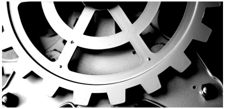


Algorithmique et structures de données
Les arbres

Gaël Mahé

slides : Elise Bonzon et Gaël Mahé

Université Paris Descartes

Licence 2

Les arbres

Les arbres

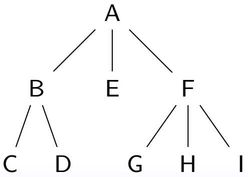
Arbres

- Un arbre est un ensemble de **nœuds** organisés de façon **hiérarchique** à partir d'un nœud distingué : la **racine**
- Structure fondamentale en informatique :
    - organisation des fichiers,
    - compilation
    - expressions arithmétiques et logiques...
- définitions, structures et algorithmes qui traitent les arbres
  ... sont fondés sur la **récursivité**

Arbres généraux

Un arbre général est composé de :

- Une racine
- De chaque nœud part un nombre quelconque de branches

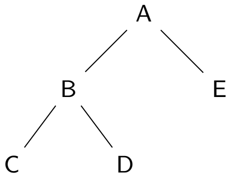


- Exemple : arbre généalogique descendant

Arbres binaires

Un arbre binaire est composé de :

- Une (et une seule) racine
- Chaque nœud peut avoir 0, 1 ou 2 branches



- Exemple :
  - Arbre généalogique ascendant
  - Tournoi

Définition formelle des arbres binaires

- Un arbre binaire est soit **vide** $\varnothing$, soit défini par :
  - **Racine** r,
  - **Sous-arbre gauche** G, et
  - **Sous-arbre droit** D

  où G et D sont eux-mêmes des arbres binaires
- L'ensemble AB des **arbres binaires** sur l'alphabet S est donc défini par induction :

  (B) $\emptyset \in AB$ (c'est l'arbre binaire vide)

  (I) $G, D \in AB \Rightarrow \forall r \in S, \langle r, G, D \rangle \in AB$ (c'est l'arbre de racine r; de fils gauche G et de fils droit D)
- Ou, dans une écriture plus compacte :

$$AB = \varnothing + \langle r, AB, AB \rangle$$

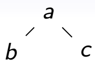
- NB : un arbre n'est pas symétrique : $\langle r, G, D \rangle \neq \langle r, D, G \rangle$

Construction des arbres binaires

- $\langle a, \emptyset, \emptyset\rangle$

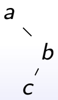
$$a$$

- $\langle a, \langle b, \emptyset, \emptyset\rangle, \langle c, \emptyset, \emptyset\rangle\rangle$

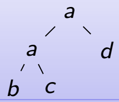


  Pour simplifier, on peut convenir de représenter l'arbre $\langle a, \emptyset, \emptyset\rangle$ par a. L'arbre précédent s'écrit alors $\langle a, b, c\rangle$

- $\langle a, \emptyset, \langle b, c, \emptyset\rangle\rangle$

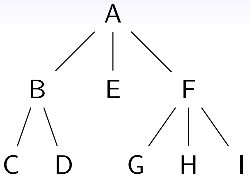


- $\langle a, \langle a, b, c\rangle, d\rangle$



Vocabulaire

- On utilise **père**, **fils**, **frère**



- A est le père de B
- F est le frère de B
- C est le fils de B

Vocabulaire

- Un nœud n'a qu'un seul père
- Un nœud sans fils est une **feuille** ou un **sommet pendant**
- Une **branche** est le chemin qui joint un nœud à la racine

Les arbres

Quelques fonctions définies sur les arbres binaires

- AB représente l'**ensemble des arbres binaires**
- S représente l'**ensemble des nœuds** auxquels on peut associer des **éléments** de E (**arbre étiqueté**)
- Constante $\mathrm{arbrevide} \in AB$
- $\mathrm{racine}\colon AB \setminus \{\mathrm{arbrevide}\} \to S$
- $\mathrm{G}, \mathrm{D}\colon AB \setminus \{\mathrm{arbrevide}\} \to AB$
- $\mathrm{contenu}\colon S \to E$

Définitions

- **Arbre filiforme** ou **dégénéré** : arbre binaire formé de nœuds qui n'ont qu'un seul fils
- **Arbre complet** : arbre binaire dont chaque niveau est complètement rempli
- **Arbre parfait** : arbre binaire dont tous les niveaux sont complètement remplis sauf peut-être le dernier, mais alors les feuilles du dernier niveau sont regroupées à gauche
- **Arbre localement complet** : arbre binaire non vide tel que tous les nœuds qui ne sont pas des feuilles ont 2 fils
  Définition récursive de l'ensemble L des arbres localement complets :
  $$L = \langle r, \emptyset, \emptyset \rangle + \langle r, L, L \rangle$$
- **Peigne gauche** : arbre binaire localement complet tel que tout fils droit est une feuille (et vice-versa)

Définitions

- **Arbre filiforme** ou **dégénéré** : arbre binaire formé de nœuds qui n'ont qu'un seul fils
- **Arbre complet** : arbre binaire dont chaque niveau est complètement rempli
- **Arbre parfait** : arbre binaire dont tous les niveaux sont complètement remplis sauf peut-être le dernier, mais alors les feuilles du dernier niveau sont regroupées à gauche
- **Arbre localement complet** : arbre binaire non vide tel que tous les nœuds qui ne sont pas des feuilles ont 2 fils
  Définition récursive de l'ensemble L des arbres localement complets :
  $$L = \langle r, \emptyset, \emptyset \rangle + \langle r, L, L \rangle$$
- **Peigne gauche** : arbre binaire localement complet tel que tout fils droit est une feuille (et vice-versa)

Définitions

- **Arbre filiforme** ou **dégénéré** : arbre binaire formé de nœuds qui n'ont qu'un seul fils
- **Arbre complet** : arbre binaire dont chaque niveau est complètement rempli
- **Arbre parfait** : arbre binaire dont tous les niveaux sont complètement remplis sauf peut-être le dernier, mais alors les feuilles du dernier niveau sont regroupées à gauche
- **Arbre localement complet** : arbre binaire non vide tel que tous les nœuds qui ne sont pas des feuilles ont 2 fils
  Définition récursive de l'ensemble L des arbres localement complets :

$$L = \langle r, \emptyset, \emptyset \rangle + \langle r, L, L \rangle$$

- **Peigne gauche** : arbre binaire localement complet tel que tout fils droit est une feuille (et vice-versa)

Définitions

- **Arbre filiforme** ou **dégénéré** : arbre binaire formé de nœuds qui n'ont qu'un seul fils
- **Arbre complet** : arbre binaire dont chaque niveau est complètement rempli
- **Arbre parfait** : arbre binaire dont tous les niveaux sont complètement remplis sauf peut-être le dernier, mais alors les feuilles du dernier niveau sont regroupées à gauche
- **Arbre localement complet** : arbre binaire non vide tel que tous les nœuds qui ne sont pas des feuilles ont 2 fils
  Définition récursive de l'ensemble L des arbres localement complets :

$$L = \langle r, \emptyset, \emptyset \rangle + \langle r, L, L \rangle$$

- **Peigne gauche** : arbre binaire localement complet tel que tout fils droit est une feuille (et vice-versa)

Définitions

- **Arbre filiforme** ou **dégénéré** : arbre binaire formé de nœuds qui n'ont qu'un seul fils
- **Arbre complet** : arbre binaire dont chaque niveau est complètement rempli
- **Arbre parfait** : arbre binaire dont tous les niveaux sont complètement remplis sauf peut-être le dernier, mais alors les feuilles du dernier niveau sont regroupées à gauche
- **Arbre localement complet** : arbre binaire non vide tel que tous les nœuds qui ne sont pas des feuilles ont 2 fils
  Définition récursive de l'ensemble L des arbres localement complets :

$$L = \langle r, \emptyset, \emptyset \rangle + \langle r, L, L \rangle$$

- **Peigne gauche** : arbre binaire localement complet tel que tout fils droit est une feuille (et vice-versa)

Définitions

Mesures permettant d'évaluer la complexité des algorithmes sur les arbres :

- Nombre de sommets : taille: $AB \to \mathbb{N}$
  - $\text{taille}(\varnothing) = 0$
  - $\text{taille}(\langle r, G, D \rangle) = 1 + \text{taille}(G) + \text{taille}(D)$
- Nombre d'ascendants d'un nœud jusqu'à la racine : hauteur: $S \to \mathbb{N}$
  - $\text{hauteur}(x) = 0$ si x est la racine
  - $\text{hauteur}(x) = 1 + \text{hauteur}(y)$ si y est le père de x

  On dit aussi la **profondeur** ou le **niveau** d'un nœud.
- Plus grande hauteur prise sur l'ensemble des sommets : profondeur: $AB \to \mathbb{N}$
  - $\text{profondeur}(A) = \max_{x \in S}\{\text{hauteur}(x)\}$
- Nombre de fils d'un nœud : degre: $S \to \mathbb{N}$
  - Dans un arbre binaire, $\text{degre}(x)$ peut prendre les valeurs 0, 1 ou 2

Définitions

Mesures permettant d'évaluer la complexité des algorithmes sur les arbres :

- Nombre de sommets : taille : $AB \to \mathbb{N}$
  - $\mathrm{taille}(\varnothing) = 0$
  - $\mathrm{taille}(\langle r, G, D\rangle) = 1 + \mathrm{taille}(G) + \mathrm{taille}(D)$
- Nombre d'ascendants d'un nœud jusqu'à la racine : hauteur : $S \to \mathbb{N}$
  - $\mathrm{hauteur}(x) = 0$ si x est la racine
  - $\mathrm{hauteur}(x) = 1 + \mathrm{hauteur}(y)$ si y est le père de x

  On dit aussi la **profondeur** ou le **niveau** d'un nœud.
- Plus grande hauteur prise sur l'ensemble des sommets : profondeur : $AB \to \mathbb{N}$
  - $\mathrm{profondeur}(A) = \max_{x \in S}\{\mathrm{hauteur}(x)\}$
- Nombre de fils d'un nœud : degre : $S \to \mathbb{N}$
  - Dans un arbre binaire, $\mathrm{degre}(x)$ peut prendre les valeurs 0, 1 ou 2

Définitions

Mesures permettant d'évaluer la complexité des algorithmes sur les arbres :

- Nombre de sommets : taille : $AB \to \mathbb{N}$
  - $\mathrm{taille}(\varnothing) = 0$
  - $\mathrm{taille}(\langle r, G, D \rangle) = 1 + \mathrm{taille}(G) + \mathrm{taille}(D)$
- Nombre d'ascendants d'un nœud jusqu'à la racine : hauteur : $S \to \mathbb{N}$
  - $\mathrm{hauteur}(x) = 0$ si x est la racine
  - $\mathrm{hauteur}(x) = 1 + \mathrm{hauteur}(y)$ si y est le père de x

  On dit aussi la **profondeur** ou le **niveau** d'un nœud.
- Plus grande hauteur prise sur l'ensemble des sommets : profondeur : $AB \to \mathbb{N}$
  - $\mathrm{profondeur}(A) = \max_{x \in S}\{\mathrm{hauteur}(x)\}$
- Nombre de fils d'un nœud : degre : $S \to \mathbb{N}$
  - Dans un arbre binaire, $\mathrm{degre}(x)$ peut prendre les valeurs 0, 1 ou 2

Définitions

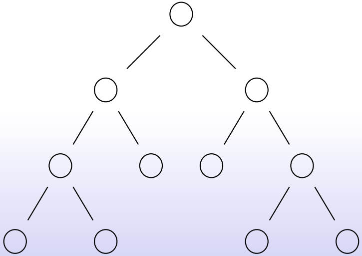
Mesures permettant d'évaluer la complexité des algorithmes sur les arbres :

- Nombre de sommets : taille: $AB \to \mathbb{N}$
  - $\mathrm{taille}(\varnothing) = 0$
  - $\mathrm{taille}(\langle r, G, D \rangle) = 1 + \mathrm{taille}(G) + \mathrm{taille}(D)$
- Nombre d'ascendants d'un nœud jusqu'à la racine : hauteur: $S \to \mathbb{N}$
  - $\mathrm{hauteur}(x) = 0$ si x est la racine
  - $\mathrm{hauteur}(x) = 1 + \mathrm{hauteur}(y)$ si y est le père de x

  On dit aussi la **profondeur** ou le **niveau** d'un nœud.
- Plus grande hauteur prise sur l'ensemble des sommets : profondeur: $AB \to \mathbb{N}$
  - $\mathrm{profondeur}(A) = \max_{x \in S}\{\mathrm{hauteur}(x)\}$
- Nombre de fils d'un nœud : degre: $S \to \mathbb{N}$
  - Dans un arbre binaire, $\mathrm{degre}(x)$ peut prendre les valeurs 0, 1 ou 2

Numérotation des nœuds d'un arbre binaire

- Les nœuds sont numérotés par les mots formés sur l'alphabet $\{0,1\}$ selon une numérotation hiérarchique :
- La racine est numérotée par le mot vide ϵ
- Si un nœud est numéroté par $\mu \in \{0,1\}^*$,
  - son fils gauche est numéroté $\mu 0$, et
  - son fils droit est numéroté $\mu 1$



Un représentation de l'arbre est $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 110, 111\}$

Propriétés des arbres binaires

- Pour une hauteur h,
  - Les arbres avec le maximum de nœuds sont les arbres complets : $n = 2^{h+1} - 1$ sommets
  - Les arbres avec le minimum de nœuds sont les arbres dégénérés : $n = h + 1$ sommets
- Conséquence : pour un arbre binaire (non vide) de taille n, de profondeur h

$$\lfloor \log_2(n) \rfloor \leq h \leq n - 1$$

Propriétés des arbres binaires

- Un arbre binaire de taille n possédant f feuilles satisfait

$$f \leq \frac{n+1}{2} \qquad (P)$$

- Démonstration par le principe d'induction

1. Relation d'ordre *"est un sous-arbre de"* bien fondée sur AB
2. (B) La propriété est vraie pour l'arbre vide ($n = 0$)
3. (I') Soit $B = \langle r, B1, B2 \rangle$ un arbre binaire de n sommets et f feuilles

4. On a montré que $\forall\, B \in AB, (\forall\, A_{\in AB} < B,\ P(A)) \Rightarrow P(B)$
Donc, d'après le principe d'induction, $\forall\, B \in AB,\ P(B)$.

Propriétés des arbres binaires

- Pour un arbre binaire de taille $n > 0$, de profondeur h,

$$f \leq \frac{n+1}{2} \quad \text{et} \quad \lfloor \log_2(n) \rfloor \leq h \leq n-1$$

- Un arbre binaire non vide satisfait donc :

$$\lceil \log_2(f) \rceil \leq \lfloor \log_2(n) \rfloor \leq h \leq n-1$$

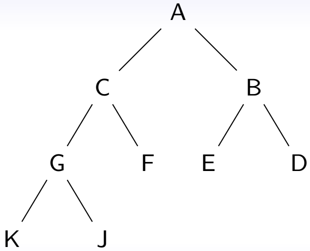
- Démonstration de $\lceil \log_2(f) \rceil \leq \lfloor \log_2(n) \rfloor$:

- Démonstration de $\lceil \log_2(n+1) \rceil = \lfloor \log_2(n) \rfloor + 1$:

Les arbres

Parcours d'un arbre binaire

- Certains algorithmes nécessitent de parcourir systématiquement tous les nœuds d'un arbre
- En **profondeur à main gauche** (parcours du gant)



- Chaque nœud est rencontré 3 fois :

Parcours d'un arbre binaire

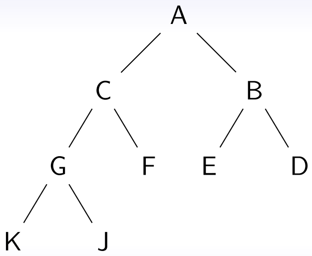
Lors de ce parcours, nous supposons qu'un traitement spécifique est fait :

- T1 au premier passage (en descendant)
- T2 la seconde fois
- T3 au dernier passage (en remontant définitivement)

Algorithme 1 : Procedure Parcours(Ab)

début
  /* ENTRÉES: Un arbre Ab */
  /* SORTIE: Application des traitements T1, T2 et T3 */
  si $Non(\mathrm{est\_vide}(Ab))$ **alors**
    Traitement T1 sur $racine(Ab)$
    Parcours($G(Ab)$)
    Traitement T2 sur $racine(Ab)$
    Parcours($D(Ab)$)
    Traitement T3 sur $racine(Ab)$
fin

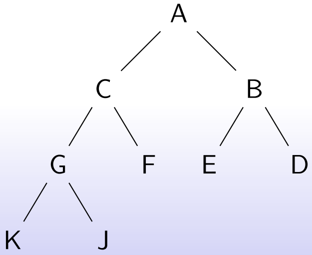
Parcours d'un arbre binaire : exemple



Ordres préfixe, symétrique, suffixe

Supposons que chaque noeud soit traité une seule fois :

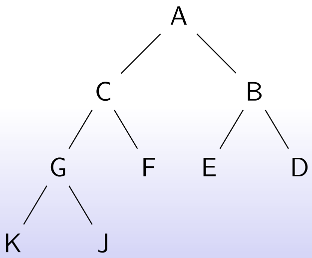
- lors de la première rencontre (T1) → **ordre préfixe**
  Exemple : A C G K J F B E D
- lors de la deuxième rencontre (T2) → **ordre symétrique** ou **infixe**
  Exemple : K G J C F A E B D
- lors de la troisième rencontre (T3) → **ordre suffixe**
  Exemple : K J G F C E D B A



Ordres préfixe, symétrique, suffixe

Supposons que chaque noeud soit traité une seule fois :

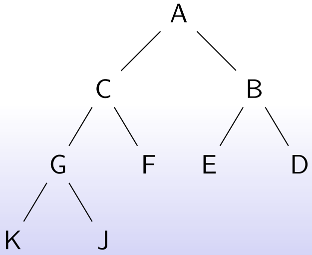
- lors de la première rencontre (T1) → **ordre préfixe**
  Exemple : A C G K J F B E D
- lors de la deuxième rencontre (T2) → **ordre symétrique** ou **infixe**
  Exemple : K G J C F A E B D
- lors de la troisième rencontre (T3) → **ordre suffixe**
  Exemple : K J G F C E D B A



Ordres préfixe, symétrique, suffixe

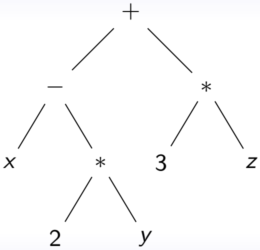
Supposons que chaque noeud soit traité une seule fois :

- lors de la première rencontre (T1) → **ordre préfixe**
  Exemple : A C G K J F B E D
- lors de la deuxième rencontre (T2) → **ordre symétrique** ou **infixe**
  Exemple : K G J C F A E B D
- lors de la troisième rencontre (T3) → **ordre suffixe**
  Exemple : K J G F C E D B A



Application : expressions arithmétiques

- Expression arithmétique ne comprenant que des opérateurs binaires
- peut être représentée par un arbre binaire localement complet
- Exemple : $(x - (2 * y)) + (3 * z)$



- Parcours en ordre préfixe $\rightarrow$ notation polonaise préfixée :
  Exemple : $+ - x * 2y * 3z$
- Parcours en ordre suffixe $\rightarrow$ notation polonaise postfixée (ou inverse) :
  Exemple : $x2y * -3z * +$
- Pourquoi pas de parcours en ordre symétrique ?

Parcours d'un arbre binaire : version itérative

- Empiler dans une pile P, avec la marque 1, les nœuds traités (T1) en descente gauche
- En remontant l'arbre gauche, effectuer T2 dans l'ordre inverse (d'où l'utilisation d'une pile)
- Rempiler chaque nœud traité T2 dans P avec la marque 2
- Le dépilement des nœuds munis de la marque 2 correspond à la remontée à droite
  On traite alors par T3 chaque nœud dépilé.

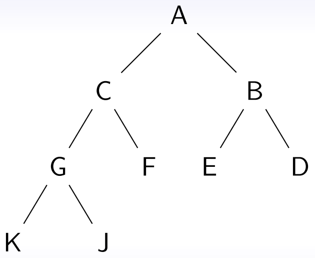
Parcours itératif

Algorithme 2 : Parcours(Ab)

début
- /* ENTRÉES: Un arbre Ab */
- $P \leftarrow pilevide$; $t \leftarrow 1$; $x \leftarrow racine(Ab)$
- **répéter**
  - **si** $t = 1$ **alors**
    - **tant que** *not(estvide*(x)*)* **faire**
      - $T1(x)$
      - $empile((x, 1), P)$
      - $x \leftarrow G(x)$
  - **si** *not(estvide*(P)*)* **alors**
    - $(x, t) \leftarrow sommet(P)$; $depile(P)$
    - **si** $t = 1$ **alors**
      - $T2(x)$
      - $empile((x, 2), P)$
      - $x \leftarrow D(x)$
    - **sinon** $T3(x)$
- **jusqu'à** $estvide(P)$

fin

Parcours d’un arbre binaire : exemple

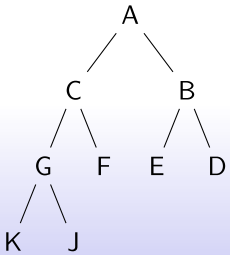


Les arbres

Tableau

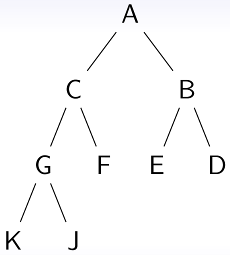
- Par un **tableau** de trois lignes
- Pour chaque indice i les trois lignes contiennent dans l'ordre :
  - l'étiquette du nœud i
  - l'indice où se trouve le fils gauche du nœud i, avec par convention 0 si ce dernier n'existe pas
  - l'indice où se trouve le fils droit du nœud i, avec par convention 0 si ce dernier n'existe pas.

Indices	1	2	3	4	5	6	7	8	9
Arbre	A	B	C	D	E	F	G	K	J
	3	5	7	0	0	0	8	0	0
	2	4	6	0	0	0	9	0	0



Tableau

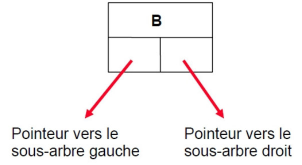
Indices	1	2	3	4	5	6	7	8	9
Arbre	A	B	C	D	E	F	G	K	J
	3	5	7	0	0	0	8	0	0
	2	4	6	0	0	0	9	0	0

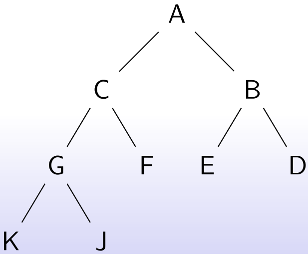


- Recherche de la racine?

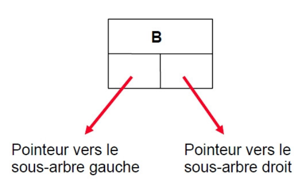
Liste chaînée

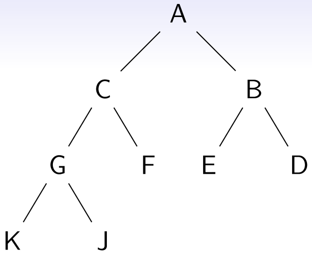
- Par une **liste chaînée**
- A chaque nœud, on associe une structure avec
  - l'étiquette du nœud
  - un pointeur vers son sous-arbre droit
  - un pointeur vers son sous-arbre gauche
- L'arbre est donné par l'adresse de sa racine





Liste chaînée





- Intérêt de cette représentation?
- Recherche du père d'un nœud?

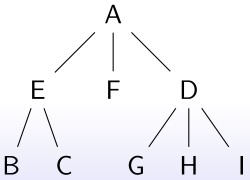
Les arbres

Expression complètement parenthésée

- On écrit une parenthèse ouvrante, puis l'étiquette du sommet
- On descend au premier fils, et on recommence :
  parenthèse ouvrante, étiquette du sommet
- Lorsqu'on ne peut plus descendre,
  on écrit une parenthèse fermante
  puis on remonte pour descendre au fils suivant

Expression complètement parenthésée

- On écrit une parenthèse ouvrante, puis l'étiquette du sommet
- On descend au premier fils, et on recommence :
  parenthèse ouvrante, étiquette du sommet
- Lorsqu'on ne peut plus descendre,
  on écrit une parenthèse fermante
  puis on remonte pour descendre au fils suivant



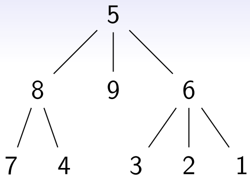
(A (E (B) (C)) (F) (D (G) (H) (I)))

- Inconvénient : peu pratique pour les manipulations

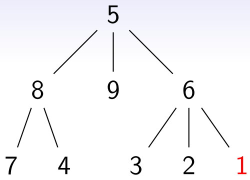
Codage de Prüfer d'un arbre général

- On suppose qu'il existe un ordre sur l'étiquetage des sommets
- On cherche une représentation minimale, linéaire, d'un arbre de taille n, par un mot de longueur $n - 1$ et l'ensemble S des sommets :
  - Chercher la feuille de plus petit numéro
  - Éliminer ce sommet et inscrire son père
  - Répéter le processus jusqu'à obtenir une arborescence réduite à la racine

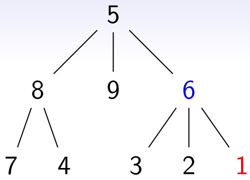
Codage de Prüfer d'un arbre général : Exemple



Codage de Prüfer d'un arbre général : Exemple

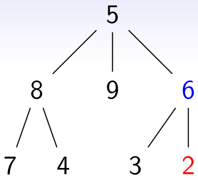


Codage de Prüfer d'un arbre général : Exemple



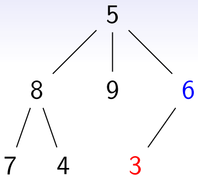
Code de Prüfer : 6

Codage de Prüfer d'un arbre général : Exemple



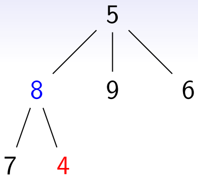
Code de Prüfer : 6 6

Codage de Prüfer d'un arbre général : Exemple



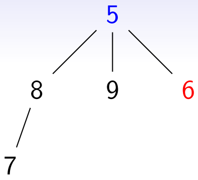
Code de Prüfer : 6 6 6

Codage de Prüfer d'un arbre général : Exemple



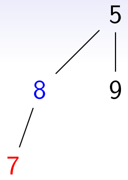
Code de Prüfer : 6 6 6 8

Codage de Prüfer d'un arbre général : Exemple



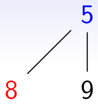
Code de Prüfer : 6 6 6 8 5

Codage de Prüfer d'un arbre général : Exemple



Code de Prüfer : 6 6 6 8 5 8

Codage de Prüfer d'un arbre général : Exemple



Code de Prüfer : 6 6 6 8 5 8 5

Codage de Prüfer d'un arbre général : Exemple

Code de Prüfer : 6 6 6 8 5 8 5 5

Codage de Prüfer d'un arbre général : Exemple

5

Code de Prüfer : 6 6 6 8 5 8 5 5

Décodage de Prüfer

- Soit S l'ensemble ordonné des sommets de l'arbre
  et P la suite formant le codage de Prüfer
- Chercher le plus petit élément s de S qui n'est pas dans P
- Relier par une arête le sommet s de S avec le premier élément p du mot P
- Supprimer s de S et p de P
- Continuer tant que S n'est pas vide

Décodage de Prüfer

- Soit S l'ensemble ordonné des sommets de l'arbre
  et P la suite formant le codage de Prüfer
- Chercher le plus petit élément s de S qui n'est pas dans P
- Relier par une arête le sommet s de S avec le premier élément p du mot P
- Supprimer s de S et p de P
- Continuer tant que S n'est pas vide

Exemple : $S = 1\ 2\ \ldots\ 9$, $P = 6\ 6\ 6\ 8\ 5\ 8\ 5\ 5$

Codage de Prüfer

- S'applique sur des arbres généraux dont les étiquettes sont ordonnées
- On peut retrouver l'arbre de manière unique à partir de son code de Prüfer
- Représentation linéaire particulièrement compacte

$\Rightarrow$ intéressant la représentation des arbres généraux en machine

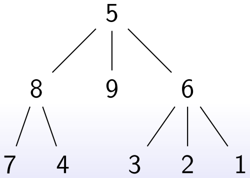
$\ominus$ mais difficilement manipulable pour faire des parcours

Tableau

Pour chaque indice *i* les lignes contiennent dans l'ordre :

- l'étiquette du nœud *i*
- l'indice où se trouve le 1er fils du nœud *i*,
- l'indice où se trouve le 2e fils du nœud *i*,
- etc

Indices	1	2	3	4	5	6	7	8	9
Arbre	5	8	9	6	7	4	3	2	1
	2	5	0	7	0	0	0	0	0
	3	6	0	8	0	0	9	0	0
	4	0	0	9	0	0	9	0	0



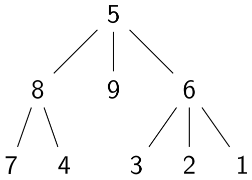
- Inconvénient : beaucoup de place consommée pour rien

Les arbres

Transformation d'un arbre général en arbre binaire

- Bijection "fils ainé gauche - frère immédiat droit"
- La racine S de l'arbre général A est la racine de l'arbre binaire AB que l'on veut créer
- Le fils ainé gauche S_1 de S dans A est le fils ainé gauche de S dans AB
- Répéter : le frère immédiat droit de S_i dans A est le fils droit de S_i dans AB
- On itére le processus jusqu'à avoir placé tous les nœuds de A dans AB

Transformation d'un arbre général en arbre binaire : exemple



Transformation d'un arbre général en arbre binaire

Intérêt de cette transformation

- Bijection, donc réversible
- Représentation peu gourmande en mémoire
- Non nécessaire de revenir à l'arbre général pour le parcourir :
  Pour un arbre général A transformée en arbre binaire B,
  - parcours préfixe de A = parcours préfixe de B
  - parcours suffixe de A = parcours symétrique de B

Les arbres

Arbres binaires de recherche : motivation

- **Recherche, insertion ou suppression** d'un élément dans une liste d'éléments **ordonnés** :

▶ Comment réduire la complexité ?

- **Recherche dichotomique**

  $\rightarrow$ $\Theta(log(n))$ comparaisons (au lieu de $\Theta(n)$ par méthode séquentielle)
  $\ominus$ insertion ou suppression peuvent nécessiter $\Theta(n)$ opérations

- **Utiliser une liste chaînée**

  $\rightarrow$ insertion et suppression en un nombre constant d'opérations
  $\ominus$ Recherche en $\Theta(n)$ opérations (car accès uniquement séquentiel)

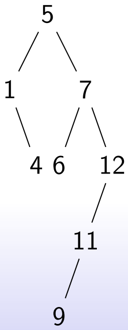
On veut que les 3 opérations : recherche, insertion et suppression aient la même complexité $\Theta(log(n))$
$\rightarrow$ Structures arborescentes

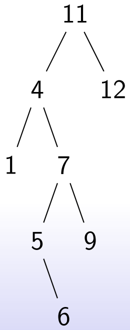
Arbres binaires de recherche : définition

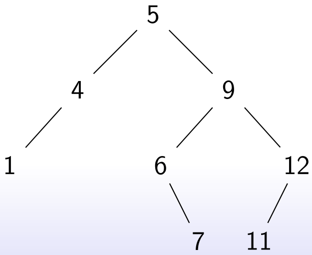
- Soir un ensemble E
  muni d'une **relation d'ordre total**
- Un arbre binaire de recherche A est un arbre binaire tel que :
  pour tout nœud n :
  - les éléments de $g(n)$ sont inférieurs ou égaux à n
  - les éléments de $d(n)$ sont supérieurs à n

Construction d'un ABR

- Insertions successives en respectant la propriété précédente
- Il existe plusieurs ABR représentant le même ensemble
- Exemple : $E = \{1, 4, 5, 6, 7, 9, 11, 12\}$

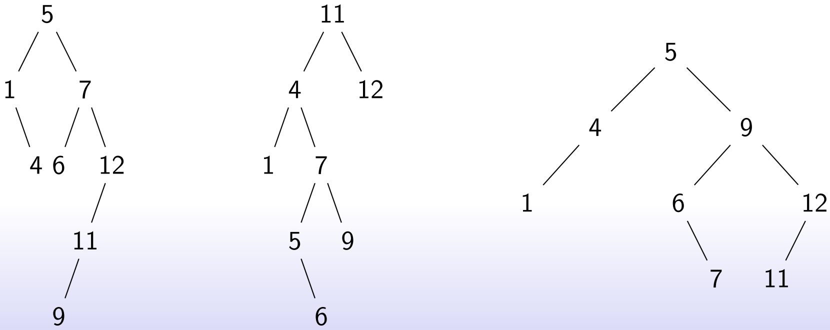






Construction d'un ABR

- Insertions successives en respectant la propriété précédente
- Il existe plusieurs ABR représentant le même ensemble
- Exemple : $E = \{1, 4, 5, 6, 7, 9, 11, 12\}$



Quel parcours donne l'arbre trié par ordre croissant?

Recherche d'un élément dans un ABR

Algorithme 3 : Rechercher(x, A)

début
- /* ENTRÉES: Un ABR A, un élément $x \in E$ */
- /* SORTIE: vrai si x appartient A, faux sinon */
- **si** est_vide(A) **alors**
  - Retour faux
- **sinon**
  - **si** x = racine(A) **alors**
    - Retour vrai
  - **sinon**
    - **si** $x >$ racine(A) **alors** Retour Rechercher($x, D(A)$)
    - **sinon** Retour Rechercher($x, G(A)$)

fin

Recherche d'un élément dans un ABR version itérative

Algorithme 4 : Rechercher_it(x, A)

début
  /* ENTRÉES: Un ABR A, un élément $x \in E$ */
  /* SORTIE: vrai si x appartient A, faux sinon */
  tant que not(est_vide(A)) et $x \neq$ racine(A) **faire**
    si $x >$ racine(A) **alors** $A \leftarrow D(A)$
    sinon $A \leftarrow G(A)$
  si est_vide(A) **alors** Retour faux
  sinon Retour vrai
fin

Complexité de la recherche

- Soit $C(A)$ = nombre de comparaisons
- Pire cas pour un arbre de hauteur h :
  - parcours de A jusqu'à une feuille $\neq x$, de hauteur la profondeur h de A

  $\rightarrow$ $h+1$ itérations (ou appels successifs), 2 comparaisons à chaque fois
  $C_{max}(A) = 2(h+1)$
- Pire cas pour les arbres de taille n et de hauteur h :
  - $h \leq n-1 \ \rightarrow \ C_{max}$ maximal $= 2n$ (arbres filaires)
  - $h \geq \log_2(n) \ \rightarrow \ C_{max}$ minimal $= 2(\log_2(n)+1)$ (arbres complets)
  - Pour une liste ordonnée de n éléments,
    la profondeur moyenne h des $n!$ ABR possibles est $\Theta(\log_2(n))$
    $\rightarrow$ en moyenne sur les ABR, $C_{max} = \Theta(\log_2(n))$

Insertion d'un élément dans un ABR

- Rechercher la place de l'élément à insérer
- Insérer l'élément
  - soit comme feuille
  - soit comme racine
- Le résultat doit être un *ABR*
  - *ajfeuille* : $E \times ABR \rightarrow ABR$
  - *ajracine* : $E \times ABR \rightarrow ABR$

Insertion d'un élément feuille dans un ABR

- Construction d'un *ABR* par adjonctions successives de feuilles
- Créer un *ABR* représentant $\{b, i, a, t, d\}$

Ajout d'une feuille dans un ABR : version récursive

Algorithme 5 : ajfeuille(x, A)

début

/* ENTRÉES: Un ABR A, un élément $x \in E$ */

/* SORTIE: Un ABR A */

si est_vide(A) **alors**

Retour $\langle x, \varnothing, \varnothing \rangle$

sinon

si $x >$ racine(A) **alors**

Retour $\langle$racine(A), $G(A)$, ajfeuille($x, D(A)$)$\rangle$

sinon Retour $\langle$racine(A), ajfeuille($x, G(A)$), $D(A)\rangle$

fin

Ajout d'une feuille : version itérative

Algorithme 6 : ajfeuille_iter(x, A)

début

  /* ENTRÉES: Un ABR A, un élément $x \in E$ */
  /* SORTIE: L'ABR A modifié*/
  $A_{sub} \leftarrow A$
  tant que not est_vide(A_{sub}) **faire**
    si $x >$ racine(A_{sub}) **alors**
      $A_{sub} \leftarrow D(A_{sub})$
    sinon
      $A_{sub} \leftarrow G(A_{sub})$
  $A_{sub} \leftarrow \langle x, \varnothing, \varnothing \rangle$

fin

Complexité de l'insertion comme feuille

- Soit $C(A) =$ nombre de comparaisons.
- Pire cas pour un arbre de hauteur h :
  - parcours de A jusqu'à une feuille $\neq x$, de hauteur la profondeur h de A

  $\rightarrow$ $h+1$ itérations (ou appels successifs), 1 comparaison à chaque fois
  $C_{max}(A) = h+1$
- De même que pour l'algorithme de recherche,
  - en moyenne sur les ABR, $C_{max} = \Theta(\log_2(n))$;
  - pour les arbres filaires, $C_{max} = n$;
  - pour les arbres complets, $C_{max} = \log_2(n) + 1$.
- Nombre d'affectations $Aff(A) = C(A)$

Insertion d'un élément racine dans un ABR

- Pour ajouter x à la racine de l'ABR A
  - Couper A en deux sous arbres G et D
  - G contient tous les éléments inférieurs ou égaux à x
  - D contient tous les éléments supérieurs à x
- Former l'arbre de racine x ayant G et D comme sous-arbres

Ajout d'une racine dans un ABR : algorithme couper

Algorithme 7 : coupe(x, A)

début

/* ENTRÉES: Un ABR A, un élément $x \in E$ */

/* SORTIE: Deux ABR G et D */

si est_vide(A) **alors** Retour $(\emptyset, \emptyset)$

sinon

  si racine$(A) \leq x$ **alors**

    $(G', D') \leftarrow$ coupe$(x, D(A))$

    Retour $(\langle \text{racine}(A), G(A), G' \rangle, D')$

  sinon

    $(G', D') \leftarrow$ coupe$(x, G(A))$

    Retour $(G', \langle \text{racine}(A), D', D(A) \rangle)$

fin

Ajout d'une racine dans un ABR : algorithme ajracine

Algorithme 8 : ajracine(x, A)

début

/* ENTRÉES: Un ABR A, un élément $x \in E$ */

/* SORTIE: Un ABR A */

$(G, D) \leftarrow$ coupe(x, A)

Retour $\langle x, G, D \rangle$

fin

Complexité de l'insertion comme racine

- Soit $C(A) =$ nombre de comparaisons.
- Pire cas pour un arbre de hauteur h :
  - parcours de A jusqu'à une feuille $\neq x$, de hauteur la profondeur h de A

  $\rightarrow$ $h+1$ itérations (ou appels successifs), 1 comparaison à chaque fois
  $C_{max}(A) = h+1$
- De même que pour l'insertion comme feuille,
  - en moyenne sur les ABR, $C_{max} = \Theta(\log_2(n))$;
  - pour les arbres filaires, $C_{max} = n$;
  - pour les arbres complets, $C_{max} = \log_2(n) + 1$.
- Nombre d'affectations $Aff(A) = 2C(A)$ (2 affectations dans le retour)

Supprimer un élément d'un ABR

- Recherche l'élément à supprimer
- Si c'est une feuille, suppression facile
- S'il a 1 fils, le remplacer par son fils
- S'il a deux fils,
  - soit on le remplace par le sommet qui lui est immédiatement inférieur = élément maximal de son sous-arbre gauche
  - soit on le remplace par le sommet qui lui est immédiatement supérieur = élément minimal de son sous-arbre droit
  - Les 2 solutions sont équivalentes si tous les noeuds sont distincts

  $\rightarrow$ besoin d'une fonction auxiliaire qui recherche et supprime le min ou le max d'un ABR

Suppression de l'élément max d'un ABR : algorithme supMax

Algorithme 9 : supMax(A)

début

/* ENTRÉES: Un ABR A */

/* SORTIE: l'élément max, l'ABR A privé de l'élément max*/

si est_vide(A) **alors** Retour A

sinon

  si est_vide($D(A)$) **alors**

    Retour (racine(A), $G(A)$)

  sinon

    $(x, B) \leftarrow$ supMax($D(A)$)

    Retour (x, $\langle$racine(A), $G(A)$, $B\rangle$)

fin

Suppression d'un élément d'un ABR

Algorithme 10 : supprime(x, A)

début

/* ENTRÉES: Un ABR A, un élément x */
/* SORTIE: L'ABR A privé de x */
si est_vide(A) **alors** Retour A
sinon
  si $x >$ racine(A) **alors**
    $B \leftarrow$ supprime$(x, D(A))$; Retour $\langle$racine$(A), G(A), B\rangle$
  sinon si $x <$ racine(A) **alors**
    $B \leftarrow$ supprime$(x, G(A))$; Retour $\langle$racine$(A), B, D(A)\rangle$
  sinon
    si est_vide$(G(A))$ **alors** Retour $D(A)$
    sinon si est_vide$(D(A))$ **alors** Retour $G(A)$
    sinon $(max, B) \leftarrow supMax(G(A))$; Retour $\langle max, B, D(A)\rangle$

fin

Complexité de la suppression

- Nombre de comparaisons :
  - supMax n'a pas de comparaisons
  - donc max de comparaisons en passant par "sinon si"
  - $C_{max}(h) = 2 + C_{max}(h-1)$ tant que $x <$ racine(A)
  - $C_{max}(h) = 2(h(x)+1) + C_{supMax}$
  - Donc C_{max} du même ordre que précédemment
- Nombre d'affectations :
  - $Aff_{supMax}(h) = 2 + Aff_{supMax}(h-1) = 2(h+1)$
  - $Aff_{max}(h) = 1 + Aff_{max}(h-1)$ tant que $x <$ racine(A)
  - $Aff_{max}(h) = h(x) + 1 + Aff_{supMax}(h - h(x) - 1) = 2h - h(x) + 1$
  - Donc Aff_{max} du même ordre que précédemment

Les arbres

Rotation gauche ou droite

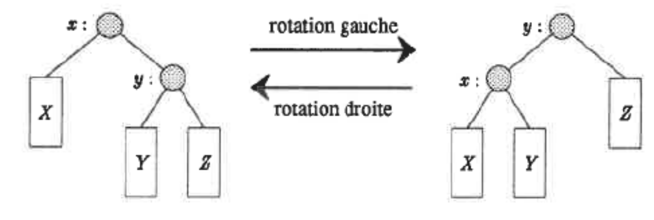
- Rotation gauche :

$$A = \langle x, X, \langle y, Y, Z \rangle\rangle \mapsto \mathcal{G}(A) = \langle y, \langle x, Y, Z \rangle, Z \rangle$$

- Rotation droite :

$$A = \langle y, \langle x, Y, Z \rangle, Z \rangle \mapsto \mathcal{D}(A) = \langle x, X, \langle y, Y, Z \rangle\rangle$$

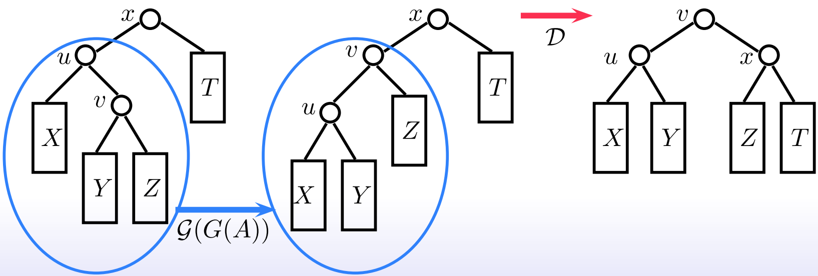
- Propriété : si A est un ABR, alors $\mathcal{G}(A)$ et $\mathcal{D}(A)$ aussi.



Figures issues de D. Beauquier, J. Berstel et Ph. Chrétienne, "Eléments d'algorithmique", Masson, 1992.

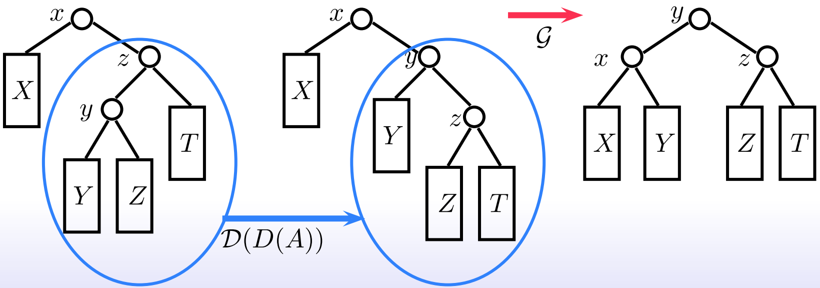
Rotation gauche-droite

- $A \mapsto \mathcal{D}\big(x, \mathcal{G}(G(A)), D(A)\big)$
- Préservation du caractère ABR.



Rotation droite-gauche

- $A \mapsto \mathcal{G}\big(x, G(A), \mathcal{D}(D(A))\big)$
- Préservation du caractère ABR.



Complexité des rotations

Algorithme 11 : Rotation gauche de A

début

/* ENTRÉES: Un ABR A */

/* SORTIE: A transformé en $\mathcal{G}(A)$ */

$B \leftarrow D(A)$

$D(A) \leftarrow G(B)$

$G(B) \leftarrow A$

$A \leftarrow B$

fin

Complexité constante quelle que soit la taille de A.

Les arbres

Arbres de Adelson-Velskii et Landis (AVL)

Définition : équilibre

Soit x un noeud d'un arbre et A_x le sous-arbre de racine x.
L'*équilibre* de x est : $\delta(x) = h\big(G(A_x)\big) - h\big(D(A_x)\big)$

Définition : AVL

Un arbre binaire de recherche A est un arbre AVL si, pour tout sommet x, les hauteurs des sous-arbres gauche et droite issus de ce sommet diffèrent d'au plus 1, *i.e.* $\delta(x) \in \{-1, 0, 1\}$.

Propriété des AVL

Pour tout arbre AVL A de hauteur h et de taille n,

$$\log_2(1+n) \leq 1 + h \leq 1,44 \log_2(2+n)$$

Conséquence :
complexité de recherche/insertion/suppression $= \Theta(\log_2(n))$

Preuve de $\log_2(n+1) \leq h+1 \leq 1,44\log_2(n+2)$

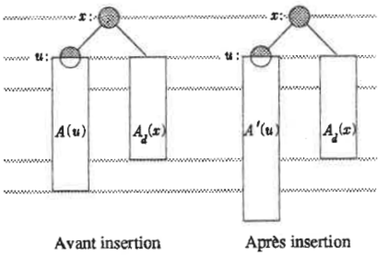
- Pour h donné, maximum de nœuds pour les arbres complets $\rightarrow n \leq 2^{h+1}-1$, donc $\log_2(n+1) \leq h+1$.
- Soit $N(h)$ = nombre minimal de nœuds des AVL non-vides de hauteur h
    - $N(0) = 1$, $N(1) = 2$
    - Pour $h \geq 2$, un des sous-arbres a pour hauteur $h-1$, l'autre a pour hauteur minimale $h-2$ (si plus petit, pas AVL). $\rightarrow N(h) = 1 + N(h-1) + N(h-2)$
    - Posons $F(h) = N(h) + 1$. Alors $F(h) = F(h-1) + F(h-2)$, avec $F(0) = 2$ et $F(1) = 3$:
      suite de Fibonacci décalée de 2 $\rightarrow F(h) = (\phi^{h+2} + \overline{\phi}^{h+2})/\sqrt{5}$
      avec $\phi = (1+\sqrt{5})/2$ et $\overline{\phi} = (1-\sqrt{5})/2$
    - $n \geq F(h) - 1 \longrightarrow h+1 \leq 1,44\log_2(n+2)$

Insertion dans un arbre AVL (1)

Insertion d'un sommet s :

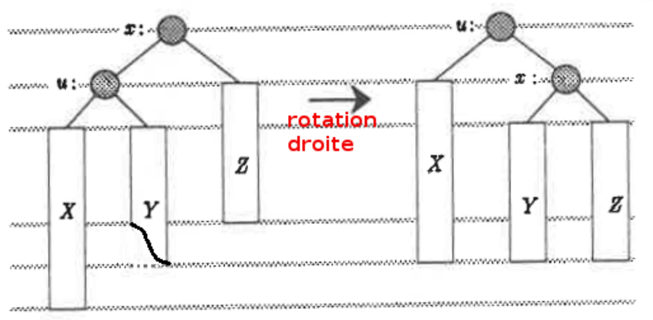
1. Insertion de s comme feuille : $ajf(s, A)$
2. Remonter l'arbre de s à la racine et vérifier l'équilibre de chaque nœud.
3. Deux cas possibles :
   - Si OK, c'est fini, l'arbre est toujours AVL
   - Sinon, soit x le premier nœud rencontré tel que $\delta(x) = \pm 2$
     Ré-équilibrer le sous-arbre issu de x.

Supposons qu'après insertion, $\delta(x) = 2$:



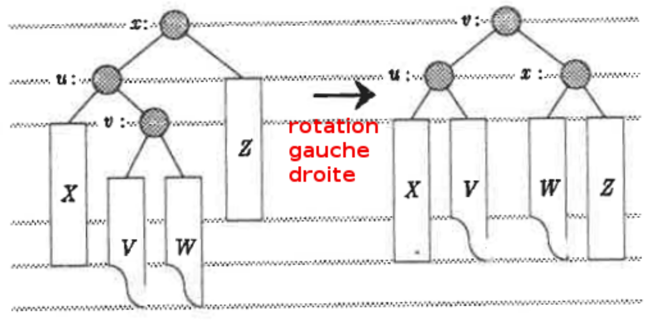
Insertion dans un arbre AVL (2)

1er cas : s a été inséré dans le sous-arbre gauche de u



Insertion dans un arbre AVL (3)

2nd cas : s a été inséré dans le sous-arbre droit de u

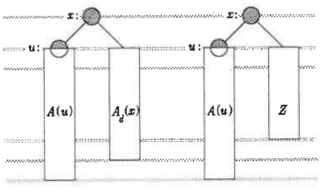


Suppression dans un arbre AVL (1)

1. Suppression d'un sommet s :
   - Si s est une feuille, on le supprime
   - Si s a 1 fils, on le remplace par son fils
   - Si s a 2 fils, on supprime le max de son sous-arbre gauche qui va remplacer s

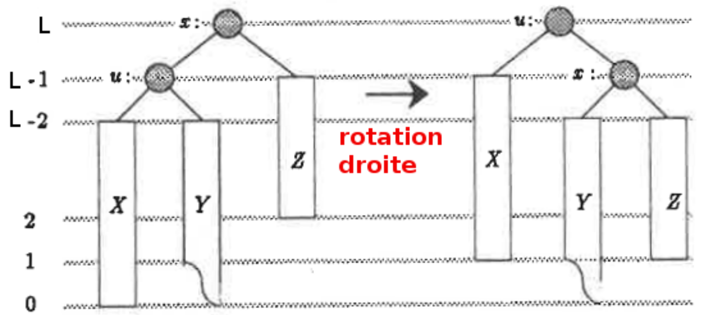
   Dans tous les cas, suppression d'un nœud $s' = s$ ou un descendant de s.
2. Remonter l'arbre de s' à la racine et vérifier l'équilibre de chaque nœud.
3. Deux cas possibles :
   - Si OK, c'est fini, l'arbre est toujours un AVL
   - Sinon, soit x le premier nœud rencontré tel que $\delta(x) = \pm 2$
     Ré-équilibrer le sous-arbre issu de x.

Supposons qu'après suppression, $\delta(x) = 2$:



Suppression dans un arbre AVL (2)

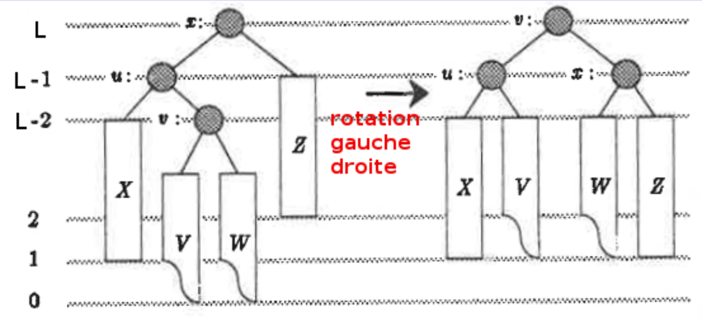
1er cas :



- Si le nouveau sous-arbre a pour hauteur L, c'est équilibré.
- Sinon, continuer à équilibrer en remontant de u vers la racine.
  (car le sous-arbre d'un nœud frère ou cousin de u peut avoir une hauteur $L+1$)

Suppression dans un arbre AVL (3)

2nd cas :



Le nouveau sous-arbre a pour hauteur $L - 1$
$\rightarrow$ continuer à équilibrer en remontant de v vers la racine.
(car le sous-arbre d'un nœud frère ou cousin de v peut avoir une hauteur $L + 1$)

Complexité totale

- Insertion/suppression : $\Theta(h)$, donc $\Theta(\log_2(n))$
- Remontée et vérification des équilibres : idem
  Nécessaire de stocker dans chaque sommet la hauteur de son sous-arbre
- Ré-équilibrage : max h ré-équilibrages, chacun de complexité constante
  Donc $\Theta(h)$, donc $\Theta(\log_2(n))$

Bilan :
complexité insertion/suppression dans un arbre AVL =
$\Theta(h) = \Theta(\log_2(n))$