# Arbres

→ Arbres (binaires)
→ Arbres binaires de recherche
→ Tas (tri en tas, files de priorité)

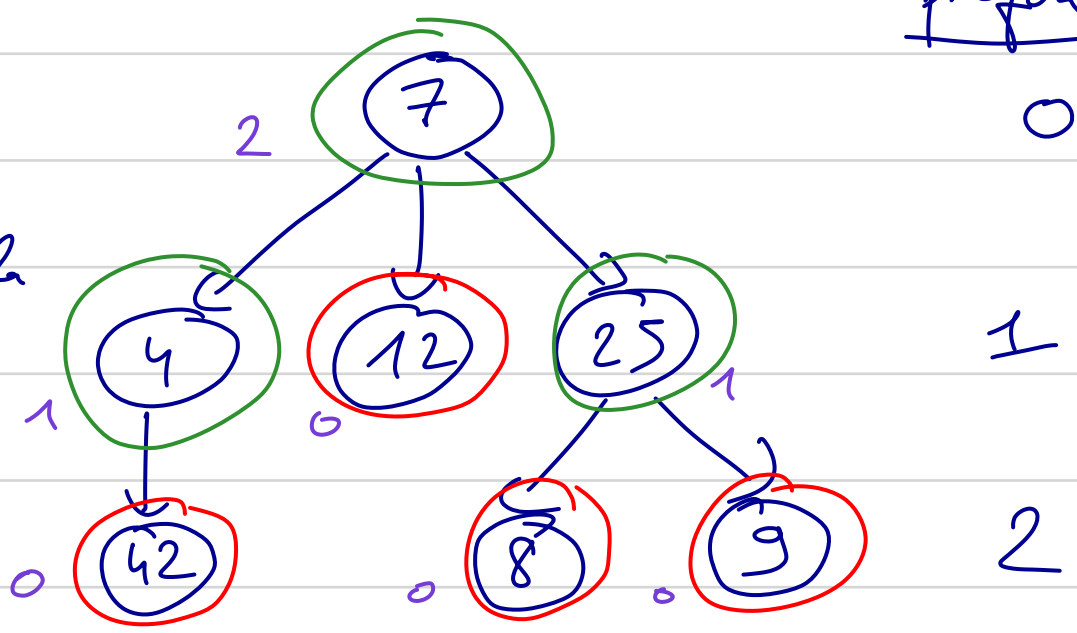
**Arbre** $\begin{cases} N \text{ ensemble fini de noeuds} \\ \text{enfants} : N \to \text{séquence fini d'éléments de } N \text{ deux à deux distincts} \\ \text{profondeur} : N \to \mathbb{N} \end{cases}$

t.q. :
- Il existe un seul noeud $r \in N$, la racine de l'arbre t.q. profondeur $(r) = 0$
- Chaque noeud $n \in N \setminus \{r\}$ a un unique parent $p(n) \in N$ tel que $n \in$ enfants $(p(n))$
- Si $p$ est le parent de $n$, alors profondeur $(p)$ = profondeur $(n) - 1$

**descendant** : clôture transitive de la relation enfant

**ancêtre** : clôture transitive de la relation parent

**hauteur**




Une **feuille** est un noeud sans enfant

Un **noeud interne** est un noeud avec $\geq 1$ enfant

La **profondeur** / **hauteur** d'un arbre est $\max_{n \in N}$ profondeur $(n)$

La **hauteur** d'un noeud est la profondeur de l'arbre formé de ce noeud et de ses descendants

Un arbre binaire est un arbre dans lequel chaque noeud interne a 1 ou 2 enfants.

Un arbre binaire complet est un arbre dans lequel chaque noeud interne a 2 enfants.

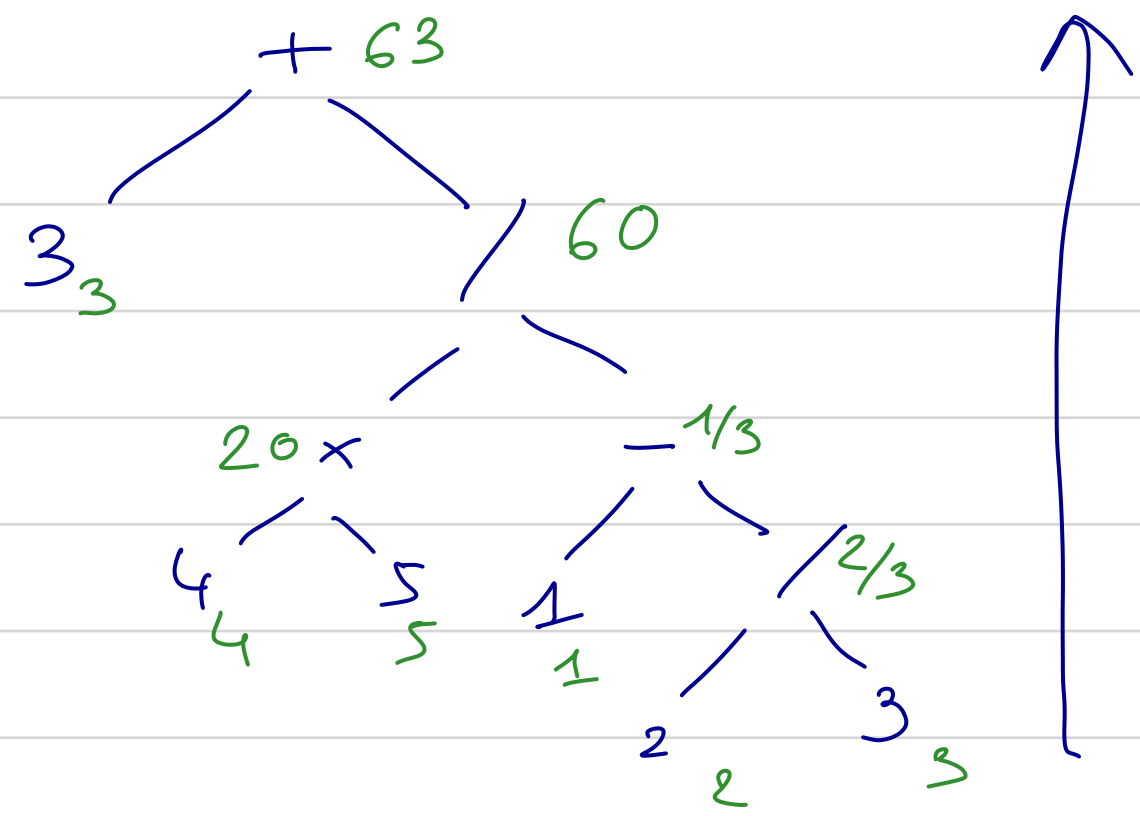
(On peut donc parler d'enfant gauche et d'enfant droit)

Application : syntaxe d'une expression arithmétique

$$3 + (4 \times 5 / (1 - 2/3))$$

Arbre de parsing, syntaxique :

Évaluation

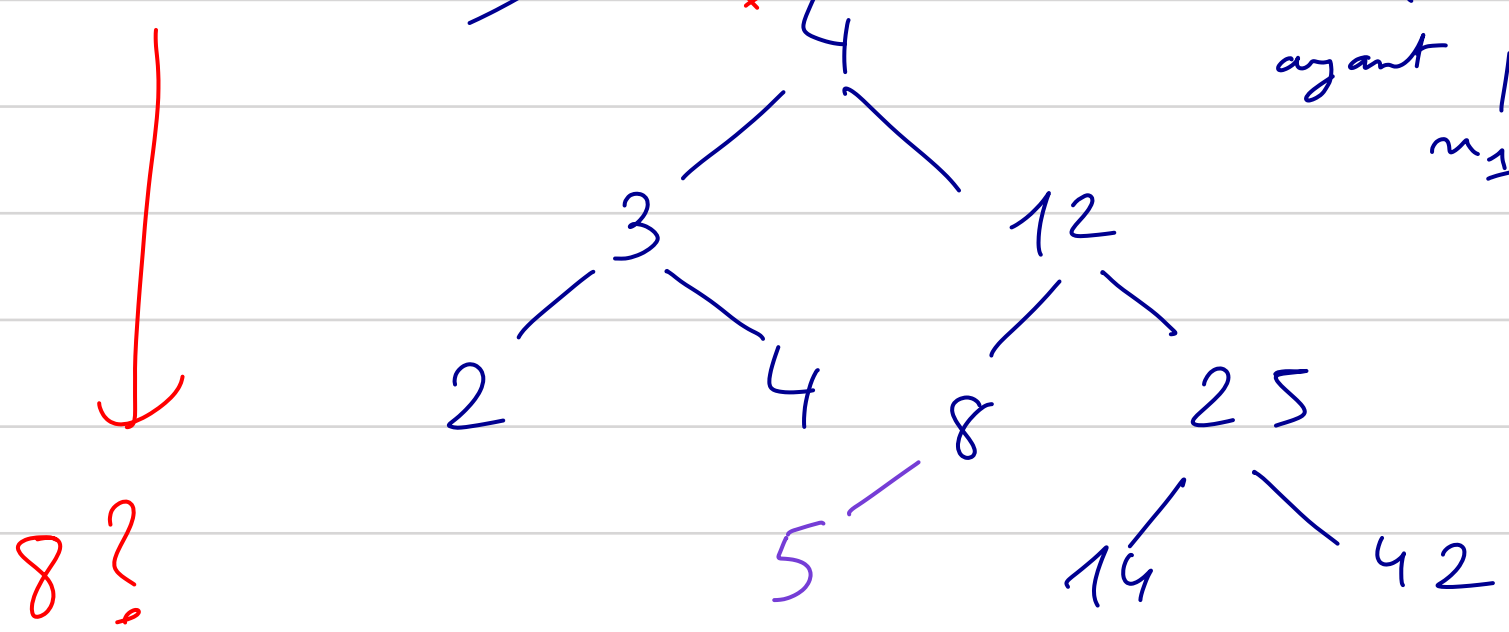


# Arbre binaire de recherche

- arbre binaire (complet)
- valeur associée à chaque noeud de l'arbre
  $v(n)$
- Si enfants $(n) = n_1, n_2$ alors

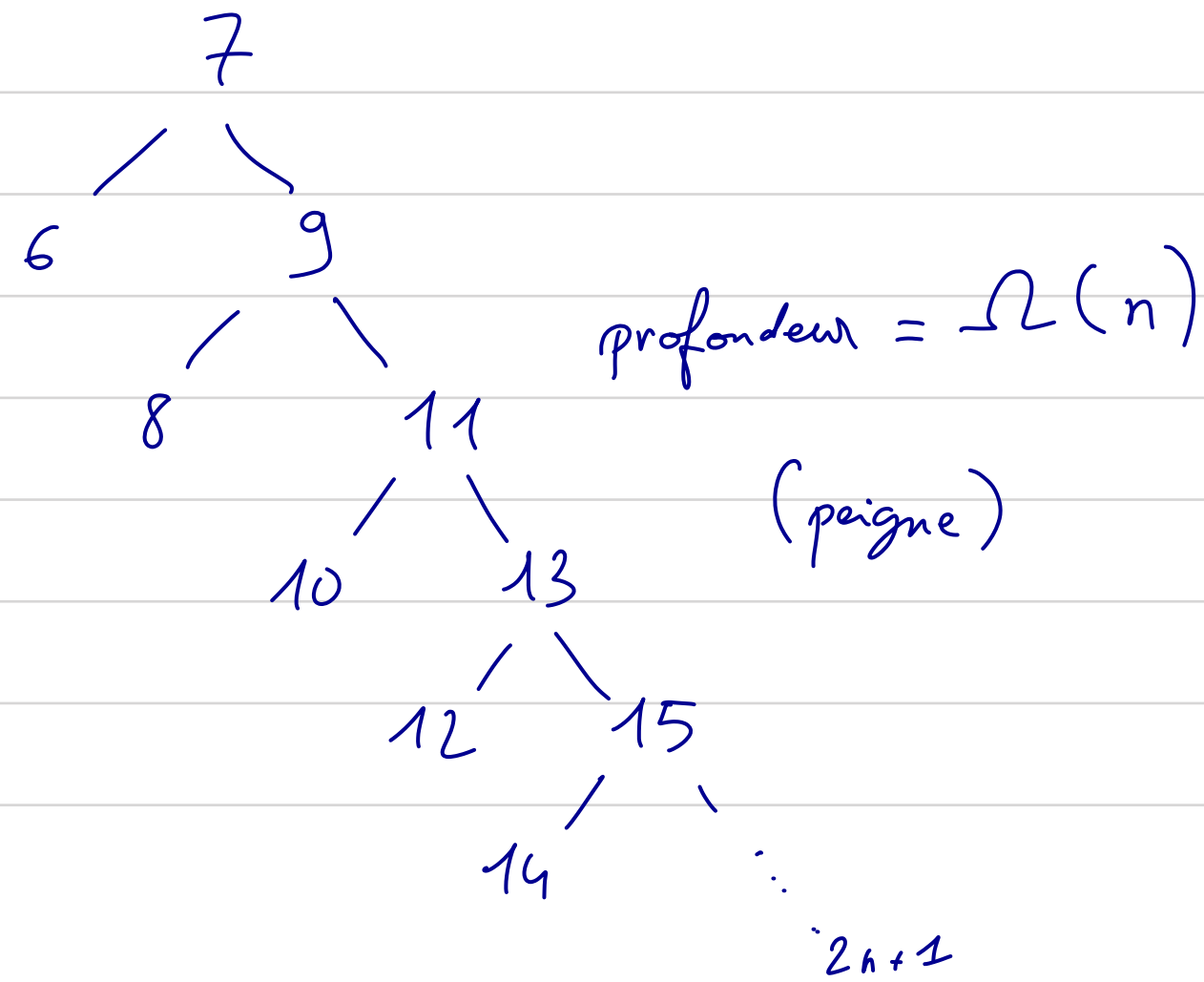
$$\left.\begin{matrix} v(n_1) \\ v(p_1) \end{matrix}\right\} \leq v(n) < \left.\begin{matrix} v(n_2) \\ v(p_2) \end{matrix}\right.$$

avec $p_1, p_2$ noeuds arbitraires du sous-arbre ayant pour racine $n_1, n_2$




Rechercher un élément dans un arbre binaire de recherche : $\Theta$(profondeur de l'arbre)

profondeur de l'arbre = $\Omega$ (log$_2$(nombre de noeuds))



On dit que l'arbre est <u>équilibré</u> si

profondeur = $O(\log(\text{nb de noeuds}))$

Complexité de la recherche dans un ABR :

→ $O(n)$ dans un ABR arbitraire

→ $O(\log n)$ dans un ABRE

Recherche, <u>Minimum</u>, <u>Maximum</u>, Insertion, Suppression,

↓ ↓ ↓ ↓ ↓

$O(h)$ $O(h)$ $O(h)$ $O(h)$ $O(h)$

Suppression / Insertion d'une valeur $v$ : <u>Successeur</u> → $O(h)$

- Rechercher la valeur
- Supprime / Ajoute la valeur à cet endroit (en s'assurant qu'on conserve une structure d'ABR)

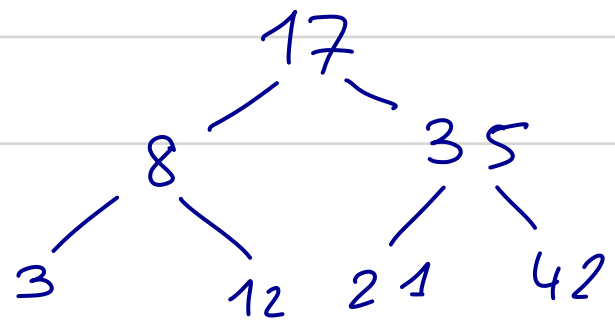
Construire un ABRE fixe à partir d'un ensemble de valeurs :

12 8 35 42 3 21 17

tri ($O(n \log n)$) ↳ 3 8 12 **17** 21 35 42

1/ Tri $O(n \log n)$

2/ Ajoute la valeur du milieu à la racine, puis je procède récursivement sur la moitié gauche et la moitié droite du tableau pour construire les sous-arbres gauche et droit $O(n)$



C++ → set (ABRE)
→ unordered_set (hachage)

Java → TreeSet (ABRE)
→ HashSet (hachage)

---

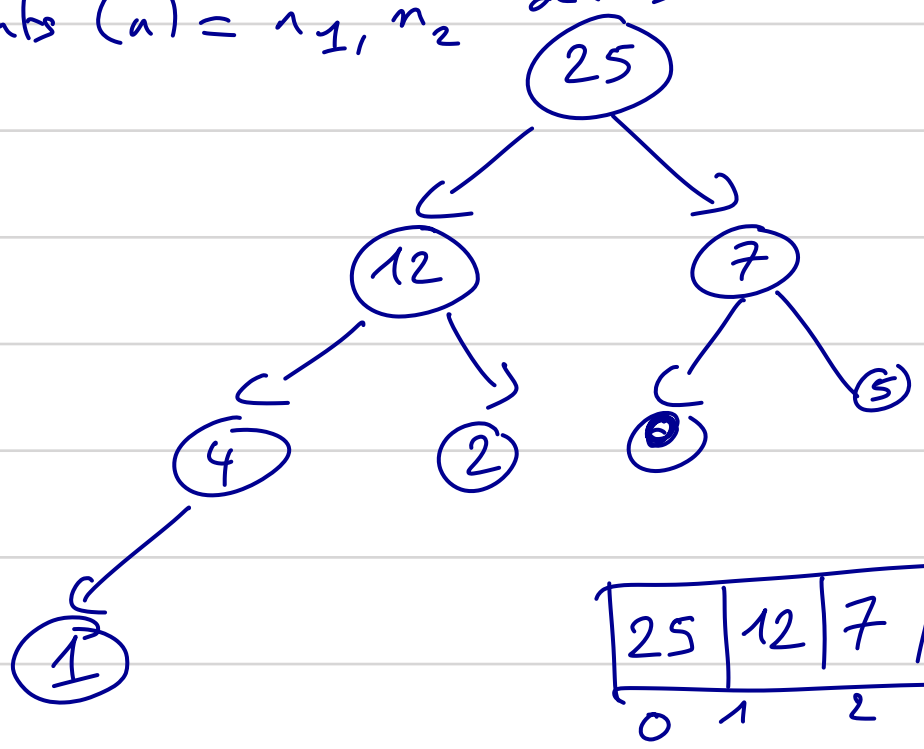
## Tas (heap)

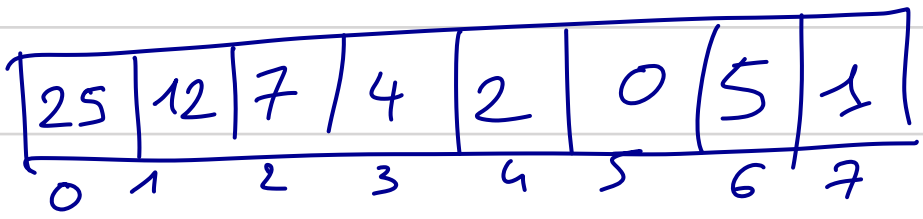
→ Arbre binaire (pas nécessairement complet) équilibré

→ Quasi-complet : tous les niveaux de l'arbre sont remplis, sauf le dernier où tous les noeuds sont le plus à gauche possible

→ Valeur $v(n)$ associée à chaque noeud $n$

→ si enfants $(n) = n_1, n_2$ alors $v(n) \geq v(n_1)$ $\geq v(n_2)$



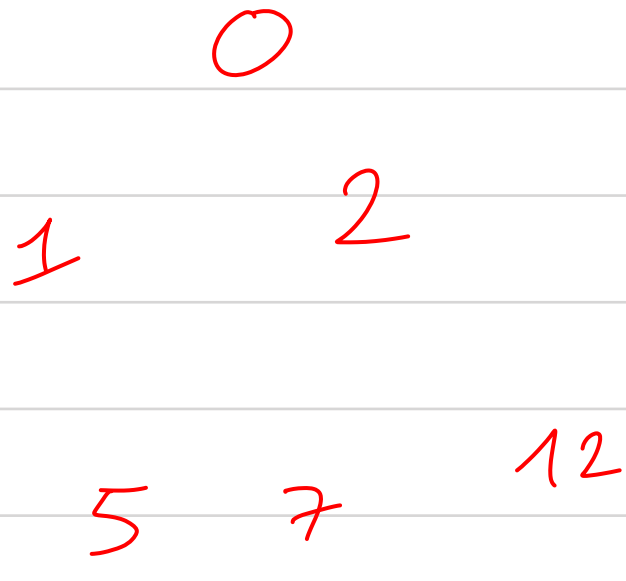

Un tas se représente par un tableau de taille fixe

$T = [t_0 | t_1 | .. | t_{n-1}]$

avec $t_0$ la valeur de la racine

et pour tout noeud interne dont la valeur est à la position $i$ :

- la valeur du fils gauche est à $2 \times i + 1$
- la valeur du fils droit est à $2 \times i + 2$
- $\lfloor (i-1)/2 \rfloor$ est la position de la valeur du parent




## Construire Tas

Pour chacun des noeuds (parcourus du dernier vers le premier), appliquer la méthode Tasser qui transforme le sous-arbre à ce noeud en un tas, en faisant descendre les valeurs plus petites que leurs enfants

$O(n \times \log n)$ en fait $O(n)$

## TriEnTas $O(n \log n)$

→ Construire Tas $O(n \log n)$ ou $O(n)$

→ Répétitivement (n fois)

$O(1)$ → Echanger la valeur à la racine avec le dernier élément du tableau

$O(1)$ → Indiquer que le dernier élément n'est plus dans le tas

$O(\log n)$ → Appeler tasser la racine

## Avantages de Tri en Tas

→ Tri en place (pas besoin de construire de nouvelles structures)

→ $O(n \log n)$ dans le pire des cas

En pratique, le tri rapide est en général plus rapide que le tri en tas.

---

Tas comme une file de priorité :

$O(\log n)$ amorti → Insérer un élément avec une valeur de priorité (deux étapes : insérer avec priorité $-\infty$ ; $O(1)$ amorti modifier la priorité)

$O(1)$ (racine) → Récupérer l'élément dont la valeur de priorité est maximale

$O(\log n)$ cf. tri en tas → Supprimer

$O(\log n)$ → Incrémenter la priorité

faire remonter l'élément en l'échangeant avec son parent si valeur plus grande que le parent

modifier la priorité