

# Algorithmie Avancée

## Mise en Contexte / Mise en Oeuvre

Année 2020-2021 par Prof. Nicolas Loménie  
Sur la base du cours de Prof. Etienne Birmelé (2016-2020)

# Structure d'arbre binaire

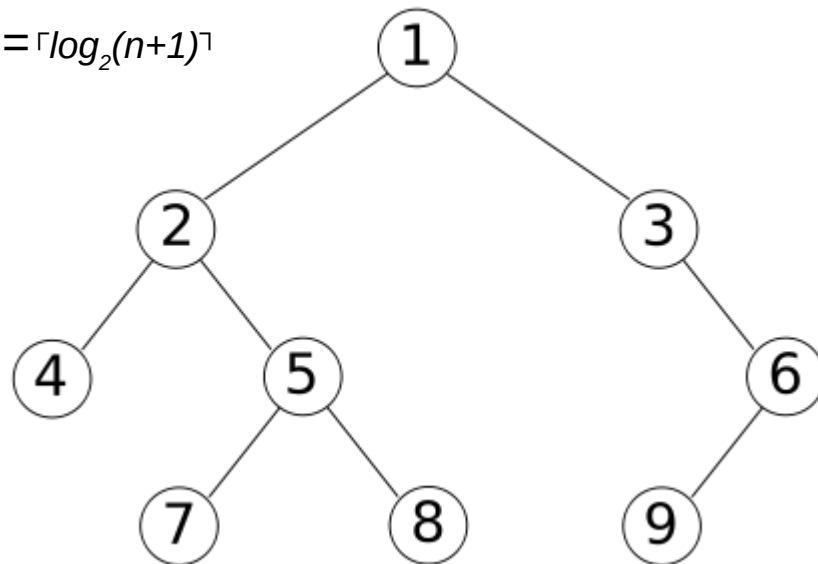
On désigne par  $h$  la hauteur d'un arbre.

Si  $h$  démarre à 0, le nombre de niveaux est égal à  $h+1$ .

Un arbre binaire à  $n$  nœuds et de hauteur  $h$  vérifie :  $h \geq \lfloor \log_2 n \rfloor$

En effet, il y a au plus  $2^i$  nœuds à hauteur  $i$  donc  $n \leq 2^{h+1} - 1$   
(Suite géométrique de raison 2)

De plus, pour tout  $n \geq 1$ ,  $1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil$



Somme des termes d'une suite géométrique  
de raison 2 à l'ordre  $n$  :  $2^{n-1} - 1$

## THÉORÈME

Soit  $q$  un nombre réel différent de 1:

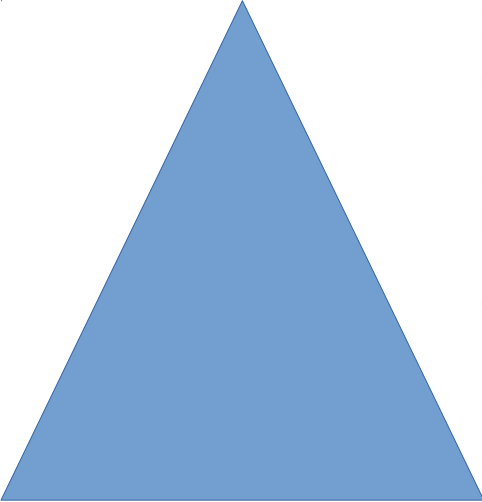
$$1 + q + q^2 + \dots + q^n = \frac{1 - q^{n+1}}{1 - q}$$

# Structure d'arbre binaire

Un arbre binaire de hauteur  $h$  est complet si chaque niveau est entièrement rempli. Cet arbre possède  $2^{h+1}-1$  sommets et  $2^h$  feuilles.

Dans un arbre binaire complet à  $n$  nœuds, le nombre de nœuds internes est égal au nombre de feuilles moins 1, en effet

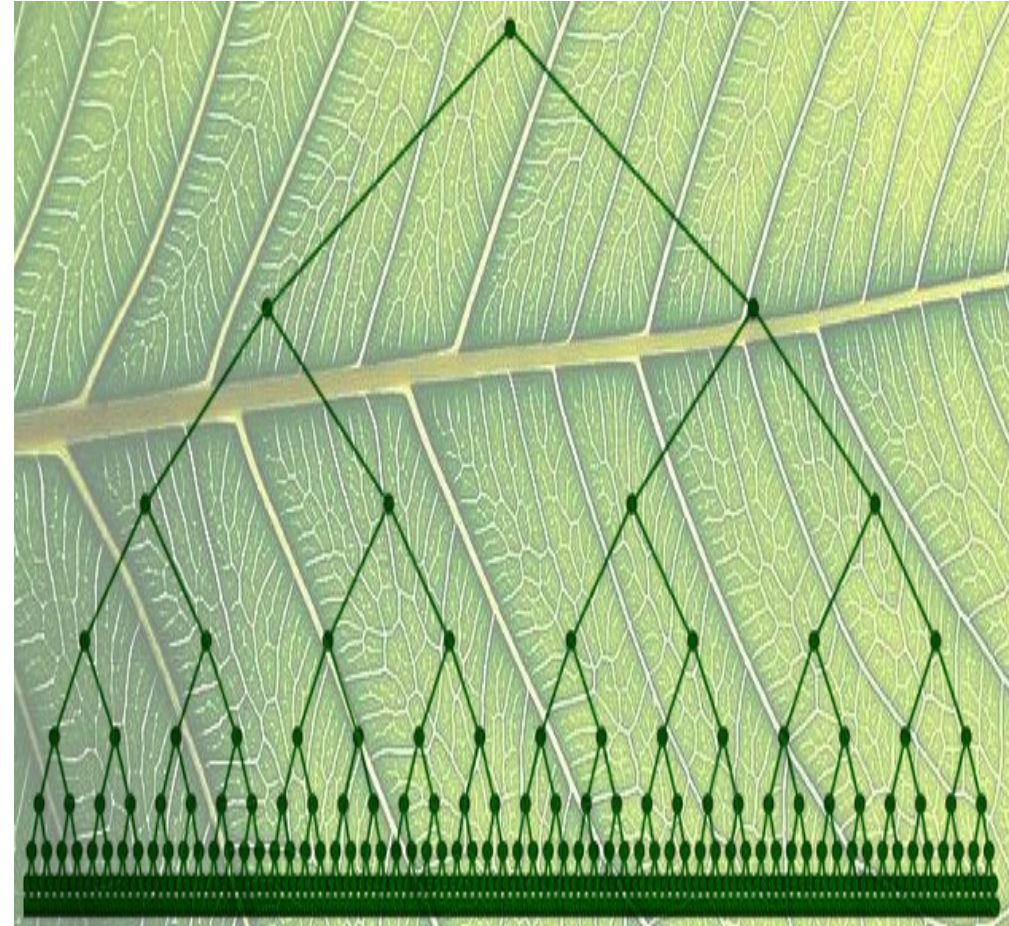
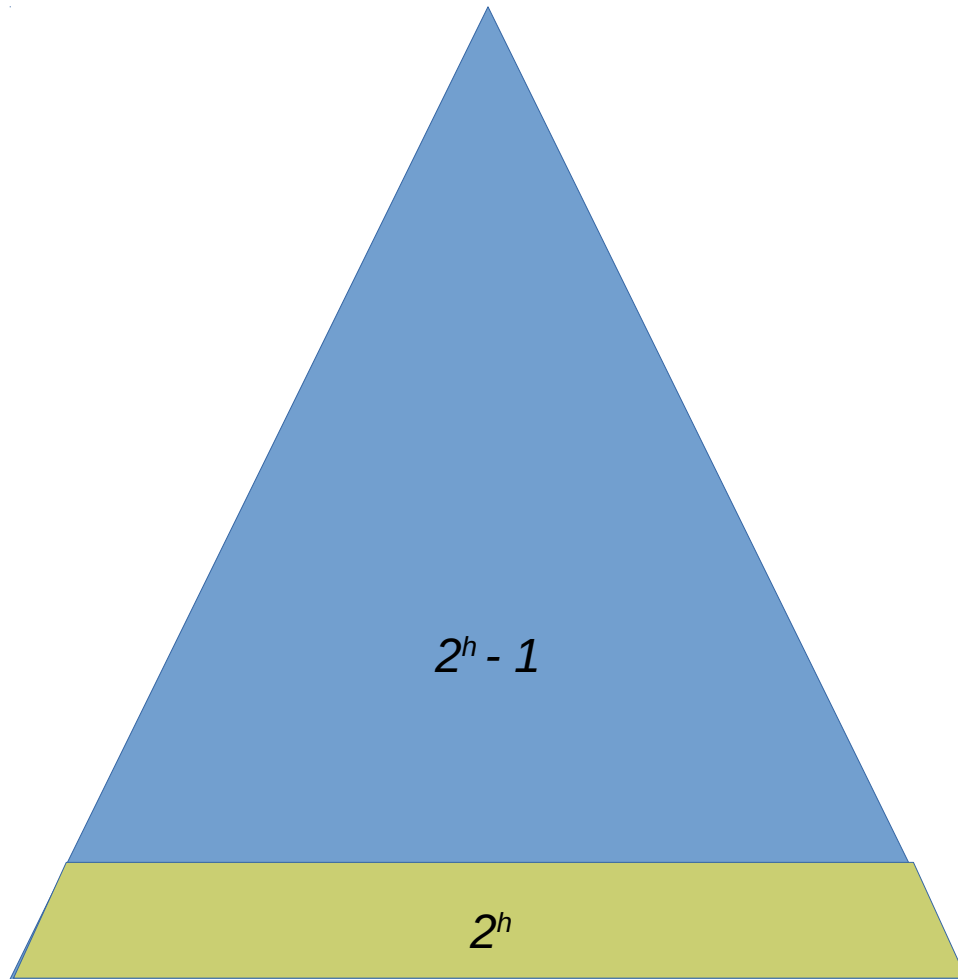
$$nb\_noeud\_internes + 2^h = 2^{h+1}-1 \text{ soit } nb\_noeuds\_internes + 1 = 2^h(2-1) = 2^h$$



**Preuve.** Notons, pour simplifier,  $f(A)$  le nombre de feuilles et  $n(A)$  le nombre de nœuds internes de l'arbre binaire complet  $A$ . Il s'agit de montrer que  $f(A) = n(A) + 1$ .

Le résultat est vrai pour l'arbre binaire de hauteur 0. Considérons un arbre binaire complet  $A = (A_g, r, A_d)$ . Les feuilles de  $A$  sont celles de  $A_g$  et de  $A_d$  et donc  $f(A) = f(A_g) + f(A_d)$ . Les nœuds internes de  $A$  sont ceux de  $A_g$ , ceux de  $A_d$  et la racine, et donc  $n(A) = n(A_g) + n(A_d) + 1$ . Comme  $A_g$  et  $A_d$  sont des arbres complets de hauteur inférieure à celle de  $A$ , la récurrence s'applique et on a  $f(A_g) = n(A_g) + 1$  et  $f(A_d) = n(A_d) + 1$ . On obtient finalement  $f(A) = f(A_g) + f(A_d) = (n(A_g) + 1) + (n(A_d) + 1) = n(A) + 1$ .  $\square$

# Allure d'un arbre binaire complet



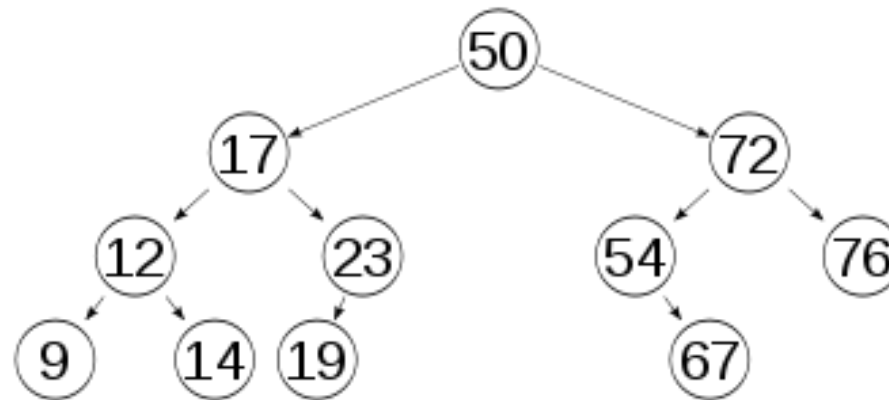
# Allure d'un arbre binaire complet

Au final pour tout arbre binaire

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

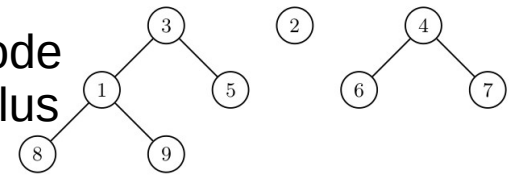
Pour être complet, on doit avoir  $h(A) = \lfloor \log_2 n \rfloor$ .

En conséquence pour optimiser les algorithmes on essaye d'équilibrer ces arbres binaires (voir arbres AVL ou Union-Find) plutôt qu'au pire avoir des arbres filiformes.



# Allure d'un arbre à $n$ nœuds « équilibré »

La hauteur d'un arbre à  $n$  nœuds équilibré par **UNION** (dans la méthode UNION-FIND voire méthode de Kruskal ) ou union pondérée est au plus  $1 + \lfloor \log_2 n \rfloor$



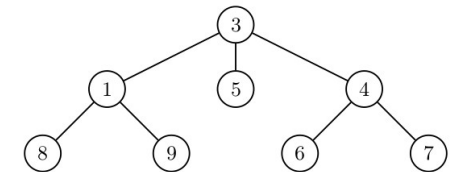
$UNION(1, 7)$

Ici il est important de noter que la taille représente le nombre de sommets.

Pour un arbre ordre et taille sont équivalents.

On branche l'arbre de plus petite taille sur celui de plus grande taille dans UNION-FIND d'où :

sommet	1	2	3	4	5	6	7	8	9
père	3	2	3	3	3	4	4	1	1



**Preuve.** Par récurrence sur  $n$ . Pour  $n = 1$ , il n'y a rien à prouver. Si un arbre est obtenu par union pondérée d'un arbre à  $m$  nœuds et d'un arbre à  $n - m$  nœuds, avec  $1 \leq m \leq n/2$ , sa hauteur est majorée par

$$\max(1 + \lfloor \log_2(n - m) \rfloor, 2 + \lfloor \log_2 m \rfloor).$$

Comme  $\log_2 m \leq \log_2(n/2) = \log_2 n - 1$ , cette valeur est majorée par  $1 + \lfloor \log_2 n \rfloor$ .  $\square$

En déduire un encadrement de la hauteur de l'arbre **si celui-ci est binaire** et bien équilibré à la façon Union-Find ? On fera le lien avec les arbres AVL un peu plus loin.

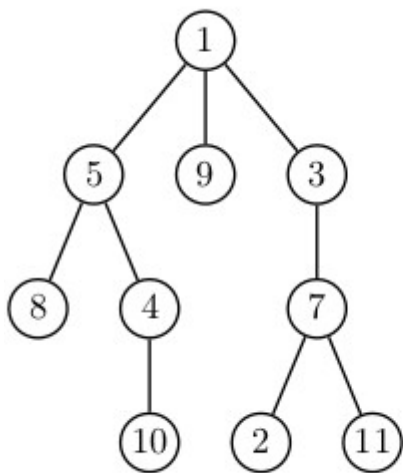


# Complexité amortie via arbres équilibrés et « compressés »

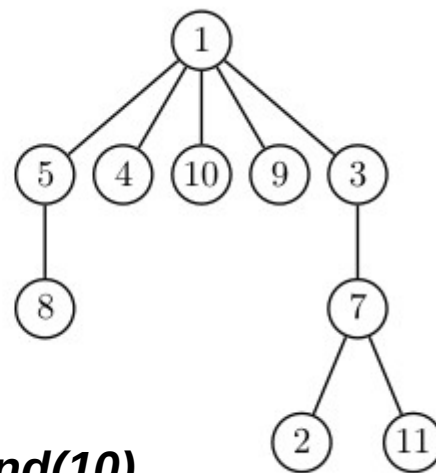
La complexité de **FIND** est donc déjà réduite à  $1 + \lfloor \log_2 n \rfloor$  pour chaque arête ( $O(h)$ )  
soit  $O(\text{FIND}) = m * \log(n)$

Une stratégie complémentaire appliquée lors de la partie **FIND** de l'algorithme de Kruskal réduit drastiquement la complexité.

Cela est obtenu en « compressant le chemin de recherche » selon la règle suivante : *Après être remonté du nœud  $x$  à sa racine  $r$ , on refait le parcours en faisant de chaque nœud rencontré un fils de  $r$ .*



**Find(10)**



```
static int trouverAvecCompression(int x)
{
    int r = trouver(x);
    while (x != r)
    {
        int y = pere[x];
        pere[x] = r;
        x = y;
    }
    return r;
}
```

# Complexité amortie via arbres équilibrés et « compressés »

C'est le principe de la complexité amortie évoquée dans le cours sur Kruskal. On investit un peu plus à un moment en réarrangeant la structure de données pour gagner au final dans la séquence d'opérations à effectuer.

**Complexité quasi-linéaire due à Tarjan :** Avec l'union pondérée et la compression des chemins, une suite de  $n - 1$  « UNIONS » de complexité unitaire  $O(1)$  et de  $m$  « FINDs » ( $m > n$ ) se réalise en temps  $O(n + m \cdot \alpha(n, m))$ , où  $\alpha$  est l'inverse d'une sorte de fonction d'Ackermann.

En fait, on a  $\alpha(n, m) \leq 2$  pour  $m \geq n$  et  $n < 2^{65536}$  et par conséquent, l'algorithme précédent **se comporte, d'un point de vue pratique, comme un algorithme linéaire** en  $n + m$  ( $\alpha(n, m)$  quasi constant). Pourtant, Tarjan a montré qu'il n'est pas linéaire en réalité et on ne connaît pas à ce jour d'algorithme strictement linéaire.

On lui doit le principe de la complexité amortie.

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$



[https://fr.wikipedia.org/wiki/Robert\\_Tarjan](https://fr.wikipedia.org/wiki/Robert_Tarjan)

[https://fr.wikipedia.org/wiki/Fonction\\_d%27Ackermann](https://fr.wikipedia.org/wiki/Fonction_d%27Ackermann)



# Notion de complexité amortie

On doit distinguer trois types de complexité :

La complexité dans le **pire cas**, qui donne une borne supérieure ;

La complexité **moyenne**, qui nécessite un calcul de probabilités ;

La complexité **amortie** qui donne une borne supérieure pour le pire cas lorsqu'une suite d'opérations est effectuée. Une opération précise peut être en  $O(n)$  par exemple mais une suite de  $m$  telles opérations en  $O(m \cdot \log(n))$  au lieu de  $O(mn)$  en amortissant les coûts successifs. La complexité amortie est une complexité moyenne calculée d'une façon non probabiliste. C'est cette complexité qui est évoquée pour le calcul d'un MST type Kruskal et inventée par Tarjan *et al.*

<https://www.cs.princeton.edu/~ret/>

# Notion de complexité amortie

Prenons l'exemple d'une pile et ses opérations :

PileVide() : retourne une pile vide ;

P.EstVide() : indique si la pile est vide ou non ;

P.Empiler(x) : ajoute x au sommet de la pile ;

P.Sommet() : retourne l'élément au sommet de la pile ;

P.Dépiler() : retire l'élément du sommet de la pile.

On ajoute une opération supplémentaire :

P.DépilerPlusieurs(k) : retire les k premiers éléments de la pile ou jusqu'à ce que la pile soit vide ;

Toutes les opérations se font en  $O(1)$  sauf la dernière qui se fait en  $O(\min(|P|, k))$ .

Supposons qu'on effectue **n** opérations parmi **Empiler**, **Dépiler** et **DépilerPlusieurs** ;  
Quel est le coût maximum possible ?

Comme la taille de la pile est au plus  $n$  et que l'opération **DépilerPlusieurs** se fait en temps  $O(n)$ , alors le coût total sera  $O(n^2)$  ; L'analyse précédente est correcte, mais pas assez précise.

■ Les piles et les files sont des sacs.

■ Une pile est un sac LIFO (Last-In First-Out)

- ▶ lorsqu'on retire un élément, on récupère toujours le dernier ajouté.
- ▶ ajouter se dit *empiler* (push), retirer se dit *dépiler* (pop).

■ Une file est un sac FIFO (First-in First-Out)

- ▶ on retire les éléments dans l'ordre de leur insertion.
- ▶ ajouter se dit *enfiler*, retirer se dit *défiler*.



# Notion de complexité amortie

En fait, si on effectue plusieurs opérations **DépilerPlusieurs** consécutives, alors elles ne peuvent pas toutes être coûteuses ;

Plus précisément, si on commence avec une pile vide, alors le nombre total d'opérations **Empiler** est plus grand ou égal que le nombre total d'opérations **Dépiler**. On en conclut que le temps total pris est  $O(n)$ , ce qui est plus précis que  $O(n^2)$  ; Le coût moyen d'une opération est donc  $O(n)/n = O(1)$  ; C'est ce qu'on appelle le coût amorti.

À noter que le coût amorti ne se base pas sur des probabilités, mais considère tous les cas possibles. La méthode la plus utilisée pour calculer ce coût amorti est la méthode du potentiel.

**Voir l'exemple d'un compteur binaire**  
**Coût amorti = 2 soit un coût total de  $2n$**   
**au lieu d'une complexité au pire en  $O(nk)$**

Valeur	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0

## Incrémentation d'un compteur binaire

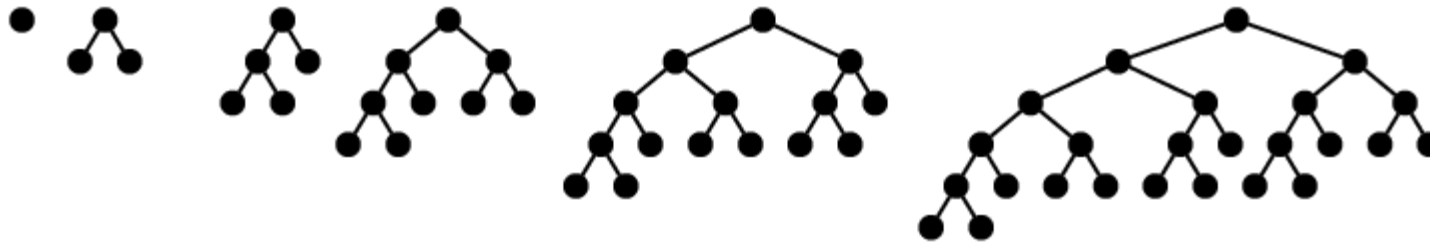
```

1: procedure INCRÉMENTER(A : tableau de bits)
2:    $i \leftarrow 0$ 
3:   tant que  $i < |A|$  et  $A[i] = 1$  faire
4:      $A[i] \leftarrow 0$ 
5:      $i \leftarrow i + 1$ 
6:   fin tant que
7:   si  $i < |A|$  alors
8:      $A[i] \leftarrow 1$ 
9:   fin si
10: fin procedure
  
```

- Soit  $k = |A|$  ;
- Alors une incrémentation se fait en  $O(k)$  ;
- Une suite de  $n$  incrémentations a un coût total de  $O(nk)$ .

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

# Où l'on va parler de Fibonacci



Un autre exemple d'arbre équilibré est fourni par les arbres de Fibonacci, qui sont des arbres binaires  $A_n$  définis récursivement tels que les sous-arbres gauche et droit de  $A_n$  sont respectivement  $A_{n-1}$  et  $A_{n-2}$ . Ce qu'on pourrait écrire récursivement  $A_n = (r, A_{n-1}, A_{n-2})$  ou  $A_n = (A_{n-1}, r, A_{n-2})$  en fonction des écoles.

## Suite de Fibonacci :

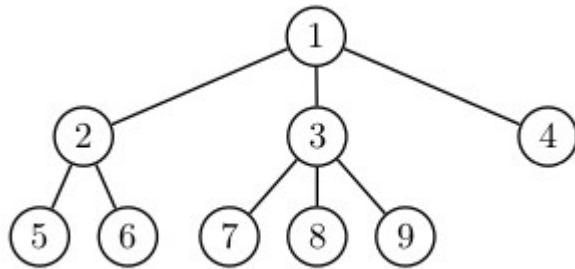
$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$	$F_9$	$F_{10}$	$F_{11}$	$F_{12}$	$F_{13}$	$F_{14}$	$F_{15}$	$F_{16}$	...	$F_n$
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	...	$F_{n-1} + F_{n-2}$

La suite est définie par  $F_0 = 0$ ,  $F_1 = 1$ , et  $F_n = F_{n-1} + F_{n-2}$ , pour  $n > 1$ .

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \varphi'^n), \quad \text{avec} \quad \varphi = \frac{1 + \sqrt{5}}{2} \quad \text{et} \quad \varphi' = \frac{1 - \sqrt{5}}{2} = -\frac{1}{\varphi}.$$



# Récurtivité et nombre de Catalan



## 3.1 Compter les arbres binaires

La figure 3.11 montre les arbres binaires ayant 1, 2, 3 et 4 nœuds.

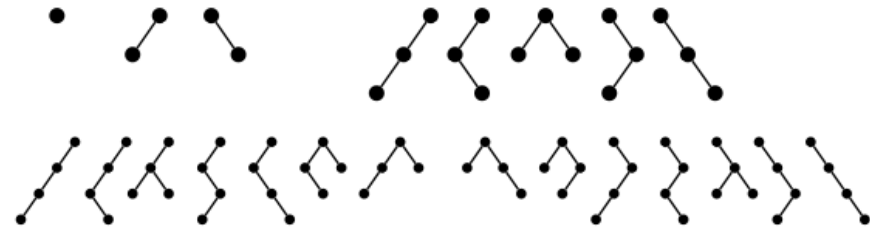


FIG. 3.11 – Les premiers arbres binaires.

Notons  $b_n$  le nombre d'arbres à  $n$  nœuds. On a donc  $b_0 = b_1 = 1$ ,  $b_2 = 2$ ,  $b_3 = 5$ ,  $b_4 = 14$ . Comme tout arbre  $A$  non vide s'écrit de manière unique sous forme d'un triplet  $(A_g, r, A_d)$ , on a pour  $n \geq 1$  la formule

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}$$

La série génératrice  $B(x) = \sum_{n \geq 0} b_n x^n$  vérifie donc l'équation

$$xB^2(x) - B(x) + 1 = 0.$$

Comme les  $b_n$  sont positifs, la résolution de cette équation donne

$$b_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

Les nombres  $b_n$  sont connus comme les *nombre de Catalan*. L'expression donne aussi  $b_n \sim \pi^{-1/2} 4^n n^{-3/2} + O(4^n n^{-5/2})$ .

$$T = (1, \{(2, \{(5), (6)\}), (3, \{(7), (8), (9)\}), (4)\})$$

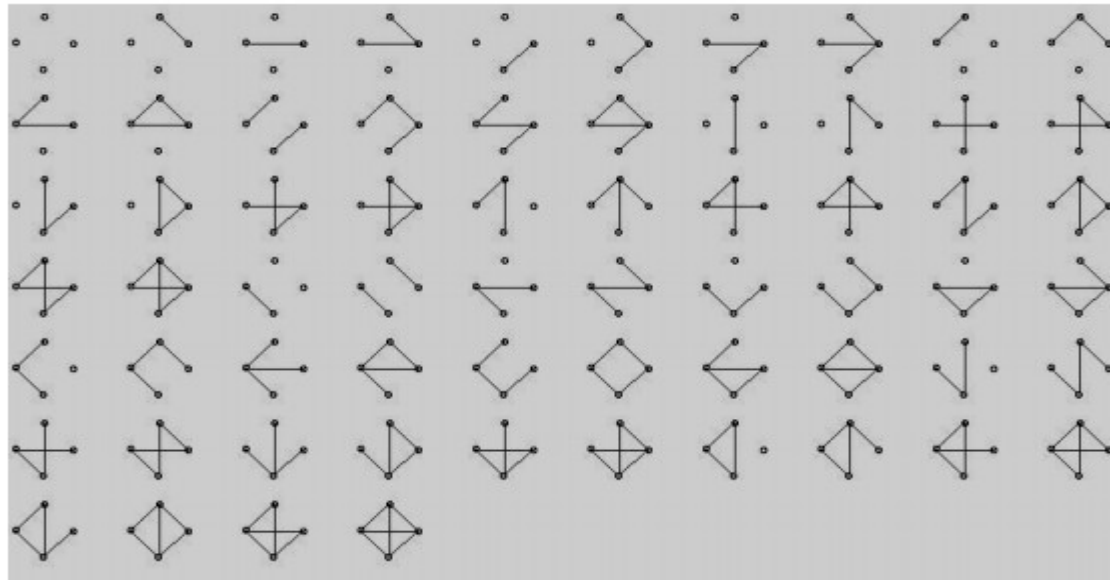
Avec cette notation récursive (*nœud*, {*AG*}, {*AD*}) on devrait pouvoir générer tous les arbres binaires d'ordre  $n$  et les compter en les énumérant. Imaginer le code ?

# Un bonus

## Tous les graphes avec $N$ sommets

On se donne  $N$  points numérotés de 0 à  $N - 1$ . On veut construire tous les graphes ayant comme sommets ces points. Pour cela il s'agit de placer des arêtes de jonction de toutes les façons possibles.

Le nombre d'arêtes maximal est le nombre de façons de choisir deux points parmi les  $N$ , soit  $C_N^2 = N(N - 1) / 2$ . Par exemple 6 arêtes pour  $N = 4$ . Chaque graphe est obtenu en prenant une partie de cet ensemble d'arêtes. Le nombre de parties de cet ensemble est  $2^{N(N - 1) / 2}$ . C'est aussi le nombre des graphes, par exemple 64 pour  $N=4$ . Nous allons maintenant les dessiner, comme ci-dessous pour  $N = 4$ .





# Equilibrage, Complexité, ABR et Tas

**Définition** :  $\mathcal{B} = \emptyset + \langle \bullet, \mathcal{B}, \mathcal{B} \rangle$  : un *arbre binaire* est

- soit vide ( $\emptyset$ ),
- soit constitué d'un nœud racine, d'un sous-arbre gauche qui est un arbre binaire et d'un sous-arbre droit qui est un arbre binaire.

**Parcours** :  $\mathcal{B} \rightarrow \text{Liste (sommets)}$

Soit  $B = \langle \bullet, G, D \rangle$

- 1 préfixe :  $\text{PREF}(B) = [\text{visit}(\bullet), \text{PREF}(G), \text{PREF}(D)]$
- 2 infixe (ou symétrique) :  $\text{INF}(B) = [\text{INF}(G), \text{visit}(\bullet), \text{INF}(D)]$
- 3 suffixe :  $\text{SUF}(B) = [\text{SUF}(G), \text{SUF}(D), \text{visit}(\bullet)]$

**Définition** : soit  $B = \langle \bullet, G, D \rangle$  un arbre binaire, le complété de  $B$ , noté  $\bar{B}$  est l'arbre obtenu en remplaçant tous les sous-arbres vides  $\emptyset$  par une feuille, notée  $\square$ .

**Lemme** : l'arbre complété  $\bar{B}$  a  $n$  nœuds (internes) a  $(n + 1)$  feuilles.

*Preuve par induction.*



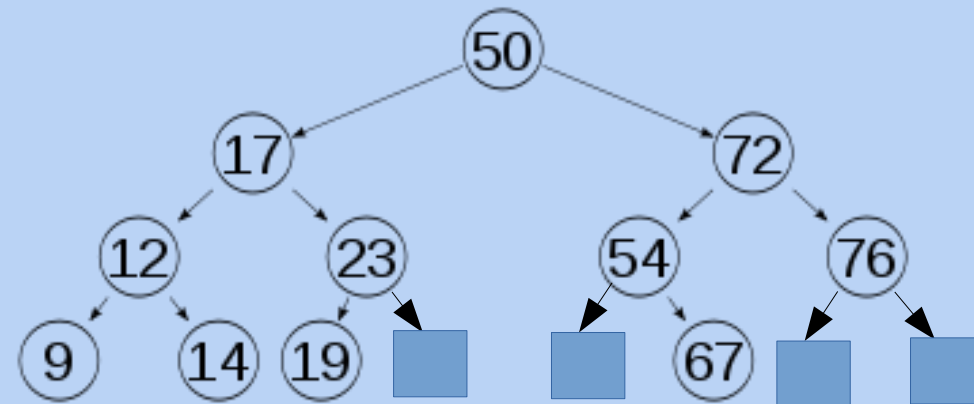
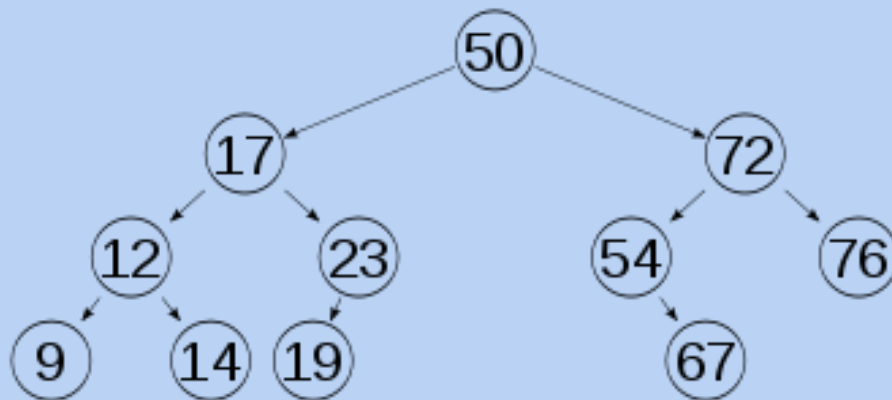
# Equilibrage, Complexité, ABR et Tas

**Définition** : un ABR est un arbre binaire étiqueté tel que en chaque nœud, l'étiquette est plus grande que toutes les étiquettes du sous-arbre gauche, et plus petite que toutes les étiquettes du sous-arbre droit.

**Propriété** : le parcours infixe INF(ABR) d'un ABR donne la suite des étiquettes en ordre croissant. (Preuve par induction)

**Utilité** : Algorithmes de recherche, ajout et suppression de clés rapides reposant sur :

- Parcours d'une branche donc en  $O(h)$
- Algorithmes simples : modifications minimales



# Equilibrage, Complexité, ABR et Tas

## Définition d'un AVL

Un *AVL* (Adelson–Velsky, Landis) est un ABR t.q. en chaque nœud, la hauteur du sous-arbre gauche et celle du sous-arbre droit diffèrent au plus de 1.

## Hauteur d'un AVL

Soit  $h$  la hauteur d'un AVL avec  $n$  nœuds :

$$\log_2(n + 1) \leq h + 1 < 1.44 \log_2 n.$$

Au pire les arbres de Fibonacci :

$$F_0 = \langle \bullet, \emptyset, \emptyset \rangle, F_1 = \langle \bullet, F_0, \emptyset \rangle, F_n = \langle \bullet, F_{n-1}, F_{n-2} \rangle$$

# Equilibrage, Complexité, AVL et Tas

On a toujours  $n \leq 2^{h+1} - 1$  donc  $\log_2(1+n) \leq 1+h$

On pose  $N(h)$  le nombre de nœuds minimum d'un arbre AVL de hauteur  $h$ .

Alors  $N(h) = 1 + N(h-1) + N(h-2)$  car un arbre AVL de hauteur  $h$  aura un sous-arbre de hauteur  $h-1$  et l'autre sous-arbre de hauteur  $h-2$ .

La suite  $F(h) = 1 + N(h)$  vérifie  $F(0)=2$ ,  $F(1)=3$  et  $F(h+2)=F(h+1)+F(h)$  pour  $h \geq 0$  donc

$F(h) = 1/\sqrt{5} (\varphi^{h+3} - (-\varphi)^{-(h+3)})$  où  $\varphi$  est le nombre d'or

Soit  $1+n \geq F(h) \geq 1/\sqrt{5}(\varphi^{h+3} - 1)$  d'où en utilisant le logarithme en base  $\varphi$

$$h+3 < \log_{\varphi}(\sqrt{5}(1+n)) < \log_2(1+n)/\log_2\varphi + 2 < 1.44 \cdot \log_2(n+2) + 2$$

$$\text{d'où } \log_2(1+n) \leq h+1 < 1.44 \cdot \log_2(n+2)$$

Par exemple, pour un arbre AVL qui a 100 000 nœuds, la hauteur est comprise entre 17 et 25. C'est le nombre d'opérations qu'il faut donc pour rechercher, insérer ou supprimer une donnée dans un tel arbre. Impressionnant n'est-ce pas ?

# Equilibrage, Complexité, AVL et Tas

Les arbres AVL sont des arbres binaires de recherche équilibrés qui préserve l'ordre infixe donc  $INF(T)=INF(AVL(T))$  si  $T$  est un arbre binaire quelconque et donc l'ordre pour un ABR.

N.B. Penser aux Bases de données qui sont des structures de recherche sur clés. Les arbres AVL constituent une structure de données concurrente pour ce type de recherche.

Ci-dessous une comparaison des complexités d'opérations de base sur les ABR.

Complexité	Moyenne	Pire
ABR	$O(\log n)$	$O(n)$
ABR Eq.	$O(\log n)$	$O(\log n)$

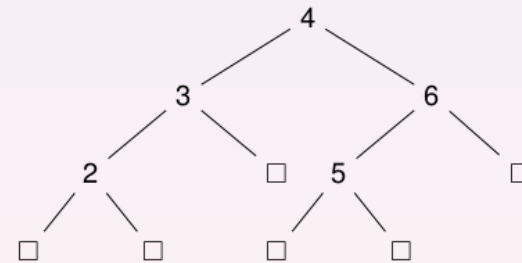
# Equilibrage, Complexité, AVL et Tas

## 1 Arbres binaires de recherche

- en moyenne hauteur en  $O(\log n)$ , mais au pire  $O(n)$  (liste)
- algorithmes Recherche, Ajout et Suppression : parcours d'une branche

## 2 Arbres de recherche équilibrés

- hauteur toujours en  $O(\log n)$
- algorithmes Recherche, Ajout et Suppression : parcours d'une branche
- algorithmes sophistiqués : modifications locales sur une branche (rotations, éclatements) pour maintenir hauteur en  $O(\log n)$

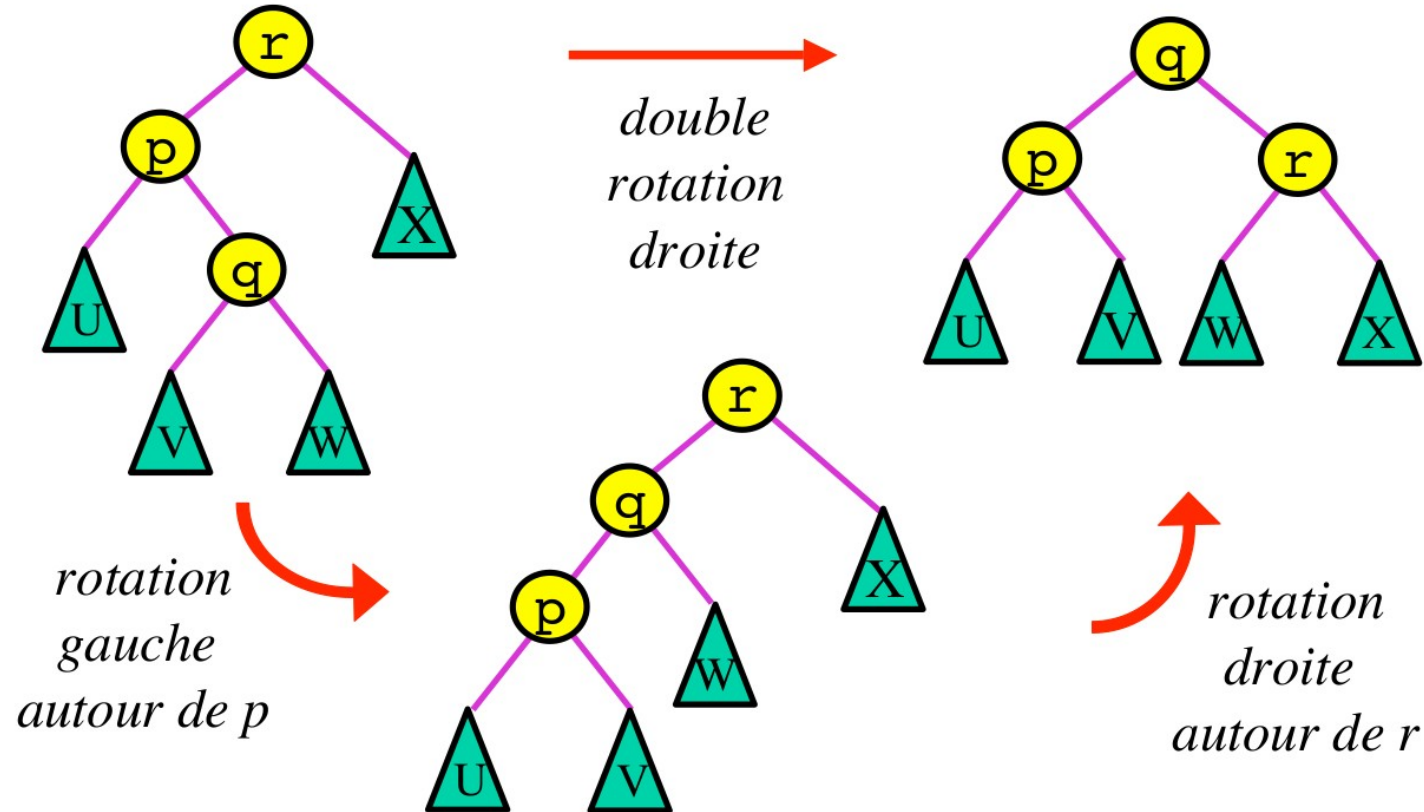


Ajout de 1 dans l'ABR parfait

# Création d'un AVL

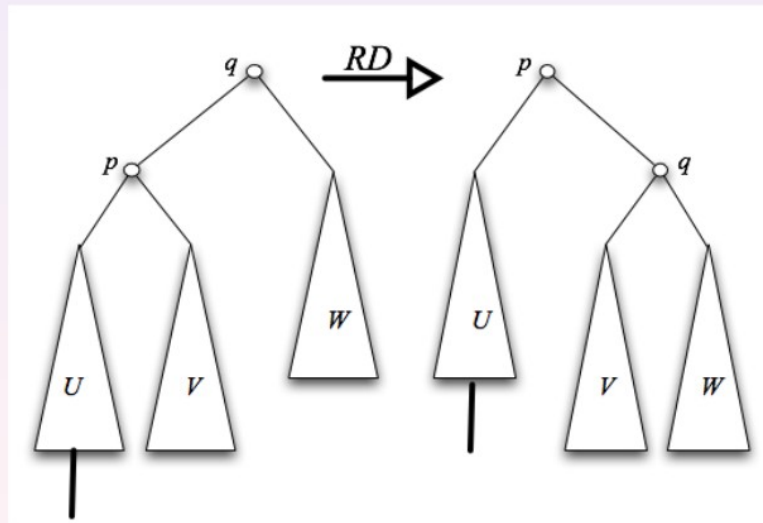
Pour rééquilibrer, nécessité de définir des opérations sophistiquées sur les arbres qu'on appelle rotations d'arbre : rotation droite, rotation gauche, double rotations

## *Double rotation*



# Création d'un AVL

- 1 Rotation droite  $A = \langle q, \langle p, U, V \rangle, W \rangle \implies RD(A) = \langle p, U, \langle q, V, W \rangle \rangle$   
*propriété d'ABR* : parcours infixe :  
 $INF(A) = INF(RD(A)) = INF(U).p.INF(V).q.INF(W)$   
hauteur :  $h(U) = h(V) = h(W) = H - 2$  . Arbre initial  $A$  : hauteur  $H$  et déséquilibre à gauche en  $q$ .  
Si insertion aux feuilles de  $U$ , arbre résultant  $RD(A)$  : hauteur  $H$  et pas de déséquilibre, ni en  $p$ , ni en  $q$ .



- 2 Rotation gauche  $A = \langle p, U, \langle q, V, W \rangle \rangle \implies RG(A) = \langle q, \langle p, U, V \rangle, W \rangle$   
*même propriété que 1)*



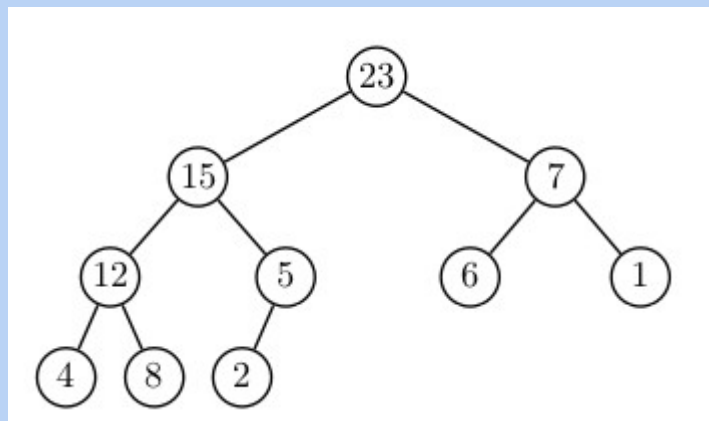
# Equilibrage, Complexité, Tas

Dans la série des arbres binaires équilibrés, je demande les arbres tassés.

Un arbre binaire est **tassé** si tous les niveaux sont entièrement remplis à l'exception peut-être du dernier niveau, et ce dernier niveau est rempli «à gauche». On parle d'arbre quasi complet.

**Proposition** : La hauteur d'un arbre binaire tassé à  $n$  nœuds est optimale et égale à  $\lfloor \log_2 n \rfloor$

Un **tas** (en anglais « **heap** ») est un **arbre tassé** tel que le contenu de chaque nœud **vérifie un ordre** (supérieur ou égal : tas-max ou inférieur ou égal : tas-min) par rapport à celui de ses fils. Ceci entraîne, par transitivité, que le contenu de chaque nœud est supérieur ou égal à celui de ses descendants (respectivement inférieur ou égal). On l'appelle aussi **arbre-tournoi**. C'est donc assez différent d'un ABR par exemple.



# Tas binaire

**Tas binaire** : arbre tournoi parfait (arbre binaire de type tas-max quasi complet)

$\text{val}(\text{noeud}) \leq \text{val}(\text{noeud\_fils})$

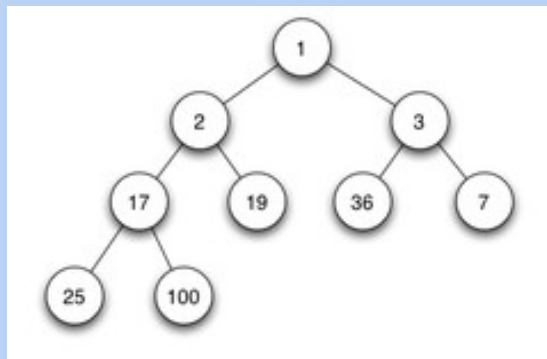
$$h = \lfloor \log_2 n \rfloor$$

+

$2^{h'}$  sommets pour tout  $h' < h$

+

$n - 2^h + 1$  feuilles de hauteur  $h$   
regroupées à gauche



A min-heap

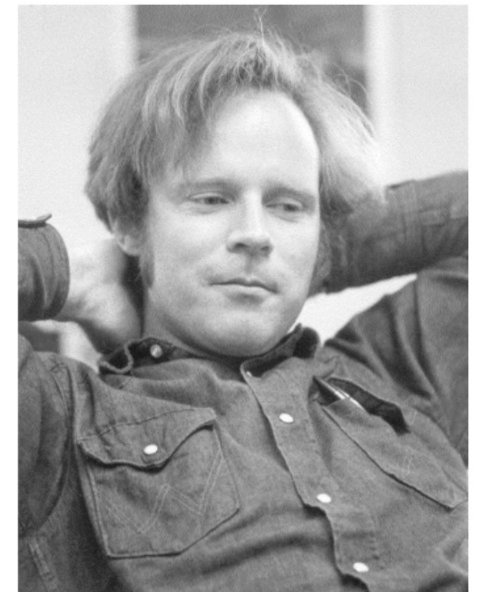


Figure 1: R. W. Floyd [1964]

[https://fr.wikipedia.org/wiki/Tas\\_binaire](https://fr.wikipedia.org/wiki/Tas_binaire)

Si la relation d'ordre choisie est "supérieure ou égale", on parle alors de tas-max (ou max-heap). Si la relation est "inférieure ou égale", on parle alors de tas-min (ou min-heap).

# Tas binaire

## 4.2 Implantation d'un tas

Un tas s'implante facilement à l'aide d'un simple tableau. Les nœuds d'un arbre tassé sont numérotés en largeur, de gauche à droite. Ces numéros sont des indices dans un tableau (cf figure 4.16).

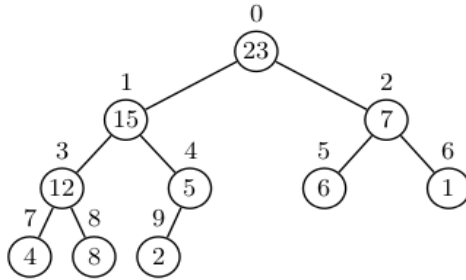


FIG. 4.16 – Un arbre tassé, avec la numérotation de ses nœuds.

L'élément d'indice  $i$  du tableau est le contenu du nœud de numéro  $i$ . Dans notre exemple, le tableau est :

$i$	0	1	2	3	4	5	6	7	8	9
$a_i$	23	15	7	12	5	6	1	4	8	2

Il existe plusieurs variantes plus efficaces. La classe des arbres binaires représentables par un tableau sont les tas binaires. Toutes les opérations de base se font pratiquement en  $O(h)$  donc en  $O(\log n)$

La construction naïve d'un tas-max de  $n$  éléments (par ajout successifs à partir d'un tableau par ex.) a un coût temporel  $O(n \log n)$  (en effet  $\sum \log(k)$  est en  $n \log(n)$ ). Mais il existe une stratégie par forêt en  $O(n)$  efficace dans certains cas.

La complexité du tri par tas est, dans le pire des cas, en  $O(n \log n)$ . En effet, on appelle  $n$  fois chacune des méthodes ajouter et supprimer. (heapsort algorithm)

racine	: nœud 0
parent du nœud $i$	: nœud $\lfloor (i-1)/2 \rfloor$
fil gauche du nœud $i$	: nœud $2i+1$
fil droit du nœud $i$	: nœud $2i+2$
nœud $i$ est une feuille	: $2i+1 \geq n$
nœud $i$ a un fils droit	: $2i+2 < n$

L'insertion d'un nouvel élément  $v$  se fait en deux temps : d'abord, l'élément est ajouté comme contenu d'un nouveau nœud à la fin du dernier niveau de l'arbre, pour que l'arbre reste tassé. Ensuite, le contenu de ce nœud, soit  $v$ , est comparé au contenu de son père. Tant que le contenu du père est plus petit que  $v$ , le contenu du père est descendu vers le fils. À la fin, on remplace par  $v$  le dernier contenu abaissé (voir figure 4.17).

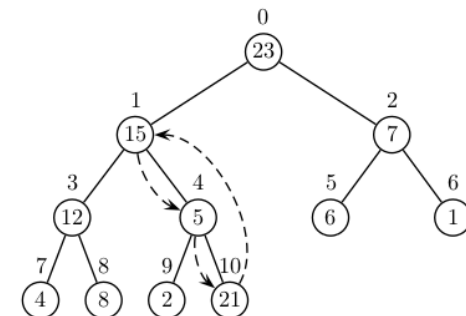


FIG. 4.17 – Un tas, avec remontée de la valeur 21 après insertion.

# Tas binaire

## Pour ce tri :

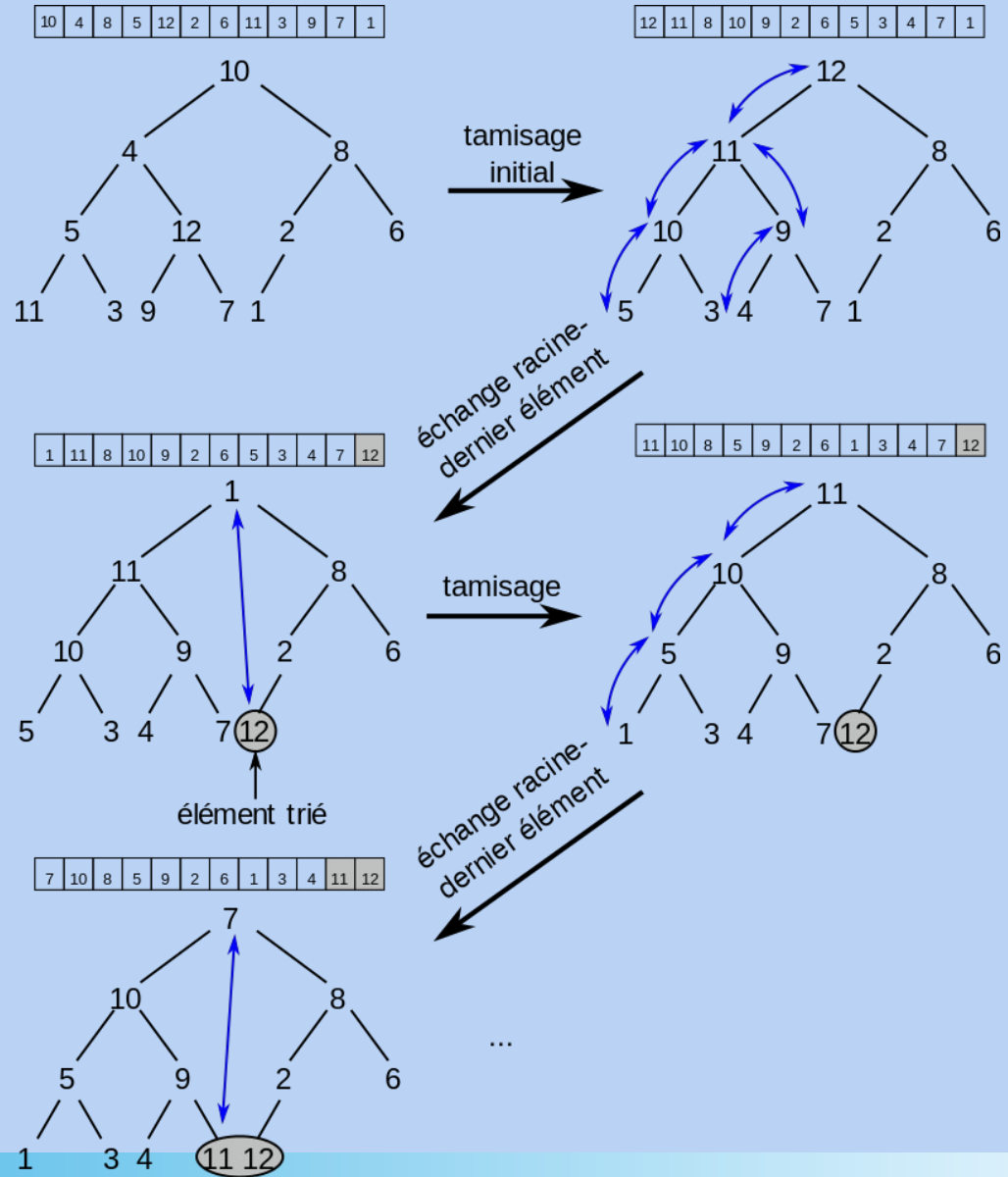
1. Construire un tas à partir du tableau ;
2. Extraire de ce tas les maximums successifs et mettre dans le tableau.

Avantage : même si pas optimal comme tri, il est « en place » : cad que tout se fait dans le même tableau sans allocation supplémentaire

```
void tri_tas(int* tableau, int taille)
```

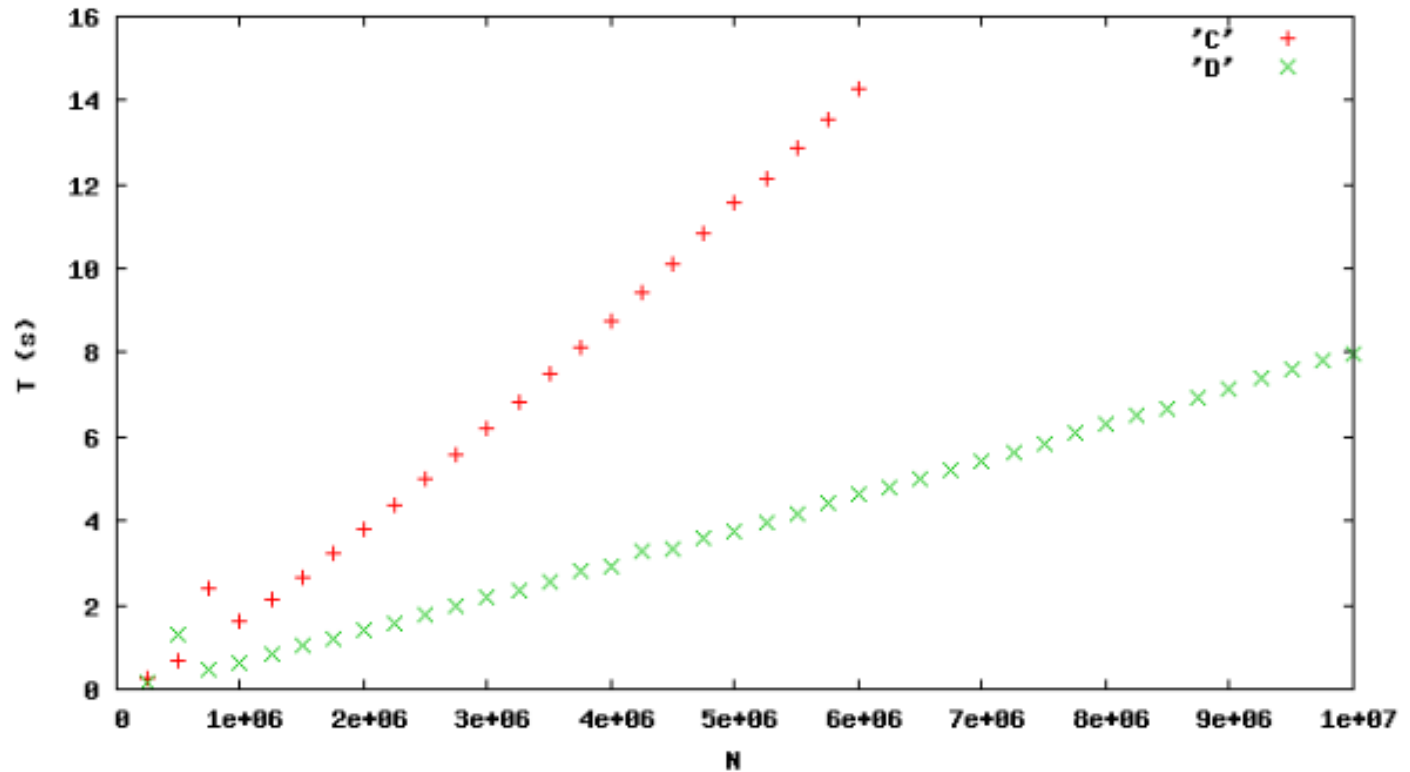
La complexité du tri par tas est, dans le pire des cas, en  $O(n \log n)$ . En effet, on appelle  $n$  fois chacune des méthodes ajouter et supprimer. (heapsort algorithm)

Il existe plusieurs variantes plus efficaces.



# Tas binaire

La librairie de java fournit une méthode de tri impressionnante.



La [méthode](#) est dans la classe [Arrays](#), du package `java.util`.

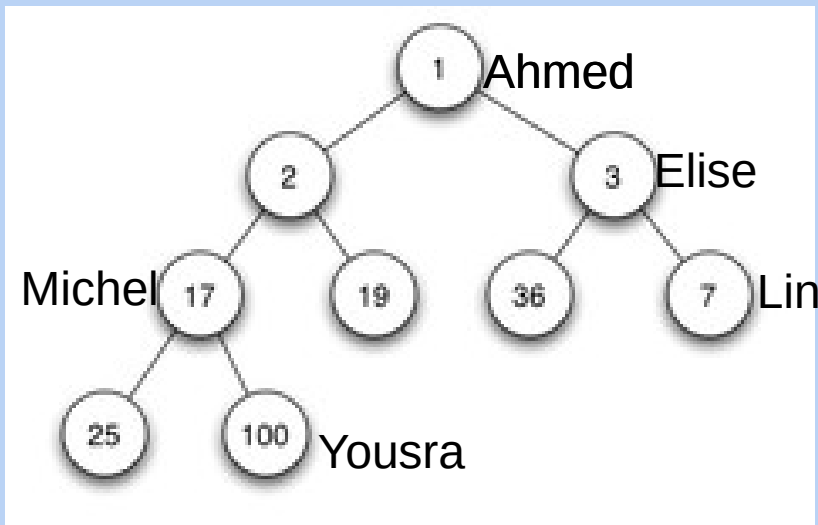
```
static void sort(int [] t) { java.util.Arrays.sort(t) ; }
```

Selon la documentation, la méthode `Arrays.sort` est écrite à partir de cet [article de Jon Bentley et M. Douglas McIlroy](#).

<http://pauillac.inria.fr/~maranget/X/421/09/bentley93engineering.pdf>

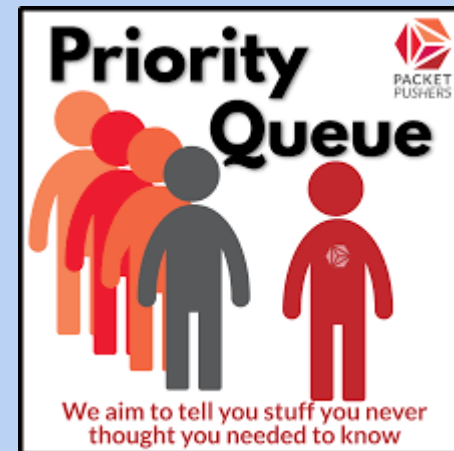
<http://gallium.inria.fr/~maranget/X/421/poly/arbre-bin.html>

# Files de Priorité



Un tas-min avec indice de patience

	Unsorted array	Sorted array	Balanced tree
Insert( $Q, x$ )	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum( $Q$ )	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum( $Q$ )	$O(n)$	$O(1)$	$O(\log n)$



<https://www.mainjava.com/java/core-java/priorityqueue-class-in-java-with-programming-example/>

En **Python**, le module 'heapq' implémente la file de priorité. En **C++**, la **STL** fournit une implémentation nommée priority\_queue. Ce type de file se base au choix sur un des conteneurs présents dans la STL (liste, vecteur, etc.) et manipule des objets d'un type défini par l'utilisateur. Il est nécessaire de fournir une fonction de comparaison qui permettra à la bibliothèque d'ordonner la file.

[https://en.wikipedia.org/wiki/Heap\\_\(data\\_structure\)](https://en.wikipedia.org/wiki/Heap_(data_structure))

En **Python**, le module 'heapq' implémente

# Files de Priorité

**File de priorité** : un multi-ensemble de valeurs dans un espace ordonné (ex. clés-valeurs). Plus explicitement, une file de priorité est un objet contenant des éléments ayant chacun une priorité représentée par un nombre entier positif.

En informatique, une file de priorité est un type abstrait élémentaire sur laquelle on peut effectuer trois opérations :

- insérer un élément ;
- extraire l'élément ayant la plus grande clé ;
- tester si la file de priorité est vide ou pas.

On veut donc pouvoir effectuer les deux opérations suivantes de manière **efficace** :

- **push(x, f)** qui ajoute un nouvel élément de priorité x dans la file ;
- **pop(f)** qui renvoie et supprime de la file l'élément ayant la plus grande priorité.



# Files de Priorité

Les files de priorité sont donc très proches des piles et des files (les opérations push et pop sont semblables) mais au lieu de ressortir les éléments dans un ordre dépendant de l'ordre dans lequel ils ont été insérés, on veut les ressortir dans un ordre qui dépend de leur priorité.

**Le tas est une bonne structure pour implémenter** une telle fonctionnalité. En effet une liste chaînée gèrerait le push en  $O(1)$  mais le pop en  $O(n)$ . On espère donc un  $O(\log n)$  généralisé.

**Applications :** gérer une file d'attente avec priorité (s'asseoir dans un bus après y être monté, être reçu à un guichet après avoir accédé à la queue ou file d'attente, gérer des priorités des processus sur un système UNIX après les avoir lancées dans la file, des ordres en bourse, etc. (à compléter avec des applications sur des piles par exemples dans la « vrai » vie)

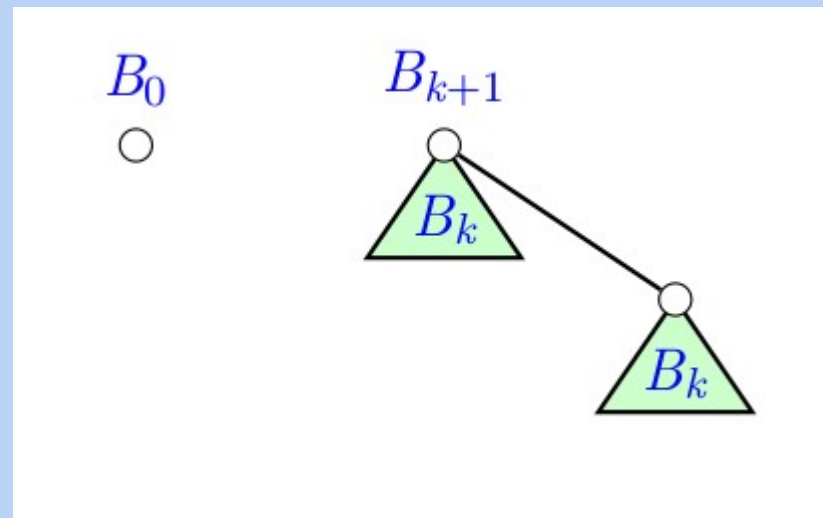
Il faut pouvoir le faire de façon très efficace et passer à l'échelle (plusieurs millions/milliards de sommets).

# Tas binomiaux

**Les arbres binomiaux** sont définis récursivement comme suit :

- L'arbre binomial d'ordre 0 est un simple nœud
- L'arbre binomial d'ordre  $k$  possède une racine de degré  $k$  et ses fils sont racines d'arbre binomiaux d'ordre 0, 1, 2,  $k-2$ ,  $k-1$  (**dans un ordre précis choisi une fois pour toutes**). Ils ne sont donc pas binaires. L'idée est d'améliorer la complexité amortie un peu dans l'idée de la compression de chemins de UNION FIND.

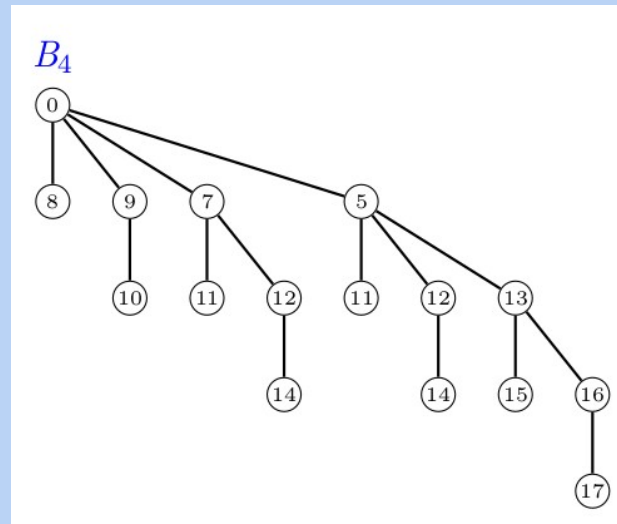
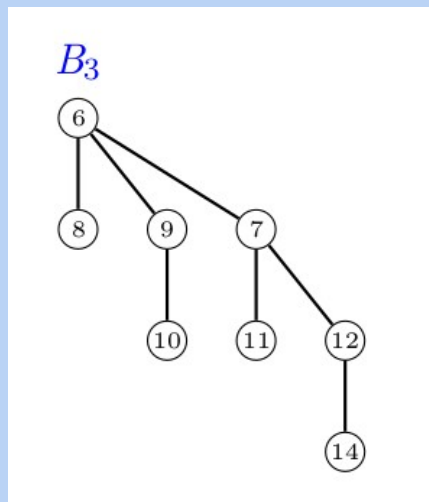
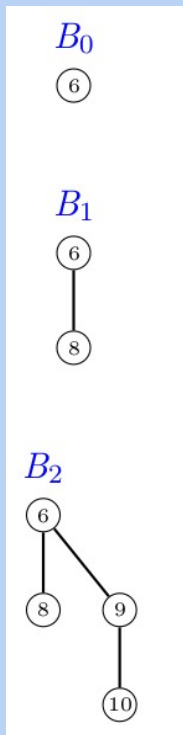
On peut les définir récursivement par : un nœud simple est l'arbre  $B_0$  et 2 arbres  $B_k$  sont réunis en un arbre  $B_{k+1}$ .



# Tas binomiaux

Un arbre binomial d'ordre  $k$  possède  $2^k$  nœuds, et a pour hauteur  $k$ . Le rang de l'arbre est le degré de sa racine  $k$  (parfois appelé degré de l'arbre).

Des variantes d'arbres binomiaux sont aussi utilisées pour construire les tas de Fibonacci.



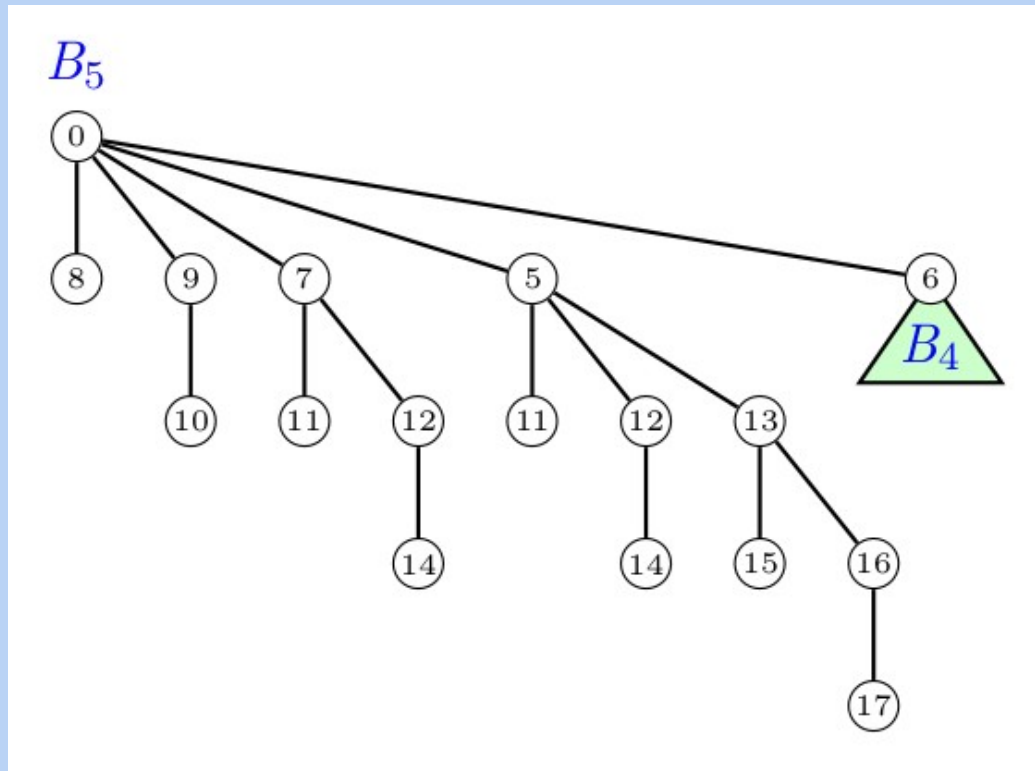
<https://www.di.ens.fr/~jv/>  
Jean Vuillemin

# Tas binomiaux

Le  $i^{\text{ème}}$  fils d'un nœud a un degré  $i-1$

Surtout, le nombre de nœuds sur chaque niveau est **un coefficient binomial** (voir TP Prog Imp L2)

Le degré maximal d'un nœud quelconque dans un arbre binomial à  $n$  nœuds est au plus  $\log(n)$  soit  $k$  pour  $B_k$



0:					1												
1:					1		1										
2:					1		2		1								
3:				1		3		3		1							
4:			1		4		6		4		1						
5:			1		5		10		10		5		1				
6:		1		6		15		20		15		6		1			
7:	1		7		21		35		35		21		7		1		
8:	1		8		28		56		70		56		28		8		1

# Triangle de Pascal

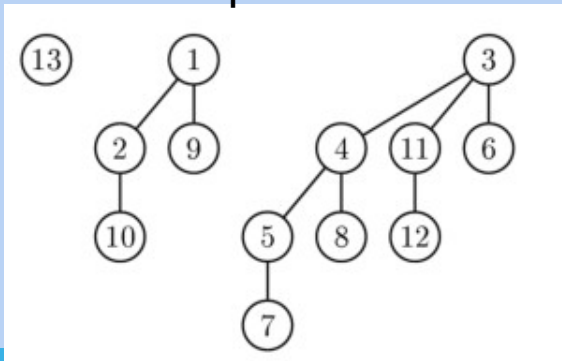
$$\binom{n}{k} = C_n^k = \frac{n!}{k!(n-k)!}$$

# Tas binomiaux

**Tas binomial** : Un tas binomial est un ensemble (ou collection ou forêt) d'arbres binomiaux (à comparer avec un tas binaire, qui correspond à un unique arbre binaire) qui respectent les règles suivantes :

- chaque arbre binomial dans le tas satisfait la contrainte de tas-minimum (ou tas-maximum au choix) : chaque clé d'un sommet est  $\geq$  à la clé du parent.
- pour tout entier  $k \geq 0$ , **il y a au plus un arbre de rang  $k$**  (ceci implique que pour  $n$  sommets, il y a au plus  $\log n + 1$  arbres binomiaux dans le tas et qu'au cours du cycle de vie d'un tas /insertion/suppression de nœuds on doit faire des fusions de tas par fusion d'arbres).

Comme on a au plus un arbre de rang  $k$   $B_k$  dans un tas binomial, et que le nombre de nœuds est  $2^k$ , si on a  $n$  éléments le tas qui le représente ne contient un arbre binomial  $B_k$  ssi le  $k^{\text{ième}}$  bit de la représentation binaire de  $n$  est à 1.



**Example:**  $n = 13$

$$\Rightarrow \text{bin}(13) = \langle 1, 1, 0, 1 \rangle$$

⇒ Si  $T$  est un tas binomial à 13 nœuds alors il est constitué des arbres binomiaux  $B_3$ ,  $B_2$  et  $B_0$  qui comportent respectivement 8, 4, et 1 nœuds, pour un total de  $n = 13$  nœuds.

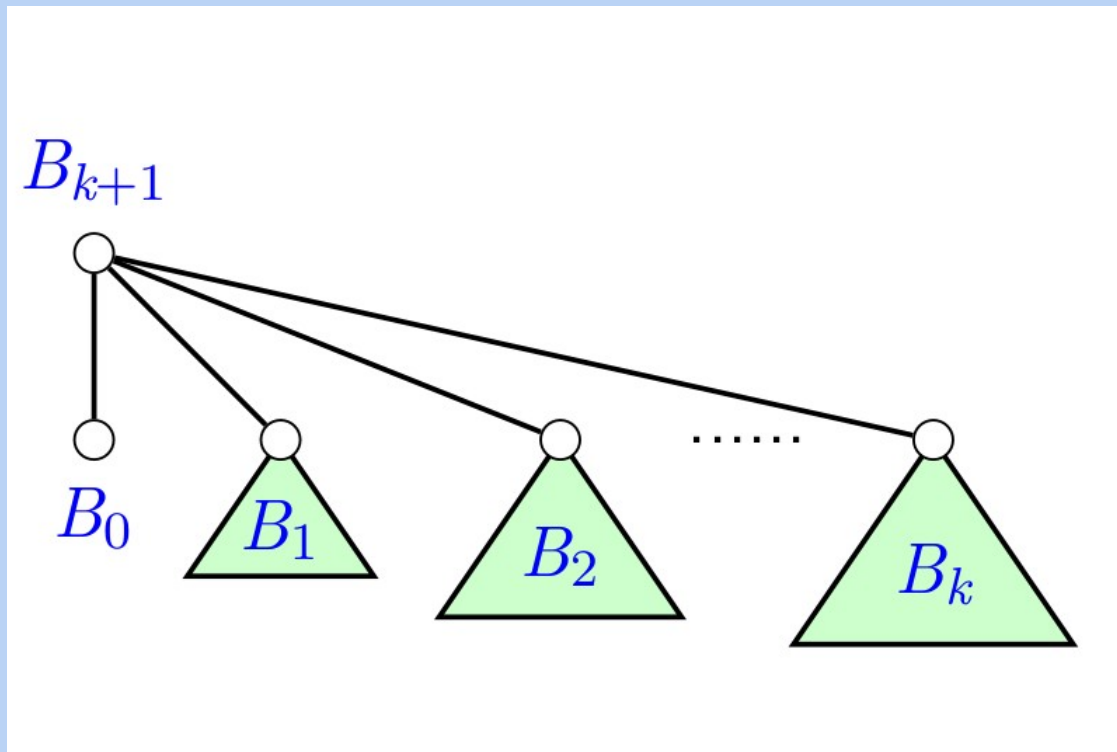
En informatique, un tas binomial est une structure de données assez proche du tas binaire, mais qui permet aussi de fusionner deux tas rapidement. Le tas binomial est donc une implémentation du **type abstrait** tas fusionnable, ie une **file à priorités** permettant des opérations de fusion. Ainsi, il supporte les opérations suivantes, toutes en  $O(\log n)$  : 34

- Insérer un nouvel élément au tas
- Trouver l'élément de plus petite clé
- Effacer du tas l'élément de plus petite clé
- Diminuer la clé d'un élément donné
- Effacer un élément donné du tas
- Fusionner deux tas en un seul

# Tas binomiaux

Si on extrait la racine  $B_{k+1}$  dans la file de priorité, que se passe-t-il ?

Du point de vue de la fusion de tas si dans le tas un autre arbre de rang entre 0 et  $k$  traîne ?



De plus, pour améliorer encore la complexité en temps, les racines des arbres binomiaux qui sont les éléments maximum de chaque arbre sont stockées dans une liste chaînée ordonnée par ordre croissant des arbres.

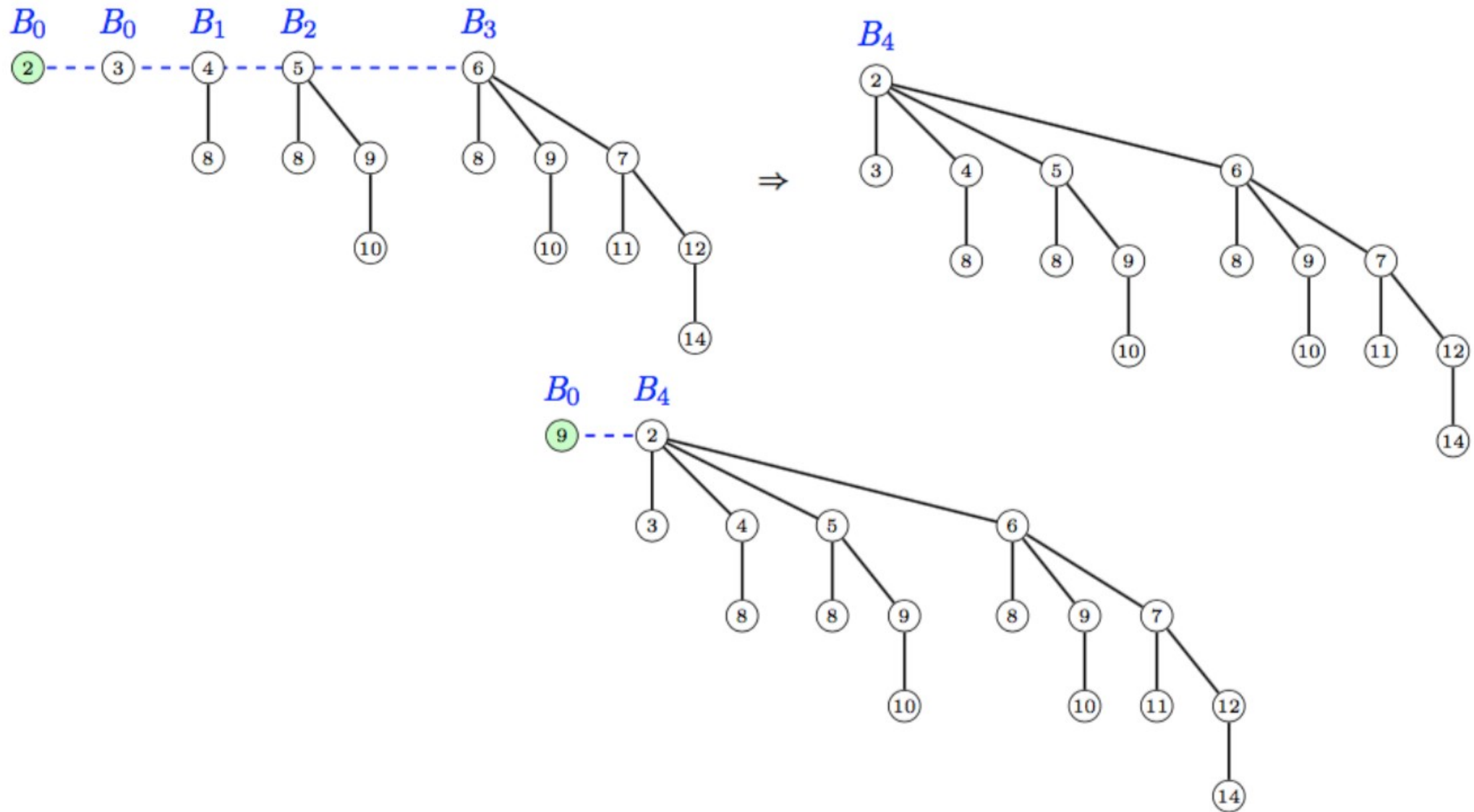
# Tas binomiaux

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap*
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

\*amortized cost



# Tas binomiaux



Insertion d'une clé suivi de fusions pour respecter les propriétés du tas

# Tas binomiaux -> Fibonacci

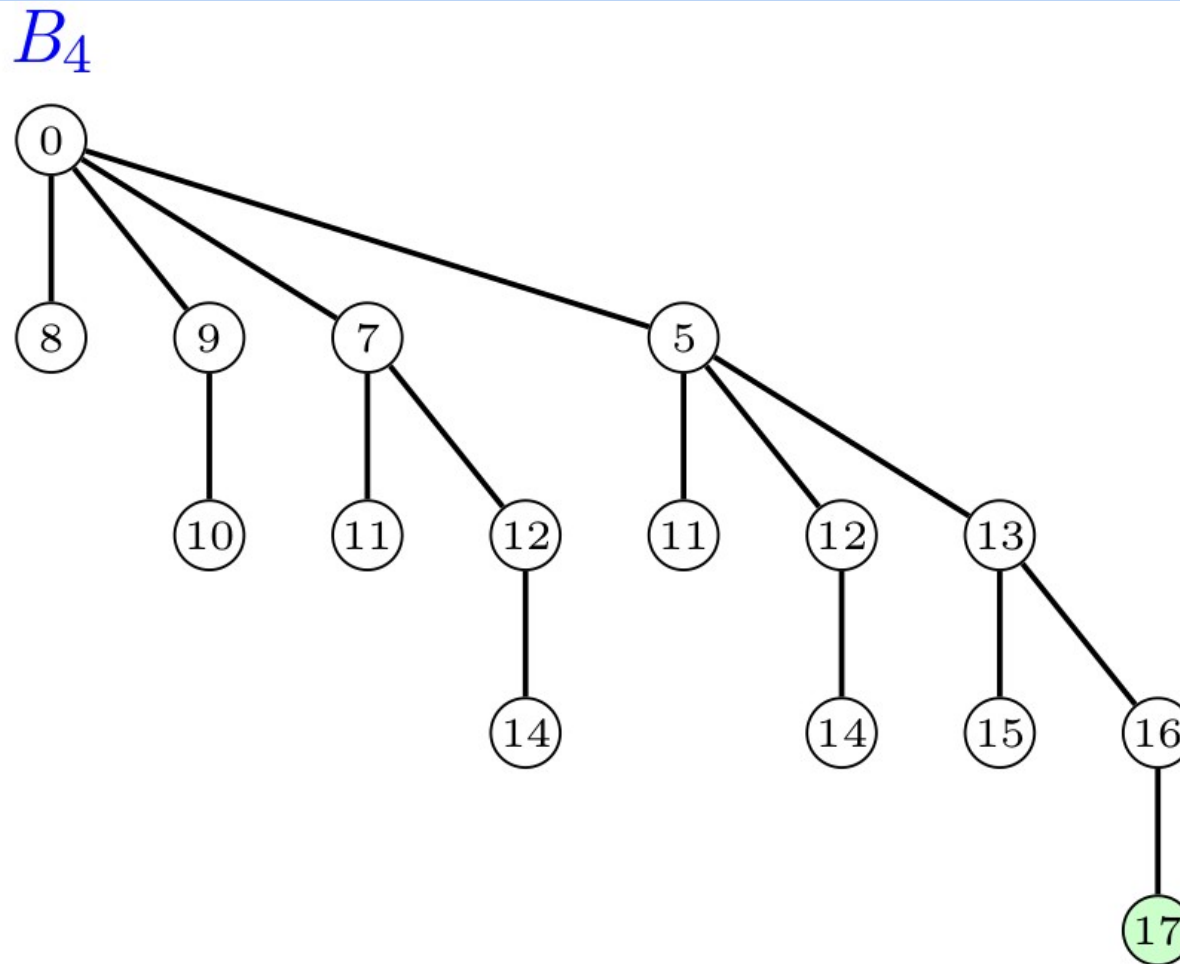


Figure 17: DECREASEKEY: 17 to 1

Percolation ou tamisage en  $O(n \log(n))$  pour rééquilibrer le tas min

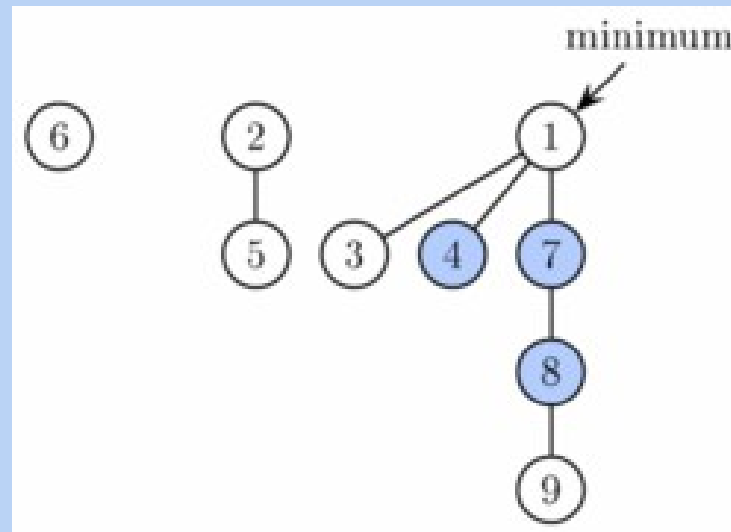
# Tas de Fibonacci

**Tas de Fibonacci** : est une sorte de tas binomial défini de façon moins stricte avec

- accès direct à la clé minimale
- pas de maintien systématique des propriétés de la structure (stratégie paresseuse)
- coût amorti  $O(1)$  pour la diminution d'une clé et l'union

En revanche, on peut montrer que tout arbre  $T$  d'un tas de Fibonacci, on a  $n \geq \phi^d$  où est le **nombre d'or** (voir suite de Fibonacci) et  $d$  le degré de la racine soit  $d = O(\log_{\phi} n) = O(\log n)$ .

Plus généralement, la taille (nombre de nœuds) d'un sous-arbre enraciné en un sommet de degré  $k$  est au plus  $F_{k+2}$  où  $F_k$  est le  $k^{\text{ième}}$  nombre de **Fibonacci**. Comme on se retrouve :-)



En informatique, un tas de Fibonacci est une structure de données similaire au tas binomial, mais avec un meilleur temps d'exécution amorti. Les tas de Fibonacci ont été conçus par Michael L. Fredman et Robert E. Tarjan en 1984 et publiés pour la première fois dans un journal scientifique en 1987. Les tas de Fibonacci sont utilisés pour améliorer le temps asymptotique de l'algorithme de Dijkstra, qui calcule les plus courts chemins dans un graphe, et de l'algorithme de Prim, qui calcule l'arbre couvrant de poids minimal d'un graphe.

Le nom de tas de Fibonacci vient des nombres de Fibonacci, qui sont utilisés pour calculer son temps d'exécution.

# Tas de Fibonacci

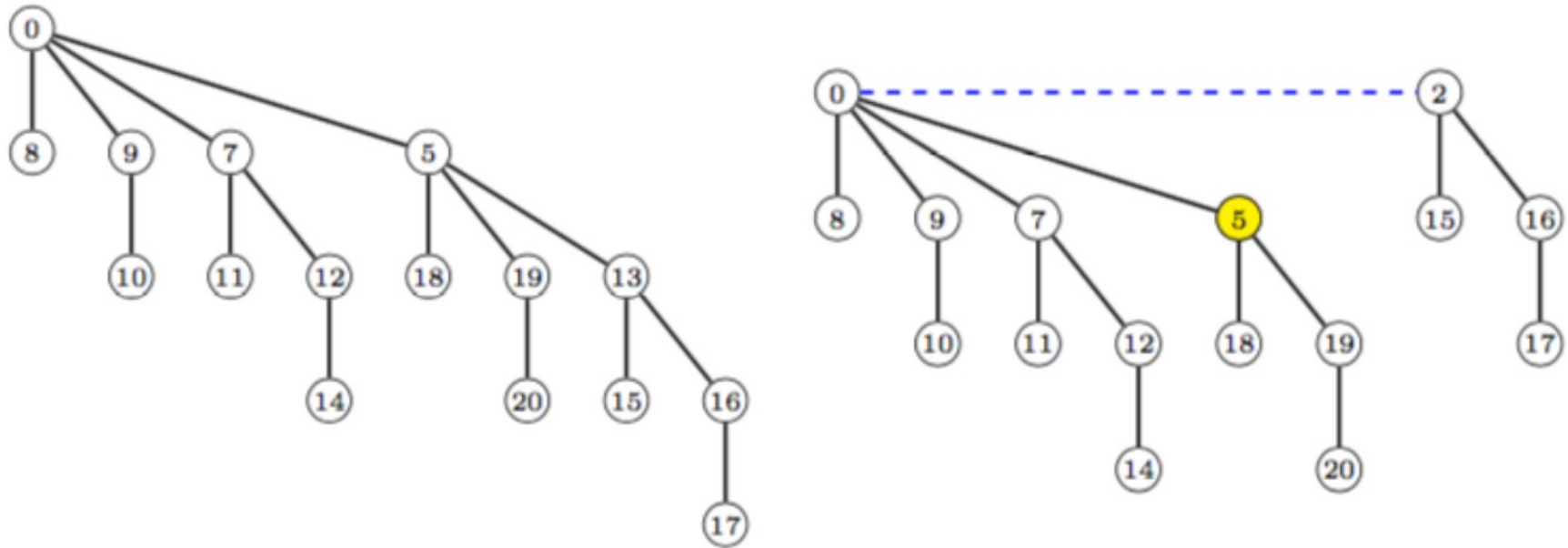


Figure 19: Heap order is violated when `DECREASEKEY` 13 to 2.

Typiquement, l'opération `DecreaseKey()` se fait en coupant une arête plutôt qu'en réarrangeant les nœuds par percolation (ou tamisage)

En informatique, un tas de Fibonacci est une structure de données similaire au tas binomial, mais avec un meilleur temps d'exécution amorti. Les tas de Fibonacci ont été conçus par Michael L. Fredman et Robert E. Tarjan en 1984 et publiés pour la première fois dans un journal scientifique en 1987. Les tas de Fibonacci sont utilisés pour améliorer le temps asymptotique de l'algorithme de Dijkstra, qui calcule les plus courts chemins dans un graphe, et de l'algorithme de Prim, qui calcule l'arbre couvrant de poids minimal d'un graphe

Le nom de tas de Fibonacci vient des nombres de Fibonacci, qui sont utilisés pour calculer son temps d'exécution.

[https://en.wikipedia.org/wiki/Fibonacci\\_heap](https://en.wikipedia.org/wiki/Fibonacci_heap)

# Tas de Fibonacci

Pour permettre la suppression et la concaténation d'arbres de façon efficace, on stocke les racines de tous les arbres dans une liste doublement chaînée. Les fils de chaque nœud sont conservés dans une liste chaînée. Pour chaque nœud, on conserve son nombre de fils et si le nœud est marqué ou non (ce marquage dépend du processus global de réorganisation des arbres lors de l'évolution du tas). On garde également un pointeur sur le minimum des racines.

**Tout ceci concourt à un coût amorti excellent et une complexité optimale pour Dijkstra ou Prim en  $O(m+n\log n)$ . Très technique : hors du cadre de ce cours en L3.**

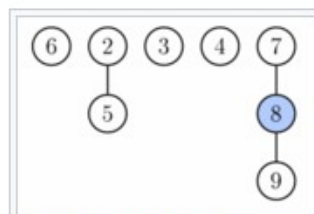


Figure 2. Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

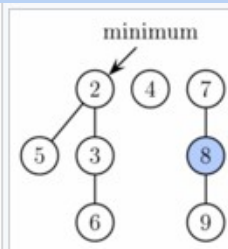


Figure 3. Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

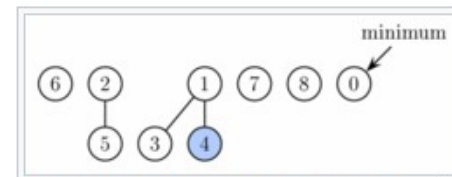


Figure 4. Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

En particulier, les opérations insertion, trouver le minimum, décroître une clé, et union ont toutes un coût amorti constant. Les opérations supprimer et supprimer le minimum ont un coût amorti en  $O(\log n)$ . C'est-à-dire qu'en partant d'une structure vide, n'importe quelle séquence de a opérations du premier groupe et b opérations du second groupe prendrait un temps  $O(a + (b \log n))$ . Dans un tas binomial, une telle séquence d'opérations prendrait un temps  $O((a + b)(\log n))$  (intéressant quand ordre  $\ll \ll$  taille graphe)

# Tas de Fibonacci

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap*	Fibonacci Heap*
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

\*amortized cost



# Tas de Fibonacci

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	$n$	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	$n$	$\log n$	$\log n$	$\log n$
UNION	1	$n$	$\log n$	1
FINDMIN	$n$	1	$\log n$	1
DIJKSTRA	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

DIJKSTRA algorithm:  $n$  INSERT,  $n$  EXTRACTMIN, and  $m$  DECREASEKEY.



# Equilibrage, Complexité, Tas

Ce cours est une ouverture sur les aspects plus calculatoires de la manipulation d'ensemble de données via des graphes/arbres. **Ce qu'il faut retenir en L3 :**

Les arbres sont des structures extrêmement riches d'un point de vue algorithmique permettant des opérations dynamiques en  $O(h)$  :

- d'extraction de valeur minimum, maximum,
- d'insertion, de suppression etc.

Pour un arbre, la taille c'est le nombre de sommets  $n$  (contrairement à un graphe pour lequel c'est l'ordre mais, comme  $n$  et  $m$  sont liés dans un arbre, taille et ordre sont les mêmes mesures ( $m=n-1$ )). La hauteur c'est  $h$ . Si l'arbre est équilibré,  $h=O(\log n)$

Si AVL,  $h$  est de l'ordre de  $1.44 \cdot \log_2(n)$  car la différence des hauteurs entre sous-arbres droit et gauche est au plus 1 pour tout sommet (un exemple les arbres de Fibonacci). Ce qui permet d'optimiser les opérations de recherche etc.

# Equilibrage, Complexité, Tas

Ce cours est une ouverture sur les aspects plus calculatoires de la manipulation d'ensemble de données via des graphes/arbres. **Ce qu'il faut retenir en L3 :**

La référence ultime pour l'optimisation, ce sont les tas de Fibonacci qui permettent de réduire la complexité de ses opérations de façon quasi optimale en coût amorti.

Au moins connaître leur existence et d'où viennent les valeurs et les noms : du **nombre d'or** :-)  
La file de priorité qui s'appuie sur les structures de tas peut notamment être utilisée comme structure d'optimisation des algorithmes où l'on doit extraire des arêtes par ordre de priorité.

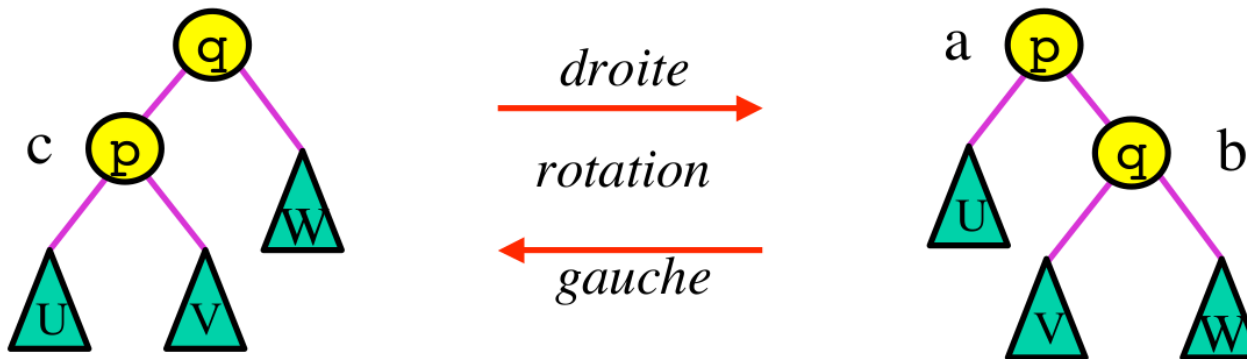
Les algorithmes suivants doivent leur efficacité à la structure de tas ou de *priority queue* disponible nativement en Java par exemple :

- Tri par tas binaire évidemment (heapsort)
- Algorithme de Dijkstra pour la recherche de plus court chemin dans un graphe ;
- Algorithme de Prim pour la recherche d'arbre couvrant de poids minimal dans un graphe.
- Algorithme A\* etc.

# Mise en Œuvre

Rotation Droite :  $((U,p,V),q,W) \rightarrow (U,p,(V,q,W))$

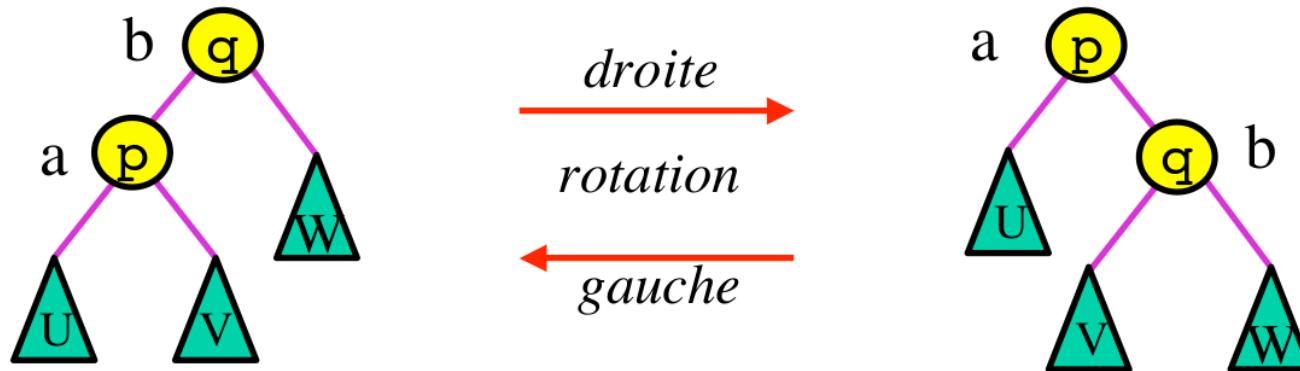
*Implantation (non destructive)*



```
static Arbre rotationG(Arbre a)
{
    Arbre b = a.filsD;
    Arbre c = new Arbre (a.filsG,
        a.contenu, b.filsG);
    return new Arbre (c, b.contenu,
        b.filsD);
}
```

# Mise en Œuvre

## *Implantation (destructive)*

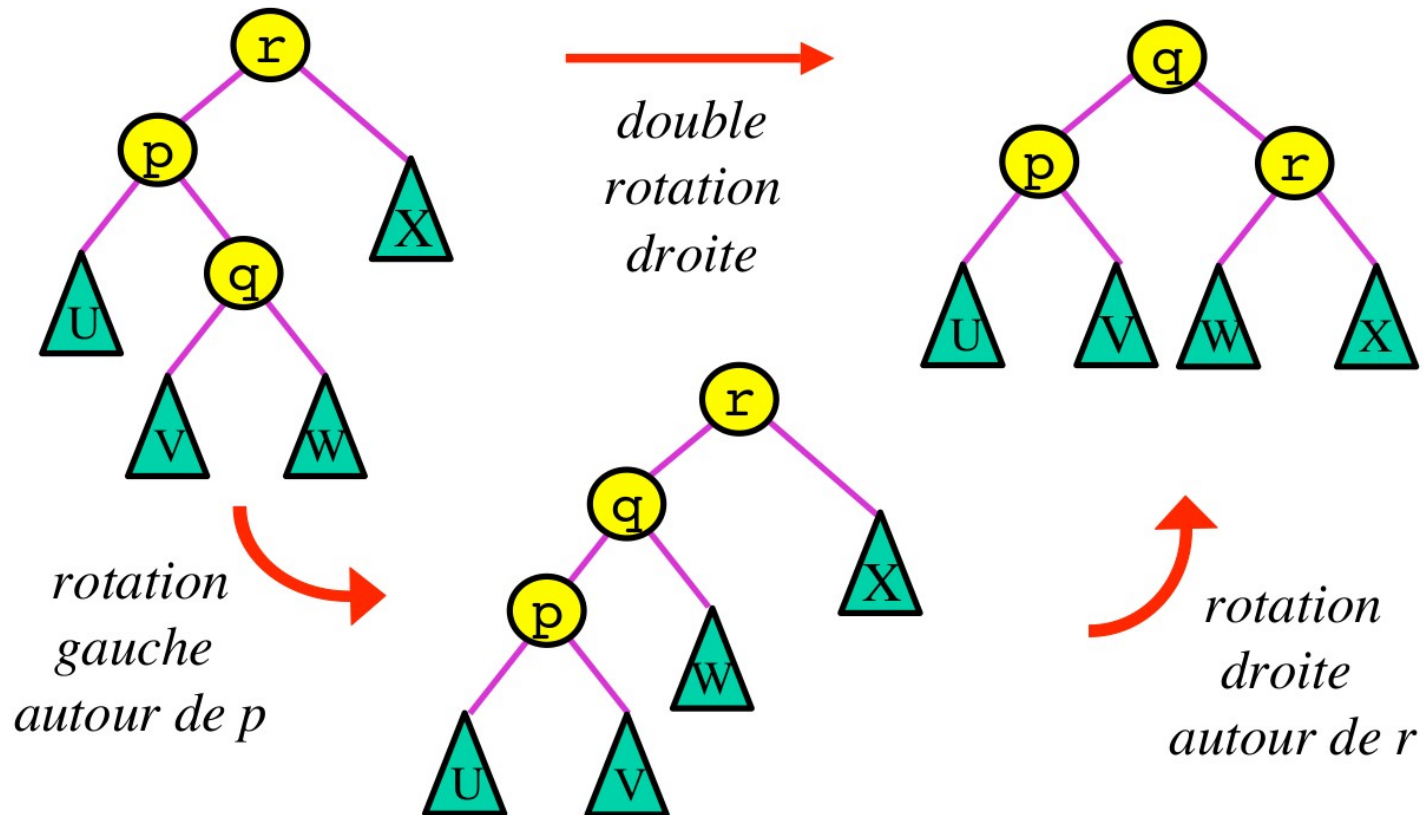


```
static Arbre rotationG(Arbre a)
{
    Arbre b = a.filsD;
    a.filsD = b.filsG;
    b.filsG = a;
    return b;
}
```

# Mise en Œuvre

Exercice reconstituer le code Java pour créer des arbres AVL avec le code Java proposé et le faire tourner sur un exemple.

## *Double rotation*



# Mise en Œuvre

Tuto AVL : <https://www.youtube.com/watch?v=rbg7Qf8GkQ4>

Tuto Binary Heap : [https://www.youtube.com/watch?v=uzqKs5t9\\_gk](https://www.youtube.com/watch?v=uzqKs5t9_gk)

Tuto Binomial Tree : <https://youtu.be/9Z5gPeuywKY>

Tuto Binomial Heap : [https://www.youtube.com/watch?v=e\\_gh1aD4v-A](https://www.youtube.com/watch?v=e_gh1aD4v-A)