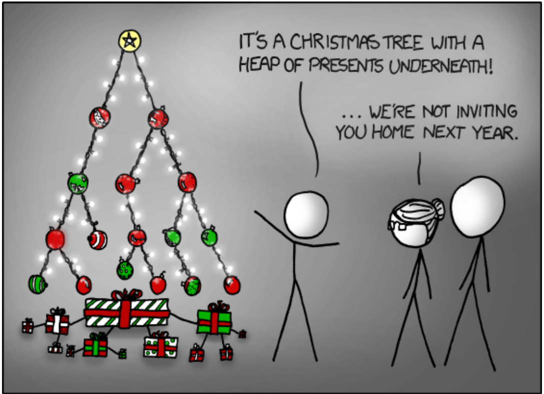
Structures de données 2 : file de priorité et tas

MP/MP* option info



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. mettre à jour un élément
4. tester si la FP est vide

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. mettre à jour un élément
4. tester si la FP est vide

Une FP max est utilisée lorsque l'on a besoin de chercher le maximum plusieurs fois.
On définit une FP min en remplaçant maximum par minimum.

File de priorité max

Implémentation avec tableau trié en décroissant:

1. extraire maximum:

File de priorité max

Implémentation avec tableau trié en décroissant:

1. extraire maximum: en $O(n)$ (il faut décaler les éléments)
2. ajouter élément:

File de priorité max

Implémentation avec tableau trié en décroissant:

1. extraire maximum: en $O(n)$ (il faut décaler les éléments)
2. ajouter élément: en $O(n)$ (idem)
3. mettre à jour: en $O(n)$ (idem)

File de priorité max

Implémentation avec liste triée en décroissant:

1. extraire maximum:

File de priorité max

Implémentation avec liste triée en décroissant:

1. extraire maximum: en $O(1)$
2. ajouter élément:

File de priorité max

Implémentation avec liste triée en décroissant:

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

File de priorité max

Implémentation avec ABR équilibré (par exemple AVL ou ARN):

1. extraire maximum:

File de priorité max

Implémentation avec ABR équilibré (par exemple AVL ou ARN):

1. extraire maximum: en $O(\log(n))$ (sommet tout à droite)
2. ajouter élément:

File de priorité max

Implémentation avec ABR équilibré (par exemple AVL ou ARN):

1. extraire maximum: en $O(\log(n))$ (sommet tout à droite)
2. ajouter élément: en $O(\log(n))$
3. mettre à jour:

File de priorité max

Implémentation avec ABR équilibré (par exemple AVL ou ARN):

1. extraire maximum: en $O(\log(n))$ (sommet tout à droite)
2. ajouter élément: en $O(\log(n))$
3. mettre à jour: en $O(\log(n))$

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

Tas max

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**.

Tas max

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

Tas max

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

Tas max

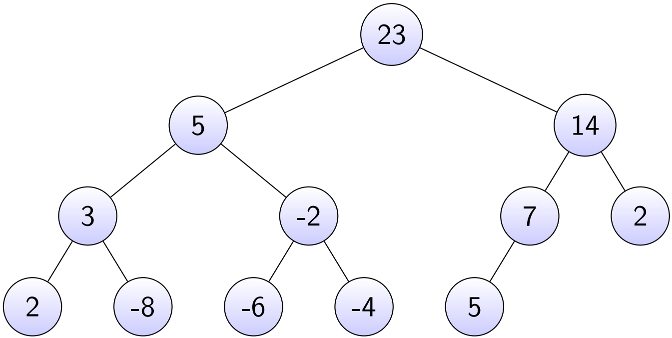
L'implémentation de FP la plus utilisée est un **tas binaire max**:

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.

Tas max

Soit un arbre binaire à n sommets et de hauteur h.

S'il est complet:

Tas max

Soit un arbre binaire à n sommets et de hauteur h.

S'il est complet:

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

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

Tas max

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:

Tas max

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:

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

$$\Longrightarrow 2^h \leq n < 2^{h+1}$$

$$\Longrightarrow h \leq \log_2(n) < h+1$$

$$\Longrightarrow \boxed{h = \lfloor \log_2(n) \rfloor}$$

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

Représentation des tas max

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

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

Le père de `t.(j)` est donc

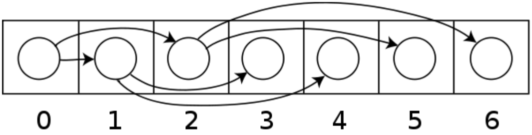
Représentation des tas max

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

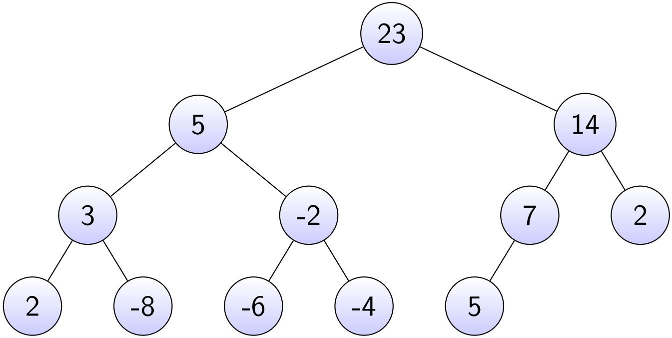
1. `t.(0)` est la racine de a.
2. `t.(i)` a pour fils `t.(2*i + 1)` et `t.(2*i + 2)`, si ceux-ci sont définis.

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

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

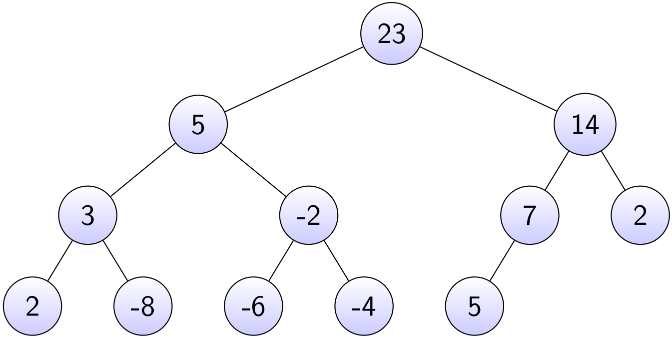


Représentation des tas max



est représenté par:

Représentation des tas max

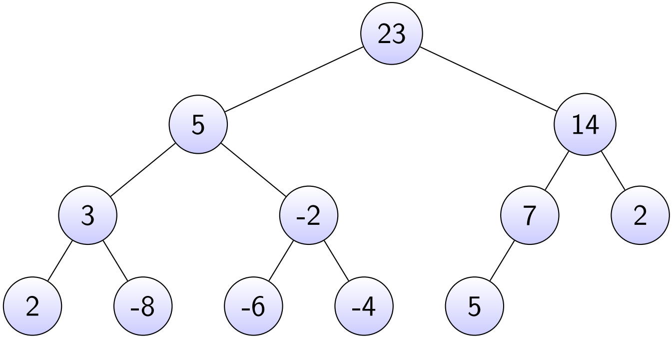


est représenté par:

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

C'est

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 utilise des fonctions utilitaires de manipulation de tas:

```
type 'a tas = { t : 'a array; mutable n : int };;

let pere i = (i - 1) / 2;;
let fg i = 2 * i + 1;;
let fd i = 2 * i + 2;;
let swap tas i j =
  let tmp = tas.t.(i) in
  tas.t.(i) <- tas.t.(j);
  tas.t.(j) <- tmp;;
```

Représentation des tas max

On utilise des fonctions utilitaires de manipulation de tas:

```ocaml
type 'a tas = { t : 'a array; mutable n : int };;

let pere i = (i - 1) / 2;;
let fg i = 2 * i + 1;;
let fd i = 2 * i + 2;;
let swap tas i j =
  let tmp = tas.t.(i) in
  tas.t.(i) <- tas.t.(j);
  tas.t.(j) <- tmp;;
```

n est le nombre d'éléments du tas (les indices de `t` après n sont ignorés).

Les feuilles sont d'indices

Représentation des tas max

On utilise des fonctions utilitaires de manipulation de tas:

```
type 'a tas = { t : 'a array; mutable n : int };;

let pere i = (i - 1) / 2;;
let fg i = 2 * i + 1;;
let fd i = 2 * i + 2;;
let swap tas i j =
  let tmp = tas.t.(i) in
  tas.t.(i) <- tas.t.(j);
  tas.t.(j) <- tmp;;
```

n est le nombre d'éléments du tas (les indices de `t` 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 `t` et un indice `i` de `t`:

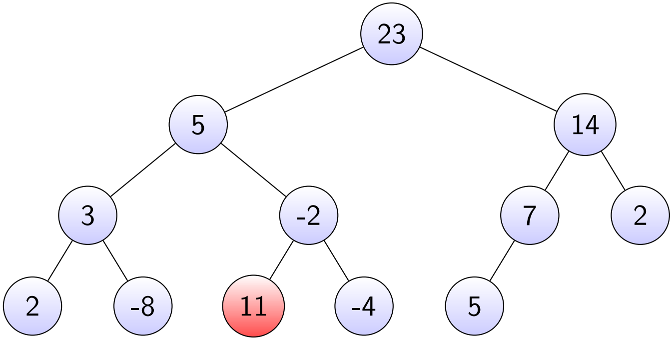
1. `monter t i`: suppose que `t` est un tas max sauf `t.(i)` qui peut être supérieur à son père.
   Fait remonter `t.(i)` de façon à obtenir un tas max.

Opérations de tas max

On utilise deux fonctions auxiliaires pour implémenter les opérations sur un tas max `t` et un indice `i` de `t`:

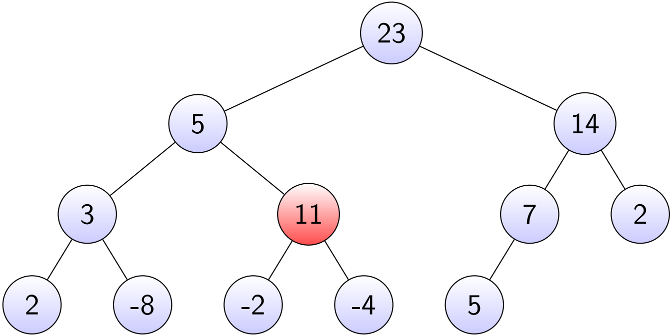
1. `monter t i`: suppose que `t` est un tas max sauf `t.(i)` qui peut être supérieur à son père.
   Fait remonter `t.(i)` de façon à obtenir un tas max.
2. `descendre t i`: suppose que `t` est un tas max sauf `t.(i)` qui peut être inférieur à un fils.
   Fait descendre `t.(i)` de façon à obtenir un tas max.

monter



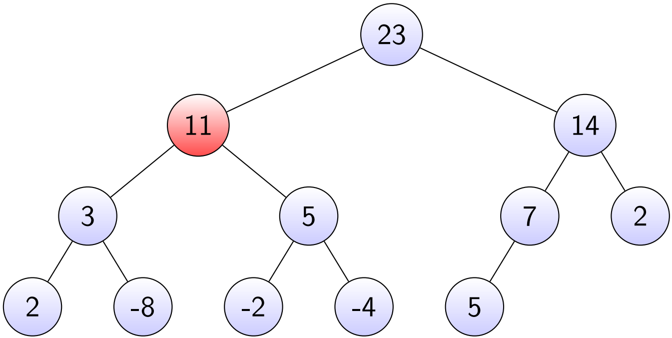
[|23; 5; 14; 3; -2; 7; 2; 2; -8; 11; -4; 5; ... |]

monter



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

monter



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

monter

Code de `monter`:

monter

Code de `monter`:

```
let rec monter tas i =
  if i <> 0 && tas.t.(pere i) < tas.t.(i)
  then begin
    swap tas i (pere i);
    monter tas (pere i)
  end;;
```

Complexité:

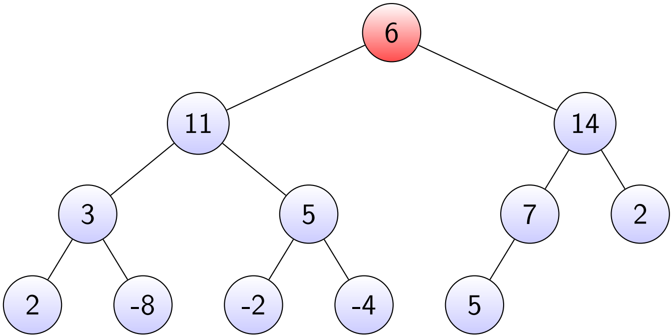
monter

Code de `monter`:

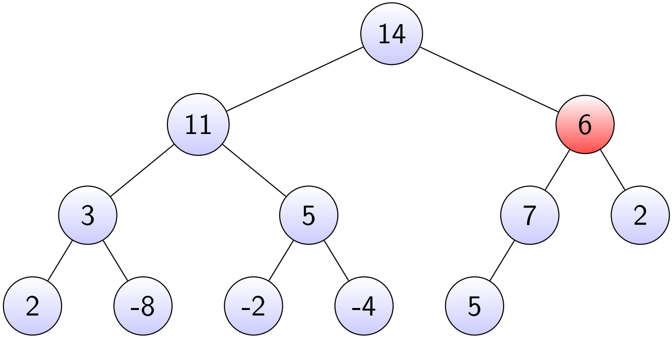
```
let rec monter tas i =
  if i <> 0 && tas.t.(pere i) < tas.t.(i)
  then begin
    swap tas i (pere i);
    monter tas (pere i)
  end;;
```

Complexité: $O(h) = O(\log(n))$.

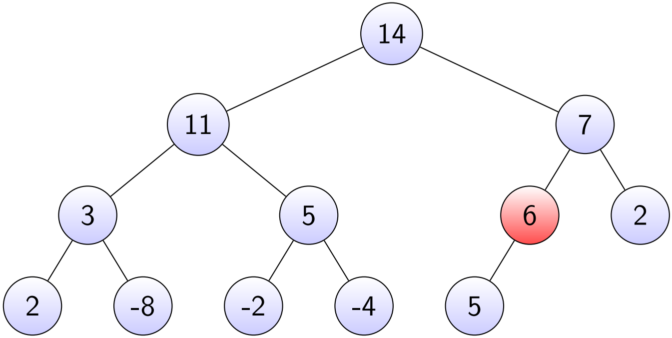
descendre



descendre



descendre



descendre

Code de `descendre`:

descendre

Code de `descendre`:

```
let rec descendre tas i =
  let j = ref i in
  if fg i < tas.n && tas.t.(fg i) > tas.t.(i) then j := fg i;
  if fd i < tas.n && tas.t.(fd i) > tas.t.(!j) then j := fd i;
  (* !j contient l'indice de l'element max parmi i et ses fils *)
  if !j <> i then begin
    swap tas i !j;
    descendre tas !j
  end;;
```

Complexité:

descendre

Code de `descendre`:

```
let rec descendre tas i =
  let j = ref i in
  if fg i < tas.n && tas.t.(fg i) > tas.t.(i) then j := fg i;
  if fd i < tas.n && tas.t.(fd i) > tas.t.(!j) then j := fd i;
  (* !j contient l'indice de l'element max parmi i et ses fils *)
  if !j <> i then begin
    swap tas i !j;
    descendre tas !j
  end;;
```

Complexité: $O(h) = O(\log(n))$.

Ajouter élément

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

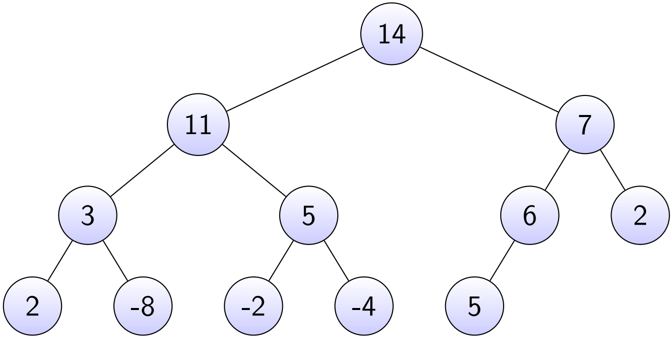
Ajouter élément

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.

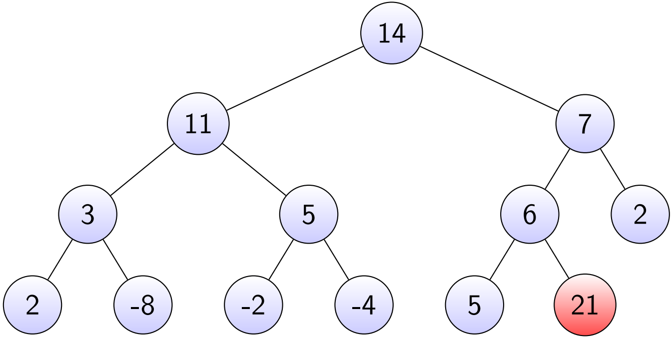
Ajouter élément

Insertion de 21:



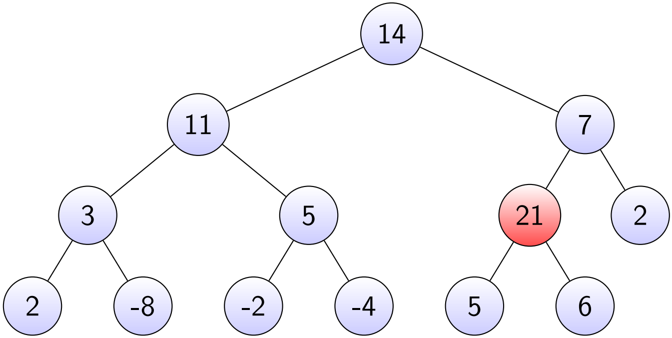
Ajouter élément

Insertion de 21:



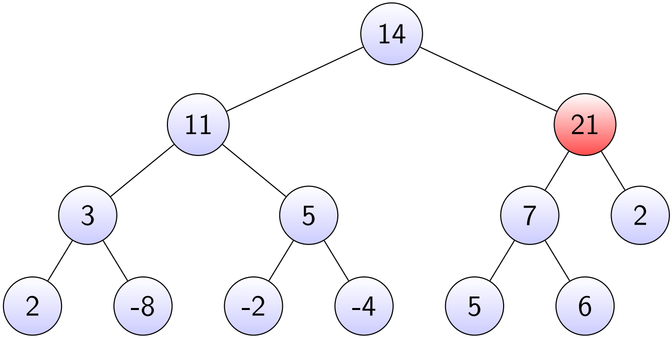
Ajouter élément

Insertion de 21:



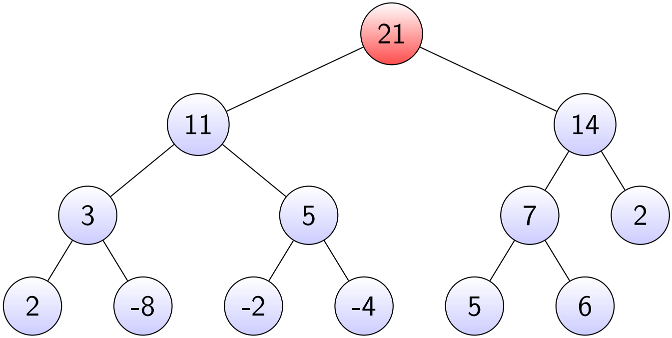
Ajouter élément

Insertion de 21:



Ajouter élément

Insertion de 21:



Ajouter élément

Code pour ajouter un élément:

```
let add e tas =
  tas.t.(tas.n) <- e;
  monter tas tas.n;
  tas.n <- tas.n + 1;;
```

Complexité:

Ajouter élément

Code pour ajouter un élément:

```
let add e tas =
  tas.t.(tas.n) <- e;
  monter tas tas.n;
  tas.n <- tas.n + 1;;
```

Complexité: $O(h) = O(\log(n))$.

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque en tas:

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque en tas:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

Correction:

Construire un tas à partir d'un tableau

Conversion d'un tableau quelconque en tas:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

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_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

Complexité:

add est en

Construire un tas à partir d'un tableau

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

```
let array_to_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

Complexité:

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

Construire un tas à partir d'un tableau

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

```
let array_to_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

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

Construire un tas à partir d'un tableau

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

```
let array_to_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

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

Plus précisément: `add tab.(i) tas` 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_tas tableau =
  let tas = { t = tableau; n = 1 } in
  for i = 1 to Array.length tableau - 1 do
    add tableau.(i) tas
  done;
  tas;;
```

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:

$$\sum_{p=0}^{h} p2^p = \ldots = \Theta(h2^h) = \Theta(\log(n)n)$$

Profondeur moyenne d'un sommet

Remarque: on a montré que la profondeur moyenne d'un sommet dans un arbre binaire complet à n sommets est:

$$\frac{\displaystyle\sum_{p=0}^{h} p2^p}{\text{nb sommets}} = \frac{\Theta(n\log(n))}{n} = \Theta(\log(n))$$

Construire un tas à partir d'un tableau

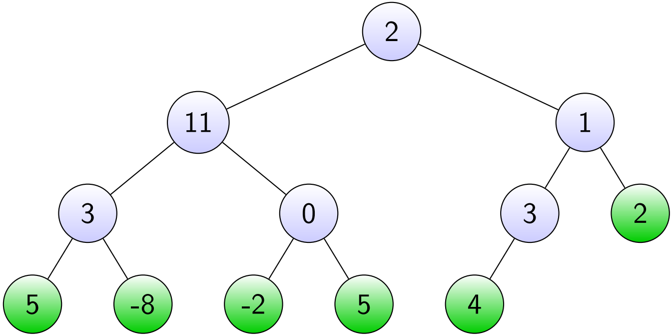
On a construit le tas en partant de la racine jusqu'aux feuilles.
Résultat: les 2^h feuilles demandent chacune h swaps...

Construire un tas à partir d'un tableau

On a construit le tas en partant de la racine jusqu'aux feuilles.
Résultat: les 2^h feuilles demandent chacune h swaps...

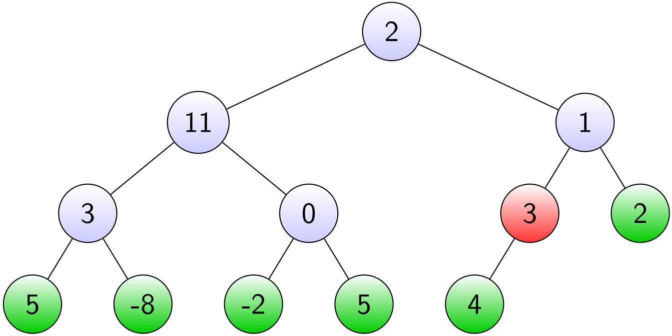
Il est plus intelligent de construire le tas en partant des feuilles: initialement seules les feuilles vérifient la condition de tas, puis les sommets de profondeur $\geq h - 1$, puis ceux de profondeur $\geq h - 2$...

Construire un tas à partir d'un tableau



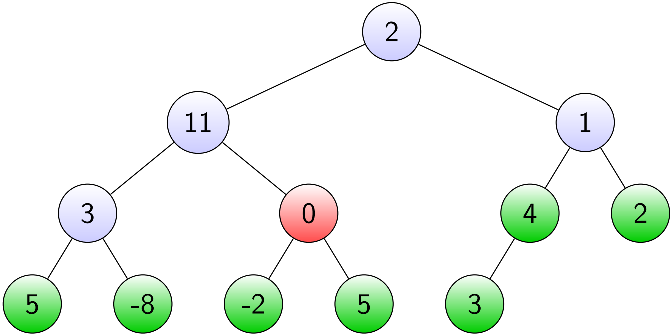
[|2; 11; 1; 3; 0; 3; 2; 5; -8; -2; 5; 4 |]

Construire un tas à partir d'un tableau



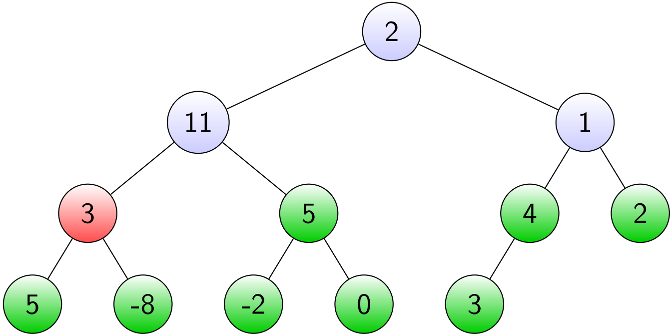
[|2; 11; 1; 3; 0; 3; 2; 5; -8; -2; 5; 4 |]

Construire un tas à partir d'un tableau



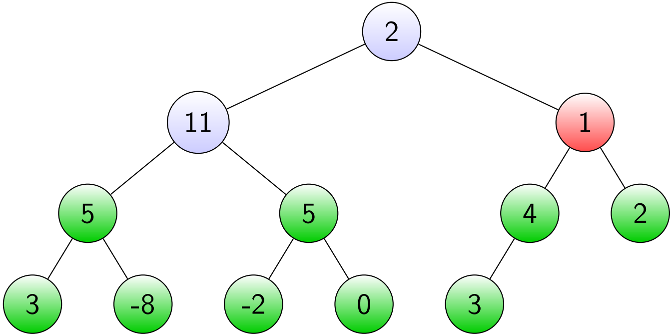
[|2; 11; 1; 3; 0; 4; 2; 5; -8; -2; 5; 3|]

Construire un tas à partir d'un tableau



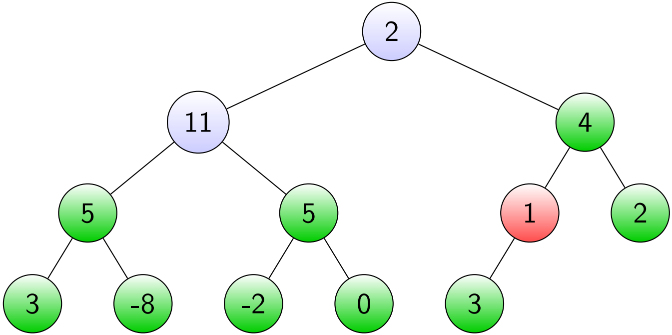
[|2; 11; 1; 3; 5; 4; 2; 5; -8; -2; 0; 3|]

Construire un tas à partir d'un tableau



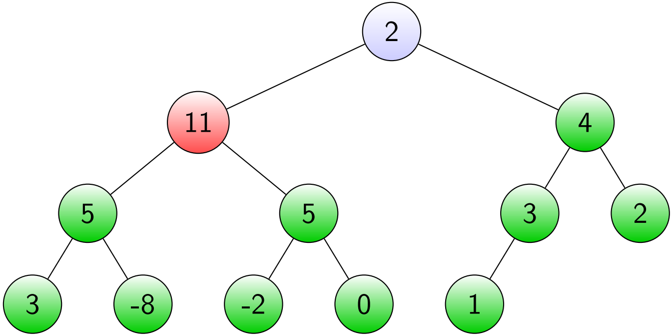
[|2; 11; 1; 5; 5; 4; 2; 3; -8; -2; 0; 3|]

Construire un tas à partir d'un tableau



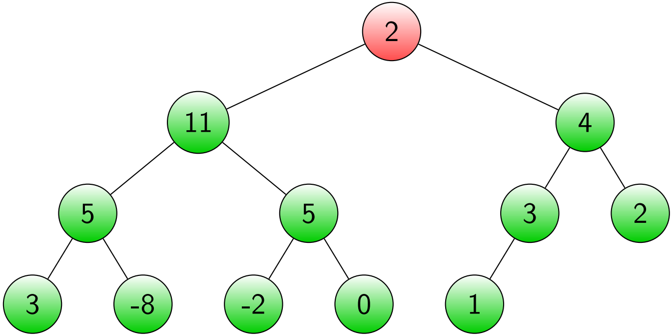
[|2; 11; 4; 5; 5; 1; 2; 3; -8; -2; 0; 3|]

Construire un tas à partir d'un tableau



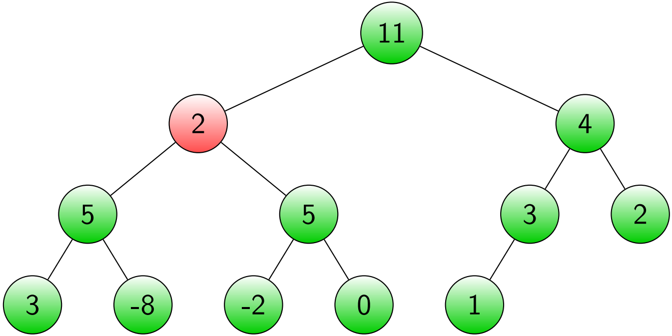
[|2; 11; 4; 5; 5; 3; 2; 3; -8; -2; 0; 1|]

Construire un tas à partir d'un tableau



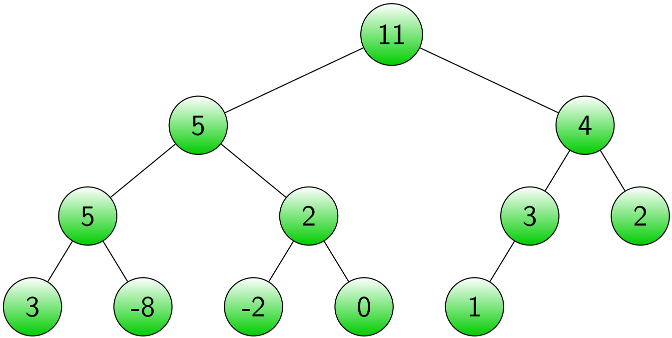
`[|2; 11; 4; 5; 5; 3; 2; 3; -8; -2; 0; 1|]`

Construire un tas à partir d'un tableau



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

Construire un tas à partir d'un tableau



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

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

Correction:

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

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

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

Complexité:

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

Complexité:

Un sommet à la profondeur p est descendu en faisant au plus

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

Complexité:
Un sommet à la profondeur p est descendu en faisant au plus $h - p$ swaps. D'où la complexité totale:

Construire un tas à partir d'un tableau

Code pour la 2ème méthode:

```
let array_to_tas tableau =
  let tas = { t = tableau; n = Array.length tableau } in
  for i = tas.n / 2 - 1 downto 0 do
    descendre tas i
  done;
  tas;;
```

Complexité:
Un sommet à la profondeur p est descendu en faisant au plus $h - p$ swaps. D'où la complexité totale:

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

On obtient une complexité **linéaire**.

Extraire le maximum

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

Extraire le maximum

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

On peut:

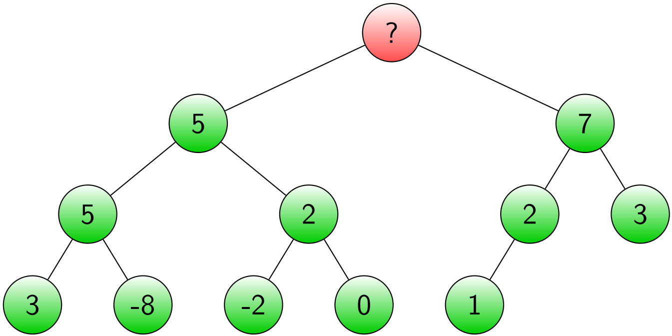
Extraire le maximum

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

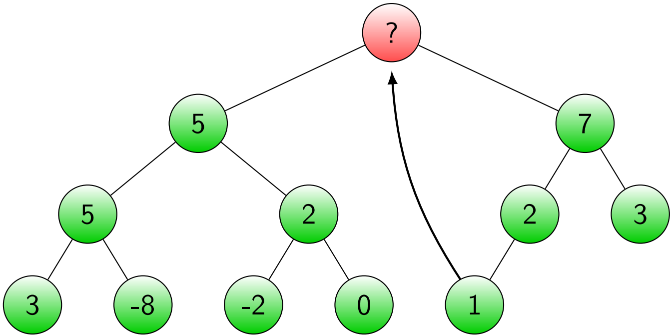
On peut:

1. Remplacer la racine par la dernière feuille.
2. Appeler descendre dessus.

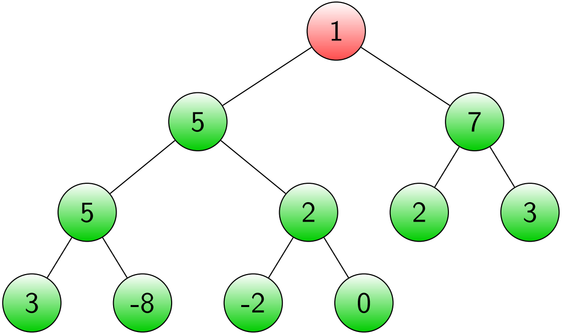
Extraire le maximum



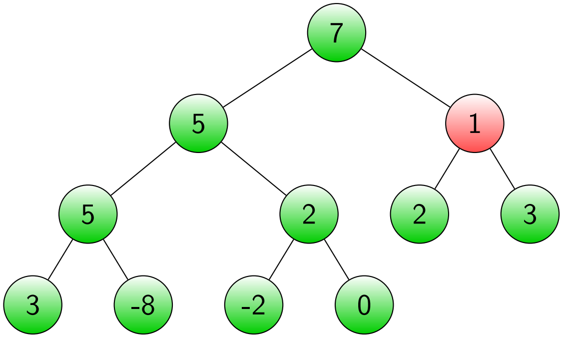
Extraire le maximum



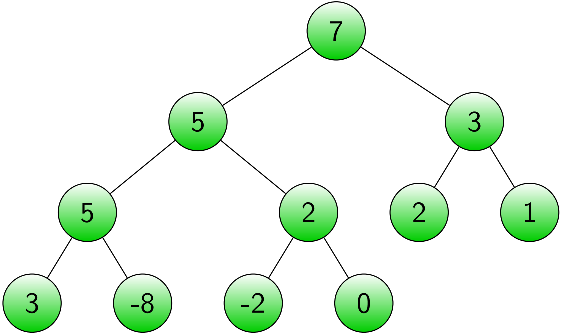
Extraire le maximum



Extraire le maximum



Extraire le maximum



Extraire le maximum

Code pour extraire la racine d'un tas:

Extraire le maximum

Code pour extraire la racine d'un tas:

```
let take_max tas = (* extrait maximum *)
  swap tas 0 (tas.n - 1);
  tas.n <- tas.n - 1;
  descendre tas 0;
  tas.t.(tas.n);;
```

Complexité:

Extraire le maximum

Code pour extraire la racine d'un tas:

```
let take_max tas = (* extrait maximum *)
  swap tas 0 (tas.n - 1);
  tas.n <- tas.n - 1;
  descendre tas 0;
  tas.t.(tas.n);;
```

Complexité: $O(\log(n))$.

Remarques:

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

Mettre à jour un élément

Pour mettre à jour un élément:

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

Pour mettre à jour un élément:

Mettre à jour un élément

Pour mettre à jour un élément:

```
let update tas i new_e =
  let prev_e = tas.t.(i) in
  tas.t.(i) <- new_e;
  if prev_e < new_e then monter tas i
  else descendre tas i;;
```

Complexité:

Mettre à jour un élément

Pour mettre à jour un élément:

```
let update tas i new_e =
  let prev_e = tas.t.(i) in
  tas.t.(i) <- new_e;
  if prev_e < new_e then monter tas i
  else descendre tas 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	AVL
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`	$O(n\log(n))$	$O(n)$	$O(n\log(n))$

Tri par tas

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.

Tri par tas

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.

1. FP implémenté avec tas $\implies$ tri en $O(n\log(n))$
2. FP implémenté avec AVL/ARN $\implies$ tri en $O(n\log(n))$
3. ...

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 tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  done;;
```

Correction:

Tri par tas

Code pour trier avec un tas:

```
let tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  done;;
```

Correction: « au début de la boucle, les éléments de `t` d'indices $n - i$ à $n - 1$ sont les i plus grands éléments triés ».

Tri par tas

Code pour trier avec un tas:

```
let tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  done;;
```

Complexité:

Tri par tas

Code pour trier avec un tas:

```
let tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  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 tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  done;;
```

Complexité **en mémoire** (espace utilisé en plus de l'entrée):

Tri par tas

Code pour trier avec un tas:

```
let tri_tas tableau =
  let tas = array_to_tas tableau in
  for i = 0 to Array.length tableau - 1 do
    take_max tas
  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.

Tri par tas

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

Tri par tas

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:

Tri par tas

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 $k \leq \dfrac{n}{\log(n)}$.

File de priorité avec un ABR

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`.