Arbre binaire de recherche

TA table de symboles

table de symboles = dictionnaire = tableau associatif
= *symbol table* = *dictionary* = *map*

Type abstrait pour recherche par clé

Objets : ensembles d'éléments avec clés
typiquement : clés comparables (abstraction : nombres naturels)

⋆ search(k) : recherche d'un élément à clé k ← peut être fructueuse ou infructueuse

Si dictionnaire modifiable («dynamique») :
⋆ insert(x) : insertion de l'élément x (clé+info)
⋆ delete(k) : supprimer élément avec clé k
Avec clés uniques : search(k) retourne l'élément x avec clé k si un tel existe, ou null sinon ; insert(x) *remplace* l'élément avec la même clé, si un tel est déjà présent dans l'ensemble.

Structures jusqu'ici

- ⋆ **tableau non-trié** ou **liste chaînée** : recherche séquentielle
  search en $\Theta(n)$ au pire (même en moyenne)
- ⋆ **tableau trié** : recherche dichotomique
  search en $\Theta(\log n)$ au pire ;
  mais insertion/suppression en $\Theta(n)$ au pire
- ⋆ **tableau de hachage** :
  opérations en $\Theta(1)$ en moyenne mais $\Theta(n)$ au pire

Arbre binaire

nœuds avec clé et valeur

```
class TreeNode<K,V> {
   TreeNode left, right;
   TreeNode parent;
   K key;
   V value;
   ...
}
```

- ⋆ x.left et x.right pour les enfants de x

  (null si l'enfant est un nœud externe)

- ⋆ x.parent pour le parent de x

  (null à la racine)

  en général, il est possible d'implémenter les ABRs sans stocker les parents

- ⋆ x.key pour la clé d'un nœud interne x

  (en général, un entier dans nos discussions)

Arbre binaire de recherche

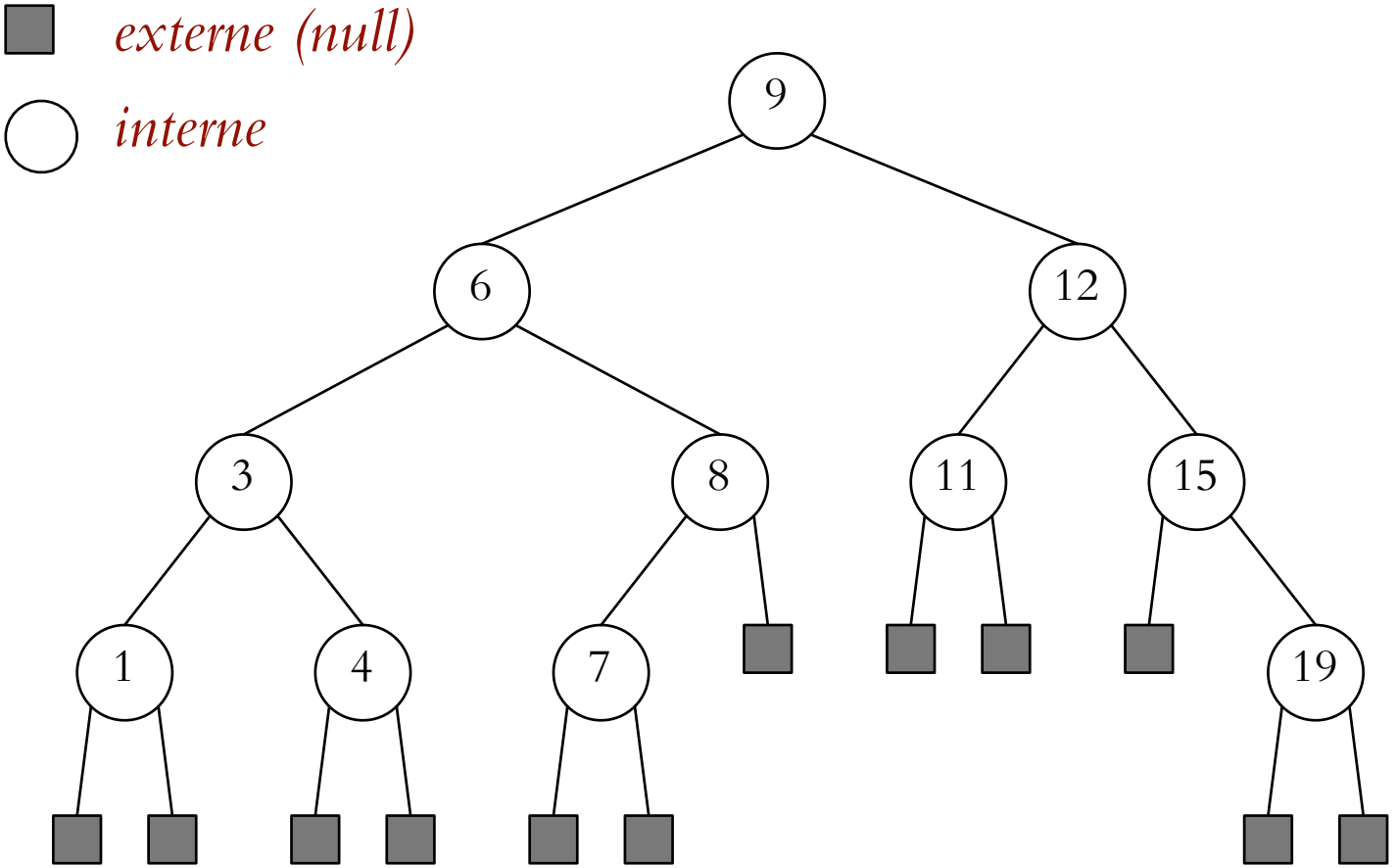
Déf. Un arbre binaire est un arbre de recherche ssi les nœuds sont énumerés lors d'un parcours infixe en ordre croissant de clés.

```
   Algo PARCOURS-INFIXE(x)
1  if x ≠ null then
2        PARCOURS-INFIXE(x.left)
3        «visiter» x
4        PARCOURS-INFIXE(x.right)
```

Thm. Soit x un nœud interne dans un arbre binaire de recherche. Si $y \neq x$ est un nœud interne dans le sous-arbre gauche de x, alors $y.\mathsf{key} < x.\mathsf{key}$. Si $y \neq x$ est un nœud interne dans le sous-arbre droit de x, alors $y.\mathsf{key} > x.\mathsf{key}$.

Preuve Le parcours infixe visite les nœuds du sous-arbre gauche avant, et les nœuds du sous-arbre droit après la racine.

Arbre binaire de recherche — exemple



Arbre binaire de recherche (cont)

À l'aide d'un arbre de recherche, on peut implémenter une table de symboles d'une manière très efficace.

Opérations naturellement supportées par la structure :

⋆ **recherche** par clé
⋆ **insertion**
⋆ **suppression**
⋆ recherche de **min** et **max**, **successeur** et **prédecesseur**
⋆ et d'autres (généralement avec variables additionnelles aux nœuds — p.e., taille du sous-arbre)

Min et max

```
class TreeNode<K,V> {
...
   K min() // clé minimale dans le sous-arbre
   {
      if (left==null){ return this.key;} else {return left.min(); }
   }
```

Algo MINNODE(x) *// cherche nœud avec clé minimale*

1 $y \leftarrow$ null
2 **while** $x \neq$ null **do** $y \leftarrow x$; $x \leftarrow x$.left
3 **return** y

MAXNODE(x) *// cherche nœud maximal*

1 $y \leftarrow$ null
2 **while** $x \neq$ null **do** $y \leftarrow x$; $x \leftarrow$ right
3 **return** y

Recherche

```
class TreeNode<K,V> {
...
   V search(K k) // recherche de clé k
   {
      if (this.key.equals(k))
         return this.value;
      else if (this.key.compareTo(k)>0)  // k inférieur à this.key
         return left==null?null:left.search(k);
      else                                // k supérieur à this.key
         return right==null?null:right.search(k);
   }
```

SEARCHNODE(x, k) // *cherche nœud avec clé k dans le sous-arbre de x*

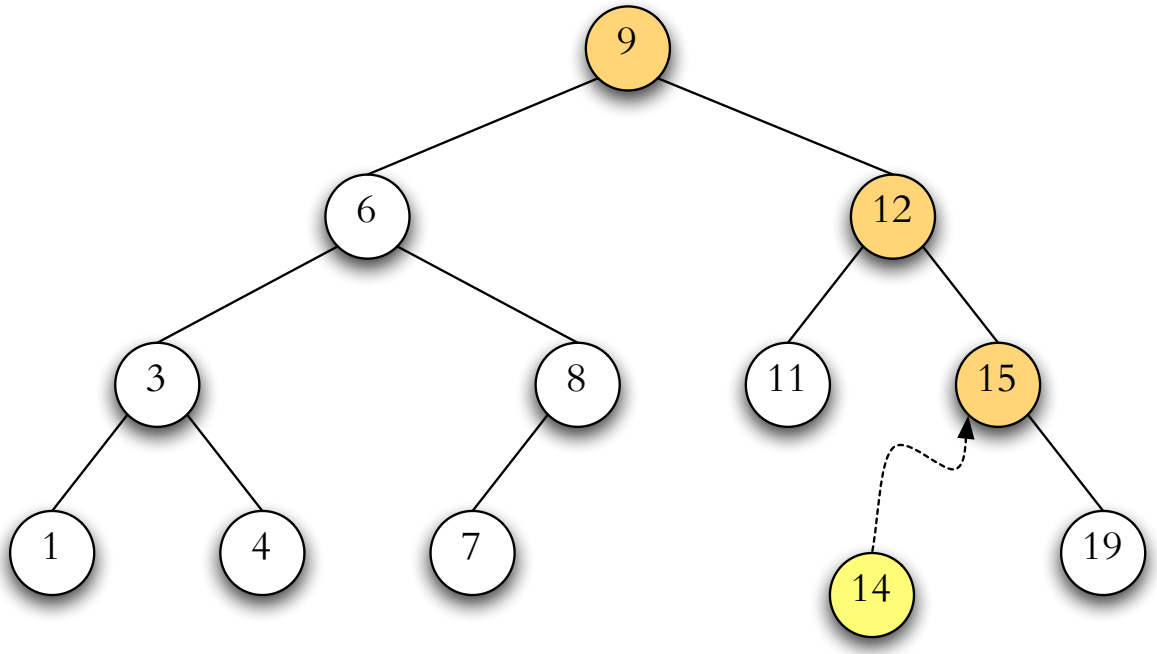
S1 **while** $x \neq$ null && $k \neq x$.key **do**
S2 **if** $k < x$.key
S3 **then** $x \leftarrow x$.left
S4 **else** $x \leftarrow x$.right
S5 **return** x // null ou le nœud avec clé k

☞s'il n'y a pas de variables x.parent, on peut mettre les ancêtres visités sur une pile ou une liste pendant la descente dans l'arbre

Insertion

On veut insérer une clé k

Idée : comme en SEARCH, on trouve la place pour k
(enfant gauche ou droit manquant)



insertion de «14»

Insertion (cont.)

insertion (pas de clés dupliquées !)

```
    INSERT(k, v)                                          // insère la clé k dans l'arbre
I1  x ← root
I2  if x = null then root ← nouveau nœud(k, v) ; return
I3  loop                                  // boucler : conditions d'arrêt testées dans le corps
I4      if k = x.key then x.value = v return                              // clé existante
I5      if k < x.key
I6      then if x.left = null                                  // y devient enfant gauche
I7          then y ← nouveau nœud(k, v) ; x.left ← y; y.parent ← x ; return
I8          else x ← x.left
I9      else if x.right = null                                 // y devient enfant droit
I10         then y ← nouveau nœud(k, v) ; x.right ← y; y.parent ← x ; return
I11         else x ← x.right
```

Insertion — récursive

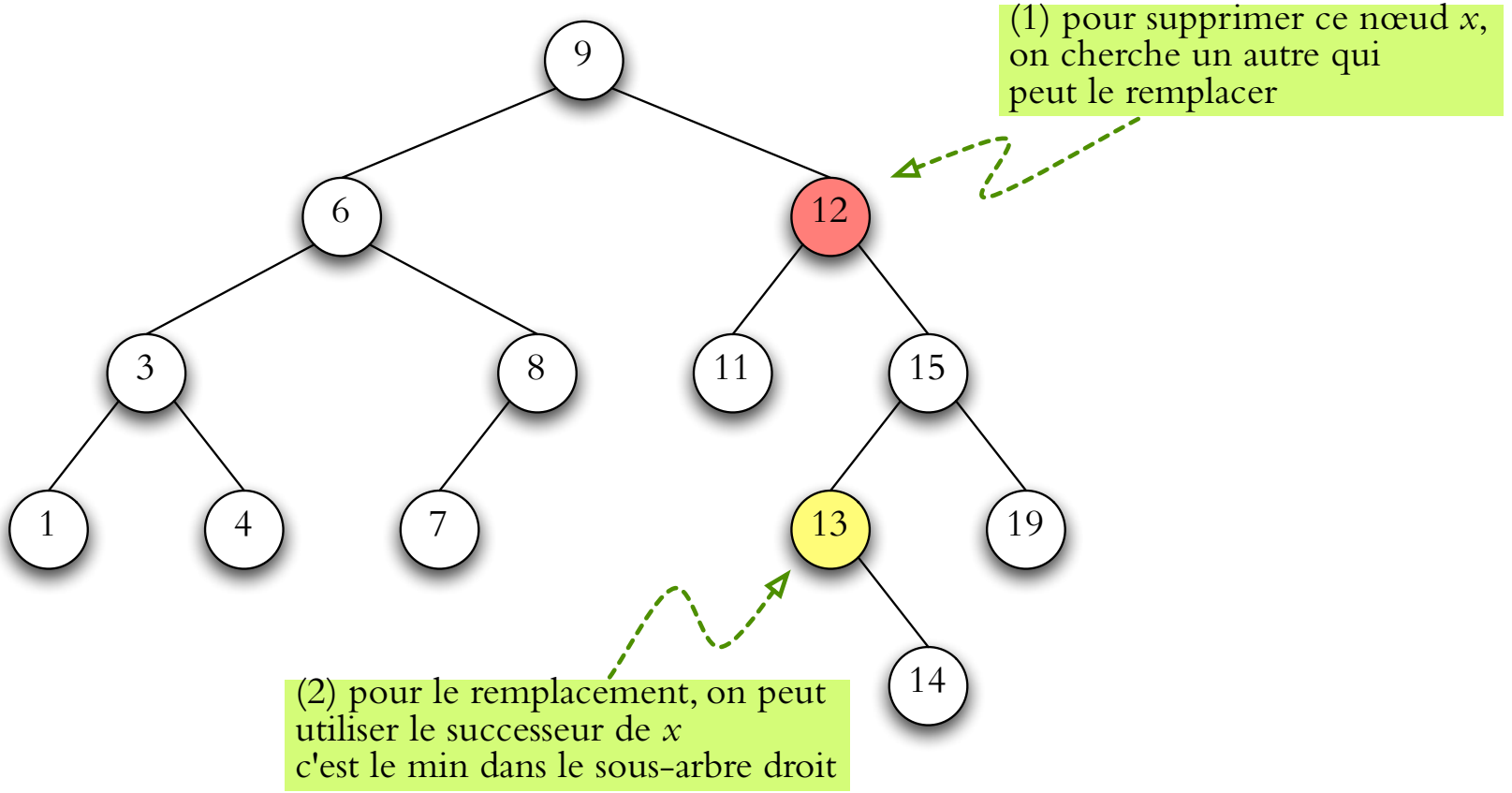
```
public class ABR<K>{
private TreeNode root;

public void insert(K key)
{
    TreeNode N = new TreeNode(key);
    if (root==null) { root = N; return;} // première insertion
    else N.insertBelow(root);
}
class TreeNode<K> {
   TreeNode insertBelow(TreeNode P)
   {
       if (P == null) return this;
       if (key.compareTo(P.key)<0)     // ==0 interdit dans cette versi
             { P.left = insertBelow(P.left); P.left.parent=P; }
       else
             { P.right = insertBelow(P.right); P.right.parent=P; }

       return P;
   }
```

Suppression

Suppression d'un nœud x

0. triviale si x n'a **pas d'enfants** internes : faire $\boxed{x.\mathsf{parent.left} \leftarrow \mathsf{null}}$ si x est l'enfant gauche de son parent, ou $\boxed{x.\mathsf{parent.right} \leftarrow \mathsf{null}}$ si x est l'enfant droit
1. facile si x a seulement **1 enfant** :
   faire $\boxed{x.\mathsf{parent.left} \leftarrow x.\mathsf{right}\,;\, x.\mathsf{right.parent} \leftarrow x.\mathsf{parent}}$ si x a un enfant droit et x est l'enfant gauche de son parent (il y a 4 cas en total dépendant de la position de x et celle de son enfant)
2. un peu plus compliqué si x a **2 enfants** : on trouve d'abord remplacement (successeur ou prédecesseur dans le parcours infixe)

Suppression — deux enfants



Lemme Le nœud avec la valeur minimale dans le sous-arbre droit de x n'a pas d'enfant gauche.

Opérations — efficacité

Dans un arbre binaire de recherche de hauteur h :

- ☞ min/max prend $O(h)$
- ☞ recherche prend $O(h)$
- ☞ insertion prend $O(h)$
- ☞ suppression prend $O(h)$

Hauteur de l'arbre

Toutes les opérations prendent $O(h)$ dans un arbre de hauteur h.

Arbre binaire complet : $2^h - 1$ nœuds dans un arbre de hauteur h, donc hauteur $h = \lceil \lg(n+1) \rceil$ pour n nœuds est possible.

Insertions consécutives de $1, 2, 3, \ldots, n$ mènent à un arbre avec hauteur $h = n$.

Est-ce qu'il est possible d'assurer que $h \in O(\log n)$?

Hauteur de l'ABR

Réponse 1 [randomisation] : la hauteur est de $O(\log n)$ *en moyenne* (permutations aléatoires de $\{1, 2, \ldots, n\}$)

Réponse 2 [optimisation] : la hauteur est de $O(\log n)$ *en pire cas* pour beaucoup de genres d'arbres de recherche équilibrés : arbre AVL, arbre rouge-noir, arbre 2-3-4 (exécution des opérations est plus sophistiquée — mais toujours $O(\log n)$)

Réponse 3 [amortisation] : exécution des opérations est $O(\log n)$ en moyenne (coût amortisé dans séries d'opérations) pour des arbres *splay*

Performance moyenne

Déf. Un **ABR aléatoire** se construit en insérant les valeurs $1, 2, \ldots, n$ selon une permutation aléatoire, choisie à l'uniforme.

☞cette notion est tout à fait différente de celle d'une structure choisie à l'uniforme : les 6 permutations des clés $\{1, 2, 3, \}$ mènent à seulement 5 arbres possibles.

Thm.[Bruce Reed & Michael Drmota] La hauteur d'un ABR aléatoire sur n clés est

$$\mathbb{E}h = \alpha \lg n - \beta \lg \lg n + O(1)$$

en espérance, où

$$\alpha \approx 2.99 \quad \text{et} \quad \beta = \frac{3}{2 \lg(\alpha/2)} \approx 1.35.$$

La variance de la hauteur aléatoire est $O(1)$.

Profondeur moyenne

On peut analyser le cas moyen en regardant la profondeur moyenne d'un nœud dans un tel arbre de recherche aléatoire : le coût de chaque opération dépend de la profondeur du nœud accédé dans l'arbre.

Déf. Soit $D(n)$ la somme des profondeurs des nœuds dans un arbre de recherche aléatoire sur n nœuds.

On va démontrer que $\frac{D(n)}{n} \in O(\log n)$.
(Donc le temps moyen d'une recherche fructueuse est en $O(\log n)$.)

Performance moyenne (cont.)

Lemme. On a $D(0) = D(1) = 0$, et

$$D(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} \Big(D(i) + D(n-1-i) \Big)$$
$$= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} D(i).$$

Preuve. (Esquissé) $i+1$ est la racine, somme des profondeurs = (n-1)+somme des profondeurs dans le sous-arbre gauche +somme des profondeurs dans le sous-arbre droit. □

D'ici, comme l'analyse de la performance du tri rapide... $D(n) \sim 2 \ln n$.

(en fait, chaque ABR correspond à une exécution de tri rapide : pivot du sous-tableau comme la racine du sous-arbre)

Arbres équilibrés

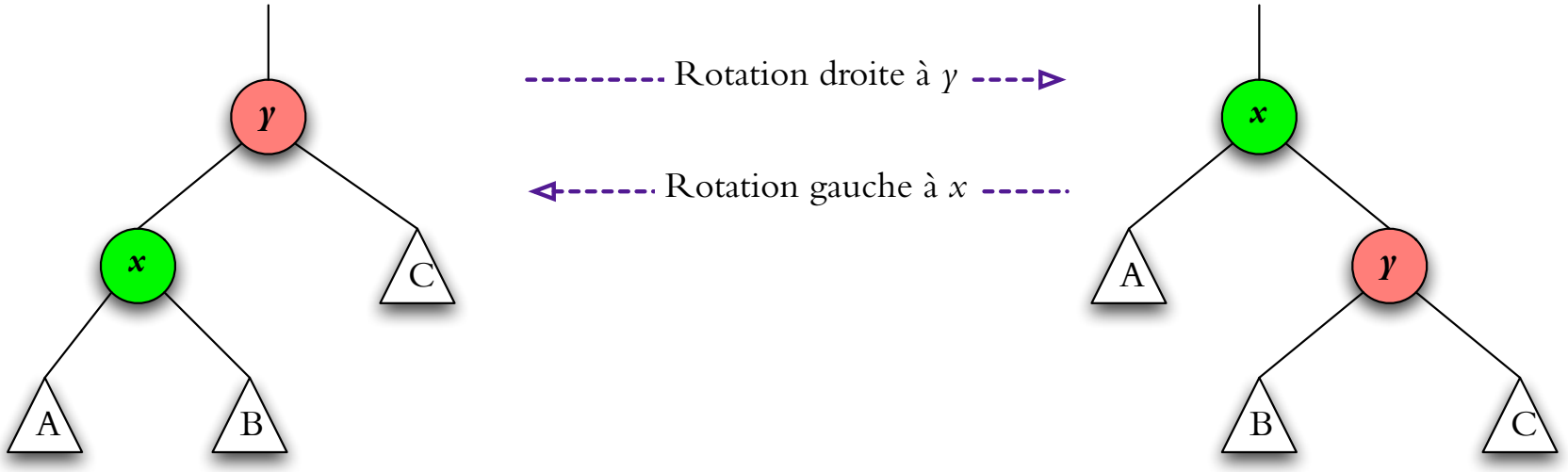
Arbre équilibré : hauteur de $O(\log n)$

on maintient une condition qui assure que les sous-arbres ne sont trop différents à aucun nœud

Si l'on veut maintenir une condition d'équilibre, il faudra travailler un peu plus à chaque (ou quelques) opérations. . . mais on veut toujours maintenir $O(\log n)$ par opération

Balancer les sous-arbres

Méthode : rotations (gauche ou droite) — préservent la propriété des arbres de recherche et prennent seulement $O(1)$



Arbres AVL

Il n'est pas possible de maintenir un arbre complet binaire avec $O(\log n)$ opérations, mais il suffit de relâcher la condition d'équilibre juste un petit peu

AVL : Adelson-Velsky et Landis (1962)

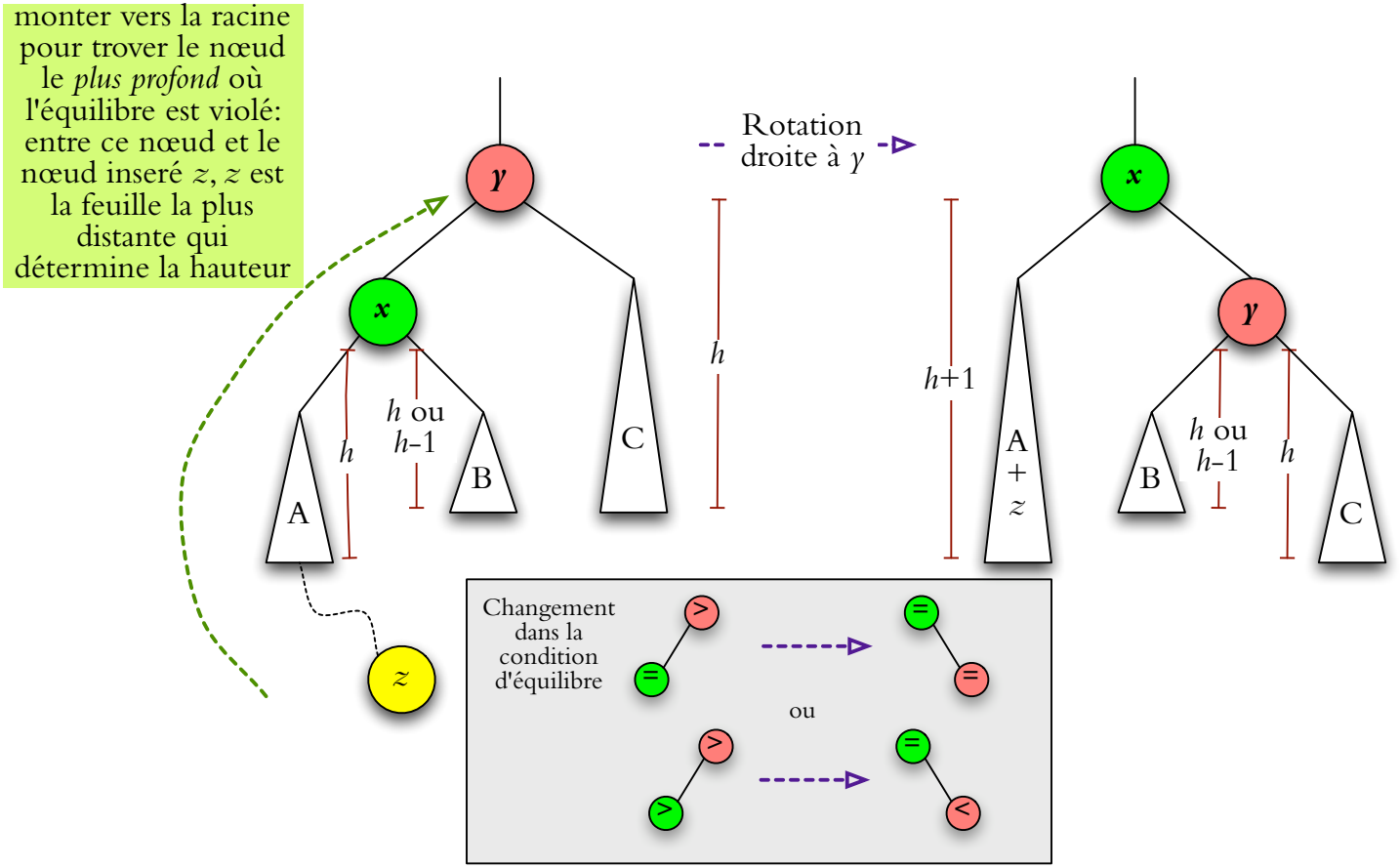
Déf. Un arbre binaire est un arbre AVL ssi à chaque nœud, la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit diffèrent par 1 au plus.

On doit stocker la différence de hauteur à chaque nœud interne.

Remarque. On peut calculer la hauteur de tous les nœuds en parcours post-fixe.

Insertion dans un arbre AVL

Insertion dans le sous-arbre gauche d'un enfant gauche : une rotation si nécessaire



(cas symétrique si sous-arbre droit d'un enfant droit ; autres cas plus compliqués)

Hauteur d'un arbre AVL

Soit $N(h)$ le nombre minimal de nœuds internes dans un arbre AVL de hauteur $h \geq 0$. On a $N(0) = 0$, $N(1) = 1$.

Lemme. Pour tout $h > 1$,

$$N(h) = N(h-1) + N(h-2) + 1$$
$$\Big(N(h) + 1\Big) = \Big(N(h-1) + 1\Big) + \Big(N(h-2) + 1\Big)$$

nombres Fibonacci : $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$

ici : $N(0) = F(2) - 1$, $N(1) = F(3) - 1$, donc

$$N(h) = F(h+2) - 1 \sim \frac{\phi^{h+2}}{\sqrt{5}}$$

avec $\phi = \frac{1+\sqrt{5}}{2}$

Hauteur d'un arbre AVL (borne)

on se sert de la borne $F(h) > \frac{\phi^h}{\sqrt{5}} - 1$:

$$N(h) = F(h+2) - 1 > \underbrace{\frac{\phi^2}{\sqrt{5}}}_{=c} \phi^h - 2$$

d'où on obtient la borne sur la hauteur de l'arbre AVL à n nœuds internes :

$$h < \log_\phi \frac{n+2}{c} = \frac{\lg(n+2) - \lg c}{\lg \phi} \sim \underbrace{\left(\lg \phi\right)^{-1}}_{=1.4404\cdots} \cdot \lg(n)$$

Arbres *splay*

Idée principale : rotations sans tests spécifiques pour l'équilibre

Quand on accède à nœud x, on performe des rotations sur le chemin de la racine à x pour monter x à la racine.

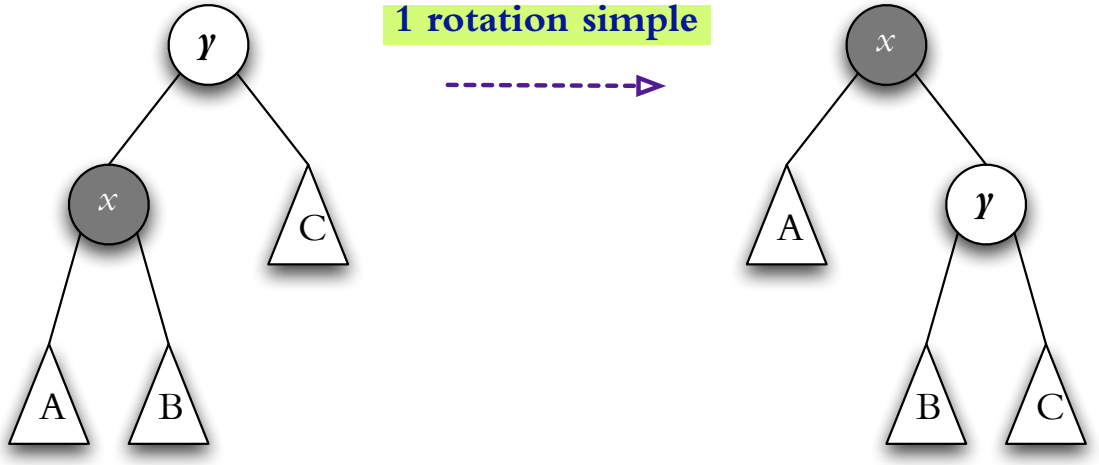
Déploiement (*splaying*) du nœud x : étapes successives jusqu'à ce x devient la racine de l'arbre

Zig et zag

Trois cas majeurs pour une étape de déploiement :

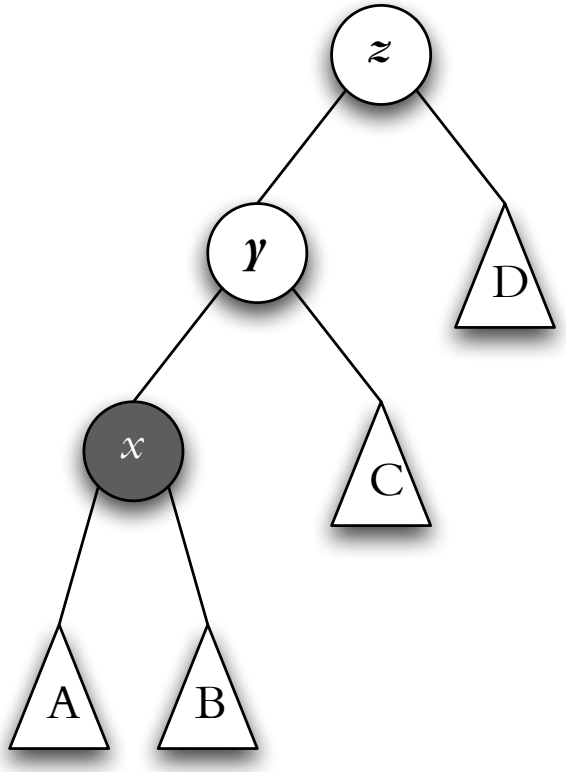
1. x sans grand-parent (zig ou zag)
2. x et son parent au même coté (gauche-gauche ou droit-droit : zig-zig ou zag-zag)
3. x et son parent à des côtés différents (gauche-droit ou droit-gauche : zig-zag ou zag-zig)

Cas 1: *zig*

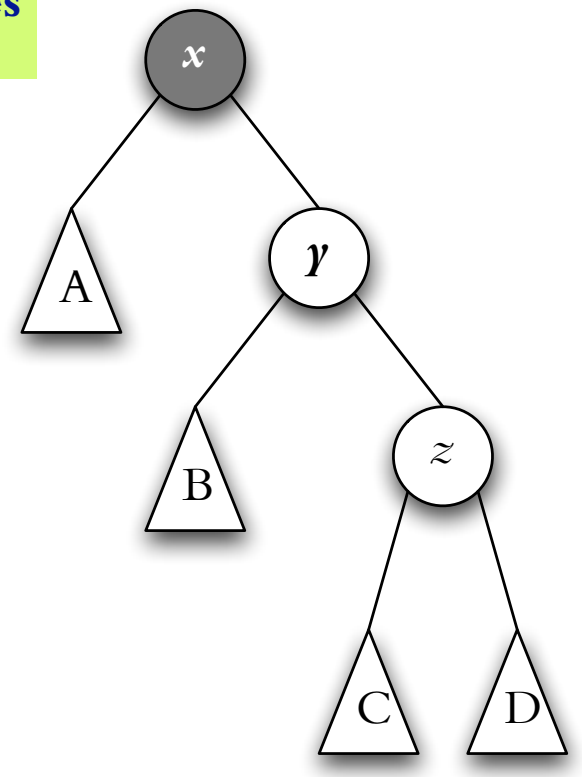


Zig-zig / zag-zag

Cas 2: *zig-zig*

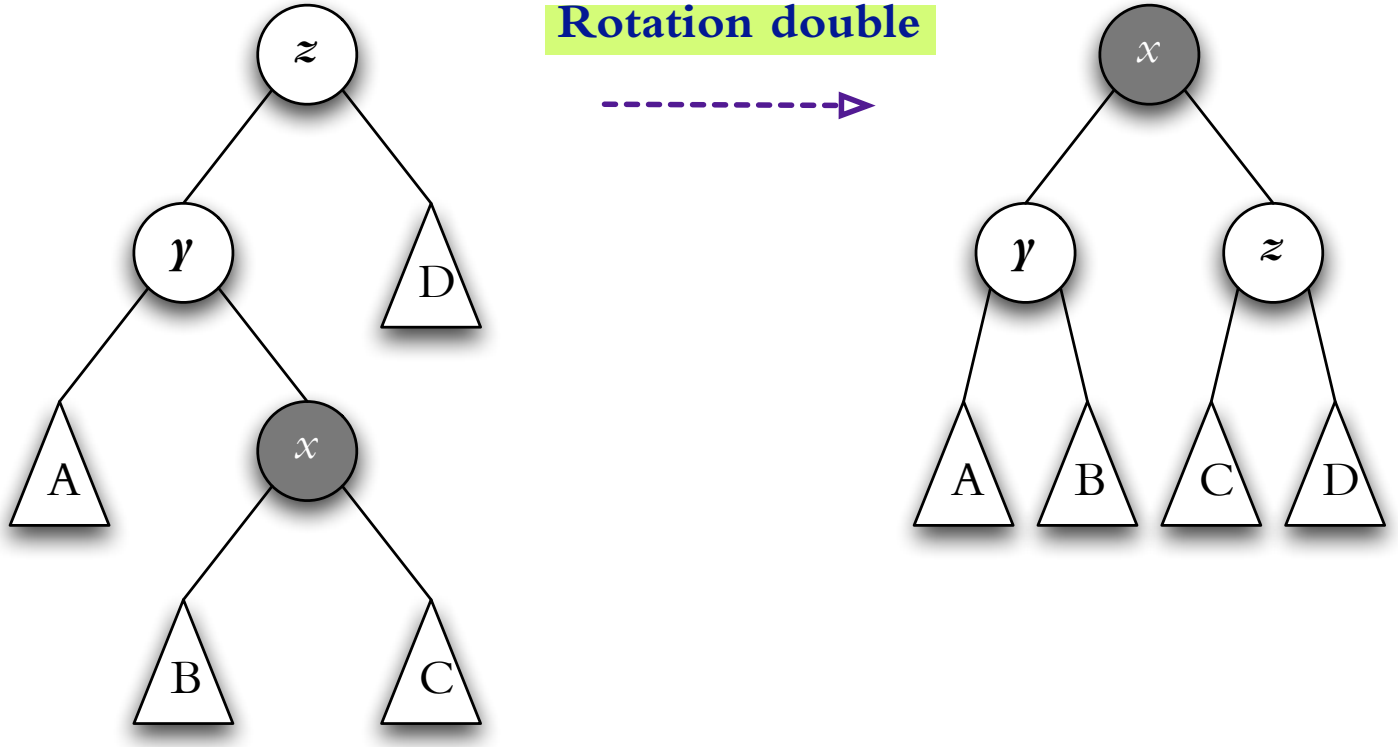


2 rotations simples
(à z et à y)



Zig-zag / zag-zig

Cas 3: *zig-zag*



Déploiement

Choix de x pour déploiement :

- insert : x est le nouveau nœud
- search : x est le nœud où on arrive à la fin de la recherche
- delete : x est le parent du nœud *effectivement* supprimé
  attention : c'est le parent ancien du successeur (ou prédecesseur) si on doit supprimer un nœud à deux enfants
  (logique : échange de nœuds, suivi par la suppression du nœud sans enfant)

Coût amorti

Temps moyen dans une **série** d'opérations

«moyen» ici : temps total divisé par nombre d'opérations
(aucune probabilité)

Théorème. Le temps pour exécuter une série de m opérations (search, insert et delete) en commençant avec l'arbre vide est de $O(m \log n)$ où n est le nombre d'opérations d'insert dans la série.

$\longrightarrow$ il peut arriver que l'exécution est très rapide au début et tout d'un coup une opération prend très long...

$\longrightarrow$ tout à fait acceptable si utilisé dans un algorithme

$\Rightarrow$ splay : ABR avec $O(\log n)$ temps amorti par opération, 0 surplus de mémoire

Arbre bicolore

Idée : une valeur entière non-négative, appellée le *rang*, associée à chaque nœud.
Notation : $\mathsf{rang}(x)$, $\mathsf{parent}(x)$ incluant x externe

Règles :

1. Pour chaque nœud x excepté la racine,

$$\mathsf{rang}(x) \leq \mathsf{rang}(\mathsf{parent}(x)) \leq \mathsf{rang}(x) + 1.$$

2. Pour chaque nœud x avec grand-parent $y = \mathsf{parent}(\mathsf{parent}(x))$,

$$\mathsf{rang}(x) < \mathsf{rang}(y).$$

3. Pour chaque nœud x externe, $\mathsf{rang}(x) = 0$ et $\mathsf{rang}(\mathsf{parent}(x)) = 1$.

Arbres RN (cont)

D'où vient la couleur ?

Les nœuds peuvent être coloriés par rouge ou noir.

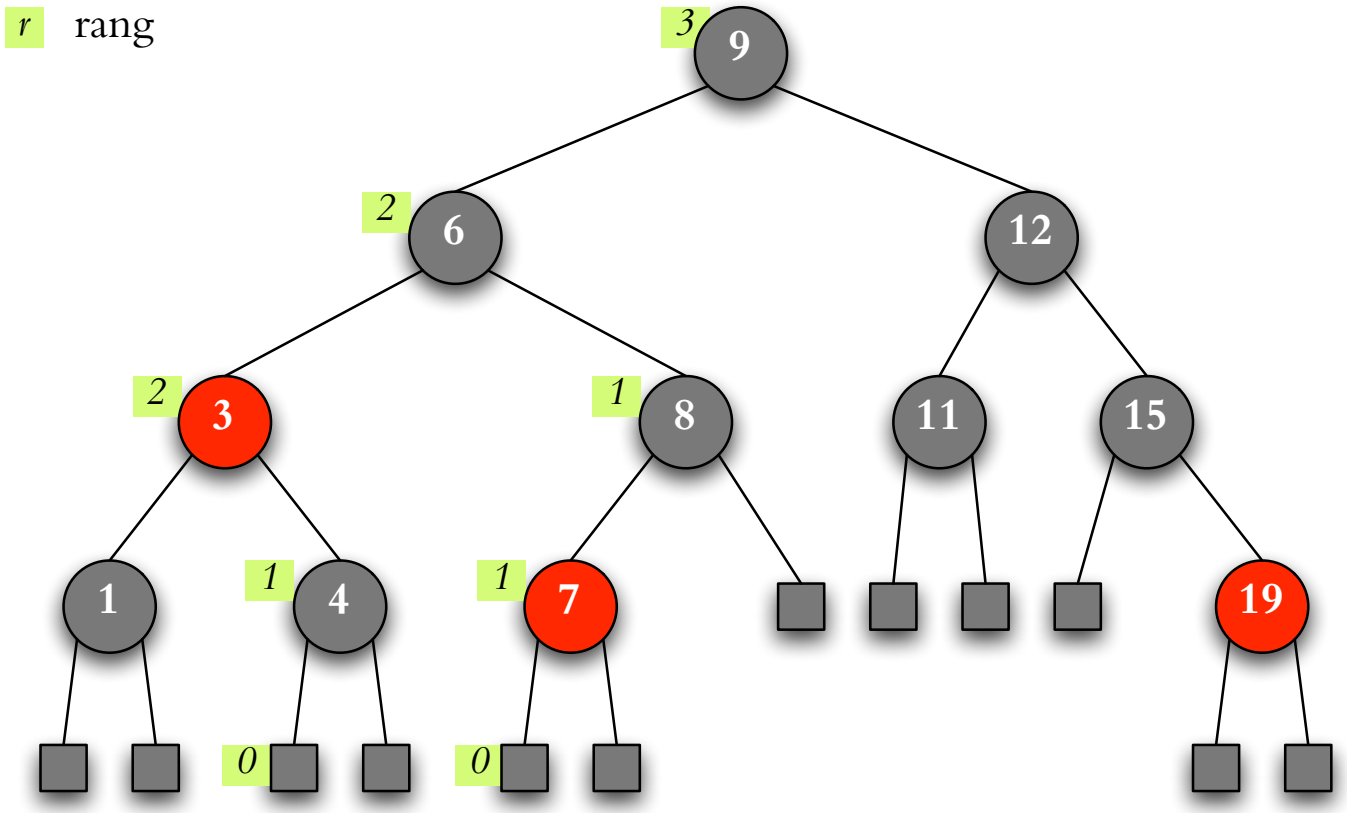
- si $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x)$, alors x est colorié par rouge
- si x est la racine ou $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x) + 1$, alors x est colorié par noir

Thm. Dans un coloriage valide,
(0) chaque nœud est soit noir soit rouge
(i) chaque nœud externe (null) est noire
(ii) le parent d'un nœud rouge est noir
(iii) tout chemin d'un nœud x à un nœud externe dans son sous-arbre contient le même nombre de nœuds noirs

Preuve En (iii), le nombre de nœuds noirs sur le chemin est égal au rang. □

` rang est parfois appelé «hauteur noire»

Arbres RN (cont)



Arbres RN (cont)

Thm. La hauteur dans un arbre RN : pour chaque nœud x,
sa hauteur $h(x) \leq 2 \cdot \mathsf{rang}(x)$.

Preuve. On doit avoir au moins autant de nœuds noirs que des nœuds rouges dans un chemin de x à un nœud externe. $\square$

Arbres RN (cont)

Thm. Le nombre de nœuds internes dans le sous-arbre de tout x est

$$n(x) \geq 2^{\mathsf{rang}(x)} - 1.$$

Démonstration.

Cas de base. Le théorème est vrai pour un nœud externe x quand $\mathsf{rang}(x) = 0$.
Hypothèse d'induction. Supposons que le théorème est vrai pour tout x avec une hauteur $h(x) < k$. Considérons un nœud x avec $h(x) = k$ et ses deux enfants u, v avec $h(u) < k, h(v) < k$.
Cas inductif. Par l'hypothèse d'induction,

$$n(x) \geq 1 + (2^{\mathsf{rang}(u)} - 1) + (2^{\mathsf{rang}(v)} - 1).$$

Or, $\mathsf{rang}(x) - 1 \leq \mathsf{rang}(u)$ et $\mathsf{rang}(x) - 1 \leq \mathsf{rang}(v)$. □

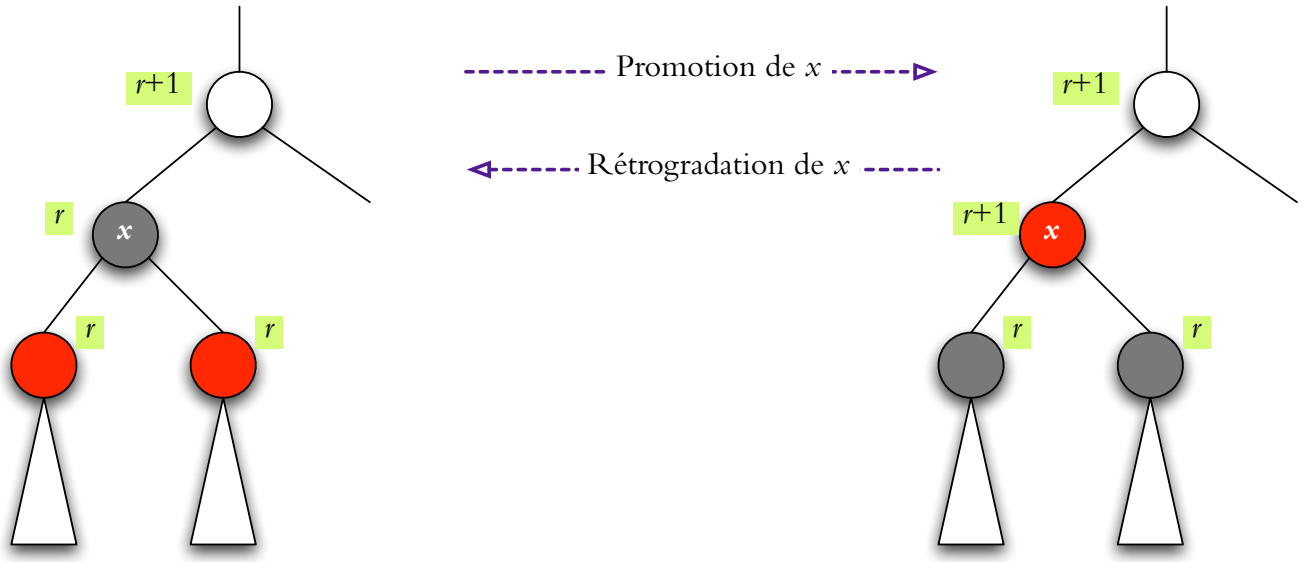
Arbres RN (cont)

Thm. Un arbre RN avec n nœuds internes a une hauteur $\leq 2\lfloor \lg(n+1) \rfloor$.

Démonstration. $h(x) \leq 2 \cdot \text{rang}(x)$ et $n(x) \geq 2^{\text{rang}(x)} - 1$. □

Arbres RN — équilibre

Pour maintenir l'équilibre, on utilise les **rotations** comme avant
+ **promotion/rétrogradation** : incrémenter ou décrementer le rang



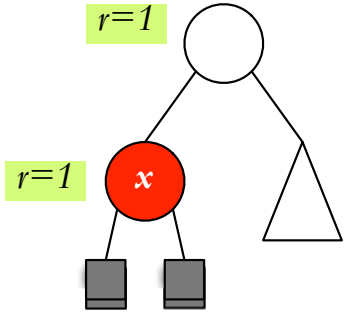
→ promotion/rétrogradation change la couleur d'un nœud et ses enfants
on peut promouvoir x ssi il est noir avec deux enfants rouges

Arbre RN — insertion

dans un arbre RN on insère un nouveau nœud x selon la procédure standard
mais comment devrait-on le colorier ?

☞ si c'est le premier nœud, il devient la racine noire

☞ sinon, x remplace un nœud externe, donc il doit être rouge
(car il a le même rang (=1) que son parent)



TEST : couleur du parent de x (qui est rouge) ?

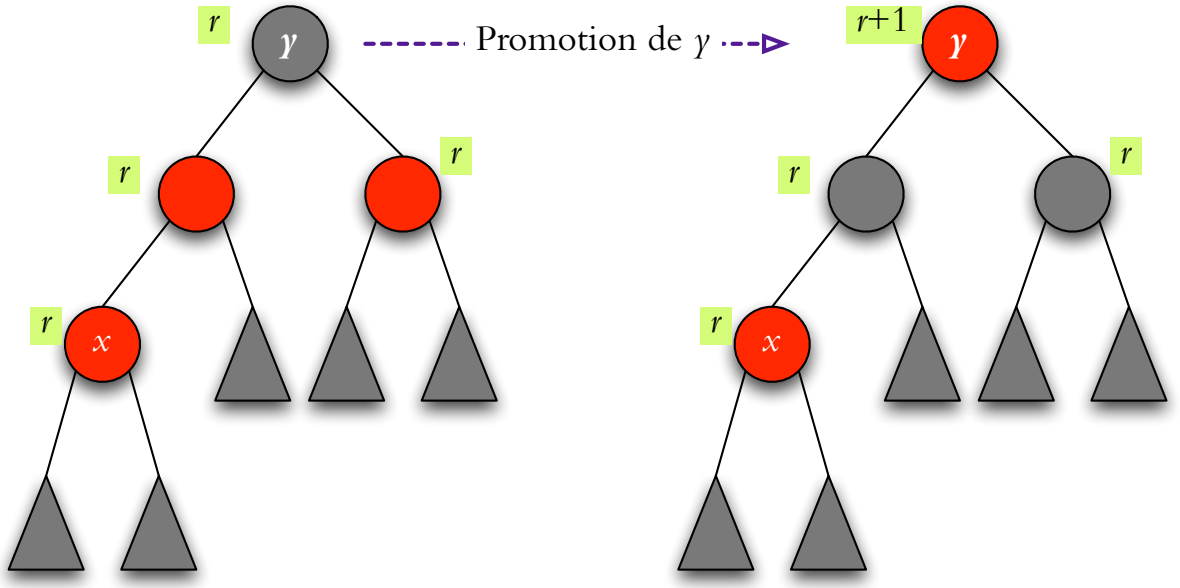
noir coloriage correct, il ne reste rien à faire

rouge règle de coloriage violée $\Rightarrow$ fixer le coloriage

Fixer le coloriage

☞si le parent de x est rouge et il n'est pas la racine
alors le grand-parent $y = \mathsf{parent}(\mathsf{parent}(x))$ est sûrement noir
(car il a un enfant rouge et le coloriage était OK avant l'insertion)

☞si l'autre enfant du grand-parent (l'«oncle» de x) est rouge aussi,
alors promouvoir y et retourner au TEST avec $x \leftarrow y$.



Fixer le coloriage 2

☞si le parent de x est rouge, mais son oncle est noir
alors faire des rotations

Cas 1 quand x et $\mathsf{parent}(x)$ sont au même côté (enfants gauches ou enfants droits) — une rotation suffit

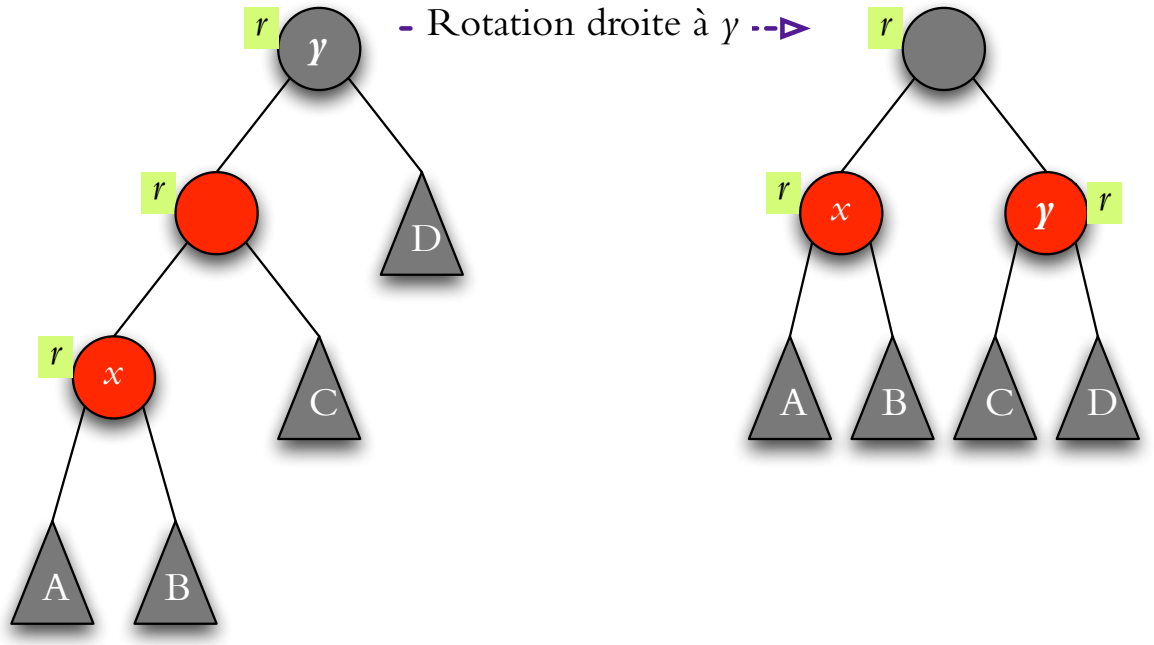
Cas 2 quand x et $\mathsf{parent}(x)$ ne sont pas au même côté (l'un est un enfant gauche et l'autre un enfant droit) — rotation double est nécessaire

On a quatre cas : 1/zig-zig, 1/zag-zag, 2/zig-zag, 2/zag-zig

mise à jour de coloriage : automatique car le rang des nœuds ne change pas dans les rotations

Arbres RN — insertion (cont)

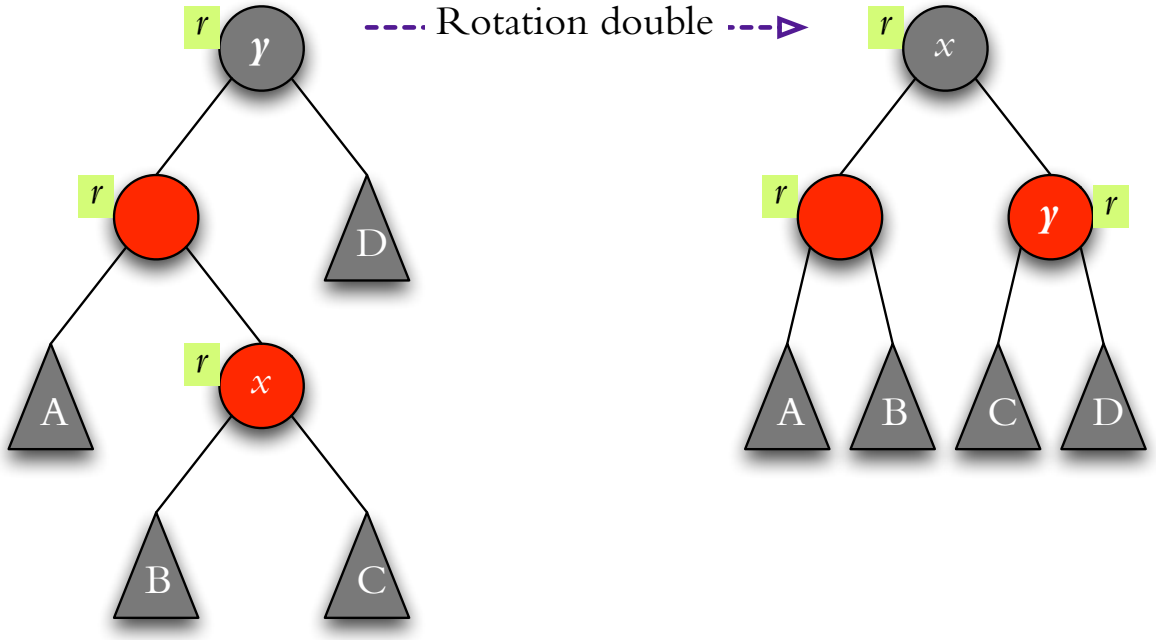
Cas 1/zig-zig : x est rouge, son parent est rouge, son oncle est noir, et x et $\mathsf{parent}(x)$ sont des enfant gauches



Cas 1/zag-zag (enfants droits) est symétrique

Arbres RN — insertion (cont)

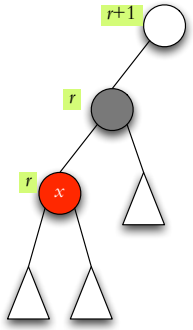
Cas 2/zig-zag : x est rouge, son parent est rouge, son oncle est noir, x est un enfant droit et $\mathsf{parent}(x)$ est un enfant gauche



Cas 2/zag-zig (x est gauche et $\mathsf{parent}(x)$ est droit) est symétrique

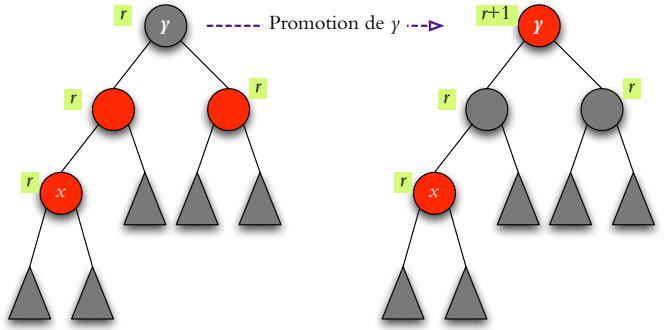
Arbre RN — insertion

Cas 0a : parent noir



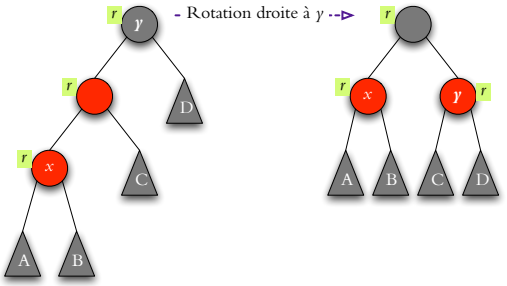
rien à faire

Cas 0b : parent rouge, oncle rouge



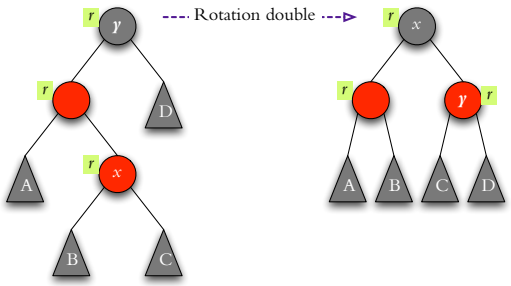
promotion du grandparent, continuer avec $x \leftarrow y$

Cas 1 : parent rouge, oncle noir, zig-zig



une rotation simple

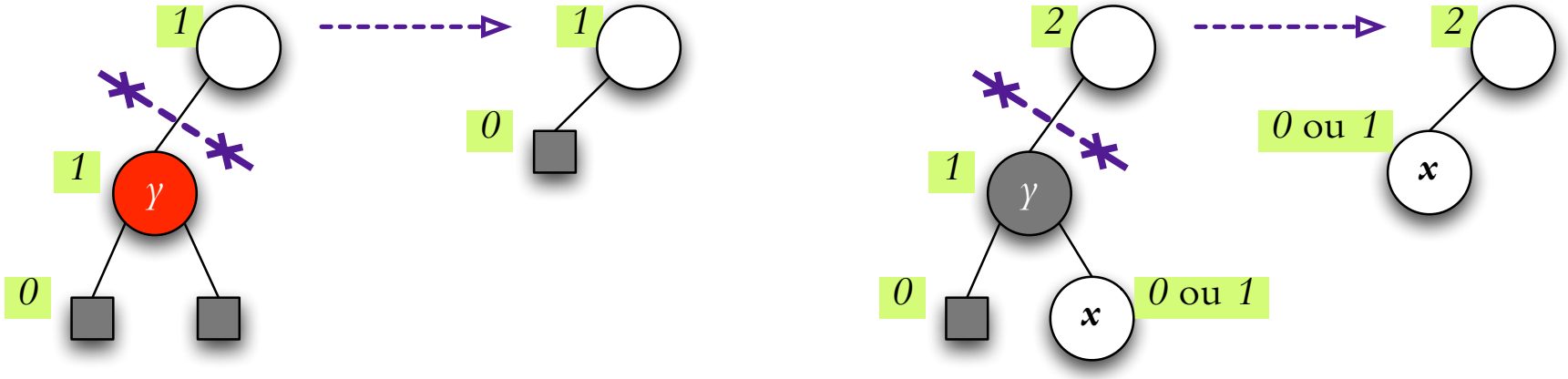
Cas 2 : parent rouge, oncle noir, zig-zag



une rotation double

Suppression et problèmes de coloriage

On enlève un nœud y : remplacement par null (si aucun enfant) ou par l'enfant non-null. Ce dernier peut être de rang trop petit (nœud noir remplacé par nœud noir x).



Plusieurs cas :

Cas 0 : nœud rouge x : rétrogradation — il devient noir, et rien plus à faire

Cas 1 : nœud noir avec une sœur noire

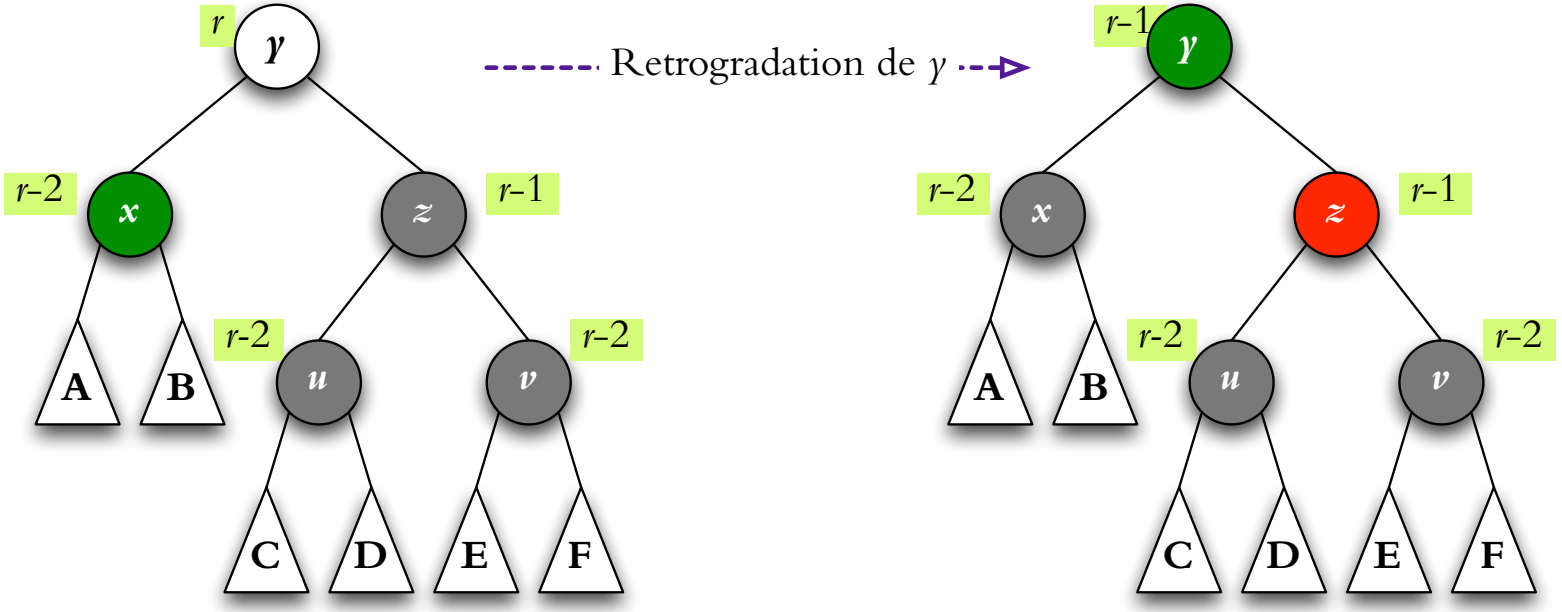
Cas 2 : nœud noir avec une sœur rouge

En cas 1, il faut vérifier la couleur des enfants de la sœur (les neveux)

Arbre RN — suppression 1A

Cas 1A : sœur noire, neveux noirs

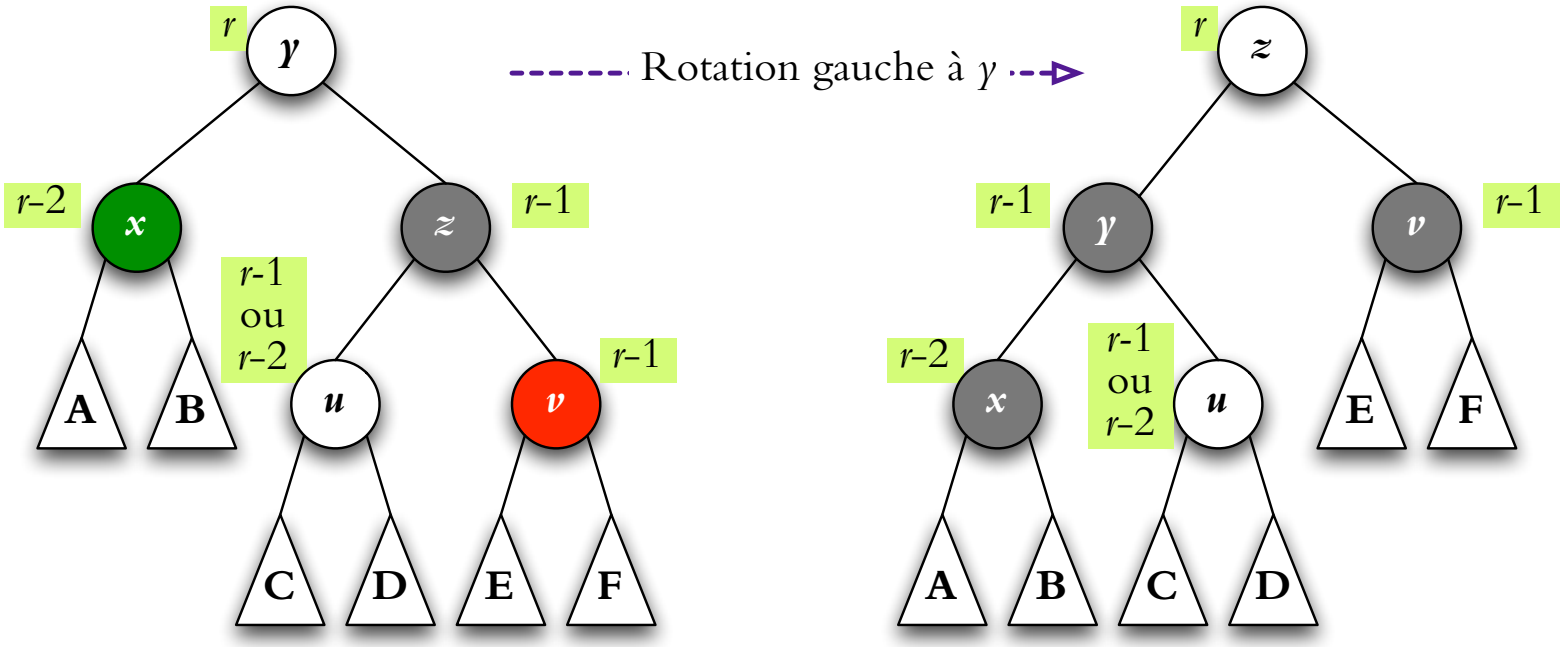
décrementer $\mathsf{rang}\big(\mathsf{parent}(x)\big)$: continuer avec $x \leftarrow \mathsf{parent}(x)$ en cas 0,1 ou 2.



Arbre RN — suppression 1B

Cas 1B : sœur noire, neveu distant (v) rouge

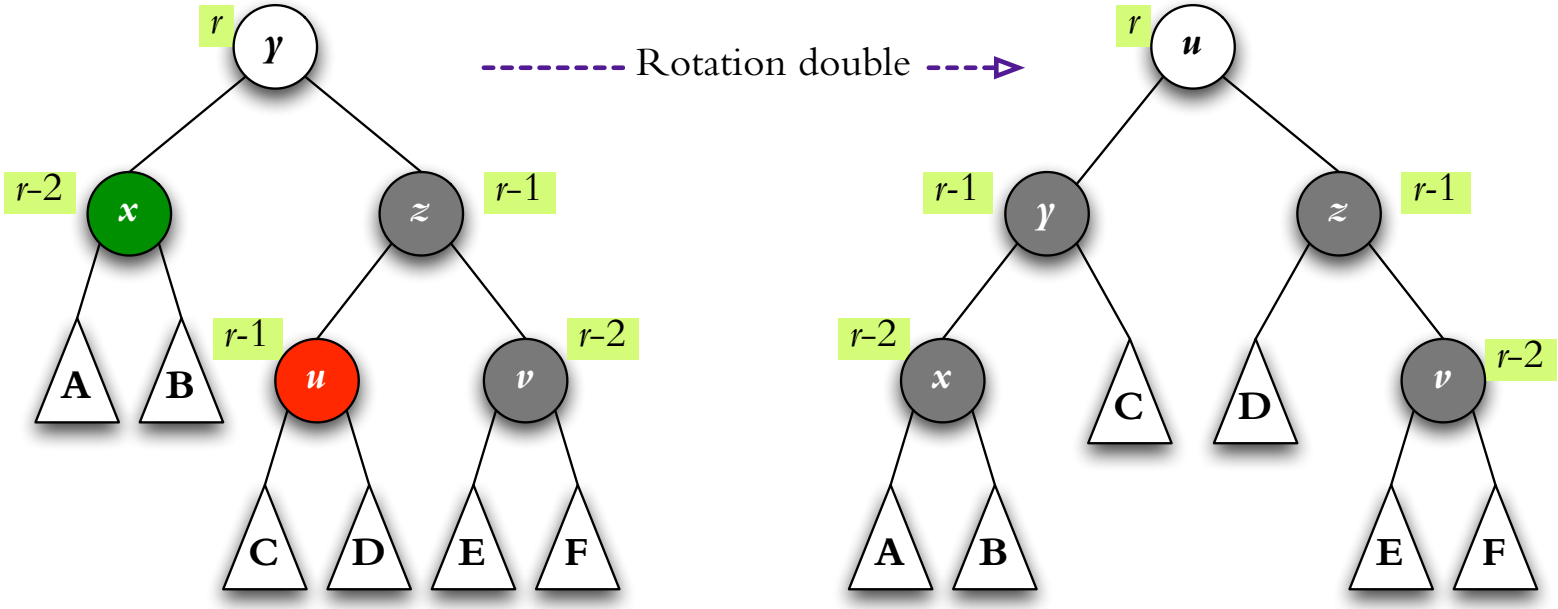
faire une rotation simple et arrêter



Arbre RN — suppression 1C

Cas 1C : sœur noire, neveu proche (u) rouge

faire une rotation double et arrêter

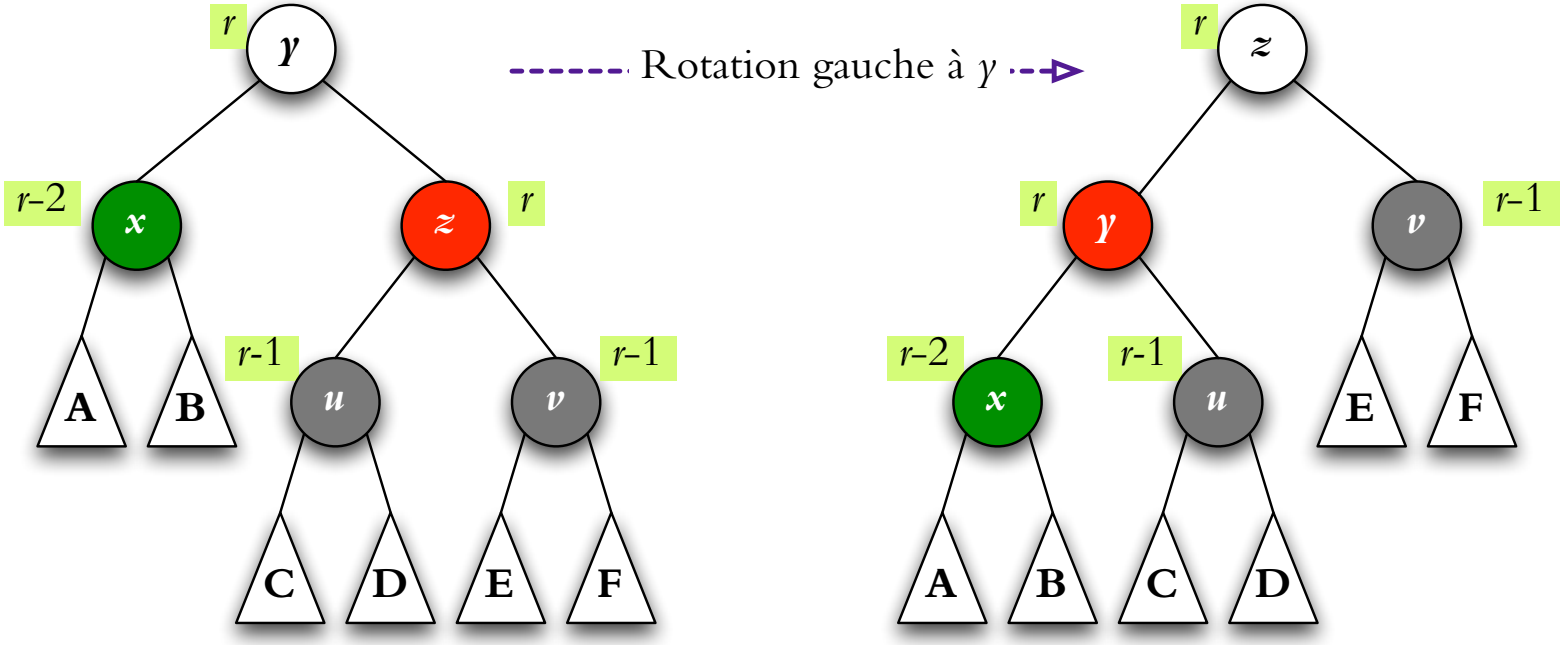


(quand $\mathsf{rang}(v) = r - 1$, on peut toujours faire cette rotation mais cas 1B est plus rapide)

Arbre RN — suppression 2

Cas 2 : sœur rouge

faire une rotation simple et continuer en cas 1 avec x

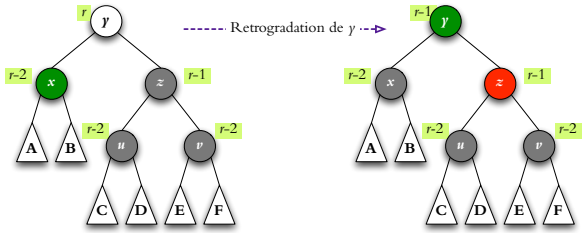


si on continue en cas 1a (les enfants de u sont noirs), alors la récursion se termine avec la rétrogradation de y qui est rouge (cas 0 pour y)

Arbre RN — suppression

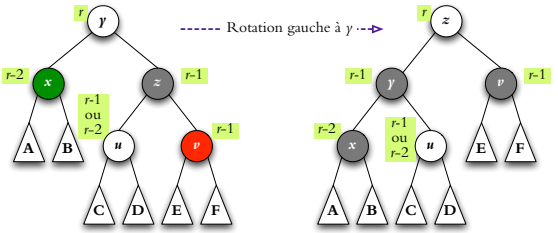
Cas 0 : nœud rouge x — il devient noir, et on arrête

Cas 1a : sœur noire, neveux noirs



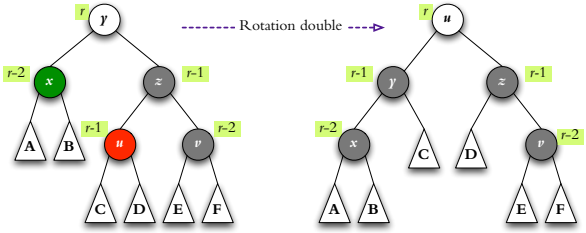
retrogradation du parent, continuer avec $x \leftarrow y$ en cas 0, 1, ou 2

Cas 1b : sœur noire, neveu distant rouge



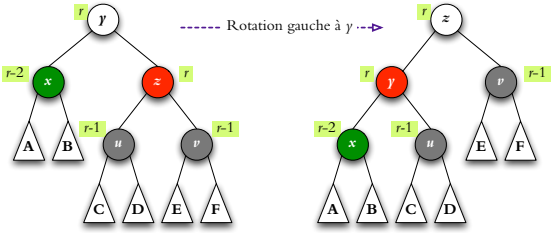
une rotation simple

Cas 1c : sœur noire, neveu proche rouge



une rotation double

Cas 2 : sœur rouge



une rotation simple, continuer avec x en cas 1 (pas de récursion en 1a)

Arbre RN — efficacité

Un arbre rouge et noir avec n nœuds internes a une hauteur $\leq 2 \lg(n+1)$

Recherche : $O(h)$ mais $h = O(\log n)$ donc $O(\log n)$

Insertion :
1. $O(h)$ pour trouver le placement du nouveau nœud
2. $O(1)$ pour initialiser les pointeurs
3. $O(h)$ promotions en ascendant si nécessaire
4. $O(1)$ pour une rotation simple ou double si nécessaire

$O(h)$ en total mais $h = O(\log n)$ donc $O(\log n)$

Suppression : $O(\log n)$

Usage de mémoire : il suffit de stocker la couleur (1 bit) de chaque nœud interne