アルゴリズムの設計と解析

教授: 黄 潤和 （W4022）

rhuang@hosei.ac.jp

SA: 広野 史明 （A4/A8）

fumiaki.hirono.5k@stu.hosei.ac.jp

Contents (L7 – Search trees)

- Searching problems
- Red Black Tree (insertion and deletion)

Outline

- What is Red-Black?
  赤黒木とは？
- From (2,4) trees to red-black trees
  2-4木から赤黒木へ
- Red-black tree　赤黒木
  - Insertion　挿入
    - restructuring　再構築
    - recoloring　再色付け
  - Deletion　削除
    - restructuring　再構築
    - recoloring　再色付け
    - adjustment　調整

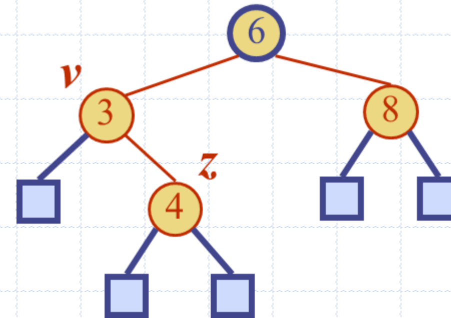
Red-Black Trees

video lecture

http://videolectures.net/mit6046jf05_demaine_lec10/

Watch about 25 minutes
to feel how a top university in the world gives lectures of CS

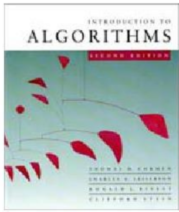
MIT - Massachusetts Institute of Technology



QS World University Rankings rates MIT No. 1 in 12 subjects for 2016 ...

news.mit.edu/2016/qs-world-university-rankings-rates-mit-no-1-in-... ▾ このページを訳す

2016/04/08 - QS World **University Rankings** has unveiled its lineup of the world's top universities for **2016**, by subject. **MIT** was honored with 12 No. 1 subject rankings, and 19 total top rankings (No. 5 or higher) out of 42 subjects.



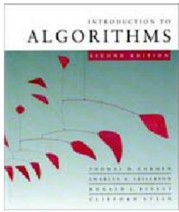
Red-black trees

This data structure requires an extra one-bit color field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

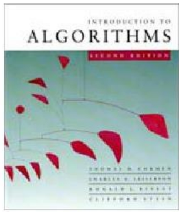
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



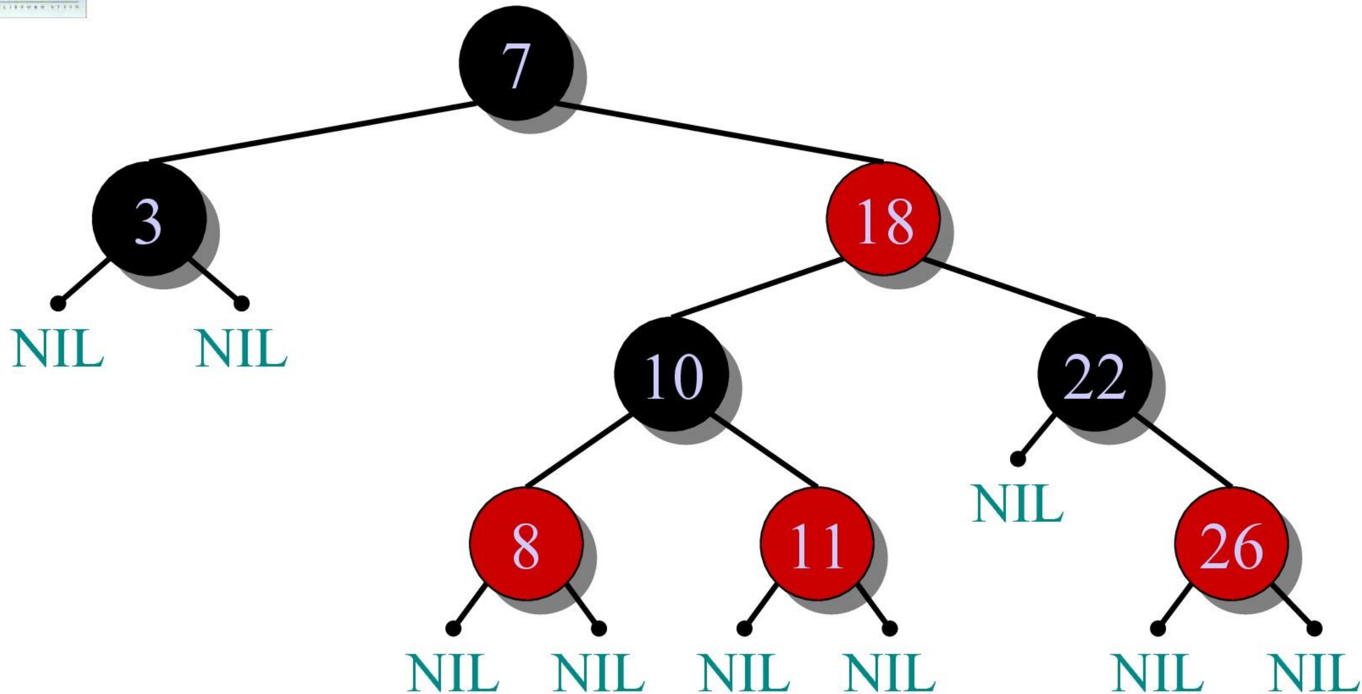
Example of a red-black tree

7
3
18
NIL
NIL
10
22
h = 4
NIL
8
11
26
NIL
NIL
NIL
NIL
NIL
NIL

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

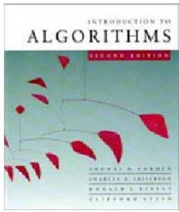


Example of a red-black tree

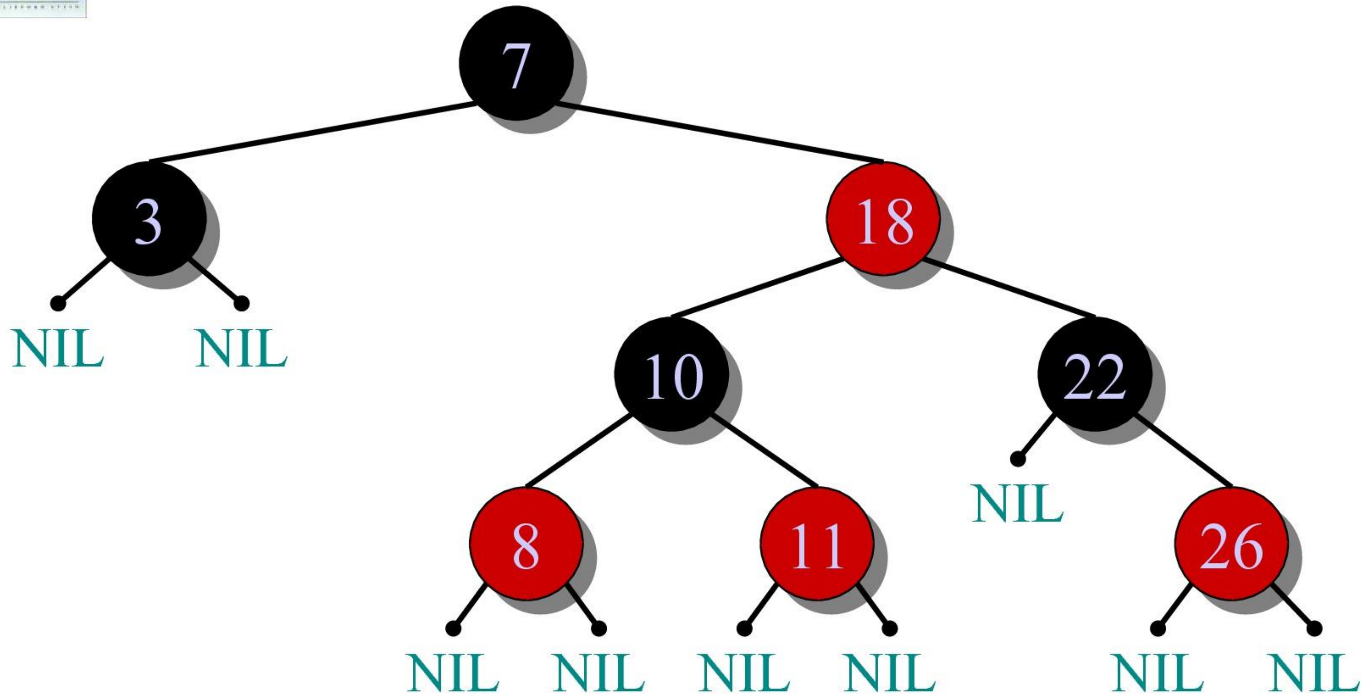


1. Every node is either red or black.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

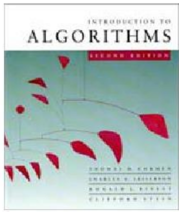


Example of a red-black tree

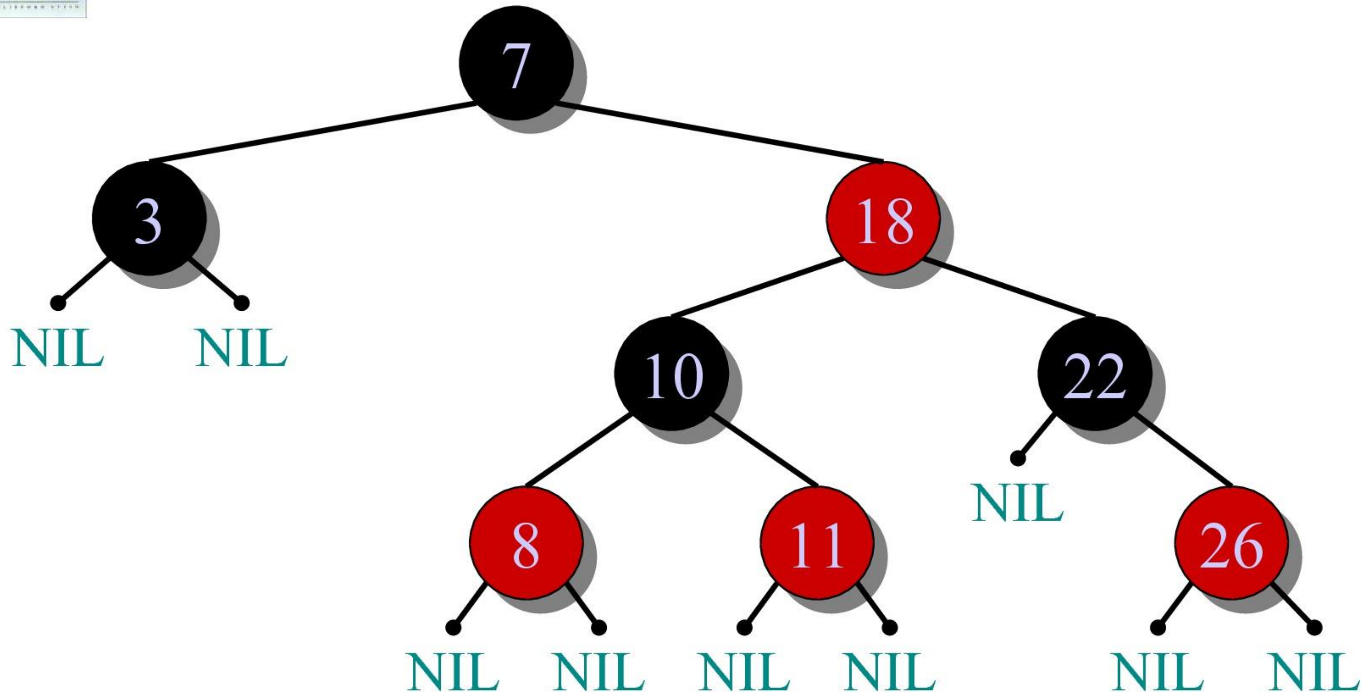


2. The root and leaves (NIL's) are black.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

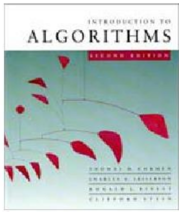


Example of a red-black tree

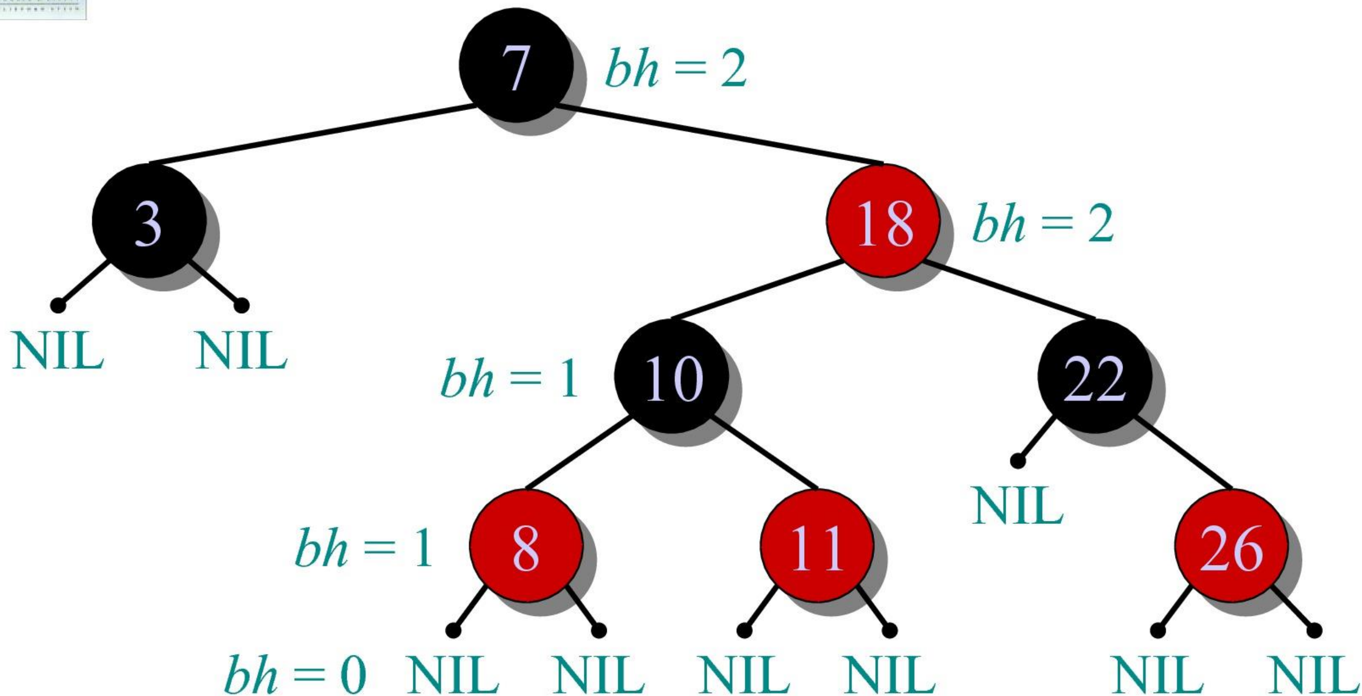


3. If a node is red, then its parent is black.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

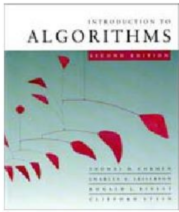


Example of a red-black tree



4. All simple paths from any node x to a descendant leaf have the same number of black nodes = *black-height*(x).

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

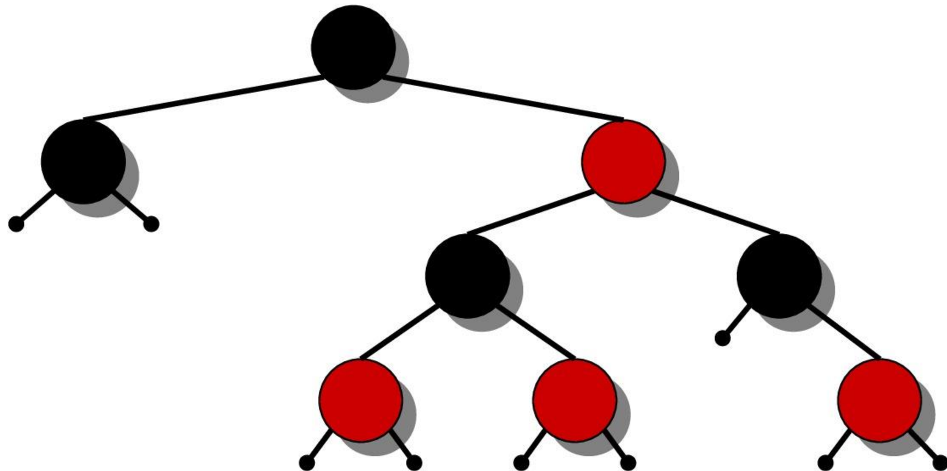
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

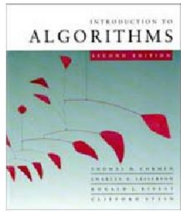
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

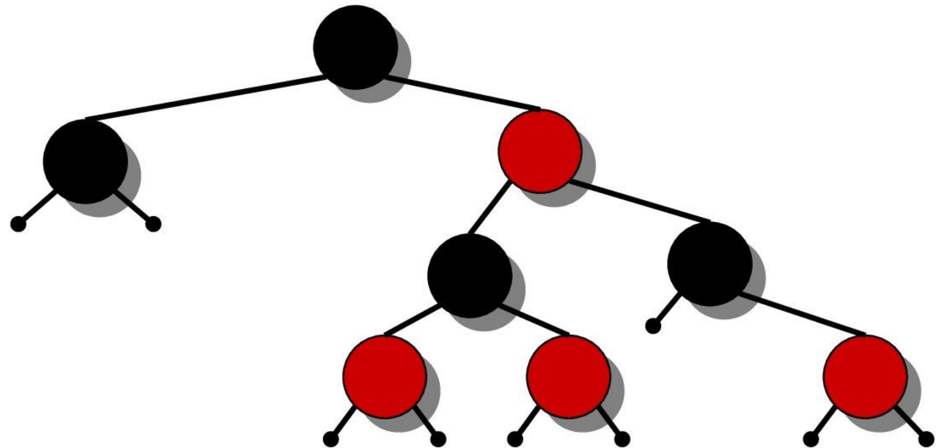
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

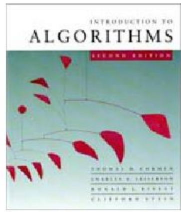
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

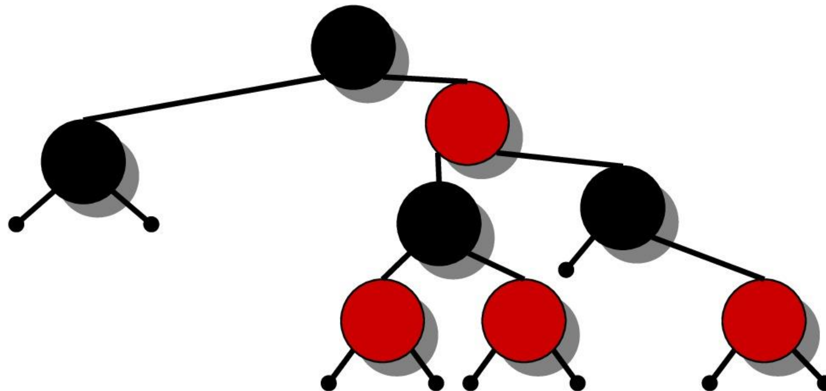
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

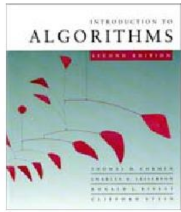
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Height of a red-black tree

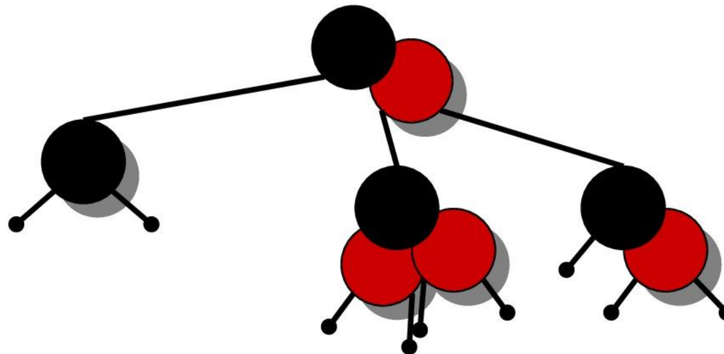
Theorem. A red-black tree with n keys has height

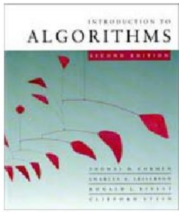
$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

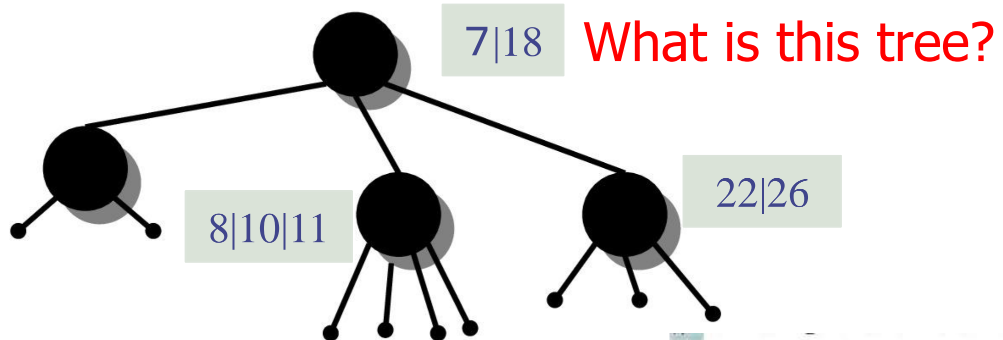
Theorem. A red-black tree with n keys has height

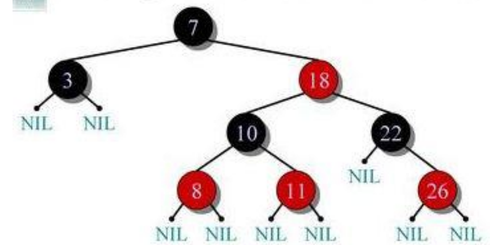
$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





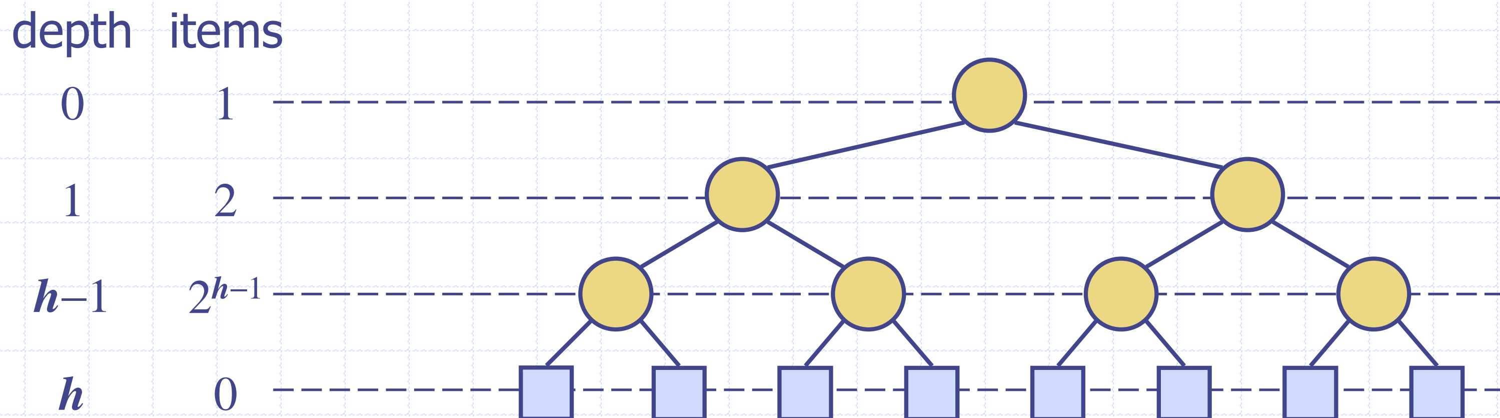
This is red-black tree?

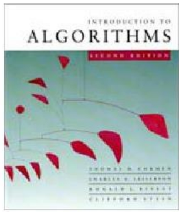
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Height of a (2,4) Tree
(2,4)木の高さ

- **Theorem:** A (2,4) tree storing n items has height $O(\log_2 n)$

  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have
  $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log_2 (n+1)$
- Searching in a (2,4) tree with n items takes $O(\log_2 n)$ time

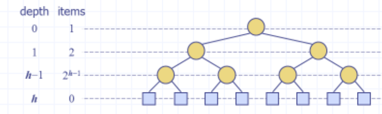




Height of a red-black tr

Height of a (2,4) Tree
(2,4)木の高さ

- Theorem: A (2,4) tree storing n items has height $O(\log_2 n)$
  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have
    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log_2 (n+1)$
- Searching in a (2,4) tree with n items takes $O(\log_2 n)$ time



16

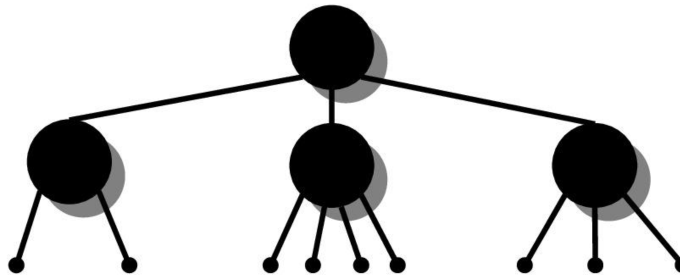
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

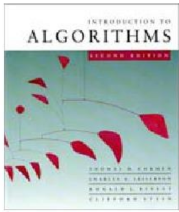
INTUITION:

- Merge red nodes into their black parents.

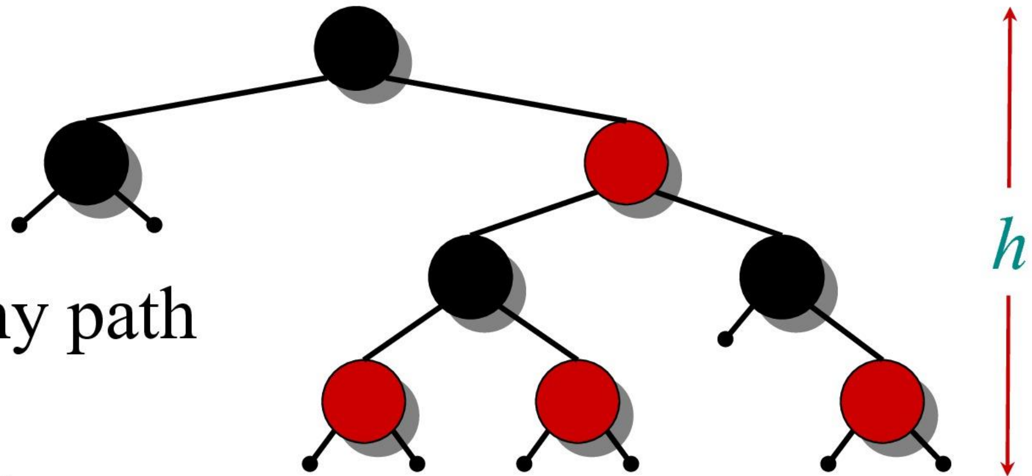


$h' <= \lg(n+1)$

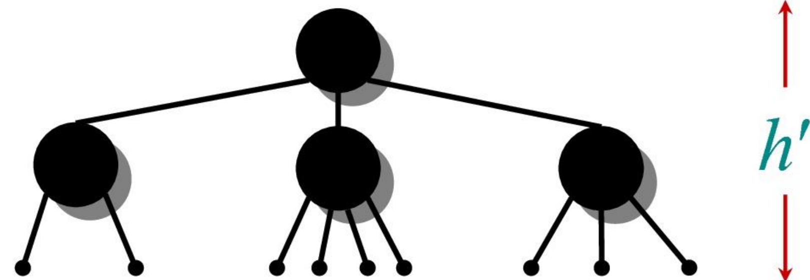
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.



Proof (continued) - Answer

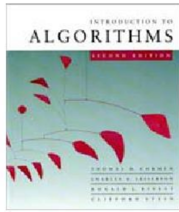


- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n + 1$
  $\Rightarrow n + 1 \geq 2^{h'}$
  $\Rightarrow \lg(n + 1) \geq h' \geq h/2$
  $\Rightarrow h \leq 2 \lg(n + 1)$. ∎



- $h' \leq \log_2 (n + 1)$

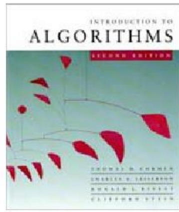
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Query operations

Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

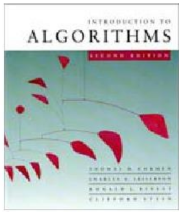
- the operation itself,
- color changes,
- restructuring the links of the tree via ***"rotations"***.

Do not forget to keep the properties
After do any operations --------------->

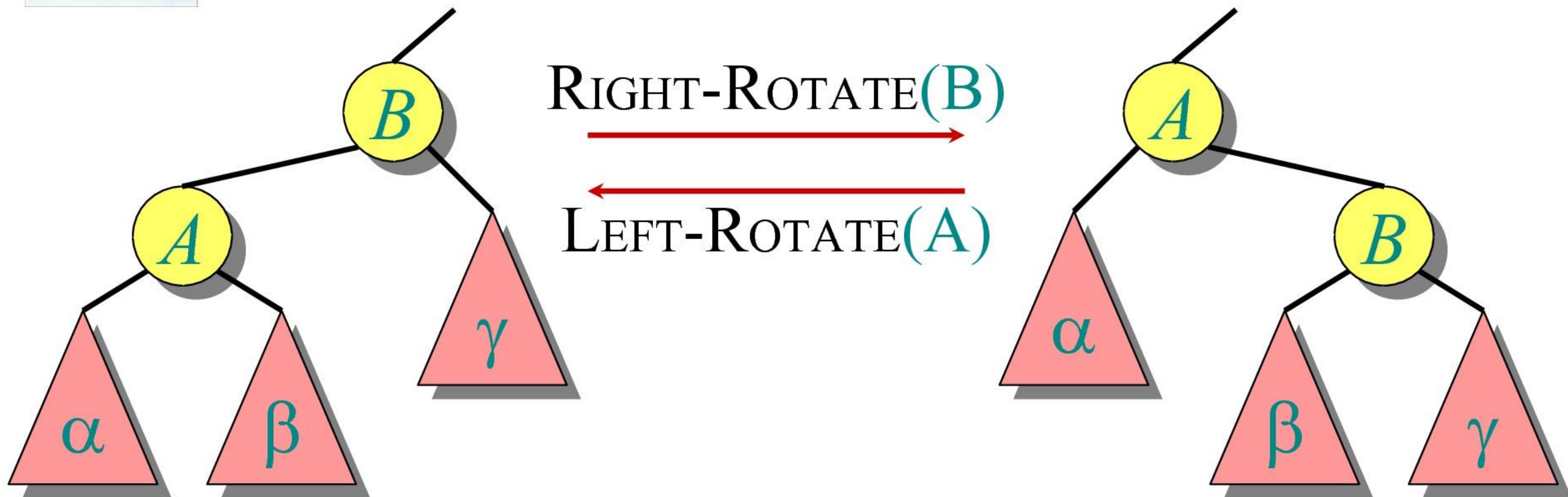
Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Rotations

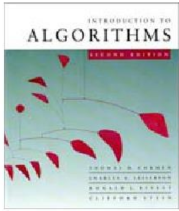


Rotations maintain the inorder ordering of keys:

- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \le A \le b \le B \le c$.

A rotation can be performed in $O(1)$ time.

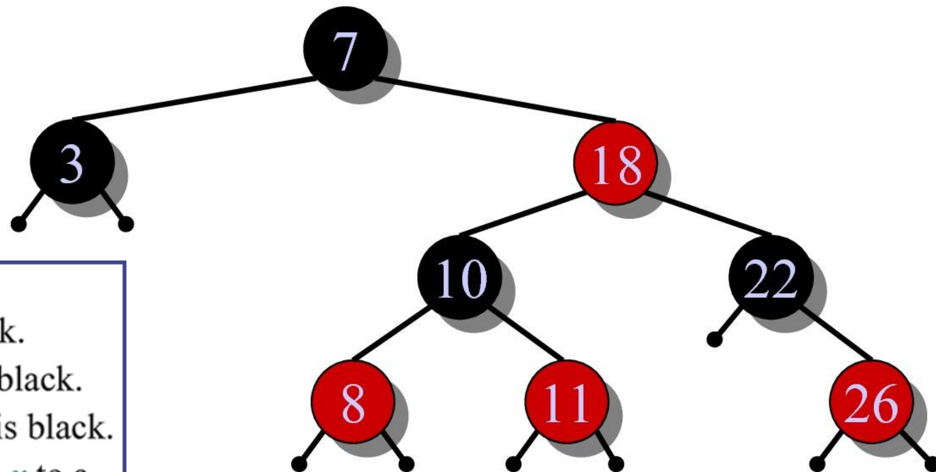
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Insertion into a red-black tree

(always insert a red node)

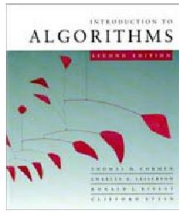
IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.



Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black. ← violation
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

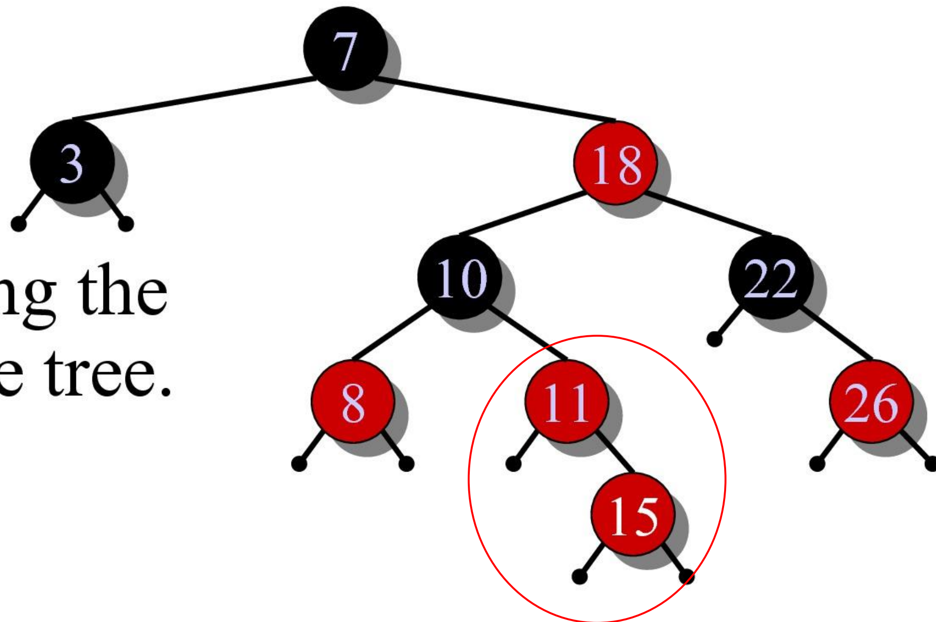


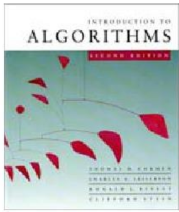
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.



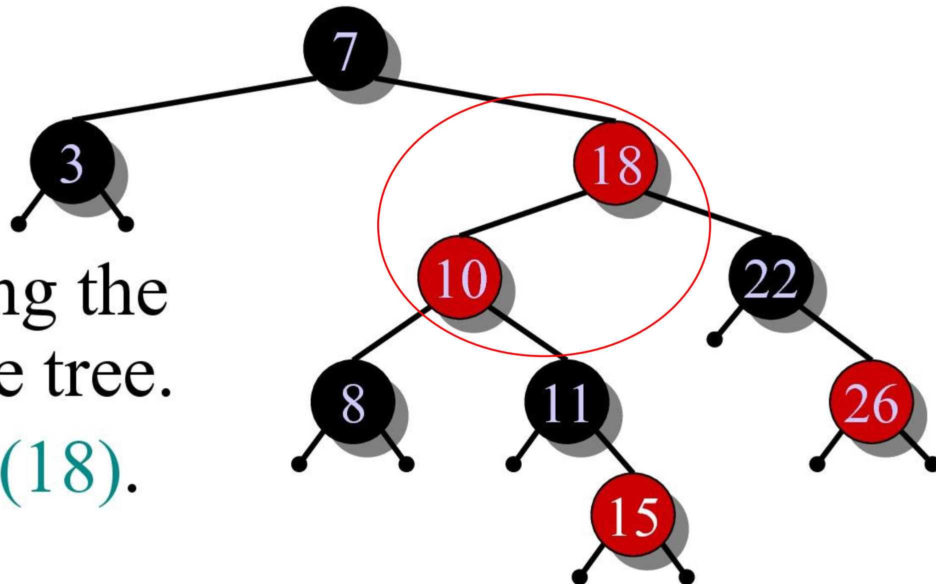


Insertion into a red-black tree

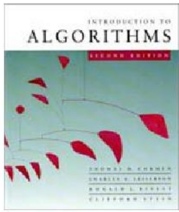
IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

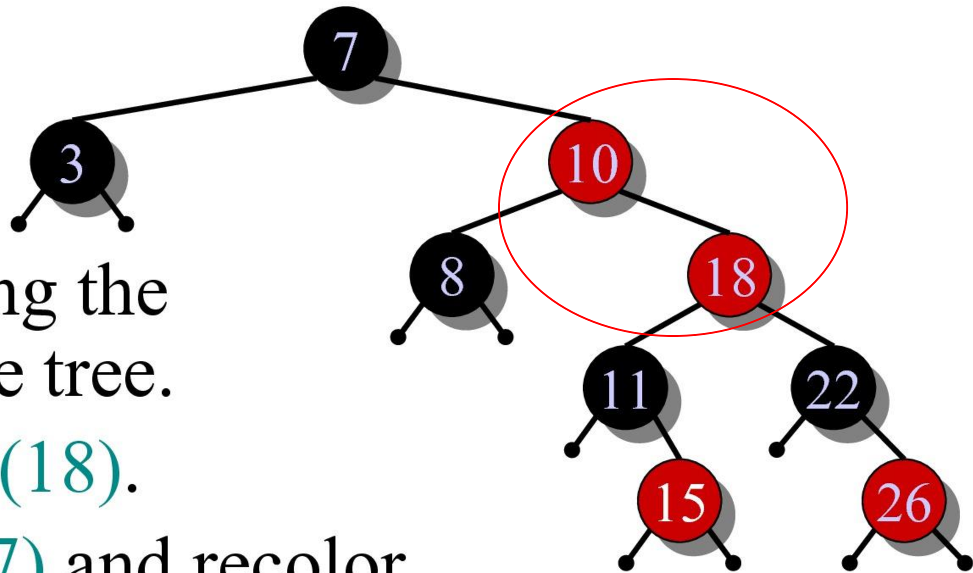


Insertion into a red-black tree

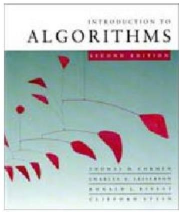
IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

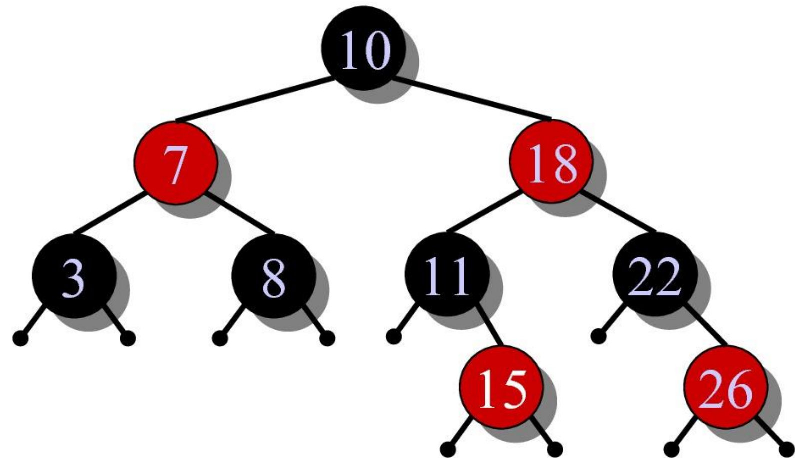


Insertion into a red-black tree

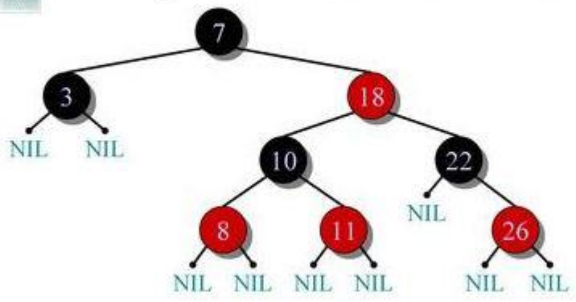
IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

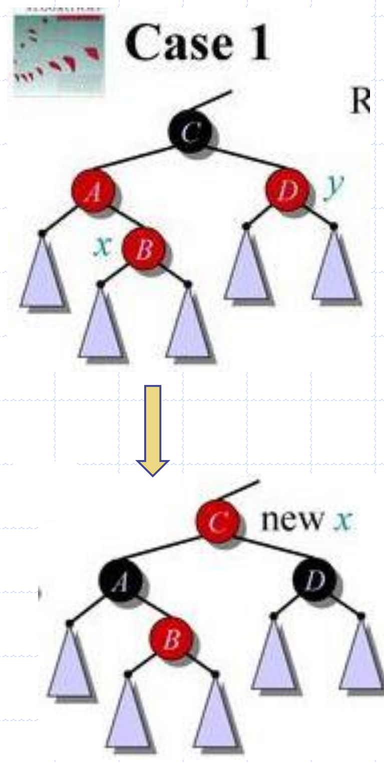
Example:

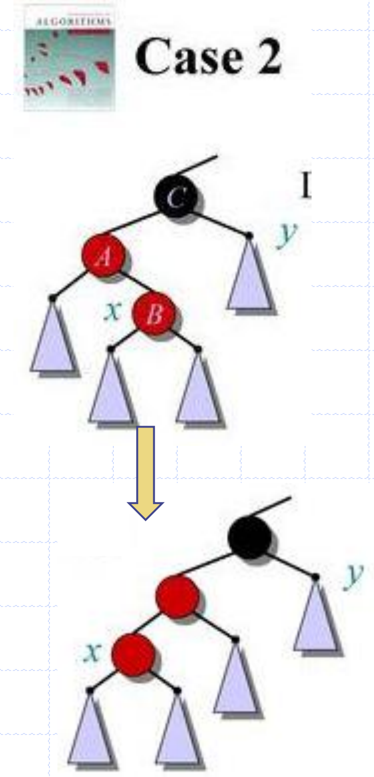
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

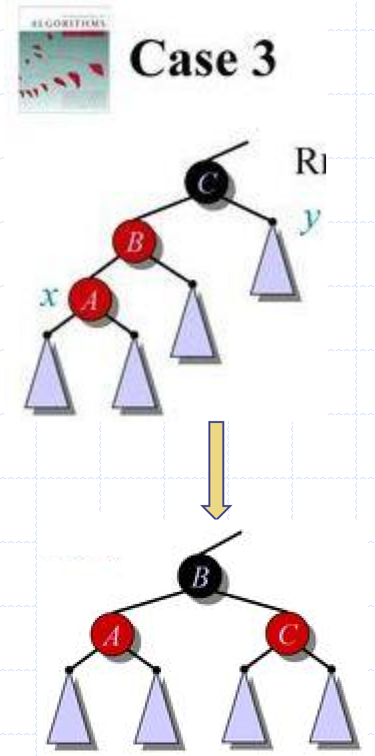


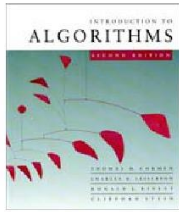
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson


7
3
18
NIL
NIL
10
22
8
11
NIL
26
NIL NIL NIL NIL NIL NIL


Case 1
R
C
A
D
y
x
B
C
new x
A
D
B


Case 2
C
y
A
x
B
y
x


Case 3
C
y
B
x
A
B
A
C

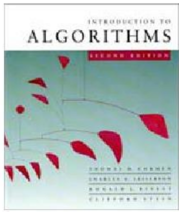


Graphical notation

Let △ denote a subtree with a black root.

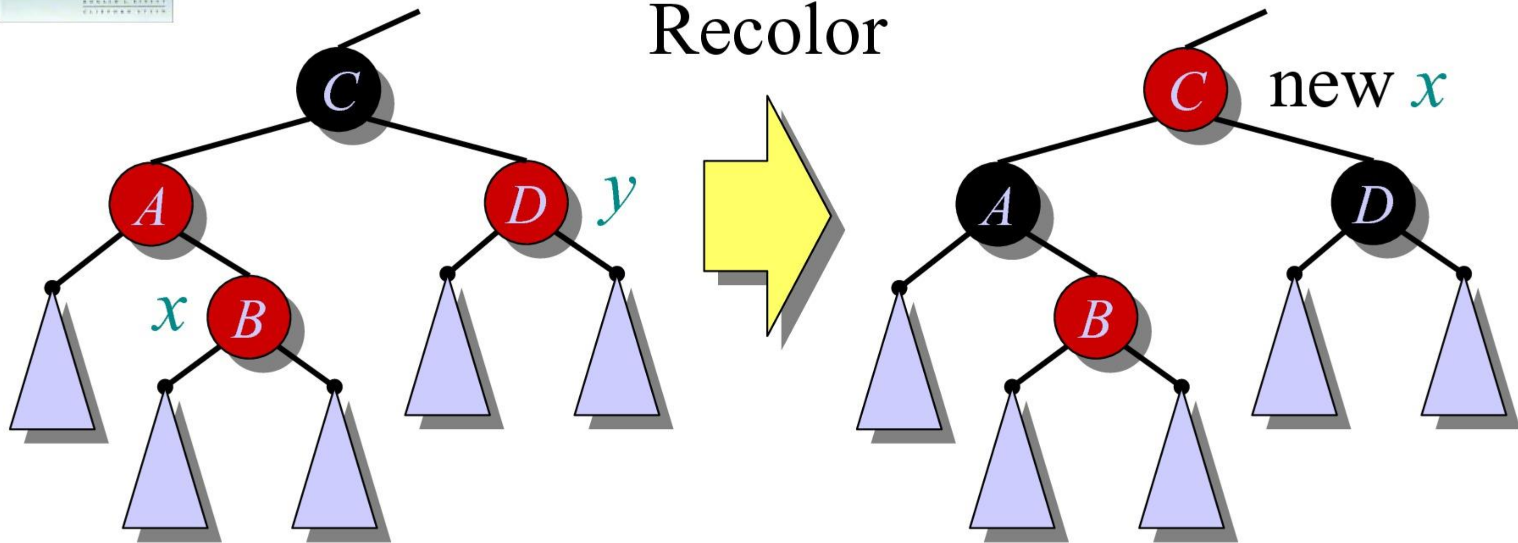
All △'s have the same black-height.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Case 1

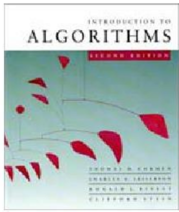
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html



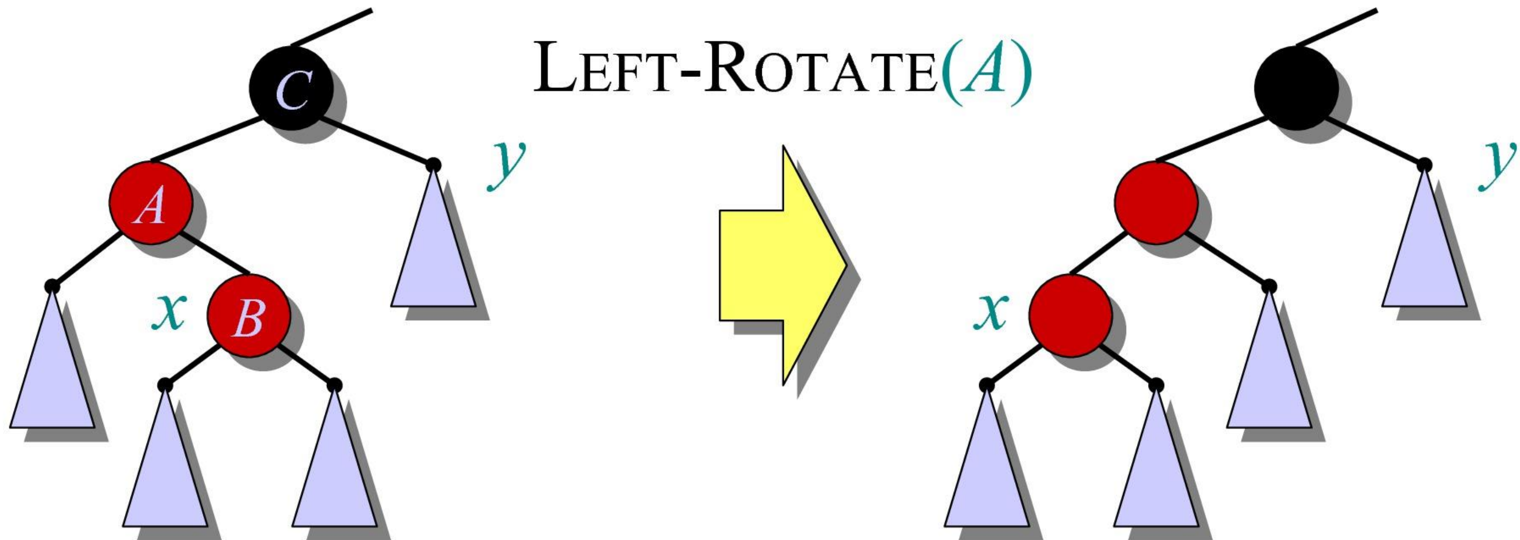
(Or, children of *A* are swapped.)

Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

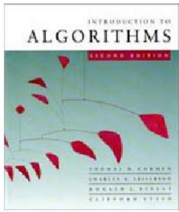


Case 2 https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

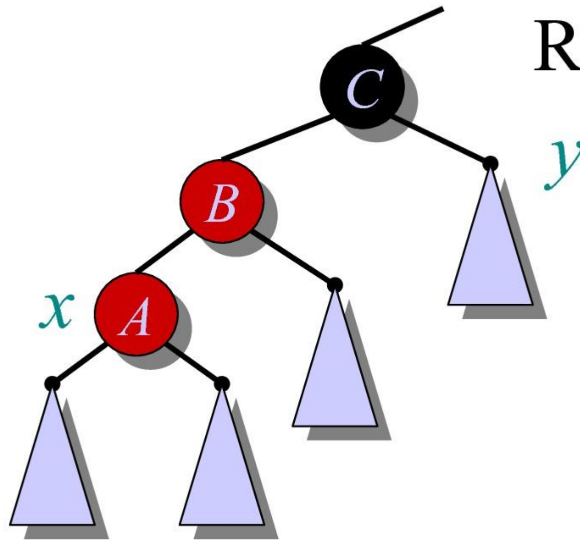


Transform to Case 3.

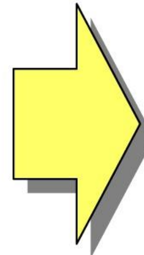
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

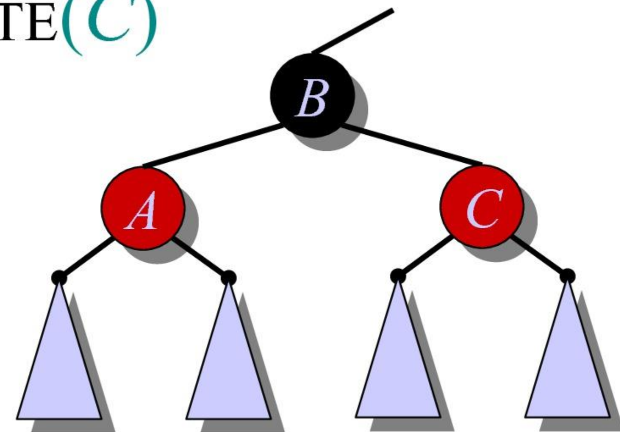


Case 3



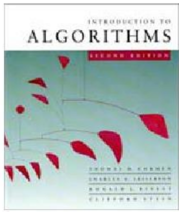
RIGHT-ROTATE(C)





Done! No more violations of RB property 3 are possible.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

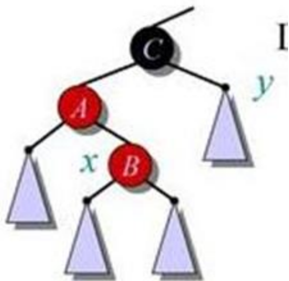


Pseudocode

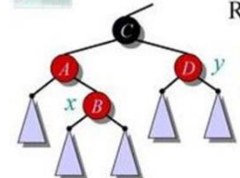
```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED      ▷ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]
            then y ← right[p[p[x]]          ▷ y = aunt/uncle of x
                 if color[y] = RED
                  then ⟨Case 1⟩
                  else  if x = right[p[x]]
                          then ⟨Case 2⟩    ▷ Case 2 falls into Case 3
                        ⟨Case 3⟩
            else ⟨"then" clause with "left" and "right" swappe
    color[root[T]] ← BLACK
```

If its parent's node is black, no problem. Otherwise, do the following (3 cases)

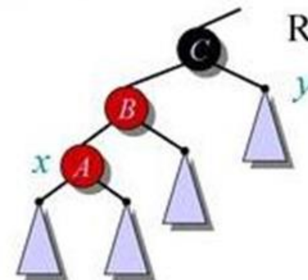
Case 2



Case 1



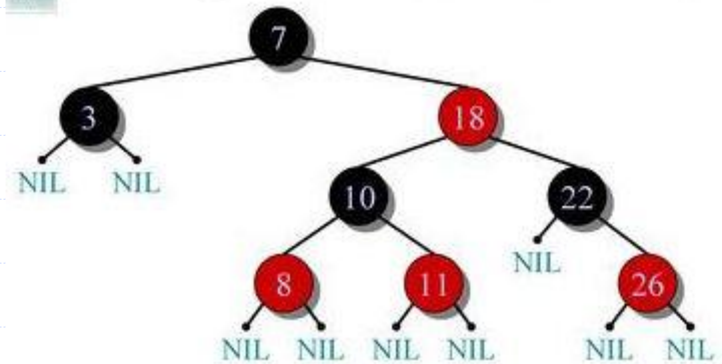
Case 3



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

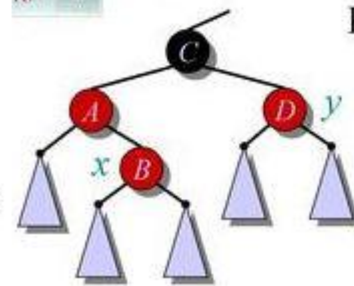
Red-black properties:

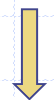
1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

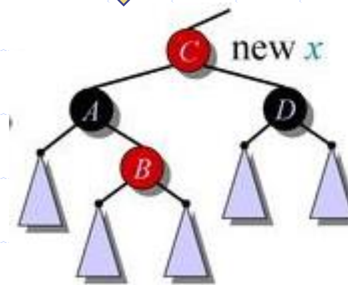



```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED      ▷ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]       ▷ y = aunt/uncle of x
                if color[y] = RED
                 then ⟨Case 1⟩
                 else  if x = right[p[x]]
                          then ⟨Case 2⟩   ▷ Case 2 falls into Case 3
                        ⟨Case 3⟩
            else ⟨"then" clause with "left" and "right" swapped⟩
    color[root[T]] ← BLACK
```

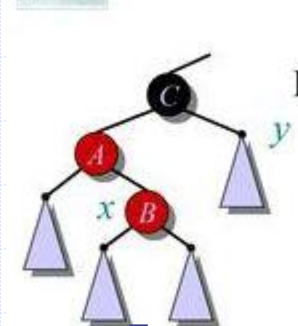
Case 1

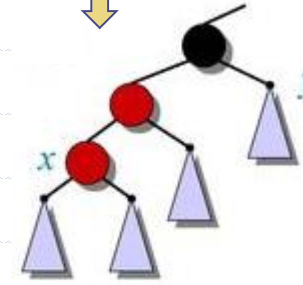




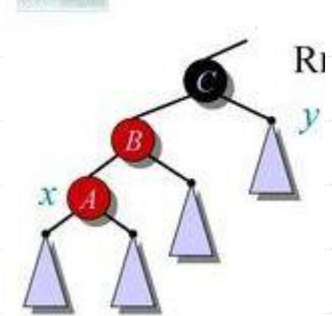


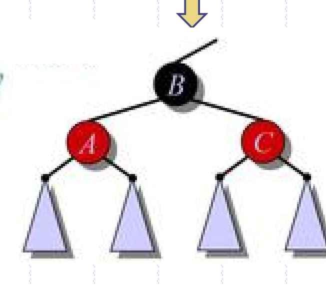
Case 2

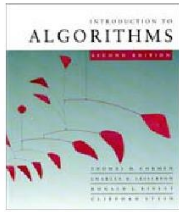




Case 3







Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running time: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Insertion program source code in Python

```
class TreeNode:
    def __init__(self,val,left = None,right = None, parent = None,color = None):
        self.val = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
        self.color = color

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return (self.parent and (self.parent.leftChild == self))

    def isRightChild(self):
        return (self.parent and (self.parent.rightChild == self))
```

```python
class BST(TreeNode):
    def __init__(self):
        self.root = None
        self.size = 0
        self.nil = TreeNode(None)
        self.nil.color = "black"

    def addNode(self,val):
        self.size += 1
        y = self.nil
        x = self.root
        if self.root == None:
            self.root = TreeNode(val,self.nil,self.nil,self.nil,"black")
        else:
            z = TreeNode(val,self.nil,self.nil, None, "red")
            while x!=self.nil:
                y = x
                if z.val < x.val:
                    x = x.hasLeftChild()
                else:
                    x = x.hasRightChild()
            z.parent = y
            if y == self.nil:
                self.root = z
            elif z.val < y.val:
                y.leftChild = z
            else:
                y.rightChild = z
            self.treeInsFixer(z)
```

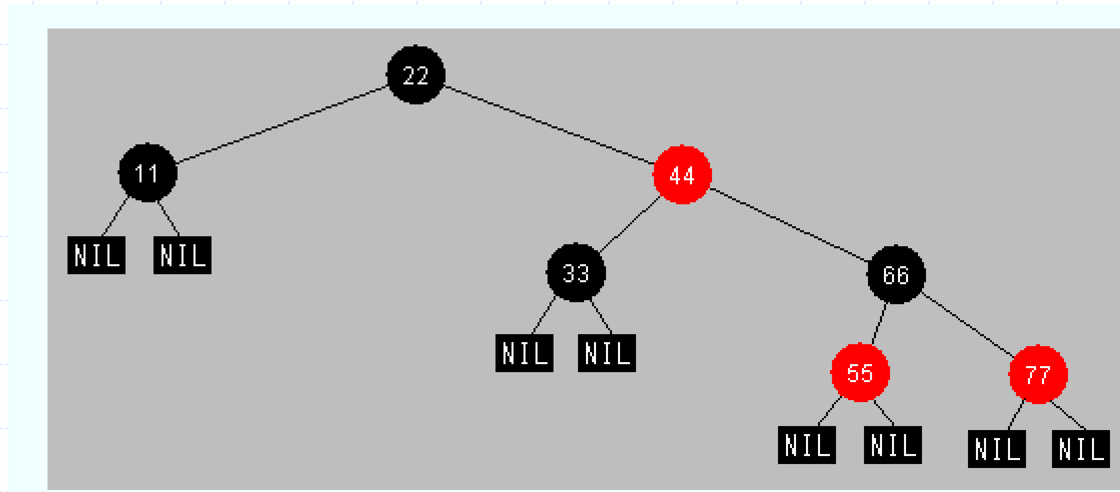
```python
def treeInsFixer(self,z):
    while z.parent.color == "red":
        if z.parent == z.parent.parent.leftChild:
            y = z.parent.parent.rightChild
            if y.color == "red":
                z.parent.color = "black"
                y.color = "black"
                z.parent.parent.color = "red"
                z = z.parent.parent
            else:
                if z == z.parent.rightChild:
                    z = z.parent
                    self.leftRotate(z)
                z.parent.color = "black"
                z.parent.parent.color = "red"
                self.rightRotate(z.parent.parent)
        elif z.parent == z.parent.parent.rightChild:
            y = z.parent.parent.leftChild
            if y.color == "red":
                z.parent.color = "black"
                y.color = "black"
                z.parent.parent.color = "red"
                z = z.parent.parent
            else:
                if z == z.parent.leftChild:
                    z = z.parent
                    self.rightRotate(z)
                z.parent.color = "black"
                z.parent.parent.color = "red"
                self.leftRotate(z.parent.parent)
    self.root.color = "black"
```

```python
def leftRotate(self,x):
    y = x.rightChild
    x.rightChild = y.leftChild
    if y.leftChild != self.nil:
        y.leftChild.parent = x
    y.parent = x.parent
    if x.parent == self.nil:
        self.root = y
    elif x == x.isLeftChild():
        x.parent.leftChild = y
    else:
        x.parent.rightChild = y
    y.leftChild = x
    x.parent = y

def rightRotate(self,x):
    y = x.leftChild
    x.leftChild = y.rightChild
    if y.rightChild != self.nil:
        y.rightChild.parent = x
    y.parent = x.parent
    if x.parent == self.nil:
        self.root = y
    elif x == x.isRightChild():
        x.parent.rightChild = y
    else:
        x.parent.leftChild = y
    y.rightChild = x
    x.parent = y

def inOrder(self,x):
    if(x!= self.nil):
        self.inOrder(x.leftChild)
        print(x.val, x.color)
        self.inOrder(x.rightChild)
```

```
a = BST()
a.addNode(5)
a.addNode(7)
a.addNode(4)
a.addNode(6)
a.addNode(11)
a.addNode(9)
a.addNode(17)
a.addNode(2)
a.addNode(14)
a.addNode(10)
a.addNode(8)
a.inOrder(a.root)
```

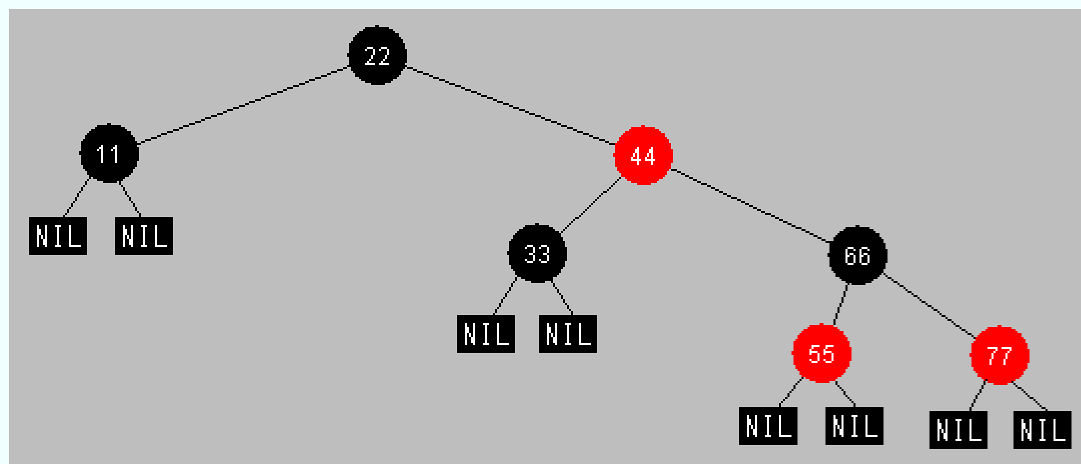
```
RESTART: C:/rhuang-2016-10-28/Algorithms/algorithm-2015/2017/lecture-notes/rb-tree-L7/red-black-insert.py
>>> a= BST()
>>> a.addNode(5)
>>> a.addNode(7)
>>> a.addNode(4)
>>> a.inOrder(a.root)
4 red
5 black
7 red
>>> a.addNode(6)
>>> a.addNode(11)
>>> a.inOrder(a.root)
4 black
5 black
6 red
7 black
11 red
>>> a.addNode(17)
>>> a.inOrder(a.root)
4 black
5 black
6 black
7 red
9 red
11 black
17 red
```

An example



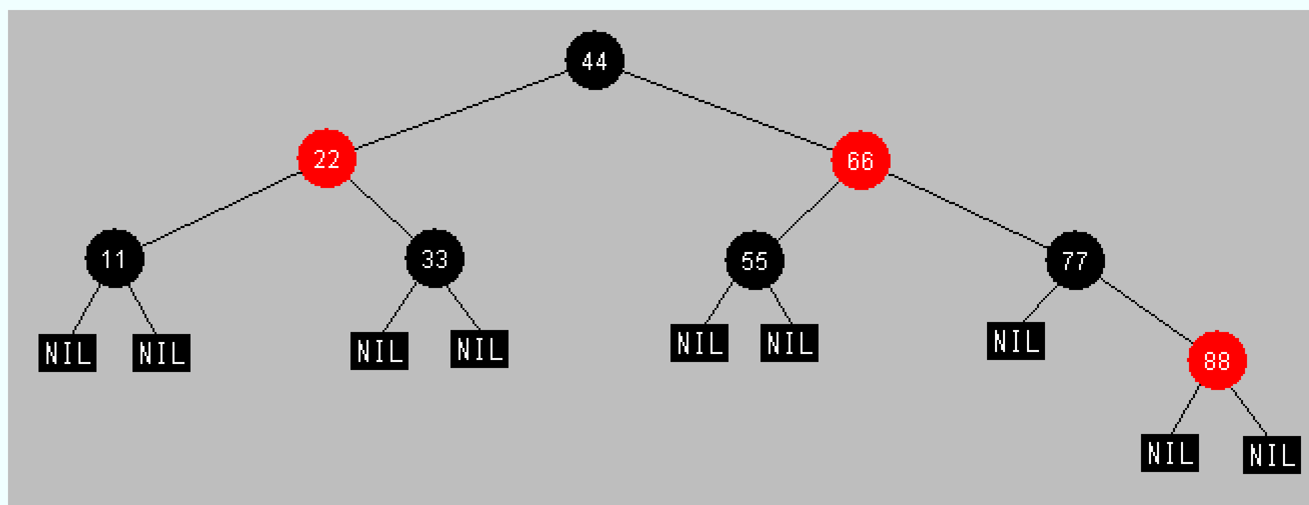
上図に 88 を挿入すると、

Red-Black Tree の例



Red/Black Tree Demonstration で作図後 NIL を加筆

上図に 88 を挿入すると下図のようになります。



Work in class

Please add the following nodes to red-black tree in order

20, 5, 40, 10, 25, 2, 35, 15, 13, 30, 33

Red-black properties:

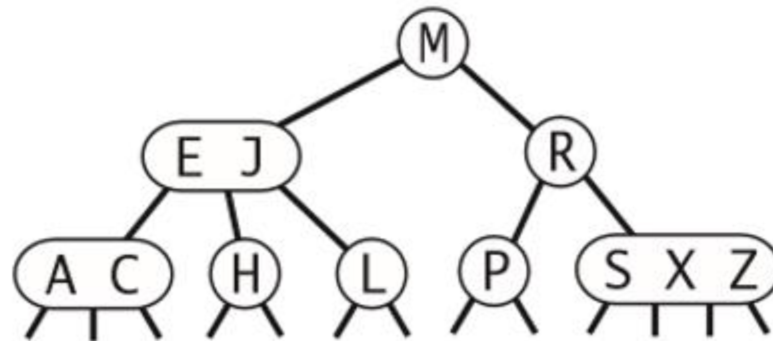
1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Demo

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

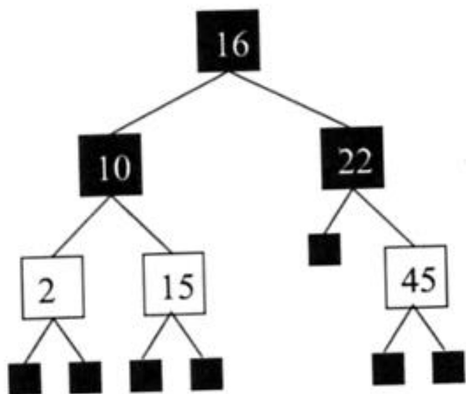
Exercise 7-1

- Please try to transfer the following 2-3-4 tree to a red-black tree

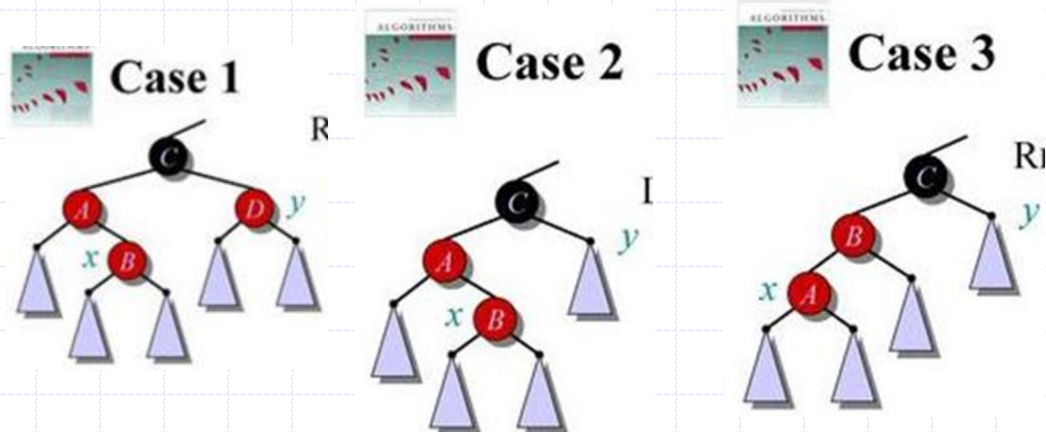


Exercise 7-2

Consider the following sequence of keys: (18, 30, 25, 12, 14). Insert the items with this set of keys in the order given into the red-black tree below. Draw the tree after each insertion.



キー配列について考える: (18, 30, 25, 12, 14)。
このキーのセットを図の赤黒木に挿入しなさい。
それぞれの挿入後の赤黒木を描きなさい。



This is another web site for your reference

http://fujimura2.fiw-web.net/java/mutter/tree/red-black-tree.html

Please read through
Its implementation is in Java