Balanced tree

Operations on a binary search tree (most of them)

take time directly proportional to the tree's height

→ it is desirable to keep the height small

Balanced tree : no leaf is much farther away from the root than any other leaf.

Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Self-balancing binary search tree :

- a binary search tree
- & keep it balanced

Popular balanced tree

- red-black tree
- AVL tree

Height-balanced tree

Height-balanced tree :

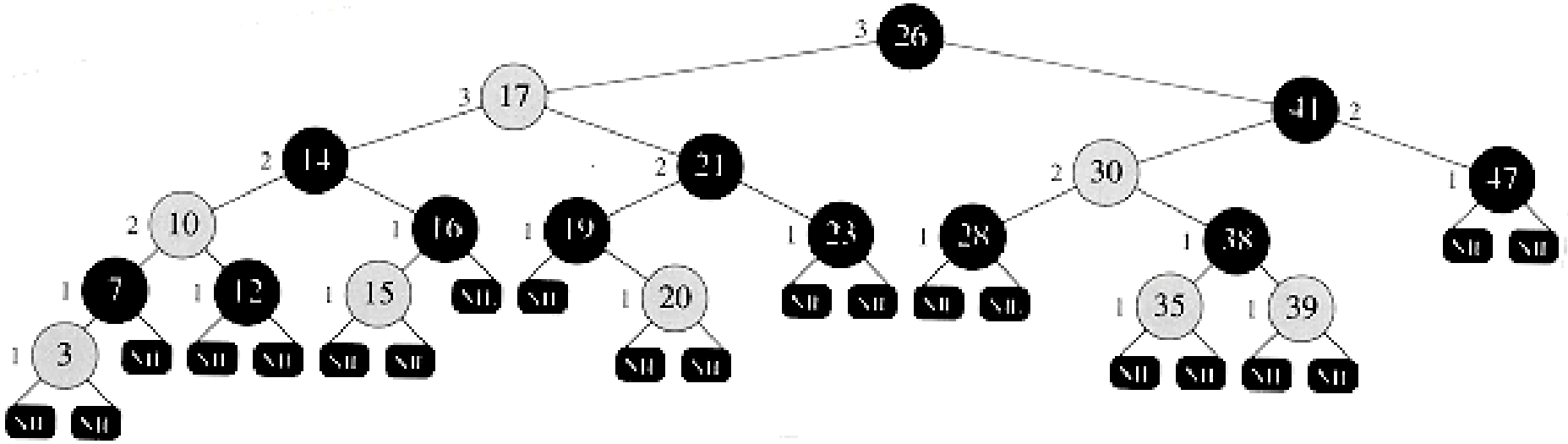
A tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

An empty tree is height-balanced.

Height-balanced tree

- AVL tree

Red-black tree



Cormen

Red-Black tree

A red-black tree is a binary search tree which satisfies:

1. Every node is either red or black.
2. Every leaf (NIL) is black.
3. If a node is red, then both its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

- one extra information per node:

  its ***color***, which can be either RED or BLACK.

Red-Black tree

- **black-height** of a node x: bh(x)

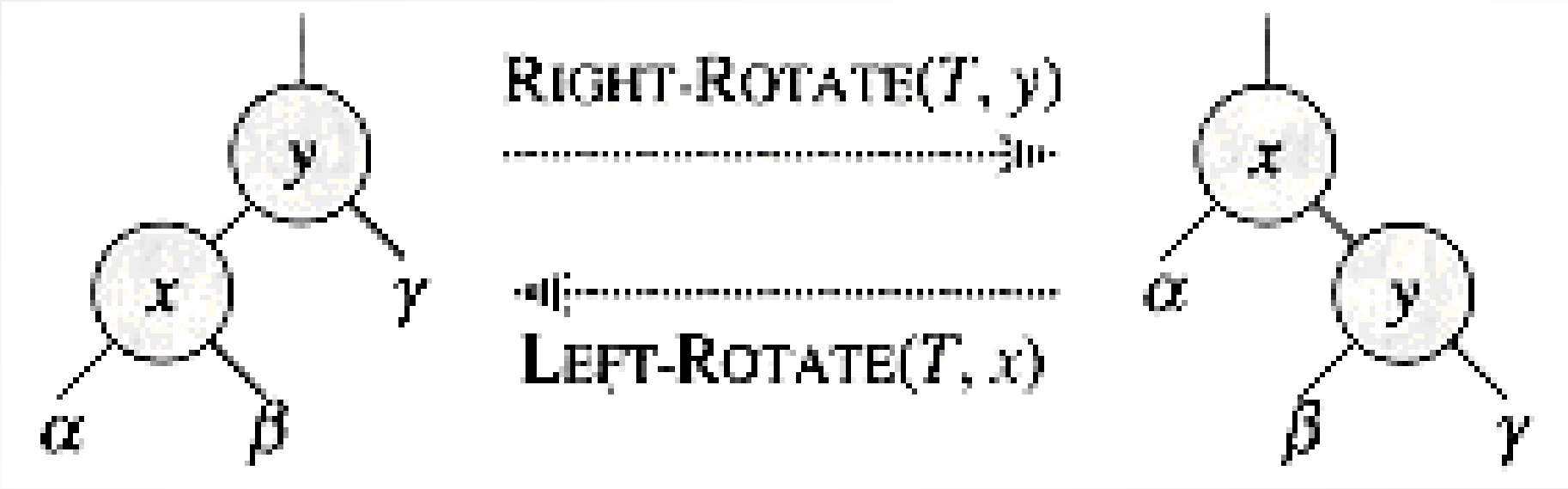
  the number of black nodes on any path from x to a leaf node
- **black-height of a red-black tree**: the black-height of its root.

Lemma

A red-black tree with n internal nodes

has height at most $2*\log_2(n + 1)$.

Rotation



DS

TColor = (red, black)

TreeNode:

info: TCE

left: ^TreeNode

right: ^TreeNode

parent:^TreeNode

color: TColor

end

Red-black tree: operation insert

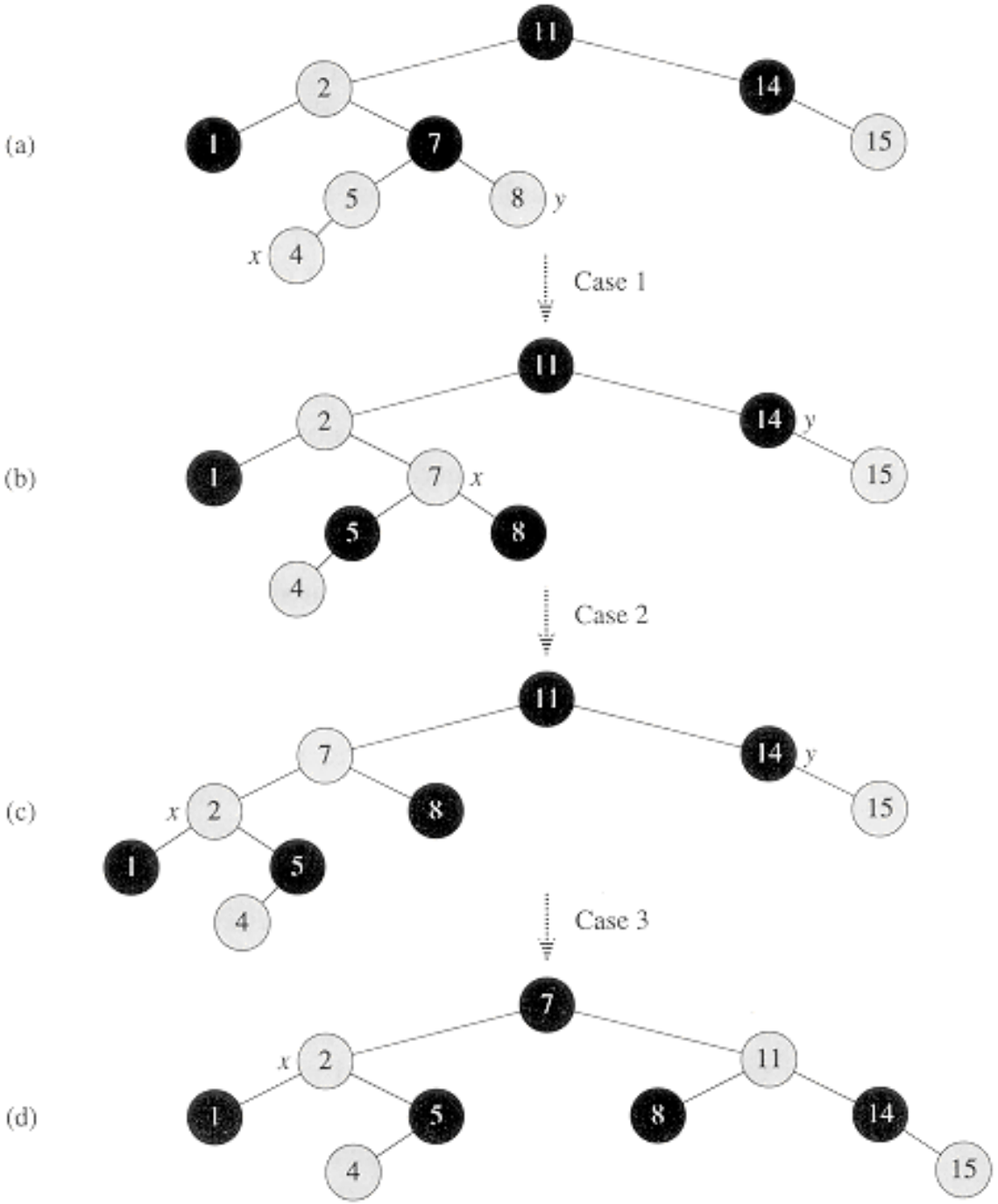
- insert in BSTree

  new node ***x***

  x is red
- if the parent of ***x*** is red

  fix the tree !!

Red-black tree: operation insert



Cormen

RBT_insert(T,e)

```
x:=BST-insert(T,e)
[x].color := red
while x<>rootPos(T) and Color(x^.parent) = red do
    // …
    if x^.parent = x^.parent^.parent^.left then
        y:= x^.parent^.parent^.right
        if Color(y)=red then
                Color(x^.parent) :=black
                Color(y):=black
                x:= x^.parent^.parent
                Color(x) := red
        else
```

Case 1

```
            if x = x^.parent^.right then
                  x:=x^.parent
                  LeftRotate(T,x)
            endif
            Color(x^.parent) :=black
            Color(x^.parent^.parent) :=red
            RightRotate(T, x^.parent^.parent)
      endif
   else
   …
   endif
endwhile
// …
```

- **Case 2**

Case 3

Red-black tree: operation delete

Delete as in BSTree

- A node to be deleted will have at most one child

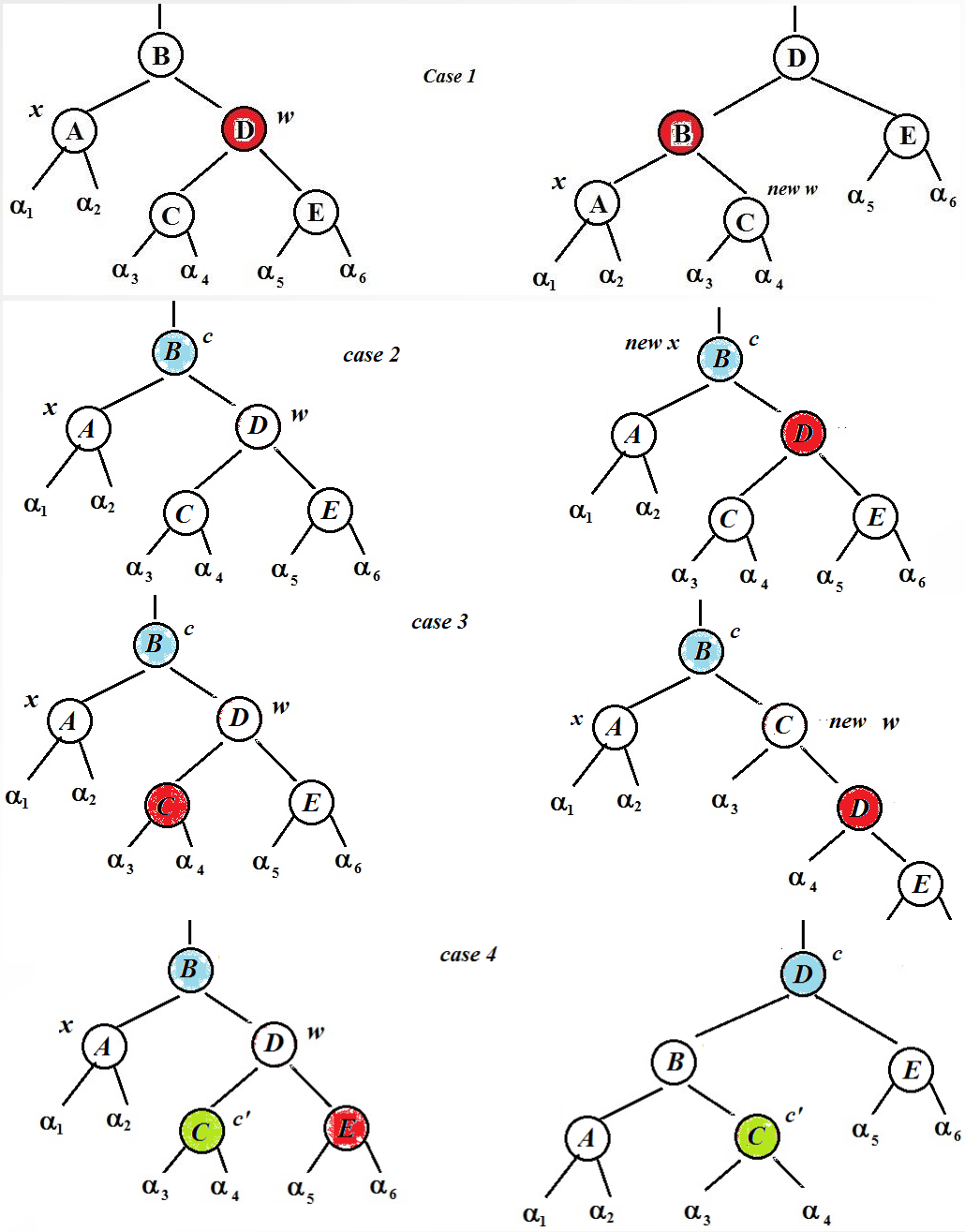
If a discrepancy arises for the red-black tree, fix it !

- If the deleted node is red

  the tree is still a red-black tree

- If the deleted node is black:
  - if its child is red, repaint the child to black.
  - otherwise: fix the tree !!

    mark the child as **double black : x** **(and fix the problem !)**


Case 1
B
x
A
D
w
α1
α2
C
E
α3
α4
α5
α6
D
B
E
x
A
new w
C
α5
α6
α1
α2
α3
α4
case 2
B
c
x
A
D
w
C
E
new x
B
c
A
D
C
E
case 3
B
c
x
A
D
w
C
E
B
c
x
A
C
new w
D
E
case 4
B
x
A
D
w
C
c′
E
D
c
B
E
A
C
c′

```
while x<> rootPos(T) and Color(x) = black do
    if x=x^.parent^.left then
        w:=x^.parent^.right
        if Color(w) =red then
                Color(w) :=black
                Color(x^.parent) :=red
                LeftRotate(T, x^.parent)
                w:= x^.parent^.right
        endif
          if Color(w^.left) =black and Color(w^.right) =black
          then
                Color(w) :=red
                x:= x^.parent
          else
```

Case 1

Case 2

```
        if Color(w^.right) =black then
                Color(w^.left) :=black
                Color(w) :=red
                RightRotate(T,w)
                w:= x^.parent^.right
        endif
        Color(w) := Color(x^.parent)
        Color(x^.parent) :=black
        Color(w^.right) :=black
        LeftRotate(T, x^.parent)
        x:=root(T)
      endif
    else
    …
    endif
endwhile
Color(x) :=black
```

Case 3

Case 4