Arbres

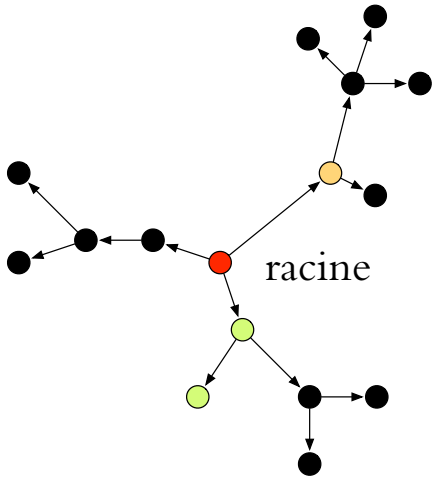
Arbres — terminologie

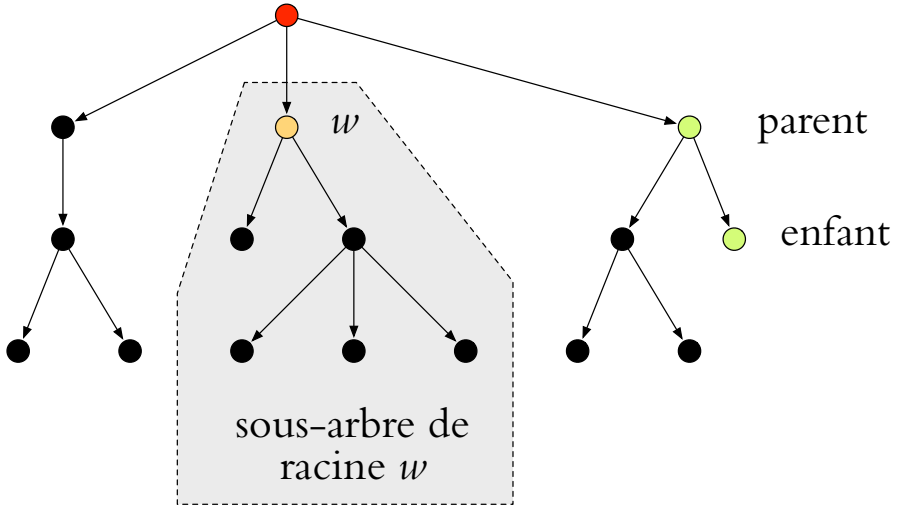
Graphe : sommet et arêtes

Arbre (en théorie des graphes) : graphe non-orienté, connexe et acyclique

Arbre raciné ou **arborescence** (en théorie des graphes) : graphe orienté, et connexe avec un sommet spécial, la **racine**, dans lequel il y a 1 chemin simple de la racine à chaque sommet.

- orientation des arêtes : relations parent-enfant
- u est dans le sous-arbre enraciné à w ssi w est sur le chemin de la racine à u





Arbres — terminologie

Ancêtre : w est l'ancêtre de u ssi u est dans le sous-arbre de racine w

Descendant : u est un descendant de w ssi u est dans le sous-arbre de racine w

Dans des graphes orientés (y inclut les arborescences) les sommets s'appellent aussi des nœuds, et les arêtes s'appellent aussi des arcs

Degré d'un nœud : nombre d'arcs sortants (ou nombre d'enfants)

Nœud externe ou feuille : aucun arc sortant (aucun enfant)

Nœud interne : tous les autres (au moins 1 enfant)

Hauteur et profondeur

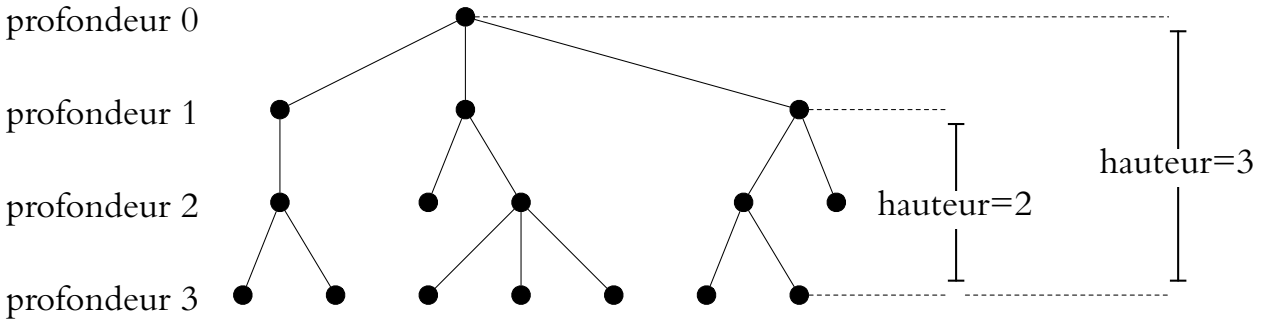
Profondeur d'un nœud u : longueur du chemin de la racine à u

Hauteur d'un nœud u : longueur du chemin à la feuille la plus distante dans le sous-arbre de u

Hauteur de l'arbre : hauteur de la racine

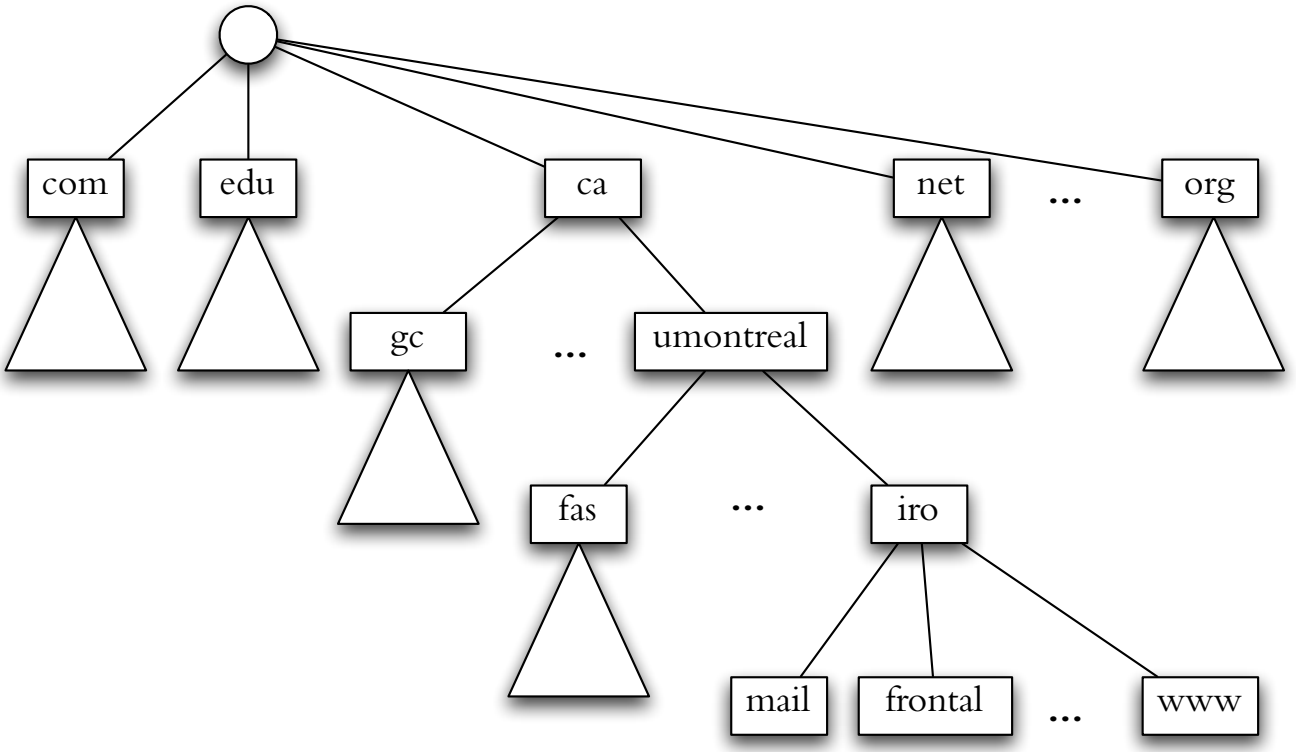
Profondeur de l'arbre : profondeur maximale

Thm. profondeur de l'arbre = hauteur de l'arbre.



Arborescence — exemples

Domaines internet :



Packages en Java, répertoires sous Unix, . . .

Arbre numeroté

Arbre numeroté : les enfants d'un nœud sont étiquetés par des entiers positifs distincts.

i-ème enfant absent : si aucun enfant n'est étiqueté par i

Arité k : ssi tous les enfants avec étiquettes $> k$ sont absents.

Arbre binaire : arbre numeroté d'arité 2

enfant gauche ou droit : enfant étiqueté par 1 ou 2

frères ou sœurs : nœuds avec le même parent

Arbre numeroté (cont.)

Définition altérnative par récurrence :

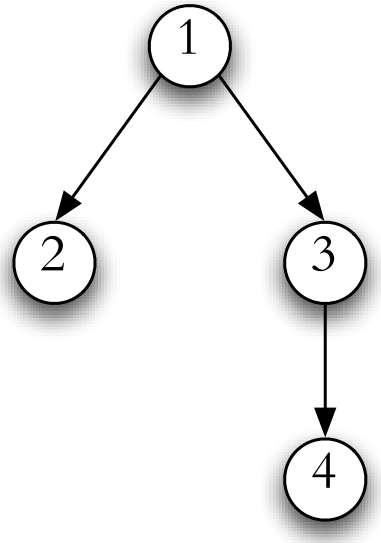
Déf. *Un arbre k-aire T est une structure définie sur un ensemble fini de nœuds qui*

1. *ne contient aucun nœud,* ***ou***
2. *est composé de $(k+1)$ ensembles de nœuds disjoints : un nœud* ***racine*** *r, et les arbres k-aires $T_1, T_2, \ldots, T_k$.*

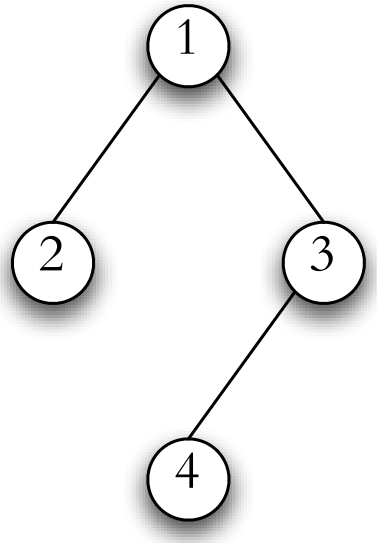
(En 2, la racine de T_i quand T_i est non-vide est l'enfant de r étiqueté par i.)

Arbre numeroté (cont.)

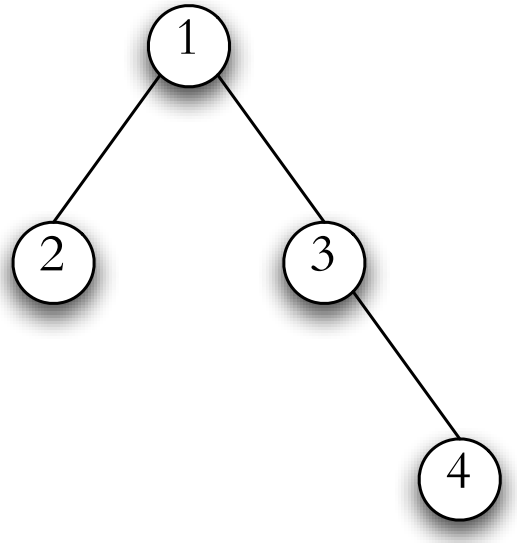
Attention : l'ordre des enfants est important dans un arbre numeroté



arborescence
(l'ordre des enfants
n'est pas important)



arbre binaire
(«4» est l'enfant gauche)



arbre binaire
(«4» est l'enfant droit)

Implantation d'un arbre numeroté

Arbre = ensemble d'objets représentant de nœuds + relations parent-enfant

En général, on veut retrouver facilement le parent et les enfants de n'importe quel nœud

Approche Java :

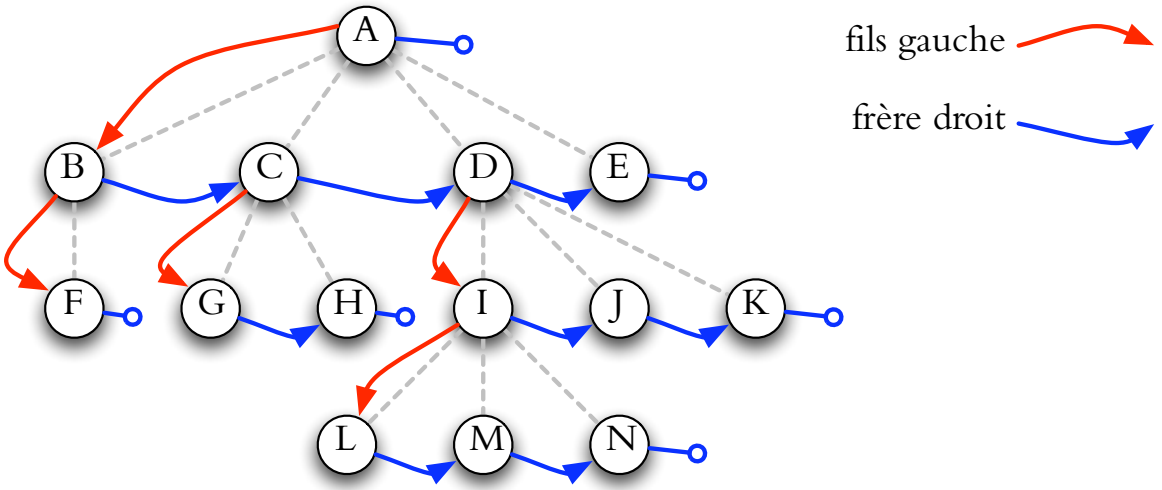
```
public class TreeNode {
   TreeNode parent;
   TreeNode enfant_gauche;
   TreeNode enfant_droit;
   ...
}
```

Si arbre k-aire, alors on peut avoir `TreeNode[] enfants`
avec `enfants.length` $= k$.

Implantation (cont)

Si l'arité de l'arbre n'est pas connu en avance (ou la plupart des nœuds ont très peu d'enfants), on peut utiliser

1. une liste pour stocker les enfants
2. représentation **fils-gauche**, **frère-droit** (*first-child, next-sibling*)



Parcours des arbres

Dans un parcours, tous les nœuds de l'arbre sont visités.

Déf. Dans un parcours préfixe (*preorder traversal*), chaque nœud est visité avant que ses enfants soient visités.

Déf. Dans un parcours postfixe (*postorder traversal*), chaque nœud est visité après que ses enfants sont visités.

Parcours préfixe et postfixe

```
   Algo PARCOURS-PRÉFIXE(x)
1  if x ≠ null then
2      imprimer de x
3      for i ← 1, ..., k do PARCOURS-PRÉFIXE(enfant(x, i))
```

```
   Algo PARCOURS-POSTFIXE(x)
1  if x ≠ null then
2      for i ← 1, ..., k do PARCOURS-POSTFIXE(enfant(x, i))
3      imprimer de x
```

(enfant(x, i) donne l'enfant de x étiqueté par i — s'il n'y en a pas, alors null)

Maintenant PARCOURS-PRÉFIXE(racine) va imprimer tous les nœuds dans l'arbre dans l'ordre préfixe.

Parcours infixe

On peut parcourir un arbre binaire aussi dans l'ordre infixe

Déf. Dans un parcours infixe (*inorder traversal*), chaque nœud est visité après son enfant gauche mais avant son enfant droit.

```
   Algo PARCOURS-INFIXE(x)
1  if x ≠ null then
2      PARCOURS-INFIXE(gauche(x))
3      imprimer x
4      PARCOURS-INFIXE(droit(x))
```

Arbre binaire — notation

Dans nos arbres binaires, chaque nœud a une **valeur** (ou clé).

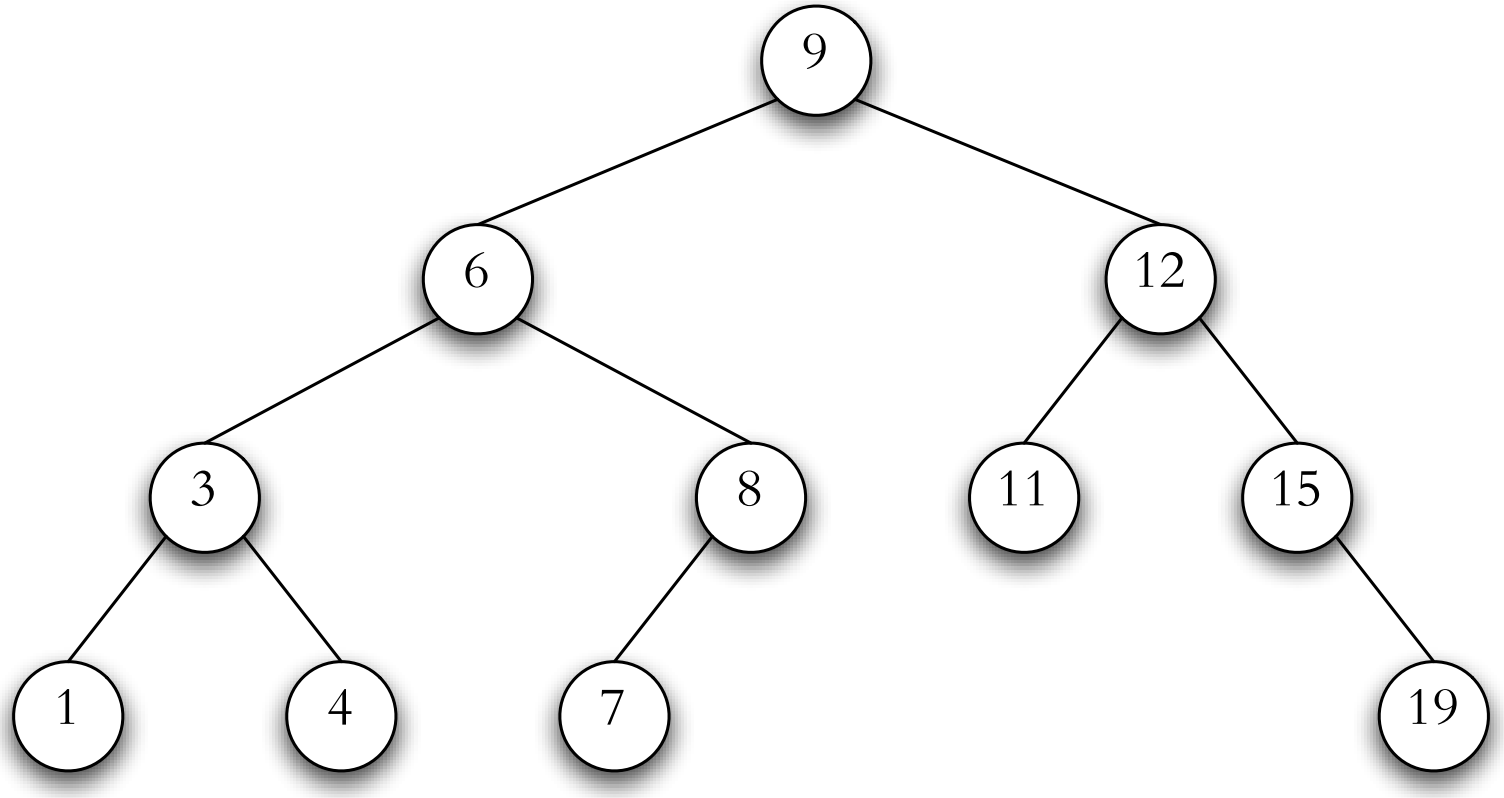
Objets pour notre ADT : nœuds (ensemble $\mathcal{N}$) et arbres binaires (ensemble $\mathcal{T}$)

Opérations sur 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)
$\mathsf{val}(x)$ pour la valeur 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 valeur.

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

Arbre de recherche — exemple



Arbre de recherche (cont)

À l'aide d'un arbre de recherche, on peut implanter des opérations sur les ensembles (éléments = valeurs des nœuds) 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.

Min et max

```
    Algo MIN() // trouve la valeur minimale dans l'arbre
1   x ← racine; y ← null
2   while x ≠ null do
3       y ← x; x ← gauche(x)
4   retourner y
```

```
    Algo MAX() // trouve la valeur maximale dans l'arbre
1   x ← racine; y ← null
2   while x ≠ null do
3       y ← x; x ← droit(x)
4   retourner y
```

Recherche

```
     Algo FIND(x, v) // trouve la valeur v dans le sous-arbre de x
F1   if x = null ou v = val(x) alors retourner x
F2   if v < val(x)
F3   then retourner FIND(gauche(x), v)
F4   else retourner FIND(droit(x), v)
```

Maintenant, FIND(racine, v) retourne
- soit un nœud dont la valeur 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 FIND(x, v) // trouve la valeur v dans le sous-arbre de x
F1   while x ≠ null et v ≠ val(x) do
F2       if v < val(x)
F3       then x ← gauche(x)
F4       else x ← droit(x)
F5   retourner x
```

Recherche — efficacité

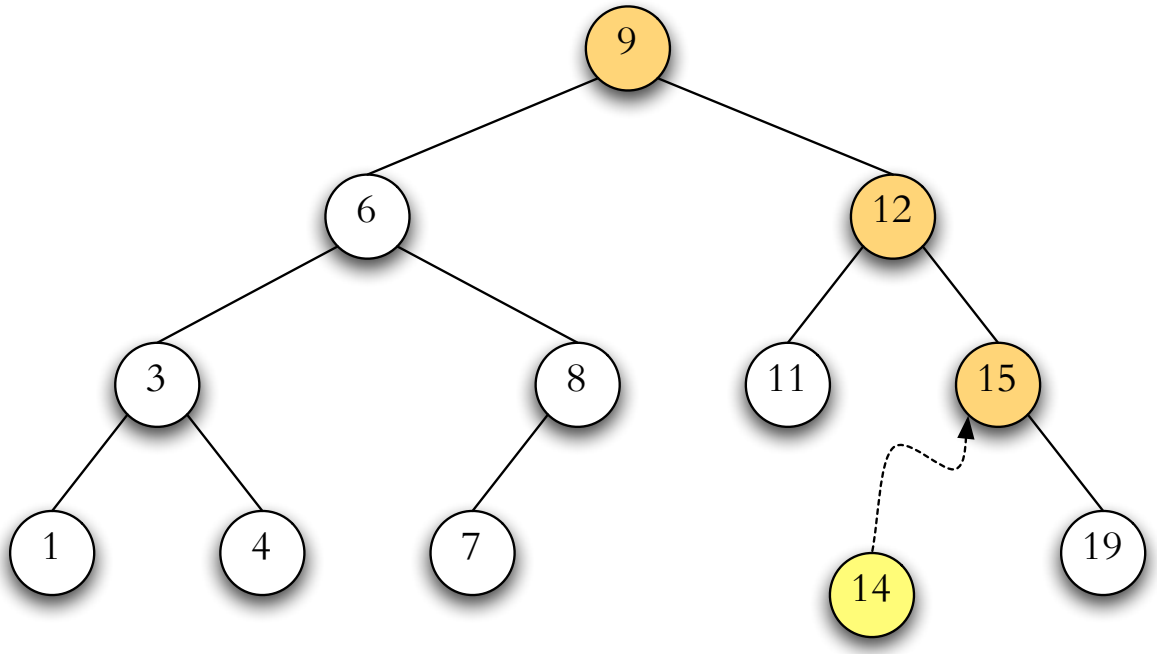
Dans un arbre binaire de recherche de hauteur h :

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

Insertion

On veut insérer une valeur v

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



insertion de «14»

Insertion (cont.)

insertion — pas de valeurs dupliquées

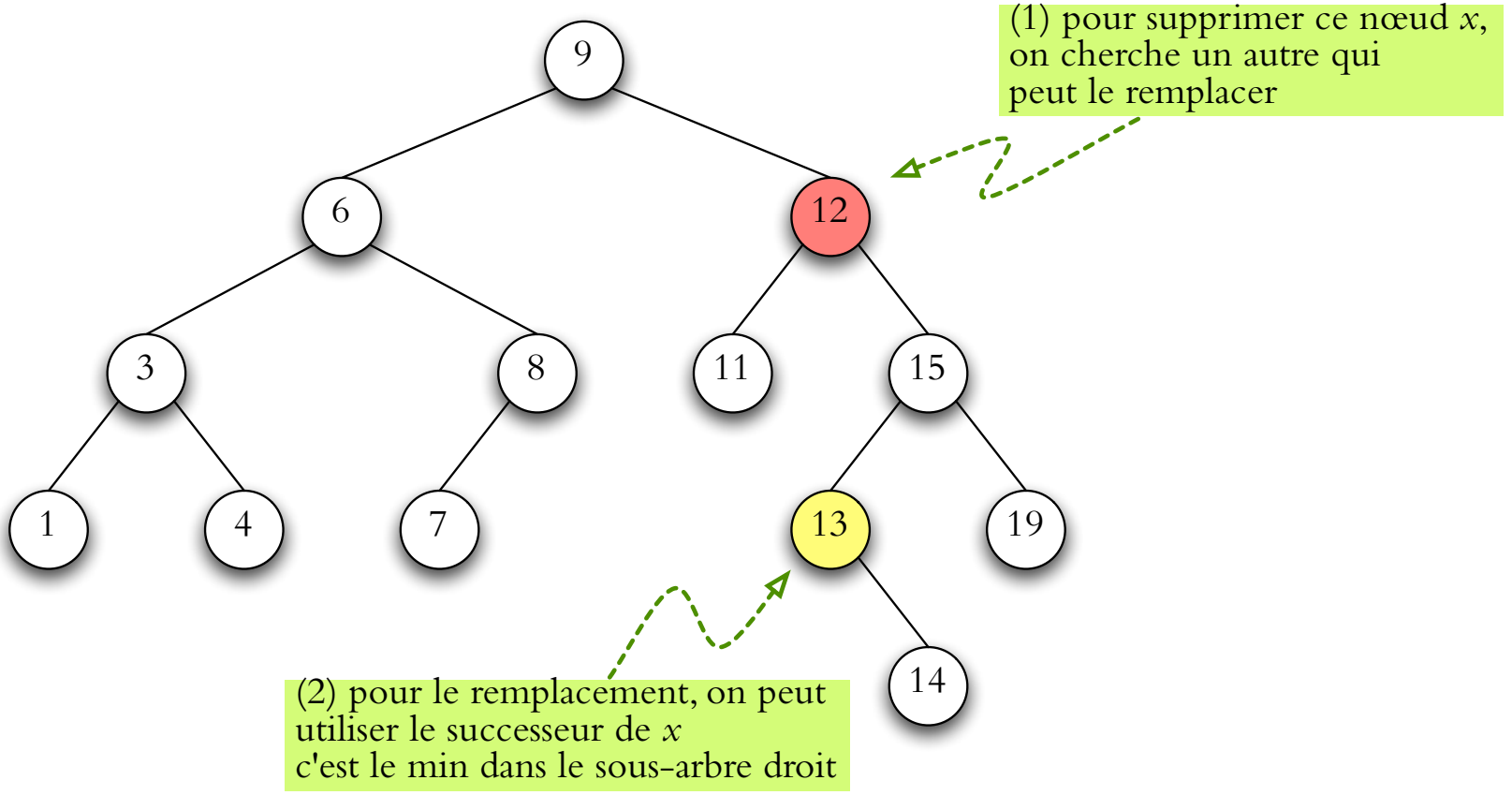
```
     Algo INSERT(v) // insère la valeur v dans l'arbre
 I1  x ← racine
 I2  if x = null then initialiser avec une racine de valeur v et retourner
 I3  while vrai do // (conditions d'arrête testées dans le corps)
 I4      if v = val(x) then retourner // (pas de valeurs dupliquées)
 I5      if v < val(x)
 I6      then if gauche(x) = null
 I7          then attacher nouvel enfant gauche de x avec valeur v et retourner
 I8          else x ← gauche(x)
 I9      else if droit(x) = null
I10          then attacher nouvel enfant droit de x avec valeur v et retourner
I11          else x ← 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 :

INSERT(v) prend $O(h)$
suppression d'un nœud 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.

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 : l'hauteur est de $O(\log n)$ *en moyenne* (permutations aléatoires de $\{1, 2, \ldots, n\}$)

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

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

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 le 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 $\frac{D(n)}{n} \in O(\log n)$.

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. □

Performance moyenne (cont.)

Par le lemme précédent,

$$\begin{aligned} nD(n) - (n-1)D(n-1) &= \left(n(n-1) + 2\sum_{i=0}^{n-1} D(i)\right) \\ &\quad - \left((n-1)(n-2) + 2\sum_{i=0}^{n-2} D(i)\right) \\ &= 2(n-1) + 2D(n-1). \end{aligned}$$

D'où on a

$$\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2n-2}{n(n+1)} = \frac{D(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}.$$

Avec $E(n) = \frac{D(n)-2}{n+1}$, on peut écrire

$$E(n) = E(n-1) + \frac{2}{n+1}.$$

Performance moyenne (cont.)

Donc,

$$E(n) = E(0) + \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n+1}$$
$$= \frac{D(0) - 2}{1} + 2(H_{n+1} - 1) = 2H_{n+1} - 4$$

où $H_n = \sum_{i=1}^{n} 1/i$ est le n-ième nombre harmonique.

En retournant à $D(n) = 2 + (n+1)E(n)$, on a alors

$$D(n) = 2(n+1)H_{n+1} - 4n - 2 < 2nH_{n+1}$$

Donc la profondeur moyenne

$$\frac{D(n)}{n} < 2H_{n+1} \in O(\log n).$$

(En fait, on a $D(n)/n < 2 \cdot \ln n$.)

Performance moyenne (cont)

L'analyse assume un arbre aléatoire : l'arbre est produit par l'insertion des valeurs $1, 2, \ldots, n$ selon une permutation aléatoire.

Et les suppressions ?

Dans l'approche présentée, on a utilisé le successeur d'un nœud u quand u est à supprimer et il a deux enfants.
$\rightarrow$ peu à peu l'arbre commence à pencher vers la gauche avec les suppressions.

Mais l'effet n'est pas visible après $o(n^2)$ paires d'insertion-deletion...

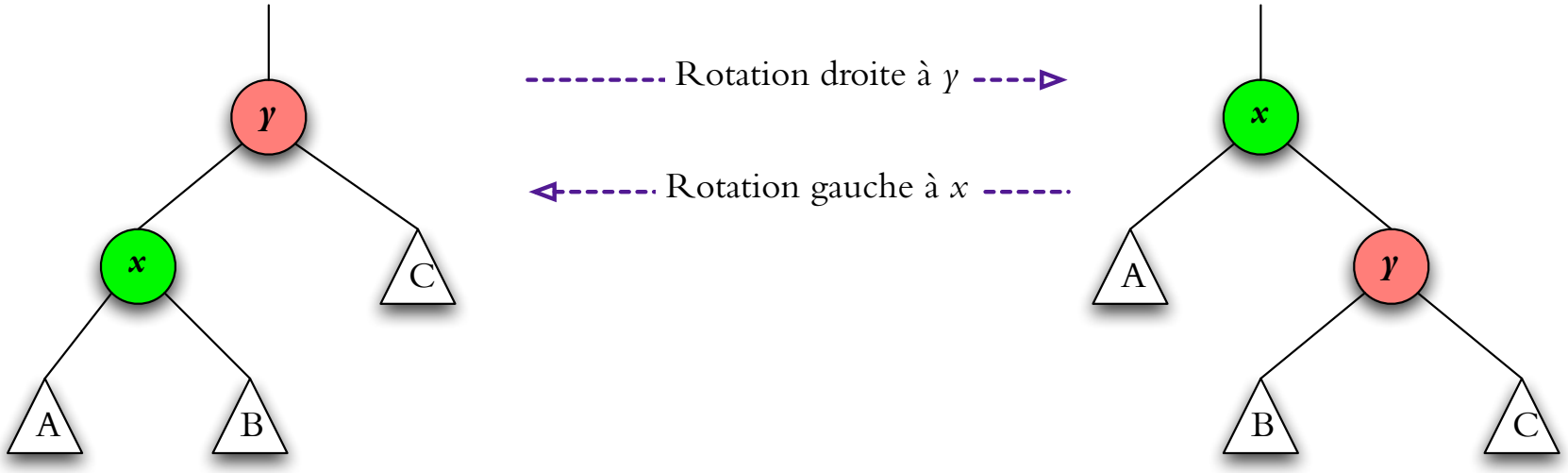
Arbres balancés

Arbres balancés (ou 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 prendent seulement $O(1)$



Arbres AVL

AVL : Adelson-Velsky et Landis (1962)

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

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

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

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

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

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. □

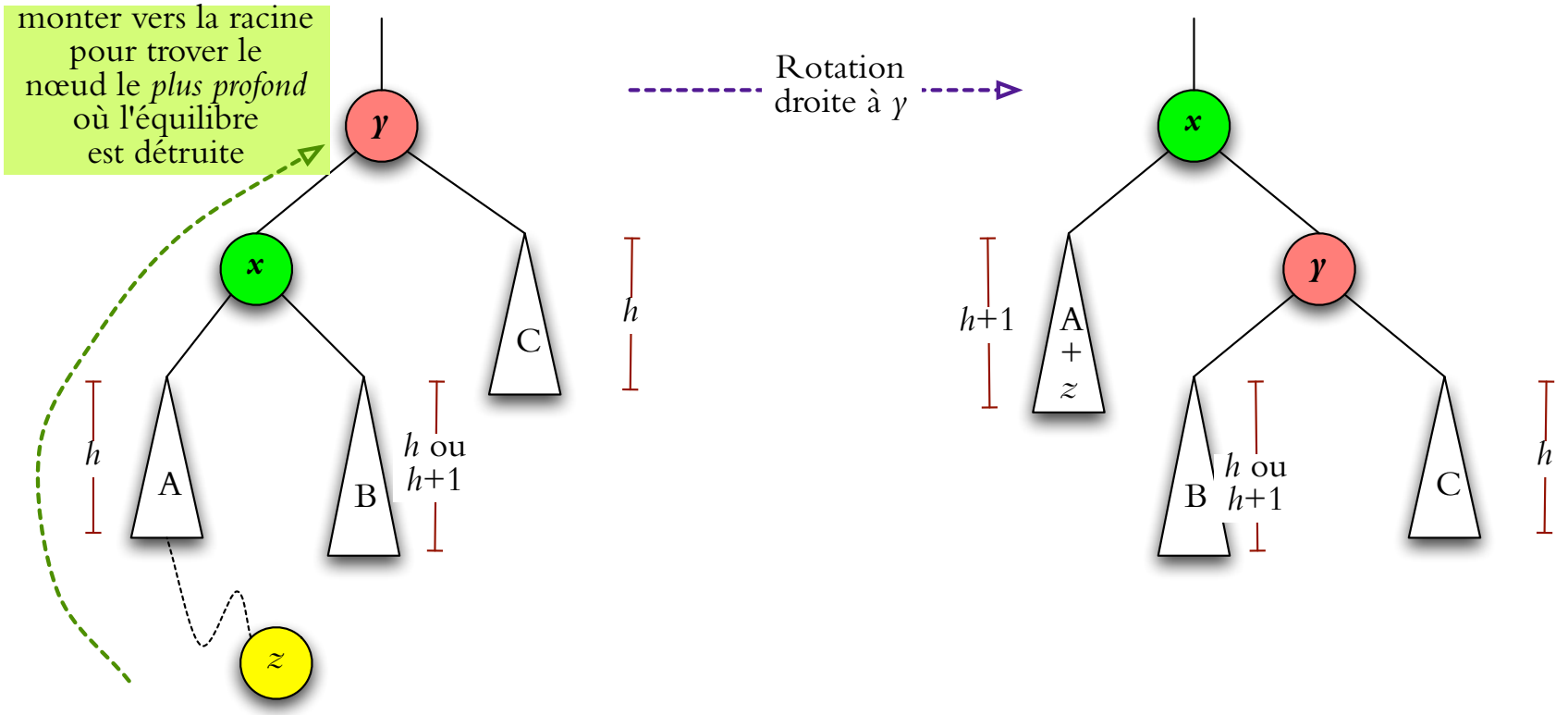
Hauteur d'un arbre AVL (cont)

Donc un arbre AVL sur n nœuds est de l'hauteur

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

Insertion dans arbre AVL

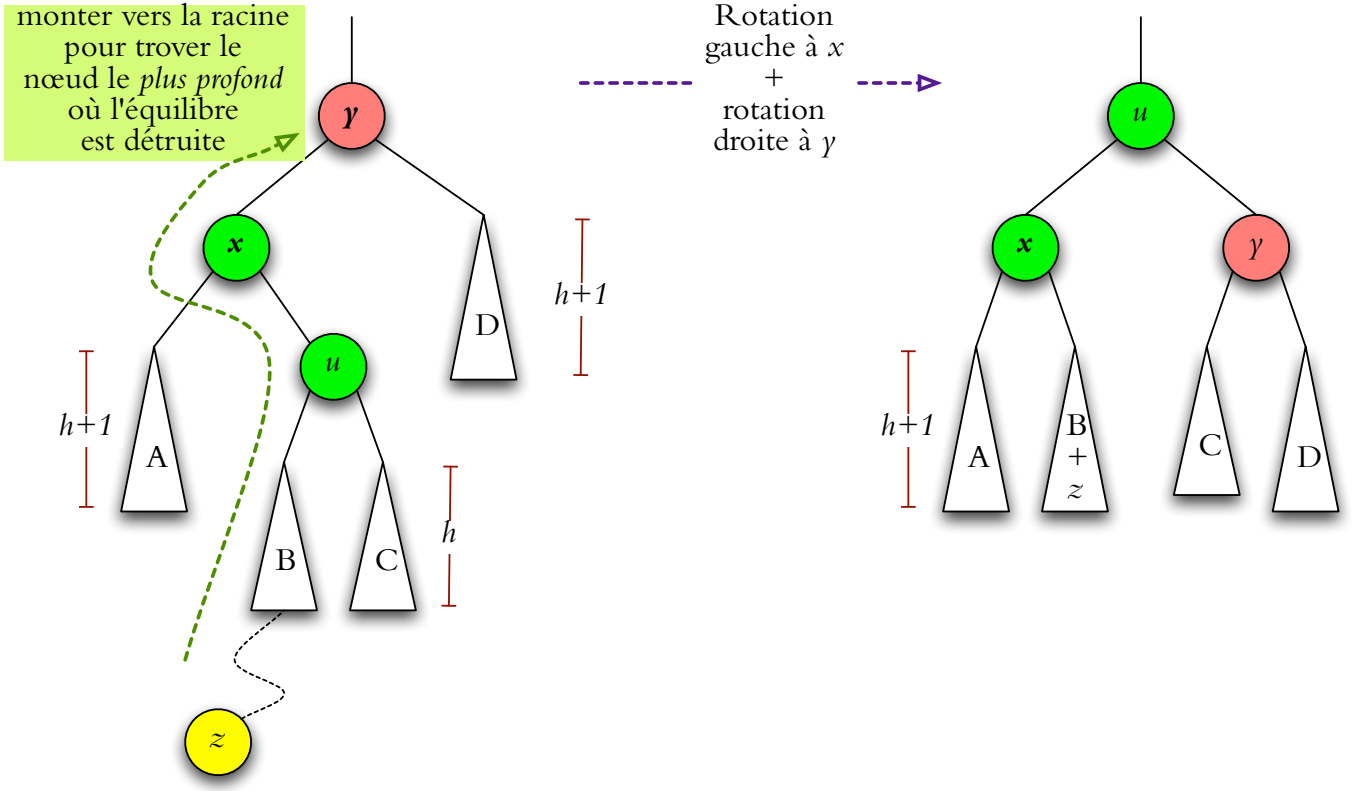
Insertion dans le sous-arbre gauche d'un enfant gauche : une rotation si nécessaire



(cas symétrique si sous-arbre droit d'un enfant droit)

Insertion dans arbre AVL (cont)

Sous-arbre droit d'un enfant gauche :



(cas symétrique si sous-arbre gauche d'un enfant droit)

Arbres AVL

Suppression : parfois $\Theta(\log n)$ rotations sont nécessaires

Après avoir trouvé le nœud affecté par l'opération (insertion ou suppression) en descendant, il faut monter de nouveau pour vérifier les conditions d'équilibre.

Il faut maintenir le hauteur de chaque sous-arbre

(En fait, on peut implanter en utilisant seulement trois valeurs $-1, 0, +1$ pour stocker la différence entres les hauteurs des sous-arbres — 2 bits par nœud)

C'est $O(\log n)$ mais prend trop de travail (beaucoup de rotations dans le pire cas de suppression) et une valeur à maintenir à chaque nœud ...

Arbres rouge-noir

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

(rang pour les arbres RN comme la hauteur pour les arbres AVL)

Pour la discussion des arbres rouge et noir, on va considérer les pointeurs null pour des enfants manquants comme des pointeurs vers des feuilles

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

Arbres RN (cont)

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 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.

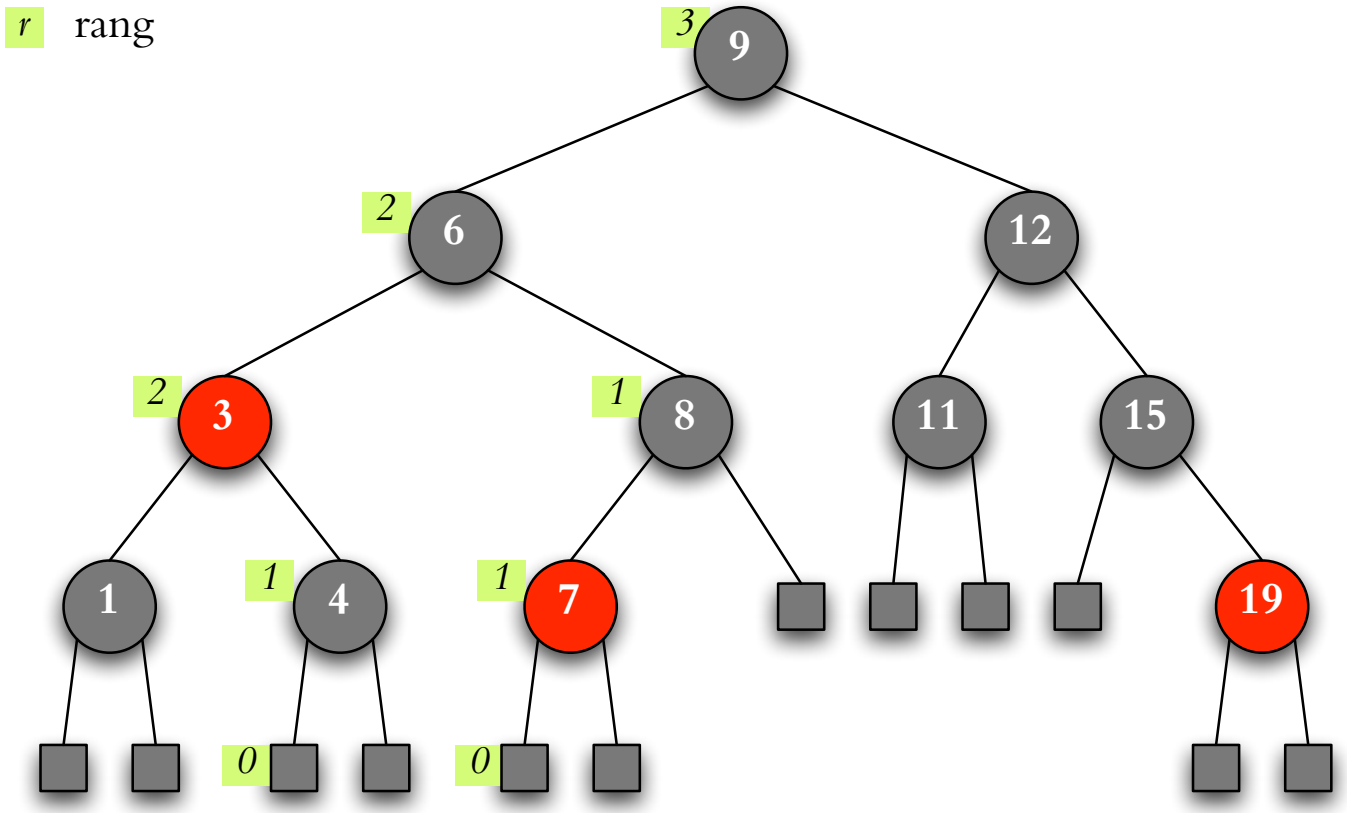
- si $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x)$, alors x est colorié par rouge
- si x est la racine ou $\mathsf{rang}(\mathsf{parent}(x)) = \mathsf{rang}(x) + 1$, alors x est colorié par noir

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. □

⇒ rang est parfois appelé «hauteur noire»

Arbres RN (cont)



Arbres RN (cont)

Thm. L'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 $\geq 2^{\mathsf{rang}(x)} - 1$.

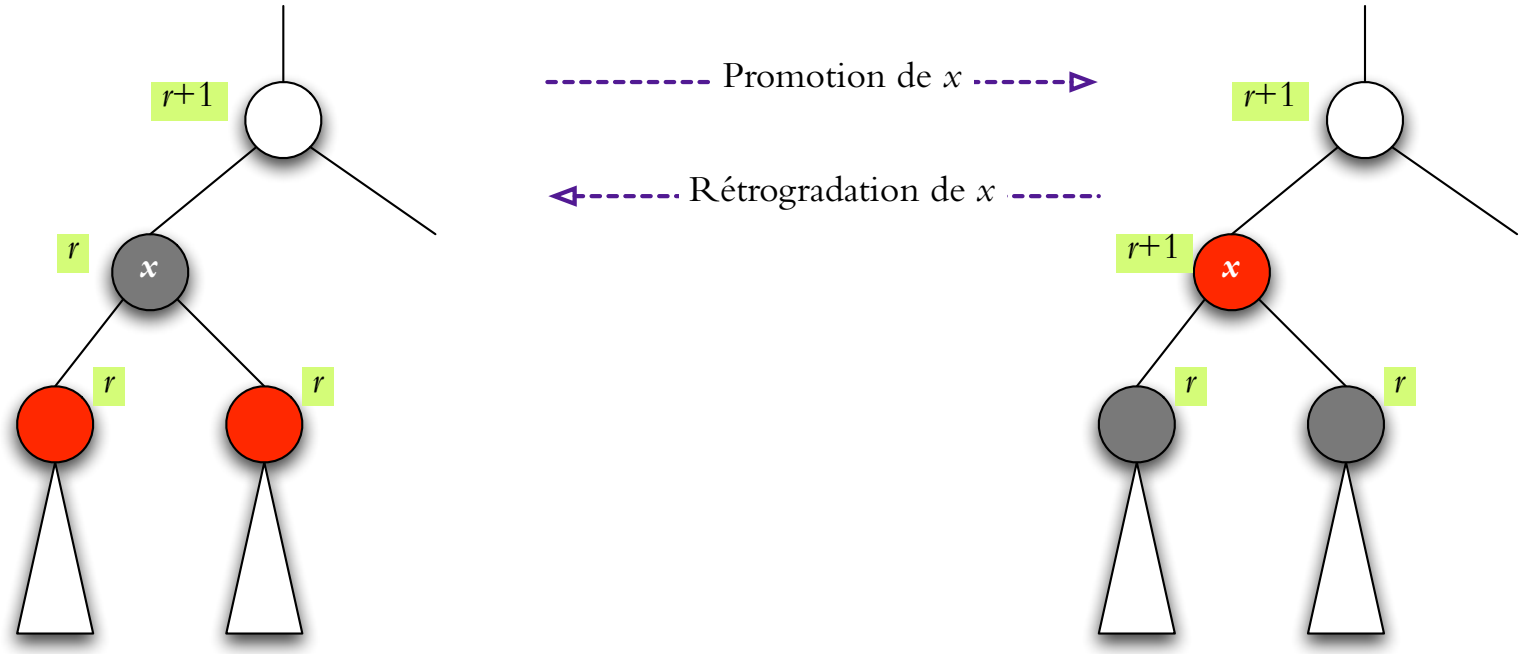
Preuve. Par induction. Le théorème est vrai pour une feuille x quand $\mathsf{rang}(x) = 0$. Supposons que le théorème est vrai pour tout x avec une hauteur $h(x) < k$. Considérons un nœud x avec $h(x) = k$ et ses deux enfants u, v avec $h(u), h(v) < k$. Par l'hypothèse d'induction, le nombre des descendants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



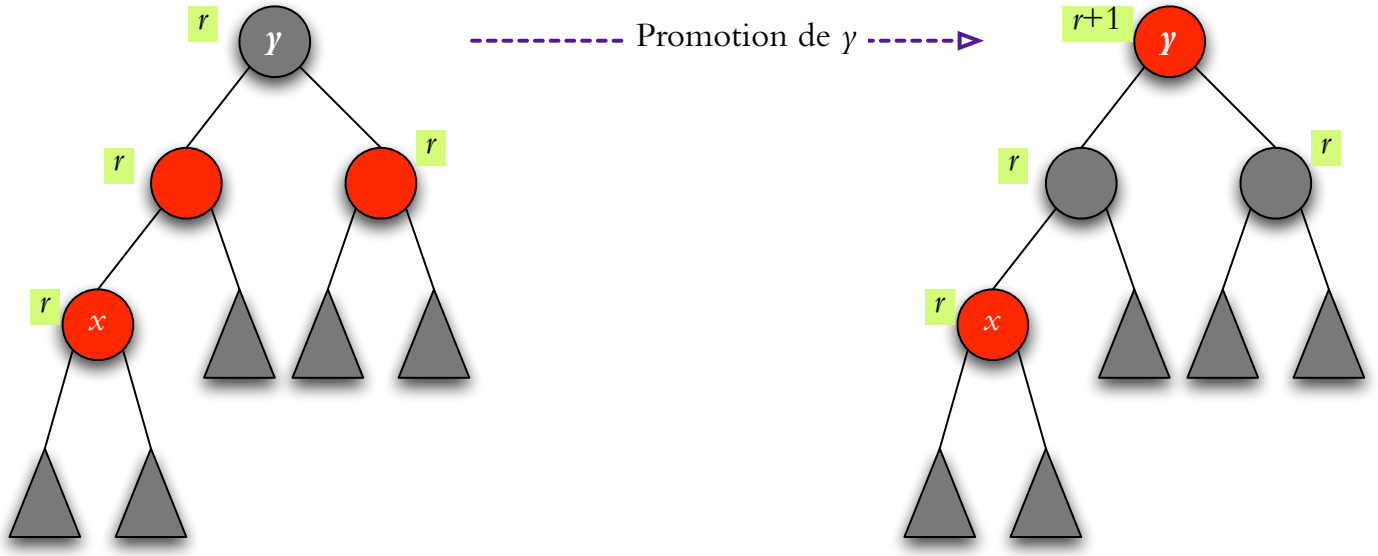
→ promotion/rétrogradation change la couleur d'un nœud et ses enfants
on peut promouvoir x ssi il est noir avec deux enfants rouges

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.

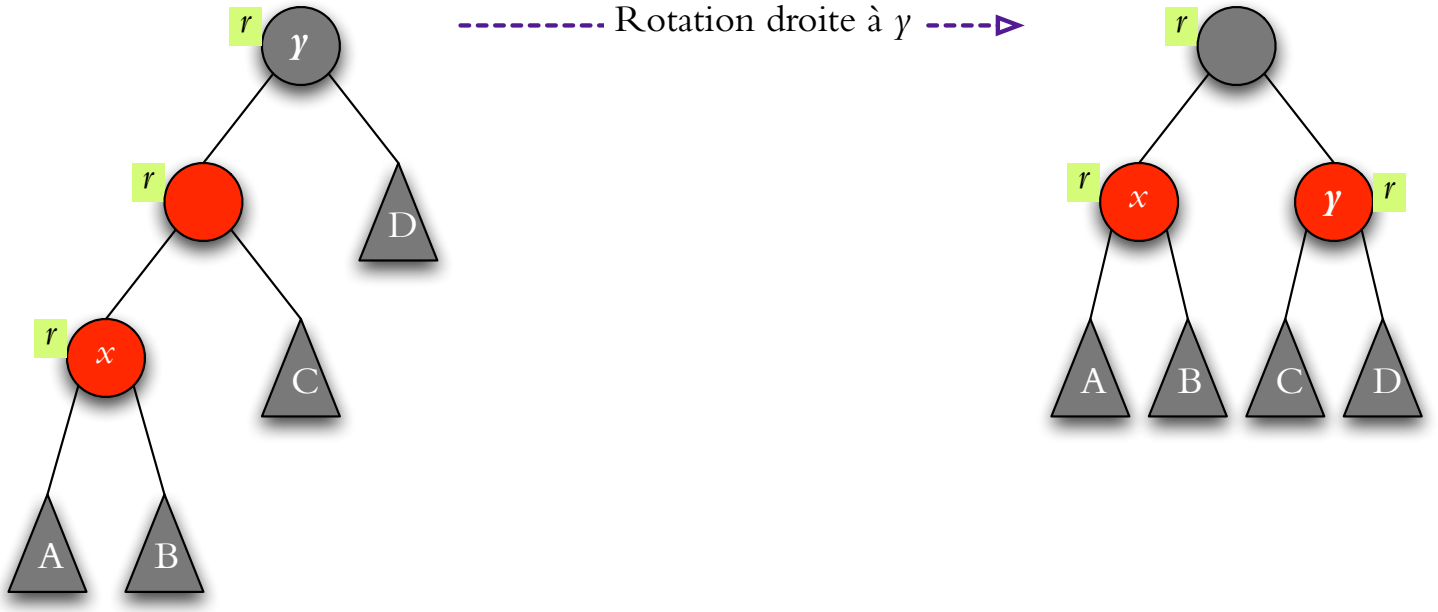
Deux cas :
Cas 1 quand x et $\mathsf{parent}(x)$ sont au même côté (enfants gauches ou enfants droits) — une rotation suffit

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)

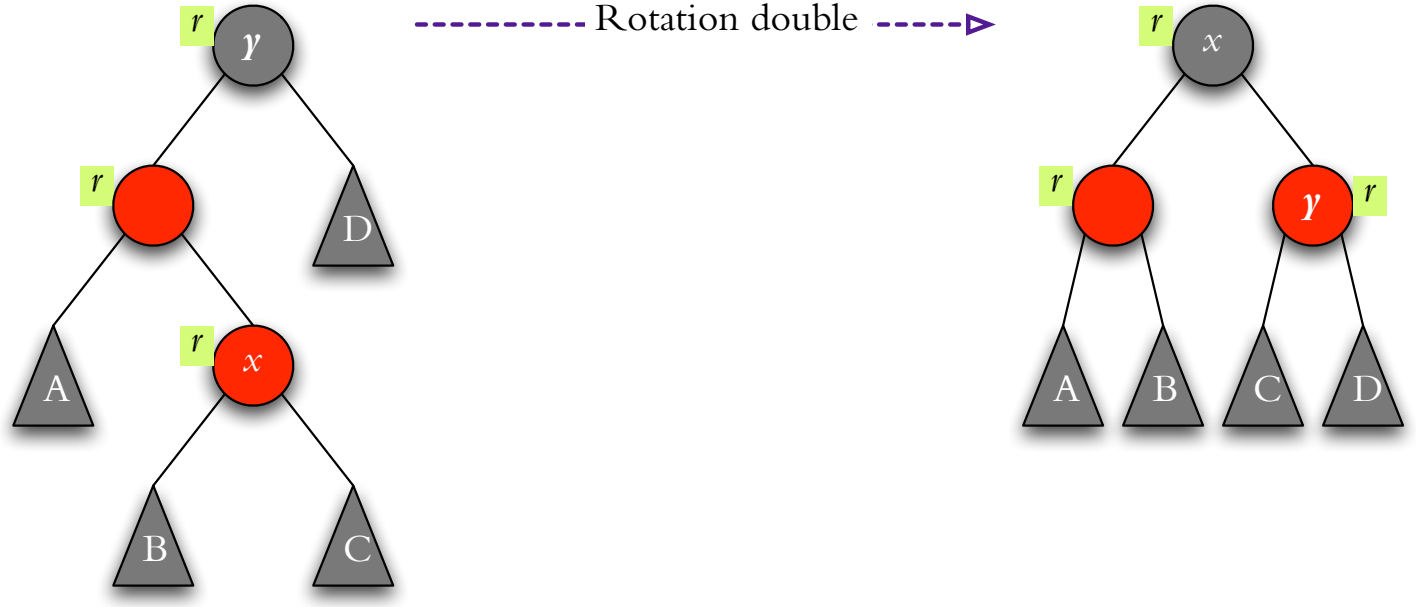
Cas 1a : x est rouge, son parent est rouge, son oncle est noir, et x et $\mathsf{parent}(x)$ sont des enfant gauches



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 retrogradations en ascendant vers la racine + $O(1)$ rotations (trois au plus) à la fin

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 $h \in O(\log n)$ dons $O(\log n)$

Suppression : $O(\log n)$

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

Arbres déployés

Variables pour maintenir l'équilibre de l'arbre

Arbre AVL : au moins deux bits (pour stocker $-1, 0, +1$)
Arbre RN : au moins un bit (couleur)

Arbre déployé (*splay tree*) : aucune variable

mais $O(\log n)$ seulement comme coût «moyen»

Arbres déployés (cont)

Idée principale : rotations sans tests spécifiques pour la balance

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

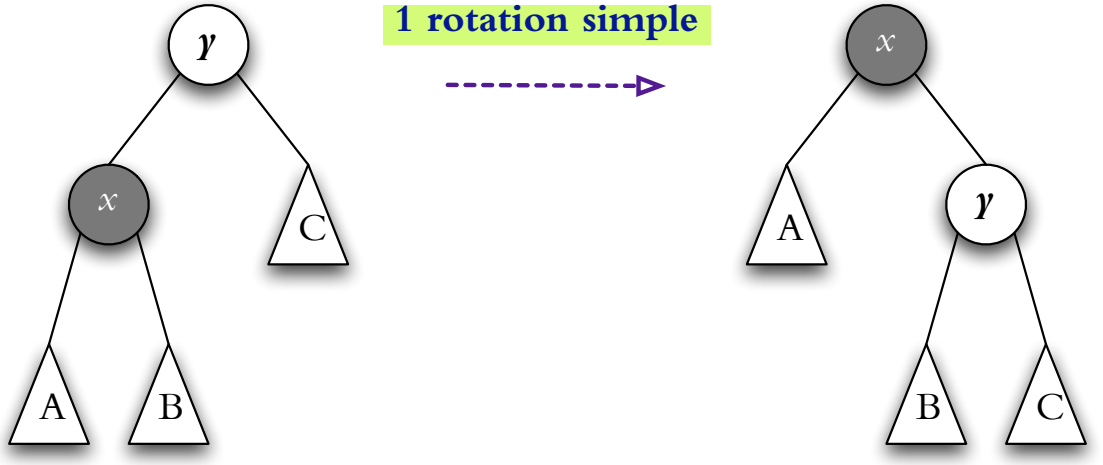
Déploiement (*splaying*) du nœud x : étapes successives jusqu'à ce que $\mathsf{parent}(x)$ devienne null
(et donc x devient la racine de l'arbre)

Zig et zag

Trois cas majeurs pour une étape de déploiement :

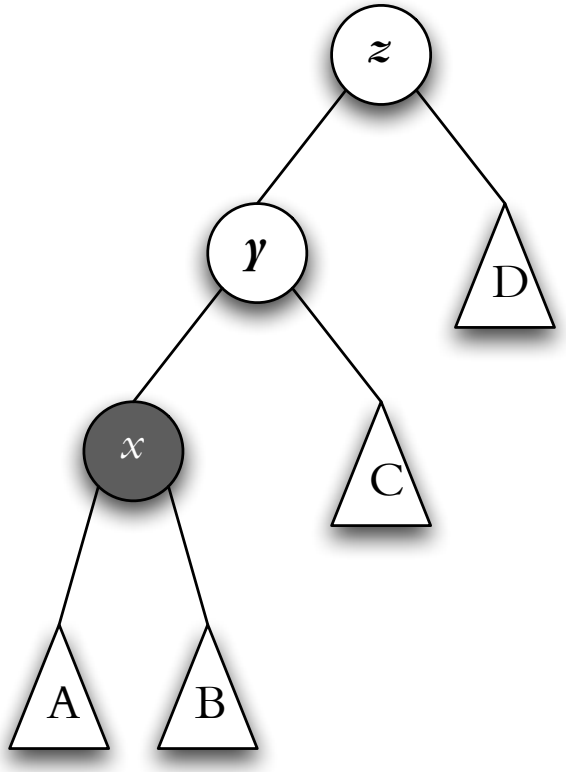
1. x sans grand-parent (zig ou zag)
2. x et $\mathsf{parent}(x)$ au même coté (gauche-gauche ou droit-droit : zig-zig ou zag-zag)
3. x et $\mathsf{parent}(x)$ à des côtés différents (gauche-droit ou droit-gauche : zig-zag ou zag-zig)

Cas 1: *zig*

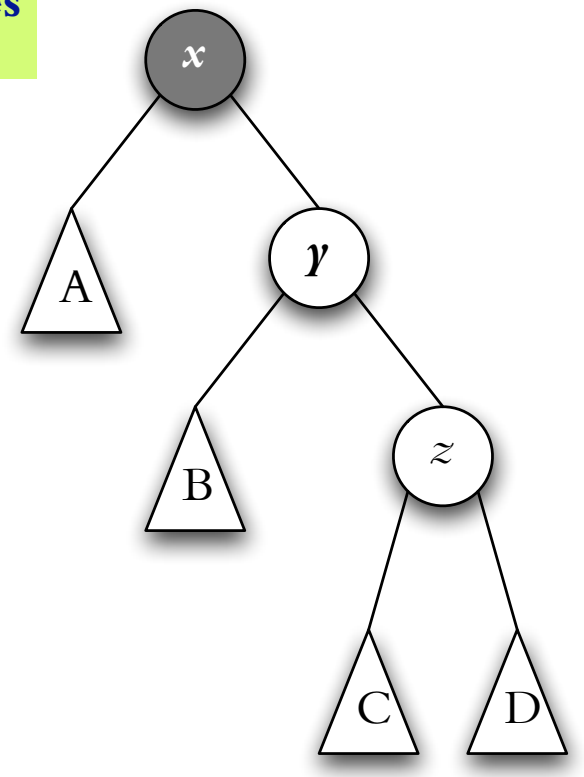


Zig et zag (cont)

Cas 2: *zig-zig*

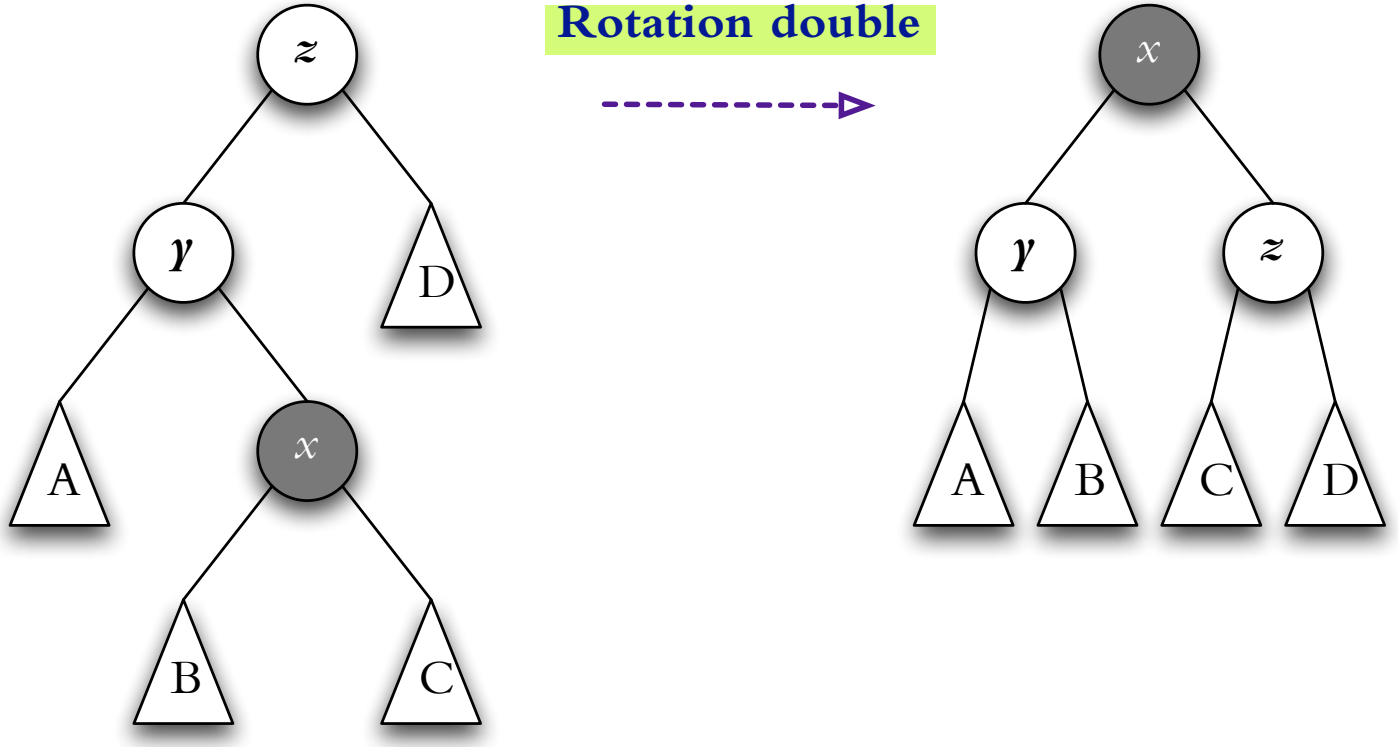


2 rotations simples
(à z et à y)



Zig et zag (cont)

Cas 3: *zig-zag*



Déploiement

Choix de x pour déploiement :

- insert : x est le nouveau nœud
- find : x est le nœud où on arrive à la find de la recherche
- delete : x est le parent du nœud 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 (find, 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.

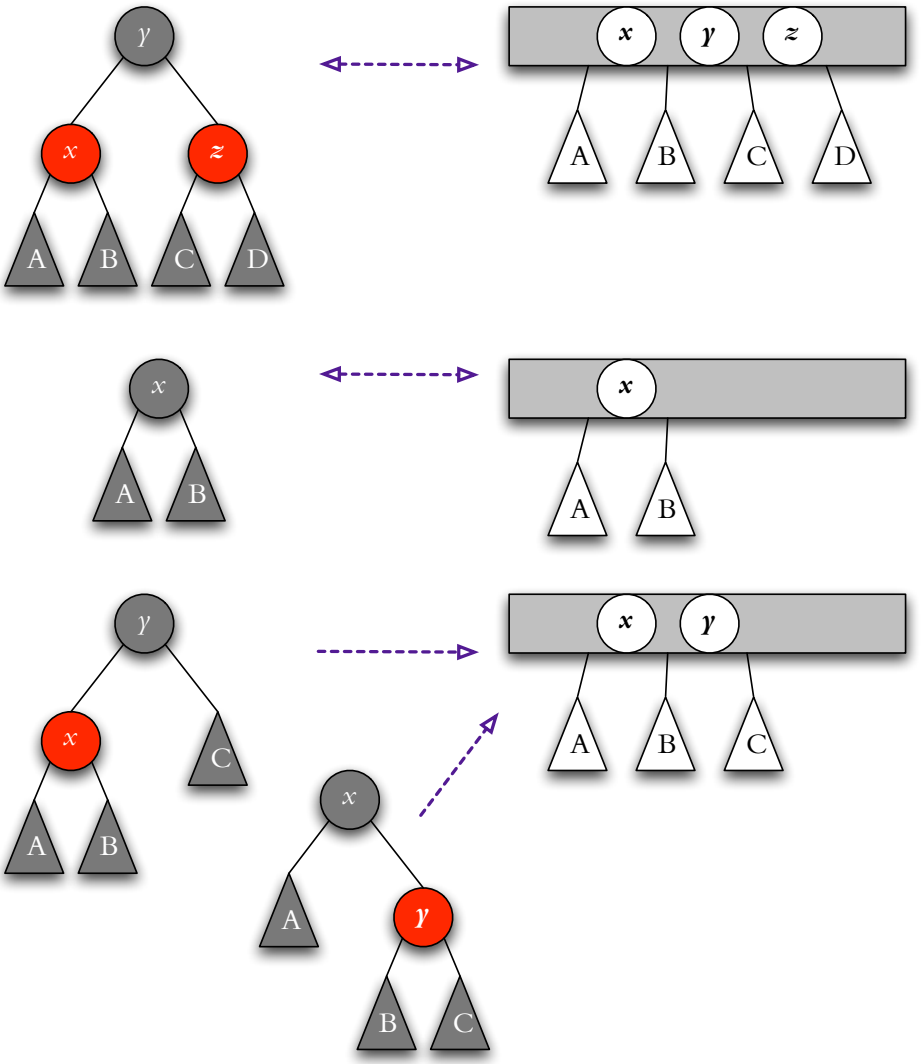
$\longrightarrow$ il peut arriver que l'exécution est très rapide au début et tout d'un coup une opération prend très long...

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 à la même profondeur

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

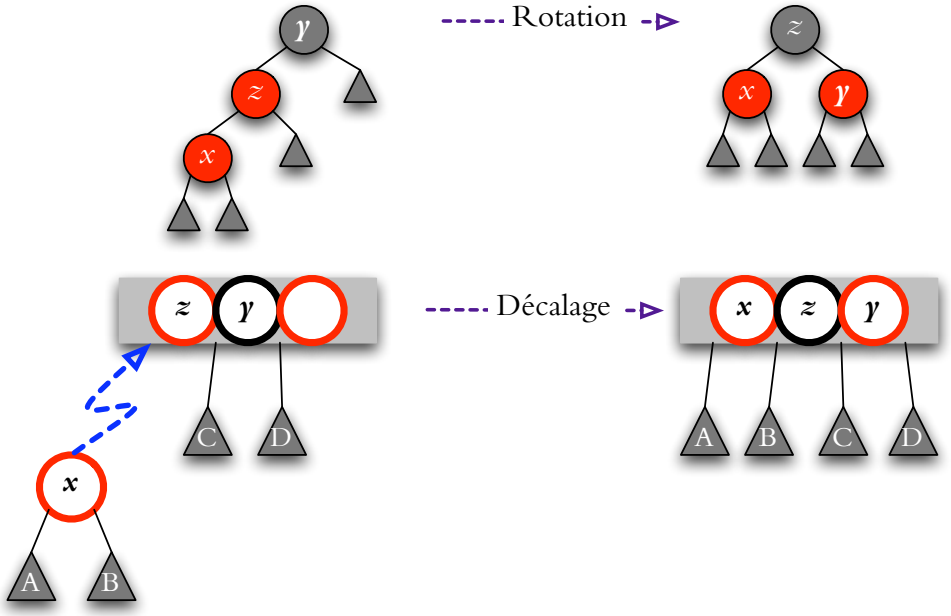
Transformation entre arbres RN et 2-3-4



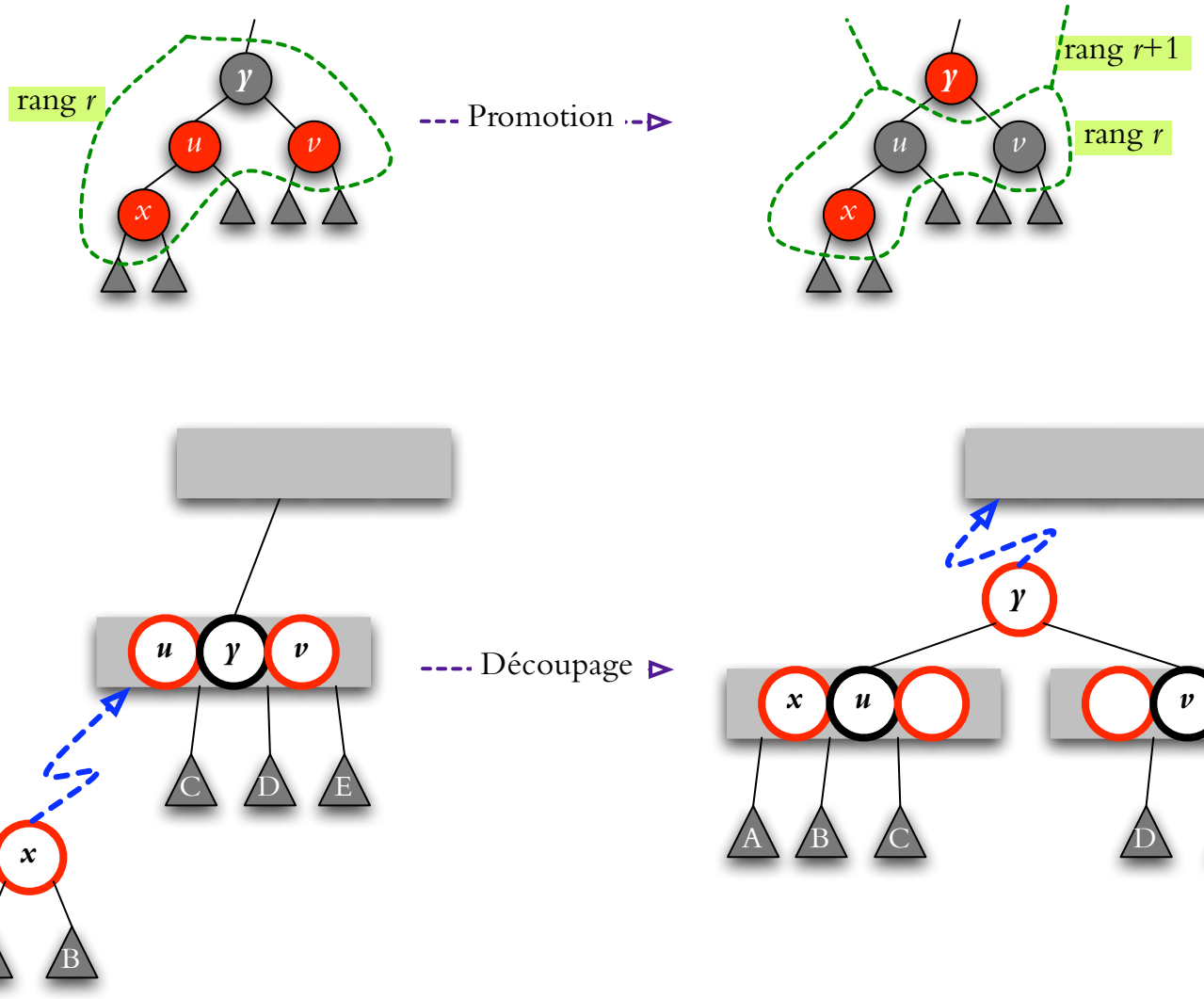
Arbre RN $\leftrightarrow$ arbre 2-3-4

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

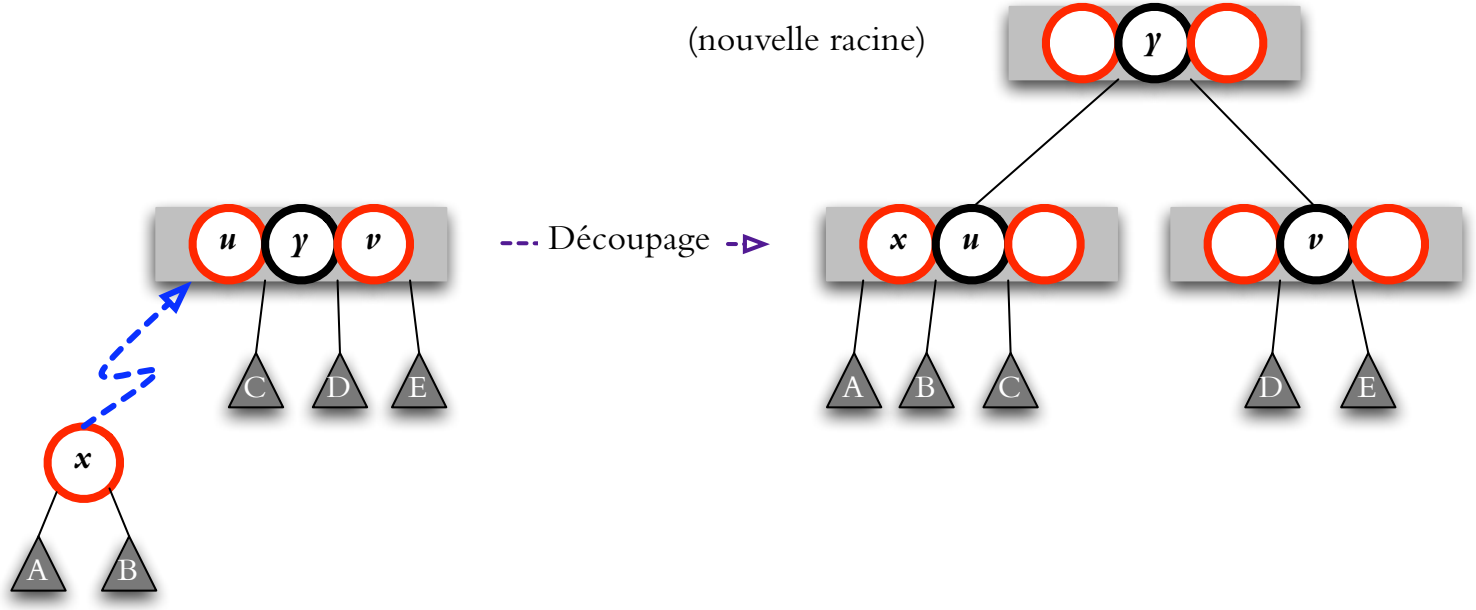


Arbre RN $\leftrightarrow$ arbre 2-3-4 (promotions)



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)

Recherche dans mémoire externe

Supposons qu'on veut stocker un grand nombre de données : **enregistrements** avec **clés**

On veut minimiser l'accès au disque dur : 10^5 fois plus de temps que l'accès à la mémoire principale

Stocker un arbre rouge et noir ??

Arbre 2-3-4 est plus efficace : on modifie «quelques» nœuds seulement

Comment améliorer ?
on généralise à M sous-arbres au lieu de 4.

Une autre bonne idée : on va mettre les données aux feuilles, et ne stocker que des clés à des nœuds internes («B^+-arbre»)

B-arbre

Données stockées aux feuilles : L enregistrements à une feuille

Clés stockés aux nœuds internes : $(M - 1)$ clés à un nœud interne, M enfants

Clé i : valeur minimale dans le sous-arbre $(i + 1)$

Racine : $2..M$ enfants
Nœuds internes : $\lceil M/2 \rceil..M$ enfants (taille : $M \cdot |\text{clé}| + (M - 1) \cdot |\text{pointeur}|$)
Feuilles : $\lceil L/2 \rceil..L$ enregistrements (taille : $L \cdot |\text{enregistrement}|$)

Feuilles ont la même profondeur

Choix de M et L : on utilise un bloc (taille typique : 4k, 8k, . . .) par nœud

B-arbre

Thm. La hauteur h de B arbre est bornée par

$$h \leq 1 + \log_{\lceil M/2 \rceil} \frac{N}{2} \leq 1 + \frac{\lg \frac{n}{L}}{\lg M - 1}$$

où N est le nombre de feuilles et n est le nombre d'enregistrements.

Exemple (du livre) : blocs de 8k, clés de 32 octets, enregistrements de 256 octets, $M = 228$, $L = 32$
$h = 4$ suffit jusqu'à $N = 2.9 \cdot 10^6$ ou $n = 47 \cdot 10^6$

$\Rightarrow$ nombre d'accès au disque est determiné par h : très peu (en plus, on peut garder la racine et peut-être même le premier niveau en mémoire principale)

B-arbre (cont)

Insertion d'un enregistrement : s'il y a de la place dans la feuille, aucun problème

s'il n'y a pas de place : **débordement** de la feuille

solution : découpage de la feuille $\rightarrow$ éléments distribués en deux feuilles de tailles $\lfloor \frac{L}{2} \rfloor + 1$ et $\lceil \frac{L}{2} \rceil$.

peut causer un débordement au parent : découpage si nécessaire en ascendant vers la racine

$\Rightarrow$ la hauteur croît en découpant la racine

B-arbre (cont)

Suppression d'un élément : si la feuille est toujours assez complète, aucun problème

et si le nombre d'éléments tombe en-dessous de $\lceil L/2 \rceil$?

1. prendre des éléments des sœurs immédiates

2. si elles sont au minimum, alors fusionner les feuilles $\longrightarrow$ le parent perd un unfant

3. continuer avec le parent de la même manière si nombre d'enfants $< \lceil M/2 \rceil$

$\Rightarrow$ la hauteur décroît en enlevant la racine (quand elle a un enfant seulement)