紅黑樹

Michael Tsai

2012/06/12

紅黑樹

- 可以幹嘛?
- 是棵平衡的樹: 保證從root到某個leaf的simple path一定不會超過從root到任何一條其他這樣的path的兩倍長
- 大概平衡→operation可以都在**O(log n)**內完成. 耶.

- 那些operation?
- 1. 找
- 2. 插入某element
- 3. 殺掉某element
- 4. 找最大or找最小element
- 5. 找某一個element的下一個(successor)或前一個element (predecessor)

紅黑樹

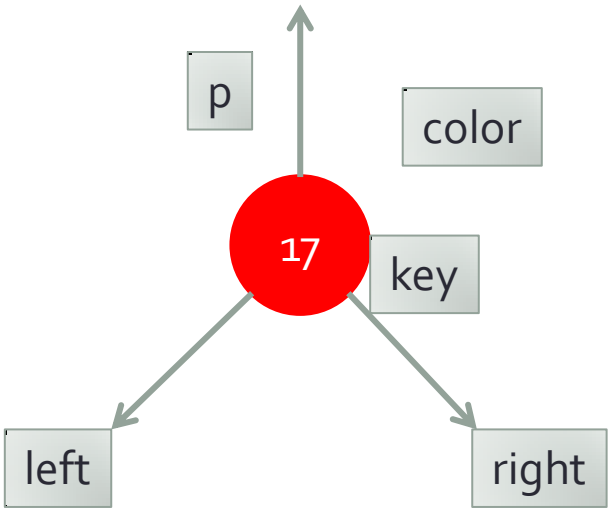
- 每個node都分配一個顏色: 紅或黑
- 沒有children的地方都補上external node, 又叫NIL

- 規則們:
- 1. 每個node不是黑就是紅
- 2. root是黑色的
- 3. 每個leaf (external node, or NIL)都是黑色
- 4. 如果一個node是紅的, 則它的兩個小孩都是黑的
- 5. 對每個node來說, 從它到他的所有子孫葉子node的路徑上含有**一樣數目的黑色node** (不包含他自己)

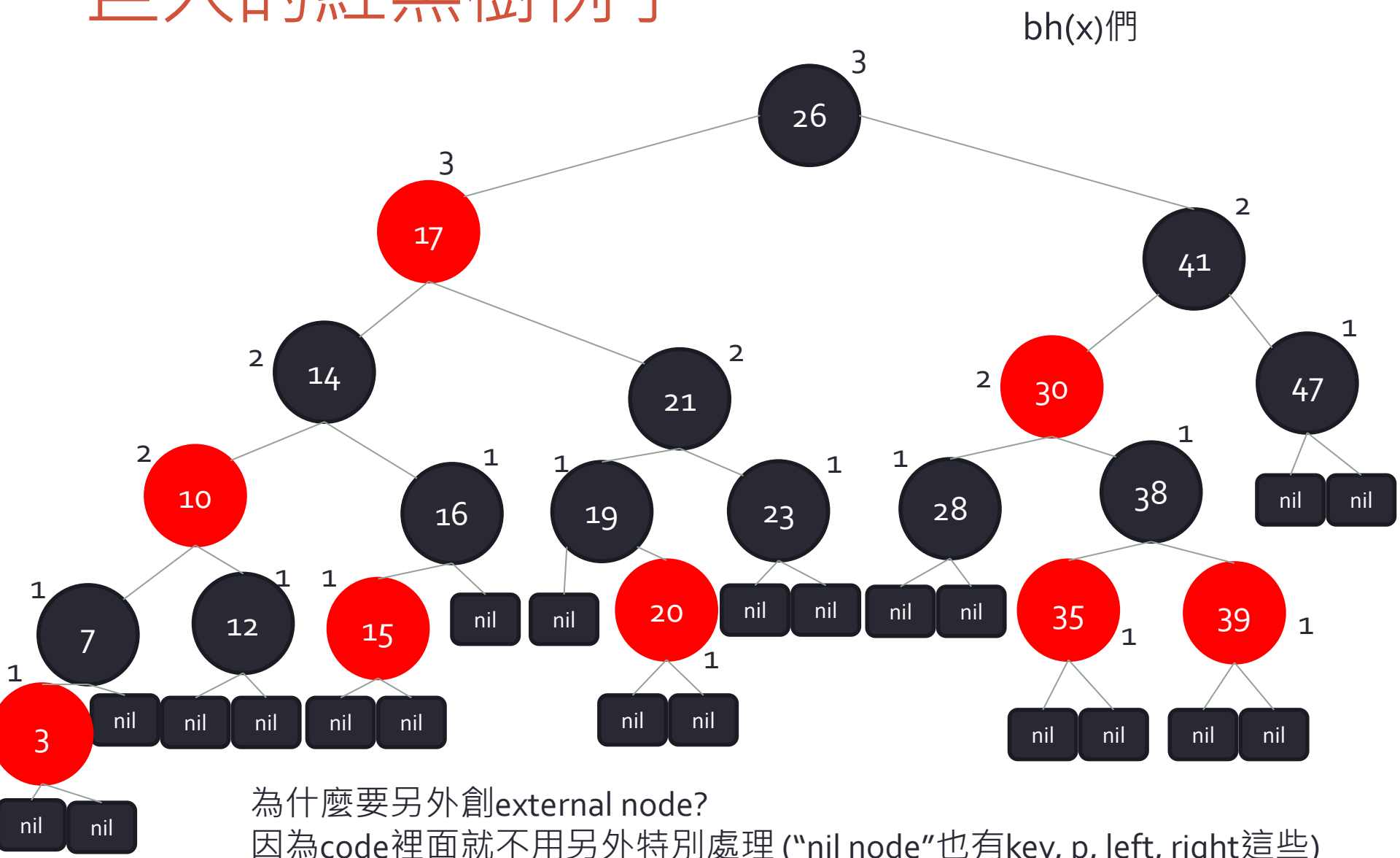
黑高度

- Black height: bh(x) = 從x到任何一個它的子孫葉子node遇到的black node個數 (因為都一樣, 所以可以是任何一個)
- 不包含node x自己 (如果它自己是黑的的話)
- external node或nil(葉子node)的black height為o

Representation

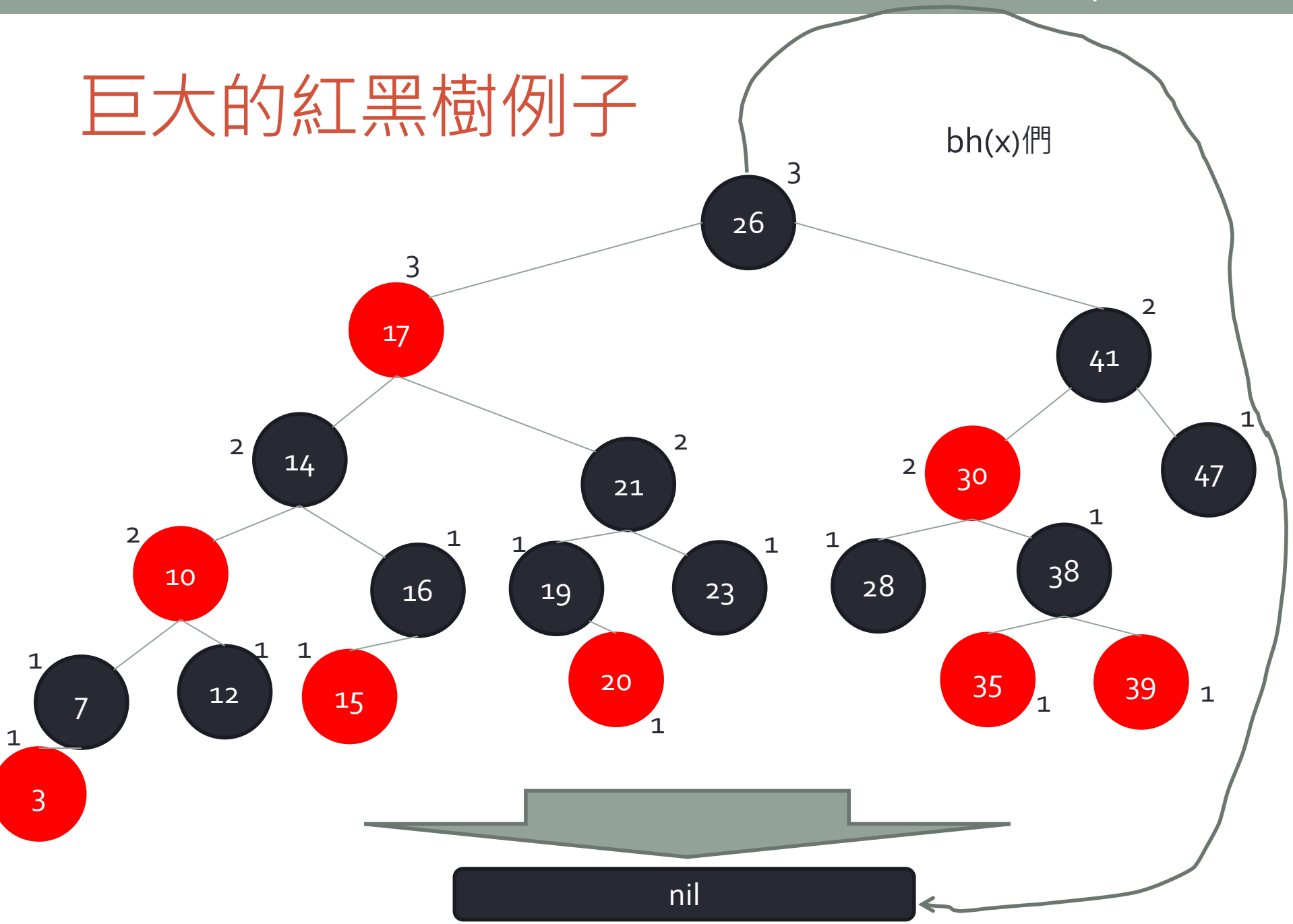


巨大的紅黑樹例子



為什麼要另外創external node?
因為code裡面就不用另外特別處理 ("nil node"也有key, p, left, right這些)

巨大的紅黑樹例子


bh(x)們
3
26
3
17
2
41
2
14
2
21
2
30
1
47
2
10
1
16
1
19
1
23
1
28
1
38
1
7
1
12
1
15
20
1
35
1
39
1
1
3
nil

來點證明

- **定理: 一個有n個node的紅黑樹, 最高為2 log(n+1)**
- 第一步驟: 證明node x底下的subtree最少有$2^{bh(x)} - 1$個internal node
- 歸納法證明:
- 1. 當x的height為0, 則x是個葉子. bh(x)=0. $2^0 - 1 = 0$. 的確subtree有0個internal node.
- 2. 假設當x的height為正整數, 且是一個有兩個children的internal node. 每個小孩的black height為bh(x) (如果小孩是紅的) or bh(x)-1 (如果小孩是黑的)
- 假設小孩的subtree都至少有$2^{bh(x)-1} - 1$個internal node
- 3. 則x底下的subtree應該有
- $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ (成功了!)

來點證明

- "4. 如果一個node是紅的, 則它的children都是黑的"
- 另外一種解釋:
- 從任一node到leaf, 至少一半以上的node是黑的
- 假設h是tree的高度, 則$bh(root) \geq \frac{h}{2}$
- "node x底下的subtree最少有$2^{bh(x)} - 1$個internal node"
- 第二步驟: root底下至少有多少個node?
- root底下最少有
- $2^{bh(x)} - 1 \geq 2^{\frac{h}{2}} - 1$個node.
- 假設node個數為n, 則$n \geq 2^{\frac{h}{2}} - 1$
- →$h \leq 2\log(n+1)$. 耶.

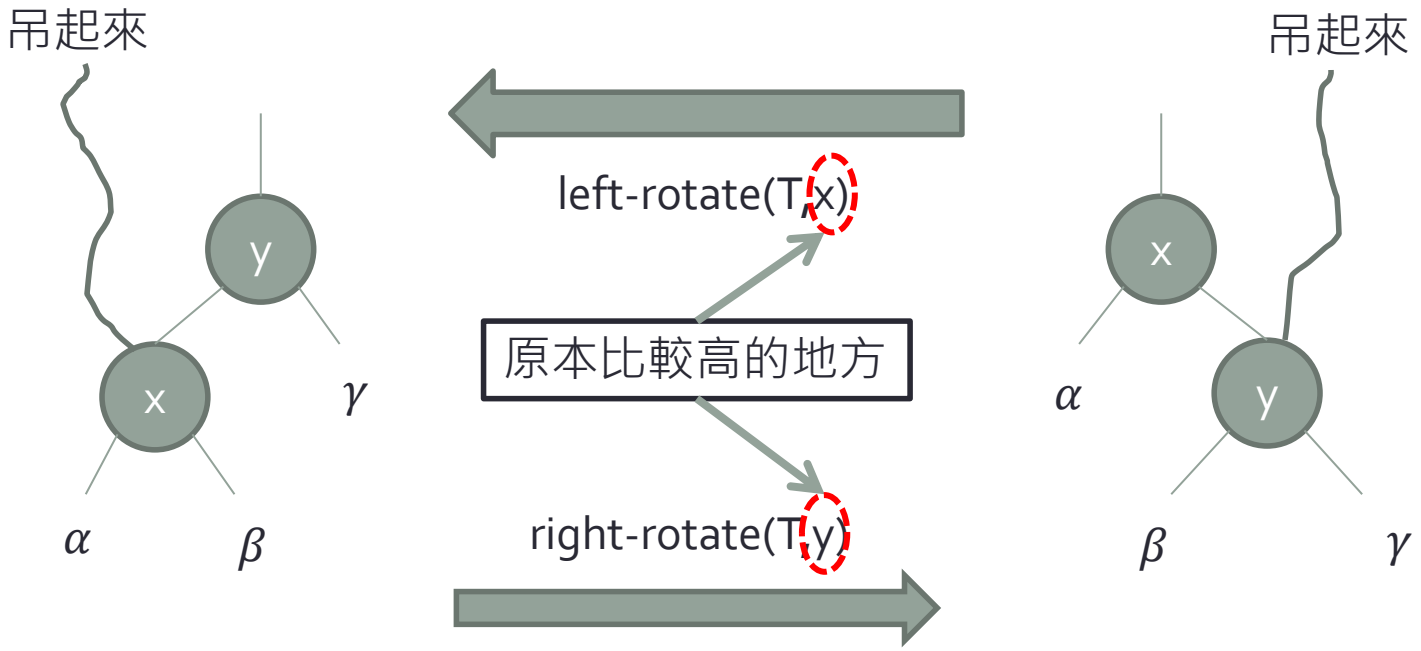
以上證明...

- 說明為什麼
- 1. 找
- 4. 找最大or找最小element
- 5. 找某一個element的下一個(successor)或前一個element (predecessor)
- 等operation可以在O(log n)內完成

- 那麼insert和delete呢?
- 插入和刪除本身所花時間滿足O(log n)
- 但是: 插入或殺掉之後, 可能不滿足紅黑樹的條件
- 要花多少時間調整呢? 是否還是O(log n)?

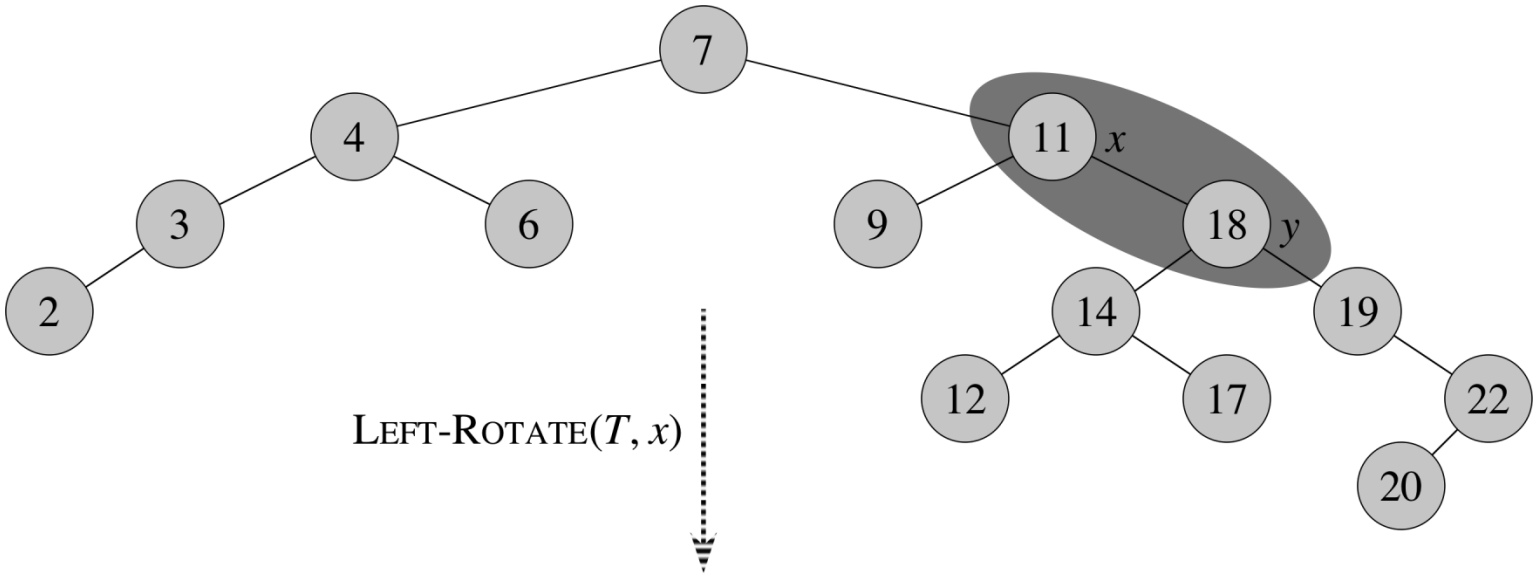
Rotate

- 等一下會用到的



- 注意rotate前後大小關係沒有變喔!
- $\alpha < x < \beta < y < \gamma$

來個例子：Left-Rotate(T,x)



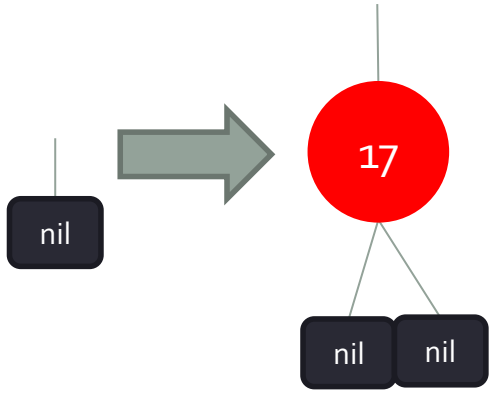
???????

Insertion

- 首先, 用原本的binary search tree插入的方法
- insert(z)
- 不同的地方:
- 1. z的兩個children都指到nil node nil
- 2. z為紅色
- 3. 我們最後要處理不符合紅黑樹規則的部分

- 怎麼處理?

會違反那些規則呢?

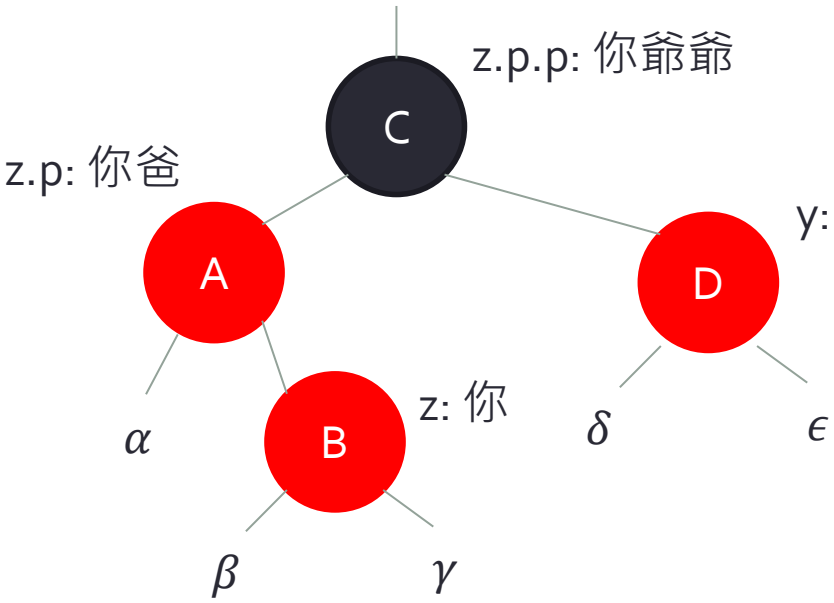


- 規則們:
- ~~1. 每個node不是黑就是紅~~
- 2. root是黑色的
- ~~3. 每個leaf (external node, or nil)都是黑色~~
- 4. 如果一個node是紅的, 則它的children都是黑的
- ~~5. 對每個node來說, 從它到他的所有子孫葉子node的路徑上含有一樣數目的黑色node~~

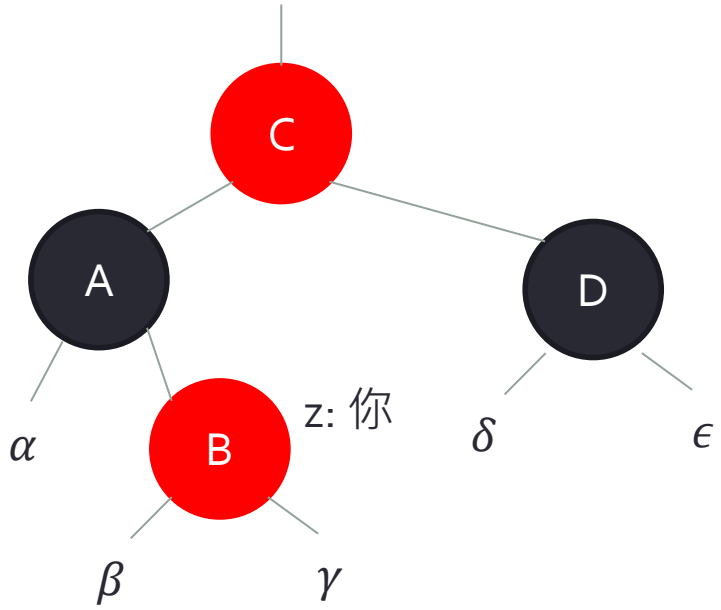
- 5.不會違反因為z是紅的, z取代掉一個nil, 而z的兩個children都是nil
- 如果違反2, 則z是root, 整棵樹只有z: 很容易處理
- 來看違反4的情形: z.p is also red

情形一：你的叔叔是紅的

如果你是你爸右邊的小孩



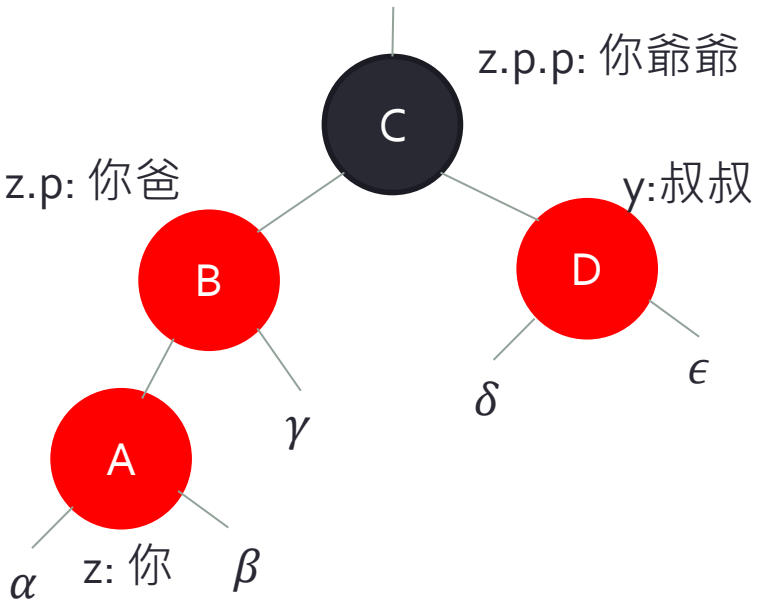
繼續看新z爸爸是不是紅的



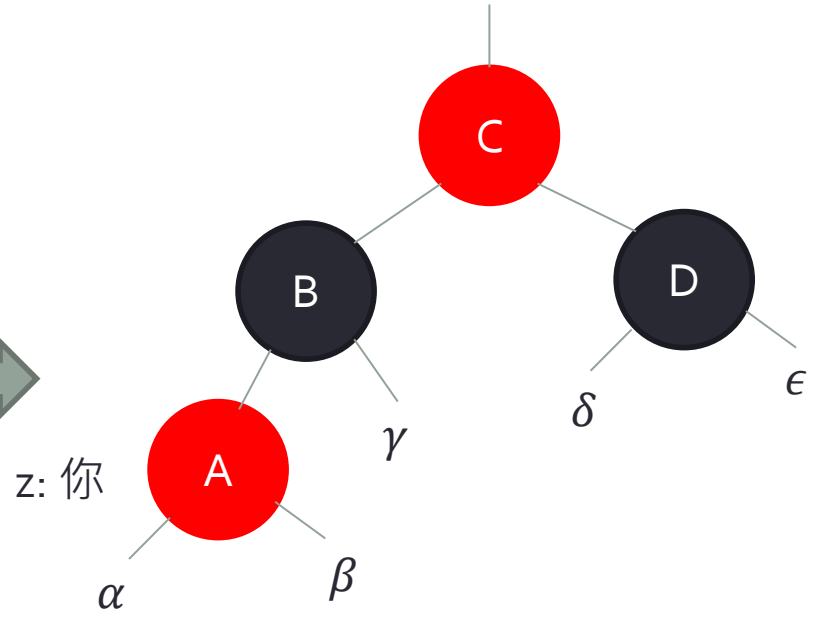
注意看每一個步驟是否有保持紅黑樹第五個條件:
從C開始走到α~ ϵ任一個分支所經過的黑色node數目在修改前後必須相同!

情形一：你的叔叔是紅的

如果你是你爸左邊的小孩

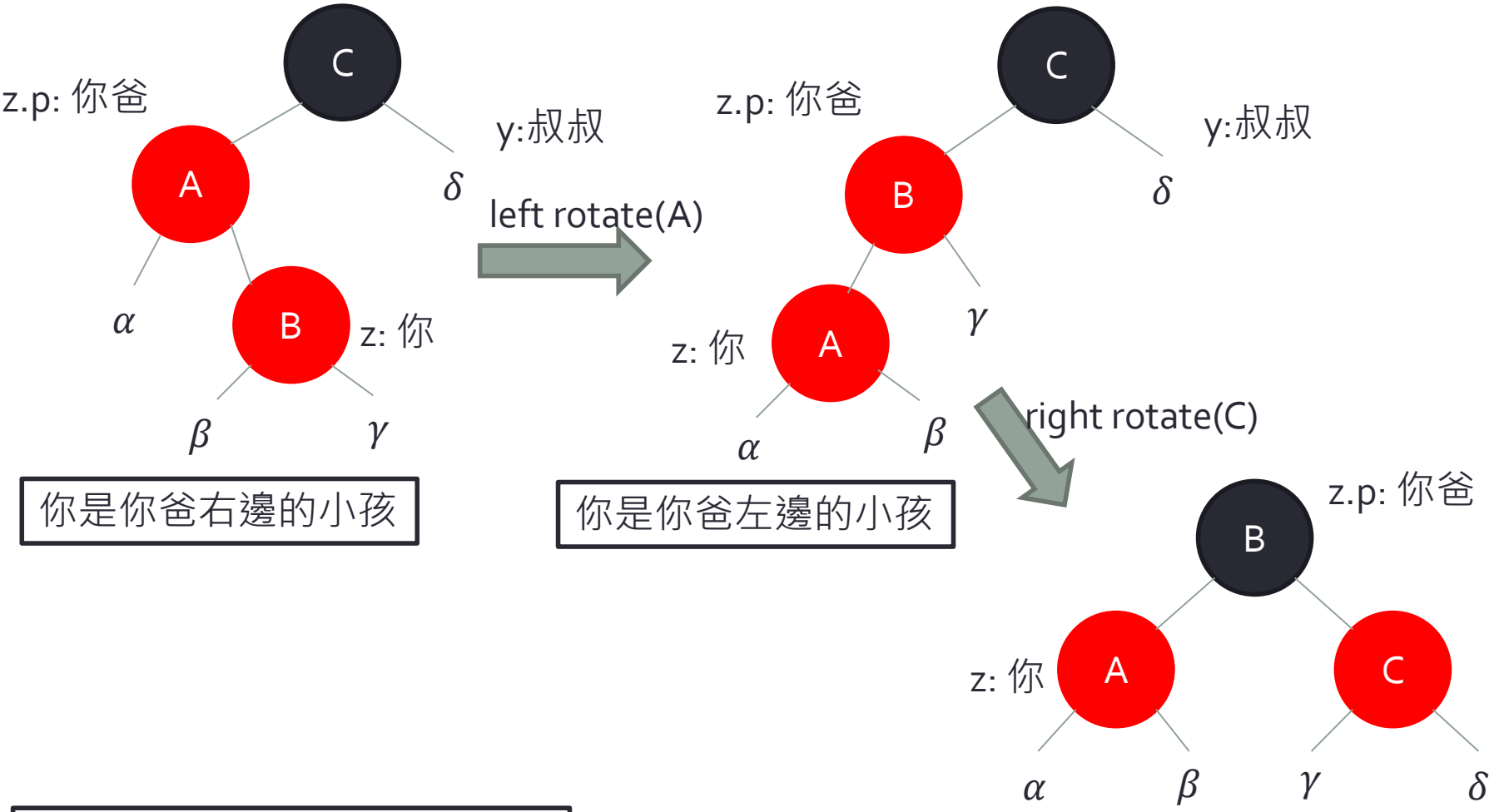


繼續看新z爸爸是不是紅的



注意看每一個步驟是否有保持紅黑樹第五個條件:
從C開始走到α~ ϵ任一個分支所經過的黑色node數目在修改前後必須相同!

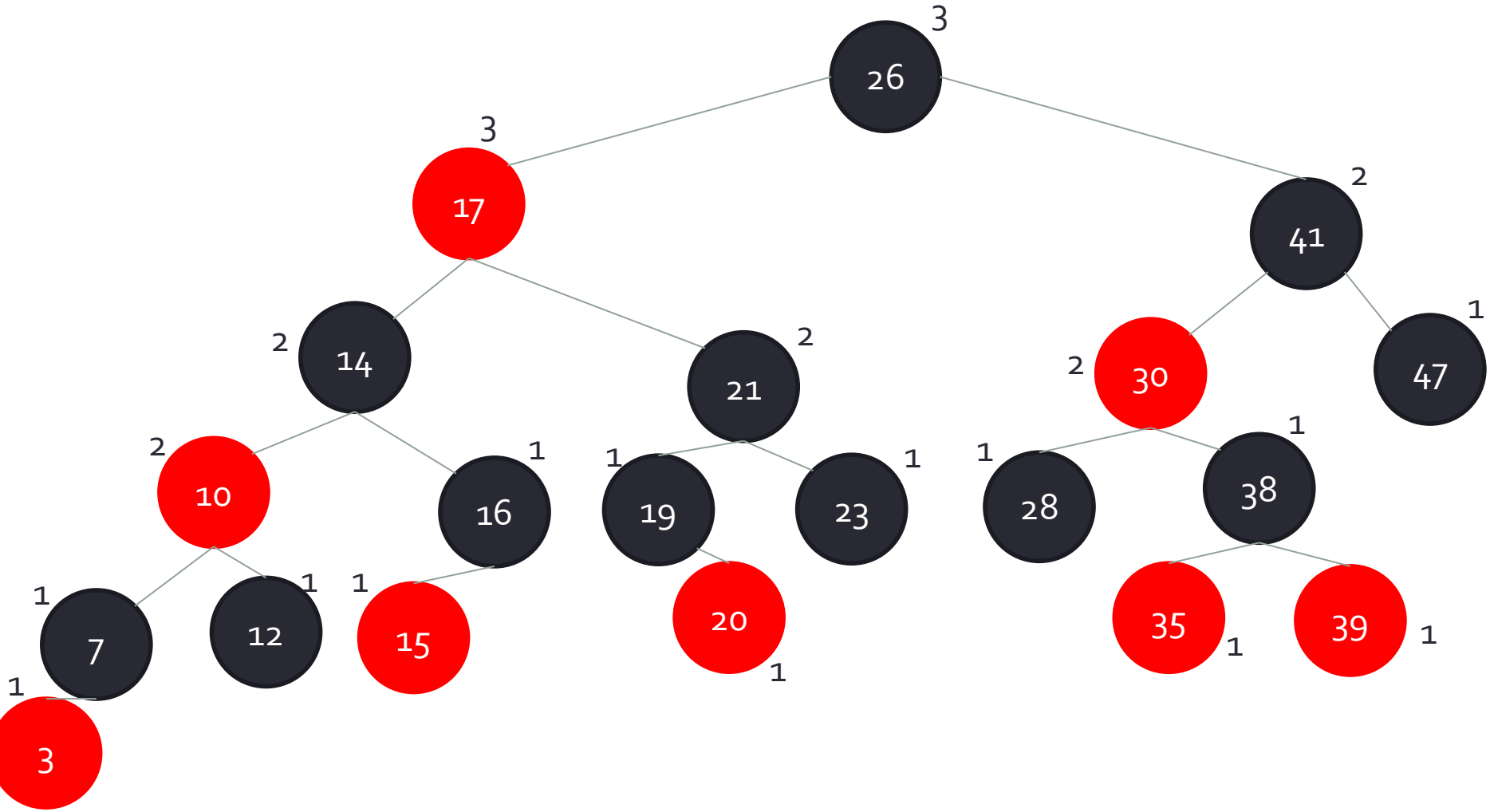
情形二與三：你的叔叔是黑的



注意看每一個步驟是否有保持紅黑樹第五個條件

需要繼續往上層看嗎? 不用! 因為B還是黑的!

練習時間



1. 加入40?

2. 加入4?

Pseudo-code: 插入後修理R-B tree

```
RB-Insert-Fixup(T,z)
While z.p.color==RED  (如果你爸是紅色的)
      if z.p==z.p.p.left (如果你爸是你爺爺左邊的小孩)
            y=z.p.p.right (那麼叔叔就是你爺爺右邊的小孩)
            if y.color==RED (如果叔叔是紅色的)
                  z.p.color=BLACK (就把爸爸設成黑色的)
                  y.color=BLACK (叔叔也設成黑色的)
                  z.p.p.color=RED (爺爺設成紅色的)
                  z=z.p.p (把自己變成爺爺)
            else if z==z.p.right (如果你是你爸右邊的小孩)
                        z=z.p (把自己變成你爸)
                        LEFT-ROTATE(T,z)
                  z.p.color=BLACK (把你爸變成黑色)
                  z.p.p.color=RED (把你爺爺變成紅色)
                  RIGHT-ROTATE(T,z.p.p)
      else if z.p==z.p.p.right (如果你爸是你爺爺右邊的小孩,
                                    跟前面一樣, 只是left, right都反
過來) ……

T.root.color=BLACK (最後把root改成黑色)
```

情形1

情形2

情形3

情形4-6: 鏡射的狀況

要花多少時間呢?

- 原本正常的binary tree insert要花O(log n)的時間
- 因為高度最高為2 log(n+1)
- 那麼花費在調整的時間呢?

- 最糟的狀況? 情形一一直重複發生, 每次z往上移兩層
- 執行時間最糟要花跟高度成正比的時間
- 也是O(log n)

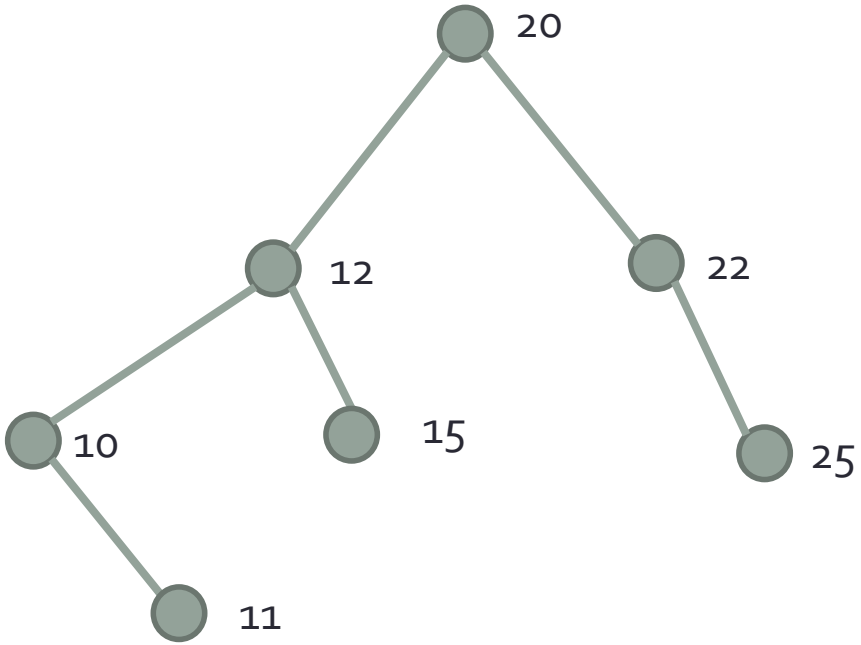
- 那如果發生情形二or三呢?
- 執行一次即完成. O(1). (所以比O(log n)小)
- 另外, 最多rotate只需要執行兩次. (不會再發生第二次情形二or三)

- 正確性證明請見Cormen p. 318-322

如何刪掉一個node? (binary search tree 複習)

- 首先要先找到那個node
- 接著, 有各種不同情形:

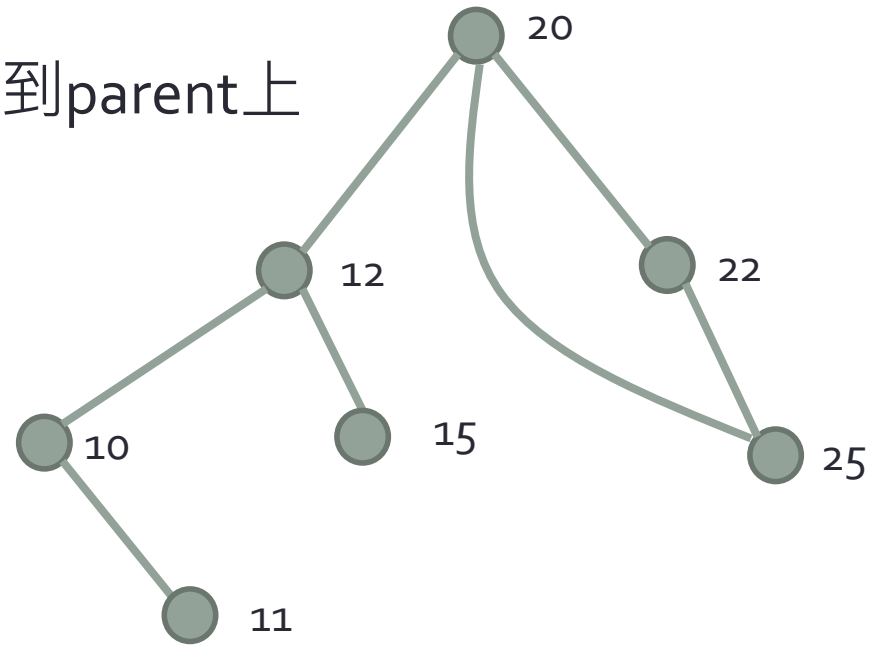
- 如果沒有手(degree=0)
- 直接拿掉



如何刪掉一個node? (binary search tree 複習)

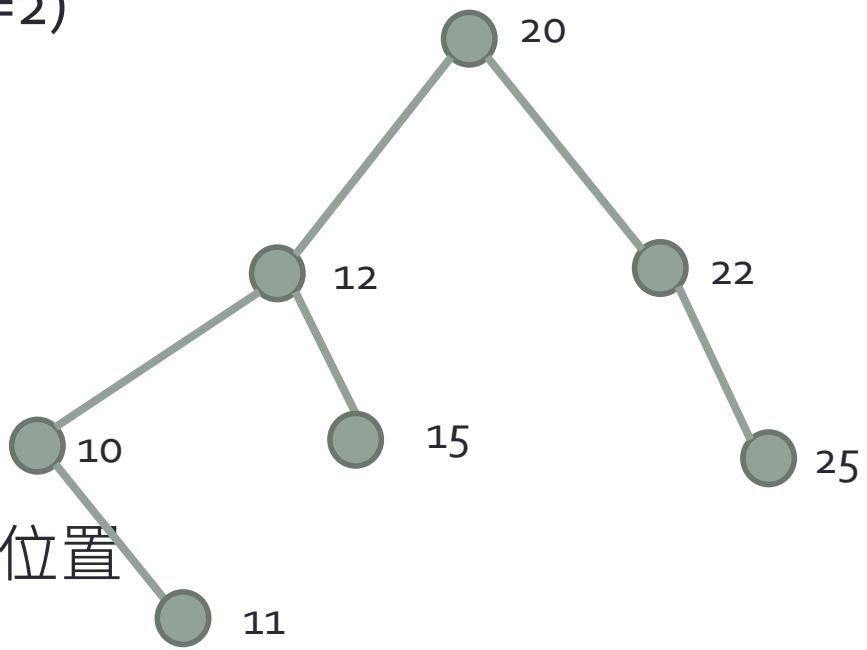
- 如果只有一隻手 (degree=1)
- 則把那個唯一的child拿上來接到parent上

- 例如: 拿掉22



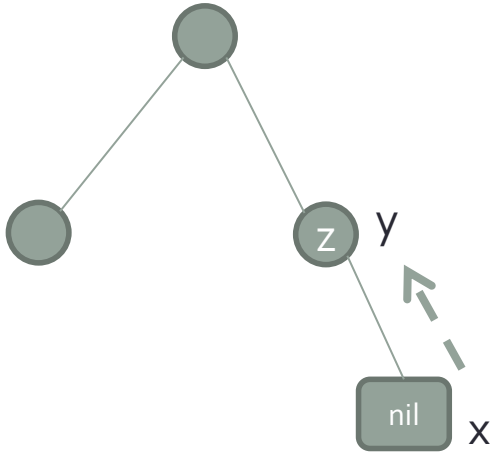
如何刪掉一個node? (binary search tree 複習)

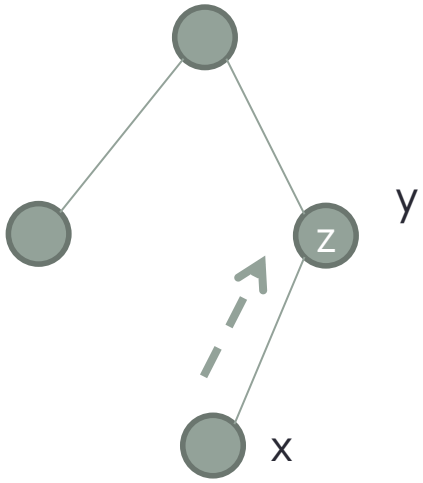
- 如果兩手都有東西呢? (degree=2)

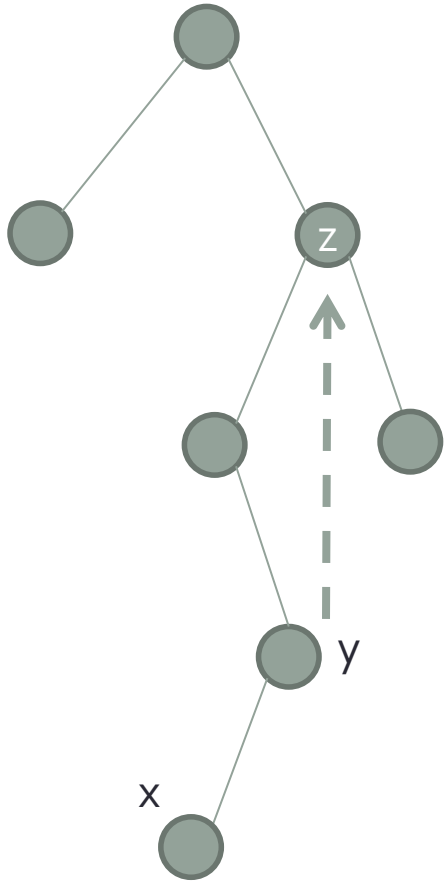


- 例如刪掉12
- 找左邊child底下最大的
- (或者是右邊child底下最小的)
- 刪掉它並把它移到原本刪掉的位置
- 問題: 那如果那個最大的底下還有child呢?
- 直接拿上來(最多只會有左邊一隻手)
- 這時被移上去的node顏色改變成新位置原本node的顏色

x，y，z的定義







z: 要被刪掉的
y: z少於兩隻手→被砍掉的
　z兩隻手 → 移動到z原本位置的
x: 移動到原本y的位置的

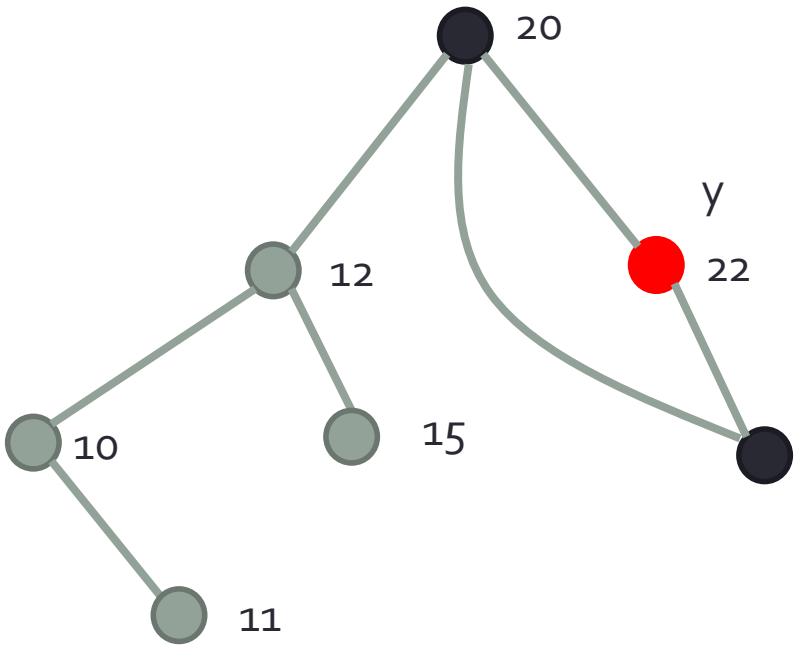
會記得y的顏色.

拿掉一個node，什麼時候會違反規則?

- 假設前述三種情形中,
- 移動(兩手都有)或是刪除(一隻手或沒有手)的node為y

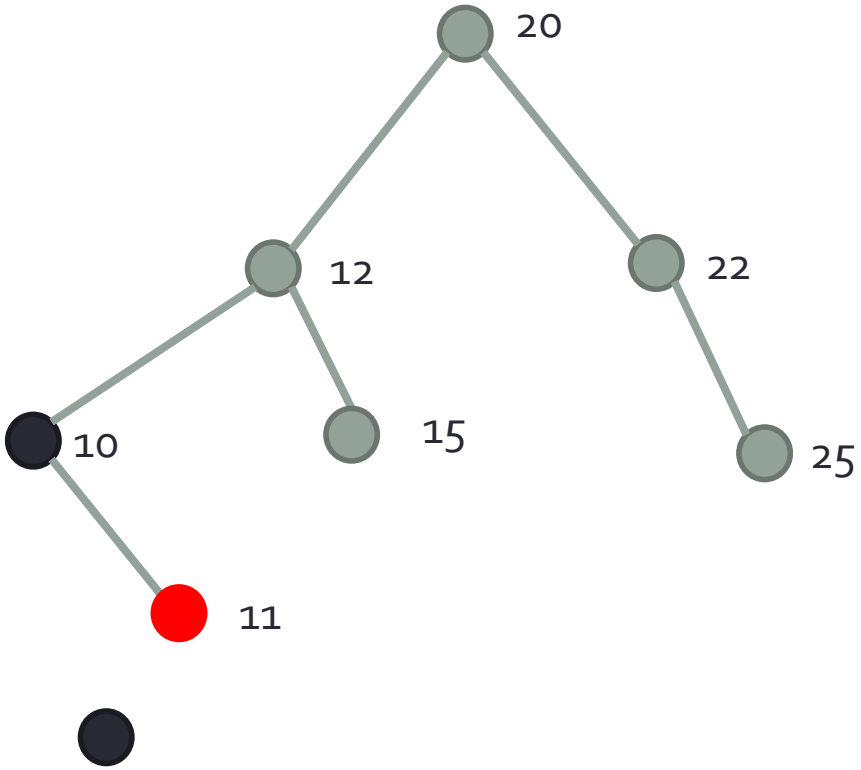
- 當y為紅色(原本的顏色), 移動或刪除會造成違反規則嗎?

- 1. black height不會改變, 因為y是紅色node
- 2. 會不會造成兩個紅色node是相鄰的呢? (父子)
- A. 如果y是被刪掉的, 因為它是紅的, 所以它的上下層都是黑的
- 不會有問題



拿掉一個node，什麼時候會違反規則?

- B. 如果y是被移動的, 假設y有children也是黑色的, 不會造成問題
- 3. y如果是紅色, 它不會是root. 所以也不會有造成違反root是黑色的規定
- 綜合以上三點, 只有當y為黑色時才可能會造成問題, 需要調整



可能違反規則的情形

- 當y是黑色的, 可能造成違反規則的有下列情形:
- 1. y是root. y刪掉以後, 它的一個children是紅色的, 變成了root
- 2. y原本的上下兩層都是紅色的, 移走或刪除以後變成兩個紅色相鄰
- 3. y刪掉或移走以後, 造成y的祖先們的black height不一致(因為y是黑的)

- y拿掉以後, y的黑色就被"趕到" x裡面了
- x表示: 這邊要有一份黑色
- 但是x本身可能原本是紅或黑色

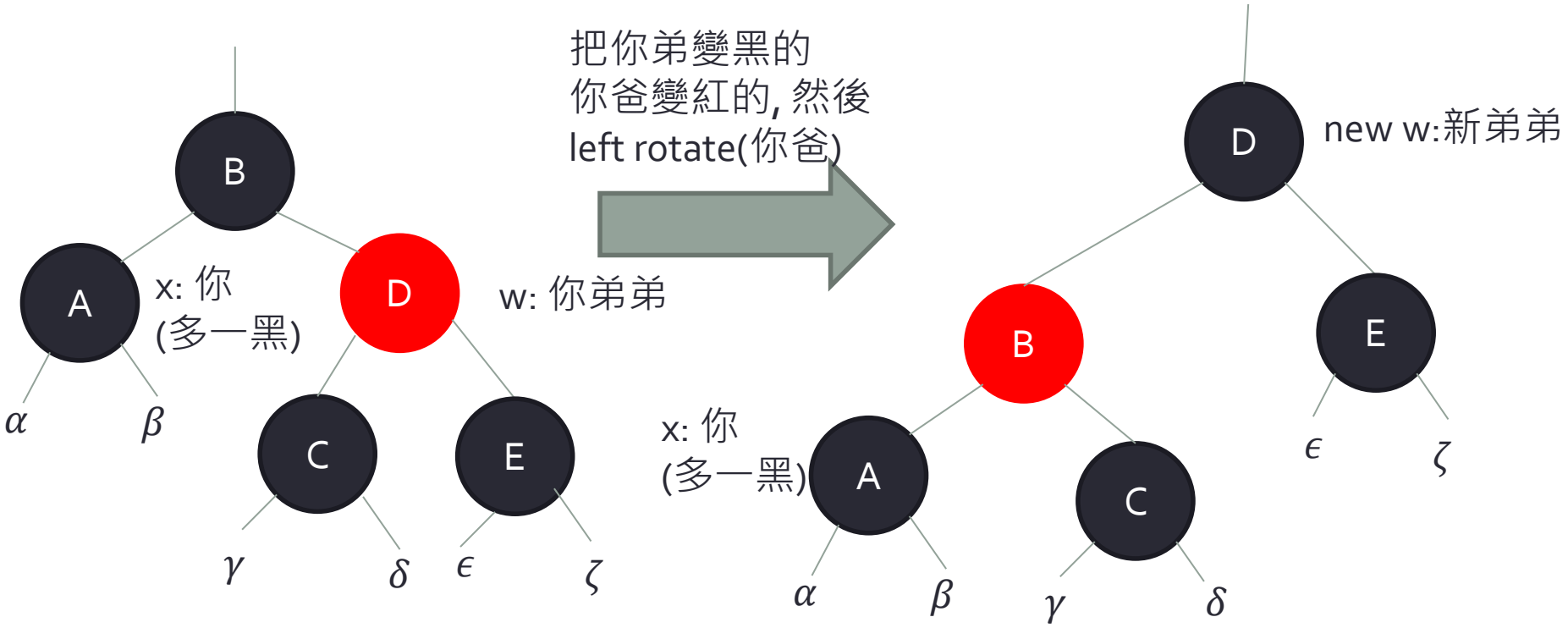
x是”紅與黑”或”黑與黑”?

- 1. x如果是”紅與黑”, 我們把它改成黑就可以了.
- (這邊要多一份黑色, 所以我就把紅色變成黑色就多一份了)

- 2. x如果是root且為”黑與黑”, 那麼直接就可以改成單純的黑色
- (反正已經到root了, 所以就算把一份黑色拿掉也不會有祖先會因此少算一份黑色的node)

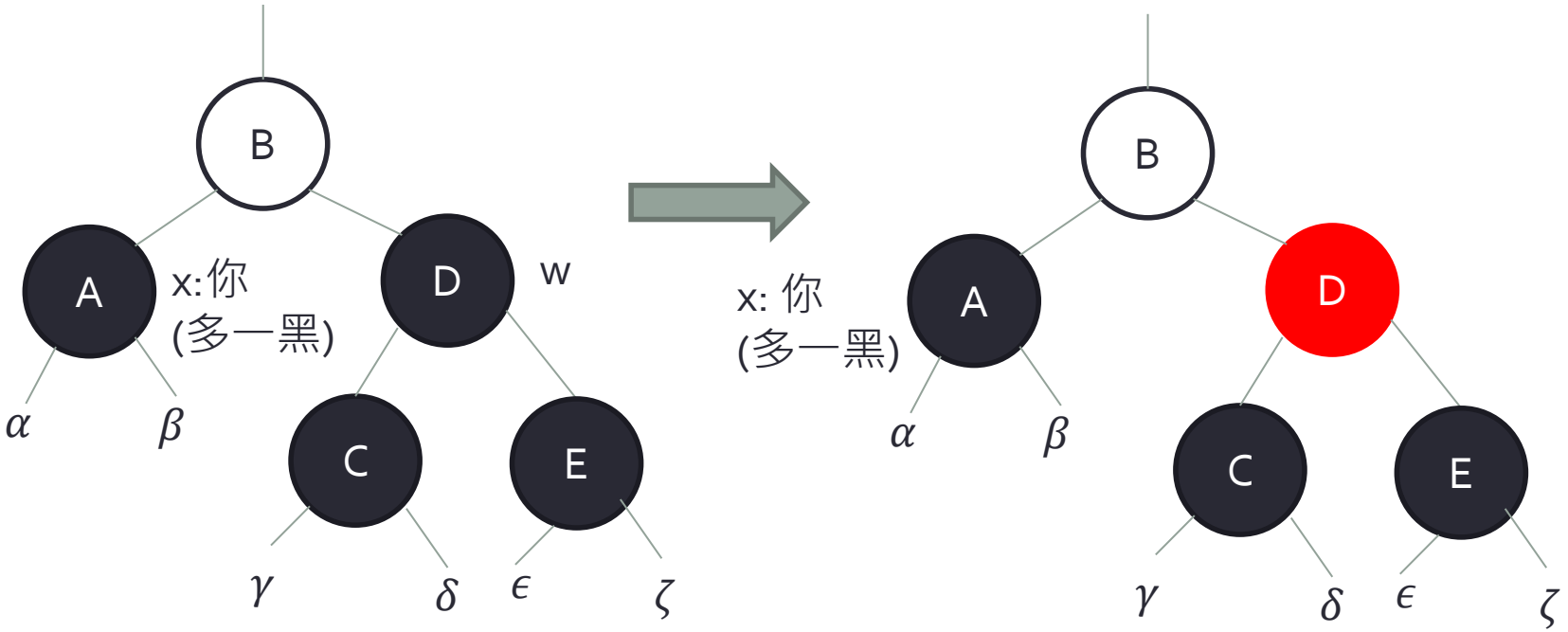
- 3. 其他的就用以下的四種case處理...(把黑與黑往root方向移動)

情形一：你的弟弟是紅的

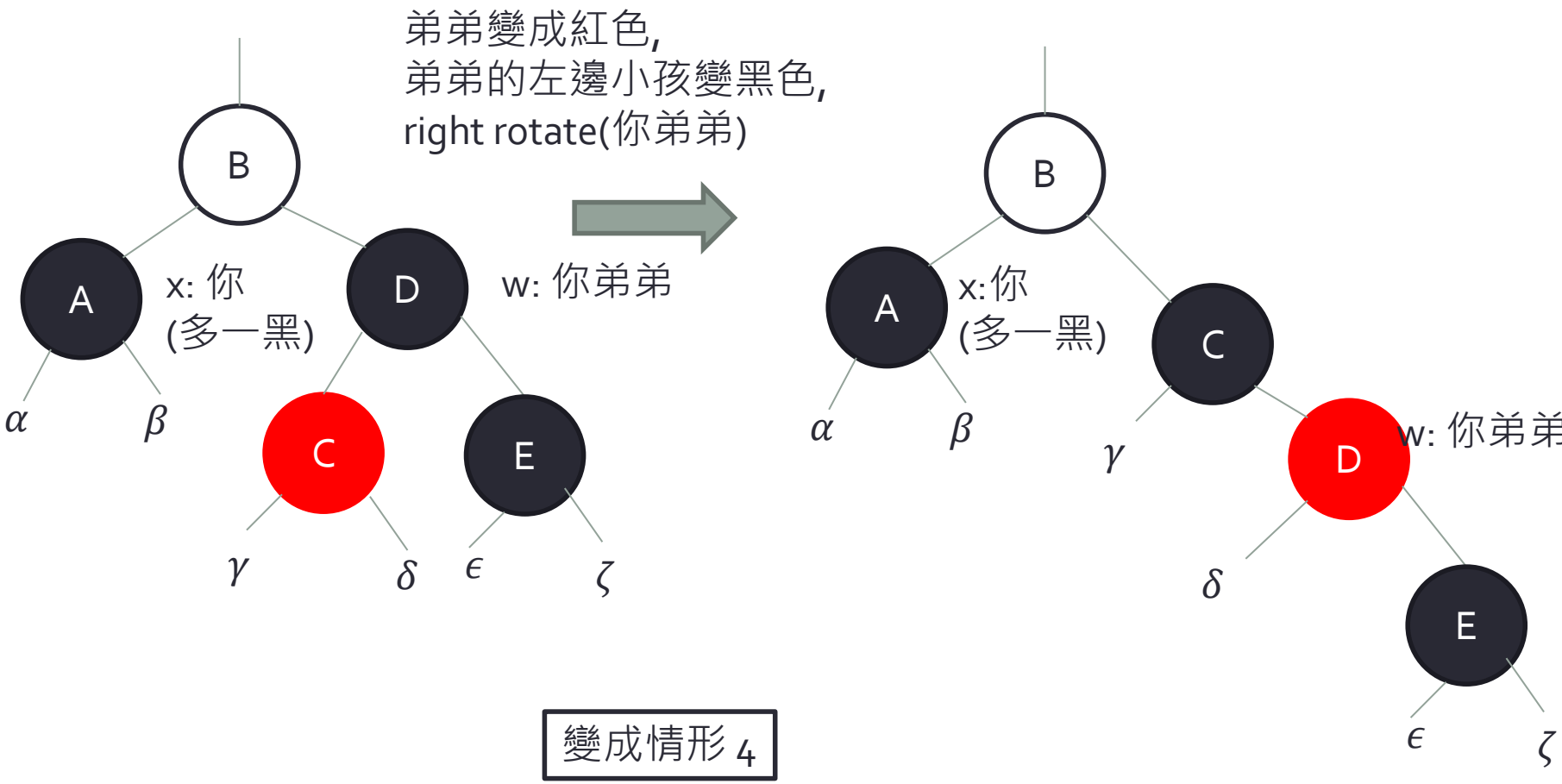


轉換成情形二、三、或四

情形二：你的弟弟是黑的& 你的姪子們都是黑的

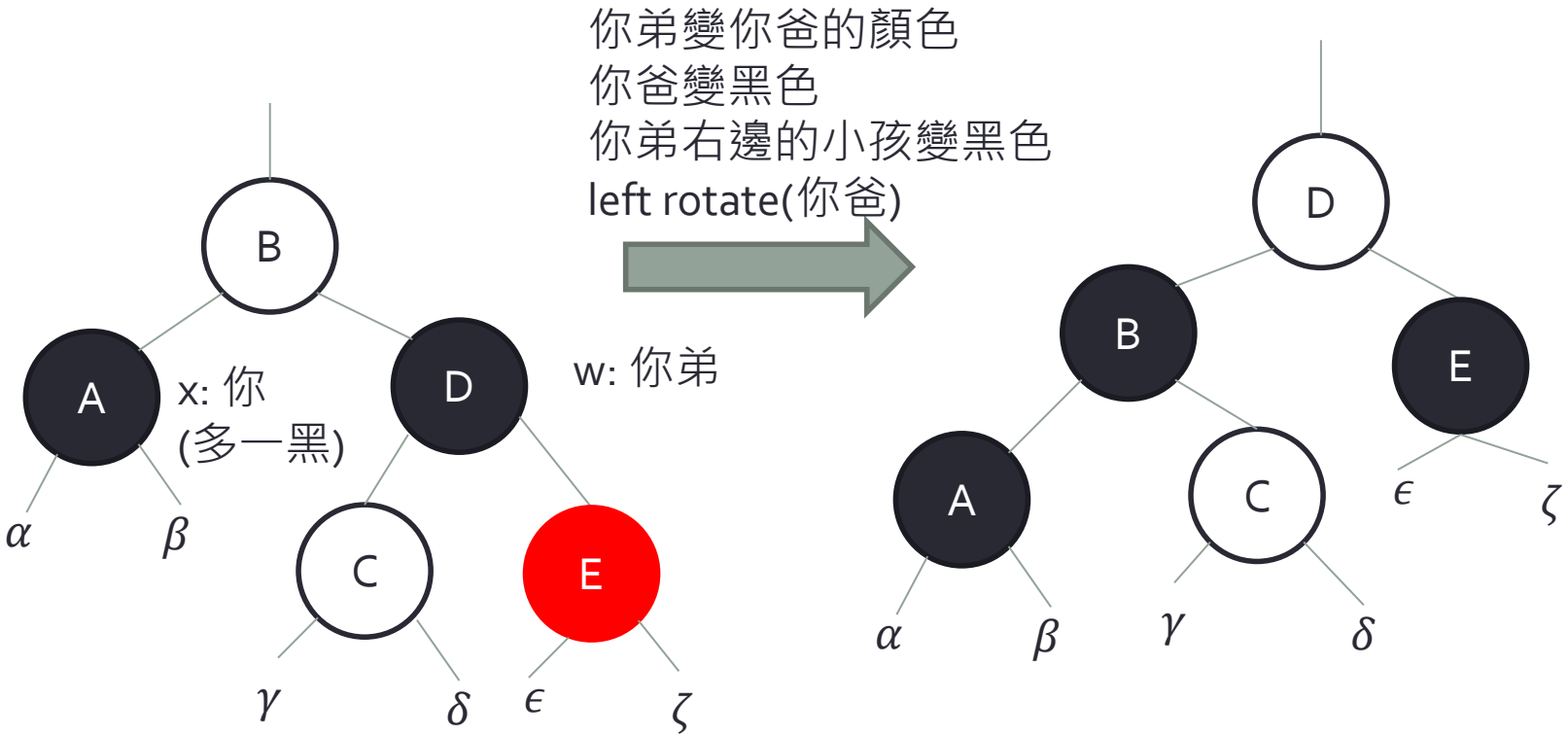


情形三：你的弟弟是黑的&
你的姪子們(弟弟的小孩)是左紅右黑



變成情形 4

情形四:你的弟弟是黑的&
你的右邊姪子(弟弟的右邊小孩)是紅的



Pseudo-code: 刪除後修理R-B tree

```
RB-Delete-Fixup(T,x)
while x!=T.root && x.color==BLACK (你不是root而且是黑的)
        if (x==x.p.left) (你是你爸的左邊小孩)
                w=x.p.right (你弟就是你爸右邊小孩)
                if w.color==RED (如果你弟是紅色的)
                        w.color=BLACK (把你弟設成黑色)
                        x.p.color=RED  (你爸設成紅色)
                        LEFT-ROTATE(T,x.p)
                        w=x.p.right      (你新弟弟是現在你爸右邊小孩)
                if w.left.color==BLACK && w.right.color==BLACK (兩個姪子都黑)
                        w.color=RED (把弟弟設成紅的)
                        x=x.p (你變成你爸)
                else if w.right.color==BLACK (你姪子右黑左紅)
                                w.left.color=BLACK  (左邊姪子設成黑的)
                                w.color=RED (你弟設成紅的)
                                RIGHT-ROTATE(T,w)
                                w=x.p.right (新弟弟是現在你爸右邊小孩)
                        w.color=x.p.color (弟弟設射程爸爸的顏色)
                        x.p.color=BLACK (爸爸設成黑的)
                        w.right.color=BLACK (右邊姪子設成黑的)
                        LEFT-ROTATE(T,x.p)
                        x=T.root (你直跳root, 準備出去)
        else (x==x.p.right) (如果你是你爸右邊的小孩,跟前面一樣, 只是left, right都反過來)
x.color=BLACK (如果跳出迴圈了, 也就是x是紅加黑, 或者是root是黑加黑, 那麼就把它設成一個黑色即可)
```

情形1

情形2

情形3

情形4

情形5-8

Delete後調整要花多少時間?

- 解決路徑圖:
- 1→ 2 → solved (從1到2的, 情形二中的圖中B原本一定是紅色, 所以可以直接解決)
- 1→3→4→solved
- 1→4→solved
- 3→4 solved
- 4 solved
- 2→解決(如果圖中B原本是紅色), or 轉到1 or 2 or 3 or 4 (x往上走一層) (如果圖中B原本是黑色)

- 所以worst case為2,2,2,...一直到root為止
- O(log n)

Today's Reading Assignment

- Cormen ch.13

- You might be interested in some other balanced binary trees:
  - AVL Tree (Cormen problem 13-3)
  - 2-3 Trees
  - B-trees, generalization of 2-3 trees (Cormen ch. 18)