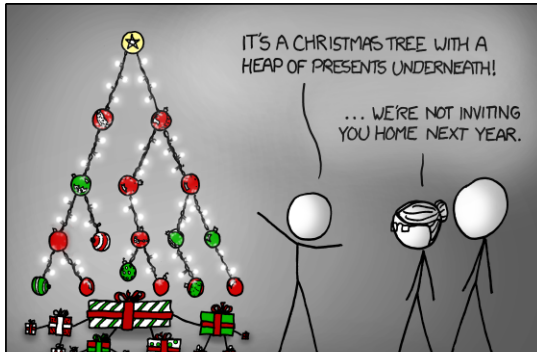


File de priorité et tas

Quentin Fortier

September 26, 2022



File de priorité (FP)

Une **file de priorité max** (FP max) est une structure de données possédant les opérations :

- 1 extraire maximum : supprime et renvoie le maximum
- 2 ajouter élément
- 3 tester si la FP est vide
- 4 (mettre à jour un élément)

Une FP max est utilisée lorsque l'on a souvent besoin de trouver le maximum.

On définit une FP min en remplaçant maximum par minimum.

Exercice

Donner des implémentations possibles de FP.

Implémentation avec liste triée en décroissant :

- ① extraire maximum : en $O(1)$
- ② ajouter élément : en $O(n)$
- ③ mettre à jour : en $O(n)$

Implémentation avec arbre binaire de recherche (ABR) équilibré (par exemple AVL ou ARN) :

- ❶ extraire maximum : en $O(\log(n))$ (sommet tout à droite)
- ❷ ajouter élément : en $O(\log(n))$
- ❸ mettre à jour : en $O(\log(n))$

C'est une bonne implémentation mais il y a plus efficace en pratique...

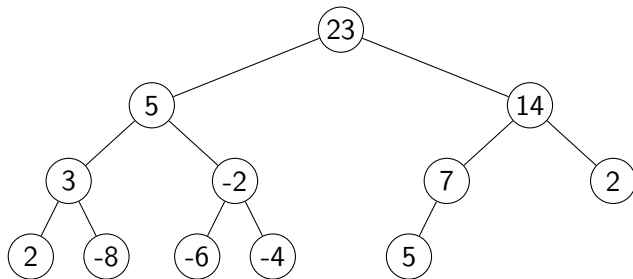
L'implémentation de FP la plus utilisée est un **tas binaire max** :

- ① un **arbre binaire**...
- ② ... **presque complet** : tous les niveaux sont complets, sauf éventuellement le dernier niveau ...
- ③ ... dont **chaque sommet est supérieur à ses éventuels fils**.

À ne pas confondre avec un ABR !

La racine contient le maximum.
(le minimum est une feuille).

Tas max



Le dernier niveau est rempli de gauche à droite.

On considère un arbre binaire à n sommets et de hauteur h .

S'il est complet :

$$n = \sum_{k=0}^h 2^k$$

$$n = 2^{h+1} - 1$$

Un arbre presque complet a son nombre de sommets n compris entre un arbre complet de hauteur $h - 1$ et un arbre complet de hauteur h :

$$\begin{aligned}2^h - 1 &< n \leq 2^{h+1} - 1 \\ \implies 2^h &\leq n < 2^{h+1} \\ \implies h &\leq \log_2(n) < h + 1 \\ \implies &\boxed{h = \lfloor \log_2(n) \rfloor}\end{aligned}$$

Donc $\boxed{h = O(\log(n))}$.

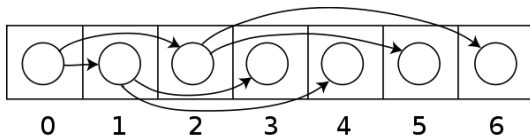
Représentation des tas max

On peut représenter efficacement un arbre binaire à presque complet (donc aussi un tas max) par un tableau a tel que :

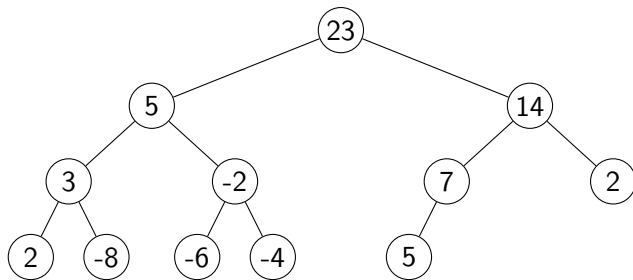
- 1 $a.(0)$ est la racine de a .
- 2 $a.(i)$ a pour fils $a.(2*i + 1)$ et $a.(2*i + 2)$, si ceux-ci sont définis.

Le père de $a.(j)$ est donc $a.((j - 1)/2)$ (si $j \neq 0$)

Ainsi, on accède au père et au fils d'un sommet en $O(1)$.



Représentation des tas max



est représenté par :

[|23; 5; 14; 3; -2; 7; 2; 2; -8; -6; -4; 5; ...|]

C'est le **parcours en largeur du tas** !

Représentation des tas max

On peut utiliser des fonctions utilitaires de manipulation de tas :

```
type 'a heap = { a : 'a array; mutable n : int }
```

```
let pred i = (i - 1)/2
let g i = 2*i + 1
let d i = 2*i + 2
let swap h i j =
  let tmp = h.a.(i) in
  h.a.(i) <- h.a.(j);
  h.a.(j) <- tmp
```

let make-heap p a =
 { a = Array.make p a,
 n = 0 }.

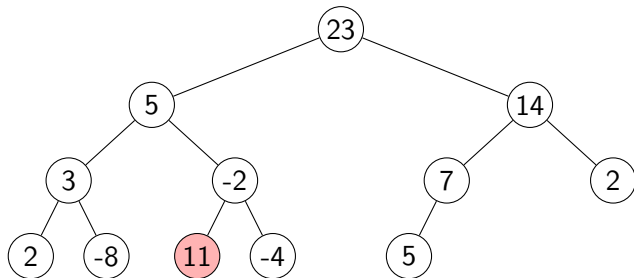
n est le nombre d'éléments du tas (les indices de a après n sont ignorés).

Les feuilles sont d'indices $\left\lfloor \frac{n}{2} \right\rfloor$ à $n - 1$.

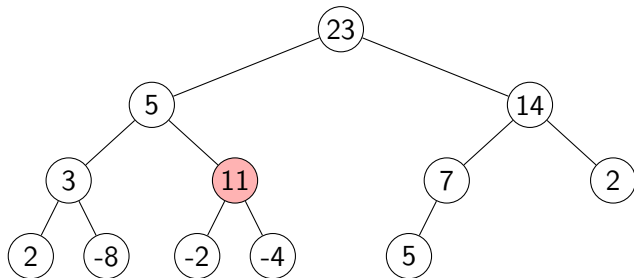
Opérations de tas max

On utilise deux fonctions auxiliaires pour implémenter les opérations sur un tas max heap et un indice i de heap.a :

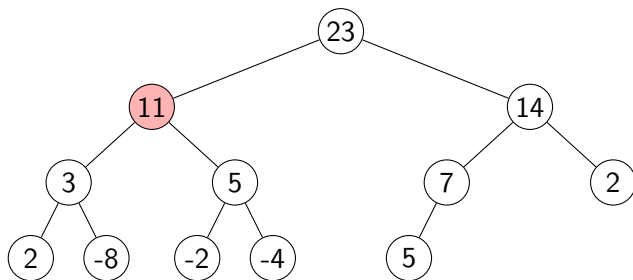
- 1 $\text{up heap } i$: suppose que heap est un tas max sauf $\text{heap.a}(i)$ qui peut être supérieur à son père.
Fait monter $\text{heap.a}(i)$ de façon à obtenir un tas max.
- 2 $\text{down heap } i$: suppose que heap est un tas max sauf $\text{heap.a}(i)$ qui peut être inférieur à un fils.
Fait descendre $\text{heap.a}(i)$ de façon à obtenir un tas max.



[| 23; 5; 14; 3; -2; 7; 2; 2; -8; 11; -4; 5; ... |]



[| 23; 5; 14; 3; 11; 7; 2; 2; -8; -2; -4; 5; ... |]

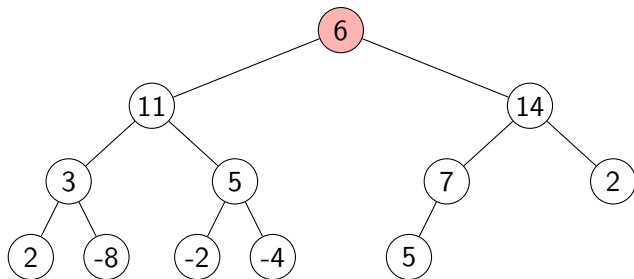


[| 23; 11; 14; 3; 5; 7; 2; 2; -8; -2; -4; 5; ... |]

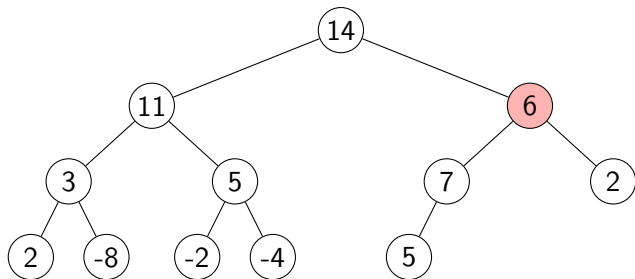
```
let rec up heap i =  
  let p = pred i in  
  if i <> 0 && heap.a.(p) < heap.a.(i) then (  
    swap heap i p;  
    up heap p  
  )
```

Complexité : $O(h) = O(\log(n))$.

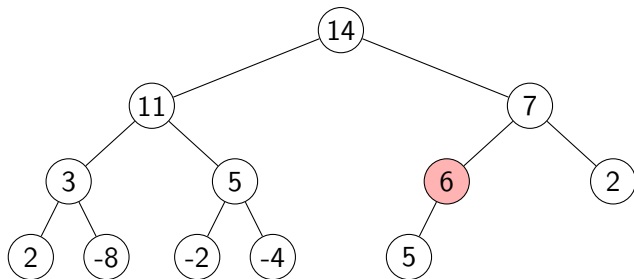
down



down



down



```
let rec down heap i =  
  let get j = (if j < heap.n then heap.a.(j) else min_int), j in  
  let m, j = max (get (2*i + 1)) (get (2*i + 2)) in  
  if heap.a.(i) < m then (  
    swap heap i j;  
    down heap j  
  )
```

Complexité :

```
let rec down heap i =  
  let get j = (if j < heap.n then heap.a.(j) else min_int), j in  
  let m, j = max (get (2*i + 1)) (get (2*i + 2)) in  
  if heap.a.(i) < m then (  
    swap heap i j;  
    down heap j  
  )
```

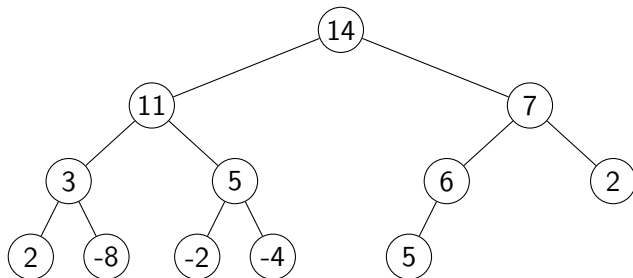
Complexité : $O(h) = O(\log(n))$.

Pour ajouter un élément (tant qu'il reste de la place dans le tableau) :

- 1 l'ajouter en tant que feuille la plus à droite (dernier indice du tableau)
- 2 le faire remonter.

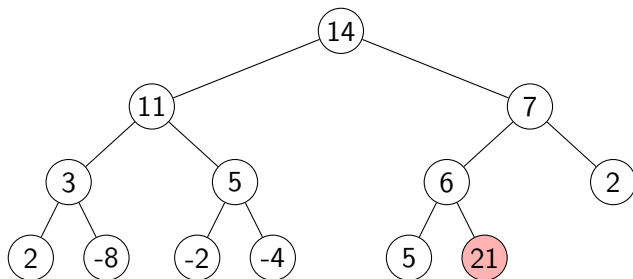
Ajouter élément

Insertion de 21 :



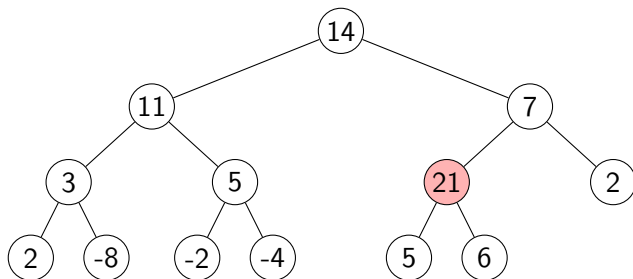
Ajouter élément

Insertion de 21 :



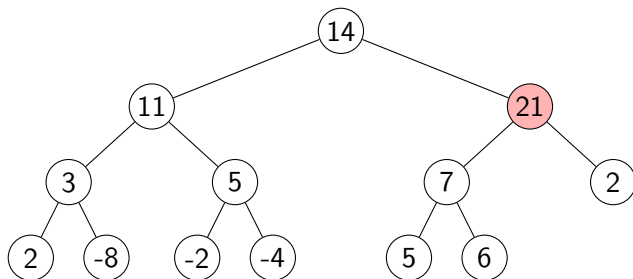
Ajouter élément

Insertion de 21 :



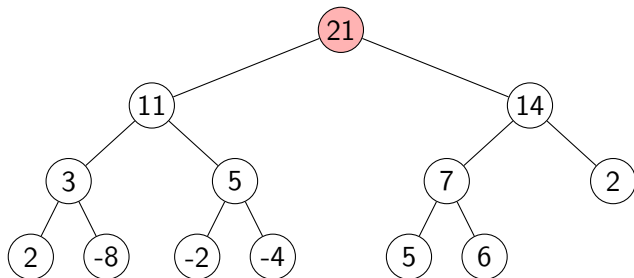
Ajouter élément

Insertion de 21 :



Ajouter élément

Insertion de 21 :



Ajouter élément

Code pour ajouter un élément :

```
let add heap e =  
  heap.a.(heap.n) <- e;  
  up heap heap.n;  
  heap.n <- heap.n + 1
```

Complexité : $O(h) = O(\log(n))$.

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque en tas :

```
let array_to_heap a = Make-Heap()  
  let heap = { a=a; n=0 } in  
  Array.iter (add heap) a;  
  heap
```

Correction :

« au début de la boucle, les i premiers éléments de `tas.t` forment un tas » est un **invariant de boucle**.

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque (de taille n) en tas :

```
let array_to_heap a =  
  let heap = { a=a; n=0 } in  
  Array.iter (add heap) a;  
  heap
```

Complexité :

add est en $O(\log(n))$ donc array_to_tas est en $O(n \log(n))$.

Plus précisément : add heap a.(i) est en $O(p)$, où p est la profondeur de l'élément rajouté.

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque (de taille n) en tas :

```
let array_to_heap a =  
  let heap = { a=a; n=0 } in  
  Array.iter (add heap) a;  
  heap
```

Complexité plus précise :

Dans le pire des cas, chaque élément ajouté à une profondeur p est remonté en racine : p échanges.

Le nombre de swaps est donc, dans le pire cas :

$$\sum_{p=0}^h p 2^p = \dots = \Theta(h 2^h) = \Theta(\log(n)n)$$

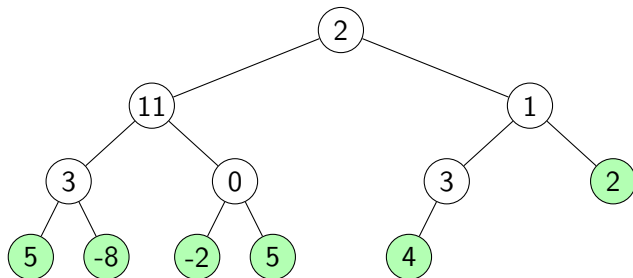
Construire un tas à partir d'un tableau

On a construit le tas en partant de la racine jusqu'aux feuilles.

→ les 2^h feuilles demandent chacune h swaps...

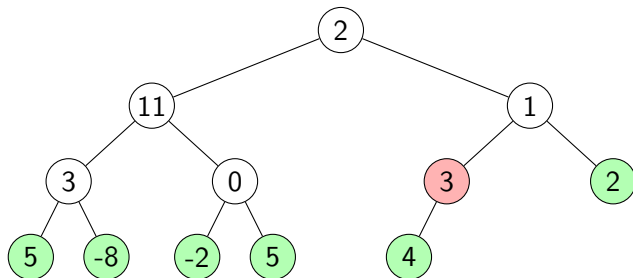
Il est plus intelligent de construire le tas en partant des feuilles :
initialement seules les feuilles vérifient la condition de tas, puis les
sommets de profondeur $\geq h - 1$, puis ceux de profondeur $\geq h - 2$...

Construire un tas à partir d'un tableau



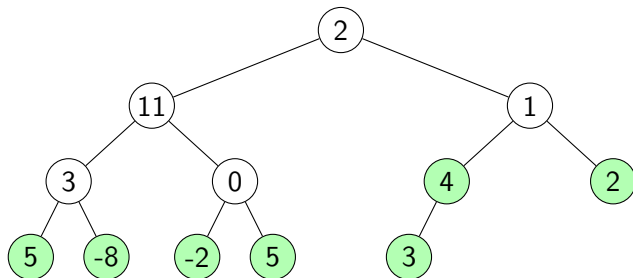
[2; 11; 1; 3; 0; 3; 2; 5; -8; -2; 5; 4]

Construire un tas à partir d'un tableau



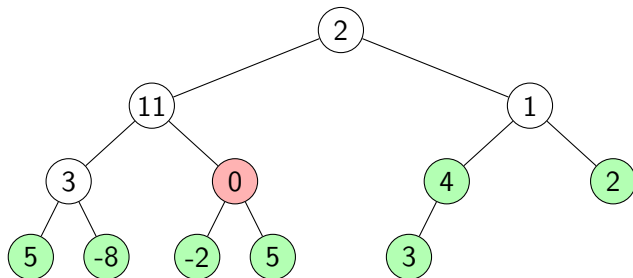
[[2; 11; 1; 3; 0; 3; 2; 5; -8; -2; 5; 4]]

Construire un tas à partir d'un tableau



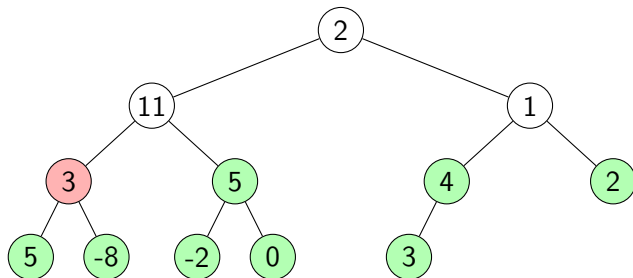
[2; 11; 1; 3; 0; 4; 2; 5; -8; -2; 5; 3]

Construire un tas à partir d'un tableau



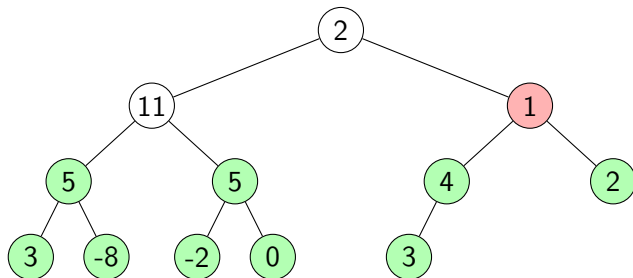
[[2; 11; 1; 3; 0; 4; 2; 5; -8; -2; 5; 3]]

Construire un tas à partir d'un tableau



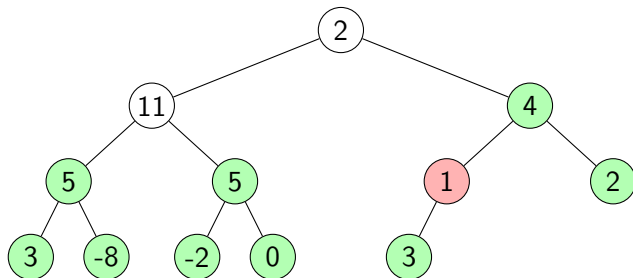
[[2; 11; 1; 3; 5; 4; 2; 5; -8; -2; 0; 3]]

Construire un tas à partir d'un tableau



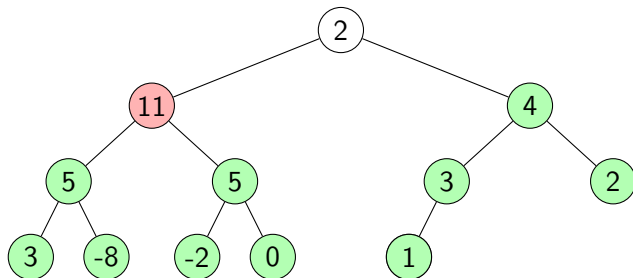
[[2; 11; 1; 5; 5; 4; 2; 3; -8; -2; 0; 3]]

Construire un tas à partir d'un tableau



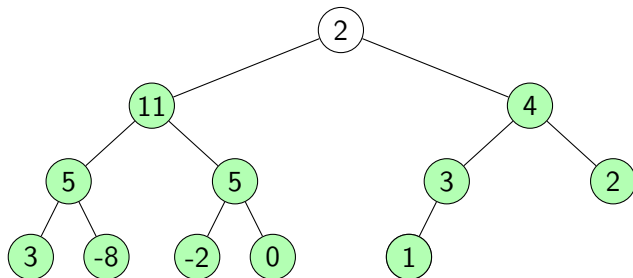
[[2; 11; 4; 5; 5; 1; 2; 3; -8; -2; 0; 3]]

Construire un tas à partir d'un tableau



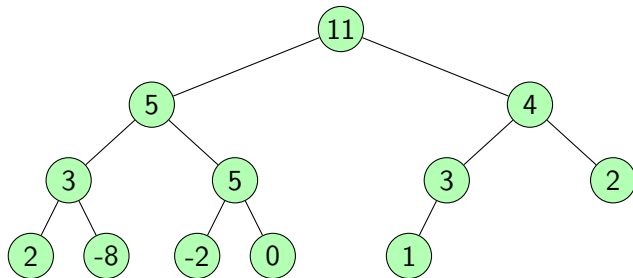
[[2; 11; 4; 5; 5; 3; 2; 3; -8; -2; 0; 1]]

Construire un tas à partir d'un tableau



[[2; 11; 4; 5; 5; 3; 2; 3; -8; -2; 0; 1]]

Construire un tas à partir d'un tableau



[| 11; 5; 4; 3; 5; 3; 2; 2; -8; -2; 0; 1 |]

Construire un tas à partir d'un tableau

Code pour cette 2ème méthode :

```
let array_to_heap a =  
  let n = Array.length a in  
  let heap = { a=a; n=n } in  
  for i = n/2 - 1 downto 0 do  
    down heap i;  
  done;  
  heap
```

Correction :

« au début de la boucle, les éléments après i dans `heap.a` vérifient la condition de tas » est un **invariant de boucle**.

Construire un tas à partir d'un tableau

Complexité :

Un sommet à la profondeur p est descendu en faisant au plus $h - p$ swaps.

D'où la complexité totale :

$$\sum_{p=0}^h (h - p)2^p = \dots = \Theta(2^h) = \Theta(n)$$

On obtient une complexité **linéaire**.

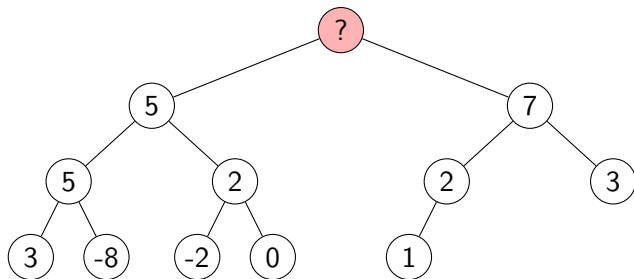
Extraire le maximum

On veut supprimer et renvoyer la racine, en conservant la structure de tas.

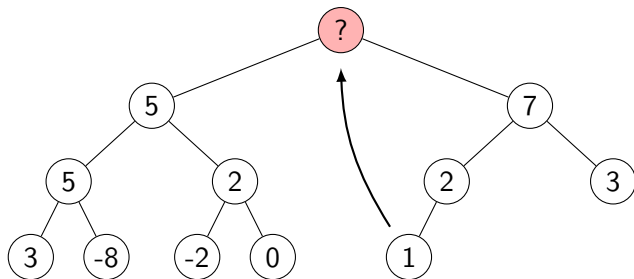
On peut :

- ➊ Remplacer la racine par la dernière feuille.
- ➋ Appeler down dessus.

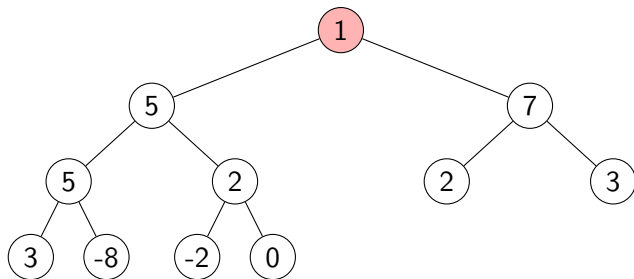
Extraire le maximum



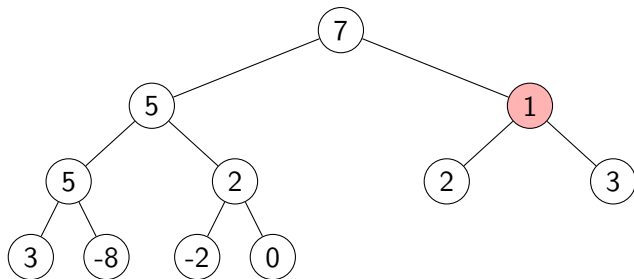
Extraire le maximum



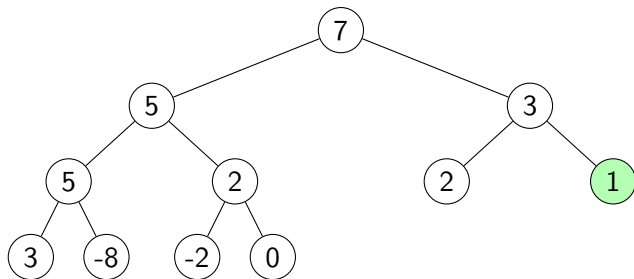
Extraire le maximum



Extraire le maximum



Extraire le maximum



Extraire le maximum

Code pour extraire la racine d'un tas :

```
let rec take_max heap =  
  swap heap.a 0 (heap.n - 1);  
  heap.n <- heap.n - 1;  
  down heap 0;  
  heap.a.(heap.n)
```

Complexité : $O(\log(n))$.

Remarques :

- on met le maximum à la fin
- cette méthode ne permet que de supprimer la racine (maximum), pas un élément quelconque

Mettre à jour un élément

Pour mettre à jour un élément :

- 1 Si on augmente son étiquette : on le monte.
- 2 Sinon : on le descend.

Mettre à jour un élément

utile pour Dijkstra

Pour mettre à jour un élément :

```
let update heap i v =  
  let p = heap.a.(i);  
  heap.a.(i) <- v;  
  if v > p then up heap i  
  else down heap i
```

Complexité : $O(\log(n))$.

Tas max : résumé

Opération	Tas max
ajouter élément	$O(\log(n))$
extraire maximum	$O(\log(n))$
valeur du maximum	$O(1)$
mettre à jour	$O(\log(n))$
créer à partir d'un tableau de taille n	$O(n)$

File de priorité

Comparaison des implémentations de files de priorités :

Opération	Liste triée	Tas	ABR équilibré
ajouter	$O(n)$	$O(\log(n))$	$O(\log(n))$
extraire max	$O(1)$	$O(\log(n))$	$O(\log(n))$
valeur du max	$O(1)$	$O(1)$	$O(\log(n))$
update	$O(n)$	$O(\log(n))$	$O(\log(n))$
Conversion depuis array	$O(n \log(n))$	$O(n)$	$O(n \log(n))$

Toute FP avec ajout et extraction du maximum en $O(f(n))$ donne un algorithme de tri en $O(nf(n))$: on ajoute un à un les éléments extraits dans une nouvelle liste.

- ❶ FP implémenté avec tas \implies tri en $O(n \log(n))$
- ❷ FP implémenté avec AVL/ARN \implies tri en $O(n \log(n))$
- ❸ ...

Mais avec un tas on peut éviter de créer un nouveau tableau (complexité $O(1)$ **en mémoire**).

Tri par tas

Code pour trier avec un tas :

```
let heap_sort a =  
  let heap = array_to_heap a in  
  for i = 0 to heap.n - 1 do  
    take_max heap  
  done
```

Correction : « au début de la boucle, les éléments de heap d'indices $n - i$ à $n - 1$ sont les i plus grands éléments triés ».

Tri par tas

Code pour trier avec un tas :

```
let heap_sort a =  
  let heap = array_to_heap a in  
  for i = 0 to heap.n - 1 do  
    take_max heap  
  done
```

Complexité : $O(n + n \log(n)) = O(n \log(n))$ (optimal pour un tri).

Tri par tas

Code pour trier avec un tas :

```
let heap_sort a =  
  let heap = array_to_heap a in  
  for i = 0 to heap.n - 1 do  
    take_max heap  
  done
```

Complexité en mémoire (espace utilisé en plus de l'entrée) : $O(1)$

On dit que le tri est **en place** : pas besoin de créer un nouveau tableau.

Exercice

Comment trier partiellement un tableau (seulement les k plus grands ou plus petits) ?

Il suffit d'arrêter la boucle au bout de k itérations.

Complexité : $O(n + k \log(n))$.

Ceci donne un algorithme linéaire pour trouver le k ème plus petit élément d'un tableau, pour $k \leq \frac{n}{\log(n)}$.

Exercice

Écrire des fonctions `take_max`, `add`, `is_empty` implémentant les opérations de FP max avec un ABR.

En déduire un algorithme de tri `'a list -> 'a list`.