




Chap 10 · Tas

Prise de note par Léo BERNARD en MP2I au Lycée du Parc.
Année 2022-2023



Pitié, autre chose que les arbres

-  Chap 10 · Tas
- I. File de priorité
- II. Tas binaire
- III. Tas impératif à l'aide d'un arbre implicite
- IV. Opérations sur les tas
 - 4.1. Lecture du minimum
 - 4.2. Insertion
 - 4.3. Extraction du minimum
 - 4.4. Entassement · Heapify
- V. Tri par tas (heapsort)

I. File de priorité

Une **file de priorité** est une structure de données contenant des couples (valeur, priorité). Les **valeur** peuvent être d'un type quelconque, mais les **priorité** doivent être choisies dans un type totalement ordonné (typiquement, des entiers ou des flottants).

Deux opérations fondamentales :

- **ajouter un élément** : (valeur, priorité)
- **extraire le « prochain élément »** : celui de priorité le plus faible (*par convention*)

 Exemples :

- la salle d'attente du service des urgences
- l'ordonnancement des personnes pour les processeurs.



Signature possible pour une implémentation impérative

```

val insert      : 'p -> 'v -> ('p, 'v) pq -> unit
val get_min     : ('p, 'v) -> 'p * 'v (*lecture du min (sans modif.)*
val extract_min : ('p, 'v) pq -> 'p * 'v (*avec effet de bord*)
val create      : unit -> ('p, 'v) pq
val cardinal    : ('p, 'v) pq -> int

```

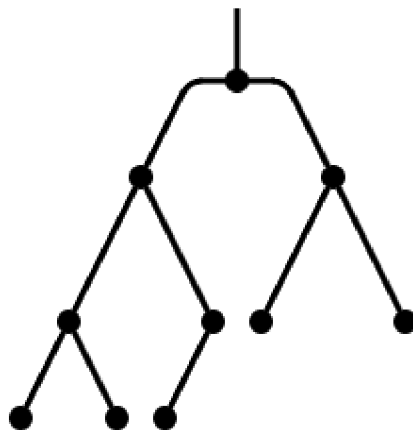
II. Tas binaire

Définition :

Un **arbre binaire complet gauche** (ABCG) de hauteur h est un arbre binaire qui vérifie :

- Tous les niveaux, sauf éventuellement le dernier, sont complets, i.e. pour $i \in \llbracket 0, k \rrbracket$, il y a exactement 2^i nœuds à profondeur i pour $i \in \llbracket 0, k \rrbracket$.
- Le dernier niveau est rempli de gauche à droite.

Exemple :



Remarque :

Tous les nœuds internes d'un ABCG possède un fils gauche.
Tous sauf éventuellement un possède un fils droit.

Remarque :

Les feuilles d'un ABCG de hauteur h sont à profondeur h ou $h - 1$.

 Proposition : Soit un ABCG de hauteur h et de taille n , on a ;

- $2^k \leq n < 2^{k+1}$
- $k = \lfloor \log_2 n \rfloor$

Démonstration :

On note n_k le nombre de nœuds à la profondeur k .

$$n_k = \begin{cases} 2^k & \text{si } k \in \llbracket 0, k \rrbracket \\ \text{entre 1 et } 2^k & \text{si } k = h \end{cases}$$

Donc :

$$1 + \sum_{k=0}^{h-1} 2^k \leq n = \sum_{k=0}^{h-1} n_k \leq \sum_{k=0}^{h-1} 2^k$$

Soit :


$$2^h \leq n \leq 2^{h+1} - 1$$

Définition :

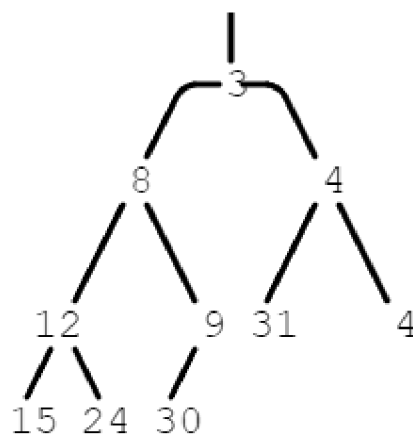
Soit T un arbre dont les nœuds sont étiquetés par les éléments d'un ensemble totalement ordonné.

- On dit que T possède la propriété des **tas-min** si l'étiquette d'un nœud est toujours inférieure ou égale à celle de ses éventuels enfants. (*On appelle également ceci un arbre tournoi min*)
- On dit que T est un **tas** (min) (binaire) si, de plus, c'est un arbre binaire complet gauche.

Note : en anglais, le tas s'appelle *heap*.

 **Remarque :** Cette définition du tas n'a pas de rapport avec le tas de la mémoire, c'est une coïncidence.

Exemple :



Remarque :

- On ne sait rien sur l'étiquette du fils gauche par rapport à l'étiquette du fils droit.
- La propriété d'ordre des tas est locale.
- Un tas max est la même chose avec des inégalités renversées.

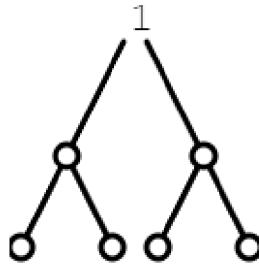
💡 **Proposition :**

- Si a est un ancêtre de b , alors $a \leq b$.
- La racine contient un élément de clé minimale.

📖 **Exercice :** Combien de tas min peut-on construire avec un ensemble de $2^n - 1$ clés deux à deux distinctes ?

On essaie avec 7 éléments.

On note ces éléments 1, 2, 3, 4, 5, 6, 7.



- $\binom{6}{3} \rightarrow$ à gauche $\rightarrow c_2$ manières de le placer
- 1 \rightarrow à droite $\rightarrow c_2$ manières de le placer

$$c_3 = \binom{6}{3} c_2^2$$

On note c_n le nombre de tas min.

- le minimum est forcément à la racine :
il reste $2^n - 2$ clés à répartir à gauche et à droite, sans contrainte.
- Cette répartition va pouvoir se faire de $\binom{2^n - 2}{2^{n-1} - 1}$ manières.
- Chacun des sous-arbres peut se construire de c_{n-1} manières différentes.

$$c_n = \binom{2^n - 2}{2^{n-1} - 1} c_{n-1}^2$$

$$c_n = \prod_{k=0}^{n-1} \binom{2^{n-k} - 2}{2^{n-k-1} - 1}^{2^k} \times \underbrace{c_0^{2^n}}_{=1}$$

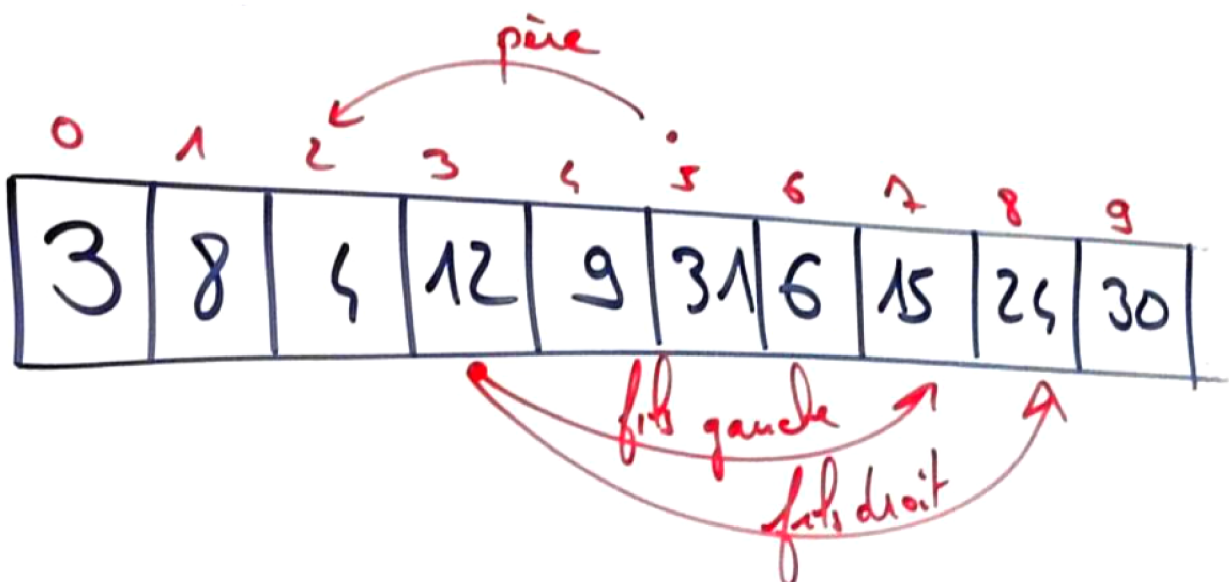
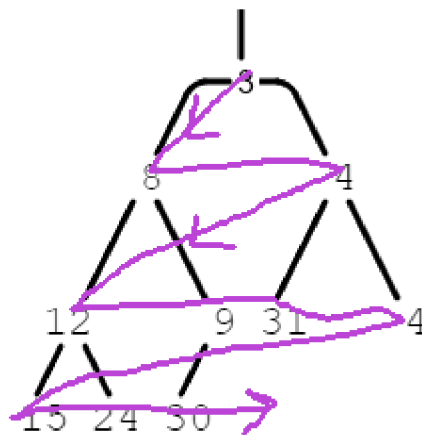
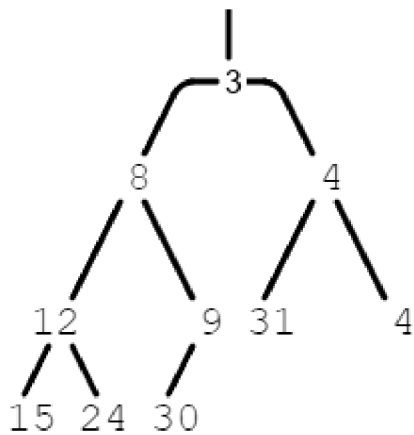
III. Tas impératif à l'aide d'un arbre implicite

(Implicite = indirect, caché)

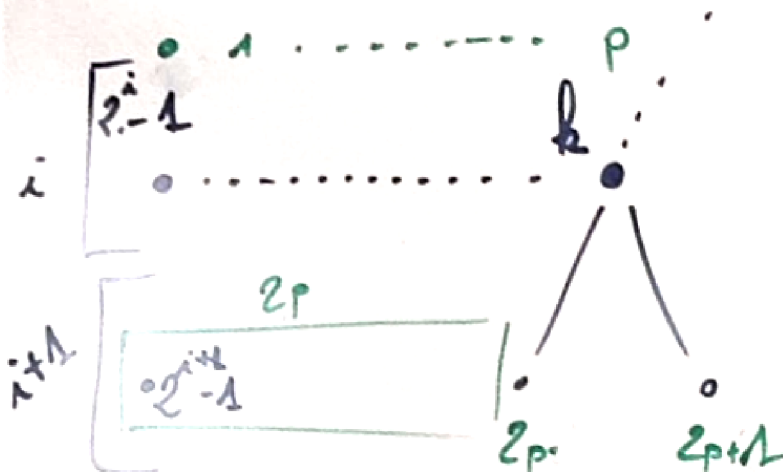
💡 **Théorème :** On peut représenter un arbre binaire complet gauche ayant n nœuds dans un tableau de largeur n , en énumérant ses éléments dans l'ordre du parcours en largeur. Cette représentation est bijective. On a les propriétés suivantes :

- La racine est à l'indice 0.
- Les nœuds de profondeur i sont stockés consécutivement à partir de l'indice $2^i - 1$.
- Les fils gauche et droit du nœud d'indice k sont respectivement aux indices $2k + 1$ et $2k + 2$.
- Le père d'un nœud d'indice $k > 0$ a pour indice $\left\lfloor \frac{k-1}{2} \right\rfloor$, soit $(k-1) \gg 1$.

📁 **Exemple :**



$$k = 2^i - 1 + p$$



Fils gauche : $2^{i+1} - 1 + 2p = 2(2^i - 1 + p) + 1$

Fils droit : $2^{i+1} - 1 + 2p + 1 = 2k + 2$

Dans l'autre sens :

$$\left\lfloor \frac{(2k + 1) - 1}{2} \right\rfloor = \left\lfloor \frac{(2k + 2) - 1}{2} \right\rfloor = k$$

En pratique, les opérations intéressantes sur un tas font varier sa taille. La largeur du tableau ne sera donc pas en général égale au nombre d'éléments du tas (il faudra stocker ce nombre d'éléments).

```
struct heap {
    int size;
    int capacity;
    datatype* data
}
```

👉 **Remarque :** C'est la même structure que pour les tableaux dynamiques : on pourra redimensionner le tableau data si besoin.

🐪 En OCaml :

```
type 'a heap = {
    mutable size: int;
    mutable data : 'a array
    (*mutable si redimensionnement*)
}
```

📁 **Quelques fonctions utilitaires**

```
// Rappel
struct heap {
    int size;
    int capacity;
    datatype* data
}

heap* heap_new(int capacity);
void heap_delete(heap* h);
void swap(datatype* data, int i, int j);
int up(int k);
int left(int k);
int right(int k);
```

```
heap* heap_new(int capacity) {
    heap* h = malloc(sizeof(heap));
    h->size = 0;
    h->capacity = capacity;
    h->data = malloc(capacity * sizeof(datatype));
    return h;
}

void heap_delete(heap* h) {
    free(h->data);
    free(h);
}

void swap(datatype* data, int i, int j) {
    datatype tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}

int up(int k) {
    return (k-1)/2;
}

int left(int k) {
    return 2*k + 1;
}

int right(int k) {
    return 2*k + 2;
}
```

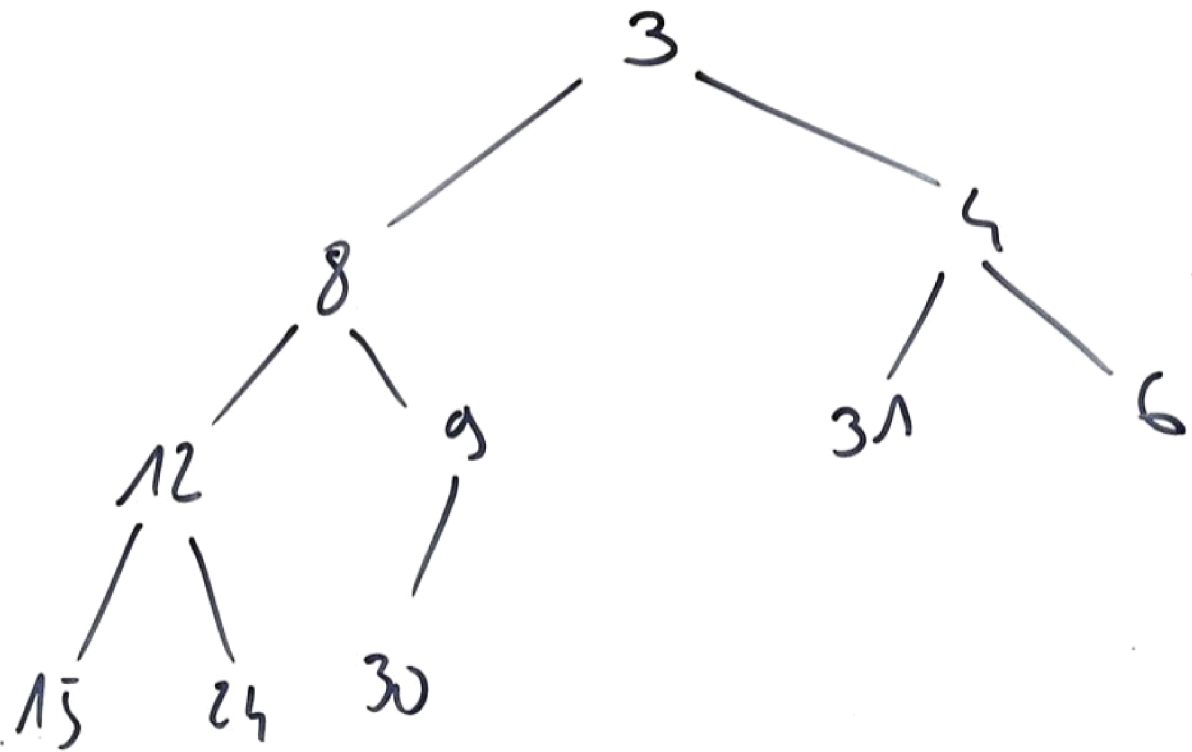
IV. Opérations sur les tas

4.1. Lecture du minimum

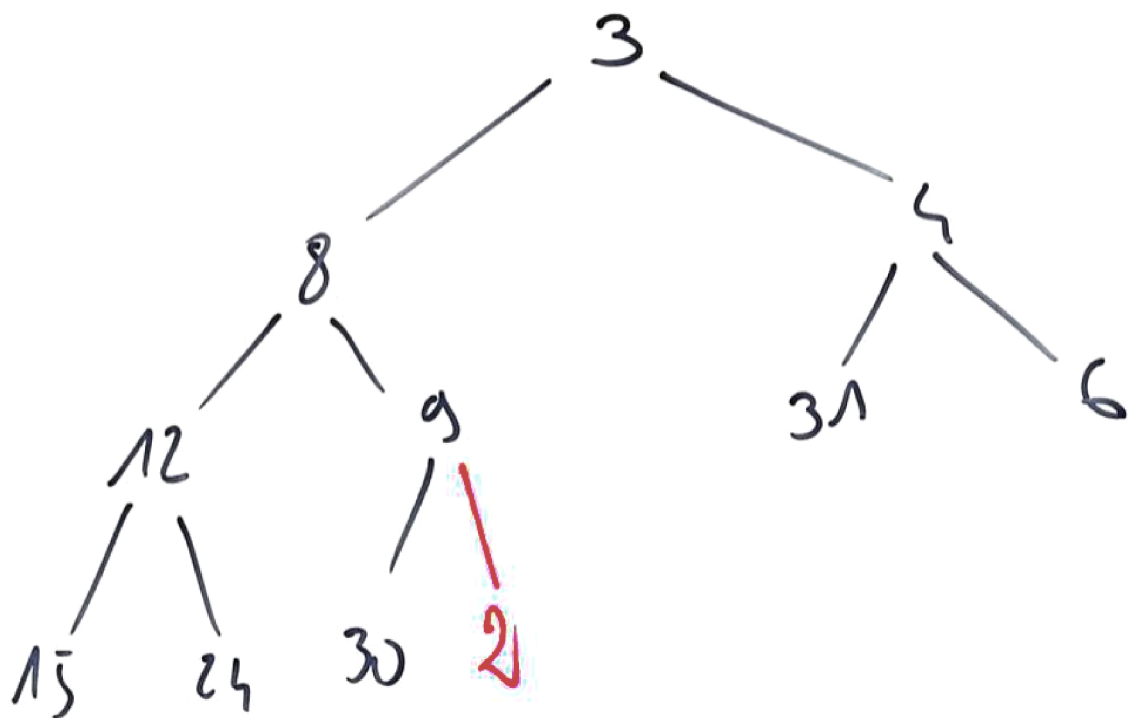
ez: racine

4.2. Insertion

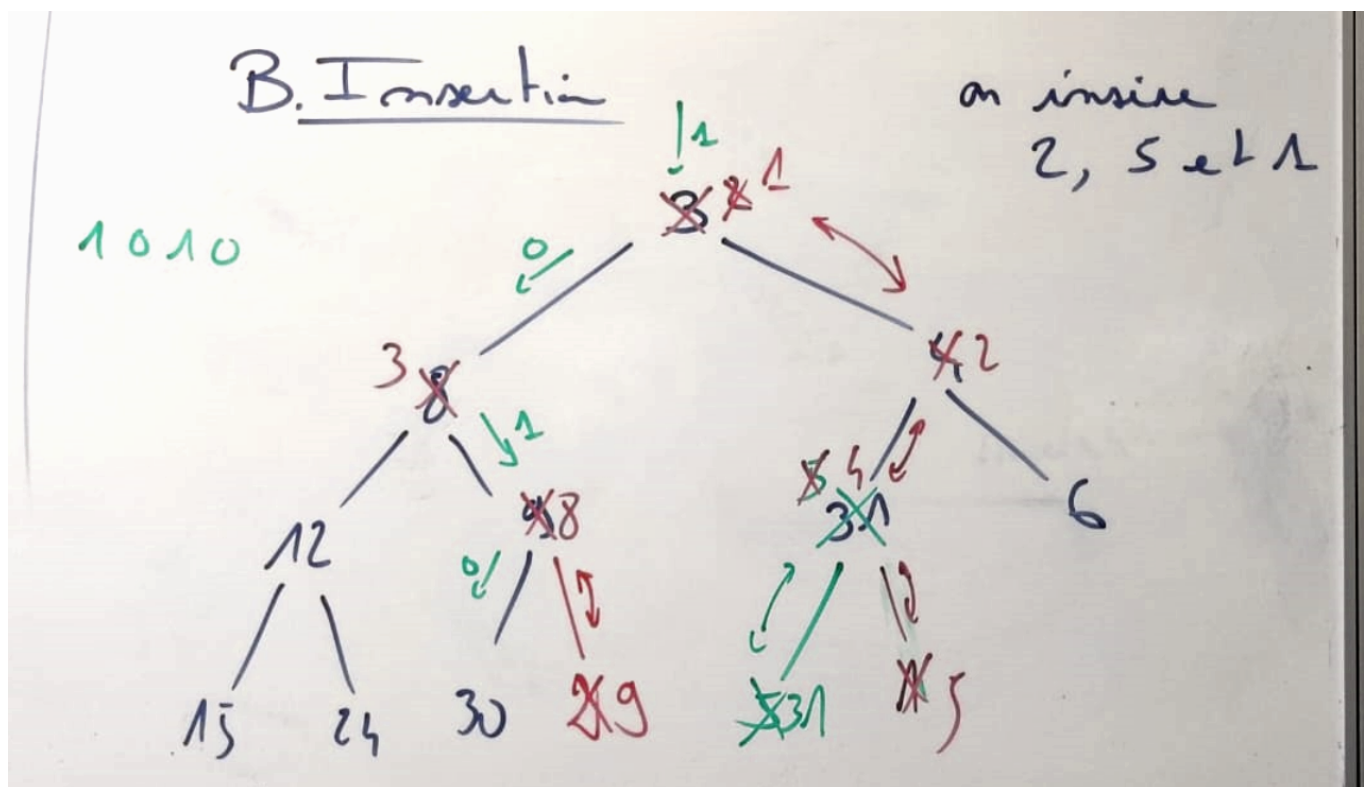
On souhaite insérer 2, 5, et 1 dans l'arbre suivant :



- On commence par rajouter l'élément pour conserver la structure complète gauche : un seul emplacement disponible.



- On fait percoler l'élément vers le haut.

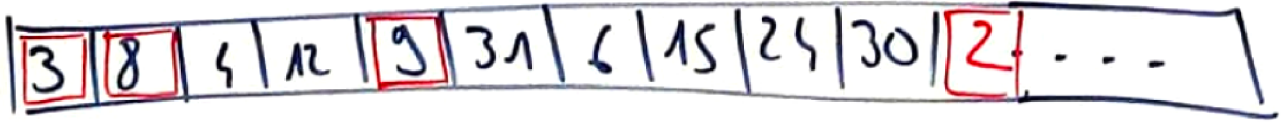


TODO : comprendre ce dessin

Pour insérer un élément dans le tas :

- on écrit cette clé dans la première case libre du tableau
- on fait une percolation vers le haut (sift-up) pour rétablir la propriété d'ordre :
 - si la clé est \geq à celle de son père (ou si on est à la racine), c'est fini.
 - sinon, on échange l'élément avec son père, et on recommence.

👉 **Remarque :** On insère 2 dans :



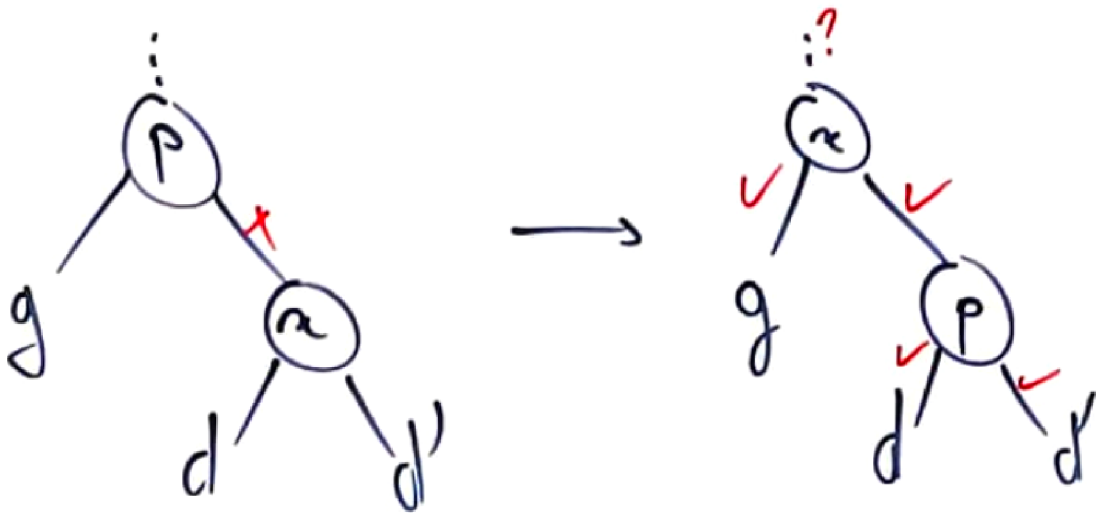
Cela revient à « insérer » le nouvel élément dans le sous-tableau trié correspondant au chemin qui le relie à la racine.

💡 **Proposition :** Si T est un tas binaire, alors le résultat T' de l'insertion de x dans T par le processus précédent est encore un tas binaire.

🔧 Démonstration :

Invariant : À tout moment, on a au plus une violation de la propriété d'ordre des tas : entre x et son père.

- C'est vrai au départ, juste après l'insertion de l'élément x .
- Si x est à la racine ou si $x \geq p$, il n'y a aucune violation : l'arbre est un tas et on s'arrête
- Sinon, on a $x \leq p$ et on effectue l'opération :



- il n'y a pas de problème entre x et p
- on n'a pas créé de problème entre x et g car $x < p \leq \min g$
- on n'a pas créé de problème entre p et d ou d' car les éléments de d et d' étaient des descendants de p dans T .

On a bien supprimé une violation, et on en a créé au plus une.

Enfin, l'algo termine forcément puisque x remonte d'un niveau à chaque étape.

💡 **Proposition :** Insérer un élément dans un tas de taille n se fait en temps $\mathcal{O}(\log n)$.

📖 Exercice :

1. En C, écrire une fonction `void sift_up(heap* h, int i)`, `i` est l'indice du nœud à faire percoler.

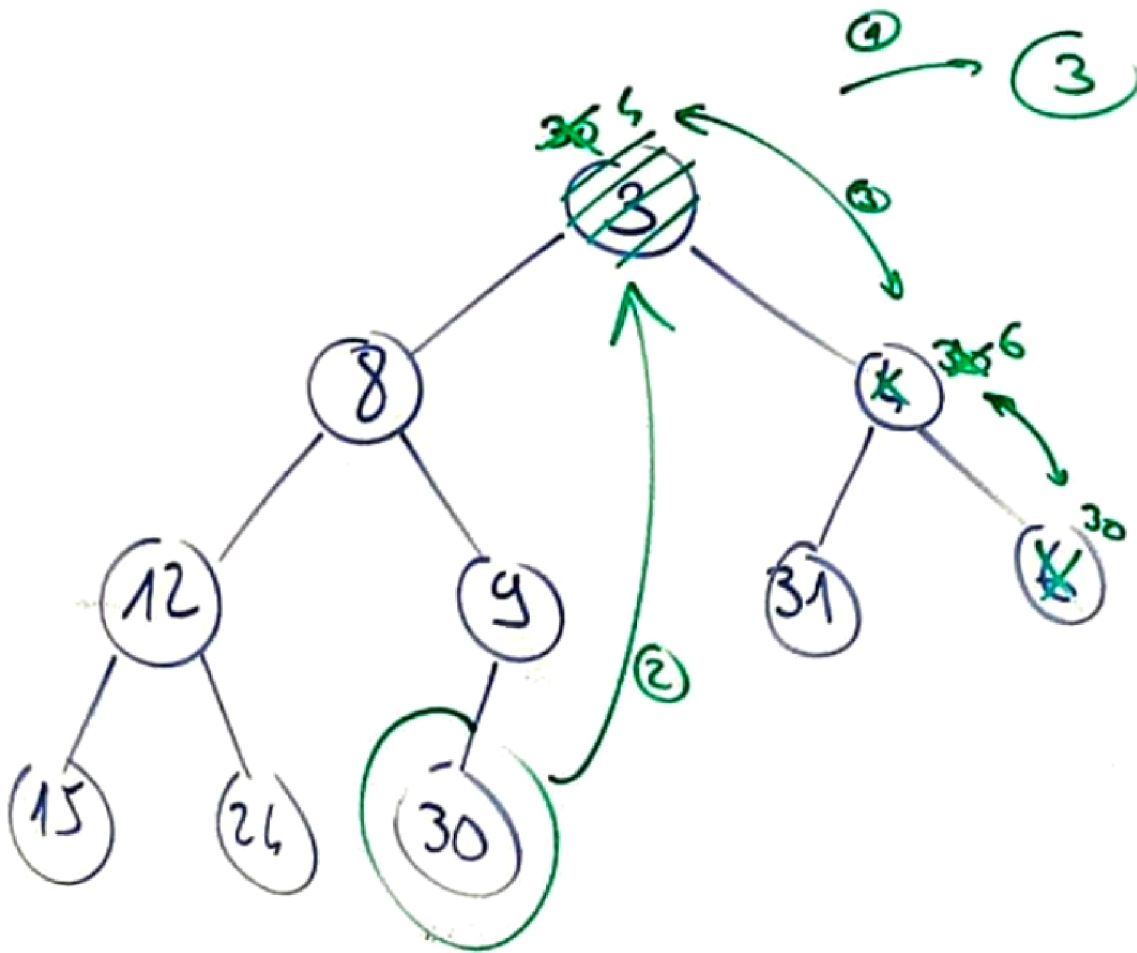
```
void sift_up(heap* h, int i) {
    int idp = (i-1) / 2; // ou int idp = up(i);
    if (h->data[i] >= h->data[idp]) {
        // si i==0, on est dans ce cas et l'algo fini
        return;
    } else {
        swap(h->data, i, idp);
        sift_up(h, idp);
        return;
    }
}
```

2. Écrire une fonction `heap_insert(heap* h, datatype x)`

```
void heap_insert(heap* h, datatype x) {
    assert(h->size < h->capacity);
    h->data[h->size] = x;
    sift_up(h, h->size);
    h->size++;
}
```

4.3. Extraction du minimum

 Exercice :



Pour extraire le minimum :

- on lit ce minimum
- on recopie le dernier élément du tableau sur la racine (la case est « supprimée »)
- on fait percoler cet élément vers le bas (sift-down)
 - on le compare à ses deux fils.
 - s'il est \leq à ses deux fils, on s'arrête.
 - sinon, on l'échange avec le plus petit des deux et on recommence.

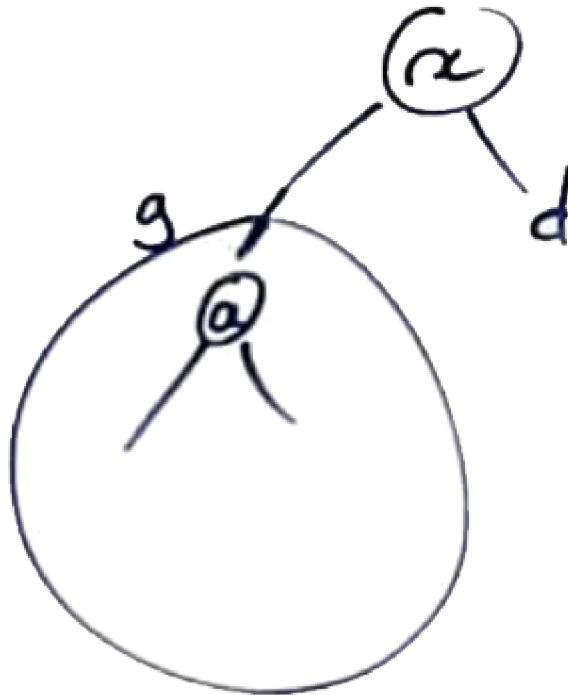
On s'arrête s'il n'y a pas de fils et on s'adapte s'il n'y en a qu'un.

💡 **Proposition :** Si T est un tas binaire non vide, le résultat T' de l'extraction du minimum de T est encore un tas binaire.

🔧 **Démonstration :** On appelle pseudo-tas un arbre binaire complet gauche dans lequel la propriété de tas est respectée sauf éventuellement à la racine.

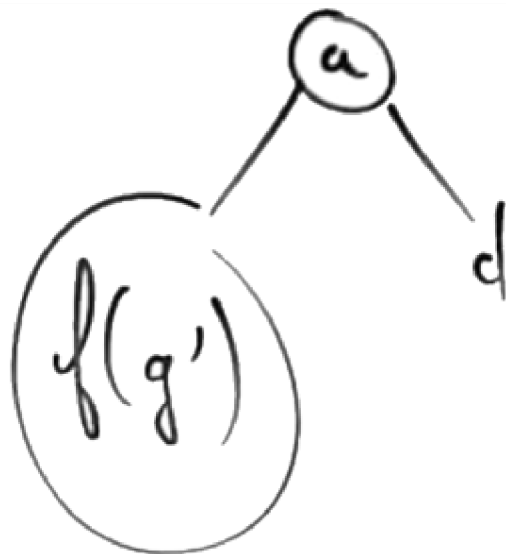
Notons $f(T)$ le résultat de la percolation vers le bas depuis la racine pour un pseudo-tas T . Montrons par induction structurelle que $f(T)$ est un tas.

- Si T est vide ou réduit à une feuille, c'est évident.
- Sinon, on peut supposer sans perte de généralité que le plus petit fils de la racine se trouve à gauche.



Notons que g et d sont des tas.

- Si $a \geq x$, alors on s'arrête, $f(T) = T$ est bien un tas car $x \leq a \leq \min d$
- Sinon, en notant g' l'arbre g dans lequel on a remplacé a par x , on obtient après échange et percolation :



- g' est un pseudo-tas donc par induction $f(g)$ est un tas.
- d est un tas
- a étant la racine de g , donc $a \leq \min d$.
- Le minimum de $f(g')$ est soit x , soit un élément de g . Mais $a = \min(g)$ et $a < x$ donc $a \leq \min(g')$

$\implies f(T)$ est bien un tas.

On peut conclure l'induction structurale. \square

💡 **Proposition :** L'extraction du minimum d'un tas contenant n élément se fait en temps $\mathcal{O}(\log n)$.

📁 Exercice :

1.

```
void sift_down(heap* h, int i) {
    int ind_min = i;
    if (left(i) < h->size && h->data[i] > h->data[left(i)]) {
        ind_min = left(i);
    }
    if (right(i) < h->size && h->data[ind_min] > h->data[right(i)]) {
        ind_min = right(i);
    }
    if (i != ind_min) {
        swap(h->data, i, ind_min);
        sift_down(h, ind_min);
    }
}
```

2.

```
datatype extract_min(heap* h) {
    assert(h->size > 0);
    h->size--;
    datatype min = h->data[0];
    h->data[0] = h->data[h->size];
    sift_down(h, 0);
    return min;
}
```

4.4. Entassement · Heapify

On se donne un tableau, on le voit comme un arbre complet gauche.
On veut le modifier pour qu'il respecte la propriété de tas.

heap h

$\text{data} = t$
$\text{size} = 0$
$\text{capacity} = t $

```
Pour chaque i de 0 à |t| - 1
| size++
| sift_up(h, i)
```

$$\mathcal{O}\left(\sum_{i=1}^n \log(i)\right) = \mathcal{O}(\log(n!))$$

$$= \mathcal{O}(n \log n)$$

$$\log(n!) \sim n \log n$$

Algorithme :

Pour un tableau de taille n

Pour i allant de $\lfloor \frac{n-2}{2} \rfloor$ à 0 (\searrow):

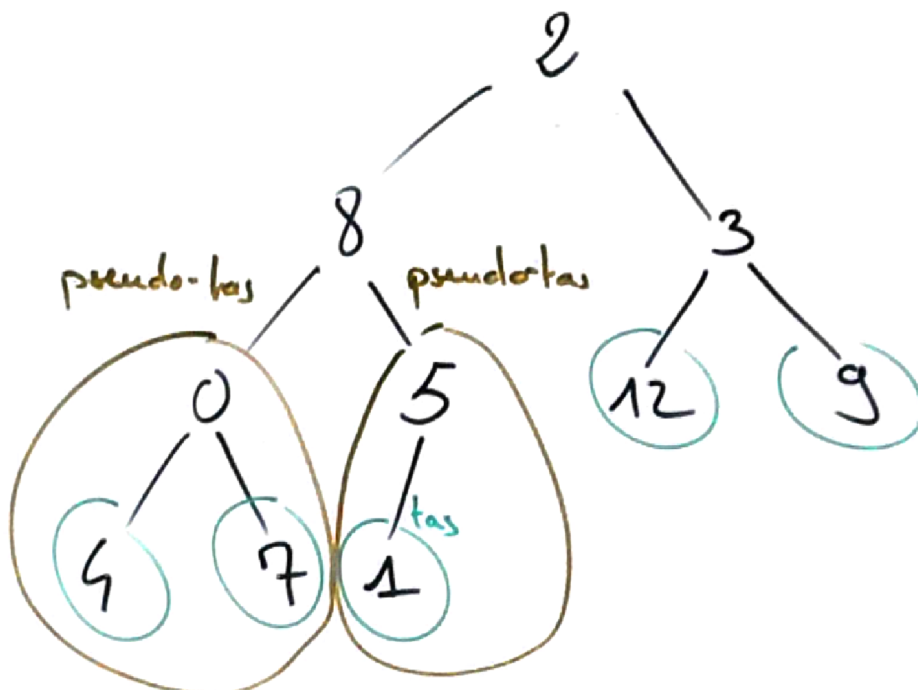
| `percolation_vers_le_bas(t, i)`

Exercice :

```
heap* heapify(datatype t[], int len) {
    heap* h = malloc(sizeof(heap));
    h->size = len;
    h->capacity = len;
    h->data = t;
    for (int i=(n-2)/2 ; i >= 0 ; i--) {
        sift_down(h, i);
    }
    return h;
}
```

Exercice :

1. En remplaçant $\lfloor \frac{n-2}{2} \rfloor$ par $n - 1$, expliquer pourquoi ça fonctionne.
2. Expliquer pourquoi on peut commencer à $\lfloor \frac{n-2}{2} \rfloor$.
3. Quelle est la complexité ?



1. Invariant à la fin de chaque itération i : les arbres enracinés en t_j avec $j \geq i$ sont des tas.

- C'est évident au départ.
- L'invariant est bien conservé : au début de l'itération i , l'arbre enraciné en t_i est un pseudo-tas. Après percolation, c'est bien un tas.
- À la fin, l'arbre enraciné en t_0 est un tas.

2. Une feuille est déjà un tas : inutile de faire la percolation.

Le premier nœud intéressant est le p_re de la dernière feuille :

- indice de cette feuille : $n - 1$
- indice de son père : $\lfloor \frac{n-1-1}{2} \rfloor$

3. Soit h la hauteur de l'arbre.

Pour $i \in \llbracket 0, h \rrbracket$, il y a au plus 2^i nœuds à la profondeur i et chacun de ces nœuds nécessite au plus $(h - i)$ échanges lors de sa percolation.

Au total :

$$\begin{aligned} c &\leq \sum_{i=0}^h 2^i (h - i) \\ &= \sum_{j=0}^h j 2^{h-j} \quad j := h - i \\ &= 2^k \underbrace{\sum_{i=0}^h \frac{i}{2^i}}_{\text{converge}} \\ &\leq A 2^h \quad (A \text{ const.}) \end{aligned}$$

Donc $c = \mathcal{O}(2^k) = \mathcal{O}(n)$

V. Tri par tas (*heapsort*)

À partir d'une file de priorité, on peut construire un algorithme de tri.

1. On insère les éléments un par un dans une file.
2. On fait ensuite n extractions du minimum.

 **Exercice :**

```
1 | void heapsort(datatype t[], int len) {  
    // tri décroissant... en place  
2 | heap* h = heapify(t, len);  
3 | for (int i = len-1; i >= 0; i--) {
```



```
4 |     t[i] = heap_extract_min(h);  
5 | }  
6 | free(h);  
7 | }
```

(On peut faire un swap dans `extract_min` pour éviter ici l'affectation)

Étude de la complexité :

L2 : $\mathcal{O}(n)$

L4 : $\mathcal{O}(n)$

L3-5 :

$$\begin{aligned}\sum_{i=0}^{n-1} \mathcal{O}(\log(i+1)) &= \mathcal{O}\left(\sum_{i=1}^n \log(i)\right) \\ &= \mathcal{O}(\log(n!)) \\ &= \mathcal{O}(n \log(n))\end{aligned}$$