ARBRES BINAIRES DE RECHERCHE

Table de symboles

Recherche : opération fondamentale

données : éléments avec clés

Type abstrait d'une **table de symboles** (*symbol table*) ou dictionnaire

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

Opérations :
$\mathsf{insert}(x, D)$: insertion de l'élément x dans D
$\mathsf{search}(k, D)$: recherche d'un élément à clé k (peut être infructeuse)

Opérations parfois supportées :
$\mathsf{delete}(k, D)$: supprimer élément avec clé k
$\mathsf{select}(i, D)$: sélection de l'i-ème élément (selon l'ordre des clés)

Structures de données

structures simples : tableau trié ou liste chaînée

arbre binaire de recherche

tableau de hachage (plus tard)

Structures simples

liste chaînée ou tableau non-trié : recherche séquentielle
temps de $\Theta(n)$ au pire (même en moyenne)

tableau trié : recherche binaire
temps de $\Theta(\log n)$ au pire

tableau trié : insertion/suppression en $\Theta(n)$ au pire cas

Arbre binaire de recherche

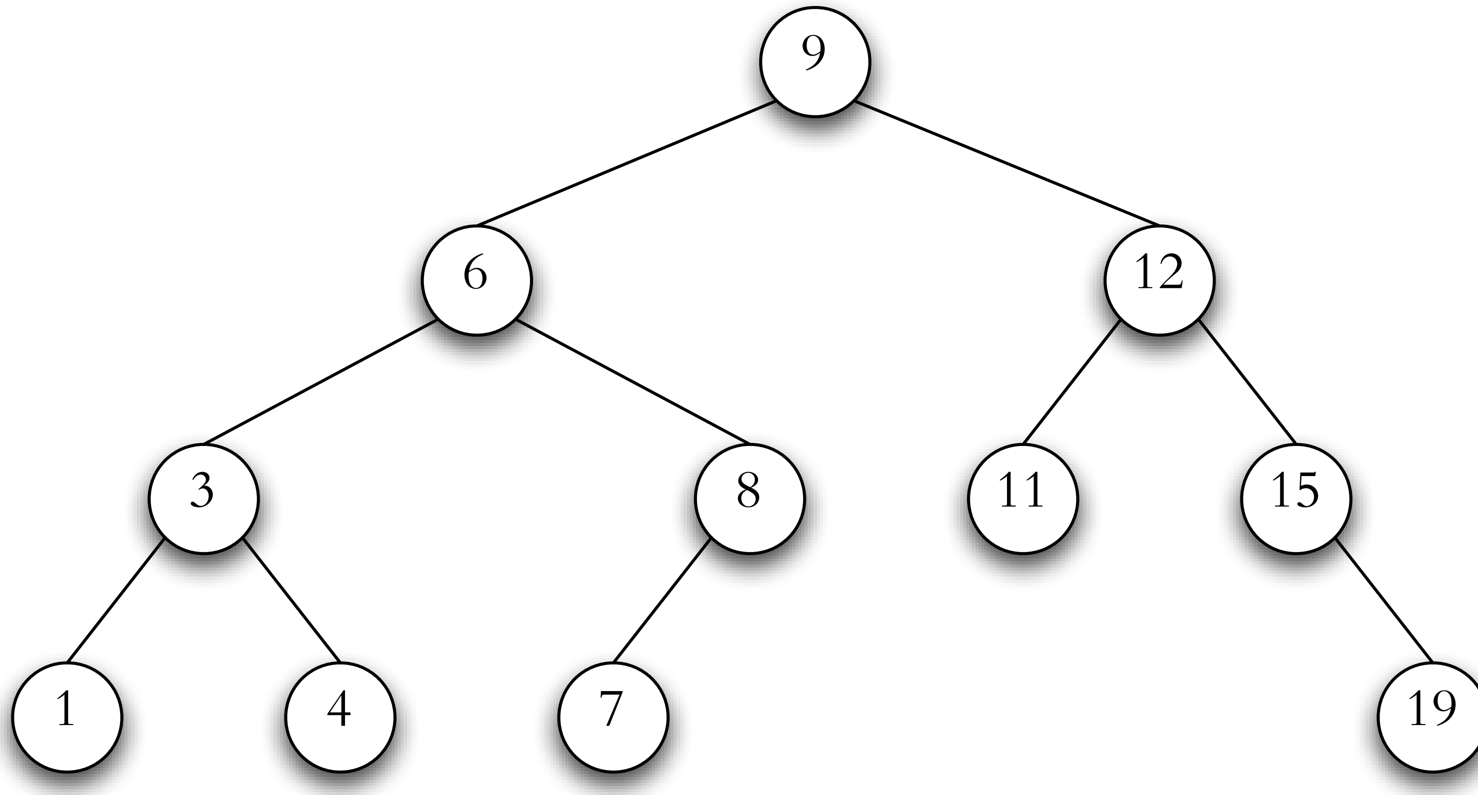
Dans un arbre binaire de recherche, chaque nœud a une clé.

Accès aux nœuds :
$\mathsf{gauche}(x)$ et $\mathsf{droit}(x)$ pour les enfants de x (null s'il n'y en a pas)
$\mathsf{parent}(x)$ pour le parent de x (null pour la racine)
$\mathsf{cle}(x)$ pour la clé de nœud x (en général, un entier dans nos discussions)

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

Thm. Soit x un nœud dans un arbre binaire de recherche. Si y est un nœud dans le sous-arbre gauche de x, alors $\mathsf{cle}(y) \leq \mathsf{cle}(x)$. Si y est un nœud dans le sous-arbre droit de x, alors $\mathsf{cle}(y) \geq \mathsf{cle}(x)$.

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 : **recherche** d'une valeur particulière, **insertion** ou **suppression** d'une valeur, recherche de **min** ou **max**, et des autres.

Pour la discussion des arbres binaires de recherche, on va considérer les pointeurs `null` pour des enfants manquants comme des pointeurs vers des feuilles ou nœuds externes

Donc toutes les feuilles sont `null` et tous les nœuds avec une valeur `cle()` sont des nœuds internes.

Min et max

Algo MIN() // trouve la valeur minimale dans l'arbre

$x \leftarrow$ racine; $y \leftarrow$ null
tandis que $x \neq$ null **faire**
    $y \leftarrow x$; $x \leftarrow$ gauche(x)
retourner y

Algo MAX() // trouve la valeur maximale dans l'arbre

$x \leftarrow$ racine; $y \leftarrow$ null
tandis que $x \neq$ null **faire**
    $y \leftarrow x$; $x \leftarrow$ droit(x)
retourner y

Recherche

Algo SEARCH(x, v) // trouve la clé v dans le sous-arbre de x

F1 **si** $x = \mathsf{null}$ ou $v = \mathsf{cle}(x)$ **alors** retourner x

F2 **si** $v < \mathsf{cle}(x)$

F3 **alors** retourner SEARCH$(\mathsf{gauche}(x), v)$

F4 **sinon** retourner SEARCH$(\mathsf{droit}(x), v)$

Maintenant, SEARCH(racine, v) retourne

- soit un nœud dont la clé est égale à v,
- soit null.

Notez que c'est une recursion terminale $\Rightarrow$ transformation en forme itérative

Recherche (cont)

Solution itérative (plus rapide) :

Algo SEARCH(x, v) // trouve la clé v dans le sous-arbre de x

tandis que $x \neq \mathsf{null}$ et $v \neq \mathsf{cle}(x)$ **faire**

  si $v < \mathsf{cle}(x)$

  alors $x \leftarrow \mathsf{gauche}(x)$

  sinon $x \leftarrow \mathsf{droit}(x)$

retourner x

Recherche — efficacité

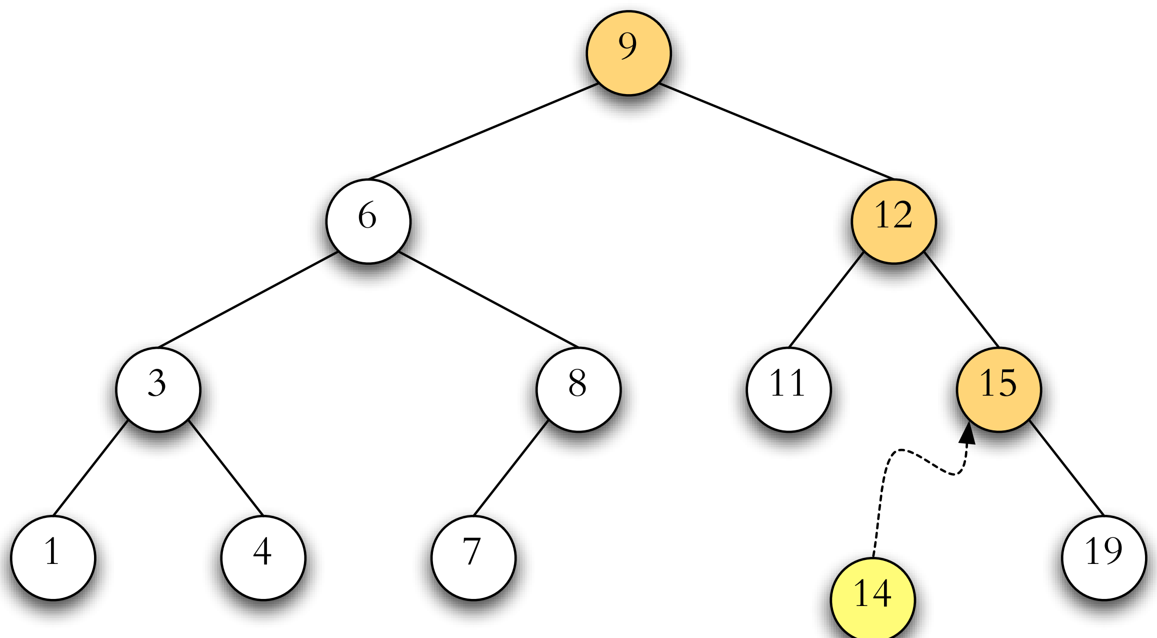
Dans un arbre binaire de recherche de hauteur h :

MIN() prend $O(h)$
MAX() prend $O(h)$
SEARCH(racine, v) prend $O(h)$

Insertion

On veut insérer une clé v

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



insertion de «14»

Insertion (cont.)

insertion — pas de clés dupliquées

Algo INSERT(v) // insère la clé v dans l'arbre

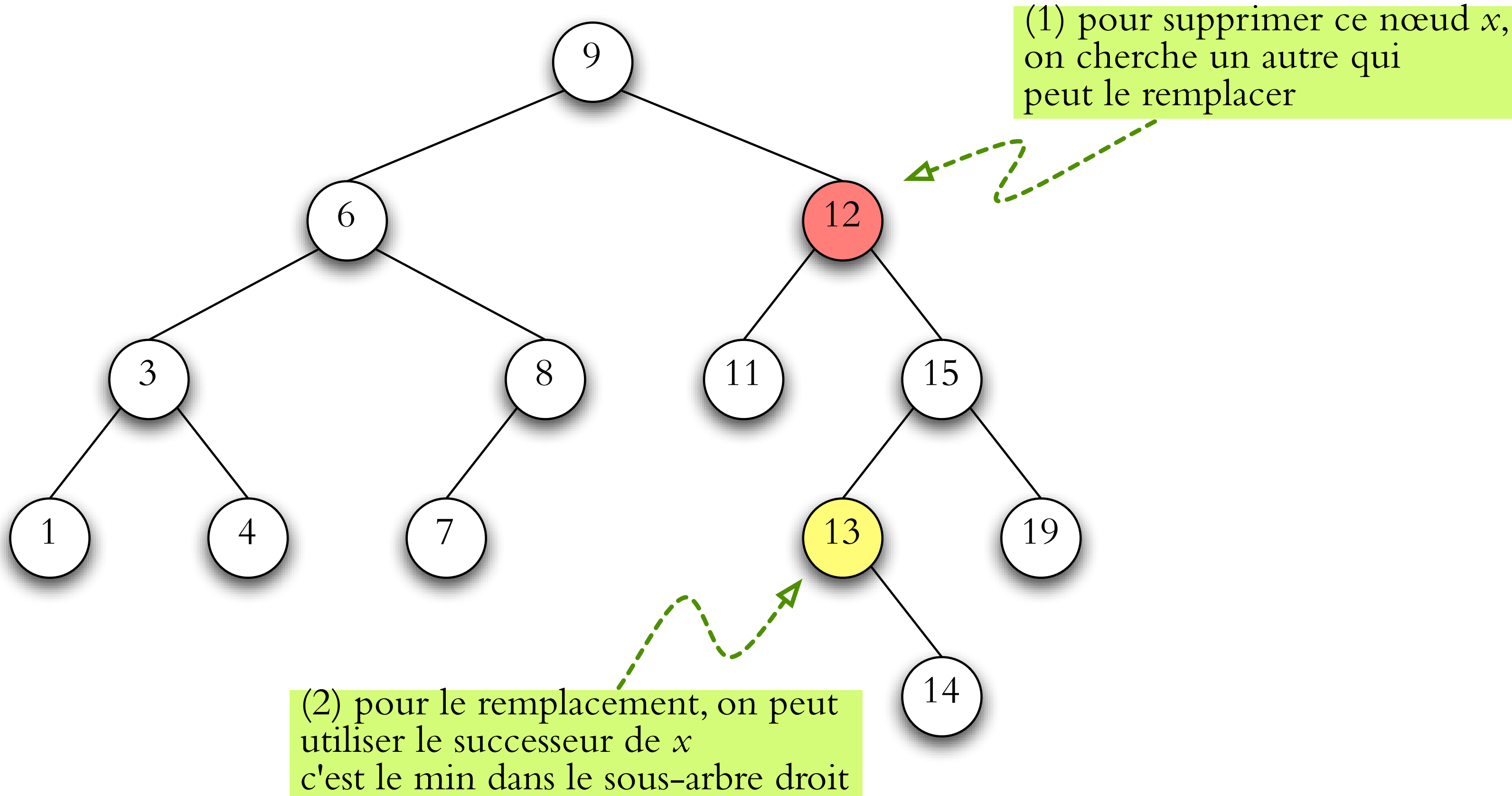
$x \leftarrow$ racine
si $x =$ null **alors** initialiser avec une racine de clé v et retourner
tandis que vrai **faire** // (*conditions d'arrête testées dans le corps*)
    si $v =$ cle(x) **alors** retourner // (*pas de valeurs dupliquées*)
    si $v <$ cle(x)
    alors si gauche(x) $=$ null
        alors attacher nouvel enfant gauche de x avec clé v et retourner
        sinon $x \leftarrow$ gauche(x)
    sinon si droit(x) $=$ null
        alors attacher nouvel enfant droit de x avec clé v et retourner
        sinon $x \leftarrow$ droit(x)

Suppression

Suppression d'un nœud x

1. triviale si x est une feuille : $\mathsf{gauche}(\mathsf{parent}(x)) \leftarrow \mathsf{null}$ si x est l'enfant gauche de son parent, ou $\mathsf{droit}(\mathsf{parent}(x)) \leftarrow \mathsf{null}$ si x est l'enfant droit
2. facile si x a seulement un enfant : $\mathsf{gauche}(\mathsf{parent}(x)) \leftarrow \mathsf{droit}(x)$ si x a un enfant droit et il est l'enfant gauche (4 cas en total dépendant de la position de x et celle de son enfant)
3. un peu plus compliqué si x a deux enfants

Suppression — deux enfants



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

Insertion et suppression — efficacité

Dans un arbre binaire de recherche de hauteur h :

$\textsc{Insert}(v)$ prend $O(h)$
suppression d'un nœud prend $O(h)$

Hauteur de l'arbre

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

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

Insertion successive de $1, 2, 3, 4, \ldots, n$ donne un arbre avec $h = n - 1$.

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

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

Thm. Hauteur moyenne d'un arbre de recherche construit en insérant les valeurs $1, 2, \ldots, n$ selon une permutation aléatoire est $\alpha \lg n$ en moyenne où $\alpha \approx 2.99$.

(preuve trop compliquée pour les buts de ce cours)

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

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

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. $\square$

D'ici, comme l'analyse de la performance du tri rapide. . .

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

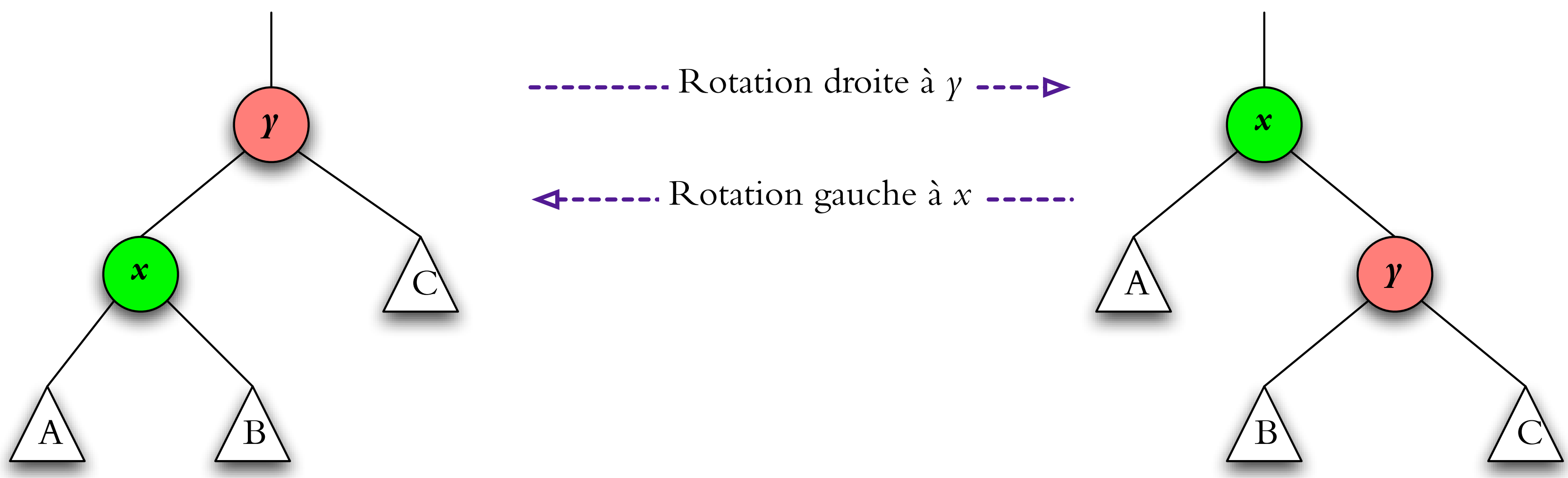
Arbres équilibrés

Arbres équilibrés : 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

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.

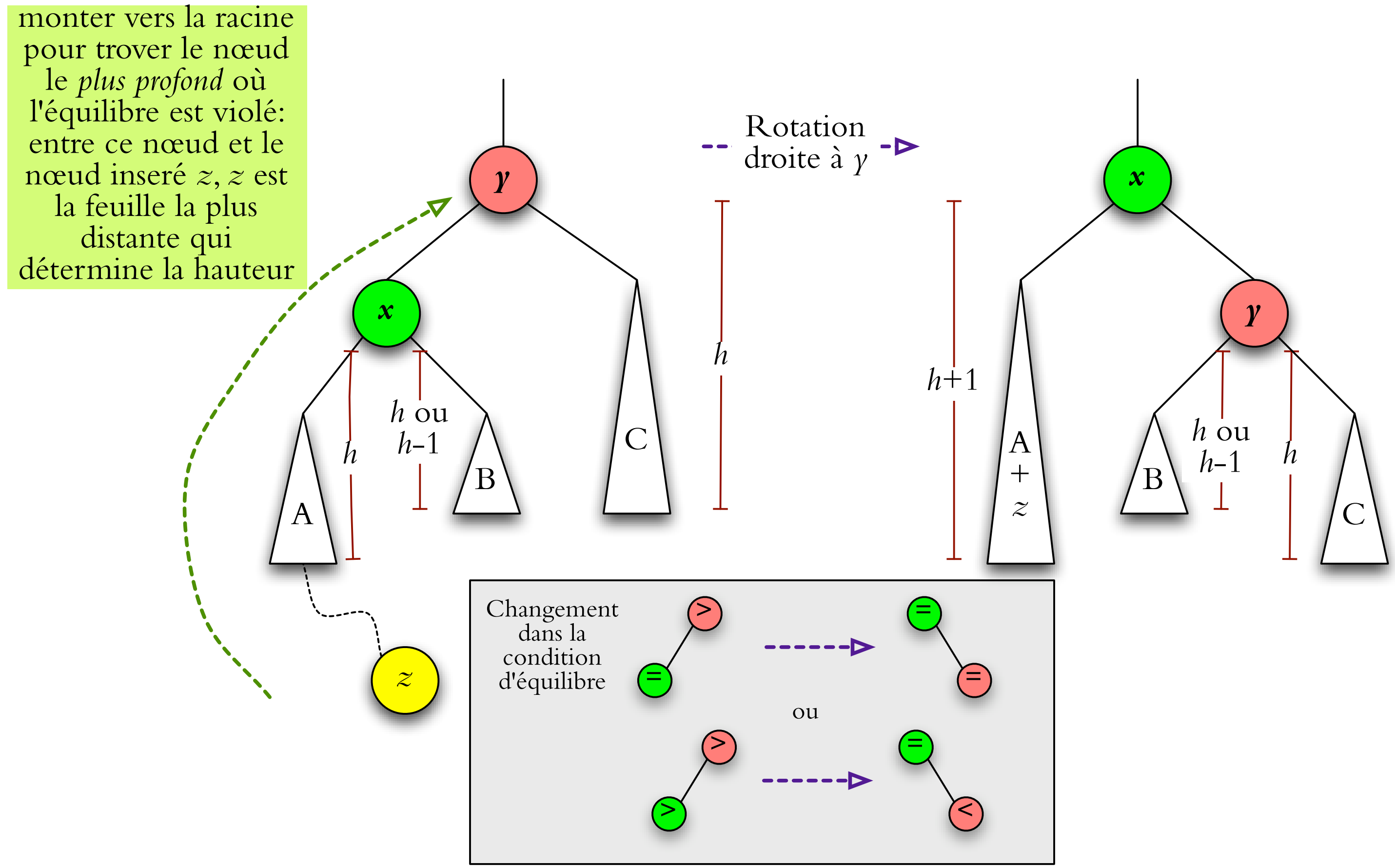
(hauteur d'un sous-arbre vide = -1.)

On va donc stocker la hauteur de chaque sous-arbre à sa racine.

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

Insertion dans 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 dans un arbre AVL de hauteur $h \geq 0$. On a $N(0) = 1$, $N(1) = 2$.

Lemme. Pour tout $h > 1$,

$$N(h) = N(h-1) + N(h-2) + 1.$$

Lemme Il existe $c > 0$ tel que $N(h) \leq c\phi^h - 1$ pour tout $h \geq 0$ où $\phi = \frac{1+\sqrt{5}}{2}$. (Notez que $\phi^2 = \phi + 1$.)

Preuve La constante c sera spécifiée plus tard. Supposons que $N(k) \leq c\phi^k - 1$ pour tout $0 \leq k < h$. Alors,

$$N(h) = N(h-1) + N(h-2) + 1 \leq c\phi^{h-1} + c\phi^{h-2} - 1 = c\phi^h - 1.$$

Si on choisit $c = 2$, la borne est correcte pour $h = 0, 1$ et donc elle est correcte pour tout h. $\square$

Hauteur d'un arbre AVL (cont)

Donc un arbre AVL sur n nœuds est de hauteur

$$h \le \log_\phi \frac{n+1}{2} = \frac{\lg(n+1)-1}{\lg\phi} \approx 1.44 \lg n \in O(\log n).$$

Arbres *splay*

On utilise souvent des variables auxiliares pour maintenir l'équilibre de l'arbre
p.e., arbre rouge et noir : au moins un bit (couleur)

Arbre *splay* : aucune variable

mais $O(\log n)$ seulement comme coût amorti

Arbres *splay* (cont)

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 que $\mathsf{parent}(x)$ devienne null
(et donc x devient la racine de l'arbre)

Zig et zag

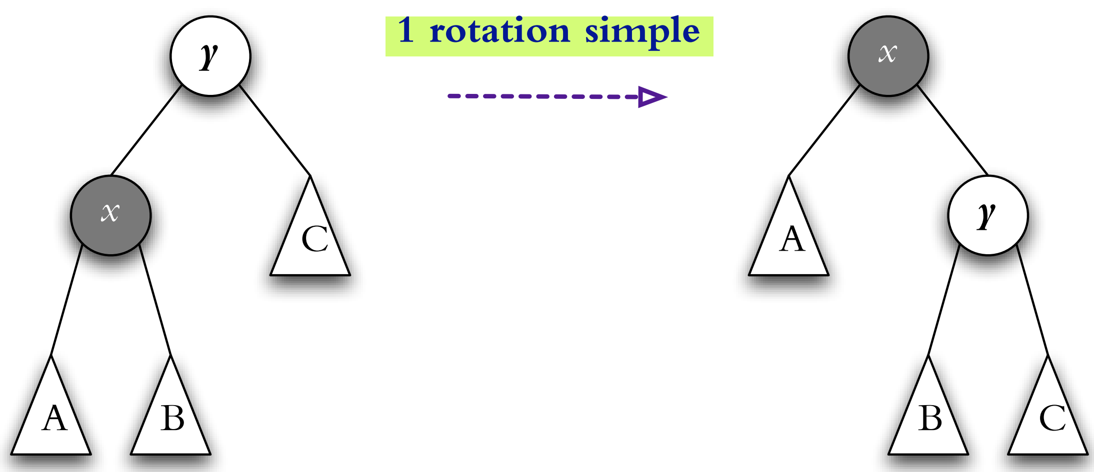
Trois cas majeurs pour une étape de déploiement :

1. x sans grand-parent (zig ou zag)

2. x et $\mathsf{parent}(x)$ au même coté (gauche-gauche ou droit-droit : zig-zig ou zag-zag)

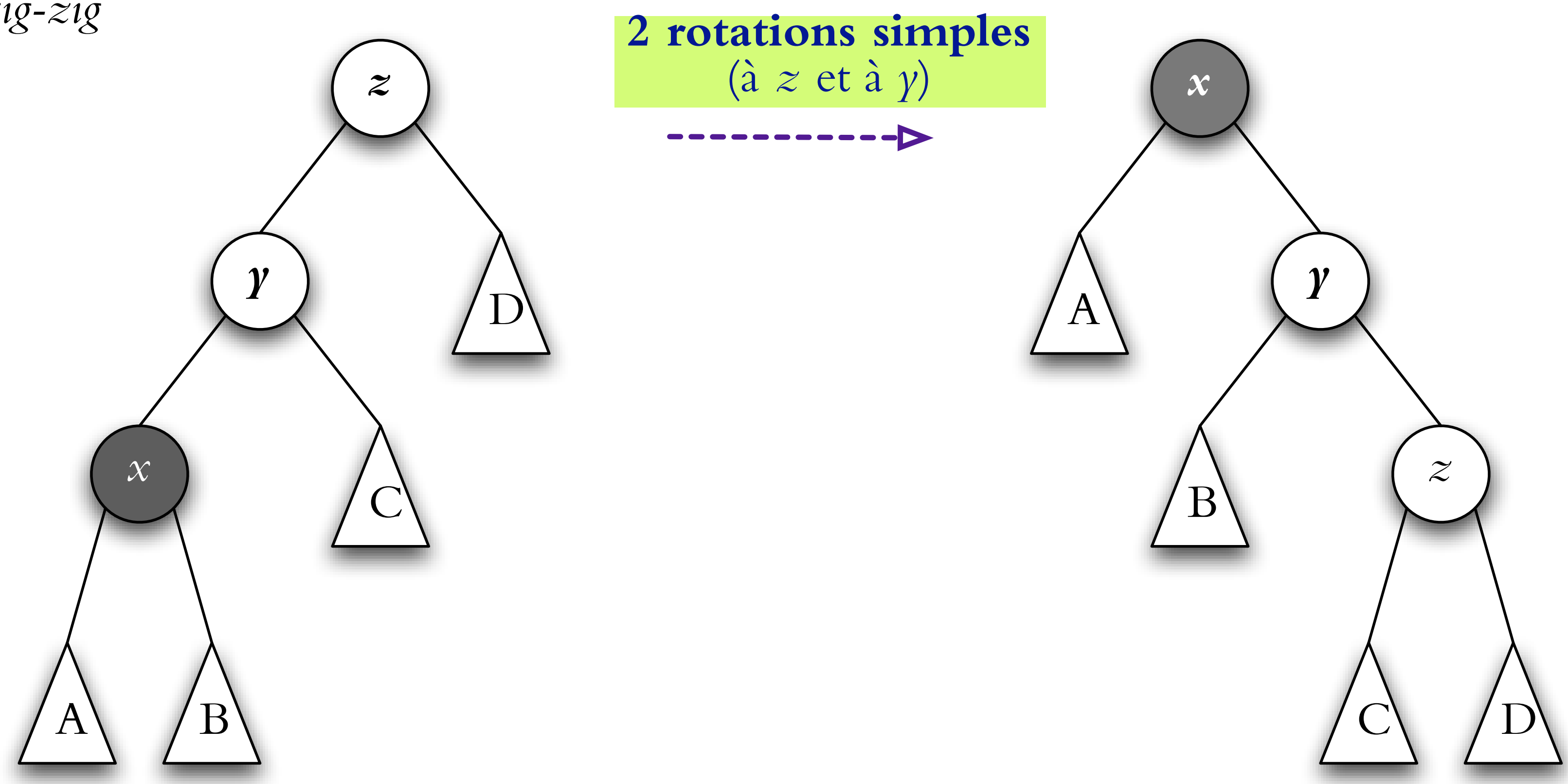
3. x et $\mathsf{parent}(x)$ à des côtés différents (gauche-droit ou droit-gauche : zig-zag ou zag-zig)

Cas 1: *zig*



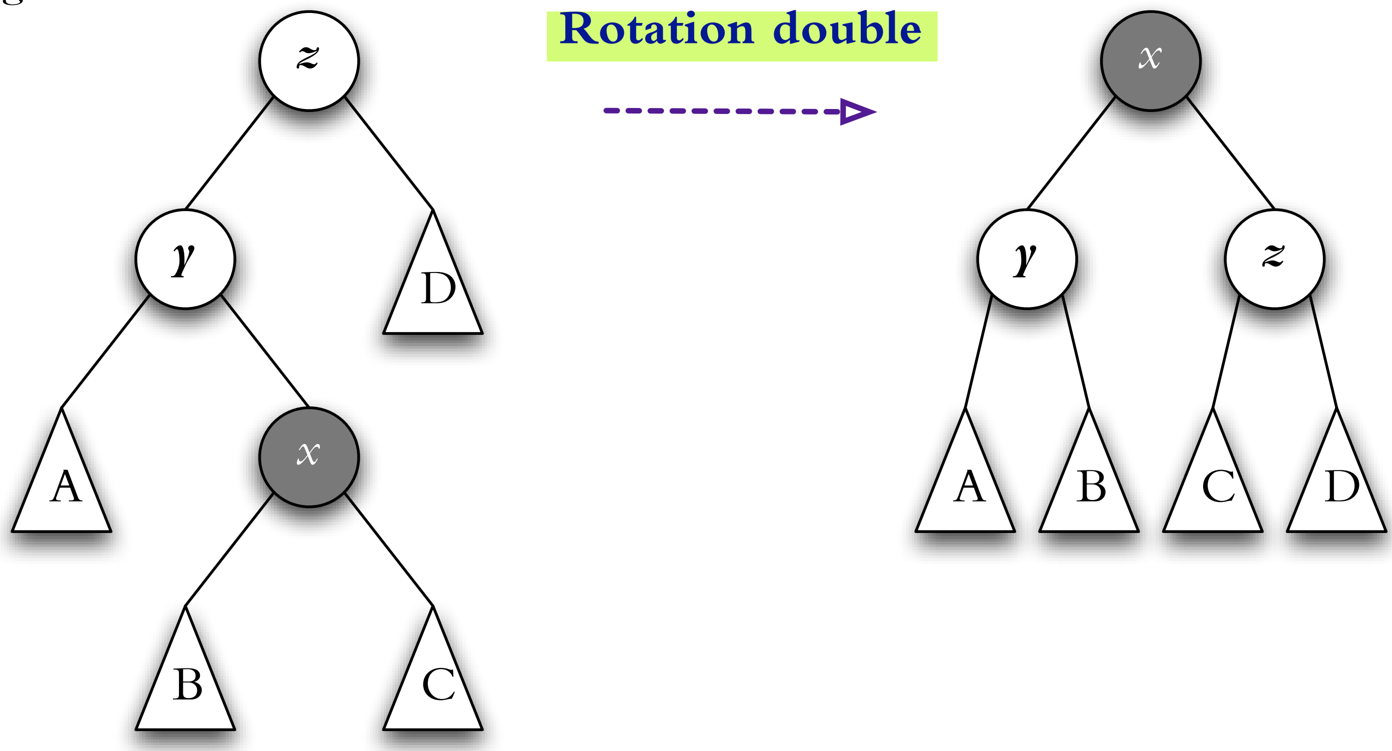
Zig et zag (cont)

Cas 2: *zig-zig*



Zig et zag (cont)

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.

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

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

Arbres rouges et noirs

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

Règles :

1. Pour chaque nœud x excepté la racine,
$$\mathsf{rang}(x) \le \mathsf{rang}(\mathsf{parent}(x)) \le \mathsf{rang}(x) + 1.$$
2. Pour chaque nœud x avec grand-parent $y = \mathsf{parent}(\mathsf{parent}(x))$,
$$\mathsf{rang}(x) < \mathsf{rang}(y).$$
3. Pour chaque feuille (null) on a $\mathsf{rang}(x) = 0$ et $\mathsf{rang}(\mathsf{parent}(x)) = 1$.

Arbres RN (cont)

D'où vient la couleur ?

Les nœuds peuvent être coloriés par rouge ou noir.

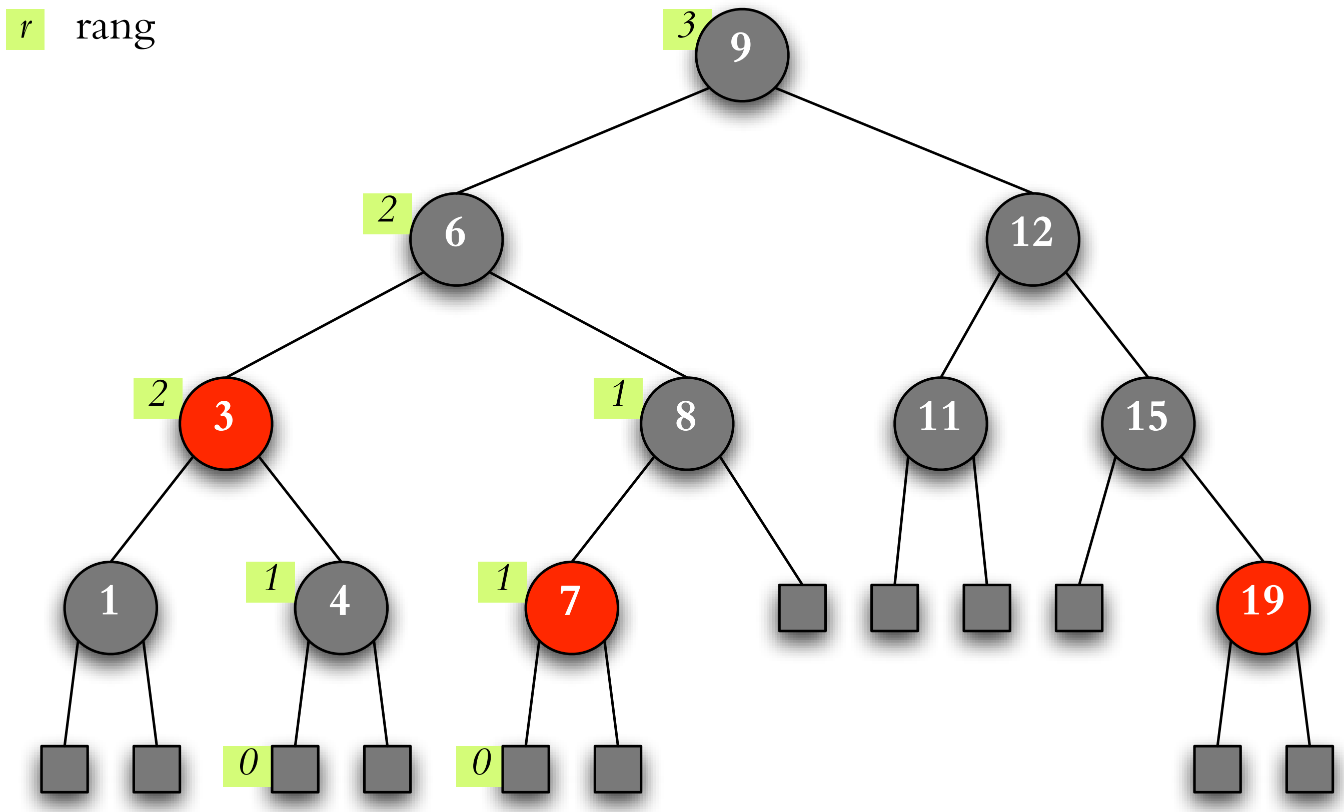
- si $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x)$, alors x est colorié par rouge
- si x est la racine ou $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x) + 1$, alors x est colorié par noir

Thm. Coloriage :
(0) chaque nœud est soit noir soit rouge
(i) chaque feuille (null) est noire
(ii) le parent d'un nœud rouge est noir
(iii) chaque chemin reliant un nœud à une feuille 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. $\square$

$\Rightarrow$ 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 une feuille. $\square$

Thm. Le nombre de déscendants internes de chaque nœud x est $\geq 2^{\mathsf{rang}(x)} - 1$.

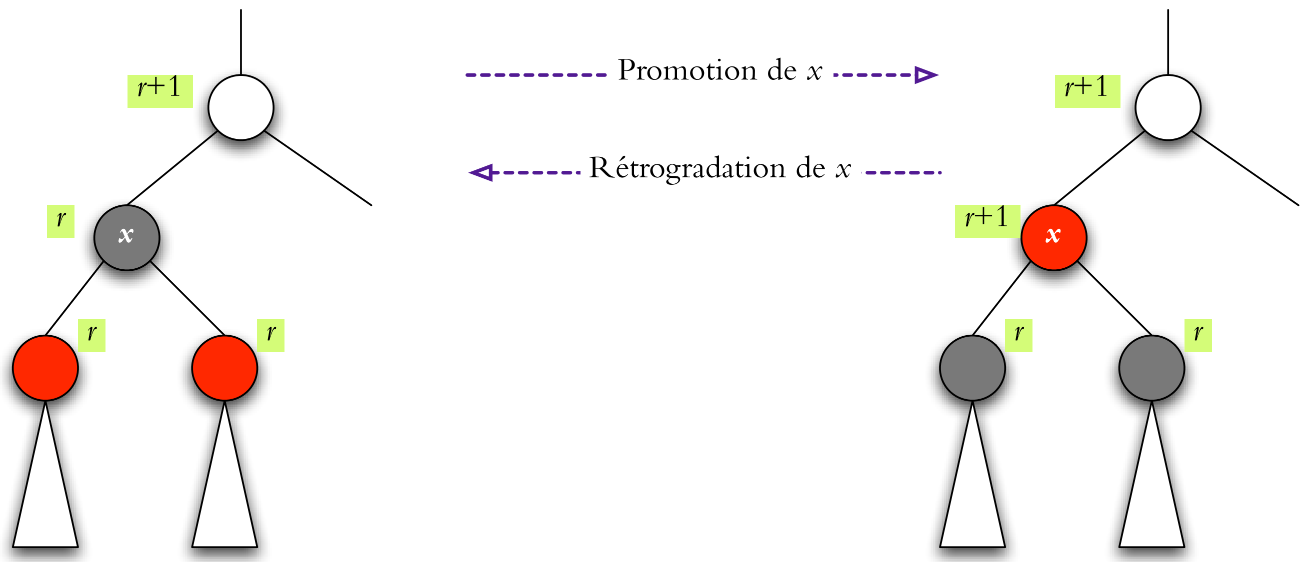
Preuve. Par induction. Le théorème est vrai pour une feuille x quand $\mathsf{rang}(x) = 0$. Supposons que le théorème est vrai pour tout x avec une hauteur $h(x) < k$. Considérons un nœud x avec $h(x) = k$ et ses deux enfants u, v avec $h(u), h(v) < k$. Par l'hypothèse d'induction, le nombre des descendants de x est $\geq 1 + (2^{\mathsf{rang}(u)} - 1) + (2^{\mathsf{rang}(v)} - 1)$. Or, $\mathsf{rang}(x) - 1 \leq \mathsf{rang}(u), \mathsf{rang}(v)$. $\square$

Arbres RN (cont)

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

Arbres RN — balance

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



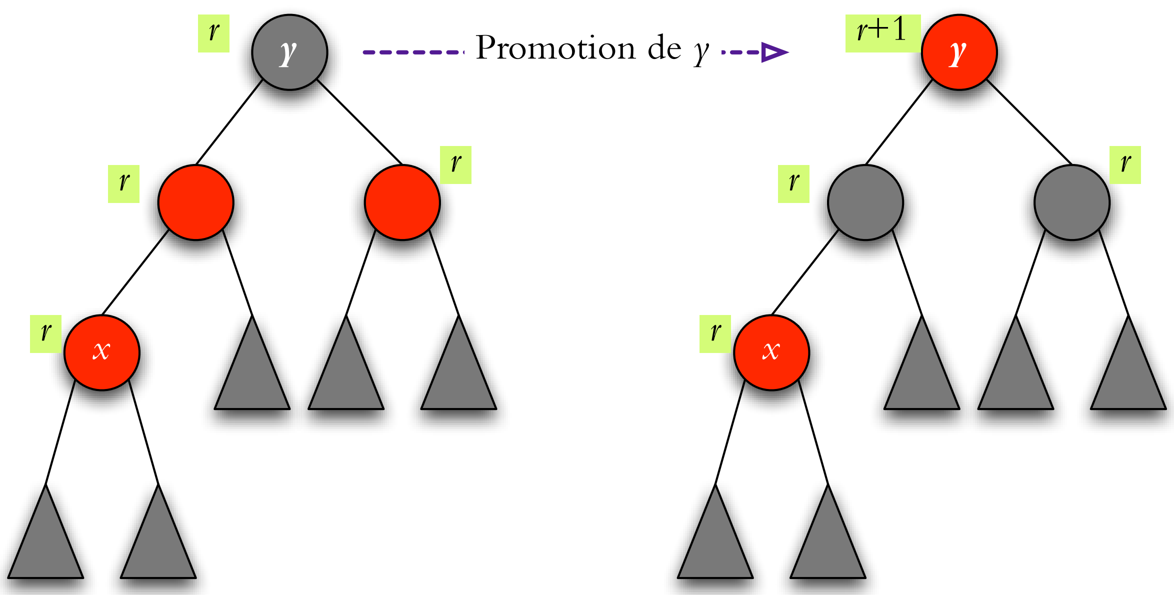
→ promotion/rétrogradation change la couleur d'un nœud et ses enfants
on peut promouvoir x ssi il est noir avec deux enfants rouges

Arbres RN — insertion

on insère x avec $\mathsf{rang}(x) = 1 \Rightarrow$ sa couleur est rouge

Test : est-ce que le parent de x est rouge ?
Si oui, on a un problème ; sinon, rien à faire

Solution : soit $y = \mathsf{parent}(\mathsf{parent}(x))$ le grand-parent — il est noir. Si y a deux enfants rouge, alors promouvoir y et retourner au test avec $x \leftarrow y$.



Arbres RN — insertion (cont)

On a fini les promotions et il y a toujours le problème que x est rouge, son parent est rouge aussi, mais l'oncle de x est noir.

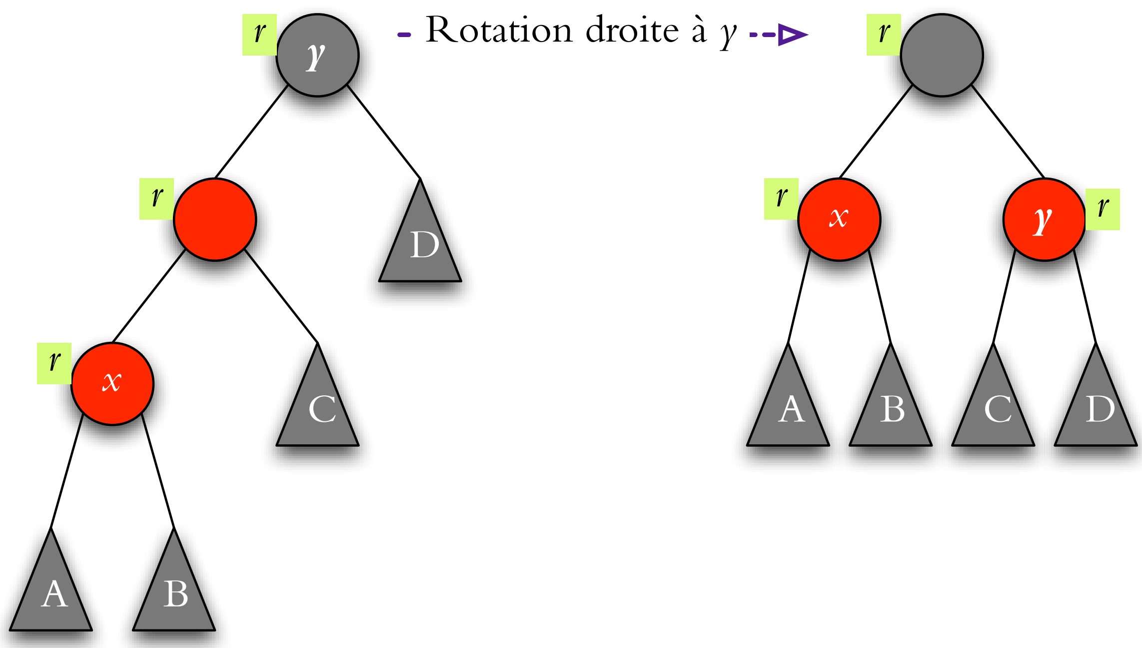
Deux cas :
Cas 1 quand x et $\mathsf{parent}(x)$ sont au même côté (enfants gauches ou enfants droits) — une rotation suffit

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

(enfin, c'est quatre cas : 1a, 1b, 2a et 2b)

Arbres RN — insertion (cont)

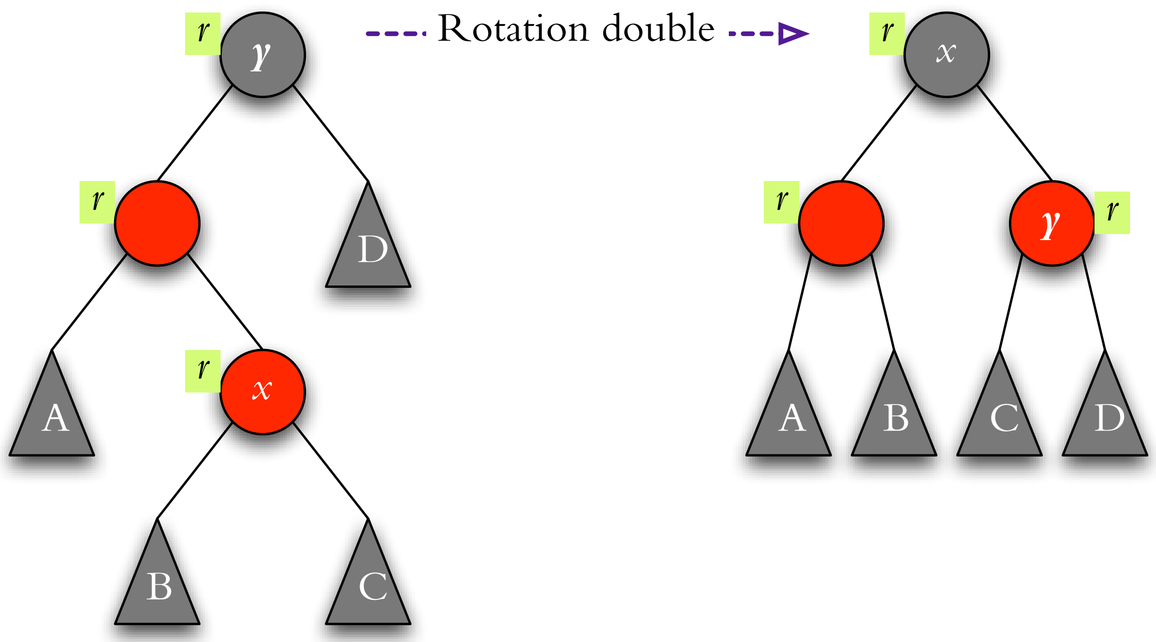
Cas 1a : x est rouge, son parent est rouge, son oncle est noir, et x et $\mathsf{parent}(x)$ sont des enfant gauches



Cas 1b (enfants droits) est symétrique

Arbres RN — insertion (cont)

Cas 2a : 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 2b (x est gauche et $\mathsf{parent}(x)$ est droit) est symétrique

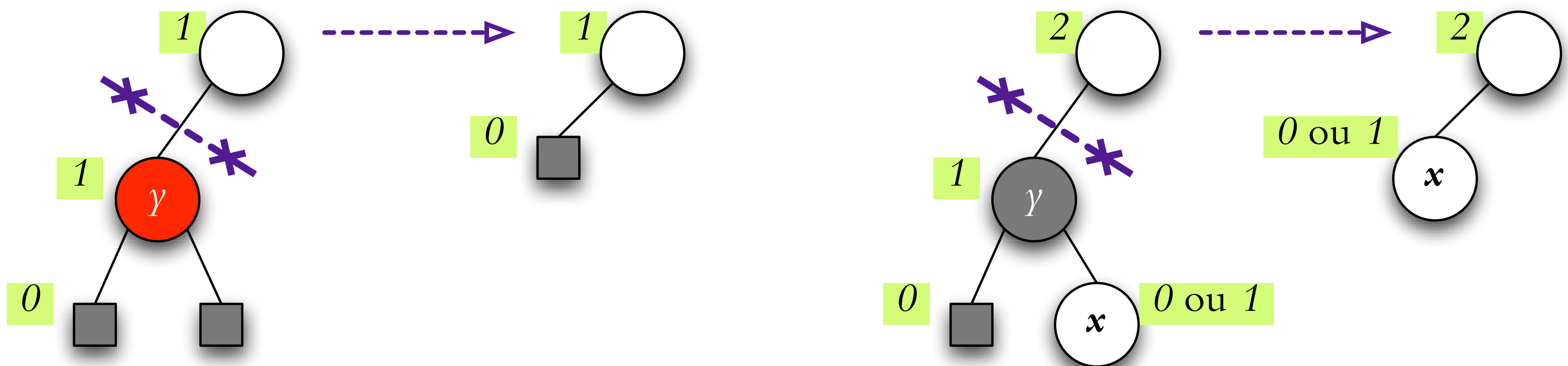
Arbres RN — suppression

Et suppression d'un nœud ?

Technique similaire : procéder comme avec l'arbre binaire de recherche, puis rétrogradations en ascendant vers la racine + $O(1)$ rotations (trois au plus) à la fin

Arbres RN — problème

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

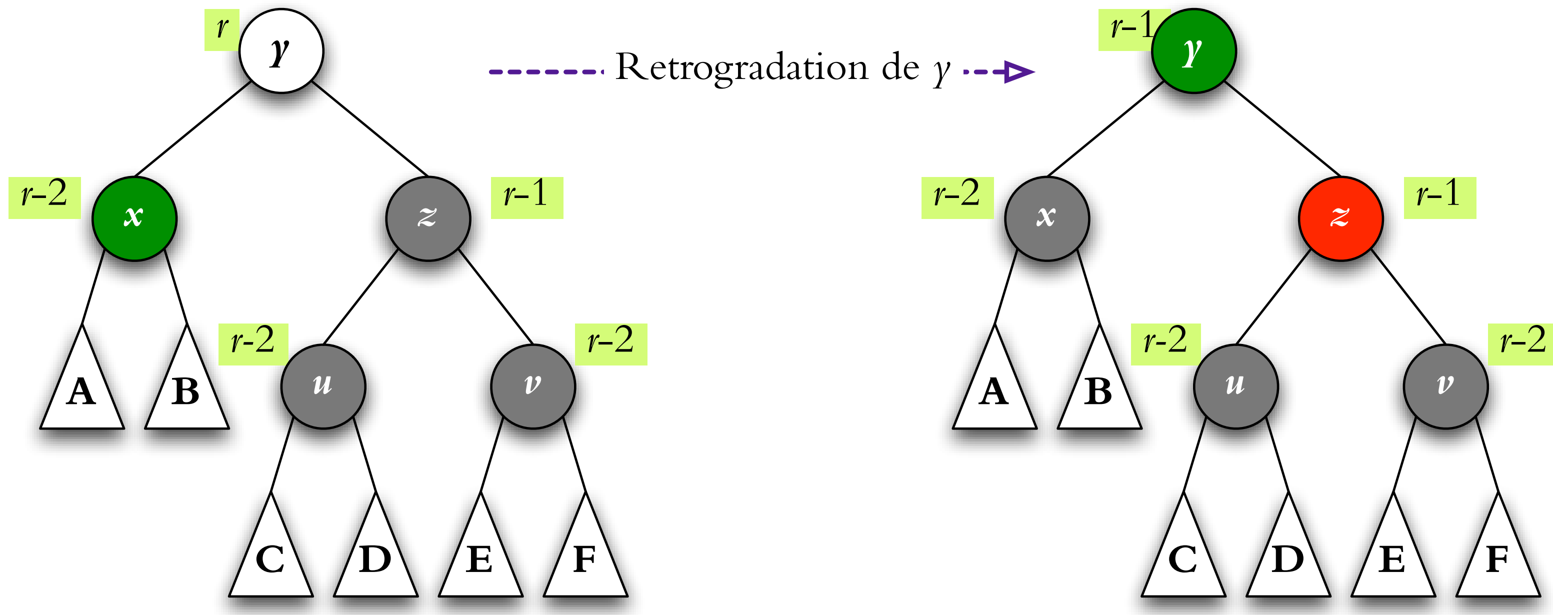
Cas 2 : nœud noir avec une sœur rouge

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

Arbres RN — suppression 1A

Cas 1A : sœur noire, neveux noirs

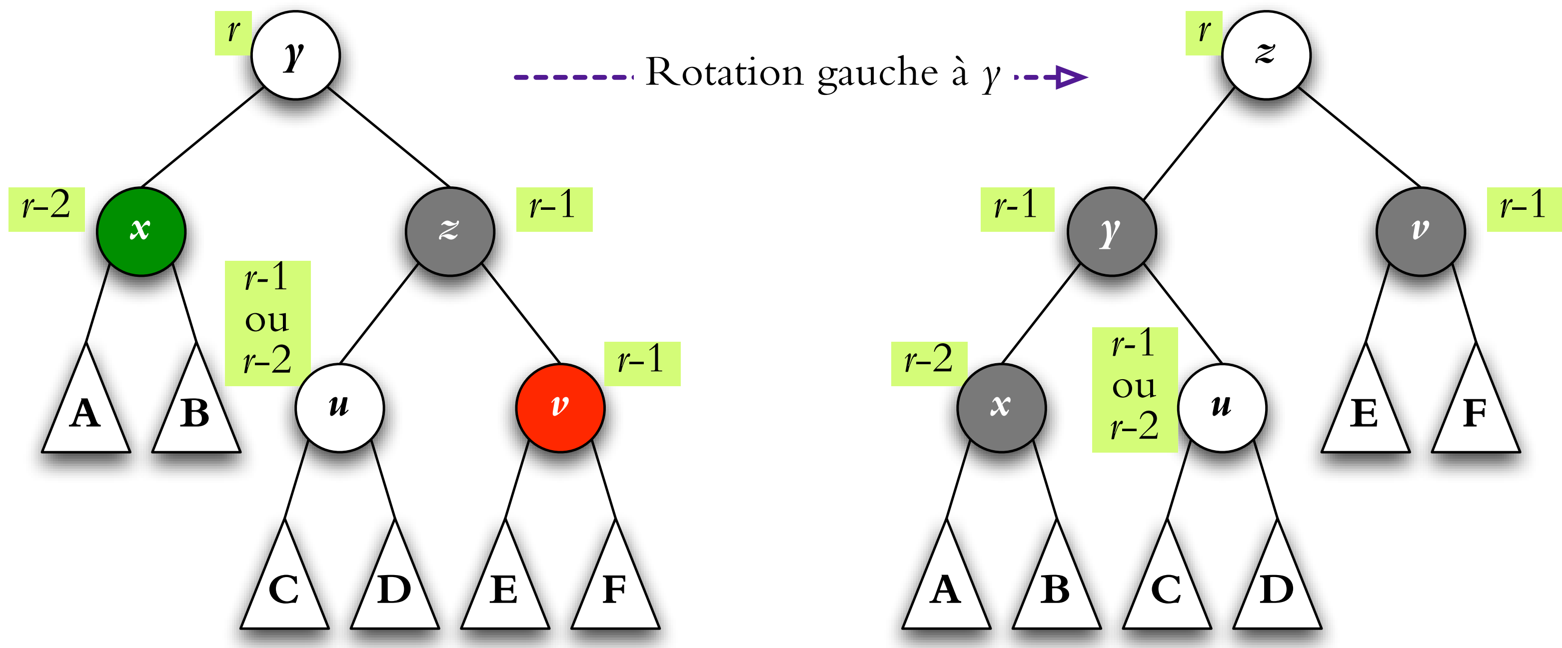
décrementer $\mathsf{rang}\Big(\mathsf{parent}(x)\Big)$: continuer avec $x \leftarrow \mathsf{parent}(x)$ en cas 0,1 ou 2.



Arbres RN — suppression 1B

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

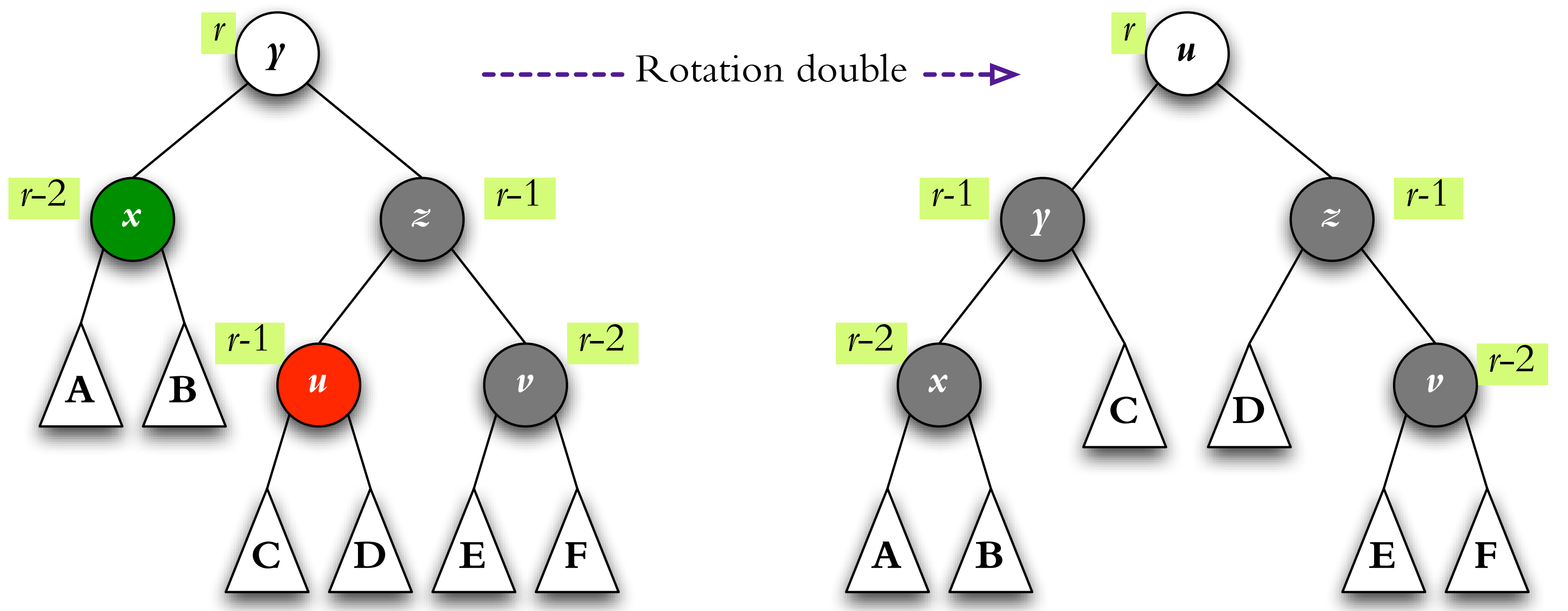
faire une rotation simple et arrêter



Arbres RN — suppression 1C

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

faire une rotation double et arrêter

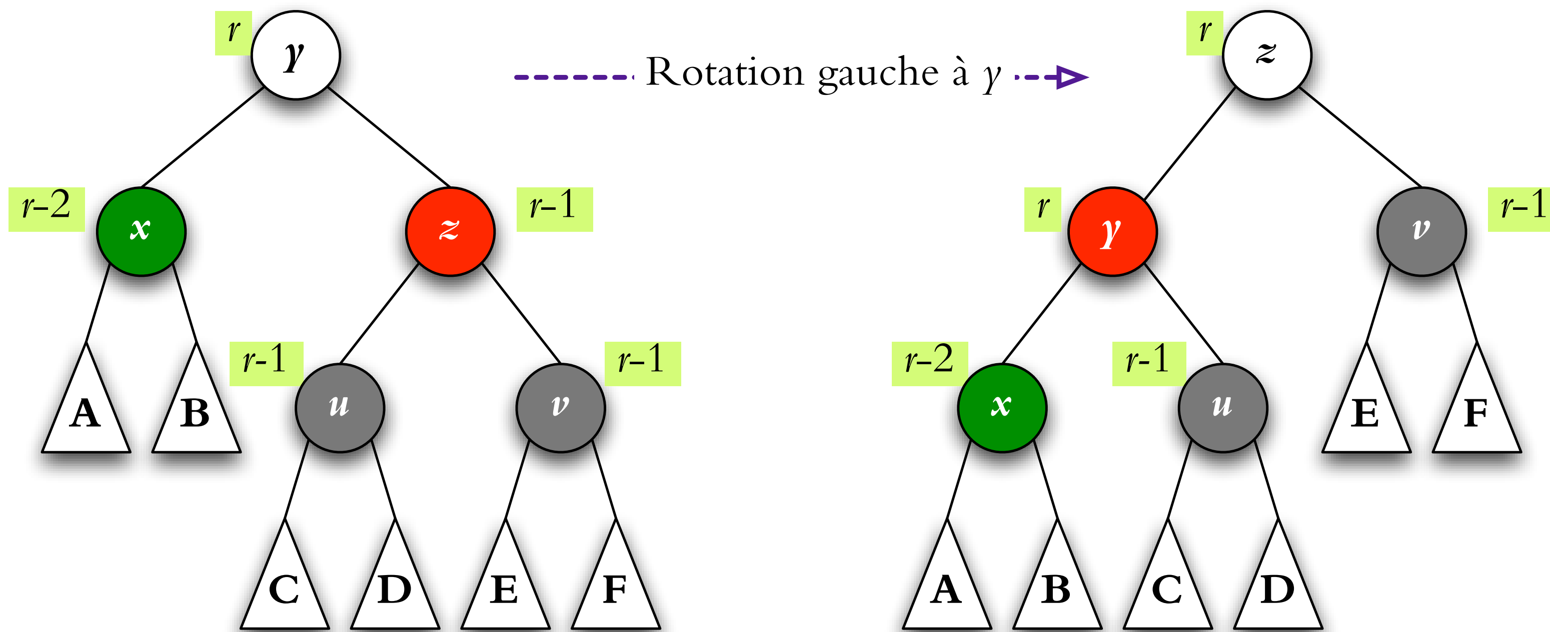


(quand $\mathsf{rang}(v) = r - 1$, on peut toujours faire cette rotation mais cas 1B est plus rapide)

Arbres 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)

Arbres RN — efficacité

Un arbre rouge et noir avec n nœuds internes et hauteur $h \in O(\log n)$.

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

Suppression : $O(\log n)$

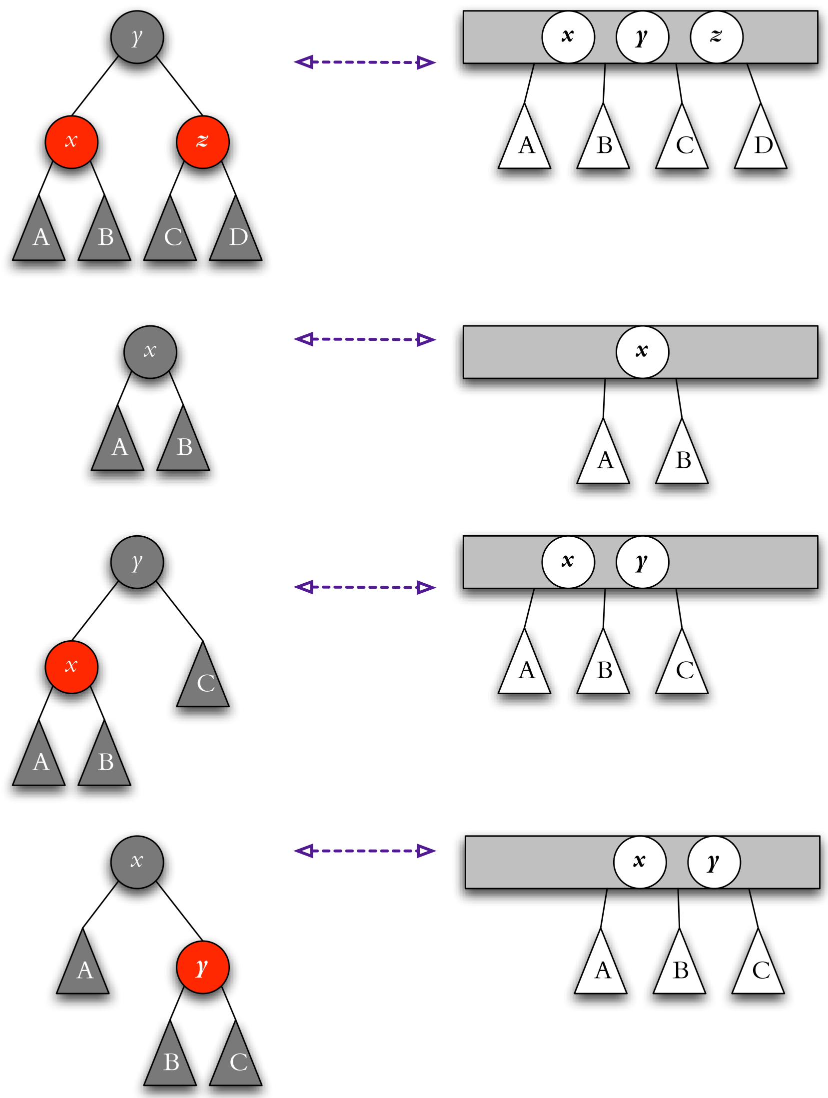
Usage de mémoire : il suffit de stocker la couleur (1 bit) de chaque nœud interne (astuce pour épargner un bit par nœud : échanger les pointeurs gauche $\leftrightarrow$ droit pour les nœuds noirs — couleur testée par la comparaison des clés aux enfants)

Arbre 2-3-4

Arbre **2-3-4** : c'est un arbre de recherche *non-binaire* où chaque nœud peut avoir 2, 3 ou 4 enfants et stocke 1,2 ou 3 valeurs
toutes les feuilles sont au même niveau

Équivaut à l'arbre rouge et noir : fusionner les nœuds rouges et leurs parents noirs.

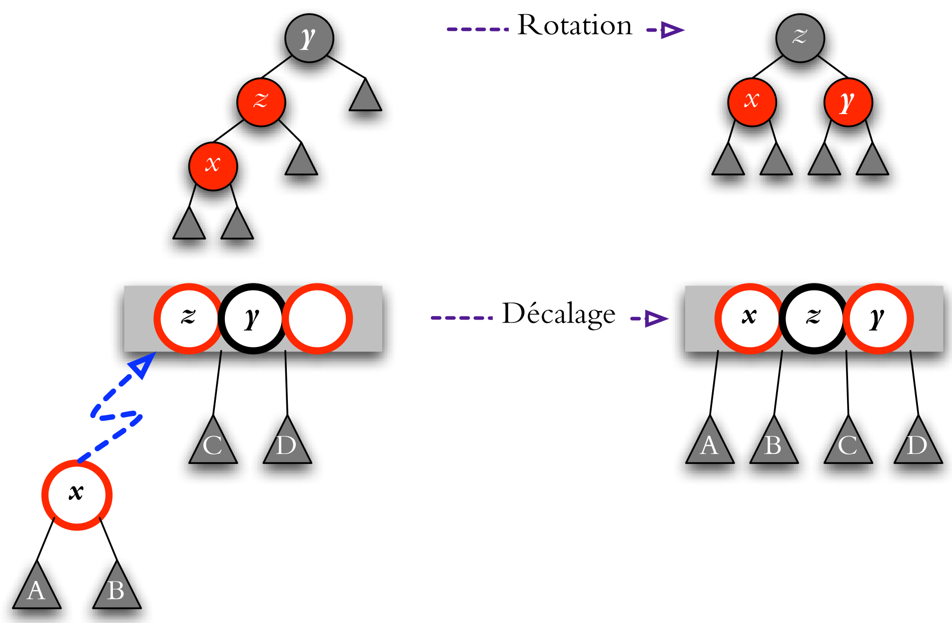
Transformation entre arbres RN et 2-3-4



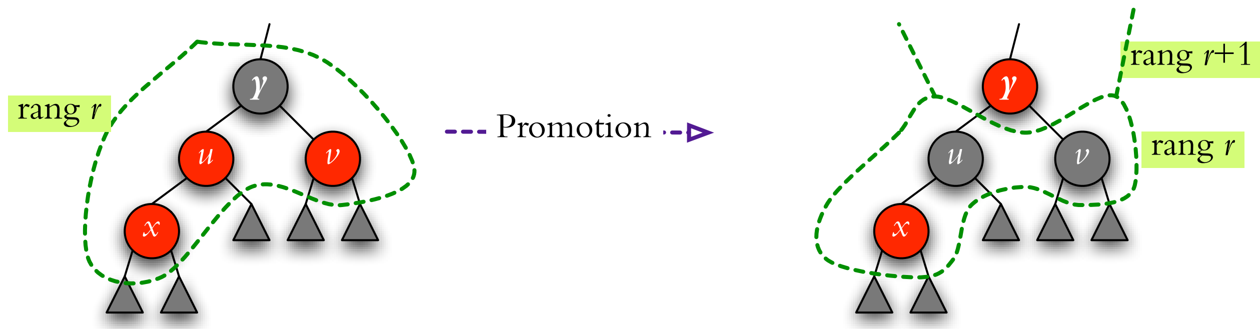
Arbre RN ↔ arbre 2-3-4

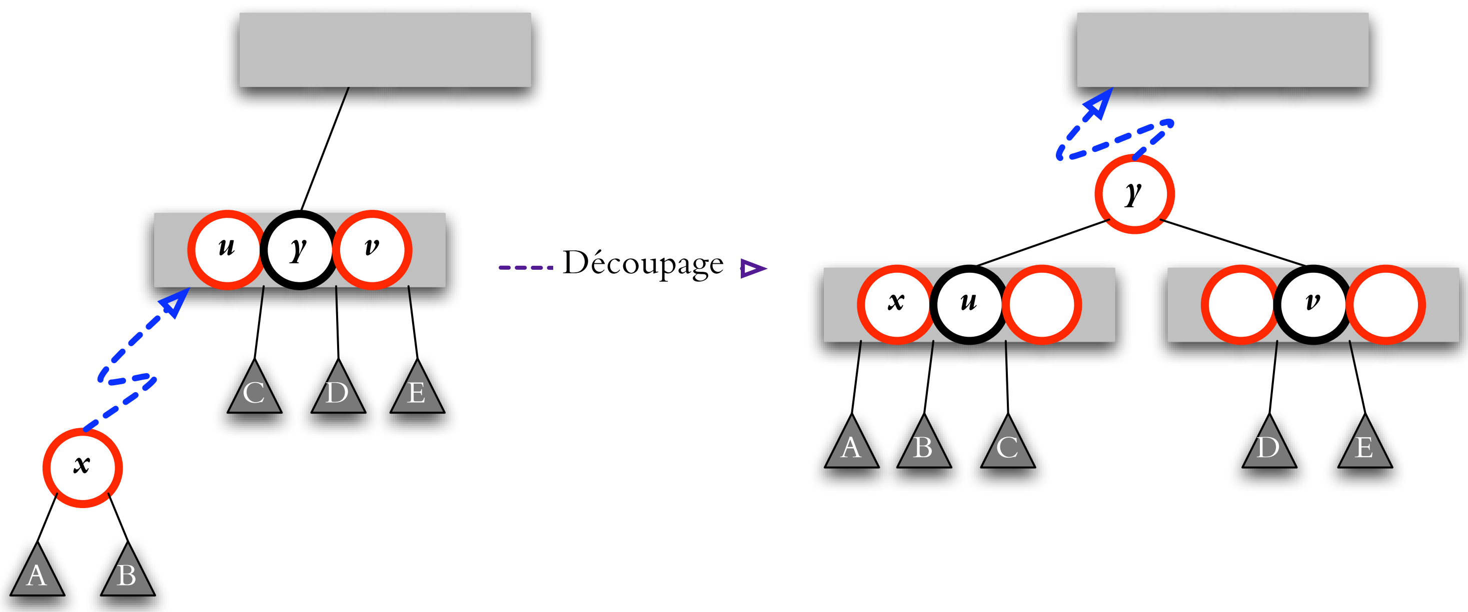
Qu'est-ce qui se passe lors d'une insertion ?
On crée un nœud rouge : promotions+rotations en ascendant vers la racine

Rotation : nœud noir avec un enfant rouge et son grand-enfant rouge transformé en un nœud noir avec deux enfants rouges



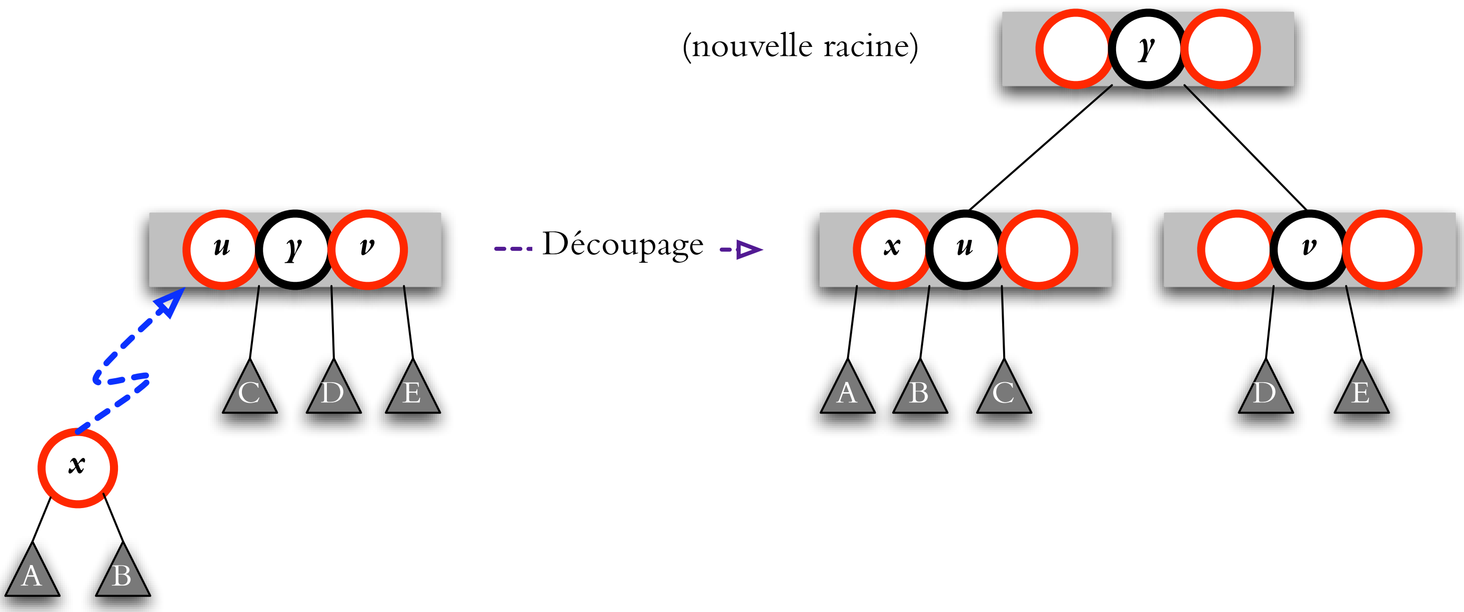
Arbre RN ↔ arbre 2-3-4 (promotions)





Arbre RN $\leftrightarrow$ arbre 2-3-4 (cont)

Cas spécial : promotion de la racine



$\Rightarrow$ la hauteur de l'arbre croît par le découpage de la racine

(arbre binaire de recherche : la hauteur croît par l'ajout de feuilles)