

Arbres binaires de recherche
Opérations de base
TreeMap
Performances
Opérations de rebalancement
Opération de rotation
Opération de restructuration
AVL
Splay
RougeNoir
Introduction par les arbres (2,4)
Arbres vs tables de hachage

Arbre binaire de recherche

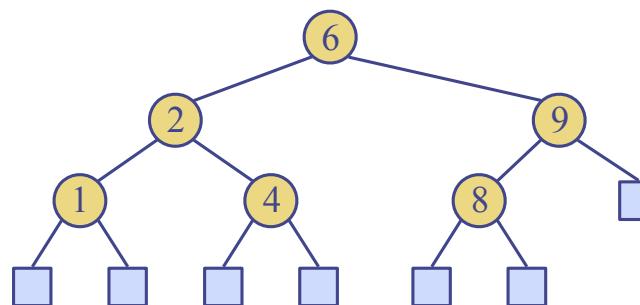
Un arbre binaire de recherche est un arbre binaire stockant des clés (ou des éléments clé-valeur) dans ses noeuds et satisfaisant la propriété suivante :

Soit u , v , et w , trois noeuds tels que u est dans le sous-arbre gauche de v et w est dans le sous-arbre droit de v . Nous avons que

$$\text{clé}(u) < \text{clé}(v) < \text{clé}(w)$$

Un parcours dans l'ordre (inorder) d'un arbre binaire de recherche visite les clés en ordre croissant.

Les carrés bleus représentent les enfants (références None) des noeuds externes.



Recherche

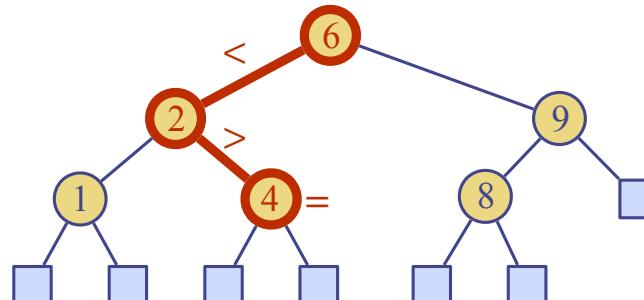
Pour **chercher** une clé k , nous suivons un chemin descendant commençant à la racine

Le prochain noeud visité dépend de la comparaison de k avec la clé du noeud visité

Si nous atteignons None, la clé n'est pas trouvée

Exemple: **chercher** 4 dans l'arbre binaire de recherche ci-dessous :

TreeSearch(T, T.racine(), 4)



```

Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)
    return p

```

succès

{appel récursif à gauche}

{appel récursif à droite}

{échec}

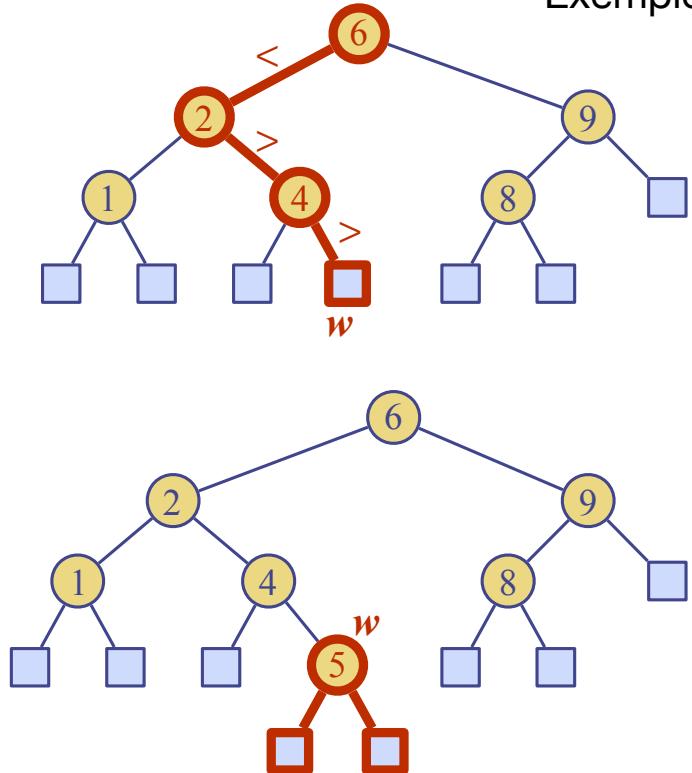
Insertion

Pour effectuer l'opération d'**insertion**(k, o), on cherche la clé k (en utilisant *TreeSearch*)

Supposons que k ne soit pas déjà dans l'arbre, et soit w la feuille où se termine la recherche

Nous insérons (k, o) au noeud w , qui devient un noeud interne

Exemple: **TreelInsert($T, 5, o$)**



Algorithm TreelInsert(T, k, v):

Input: A search key k to be associated with value v
 $p = \text{TreeSearch}(T, T.\text{root}(), k)$

if $k == p.\text{key}()$ **then**
 Set p 's value to v

else if $k < p.\text{key}()$ **then**
 add node with item (k, v) as left child of p

else
 add node with item (k, v) as right child of p

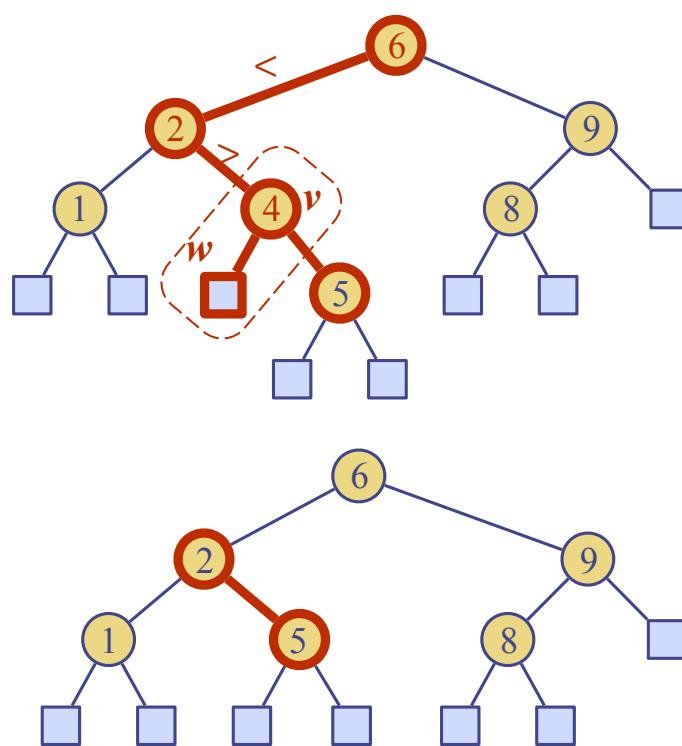
Suppression

Pour effectuer l'opération **supprimer(k)**, on cherche la clé k

Supposons que la clé k est dans l'arbre, et que v soit le nœud stockant k

Si le nœud v possède au plus un enfant, nous le supprimons avec l'opération **_delete(v)** (de la classe *LinkedBinaryTree* - voir IFT2015/3 Arbres diapositives #27 à #29), qui supprime v et le remplace par son enfant s'il en a un

Exemple: **supprimer 4**



```

# delete une Position si valide
# la remplace par son enfant s'il y en a un (mais pas 2!)
# retourne l'élément deleté
def _delete( self, p ):
    # validation de la Position
    node = self._validate( p )
    # doit avoir au plus 1 enfant
    if self.num_children( p ) == 2: raise ValueError( 'p has two children' )
    # on prend l'enfant existant ou None s'il n'y en a aucun
    child = node._left if node._left else node._right
    # s'il y a un enfant, il est adopté par son grand-parent
    # ou par personne s'il n'en a pas
    if child is not None:
        child._parent = node._parent
    # si la Position était la racine, la nouvelle racine
    # devient l'enfant
    if node is self._root:
        self._root = child
    # sinon, on remplace le noeud par son enfant
    # de gauche s'il était enfant gauche
    # de droite s'il était enfant droit
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    # on décrémente la taille
    self._size -= 1
    # on rend la Position invalide (en mettant parent sur lui même)
    node._parent = node
    # on retourne l'élément deleté
    return node._element
  
```

Suppression (cas d'un noeud interne avec 2 enfants)

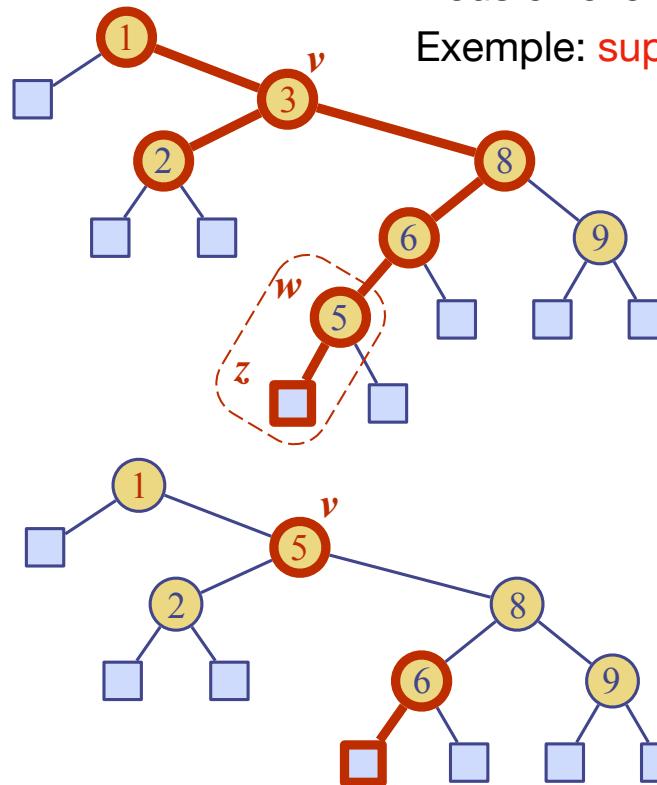
Nous considérons le cas où la clé k à supprimer est stockée sur un nœud v dont les enfants sont tous les deux internes

Nous trouvons le nœud interne w qui suit v dans un parcours en ordre (inorder)

Nous copions l'élément ($w, value$) dans le nœud v

Nous enlevons le noeud w au moyen de l'opération `_delete(w)`

Exemple: **supprimer 3**



TreeMap utilise un arbre binaire de recherche pour implémenter Map

```

from LinkedBinaryTree import LinkedBinaryTree
from Map import Map

class TreeMap( LinkedBinaryTree, Map ):

    #on ajoute des méthodes d'accès à une Position
    #pour la clé et la valeur
    class Position( LinkedBinaryTree.Position ):

        def key( self ):
            return self.element().__key

        def value( self ):
            return self.element().__value

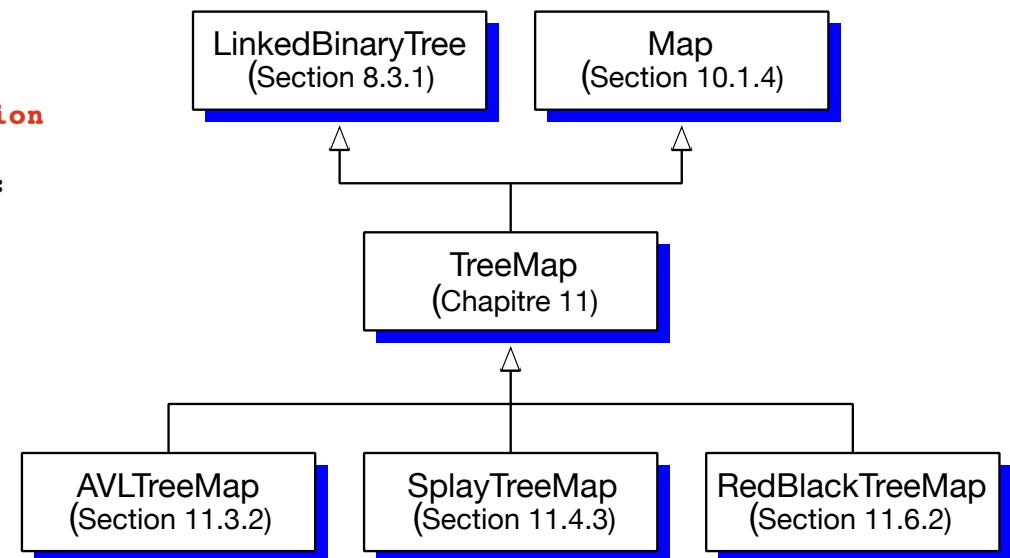
    #--- méthodes privées ---

    #pour rebalancer après une suppression
    def _rebalance_delete( self, p ):
        pass

    #pour rebalancer après une accession
    def _rebalance_access( self, p ):
        pass

    #pour rebalancer après une insertion
    def _rebalance_insert( self, p ):
        pass

```



Les spécialisations de *TreeMap*, *AVL*, *Splay* et *RedBlack*, diffèrent de par leur manière de manipuler la structure de l'arbre, et en particulier de rebalancement par leurs versions des méthodes *_rebalance*

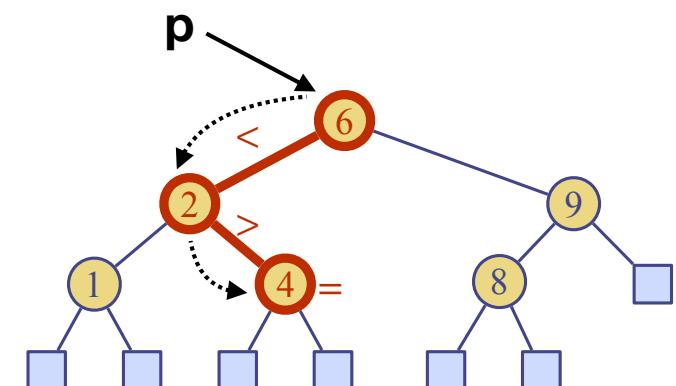
```

#recherche à partir d'une position l'élément de clé k
def _subtree_search( self, p, k ):
    #si la position possède l'élément de clé k
    #on retourne la position correspondante
    if k == p.key():
        return p
    #sinon, si la clé est < à celle de l'élément à position p
    elif k < p.key():
        #on poursuit la recherche dans le sous-arbre de gauche
        if self.left( p ) is not None:
            return self._subtree_search( self.left( p ), k )
    else:
        #sinon, k est plus grande et on poursuit dans le sous-arbre de droite
        if self.right( p ) is not None:
            return self._subtree_search( self.right( p ), k )
    #on arrive à un sous-arbre de p qui est vide
    #on retourne p, le dernier noeud visité
    #sa clé peut être >, = ou < à k selon le cas
    return p

#trouve et retourne la position du noeud qui possède
#l'élément de clé k
def _find_position( self, k ):
    if self.is_empty():
        return None
    else:
        p = self._subtree_search( self.root(), k )
        self._rebalance_access( p )
        return p

```

`_subtree_search(p, 4)`



```

#accession à la position avec clé k
def __getitem__( self, k ):
    #si arbre vide, on lance un exception
    if self.is_empty():
        raise KeyError
    else:
        #sinon, on cherche la position de clé k
        p = self._subtree_search( self.root(), k )
        #on rebalance post-accession
        self._rebalance_access( p )
        #si la clé k est absente, on lance une exception
        if k != p.key():
            raise KeyError
        #sinon, on retourne la valeur de la position de clé k
        return p.value()

#assignation ou insertion de l'élément (k, v)
def __setitem__( self, k, v ):
    #si l'arbre est vide, on crée la racine
    if self.is_empty():
        leaf = self._add_root( self._Item( k, v ) ) #de LinkedBinaryTree
    else:
        #sinon, on cherche si la position de clé k existe
        p = self._subtree_search( self.root(), k )
        #si elle existe
        if p.key() == k:
            #on change sa valeur
            p.element().__value = v
            #on rebalance post-accession
            self._rebalance_access( p )
            #on sort
            return
        else:
            #sinon, on crée le nouvel item et la nouvelle position
            item = self._Item( k, v )
            #à droite si la clé k est > que la clé de la position feuille
            if p.key() < k:
                leaf = self._add_right( p, item ) #de LinkedBinaryTree
            else:
                #sinon, à gauche
                leaf = self._add_left( p, item ) #de LinkedBinaryTree
    #on rebalance post-insertion
    self._rebalance_insert( leaf )

```

```

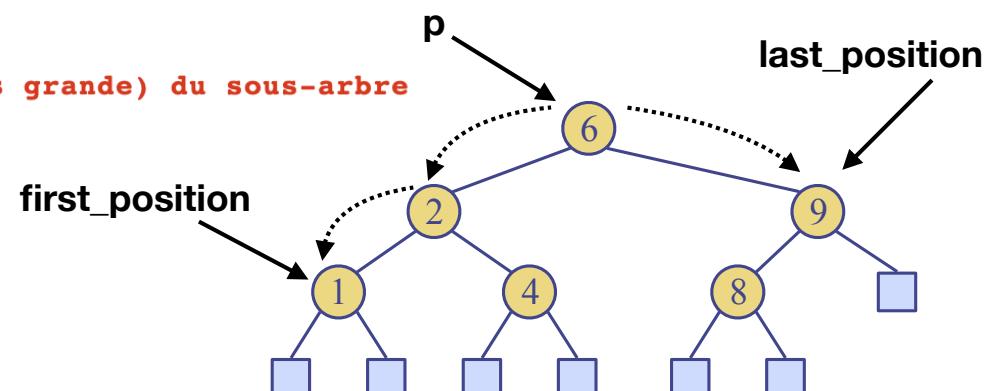
#accède et retourne la première position (clé la plus petite) du sous-arbre
#de racine p
def _subtree_first_position( self, p ):
    walk = p
    #on descend le plus loin possible à gauche
    while self.left( walk ) is not None:
        walk = self.left( walk )
    return walk

#accède et retourne la dernière position (clé la plus grande) du sous-arbre
#de racine p
def _subtree_last_position( self, p ):
    walk = p
    #on descend le plus loin possible à droite
    while self.right( walk ) is not None:
        walk = self.right( walk )
    return walk

#version racine de première position
def first( self ):
    return self._subtree_first_position( self.root() ) if len( self ) > 0 else None

#version racine de dernière position
def last( self ):
    return self._subtree_last_position( self.root() ) if len( self ) > 0 else None

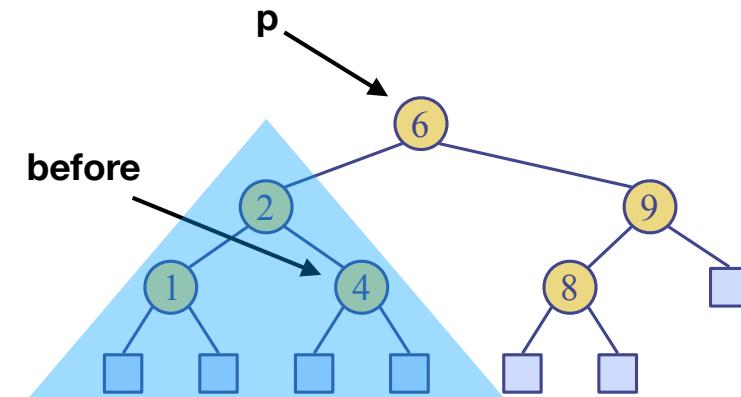
```



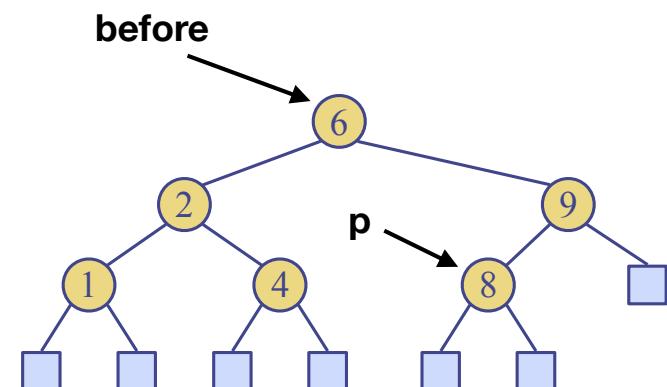
```

#retourne la position juste avant p
def before( self, p ):
    #on valide p
    self._validate( p )
    #la position précédente est soit la dernière position du sous-arbre gauche
    if self.left( p ):
        return self._subtree_last_position( self.left( p ) )
    #ou le premier ancêtre avec une clé plus petite que p
    #on remonte jusqu'à un virage à gauche
    else:
        walk = p
        above = self.parent( walk )
        while above is not None and walk == self.left( above ):
            walk = above
            above = self.parent( walk )
    return above

```

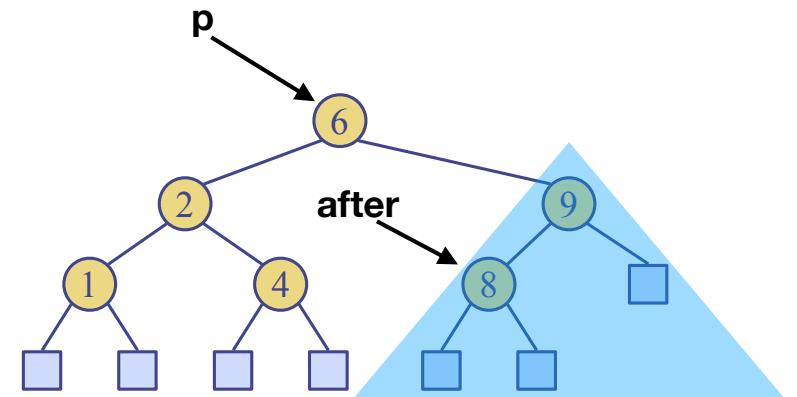


before est la dernière position du sous-arbre gauche

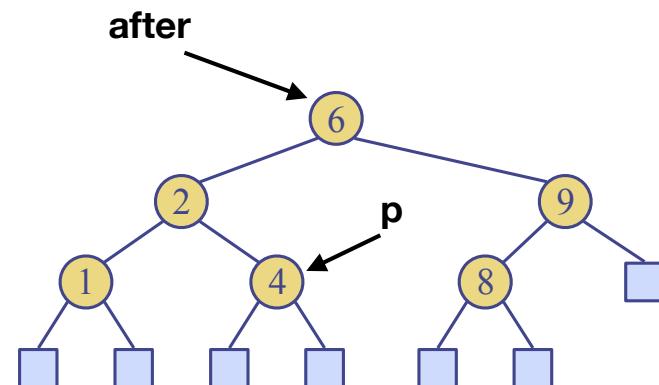


before est le premier ancêtre dont la clé est plus petite que celle de *p*

```
#retourne la position juste après p
def after( self, p ):
    #on valide p
    self._validate( p )
    #la position après est soit la première position du sous-arbre droit
    if self.right( p ):
        return self._subtree_first_position( self.right( p ) )
    #ou le premier ancêtre avec une clé plus grande que p
    #on remonte jusqu'à un virage à droite
    else:
        walk = p
        above = self.parent( walk )
        while above is not None and walk == self.right( above ):
            walk = above
            above = self.parent( walk )
    return above
```



after possède la plus petite clé du sous-arbre droit



after est le premier ancêtre dont la clé est plus grande que celle de *p*

```

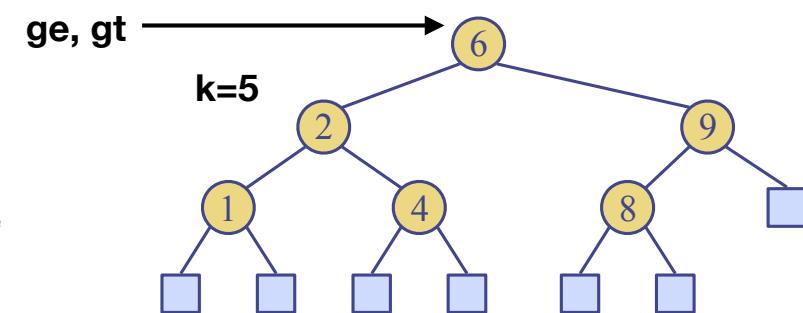
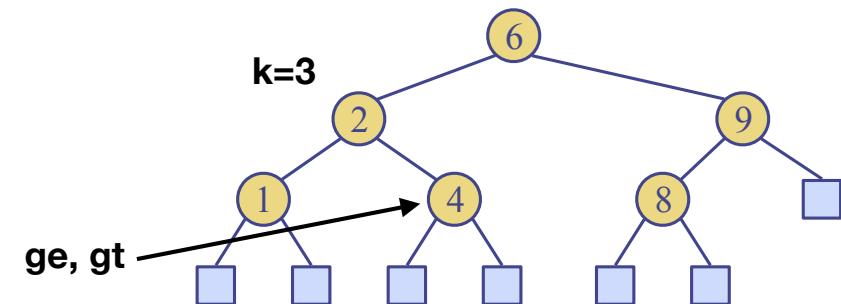
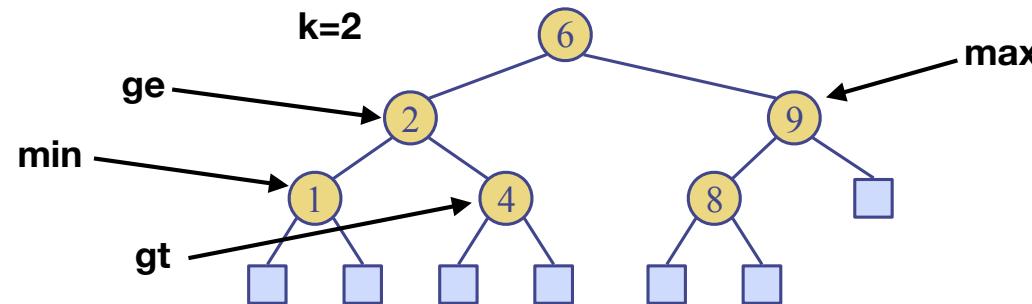
#retourne l'élément avec la clé minimum, soit first
def find_min( self ):
    if self.is_empty():
        return None
    else:
        p = self.first()
        return (p.key(), p.value())

#retourne l'élément avec la clé maximum, soit last
def find_max( self ):
    if self.is_empty():
        return None
    else:
        p = self.last()
        return (p.key(), p.value())

#retourne l'élément avec la clé plus grande ou égale à k
def find_ge( self, k ):
    if self.is_empty():
        return None
    else:
        #on trouve la position du noeud de clé k
        p = self._find_position( k )
        #si le noeud trouvé possède une clé < k
        #on retourne la suivante qui est > k (car clé k non trouvée)
        #sinon, soit on a trouvé la clé k ou on s'est arrêté
        #sur un noeud dont la clé est > k, donc on la retourne
        if p.key() < k:
            p = self.after( p )
        return (p.key(), p.value()) if p is not None else None

#retourne l'élément avec la clé plus grande que k
def find_gt( self, k ):
    if self.is_empty():
        return None
    else:
        #on trouve la position du noeud de clé k
        p = self._find_position( k )
        #si le noeud trouvé possède une clé <= k, on retourne la suivante
        #sinon, on s'est arrêté sur une clé > k, donc on la retourne
        if p.key() <= k:
            p = self.after( p )
        return (p.key(), p.value()) if p is not None else None

```

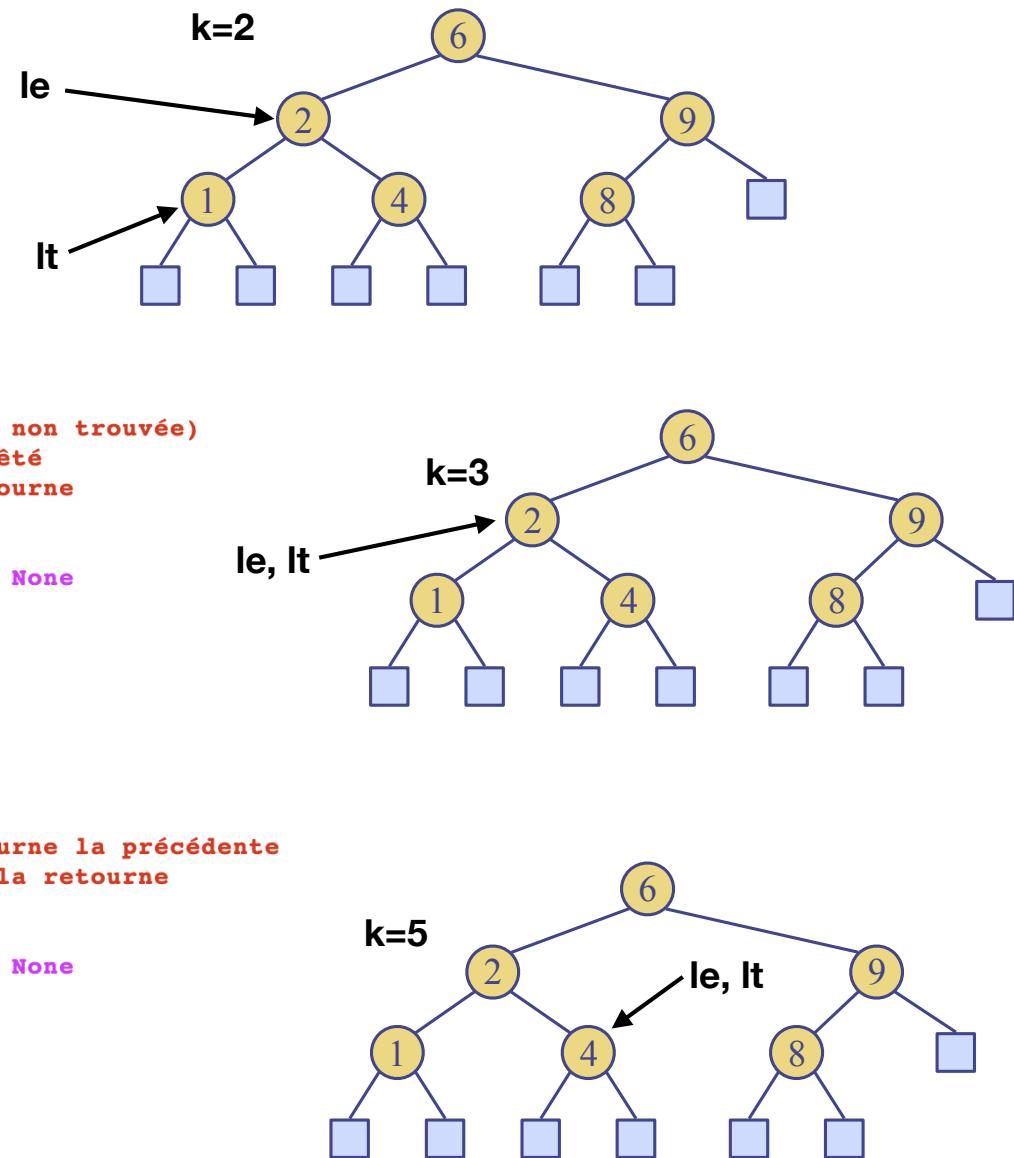


```

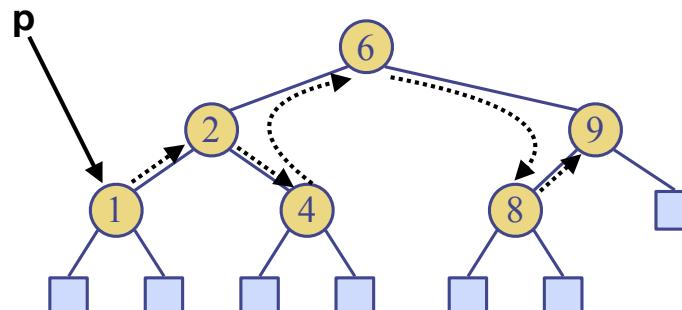
#retourne l'élément avec la clé plus petite ou égale à k
def find_le( self, k ):
    if self.is_empty():
        return None
    else:
        #on trouve la position du noeud de clé k
        p = self._find_position( k )
        #si le noeud trouvé possède une clé > k
        #on retourne la précédente qui est < k (car clé k non trouvée)
        #sinon, soit on a trouvé la clé k ou on s'est arrêté
        #sur un noeud dont la clé est < k, donc on la retourne
        if p.key() > k:
            p = self.before( p )
        return (p.key(), p.value()) if p is not None else None

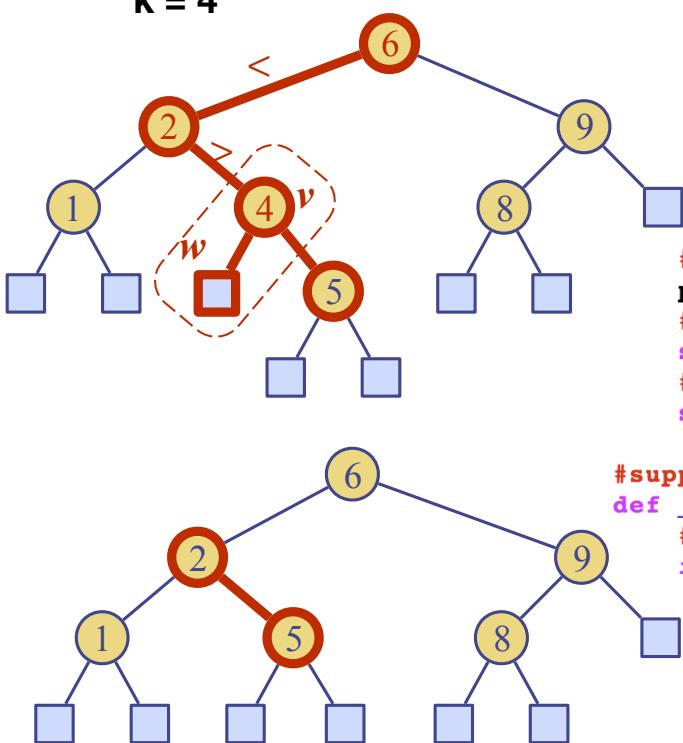
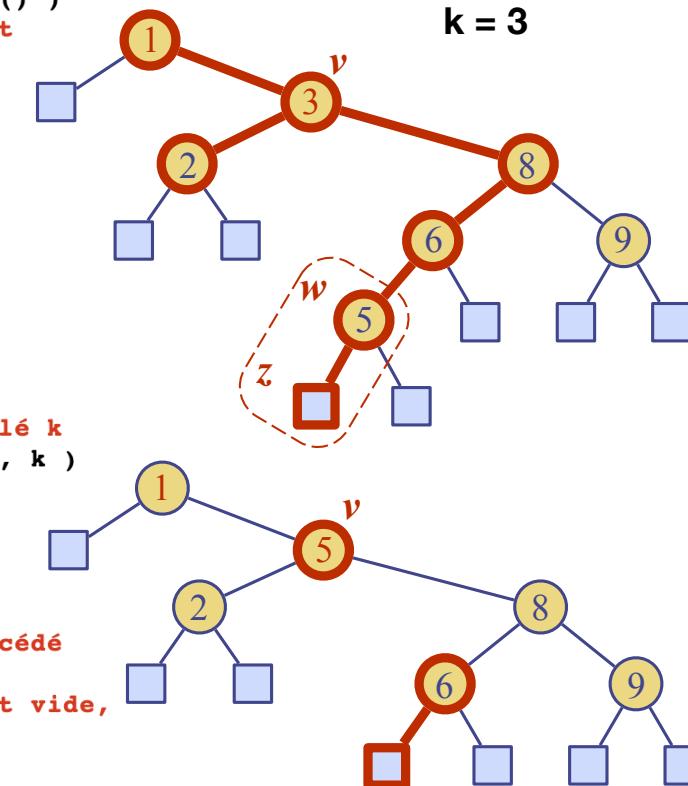
#retourne l'élément avec la clé plus petite que k
def find_lt( self, k ):
    if self.is_empty():
        return None
    else:
        #on trouve la position du noeud de clé k
        p = self._find_position( k )
        #si le noeud trouvé possède une clé >= k, on retourne la précédente
        #sinon, on s'est arrêté sur une clé < k, donc on la retourne
        if p.key() >= k:
            p = self.before( p )
        return (p.key(), p.value()) if p is not None else None

```



```
#itérateur sur les clés entre [start, stop[
def find_range( self, start = None, stop = None ):
    if not self.is_empty():
        if start is None:
            #si start est None, on commence avec la plus petite clé
            p = self.first()
            print( "start is None, p = ", p.key() )
        else:
            #sinon, on trouve la position de start
            p = self._find_position( start )
            #si elle n'existe pas et qu'on s'est arrêté sur
            #le noeud de clé < start, on commence à la clé suivante
            if p.key() < start:
                p = self.after( p )
        #on parcourt les clés à partir de start jusqu'à stop exclusivement
        #ou toutes les clés après start si stop est None
        while p is not None and (stop is None or p.key() < stop):
            yield (p.key(), p.value())
            p = self.after( p )
```

find_range()**stop = None****start = None****while p is not None and True****yield 1; p = 2****yield 2; p = 4****yield 4; p = 6****yield 6; p = 8****yield 8; p = 9****yield 9; p = None**

$k = 4$  $k = 3$ 

```

def delete( self, p ):
    #on valide p
    self._validate( p )
    #si p possède des sous-arbres gauche et droit
    #on substitut son élément par celui de la clé suivante
    #et on supprime le noeud qui la contient (le substitut)
    #sinon, on le supprime avec delete de LinkedBinaryTree.py
    if self.left( p ) and self.right( p ):
        #on trouve la clé de remplacement
        #par convention la première clé du sous-arbre droit
        replacement = self._subtree_first_position( self.right( p ) )
        #on modifie l'élément à position p par celui du substitut
        #replace est dans LinkedBinaryTree.py
        self._replace( p, replacement.element() )
        #le noeud à supprimer est le substitut
        p = replacement
    #on note le parent du noeud à supprimer
    parent = self.parent( p )
    #on supprime le noeud à position p
    self._delete( p )
    #on rebalance autour du parent
    self._rebalance_delete( parent )

#suppression d'une clé dans la Map
def __delitem__( self, k ):
    #si la Map n'est pas vide
    if not self.is_empty():
        #on cherche la position du noeud de clé k
        p = self._subtree_search( self.root(), k )
        #si trouvé, on la supprime
        if k == p.key():
            self.delete( p )
            return
        #sinon, la clé n'existe pas et on
        #rebalance autour du dernier noeud accédé
        self._rebalance_access( p )
    #si la clé n'est pas trouvée ou la Map est vide,
    #on lance une exception
    raise KeyError( "Key Error: ", k )

```

Performances

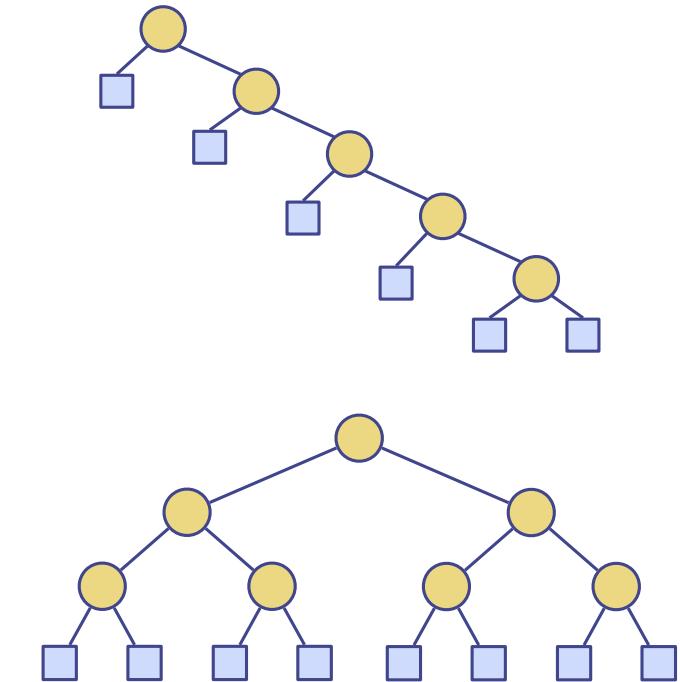
Considérons une *Map* avec n objets implémentée au moyen d'un arbre binaire de recherche de hauteur h

L'espace utilisé est dans $O(n)$

Les méthodes de recherche et de mise à jour prennent $O(h)$ -temps

La hauteur h est dans $O(n)$ dans le pire cas et $O(\log n)$ dans le meilleur cas

Operation	Running Time
$k \text{ in } T$	$O(h)$
$T[k], T[k] = v$	$O(h)$
$T.\text{delete}(p), \text{del } T[k]$	$O(h)$
$T.\text{find_position}(k)$	$O(h)$
$T.\text{first}(), T.\text{last}(), T.\text{find_min}(), T.\text{find_max}()$	$O(h)$
$T.\text{before}(p), T.\text{after}(p)$	$O(h)$
$T.\text{find_lt}(k), T.\text{find_le}(k), T.\text{find_gt}(k), T.\text{find_ge}(k)$	$O(h)$
$T.\text{find_range}(\text{start}, \text{stop})$	$O(s + h)$
$\text{iter}(T), \text{reversed}(T)$	$O(n)$

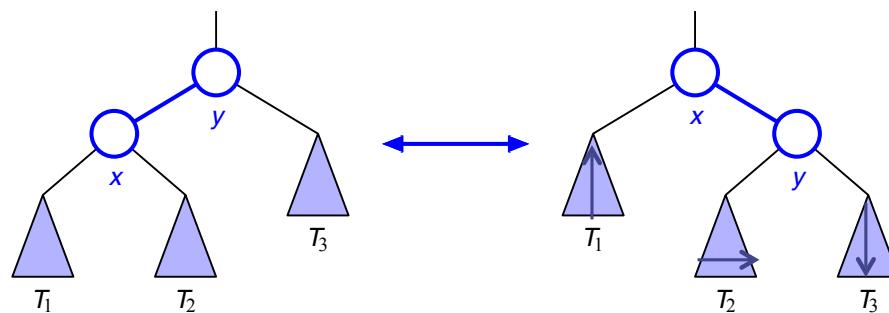


nb. s est le nombre de résultats.

Rebalancement

Le meilleur cas pour un ABR est lorsque l'arbre est balancé, c'est que sa hauteur est $\log n$.

Par conséquent, nous considérons maintenant des techniques d'arbres binaires de recherche balancés pour notre implémentation de *TreeMap*, c'est-à-dire qui gardent un ABR balancé après chaque opération.



```
def _rebalance_delete( self, p ):
    pass

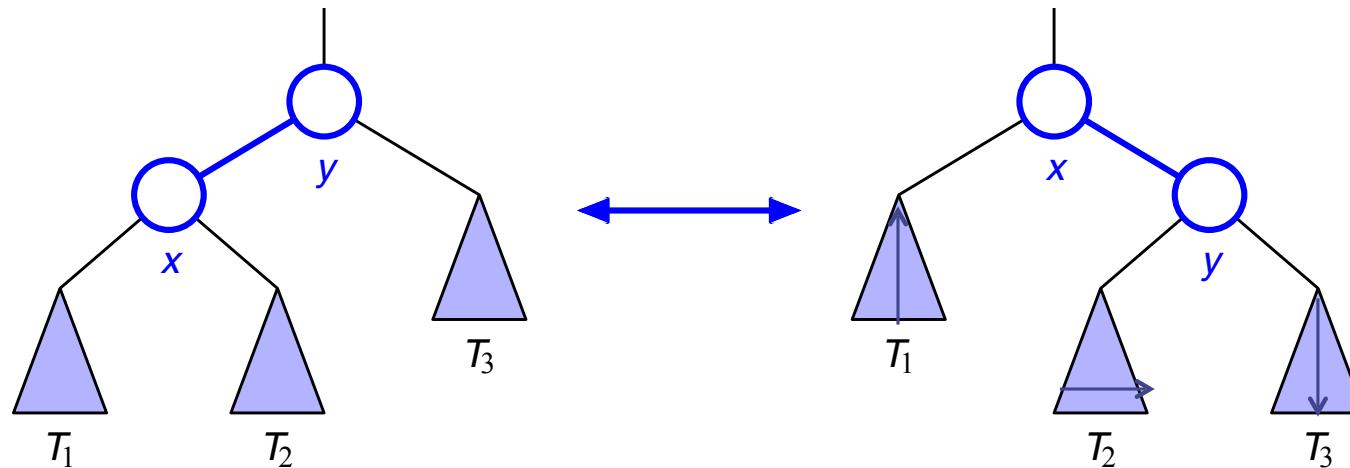
def _rebalance_access( self, p ):
    pass

def _rebalance_insert( self, p ):
    pass
```

À tout moment pendant les opérations de recherche et de mise à jour, on peut avoir à “rebalancer” l’arbre de recherche. Ces opérations de rebalancement seront implémentés dans 3 fonctions privées et appelées par les 3 méthodes : suppression, recherche et insertion.

Les structures de données d'arbres binaires de recherche balancés (AVL, Splay et RedBlack) se distinguent de par la manière dont leurs opérations de rebalancement sont implémentées.

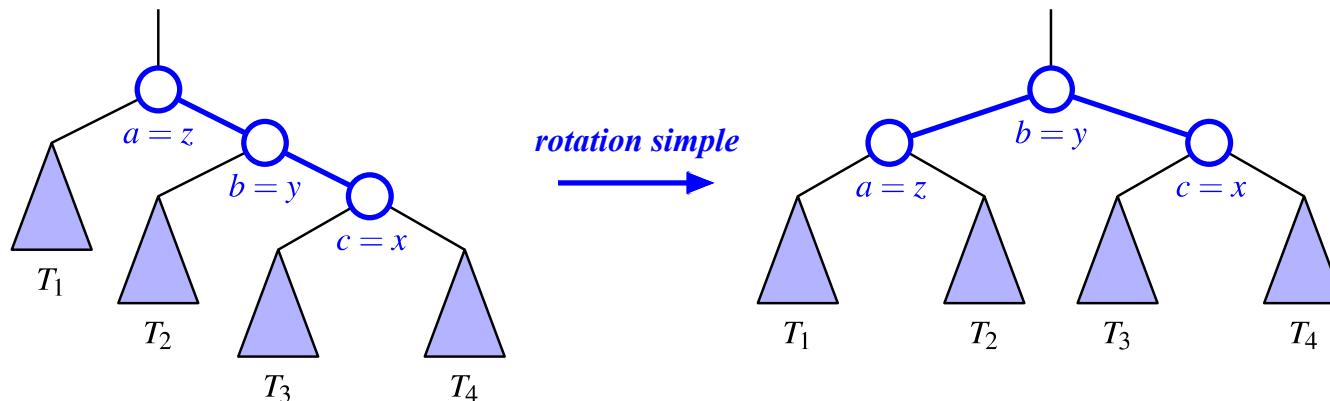
Pour rebalancer un ABR, il faut faire des rotations



L'opération principale pour balancer un ABR est une rotation

- Modifie la structure en $O(1)$ -temps
- Maintient la propriété de l'arbre binaire de recherche

Restructuration de trois noeuds (rotation simple)



Nous considérons une position x , son parent y et son grand-parent z . L'objectif est de restructurer le sous-arbre enraciné à z pour réduire la longueur du chemin jusqu'à x et ses sous-arbres.

Restructuration de trois noeuds (cas vers la droite)

_restructure(x) :

entrée: position x et ABR T

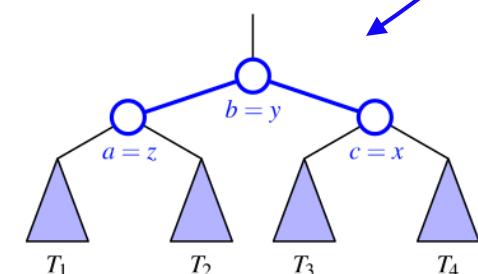
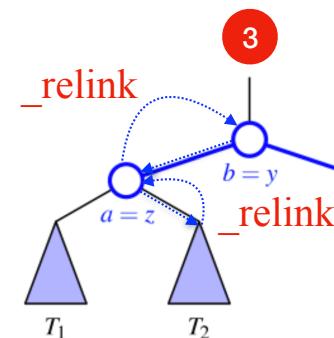
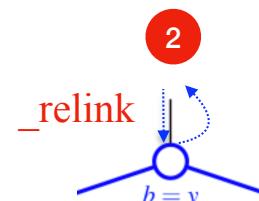
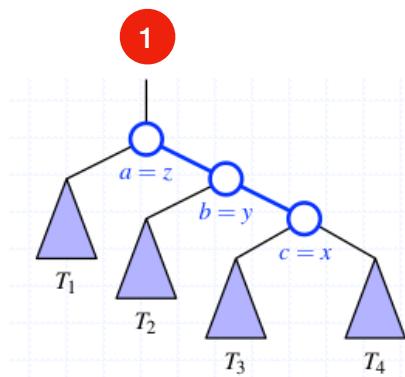
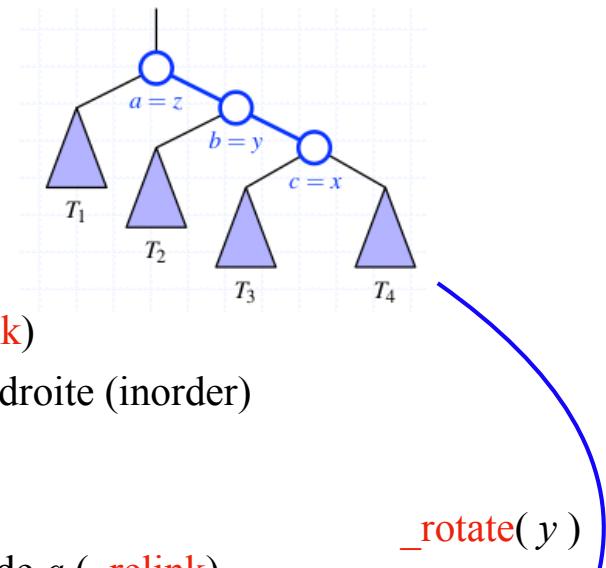
sortie: T après la restructuration des trois noeuds

opération: rotation simple autour de y (_rotate) en 3 (re)liaisons (_relink)

- 1 On définit (a, b, c) comme étant les noeuds x, y et z dans l'ordre de gauche à droite (inorder) et (T_1, T_2, T_3, T_4) les 4 sous-arbres de x, y et z
- 2 On met b comme enfant de z (_relink) à gauche ou droite selon le cas
- 3 On met a comme enfant gauche de b (_relink) et T_2 comme sous-arbre droit de a (_relink)

Ce qui n'a pas bougé :

c est l'enfant droit de b , T_3 et T_4 les sous-arbres gauche et droit de c , et T_1 le sous-arbre gauche de a .



Restructuration de trois noeuds (cas symétrique vers la gauche)

_restructure(x) :

entrée: position x et ABR T

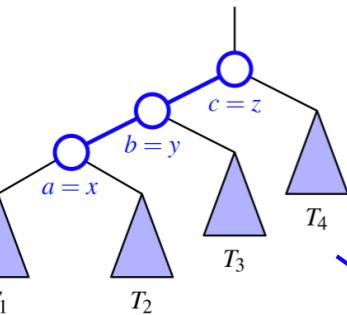
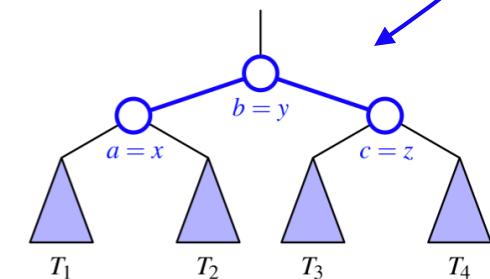
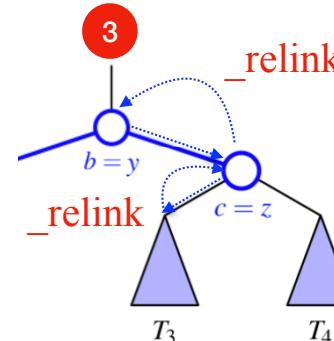
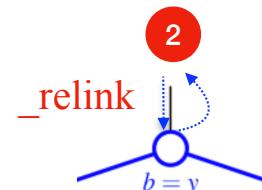
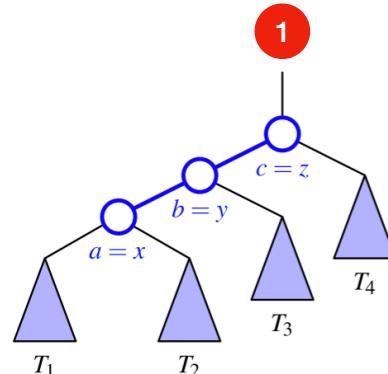
sortie: T après la restructuration des trois noeuds

opération: rotation simple autour de y (_rotate) en 3 (re)liaisons (_relink)

- 1 On définit (a, b, c) comme étant les noeuds x, y et z dans l'ordre de gauche à droite (inorder) et (T_1, T_2, T_3, T_4) les 4 sous-arbres de x, y et z
- 2 On met b comme enfant de z (_relink) à gauche ou droite selon le cas
- 3 On met c comme enfant droit de b (_relink) et T_3 comme sous-arbre gauche de c (_relink)

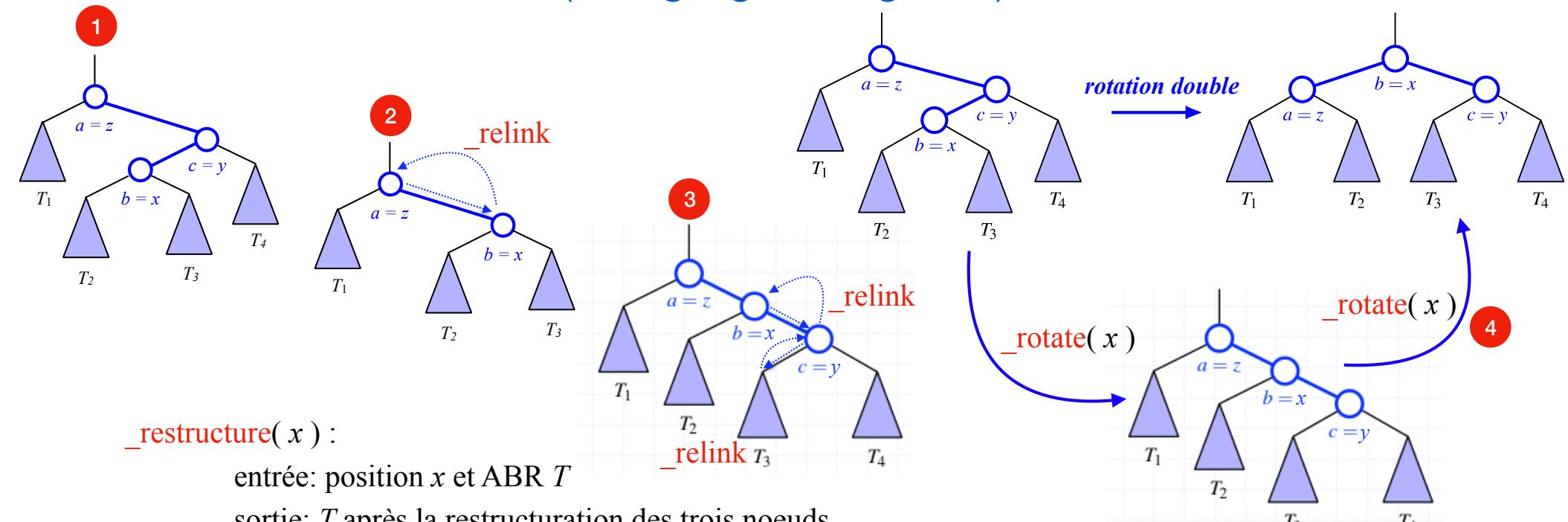
Ce qui n'a pas bougé :

a est l'enfant gauche de b , T_1 et T_2 les sous-arbres gauche et droit de a et T_4 le sous-arbre droit de c .



_rotate(y)

Restructuration de trois noeuds (cas zig-zag droite-gauche)



_restructure(x) :

entrée: position x et ABR T

sortie: T après la restructuration des trois noeuds

opération: rotation double autour de x (rotate) en 3 (re)liaisons (relink) chacune

1 On définit (a, b, c) comme étant les noeuds x, y et z dans l'ordre de gauche à droite (inorder) et (T_1, T_2, T_3, T_4) les 4 sous-arbres de x, y et z

2 On met b comme enfant droit de z (relink)

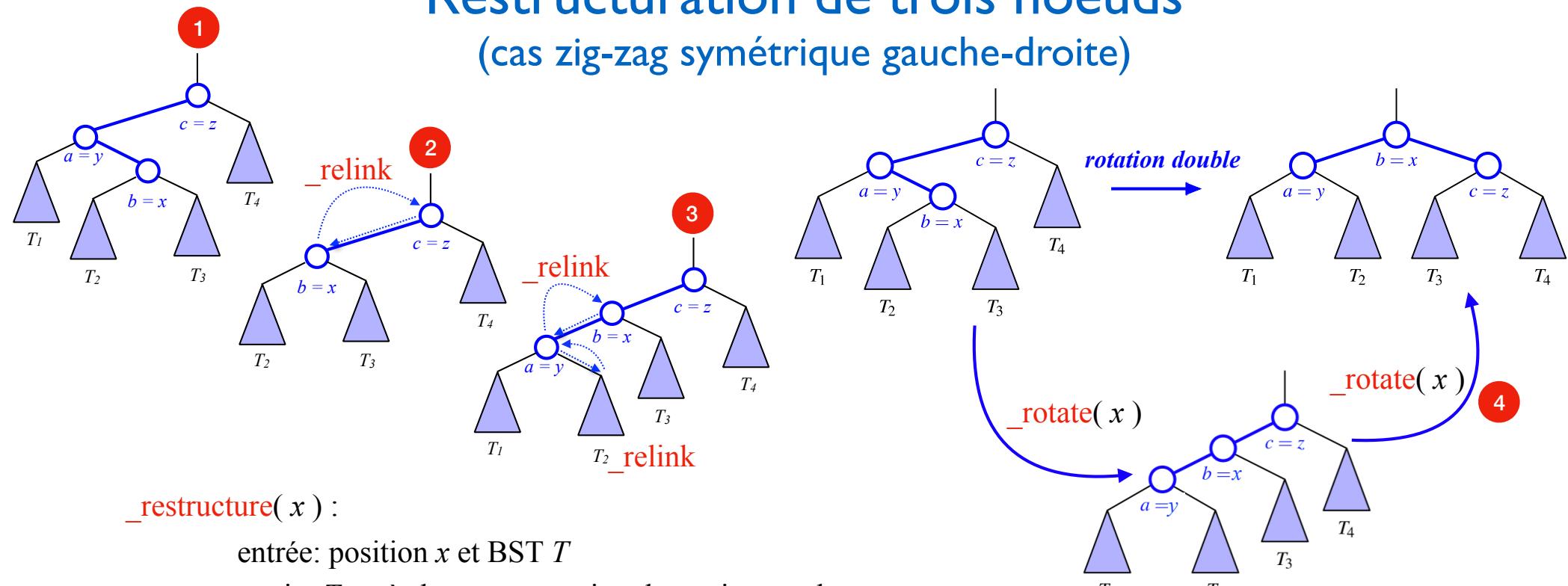
3 On met c comme enfant droit de b (relink) et T_3 comme sous-arbre gauche de c (relink)

Ce qui n'a pas bougé :

T_1 est le sous-arbre gauche de a , T_2 est le sous-arbre gauche de b et T_4 est le sous-arbre droit de c .

4 On se retrouve dans un cas de rotation simple, donc on applique une 2ème rotation autour de x (rotate)

Restructuration de trois noeuds (cas zig-zag symétrique gauche-droite)



_restructure(x) :

entrée: position x et BST T

sortie: T après la restructuration des trois noeuds

opération: double rotation de x (rotate) en 3 (re)liaisons (relink) chacune

① On définit (a, b, c) comme étant les noeuds x, y et z dans l'ordre de gauche à droite (inorder) et (T_1, T_2, T_3, T_4) les 4 sous-arbres de x, y et z

② On met b comme enfant gauche de z (relink)

③ On met a comme enfant gauche de b (relink) et T_2 comme sous-arbre droit de a (relink)

Ce qui n'a pas bougé :

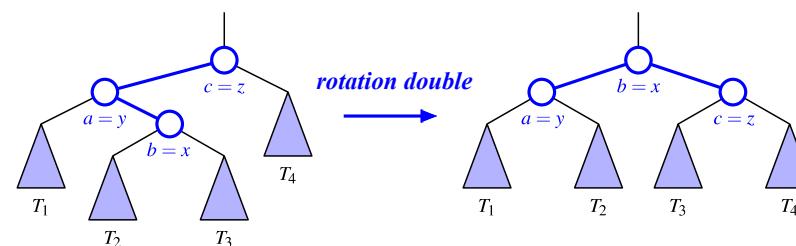
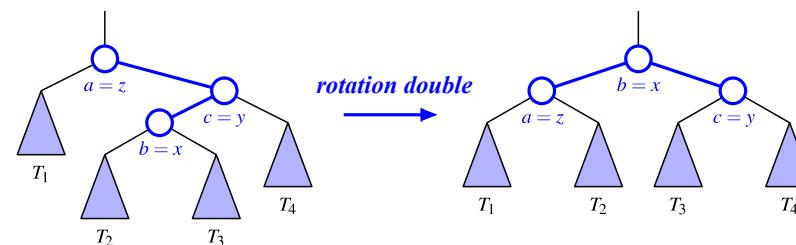
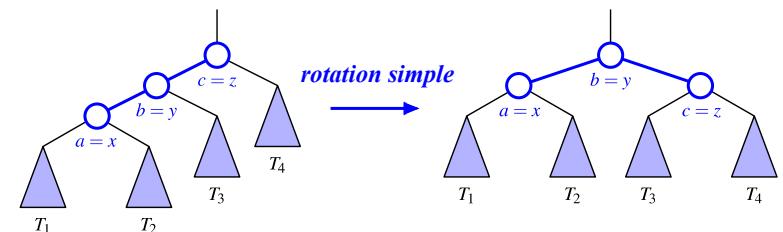
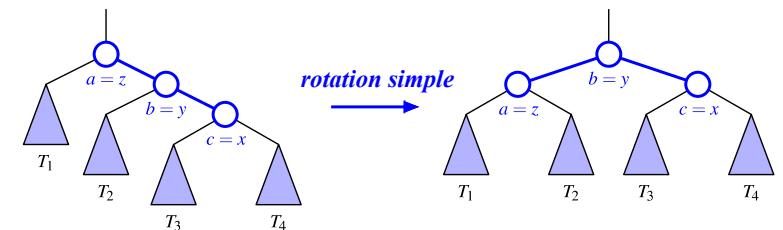
T_1 est le sous-arbre gauche de a , T_3 est le sous-arbre droit de b et T_4 est le sous-arbre droit de c .

④ On se retrouve dans un cas de rotation simple : rotation simple de x (rotate)

```

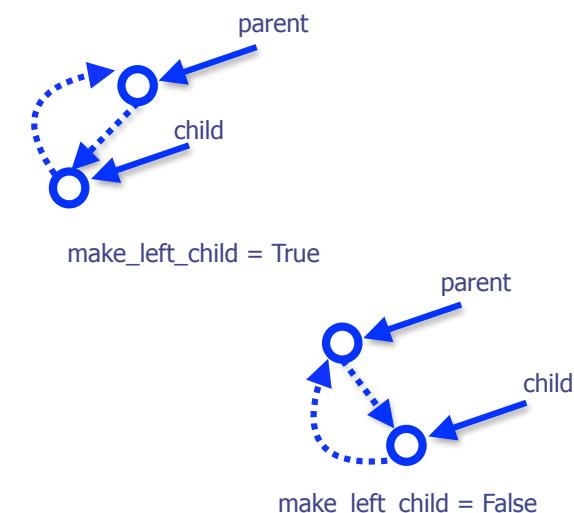
#restructuration pour le noeud x
def _restructure( self, x ):
    #y est le parent de x
    y = self.parent( x )
    #z est le grand-parent de x
    z = self.parent( y )
    #on identifie le cas
    #pour les cas de rotation simple (pas de zig zag)
    #si x est enfant droit de y et y enfant droit de z, ou
    #si x est enfant gauche de y et y enfant gauche de z
    if( x == self.right( y ) ) == (y == self.right( z ) ):
        #rotation simple autour de y
        self._rotate( y )
        return y
    #sinon, on est dans un cas de rotation double (zig zag)
    else:
        #on applique 2 fois la rotation sur x
        self._rotate( x )
        #après la 1ère rotation, on est dans un des 2 premiers cas
        self._rotate( x )
        return x

```



Lier un enfant à gauche ou à droite d'un parent (_relink)

```
#liaison d'un parent et son enfant (qui peut être None)
def _relink( self, parent, child, make_left_child ):
    #à gauche
    if make_left_child:
        parent._left = child
    #ou à droite (selon make_left_child)
    else:
        parent._right = child
    #on relie l'enfant au (nouveau) parent
    if child is not None:
        child._parent = parent
```

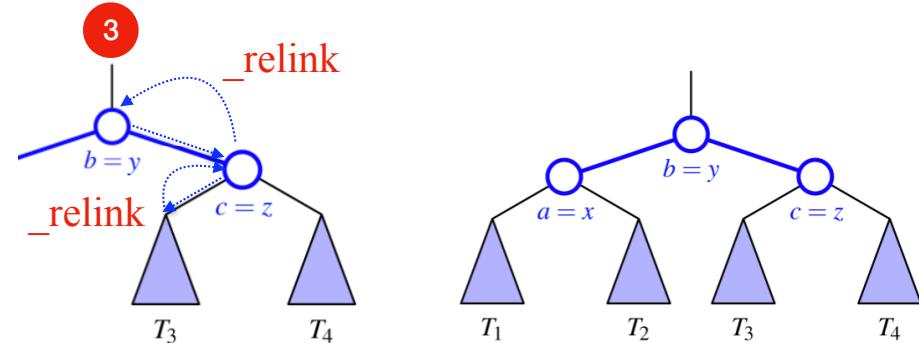
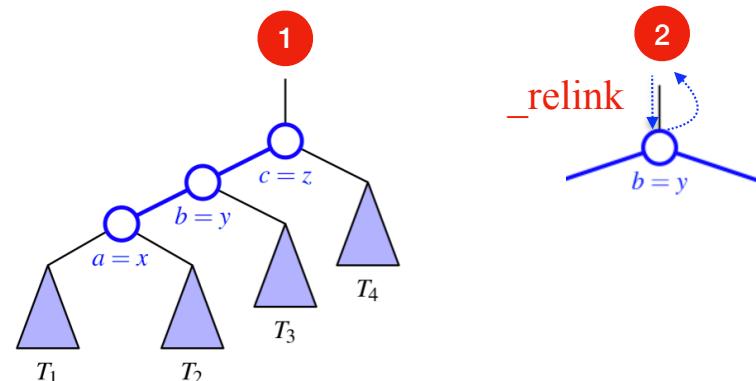
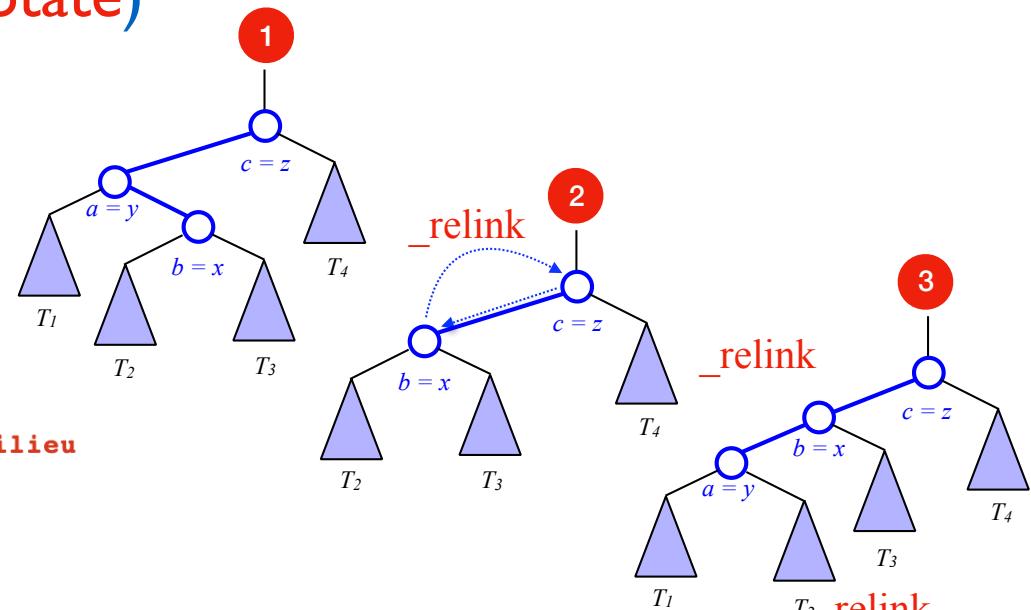


Rotation simple (_rotate)

```

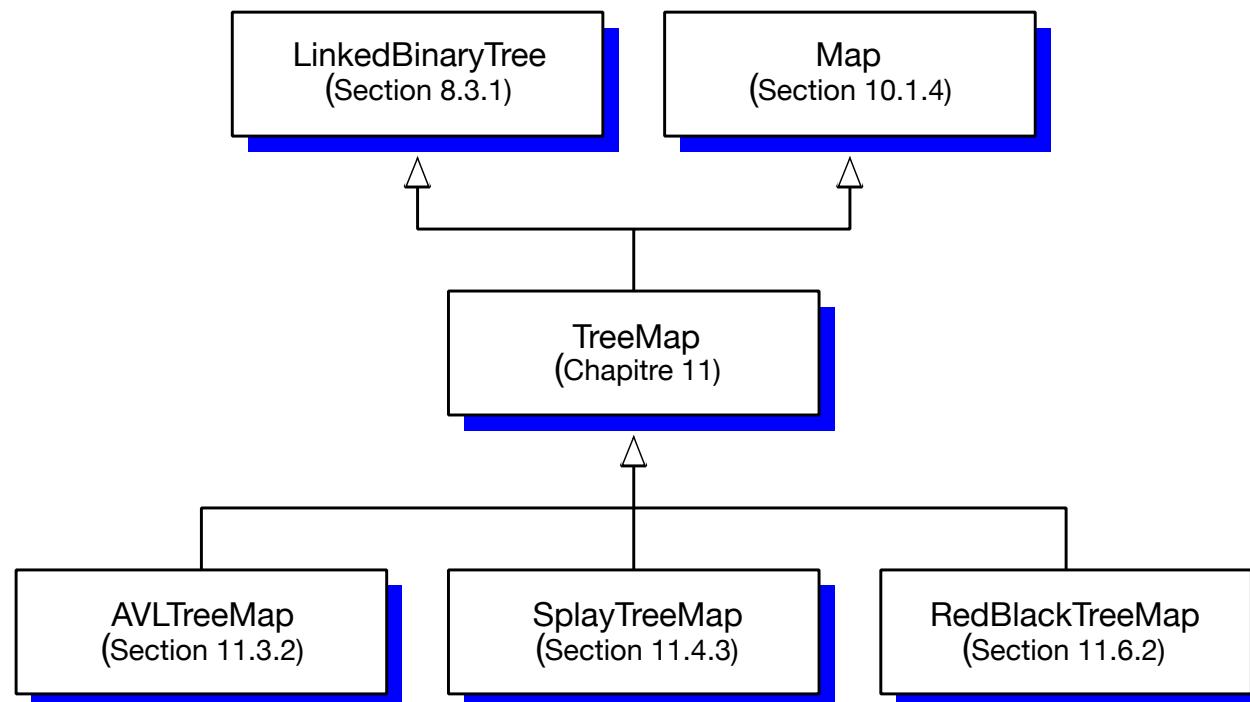
#rotation autour de p
def _rotate( self, p ):
    #x est le noeud p
    x = p._node
    #y est son parent
    y = x._parent
    #z est son grand-parent
    z = y._parent
    #si z est None, on a 2 noeuds, pas de grand-parent
    if z is None: #x remplacera y à la racine
        self._root = x
        x._parent = None
    else:
        #x est adopté par son grand-père
        self._relink( z, x, y == z._left )
    #rotation de x et y et transfert du sous-arbre du milieu
    if x == y._left:
        self._relink( y, x._right, True )
        self._relink( x, y, False )
    else:
        self._relink( y, x._left, False )
        self._relink( x, y, True )

```



Trois prochaines sections

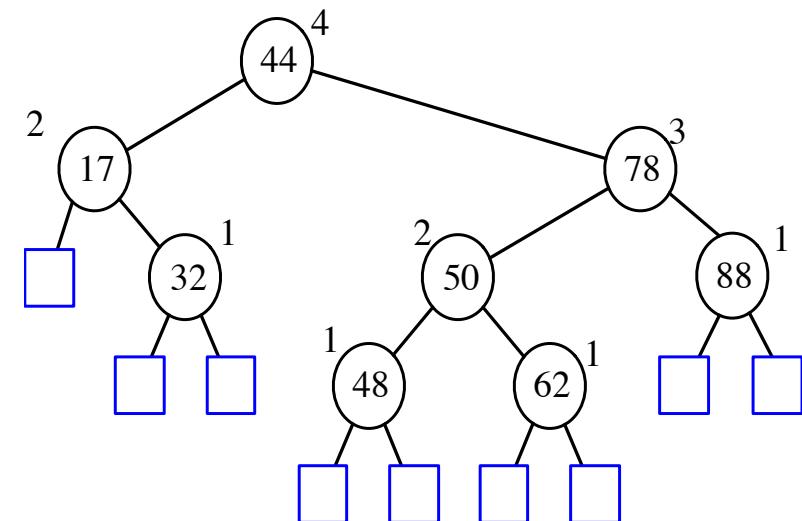
AVL, Splay, 2,4 (en introduction aux RedBlack) et RedBlack



Arbres AVL

Inventés en 1962 par
G.M. Adelson et Velskii E.M. Landis

- Les arbres AVL sont équilibrés/balancés !
- Un arbre AVL est un arbre de recherche binaire tel que pour chaque nœud interne v, les hauteurs des enfants de v peuvent différer d'au plus 1.



Un exemple d'arbre AVL avec les hauteurs des noeuds internes. Par convention, la hauteur des noeuds externes est 1.
La hauteur d'un noeud interne est la hauteur maximum d'un de ses enfants + 1.

Hauteur d'un arbre AVL

La hauteur d'un arbre AVL stockant n clés est dans $O(\log n)$.

Prenons $n(h)$, le nombre minimum de nœuds internes d'un arbre AVL de hauteur h .

On voit (en bas à droite) que :

$n(1) = 1$, un AVL de hauteur 1 doit avoir exactement un nœud

$n(2) = 2$, un AVL de hauteur 2 doit avoir au moins deux nœuds

Un AVL avec un minimum de nœuds internes de hauteur $h \geq 3$ possède deux sous-arbres AVL avec le minimum de nœuds, soit un avec une hauteur $h-1$ et l'autre de hauteur $h-2$

En prenant en compte la racine, pour $h \geq 3$ on obtient la formule suivante :

$$n(h) = 1 + n(h-1) + n(h-2)$$

En utilisant le fait que $n(h-1) > n(h-2)$, on remplace $n(h-1)$ par $n(h-2)$ pour minorer et on laisse tomber le 1 :

$$n(h) > 2 n(h-2)$$

Cette formule nous indique que $n(h)$ double au moins chaque fois que h augmente de 2, donc que $n(h)$ croît exponentiellement. On peut montrer ce fait en appliquant cette formule à répétition, ce qui donne la série suivante :

$$n(h) > 2 n(h-2),$$

$$n(h) > 4 n(h-4),$$

$$n(h) > 8 n(h-6), \dots \text{(par induction)},$$

$$\mathbf{n(h) > 2^i n(h-2i)}$$

Donc, $n(h) > 2^i n(h-2i)$, pour tout entier i , tel que $h-2i \geq 1$. On connaît les valeurs de $n(1)$ et $n(2)$,

donc remplaçons i tel que $h - 2i$ est égal à 1 ou 2, prenons

$$i = \lceil h/2 \rceil - 1, \text{ ex: } h = 13, \text{ donc } i = 7-1 = 6; h - 2i = 13 - 2 \times 6 = 1$$

On substitue i dans l'inéquation et on obtient pour $h \geq 3$:

$$n(h) > 2^{\lceil h/2 \rceil - 1} n(h - 2(\lceil h/2 \rceil - 1))$$

$n(h) > 2^{\lceil h/2 \rceil - 1} n(1)$, on a défini $i = \lceil h/2 \rceil - 1$ pour que $h - 2i = 1$ ou 2.

$$n(h) > 2^{\lceil h/2 \rceil - 1}$$

On prend les logarithmes de chaque côté :

$$\log(n(h)) > h/2 - 1,$$

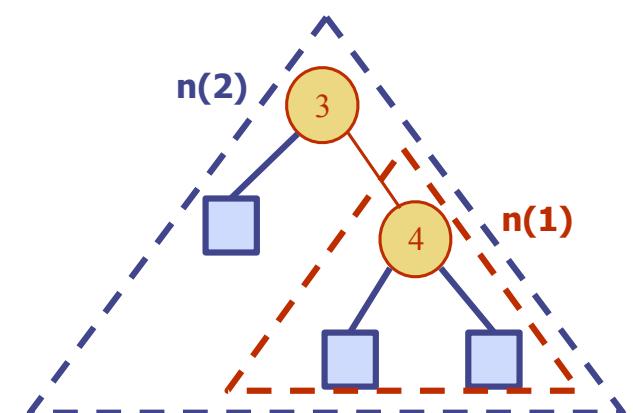
à partir de laquelle on obtient :

$$h < 2 \log(n(h)) + 2$$

Ainsi, la hauteur d'un arbre AVL contenant n noeuds internes est dans $O(\log n)$.

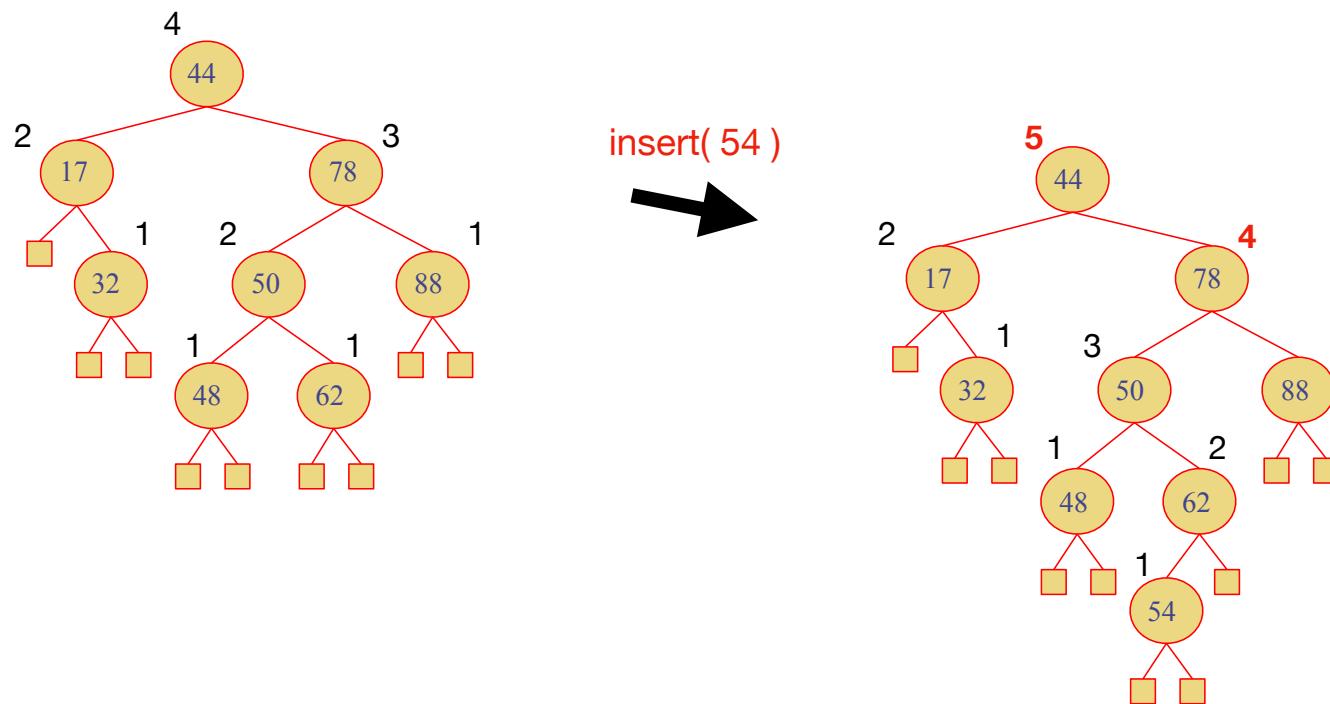
Notez que les arbres AVL avec un nombre minimum de nœuds sont les pires exemples pour le ratio n/h ; avec la même hauteur on peut avoir des arbres AVL avec plus de nœuds.

Si nous pouvons limiter la hauteur de ces pires exemples, nous aurons réussi à limiter la hauteur de tous les arbres AVL.

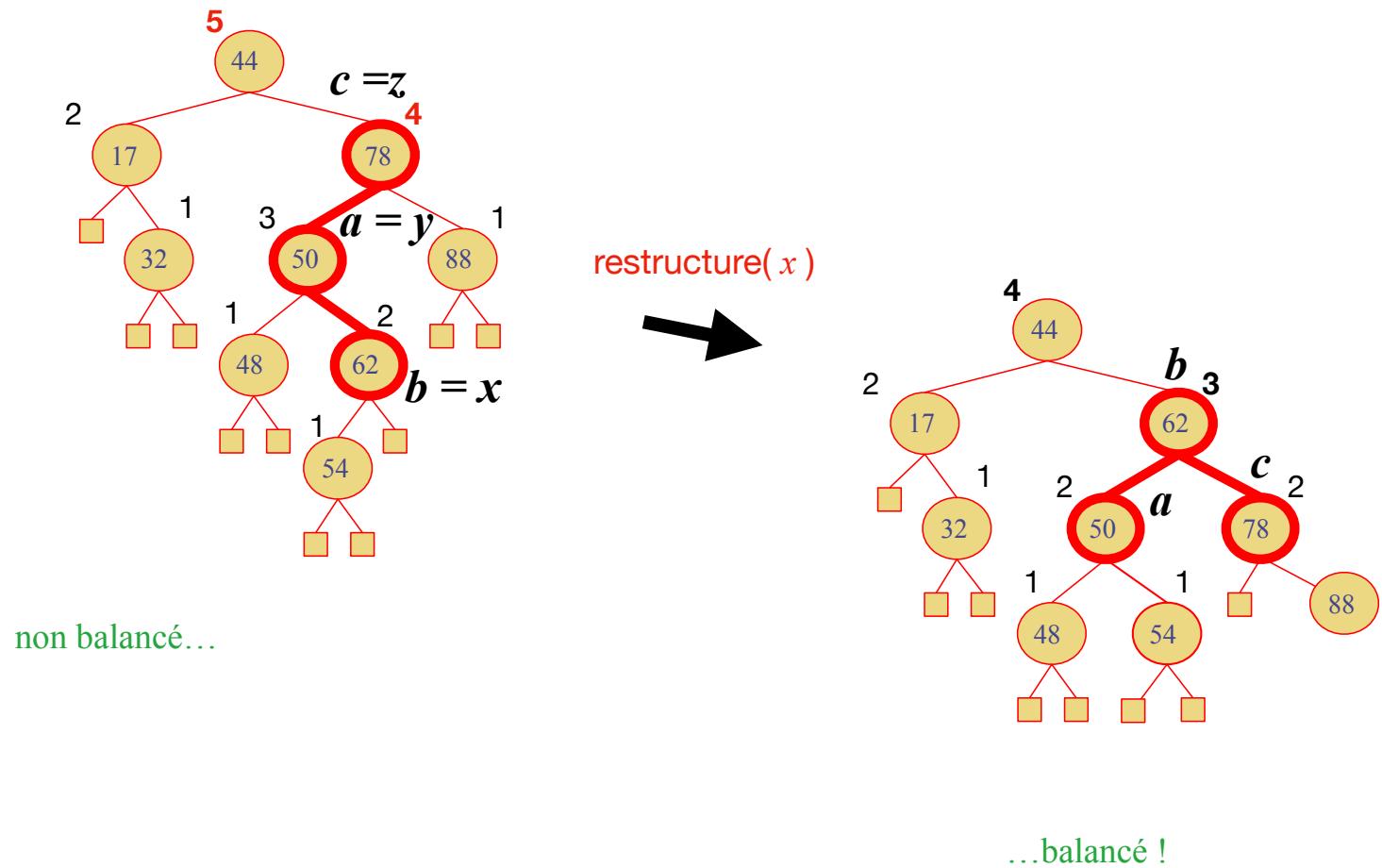


AVL insertion

L'insertion est comme dans un arbre de recherche binaire.
Toujours en développant un noeud externe.

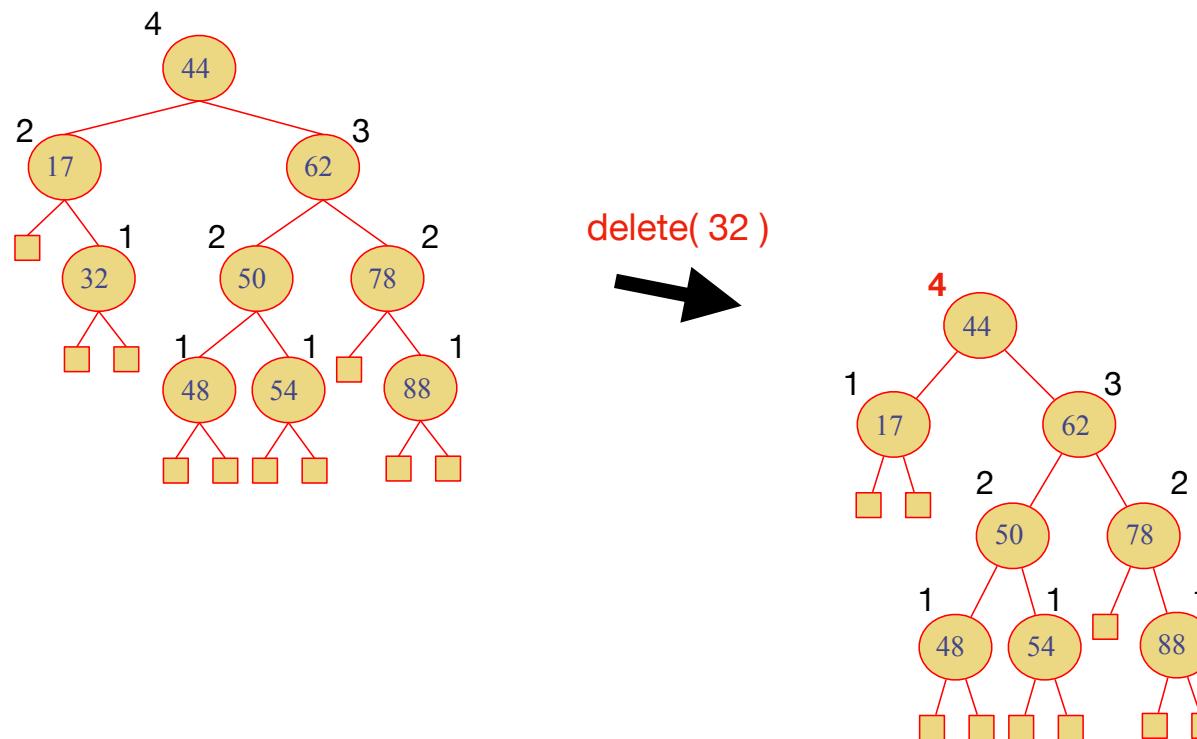


AVL insertion

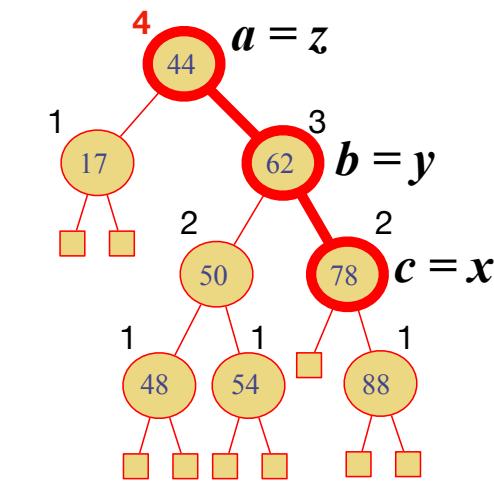


AVL suppression

La suppression commence comme dans un arbre de recherche binaire, ce qui signifie que le noeud supprimé deviendra un noeud externe vide. Son parent, w, peut causer un déséquilibre.

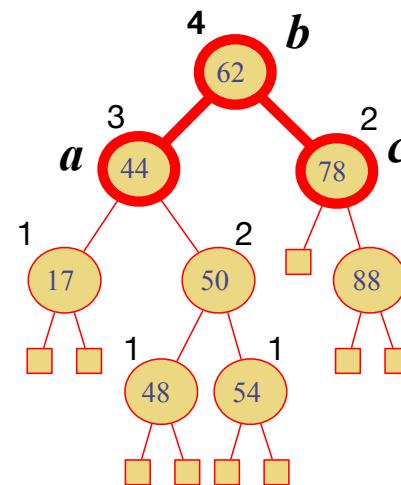


AVL delete



non balancé...

restructure(x)



...balancé !

```
from TreeMap import TreeMap

#classe AVLTreeMap
class AVLTreeMap( TreeMap ):

    #ajout de la hauteur d'un noeud
    class _Node( TreeMap._Node ):
        __slots__ = '_height' #membre additionel

    #constructeur : on appelle celui de TreeMap et on met la hauteur par défaut à 0
    def __init__( self, element, parent = None, left = None, right = None ):
        super().__init__( element, parent, left, right )
        self._height = 0

    #accès à la hauteur du sous-arbre gauche
    def left_height( self ):
        return self._left._height if self._left is not None else 0

    #accès à la hauteur du sous-arbre droit
    def right_height( self ):
        return self._right._height if self._right is not None else 0
```

```
#recalcul de la hauteur d'un noeud
def _recompute_height( self, p ):
    p._node._height = 1 + max( p._node.left_height(), p._node.right_height() )

#noeud balancé si différence des hauteurs de ses enfants est au plus 1
def _isbalanced( self, p ):
    return abs( p._node.left_height() - p._node.right_height() ) <= 1

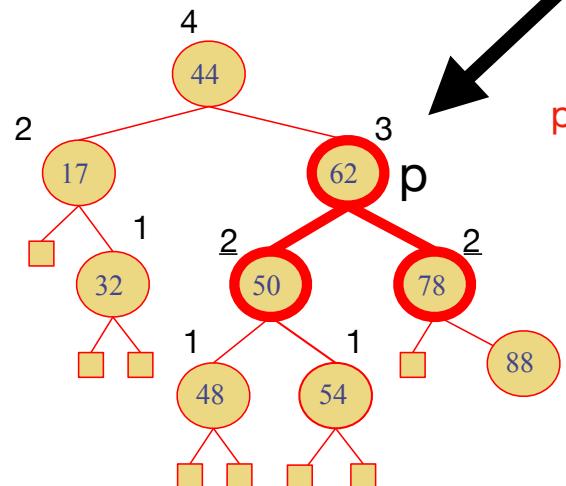
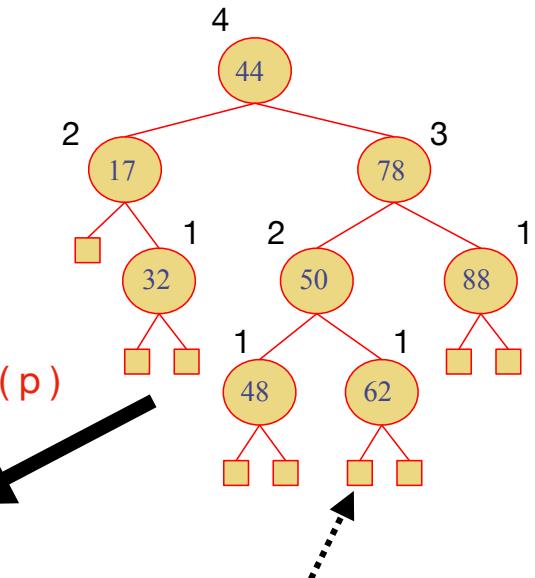
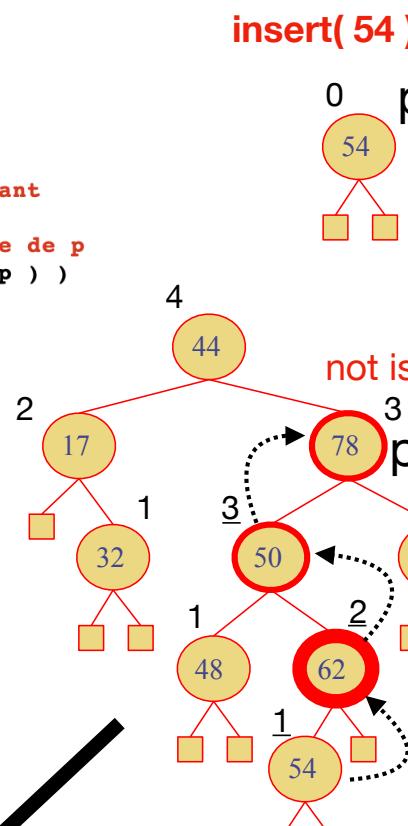
#retourne le plus grand (haut) des enfants de p
#si favorleft, on lui ajoute 1 pour le choisir en cas d'égalité
def _tall_child( self, p, favorleft = False ):
    if p._node.left_height() + ( 1 if favorleft else 0 ) > p._node.right_height():
        return self.left( p )
    else:
        return self.right( p )

#retourne le plus grand (haut) des petits enfants de p
def _tall_grandchild( self, p ):
    child = self._tall_child( p )
    #si enfant gauche, on favorise le petit enfant gauche
    #si enfant droit, on favorise le petit enfant droit
    #pour choisir une rotation simple plutôt que (zig-zag) double
    alignment = ( child == self.left( p ) )
    return self._tall_child( child, alignment )
```

```
#rebalancement autour de p, propagé vers le haut
def _rebalance( self, p ):
    while p is not None:
        #on note l'ancienne hauteur
        old_height = p._node._height
        #si on note un débalancement autour de p
        if not self._isbalanced( p ):
            #on restructure autour du plus grand petit enfant
            #on met p sur la racine du résultat
            #on recalcule les hauteurs à gauche et à droite de p
            p = self._restructure( self._tall_grandchild( p ) )
            self._recompute_height( self.left( p ) )
            self._recompute_height( self.right( p ) )
        #on recalcule la hauteur de p
        self._recompute_height( p )
        #si la hauteur est inchangée, on sort
        if p._node._height == old_height:
            p = None
        else:
            #sinon, on propage au parent
            p = self.parent( p )

#on rebalance après un insertion
def _rebalance_insert( self, p ):
    self._rebalance( p )

#on rebalance après une suppression
def _rebalance_delete( self, p ):
    self._rebalance( p )
```



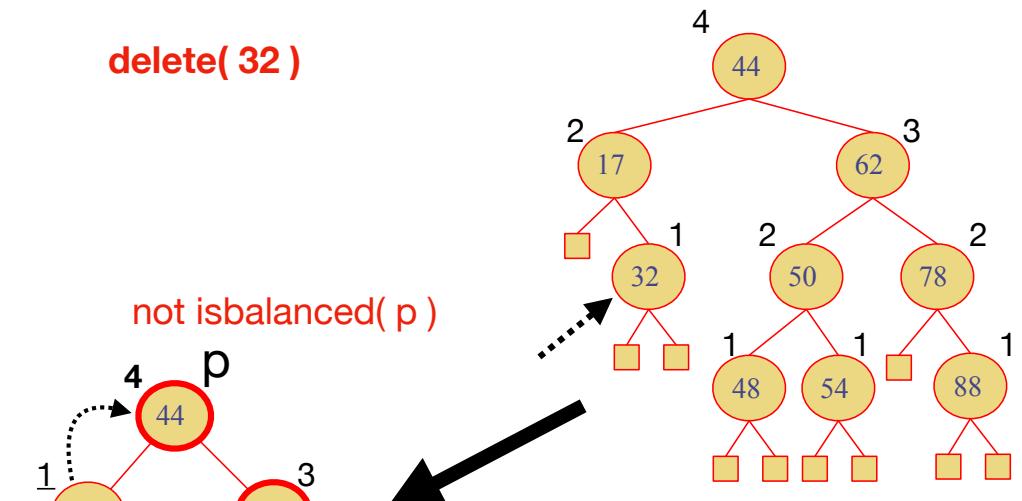
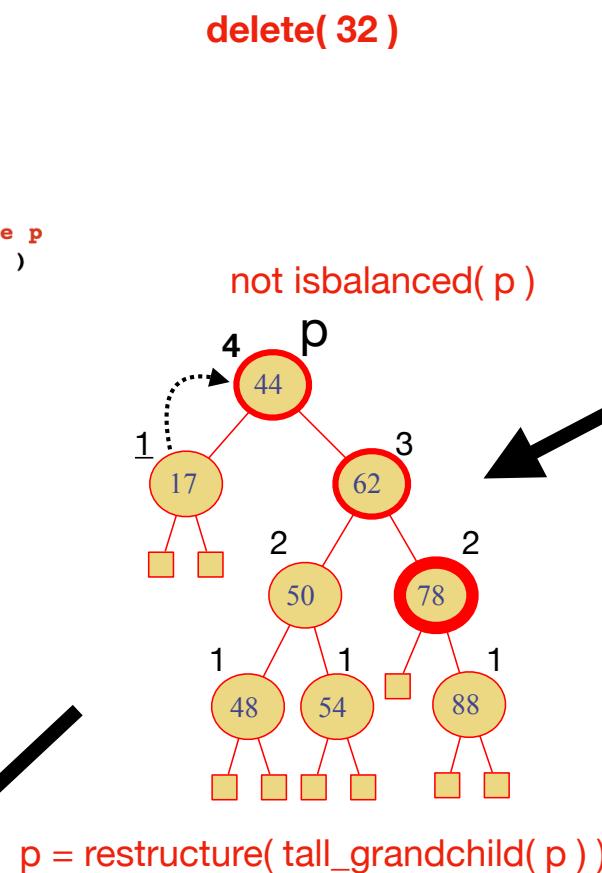
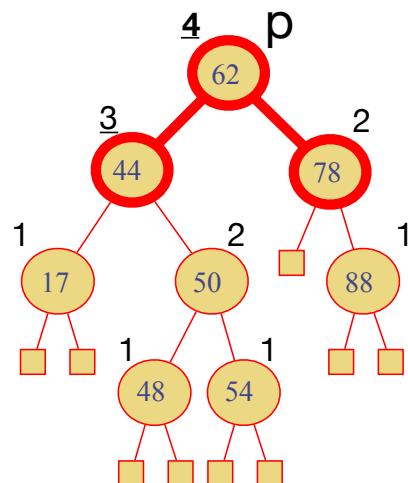
```

#rebalancement autour de p, propagé vers le haut
def _rebalance( self, p ):
    while p is not None:
        #on note l'ancienne hauteur
        old_height = p._node._height
        #si on note un débordement autour de p
        if not self._isbalanced( p ):
            #on restructure autour du plus grand petit enfant
            #on met p sur la racine du résultat
            #on recalcule les hauteurs à gauche et à droite de p
            p = self._restructure( self._tall_grandchild( p ) )
            self._recompute_height( self.left( p ) )
            self._recompute_height( self.right( p ) )
        #on recalcule la hauteur de p
        self._recompute_height( p )
        #si la hauteur est inchangée, on sort
        if p._node._height == old_height:
            p = None
        else:
            #sinon, on propage au parent
            p = self.parent( p )

#on rebalance après un insertion
def _rebalance_insert( self, p ):
    self._rebalance( p )

#on rebalance après une suppression
def _rebalance_delete( self, p ):
    self._rebalance( p )

```



Performances des arbres AVL

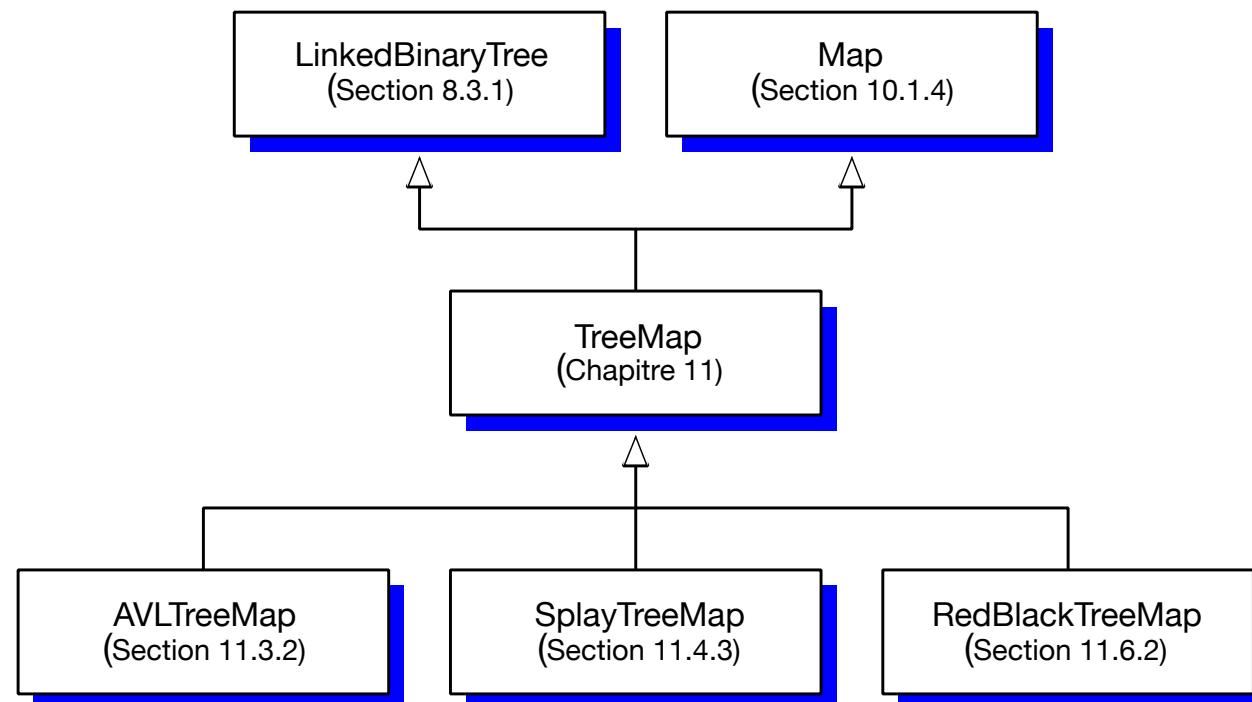
- Une restructuration est dans $O(1)$
 - On utilise un arbre binaire lié.
- La recherche prend $O(\log n)$
 - La hauteur d'un AVL est dans $O(\log n)$; aucune restructuration nécessaire
- L'insertion est dans $O(\log n)$
 - La recherche est dans $O(\log n)$
 - Restructurer l'arbre et maintenir les valeurs des hauteurs est dans $O(\log n)$
- La suppression est dans $O(\log n)$
 - La recherche est dans $O(\log n)$
 - Restructurer l'arbre et maintenir les valeur des hauteurs est dans $O(\log n)$

Data structure	Insertion (sec.)	Search (sec.)	Delete (sec.)
Sorted List	$O(n)$	$O(\log n)$	$O(n)$
1M	50.1	12.2	36.7
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$
1M	26.2	19.0	18.6
ChainHashMap	exp. $O(1)$	exp. $O(1)$	exp. $O(1)$
1M	11.3	3.1	3.4
ProbeHashMap	exp. $O(1)$	exp. $O(1)$	exp. $O(1)$
1M	8.0	3.5	3.7
AVLTreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$
1M	74.3	60.3	66.0



Deux prochaines sections

Splay, 2,4 (en introduction aux RedBlack) et RedBlack



Arbres Splay sont des arbres binaires de recherche

- Règles des arbres Splay :
 - Pas de limite logarithmique liée à la hauteur de l'arbre
 - Pas de hauteurs supplémentaires, d'équilibre ou d'autres données auxiliaires
 - L'opération **bouge-à-la-racine** appelle **splaying** à chaque insertion, suppression et recherche
 - Les éléments les plus fréquemment consultés restent plus près de la racine, ce qui réduit leurs temps de recherche
 - Étonnamment, cela assure un temps amorti logarithmique sur toutes les opérations

Splaying...

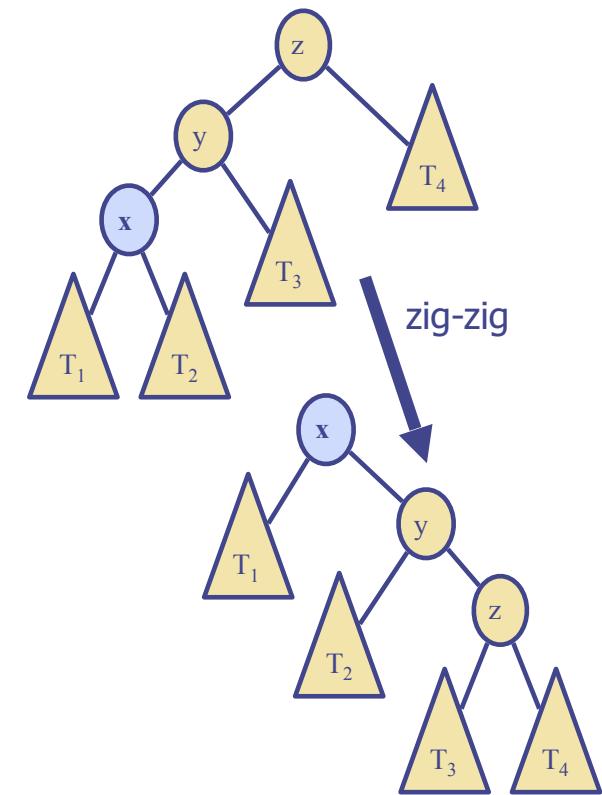
Étant donné un nœud x dans un ABR T , nous **splayons** x en le déplaçant à la racine de T à travers une séquence de restructurations.

On va s'arranger pour que les restructurations effectuées, pas n'importe quelles, amènent x à la racine.

La restructuration spécifique à effectuer pour déplacer x vers le haut dépend des positions relatives de x , de son parent y et (s'il existe) du grand-parent z .

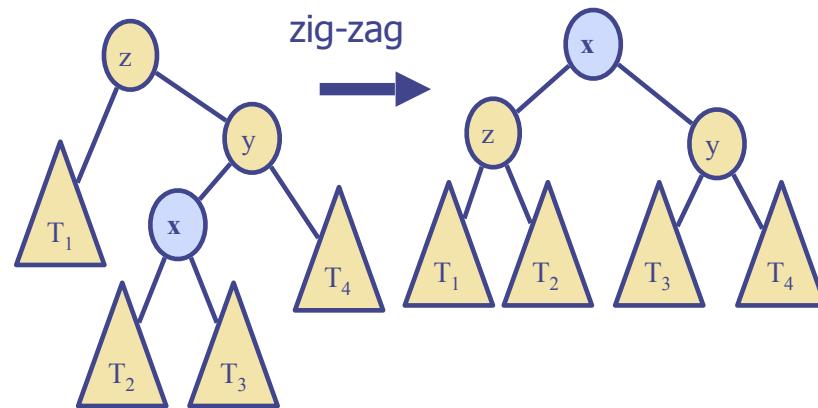
Il y a 3 cas à considérer :

zig-zig : Le nœud x et son parent y sont tous les deux des enfants gauches ou tous les deux des enfants droits (deux cas symétriques). Nous faisons monter x en faisant de y un enfant de x et z un enfant de y , ce qui maintient la relation d'ordre des nœuds dans T .



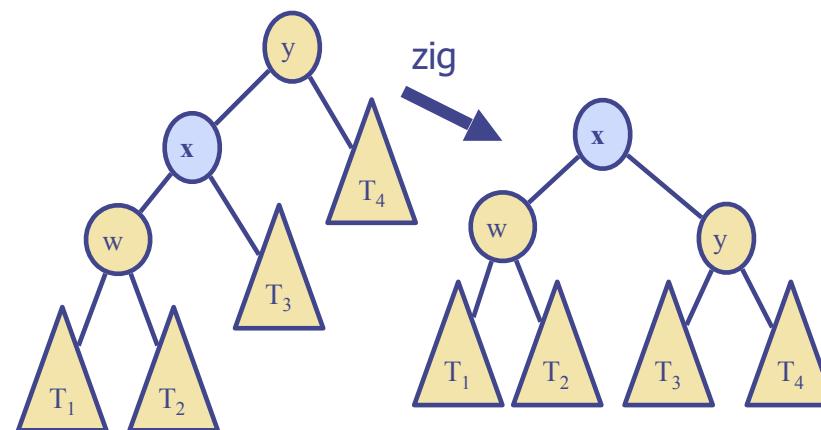
Splaying...

zig-zag : L'un des nœuds x et y est un enfant gauche et l'autre un enfant droit (deux cas symétriques). Nous faisons monter x , en faisant y et z des enfants de x , ce qui maintient la relation d'ordre des nœuds de T .



Splaying...

zig : Le noeud x n'a pas de grand-parent. Dans ce cas, nous effectuons une seule rotation pour faire monter x par rapport à y , en faisant de y un enfant de x , ce qui préserve la relation d'ordre des nœuds dans T .



Coût du splay

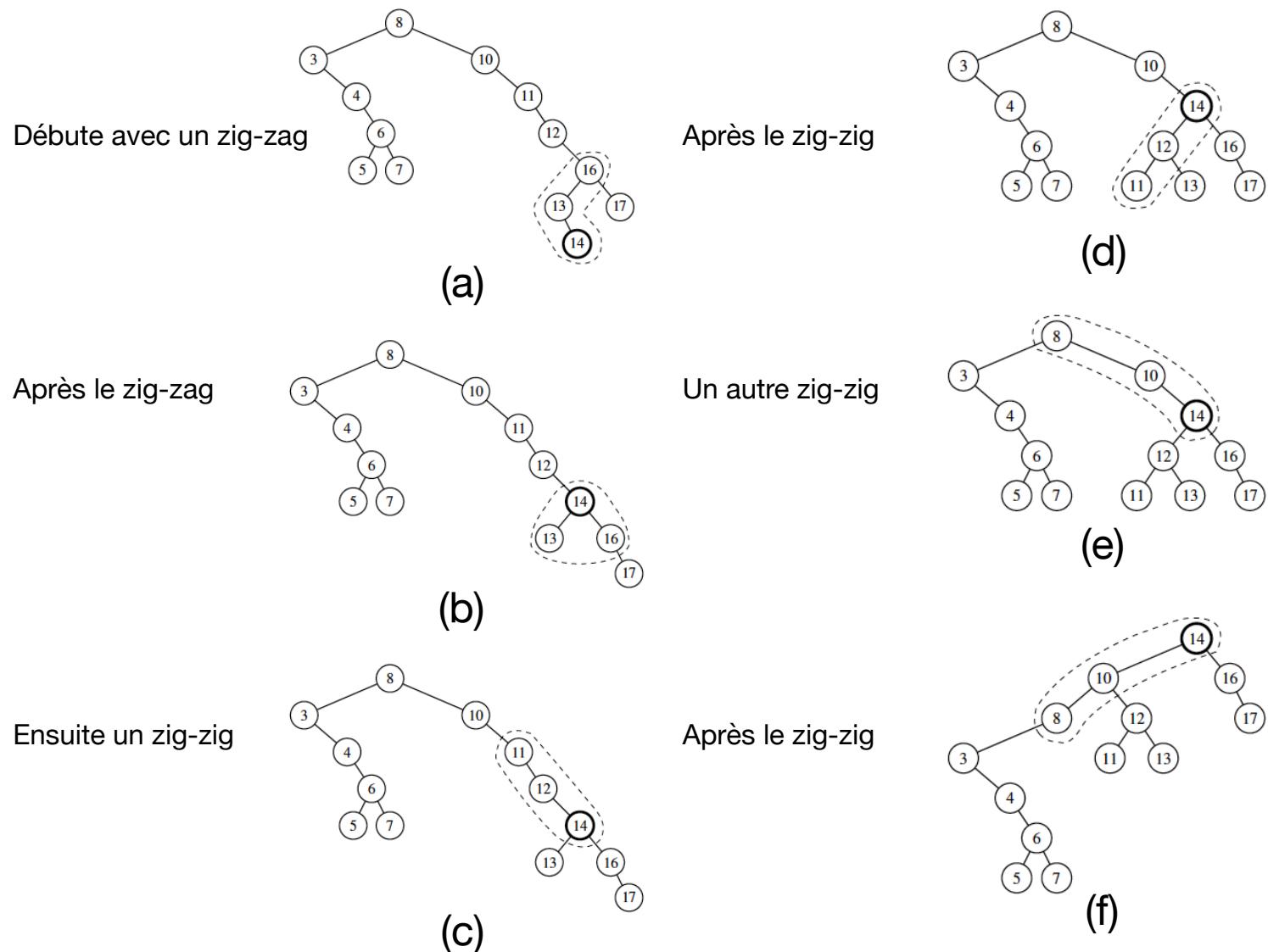
Un arbre Splay est un arbre binaire de recherche dans lequel on **splay** un nœud lorsqu'il est accédé (pour une recherche ou pour une mise à jour)

Le coût de **splayer** est dans $O(h)$, où h est la hauteur de l'arbre, qui est ici $O(n)$ en pire cas

On effectue $O(h)$ rotations (pire cas lorsque le noeud accédé est à hauteur h), chacune étant dans $O(1)$

Exemple : Splayer le noeud 14

Nous effectuons des ***zig-zig*** ou ***zig-zag*** lorsque x a un grand-parent, et nous effectuons des ***zig*** lorsque x a un parent, mais pas de grand-parent.
Splayer consiste à répéter ces restructurations de x jusqu'à ce que x devienne la racine.

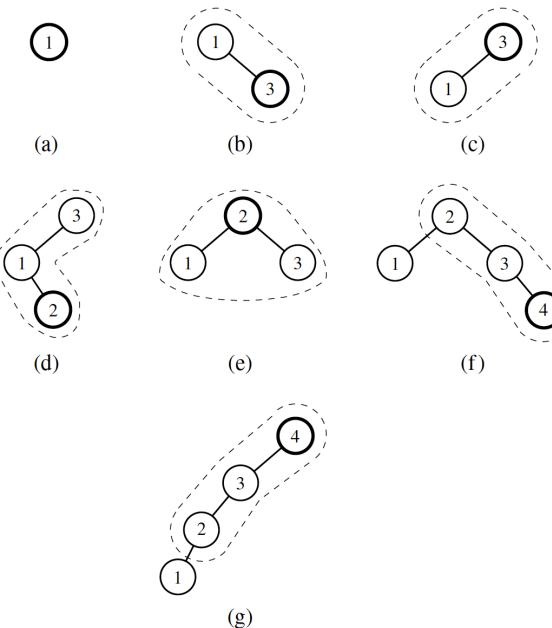
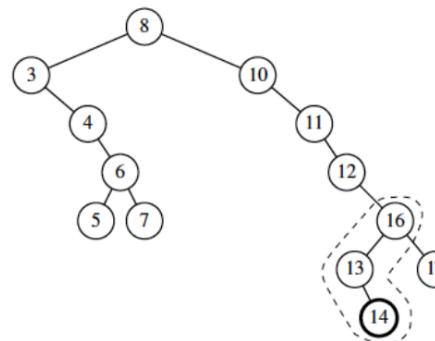


Quand et quel noeud splay ?

Recherche : Lors de la recherche de la clé k , si k est trouvée à la position p , on **splay** p , sinon on fait monter la position de la feuille où s'arrête la recherche.

Dans l'exemple précédent, le **splay** est effectué lorsque la recherche réussie pour 14, ou si on avait cherché sans succès la clé 15.

Insertion : Lors de l'insertion d'une clé k , on splay le nœud interne nouvellement créé, là où il est inséré.



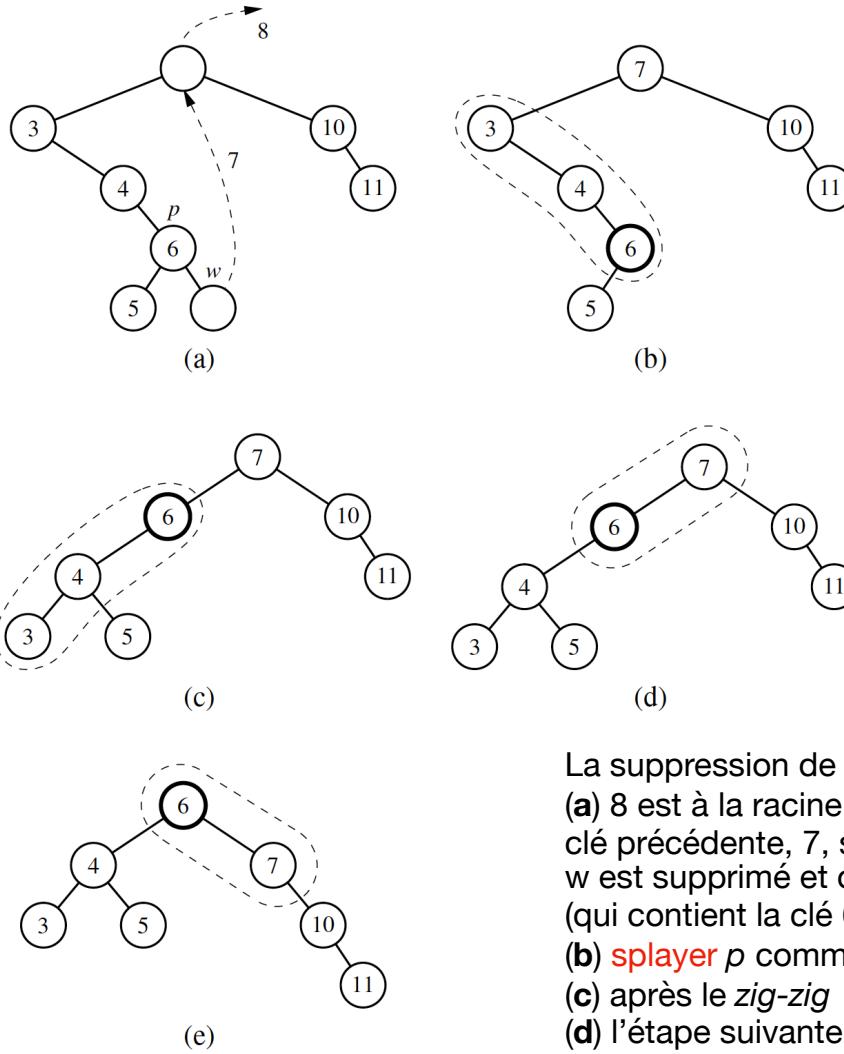
Une séquence d'insertions :

- (a) arbre initial
- (b) après l'insertion de 3 avant le zig
- (c) après **splayer** 3
- (d) après l'insertion de 2 avant le zig-zag
- (e) après **splayer** 2
- (f) après l'insertion de 4 avant le zig-zig
- (g) après **splayer** 4

Quand et quel noeud splayer ?

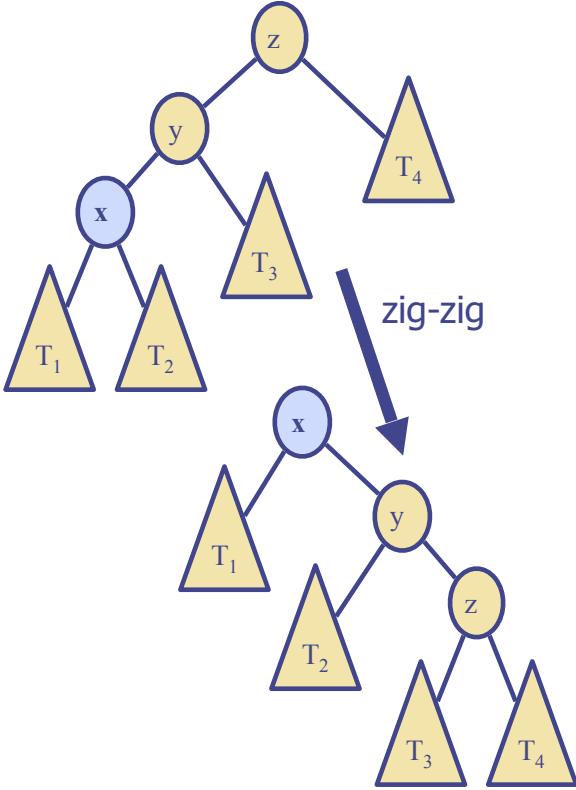
Suppression : Lors de la suppression d'une clé k , nous **splayons** la position du parent du nœud supprimé.

Rappelons l'algorithme de suppression dans un ABR : le nœud supprimé peut être celui qui contient k , ou un nœud avec une clé de remplacement.



La suppression de 8 :

- (a) 8 est à la racine, on le remplace par la clé précédente, 7, stockée au noeud w . w est supprimé et on splay son parent, p , (qui contient la clé 6)
- (b) **splayer** p commence avec un *zig-zig*
- (c) après le *zig-zig*
- (d) l'étape suivante est un *zig*
- (e) après le *zig*



```

from TreeMap import TreeMap

#classe SplayTreeMap
class SplayTreeMap( TreeMap ):

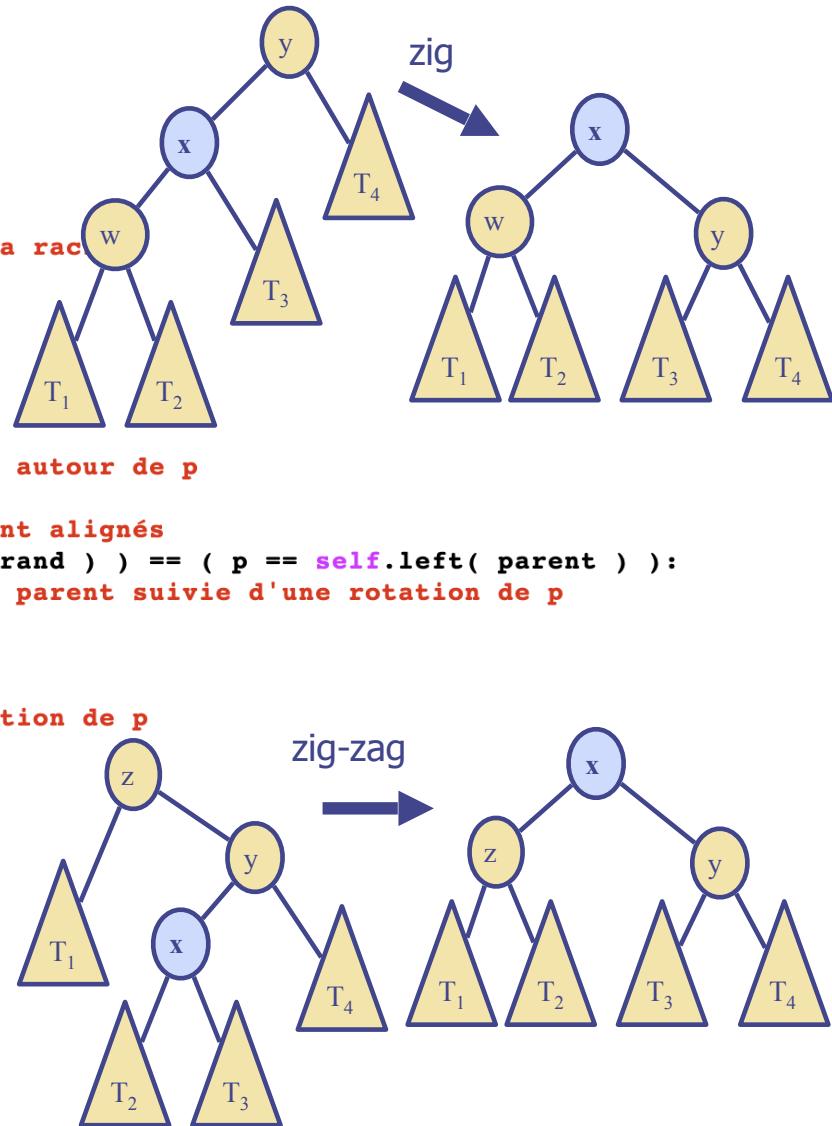
    #opération splay d'un noeud p
    def _splay( self, p ):
        #on splay jusqu'à ce que p soit la racine
        while p != self.root():
            #identification du cas
            parent = self.parent( p )
            grand = self.parent( parent )
            #pas de grand-parent => zig
            if grand is None:
                #cas zig, rotation simple autour de p
                self._rotate( p )
            #enfant, parent et grand-parent alignés
            elif ( parent == self.left( grand ) ) == ( p == self.left( parent ) ):
                #cas zig-zig, rotation du parent suivie d'une rotation de p
                self._rotate( parent )
                self._rotate( p )
            else:
                #cas zig-zag, double rotation de p
                self._rotate( p )
                self._rotate( p )

    #rebalancement après insertion
    def _rebalance_insert( self, p ):
        self._splay( p )

    #rebalancement après suppression
    def _rebalance_delete( self, p ):
        if p is not None:
            self._splay( p )

    #rebalancement après accession
    def _rebalance_access( self, p ):
        self._splay( p )

```



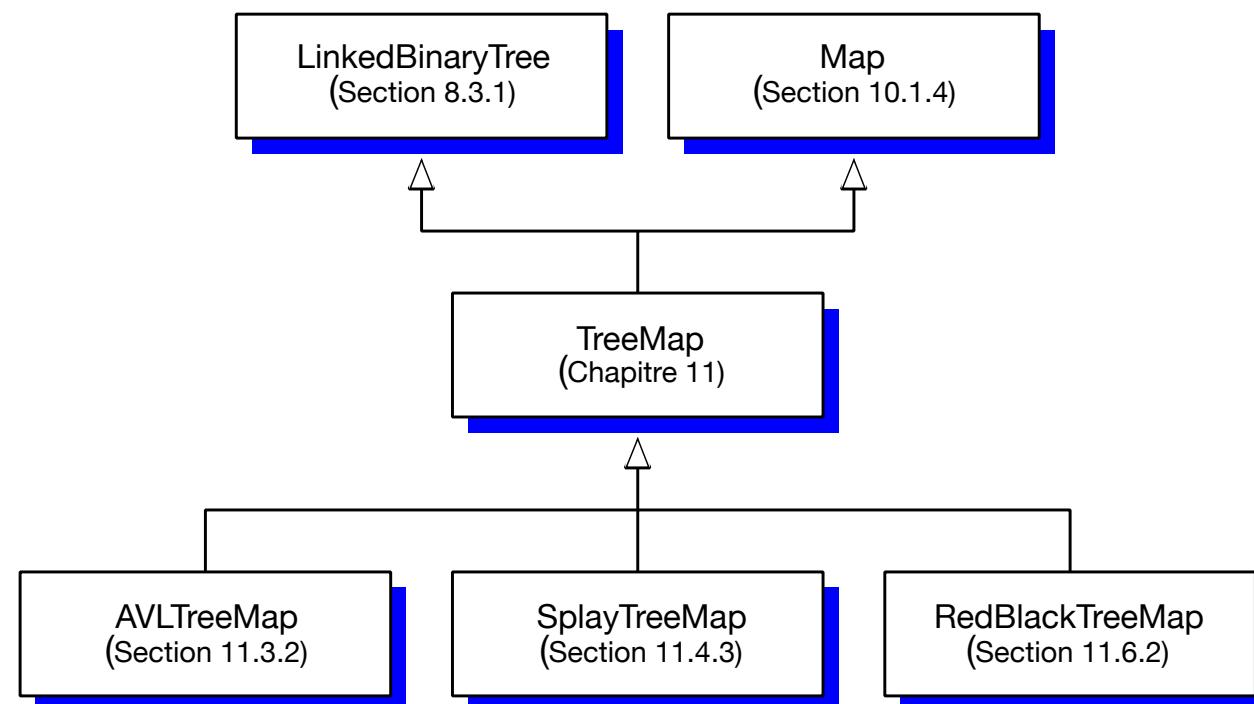
Performances des arbres Splay

- Le coût amorti de toute opération **splay** est dans $O(\log n)$ (voir livre pages 498-501)
- Les arbres splay peuvent s'adapter de manière à effectuer des recherches sur les éléments fréquemment demandés beaucoup plus rapidement qu'en $O(\log n)$, dans certains cas.

Data structure	Insertion (sec.)	Search (sec.)	Delete (sec.)
Sorted List	$O(n)$	$O(\log n)$	$O(n)$
1M	50.1	12.2	36.7
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$
1M	26.2	19.0	18.6
ChainHashMap	exp. $O(1)$	exp. $O(1)$	exp. $O(1)$
1M	11.3	3.1	3.4
ProbeHashMap	exp. $O(1)$	exp. $O(1)$	exp. $O(1)$
1M	8.0	3.5	3.7
AVLTreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$
1M	74.3	60.3	66.0
SplayTreeMap	$O(\log n)$ amorti	$O(\log n)$ amorti	$O(\log n)$ amorti
1M	197.8	203.7	195.9

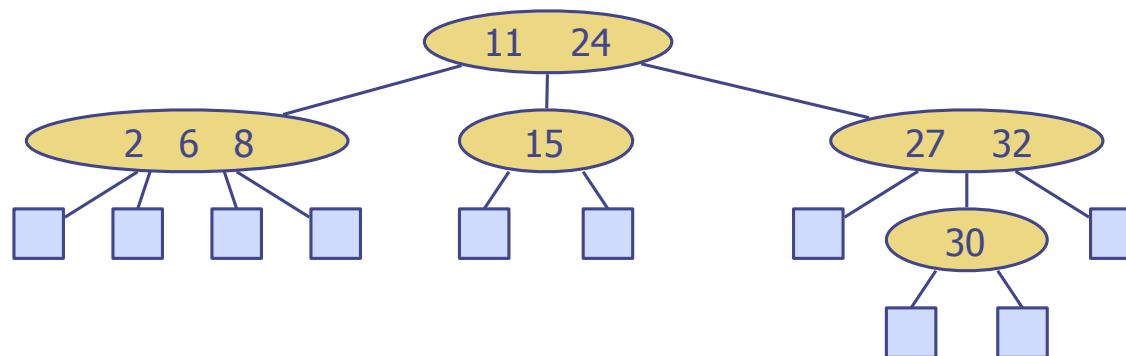


Prochaine section : RedBlack introduction par les (2,4)



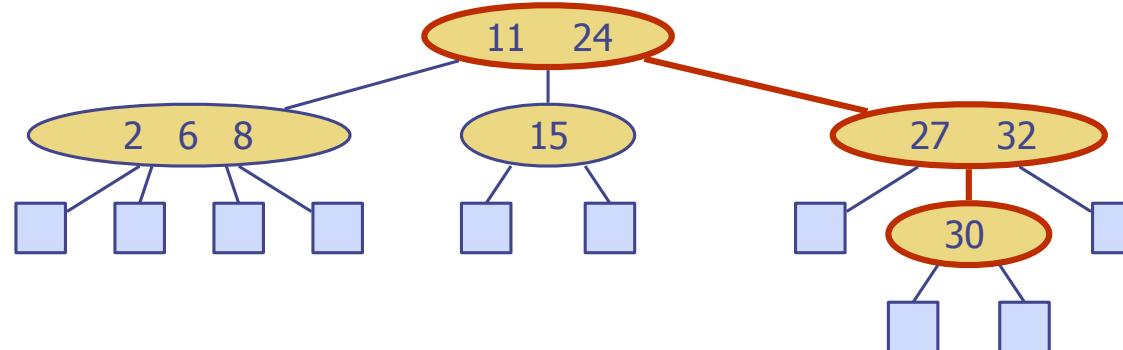
Arbres de recherche généralisés

- Un **arbre de recherche généralisé (2,d)** est un arbre ordonné tel que :
 - Chaque noeud interne possède entre deux et d enfants et stocke au maximum $d-1$ éléments clé-objet, (k_i, o_i)
 - Pour un noeud avec les enfants $v_1 v_2 \dots v_d$ stockant les clés $k_1 k_2 \dots k_{d-1}$
 - les clés du sous-arbre de v_1 sont inférieures à k_1
 - les clés du sous-arbre de v_i sont entre k_{i-1} et k_i ($i = 2, \dots, d-1$)
 - les clés du sous-arbre de v_d sont supérieures à k_{d-1}
 - Les feuilles ne stockent aucun élément



Chercher

- Similaire à la recherche dans un arbre binaire de recherche
- A chaque nœud interne avec les enfants $v_1 v_2 \dots v_d$ et les clés $k_1 k_2 \dots k_{d-1}$
 - $k = k_i (i = 1, \dots, d-1)$: la recherche se termine avec succès
 - $k < k_1$: nous continuons la recherche dans l'enfant v_1
 - $k_{i-1} < k < k_i (i = 2, \dots, d-1)$: nous continuons la recherche dans l'enfant v_i
 - $k > k_{d-1}$: nous continuons la recherche dans l'enfant v_d
- Atteindre un nœud externe met fin à la recherche sans succès
- Exemple : recherche de 30

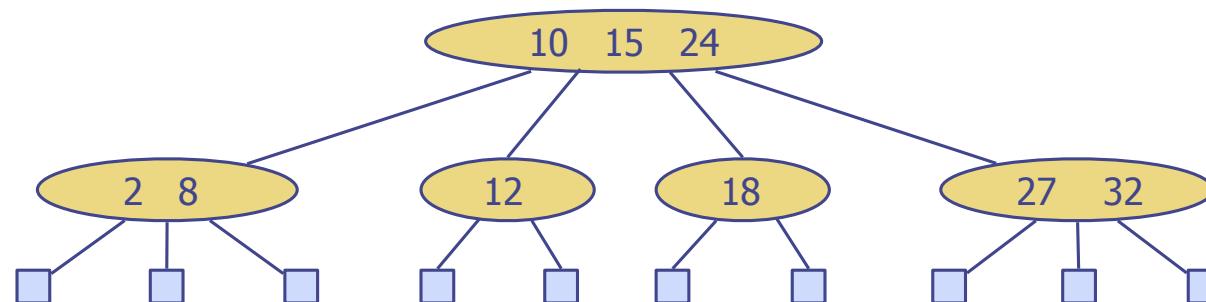


Arbres (2,4)

Un **arbre (2,4)** (aussi appelé arbre 2-4 ou arbre 2-3-4) est un arbre de recherche généralisé avec les propriétés suivantes :

- Propriété de taille : chaque noeud interne a au plus quatre enfants
- Propriété de profondeur : tous les nœuds externes ont la même profondeur

Selon le nombre d'enfants, un nœud interne est appelé soit un nœud-2, nœud-3 ou nœud-4.



Hauteur d'un arbre (2,4)

Théorème: Un arbre (2,4) stockant n éléments a une hauteur dans $O(\log n)$

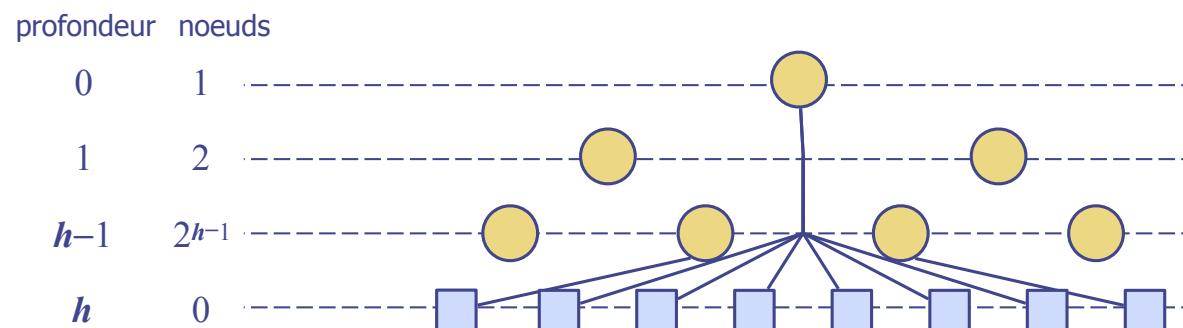
Preuve:

Soit h la hauteur d'un arbre (2,4) avec n éléments

Puisqu'il y a au moins 2^i noeuds à la profondeur $i = 0, \dots, h-1$ et aucun élément en profondeur h , on a $n \geq 1 + 2 + 4 + \dots + 2^{h-1} = 2^h - 1$

Ainsi, $h \leq \log(n + 1)$

La recherche dans un arbre (2,4) de n éléments est donc dans $O(\log n)$ -temps

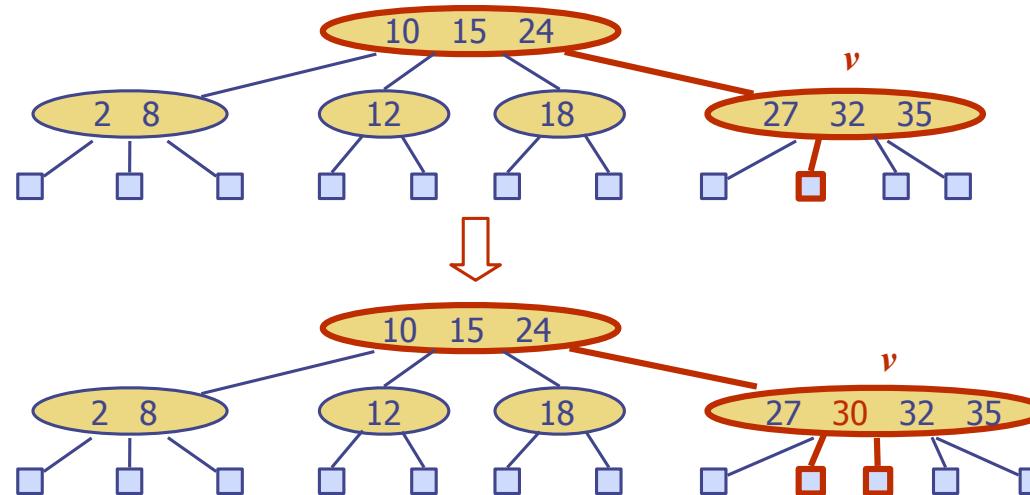


Insertion

Nous insérons un nouvel élément, (k, o) , au parent v de la feuille atteinte en cherchant k

- Nous préservons la propriété de profondeur mais cela peut provoquer un débordement (c'est-à-dire que le noeud v peut devenir un noeud-5)

Exemple : l'insertion de la clé 30 provoque un débordement !



Débordement et partage/scission (split)

Nous gérons un débordement sur un noeud-5, v , avec une opération de partage :

soit $v_1 \dots v_5$ les enfants de v et $k_1 \dots k_4$ les clés de v

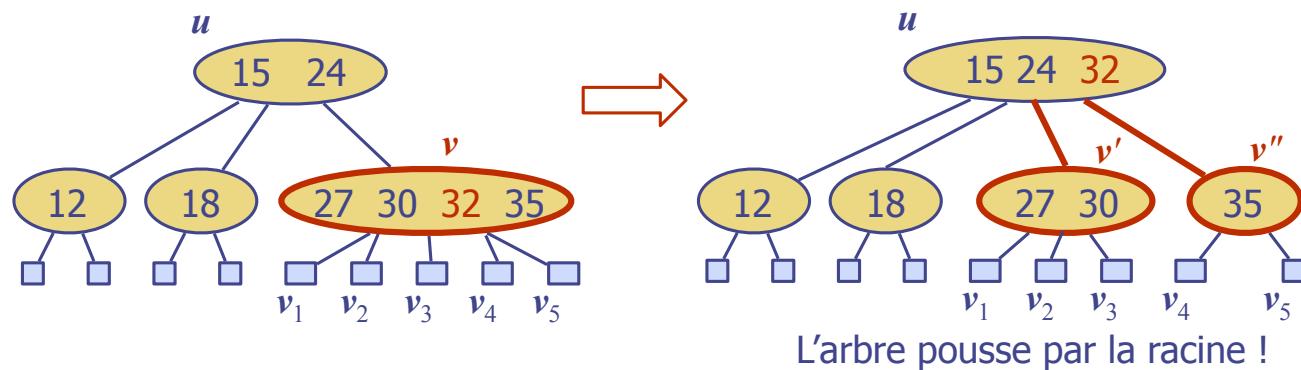
le noeud v est remplacé par les noeuds v' et v''

v' est un nœud-3 avec les clés $k_1 k_2$ et les enfants $v_1 v_2 v_3$

v'' est un nœud-2 avec la clé k_4 et les enfants $v_4 v_5$

la clé k_3 est insérée dans le parent u de v (une nouvelle racine peut ainsi être créée)

Le débordement peut se propager au nœud parent



Analyse de l'insertion

Algorithme insérer(k, o)

1. Nous recherchons la clé k pour localiser le noeud d'insertion v
2. Nous ajoutons la nouvelle entrée (k, o) au nœud v
3. tantque débordement(v)
 - si estRacine(v)
 - créer une nouvelle racine vide au-dessus de v
 - v = scinder(v)

Soit T un arbre (2,4) avec n éléments

T a une hauteur dans $O(\log n)$

L'étape 1 est dans $O(\log n)$ -temps car nous visitons $O(\log n)$ noeuds

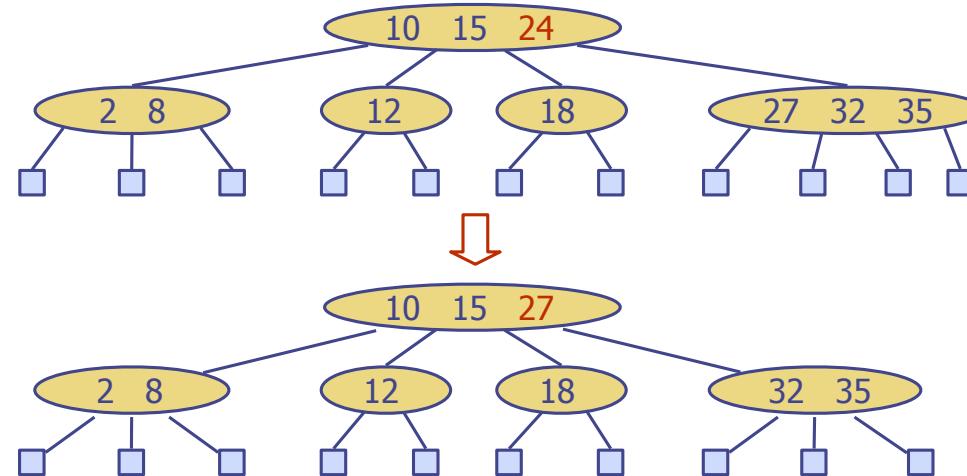
L'étape 2 est dans $O(1)$ -temps

L'étape 3 est dans $O(\log n)$ -temps car chaque partage est dans $O(1)$ -temps et nous effectuons dans $O(\log n)$ partages

Ainsi, une insertion dans un arbre (2,4) est dans $O(\log n)$ -temps

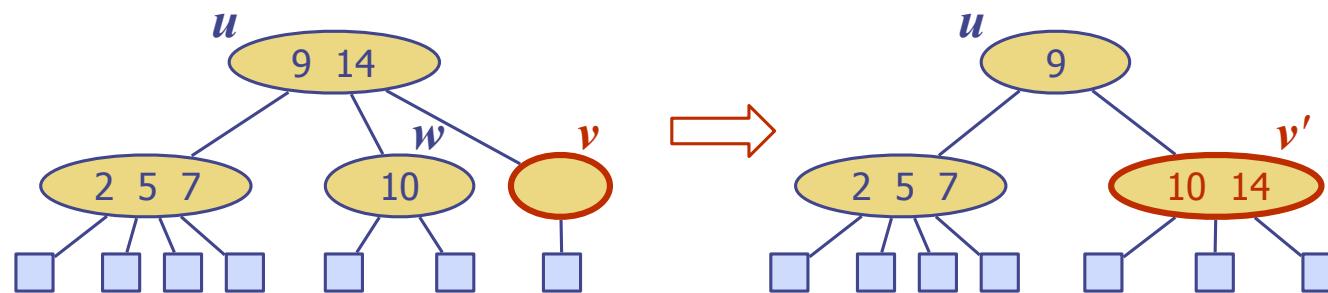
Suppression

- On réduit la suppression à un noeud avec des feuilles
- Sinon, on remplace l'élément par son successeur (ou, de manière symétrique, par son prédécesseur) et on supprime ce dernier
- Exemple : pour **supprimer** la clé 24, on la remplace par 27



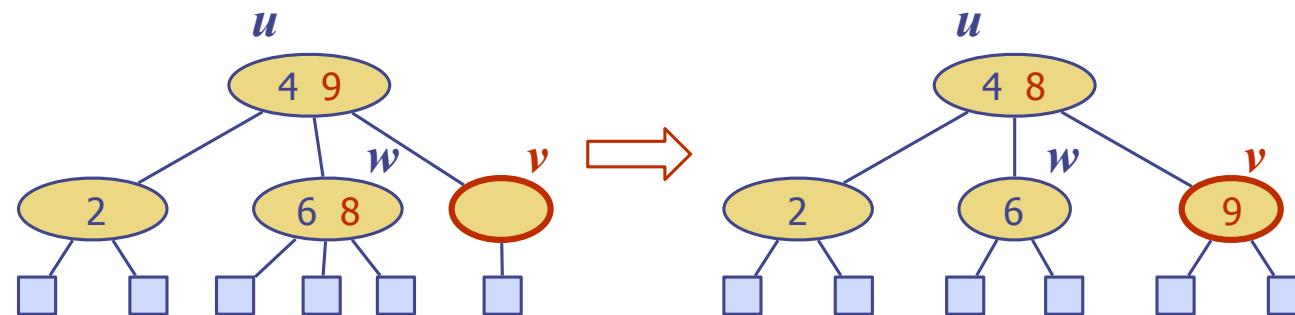
Dépassemement et fusion

- La suppression d'un nœud v peut provoquer un dépassement de capacité, où le nœud v devient un nœud-1 sans clé
- Pour gérer un dépassement au noeud v de parent u , on considère deux cas :
- **Cas 1** : présence d'un congénère noeud-2 adjacent à v
 - Opération de fusion : on fusionne v avec le congénère w et déplace une entrée de u vers le noeud fusionné v'
 - Après la fusion, le dépassement peut se propager au parent



Dépassemement et transfert

- **Cas 2** : un congénère noeud-3 ou noeud-4, w , adjacent à v
- Opération de transfert :
 1. on déplace un enfant de w à v
 2. on déplace un élément de u à v
 3. on déplace un élément de w à u
- Après un transfert, aucun dépassement n'est possible



Analyse de la suppression

- Soit T un arbre (2,4) de n éléments
 - T a une hauteur dans $O(\log n)$
- Dans une opération de suppression :
 - On visite $O(\log n)$ noeuds pour localiser le noeud de suppression
 - On traite un dépassement avec une série de fusions dans $O(\log n)$, ou d'au plus un transfert
 - Chaque fusion et transfert est dans $O(1)$ -temps
- Ainsi, supprimer un élément d'un arbre (2,4) est dans $O(\log n)$ -temps

Comparaison des implémentations de l'ADT Map jusqu'ici

	Recherche	Insertion	Suppression	Notes
Table de hachage	1 espéré	1 espéré	1 espéré	<ul style="list-style-type: none">○ pas de parcours dans l'ordre (pas ADT Map)○ facile à implémenter
SkipList	$\log n$ haute probabilité	$\log n$ haute probabilité	$\log n$ haute probabilité	<ul style="list-style-type: none">○ insertion aléatoire○ facile à implémenter
Arbres AVL et (2,4)	$\log n$ pire cas	$\log n$ pire cas	$\log n$ pire cas	<ul style="list-style-type: none">○ complexe à implémenter (à cause de la restructuration)