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 :

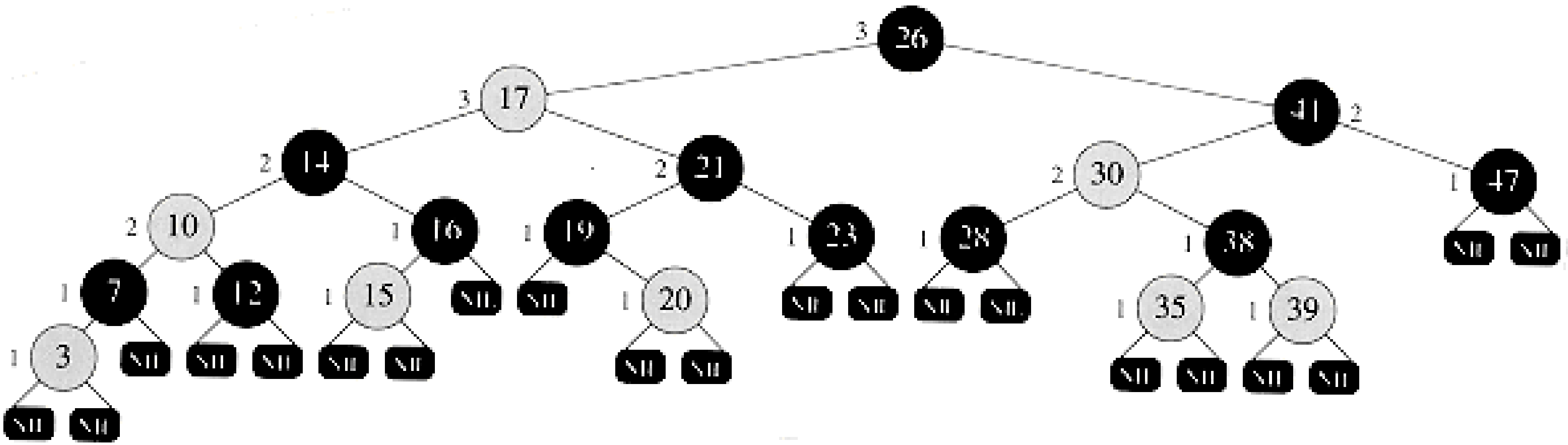
A tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

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*)

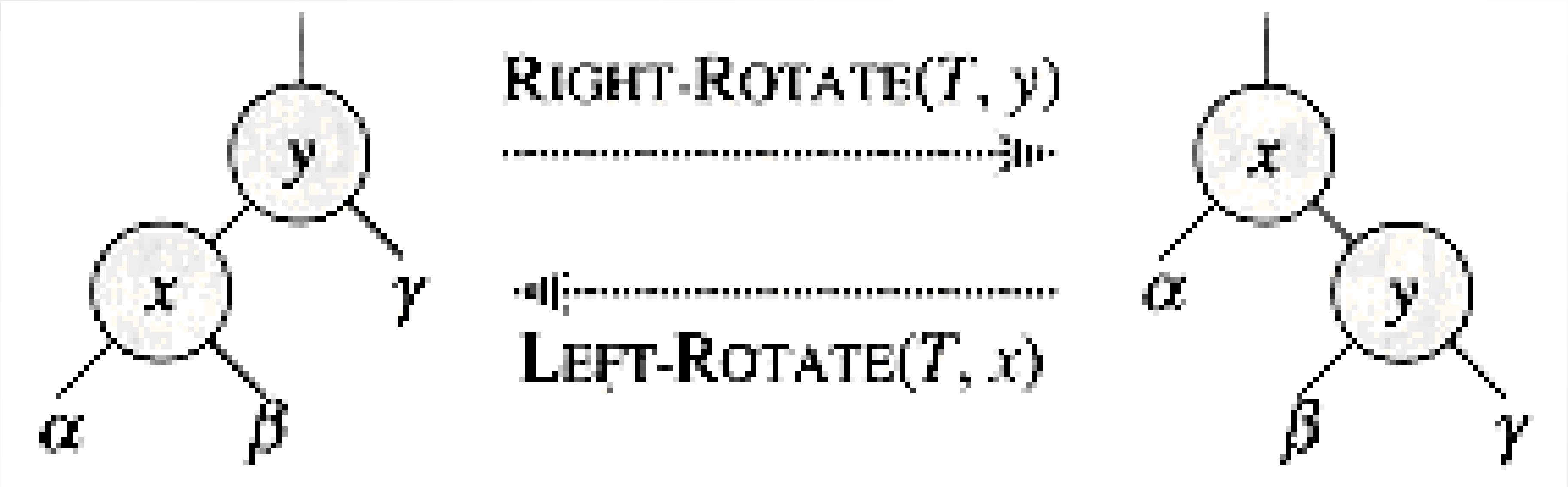
  the number of black nodes on any path from *x* to a leaf node
- **black-height of a red-black tree**: the black-height of its root.

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

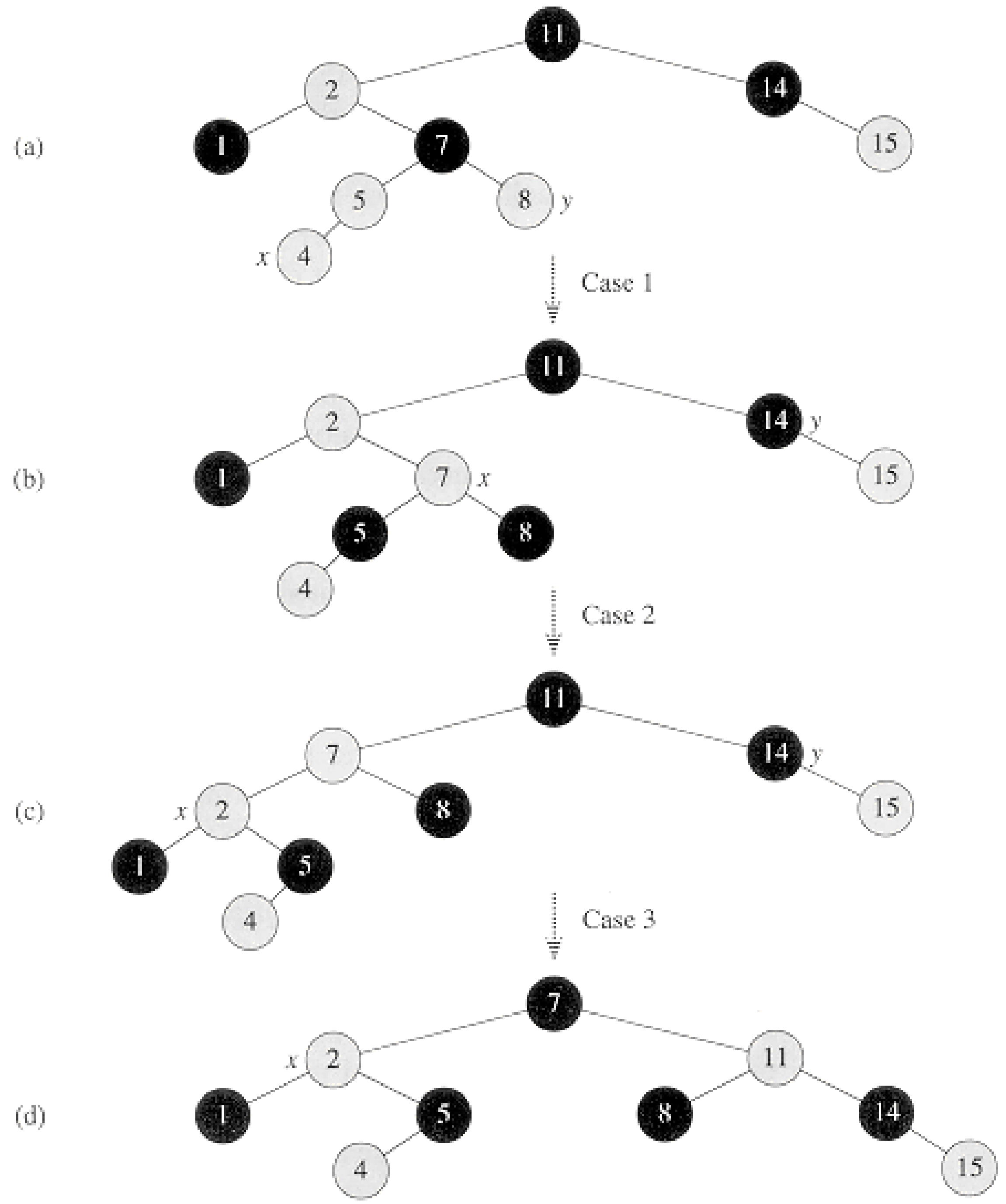
- insert in BSTree

  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

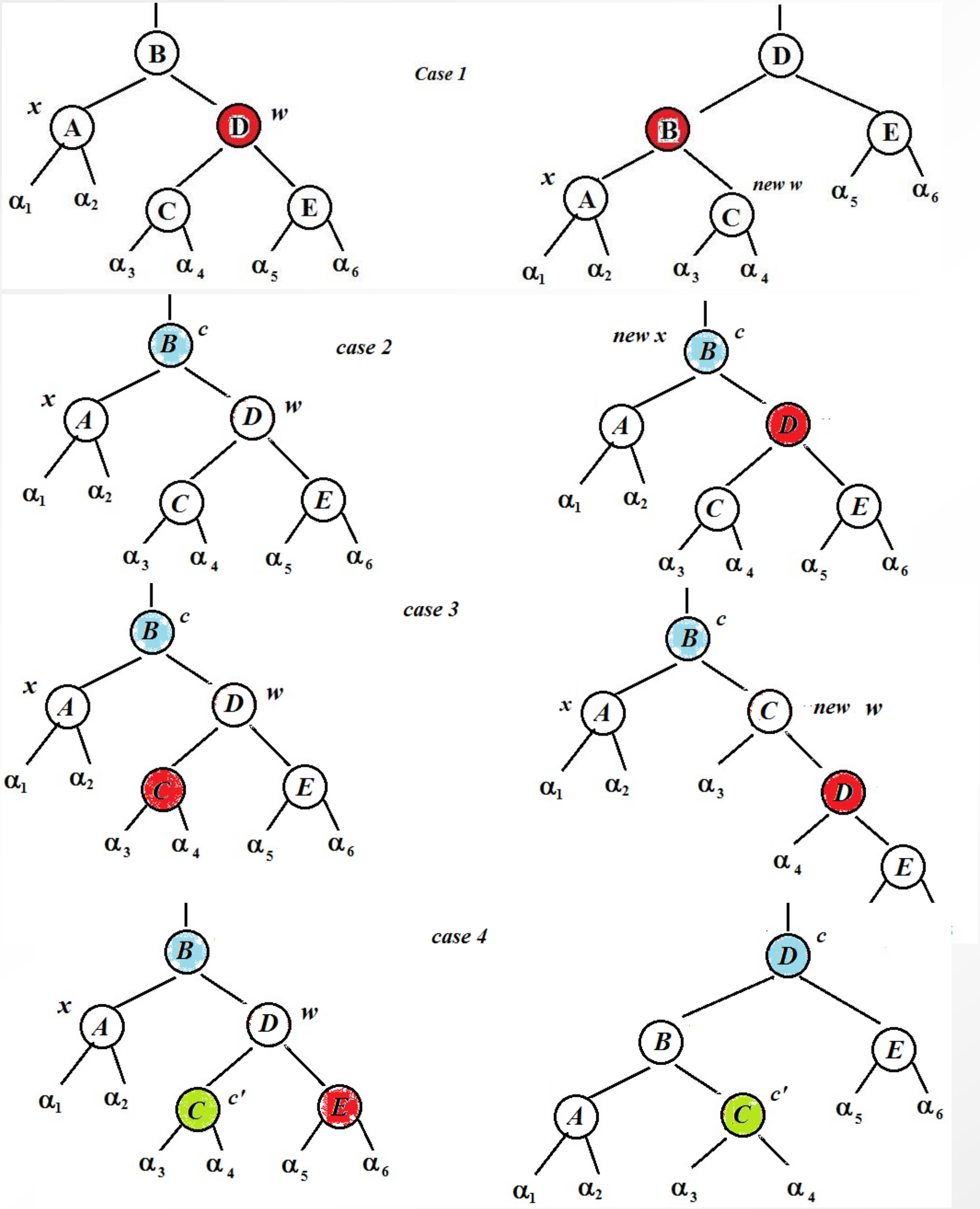
If a discrepancy arises for the red-black tree, fix it !

- 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
x
w
new w
case 2
new x
c
case 3
new w
case 4
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                                Case 1
                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                                  Case 2
                x:= x^.parent
          else
```

```
        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

AVL

Definition
An AVL tree is a binary search tree which satisfies:
the heights of the two child sub trees of any node differ by at most one

Remark:
Representation stores the balance factor or the height of the node

Operations over AVL

- search, insert and delete
  all take O(log n) time in average and worst cases
  where n is the number of nodes in the tree prior to the operation.

Consider the next representation:

```
AVLTreeNode  =       record
                          info: TComparable
                          left: ^ AVLTreeNode
                          right: ^ AVLTreeNode
                          h: Integer
                     end
```

Search

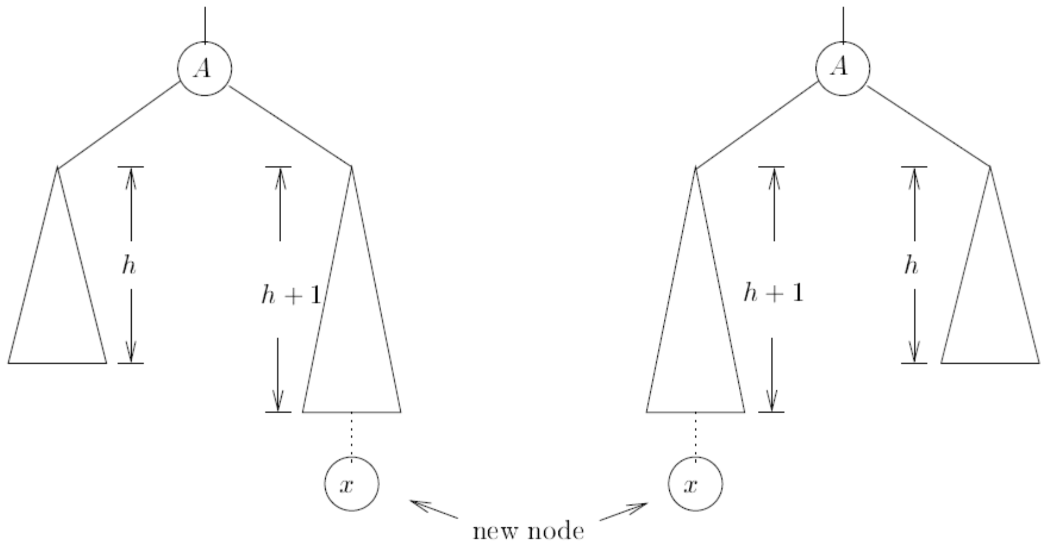
- BST search

Insert

Insertion:

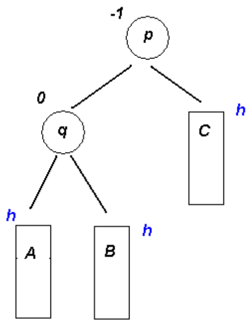
- require the tree to be rebalanced
  - insert an element like in BST case
  - rebalance the tree (if it is the case)
    consider all the ancestors (to the root)
    rebalance → one or more tree rotations.

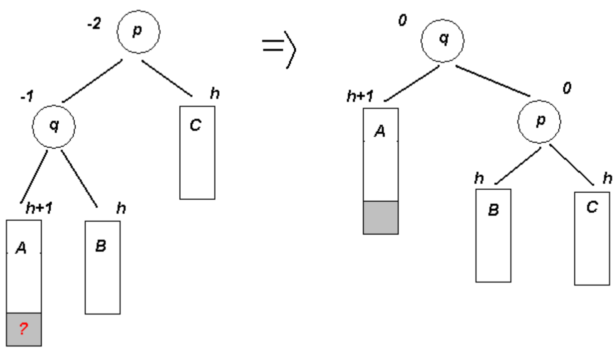
When to rebalance :

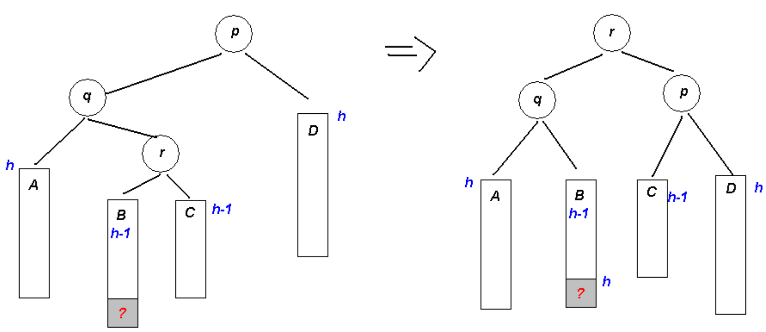


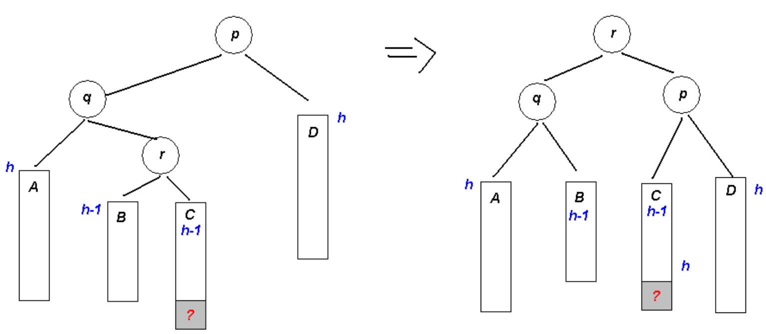
Insert cases - examples

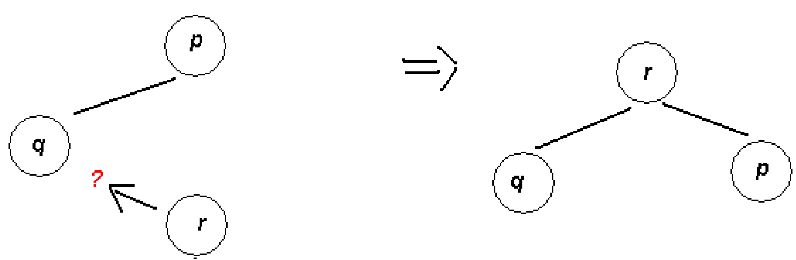
Assume: (next) initial situation & new node on the left subtree



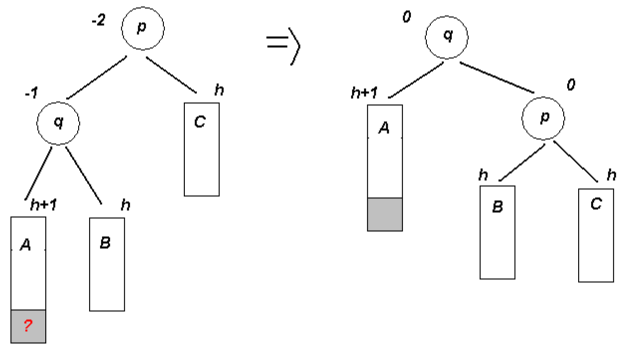








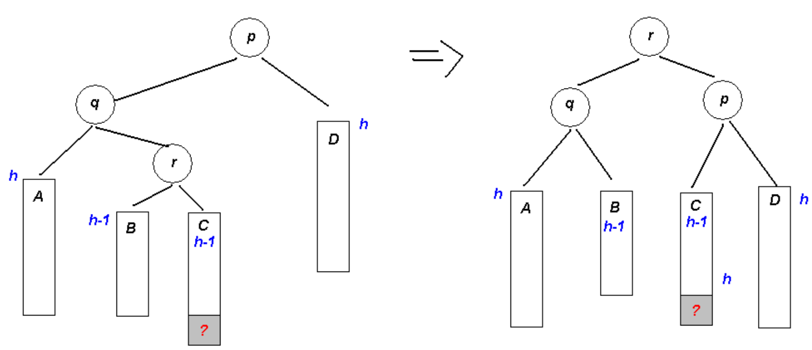
Rotations



/* ***Representation without link to parent*** */
/* Update heights, then return new root */

```
Function RotateRight ( p )
    q := p^.left
    p^.left := q^.right;
    q^.right := p;
    p^.h := Max( Height( p^.left ), Height( p^.right ) ) + 1;
    q^.h := Max( Height( q^.left ), Height ( q^.right ) ) + 1;
   RotateRight := q  /* New root */
end_RotateRight

Functin RotateLeft ( p )
    q := p^.right;
    p^.right = q^.left
    q^.left = p
    p^.h = Max( Height( p^.left ), Height( p^.right ) ) + 1
    q^.h = Max( Height( q^.left ), Height( q^.right ) ) + 1
   RotateLeft :=q
end_RotateLeft
```




```
Function DblRotateLeftRight ( p)
    p^.left := RotateLeft (p^.left )
    DblRotateLeftRight := RotateRight ( p );
end_DblRotateLeftRight
```

```
Function insert_rec(p , el)
// ElementType el, AvlTreeNode p
// return the new p
  if( p = NIL)
        p := new AvlTreeNode
        p^.info := el
        p^.h := 0;
        p^.left := NIL
        p^.right := NIL
else
        if( el < p^.info)  then
                p^.left := insert_rec(p^.left , el )
                if(Height(p^. right) - Height( p^. left ) = -2 )
                        if( el < p^.left^.info)
                                p := RotateRight ( p )
                        else
                                p := DblRotateLeftRight ( p )
                        endif
                endif
        else            // el >= [p].info
                p^.right = insert_rec(p^.right , el )
                if( Height( p^.right ) - Height( p^.left ) = 2 )
                        if( el > p^.right^.info ) then
                                p := RotateLeft ( p )
                        else
                                p := DblRotateRightLeft( p );
                        endif
                endif
        endif
        p^.h := Max( Height( p^.left ), Height( p^.right ) ) + 1;
endif
insert_rec := p
End_insert_rec

Subalg. insert(T , el)
        p := getRoot(T)
        np :=insert_rec(p, el)
        setRoot(T, np)
end_insert
```

Delete

- find the node x where k is stored
- delete the contents of node x ~similar with BST

  Deleting a node in an AVL tree can be reduced to deleting a leaf

  (next) alg. delete_rec – delete leaves

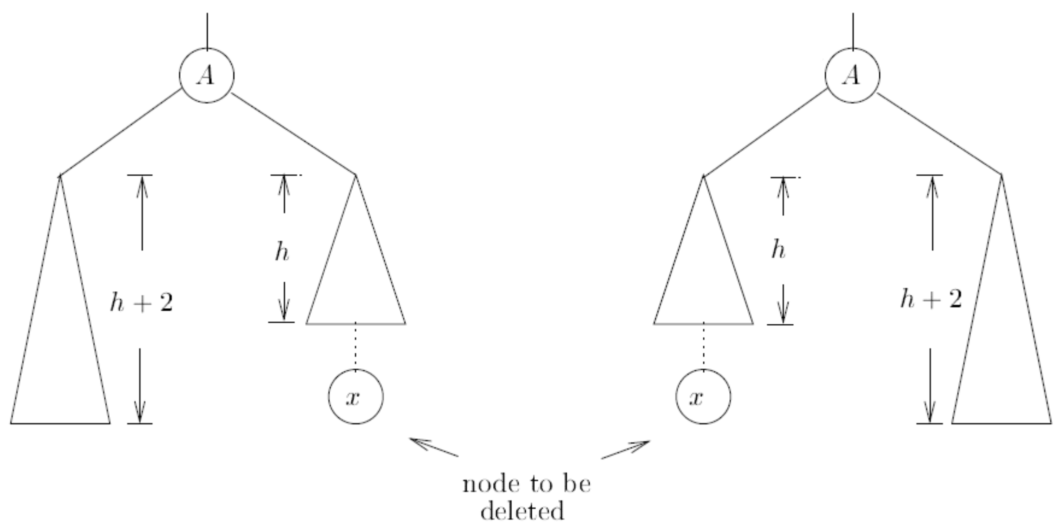
rebalance

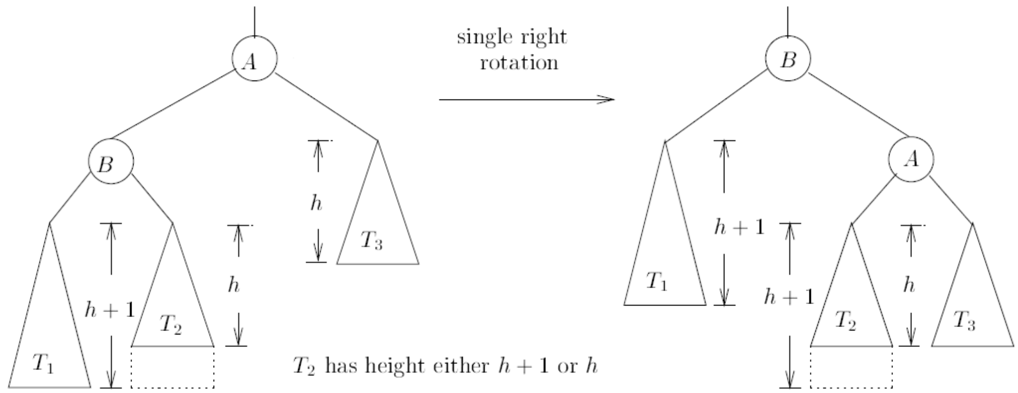
  go from the deleted leaf towards the root

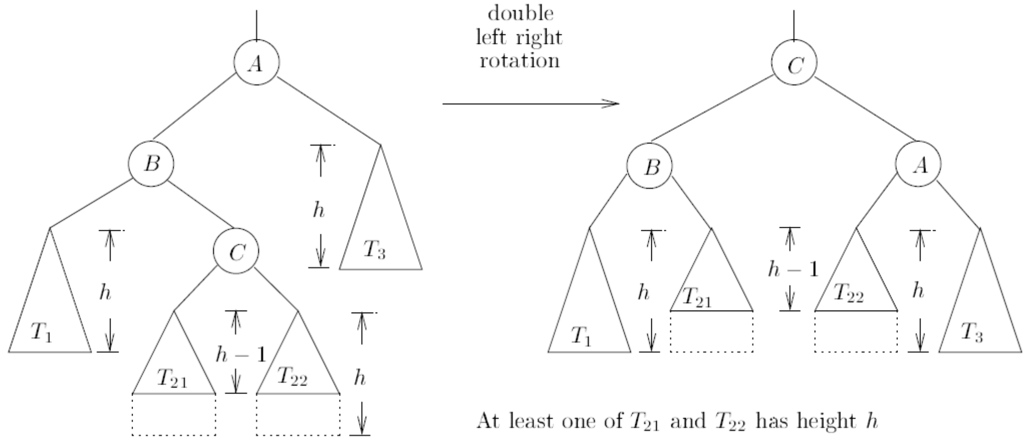
    -update the balance factor

    -rebalance with rotations if necessary.

Rebalance cases








```
Function delete_rec (p , el)
if p = NIL then return NIL endif
if el = p^.info then
        if p^.left=NIL and p^.right=NIL then
                delete p; p:=NIL
        else    q:=p
                if p^.left <> NIL       then    p:=p^.left
                                        else    p:=p^.right
                endif
                delete q
        endif
else
        if( el > p^.info) then
                p^.right = delete_rec ( p^.right , el );
                if( Height( p^.right ) - Height( p^.left ) = -2 )  then
                        if(Height( p^.left ^.left) = Height( p^.right ) +1) then
                                p = RotateRight ( p )
                        else // Height( [[p].left ].left = Height( [p].right )
                                p = DblRotateLeftRight(p)
                        endif
                endif
        else // el( el < [p].info)
                        //…
        endif
endif
if p<> NIL then p^.h = Max( Height( p^.left ), Height( p^.right ) ) + 1 endif
```

delete_rec := p
End_ delete_rec