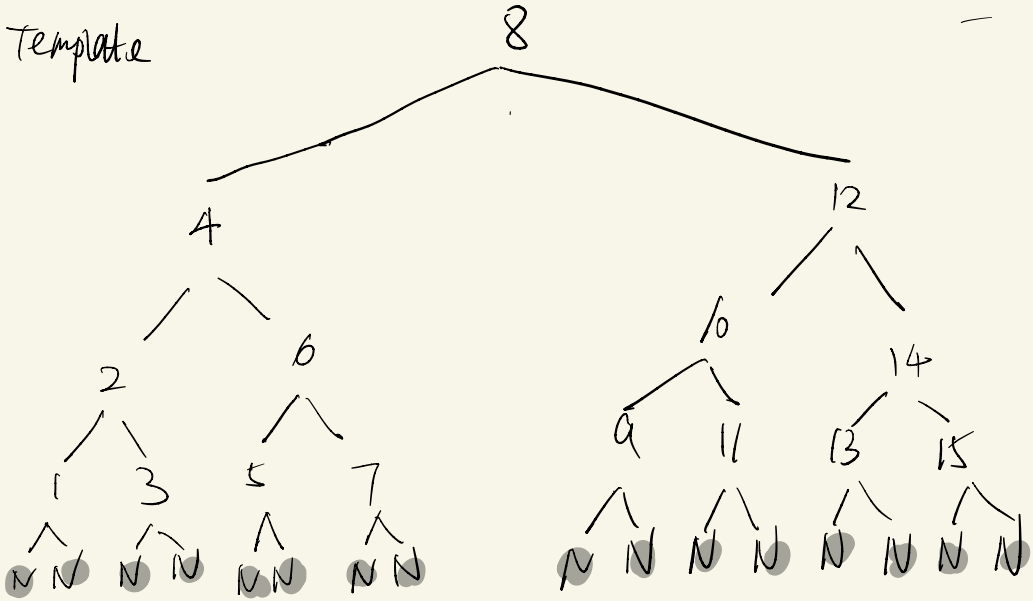
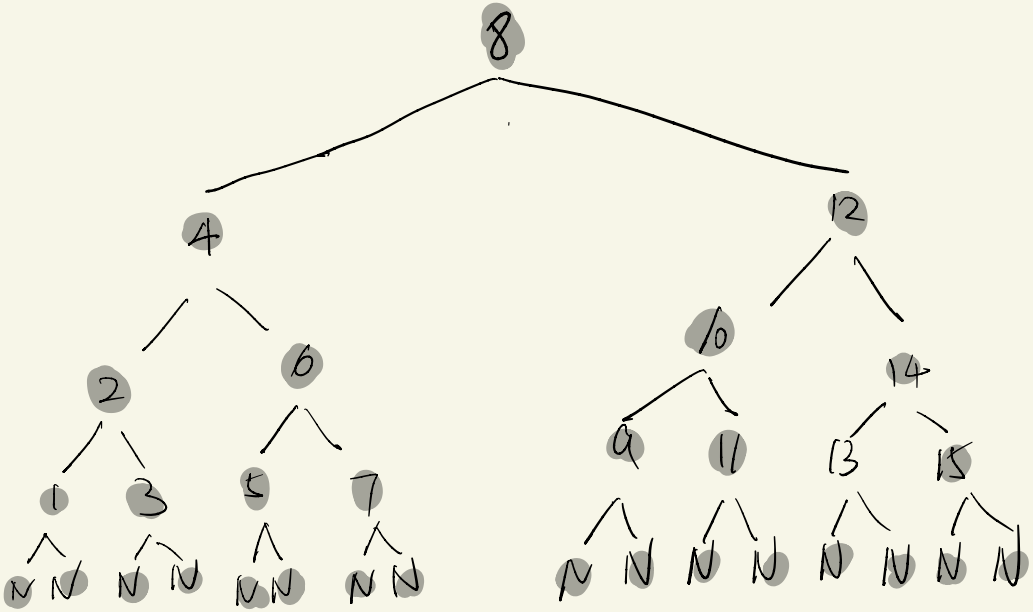
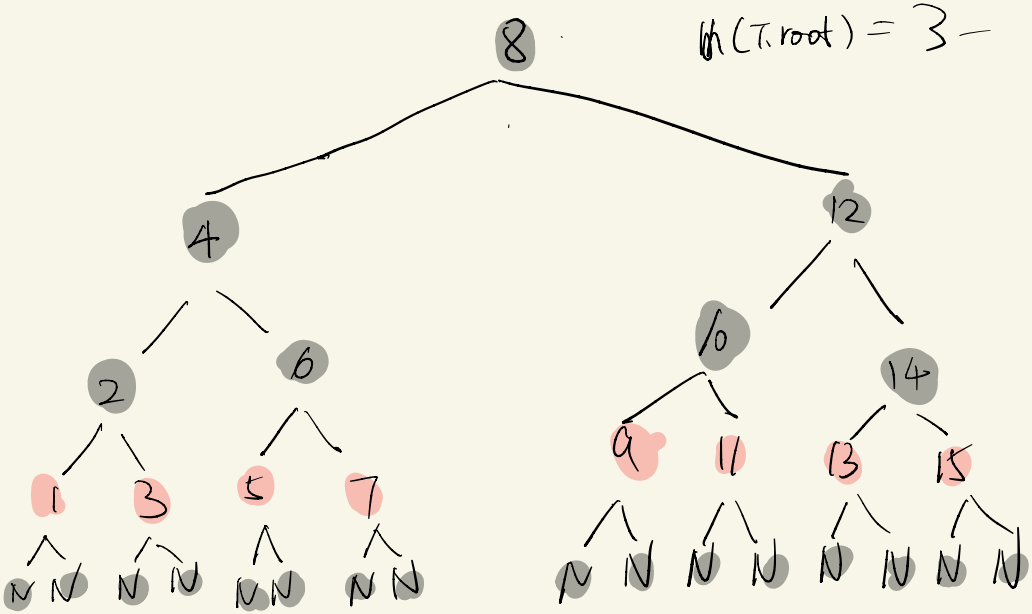
13.17

Template

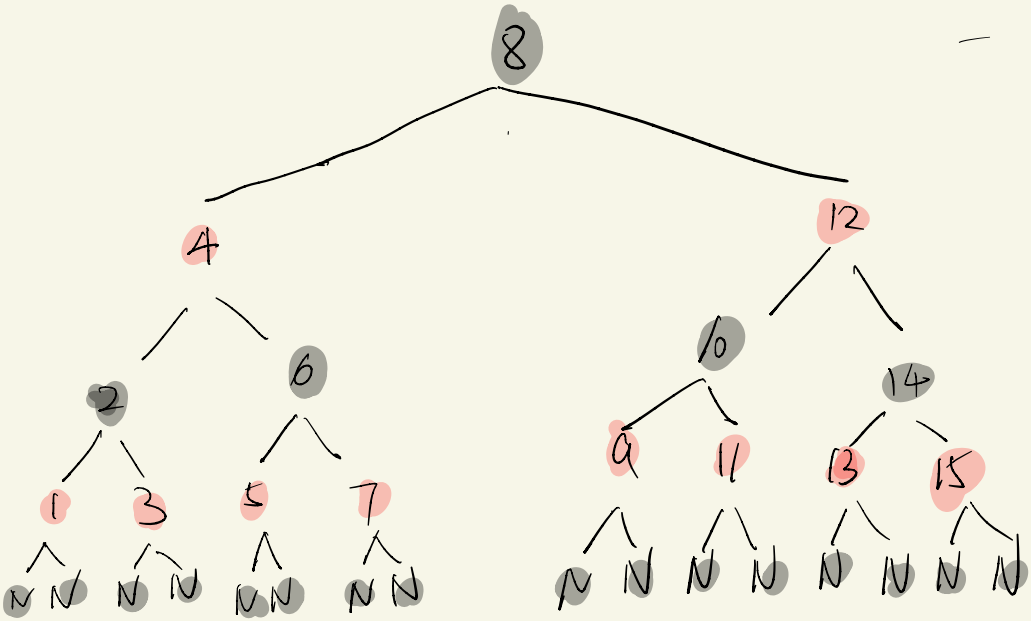


bh(T.root) = 4

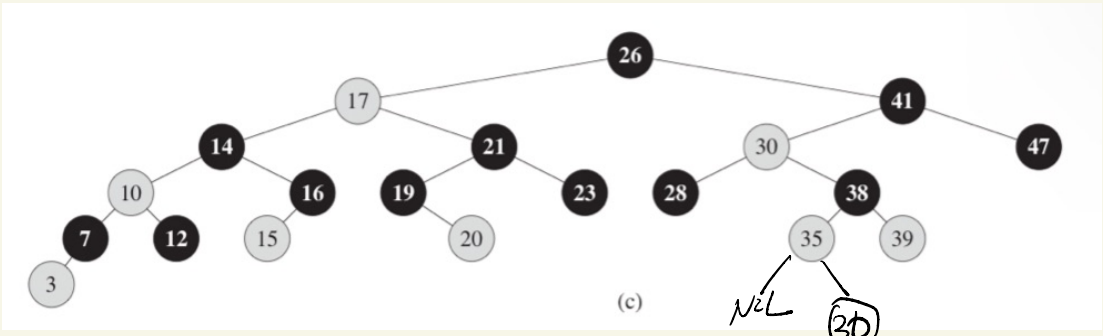



bh(T, root) = 3
8
4
12
2
6
10
14
1
3
5
7
9
11
13
15
N

bh(T, root) = 2.


8
4
12
2
6
10
14
1
3
5
7
9
11
13
15
N

13.1-2



if the inserted node is colored red, the resulting tree is not a Red-Black-Tree.

It violates that the childeren of a red parent should all be black.

No, if it is black.

It violates property 5. for each nodes, the number of black nodes should be the same for all simple paths from the node to descendant leaves.

13.1-3

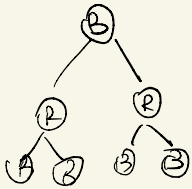
Yes, it is.

Since children of a black parent can be all black as the children of a red parent.

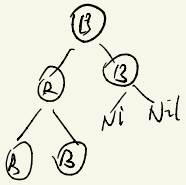
13.1-4

possible degrees:

ignore the NILs

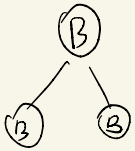


4

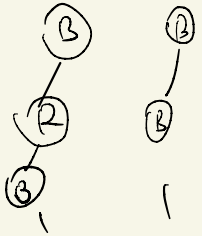


3

4, 3, 2, 1, 0.

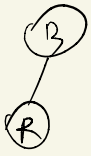


2



1

1



0

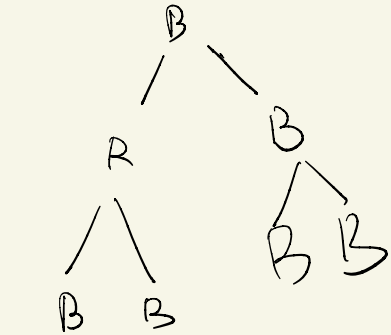
Depths of the leaves.

$$bh(T.root) > \frac{h}{2}$$

$$h' = \Omega\left(\frac{h}{2}\right).$$

13.1-5

property 5



$C_{max} \leq 2 C_{min}$

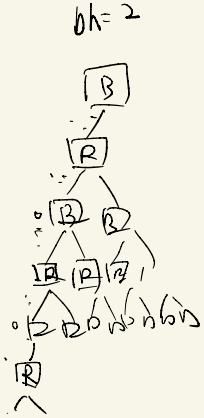
→ all black nodes

black and red nodes alternates

So that property 4 and 5 maintains

13.1-6

$h = k$



$\#IN_{max} = 2^{2k} - 1$

$\#IN_{min} = 2^{k} - 1$

2^0

$2^3 - 1$ $2^4 - 1$

13.1-7

ratio = $\frac{\text{red}}{\text{black}}$

Largest ratio = $\frac{2^1 + 2^3 + 2^5 + \cdots + 2^{2k-1}}{2^0 + 2^2 + 2^4 + \cdots + 2^{2k}} = \frac{2^1(4^0 + 4^1 + 4^2 + \cdots + 4^{k-1})}{4^0 + 4^1 + 4^2 + \cdots + 4^k}$

$= 2^1 \cdot \frac{1 - 4^{k-1+1}}{1-4}$

$\frac{1-4^{k+1}}{1-4}$

$= \frac{2 \cdot (4^k - 1)}{(4 \cdot 4^k - 1)}$

$= \frac{\frac{1}{2}(4 \cdot 4^k - 1) - \frac{3}{2}}{4 \cdot 4^k - 1}$

$= \frac{1}{2} - \frac{\frac{3}{2}}{4^{k+1} - 1}$

$= \frac{1}{2}$

k is bh of Tree

$k \to \infty$

smallest ratio = 0

Any black Tree.

This is all the nodes

---

ratio = $\frac{\#\text{IN Red}}{\#\text{IN Black}} = \frac{2^1 + 2^3 + \cdots + 2^{2k+1}}{2^0 + 2^2 + 2^4 + \cdots + 2^{2k}} = 2$

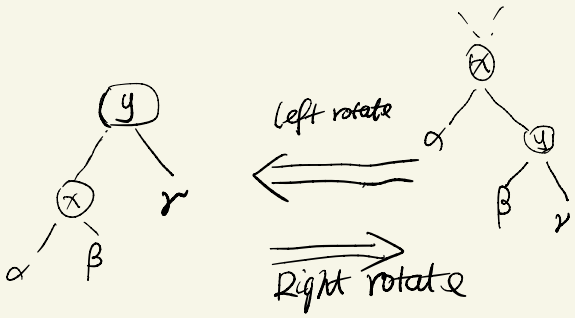
the largest ratio = 2

the smallest ratio = 0.

13.2

LEFT-ROTATE(T, x)

```
y = x.right
if y.left ≠ NIL
    y.left.p = x
x.right = y.left
if x.p == NIL
    T.root = y
elif x.p.left == x
    x.p.left = y
else x.p.right = y
y.p = x.p
x.p = y
y.left = x
```



RIGHT-ROTATE(T, y)

```
x = y.left
if x.right ≠ NIL
    x.right.p = y
y.left = x.right
if y.p == NIL
    T.root
else if y.p.left == y
    y.p.left = x
else y.p.right = x
x.p = y.p
y.p = x
x.left = y
```

13.2-2

every node can rotate with its parent

Only the root whose parent is NIL can rotate

13.2-3

a will increase by 1

b will remain the the same

c will decrease by 1.

13.2-4

① Let root and all the successive right children will be the initial right chain

② For the nodes who is a left child of the nodes in the right chain, perform a right rotation. The nodes in the right rotation will only increase, since no nodes are rotated out of the right chain.

③ keep Doing Step 2 until all the nodes are in the right chains.

The number of rotations = O(number of nodes)

$= O(n) = O(n)$

Assuming 2 arbitary binary search trees with n nodes T_1 and T_2

Assume

$\{x_1, x_2 \cdots x_p\}$

are rotations for T_1 to become a right chain.

$\{y_1, y_2 \cdots y_q\}$ are rotations for T_2 to become a left chain.

$T_1 \Rightarrow T_2$

$\{x_1, x_2 \cdots x_p\}$ $O(n)$

$\{y_1', \cdots y_q'\}$ $O(n)$

← opposite of y_1 vice versa

12.1-1

$T(n)=2^n$

$T(n)=1+\sum_{j}^{n-1} T(j)$

$n=0$ $T(0)=1=2^0$

$n=1$ $T(1)=1+\sum_{j}^{0} T(j)=2=2^1$

$n=2$ $T(2)=1+\sum_{j}^{1} T(j)=1+T(1)+T(0)=1+1+2^1=4=2^2$

Assume for $n=0,1\cdots k$ $T(n)=2^n$

$n=k+1$

$T(k+1)=1+\sum_{j}^{k} T(j)$

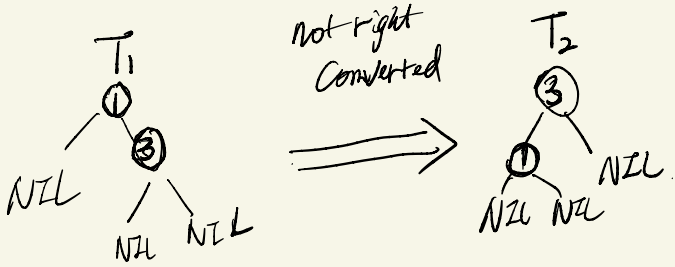
$=1+(T(0)+T(1)+\cdots T(k)$

$=1+2^0+2^1+2^2+\cdots 2^k$

$=1+\frac{1\cdot(1-2^{k+1})}{1-2}=2^{k+1}-1+1=2^{k+1}$

q.e.d.

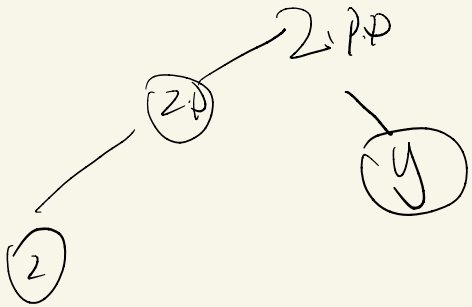
13.2-5



★ we can use $O(n)$ to rotate the node in T_2 which is T_1's root, to the root of T_1. Repeat the same process for the 2 subtrees.

Thus $O(n^2)$ right rotations can transform T_1 who is right convertible to T_2.

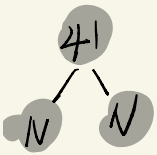
13.3



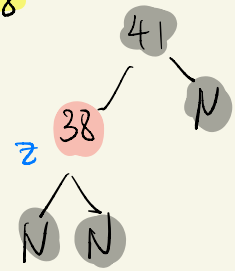
13.3-1

It violated the property 5 if we do so.

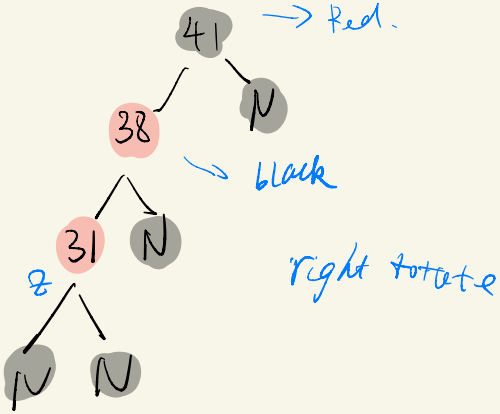
insert 41



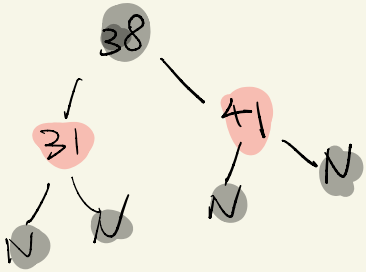
insert 38



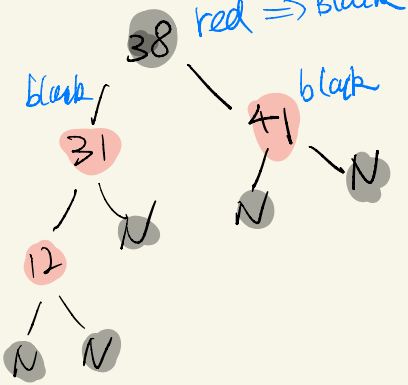
insert 31

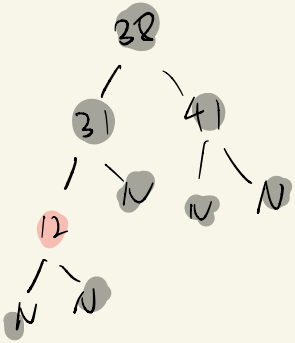


case 3



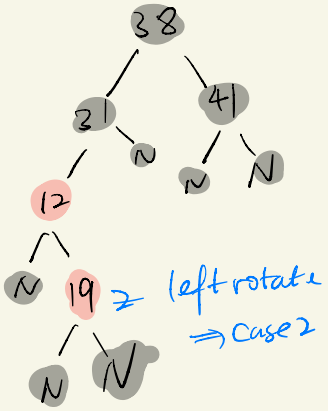
insert 12





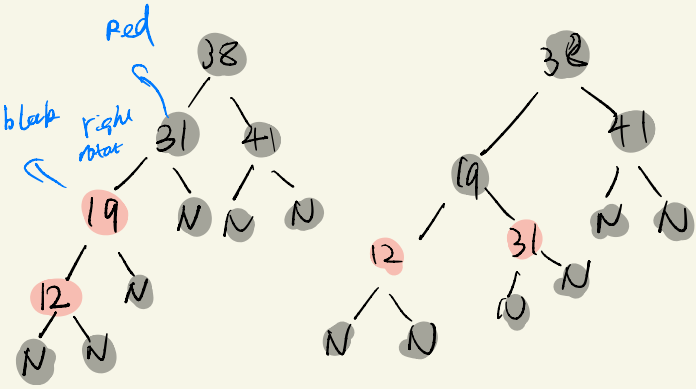
case 1: red uncle

insert 19



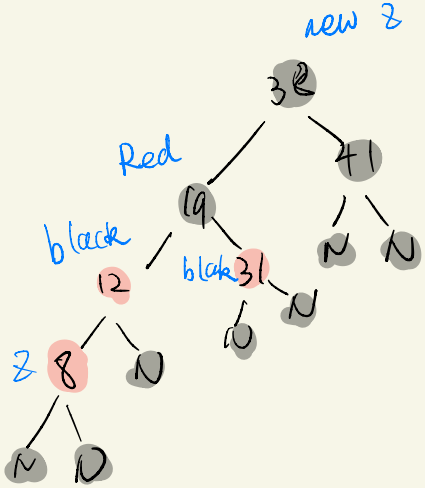
left rotate
⇒ case 2

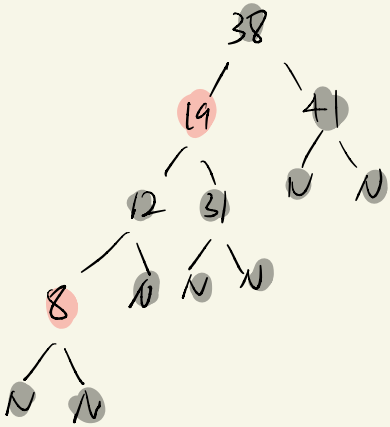
case 3 black uncle
right child



case 2 black uncle
left child

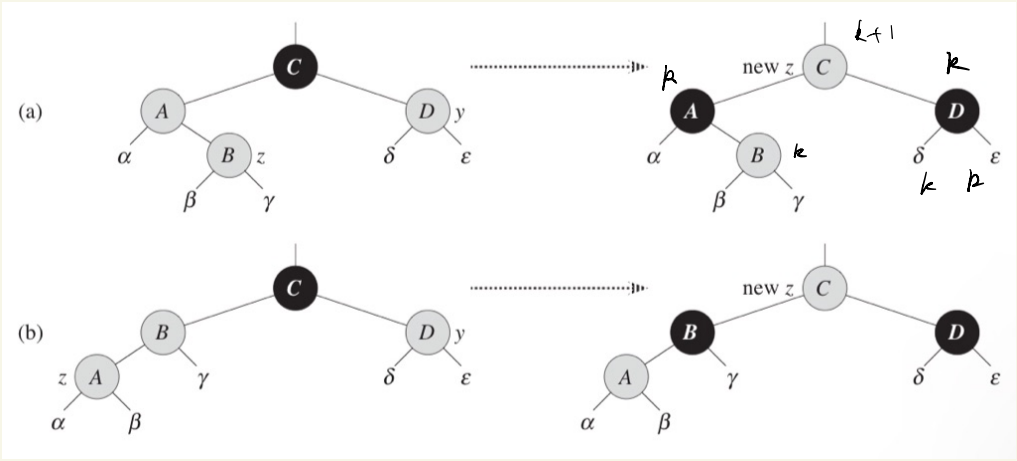
insert 8

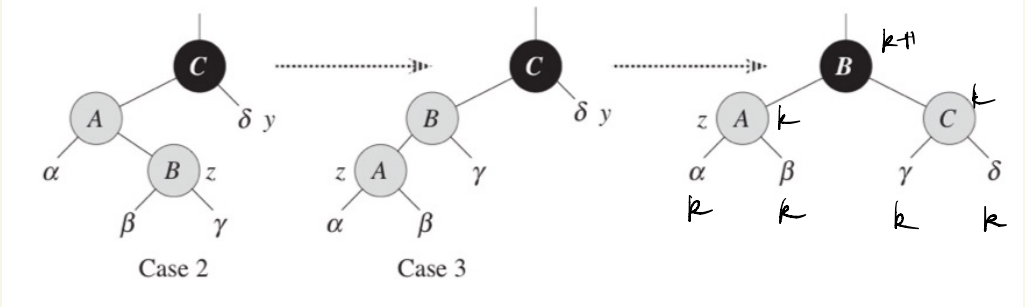




Case #1 : Red uncle

13.3-3


(a)
C
A
D
y
α
B
z
δ
ε
β
γ
new z
C
k+1
k
h
A
D
α
B
k
δ
ε
k
h
β
γ
(b)
C
B
D
y
z
A
γ
δ
ε
α
β
new z
C
B
D
A
γ
δ
ε
α
β


C
A
δ y
α
B
z
β
γ
Case 2
C
B
δ y
z
A
γ
α
β
Case 3
B
k+1
z
A
k
C
k
α
β
γ
δ
k
k
k
k

13.3-4

Since when z is T.root

z.p is T.nil whose color is black

Then the loop will terminates when z.p.color == black.

When the while loop terminates. there is no possibility that we can set T.Nil to RED. Since the final line will reset the color of T.NIL to Black.

13.3-5

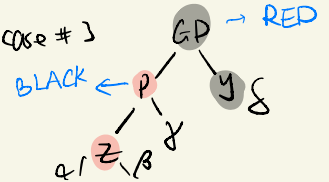
The node z is always red.

13.3-6

use a stack to record the path

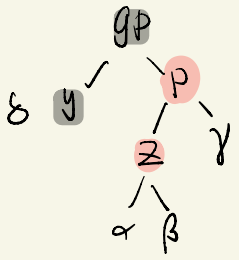
```
RB-INSERT(T, z)
y = T.NIL
x = T.root
S = new STACK()

while x != NIL
    S.PUSH(y)
    y = x

    if x.key > z.key
        x = x.left
    else
        x = x.right
if y == T.NIL
    T.root = z

else if y.key > z.key
        y.left = z

else    y.right = z

z.color = RED
z.left = T.NIL
z.right = T.NIL
RB-INSERT-FIXUP(T, z, S)
```

case # 3

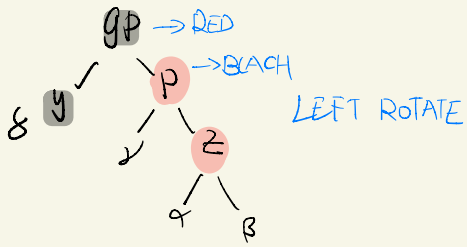



```
RB-INSERT-FIXUP(T, z, S)
parent = S.pop()
while parent.color == RED
    grandparent = S.pop()
    if parent == grandparent.left
        y = grandparent.right
        if y.color == RED                    # case 1
            grandparent.color = RED
            parent.color = BLACK
            y.color = BLACK
            z = grandparent
            parent = S.pop()
        else
            if z == parent.right             # case 2
                LEFT-ROTATE(T, parent)
                z = parent
            grandparent.color = RED          # case 3
            parent.color = BLACK
            RIGHT-ROTATE(T, grandparent)
    else parent == grandparent.right
        y = grandparent.left
        if y.color == RED
            grandparent.color = RED
            parent.color = BLACK
            y.color = BLACK
            z = grandparent
            parent = S.pop()
```

case # 2



RIGHT-ROTATE



Case# 3

LEFT ROTATE

---

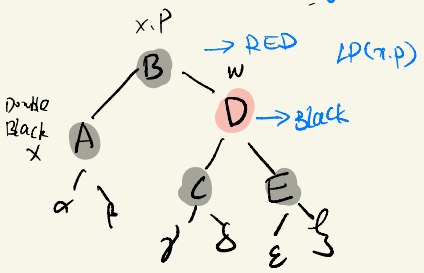
case # 2

```
else
    if z == p.left
        RIGHT-ROTATE(T, p)
        z = p
    grandparent.color = RED
    parent.color = BLACK
    LEFT-ROTATE(T, grandparent)
```

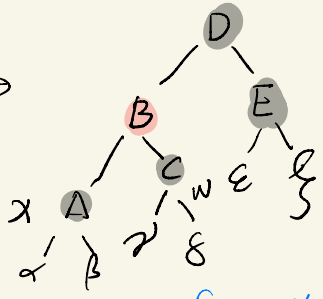
case # 3

T.root.color = RED

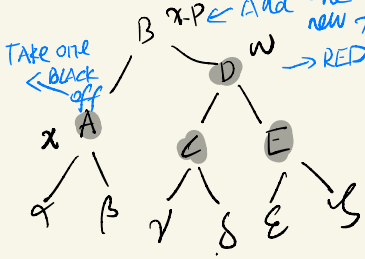
13.4 case# 1: red sibling for x — case # 2, 3, 4: Black sibling for x

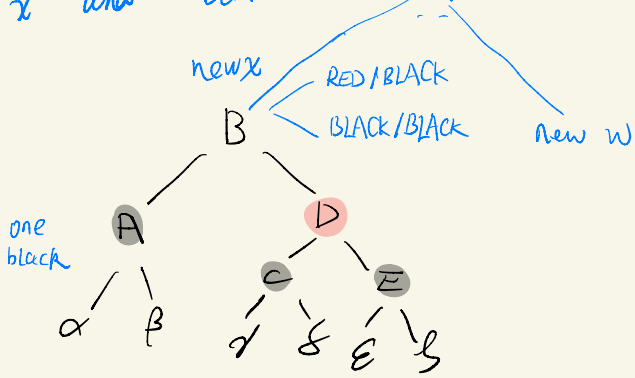


Case 1



Case #2: Black sibling for x and black children for w

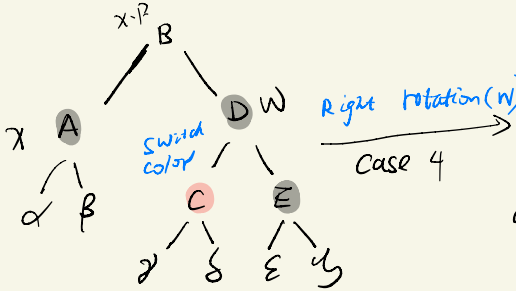




case 1 ⟹ case 2 now x. B.color = RED.

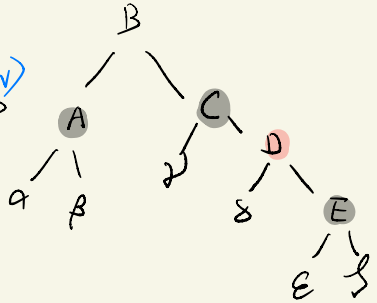
the loop terminates.

case #3: Black sibling for x, left red child for w and right black child for w.

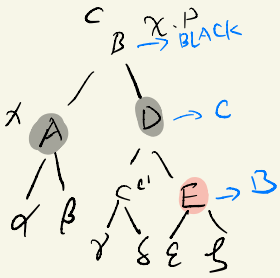


Right rotation(w)

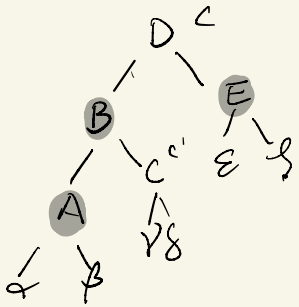
Case 4



Case 4: Black sibling for x, black left child and red right child for w



LEFT-ROTATE(B)



new $x := T.root$

w RED → case 1 → case 2 → w children all black → $x.color == RED$ or $x == T.root$ loop terminates

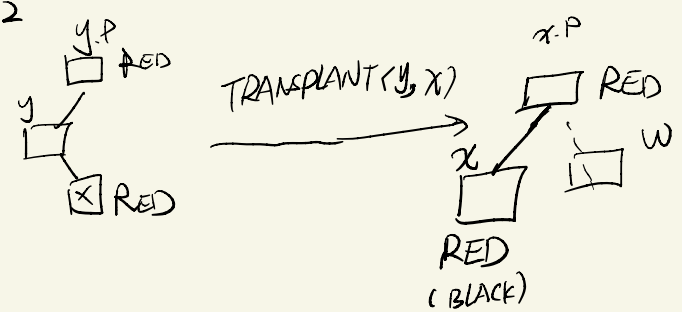
w Black: case 3 → w left child RED, right child Black → case 4 → w right child RED ⟹ $x = T.root$ loop terminates

13.4-1

if the while loop terminates with the condition that $x \neq T.root$, then line 23 maintains property 2.

if the while loop terminates with the condition that $x.color == BLACK$, we haven't touch the root yet.

13.4-2

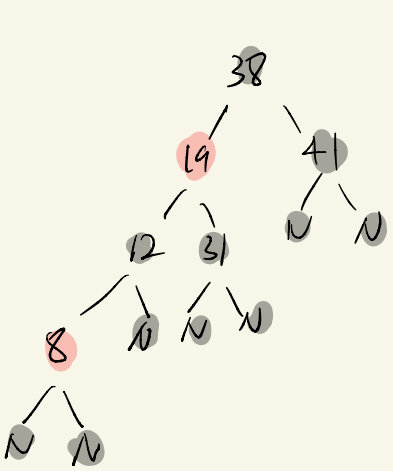


The while loop ends since it violates the condition "$x.color == BLACK$". In other words, we have a RED-and-BLACK x node.

The line 23 colors x to BLACK.

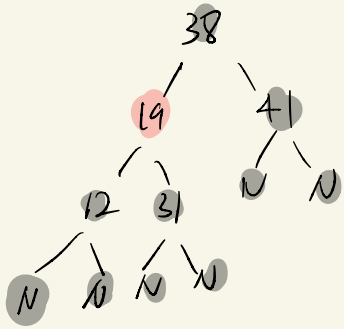
Thus property 4 is restored.

13.4-3

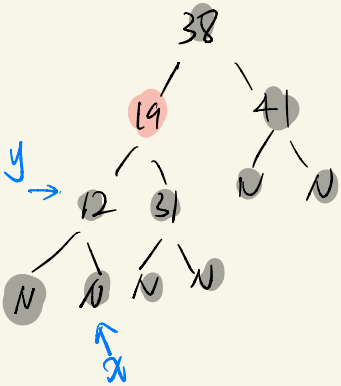


RB-DELETE(8)

No need for FIXUP.

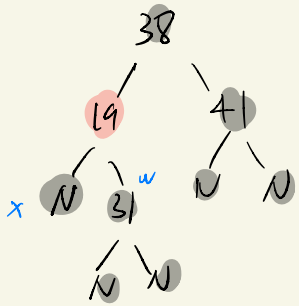


After DELET



RB-DELETE(12)

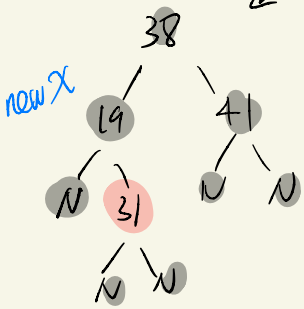
Run FIXUP

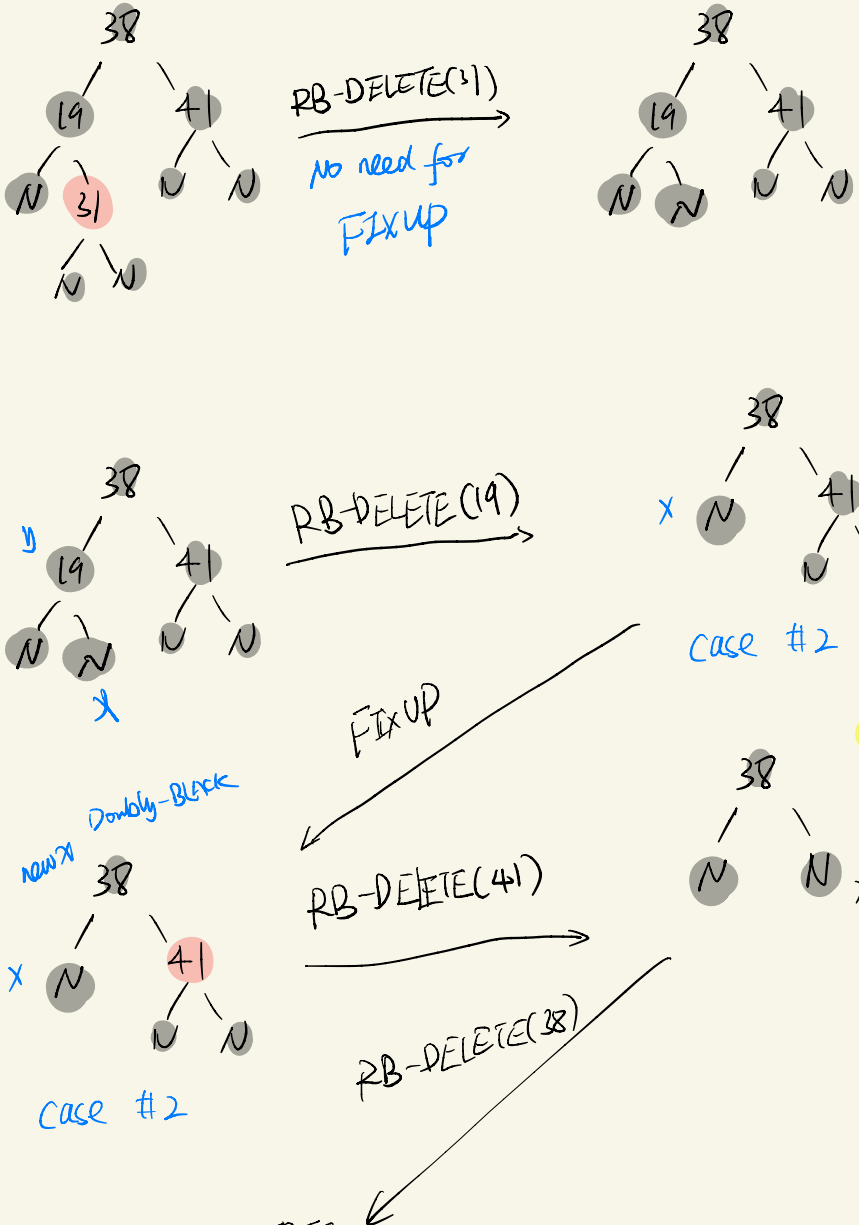


BLACK sibling

BLACK children for the sibling

case #2




38
19
41
N
31
N
N
N
N
RB-DELETE(31)
No need for FIXUP
38
19
41
N
N
N
N
38
y
19
41
N
N
N
N
x
RB-DELETE(19)
38
x
N
41
N
N
Case #2
FIXUP
x is right child
Doubly-Black
new x
38
x
N
41
N
N
38
N
N
x
RB-DELETE(41)
Case #2
RB-DELETE(38)
EMPTY-RB-TREE
T.NIL

13.4-4

w can be nil when y is red.

Otherwise the property 5 (equal Black height) doesn't hold.

In this case, the RB-DELETE-FIXUP is not executed.

Thus, only x can be nil.

In the while loop x can't be the root. so x.p is not nil.

Then Line 23 "x.color = BLACK" is the line that modify or examine (could)

T.Nil.

w's children could both be NIL, which is included in case 2

Line 9 could examine the T.Nil.

w's right child is NIL, which is the case 3.

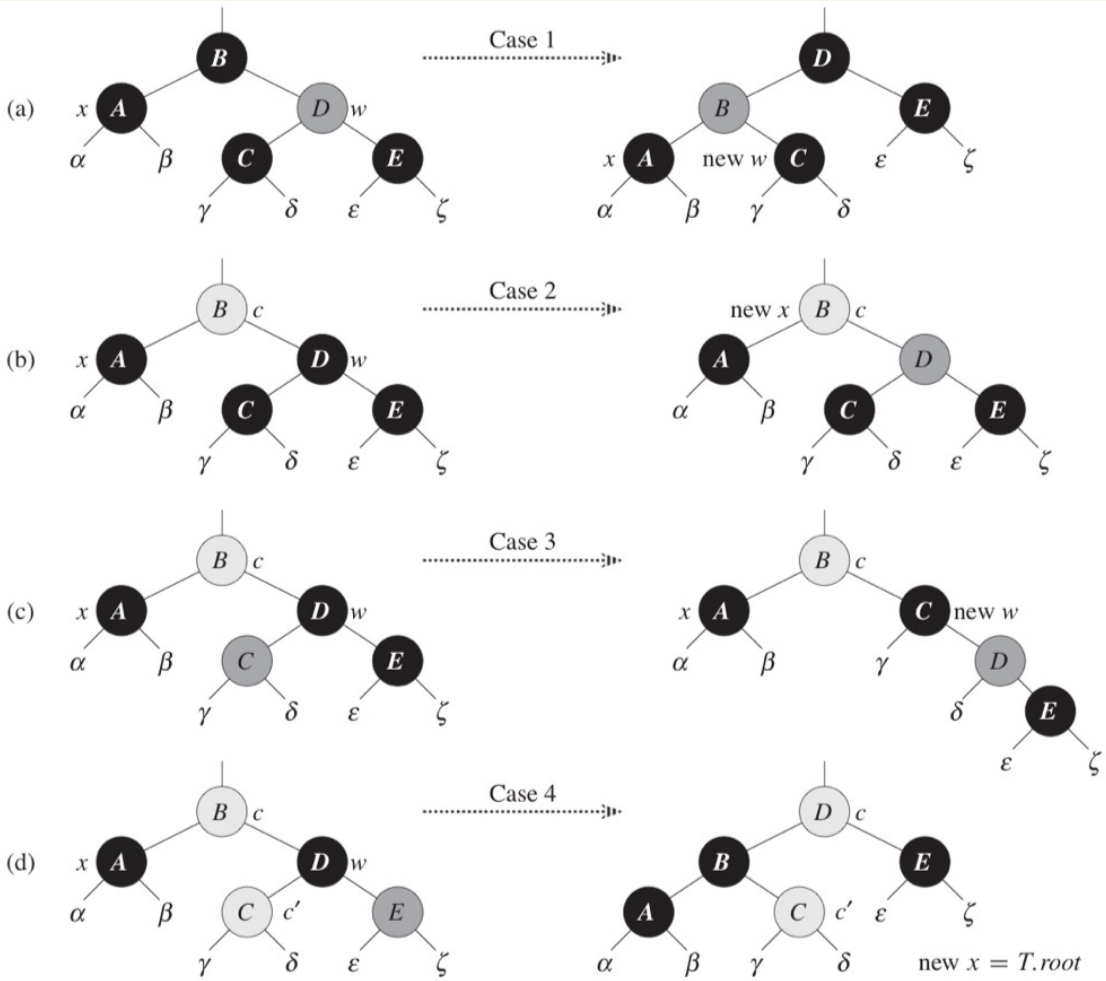
line 12 could examine the T.NIL.

w's left child could be NIL, which is the case 4.

But no line examine or modify T.NIL.

In summary, Line 23 could modify the T.NIL node, while line 9 and line 12 could only examine the T.NIL node.

B.4.5



	before						after					
	α	β	γ	δ	ε	ζ	α	β	γ	δ	ε	ζ
case 1	3	3	2	2	2	2	3	3	2	2	2	2
case 2	2+c	2+c	2+c	2+c	2+c	2+c	2+c	2+c	2+c	2+c	2+c	2+c
case 3	2+c	2+c	1+c	1+c	2+c	2+c	2+c	2+c	1+c	1+c	2+c	2+c
case 4	2+c	2+c	1+c+c'	1+c+c'	1+c	1+c	2+c	2+c	1+c+c'	1+c+c'	1+c	1+c

13.4-6

since W is RED in case #1.

x.p is also the parent of W.

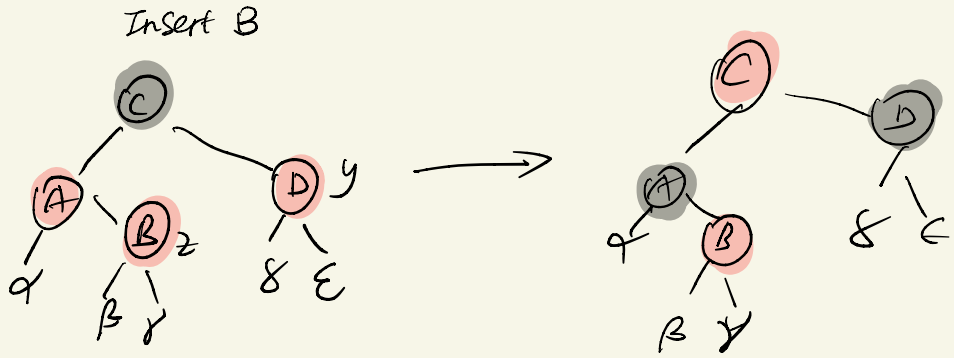
Thus, x.p can not be red but black.

Otherwise, property (Red parent, Black children) is violated.

13.4-7

NO.

Insert case 1



RB-DELETE(B). The Color of C, A, D is different as before.

13-1

a. INSERT. from the root the leaf node inserted

DELETE From the root to the successor of the node deleted.

b. PERSISTENT-TREE-INSERT (T, k)

```
y = NIL
x = T.root
x' = COPY-NODE(x)
T'.root = x'
while x ≠ NIL
    y' = x'
    if x.key < k
        x' = COPY-NODE(x.right)
        y'.right = x'
        y'.left = x.left
        x = x.right
    else
        x' = COPY-NODE(x.left)
        y'.left = x'
        y'.right = x.right
        x = x.left
newNode = new NODE(key = k)
if y' == NIL
    T'.root = newNode
else if k < y'.key
        y'.left = newNode
else    y'.right = newNode
return T'.root.
```

```
COPY-KEY(node)
    if node == NIL
        return NIL
    else
        newNode.key = node.key
        return newNode
```

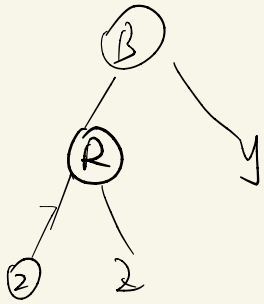
c. Both $O(h)$

d. Because a node can't have 2 parents.
So we are copying all the nodes in the tree.

e. Both RB-DELETE, RB-INSERT changes
$O(\lg n)$ nodes, which will be copied.
Thus the space are $O(\lg n)$

13.2

a.




```
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                          Case #1
            y.color = BLACK
            z.p.color = BLACK
            z = z.p.p
            z.color = RED
        else if z == z.p.right                     Case #2
            z = z.p
            LEFT-ROTATE(T, z)
        z.p.color = BLACK                          Case #3
        z.p.p.color = RED
        RIGHT-ROTATE(T, z.p.p)
    else (symmetric (switch "right" and "left"))

if T.root.color == RED
    T.root.color = BLACK
    T.bh = T.bh + 1
```

Constant O(1) in space and time.

RB-DELETE

if x == T.root

T.bh = T.bh - 1

O(1) in time and space

x.color = BLACK

we can determine the bh height of every node in O(n), or, amortized O(1).

13-2

RB-MAXIUM-BLACK-HEIGHT(T_1, T_2)

keep going right until we have visited (T_1.bh − T_2.bh) # black nodes

$x = T_1$.root

$y = T_1$.NIL

BNVisited = 0

while $x \neq T_1$.NIL and BNVisited $\leq$ (T_1.bh − T_2.bh)

    y = x

    if x.color == BLACK

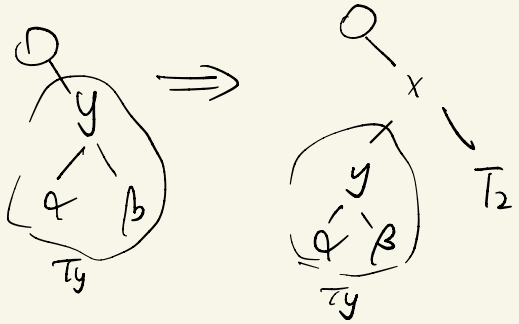
        BNVisited += 1

    x = x.right

return y.key

$O(T_1.bh - T_2.bh) = O(T_1.bh) = o(\lg n)$

c. since $x > y$



y.P.right = ($T_y \cup (x) \cup (T_2)$).root

$x > y > 0$ no binary search property is violated.

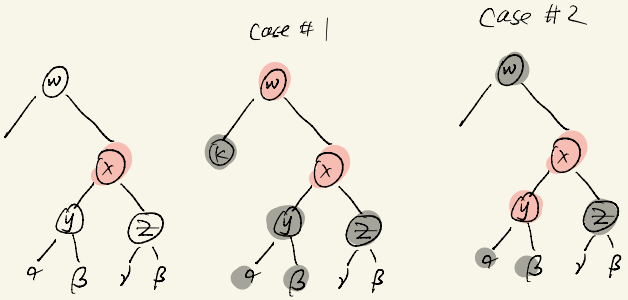
d. x.color == RED

x can't be T.NIL. Thus, property 3 (T.NIL is BLACK) holds.

property 1 (Every node is either BLACK or RED) holds obviously.

property 5: Since both T_1 and T_2 have the same black height.

Thus property 5 (every simple path from a node to a leaf node contains the same number of BLACK nodes).



RB-JOIN-FIXUP(T_1, w, x, y)
if w.color is RED
  RB-INSERT-FIXUP(T_1, x)
else if y.color is RED
  RB-INSERT-FIXUP(T_1, y)

$O(h) = O(\lg n)$

e. if $T_1.bh \le T_2.bh$
  Then T_y comes from T_2

f. b $O(\lg n)$
  c $O(1)$
  d $O(\lg n)$

Then RB-JOIN $O(\lg n)$

13-3

a $T(h) \geq T(h-1) + T(h-2) + 1$

$T(h)$ is the minimum nodes for height h.

since $T(h-1) > T(h-2)$

$T(h) \geq 2T(h-2)$ $\quad T(0) = 1$ $\quad T(1) = 2$

Then $T(h) \geq 2^{\frac{h}{2}} \iff \log T(h) \geq \frac{h}{2} \iff h < 2 \log T(h)$

Thus $h = O(\log n)$

https://people.csail.mit.edu/alinush/6.006-spring-2014/avl-height-proof.pdf

b. BALANCE(x)

case #1: left-left

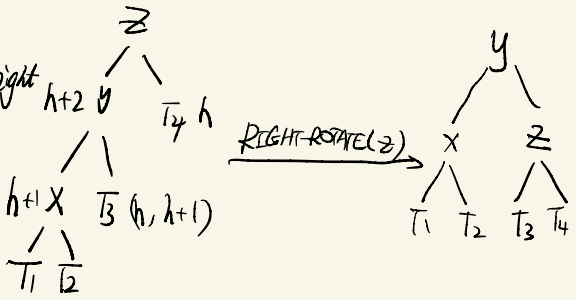
if $x.left.height - x.right.height == 2$

  if $x.left.right.height > x.left.left.height$

    LEFT-ROTATE(T, x.left.right)

  T.root = x.left

  RIGHTROTATE(T, x)



else if $x.right.height - x.left.height == 2$ case #2: left-right

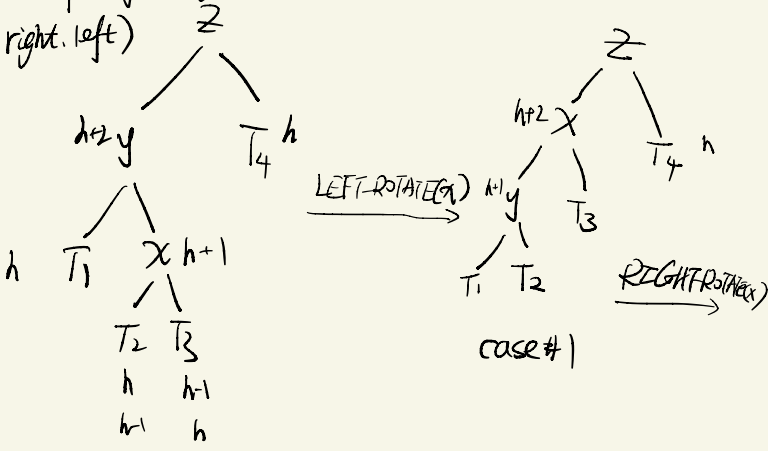
  if $x.right.left.height > x.left.right.height$

    RIGHT-ROTATE(T, x.right.left)

  T.root = x.right

  LEFT-ROTATE(x)

return T.root



the heights are updated in ROTATIONS.

case #3: right-left