# Chapitre 0

# Arbres Binaires

**Objectifs du chapitre**

Les arbres sont des structures de données omniprésentes en algorithmique, y compris dans les domaines de recherche actuels. Dans ce cours, ils sont avant tout le support de l'application de raisonnements inductifs.
De plus, ces derniers sont utilisés pour construire des structures de données élaborées, adaptées à des algorithmes, des besoins précis.

A Dans une première partie, nous allons rappeler la définition des arbres, la terminologie associée, puis nous nous attarderons sur la notion d'*arbre binaire*, sujet de ce chapitre, avant d'en définir les propriétés usuelles par définition inductive et le raisonnement associé appelé *induction structurelle*.

B Dans un second temps, nous étudierons la structure de donnée classique des *arbres binaires de recherche*, leur *déséquilibre*, la principale difficulté associée avant de s'attarder sur leur usage principal, les *dictionnaires*.

C Dans une troisième partie, nous allons étudier la structure de donnée classique des *tas binaires*, du classique tris par tas puis de leur implémentation à l'aide d'un tas stocké dans un tableau.

D Dans une dernière partie, nous utiliserons la notion de backtracking, parcours en profondeur de l'arbre des solutions, au travers du problème des huit reines.

Sommaire


A Définition . . . . . 2
1 Terminologie . . . . . 2
2 Arbre binaires . . . . . 3
3 Parcours en profondeur . . . . . 3
4 Définition et propriétés inductives 4
B Arbres Binaires de Recherche . . . . . 6
1 Définition . . . . . 6
2 Déséquilibre . . . . . 9
3 Dictionnaire . . . . . 9
C File de priorité . . . . . 10
1 Implémentations naïves . . . . . 10
2 Définition . . . . . 11
3 Tri par tas . . . . . 13
4 Stockage dans un tableau . . . . 13
D Backtracking . . . . . 14
1 Le problème des huit reines . . . 14

# A Définition

## 1 Terminologie

**Définitions : arbre** ●●●

Un arbre est une structure de données contenant des nœuds, dont l'un d'eux est une racine, c'est-à-dire le nœud sans parents, les autres en possédant un unique.

Autrement dit, un arbre est un ensemble de nœuds dont :

- **une racine** qui n'a pas de parents
- **d'autres nœuds** qui ont un et un unique parent

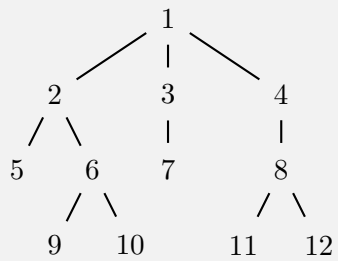
**Exemple.** Représentation `Ocaml` d'un arbre : `type 'a arbre = N of 'a * 'a arbre list`

**Définitions : terminologie des arbres** ●●●

Il convient maintenant d'introduire la terminologie usuelle sur les arbres :

- **père et fils :** Si une arête mène du sommet $i$ vers le sommet $j$, alors $i$ est le père de $j$ et $j$ est le fils de $i$
- **feuille :** Une feuille est un nœud qui n'a pas de fils
- **sous-arbre :** Chaque nœud est racine d'un sous-arbre constitué de lui-même et de tous ses descendants
- **arité :** L'arité d'un nœud est son nombre de fils

**Exemple d'arbre et application de la terminologie**




Quelques exemples d'application de la terminologie :

- le nœud d'étiquette 6 est le père de celui d'étiquette 10
- le nœud d'étiquette 6 est le père de celui d'étiquette 10
- le nœud d'étiquette 5 est le fils de celui d'étiquette 2
- les feuilles sont les nœuds d'étiquette 5, 9, 10, 7, 11, 12
- les nœuds d'étiquette 4, 8, 11, 12 forment un le sous-arbre dont la racine est le nœud d'étiquette 4
- l'arité du nœud d'étiquette 1 est 3

## 2 Arbre binaires

**Définition : Arbres binaires**

Un arbre binaire est un arbre dans lequel chaque nœud a au plus deux fils.
Bien souvent, on rajoute à l'ensemble des arbres binaires l'arbre `Vide` qui permet des définitions inductives plus simples.

Autrement dit, pour un ensemble $E$, l'ensemble des arbres binaires étiquetés par $E$ est défini par :

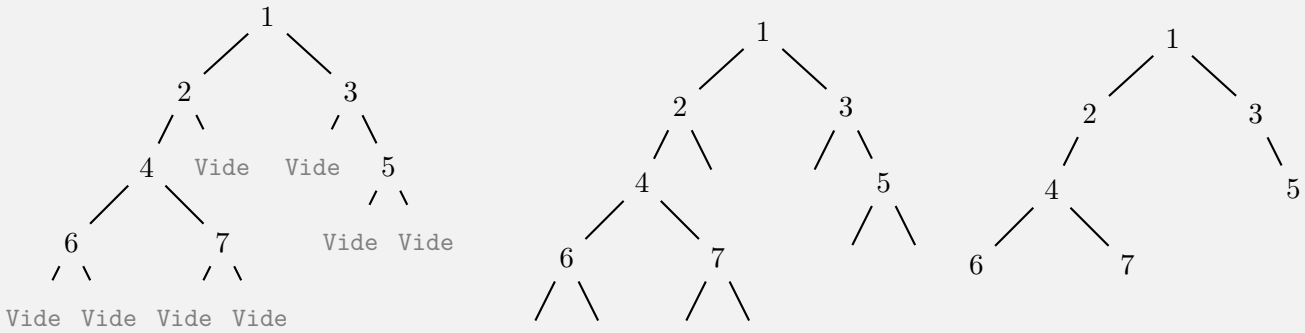
- `Vide` est un arbre binaire sur $E$
- si $x \in E$, et $F_g$, $F_d$ deux arbres binaires étiquetés par $E$ alors $(x, F_g, F_d)$ est un arbre binaire étiqueté par $E$.[a]

[a] ici, $x$ est l'étiquette du nœud, $F_d$ et $F_g$ respectivement ses sous-arbres droit et gauche

**Exemple.** Représentation `Ocaml` d'un arbre binaire : `type 'a arbre = Vide | N of 'a*'a arbre*'a arbre`

**Exemple de représentations d'arbres binaires**

```
let a = N(1,N(2,N(4,N(6,Vide,Vide),N(7,Vide,Vide)),Vide),N(3,Vide,N(5,Vide,Vide)));;
```

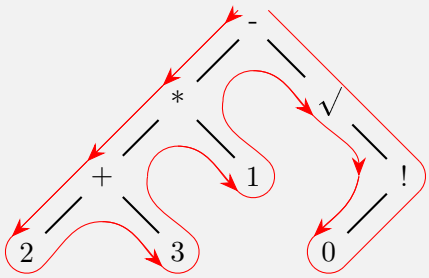



Voici trois représentations du même arbre binaire.

## 3 Parcours en profondeur d'un arbre binaire

Lors d'un parcours en profondeur, d'un nœud contenant des sous arbres, chaque sous arbre est exploré en entier avant d'explorer le sous arbre suivant.

**Parcours préfixe d'un arbre binaire**

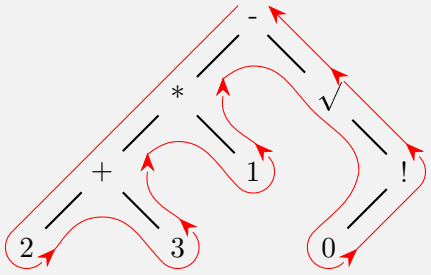



Le parcours préfixe d'un arbre consiste à s'occuper de la racine avant d'explorer le sous arbre gauche puis droit. D'une manière générale, on peut l'obtenir selon l'ordre du *passage à gauche* dans le parcours suivant. Cela revient à `- * + 2 3 1 √ ! 0` dans l'exemple.

```
let rec parcours_prefixe a = match a with
    | Vide -> ()
    | N(v, fg, fd) -> print_int v;
                      parcours_prefixe fg;
                      parcours_prefixe fd;;
```

**Remarque :** Le parcours préfixe d'un arbre représentant une expression arithmétique permet de retrouver la notation polonaise.

**Parcours suffixe d'un arbre binaire**

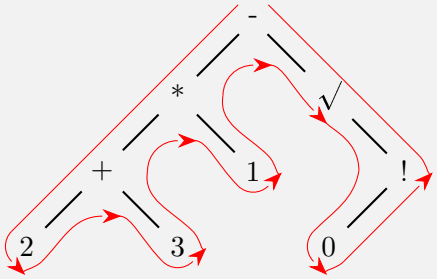



Le parcours suffixe d'un arbre consiste à explorer les sous arbre gauche et droit avant de s'occuper de la racine. D'une manière générale, on peut l'obtenir selon l'ordre du *passage à droite* dans le parcours suivant.
Cela revient à `2 3 + 1 * 0 ! √ -` dans l'exemple.

```
let rec parcours_suffixe a = match a with
    | Vide -> ()
    | N(v, fg, fd) -> parcours_prefixe fg;
                      parcours_prefixe fd;
                      print_int v;;
```

**Remarque :** Le parcours suffixe d'un arbre représentant une expression arithmétique permet de retrouver la notation polonaise inverse.

**Parcours infixe d'un arbre binaire**




Le parcours infixe d'un arbre consiste à explorer le sous arbre gauche puis à s'occuper de la racine avant d'explorer le sous arbre droit. D'une manière générale, on peut l'obtenir selon l'ordre du *passage sous* un nœud dans le parcours suivant.
Cela revient à `(2 + 3) * 1 - √ (0 !)` dans l'exemple.

```
let rec parcours_infixe a = match a with
    | Vide -> ()
    | N(v, fg, fd) -> parcours_prefixe fg;
                      print_int v;
                      parcours_prefixe fd;;
```

**Remarque :** Si l'arbre représente l'opération arithmétique $(2+3)*1-\sqrt{0!}$, seul le parcours infixe qui est ambiguë nécessite des parenthèses, sinon l'on peut confondre avec $2+3*1-\sqrt{0!}$ .

## 4 Définition et propriétés inductives

**Définition par induction structurelle** *4*

Soit $E$, $F$ des ensembles et $\mathscr{A}$ l'ensemble des arbres binaires étiquetés par $E$,
On peut définir des fonctions $f : \mathscr{A} \to F$ via un élément $a \in F$ et $\phi : E \times F \times F \to F$ ainsi que les propriétés suivantes :

- $f(\texttt{Vide}) = a$
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}, f(x, F_g, F_d) = \phi(x, f(F_g), f(F_d))$

**Définition : taille d'un arbre** ●●●

La taille d'un arbre est son nombre de nœuds.

```
let rec taille a = match a with
  | Vide -> 0
  | N(_, fg, fd) -> 1 + (taille fg) + (taille fd)
```

On note généralement $|A|$, la taille d'un arbre $A$.

**Définition : hauteur d'un arbre** ●●●

La hauteur d'un arbre est le maximal de la profondeur de ses nœuds.

```
let rec hauteur a = match a with
  | Vide -> -1
  | N(_, fg, fd) -> max (hauteur fg) (hauteur fd) + 1
```

On note généralement $h(A)$, la hauteur d'un arbre $A$.

**Définition : nombre de feuilles** ●●●

Le nombre de feuilles d'un arbre est le nombre de ses nœuds d'arité nulle.

```
let rec nbfeuilles a = match a with
  | Vide -> 0
  | N(_, Vide, Vide) -> 1
  | N(_, fg, fd) -> (nbfeuilles fg) + (nbfeuilles fd)
```

**Théorème : Preuve par induction structurelle** *4*

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

- $\mathscr{R}(\texttt{Vide})$ est vraie
- $\forall x \in E, \forall (F_g, F_d) \in \mathscr{A}, ((\mathscr{R}(F_g) \text{ et } \mathscr{R}(F_d)) \Rightarrow \mathscr{R}(x, F_g, F_d))$ est vraie.

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

**Démonstration.** par récurrence forte en la hauteur de l'arbre

**Encadrement du nombre de nœuds d'un arbre binaire** *4*

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

**Démonstration.** Par induction structurelle sur $A$ :

- Si $A = \texttt{Vide}$, $|A| = 0$ et $h(A) = -1$, $-1 + 1 \leqslant 0 \leqslant 2^{-1+1} - 1$, le résultat est vérifié.
- Si $A = (x, F_g, F_d)$, on suppose le résultat pour $F_g$ et $F_d$.

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

$$\begin{aligned} |A| = 1 + |F_g| + |F_d| &\geqslant h(F_g) + h(F_d) + 3 \\ &\geqslant max(h(F_g), h(F_d)) + 3 \\ &\geqslant h(A) + 1 \end{aligned}$$

**Remarque.** Les démonstrations par induction structurelles reviennent à éviter de faire des récurrences fortes en la hauteur de l'arbre, la preuve en devient plus légère.

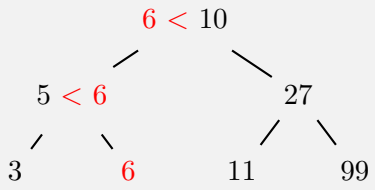
# B Arbres Binaires de Recherche

**Analogie entre dichotomie et arbre binaire de recherche**

Soit le tableau trié `[3; 5; 6; 10; 11; 27; 99]`. Si je souhaite rechercher rapidement un élément de ce tableau, je vais procéder par recherche dichotomique.

Prenons `6` par exemple, je vais d'abord comparer `6` à `10`. puis `6` à `5` avant de trouver `6`.

En réalité, on peut représenter les pivots de cette recherche dichotomique dans ce qu'on appelle un arbre binaire de recherche.
Pour faire une recherche dans cet arbre, il suffit de comparer succesivement notre élément avec les différentes étiquettes (ou pivôts) choisissant le prochain sous-arbre en fonction de cette comparaison.

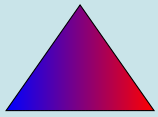



Les arbres binaires de recherches sont des arbres d'une recherche dichotomique, en effet les pivots y sont fixés.

## 1 Définition

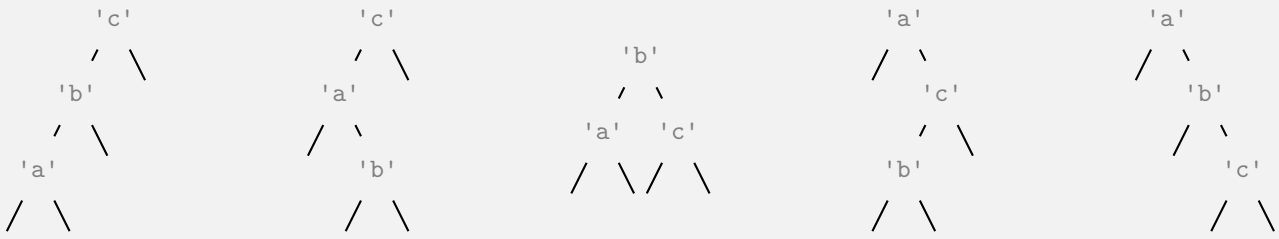
**Définition : arbre binaire de recherche** ●●●

Un *arbre binaire de recherche* est un arbre binaire où chaque noeud, est plus grand que tout les noeuds de son fils gauche et plus petit que tout les noeuds de son fils droit.



**Remarque.** Un arbre binaire de recherche peut être étiqueté par n'importe quel ensemble ordonné.

**Exemple de représentations d'ABR à trois éléments**




Il existe plusieurs structures d'ABR pour le même ensemble d'étiquettes.

### 1.1 Parcours infixe d'un ABR

**Théorème :** *4*

Lors du parcours infixe d'un arbre binaire de recherche A, les clés sont parcourues par ordre croissant.

**Démonstration.** Par induction structurelle sur $A$ :

- Si $A =$ `Vide`, il n'y a rien à prouver
- Si $A = (x, F_g, F_d)$, on suppose les clés sont parcourues par ordre croissant dans les parcours infixes des sous-arbes $F_g$ et $F_d$.
  Or toutes clé de $F_g$ est inférieure à x elle même inférieure à toute clé de $F_d$ donc le parcours $F_g$ puis $x$ puis $F_d$ est bien effectué par ordre croissant de clé.

**Corrolaire :** *4*

Soit A1 et A2 deux ABR de structure fixée et de mêmes étiquettes, alors A1 est égal à A2.

**Démonstration.** Les deux arbres ayant la même structure, le parcours infixe renvoie les étiquettes des même noeuds. Or les clés étant parcourues par odre croissant pour ce même parcours, les noeuds ayant la même place dans la structure des deux arbres possèdent les même étiquettes. Les arbres sont alors égaux.

## 1.2 Accès à un ABR

### Recherche d'une clé

**Recherche d'un clé** ●●●

La recherche d'une clé se fait donc par dichotomie, en utilisant les pivôts imposés par l'arbre binaire de recherche.

```
let rec recherche abr x = match abr with
  | Vide -> None
  | N(n, _, _)   when x = n  -> Some n
  | N(n, fg, _)  when x < n  -> recherche fg x
  | N(n, _, fd)(*when n < x*)-> recherche fd x;;
```

### Recherche de la clé minimale/maximale

**Recherche d'une clé minimale / maximale** ●●●

La recherche du minimum se poursuit dans le fils gauche tant qu'il existe, lorsqu'il est vide, les étiquettes du fils droit étant supérieures, on a trouvé le minimum.

```
let rec minimum abr = match abr with
  | Vide -> None
  | N(n, Vide, _) -> Some n
  | N(_, fg, _)   -> minimum fg;;
```

### Recherche de la clé prédécesseur/sucesseur

**Recherche de la clé prédécesseur/sucesseur**

L'arbre binaire de recherche se prête bien à la recherche d'un successeur, l'idée est d'explorer le graphe en restant aussi proche que possible de la clé recherché. Il faut tout de même garder à l'esprit que le successeur peut être dans un noeud parent donc stocké dans un accumulateur `acc` ici.

```
let sucesseur x arb =
  let rec aux x arb acc = match arb with
    | Vide -> acc
    | N(n, _, fd)   when n <= x  -> aux x fd acc
    | N(n, fg, _) (*when n > x*) -> aux x fg (Some n) in
  aux x arb None;;
```

Toutes les opérations de recherche définies précédemment peuvent s'effectuer en $O(h(A))$, en effet, une comparaison par profondeur dans l'arbre est suffisant pour trouver l'élément recherché.

## 1.3 Implémentation Naïve

### Insertion naïve

**Insertion dans une feuille**

L'idée est de parcourir l'arbre en respectant les inégalités d'un abr pour notre étiquette, de cette manière, en arrivant à une feuille, on peut faire une insertion sans problèmes.

```
let rec insertion_naïve abr x = match abr with
  | Vide -> N(x, Vide, Vide)
  | N(n, _, _)      when x = n    -> abr
  | N(n, fg, _)     when x < n    -> insertion fg n
  | N(n, _, fd) (* when n < x *) -> insertion fd n;;
```

### Suppression naïve

**Suppression d'une clé**

Si l'un des sous-arbre de l'étiquette recherchée est vide, alors on peut simplement la remplacer par la racine de l'autre. Sinon, on va croquer dans le maximum du sous arbre gauche ou le minimum du sous-arbre droit pour remplacer la racine.

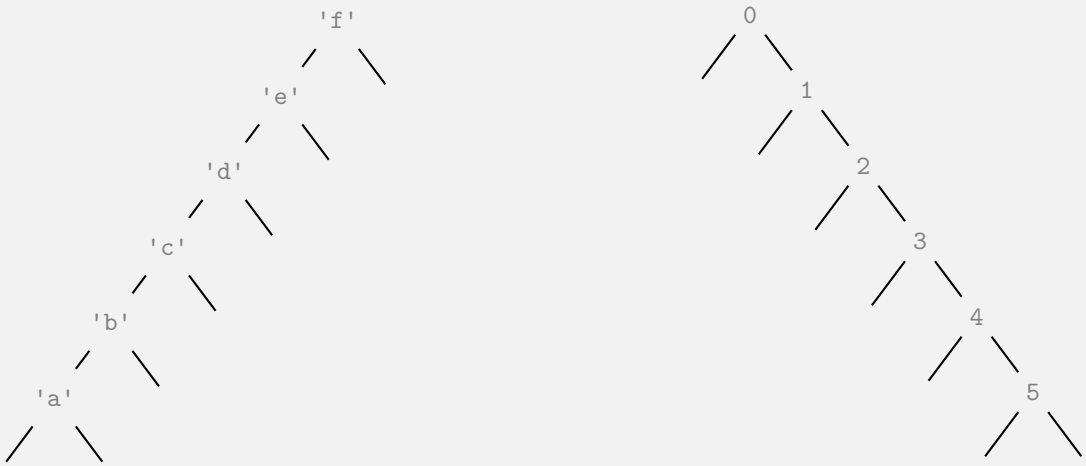
```
let supprime_max abr = match abr with
    | Vide -> failwith "pas de max"
    | N(n, fg, Vide) -> (n, fg)
    | N(n, fg, fd)   -> let (max, fg') = supprime_max fd in
                        (max, N(n, fg', fd));;

let rec suppression_naïve abr x = match abr with
  | Vide -> Vide
  | N(n, fg, fd)       when x < n    -> N(n, suppression_naïve fg x, fd)
  | N(n, fg, fd)       when n < x    -> N(n, fg, suppression_naïve fd x)
  | N(n, fg, Vide)  (* when x = n *) -> fg
  | N(n, Vide, fd)  (* when x = n *) -> fd
  | N(n, fg, fd)    (* when x = n *) -> let (max, fg') = supprime_max fg in
                                        N(max, fg', fd);;
```

Ici, la suppression se fait encore en $O(h(A))$. En effet, la fonction `supprime_max` à une complexité en $O(h(A))$, et à chaque execution de la fonction `suppression_naïve`, on fait appel soit récurssivement à `suppression_naïve`, sur un arbre de hauteur plus petit de 1, soit on fait appel à `supprime_max`, sur un arbre de hauteur plus petit de 1.

## 2 Déséquilibre

On définit usuellement la taille de l'entrée en la taille de l'arbre soit $n = |A|$.
Toutes les fonctions usuelles sur les ABR ont une complexité temporelle en $O(h(A))$. Soit $O(n)$ dans un ABR quelconque.
Et nous avons aussi vu que plusieurs structures d'ABR existent pour le même ensemble d'étiquettes. Or la seule hypothèse dont nous disposons est que $h(A) < |A|$, le cas pathologique de l'arbre peigne permet d'llustrer ce pire cas :

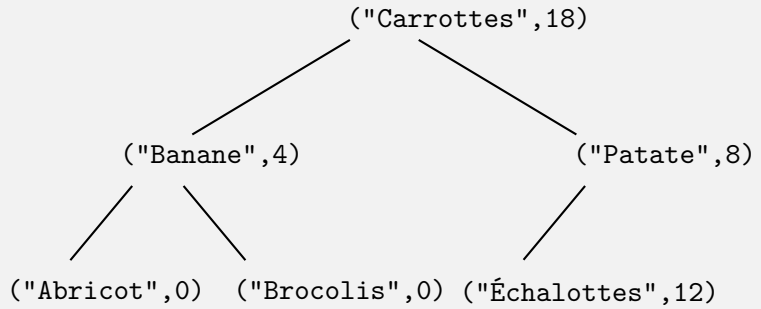



Cependant il existe des arbres binaires équilibrés qui garantissent $O(h(A)) = O(log(n))$, on peut citer par exemple les *arbres presque complets*, les *arbres AVL* ou encore les *arbres rouge-noir*.

## 3 Dictionnaire

Si l'on définit l'ensemble $E = C \times V$ avec $C$ un ensemble ordonné, on peut définir un ABR d'étiquette $E$ ordonnée par l'ensemble $C$. Cet ABR représente alors un dictionnaire dont l'ensemble $C$ représente les clefs et $V$ les valeurs.

On peut par exemple prendre des mots comme clés, en utilisant l'ordre lexicographique comme relation d'ordre, et des entiers comme valeur pour représenter un inventaire :



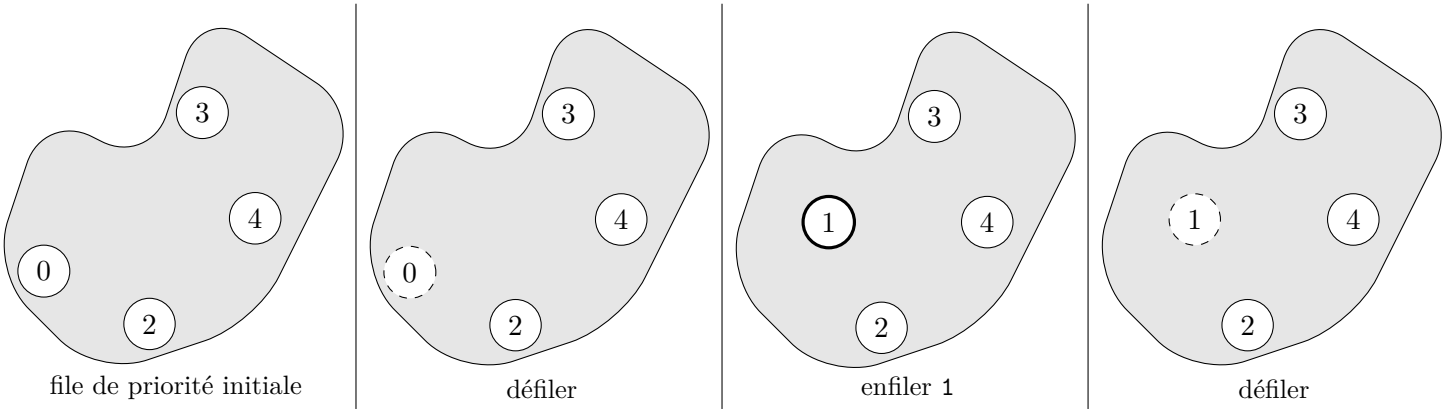

On garde alors les propriétées d'un ABR, en particulier la recherche en $O(h(A))$.

# C File de priorité

**Définition : File de priorité**

Une file de priorité est un type abstrait sur un ensemble ordonné qui comporte trois opérations :

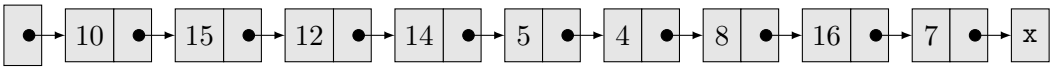
- **enfiler** un élément
- **défiler** l'élément ayant la plus petite clef
- tester si la file de priorité **est vide** ou non



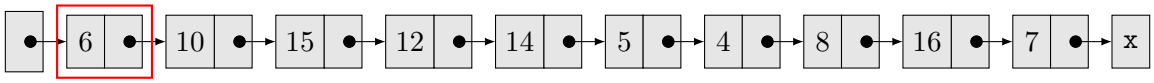

## 1 Implémentations naïves

### 1.1 La liste

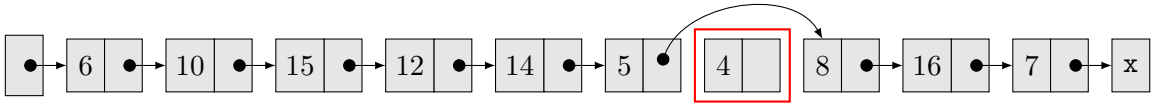
On peut implémenter une file de priorité à l'aide d'une liste quelconque.
Voici par exemple une file de priorité contenant les éléments $\{4, 5, 7, 8, 10, 12, 14, 15, 16\}$.




**Enfiler** On peut simplement enfiler un élément en l'ajoutant en tête de liste, ajoutons 6 par exemple.




**Défiler** Si l'on veut défiler, il va nous falloir retrouver le minimum de la liste, ici 4. Cela se fait en comparant un à un les éléments de cette liste.

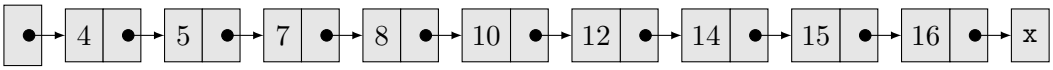


**Files de priorités implémentées par des listes :**

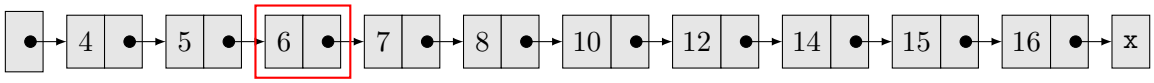
| opération | complexité | méthode |
|---|---|---|
| **enfiler** | $\mathcal{O}(1)$ | ajout en tête de liste |
| **défiler** | $\mathcal{O}(n)$ | minimim de la liste |
| **est_vide** | $\mathcal{O}(1)$ | test si la liste est vide |

### 1.2 La liste triée

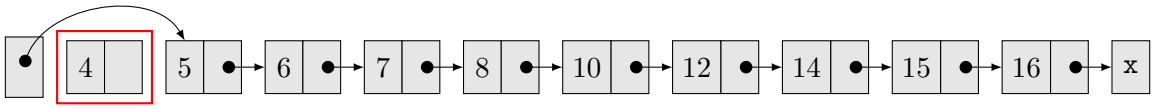
On peut aussi implémenter une file de priorité à l'aide d'une liste triée.
Reprenons l'ensemble précédent $\{4, 5, 7, 8, 10, 12, 14, 15, 16\}$.




**Enfiler** Si l'on veut enfiler, il va nous falloir placer l'élément au bon endroit, plaçons 6. Cela se fait en comparant un à un les éléments de cette liste.




**Défiler** On peut simplement défiler un élément en supprimant la tête de liste, ici 4.



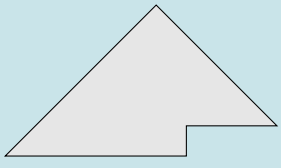

**Files de priorités implémentés par des listes triées :**

| opération | complexité | méthode |
|---|---|---|
| **enfiler** | $\mathcal{O}(n)$ | insertion par comparaisons successives conservant le tri |
| **défiler** | $\mathcal{O}(1)$ | suppression de la tête de liste |
| **est_vide** | $\mathcal{O}(1)$ | test si la liste est vide |

## 2 Implémentation par tas

**Définition : Arbre binaire presque complet** ●●●

Un arbre binaire est *presque complet* si tous ses niveaux sont complets sauf éventuellement le dernier.



**Proposition :** *4*

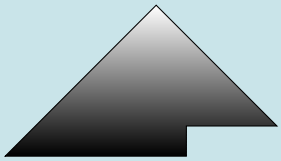
Soit $A$ un arbre binaire presque complet à $n$ nœuds, $h(A) \leqslant log_2(n)$.

**Démonstration.** Soit $A$ un arbre binaire presque complet, il y a au moins un nœud sur le niveau $h(A)$ et tous les autres sont complets. On peut donc minorer le nombre de nœud en fonction de la hauteur de l'arbre tel que :

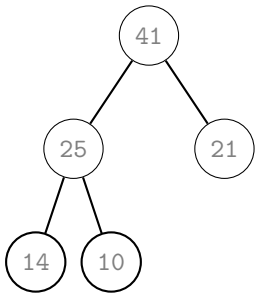
$$1 + \sum_{i=0}^{h(A)-1} (2^i) \leqslant n$$
$$2^{h(A)} \leqslant$$
$$h(A) \leqslant log_2(n)$$

**Définition : Tas** ●●●

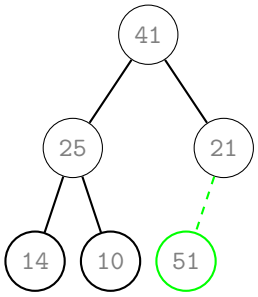
Un tas est un arbre binaire presque complet où chaque nœud a une valeur plus prioritaire que celle de ses fils. Lorsque les relations d'ordre sur les nombres $\leqslant$ $\geqslant$ sont utilisées, on parle alors de Tas-min ou de Tas-max.



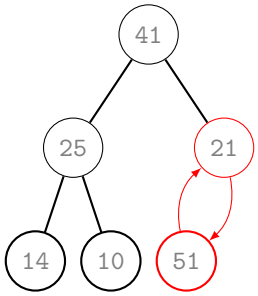
**Enfiler** Pour enfiler un élément dans un tas, on va commencer par compléter l'arbre par l'élément avant de le faire remonter grâce à du *bubble up* pour retrouver notre structure d'arbre tournois.



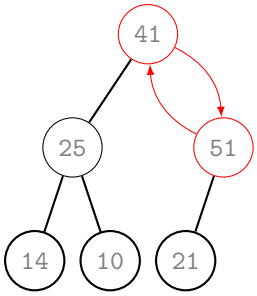
Arbre Initial



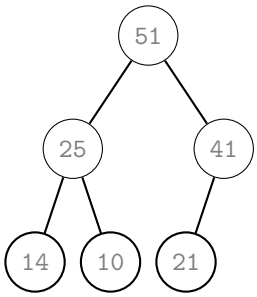
Ajout de 51



Bubble Up car $21 \leqslant 51$

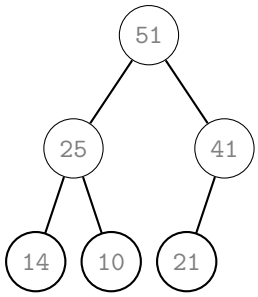


Bubble Up car $41 \leqslant 51$

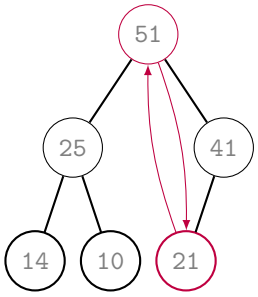


Arbre Final

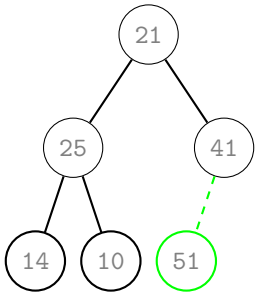
**Défiler** Pour défiler, on va échanger la racine avec le dernier noeud complété. On peut donc ensuite la retirer. Afin de rétablir la structure on va *Bubble Down* la nouvelle racine.



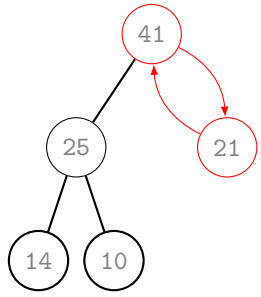
Arbre Final



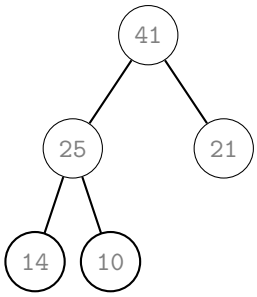
Echange Racine



Suppression de 51



Bubble Down car $21 \leqslant 25 \leqslant 41$



Arbre Initial

**Files de priorités implémentés par des tas :**

| opération | complexité | méthode |
|---|---|---|
| **enfiler** | $\mathcal{O}(log_2 n)$ | *Bubble Up* |
| **défiler** | $\mathcal{O}(log_2 n)$ | *Bubble Down* |
| **est_vide** | $\mathcal{O}(1)$ | tester si l'arbre est `Vide` |

## 3 Tri par tas

**Algorithme : tri par tas**

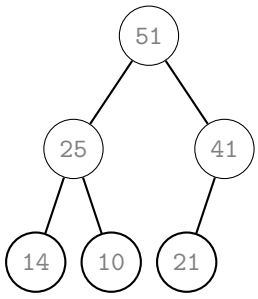
L'algorithme est très simple, il consiste à enfiler tous les éléments d'une liste `L` puis de défiler le tas. On obient alors une liste `L'` triée.

**Théorème : Complexité du tri par tas** *4*

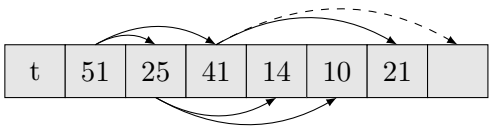
Pour une liste `L` de taille $n$, on peut la trier en utilisant des tas en complexité temporelle $\mathcal{O}(nlog_2(n))$ et en complexité en espace $\mathcal{O}(1)$[a].

[a] En stockant le tas dans un tableau, on peut faire le tris "sur-place"

## 4 Stockage dans un tableau



Arbre Final




**Numérotation de Sosa-Stradonitz :** Lors du stockage d'un arbre dans un tableau, les fils du noeud $i$ sont respectivement $2*i$ et $2*i+1$ en indexant à partir d'un dans le tableau. Ici, la hauteur maximale d'un arbre pouvant être représenté dans ce tableau est 2.

En `Ocaml`, les tableaux sont indexés à partir de `0`, on a donc deux solutions pour parer à ça :

**Si on travaille sur un tas d'entiers** On peut alors réserver bien la première case pour stocker la taille de l'arbre, nécessaire pour enfiler et défiler.

**Si on a besoin du polymorphismes** Si l'on souhaite conserver le polymorphismes, on peut recalculer la valeur des fils en partant d'un index 0, soit $2*i+1$ et $2*i+2$ et définir une structure pour stocker la taille de l'arbre, toujours nécessaire.

```
type 'a tas = {mutable longueur : int ; contenu : 'a array};;
```