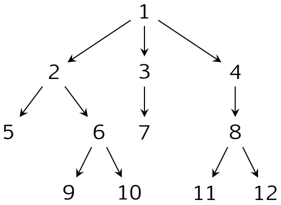
Arbres binaires

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

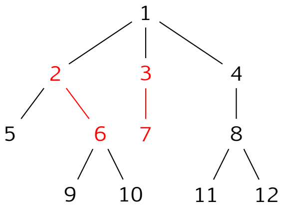
Définition d'un arbre binaire

Théorie des graphes : un arbre est un graphe connexe acyclique enraciné.



Définition d'un arbre binaire

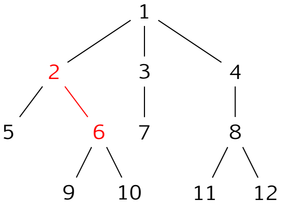
Théorie des graphes : un arbre est un graphe connexe acyclique enraciné.



Pères et fils : le sommet 3 est le père de 7, le sommet 6 est le fils de 2.
Il est d'usage de dessiner un arbre en plaçant un père au dessus de ses fils → l'orientation du graphe devient implicite.

Définition d'un arbre binaire

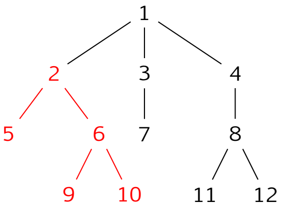
Théorie des graphes : un arbre est un graphe connexe acyclique enraciné.



Pères et fils : le sommet 3 est le père de 7, le sommet 6 est le fils de 2.
Il est d'usage de dessiner un arbre en plaçant un père au dessus de ses fils → l'orientation du graphe devient implicite.
Dans ce contexte, on parle de *nœud* au lieu de sommet. Un nœud qui n'a pas de fils est une feuille ou nœud externe, les autres sont des nœuds internes.

Définition d'un arbre binaire

Théorie des graphes : un arbre est un graphe connexe acyclique enraciné.



Pères et fils : le sommet 3 est le père de 7, le sommet 6 est le fils de 2.
Il est d'usage de dessiner un arbre en plaçant un père au dessus de ses fils → l'orientation du graphe devient implicite.
Dans ce contexte, on parle de *nœud* au lieu de sommet. Un nœud qui n'a pas de fils est une feuille ou nœud externe, les autres sont des nœuds internes.
Chaque nœud est la racine d'un arbre constitué de lui-même et de l'ensemble de ses descendants ; on parle alors de sous-arbre de l'arbre initial.

Définition d'un arbre binaire

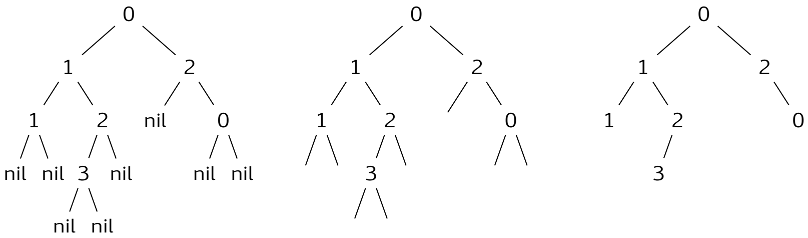
arité d'un nœud (ou degré sortant) : nombre de branches qui en partent.
arbres binaires : chaque nœud a pour arité 0, 1 ou 2.

Définition d'un arbre binaire

arité d'un nœud (ou degré sortant) : nombre de branches qui en partent.
arbres binaires : chaque nœud a pour arité 0, 1 ou 2.

Un ensemble E étant donné, on convient que :

- nil est un arbre binaire sur E appelé l'arbre vide ;
- si $x \in E$ et si F_g et F_d sont deux arbres binaires étiquetés par E, alors $A = (F_g, x, F_d)$ est un arbre binaire étiqueté par E.

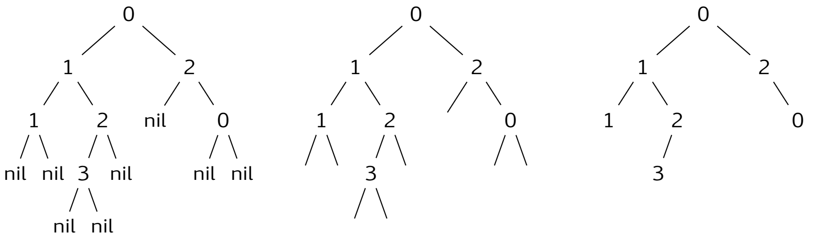


Définition d'un arbre binaire

arité d'un nœud (ou degré sortant) : nombre de branches qui en partent.
arbres binaires : chaque nœud a pour arité 0, 1 ou 2.

Un ensemble E étant donné, on convient que :

- nil est un arbre binaire sur E appelé l'arbre vide ;
- si $x \in E$ et si F_g et F_d sont deux arbres binaires étiquetés par E, alors $A = (F_g, x, F_d)$ est un arbre binaire étiqueté par E.



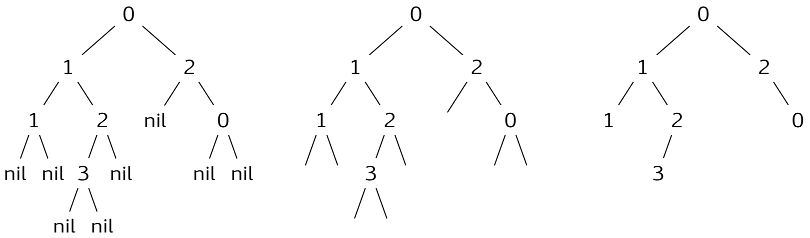
Suivant la représentation choisie une feuille désignera l'arbre vide (**`nil`**) ou un nœud dont les fils gauche et droit sont vides.

Définition d'un arbre binaire

arité d'un nœud (ou degré sortant) : nombre de branches qui en partent.
arbres binaires : chaque nœud a pour arité 0, 1 ou 2.

Un ensemble E étant donné, on convient que :

- nil est un arbre binaire sur E appelé l'arbre vide ;
- si $x \in E$ et si F_g et F_d sont deux arbres binaires étiquetés par E, alors $A = (F_g, x, F_d)$ est un arbre binaire étiqueté par E.



$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

```
type 'a arbre = Nil | Noeud of ('a arbre * 'a * 'a arbre) ;;
```

Fonctions inductives

Preuve par induction structurelle

Soit $\mathscr{R}$ une assertion définie sur l'ensemble $\mathscr{A}$ des arbres étiquetés par E. On suppose que :

- $\mathscr{R}(\text{nil})$ est vraie ;
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}^2$, l'implication $\big(\mathscr{R}(F_g) \text{ et } \mathscr{R}(F_d)\big) \Longrightarrow \mathscr{R}(F_g, x, F_d)$ est vraie ;

Alors la propriété $\mathscr{R}(A)$ est vraie pour tout arbre A de $\mathscr{A}$.

Fonctions inductives

Preuve par induction structurelle

De nombreuses fonctions $f : \mathscr{A} \to F$ se définissent par la donnée d'un élément $a \in F$, d'une fonction $\varphi : F \times E \times F \to F$ et les relations :

- $f(\text{nil}) = a$;
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}^2, f(F_g, x, F_d) = \varphi(f(F_g), x, f(F_d))$.

Fonctions inductives

Preuve par induction structurelle

De nombreuses fonctions $f : \mathscr{A} \to F$ se définissent par la donnée d'un élément $a \in F$, d'une fonction $\varphi : F \times E \times F \to F$ et les relations :

- $f(\text{nil}) = a$;
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}^2, f(F_g, x, F_d) = \varphi(f(F_g), x, f(F_d))$.

La taille $|A|$ d'un arbre A est définie inductivement par les relations :

- $|\text{nil}| = 0$;
- Si $A = (F_g, x, F_d)$ alors $|A| = 1 + |F_g| + |F_d|$.

```
let rec taille = function
  | Nil                -> 0
  | Noeud (fg, _, fd) -> 1 + taille fg + taille fd ;;
```

$|A|$ est le nombre de nœuds d'un arbre.

Fonctions inductives

Preuve par induction structurelle

De nombreuses fonctions $f : \mathscr{A} \to F$ se définissent par la donnée d'un élément $a \in F$, d'une fonction $\varphi : F \times E \times F \to F$ et les relations :

- $f(\text{nil}) = a$;
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}^2, f(F_g, x, F_d) = \varphi(f(F_g), x, f(F_d))$.

La hauteur $h(A)$ d'un arbre A se définit inductivement par les relations :

- $h(\text{nil}) = -1$;
- Si $A = (F_g, x, F_d)$ alors $h(A) = 1 + \max(h(F_g), h(F_d))$.

```
let rec hauteur = function
  | Nil              -> -1
  | Noeud (fg, _, fd) -> 1 + max (hauteur fg) (hauteur fd) ;;
```

$h(A)$ est la longueur du plus long chemin entre la racine et une feuille (la profondeur maximale d'un nœud).

Fonctions inductives

Preuve par induction structurelle

Soit A un arbre binaire. Alors $h(A)+1 \leqslant |A| \leqslant 2^{h(A)+1}-1$.

Fonctions inductives

Preuve par induction structurelle

Soit A un arbre binaire. Alors $h(A) + 1 \leqslant |A| \leqslant 2^{h(A)+1} - 1$.

On raisonne par induction structurelle.

- Si $A = \text{nil}$, $|A| = 0$ et $h(A) = -1$; le résultat annoncé est bien vérifié.

Fonctions inductives

Preuve par induction structurelle

Soit A un arbre binaire. Alors $h(A)+1 \leqslant |A| \leqslant 2^{h(A)+1}-1$.

On raisonne par induction structurelle.

- Si $A = \text{nil}$, $|A| = 0$ et $h(A) = -1$; le résultat annoncé est bien vérifié.
- Si $A = (F_g, x, F_d)$, supposons le résultat acquis pour F_g et F_d.

Fonctions inductives

Preuve par induction structurelle

Soit A un arbre binaire. Alors $h(A)+1 \leqslant |A| \leqslant 2^{h(A)+1}-1$.

On raisonne par induction structurelle.

- Si $A = \text{nil}$, $|A| = 0$ et $h(A) = -1$; le résultat annoncé est bien vérifié.
- Si $A = (F_g, x, F_d)$, supposons le résultat acquis pour F_g et F_d.

$$\begin{aligned} |A| = 1 + |F_g| + |F_d| &\geqslant 1 + h(F_g) + 1 + h(F_d) + 1 \\ &\geqslant 2 + \max(h(F_g), h(F_d)) = 1 + h(A) \end{aligned}$$

Fonctions inductives

Preuve par induction structurelle

Soit A un arbre binaire. Alors $h(A)+1 \leqslant |A| \leqslant 2^{h(A)+1}-1$.

On raisonne par induction structurelle.

- Si $A = \text{nil}$, $|A| = 0$ et $h(A) = -1$; le résultat annoncé est bien vérifié.
- Si $A = (F_g, x, F_d)$, supposons le résultat acquis pour F_g et F_d.

$$\begin{aligned} |A| = 1 + |F_g| + |F_d| &\geqslant 1 + h(F_g) + 1 + h(F_d) + 1 \\ &\geqslant 2 + \max(h(F_g), h(F_d)) = 1 + h(A) \end{aligned}$$

$$\begin{aligned} |A| = 1 + |F_g| + |F_d| &\leqslant 2^{h(F_g)+1} + 2^{h(F_d)+1} - 1 \\ &\leqslant 2 \times 2^{\max(h(F_g),h(F_d))+1} - 1 = 2^{h(A)+1} - 1 \end{aligned}$$

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

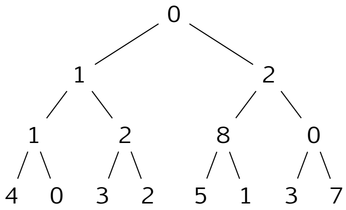
Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Cas optimal : $h(A) = \log(|A|+1)-1 \rightarrow$ arbres binaires complets.
Pour que $A = (F_g, x, F_d)$ soit complet il faut et il suffit que F_g et F_d soient complets et de même hauteur.
Un arbre binaire est complet si et seulement si toutes ses feuilles sont à la même profondeur.



$$|A| = 15 \quad \text{et} \quad h(A) = 3.$$

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Certaines catégories d'arbres garantissent l'équilibrage : arbres rouge-noir, arbres AVL, etc.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

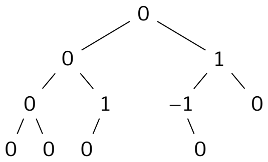
Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

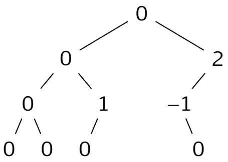
Le déséquilibre d'un arbre $A = (F_g, x, F_d)$ est égal à $h(F_g) - h(F_d)$.

Un arbre binaire A est un arbre *AVL* lorsqu'il est vide ou égal à (F_g, x, F_d) avec :

- F_g et F_d sont des arbres AVL ;
- le déséquilibre de A est égal à $-1, 0$ ou 1.

Le déséquilibre de chaque sous-arbre est égal à $-1, 0$ ou 1.





Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Tout arbre AVL est équilibré.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Tout arbre AVL est équilibré.

On considère la suite de FIBONACCI $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$.
On prouve par induction structurelle que tout arbre AVL A de hauteur h contient au moins f_h nœuds.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Tout arbre AVL est équilibré.

On considère la suite de FIBONACCI $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$.
On prouve par induction structurelle que tout arbre AVL A de hauteur h contient au moins f_h nœuds.

- Si $A = \text{nil}$ alors $|A| = 0 = f_0$.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Tout arbre AVL est équilibré.

On considère la suite de Fibonacci $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$.
On prouve par induction structurelle que tout arbre AVL A de hauteur h contient au moins f_h nœuds.

- Si $A = \text{nil}$ alors $|A| = 0 = f_0$.
- Si $A = (F_g, x, F_d)$, l'un des deux sous-arbres F_g ou F_d est de hauteur $h-1$ donc contient au moins f_{h-1} nœuds, l'autre est au moins de hauteur $h-2$ donc contient au moins f_{h-2} nœuds.
  Donc A contient au moins $f_{h-1} + f_{h-2} + 1 = f_h + 1$ nœuds.

Arbres binaires équilibrés

si A est un arbre binaire alors $\log(|A|+1)-1 \leqslant h(A) \leqslant |A|-1$.

Un arbre binaire A est dit équilibré lorsque $h(A) = \mathrm{O}(\log(|A|))$.

Tout arbre AVL est équilibré.

On considère la suite de Fibonacci $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$.
On prouve par induction structurelle que tout arbre AVL A de hauteur h contient au moins f_h nœuds.

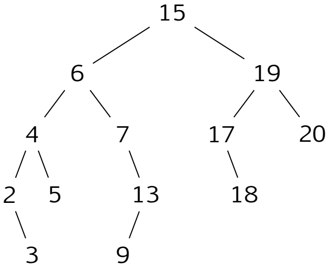
- Si $A = \text{nil}$ alors $|A| = 0 = f_0$.
- Si $A = (F_g, x, F_d)$, l'un des deux sous-arbres F_g ou F_d est de hauteur $h-1$ donc contient au moins f_{h-1} nœuds, l'autre est au moins de hauteur $h-2$ donc contient au moins f_{h-2} nœuds.
  Donc A contient au moins $f_{h-1} + f_{h-2} + 1 = f_h + 1$ nœuds.

Sachant que $\varphi^h = \mathrm{O}(f_h)$ avec $\varphi = \dfrac{1+\sqrt{5}}{2}$ on en déduit : $\varphi^{h(A)} = \mathrm{O}(|A|)$, soit $h(A) = \mathrm{O}(\log|A|)$.

Arbres binaires de recherche

On considère un ensemble ordonné de clés C et un ensemble de valeurs V, et on utilise des arbres binaires étiquetés par $E = C \times V$.
un arbre binaire A est un arbre binaire de recherche s′il est vide ou égal à $(F_g, (c, v), F_d)$ où :

- F_g et F_d sont des arbres binaires de recherche ;
- toute clé de F_g est inférieure ou égale à c ;
- toute clé de F_d est supérieure ou égale à c.



Tout nœud est associé à une clé supérieure ou égale à toute clé de son fils gauche, et inférieure ou égale à toute clé de son fils droit.

Arbres binaires de recherche

On considère un ensemble ordonné de clés C et un ensemble de valeurs V, et on utilise des arbres binaires étiquetés par $E = C \times V$.
un arbre binaire A est un arbre binaire de recherche s'il est vide ou égal à $(F_g, (c, v), F_d)$ où :

- F_g et F_d sont des arbres binaires de recherche ;
- toute clé de F_g est inférieure ou égale à c ;
- toute clé de F_d est supérieure ou égale à c.

Dans la suite du cours, on utilisera le type :

```
type ('a, 'b) data = {Key : 'a; Value : 'b} ;;
```

et les ABR seront représentés par le type *('a, 'b) data arbre*.

Arbre binaire de recherche

Parcours infixe

La propriété des ABR permet d'afficher toutes les valeurs de l'arbre par ordre croissant de clé à l'aide d'un *parcours infixe* : exploration en profondeur de l'arbre (F_g, x, F_d) dans l'ordre : $F_g \longrightarrow x \longrightarrow F_d$.

Arbre binaire de recherche

Parcours infixe

La propriété des ABR permet d'afficher toutes les valeurs de l'arbre par ordre croissant de clé à l'aide d'un *parcours infixe* : exploration en profondeur de l'arbre (F_g, x, F_d) dans l'ordre : $F_g \longrightarrow x \longrightarrow F_d$.

Preuve par induction :

- si $A = \text{nil}$, il n'y a rien à prouver ;
- si $A = (F_g, x, F_d)$, on suppose que les parcours infixes de F_g et de F_d se font par ordre de clés croissantes.
  Toute clé de F_g est inférieure à la clé de x et toute clé de F_d supérieure à cette dernière, donc le parcours $F_g \longrightarrow x \longrightarrow F_d$ est toujours effectué par ordre croissant de clé.

Arbre binaire de recherche

Parcours infixe

La propriété des ABR permet d'afficher toutes les valeurs de l'arbre par ordre croissant de clé à l'aide d'un *parcours infixe* : exploration en profondeur de l'arbre (F_g, x, F_d) dans l'ordre : $F_g \longrightarrow x \longrightarrow F_d$.

Si **traitement** est une fonction de type *'b -> unit*, le parcours d'un ABR prendra la forme :

```
let rec parcours_infixe = function
  | Nil -> ()
  | Noeud (fg, x, fd) -> parcours_infixe fg ;
                         traitement x.Value ;
                         parcours_infixe fd ;;
```

Le coût temporel de ce parcours est un $\Theta(n)$ lorsque $n = |A|$.

Requêtes dans un ABR

Opérations usuelles sur un ABR :

- *recherche* d'un valeur associée à une clé donnée ;
- recherche de la valeur associée à la clé *maximale* (ou *minimale*) ;
- recherche du *successeur* d'une clé c ;
- recherche du *prédécesseur* d'une clé ;
- l'*insertion* et *suppression* d'un nouveau couple clé/valeur.

Requêtes dans un ABR

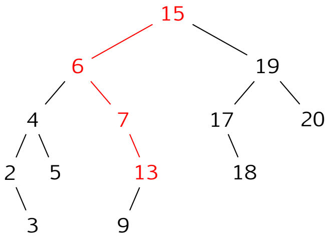
Recherche d'une clé

Recherche d'une clé k dans l'ABR $A = (F_g, (c, v), F_d)$:

- si $k = c$, retourner v ;
- si $k < c$, rechercher k dans F_g ;
- si $k > c$, rechercher k dans F_d.

```
let rec recherche k = function
  | Nil                              -> raise Not_found
  | Noeud (_, x, _) when k = x.Key   -> x.Value
  | Noeud (fg, x, _) when k < x.Key -> recherche k fg
  | Noeud (_, _, fd)                 -> recherche k fd ;;
```

Recherche de la clé $k = 13$:



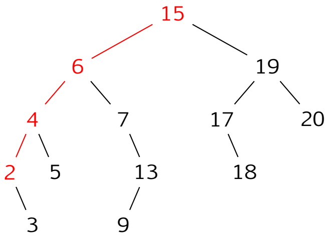
Requêtes dans un ABR

Recherche de la clé minimale / maximale

La recherche de la clé minimale se poursuit dans le fils gauche tant que ce dernier n'est pas vide :

```
let rec minimum = function
  | Nil                  -> raise Not_found
  | Noeud (Nil, x, _) -> x.Value
  | Noeud (fg, _, _)  -> minimum fg ;;
```

Recherche de la clé minimale :



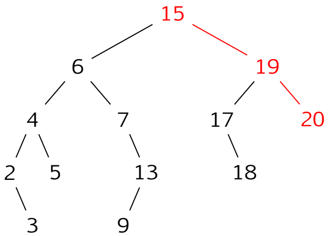
Requêtes dans un ABR

Recherche de la clé minimale / maximale

La recherche de la clé maximale se poursuit dans le fils droit tant que ce dernier n'est pas vide :

```
let rec maximum = function
  | Nil                  -> raise Not_found
  | Noeud (_, x, Nil) -> x.Value
  | Noeud (_, _, fd)  -> maximum fd ;;
```

Recherche de la clé maximale :



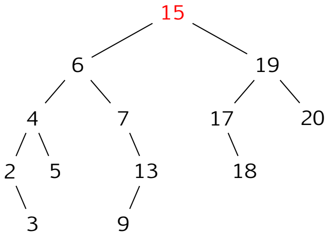
Requêtes dans un ABR

Recherche du prédécesseur / successeur

Successeur de la clé *k* : la plus petite des clés *c* vérifiant : $k < c$.

```
let rec successeur k = function
  | Nil                                -> raise Not_found
  | Noeud (_, x, fd) when x.Key <= k -> successeur k fd
  | Noeud (fg, x, _)                   -> try successeur k fg
                                          with Not_found -> x.Value ;;
```

Le successeur de 13 est 15 : il n'y a pas de successeur possible dans le fils gauche.



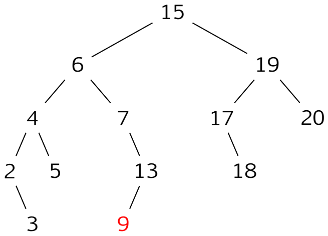
Requêtes dans un ABR

Recherche du prédécesseur / successeur

Prédécesseur de la clé k : la plus grande des clés c vérifiant : $c < k$.

```
let rec predecesseur k = function
  | Nil                                   -> raise Not_found
  | Noeud (fg, x, _) when x.Key >= k -> predecesseur k fg
  | Noeud (_, x, fd)                      -> try predecesseur k fd
                                             with Not_found -> x.Value ;;
```

Le prédécesseur de 10 est 9 : ont tout d'abord été envisagés 6 et 7 avant de trouver 9.



Requêtes dans un ABR

Coût d'une requête

Toutes ces requêtes ont un coût temporel en $\mathrm{O}(h(A))$, ce qui explique tout l'intérêt qu'il peut y avoir à ce que l'arbre binaire de recherche soit équilibré :

- dans un ABR quelconque d'ordre $n = |A|$, le coût d'une requête est un $\mathrm{O}(n)$;
- dans le cas d'un ABR équilibré, le coût d'une requête est un $\mathrm{O}(\log n)$.

Insertion dans un ABR

Insertion au niveau des feuilles

Pour insérer le couple $(k, u) \in C \times V$ au niveau des feuilles on procède ainsi :

- si $A = \text{nil}$, on retourne l'arbre $(\text{nil}, (k, u), \text{nil})$;
- si $A = (F_g, (c, v), F_d)$ alors :
  - si $c = k$ on remplace le couple (c, v) par (k, u) et on insère (c, v) dans F_g ou F_d ;
  - si $c > k$ on insère (k, u) dans F_g ;
  - si $c < k$ on insère (k, u) dans F_d.

```
let rec insere_feuille y = function
  | Nil                                 -> Noeud (Nil, y, Nil)
  | Noeud (fg, x, fd) when x.Key = y.Key -> Noeud (fg, y, insere_feuille x fd)
  | Noeud (fg, x, fd) when x.Key < y.Key -> Noeud (insere_feuille y fg, x, fd)
  | Noeud (fg, x, fd)                   -> Noeud (fg, x, insere_feuille y fd) ;;
```

Insertion dans un ABR

Insertion au niveau de la racine

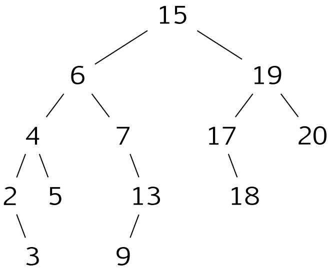
Pour insérer le nouvel élément à la racine on partitionne l'arbre en plaçant tous les éléments inférieurs à la nouvelle clé dans le fils gauche et les autres dans le fils droit :

```
let insere_racine y a =
  let rec partition = function
    | Nil                                     -> Nil, Nil
    | Noeud (fg, x, fd) when x.Key < y.Key -> let a1, a2 = partition fd in
                                                Noeud (fg, x, a1), a2
    | Noeud (fg, x, fd)                       -> let a1, a2 = partition fg in
                                                a1, Noeud (a2, x, fd)
  in let fg, fd = partition a in Noeud (fg, y, fd) ;;
```

Insertion dans un ABR

Comparaison des deux méthodes

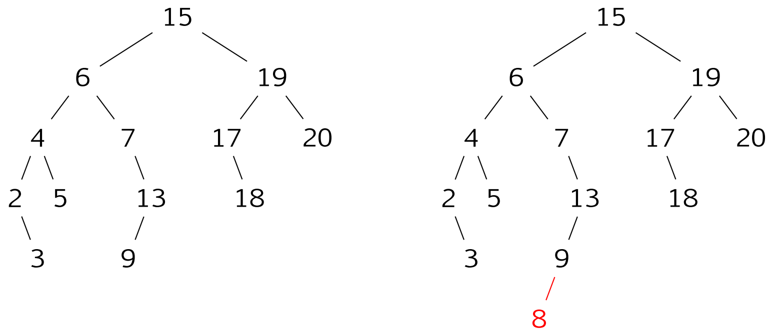
Insertion de la clé 8 dans l'arbre ci-dessous :



Insertion dans un ABR

Comparaison des deux méthodes

Insertion de la clé 8 dans l'arbre ci-dessous :



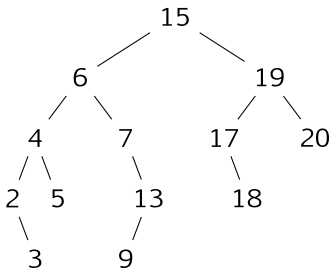
Insertion au niveau des feuilles.

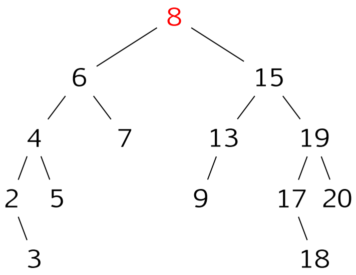
Cette méthode a un coût en $\mathrm{O}(h(A))$.

Insertion dans un ABR

Comparaison des deux méthodes

Insertion de la clé 8 dans l'arbre ci-dessous :





Insertion au niveau de la racine.

Cette méthode a un coût en $\mathrm{O}(h(A))$.

Suppression dans un ABR

Pour supprimer une clé k on procède ainsi :

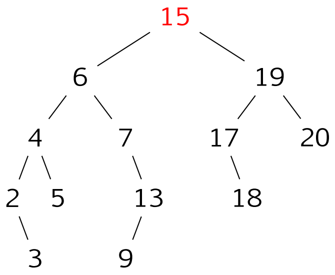
- si $k < c$, on supprime un élément de clé k dans F_g ;
- si $k > c$, on supprime un élément de clé k dans F_d ;
- si $k = c$, alors :
  - si $F_g = \text{nil}$ on renvoie F_d ;
  - si $F_d = \text{nil}$ on renvoie F_g ;
  - sinon, on supprime de F_d une clé minimale m et on renvoie (F_g, m, F'_d).

```
let rec supprime_min = function
  | Nil                  -> failwith "supprime_min"
  | Noeud (Nil, m, fd) -> m, fd
  | Noeud (fg, x, fd)  -> let m, f = supprime_min fg in m, Noeud (f, x, fd) ;;

let rec supprime k = function
    | Nil                    -> raise Not_found
    | Noeud (fg, x, fd) when x.Key < k -> Noeud (fg, x, supprime k fd)
    | Noeud (fg, x, fd) when x.Key > k -> Noeud (supprime k fg, x, fd)
    | Noeud (Nil, x, fd) -> fd
    | Noeud (fg, x, Nil) -> fg
    | Noeud (fg, x, fd)  -> let m, f = supprime_min fd in Noeud (fg, m, f) ;;
```

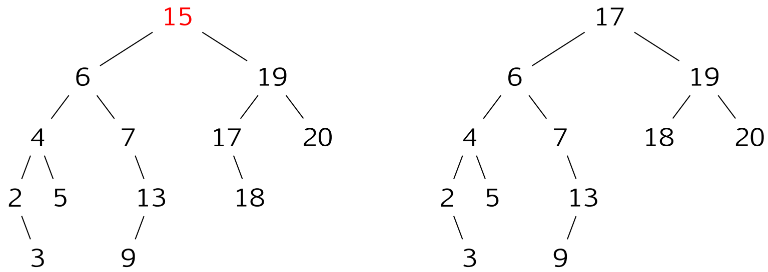
Suppression dans un ABR

Suppression de la clé 15 dans l'arbre ci-dessous :



Suppression dans un ABR

Suppression de la clé 15 dans l'arbre ci-dessous :



Cette méthode a un coût en $\mathrm{O}(h(A))$.

Le problème du déséquilibre

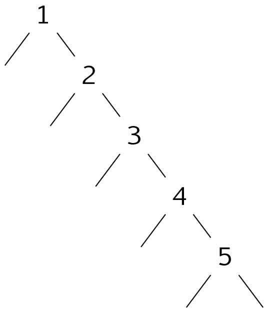
Toutes les fonctions de requêtes, d'insertion et de suppression dans un ABR ont un coût en $O(h(A))$, autrement dit, en posant $n = |A|$:

- un coût linéaire $O(n)$ dans le cas d'un arbre de recherche quelconque ;
- un coût logarithmique $O(\log n)$ dans le cas d'un arbre de recherche maintenu équilibré.

Le problème du déséquilibre

Toutes les fonctions de requêtes, d'insertion et de suppression dans un ABR ont un coût en $\mathrm{O}(h(A))$, autrement dit, en posant $n = |A|$:

- un coût linéaire $\mathrm{O}(n)$ dans le cas d'un arbre de recherche quelconque ;
- un coût logarithmique $\mathrm{O}(\log n)$ dans le cas d'un arbre de recherche maintenu équilibré.

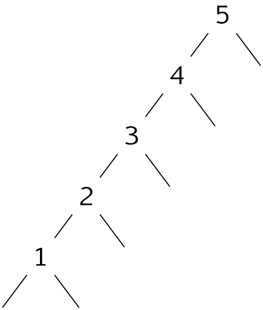


Insertion de 1, 2, 3, 4, 5 au niveau des feuilles.

Le problème du déséquilibre

Toutes les fonctions de requêtes, d'insertion et de suppression dans un ABR ont un coût en $\mathrm{O}(h(A))$, autrement dit, en posant $n = |A|$:

- un coût linéaire $\mathrm{O}(n)$ dans le cas d'un arbre de recherche quelconque ;
- un coût logarithmique $\mathrm{O}(\log n)$ dans le cas d'un arbre de recherche maintenu équilibré.



Insertion de 1, 2, 3, 4, 5 au niveau de la racine.

Le problème du déséquilibre

On peut démontrer que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable : si un arbre binaire de recherche est créé en insérant n clés distinctes au niveau des feuilles dans un ordre aléatoire, il existe c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c \log n$.

Le problème du déséquilibre

On peut démontrer que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable : si un arbre binaire de recherche est créé en insérant n clés distinctes au niveau des feuilles dans un ordre aléatoire, il existe c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c \log n$.

Par exemple, les six arbres que l'on obtient en insérant dans un ordre arbitraire les trois entiers 1, 2 et 3 sont :

```
  1 – 2 – 3  |  3 – 1 – 2  |  2 – 3 – 1  |  2 – 1 – 3  |  1 – 3 – 2  |  3 – 2 – 1
-------------+-------------+-------------+-------------+-------------+-------------
     1       |       3     |      2      |      2      |    1        |        3
    / \      |      / \    |     / \     |     / \     |   / \       |       / \
       2     |     1       |    1   3    |    1   3    |      3      |      2
      / \    |    / \      |   / \ / \   |   / \ / \   |     / \     |     / \
         3   |       2     |             |             |    2        |    1
        / \  |      / \    |             |             |   / \       |   / \
```

Le problème du déséquilibre

On peut démontrer que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable : si un arbre binaire de recherche est créé en insérant n clés distinctes au niveau des feuilles dans un ordre aléatoire, il existe c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c \log n$.
Par exemple, les six arbres que l'on obtient en insérant dans un ordre arbitraire les trois entiers 1, 2 et 3 sont :

```
  1-2-3   |  3-1-2   |  2-3-1   |  2-1-3   |  1-3-2   |  3-2-1
----------+----------+----------+----------+----------+----------
  1       |      3   |    2     |    2     |  1       |      3
 / \      |     / \  |   / \    |   / \    | / \      |     / \
    2     |    1     |  1   3   |  1   3   |    3     |    2
   / \    |   / \    | / \ / \  | / \ / \  |   / \    |   / \
      3   |      2   |          |          |  2       |  1
     / \  |     / \  |          |          | / \      | / \
```

Ceci ne revient pas à supposer que chaque arbre binaire de recherche à n nœuds est équiprobable : la hauteur moyenne d'un arbre binaire de recherche de taille n s'ils sont tous équiprobables est équivalente à $2\sqrt{\pi n}$.

Le problème du déséquilibre

On peut démontrer que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable : si un arbre binaire de recherche est créé en insérant n clés distinctes au niveau des feuilles dans un ordre aléatoire, il existe c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c \log n$.

Il est néanmoins possible de garantir une complexité dans le pire des cas en $O(\log n)$ à condition de maintenir en place une structure d'arbre qui garantisse l'équilibrage : arbres AVL, arbres rouge-noir, etc.

ABR et dictionnaires

Dans le cours de première année a été étudiée la notion de *table d'association*, ou dictionnaire : si C désigne l'ensemble des clés et V l'ensemble des valeurs, une table d'association T est un sous-ensemble de $C \times V$ tel que pour toute clé $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$.

Une table d'association supporte en général les opérations suivantes :

- *ajout* d'une nouvelle paire $(c, v) \in C \times V$ dans T ;
- *suppression* d'une paire (c, v) de T ;
- *lecture* de la valeur associée à une clé dans T.

ABR et dictionnaires

Dans le cours de première année a été étudiée la notion de *table d'association*, ou dictionnaire : si C désigne l'ensemble des clés et V l'ensemble des valeurs, une table d'association T est un sous-ensemble de $C \times V$ tel que pour toute clé $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$.

Une table d'association supporte en général les opérations suivantes :

- *ajout* d'une nouvelle paire $(c, v) \in C \times V$ dans T ;
- *suppression* d'une paire (c, v) de T ;
- *lecture* de la valeur associée à une clé dans T.

Une table de hachage permet la réalisation d'une structure impérative de dictionnaire, avec les coûts :

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

ABR et dictionnaires

Dans le cours de première année a été étudiée la notion de *table d'association*, ou dictionnaire : si C désigne l'ensemble des clés et V l'ensemble des valeurs, une table d'association T est un sous-ensemble de $C \times V$ tel que pour toute clé $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$.

Une table d'association supporte en général les opérations suivantes :

- *ajout* d'une nouvelle paire $(c, v) \in C \times V$ dans T ;
- *suppression* d'une paire (c, v) de T ;
- *lecture* de la valeur associée à une clé dans T.

Un ABR permet la réalisation d'une structure persistante de dictionnaire, avec les coûts :

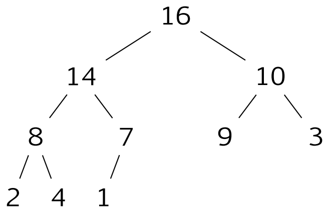
pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

à condition de maintenir équilibrés les ABR.

Tas binaire

Tas-min et tas-max

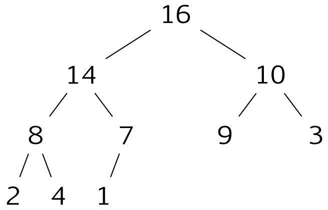
Un arbre binaire est dit parfait lorsque tous les niveaux hiérarchiques sont remplis sauf éventuellement le dernier, partiellement rempli de la gauche vers la droite.



Tas binaire

Tas-min et tas-max

Un arbre binaire est dit parfait lorsque tous les niveaux hiérarchiques sont remplis sauf éventuellement le dernier, partiellement rempli de la gauche vers la droite.



Si A est un arbre parfait, alors

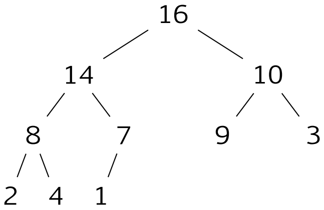
$$1 + 2 + \cdots + 2^{h(A)-1} < |A| \leqslant 1 + 2 + \cdots + 2^{h(A)}$$

donc $2^{h(A)} \leqslant |A| < 2^{h(A)+1}$; c'est un arbre équilibré.

Tas binaire

Tas-min et tas-max

Un arbre binaire est dit parfait lorsque tous les niveaux hiérarchiques sont remplis sauf éventuellement le dernier, partiellement rempli de la gauche vers la droite.

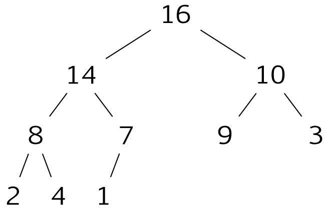


On appelle tas-max un arbre parfait étiqueté tel que l'étiquette de chaque nœud autre que la racine soit inférieure ou égale à l'étiquette de son père.

Tas binaire

Tas-min et tas-max

Un arbre binaire est dit parfait lorsque tous les niveaux hiérarchiques sont remplis sauf éventuellement le dernier, partiellement rempli de la gauche vers la droite.



On appelle tas-max un arbre parfait étiqueté tel que l'étiquette de chaque nœud autre que la racine soit inférieure ou égale à l'étiquette de son père.

Un tas-min possède la propriété opposée : l'étiquette de chaque nœud autre que la racine est supérieure ou égale à l'étiquette de son père.

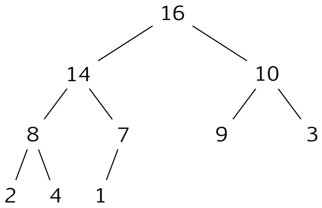
Tas binaire

Implémentation

On représente un tas par un tableau en utilisant la numérotation :

- la racine porte le numéro 1 ;
- si un nœud porte le numéro k, son fils gauche porte le numéro $2k$ et son fils droit le numéro $2k+1$.

Le père d'un nœud de numéro k porte donc le numéro $\left\lfloor \frac{k}{2} \right\rfloor$.



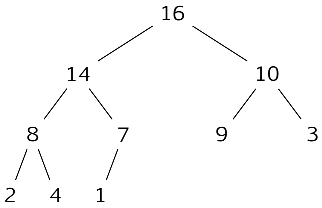
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Tas binaire

Implémentation

Sachant que les tableaux Caml sont indexés à partir de 0, on décale les indices :

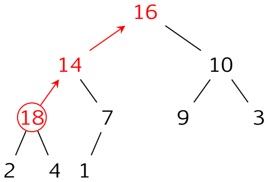
- la racine est stockée dans la case d'indice 0 ;
- si un nœud est stocké dans la case d'indice k, son fils gauche est stocké dans la case d'indice $2k+1$ et son fils droit dans la case d'indice $2k+2$;
- si un fils est stocké dans le case d'indice k, son père est stocké dans la case d'indice $\left\lfloor \frac{k-1}{2} \right\rfloor$.



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

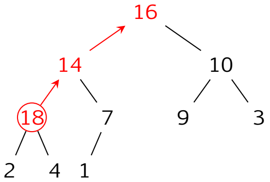
Préservation de la structure de tas

Comment reconstituer un tas après qu'un élément a été modifié ?
Si la nouvelle valeur de cet élément est supérieure à la précédente, il se peut que cet élément doive monter dans l'arbre pour reconstituer un tas.



Préservation de la structure de tas

Comment reconstituer un tas après qu'un élément a été modifié ?
Si la nouvelle valeur de cet élément est supérieure à la précédente, il se peut que cet élément doive monter dans l'arbre pour reconstituer un tas.



On permute l'élément avec son père tant que le tas n'est pas reconstitué :

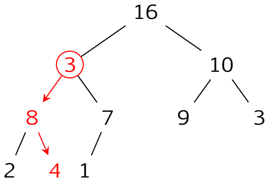
```
let swap t i j =
  let x = t.(i) in t.(i) <- t.(j) ; t.(j) <- x ;;

let monte t k =
  let rec aux = function
    | 0 -> ()
    | i -> let j = (i-1)/2 in
           if t.(i) > t.(j) then (swap t i j ; aux j)
  in aux k ;;
```

Le temps d'exécution est un $\mathrm{O}(h(T)) = \mathrm{O}(\log n)$.

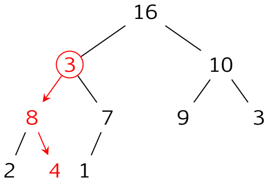
Préservation de la structure de tas

Comment reconstituer un tas après qu'un élément a été modifié ?
Si la nouvelle valeur de cet élément est inférieure à la précédente, il se peut que cet élément doive descendre dans l'arbre.



Préservation de la structure de tas

Comment reconstituer un tas après qu'un élément a été modifié ?
Si la nouvelle valeur de cet élément est inférieure à la précédente, il se peut que cet élément doive descendre dans l'arbre.



Pour la descente, il faut permuter l'élément modifié avec le plus grand de ses fils.

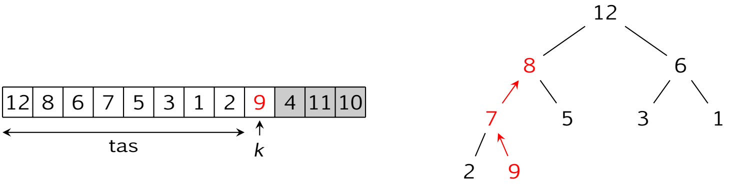
```
let descend t n k =
  let rec aux = function
    | i when 2*i+2 > n -> ()
    | i -> let j = if 2*i+2 = n || t.(2*i+1) > t.(2*i+2) then 2*i+1 else 2*i+2 in
          if t.(i) < t.(j) then (swap t i j ; aux j)
  in aux k ;;
```

Le temps d'exécution est un $\mathrm{O}(h(T)) = \mathrm{O}(\log n)$.

Construction d'un tas

Première méthode

On maintient l'invariant : « $t[0..k]$ est un tas-max » en faisant remonter l'élément t_k dans le tas situé à sa gauche à l'aide de la fonction **monte**.

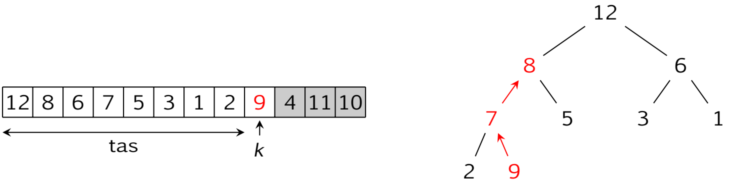



```
let cree_tas t =
  for k = 1 to vect_length t - 1 do monte t k done ;;
```

Construction d'un tas

Première méthode

On maintient l'invariant : « $t[0..k]$ est un tas-max » en faisant remonter l'élément t_k dans le tas situé à sa gauche à l'aide de la fonction **monte**.




```
let cree_tas t =
  for k = 1 to vect_length t - 1 do monte t k done ;;
```

Dans le pire des cas chaque remontée aboutit à la racine ; dans ce cas, le nombre de permutations effectuées est égal à :

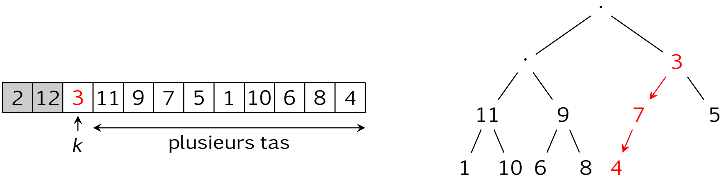
$$\sum_{k=1}^{p-1} k2^k + p(n-2^p+1) = (n+1)p - 2^{p+1} + 2 = \Theta(n\log n)$$

avec $p = \lfloor \log n \rfloor$ (la hauteur du tas).

Construction d'un tas

Deuxième méthode

On maintient l'invariant : « chaque nœud de $t[k..n-1]$ est la racine d'un tas-max » en faisant descendre chacun des nœuds internes de l'arbre.




```
let cree_tas t =
  let n = vect_length t in
  for k = n/2-1 downto 0 do descend t n k done ;;
```

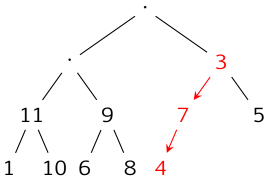
Construction d'un tas

Deuxième méthode

On maintient l'invariant : « chaque nœud de $t[k..n-1]$ est la racine d'un tas-max » en faisant descendre chacun des nœuds internes de l'arbre.

2	12	3	11	9	7	5	1	10	6	8	4

k plusieurs tas



Dans le pire des cas chaque descente aboutit au descendant le plus éloigné et réalise donc h échanges, où h est la hauteur de l'arbre dont il est la racine. Par ailleurs, le nombre de nœuds ayant la hauteur h est majoré par $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ donc le nombre total d'échanges est majoré par :

$$\sum_{h=1}^{p} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \leqslant \frac{p(p+1)}{2} + n \sum_{h=1}^{p} \frac{h}{2^{h+1}} = \Theta(n).$$

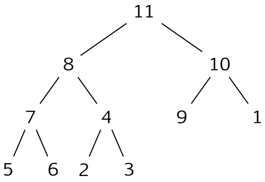
Tri par tas

Tri par tas (*heap sort*) :

- on transforme le tableau en tas-max ;
- on permute t_0 et t_{n-1} puis on reforme le tas-max en faisant descendre t_{n-1} ;
- on réitère ce processus dans la partie non triée.

11	8	10	7	4	9	1	5	6	2	3	12	13	14	15	16	17

tas ← → partie triée




```
let tri_tas t =
  cree_tas t ;
  for k = vect_length t - 1 downto 1 do
    swap t 0 k ;
    descend t k 0
  done ;;
```

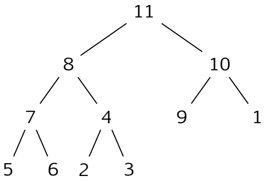
Tri par tas

Tri par tas (*heap sort*) :

- on transforme le tableau en tas-max ;
- on permute t_0 et t_{n-1} puis on reforme le tas-max en faisant descendre t_{n-1} ;
- on réitère ce processus dans la partie non triée.

11	8	10	7	4	9	1	5	6	2	3	12	13	14	15	16	17

tas ← → partie triée



La validité est assurée par l'invariant : « au début de chaque itération le sous-tableau $t[0 \dots k]$ est un tas contenant les $k+1$ plus petits éléments de t et $t[k+1 \dots n-1]$ un tableau trié contenant les $n-k-1$ plus grands éléments de t ».

Le coût total de cette méthode de tri est un $\mathrm{O}(n \log n)$.

Files de priorité

Une file de priorité permet de gérer un ensemble V de valeurs associées chacune à une priorité et supporte les opérations suivantes :

- *création* d'une file de priorité vide ;
- *insertion* d'un couple (v, p) de valeur/priorité ;
- *retrait* de la valeur v associée à la priorité maximale p ;
- *accroissement* de la priorité associée à une valeur donnée.

Files de priorité

Une file de priorité permet de gérer un ensemble V de valeurs associées chacune à une priorité et supporte les opérations suivantes :

- *création* d'une file de priorité vide ;
- *insertion* d'un couple (v, p) de valeur/priorité ;
- *retrait* de la valeur v associée à la priorité maximale p ;
- *accroissement* de la priorité associée à une valeur donnée.

Nous définissons les types :

```
type ('a, 'b) data = {Priority: 'a; Value: 'b} ;;

type 'a tas = {mutable N: int; Tbl: 'a vect} ;;
```

et les deux exceptions :

```
exception Empty ;;
exception Full ;;
```

Files de priorité

On modifie légèrement les définitions des fonctions **monte** et **descend** pour tenir compte du type de données utilisé :

```
let monte t k =
  let rec aux = function
    | 0 -> ()
    | i -> let j = (i-1)/2 in
           if t.(i).Priority > t.(j).Priority then (swap t i j ; aux j)
  in aux k ;;
```

```
let descend t n k =
  let rec aux = function
    | i when 2*i+1 >= n -> ()
    | i -> let j = if 2*i+2 = n || t.(2*i+1).Priority > t.(2*i+2).Priority
                   then 2*i+1 else 2*i+2 in
           if t.(i).Priority < t.(j).Priority then (swap t i j ; aux j)
  in aux k ;;
```

Files de priorité

Création d'une file de priorité :

```
let cree_file n (p, v) = {N = 0 ;
                          Tbl = make_vect n {Priority = p; Value = v}} ;;
```

Files de priorité

Création d'une file de priorité :

```
let cree_file n (p, v) = {N = 0 ;
                          Tbl = make_vect n {Priority = p; Value = v}} ;;
```

Insertion d'un nouvel élément :

```
let ajout (p, v) f =
  if f.N = vect_length f.Tbl then raise Full ;
  f.Tbl.(f.N) <- {Priority = p; Value = v} ;
  monte f.Tbl f.N ;
  f.N <- f.N + 1 ;;
```

Files de priorité

Création d'une file de priorité :

```
let cree_file n (p, v) = {N = 0 ;
                          Tbl = make_vect n {Priority = p; Value = v}} ;;
```

Insertion d'un nouvel élément :

```
let ajout (p, v) f =
  if f.N = vect_length f.Tbl then raise Full ;
  f.Tbl.(f.N) <- {Priority = p; Value = v} ;
  monte f.Tbl f.N ;
  f.N <- f.N + 1 ;;
```

Retrait de l'élément de priorité maximale :

```
let retrait f =
  if f.N = 0 then raise Empty ;
  let v = f.Tbl.(0).Value in
  f.N <- f.N - 1 ;
  f.Tbl.(0) <- f.Tbl.(f.N) ;
  descend f.Tbl f.N 0 ;
  v ;;
```

Files de priorité

Création d'une file de priorité :

```
let cree_file n (p, v) = {N = 0 ;
                          Tbl = make_vect n {Priority = p; Value = v}} ;;
```

Insertion d'un nouvel élément :

```
let ajout (p, v) f =
  if f.N = vect_length f.Tbl then raise Full ;
  f.Tbl.(f.N) <- {Priority = p; Value = v} ;
  monte f.Tbl f.N ;
  f.N <- f.N + 1 ;;
```

Accroissement de la valeur d'une clé :

```
let incr f k c =
  if c < f.Tbl.(k).Priority then failwith "incr" ;
  f.Tbl.(k) <- {Priority = c; Value = f.Tbl.(k).Value} ;
  monte f.Tbl k ;;
```

(En général on a pas besoin de faire décroitre la valeur d'une clé.)