Algorithms
Chapter 13 Red-Black Trees

可控制高度的二元樹

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

binary search tree 所需時間為 O(h)

h = { n 最差，跟 linked list 一樣
lgn 最好

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.

紅黑樹：binary search tree + color attribute
可控制高度在 lgn 的常數倍

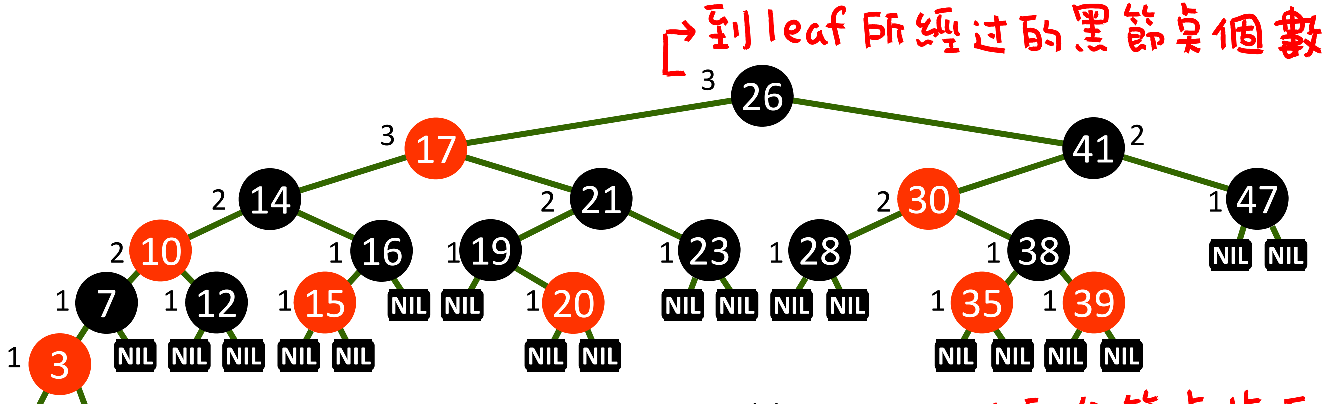
Properties of red-black trees$_{1/2}$

parent
key + color
left child right child

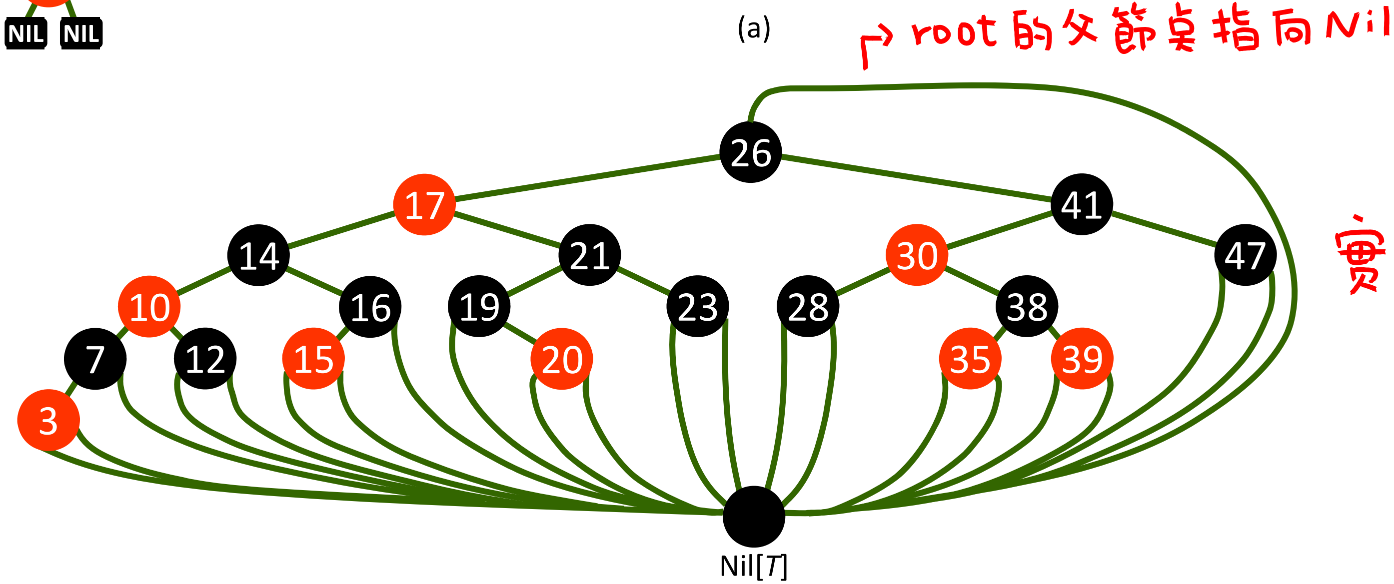
- A **red-black tree** = a binary search tree + 1 bit per node: an attribute **color**, which is either red or black.
  - Each node of the tree now contains the fields **color**, **key**, **left**, **right**, and **p**.
  - If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL.

- **Red-black properties** 性質
  1. Every node is either red or black. 每点不是紅就是黑
  2. The root is black. 根節点必為黑
  3. Every leaf (Nil) is black. 葉子必為黑
  4. If a node is red, then both its children are black. 若自己為紅 ⇒ 兩子節点必為黑 (紅紅不能相連)
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes. (黑高要一樣)

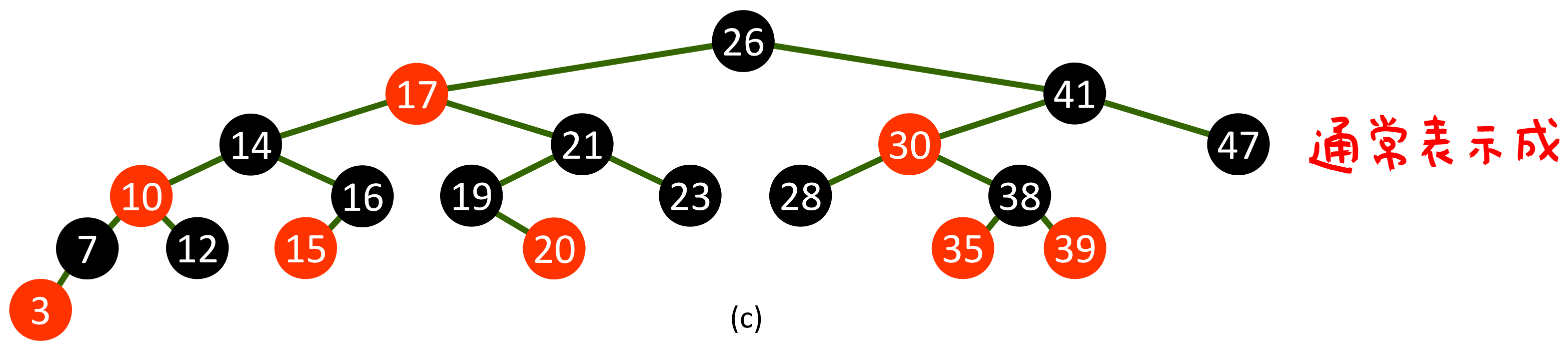
每一点到不同leaf的过程中,所經过的黑節点個數皆相同



(a)



(b)



(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. 用一個点 nil [T] 代表所有 leaf
  - color[nil[*T*]] is black. nil [T] 是黑的
  - The root's parent is also nil[*T*]. root 的父節点也是 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.

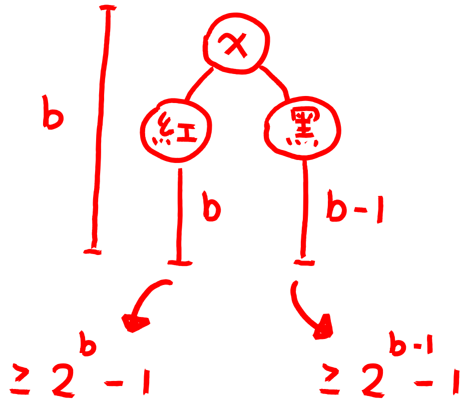
紅黑樹的高度
- node 高度: node 到 leaf 所經过最多 edge 數
- node 黑高: node 到 leaf 所經过黑節点個數

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. 以x為root的子樹的internal node個數 $\geq 2^{bh(x)} - 1$

  Proof: By induction on the height of x.

- **Basis:** 用樹高作歸納；樹高比x低的都成立
  - 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 bh(x) = b. 設黑高為b
  - Any child of x has black-height either b (if the child is red) or b − 1 (if the child is black). 若小孩為紅 ⇒ 黑高為b；若小孩為黑 ⇒ 黑高為b-1
  - 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.)

最差情形：兩子節點皆為黑 ⇒ 子節點 intenal node $\geq 2^{b-1} - 1$

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

- **Lemma 1** A red-black tree with n internal nodes has height $\leq 2\lg(n+1)$. 紅黑樹高度 ≤ lgn 的常數倍

Proof: h：樹高 b：黑高

- 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. root 到 leaf 所經过的紅節点個數 ≤ $\frac{h}{2}$
- Hence $\geq h/2$ are black, i.e., $b \geq h/2$. 黑節点個數 ≥ $\frac{h}{2}$
- Thus, $n \geq 2^b - 1 \geq 2^{h/2} - 1$.
- This implies $h \leq 2\lg(n+1)$.

Operations on red-black trees

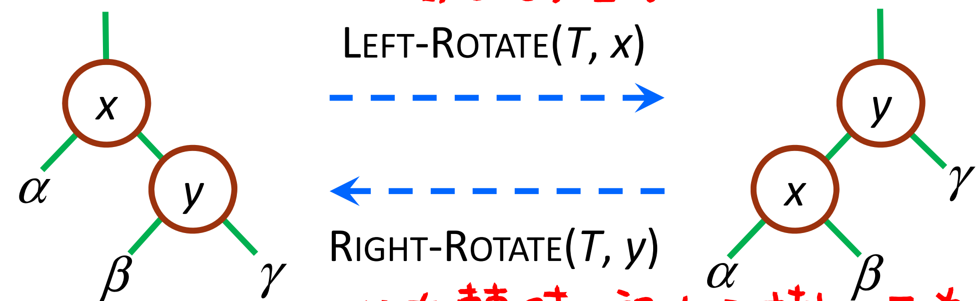
- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O(height) time. Thus, they take $O(\lg n)$ time on red-black trees.
- **Insertion** and **deletion** are not so easy. → 会改变樹的性质
- For example:
  - If we insert, what color to make the new node?
    - Red? Might violate property 4. 插入節点為紅→可能違反性质4
    - Black? Might violate property 5. 插入節点為黑→可能違反性质5

Outline

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

Rotations 旋轉：用來維持紅黑樹性質

- A local operation in a search tree that preserves the binary-search-tree property. 轉完後二元樹性質仍維持 (左子樹 < 自己，右子樹 ≥ 自己)
- There are two kinds of rotations:
  - **Left rotations** and **right rotations**. 旋轉 = 左轉 + 右轉
  - They are inverses of each other. 左轉 + 右轉 = 沒有轉
- When we do a left rotation on a node x, we assume that its right child y is not nil[T]. 以x左轉時，設右子樹y不為空
x成為y的左子



以y右轉時，設左子樹x不為空
y成為x的右子

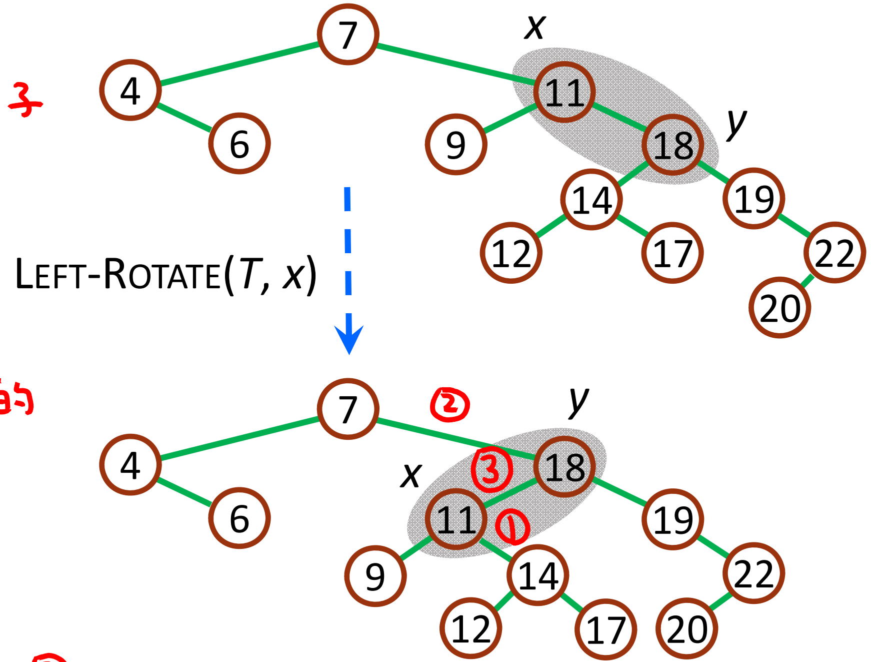
LEFT-ROTATE pseudocode

```
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
```

設定x與新右子的關係 ①

設定y與新父親的關係 ②

設定x與y的關係 ③



- Time: $O(1)$. 旋轉只需常數時間
- The code for RIGHT-ROTATE is symmetric.

Outline

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

RB-Insertion$_{1/2}$ 插入

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

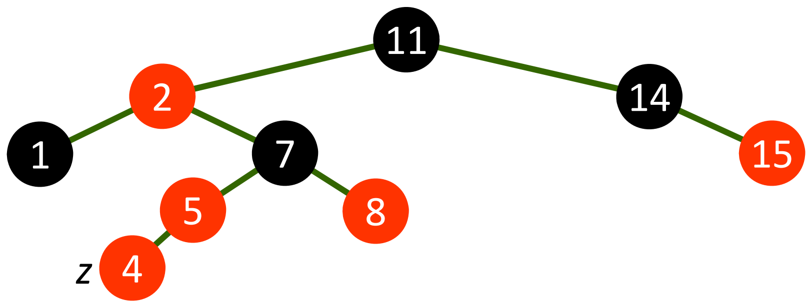
RB-INSERT(T, z)

和之前相同

```
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)
```

設屬性

修正



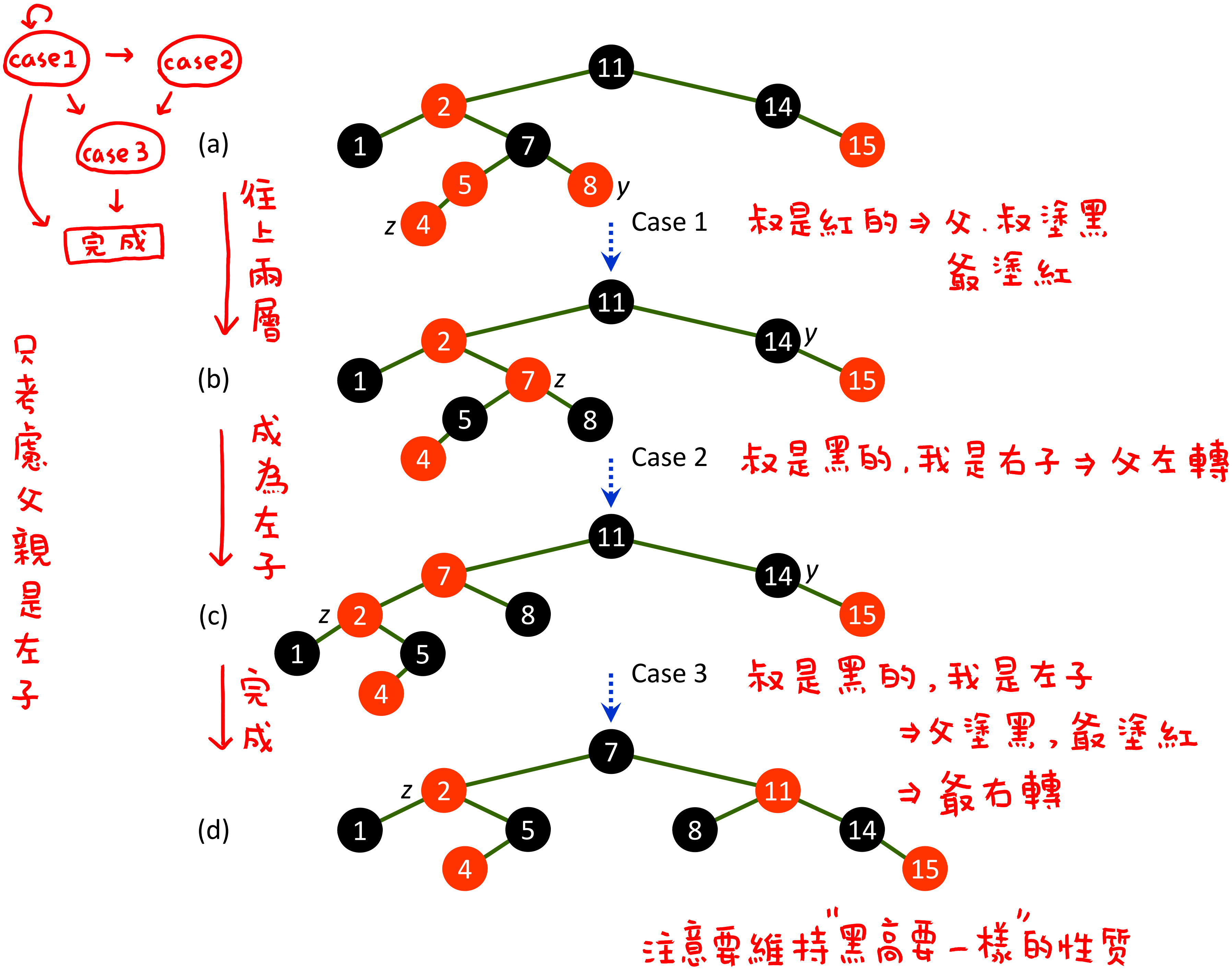
RB-Insertion$_{2/2}$

- RB-INSERT ends by coloring the new node *z* red. 將插入的點塗成紅色
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property. 用Fixup修正
- Which property might be violated? 插入違反哪些性質
  1. OK.
  2. **If *z* is the root**, then there's a violation. Otherwise, OK.
     如果z是root則違反
  3. OK.
  4. **If *p*[*z*] is red**, there's a violation: both *z* and *p*[*z*] are red.
     如果z的父親是紅色則違反
  5. OK.

RB-INSERT-FIXUP procedure 修正

```
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
```

只考慮父親是左子



case1
case2
case3
完成
往上兩層
只考慮父親是左子
成為左子
完成
(a)
(b)
(c)
(d)
11
2
14
1
7
15
5
8
y
z
4
Case 1
叔是紅的 ⇒ 父.叔塗黑
爺塗紅
Case 2
叔是黑的，我是右子 ⇒ 父左轉
Case 3
叔是黑的，我是左子
⇒ 父塗黑，爺塗紅
⇒ 爺右轉
注意要維持"黑高要一樣"的性質

正確性證法：loop invariant 要維持的性質

每一次 while loop 開始之前具有下列性質

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. z是紅的

  如果z的父親是root，則P[z]是黑色的

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

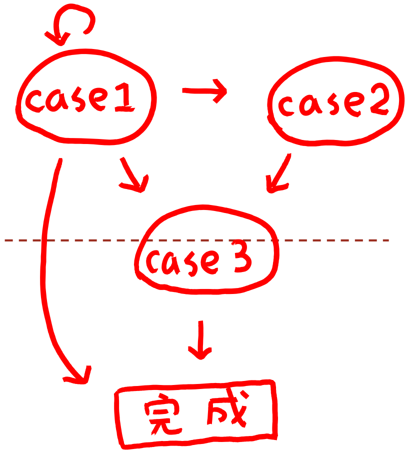
  c. There is at most one red-black violation: 最多違反性質2或性質4

  證明修正後黑高一樣

  - 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. case 1 每次往上 2 個 level
  - 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.

最多轉 2 次 $\begin{cases} \text{case2 左轉 1 次} \\ \text{case3 右轉 1 次} \end{cases}$

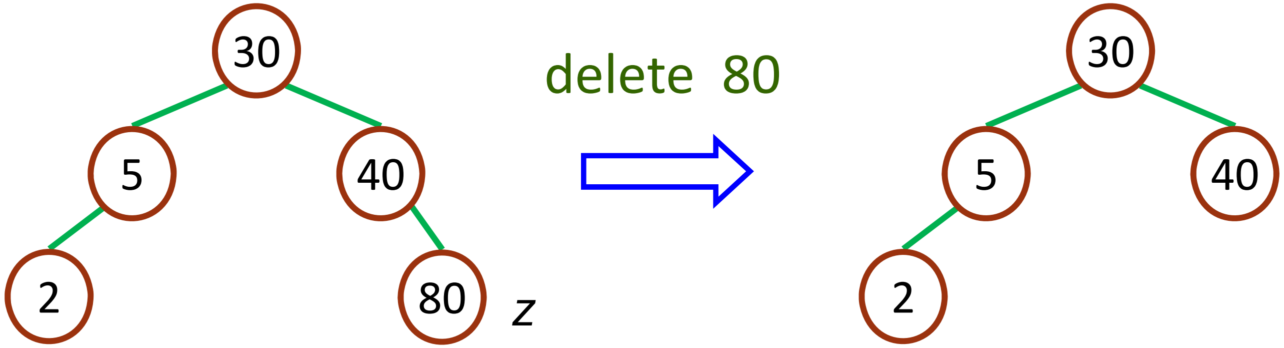
- Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Outline

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

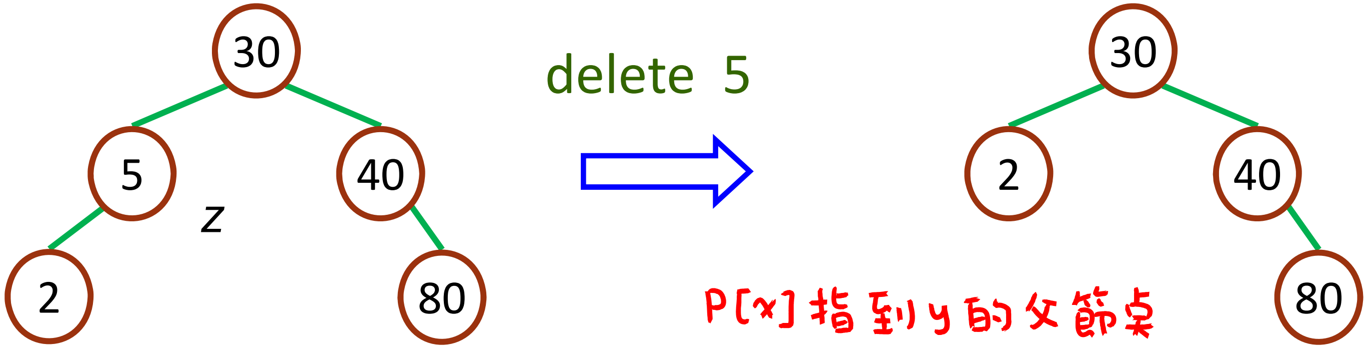
Case 1:

沒有兒子，
直接刪



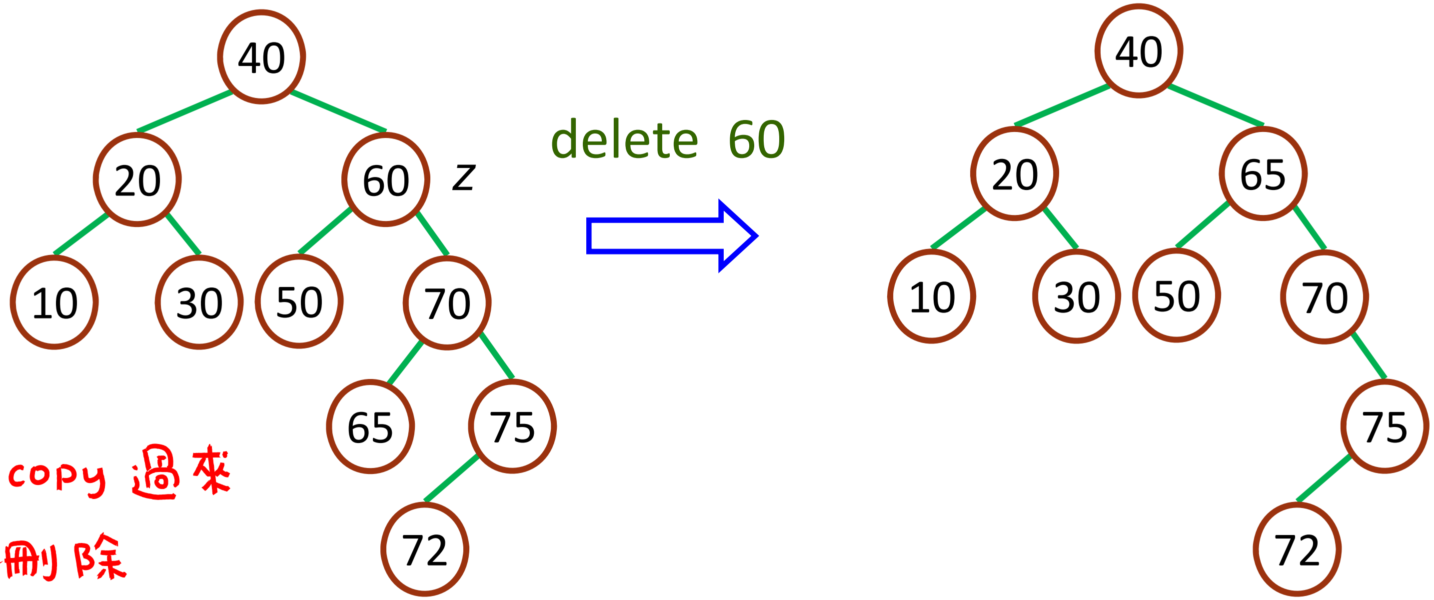
Case 2:

有一個兒子，
將兒子連到
父親



Case 3:

有2個兒子，
① 找下一個
② 將下一個 copy 過來
③ 將下一個刪除



刪除

RB-Deletion$_{1/3}$

P[x]指到y的父節點

y是真正要刪的點

x是 { y的唯一子節點 / nil[T] }

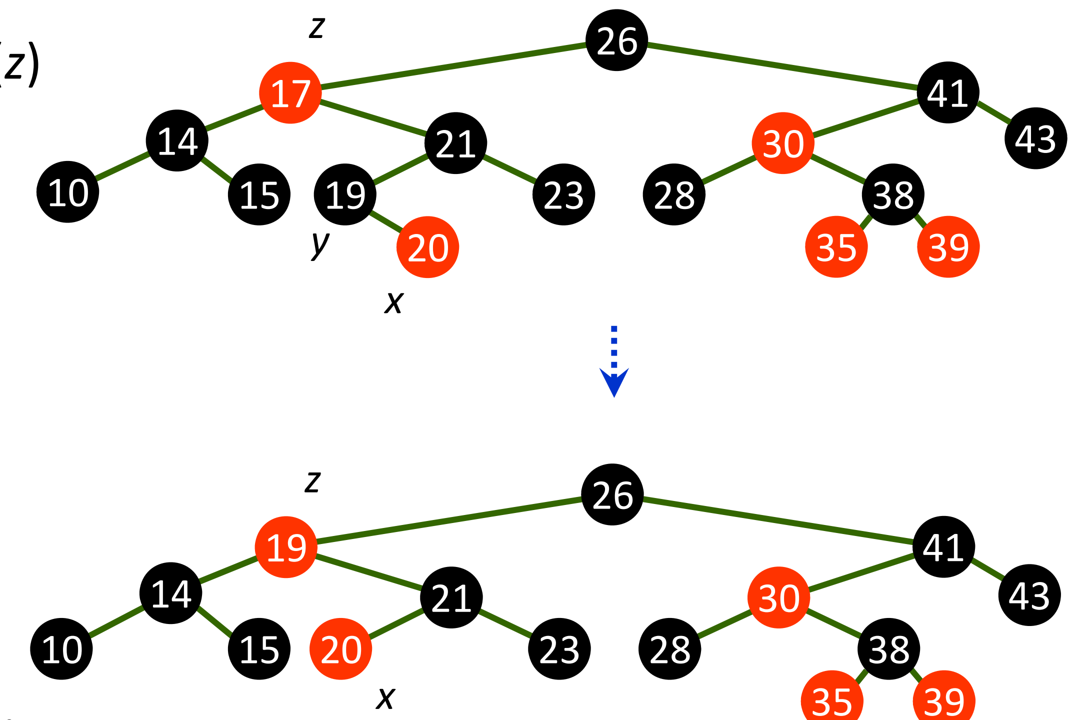
RB-DELETE(*T*, *z*)

與之前相同

```
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
```

刪掉的點是黑的 ⇒ 修正

y是真正要刪的點



RB-Deletion$_{2/3}$

- *y* is the node that was actually spliced out. y是真正要刪的點
- *x* is either x是 { y的唯一子節點 / nil[T] }
  - *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. P[x]指到y的父節點
- If *y* is red, the red-black properties still hold when *y* is spliced out, for the following reasons: 若y是紅的，刪除不違反紅黑樹性質
  - no black-heights in the tree have changed, 黑高不改变
  - no red nodes have been made adjacent, and 紅節點不相連
  - since *y* could not have been the root if it was red, the root remains black. ∵ y非root ∴ root仍為黑

RB-Deletion$_{3/3}$ 若y是黑的，刪除可能違反紅黑樹性質

- 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. 如果y是root且x是紅色則違反
  3. OK.
  4. Violation if *p*[*y*] and *x* are both red. 會違反如果p[y]和x都是紅色
  5. Any path containing *y* now has 1 fewer black node. path會經過y的桌黑高都少1
- Correct this problem by giving *x* an "extra black". 讓x多黑色屬性
  - Now property 5 is OK, but property 1 is not.
  - *x* is either **doubly black** or **red & black**. x變成黑黑 或 x變成紅黑

RB-DELETE-FIXUP

兄是紅 ①
兄是黑 右姪黑 左姪黑②
左姪紅③
右姪紅 ④

- **Idea:** Move the extra black up the tree until 將多的黑色往上移，直到
  - **1.** *x* points to a red & black node → turn it into a black node, 遇到紅点
  - **2.** *x* points to the root → just remove the extra black, or 遇到 root
  - **3.** suitable rotations and recolorings can be performed.

過程中可以適當的旋轉和將点的顏色重塗

- Within the while loop:

  x 指到 doublely black node 且非 root

  - *x* always points to a nonroot doubly black node.
  - *w* is *x*'s sibling. w 是 x 的兄弟
  - *w* cannot be *nil*[*T*], since that would violate property 5 at *p*[*x*].

  w 不会是 leaf，否則違反性質 5 (黑高要一樣)

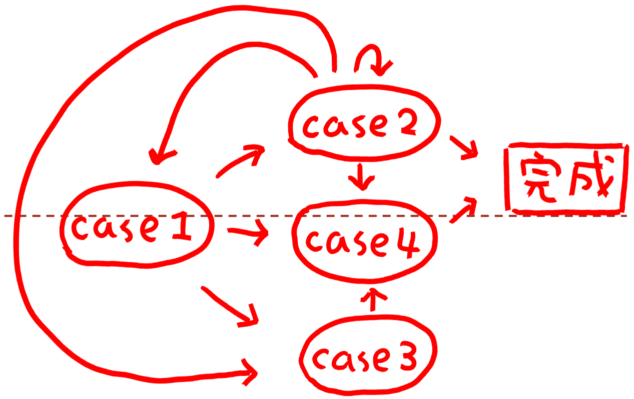
- There are 8 cases, 4 of which are symmetric to the other 4.

  共 8 種情形

- As with insertion, the cases are not mutually exclusive. We'll look at cases in which *x* is a left child. 只考慮 x 在左子的情形

  4 種

RB-DELETE-FIXUP procedure

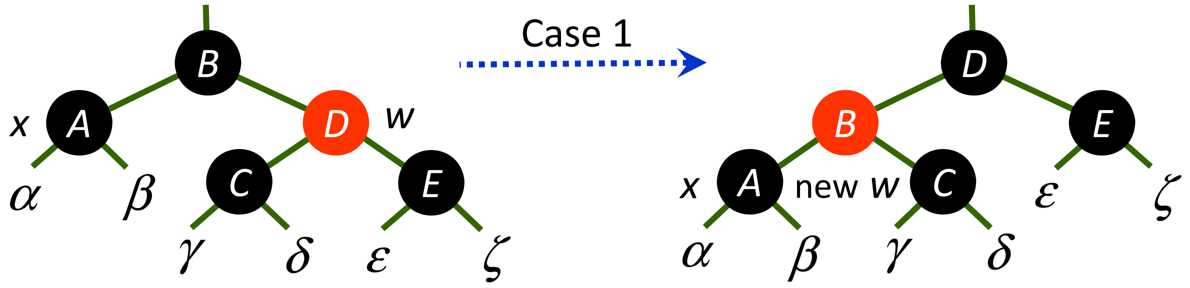



```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T ] and color[x] = BLACK
    if x = left[p[x]]
        w ← right[p[x]]
        if color[w] = RED
            color[w] ← BLACK                                Case 1
            color[p[x]] ← RED                               Case 1
            LEFT-ROTATE(T, p[x])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

注意要維持"黑高要一樣"的性質

Case 1: *w* is red 兄是紅

x有"2黑高"



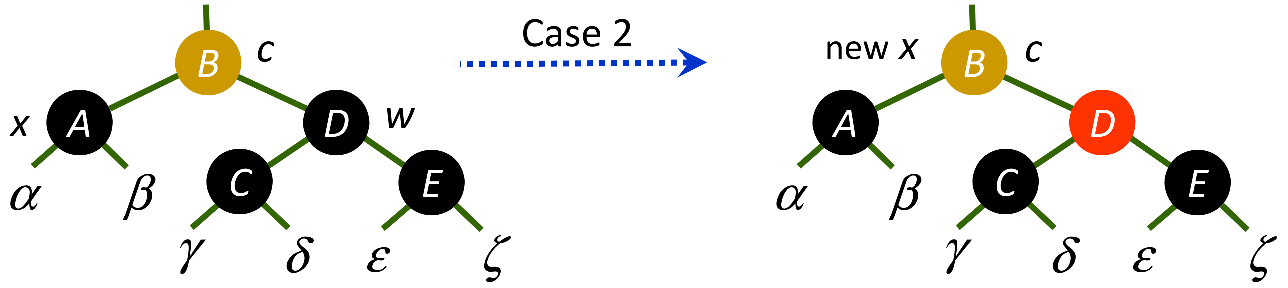
假設黑高(α)=黑高(β)=b 則黑高(γ)=黑高(δ)=黑高(ε)=黑高(ζ)=b+1

- *w* must have black children. ① 兄塗黑，父塗紅
- Make *w* black and *p*[*x*] red. ② 父左轉
- Then left rotate on *p*[*x*]. ③ 兄立刻成為黑
- New sibling of *x* was a child of *w* before rotation ➔must be black.
- Go immediately to case 2, 3, or 4.

兄是黑，兩個姪子也是黑

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

土黃：可能是黑 or 紅



黑高(α)=黑高(β)=黑高(γ)=黑高(δ)=黑高(ε)=黑高(ζ)=b

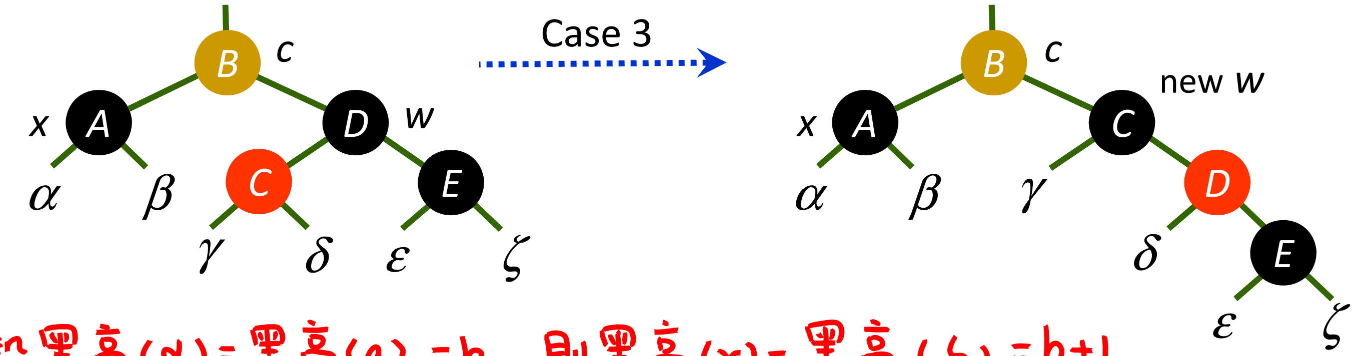
- Take 1 black off *x* (➔singly black) and off *w* (➔red).
- Move that black to *p*[*x*].
- Do the next iteration with *p*[*x*] as the new *x*.
- If entered this case from case 1, then *p*[*x*] was red ➔ new *x* is red & black ➔ color attribute of new *x* is RED ➔ loop terminates. Then new *x* is made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

①我和兄的黑移轉一個給父親
②父成為新的x
③父如果是紅⇒結束

如果是case1過來的⇒父為紅 ⇒結束

w is black, 兄是黑，左姪是紅，右姪是黑

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



假設黑高(α)=黑高(β)=b 則黑高(γ)=黑高(δ)=b+1

黑高(ε)=黑高(ζ)=b

- 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.

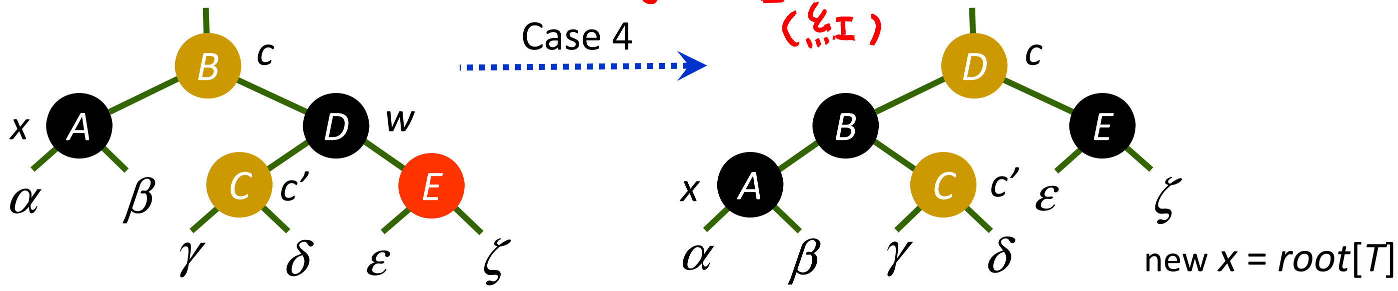
① 兄塗紅，左姪塗黑
② 兄右轉
③ 成為 case 4

兄是黑，右姪是紅　　注意要維持"黑高要一樣"的性質

Case 4: *w* is black, and *w*'s right child is red

黑　黃　黑

我 → 父 → 兄 → 右姪 (紅)



假設黑高(α)=黑高(β)=b

c′的顏色 { 紅 ⇒ 黑高(γ)=黑高(δ)=b+1 ; 黑 ⇒ 黑高(γ)=黑高(δ)=b }

- Make *w* be *p*[*x*]'s color (*c*). 黑高(ε)=黑高(ζ)=b+1
- 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.

① w塗成父的顏色
② 父塗黑，右姪塗黑
③ 父左轉
④ 移除x的一個黑
⇒完成

Time complexity of RB-DELETE

- **Analysis**:
  - Case 2 is the only case in which more iterations occur. case 2 会重覆做
    - *x* moves up 1 level. 每次上升 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)

如果是 case 1 ⇒ case 2 ⇒ 結束