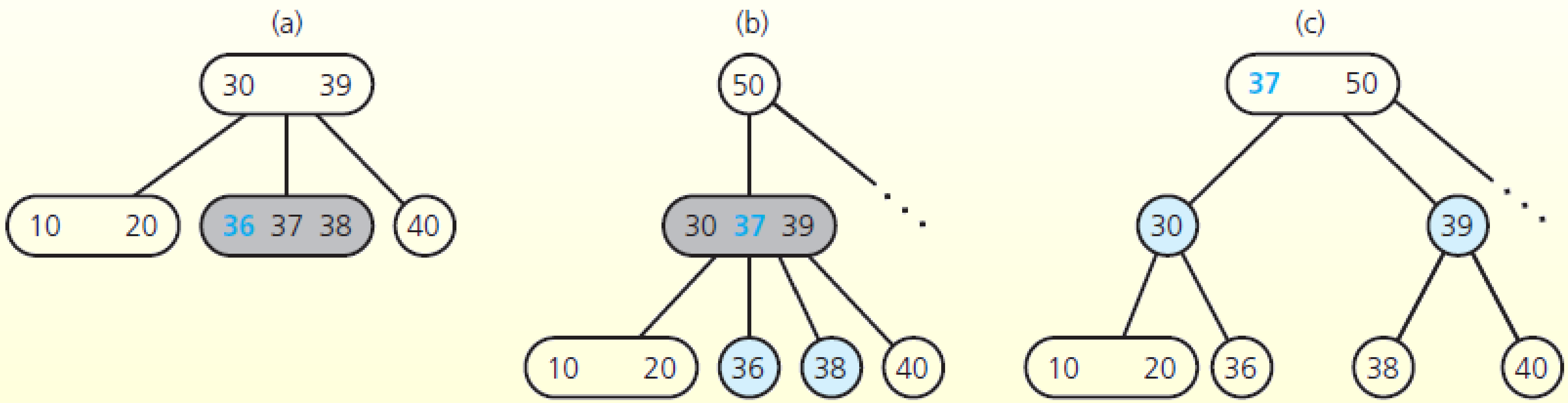
Inserting Data into a 2-3 Tree



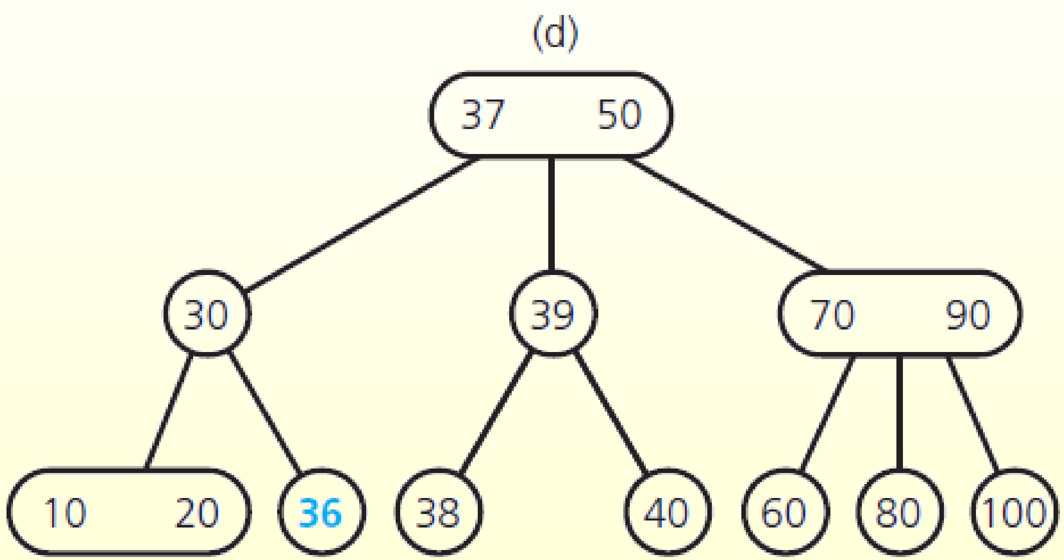
Inserting Data into a 2-3 Tree



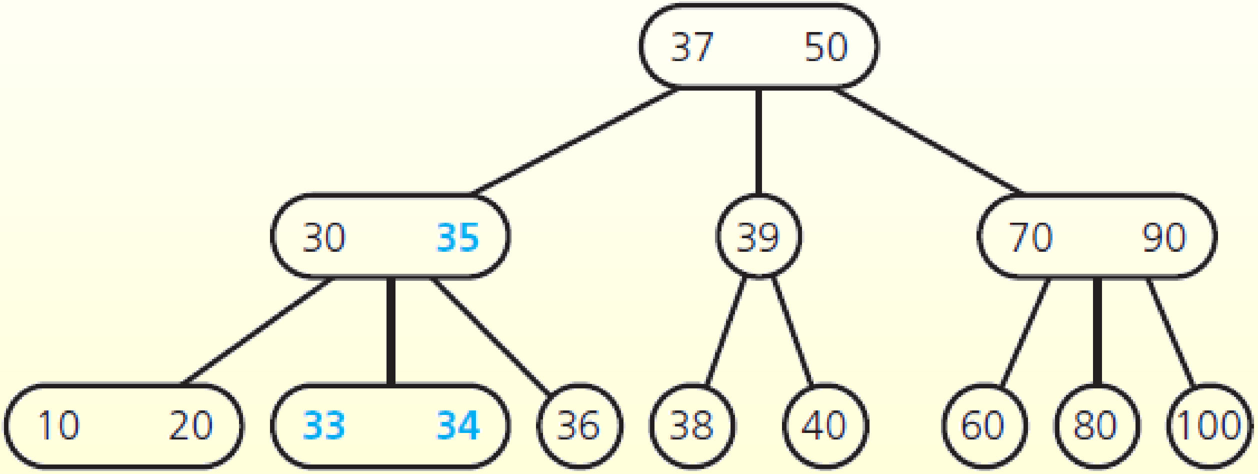
Inserting Data into a 2-3 Tree



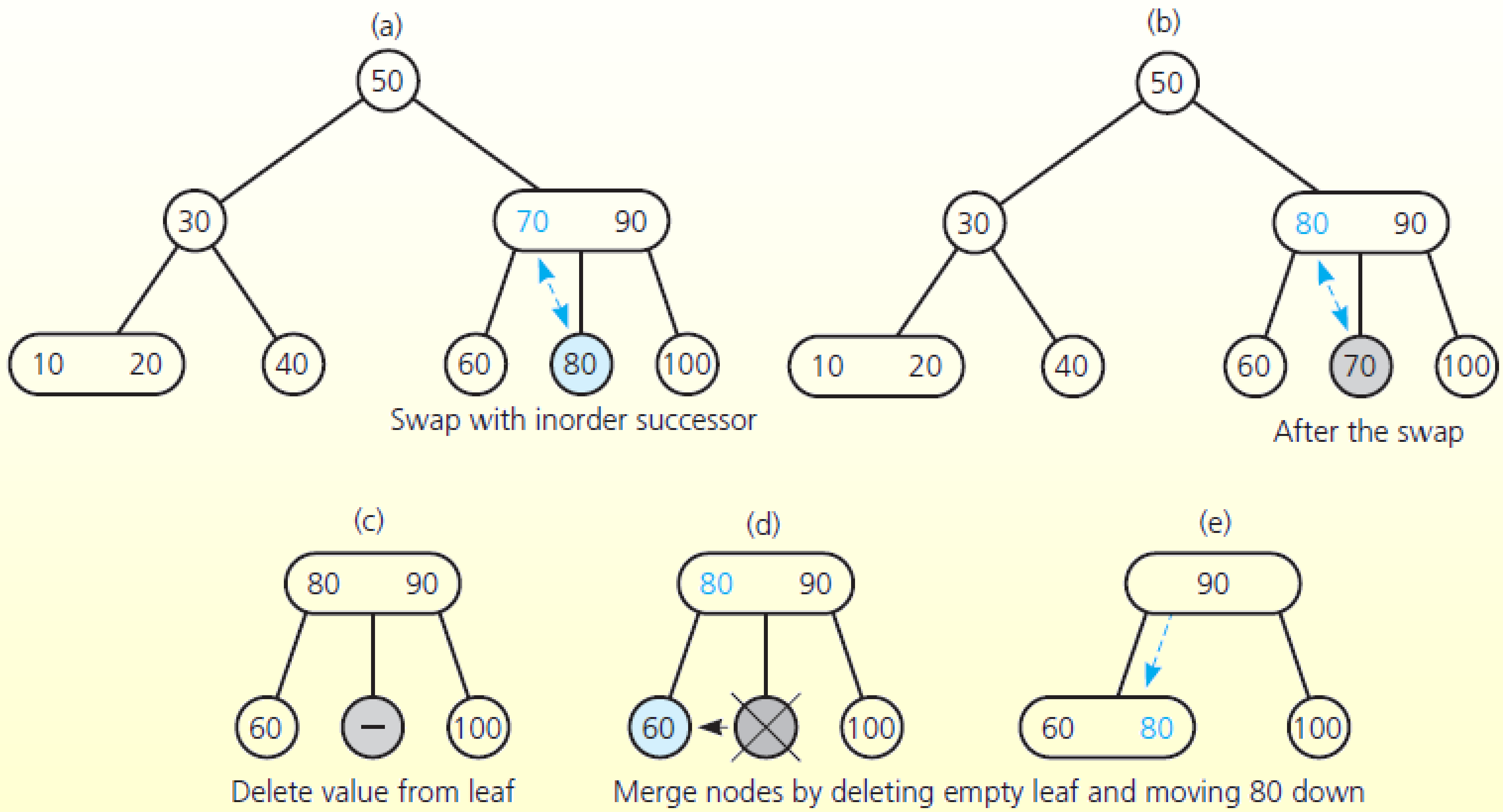
Inserting Data into a 2-3 Tree



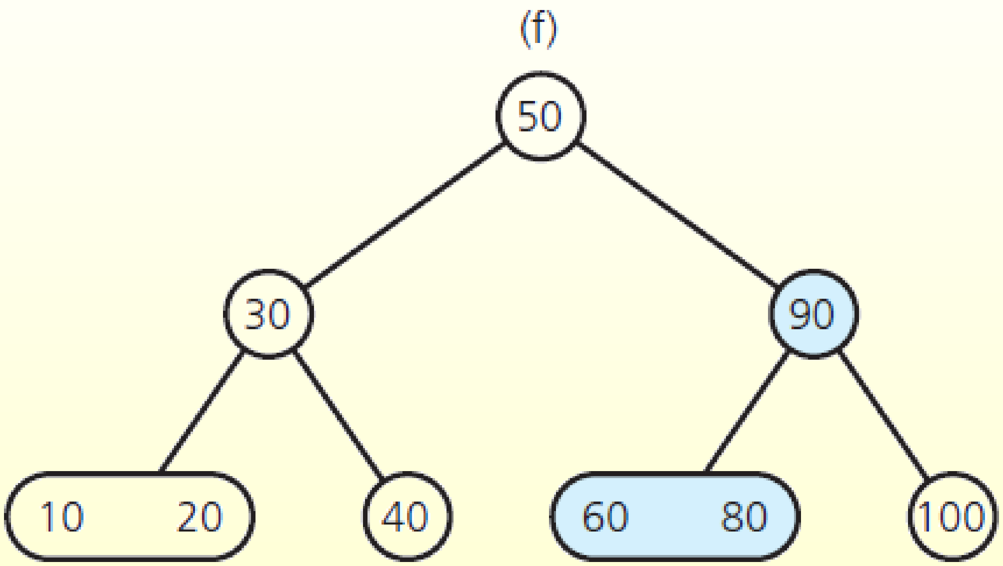
Inserting Data into a 2-3 Tree



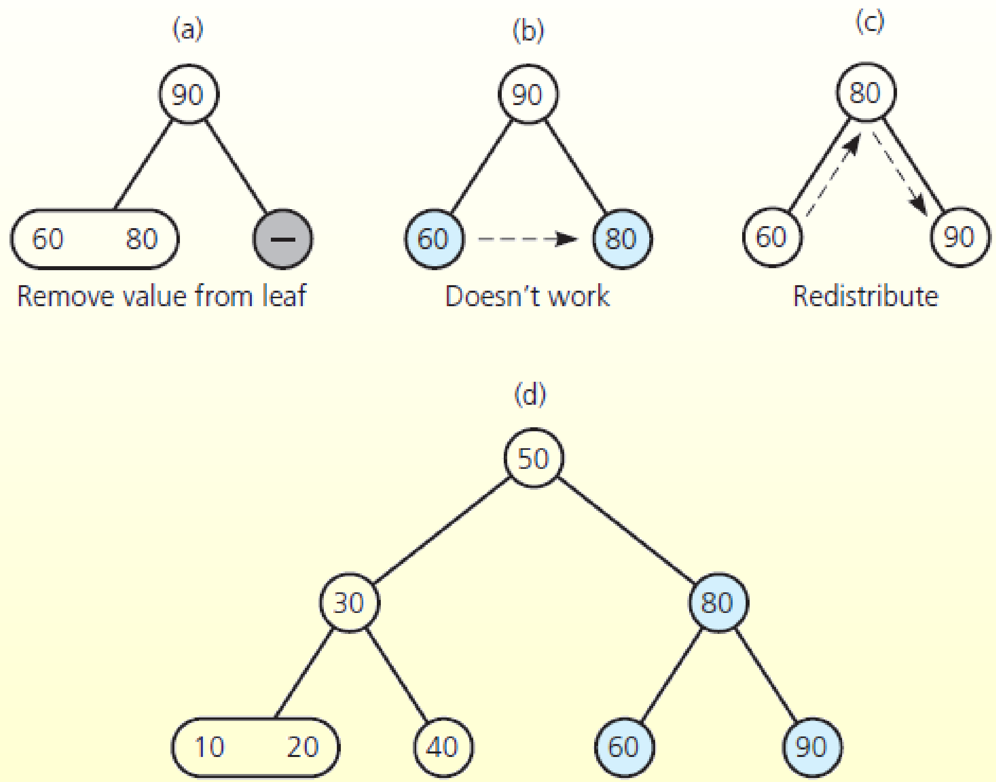
Deleting Data from a 2-3 Tree



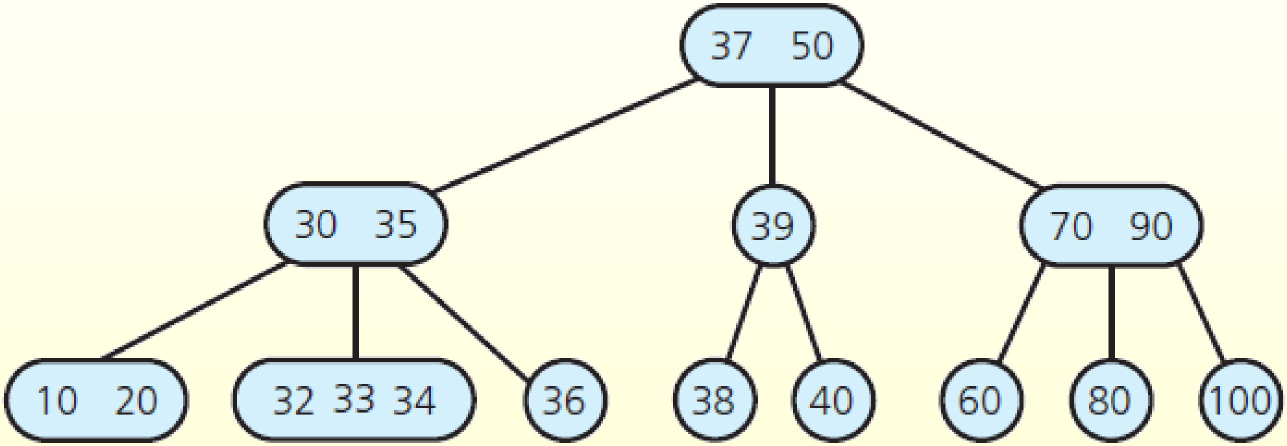
Deleting Data from a 2-3 Tree



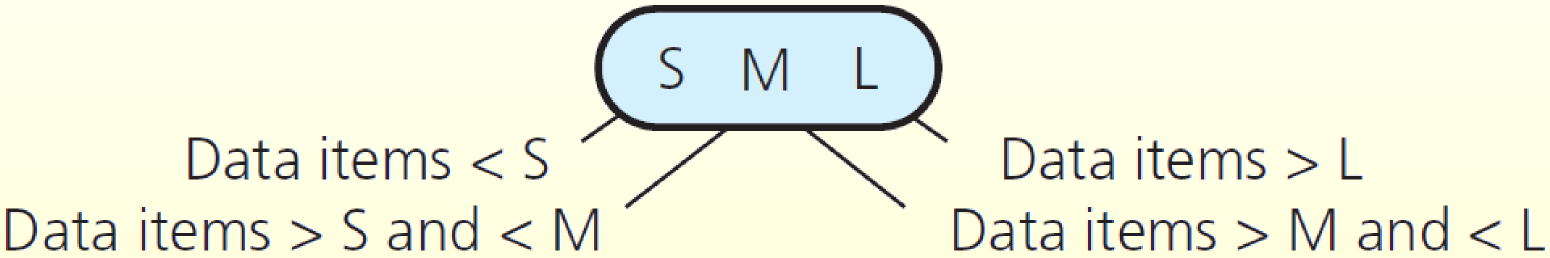
Deleting Data from a 2-3 Tree



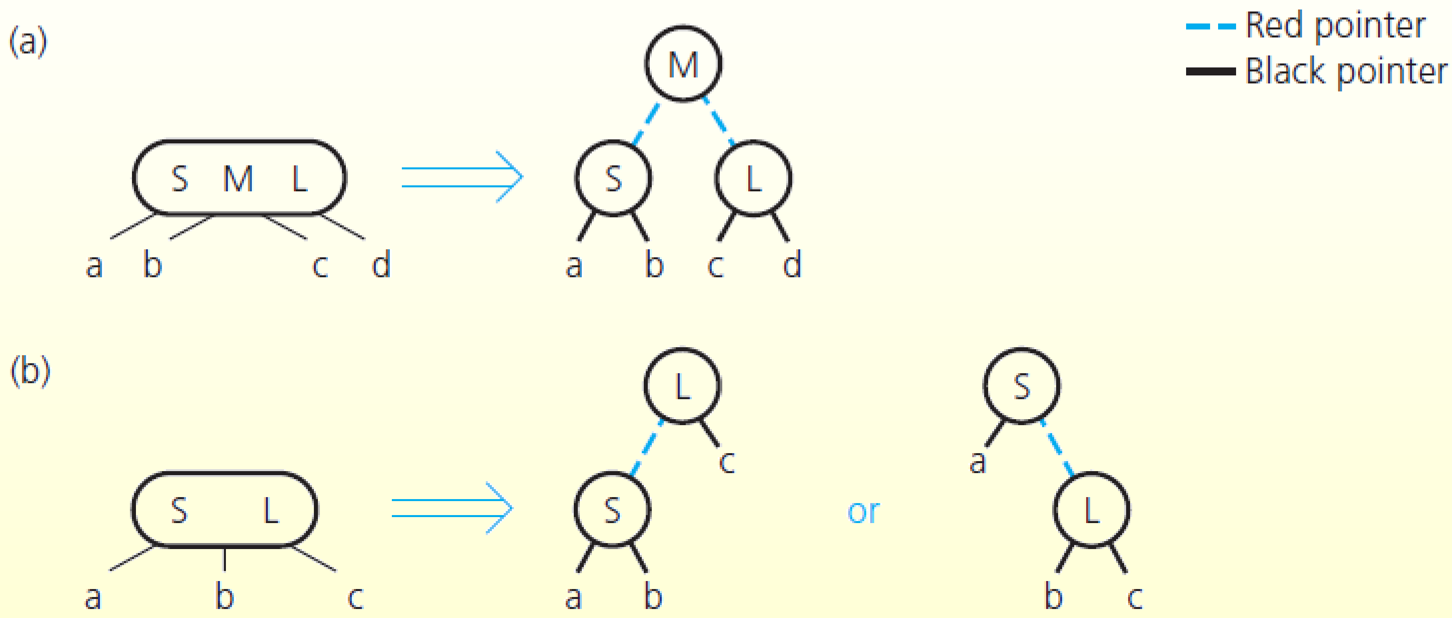
2-3-4 Trees



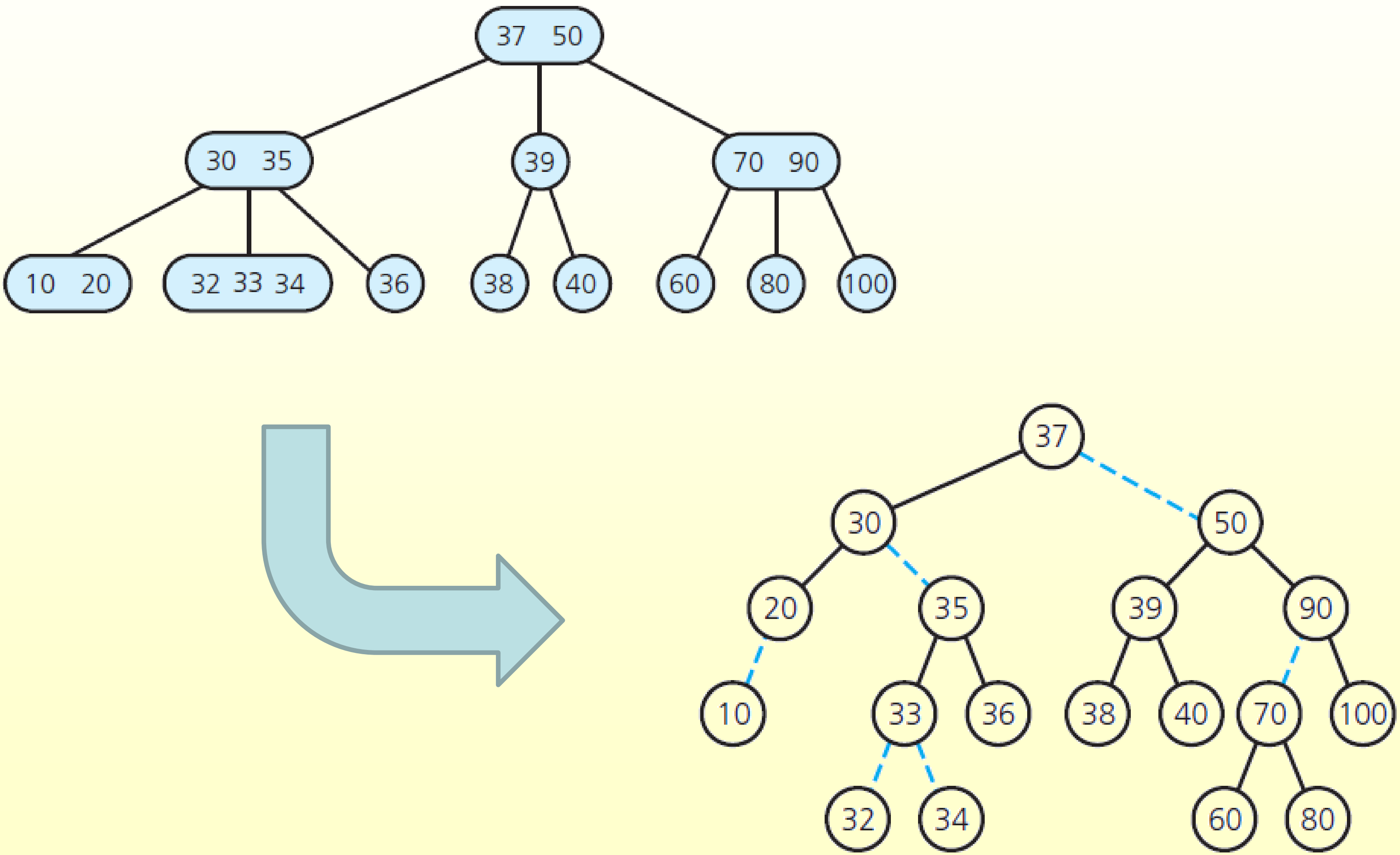
2-3-4 Trees


S M L
Data items < S
Data items > S and < M
Data items > M and < L
Data items > L

Red-Black Trees = 2-3-4 Trees



Red-Black Trees



AVL Trees

- Named for inventors, Adel’son-Vel’skii and Landis
- A balanced binary search tree
  - any node in an AVL tree has left and right subtrees whose heights differ by more than 1
- Has guaranteed O(lg n) height
- Not as efficient as red-black trees
- May require Ω(lg n) rotations to fix