Arbres

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

Arbre

- N ensemble fini de noeuds
- enfants : $N \to$ séquence finie d'éléments de N deux à deux distincts
- profondeur : $N \to \mathbb{N}$

t.q. :

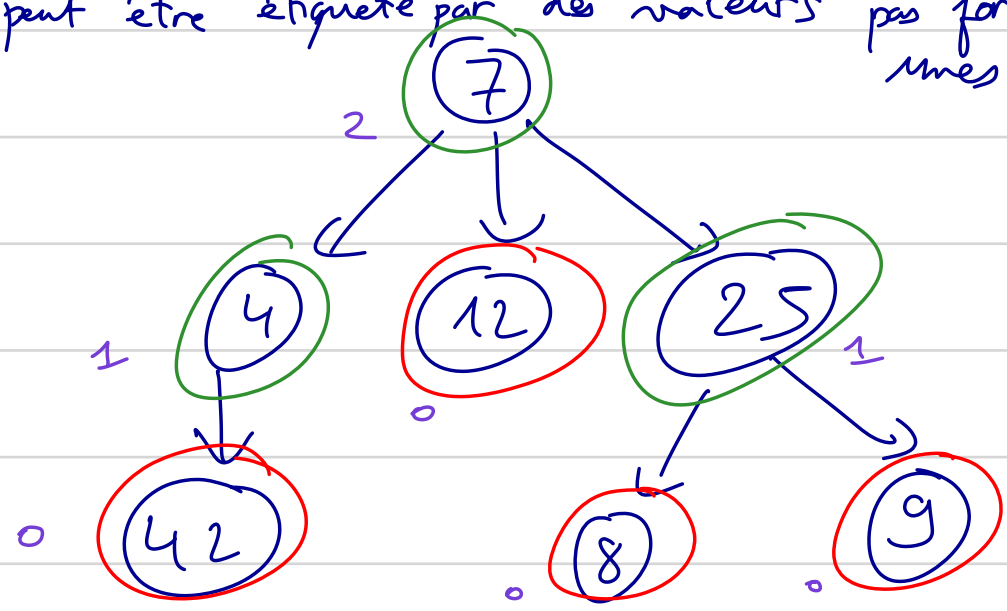
1) Il existe un seul noeud $r \in N$, la racine de l'arbre, t.q. profondeur $(r) = 0$.

2) Chaque noeud $n \in N \setminus \{r\}$ a un (unique) parent $p(n) \in N$ t.q. $n \in$ enfants $(p(n))$

3) Si p est le parent de n alors

profondeur (p) = profondeur $(n) - 1$.

(Un arbre peut être étiqueté par des valeurs pas forcément distinctes les unes des autres)



profondeur

Une feuille est un noeud sans enfant

Un noeud interne est un noeud avec ≥ 1 enfant.

Les descendants d'un noeud sont la clôture transitive de la relation enfant.

Les ancêtres d'un noeud sont ——— parent.

La 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 (sous-arbre)

Un arbre <u>binaire</u> est un arbre dans lequel chaque noeud a ≤ 2 enfants.

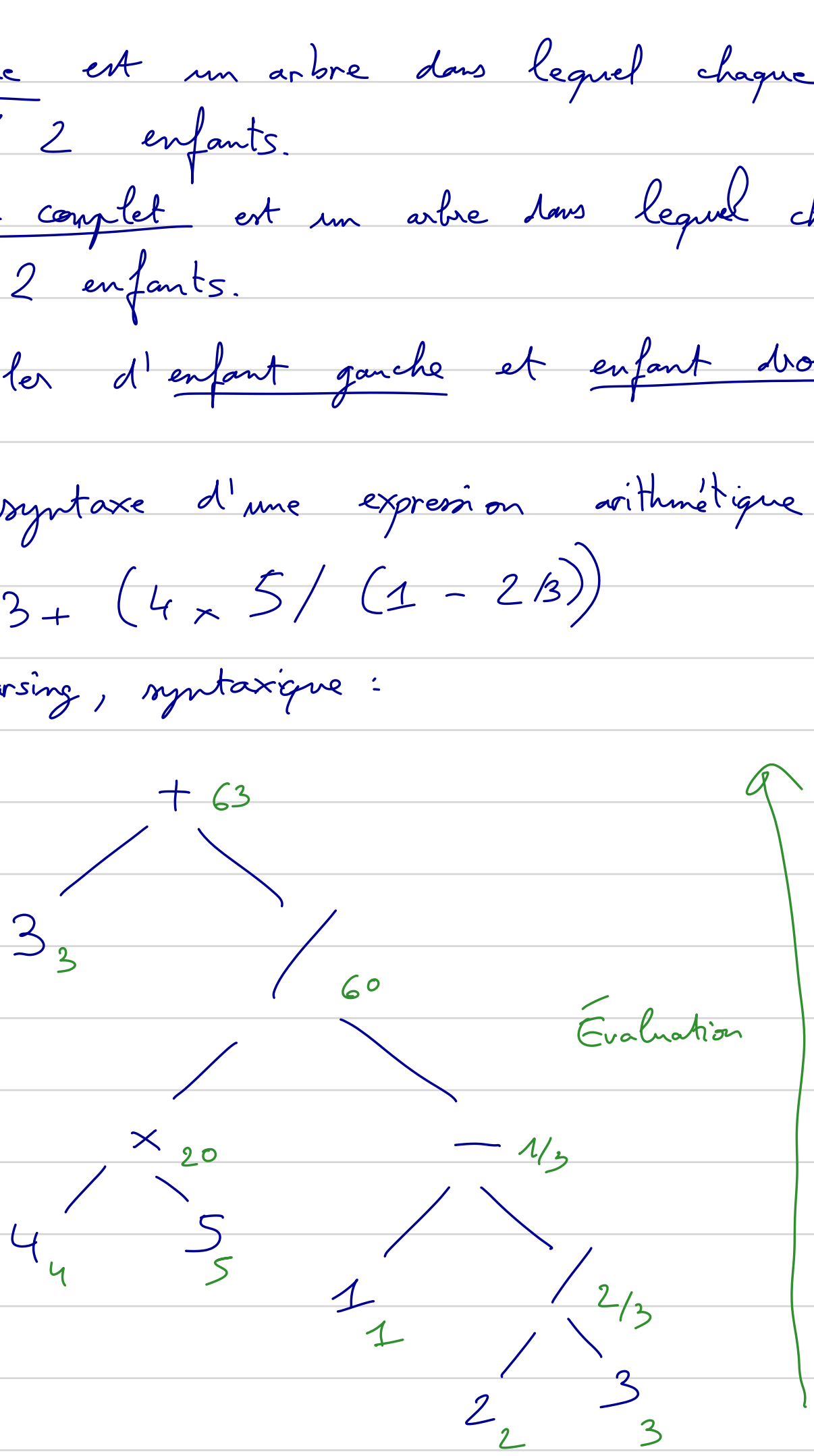
Un arbre <u>binaire complet</u> est un arbre dans lequel chaque noeud interne a 2 enfants.

↳ on peut parler d'<u>enfant gauche</u> et <u>enfant droit</u>

<u>Application</u> : syntaxe d'une expression arithmétique

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

Arbre de parsing, syntaxique :

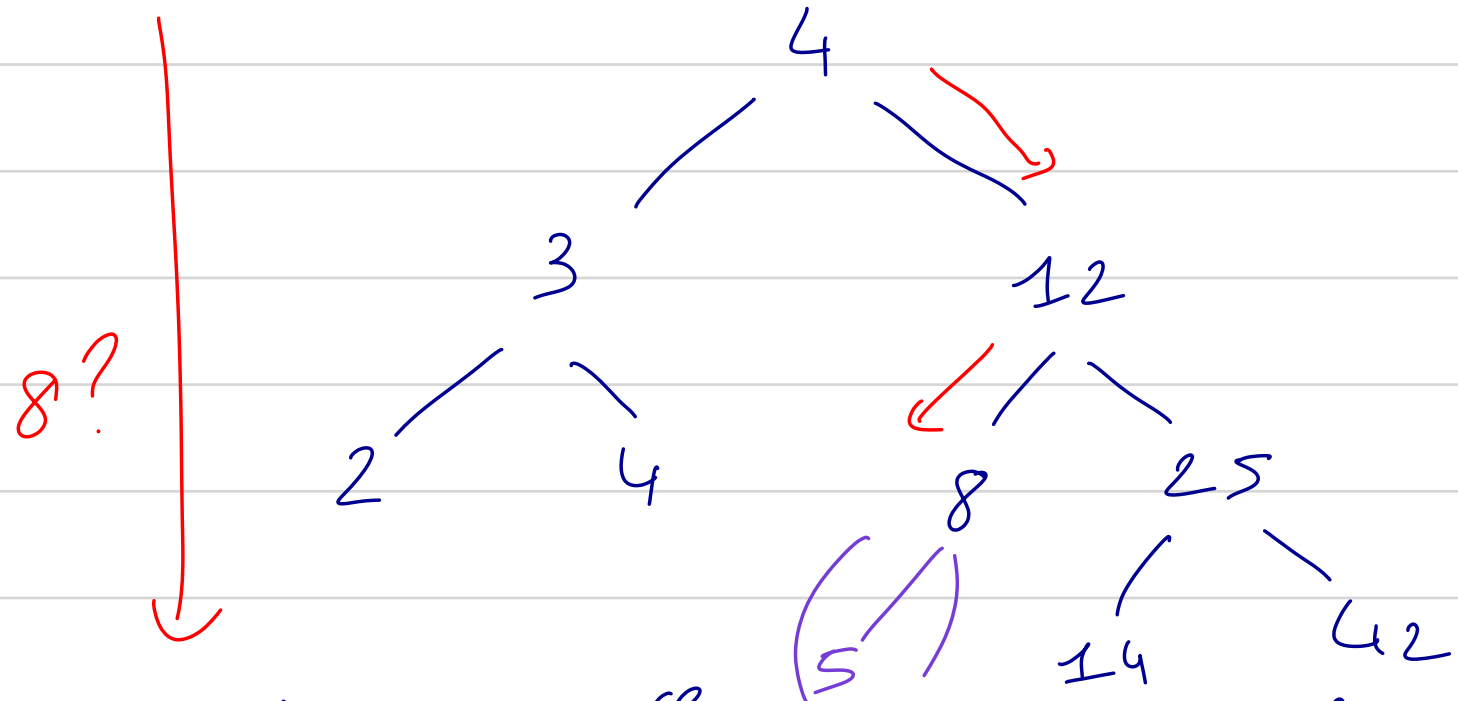


Arbre binaire de recherche

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

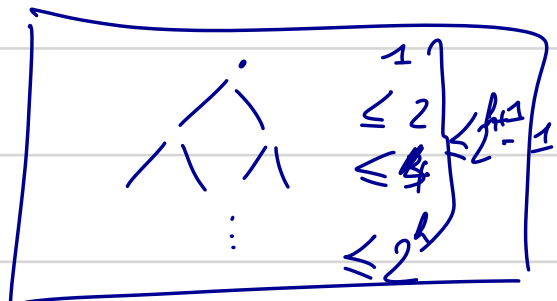
$$\left.\begin{matrix} v(n_1) \\ v(p_1) \end{matrix}\right\} \leqslant v(n) < \left\{\begin{matrix} v(n_2) \\ v(p_2) \end{matrix}\right.$$

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

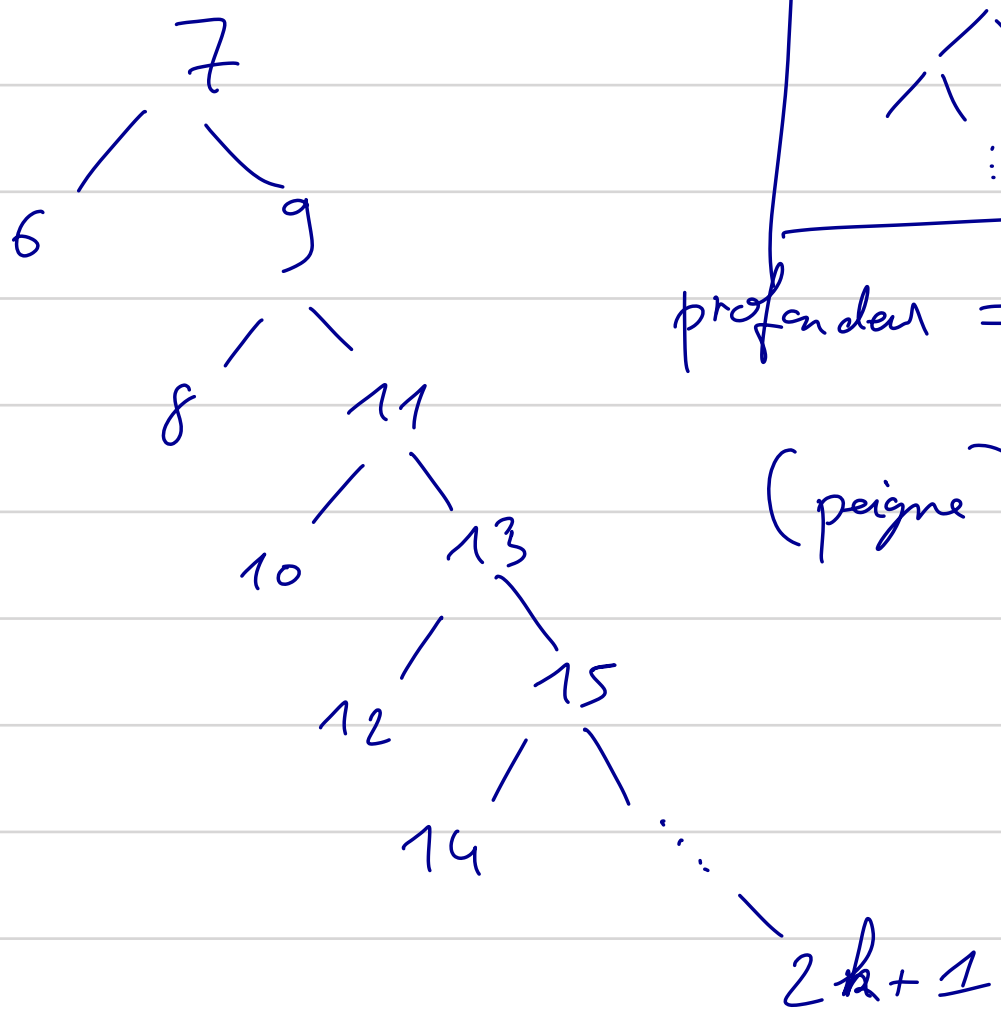


Rechercher si un élément est dans l'arbre :
- (14) (profondeur de l'arbre)
- (pour les éléments au plus bas de l'arbre)

profondeur de l'arbre $= \Omega(\log_2(\text{nombre de noeuds}))$



profondeur $= \Omega(n)$



(peigne)

On dit qu'un ABR est <u>équilibré</u> (ABRE) si

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

<u>Complexité de la recherche dans un ABR : $O(h)$</u>

→ $O(n)$ dans un ABR arbitraire

→ $\Theta(\log n)$ dans un ABRE

Recherche, Minimum, Maximum, Insertion, Suppression, Successeur

↓ ↓ ↓ ↓ ↓ ↓

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

$O(h)$ possible, mais pas très intéressant parce qu'on peut transformer un ABRE en un peigne

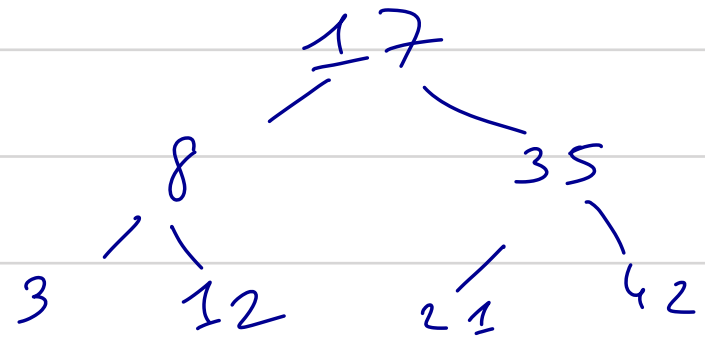
Construire un ABRE <u>fixé</u> à partir d'un ensemble de valeurs :

12 8 35 42 3 21 17

Tri ↳ 3 8 12 **17** 21 35 42

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

$\Theta(n)$ 2/ Ajouter la valeur du milieu à la racine de l'arbre et procéder récursivement sur les moitiés gauche et droite pour construire les sous-arbres gauche et droit.



C++ → set (ABRE)

→ unordered_set (hachage)

Java → TreeSet (ABRE)

→ HashSet (hachage)

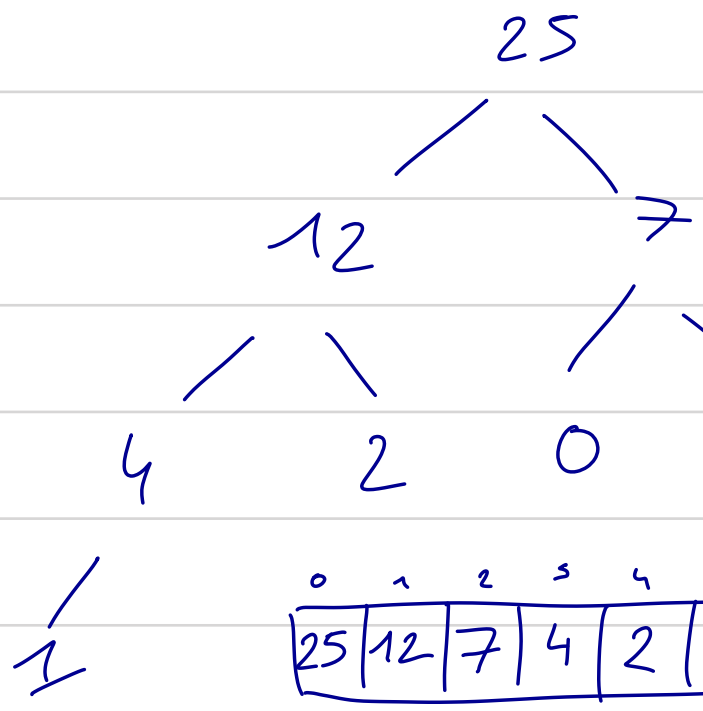
Tas (heap) : pas un ABR

→ Arbre binaire (pas nécessairement complet) équilibré

→ Quasi-complet : tous les niveaux de l'arbre sont remplis sauf le dernier (prof. maximale) 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 $v(n) \geq v(n_1)$
$\geq v(n_2)$



niveau 0 rempli

1 ——

2 ——

3 noeuds les plus à gauche possibles

Un tas se représente par un tableau de taille fixe

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

avec t_0 valeur de racine

et pour tout noeud interne dont la valeur est à la position i : } la valeur de l'enfant gauche est à $2 \times i + 1$
——— droit —— $2 \times i + 2$
$\lfloor \frac{(i-1)}{2} \rfloor$ } est la position de la valeur du parent

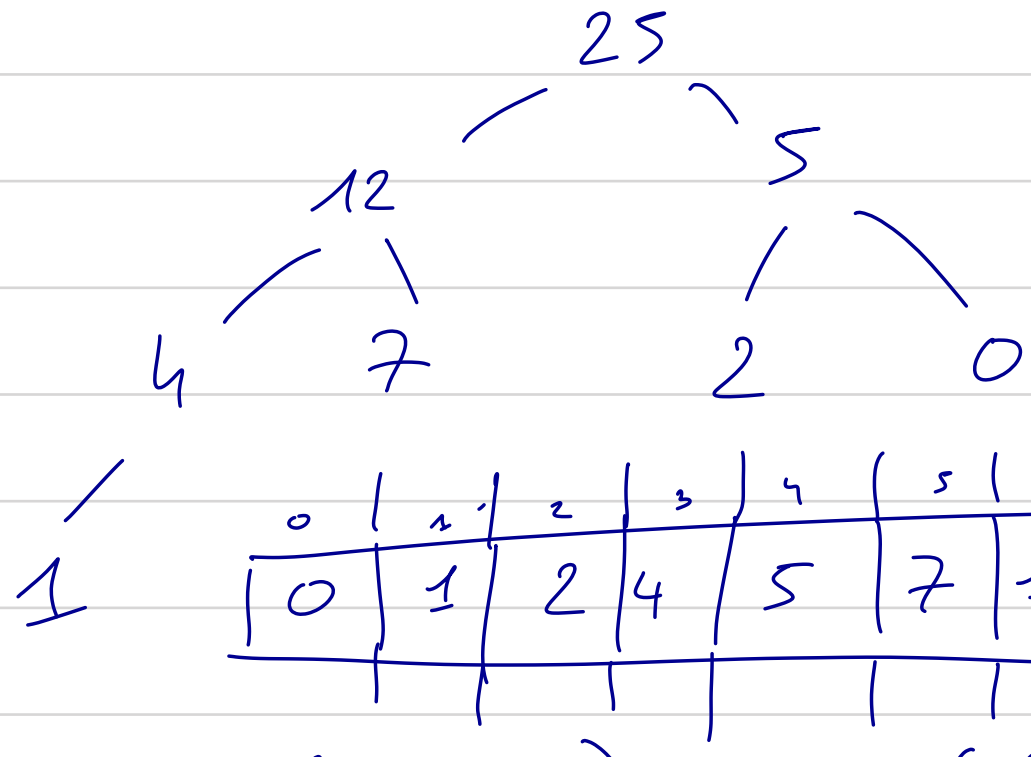
Construire un tas

| 7 | 12 | 5 | 4 | 25 | 2 | 0 | 1 |

Pour chacun des noeuds (parcours 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

Complexité : $O(n \times \log n)$ (en fait $\Theta(n)$, voir le livre de référence)

7	12	5	4	25	2	0	1



Tri En Tas (Heap Sort) ⇝ $O(n \log n)$

→ ConstruireTas $O(n \log n)$ (ou ici en $\Theta(n)$)

→ Répétitivement ($n-1$ fois)

- $O(1)$ → Échanger la valeur à la racine avec le dernier élément du tableau
- $O(1)$ → Indiquer que le dernier élément du tableau n'est plus dans le tas
- $O(\log n)$ → Applique Tasser à la racine

Avantages de TriEnTas

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

→ $O(n \log n)$ dans le pire des cas

Tas comme une file de priorité

- → Insérer un élément avec une valeur de priorité — $O(\log n)$ amorti
  - * Insère en bas du tas (priorité $-\infty$) — $O(1)$ amorti
  - * Augmente Priorité — $O(\log n)$
- → Récupérer l'élément dont la valeur de priorité est maximale [Racine] — $O(1)$
- → Supprimer — $O(\log n)$
  - Echange Racine et dernier élément — $O(1)$
  - Applique Tasser sur la nouvelle racine $O(\log n)$ (cf. TriEnTas)

AugmenteePriorité

$O(\log n)$ Faire remonter l'élément en l'échangeant avec son parent si nouvelle valeur plus grande que celle du parent.

Ex. : Insérer 20