

Structures de Données & Algorithmes II

-

Arbres & Forêts

Pascal Mérindol (CM, TD, TP)

merindol@unistra.fr

<http://www-r2.u-strasbg.fr/~merindol>

Contenu

- **Les tris (6-8h)**
- **Arbres & Forêts (12-14h)**
 - Définitions et propriétés
 - Spécifications et implémentations
 - Parcours (récurrsifs) en profondeur
 - Dérécursivation (infixe / préfixe)
 - Arbres binaires triés (BST) et équilibrés (AVL,...)
 - + Quelques bonus
- **Les graphes (14-16h)**
- **Les tables (6-8h)**

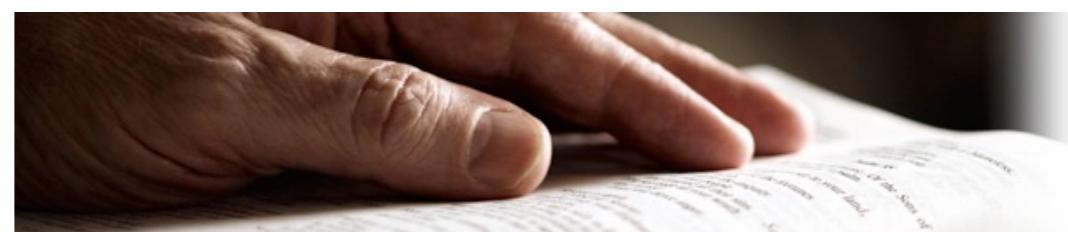
Références

<http://users.dcc.uchile.cl/~rbaeza/handbook/hbook.html>

...

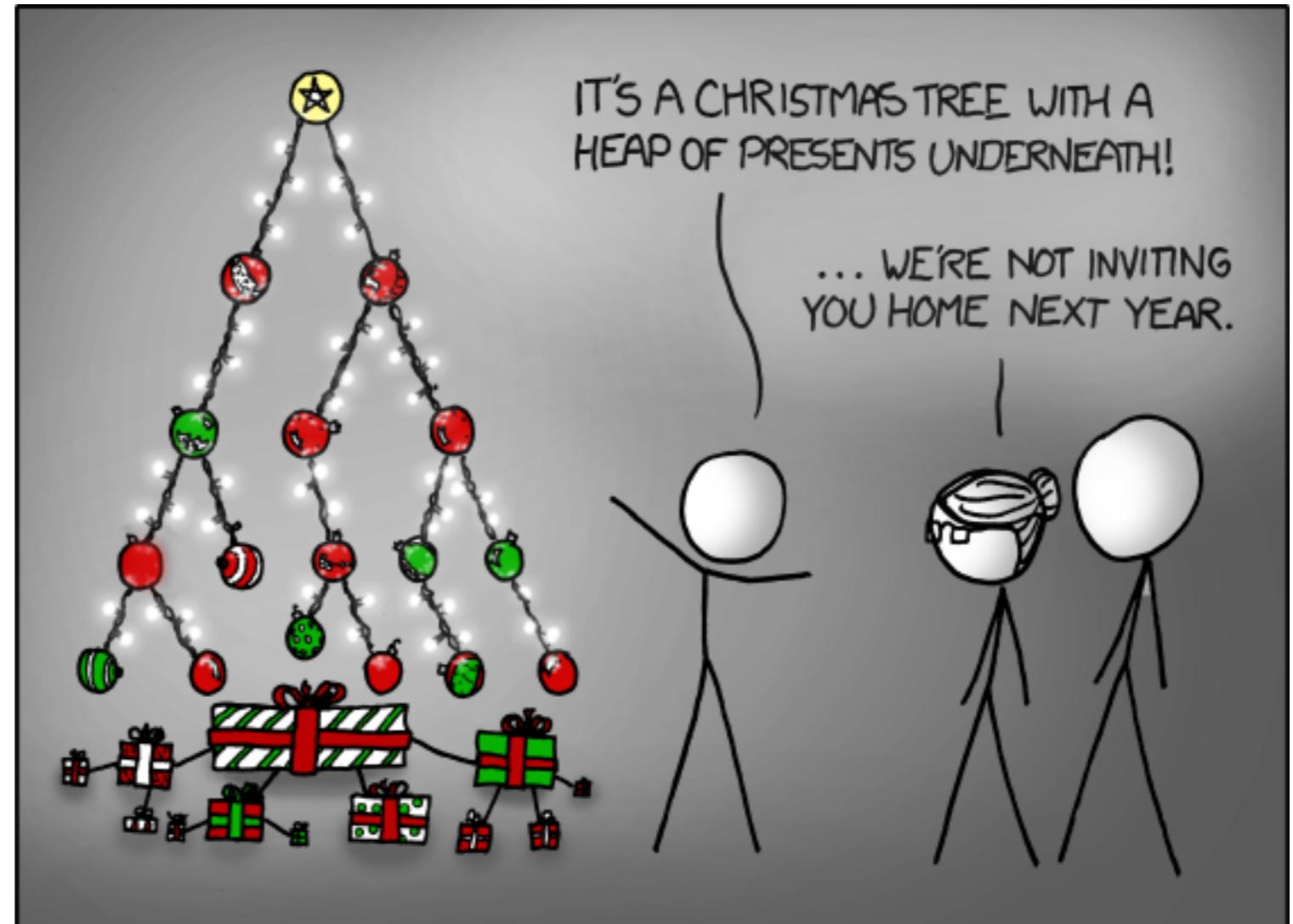


- * ***Spécification algébrique, algorithmique et programmation*** : Jean-François Dufourd, Dominique Bechmann, Yves Bertrand.
- * ***The Art of Computer Programming, Volume 1 (+3), Fundamental Algorithms (Chapter 2.3 + Chapter 6)*** : Donald Knuth.
- * ***Introduction to Algorithms*** : Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.



Motivations

- Etudes combinatoires
- Données hiérarchiques
- Recherche d'infos efficace
- Tris efficaces
- Routage (arbres de cheminement) & commutation (tries pour lookup rapide) !
- ...

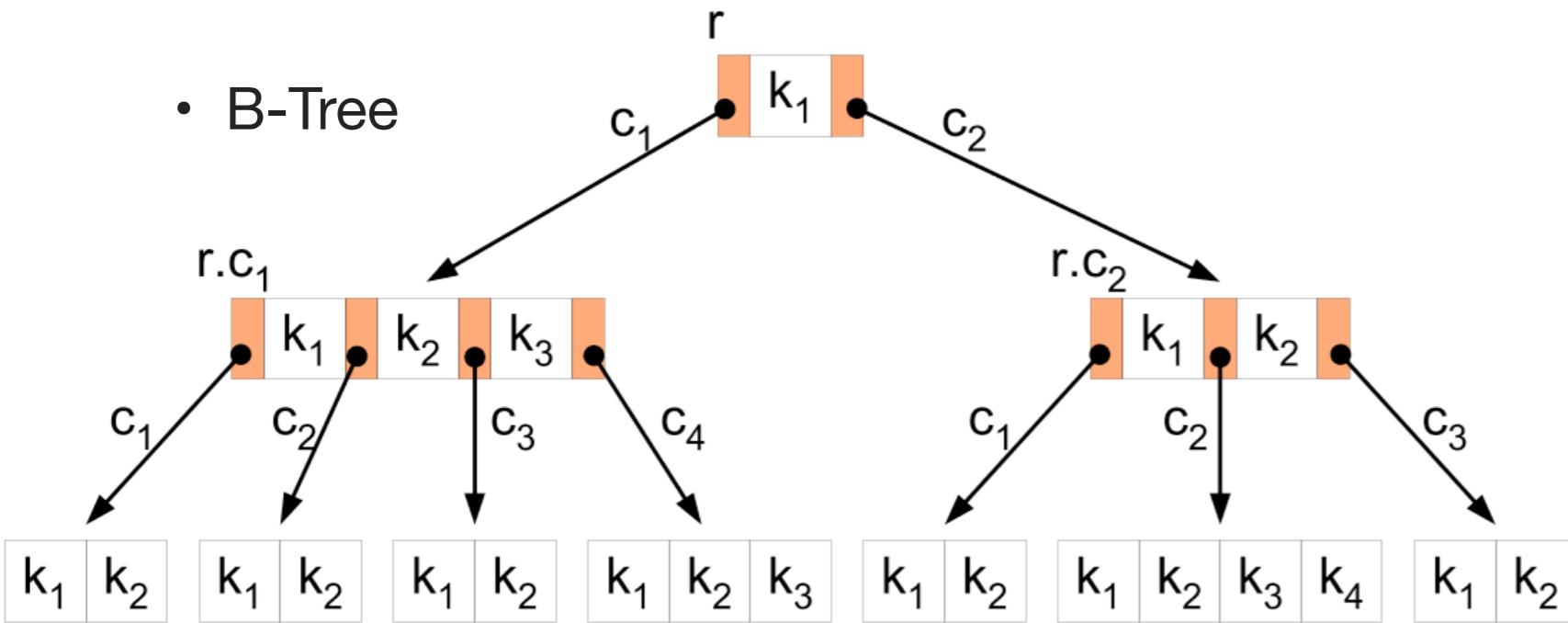


Dans ce cours

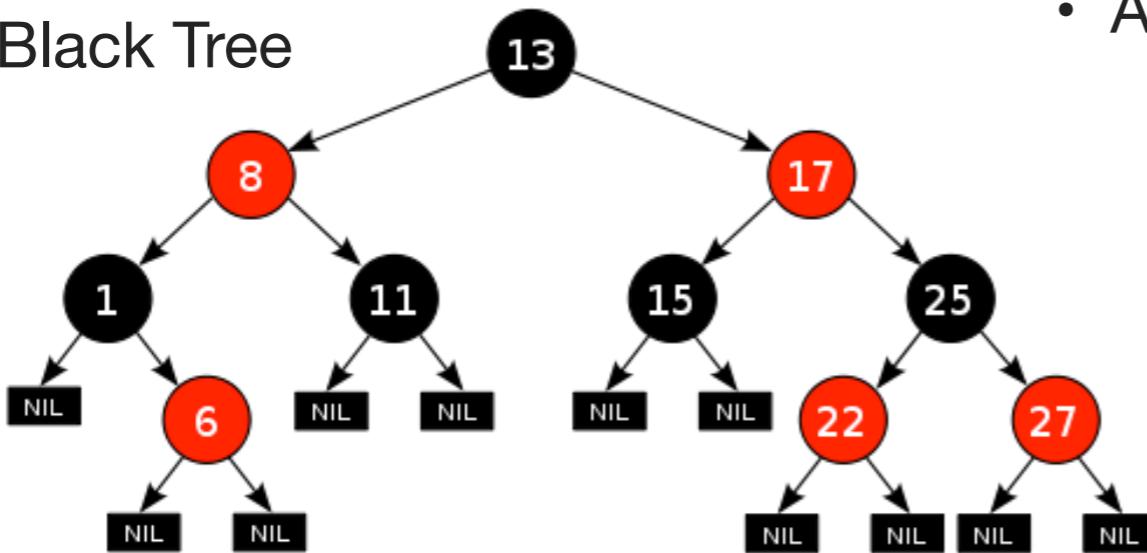
- **Les arbres et forêts spécifiques aux structures de données informatique**
 - orientés, enracinés, ordonnés, étiquetés !
 - définitions, propriétés, spécifications, représentations, ..
 - parcours, dérécursivation, puis «arbres binaires triés» : specs & complexité
- **Trois principaux types de structures de données basées sur les arbres**
 - Arbres de recherche / Search Trees
 - Binary Search Tree, AVL, Red-Black Tree, B-Tree, 2-3 Tree , AA Tree, etc.
 - Tas / Heaps
 - Binary Heap, Binomial Heap, Fibonnaci Heap, etc.
 - Tries (arbres lexicographiques) / Prefix Trees
 - Radix, Patricia, etc.

Balanced Search Tree

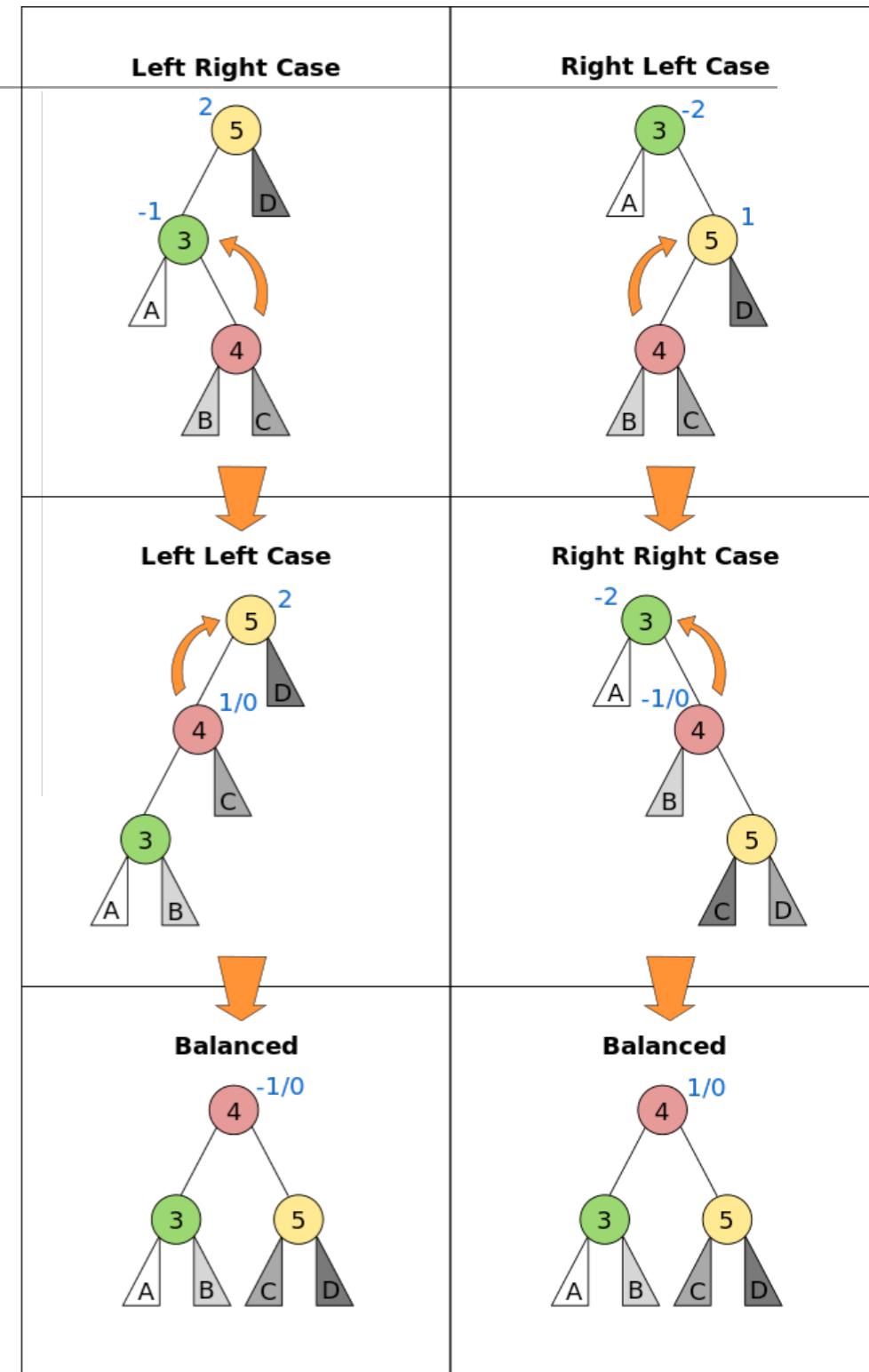
- B-Tree



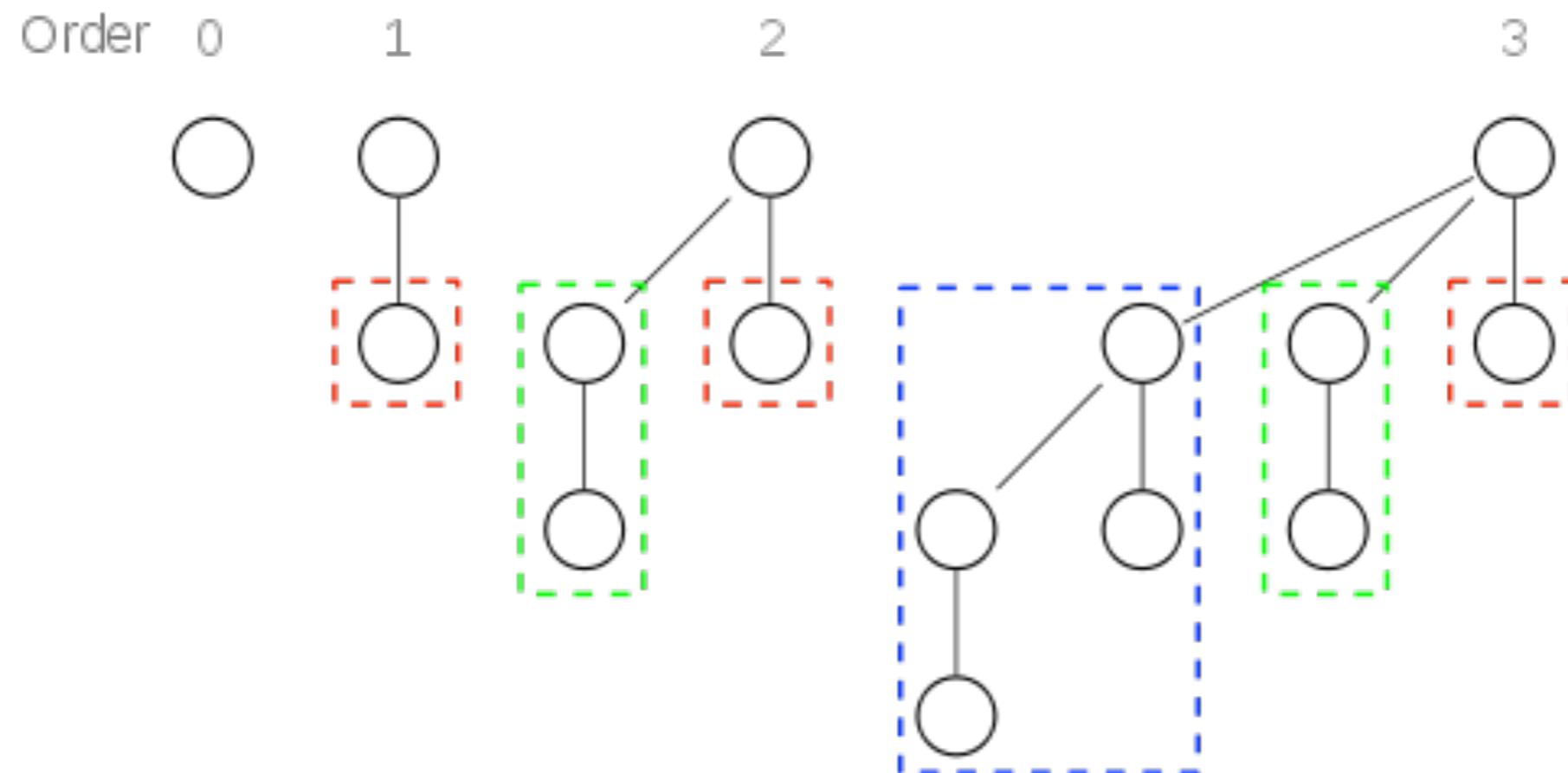
- Red-Black Tree



- AVL Tree

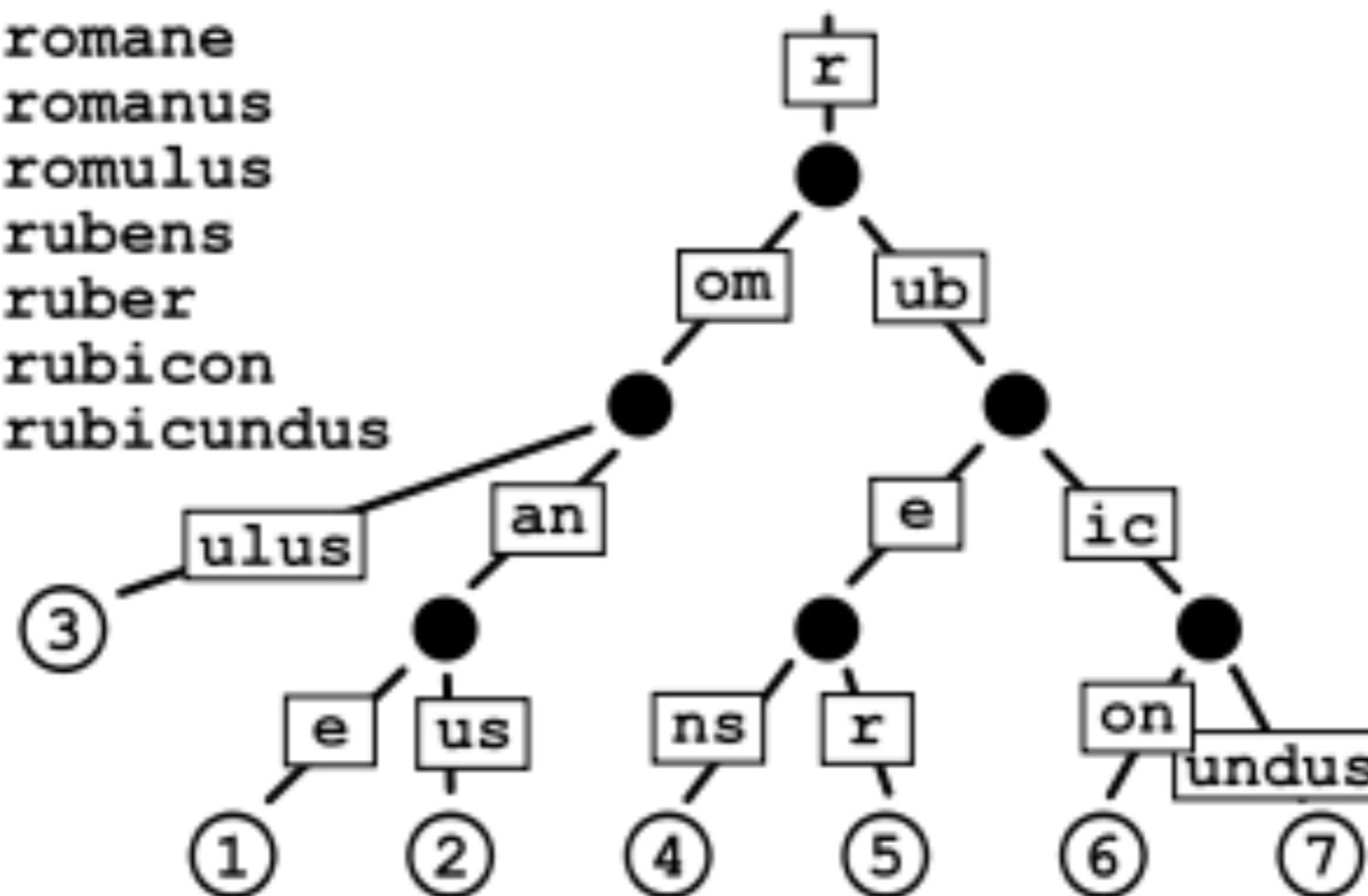


Heaps (arbres binomiaux sur l'exemple)



Tries

- 1 romane
- 2 romanus
- 3 romulus
- 4 rubens
- 5 ruber
- 6 rubicon
- 7 rubicundus



Définitions & Propriétés

- **Forêt orienté := un couple (N, T)**
 - un ensemble de noeuds N formant un graphe simple acyclique
 - $T : N \rightarrow N$ la relation d'arborescence (père \rightarrow fils) tel que $T^{-1}(x)$ est composé
 - d'aucun prédecesseur : racines des arborescences i.e. des arbres disjoints
 - sinon d'un père :
 - noeud interne père $\rightarrow \dots \rightarrow$ père \rightarrow fils i.e. $\text{pred } T^{-1}(x)$ et $\text{succ } T(x) \setminus \emptyset$
 - feuilles i.e. $T(x)$ est vide
- **Notes :**
 - Seules les racines sont connectées à l'ensemble des noeuds dans leurs composantes d'équivalence non orienté (un arbre simple connecté).
 - On peut ordonner par la hauteur avec une lecture de gauche à droite (convention) :
 - $i.j.k\dots$ avec $i.j.k\dots$ mot dans $N.N.N\dots$

Terminologie de base

- **frère** : noeuds de même père
- **ascendant/descendant (strict, large) = père/fils**
 - -> 2 noeuds sont dans la même arborescence s'ils ont un ascendant commun (large).
- **hauteur d'un noeud = niveau/profondeur**
- **hauteur d'un arbre = hauteur de sa feuille la plus profonde (taille =#noeuds)**
- **arité = degré de branchement**
- **m-aires : #fils autorisés par père**
- **branche : suite de noeud d'une racine vers une feuille**
 - -> en appliquant la fonction T^{-1} depuis la feuille
- **Enracinement d'une forêt** : définir une racine virtuelle pour fusionner les arbres

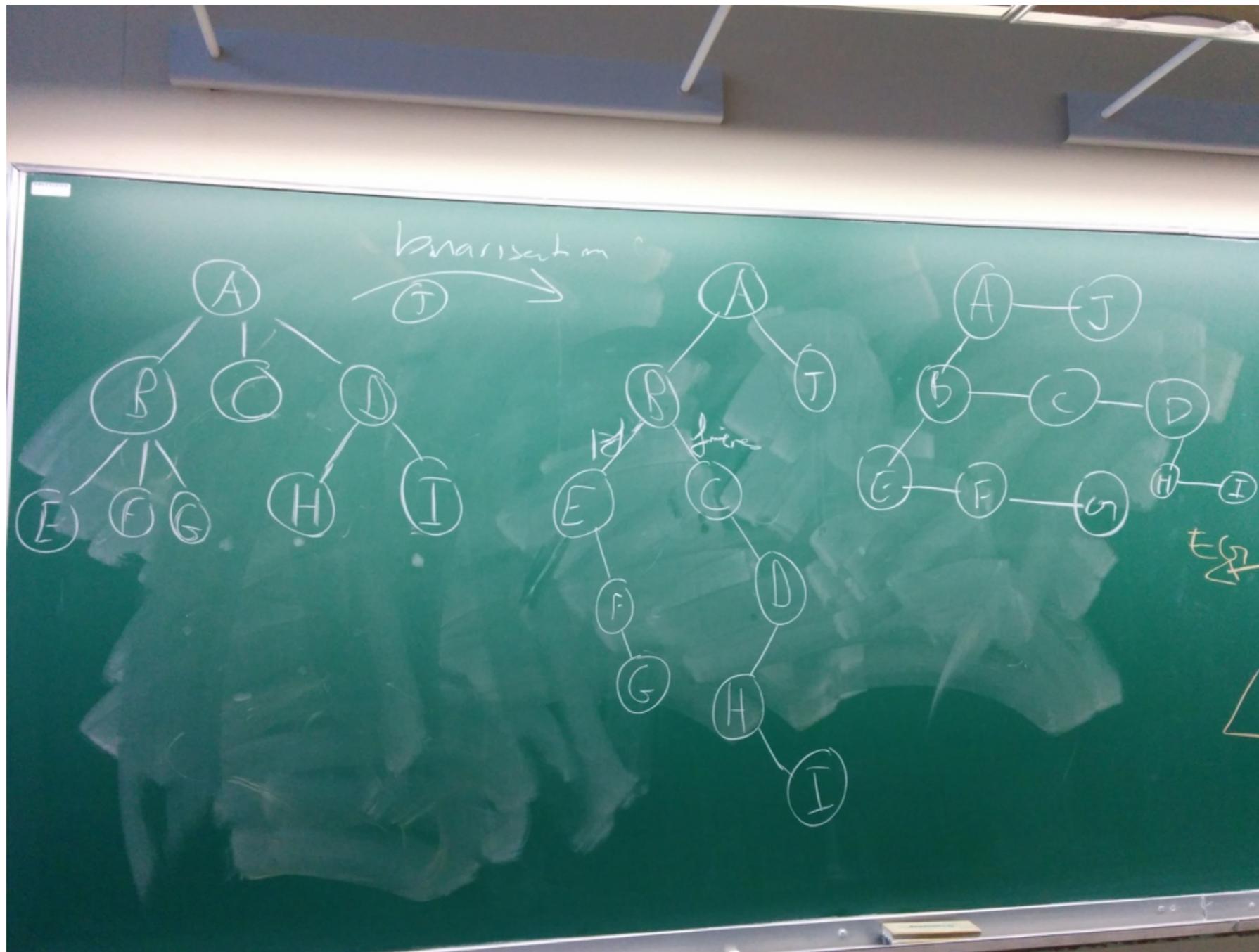
Mesures d'arbres

- **Longueurs de cheminement**
 - simple : somme de la hauteur de tous les noeuds
 - externe : somme de la hauteur de toutes les feuilles
 - interne : somme de la hauteur de tous les noeuds internes
- => hauteurs/profondeurs moyennes si division par #noeuds d'une catégorie
- **exercices pour plus tard...**

Arbre binaire

- **Définitions :**
 - Un arbre binaire est une arborescence 2-aire avec distinction, en chaque noeud interne, du fils gauche et du fils droit.
 - Bords gauche / droit (pas tjs une branche -> quel cas ?)
 - Extrémités gauche / droit (référence vers sous-arbres)
- **Binarisation d'un arbre général**
 - => exercice : quelle représentation visuelle ?
- **Propriétés :**
 - Un arbre binaire de hauteur p et de $n > 0$ noeuds vérifie : $\lfloor \log_2(n) \rfloor \leq p \leq n - 1$
 - si «parfait» alors $p \sim \log_2(n)$
 - si «filiforme» alors $p \sim n-1$
- **Exercice : démontrer les propriétés**

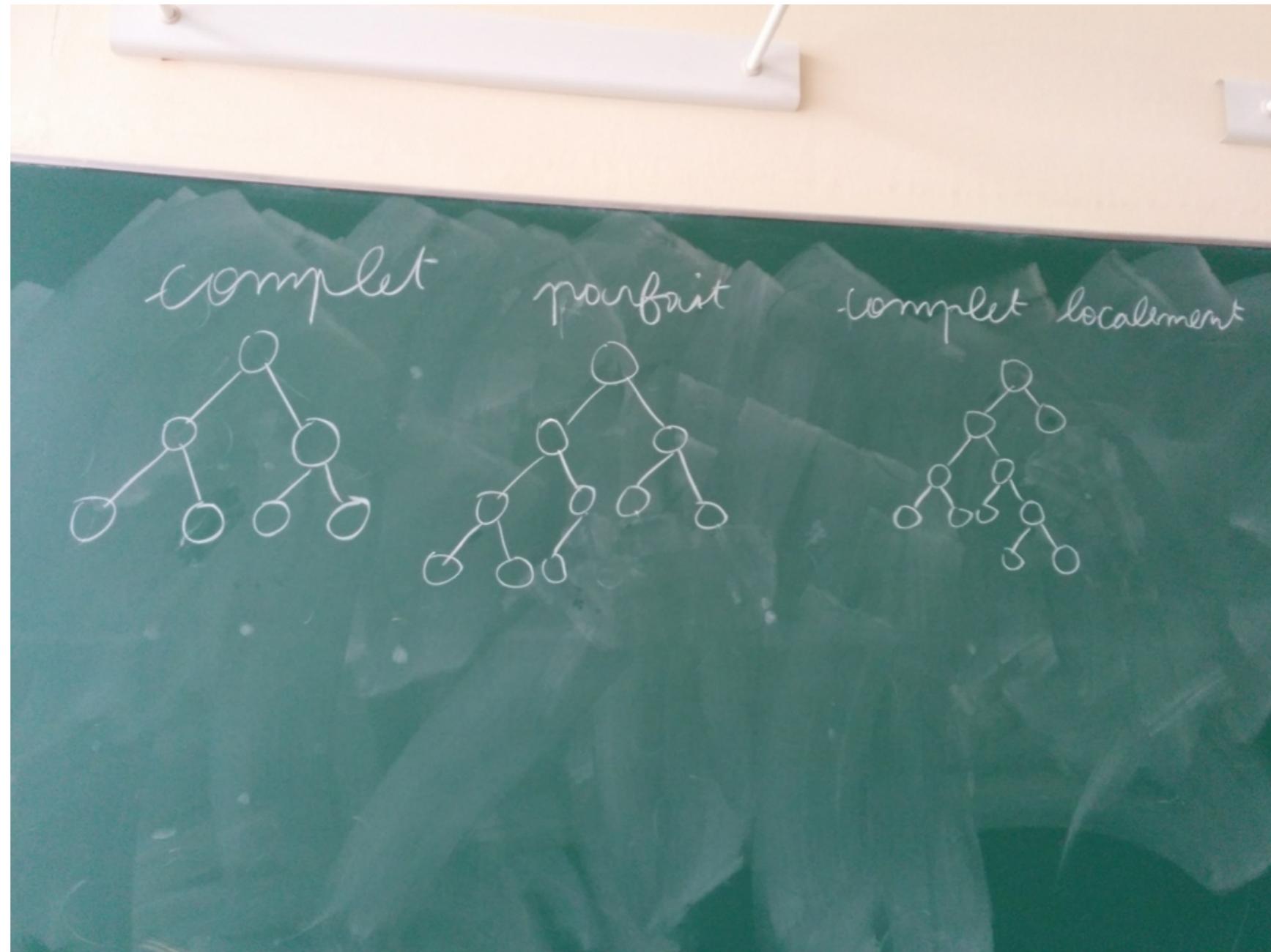
Binarisation



Arbre binaire : complétude & perfection

- **Complétude locale** : un arbre est localement complet si tous ses noeuds internes ont exactement deux fils.
 - -> taille impaire $2n'+1$ avec n' noeuds internes et $n'+1$ feuilles
- **Un niveau k est complet s'il contient exactement 2^k noeuds**
- **Perfection** : un arbre binaire est parfait si tous ses niveaux sont complets sauf éventuellement le dernier \Leftrightarrow les feuilles sont répartis sur deux niveaux au pire (~tas).
 - les feuilles sont regroupés le plus à gauche possible...
 - -> hauteur $p = \lfloor \log_2(n) \rfloor$ avec $\lceil n / 2 \rceil$ feuilles dont
 - $n - 2^p + 1$ de niveau p
 - $2^p - \lceil n / 2 \rceil - 1$ de niveau p-1
- **Complétude** : un arbre binaire est complet s'il est localement complet, (parfait) et avec toutes ses feuilles au même niveau.
- => **exercices**
 - dessiner plusieurs exemples vérifiant l'une ou l'autre propriété + preuves

Complet vs. Parfait vs. Localement Complet



Spécifications

- **spéc ARBIN étend EG ($S, _==_, _!=_$)**
- **sorte Arbin**
- **opérations**
 - $\bar{\wedge} : - \rightarrow \text{Arbin} /* \text{ arbre vide } */$
 - $e : \text{Arbin} S \text{ Arbin} \rightarrow \text{Arbin} /* \text{ enracinement } */$
 - $ag, ad : \text{Arbin} \rightarrow \text{Arbin} /* \text{ sous arbres g,d } */$
 - $r : \text{Arbin} \rightarrow S /* \text{ étiquette à la racine } */$
 - $v : \text{Arbin} \rightarrow \text{Bool} /* \text{ test de vacuité } */$
- **préconditions :**
 - pré $r(a) = \neg v(a)$
- **axiomes ?**

SPEC ARBIN

- **axiomes !**

- $\text{ag}(\bar{\wedge}) = \text{ad}(\bar{\wedge}) = \bar{\wedge}$
- $\text{ag}(e(a_1, r, a_2)) = a_1$
- $\text{ad}(e(a_1, r, a_2)) = a_2$
- $r(e(a_1, r, a_2)) = r$
- $v(\bar{\wedge}) = \text{vrai}$
- $v(e(\dots)) = \text{faux}$
- plus d'opérations basiques ?

SPEC ARBIN1 étend ARBIN

- **opérations**

- $h : \text{Arbin} \rightarrow \text{Int} /* \text{ hauteur } */$
- $\text{eg}, \text{ed} : \text{Arbin} \rightarrow \text{Arbin} /* \text{ extrémité } */$
- $f : \text{Arbin} \rightarrow \text{Bool} /* \text{ feuille ? } */$
- $\text{nn}, \text{ni}, \text{nf} : \text{Arbin} \rightarrow \text{Int} /* \#noeuds, \#n.internes, \#n.feuilles */$

- **axiomes en exercice !**

- + spécifications des opérations de base, de cheminement, d'appartenance, etc.

- **Convention : la hauteur de l'arbre vide est de -1.**

Spécifications des axiomes

The left photograph shows handwritten definitions for several functions:

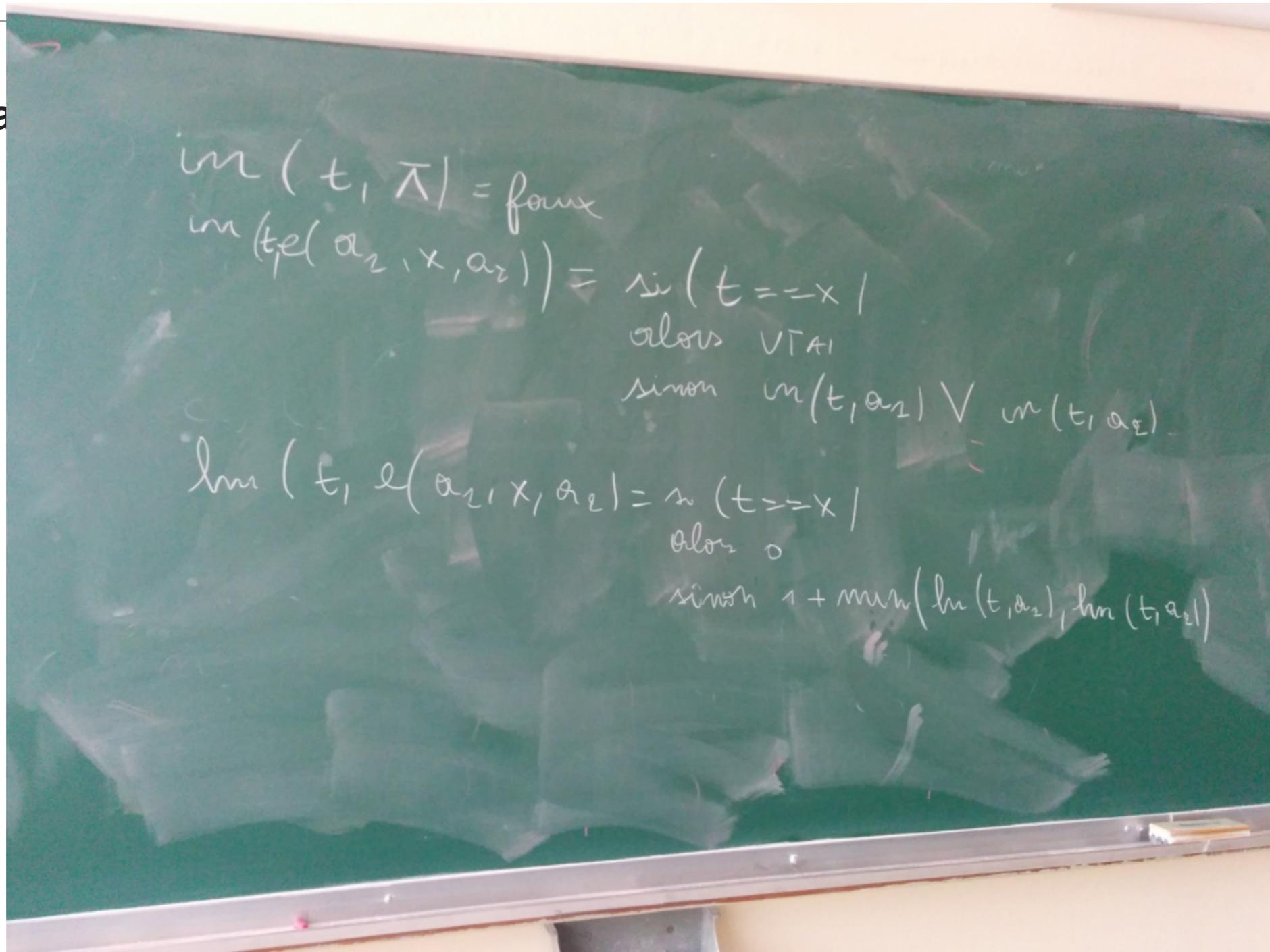
- $h(\Delta) = -1$
- $h(e(a_1, x, a_2)) = 1 + \max(h(a_1), h(a_2))$
- $eg(e(a_1, x, a_2)) = \begin{cases} v(a_1) & \text{alors } e(a_1, x, a_2) \\ \text{sinon } eg(a_1) & \end{cases}$
- $cd(e(a_1, x, a_2)) = \begin{cases} v(a_2) & \text{alors } e(a_1, x, a_2) \\ \text{alors } 1 & \\ \text{sinon } cd(a_2) & \end{cases}$
- $nf(e(a_1, x, a_2)) = \begin{cases} v(a_2) & \text{alors } e(a_1, x, a_2) \\ \text{alors } 1 & \\ \text{sinon } nf(a_1) + nf(a_2) & \end{cases}$
- $f(e(a_1, x, a_2)) = \begin{cases} v(a_1) \wedge v(a_2) & \\ mn(e(a_1, x, a_2)) = \\ 1 + mn(a_1) + mn(a_2) & \\ mn(\Delta) = 0 & \end{cases}$

The right photograph shows the definition of mn and $mn(\Delta)$:

$$\begin{cases} mn(e(a_1, x, a_2)) = \\ 1 + mn(a_1) + mn(a_2) \\ mn(\Delta) = 0 \end{cases}$$

Spécification des opérations

- Appartenance



Spécification des opérations

- **Longueur de cheminement**

$$l_c(e(a_1, n, a_2)) = l_c(a_1) + l_c(a_2) + mn(a_1) + mn(a_2)$$

(+ 3 cas spéciaux : arbre vide
· a_1 vide
· a_2 vide)

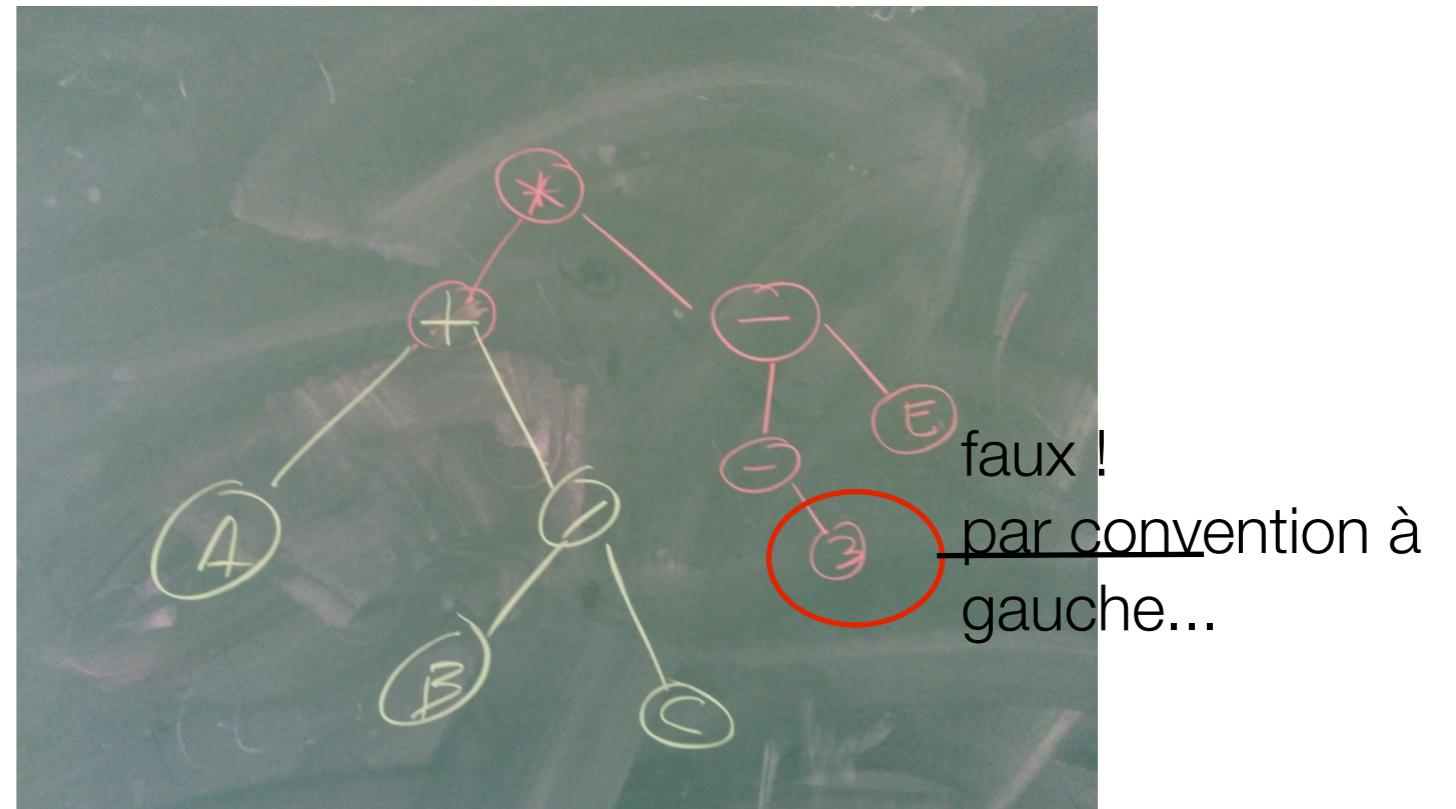
Représentations spécifiques

- **Représentation chainée pour les arbres binaires**
- **typedef struct arbibin {**
 - **struct arbibin * g;**
 - **S rac;**
 - **struct arbibin * d;**
- **}** **Noeud, *Arbin;**
- **Représentation des arbres généraux par liste d'adjacence**
- **typedef struct foforet {**
 - **S rac;**
 - **struct foforet * pf;**
 - **Liste lf; // éléments de type Noeud**
- **}** **Noeud, *Foret;**

Visualiser et manipuler ces représentations...

- **Exercice 1 :**

- Construire l'arbre binaire sur l'expression arithmétique $(A+B/C)^*((-3)-E)$
- Forêt quelconque (~ binarisation)



- **Exercice 2 : insertion itérative d'un (sous) arbre dans une forêt chainée**

- b inséré en tant que k^{ème} fils de a

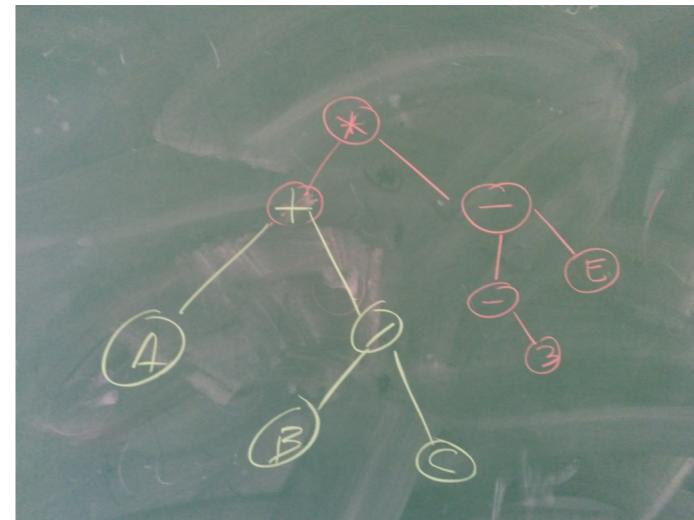
Représentation chainée dans un tableau

- **Un tableau pour les étiquettes**
- **Un tableau pour le «chainage»**
 - les «pointeurs» sont alors les indices du tableau
 - un des liens peut-être représenté par contiguïté
- **Cas général : la fonction père «est la chainage»**
 - conventions : père à 0 si racine ou à -1 si en dehors de la forêt
- **Cas particuliers (e.g., expression arithmétique binarisée) :**
 - lien droit par chainage, lien gauche par contiguïté
 - exercice sur l'expression par terme : $(A+B/C)^*((-3)-E)$

Lien droit par chainage, lien gauche par contiguïté

d	0	1	2	3	4	5	6	7	8	9
g	*	+	A	/	B	C	-	3	E	

- $(A+B/C)^*((-3)-E)$



Erratum bis (convention) :

- * -2 feuille
- * -1 pas de fils droit

6	3	-2	5	-2	-2	9	-1	-2	-2
*	+	A	/	B	C	-	-	3	E

Représentation contigue (arbres binaires parfaits ~ tas)

- **Depuis un noeud d'indice k :**

- descente dans l'arbre
 - à gauche : $2k + 1$
 - à droite : $2k + 2$
- montée dans l'arbre : $\lfloor (k - 1) / 2 \rfloor$

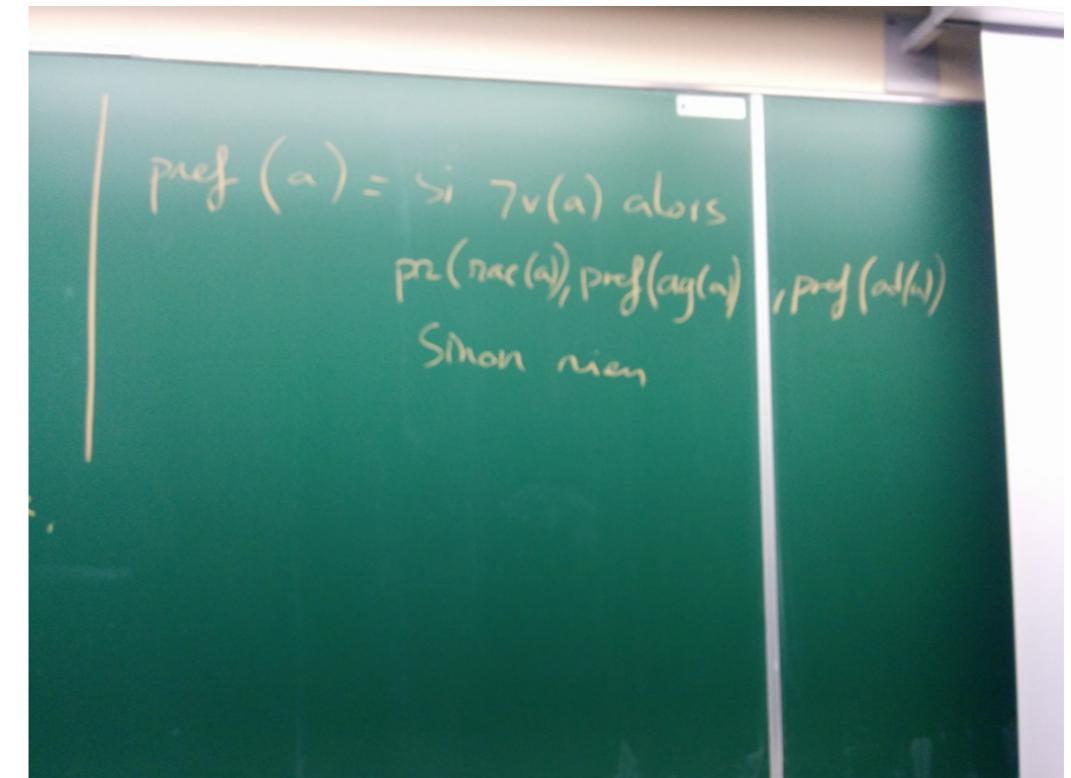
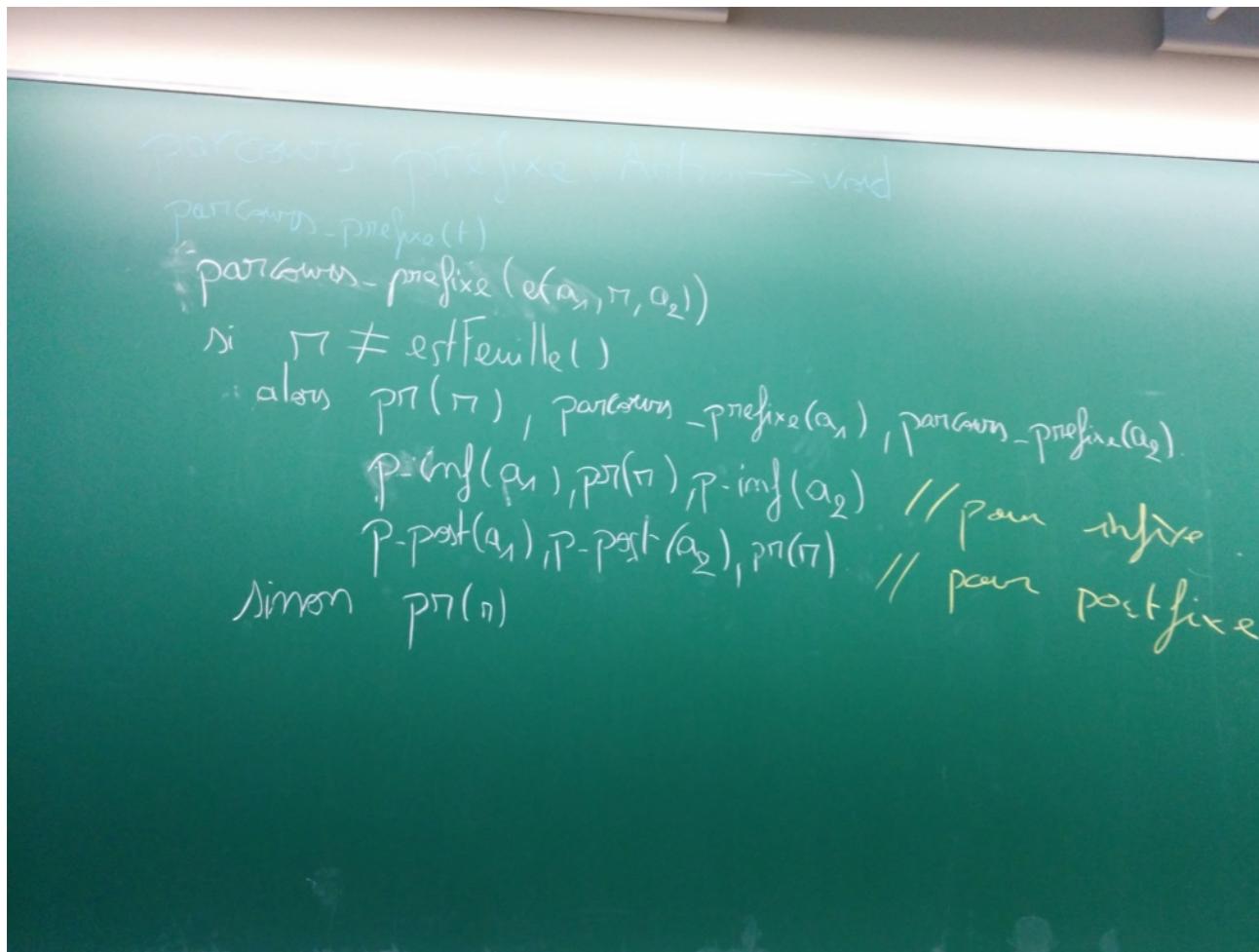
- **typedef struct arbibint {**

- **int nbn;**
- **S v[n];**
- **} Arbin, *Arbin;**

Parcours

- **Procédure *pr* : S -> void sur tous les sommets de notre foret**
 - simple si appliquée une seule fois sur chaque sommet
 - multiple sinon.
- **Ici : parcours en profondeur d'abord sur arbres binaires étiquetés**
 - 3 stratégies
 - ordre préfixe : r g d
 - ordre infixé : g r d
 - ordre postfixe : g d r
- **Exercices : spécifier/implémenter ces trois parcours en récursif !**

Spécifications parcours



Dérécursivation

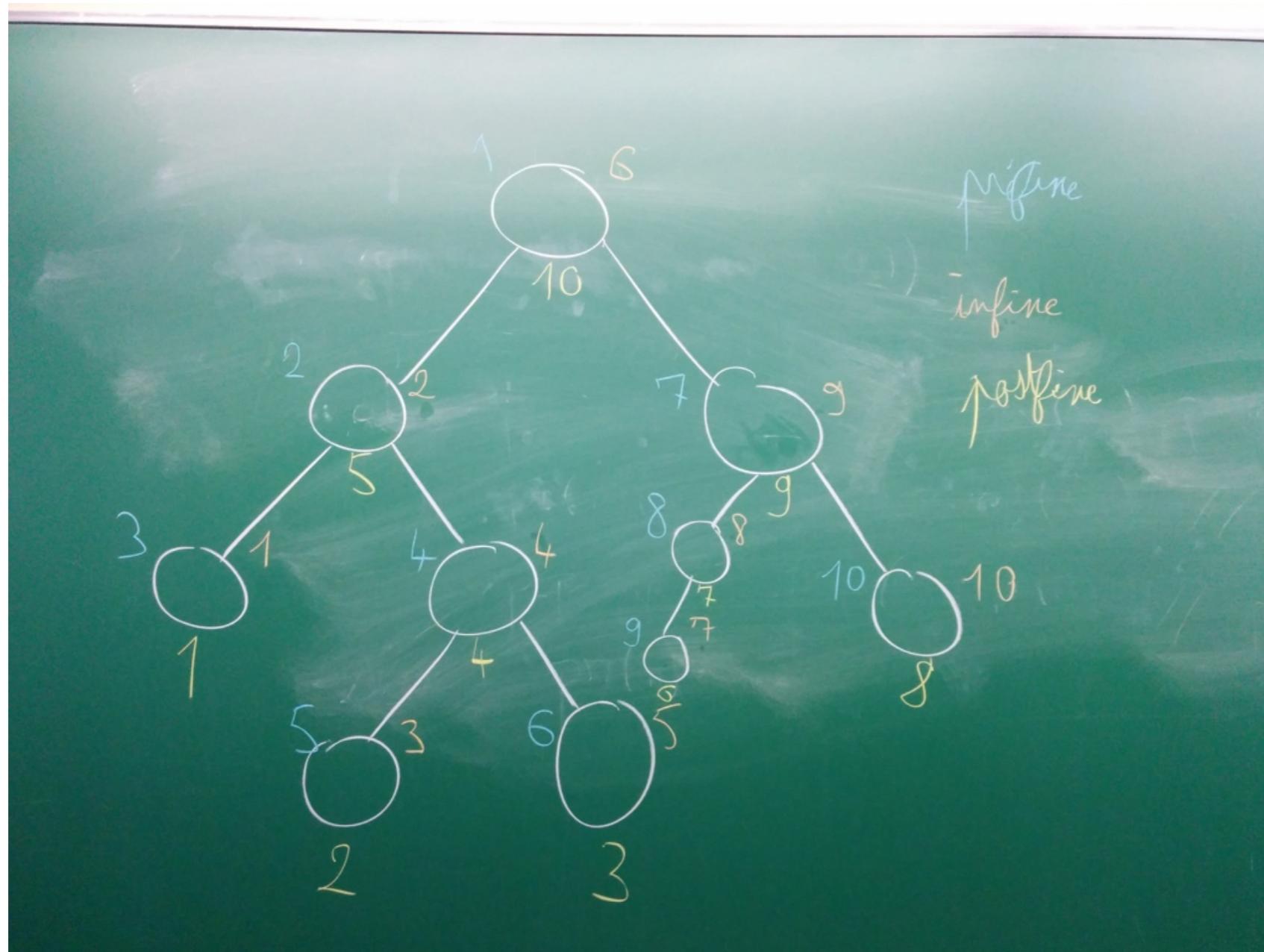
- **Exemple sur parcours ordre infixé**
 - => «retrouver» l'ordre de parcours des sommets
- **Liste avec une tête et une queue**
 - extrémité gauche vs. extrémité droite
 - descendre (facile !) et remonter ?
 - fil d'ariane = référence aux sous-arbres par lesquels la descente s'est fait !
- => pile d'arbres +/- infos spécifiques
 - empiler ~ descendre
 - dépiler ~ monter
 - exercices : à spécifier/implémenter pour chaque ordre !

Ordre : représentation visuelle

- **Exercices :**

- Donner pour chaque noeud, et sur le même arbre quelconque, le numéro de l'étape de parcours (décrivant l'ordre de visite), pour
 - l'ordre infixé
 - l'ordre préfixé
 - l'ordre postfixé
- Spécifier en mode itératif les trois ordres
 - postfixe un peu plus dur (pour l'exam ?)

Parcours : représentation visuelle



Dérécursivation : spécification ordre infixé

- **desg** : Arbin Pile \rightarrow Pile
 - pré **desg**(a , p) = $\neg v(a)$
 - **desg**(a , p) = p'
 - avec (a' , p') = init (a , p) tant que $\neg v(a')$
 - répéter (**ag**(a'), **empiler**(p' , a'))
 - **infix**(a , pr) = rien
 - avec (p , $todo$) = init (**desg**(a , **pilenouv**), rien) tant que $\neg \text{vide}(p)$
 - répéter si $v(\text{ad}(a'))$
 - alors (p' , $todo'$)
 - sinon (**desg**($\text{ad}(a')$, p'), $todo'$)
 - avec (a' , p' , $todo'$) = (**sommet**(p), **dépiler**(p), $pr(r(a'))$)

Spécification Ordre Préfixe

$\text{prefix}(a) = \text{rien}$

avec $(p) = \text{init}(a, p)$ tant que $\neg \text{vide}(p)$

répéter $(a', p') = (\text{desg}(\text{sommet}(p')), \text{dep.bu}(p'))$

$\text{desg} : \text{Arbim}, \text{Pile} \rightarrow \text{Pile}$

$\text{pré} : \text{desg}(a, p) = \neg V(a)$

$\text{desg}(a, p) = p'$

avec $(a', p') = \text{init}(a, p)$ tant que $\neg V(a')$

répéter $(p \sqcap(a'), \text{ag}(a'), \text{test})$

avec $\text{test} \Leftarrow \neg V(\text{ad}(a'))$

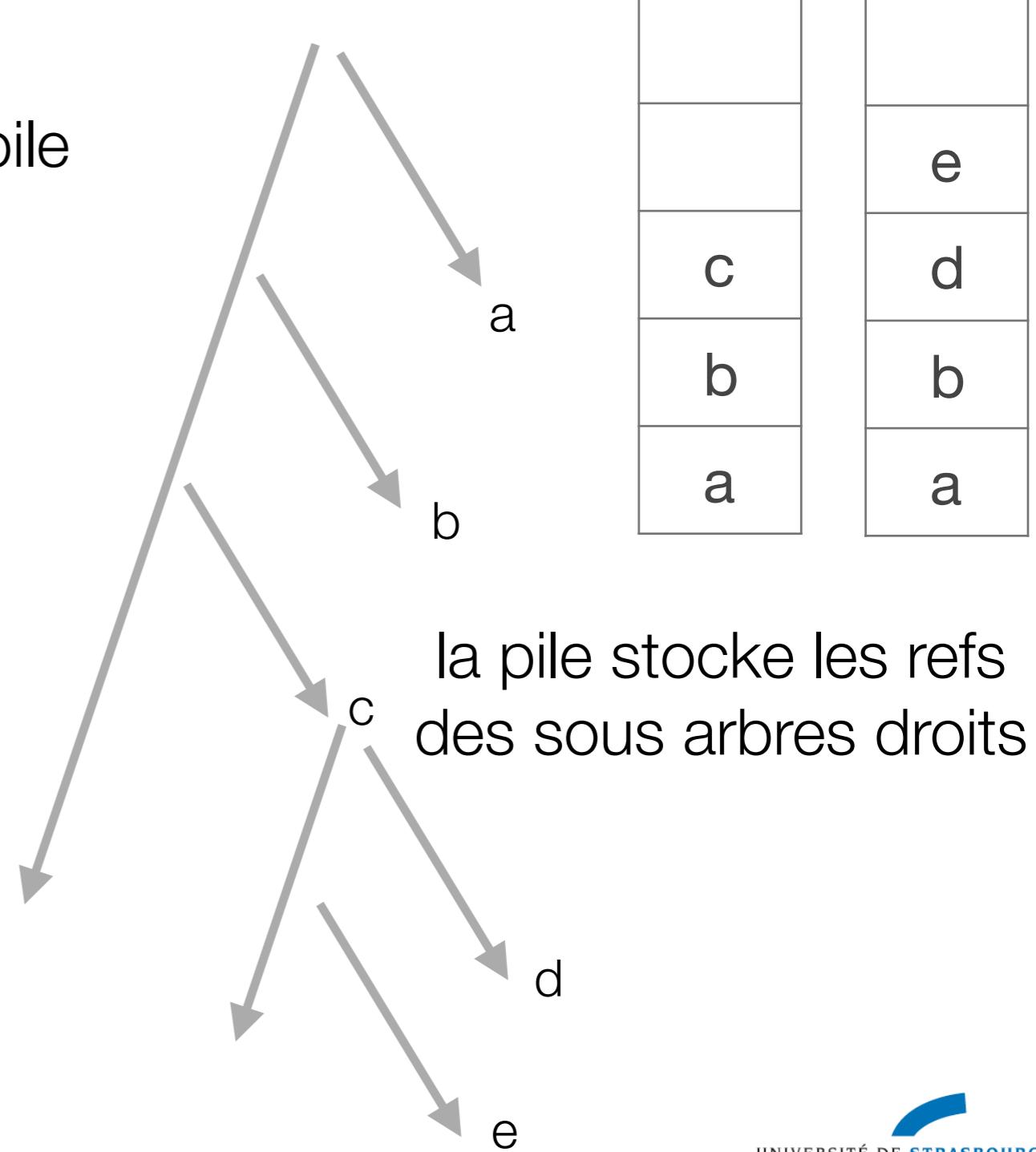
alors $\text{test} = \text{empiler}(p', \text{ad}(a'))$

sinon $\text{test} = p'$

Visualisation du parcours

prefix :
appelle desg sur le sommet de la pile
tant qu'elle n'est pas vide

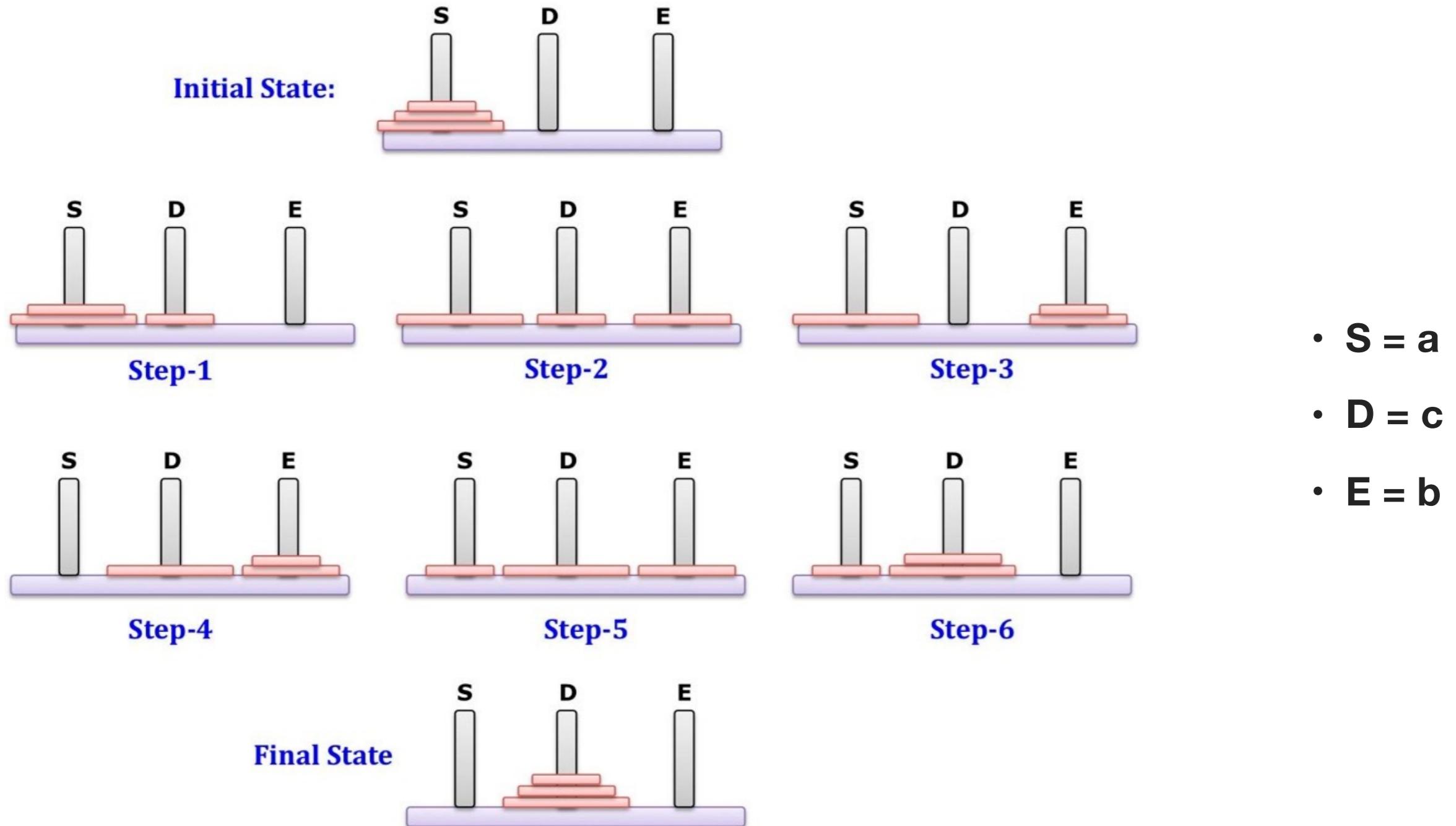
desg :
empile les refs des sous arbres droits
tant qu'on peut descendre



Hanoi & dérécursivation double

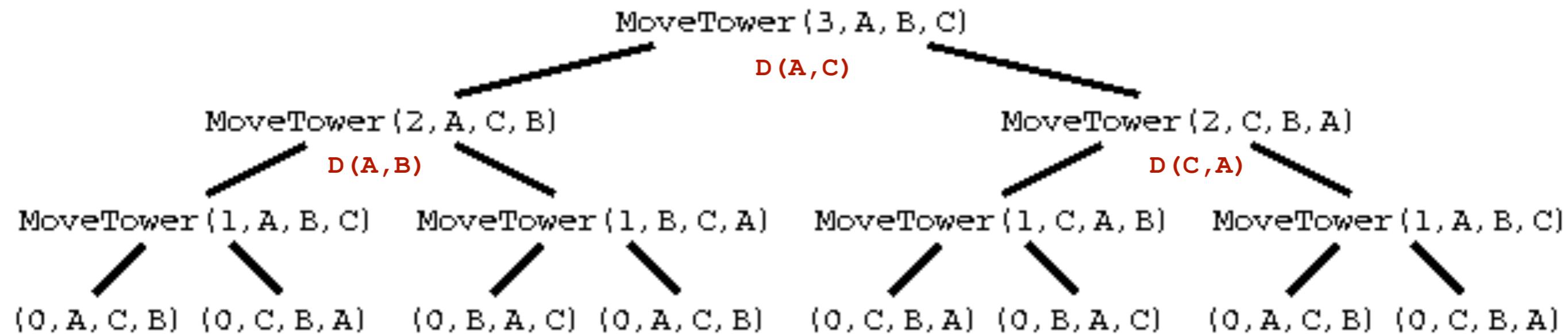
- **Définition récursivité double**
 - un appel de la fonction génère deux appels récursif de cette même fonction
- **Hanoi**
 - Trois tours a, b, c et n disques initialement sur a à déplacer vers c
 - les n disques sont rangés par ordre décroissant de diamètre
 - un disque de diamètre x ne peut pas être déplacé sur un disque de diamètre $y > x$
 - déplacement disque par disque
- **Spécification récursive**
 - `H : Int^4 -> Void`
 - `H(n, a, b, c) = si n > 0 H (n-1, a, c, b) D(a, c) H(n-1, b, a, c)`
 - sinon rien

Illustration des 7 étapes ($n=3$)



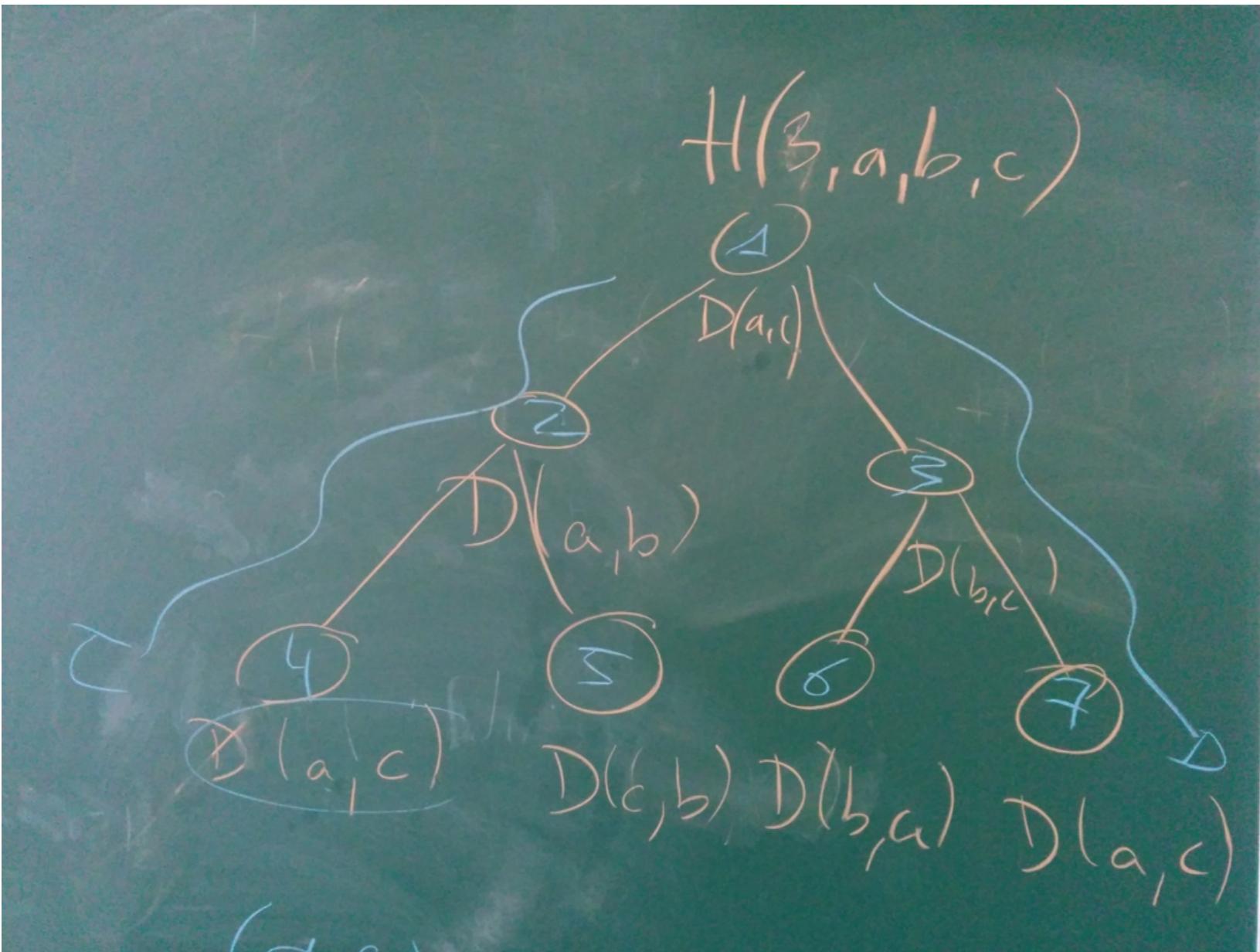
Supprimer la récursivité double !

- Arbres des appels



- => à étudier en détail (quel type d'arbre ? -> lister les propriétés !)
- => **exercice : spécif et implémentation en profitant des propriétés spécifiques**
- indice : pas besoin d'une pile d'arbres, un entier (ou pile de booléen) suffit :)

Hanoi n=3



0	gauche
1	droite
0	gauche

*2 -> à gauche

*2+1 -> à droite

/2 -> remonter, le reste indiquant si
on vient de droite ou de gauche

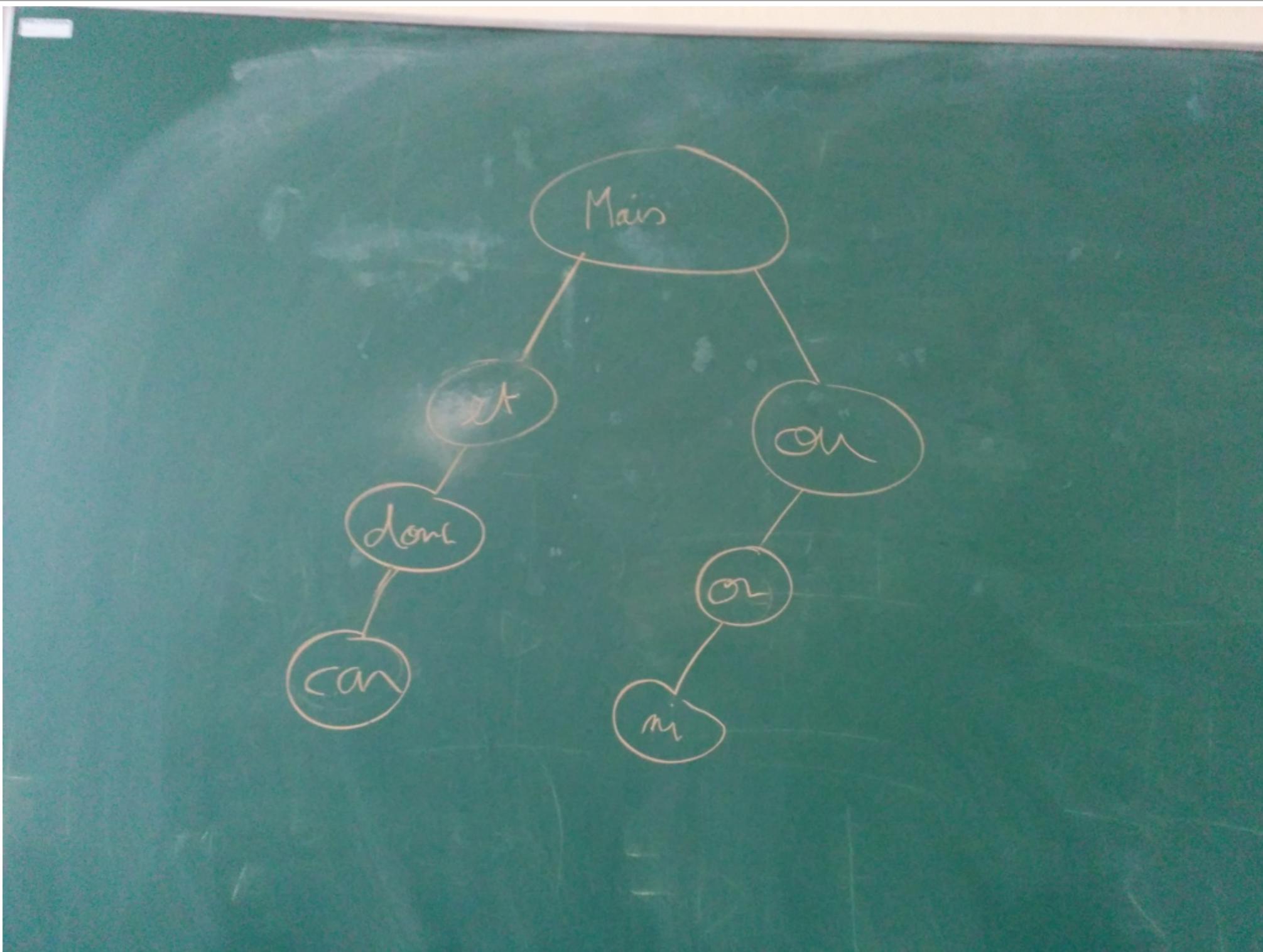
Exercices : on revient sur quoi ?

- **Spécifications de base**
 - appartenance, dimensions, extrémités, cheminement, etc.
- **Arbres généraux, binaires & parcours**
 - axiomes, représentations, insertion, parcours récursif/itératif + implem
- **Hanoi**
 - spécification itérative + implem en C !
- **Preuve des propriétés**
 - hauteur, # chaque type de noeuds, etc.
- **Construction d'une expression arithmétique sous la forme d'un arbre binaire**
 - Version itérative (gauche -> droite)
 - Arbre d'analyse par descente récursive (droite -> gauche)

Arbres binaires de recherche (~triés ou dictionnaire)

- **Hypothèse : S est munie d'un ordre total (\leq)**
- **Définition : Un arbre binaire étiqueté par S est dit trié (ou de recherche) si, en chacun de ses noeuds b, les étiquettes du sous arbre gauche vérifient $\leq \text{rac}(b)$, et celles du sous arbre droit vérifient $> \text{rac}(b)$.**
 - simplification : les étiquettes sont uniques
- **Exemple/exercice : construire un arbre de recherche pour les conjonction de coordination (avec un ordre lexicographique).**
- **Remarques :**
 - parcours infixé : ordre croissant
 - min et max aux extrémités
 - opération de suppression plus sophistiquée que l'insertion -> exercice sur la racine

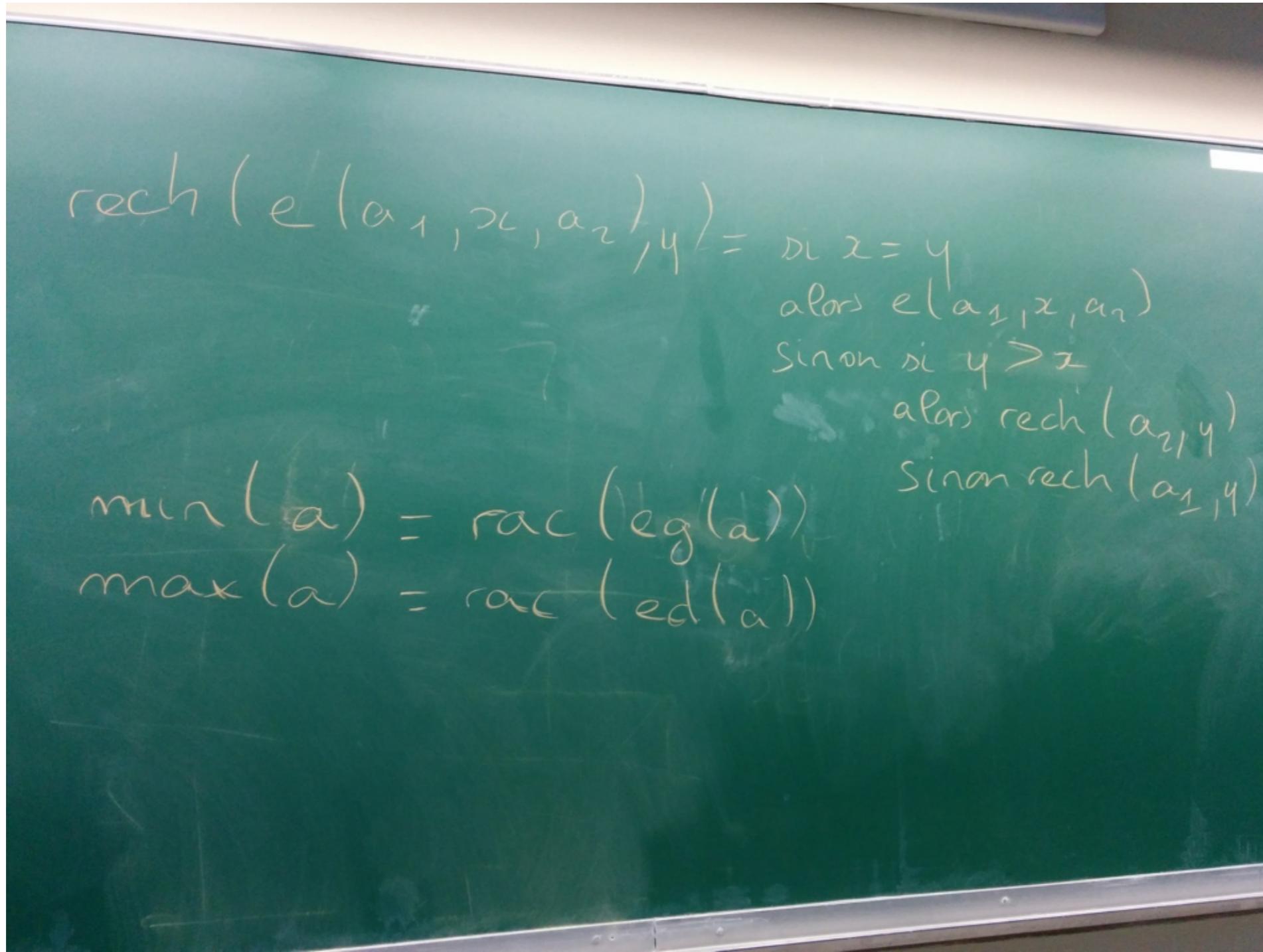
Arbres des conjonctions de coordination (insf)



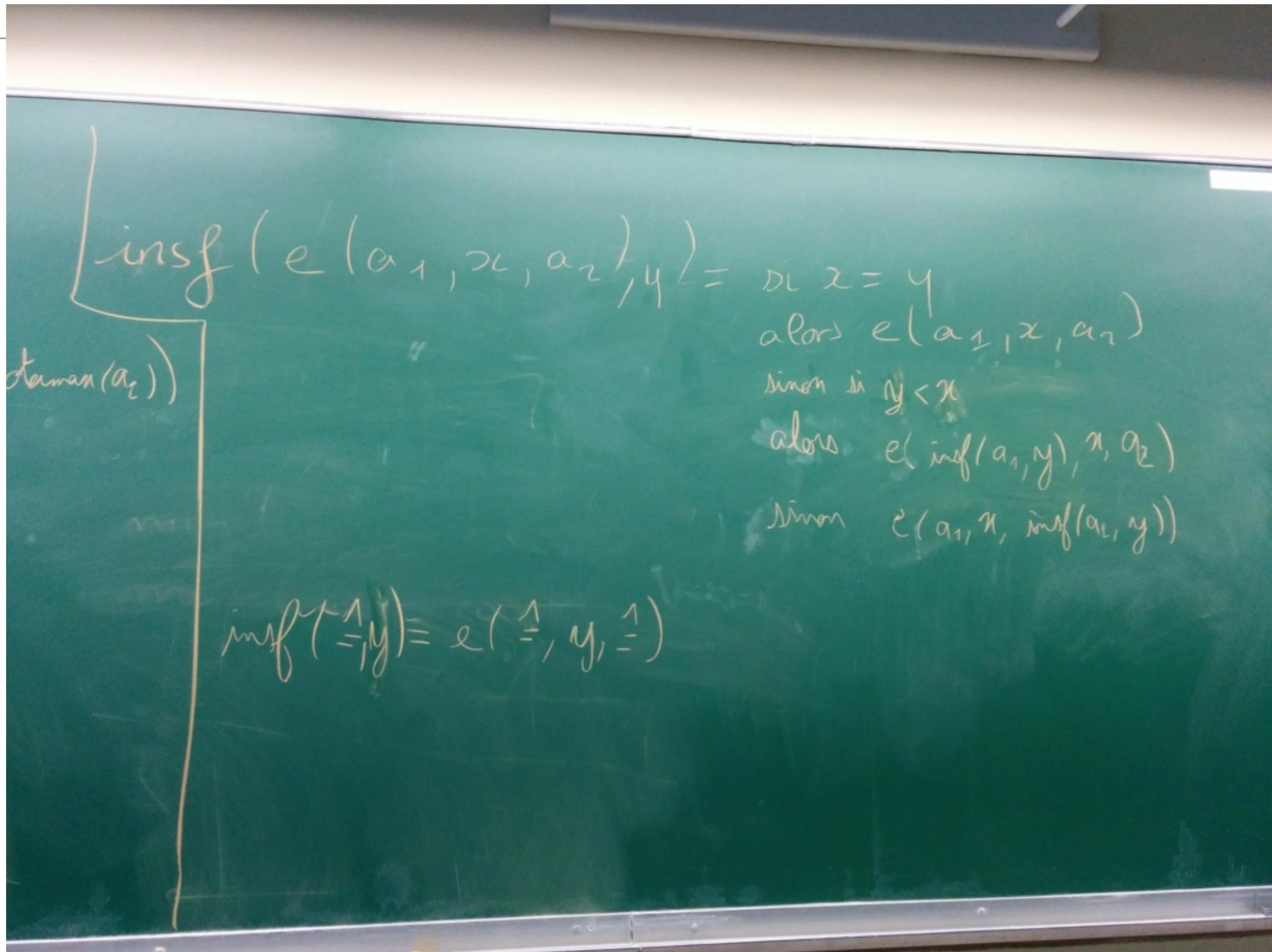
Spécification

- **spec ARBINT étend ARBIN1 (ORDT)**
- **sorte Arbint**
- **opérations**
 - //comme avant pour les opérations de base +
 - **min, max : Arbint -> S // max = rac(ed())**
 - **rech : Arbint S -> Arbint /*sous arbre pour une étiquette donnée */**
 - **insf : Arbint S -> Arbint /* insertion à une feuille */**
 - **insr : Arbint S -> Arbint /* insertion à la racine */
 - **coup : Arbint -> Arbint Arbint /* auxiliaire de coupure en 2 */****
 - **sup : Arbint S -> Arbint /* suppression d'une étiquette */
 - **otermax : Arbint -> Arbint /* auxiliare pour oter le max */****
- **axiomes faciles en exercice (rech et insf) !**

Recherche, min et max

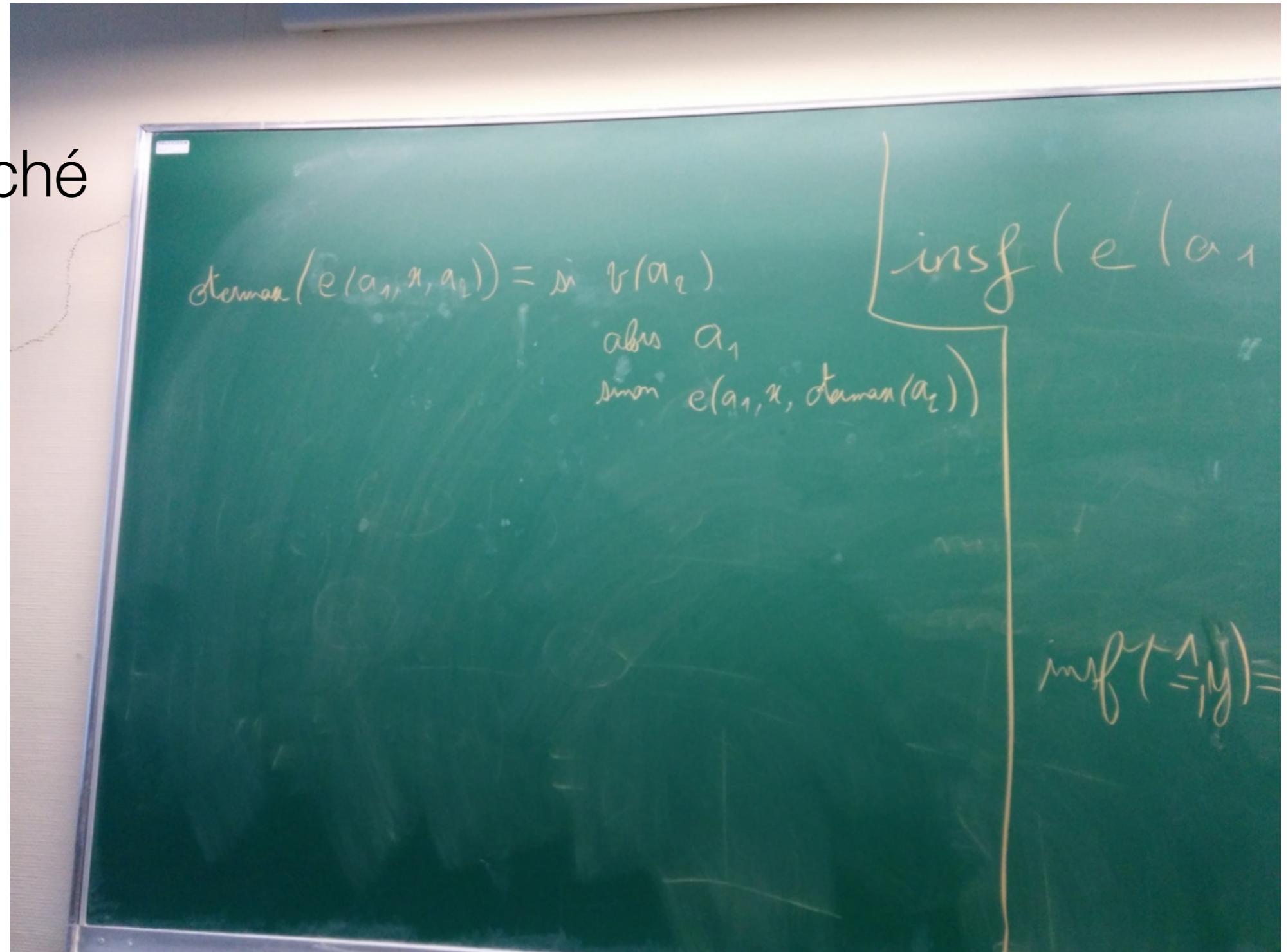


Insertion feuille

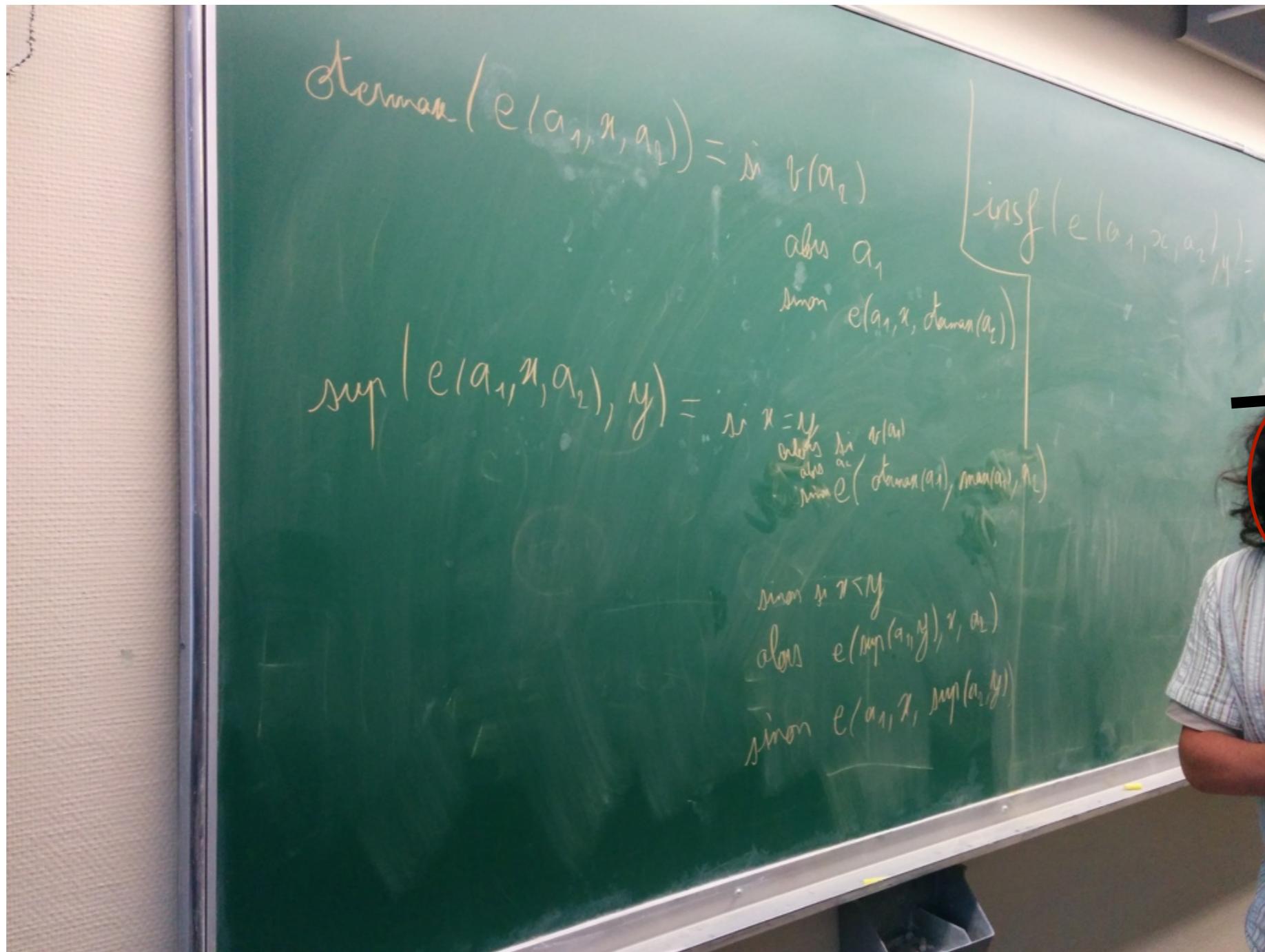


Otermax (extrémité droite de l'arbre gauche !...)

...mais cherché
dans l'arbre
droit...



Insertion feuille et suppression



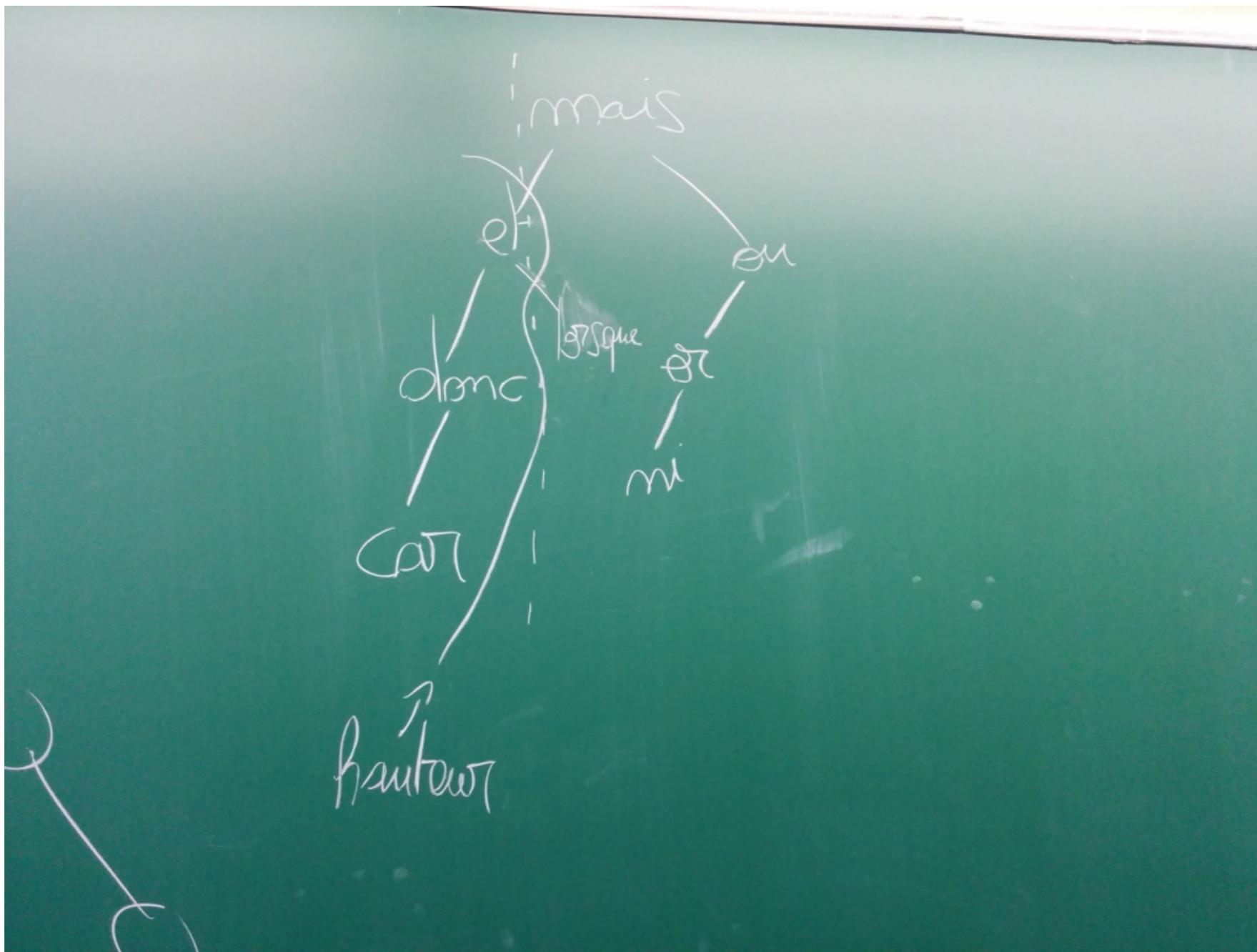
Axiomes plus «difficiles»

- **Suppression**
- $\text{sup}(\bar{\wedge}, x) = \bar{\wedge}$
- $\text{sup}(e(a_1, r, a_2), x) =$
 - si $r == x$ alors
 - si $v(a_1)$ alors a_2
 - sinon si $v(a_2)$ alors a_1
 - sinon $e(\text{otermax}(a_1), \text{max}(a_1), a_2)$
 - sinon si $x < r$ alors $e(\text{sup}(a_1, x), r, a_2)$
 - sinon $e(a_1, r, \text{sup}(a_2, x))$

Axiomes plus «difficiles»

- **Insertion à la racine (avec coupure)**
- **coup** (\emptyset , x) = (\emptyset , \emptyset)
- **coup** ($e(a_1, r, a_2)$, x) =
 - si $x == r$ alors (a_1, a_2)
 - sinon si $x < r$ alors ($g, e(d, r, a_2)$)
 - avec (g, d) = **coup** (a_1, x)
 - sinon ($e(a_1, r, g), d$)
 - avec (g, d) = **coup** (a_2, x)
- **exercice (facile) :**
 - dessiner ce que fait coupure lors de l'insertion **insf** d'un mot (e.g., «lors»)
 - puis donner **insr** en utilisant **coup**

Coupure : illustration (sur le mot hauteur)



Insertion à la racine avec coupure

Axiomes plus «difficiles»

- **Insertion à la racine (sans coupure)**

- **insr(\top , x) = e(\top , x, \top)**
- **insr(e(a1, r, a2), x) =**
 - si $x == r$ alors $e(a1, r, a2)$
 - sinon si $x < r$ alors $e(ag(b), x, e(ad(b), r, a2))$
 - avec $b = \text{insr}(a1, x)$
 - sinon $e(e(a1, r, ag(b)), x, ad(b))$
 - avec $b = \text{insr}(a2, x)$

Fusion d'opérations

- Pour gagner en efficacité max et otermax peuvent être fusionnés
 - -> exercice : spécifier **max_otermax**
- Rechercher et insérer à une feuille si pas trouvé
 - -> exercice : spécifier **rechinsf**
- Etc
 - But : éviter de doubler les opérations

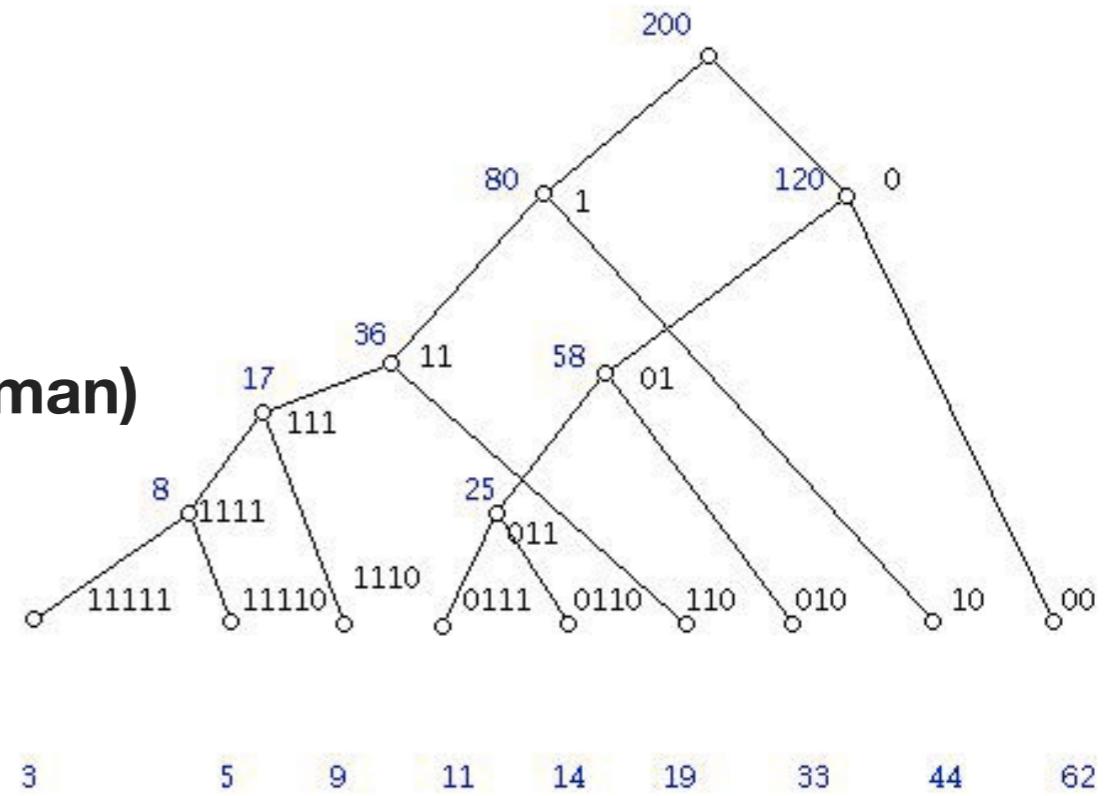
Dérécursivation effective

- **Trois étapes :**
 - préparer une fonction auxiliaire qui ne gère pas le cas vide total mais le cas général
 - avec sous-exceptions si besoin
 - trouver sa version itérative (pas besoin de pile - juste une descente, pas un parcours)
 - puis implémenter en C !
- **Exercice : le faire avec `insf1`**
 - De la spéci à l'implémentation...

Insertion feuille en C

Complexités

- **Arbre «dégénérés» : recherche en $O(n)$**
- **Arbre équilibré (parfait par exemple) : recherche en $O(\log(n))$**
- **En moyenne (arbres filiformes rares) : $O(\log(n))$**
 - ~ équivalent aux arbres équilibrés :)
- **Arbres optimaux : vers un codage efficace (Huffman)**
 - Soit une probabilité d'«être cherché»
 - -> remonter les étiquettes populaires
 - => arbre qui minimise le cout de recherche !



Arbres binaires équilibrés : (Self-)Balanced BST

- **Définitions**

- En tout noeud, les hauteurs des deux sous arbres diffèrent de 1 max
 - (déséquilibré sinon)

- **Indicateur de déséquilibre**

- $\text{déséq} : \text{Arbin} \rightarrow \text{Int}$
- $\text{déséq}(e(a1, x, a2)) = h(a1) - h(a2)$
- $\rightarrow \text{déséq} = -1, 0 \text{ ou } 1$

- **Propositions**

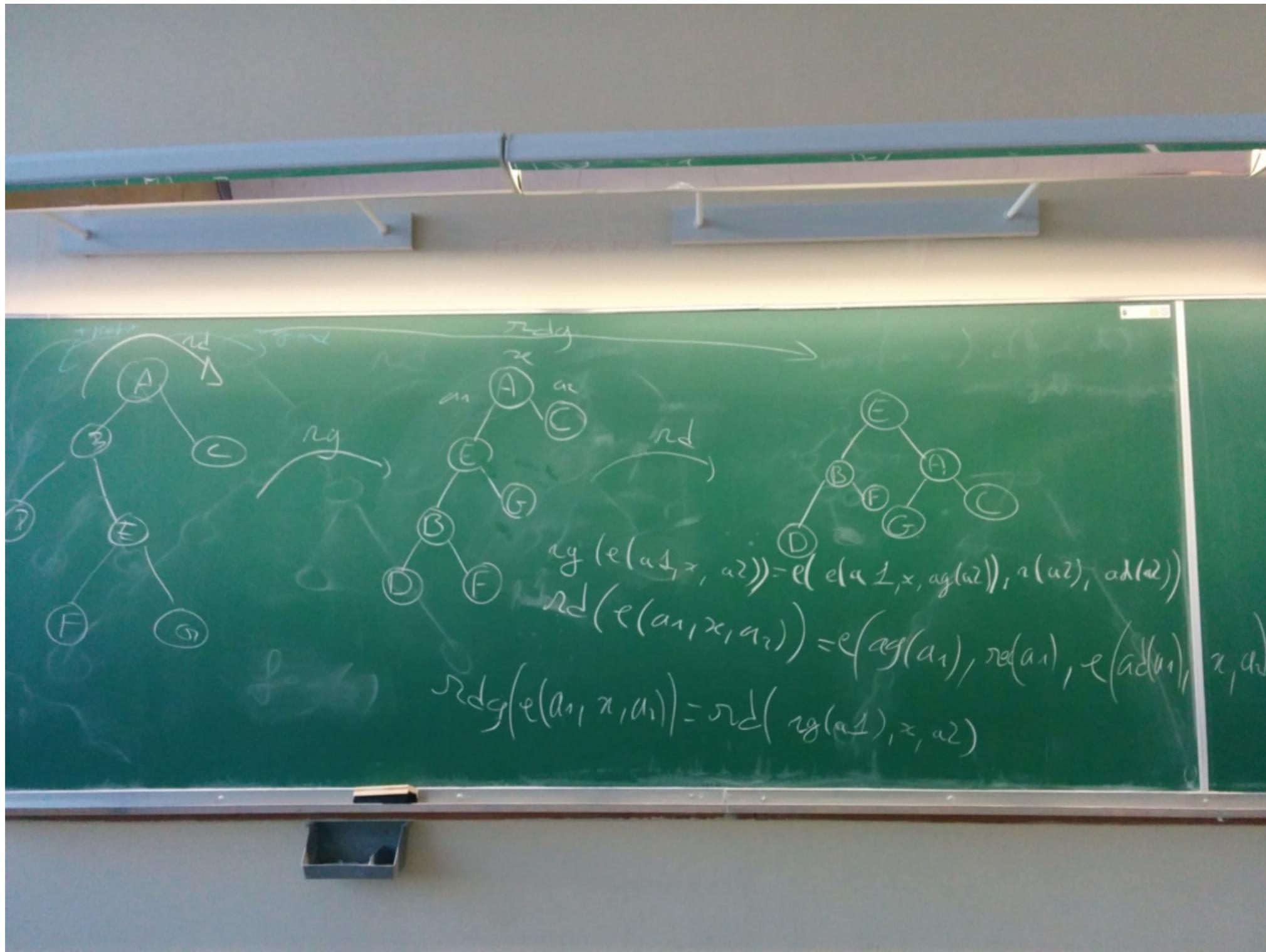
- La hauteur p d'un B-BST vérifie : $\log(n+1) \leq p < 1,4405 \log(n+1) + 1$

- **Exemples : dessiner les trois cas de déséq**

Equilibrage par rotation

- **Simple à gauche, à droite : rg , rd**
 - exercice : à dessiner (rd : gauche vers droite) et à spécifier
- **Double gauche-droite, droite-gauche : rdg , rgd**
 - exercice : à dessiner (rdg := rg du ss arbre gauche et rd de tout l'arbre) et à spécifier
- **Ces rotations transforment un BST en un (B-)BST**

Rotations : visualisation et specs



Rotations : spécifications

- **Pré-conditions : arbre non vide**
- $\text{rd}(\text{e}(a1, x, a2)) = \text{e}(\text{ag}(a1), \text{r}(a1), \text{e}(\text{ad}(a1), x, a2))$
- $\text{rgd}(\text{e}(a1, x, a2)) = \text{rd}(\text{e}(\text{rg}(a1), x, a2))$
- => **Exercices :**
 - rotation gauche ?
 - rotation droite-gauche ?

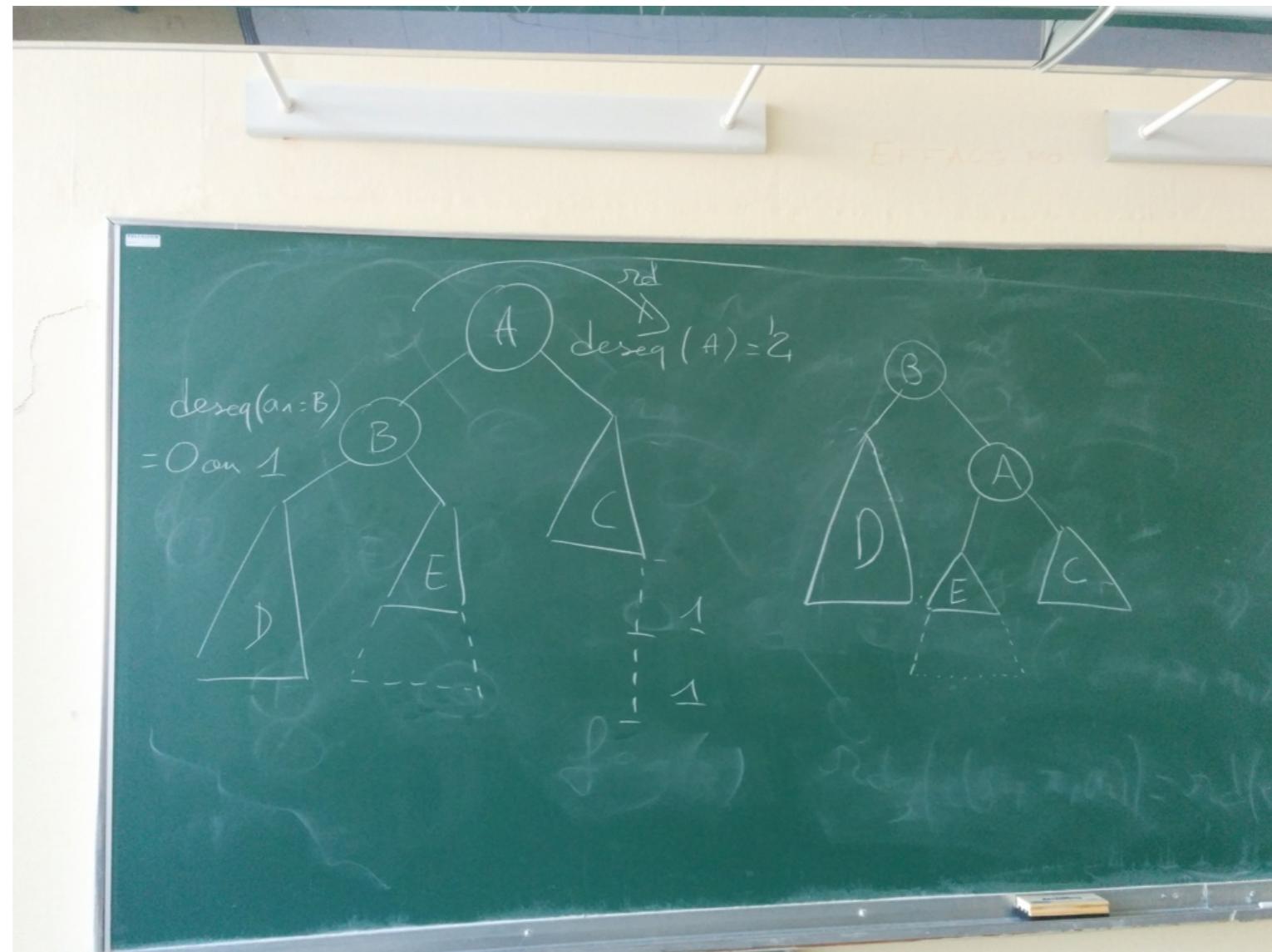
L'exemple AVL

- **Adelson-Velskii et Landis (les deux noms des inventeurs)**
 - arbre binaire de recherche équilibré
- **Ré-équilibrage périodique ou à chaque mise à jour**
 - déséquilibre de 2 max si systématique à chaque mise à jour
- **Soit $a = e(a_1, x, a_2)$ si $\text{déséq}(a) = 2$ avec a_1 et a_2 équilibrés, deux cas :**
 - $\text{déséq}(a_1) = 0$ ou 1 : rotation droite en $a' = \text{rd}(a)$
 - $\text{déséq}(a_1) = -1$: double rotation gauche-droite en $a' = \text{rdg}(a)$
- **Proposition : l'arbre a' est un AVL et si, $\text{deseq}(a_1) = +/- 1$, alors $h(a') = h(a) - 1$, sinon $h(a) = h(a')$**

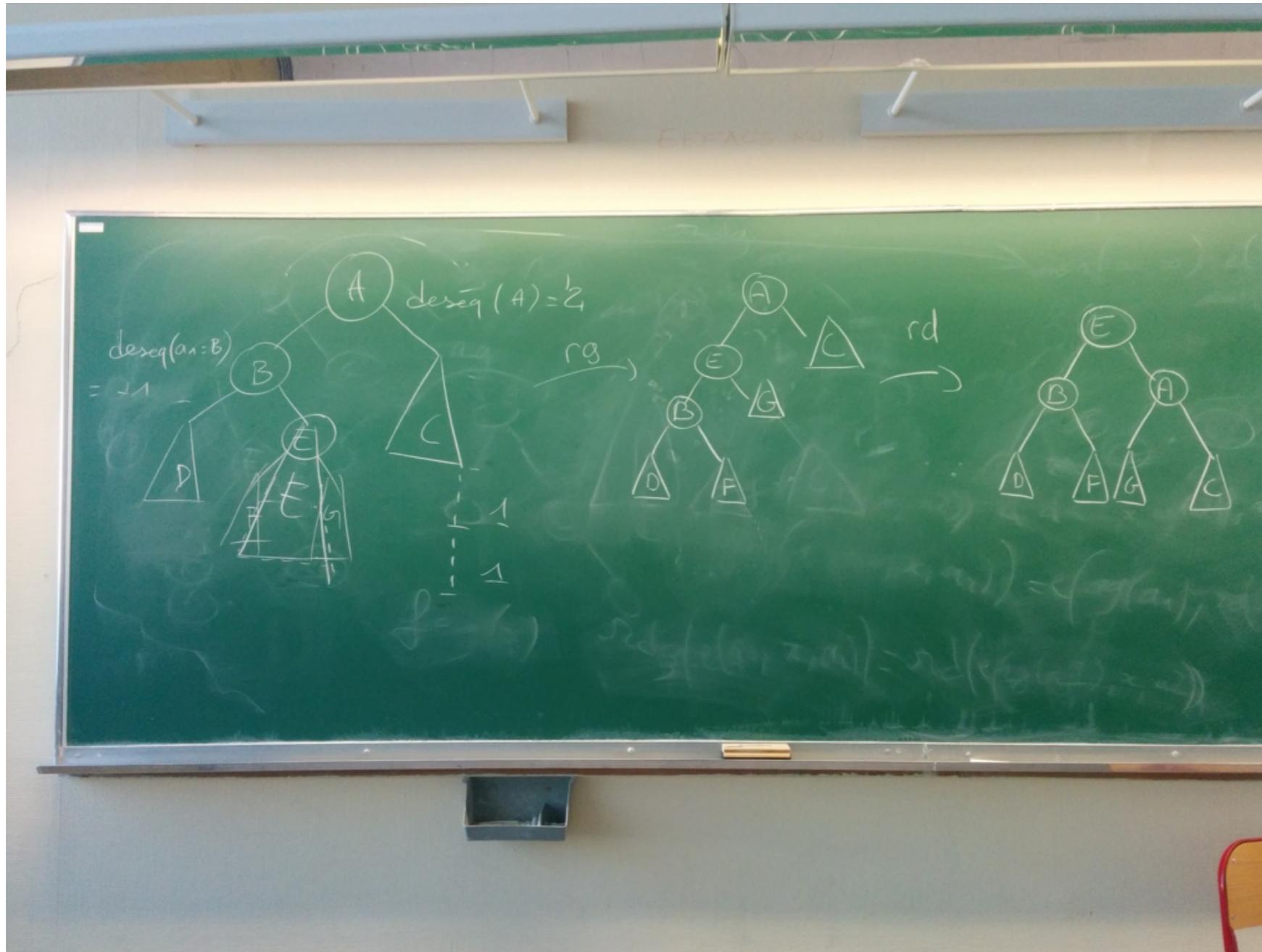
Rééquilibrage : visualisation

- **Exercices (avec sous pyramides)**

- rd ($a_1 : 0$ ou 1)



rgd (a1 : -1)



Rééquilibrage : spécification

- **rééq** : Arbint -> Arbint
- **rééq(a) =**
 - si déséq(a) = +2 ET $0 \leq \text{déséq}(\text{ag}(a))$ alors rd(a)
 - sinon si = -2 ET $\text{déséq}(\text{ad}(a)) \leq 0$ alors rg(a)
 - sinon si = +2 ET $\text{déséq}(\text{ag}(a)) = -1$ alors rgd(a)
 - sinon si = -2 ET $\text{déséq}(\text{ad}(a)) = +1$ alors rdg(a)
 - sinon a

Insertion et suppression :

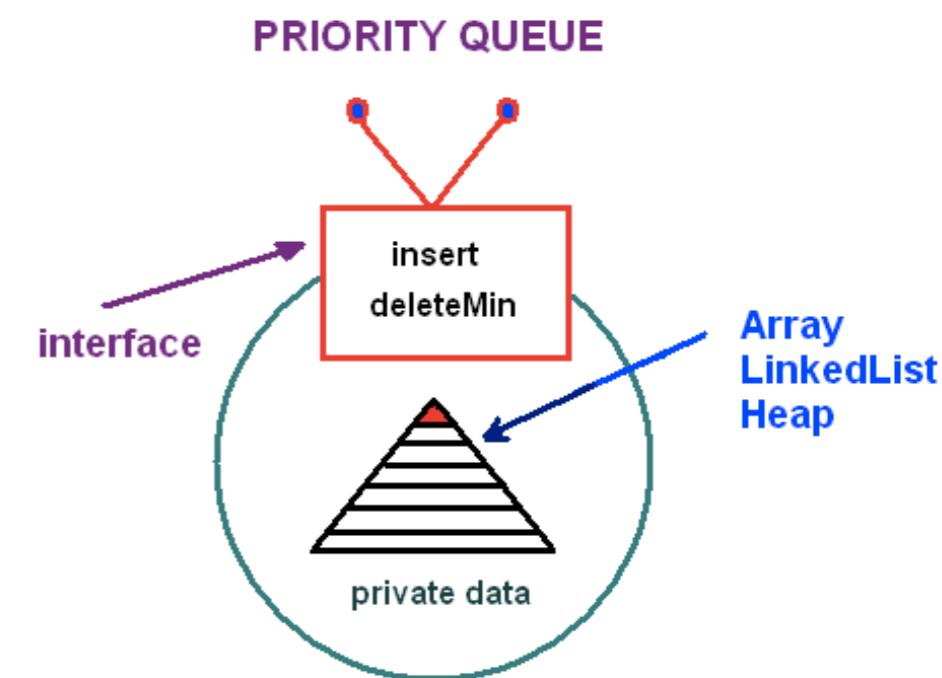
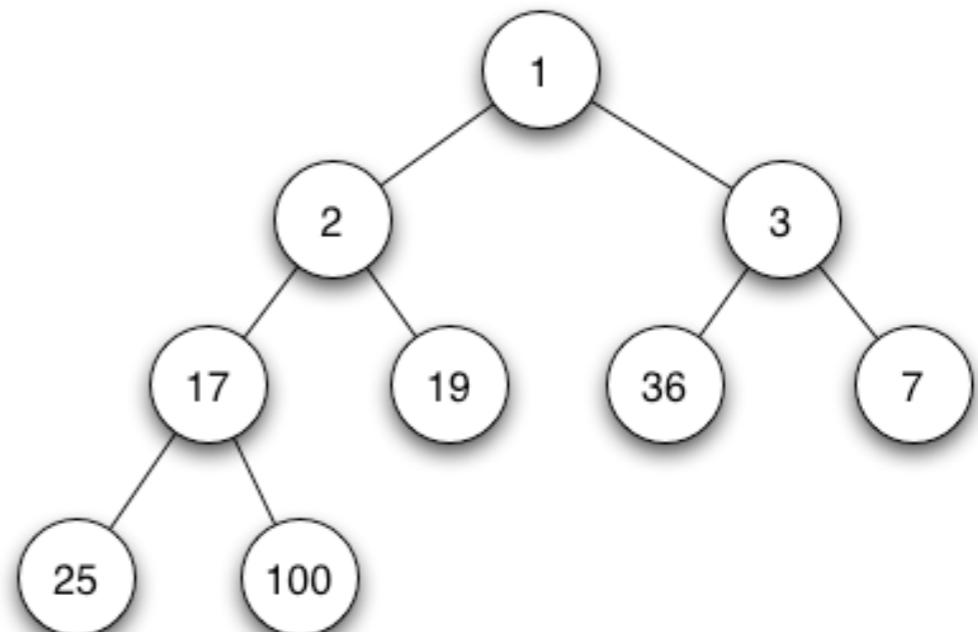
- L'insertion à une feuille nécessite au pire une seule rotation (simple ou double)
- **insval** : Arbint S -> Arbint
- **insval(e(a1,x,a2), y) =**
 - si $x = y$ alors $e(a1,x,a2)$
 - sinon si $y < x$ alors $\text{rééq}(e(\text{insval}(a1,y),x,a2))$
 - sinon $\text{rééq}(e(a1,x,\text{insval}(a2,y)))$

Suppression BST - AVL

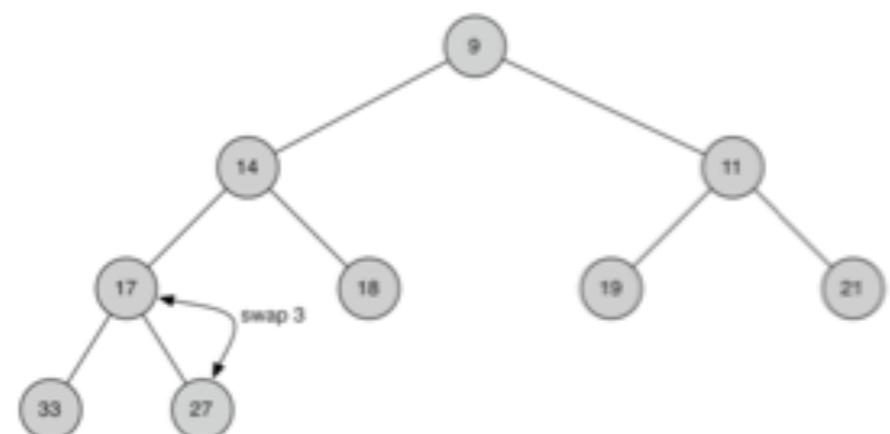
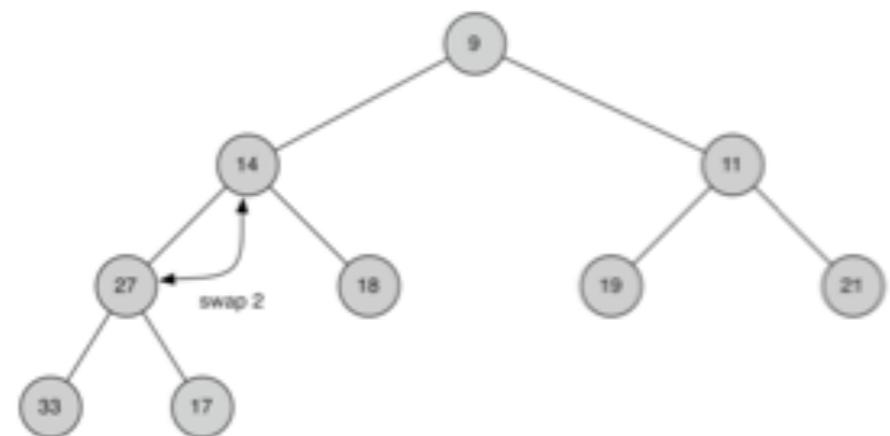
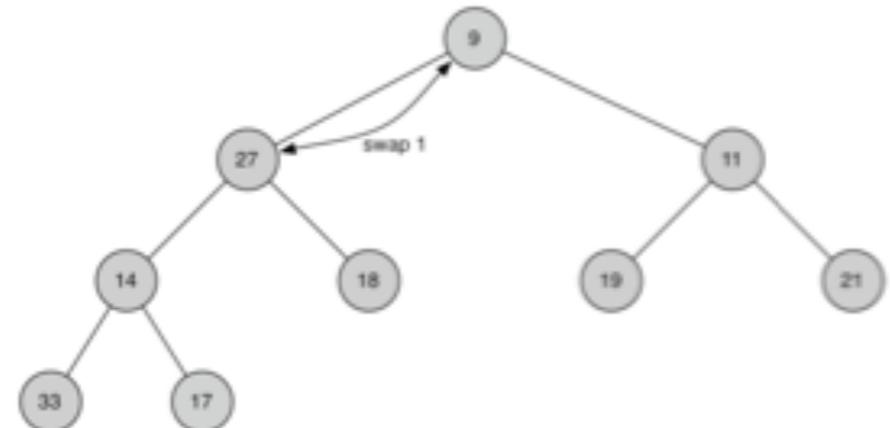
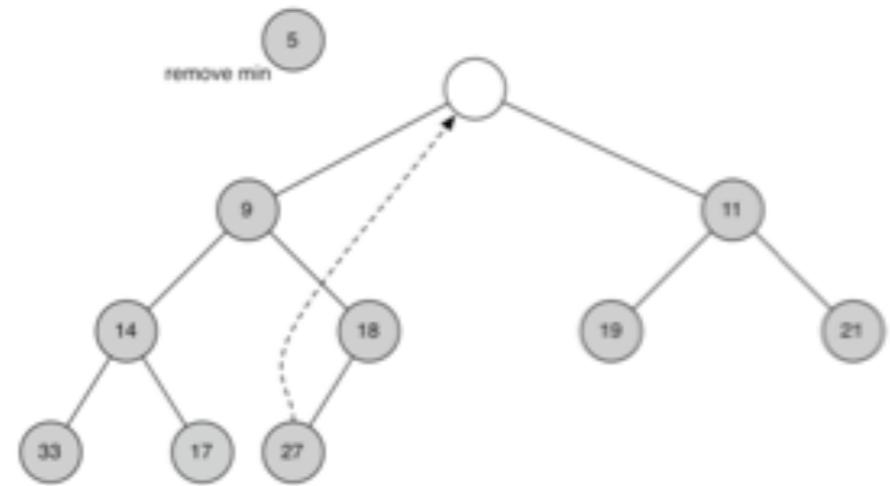
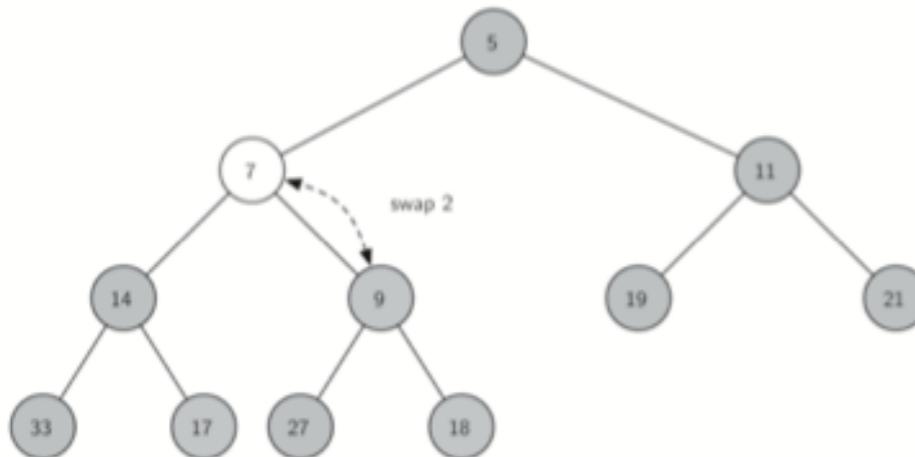
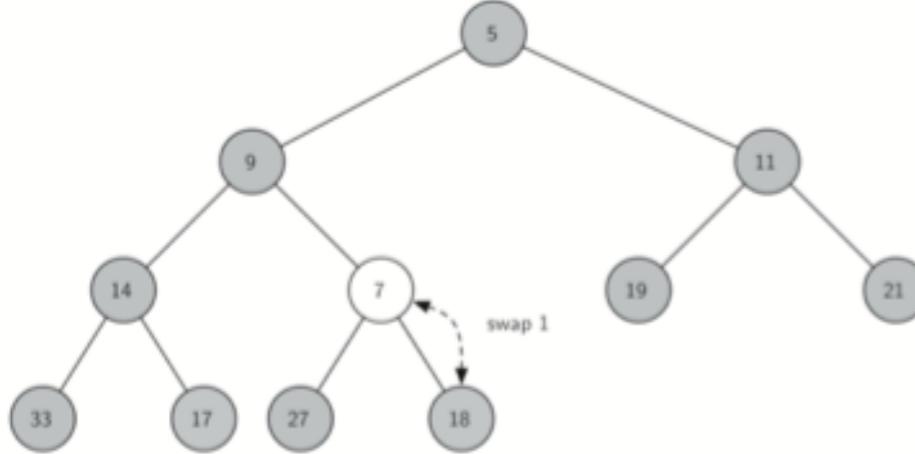
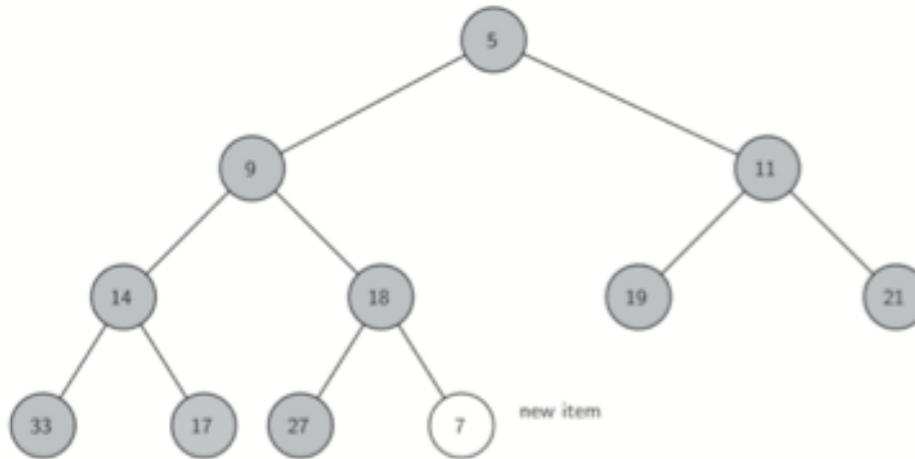
- **supval : Arbint S -> Arbint**
- **supval(e(a1,x,a2),y) =**
- **si y = x alors si v(a2) alors a1**
 - sinon si v(a1) alors a2
 - sinon rééq(e(oterm(a1),max(a1),a2))
- **sinon si y < x alors rééq(e(supval(a1,y),x,a2))**
 - sinon rééq(e(a1,x,supval(a2,y)))

Binary Heap

- **Rappels :**
 - Arbre binaire parfait
- **Opérations : percolate up-down**
 - insérer un noeud (tas facile : montée - up)
 - retirer le min/max (descente - down)
- => Spécifier et implémenter

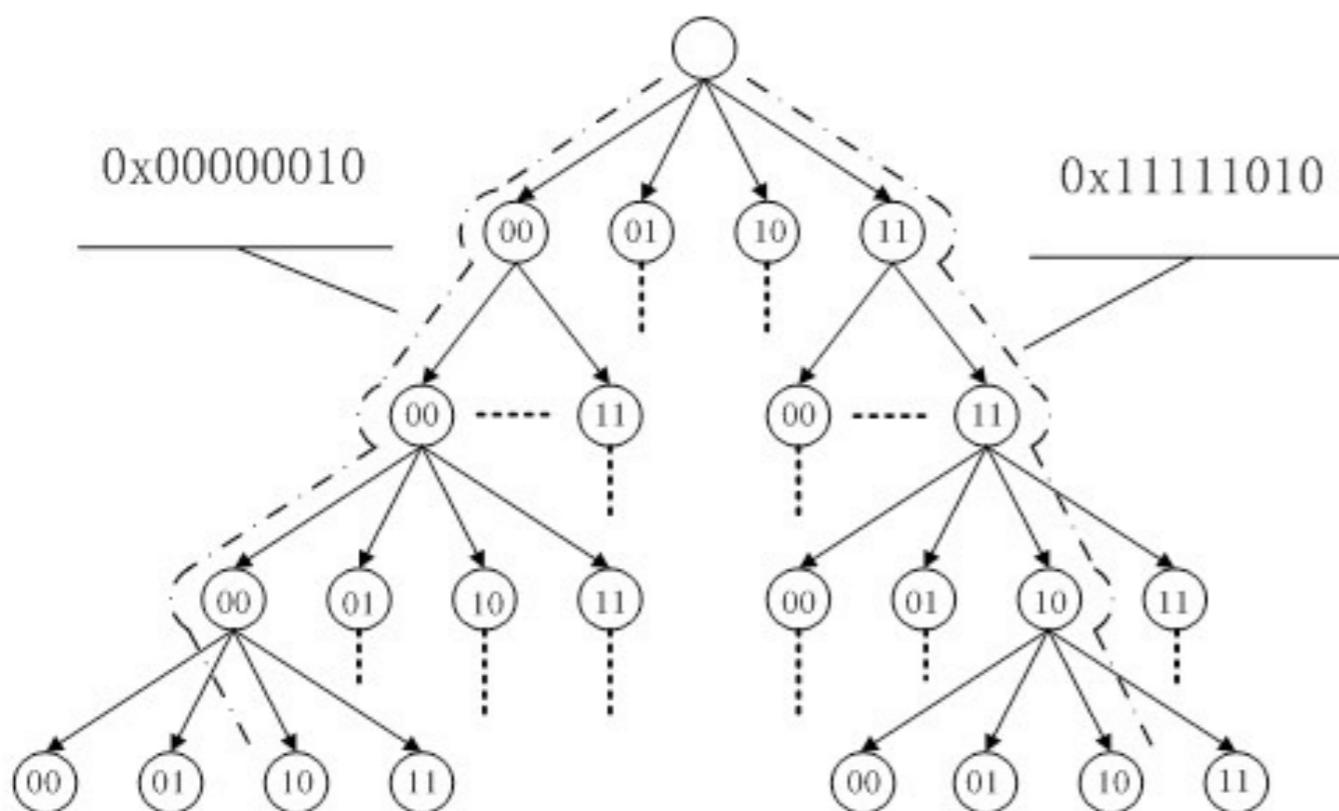


B-HEAP

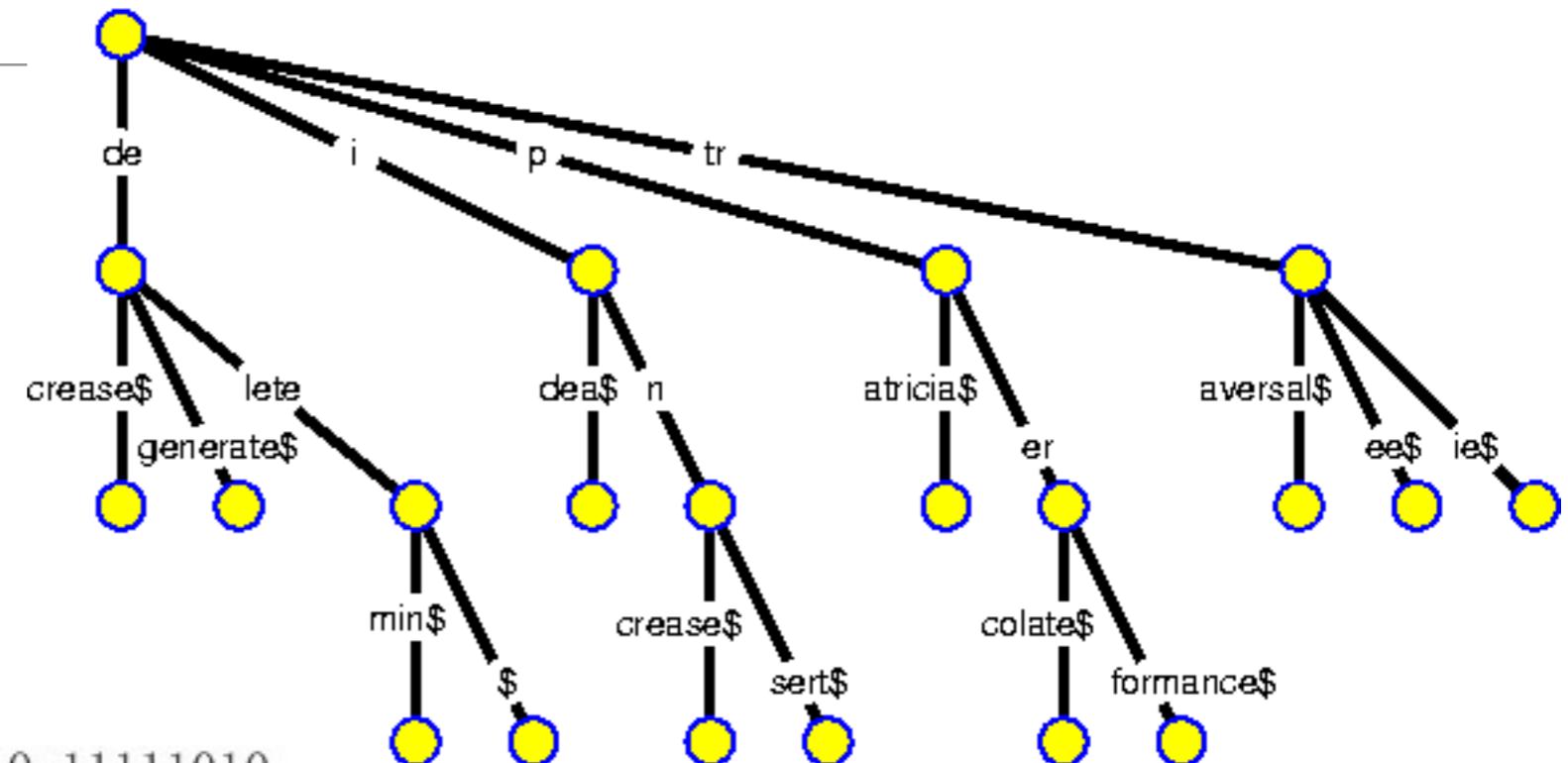


Patricia Tree

- Prefix Tree, Radix Tree



PATRICIA TRIE WITH 13 ENTRIES

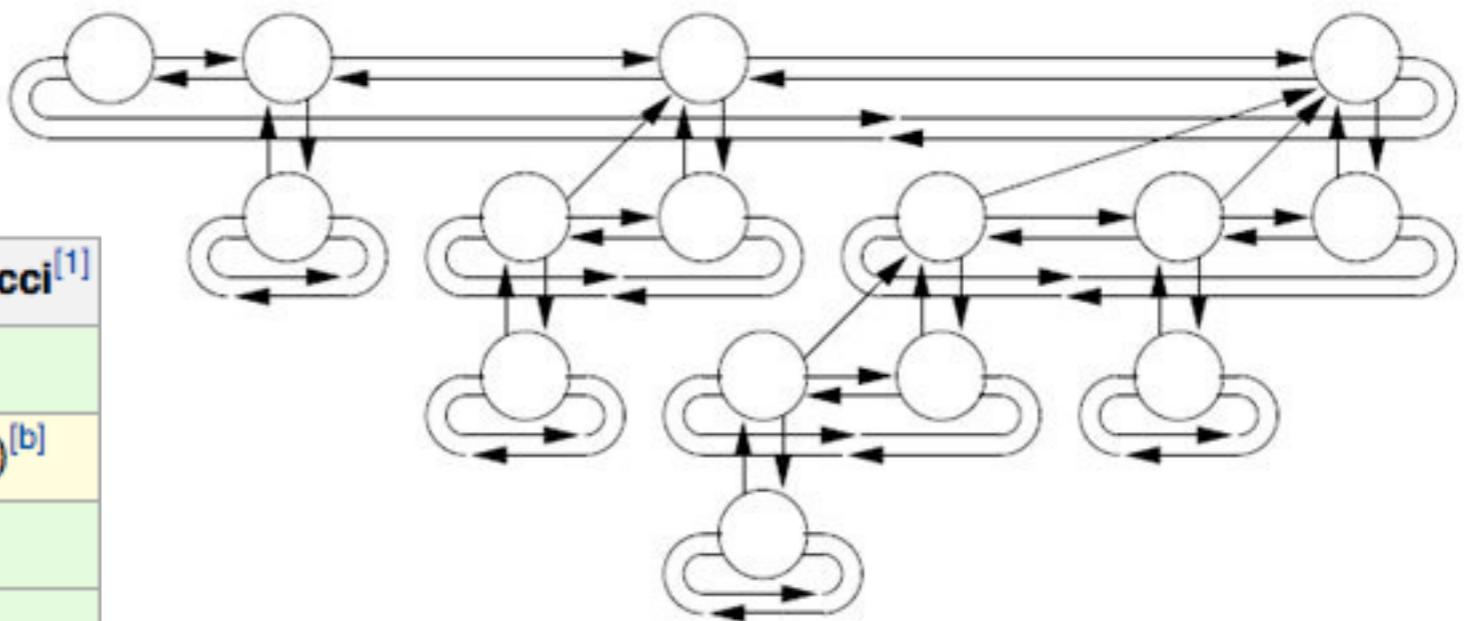


- Exercices :
- Spécification
- Implémentation

Tas de Fibonacci

- Variante d'un tas binomial relâché (notion de complexité amortie)
- Objet plus théorique que pratique...
- Fibonnaci..? rapport avec la suite ?
 - un père de degré k (k fils) doit avoir une descendance totale de F_{k+2}

Operation	Binary ^[1]	Binomial ^[1]	Fibonacci ^[1]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$ ^[b]
insert	$\Theta(\log n)$	$\Theta(1)$ ^[b]	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ ^[b]
merge	$\Theta(m \log(n+m))$ ^[d]	$O(\log n)$ ^[e]	$\Theta(1)$



Corrections

- **Hanoi**
- **Postfixe**
- **Parcours généralisé aux arbres non binaires**
- **Preuves**
- **Spécifs / implem : BST, AVL, BH, etc.**
- ...