Algorithms
Chapter 13 Red-Black Trees

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Properties of red-black trees**
- Rotations
- Insertion
- Deletion

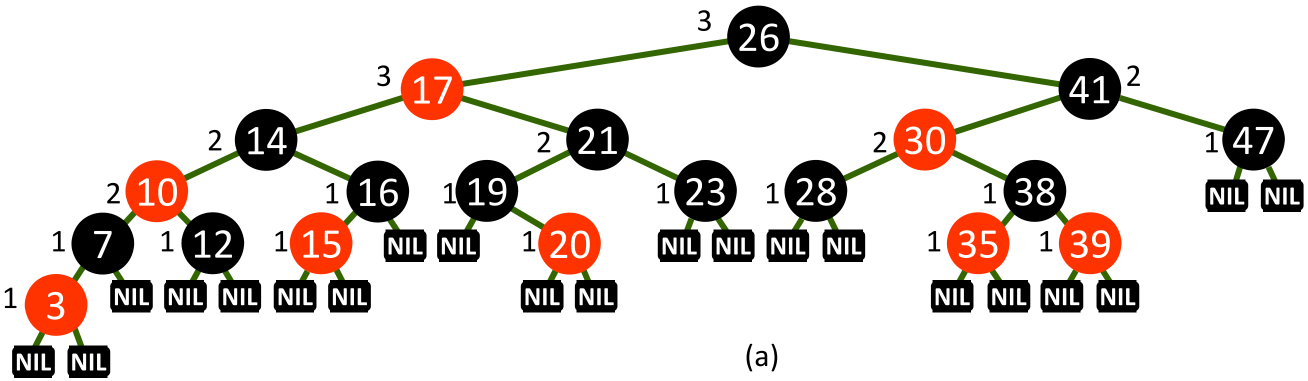
Overview

- A binary search tree of height h can implement any of the basic dynamic-set operations such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE in $O(h)$ time.
- Thus, the set operations are fast if the height of the search tree is small; but if its height is large, their performance may be no better than with a linked list.
- **Red-black trees**
  - A variation of binary search trees.
  - **Balanced**: height is $O(\lg n)$, where n is the number of nodes.
  - Operations will take $O(\lg n)$ time in the worst case.

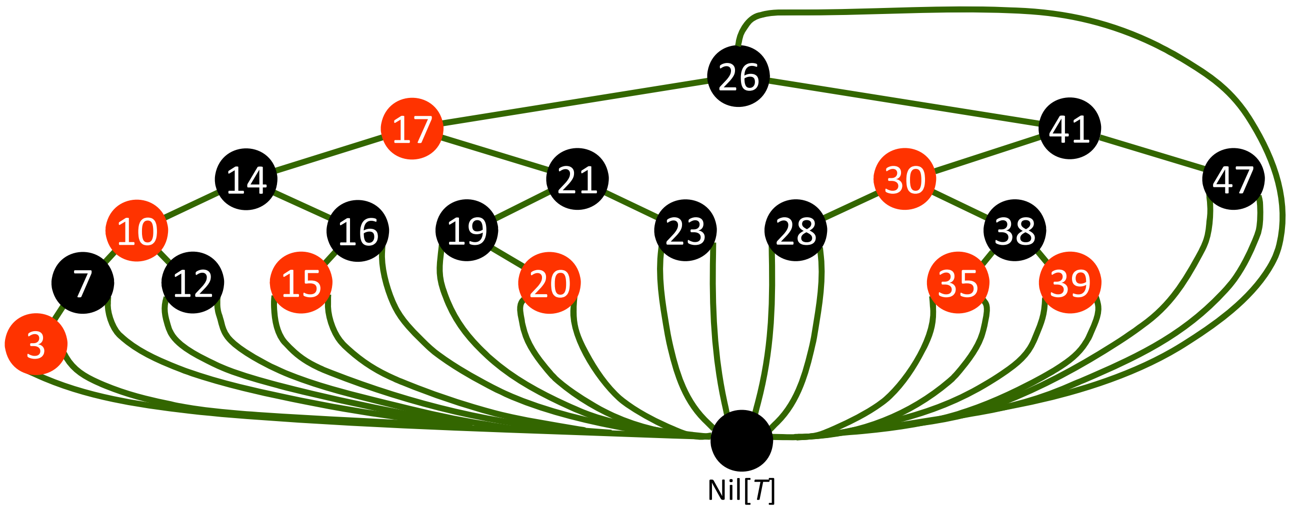
Properties of red-black trees$_{1/2}$

- A **red-black tree** = a binary search tree + 1 bit per node: an attribute **color**, which is either red or black.
  - 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.

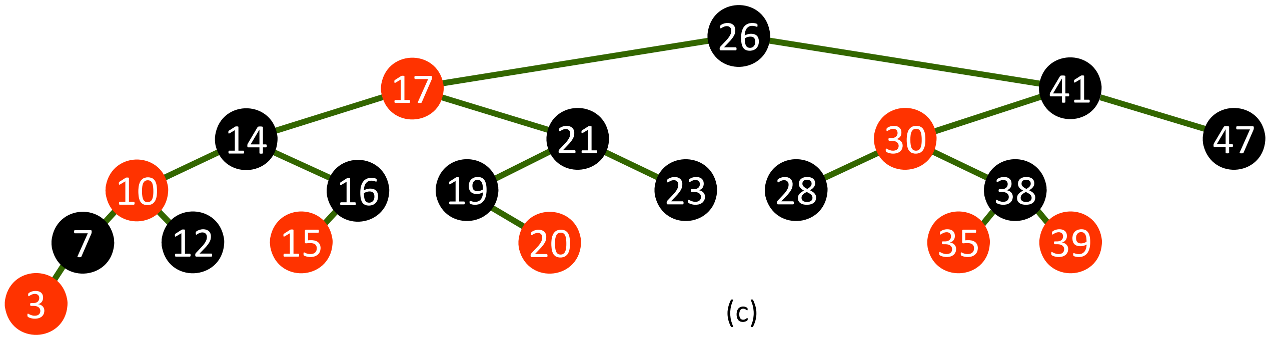
- **Red-black properties**
  1. Every node is either red or black.
  2. The root is black.
  3. Every leaf (Nil) is black.
  4. If a node is red, then both its children are black.
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.


26 17 41 14 21 30 47 10 16 19 23 28 38 7 12 15 20 35 39 3
3 3 2 2 2 2 1 2 1 1 1 1 1 1 1 1 1 1 1 1
NIL

(a)


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

(b)


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

(c)

Properties of red-black trees$_{2/2}$

- All leaves are empty (nil) and colored black.
  - We use a single sentinel, nil[T], for all the leaves as a matter of convenience.
  - color[nil[T]] is black.
  - The root's parent is also nil[T].
- **Height of a red-black tree**
  - **Height of a node** is the number of edges in a longest path to a leaf.
  - **Black-height** of a node x: bh(x) is the number of black nodes (including *nil*[T]) on the path from x to leaf, not counting x.
    By property 5, black-height is well defined.

The height of a red-black tree$_{1/2}$

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

  Proof: By induction on the height of *x*.

- **Basis:**
  - Height of $x = 0$ ➔ *x* is a leaf ➔ $\text{bh}(x) = 0$.
  - The subtree rooted at *x* has 0 internal nodes. $2^0 - 1 = 0$.
- **Inductive step:**
  - Let $\text{bh}(x) = b$.
  - Any child of *x* has black-height either *b* (if the child is red) or $b - 1$ (if the child is black).
  - By the inductive hypothesis, each child has $\geq 2^{\text{bh}(x)-1} - 1$ internal nodes.
  - Thus, the subtree rooted at *x* contains $\geq 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes. (The +1 is for *x* itself.)

The height of a red-black tree$_{2/2}$

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

  Proof:

  - Let h and b be the height and black-height of the root, respectively.
  - By Claim 1, we have $n \geq 2^b - 1$.
  - By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red.
  - Hence $\geq h/2$ are black, i.e., $b \geq h/2$.
  - Thus, $n \geq 2^b - 1 \geq 2^{h/2} - 1$.
  - This implies $h \leq 2\lg(n+1)$.

Operations on red-black trees

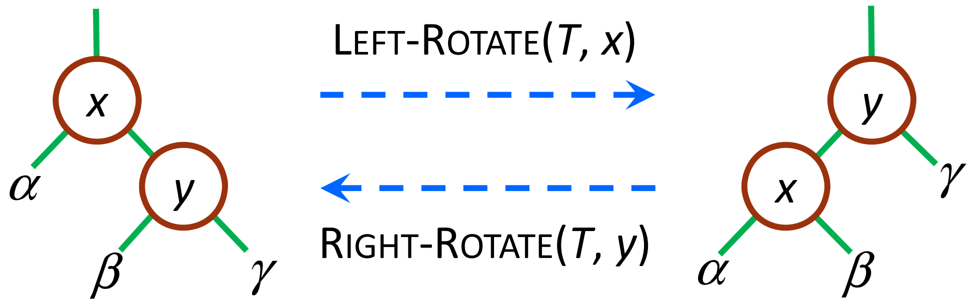
- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.
- **Insertion** and **deletion** are not so easy.
- For example:
  - If we insert, what color to make the new node?
    - Red? Might violate property 4.
    - Black? Might violate property 5.

Outline

- Properties of red-black trees
- **Rotations**
- Insertion
- Deletion

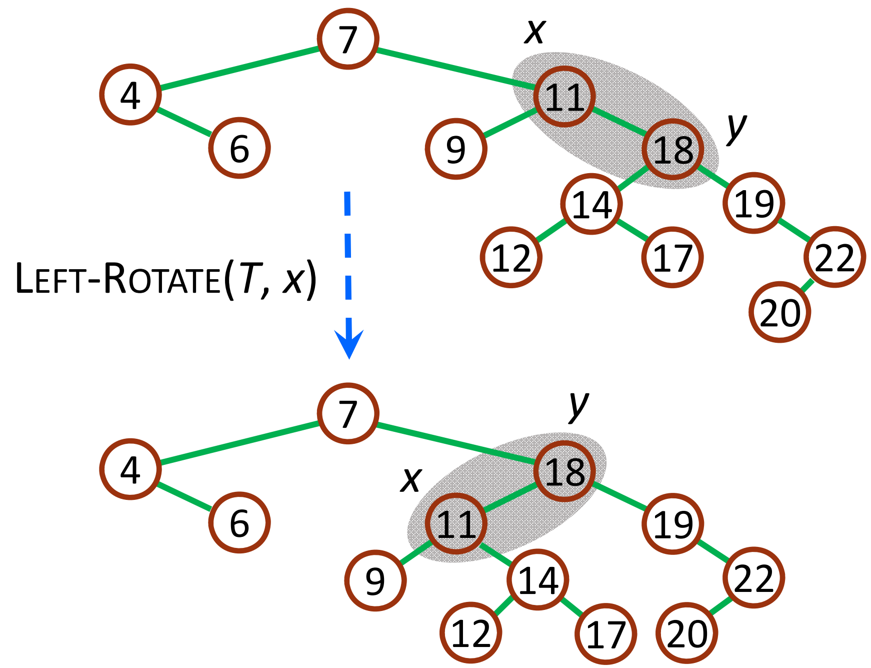
Rotations

- A local operation in a search tree that preserves the binary-search-tree property.
- There are two kinds of rotations:
  - **Left rotations** and **right rotations**.
  - They are inverses of each other.
- When we do a left rotation on a node *x*, we assume that its right child *y* is not nil[*T*].



LEFT-ROTATE pseudocode

```
LEFT-ROTATE(T, x)
    y ← right[x]
    right[x] ← left[y]
    if left[y] ≠ nil[T ]
        p[left[y]] ← x
    p[y] ← p[x]
    if p[x] = nil[T ]
        root[T ] ← y
    else if x = left[p[x]]
        left[p[x]] ← y
    else right[p[x]] ← y
    left[y] ← x
    p[x] ← y
```



- Time: $O(1)$.
- The code for RIGHT-ROTATE is symmetric.

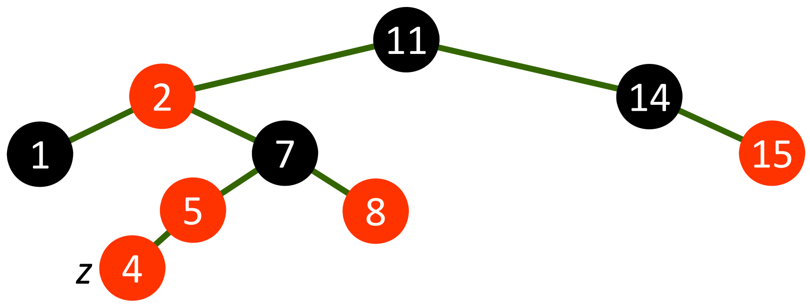
Outline

- Properties of red-black trees
- Rotations
- **Insertion**
- Deletion

RB-Insertion$_{1/2}$

- Start by doing regular binary-search-tree insertion.

```
RB-INSERT(T, z)
    y ← NIL; x ← root[T ]
    while x ≠ NIL
        y ← x
        if key[z] < key[x]
            x ← left[x]
        else x ← right[x]
    p[z] ← y
    if y = NIL
        root[T ] ← z      /* Tree T was empty */
    else if key[z] < key[y]
        left[y] ← z
    else right[y] ← z
    left[z] ← nil[T ]; right[z] ← nil[T ]; color[z] ← RED
    RB-INSERT-FIXUP(T, z)
```

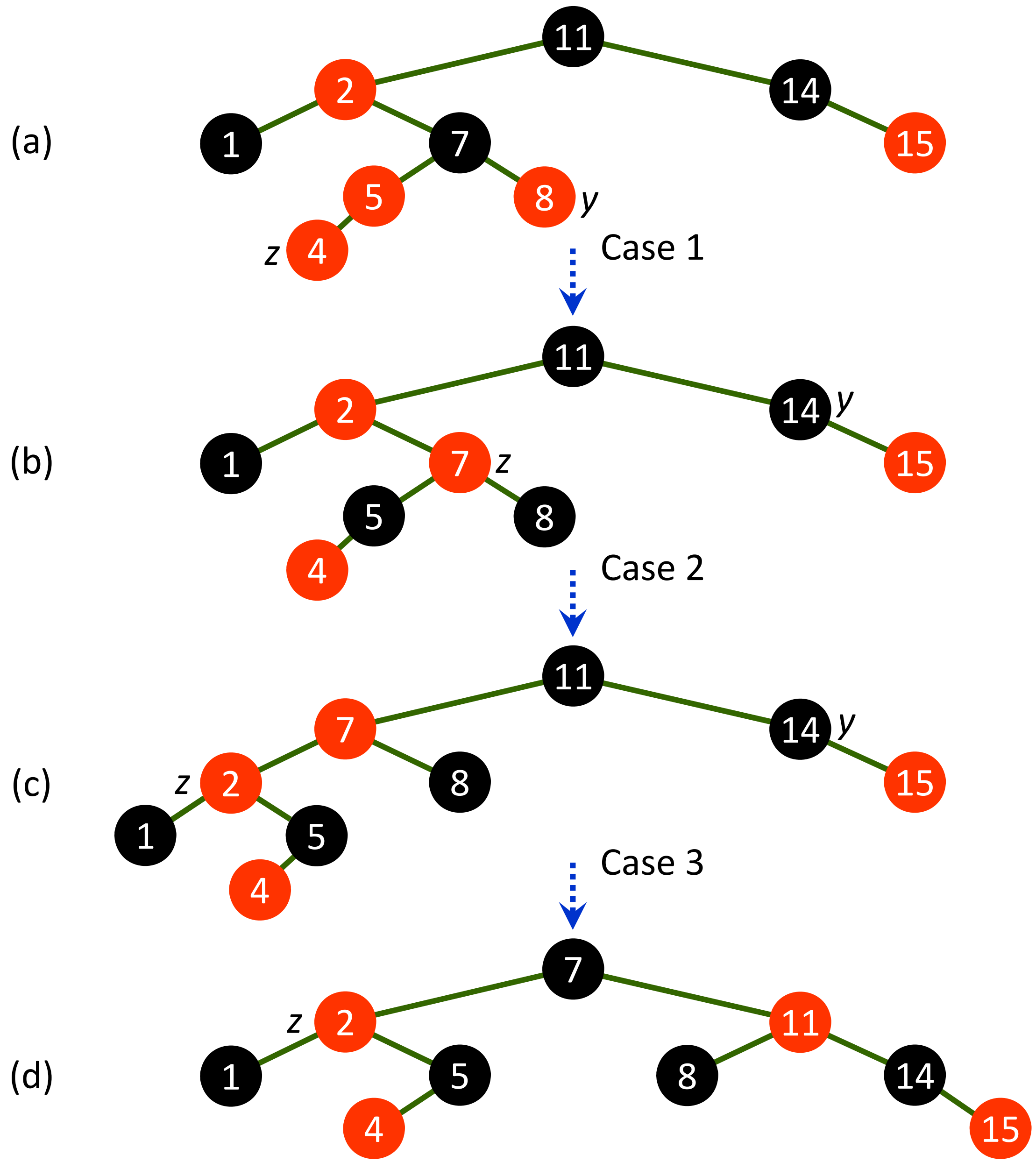


RB-Insertion$_{2/2}$

- 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 property might be violated?
  1. OK.
  2. **If *z* is the root**, then there's a violation. Otherwise, OK.
  3. OK.
  4. **If *p*[*z*] is red**, there's a violation: both *z* and *p*[*z*] are red.
  5. OK.

RB-INSERT-FIXUP procedure

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



(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

Correctness of RB-INSERT

- **Loop invariant**: At the start of each iteration of the **while** loop of lines 1-16,

  a. Node *z* is red.

  b. If *p*[*z*] is the root, then *p*[*z*] is black.

  c. There is at most one red-black violation:

    - Property 2, *z* is the root and is red.
    - Property 4, both *z* and *p*[*z*] are red.

- We omit the further details for proving the correctness.

Time complexity of RB-INSERT

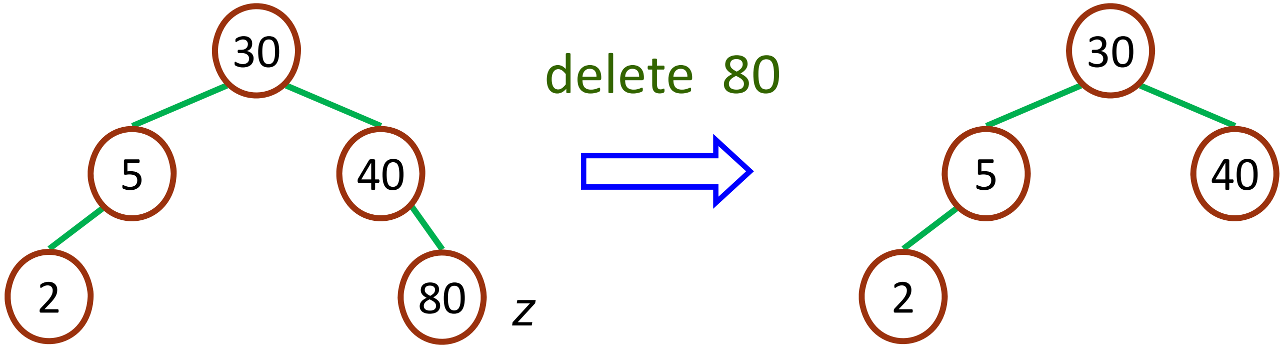
- **Analysis**:
  - Each iteration takes $O(1)$ time.
  - The **while** loop repeats only if case 1 is executed, and then the pointer z moves two levels up the tree.
  - The **while** loop terminates if case 2 or case 3 is executed.
  - $O(\lg n)$ levels ➔ $O(\lg n)$ time.
  - Also note that there are at most 2 rotations overall.

- Thus, insertion into a red-black tree takes $O(\lg n)$ time.

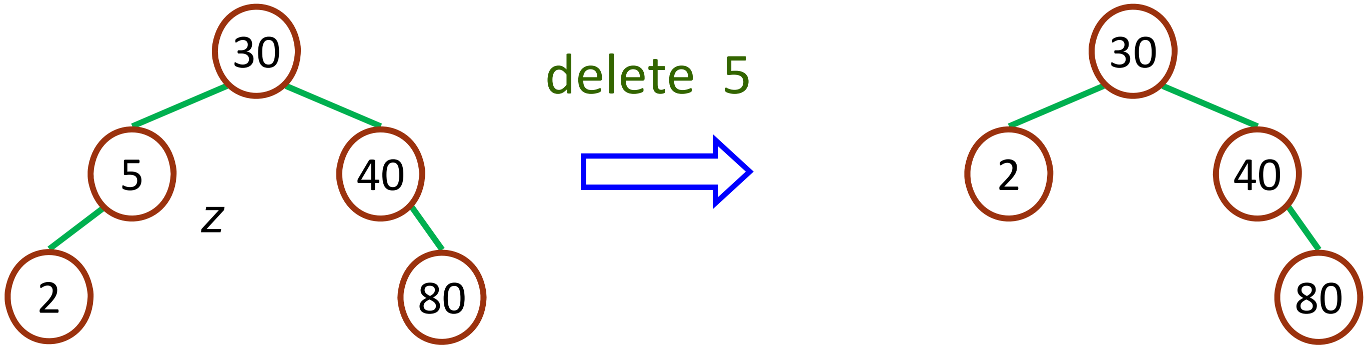
Outline

- Properties of red-black trees
- Rotations
- Insertion
- **Deletion**

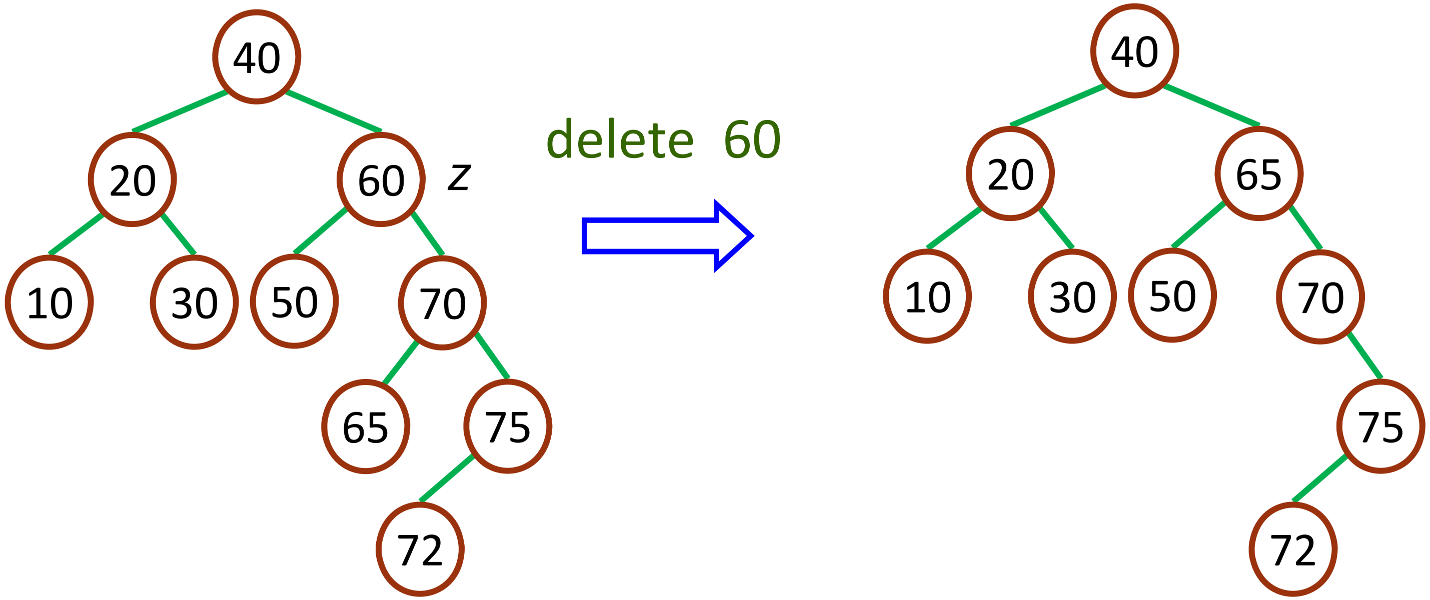
Case 1:



Case 2:

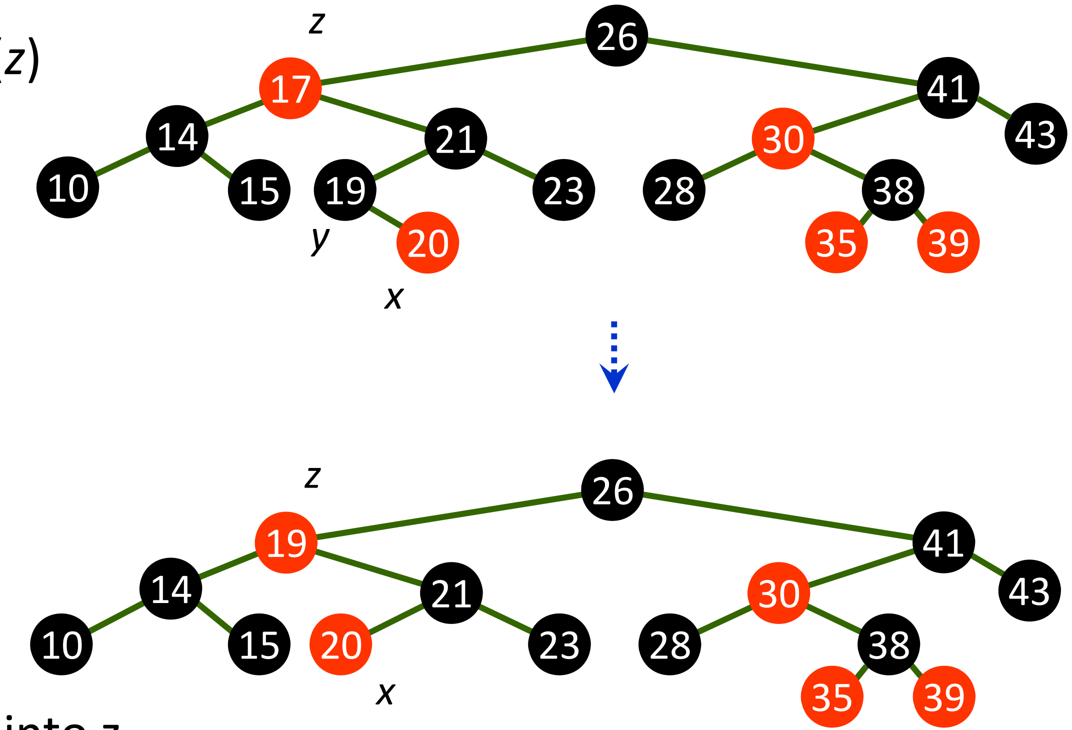


Case 3:



RB-Deletion$_{1/3}$

```
RB-DELETE(T, z)
    if left[z] = NIL or right[z] = NIL
        y ← z
    else  y ← TREE-SUCCESSOR(z)
    if left[y] ≠ NIL
        x ← left[y]
    else  x ← right[y]
    p[x] ← p[y]
    if p[y] = NIL
        root[T] ← x
    else if y = left[p[y]]
            left[p[y]] ← x
    else right[p[y]] ← x
    if y ≠ z
        key[z] ← key[y]
        copy y's satellite data into z
    if color[y] = BLACK
        RB-DELETE-FIXUP(T, x)
    return y
```



RB-Deletion$_{2/3}$

- *y* is the node that was actually spliced out.
- *x* is either
  - *y*'s sole non-sentinel child before *y* was spliced out, or
  - the sentinel, if *y* had no children.
- In both cases, *p*[*x*] is now the node that was previously *y*'s parent.
- If *y* is red, the red-black properties still hold when *y* is spliced out, for the following reasons:
  - no black-heights in the tree have changed,
  - no red nodes have been made adjacent, and
  - since *y* could not have been the root if it was red, the root remains black.

RB-Deletion$_{3/3}$

- If *y* is black, we could have violations of red-black properties:
  1. OK.
  2. If *y* is the root and *x* is red, then the root has become red.
  3. OK.
  4. Violation if *p*[*y*] and *x* are both red.
  5. Any path containing *y* now has 1 fewer black node.
- Correct this problem by giving *x* an "extra black".
  - Now property 5 is OK, but property 1 is not.
  - *x* is either **doubly black** or **red & black**.

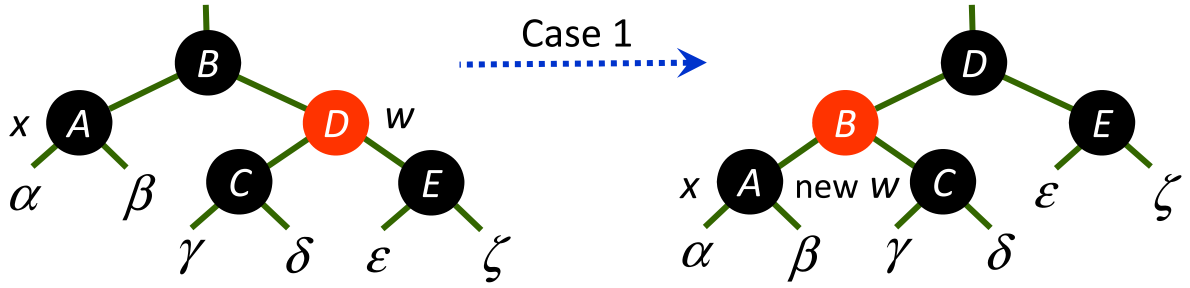
RB-DELETE-FIXUP

- **Idea:** Move the extra black up the tree until
  1. *x* points to a red & black node → turn it into a black node,
  2. *x* points to the root → just remove the extra black, or
  3. suitable rotations and recolorings can be performed.
- Within the while loop:
  - *x* always points to a nonroot doubly black node.
  - *w* is *x*'s sibling.
  - *w* cannot be *nil*[*T*], since that would violate property 5 at *p*[*x*].
- There are 8 cases, 4 of which are symmetric to the other 4.
- As with insertion, the cases are not mutually exclusive. We'll look at cases in which *x* is a left child.

RB-DELETE-FIXUP procedure

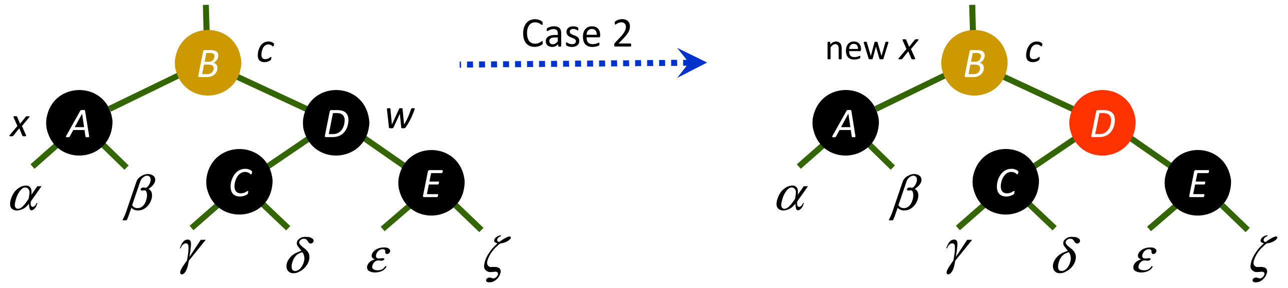
```
RB-DELETE-FIXUP(T, x)
    while x ≠ root[T ] and color[x] = BLACK
        if x = left[p[x]]
            w ← right[p[x]]
            if color[w] = RED
                color[w] ← BLACK                                   Case 1
                color[p[x]] ← RED                                  Case 1
                LEFT-ROTATE(T, p[x])                               Case 1
                w ← right[p[x]]                                    Case 1
            if color[left[w]] = BLACK and color[right[w]] = BLACK
                color[w] ← RED                                     Case 2
                x ← p[x]                                           Case 2
            else  if color[right[w]] = BLACK
                    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
```

Case 1: *w* is red



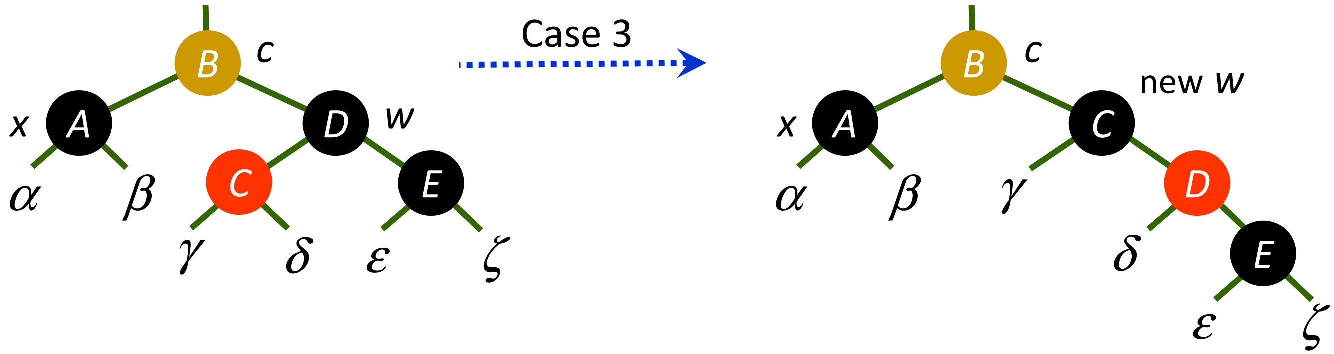
- *w* must have black children.
- Make *w* black and *p*[*x*] red.
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation ➔must be black.
- Go immediately to case 2, 3, or 4.

Case 2: *w* is black & both of *w*'s children are black



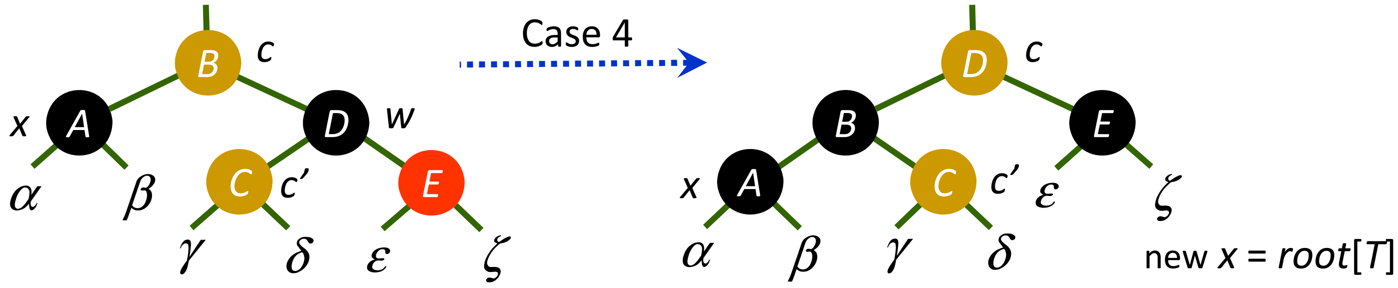
- Take 1 black off *x* (➔singly black) and off *w* (➔red).
- Move that black to *p*[*x*].
- Do the next iteration with *p*[*x*] as the new *x*.
- If entered this case from case 1, then *p*[*x*] was red ➔ new *x* is red & black ➔ color attribute of new *x* is RED ➔ loop terminates. Then new *x* is made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

w is black,
Case 3: *w*'s left child is red, and *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, and *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.

Time complexity of RB-DELETE

- **Analysis**:
  - 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 ➔ ≤ 3 rotations in all.

- Thus, the overall time for RB-DELETE is therefore $O(\lg n)$.

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html (Red-black Tree Animation)