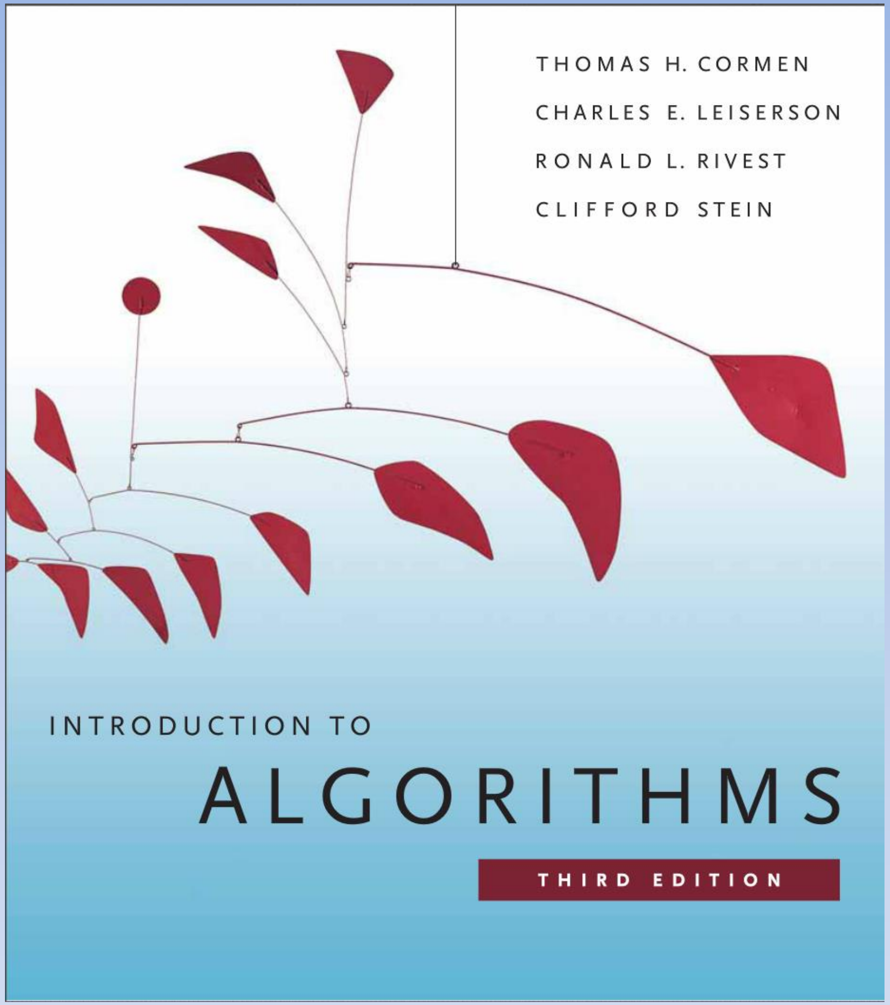


174 : Chapter 13

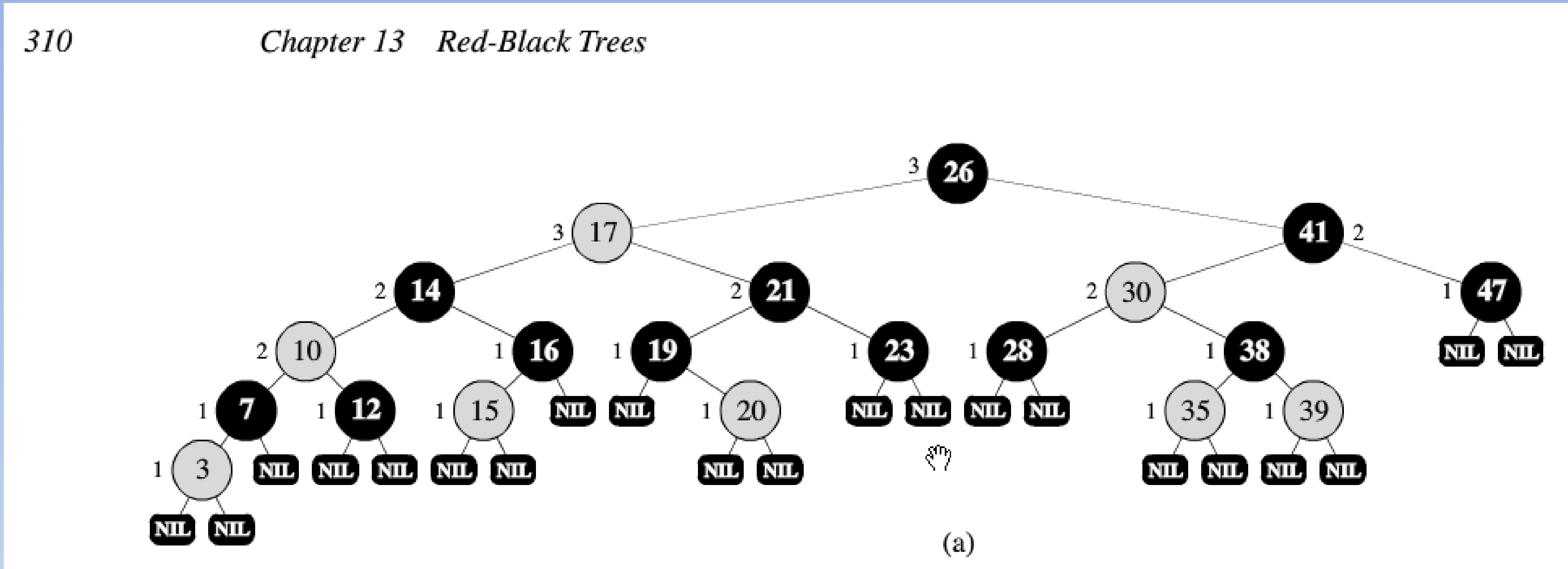
Red-Black Trees

Binary Search Trees

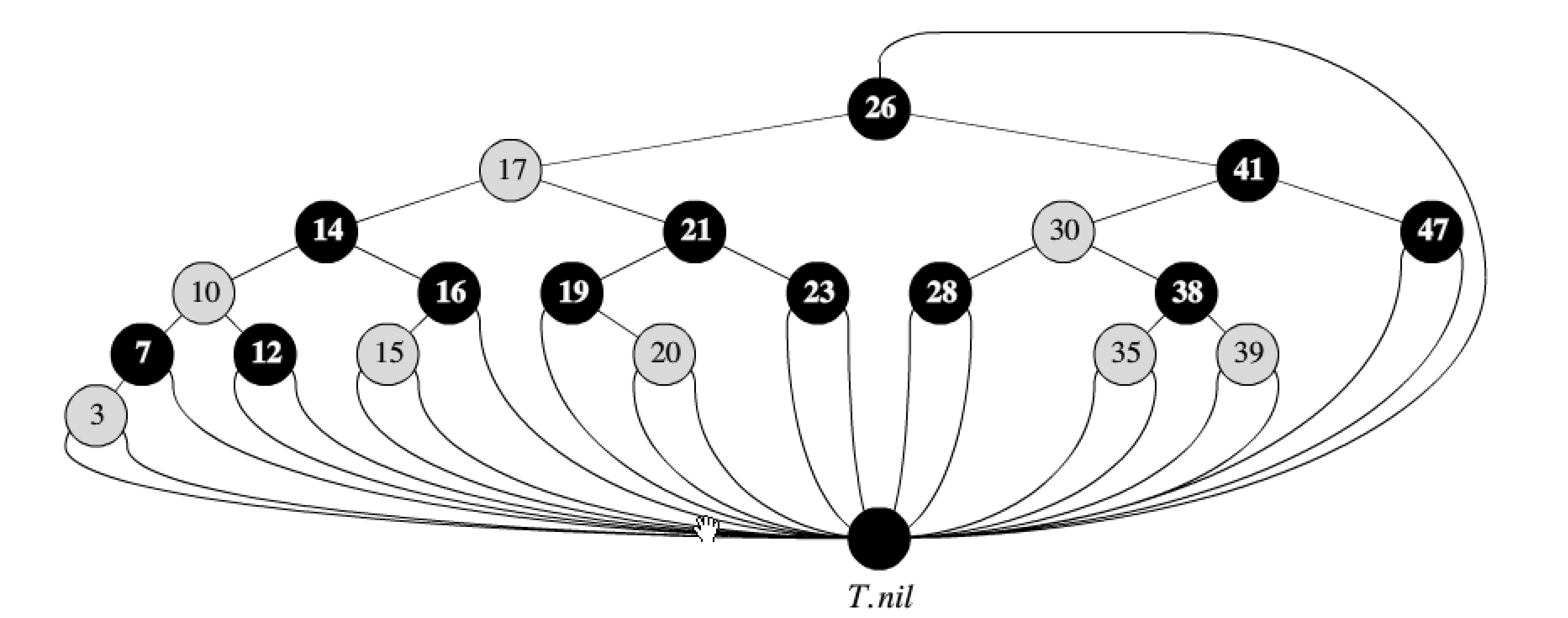
- Great average performance
  - Searching in O(lg n)
- But worst-case performance is much worse!
  - Searching in O(n) !!
- Need to prevent tree from structures generating worst case performance!

Red-Black Properties

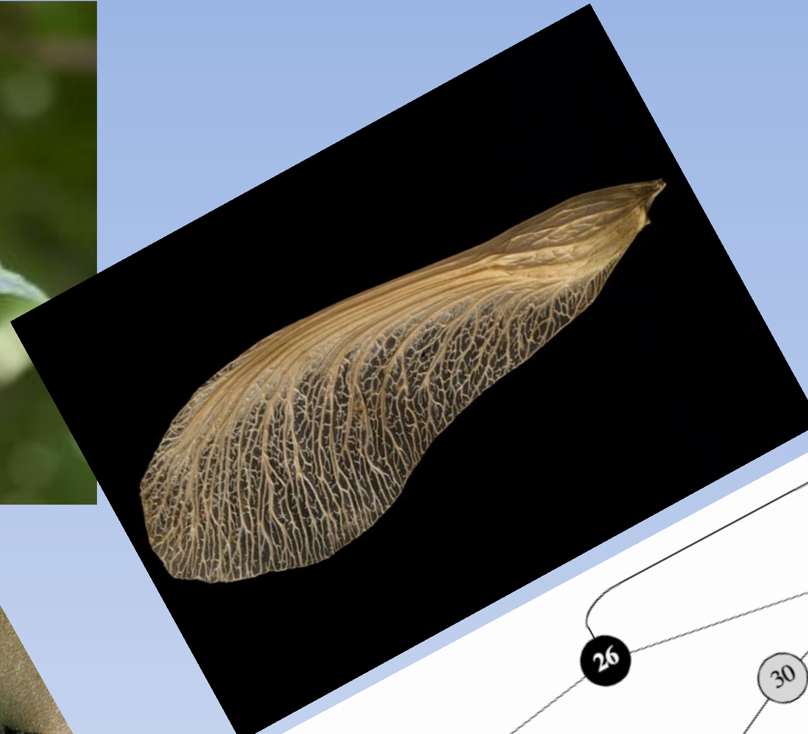
1. Every node is either red or black.
2. The root is black.
3. Every leaf (T.*nil*) is black.
4. If a node is red, then both its children are black.
   – No two reds in a row on a simple path from the root to a leaf.
5. All simple paths from a node to descendant leaves contain the same number of black nodes.

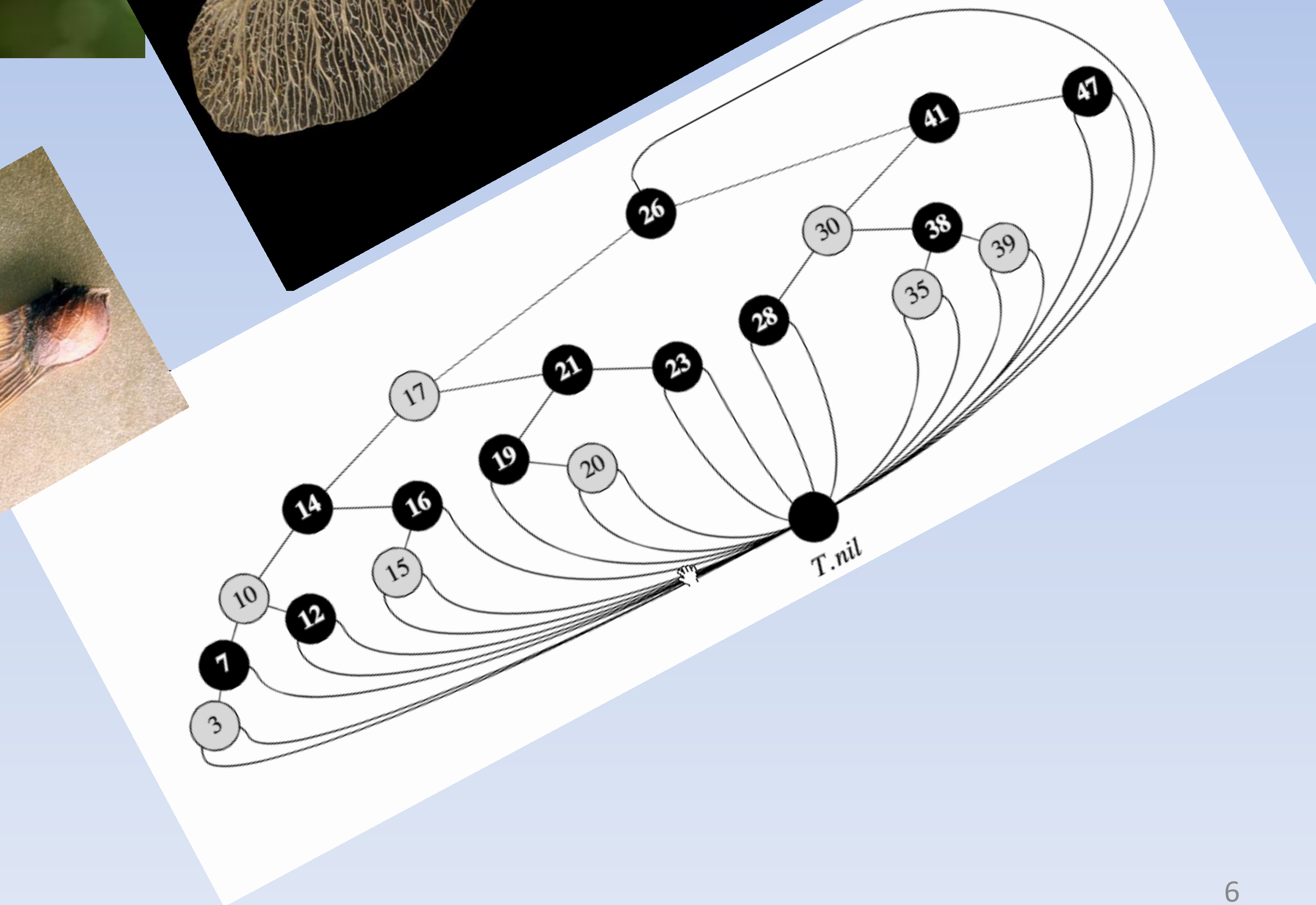


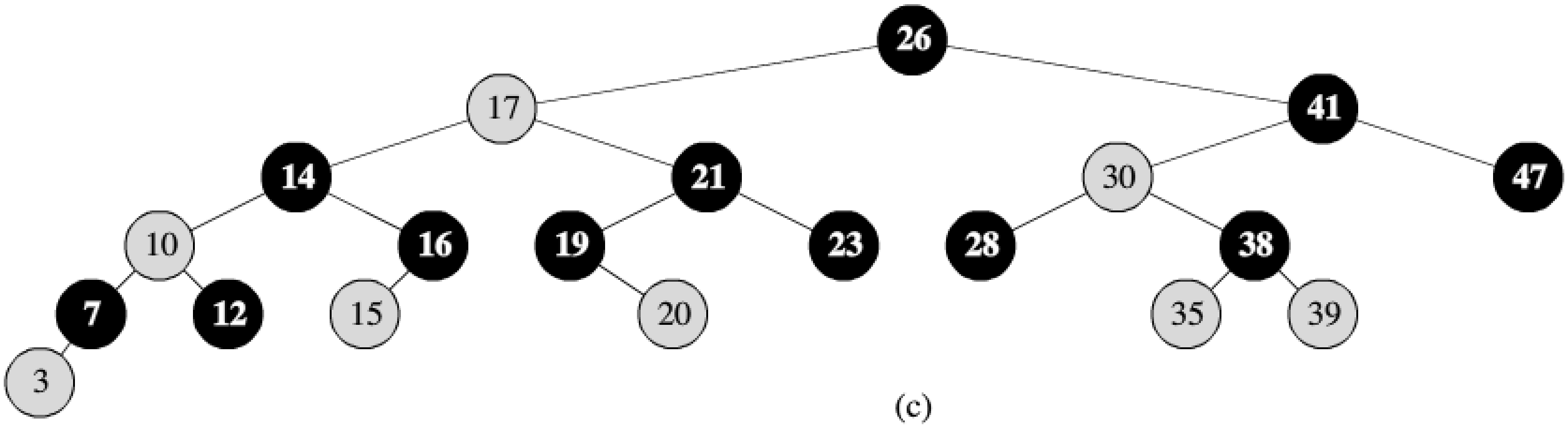
- Representation



- Implementation




3
7
10
12
14
15
16
17
19
20
21
23
26
28
30
35
38
39
41
47
T.nil



(c)

- Normally not drawn with nil!

Black-Height

- **black-height :**
  - number of black nodes on any simple path from, but not including, a node x down to a leaf of the node,
  - denoted bh(x).
- By property 5,
  - black-height is well defined, since all descending simple paths from the node have the same number of black nodes.
  - We define the black-height of a red-black tree to be the black-height of its root.

Why Red-Black Trees

- ***Claim***
  - Any node with height h has black-height >= h/2.
- ***Proof***
  - By property 4, <=h/2 nodes on the path from the node to a leaf are red.
  - Hence >= h/2 are black.

Why Red-Black Trees

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

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

5. All simple paths from a node to descendant leaves contain the same number of black nodes.

Red-Black Trees

- ***Proof*** By induction on height of x.
- **Basis:** Height of x = 0 => x is a leaf => bh(x) = 0. The subtree rooted at x has 0 internal nodes: $2^0 - 1 = 0$.
- **Inductive step:**
- Let the height of x be h and bh(x)=b.
  - Any child of x has height h - 1 and black-height either b (if the child is red) or b - 1 (if the child is black).
  - By the inductive hypothesis, each child has at least $2^{bh(x)-1} - 1$ internal nodes.
- Thus, the subtree rooted at x contains
  - $2 * (2^{bh(x)-1} – 1)+1 = 2^{bh(x)} - 1$ internal nodes. (The +1 is for x itself.)

Red-Black Trees

- ***Lemma***
- A red-black tree with n internal nodes has height of at most 2lg(n+1)
- ***Proof*** Let h and b be the height and black-height of the root, respectively.
- By the previous two claims,
  - $n >= 2^b - 1 >= 2^{h/2} - 1$
- Adding 1 to both sides and then taking logs gives
  - lg(n+1) >= h/2,
- which implies that
  - h <= 2lg(n+1).

Operations on Red-Black Trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O(height) time.
- Thus, they take O(lg n) time on red-black trees.

Operations on Red-Black Trees

- Insertions and Deletion are not so easy
- If we insert, what color to make the new node?
  - Red? Might violate property 4
  - Black? Might violate property 5

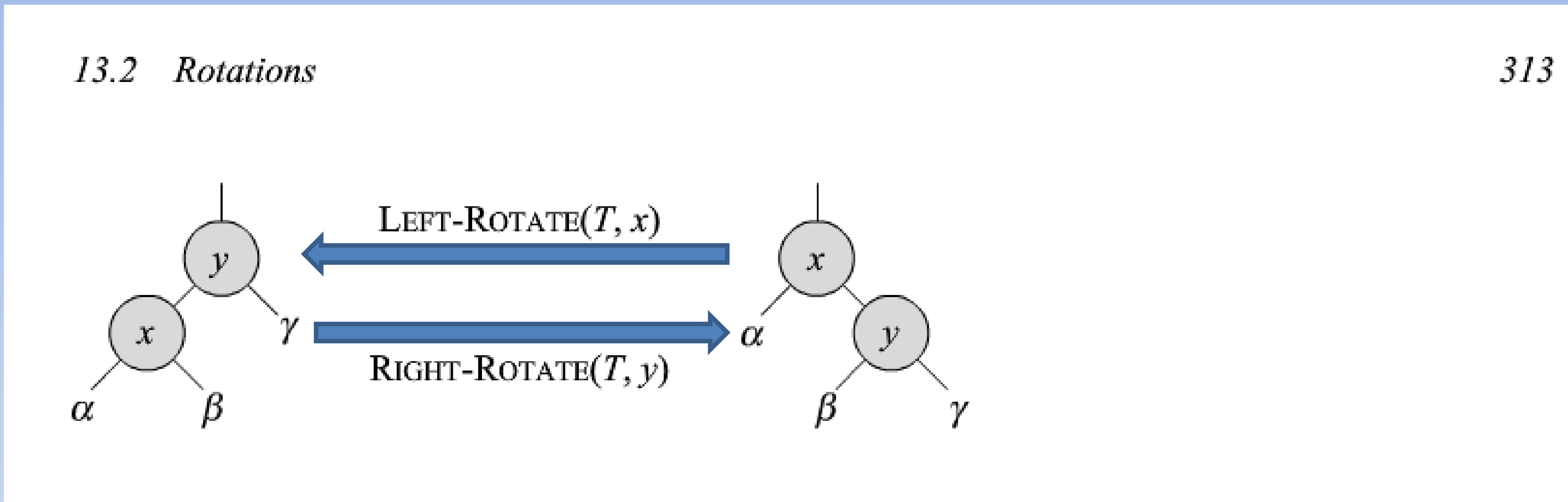
Operations on Red-Black Trees

- When deleting: what color node that was removed??
  - Red?
    - OK,
      - nO black-heights changed,
      - NO two red nodes in a row.
    - Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
  - Black?
    - Could cause there to be two reds in a row (violating property 4) and can also cause a violation of property 5.
    - Could also cause a violation of property 2, if the removed node was the root and its child – which becomes the new root – was red.

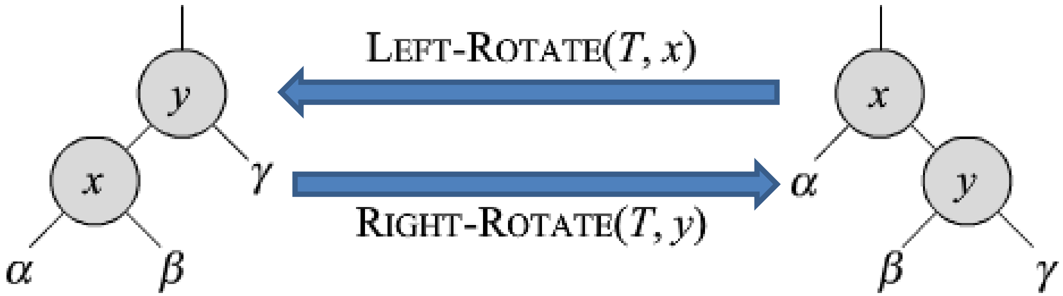
Rotations

- The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take O(lgn) time.
- Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1.
- To restore these properties, we must
  - change the colors of some of the nodes in the tree
  - and also change the pointer structure.
- Pointer structure changed through ROTATION,
  - which is a local operation in a search tree that preserves the binary-search-tree property.

Rotations

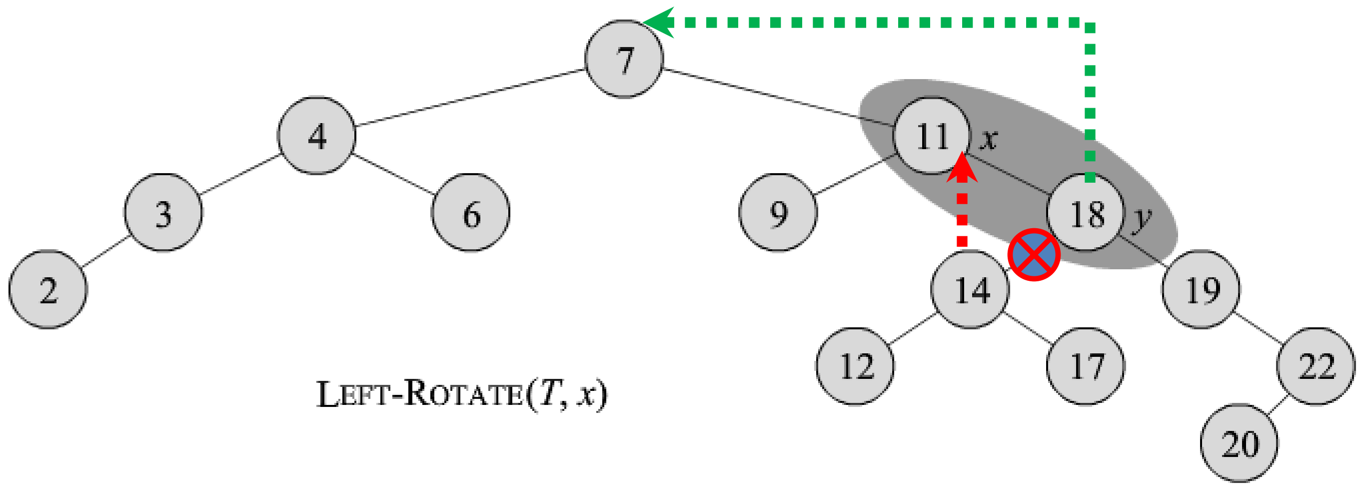


- Left-Rotate:
  - Shifts weight to the left.
- Right-Rotate:
  - Shifts weight to the right!



LEFT-ROTATE(T, x)

```
y = x.right                   // set y
x.right = y.left              // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                     // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                    // put x on y's left
x.p = y
```



LEFT-ROTATE(T, x)

LEFT-ROTATE(T, x)

```
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                     // put x on y's left
x.p = y
```

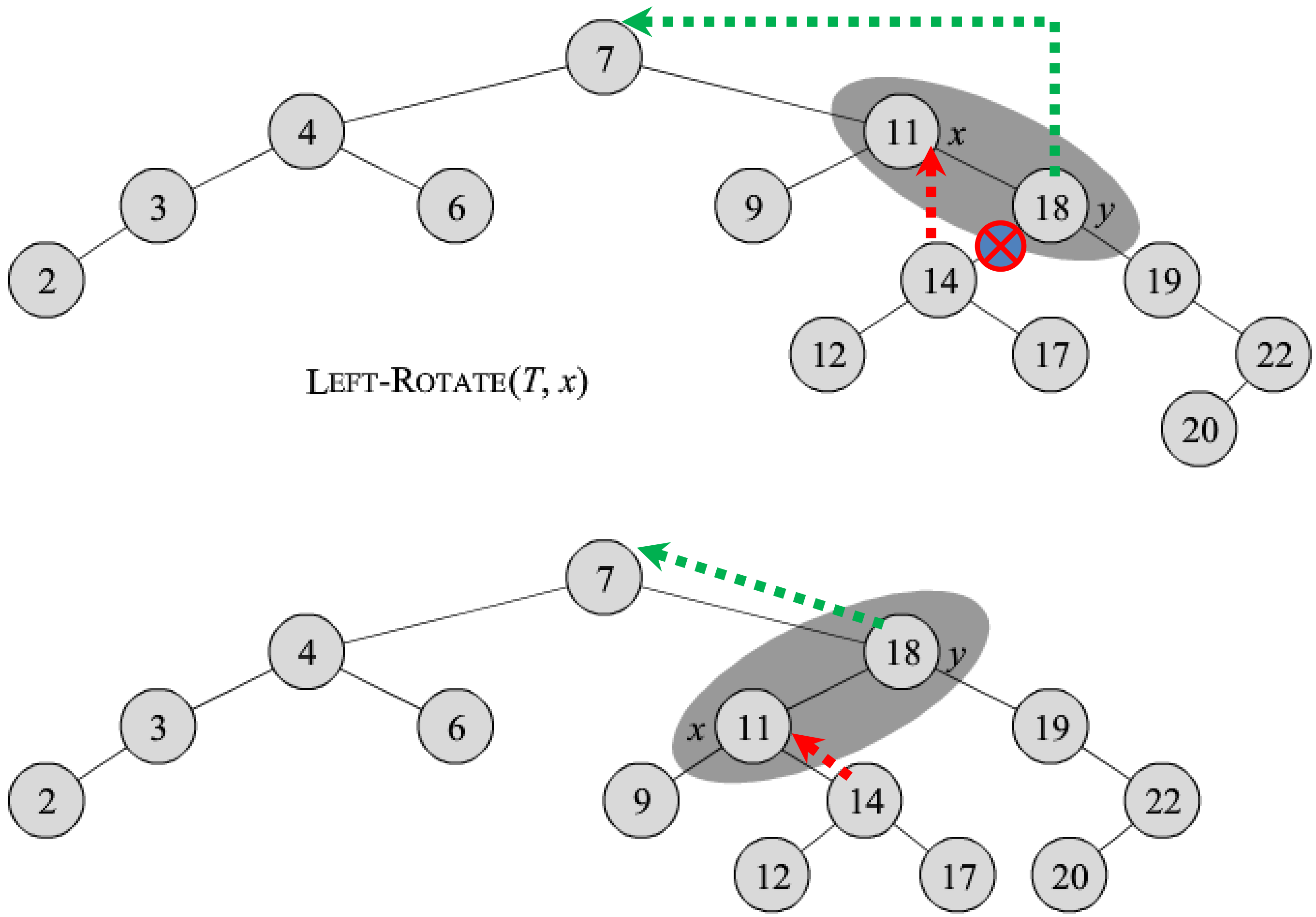


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Rotations

- ***Time***
  - O(1) for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.
- ***Notes***
  - Rotation is a very basic operation, also used in AVL trees and splay trees.
  - Some books talk of rotating on an edge rather than on a node.

Insertions
Start w/ BST Insertion

```
RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
```

Insertions
Start w/ BST Insertion

- RB-Insert ends by coloring the new node z RED
- Then it calls RB-Insert-Fixup because we could have violated a red-black property

Which Properties might be Violated?

- Property 1: OK
- Property 2:
  - If z is the root, then there's a violation.
  - Otherwise, OK.
- Property 3: OK.
- Property 4.
  - If z.*p* is red, there's a violation: both z and z.*p* are red.
- Property 5: OK.

Remove the violation by calling RB-INSERT-FIXUP:

RB-Insert-Fixup(T, z)

```
RB-INSERT-FIXUP(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK                          // case 1
              y.color = BLACK                            // case 1
              z.p.p.color = RED                          // case 1
              z = z.p.p                                  // case 1
          else if z == z.p.right
                  z = z.p                                // case 2
                  LEFT-ROTATE(T, z)                      // case 2
              z.p.color = BLACK                          // case 3
              z.p.p.color = RED                          // case 3
              RIGHT-ROTATE(T, z.p.p)                     // case 3
      else (same as then clause with "right" and "left" exchanged)
  T.root.color = BLACK
```

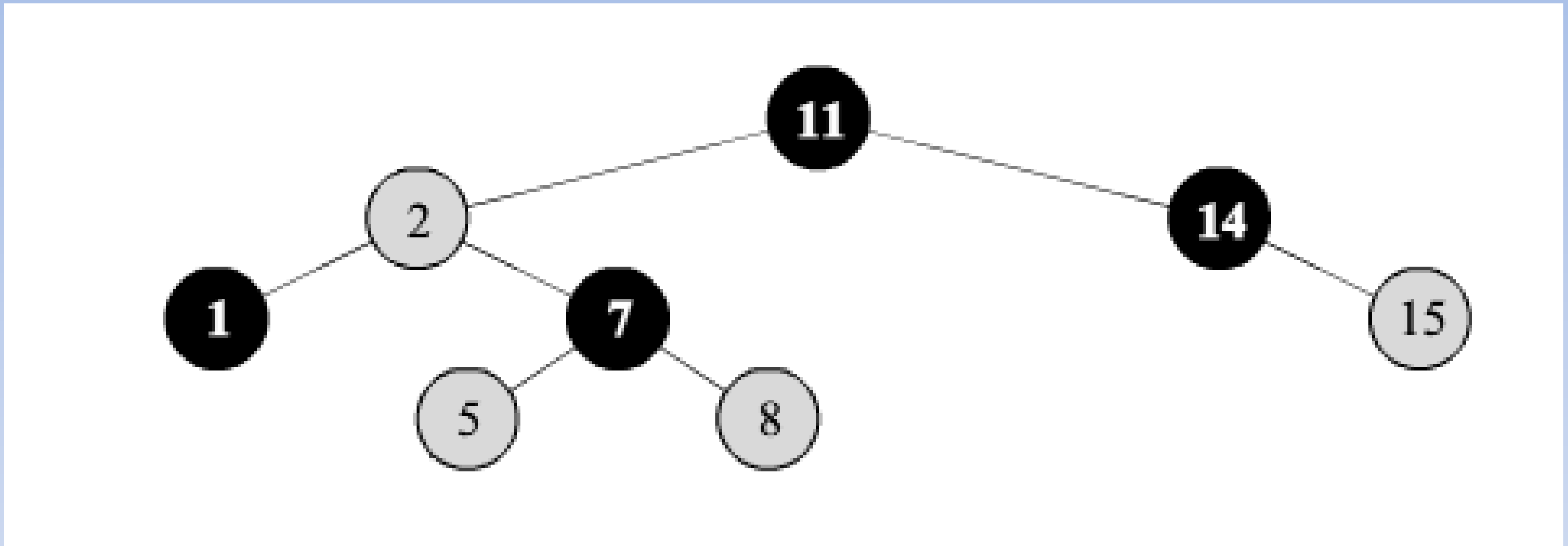
Loop Invariant

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

Loop Invariant

- **Initialization:** true from the insert.
- **Termination:** The loop terminates because z.*p* is black. Hence, property 4 is OK.
  - Only property 2 might be violated, and the last line fixes it.
- **Maintenance:** We drop out when z is the root (since then z.*p* is the sentinel T.*nil*, 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.
  - The cases are not mutually exclusive.
  - We'll consider cases in which z.*p* is a left child.
  - Let y be z's uncle (z.*p*'s sibling).

Example: Insert (4)



- Insert 4
- Initialize Color=Red

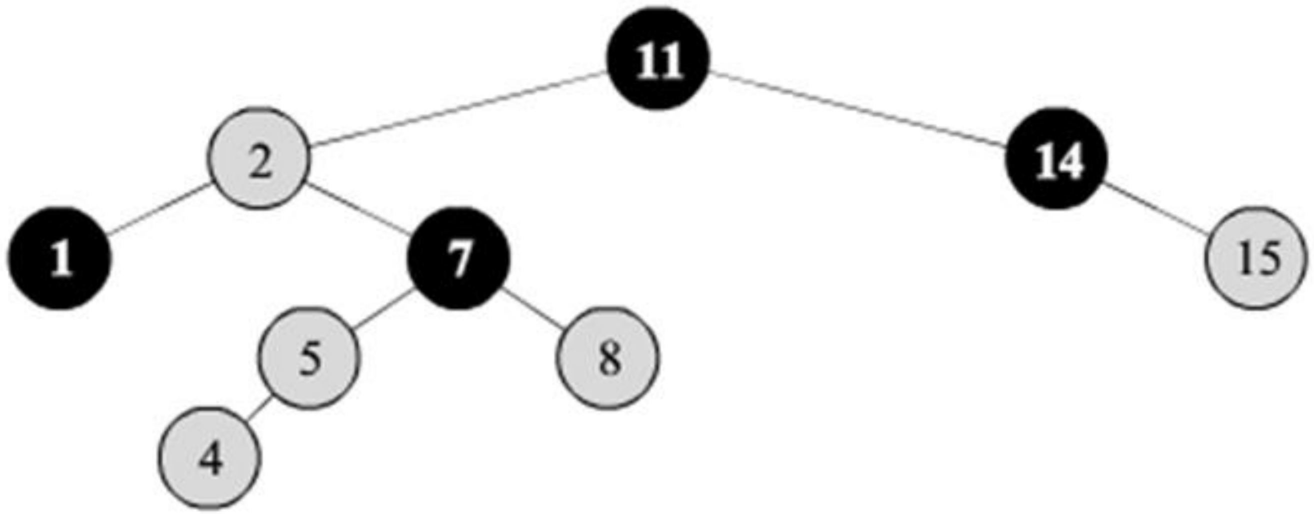
Insertions
Start w/ BST Insertion

- RB-Insert ends by coloring the new node z RED
- Then it calls RB-Insert-Fixup because we could have violated a red-black property

Insertions
Start w/ BST Insertion

```
RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
```

Example: (4) Inserted



- 4 Inserted
- Color Initialized Red

Red-Black Properties

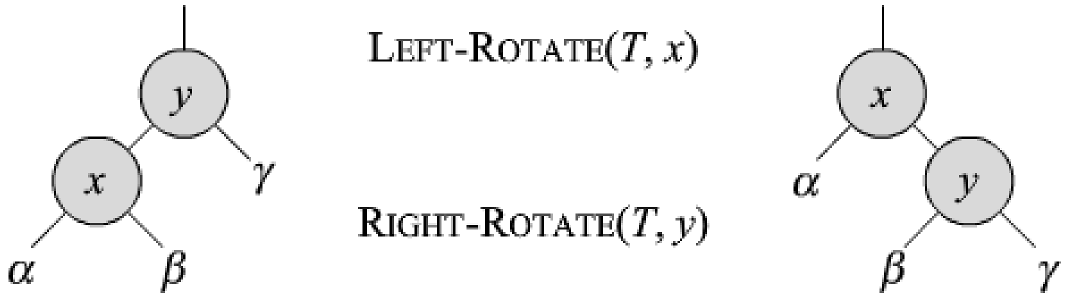
1. Every node is either red or black.
2. The root is black.
3. Every leaf (T.*nil*) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Which Properties might be Violated?

- Property 1: OK
- Property 2:
  - If z is the root, then there's a violation.
  - Otherwise, OK.
- Property 3: OK.
- Property 4.
  - If z.*p* is red, there's a violation: both z and z,*p* are red.
- Property 5: OK.
- Remove the violation by calling RB-INSERT-FIXUP:

RB-Insert-Fixup(T, z)

```
RB-INSERT-FIXUP(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK                          // case 1
              y.color = BLACK                            // case 1
              z.p.p.color = RED                          // case 1
              z = z.p.p                                  // case 1
          else if z == z.p.right
                  z = z.p                                // case 2
                  LEFT-ROTATE(T, z)                      // case 2
              z.p.color = BLACK                          // case 3
              z.p.p.color = RED                          // case 3
              RIGHT-ROTATE(T, z.p.p)                     // case 3
      else (same as then clause with "right" and "left" exchanged)
  T.root.color = BLACK
```

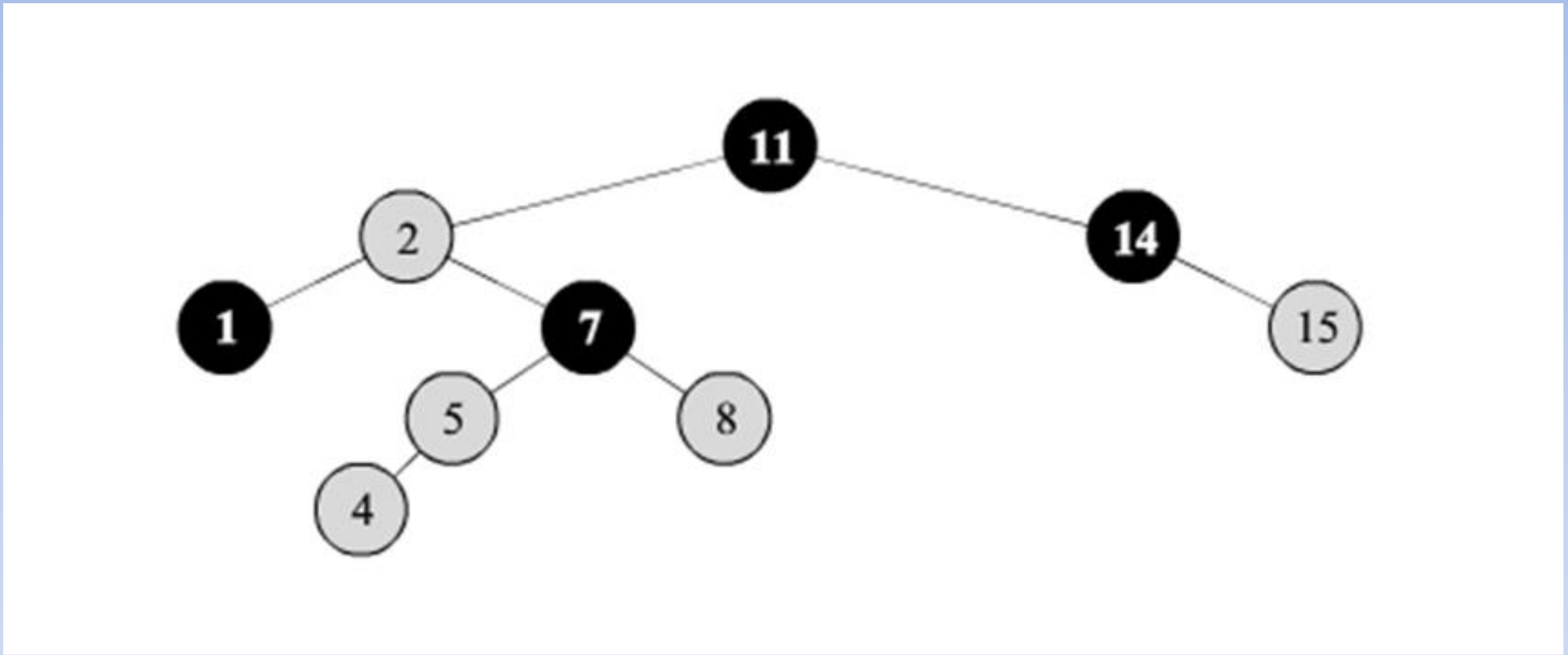



```
LEFT-ROTATE(T, x)
y = x.right                      // set y
x.right = y.left                 // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                        // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                       // put x on y's left
x.p = y
```

Loop Invariant

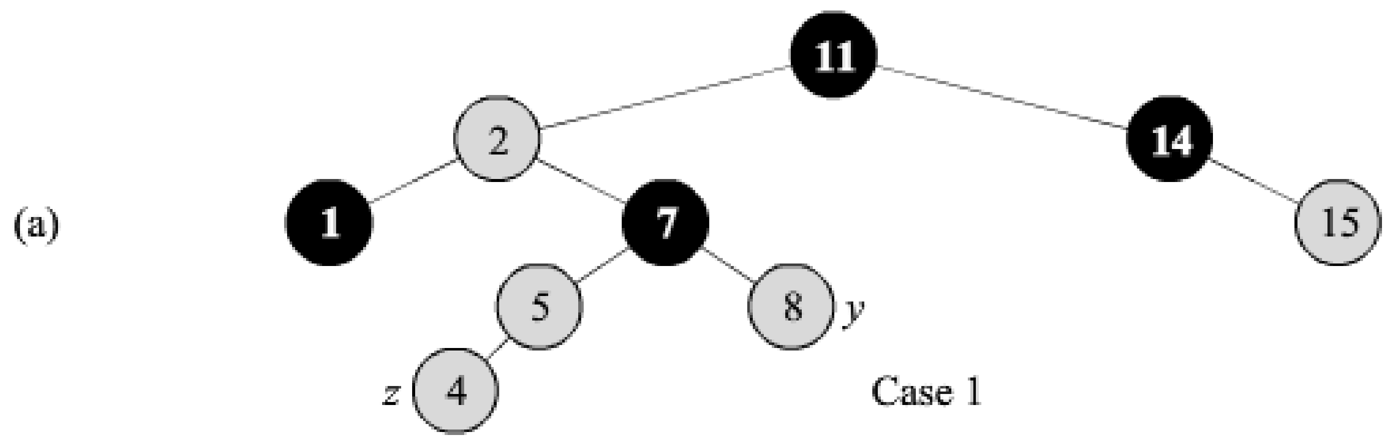
- At the start of each iteration of the **while** loop,
- z is red.
- There is at most one red-black violation:
  - Property 2: z is a red root, or
  - Property 4: z and z.*p* are both red.

Example: (4) Inserted



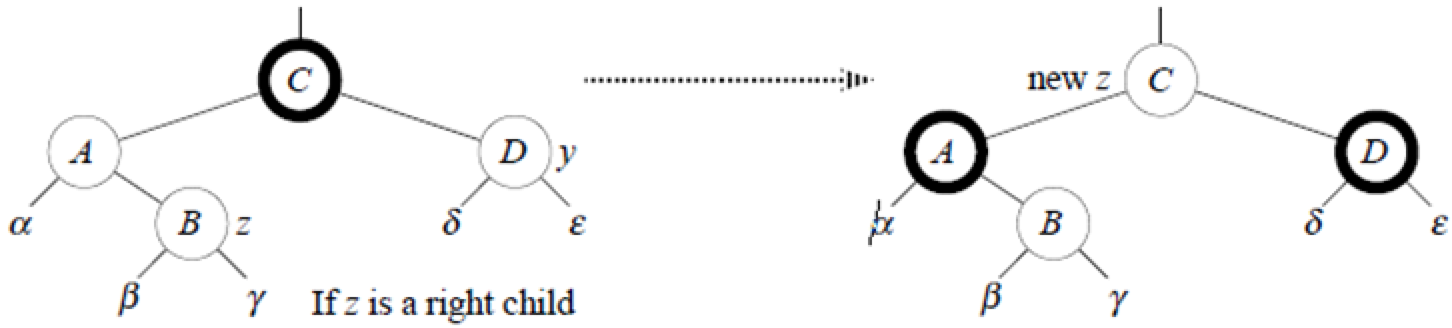
- 4 Inserted
- Color Initialized Red

Case 1: y is red



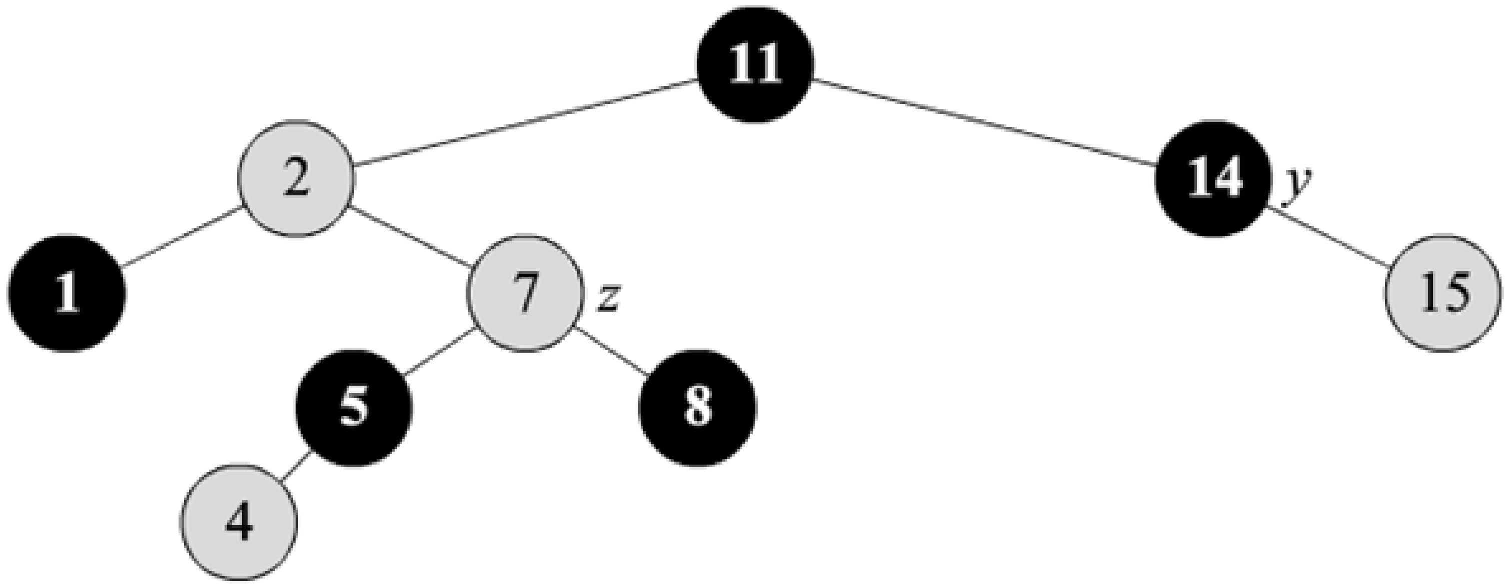
- z inserted
- y is z’s red uncle!
- Plan:
  - Turn z’s grandfather’s children black!
  - Turn z’s grandfather red – recurse from grandfather

Case 1: y is red

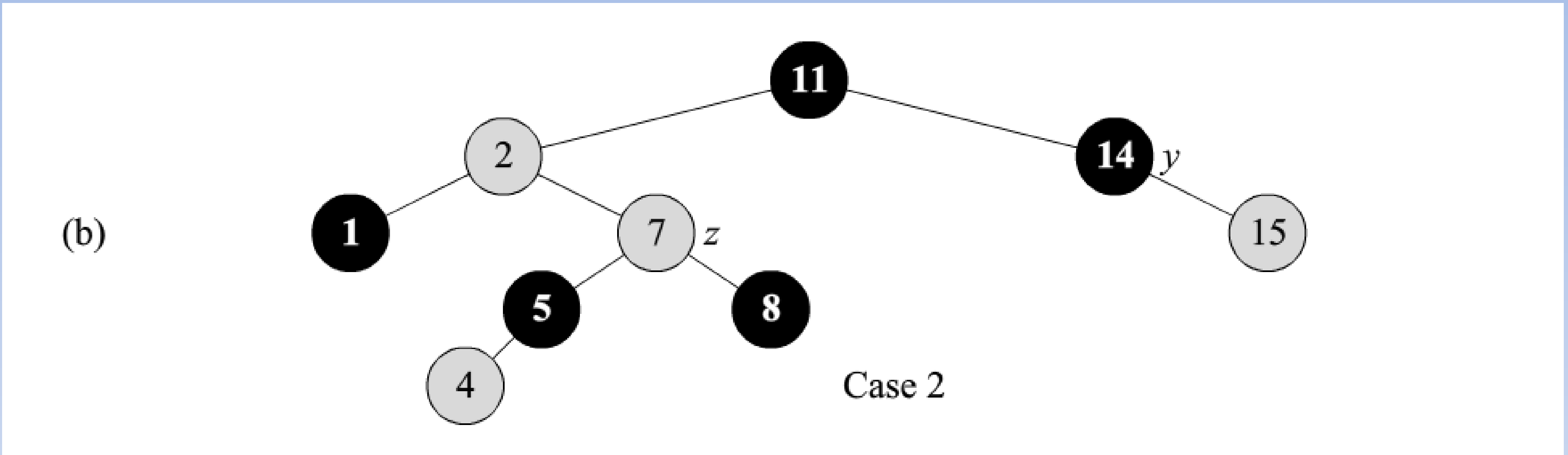


- $z.p.p$ (z's grandparent) must be black, since z and $z.p$ are both red and there are no other violations of property 4.
- Make $z.p$ and y black $\Rightarrow$ now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red $\Rightarrow$ restores property 5.
- The next iteration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

1 Iteration Completed

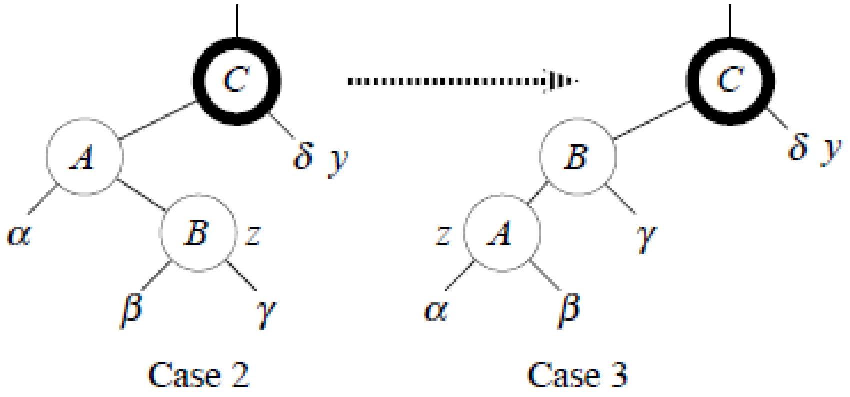


Now w/ Case 2:
y is Black, z is Right Child



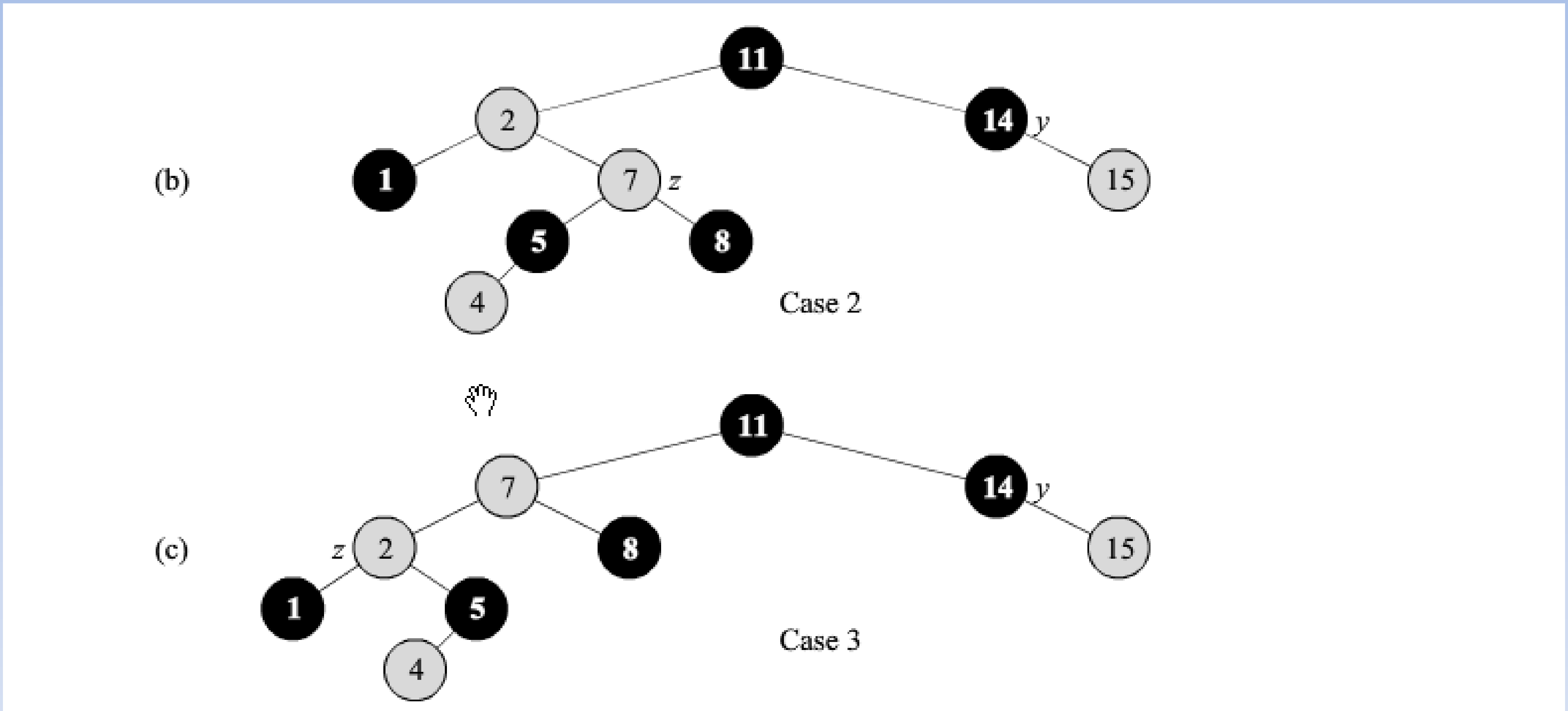
- z’s uncle is black!
  - Need to adjust!

Case 2: y is black, z is a right child

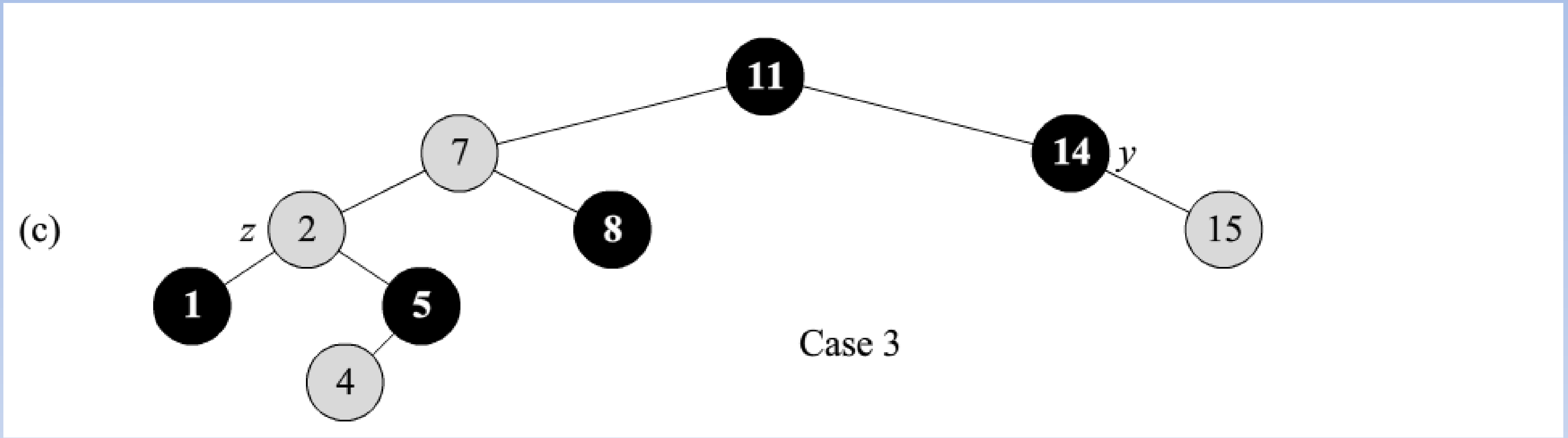


- Left rotate around $z.p \Rightarrow$ now z is a left child, and both z and $z.p$ are red.
- Takes us immediately to case 3.

Case 2:
y is Black, z is Right Child



Case 3: y is Black, z is Left Child

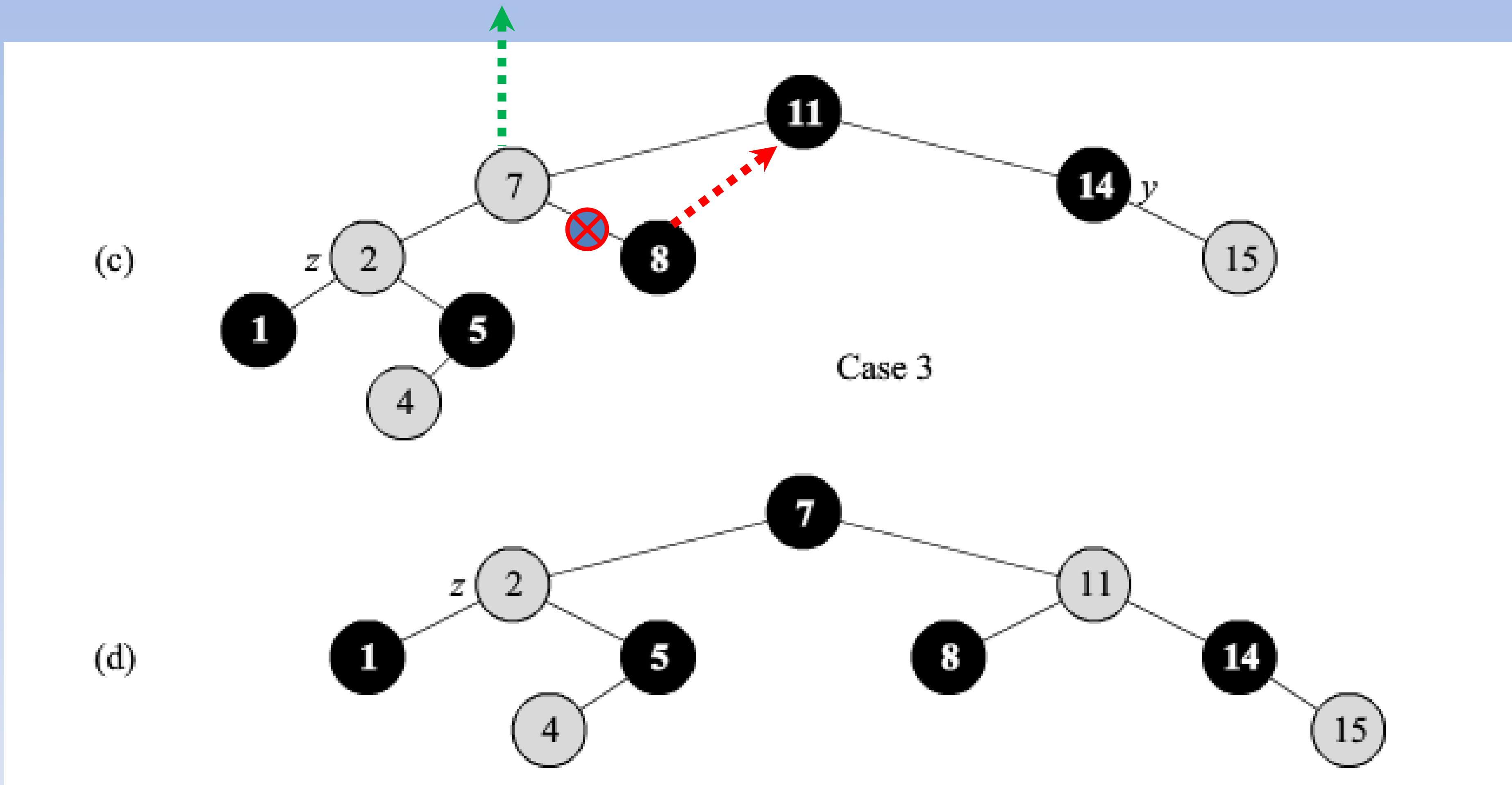


- z’s uncle is still black

Case 3

- **Case 3:** y is black, z is a left child
- Make z.*p* black and z.*p*.*p* red.
- Then right rotate on z.*p*.*p*.
- No longer have 2 reds in a row.
- Z.*p* is now black => no more iterations.

Resulting Legal Red-Black Tree



Loop Invariant

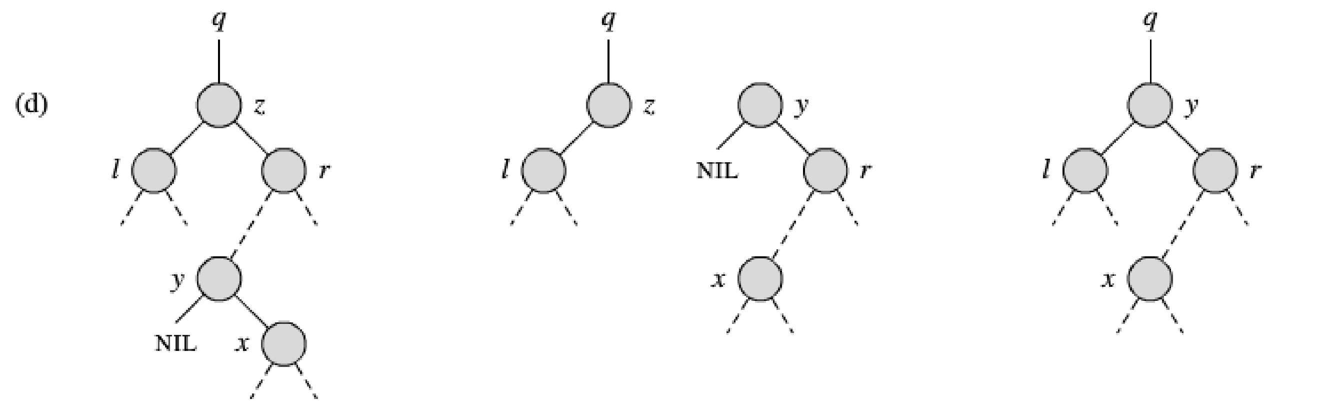
- **Initialization:** true from at start from insert.
- **Termination:** The loop terminates because z.*p* is black. Hence, property 4 is OK.
  - Only property 2 might be violated, and the last line fixes it.
- **Maintenance:** We drop out when z is the root (since then z.*p* is the sentinel T.*nil*, 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.
  - The cases are not mutually exclusive.
  - We'll consider cases in which z.*p* is a left child.
  - Let y be z's uncle (z.*p*'s sibling).

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 is either the last one or it moves z up 2 levels.
  - O(lg n) levels => O(lg n) time.
  - Also note that there are at most 2 rotations overall.
- Thus, insertion into a red-black tree takes O(lg n) time.

Deletion

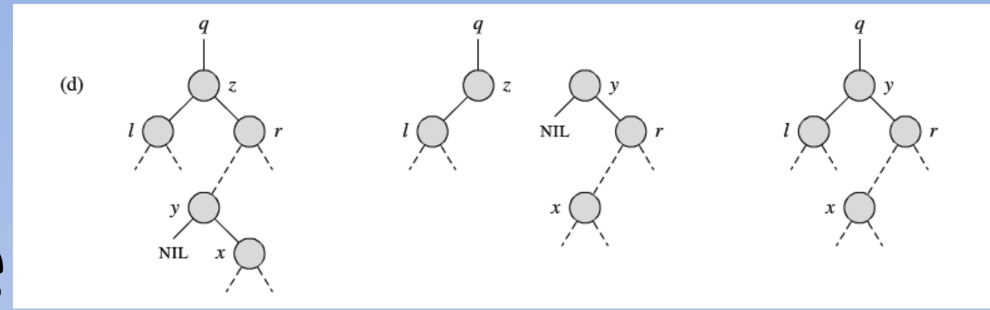
Deletion




```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

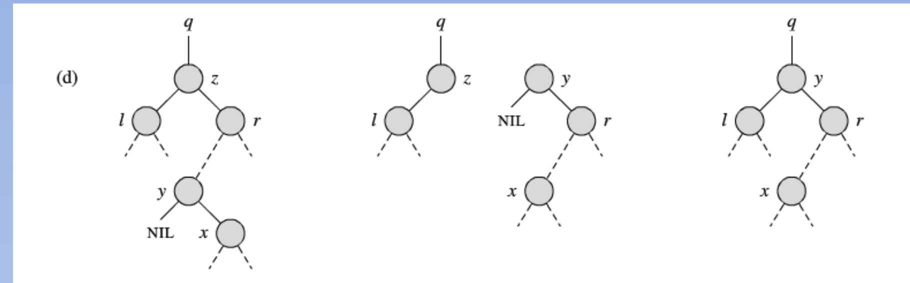
Deletion w/ Red-Black Tree




```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

Deletion w/ Red-Black Tree




```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion w/ Red-Black Tree

- y is node
  - removed
  - OR moved (successor).
- Line 1 sets y to point to node z when z has fewer than two children
- When z has two children, line 9 sets y to point to z's successor,
  - y will move into z's position in the tree.

```
RB-DELETE(T, z)
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
7      x = z.left
8      RB-TRANSPLANT(T, z, z.left)
9  else y = TREE-MINIMUM(z.right)
10     y-original-color = y.color
11     x = y.right
12     if y.p == z
13         x.p = y
14     else RB-TRANSPLANT(T, y, y.right)
15         y.right = z.right
16         y.right.p = y
17     RB-TRANSPLANT(T, z, y)
18     y.left = z.left
19     y.left.p = y
20     y.color = z.color
21 if y-original-color == BLACK
22     RB-DELETE-FIXUP(T, x)
```

Deletion w/ Red-Black Tree

- y-original-color stores y's color before any changes occur.
  - Lines 2 and 10 set this variable.
- When z has two children, then y $\neq$z and node y moves into node z's original position in the red-black tree;
  - line 20 gives y the same color as z.
- Y's original color is used to determine if Fixup is needed!

```
RB-DELETE(T, z)
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Deletion w/ Red-Black Tree

- x replaces y in tree.
- The assignments in lines 4, 7, and 11 set x to point to either
  - y's only child or,
  - T.nil
- x is equal to the third parameter in RB-Transplant for all but last case.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Deletion: y was red – No Problem

- No black-heights changed.
- No red nodes have been made adjacent.
  - if y was not z's right child,
    - then y's original right child x replaces y in the tree.
    - If y is red, x must be black,
    - so replacing y by x cannot cause two red nodes to become adjacent.
- y could not have been root

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion w/ Red-Black Tree

- If node y was black, Fixup may be needed.

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

RB-Delete-Fixup – We Deleted Black

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                              // case 1
            x.p.color = RED                              // case 1
            LEFT-ROTATE(T, x.p)                          // case 1
            w = x.p.right                                // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                // case 2
            x = x.p                                      // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                     // case 3
                w.color = RED                            // case 3
                RIGHT-ROTATE(T, w)                       // case 3
                w = x.p.right                            // 3
            w.color = x.p.color                          // ase 4
            x.p.color = BLACK                            // case 4
            w.right.color = BLACK                        // case 4
            LEFT-ROTATE(T, x                             // case 4
            x = T.ro                                     // case 4
    else (same as ther         with "right" and "left" exchanged)
x.color = BLACK
```

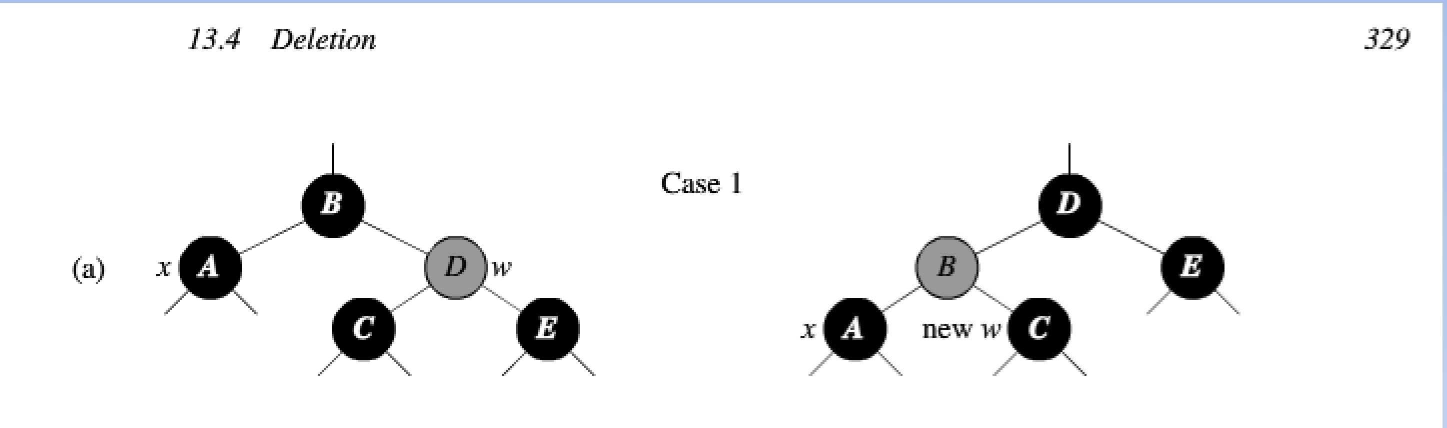
Once x is red – turn it black

RB-Delete-Fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

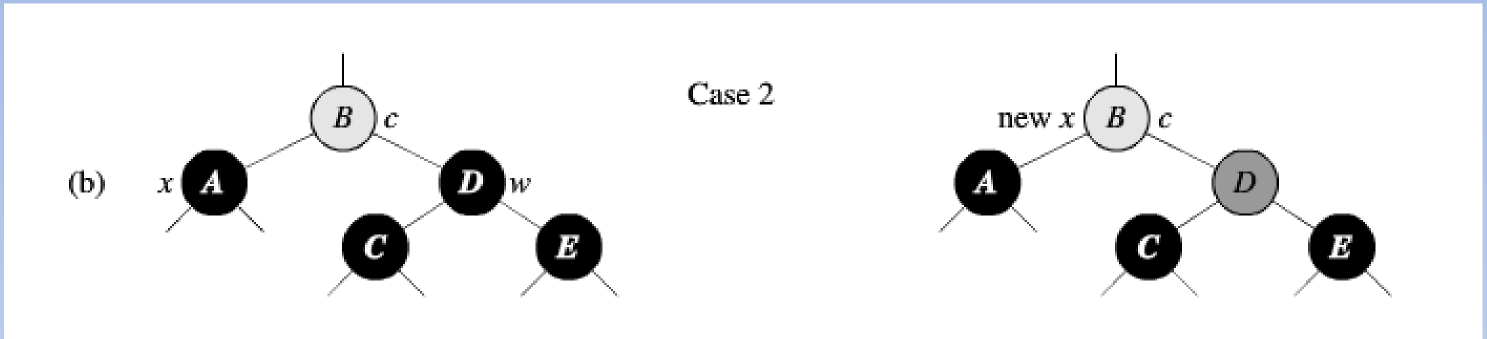
w is x's sibling

Case 1:
x's sibling (w) is red



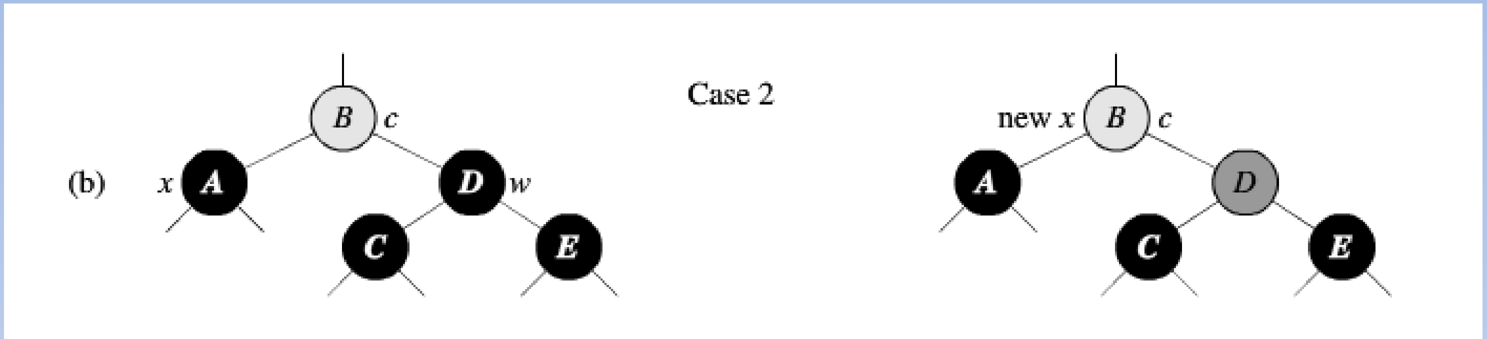
- switch colors of x.p & w
- Left rotate with their parent.
- Since both of w's children must be black
  - Case 1 is now 2, 3 or 4

Cases 2, 3, 4



- Case 2, 3, & 4 have x’s sibling w as black with
  - Case 2: 2 black children
  - Case 3: Left red child
  - Case 4: Right red child

Case: 2

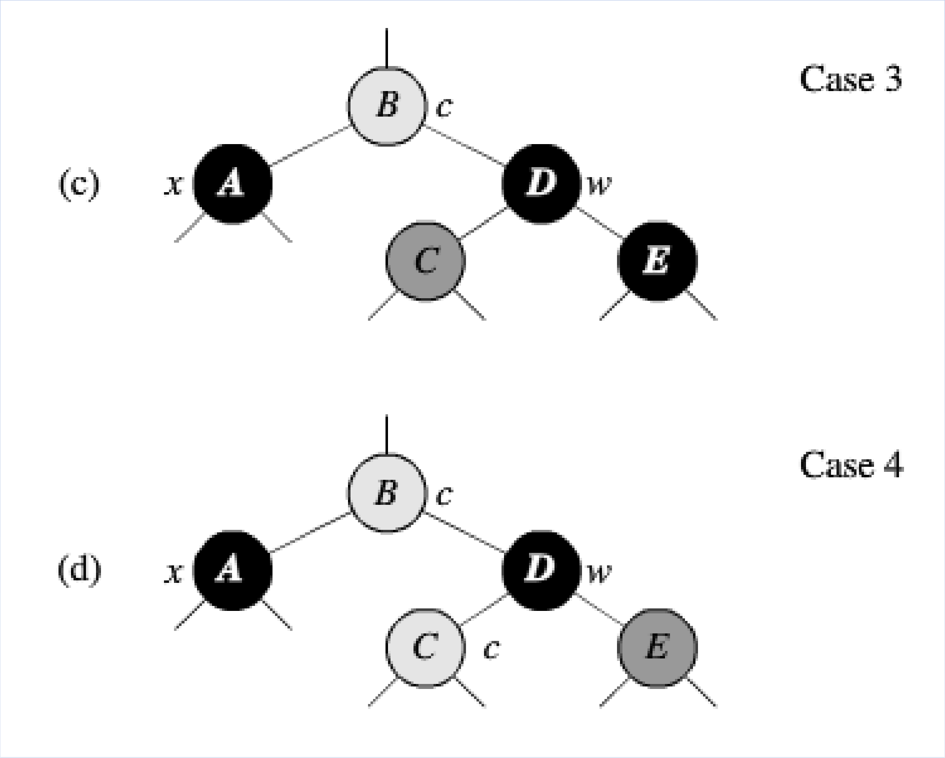


- x's sibling is black w/ 2 black children
- Since a black was deleted from x we can delete a black from d
  - Turn d RED
- Recurse now @ b since it has lost some black height.

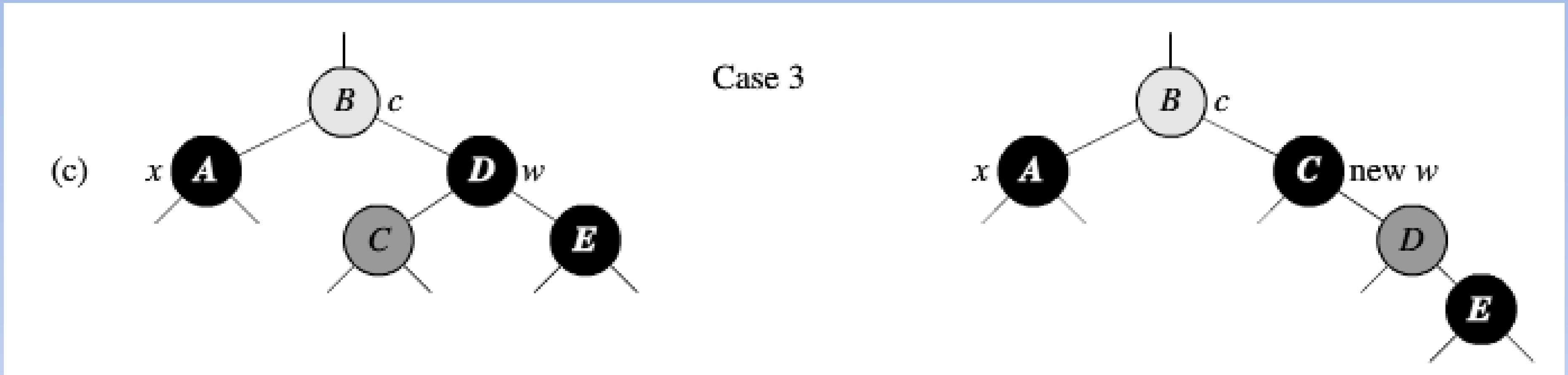
```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // case 2
            x = x.p                                           // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // case 3
                w.color = RED                                 // case 3
                RIGHT-ROTATE(T, w)                            // case 3
                w = x.p.right                                 // case 3
            w.color = x.p.color                               // case 4
            x.p.color = BLACK                                 // case 4
            w.right.color = BLACK                             // case 4
            LEFT-ROTATE(T, x.p)                               // case 4
            x = T.root                                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 3 & 4 are Symetric

- x’s sibling is black, but with one black child!
- Case 3: x’s sibling’s left child is red
- Case 4: x’s sibling’s right child is red

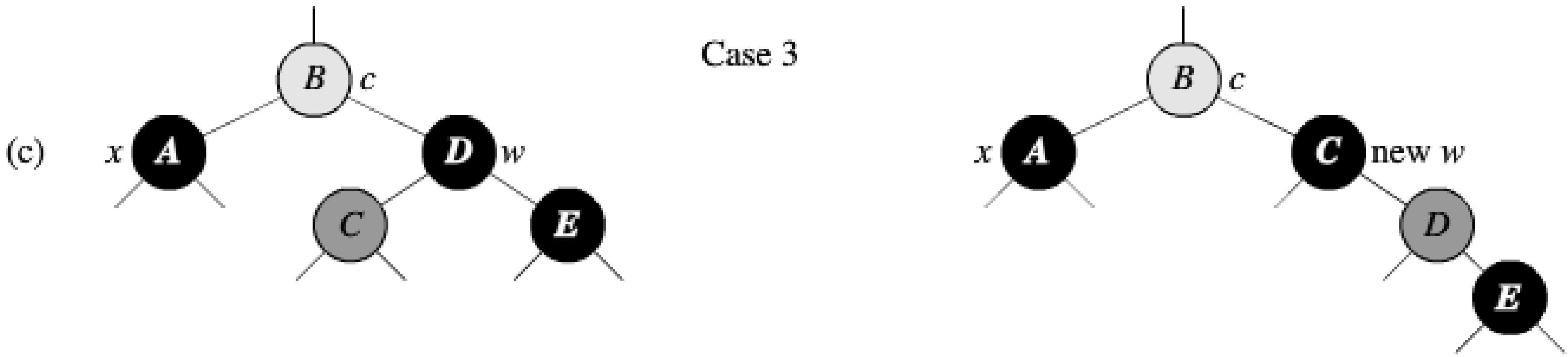


Case 3



- x's sibling is red with a left red child and black right child.
- PLAN: Transform this into CASE 4!
  - x's sibling is w
  - swap the colors between w and w.left (red child).
  - Right Rotate at w.
  - Now w has red right

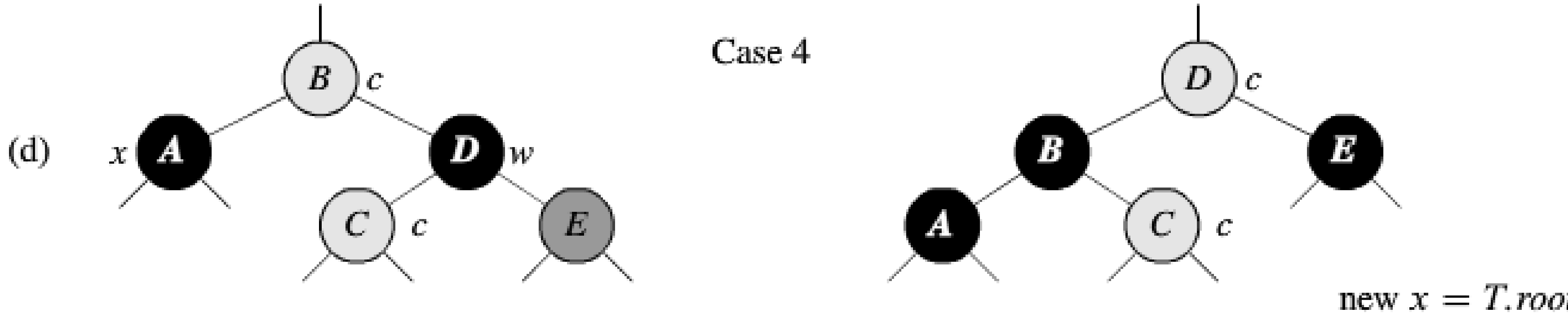
Case 3




```
else if w.right.color == BLACK
        w.left.color = BLACK                  // case 3
        w.color = RED                         // case 3
        RIGHT-ROTATE(T, w)                    // case 3
        w = x.p.right                         // case 3
    w.color = x.p.color                       // case 4
    x.p.color = BLACK                         // case 4
    w.right.color = BLACK                     // case 4
    LEFT-ROTATE(T, x.p)                       // case 4
    x = T.root                                // case 4
```

- x's siblir
  right ch
- PLAN: T
  - x's sib
  - swap
  - Right Rotate at w.

Case 4




```
else if w.right.color == BLACK
        w.left.color = BLACK                    // case 3
        w.color = RED                           // case 3
        RIGHT-ROTATE(T, w)                      // case 3
        w = x.p.right                           // case 3
    w.color = x.p.color                         // case 4
    x.p.color = BLACK                           // case 4
    w.right.color = BLACK                       // case 4
    LEFT-ROTATE(T, x.p)                         // case 4
    x = T.root                                  // case 4
```

- Node x’s sibling w is black and w’s right child is red.
- Color changes and left rotation on x.p, removes extra black on x
- x = T.root causes loop to terminate.

Analysis of RB-DELETE

- Without RB-Delete-Fixup work is O(lg n) since height of tree is lg n.
- RB-Delete-Fixup : Each operation does some constant amount of work
- RB-Delete-Fixup : Cases 1, 3, 4
  - Do some constant number of color changes.
  - At most 3 rotations.
- RB-Delete-Fixup : Case 2
  - Recurses up tree
  - O(lg n)

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [35], called "B-trees."

Red-black trees were invented by Bayer [34] under the name "symmetric binary B-trees." Guibas and Sedgewick [155] studied their properties at length and introduced the red/black color convention. Andersson [15] gives a simpler-to-code

- Chapter 18 has another type of balanced trees called "B-trees"
- Red black trees were invented under the name "symmetric binary B-trees"

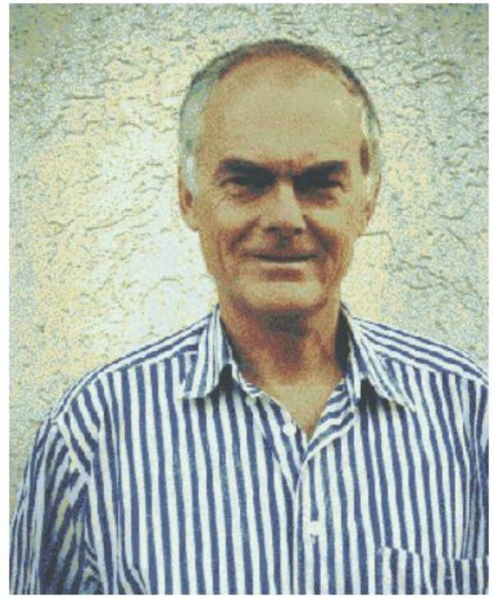
Rudolf Bayer

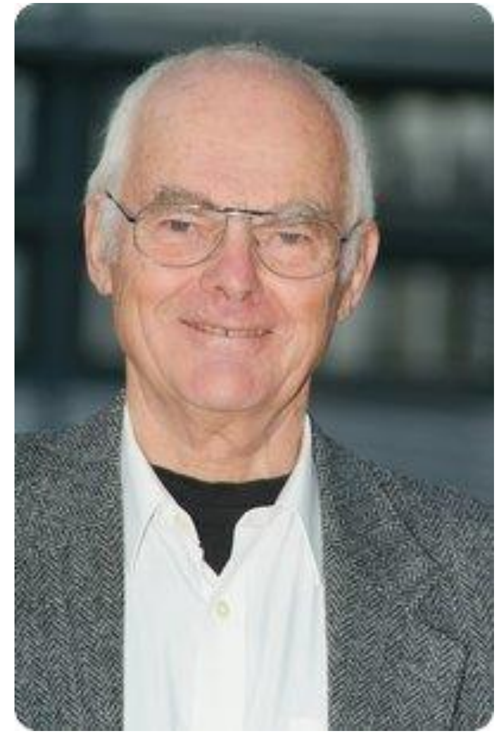
Born	May 7, 1939 (age 76)
Nationality	German
Institutions	Technical University Munich
Alma mater	University of Illinois at Urbana–Champaign
Thesis	*Automorphism Groups and Quotients of Strongly Connected Automata and Monadic Algebras* (1966)
Doctoral advisor	Franz Edward Hohn[1]
Known for	B-tree UB-tree red-black tree
Notable awards	Cross of Merit, First class (1999), SIGMOD Edgar F. Codd Innovations Award (2001)

Key to Indexing in DB

rsonen | Lehre | Lehrstuhl | Forschung | Suche

essoren | Sekretariat | Mitarbeiter | ehem. Mitarbeiter | Gäste





Titel:	Univ.-Prof.
Vorname:	Rudolf
name:	Bayer
ensuffix:	Ph. D.
on:	+ 49 89 289-17278
Telefax:	+ 49 89 289-17255
Raum:	02.11.059
Bereich:	◦ Datenschutz der Fakultät ◦ DBLP-record ◦ ACM/SIGMOD Innovations Award (2001)

[34] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[35] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

SYMMETRIC BINARY B-TREES: DATA STRUCTURE AND ALGORITHMS FOR RANDOM AND SEQUENTIAL INFORMATION PROCESSING*

Rudolf Bayer
Computer Sciences
Purdue University
Lafayette, Indiana 47907

CSD TR 54

November 1971

ABSTRACT

A class of binary trees is described for maintaining ordered sets of data. Random insertions, deletions, and retrievals of keys can be done in time proportional to log N where N is the cardinality of the data-set. Symmetric B-trees are a modification of B-trees described previously by Bayer and McCreight. This class of trees properly contains the balanced trees.

* This work was partially supported by an NSF grant.

[155] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas
Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University

and

Robert Sedgewick*
Program in Computer Science
Brown University
Providence, R. I.

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this framework the best known balanced tree techniques and then use the framework to develop new algorithms which perform the update and rebalancing in one pass, on the way down towards a leaf. We conclude with a study of performance issues and concurrent updating.

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its behavior are proved.

In both sections 1 and 2 particular attention is paid to practical implementation issues, and complete implementations are given for all of the important algorithms. This is significant because one

- Robert Sedgewick and Leo Guibas in 1978 coursera

Catalog | Search catalog

Institutions Log In Sign Up

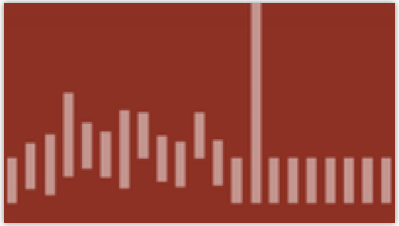
Robert Sedgewick

Professor

Computer Science

Princeton University

Robert Sedgewick is the William O. Baker Professor of Computer Science at Princeton, where he was the founding chair of the Department of Computer Science. He received the Ph.D. degree from Stanford University, in 1975. Prof. Sedgewick also served on the faculty at Brown University and has held visiting research positions at Xerox PARC, Palo Alto, CA, Institute for Defense Analyses, Princeton, NJ, and INRIA, Rocquencourt, France. He is a member of the board of directors of Adobe Systems. Prof. Sedgewick's interests are in analytic combinatorics, algorithm design, the scientific analysis of algorithms, curriculum development, and innovations in the dissemination of knowledge. He has published widely in these areas and is the author of several books.



Algorithms, Part II

October 30, 2015

Analysis of Algorithms

Leonidas J. Guibas

Paul Pigott Professor of Computer Science and
Electrical Engineering (courtesy)
in the School of Engineering

Research Statement

Professor Guibas heads the Geometric Computation group in the Computer Science Department of Stanford University and is a member of the Computer Graphics and Artificial Intelligence Laboratories. He works on algorithms for sensing, modeling, reasoning, rendering, and acting on the physical world. Professor Guibas' interests span computational geometry, geometric modeling, computer graphics, computer vision, sensor networks, robotics, and discrete algorithms --- all areas in which he has published and lectured extensively. Examples of current and recent activities include:

AVL Trees

- AVL Trees: Adel'son-Vel'skii & Landis 1962
- For every node, require heights of left & right children to differ by at most 1.

IDEA of Balancing

Bibliography

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(5):1259–1263, 1962.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [35], called "B-trees."

Red-black trees were invented by Bayer [34] under the name "symmetric binary B-trees." Guibas and Sedgewick [155] studied their properties at length and introduced the red/black color convention. Andersson [15] gives a simpler-to-code

An algorithm for the organization of information
by G. M. Adel'son-Vel'skii and E. M. Landis
Soviet Mathematics Doklady, 3, 1259-1263, 1962
http://monet.skku.ac.kr
Networkging Lab.

Thus (5.1) is not tr 1962 $r = 1$.

Received 22/MAR/62

BIBLIOGRAPHY

[1] C. Miranda, *Equazioni alle derivate parziali di tipo ellittico*, Springer, Berlin, 1955.

[2] S. M. Nikol'skiĭ, Sibirsk. Mat. Ž. 1 (1960), 78.

Translated by:
F. M. Goodspeed

AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

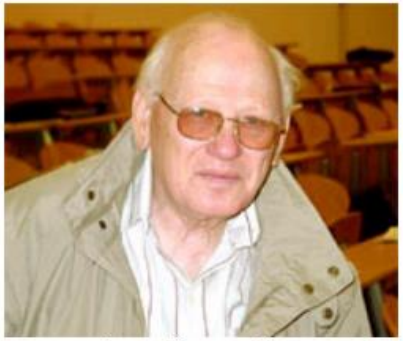
G. M. ADEL'SON-VEL'SKIĬ AND E. M. LANDIS

In the present article we discuss the organization of information contained in the cells of an automatic calculating machine. A three-address machine will be used for this study.

Statement of the problem. The information enters a machine in sequence from a certain reserve. The information element is contained in a group of cells which are arranged one after the other. A certain number (the information estimate), which is different for different elements, is contained in the information element. The information must be organized in the memory of the machine in such a way that at any moment a very large number of operations is not required to scan the information with the given evaluation and to record the new information element.

Георгий Максимович Адельсон-Вельский

Домашняя страница



Автобиография

Первая научная работа была сделана мной в 1944 г., когда я был студентом 4-го курса. Я обобщил одну теорему С.Н. Бернштейна о порядке роста значений функции двух переменных, чей график в трехмерном пространстве значений аргументов и функций имеет отрицательную кривизну, заменив это требование другим, не предполагающим существования производных у рассматриваемых функций: отсутствием ограниченных связных частей во множестве точек любой плоскости этого трехмерного пространства, полученного в результате исключения точек ее пересечения с графиком рассматриваемой функции. Работа получила 3-ю премию на конкурсе студенческих научных работ. Этот подход и его результат были использованы в следующем году в совместном с А.С. Кронродом решении проблемы, поставленной Лузиным: доказать, что моногенная в некоторой области функция аналитична в этой области, пользуясь только ее качественными свойствами, а не интегралом Коши. За эту работу мы получили премию

Evgenii Landis

From Wikipedia, the free encyclopedia

Evgenii Mikhailovich Landis (Russian: Евге́ний Миха́йлович Ла́ндис, *Yevgeny Mikhaylovich Landis*; October 6, 1921 – December 12, 1997) was a Soviet mathematician who worked mainly on partial differential equations.

Landis at a conference on potential theory in Prague, 1987

Life [edit]

Landis was born in Kharkiv, Ukrainian SSR, Soviet Union. He studied and worked at the Moscow State University, where his advisor was Alexander Kronrod, and later Ivan Petrovsky. In 1946, together with Kronrod, he rediscovered Sard's lemma unknown in Russia at the time.

Later he worked on uniqueness theorems for elliptic and parabolic differential equations, Harnack inequalities, and Phragmén–Lindelöf type theorems. With Georgy Adelson-Velsky, he invented the AVL tree datastructure (where "AVL" stands for **A**delson-**V**elsky **L**andis).

He died in Moscow. His students include Yulij Ilyashenko.

External links [edit]

- Evgenii Landis at the Mathematics Genealogy Project
- Biography of Y.M. Landis at the International Centre for Mathematical Sciences.

AVL Trees

- For every node, require heights of left & right children to differ by at most 1.
- Treat nil tree as height = -1
- Each node stores its height (DATA STRUCTURE AUGMENTATION)

AVL Tree

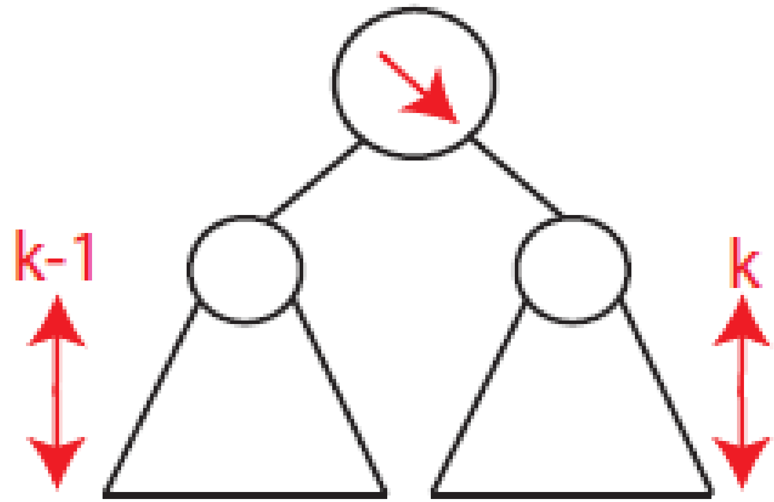


Figure 4: AVL Tree Concept

AVL Balance

- Worst when every node differs by 1
- Let N_k = (min) # nodes in height-h AVL Tree

$\Rightarrow N_h = N_{h-1} + N_{h-2} + 1$

$\Rightarrow \quad > 2N_{h-2}$

$\Rightarrow N_h > 2^{h/2}$

$\Rightarrow h < 2\lg N_h$

AVL Balance (Alternate)

- $N_h > F_h$ (Nth Fibonacci Number)
- $N_h = F_{n+1} - 1$ (simple induction)
- Max h approximately 1.440lg n

AVL Tree

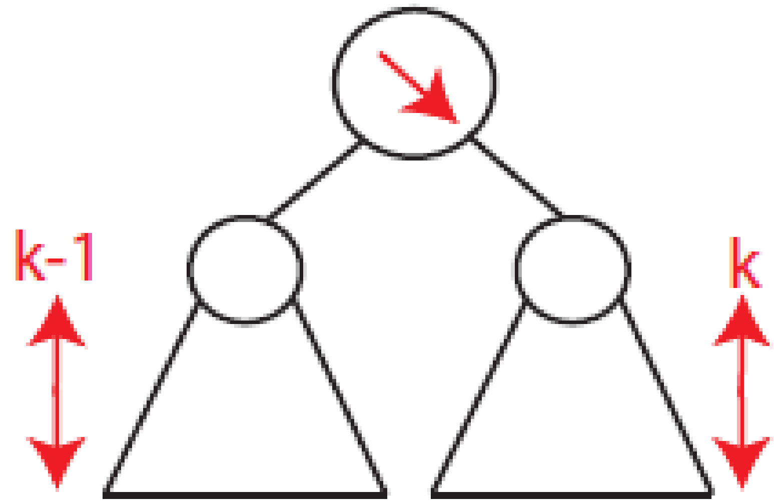
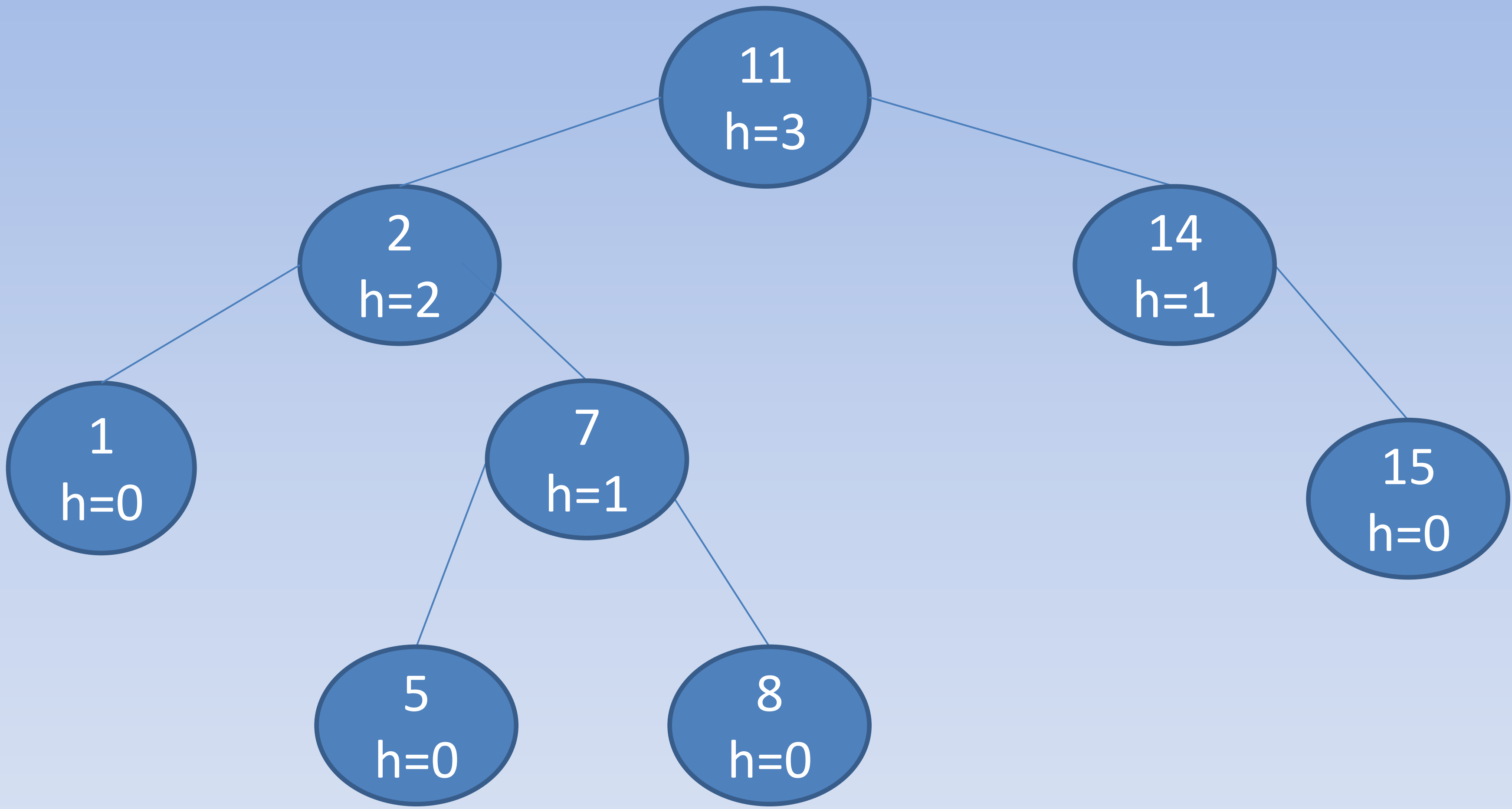
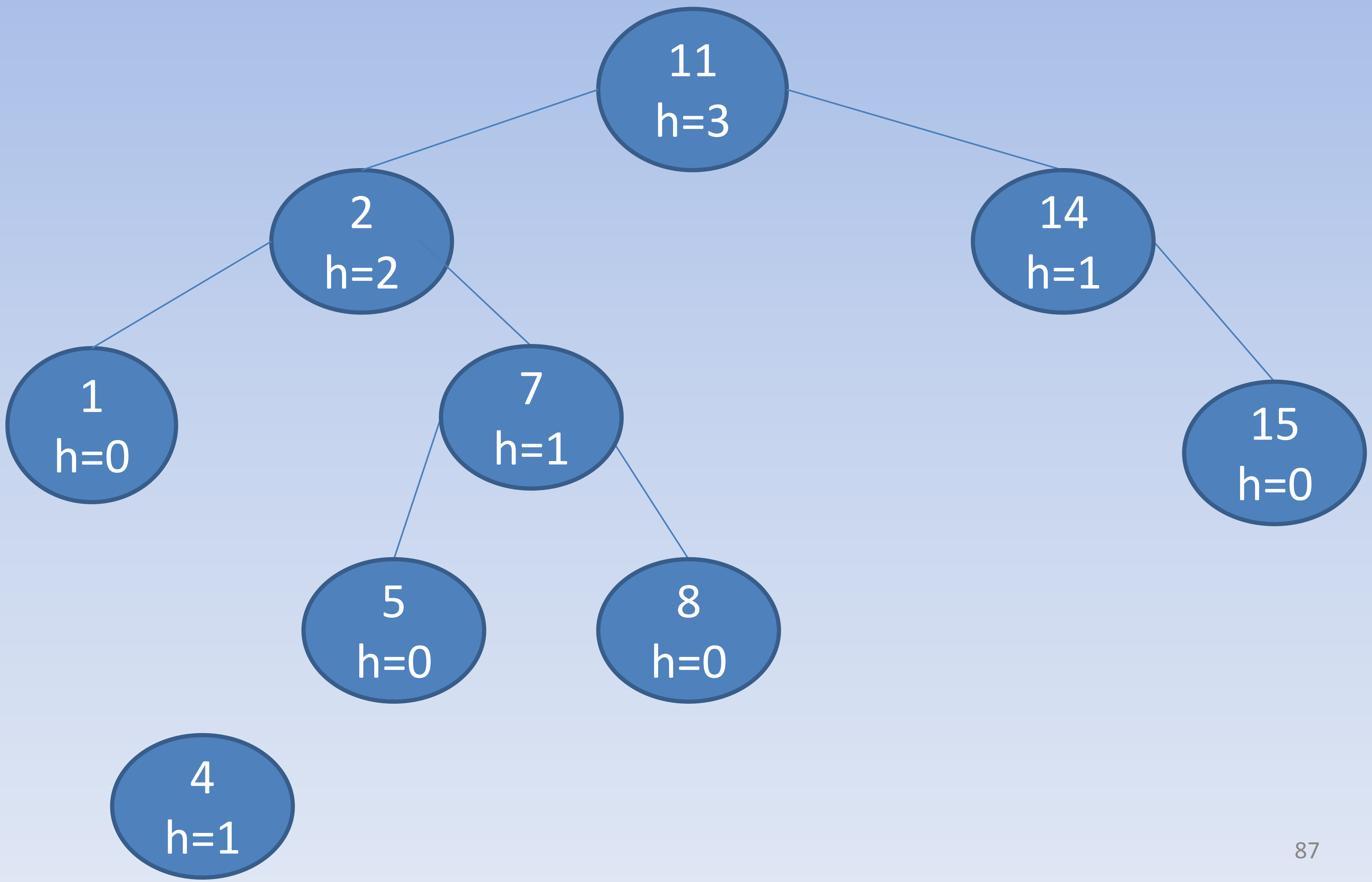


Figure 4: AVL Tree Concept

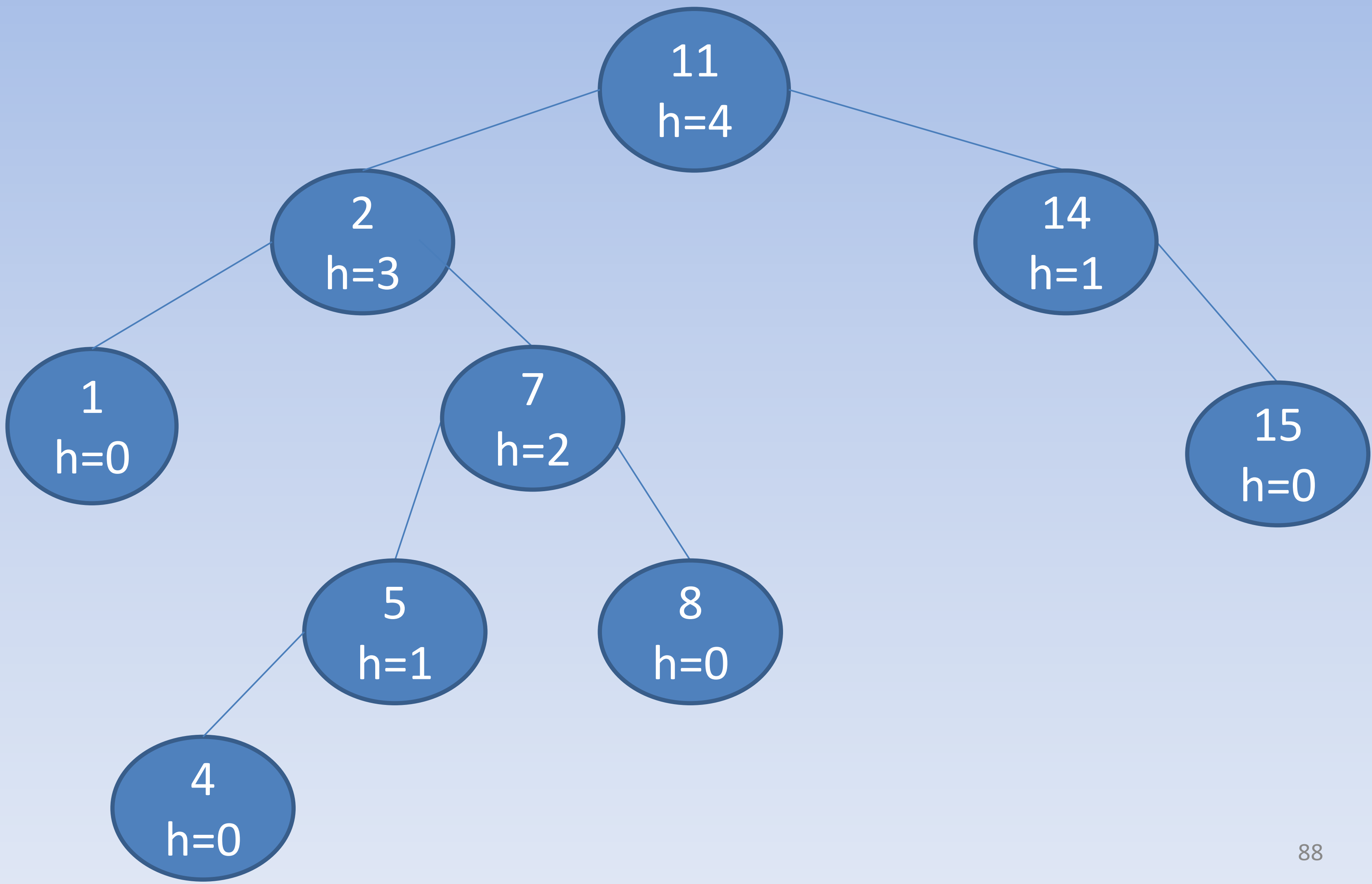
AVL Tree



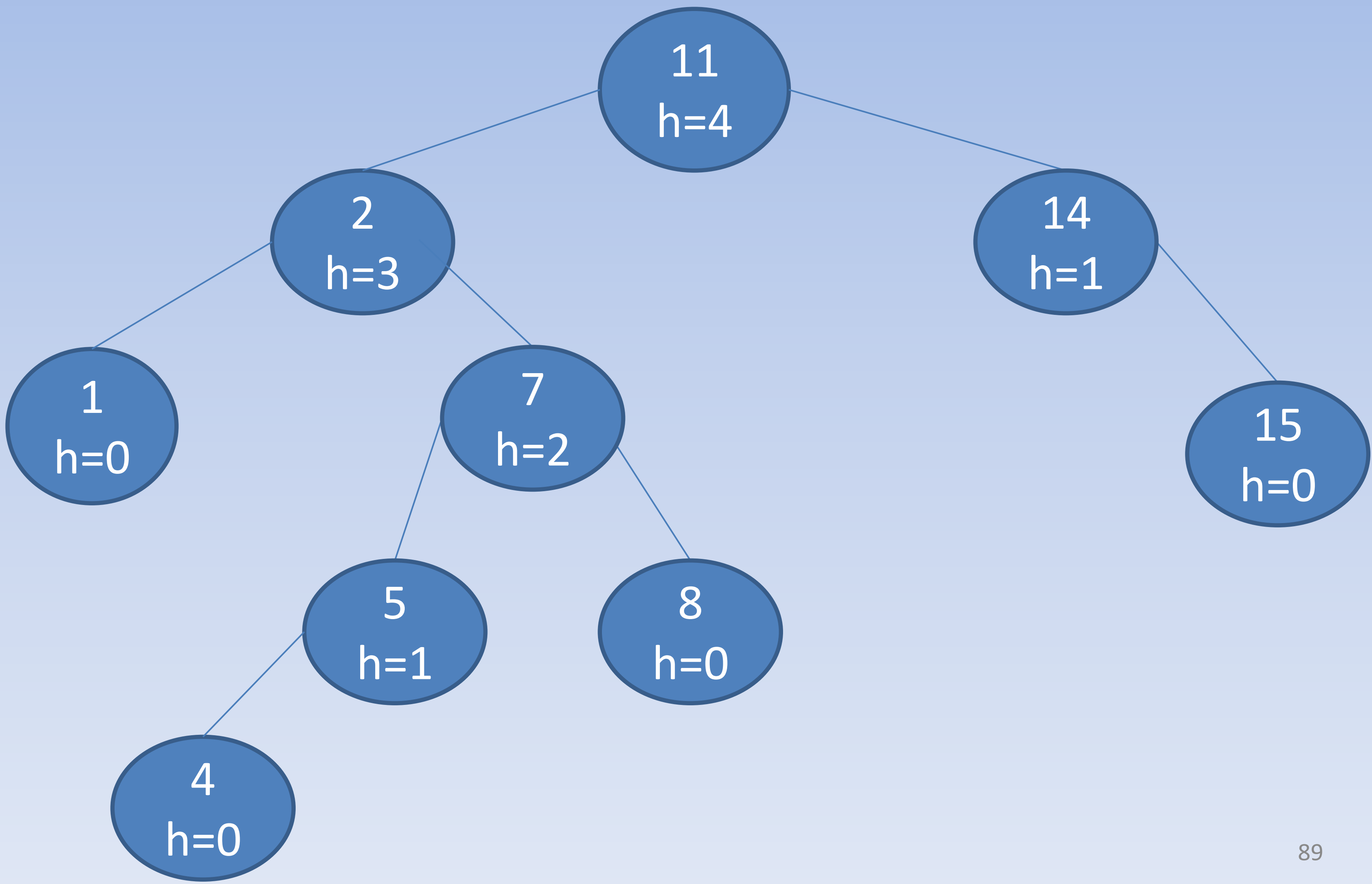
AVL Tree: Insert 4



AVL Tree: Update Heights

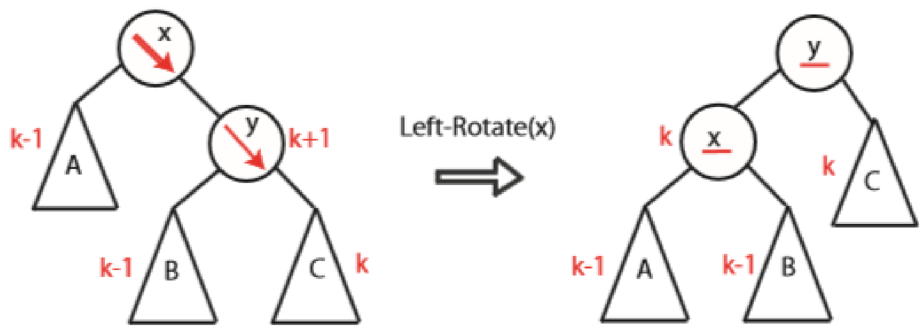


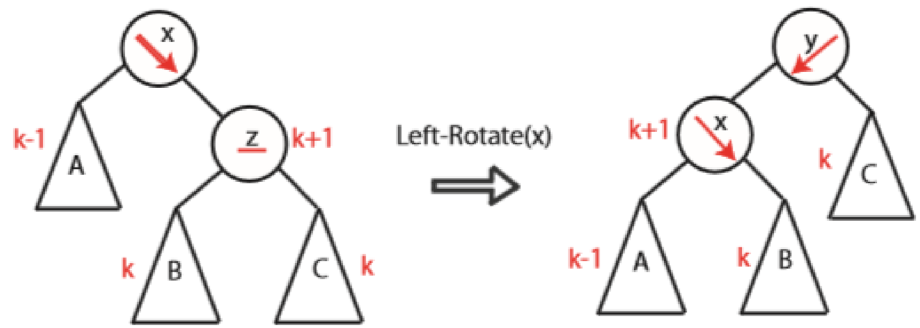
AVL Tree: AVL Property Violation



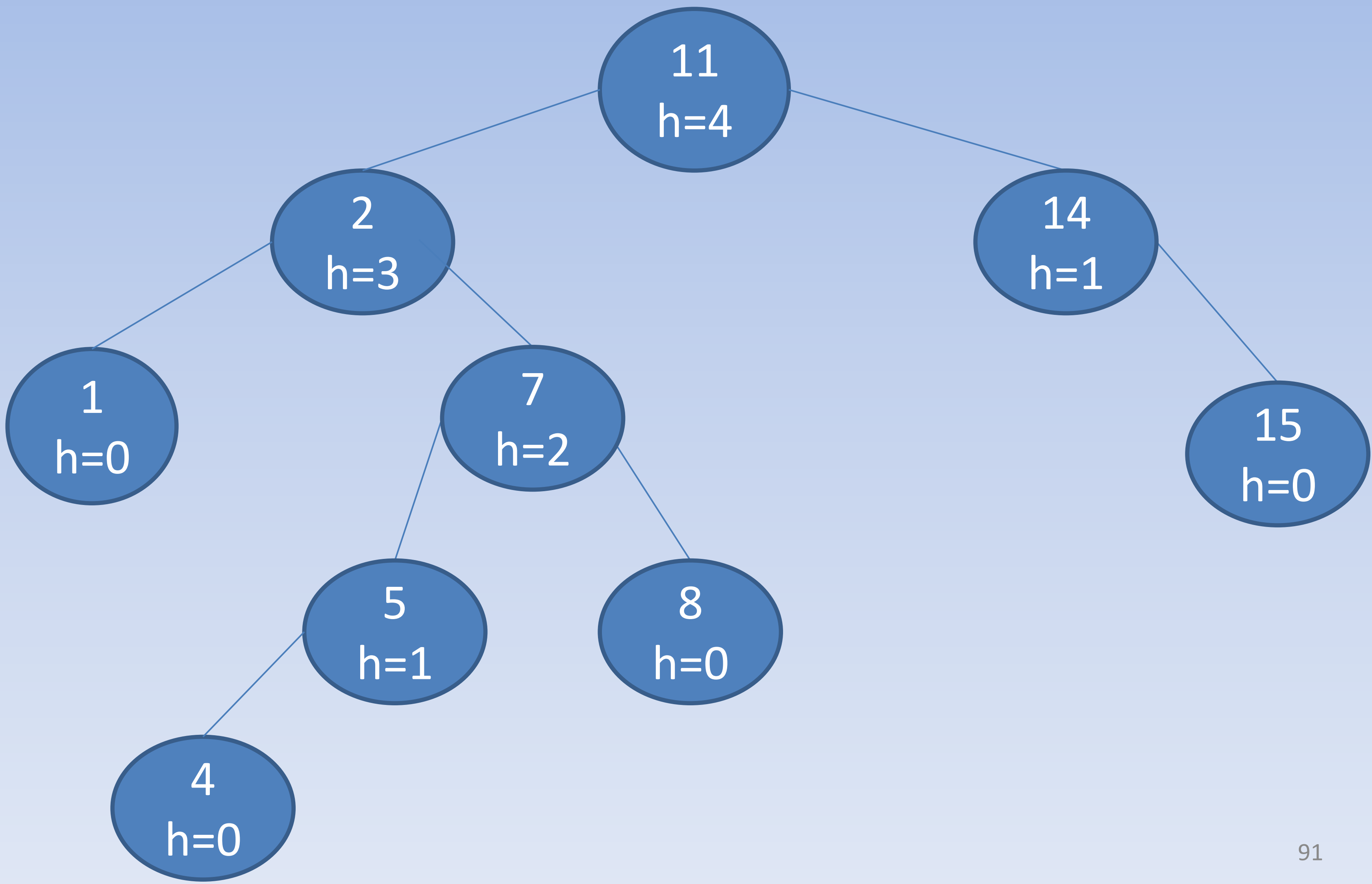
Each Step:

- suppose x is lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- if x's right child is right-heavy or balanced:



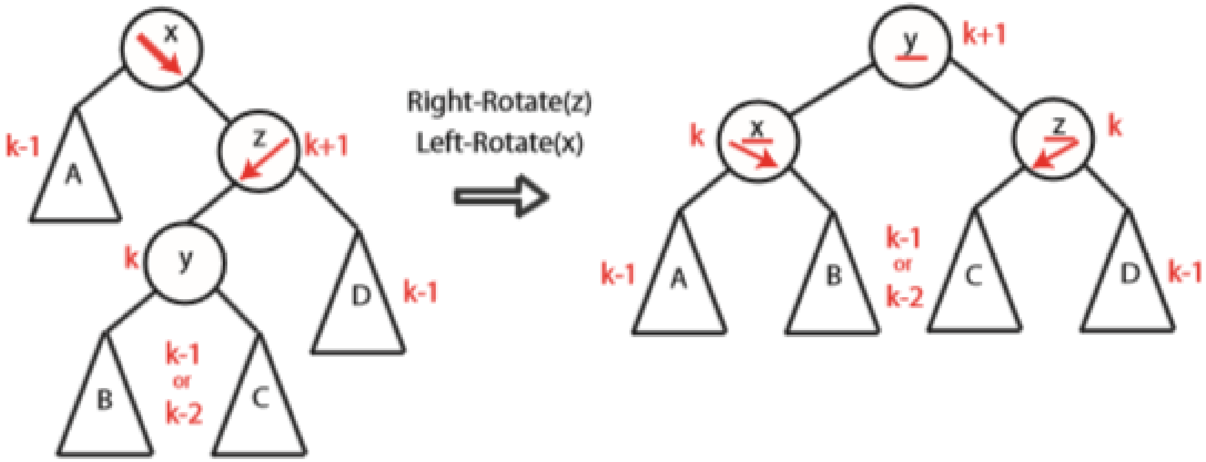


AVL Tree: AVL Property Violation

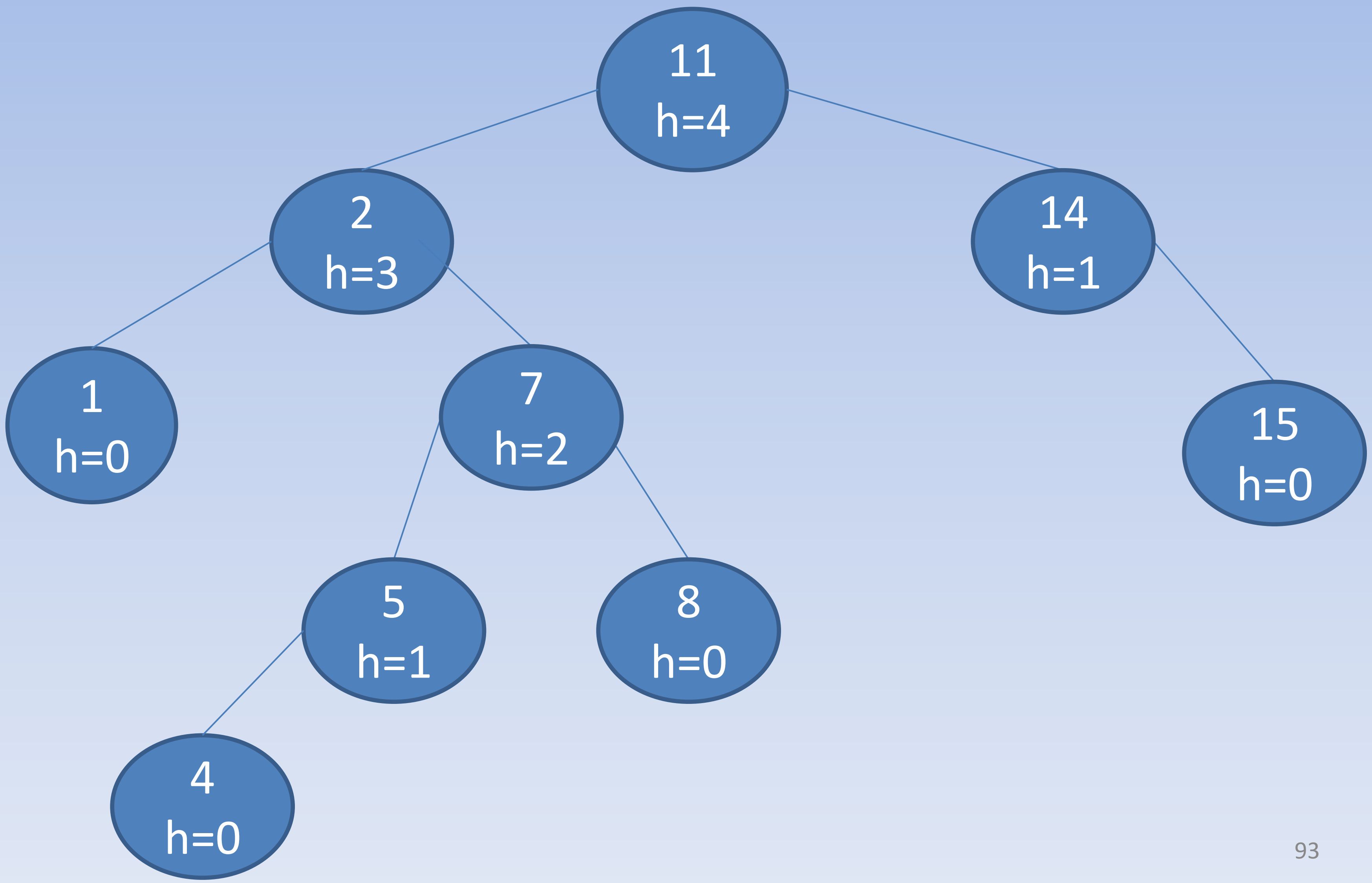


x's right child is NOT right heavy

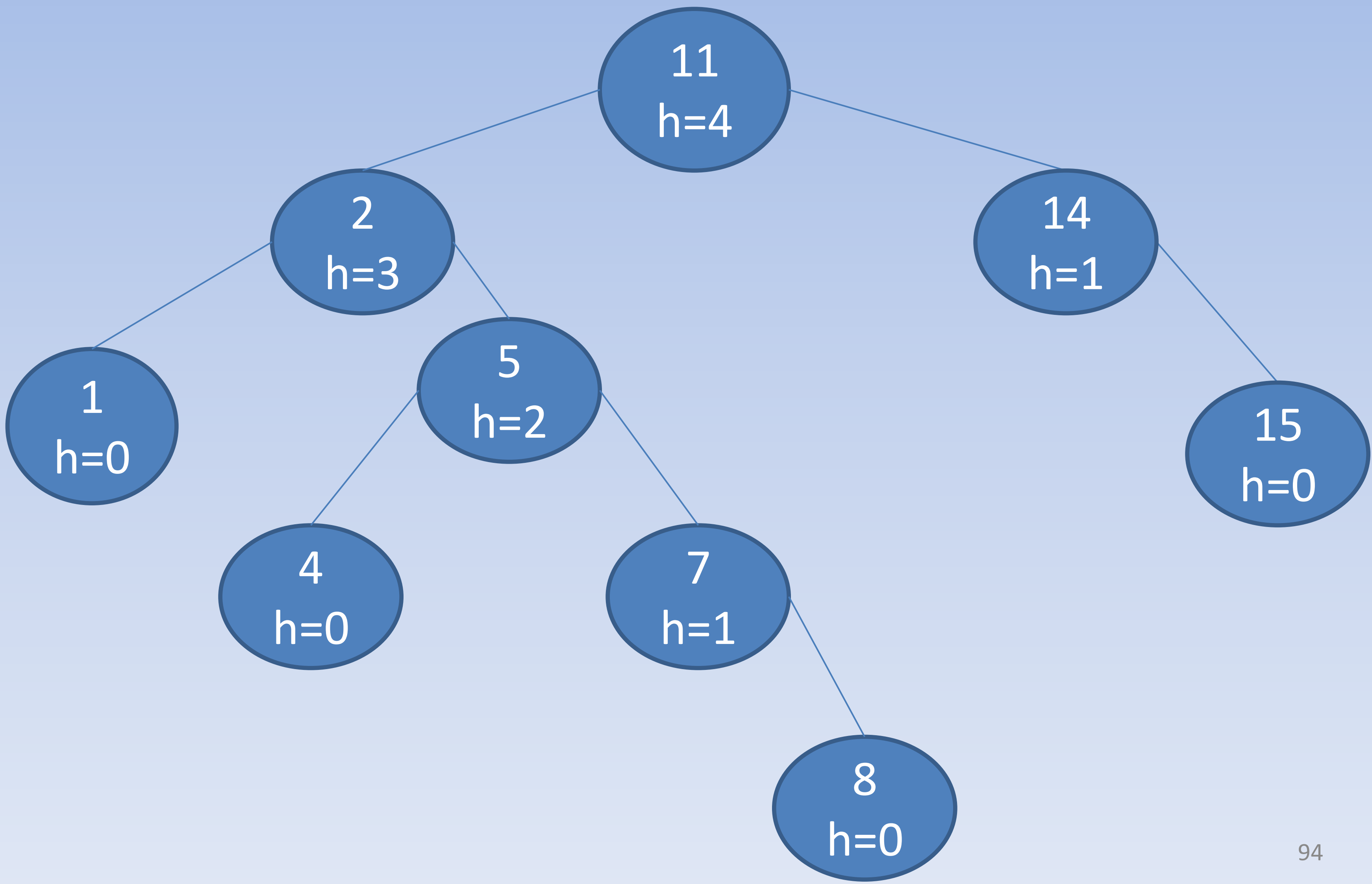
- else: follow steps



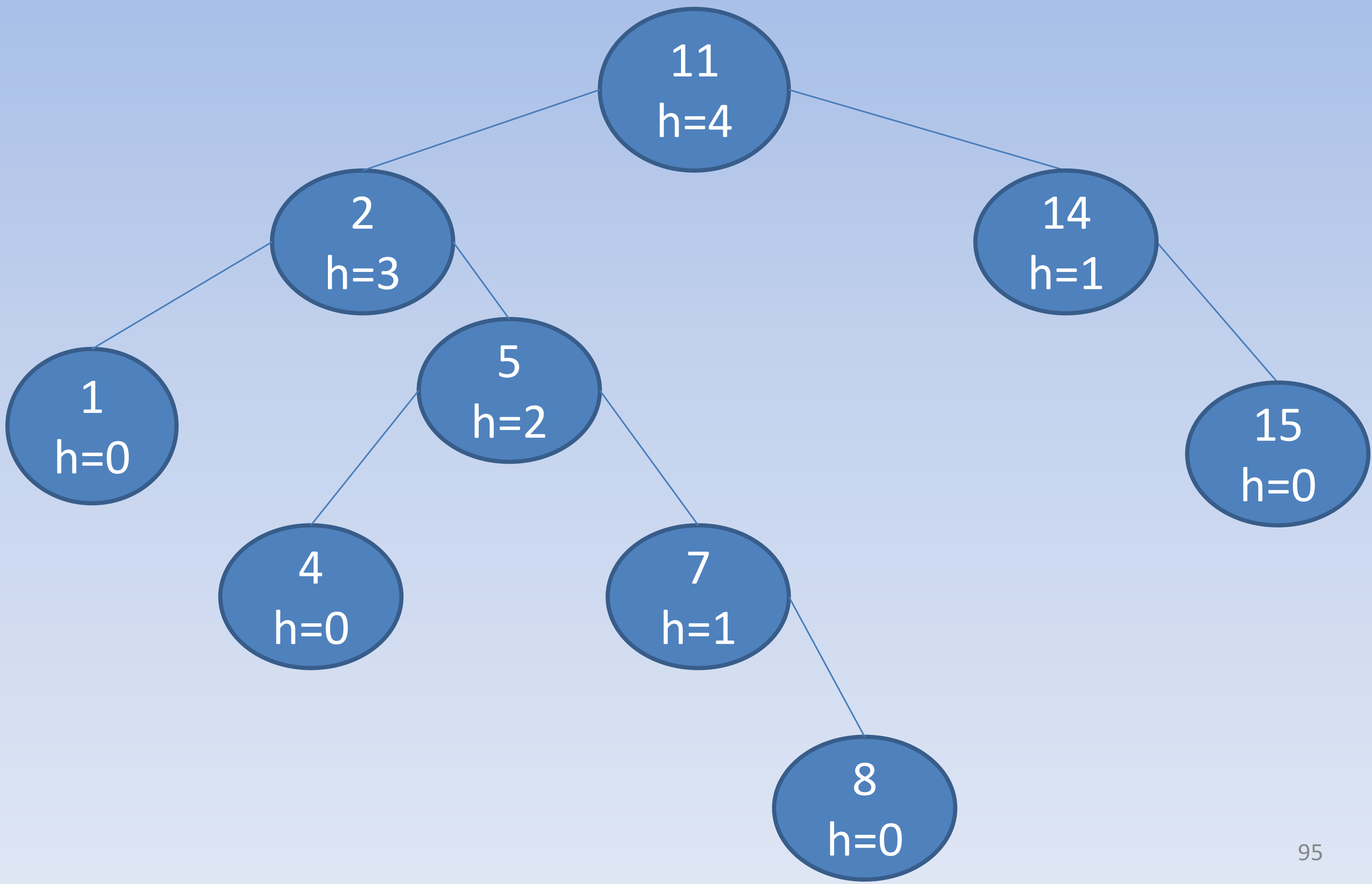
Right-Rotate @ 7



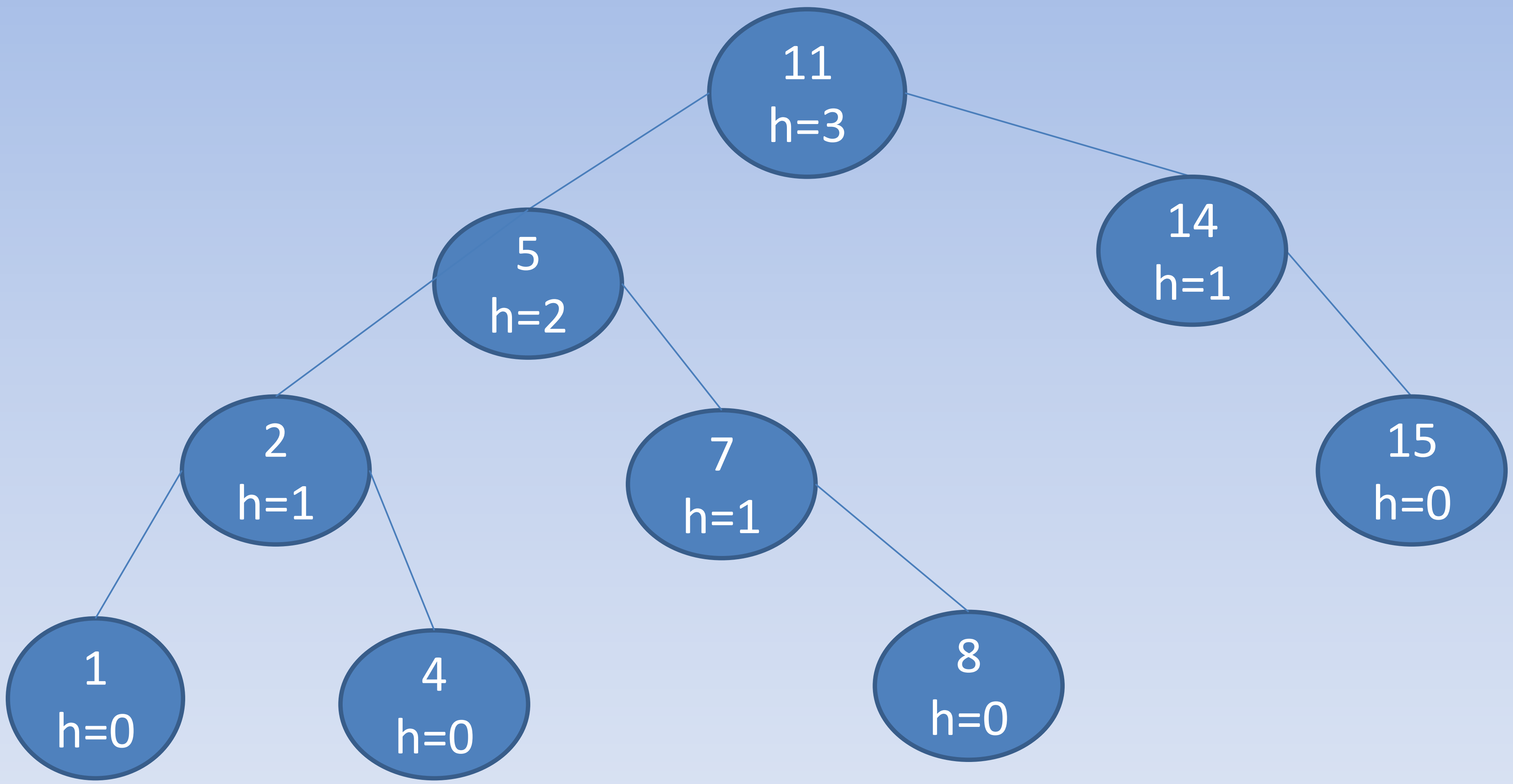
Result of Right-Rotate @ 7



Left-Rotate @ 2 (Lowest Violation)



Result of Left-Rotate @ 2



AVL Balance

- Worst when every node differs by 1
- Let N_k = (min) # nodes in height-h AVL Tree

$\Rightarrow N_h = N_{h-1} + N_{h-2} + 1$

$\Rightarrow \quad > 2N_{h-2}$

$\Rightarrow N_h > 2^{h/2}$

$\Rightarrow h < 2\lg N_h$

AVL Balance (Alternate)

- $N_h > F_h$ (Nth Fibonacci Number)
- $N_h = F_{n+1} - 1$ (simple induction)
- Max h approximately 1.440lg n

```python
def height(node):
    if node is None:
        return -1
    else:
        return node.height

def update_height(node):
    node.height = max(height(node.left), height(node.right)) + 1
```

Nil is -1

```python
def insert(self, t):
    """Insert key t into this tree, modifying it in-place."""
    node = bst.BST.insert(self, t)
    self.rebalance(node)

def rebalance(self, node):
    while node is not None:
        update_height(node)
        if height(node.left) >= 2 + height(node.right):
            if height(node.left.left) >= height(node.left.right):
                self.right_rotate(node)
            else:
                self.left_rotate(node.left)
                self.right_rotate(node)
        elif height(node.right) >= 2 + height(node.left):
            if height(node.right.right) >= height(node.right.left):
                self.left_rotate(node)
            else:
                self.right_rotate(node.right)
                self.left_rotate(node)
        node = node.parent
```

Insert 0..9 into BST

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
        /\

0
/\
 1
 /\
  2
  /\
   3
   /\
    4
    /\
     5
     /\
      6
      /\
       7
       /\
        8
        /\
         9
         /\
>>> 
Ln: 251 Col: 4
```

Insert 0..9 into AVL

```
Inserting:  0
0
/\
Balancing (  LeftHeight = -1 | RightHeight = -1 )
```

Insert 0..9 into AVL

```
Inserting:  1
0
/\
 1
 /\
Balancing (  LeftHeight = -1 | RightHeight = 0 )
```

Insert 0..9 into AVL

```
Inserting:  2
0
/\
 1
 /\
  2
  /\
Balancing (  LeftHeight = -1 | RightHeight = 1 )
Difference = 2

After Balancing (  0 | 0 )
  1
 / \
0  2
/\ /\
```

```
Inserting:  3
  1
 / \
0  2
/\ /\
    3
    /\
Balancing (  0 | 1 )

After Balancing (  0 | 1 )
  1
 / \
0  2
/\ /\
    3
    /\
```

```
Inserting:  4
  1
 / \
0  2
/\ /\
    3
    /\
     4
     /\
Balancing (  0 | 2 )

After Balancing (  0 | 1 )
  1.
 /  \
0    3
/\  / \
   2  4
   /\ /\
```

```
Inserting:  5
  1.
 /  \
0    3
/\  / \
   2  4
   /\ /\
      5
      /\
Balancing (  0 | 2 )

After Balancing (  1 | 1 )
   .3.
  /   \
  1   4
 / \  /\
0  2   5
/\ /\  /\
```