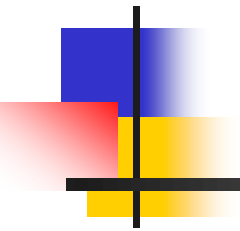


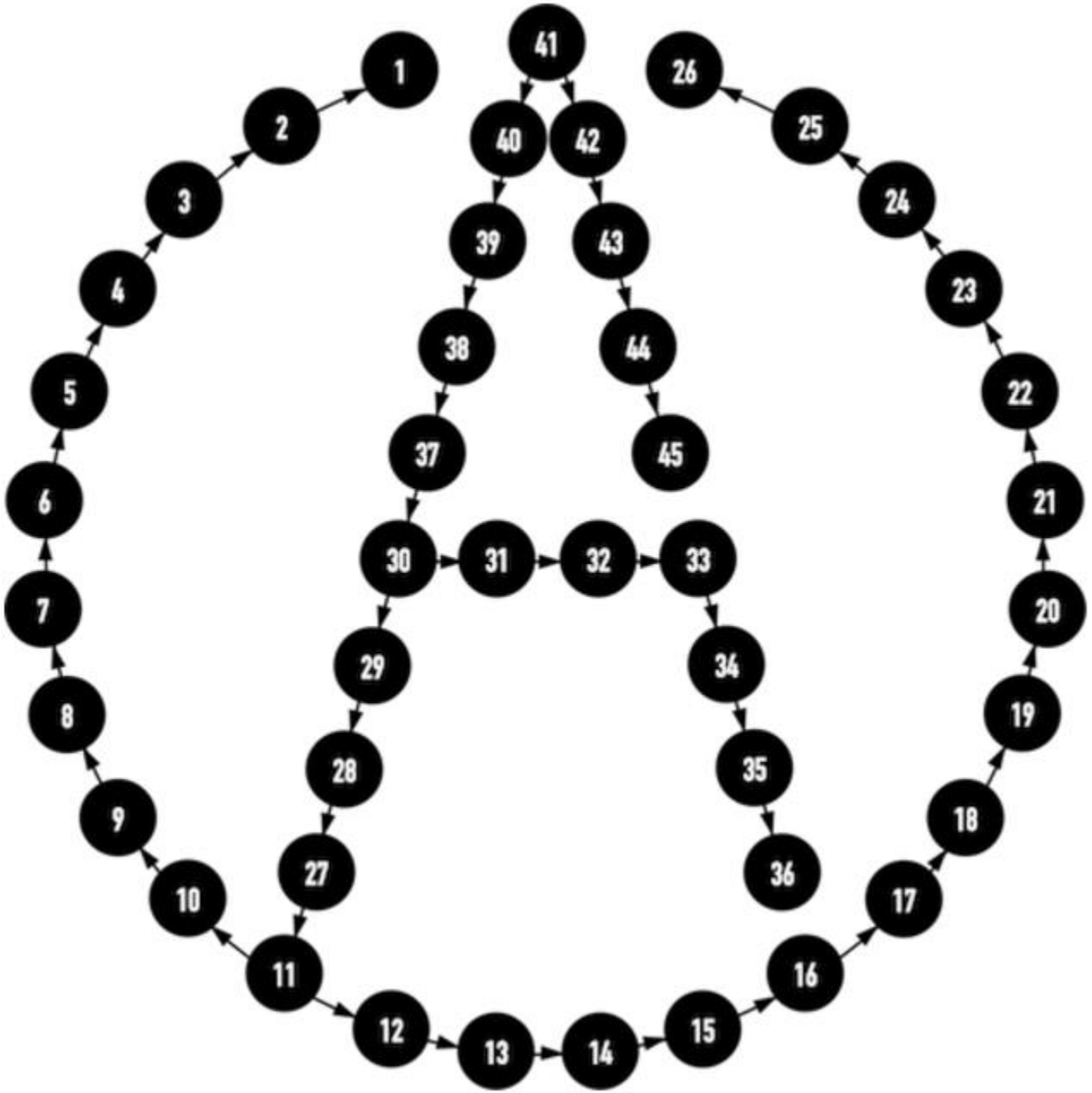
13. Red-Black Tree

Yu-Shuen Wang, CS, NCTU

Thanks the images from http://alrightchiu.github.io/SecondRound/mu-lu-yan-suan-fa-yu-zi-liao-jie-gou.html

Red-black trees

One of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take O(lg n) time in the worse case.


41
1
26
2
40
42
25
3
24
39
43
4
23
38
44
5
22
37
45
6
21
30
31
32
33
7
20
29
34
8
19
28
35
9
18
27
36
10
17
11
16
12
13
14
15

13.1 Properties of red-black tree

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more then twice as long as any other, so that three is approximately balanced.

13.1 Properties of red-black tree

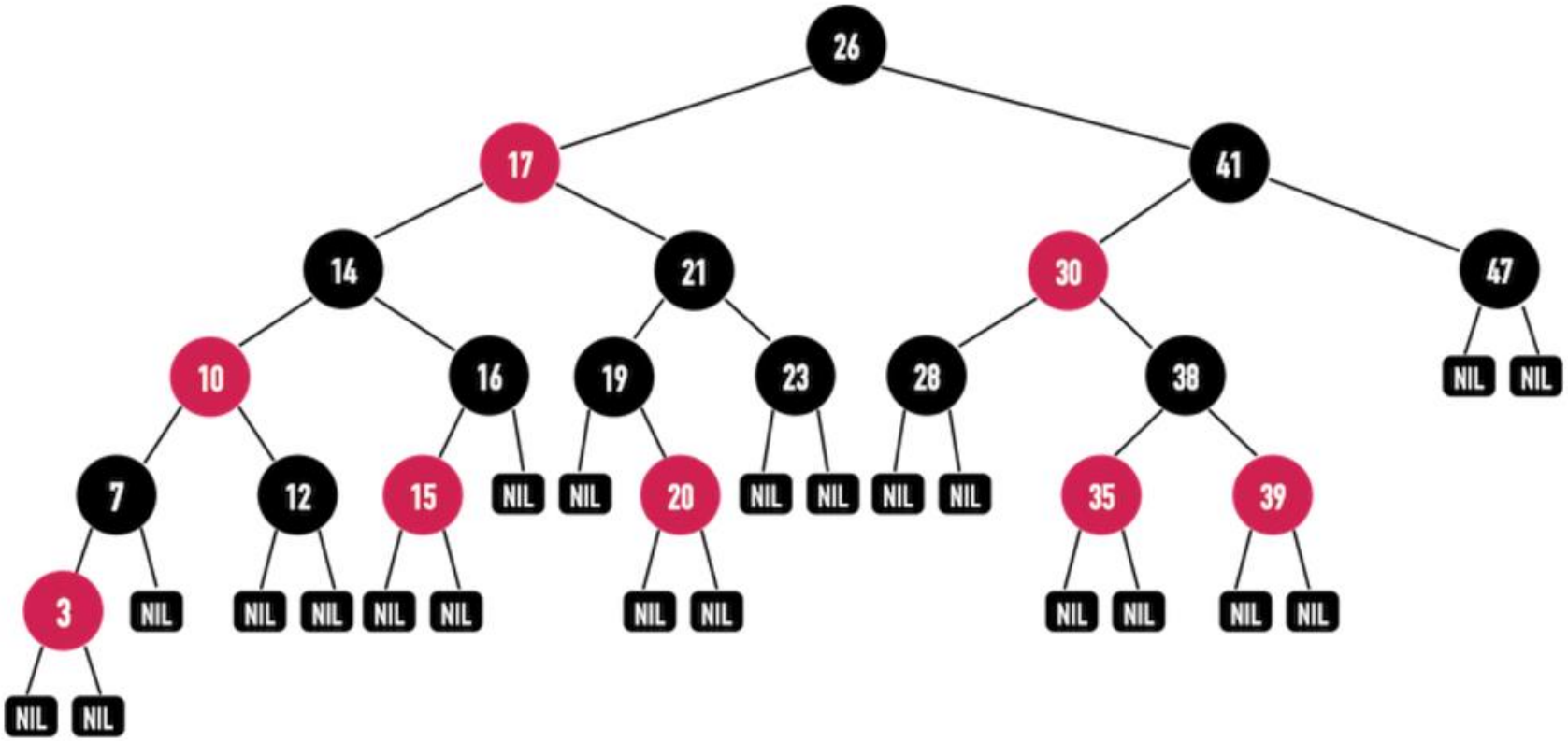
Each node of the tree now contains the fields *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

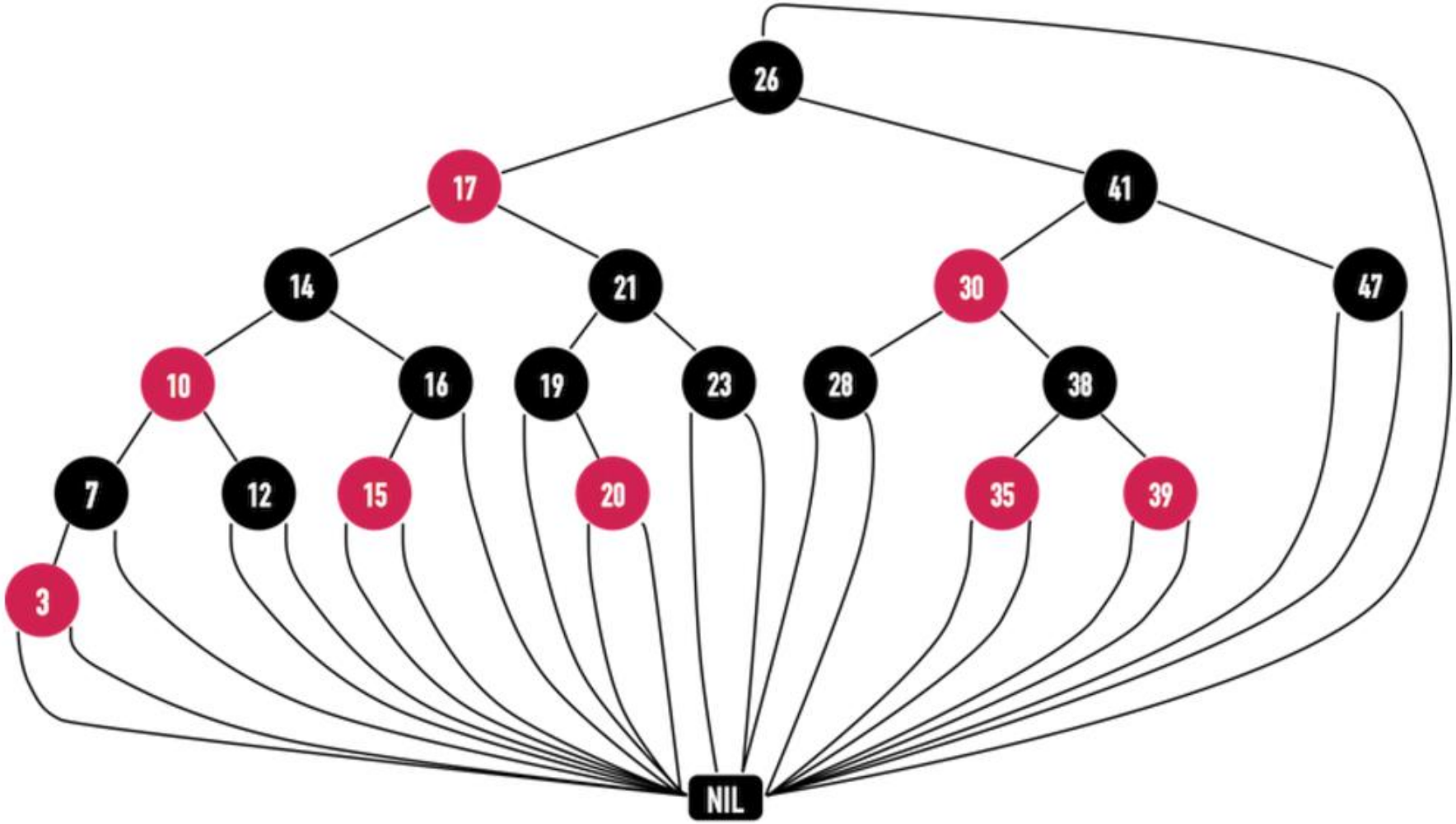
13.1 Properties of red-black tree

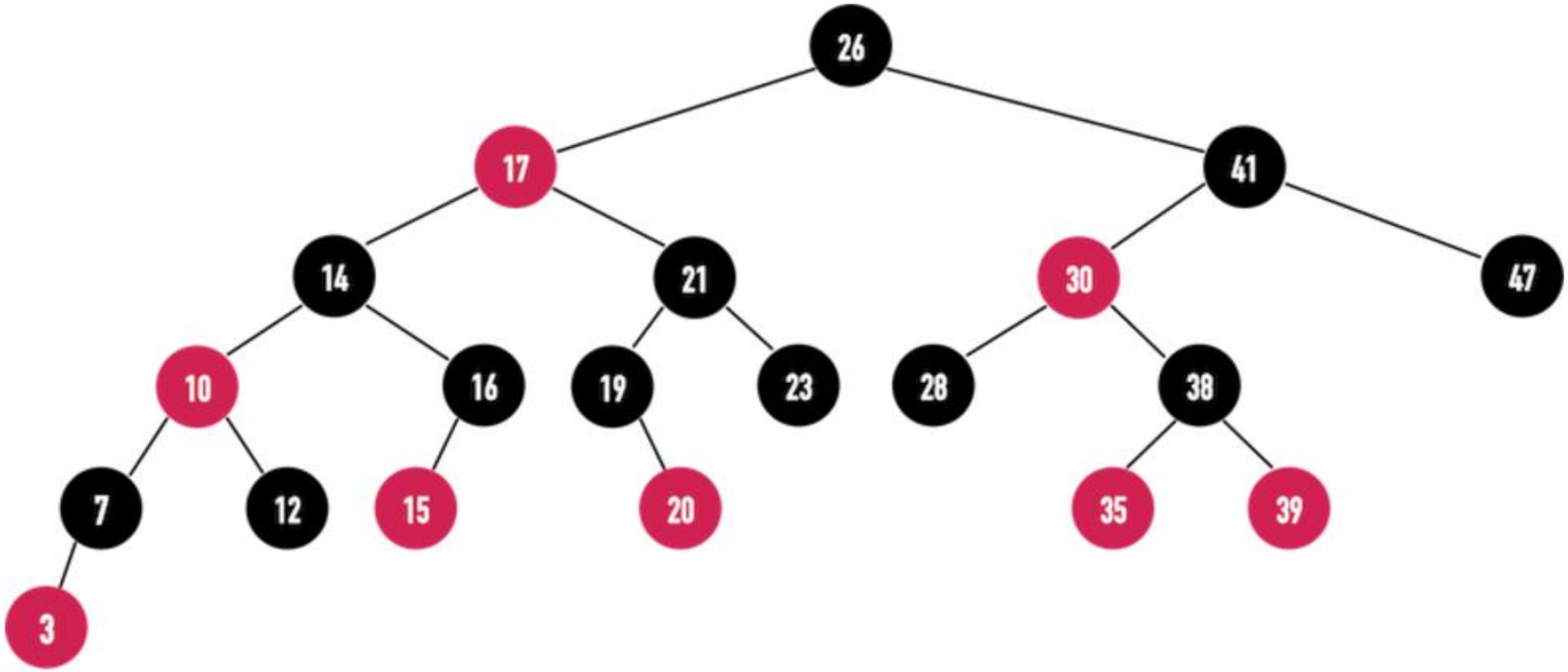
A binary search tree is a red-black tree if it satisfies the following red-black properties:

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.

A red-black tree



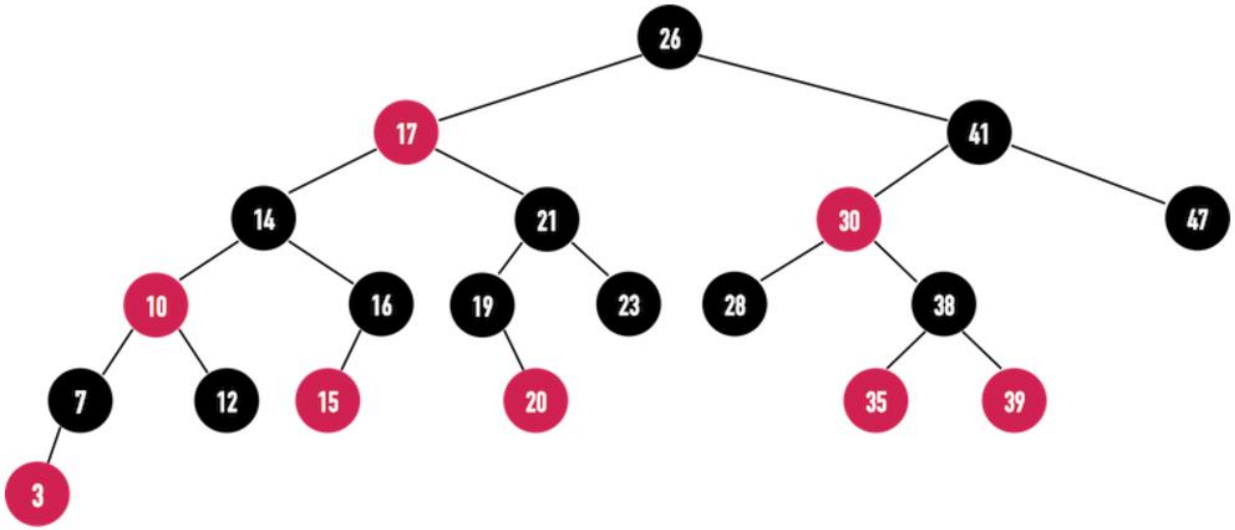

26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NIL


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

Lemma 13.1

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

Proof.



- We start by showing that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($\text{NIL}[T]$), and the sub tree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

bh(x): black height of sub-tree rooted at x

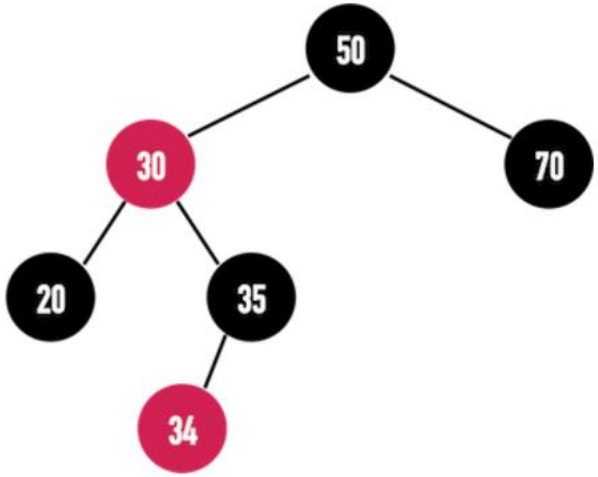
For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

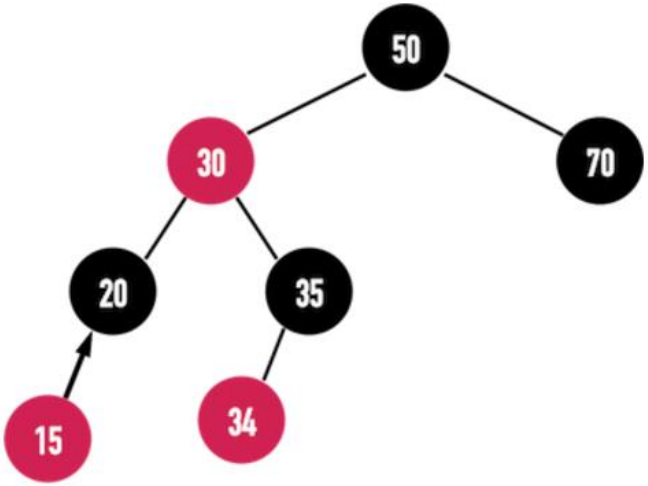
$$n \geq 2^{h/2} - 1.$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n + 1) \geq h/2$, or $2\lg(n+1) \geq h$.

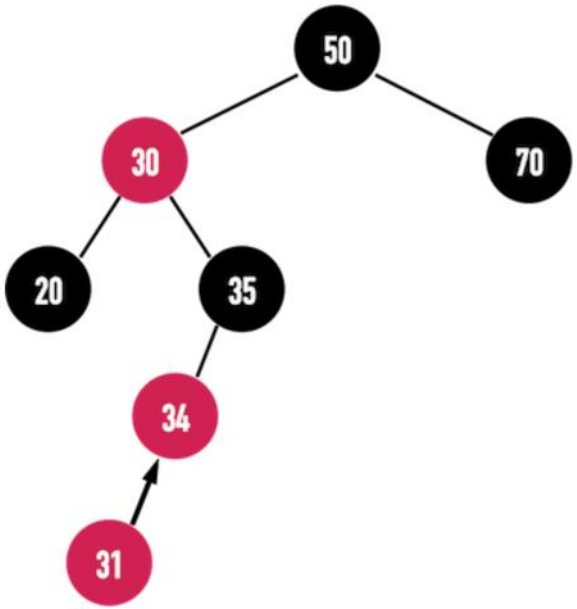
13.2 Rotations – why?



original

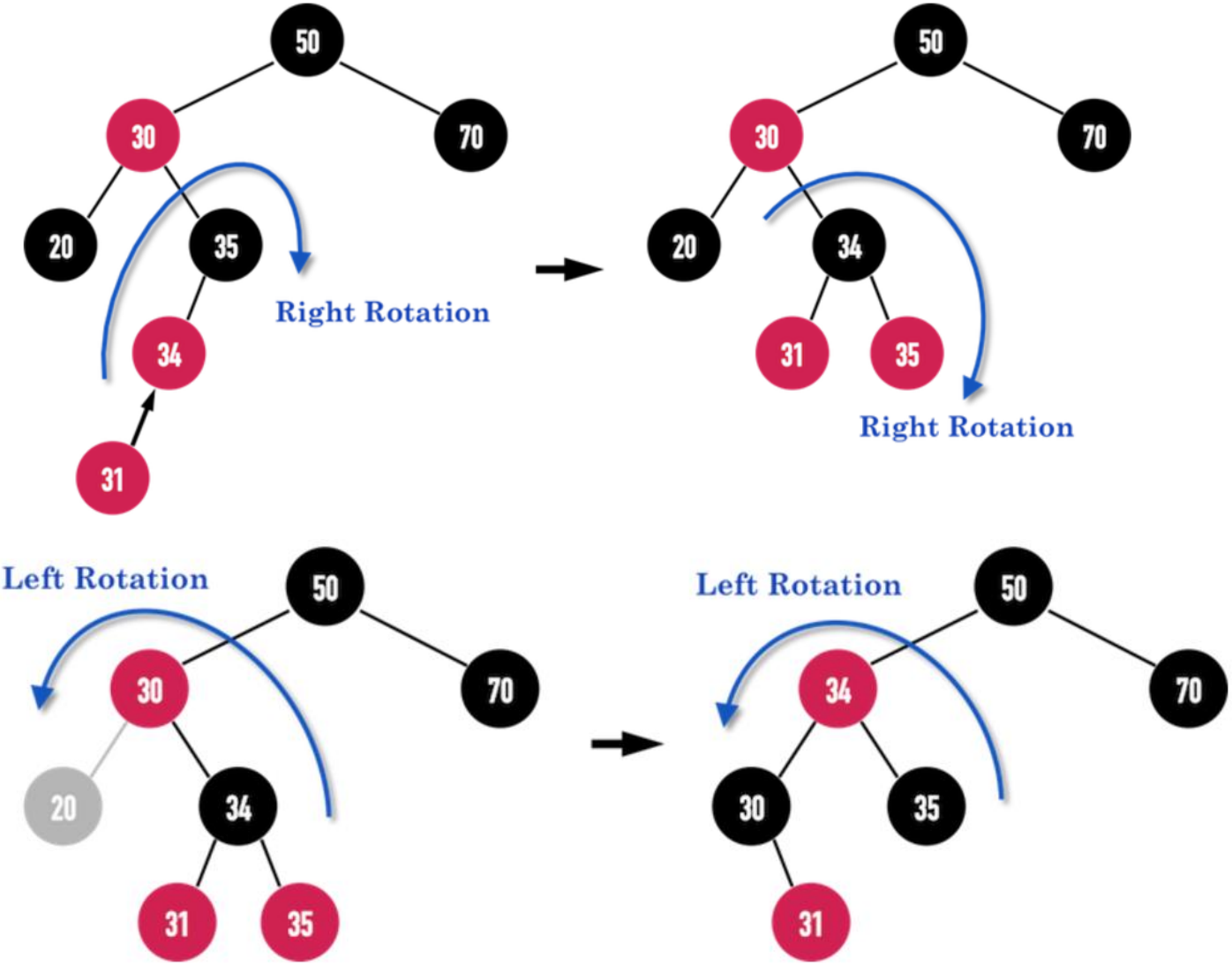


insert 15



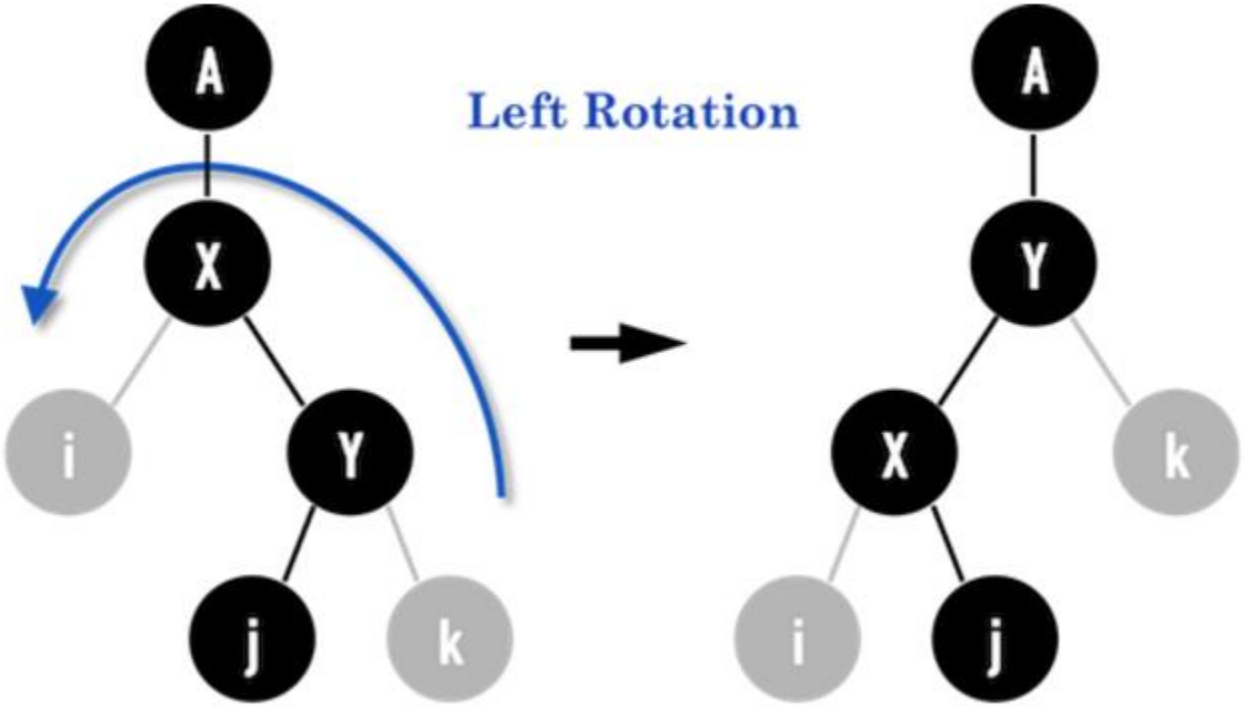
insert 31

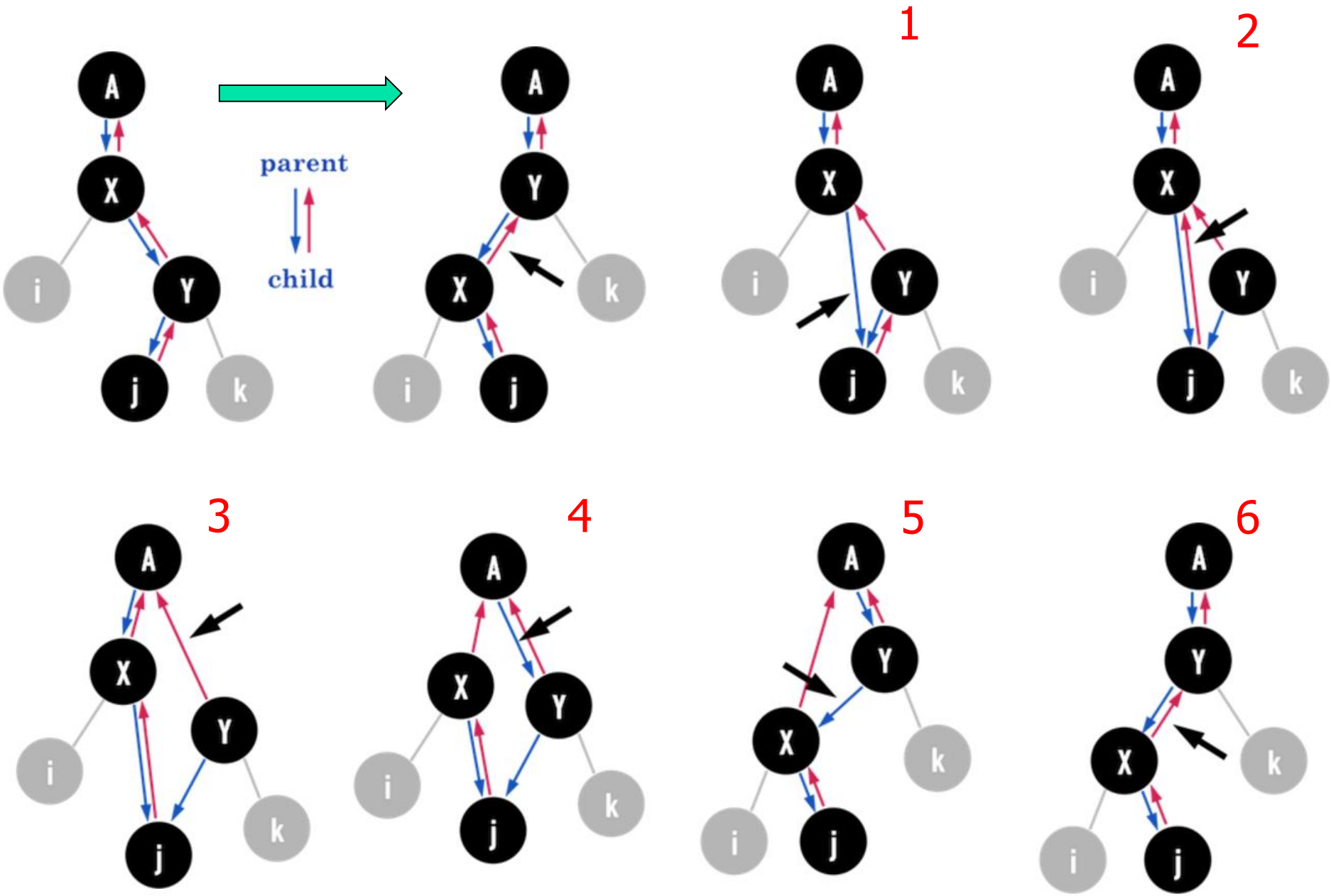
13.2 Rotations



delete 20

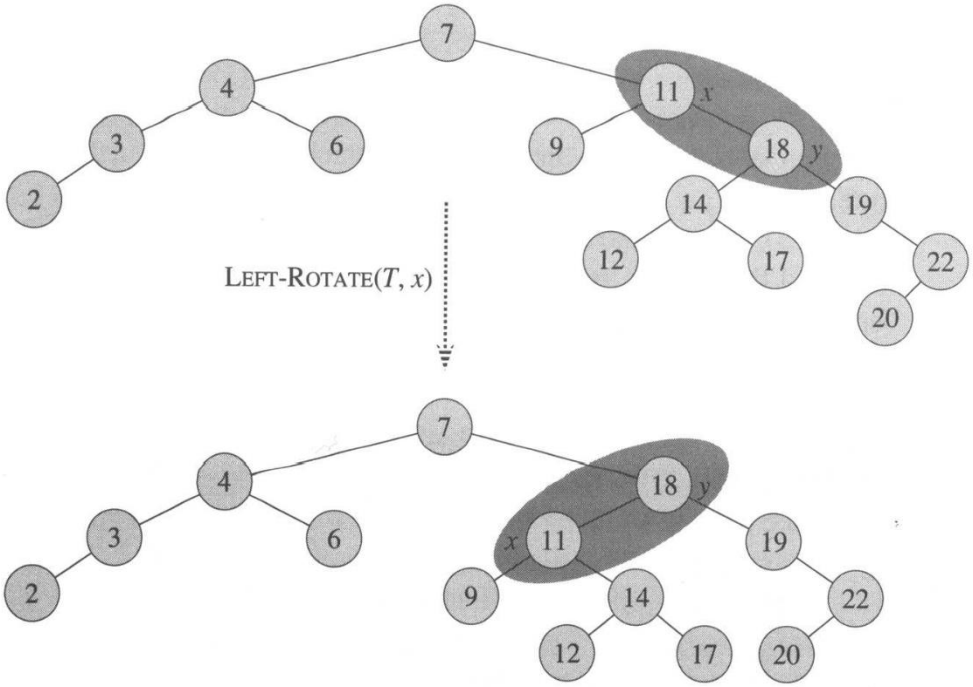
An example of LEFT-ROTATE(T,x)




A
X
Y
i
j
k
parent
child
1
2
3
4
5
6

LEFT-ROTATE(T,x)

```
    1  y ← right[x]
11  2  right[x] ← left[y]
&   3  p[left[y]] ← x
14
    4  p[y] ← p[x]
    5  If p[x] = nil[T]
7   6    then root[T] ← y
&   7    else if x = left[p[x]]
18  8         then left[p[x]] ← y
    9         else  right[p[x]] ← y
11 10  left[y] ← x
&  11  p[x] ← y
18
```



The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in O(1) time.

13.3 Insertion

RB-INSERT(T,z)

```
1  y ← nil[T]
2  x ← root[T]
3  while x ≠ nil[T]
4        do y ← x
5              if  key[z] < key[x]
6                 then  x ← left[x]
7                 else  x ← right[x]
8   p[z] ← y
```

- unsuccessful search

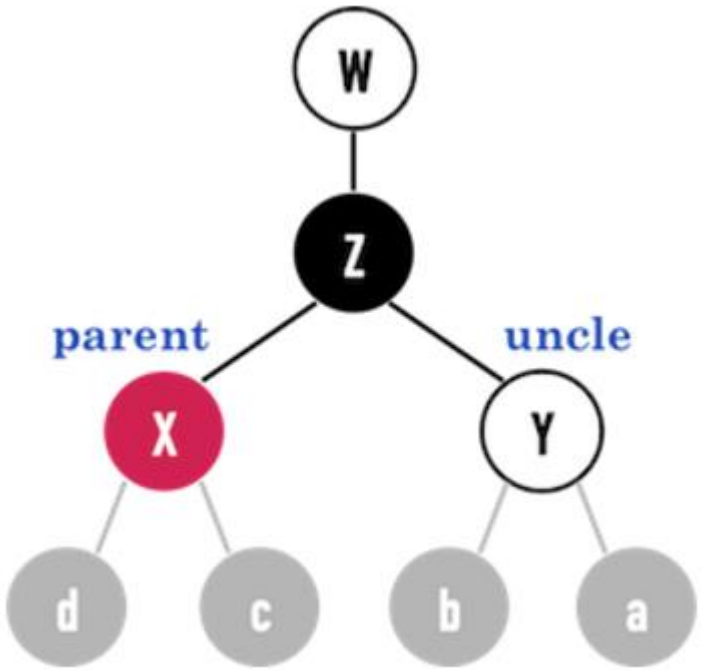
13.3 Insertion

```
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12                then left[y] ← z
13                else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

- tree T was empty
- link to child

new in RB-tree

RB-INSERT-FIXUP



Case 1: uncle is red

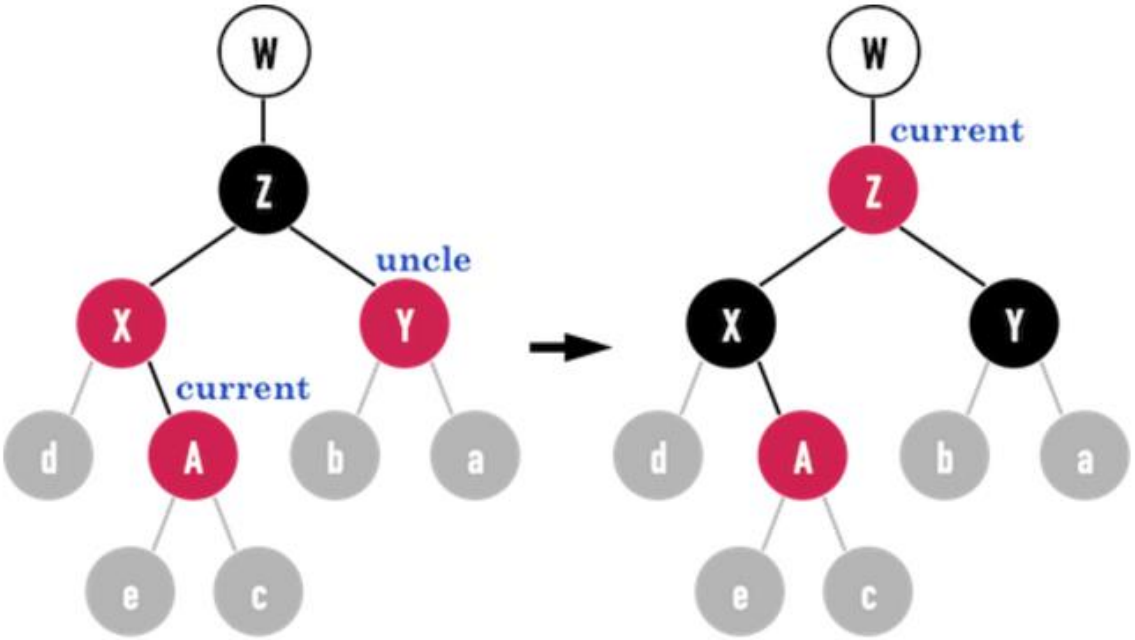
Case 2: uncle is black, inserted node is right(X)

Case 3: uncle is black, inserted node is left(X)

White means the node could be red or black.

PS: uncle can be nil (black)

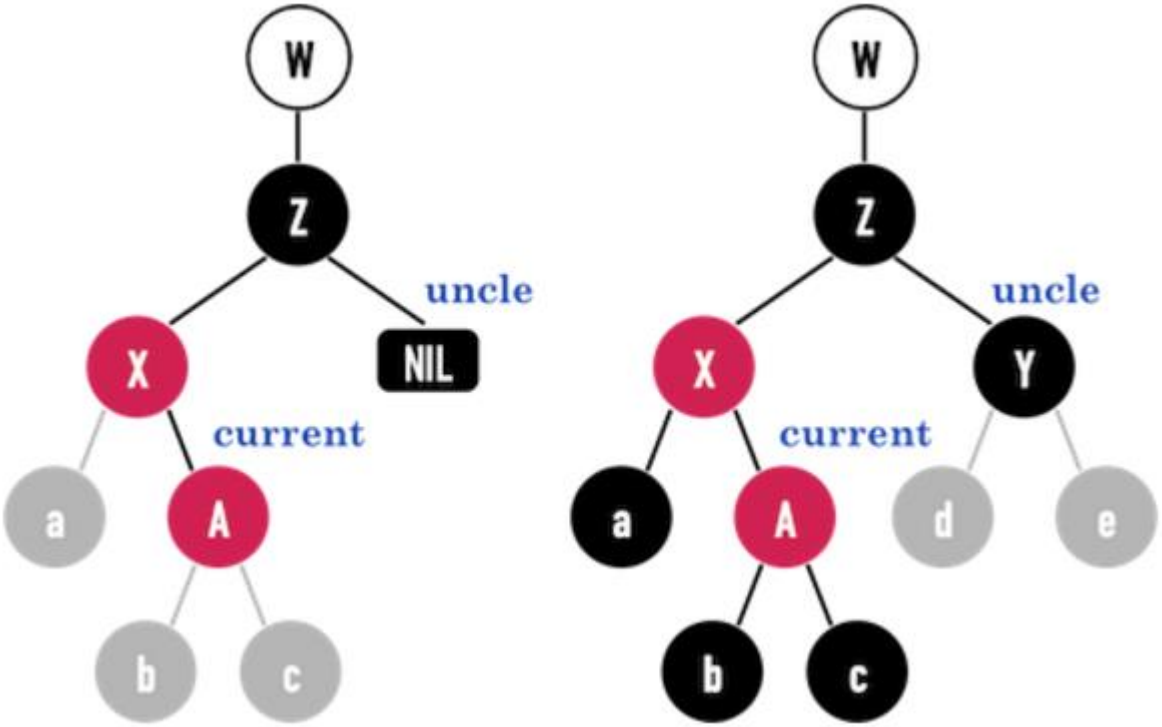
RB-INSERT-FIXUP – case 1



- parent turn black
- uncle turn black
- grandparent turn red
- current -> grandparent(current)

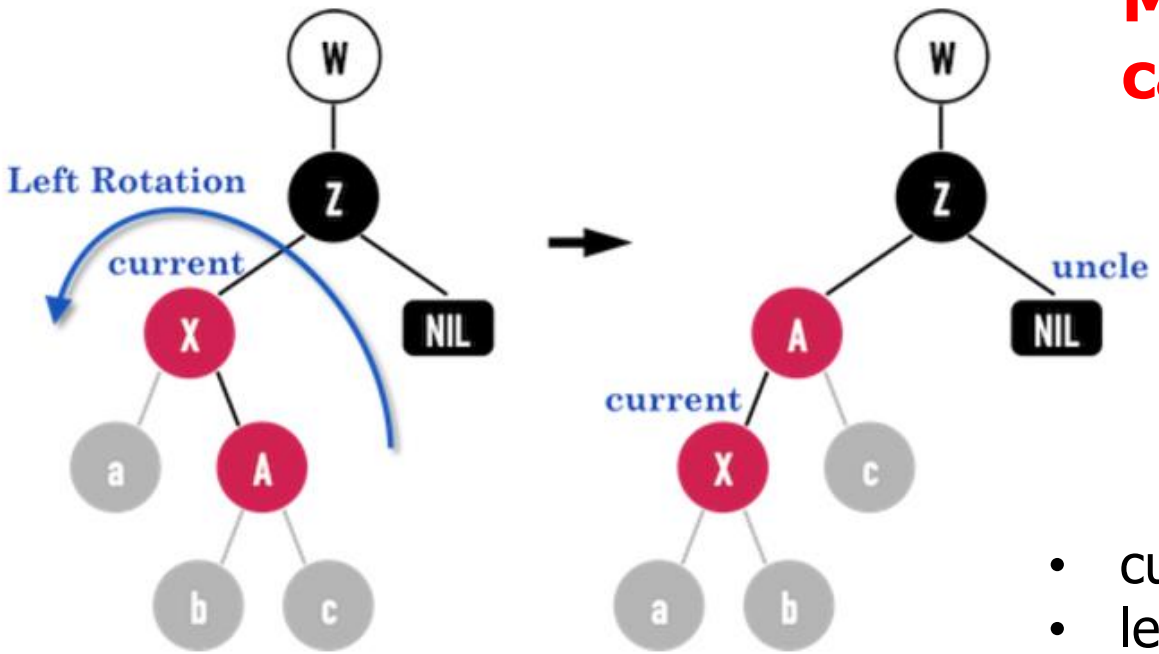
- check z (current) and w, and fix up iteratively.

RB-INSERT-FIXUP – case 2



If A is the newly inserted node, then uncle must be NIL. If not, uncle could be an ordinary node.

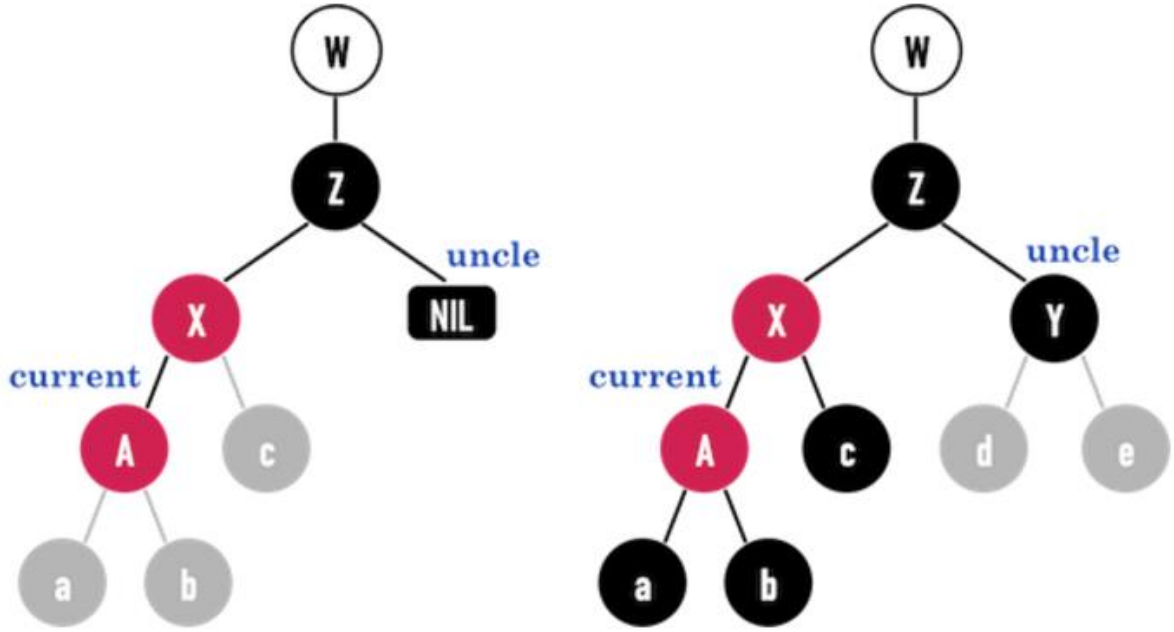
RB-INSERT-FIXUP – case 2



Make case 2 become case 3!

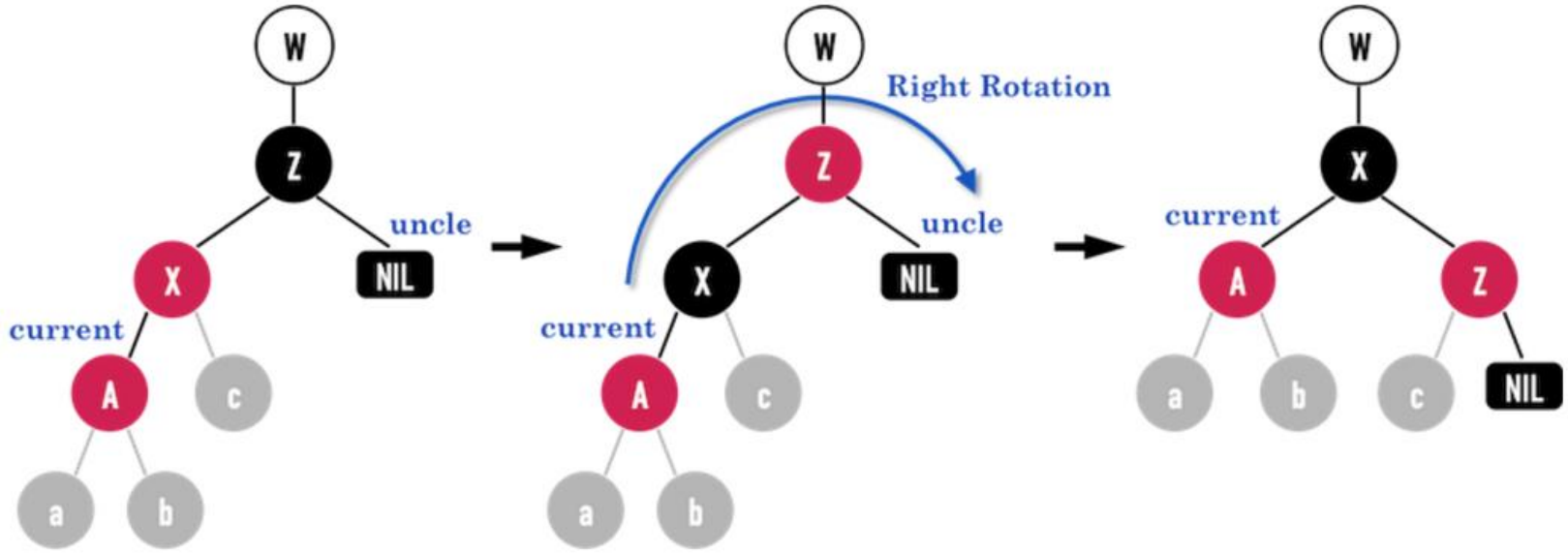
- current -> parent(current)
- left rotate at current (X)

RB-INSERT-FIXUP – case 3



Similar to case 2, if A is the newly inserted node, then uncle must be NIL. If not, uncle could be an ordinary node.

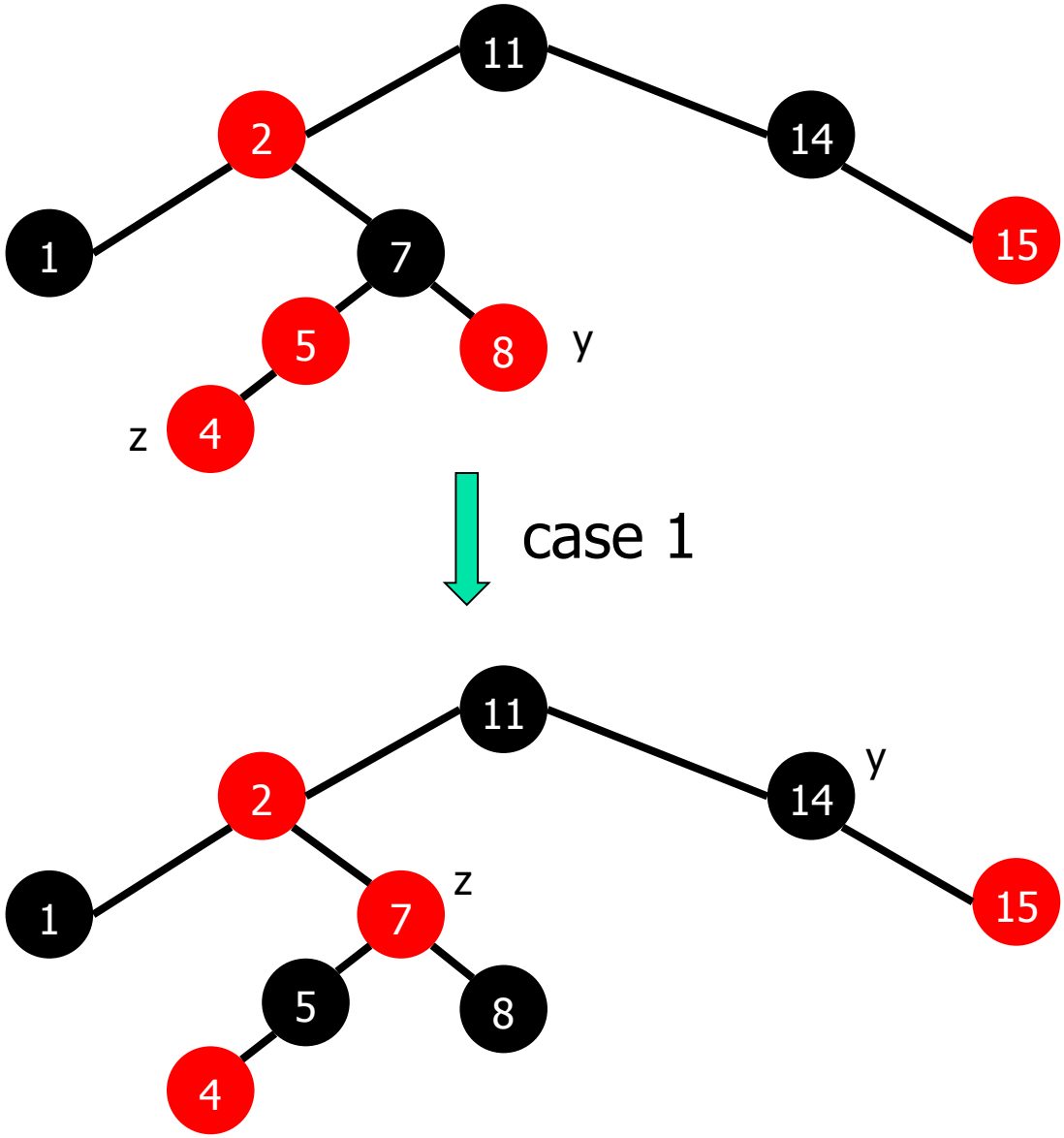
RB-INSERT-FIXUP – case 3

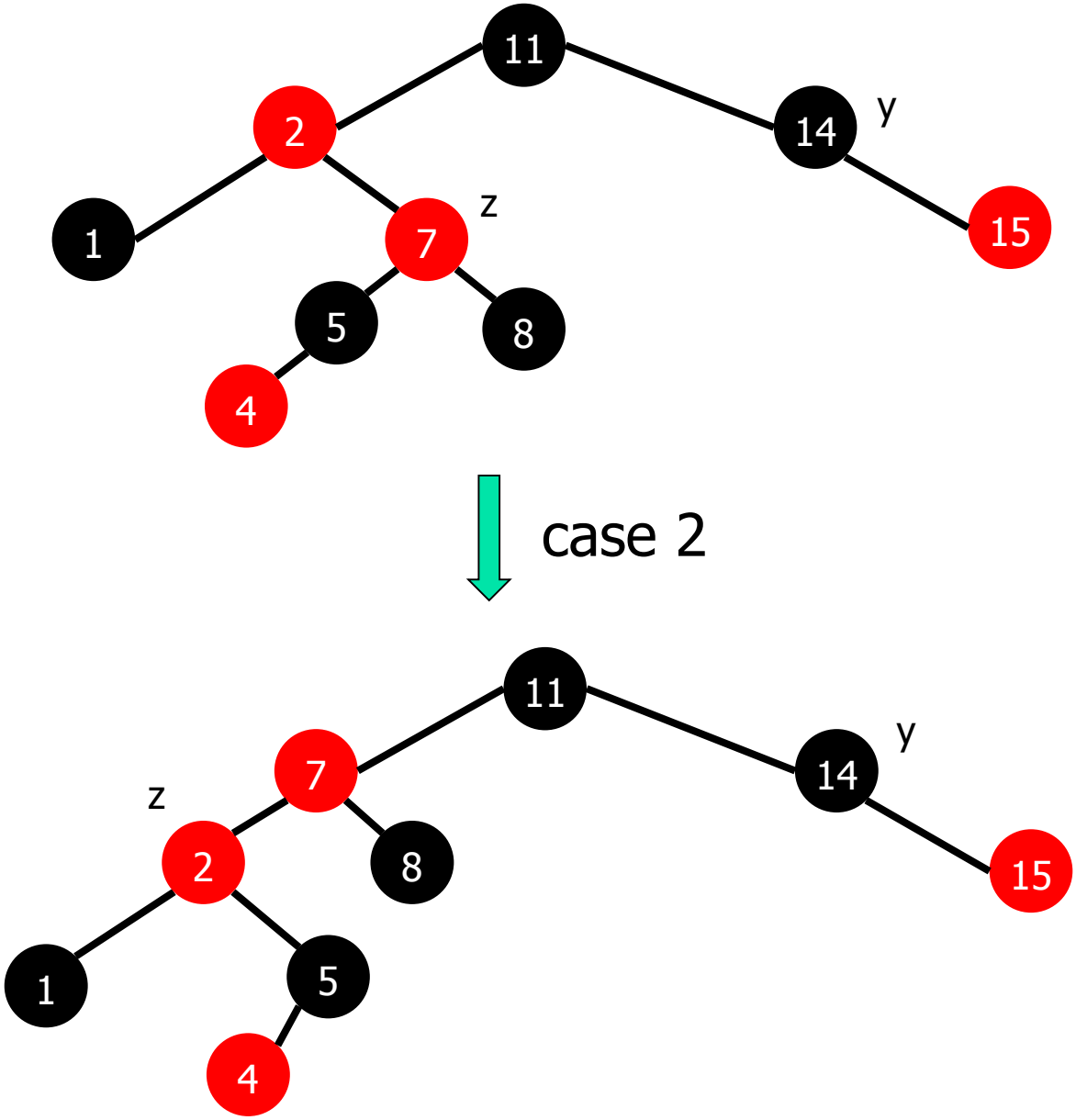


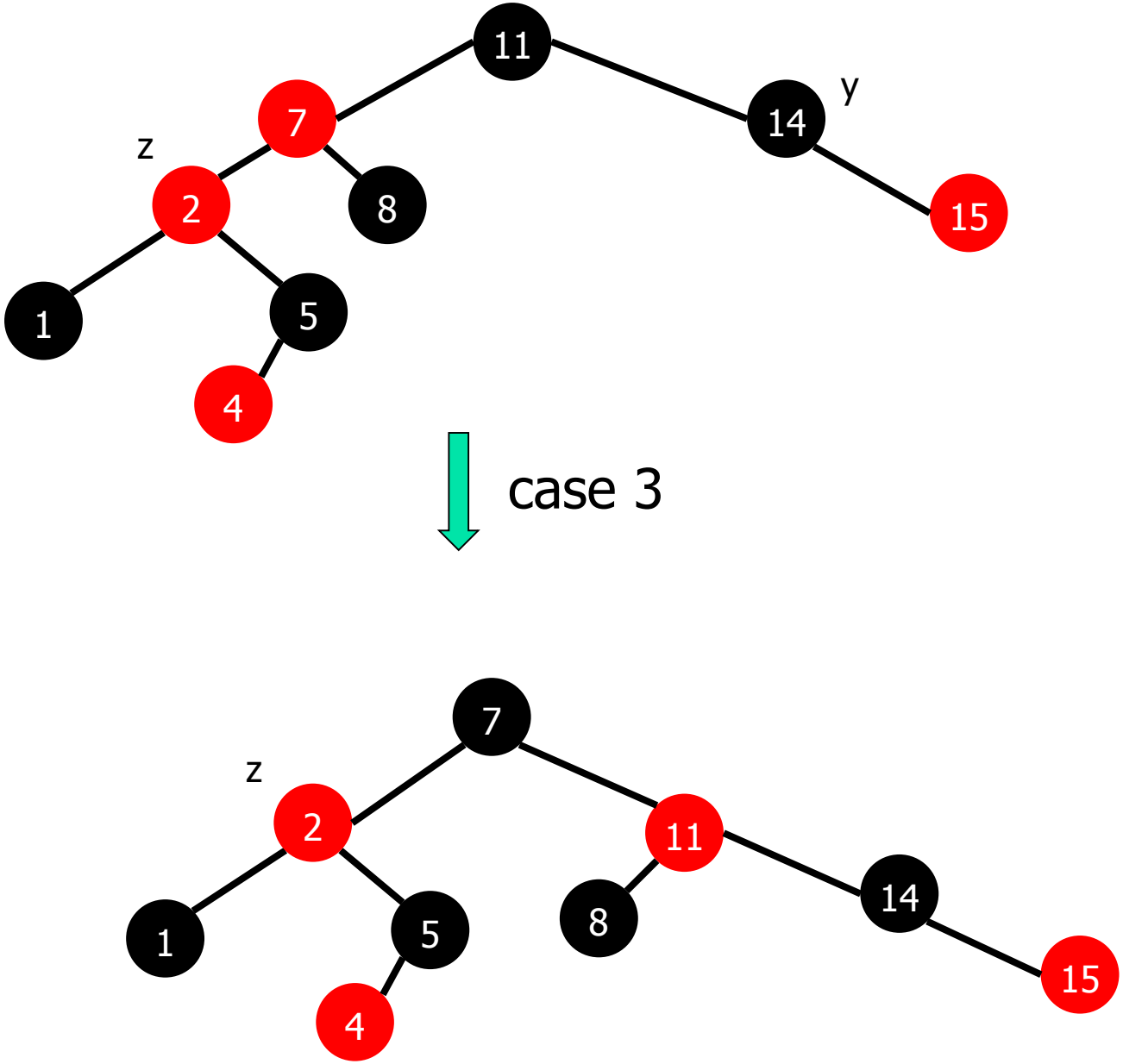
- parent (X) turn black
- grandparent (Z) turn red
- right rotate at grandparent (Z)

Why does it work?

In the beginning, let bh(A) be M, then bh(Z) is M+1. After the rotation, both bh(A) and bh(Z) are M.


11
2
14
1
7
15
5
8
y
z
4
case 1
11
2
14
y
1
7
z
15
5
8
4


11
2
14
y
1
7
z
15
5
8
4
case 2
11
7
14
y
z
2
8
15
1
5
4


11
7
14
y
z
2
8
15
1
5
4
case 3
7
z
2
11
1
5
8
14
4
15

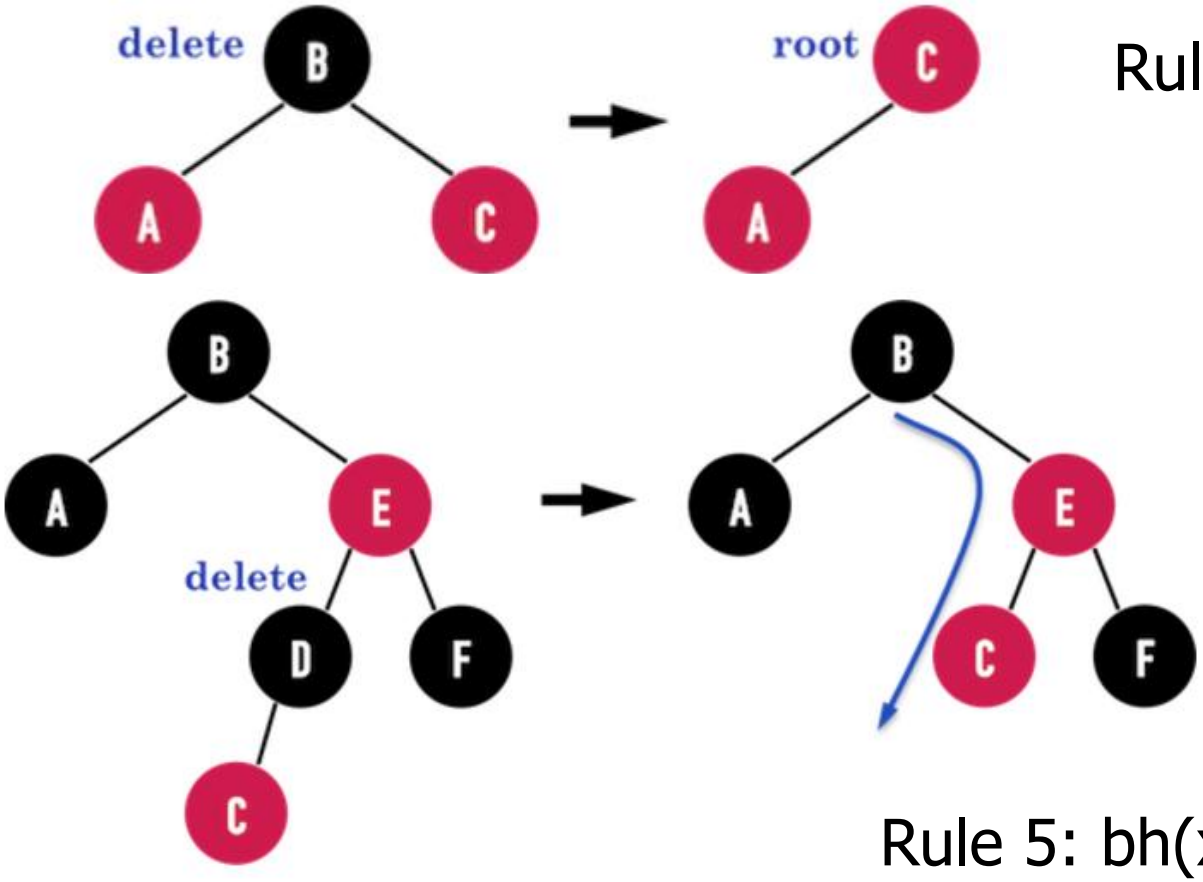
RB-INSERT-FIXUP(T,z)

```
1  while color[p[z]] = RED
2        do if p[z] = left[p[p[z]]]
3           then y ← right[p[p[z]]]
4                if  color[y] = RED
5                   then color[p[z]] ←BLACK                    Case 1
6                        color[y] ← BLACK                      Case 1
7                        color[p[p[z]]] ← RED                  Case 1
8                        z ← p[p[z]]                           Case 1
9                if else if z = right[p[z]]
10                  then z ← p[p[z]]                           Case 2
11                       LEFT-ROTATE(T,z)                      Case 2
12               color[p[z]] ← BLACK                           Case 3
13               color[p[p[z]]] ← RED                          Case 3
14               RIGHT-ROTATE(T,p[p[z]])                       Case 3
15       else (same as then clause with "right" and "left" exchanged)
16 color[root[T]] ← BLACK
```

Analysis

- RB-INSERT take a total of $O(\lg n)$ time.
- It never performs more than two rotations, since the while loop terminates if case 2 or case 3 executed.

13.4 Deletion



Rule 2: the root is black.

Rule 4: parent & child shouldn't be red simultaneously

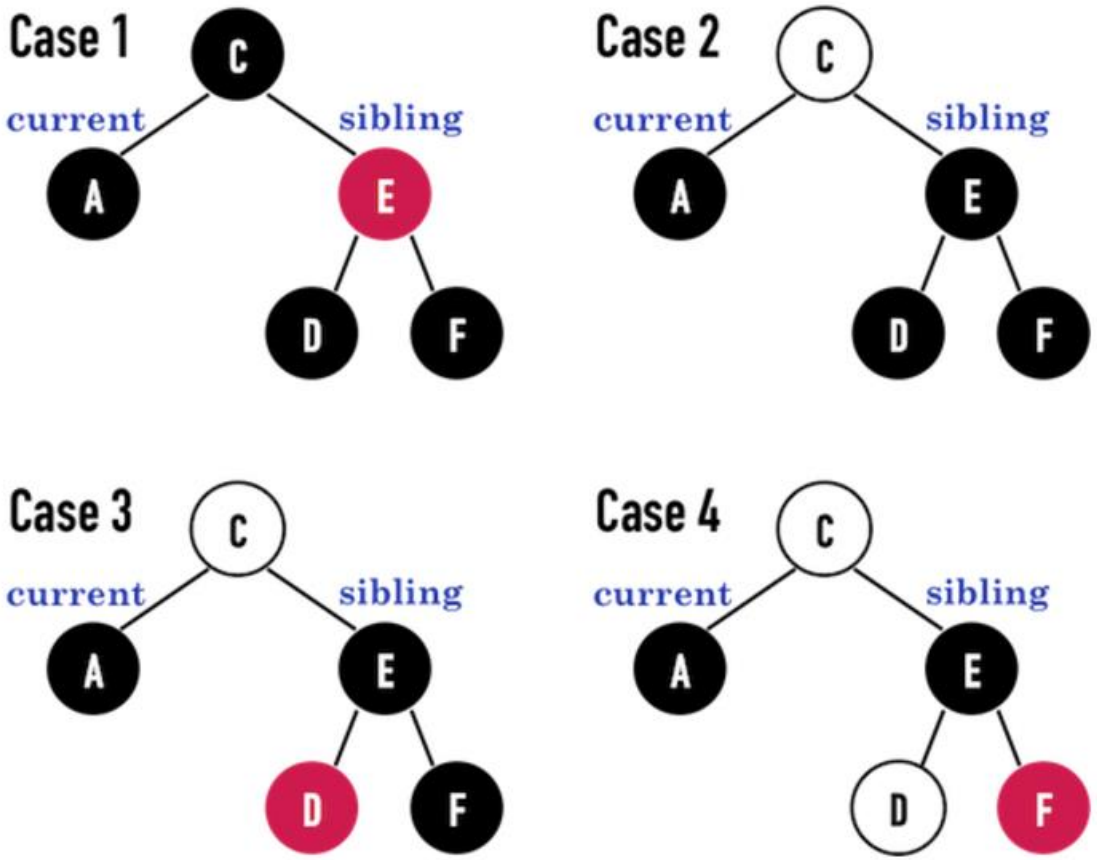
Rule 5: bh(x) of all paths from x to leaves are the same.

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]
 8  if p[y] = nil[T]
 9         then root[T] ← x
10      else if y = left[p[y]]
11                then left[p[y]] ← x
12                else right[p[y]] ← x
13      if y ≠ z
14    then key[z] ← key[y]
15           copy y's satellite data into z
16  if color[y] = BLACK
17       then RB-DELETE-FIXUP(T, x)
18  return y
```

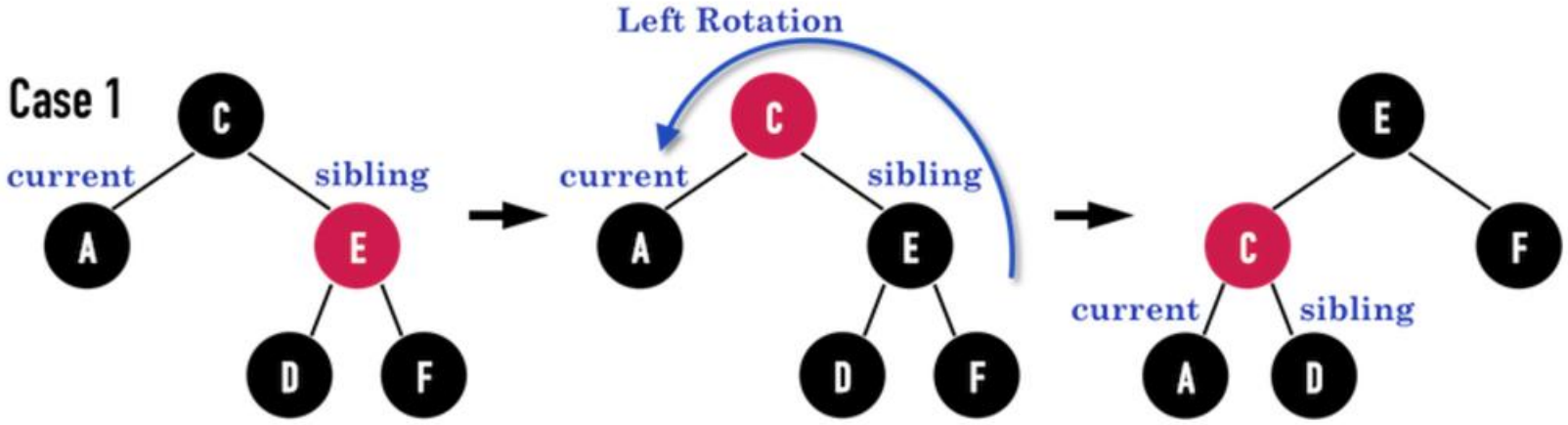
- one or no child
- two children
- set x to be y's child
- connect the child to its parent
- y is root
- y will be deleted, x becomes root
- connect parent to child

RB-DELETE-FIXUP



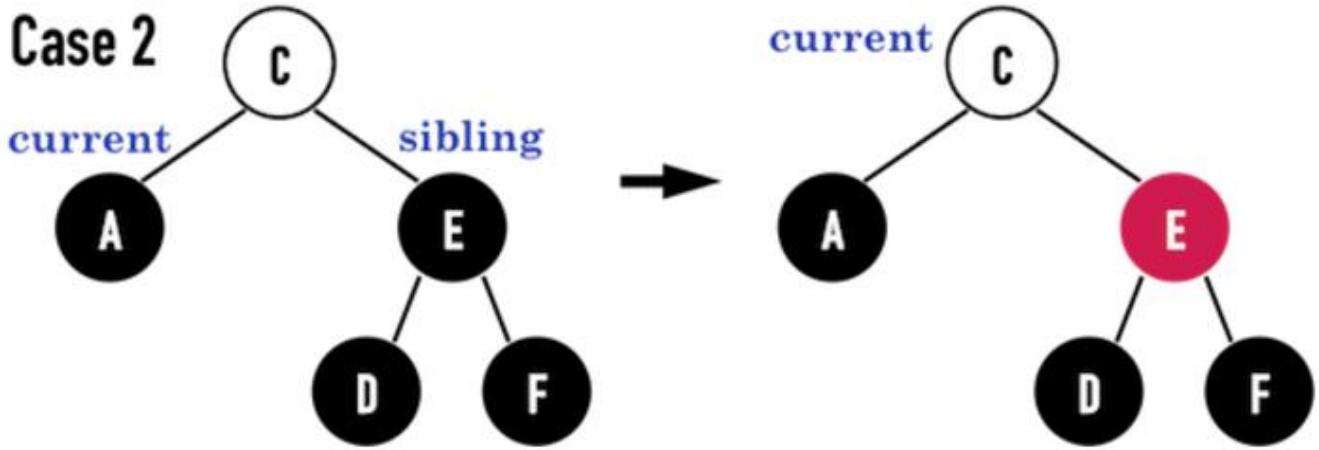
Delete A

Case 1. sibling is red
Case 2. sibling is black, and its children are black
Case 3. sibling is black, and its left child is red, right child is black
Case 4. sibling is black, and its right child is red



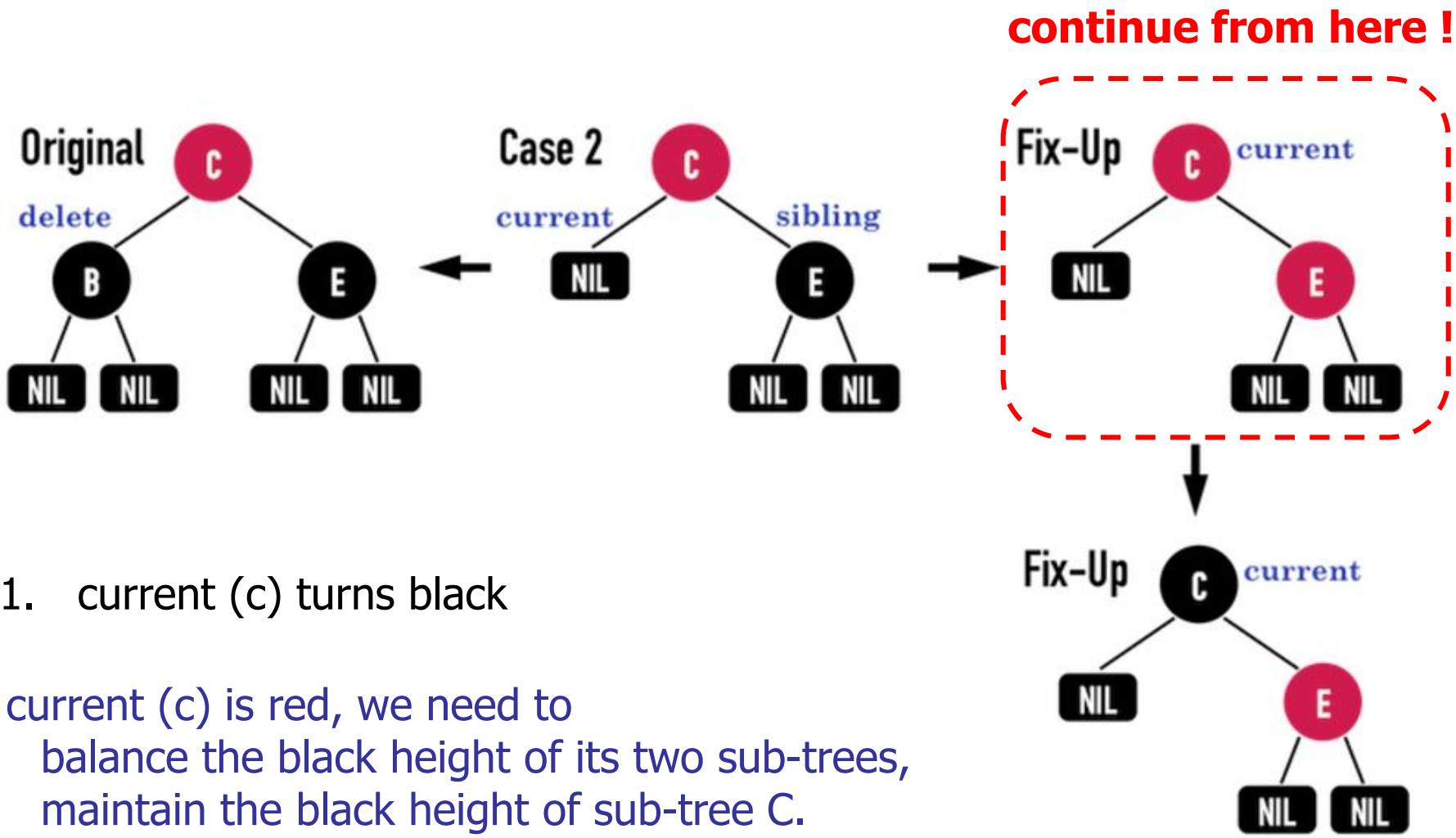
1. sibling turns black
2. current->parent turns red
3. left rotation at current->parent
4. update sibling's position

Since current node (A) is still black, the problem still exists, and will becomes case 2, 3, or 4.



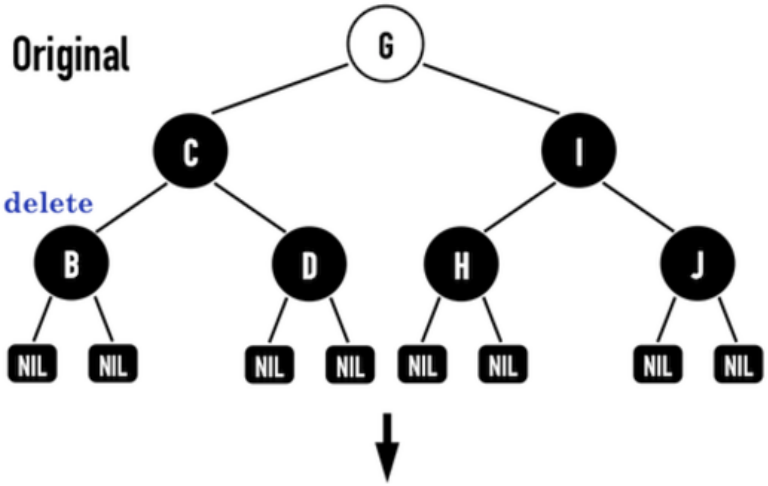
1. sibling turns red
2. current = current->parent

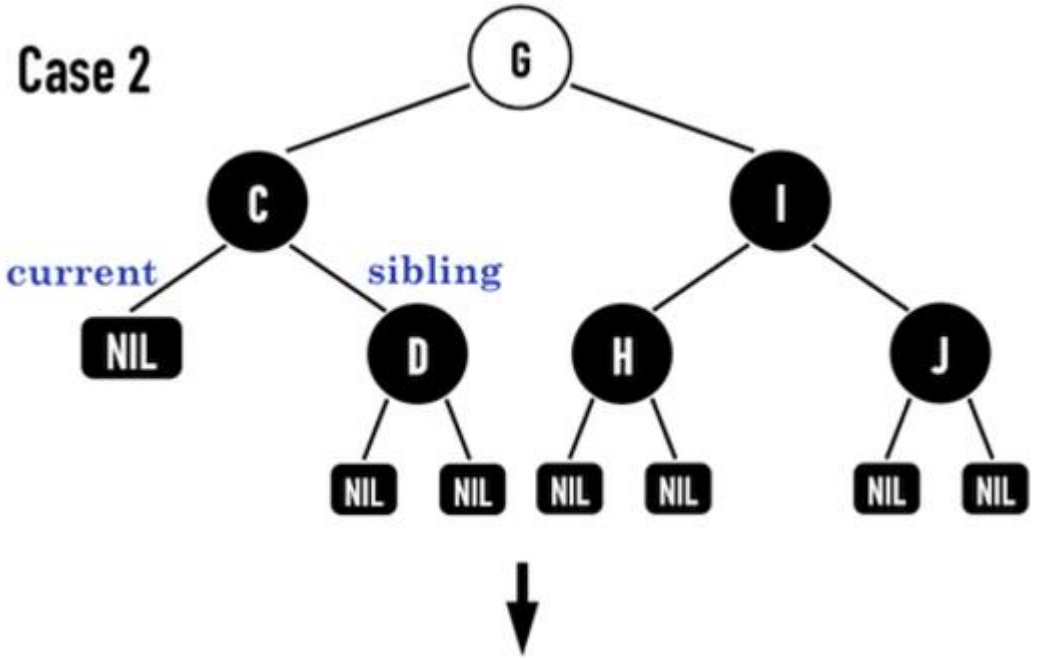
current (c) could be red or black after the operation

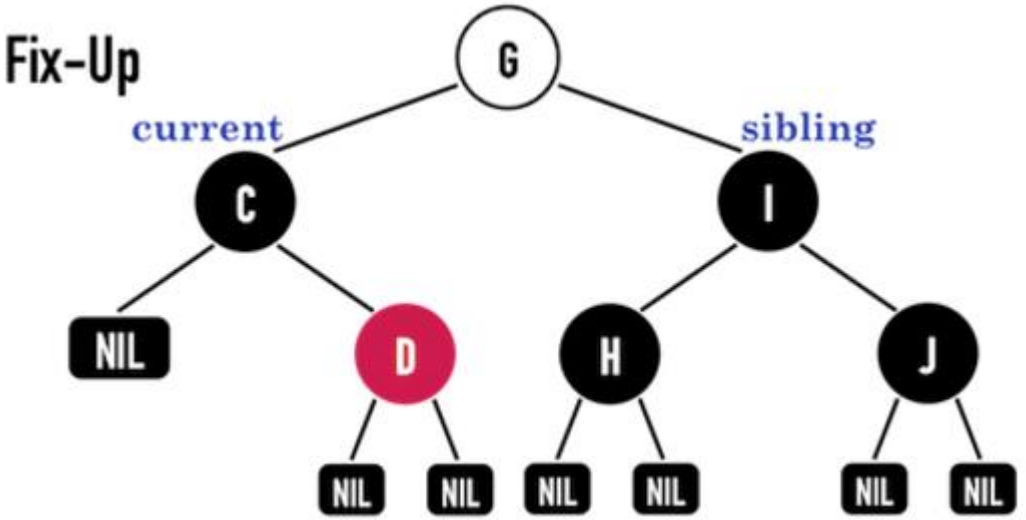


1. current (c) turns black

If current (c) is red, we need to
1. balance the black height of its two sub-trees,
2. maintain the black height of sub-tree C.

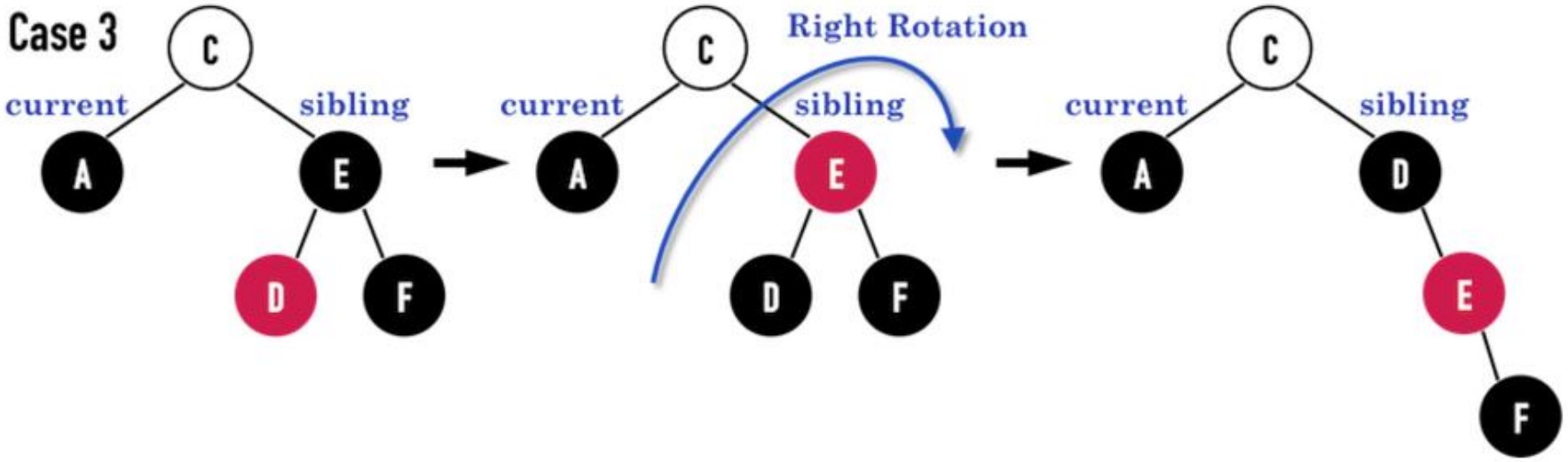






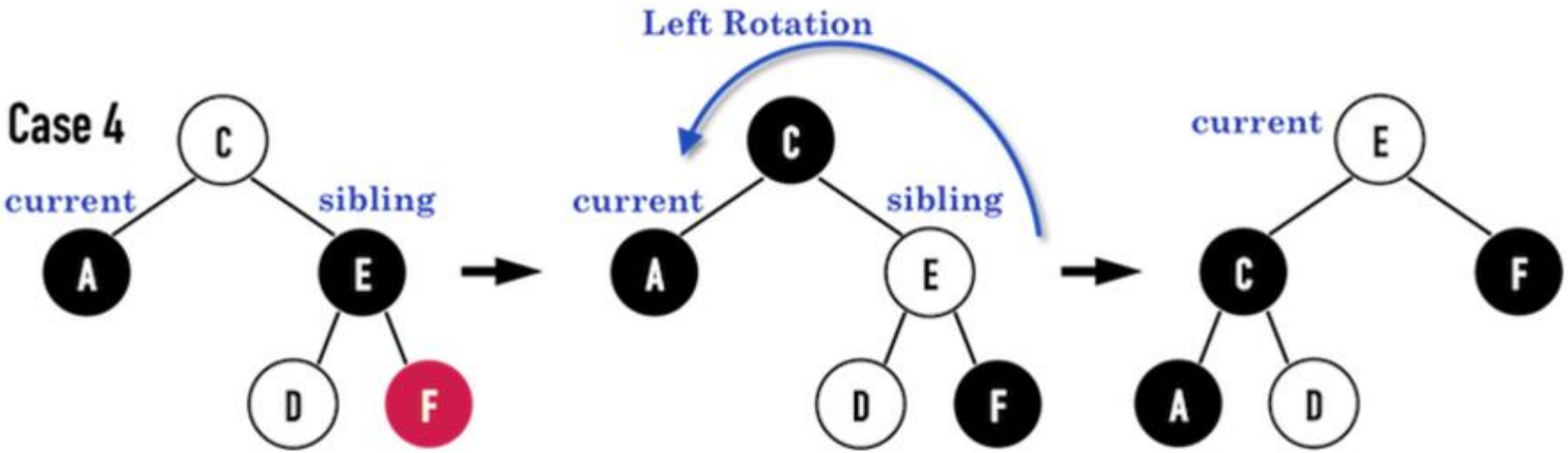
If current (c) is black, then we have to start over again. Note that bh(c) and bh(I) are different.

PS: go to case 2 in this example, but can be cases 3 and 4.



1. sibling -> left child turns black
2. sibling turns red
3. right rotation at sibling

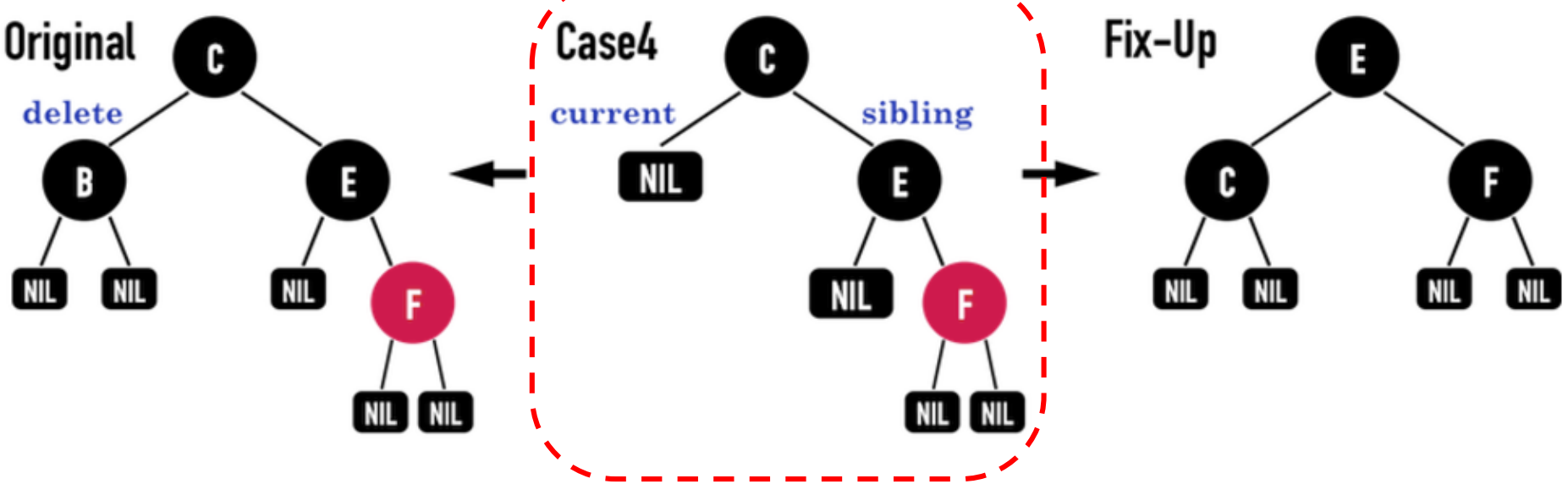
The goal is change case 3 to case 4.



1. exchange the color of sibling (E) and parent (C)
2. sibling right child (F) turns black
3. left rotate at parent (c)
4. current -> parent

The goal is to balance the black heights of sub-trees (c) after A is deleted.

An example


Original
C
delete
B
E
NIL
NIL
NIL
F
NIL
NIL
Case4
C
current
sibling
NIL
E
NIL
F
NIL
NIL
Fix-Up
E
C
F
NIL
NIL
NIL
NIL

RB-DELETE-FIXUP(T, x)

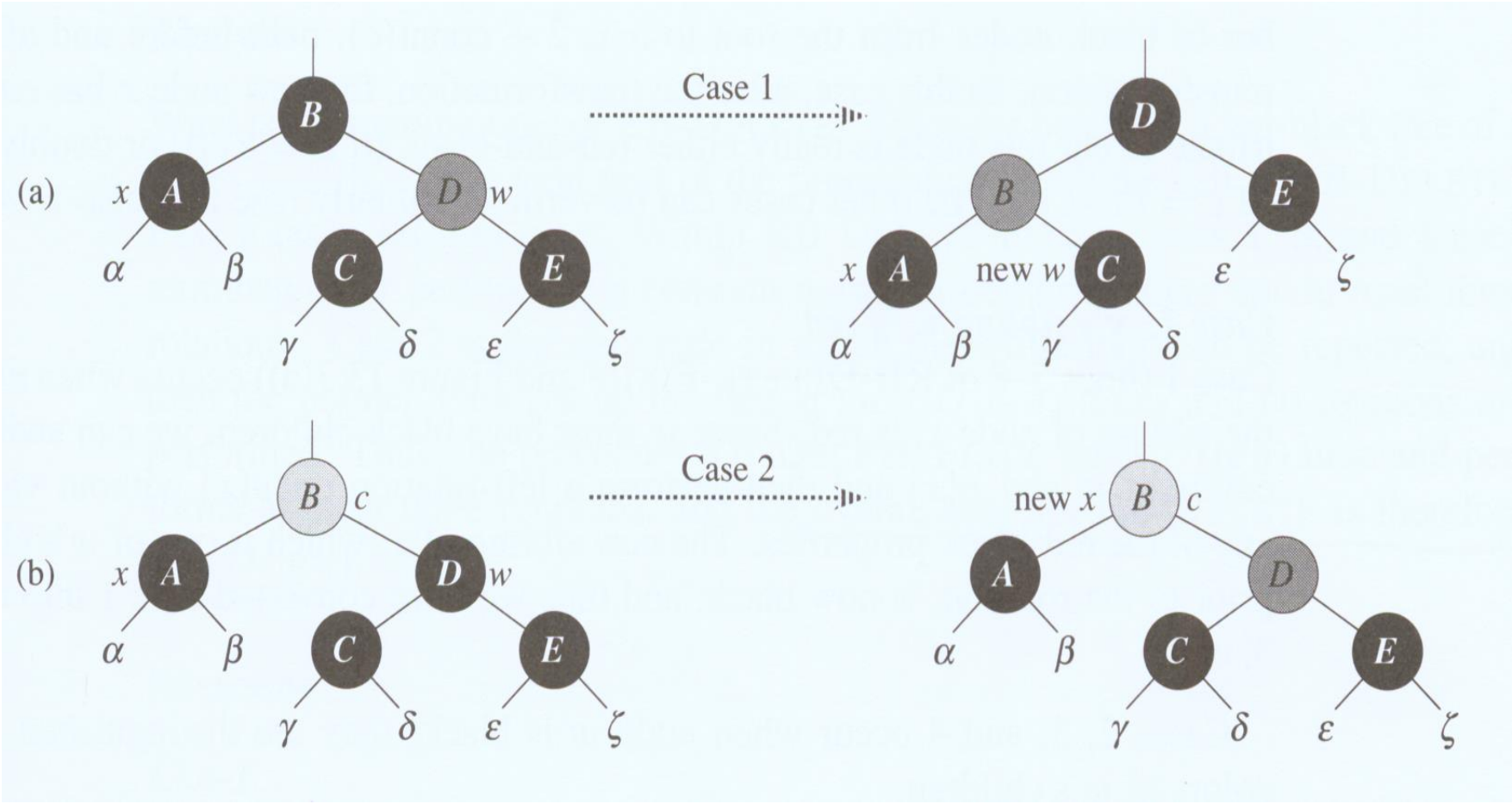
- RB-DELETE-FIXUP

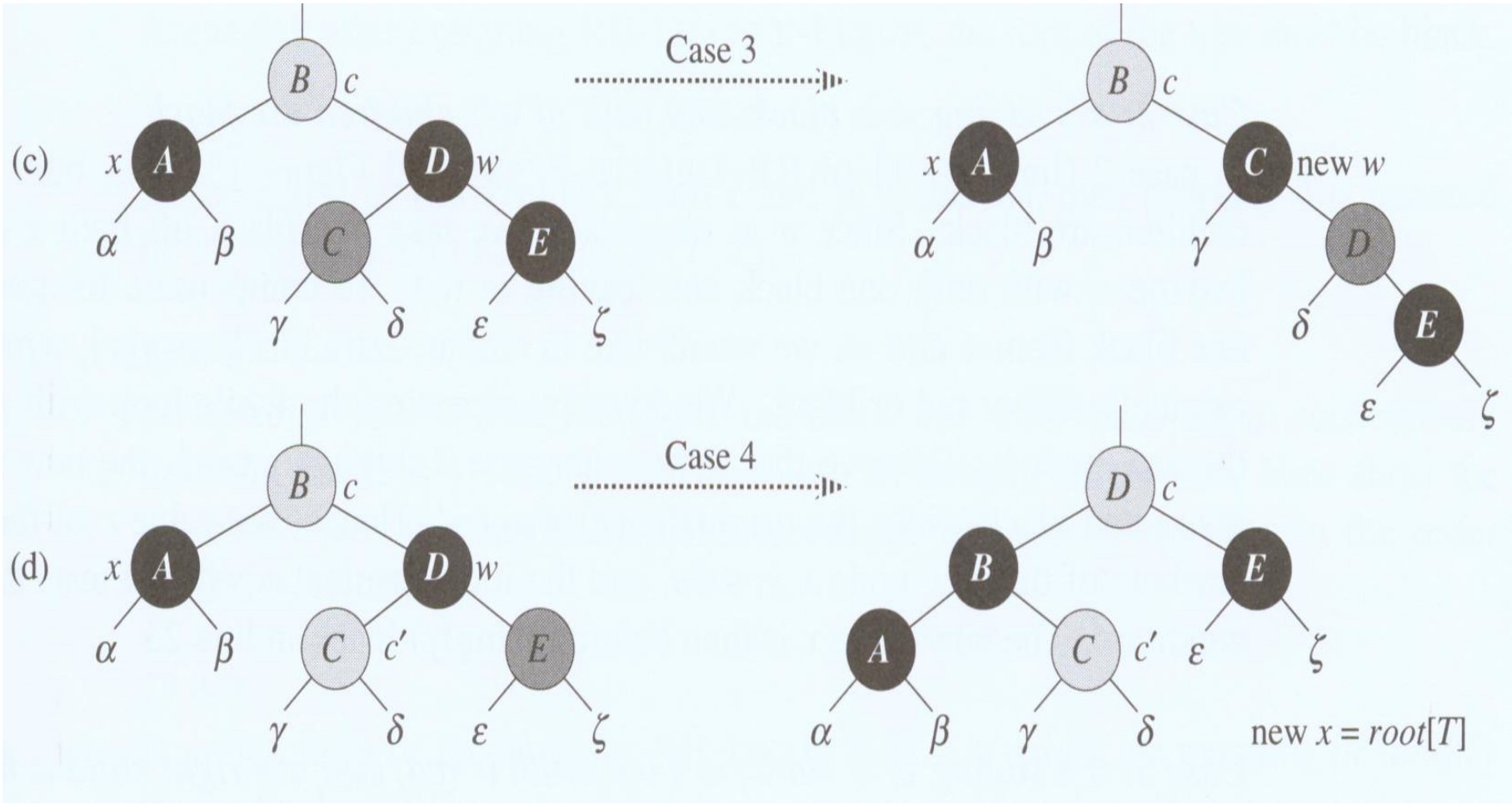
```
1   while x ≠ root[T] and color[x] = BLACK
2       do if x = left[p[x]]
3             then w ← right[p[x]]
4                  if color[w] = RED
5                        then color[w] ← BLACK          Case1
6                             color[p[x]] = RED         Case1
7                             LEFT-ROTATE(T,p[x])       Case1
8                             w ← right[p[x]]           Case1
```

```
 9        if color[right[w]] = BLACK and
                                  color[right[w]= BLACK
10           then color[w] ← RED                          Case2
11                x ← p[x]                                Case2
12           else if color[right[w]] = BLACK
13                then color[left[w]] ← BLACK             Case3
14                     color[w] ← RED                     Case3
15                     RIGHT-ROTATE(T,w)                  Case3
16                     w ← right[p[x]]                    Case3
17                color[w] ← color[p[x]]                  Case4
```

```
18                  color[p[x]] ← BLACK                    Case4
19                  color[right[w]] ← BLACK                Case4
20                  LEFT-ROTATE(T,p[x])                    Case4
21                  x ← root[T]                            Case4
22                  else (same as then clause with "right"
                          and "left" exchanged)
23  color[x] ← BLACK
```

the case in the while loop of RB-DELETE-FIXUP




(c)
B
c
x
A
D
w
α
β
C
E
γ
δ
ε
ζ
Case 3
B
c
x
A
C
new w
α
β
γ
D
δ
E
ε
ζ
(d)
B
c
x
A
D
w
α
β
C
c′
E
γ
δ
ε
ζ
Case 4
D
c
B
E
A
C
c′
ε
ζ
α
β
γ
δ
new x = root[T]

Analysis

- The RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations.
- The overall time for RB-DELETE is therefore also $O(\lg n)$