Chapitre 1

option informatique

Arbres binaires

1. Introduction

Dans son acceptation la plus générale, un arbre est un graphe connexe acyclique enraciné [1] : tous les sommets, à l'exception de la racine, ont un unique parent.

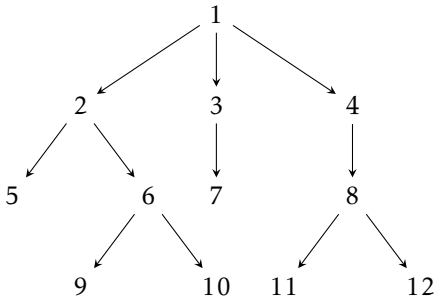


Figure 1 – Un exemple d'arbre enraciné en 1.

Si une arête mène du sommet i au sommet j, on dit que i est le *père* de j, et en conséquence que j est le *fils* de i. Par exemple, 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, si bien que l'on peut sans ambigüité représenter les arêtes par des traits simples à la place des flèches, ce que l'on fera par la suite [2].

Dans ce contexte, il est fréquent de parler de *nœud* au lieu de sommet. Un nœud qui n'a pas de fils est appelé une *feuille* (ou nœud externe), les autres sont appelés des *nœuds internes*. Dans l'exemple ci-dessus, les feuilles sont les sommets 5, 7, 9, 10, 11, et 12 et les nœuds internes les sommets 1, 2, 3, 4, 6, et 8.

Enfin, on notera que 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. Par exemple, les sommets 2, 5, 6, 9 et 10 constituent un sous-arbre de l'arbre représenté ci-dessus.

1.1 Définition formelle d'un arbre binaire

On appelle *arité* d'un nœud le nombre de branches qui en partent [3]. Dans la suite de notre cours, nous nous intéresserons plus particulièrement aux *arbres binaires*, c'est à dire ceux dont chaque nœud a au plus deux fils. Pour ne pas avoir à distinguer les nœuds suivant leur arité, il est pratique d'ajouter à l'ensemble des arbres binaires un arbre particulier appelé l'*arbre vide*. Ceci conduit à adopter la définition qui suit.

Un ensemble E étant donné, on définit par induction les arbres binaires étiquetés par E en convenant que :

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

x est l'étiquette de la *racine* de A ; quant à F_g et F_d, ils sont appelés respectivement le *sous-arbre gauche* et le *sous-arbre droit* de l'arbre binaire A.
De manière usuelle, on convient de ne pas faire figurer l'arbre vide dans les représentations graphiques des arbres binaires (voir la figure 2). Ainsi, suivant la représentation choisie les feuilles pourront désigner exclusivement l'arbre vide (et dans ce cas tous les nœuds seront d'arité égale à 2) ou alors les nœuds dont les deux fils sont vides (dans ce cas l'arité d'un nœud pourra être égale à 0, 1 ou 2). C'est cette seconde convention qui sera utilisée dans la suite de ce cours.

1. La définition précise de ces termes sera donnée dans le chapitre suivant.
2. Plus précisément, nous démontrerons qu'une fois la racine choisie, il existe une unique orientation des arêtes qui permette de se déplacer de la racine vers chacun des autres sommets.
3. son *degré sortant*, pour employer le vocabulaire des graphes.

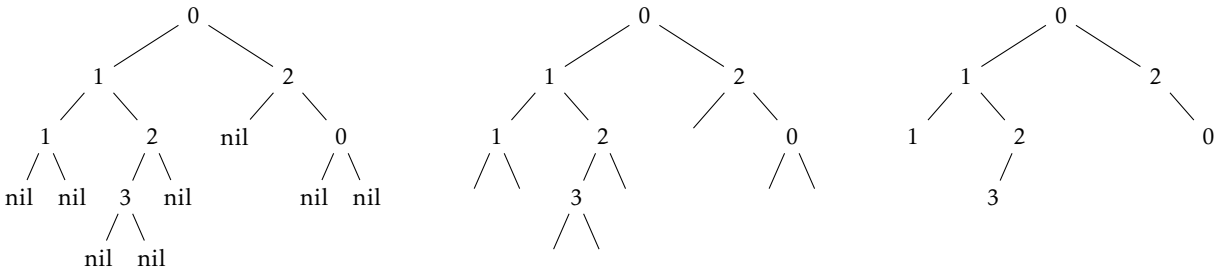


FIGURE 2 – Trois représentations du même arbre binaire.

Mise en œuvre pratique
La définition formelle que nous avons adoptée peut se résumer à :

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

et conduit à la définition CAML suivante :

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

1.2 Fonctions inductives

Preuve par induction structurelle
La plupart des résultats qui concernent les arbres binaires se prouvent par *induction structurelle*, c'est-à-dire en utilisant le résultat suivant :

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

(i) $\mathscr{R}(nil)$ *est vraie ;*

(ii) $\forall x \in \mathrm{E}, \forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}^2$, *l'implication* $\big(\mathscr{R}(\mathrm{F}_g)$ *et* $\mathscr{R}(\mathrm{F}_d)\big) \Longrightarrow \mathscr{R}(\mathrm{F}_g, x, \mathrm{F}_d)$ *est vraie ;*

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

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

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

DÉFINITION. — *La* taille $|\mathrm{A}|$ *d'un arbre* A *est définie inductivement par les relations :*

- $|nil| = 0$;
- *Si* $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$ *alors* $|\mathrm{A}| = 1 + |\mathrm{F}_g| + |\mathrm{F}_d|$.

La hauteur $h(\mathrm{A})$ *d'un arbre* A *se définit inductivement par les relations :*

- $h(nil) = -1$;
- *Si* $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$ *alors* $h(\mathrm{A}) = 1 + \max(h(\mathrm{F}_g), h(\mathrm{F}_d))$.

Avec ces conventions, $|\mathrm{A}|$ est le nombre de nœuds d'un arbre et $h(\mathrm{A})$ la longueur maximale du chemin reliant la racine à une feuille, autrement dit la profondeur maximale d'un nœud.
La définition CAML de ces fonctions est immédiate :

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

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

Autre exemple, pour calculer le nombre de feuilles d'un arbre binaire on utilise la fonction :

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

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

Preuve. On raisonne par induction structurelle.

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

$$|A| = 1 + |F_g| + |F_d| \geqslant 1 + h(F_g) + 1 + h(F_d) + 1 \geqslant 2 + \max(h(F_g), h(F_d)) = 1 + h(A)$$

$$\text{et} \quad |A| = 1 + |F_g| + |F_d| \leqslant 2^{h(F_g)+1} + 2^{h(F_d)+1} - 1 \leqslant 2 \times 2^{\max(h(F_g),h(F_d))+1} - 1 = 2^{h(A)+1} - 1$$

□

1.3 Arbres binaires équilibrés

Revenons sur le résultat énoncé dans le dernier théorème, qu'on peut aussi écrire : si A est un arbre binaire alors $\log(|A|+1) - 1 \leqslant h(A) \leqslant |A| - 1$. Dans la suite de ce cours nous aurons à plusieurs reprises intérêt à utiliser des arbres qui, pour une taille donnée, ont une hauteur minimale, autrement dit pour lesquels $h(A) = O(\log(|A|))$. De tels arbres seront dits *équilibrés*.

Étudions tout d'abord le cas optimal $h(A) = \log(|A|+1) - 1$, correspondant aux arbres binaires *complets*. D'après la preuve du théorème précédent, 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. Il est dès lors facile d'établir par induction qu'un arbre binaire est complet si et seulement si toutes ses feuilles sont à la même profondeur.

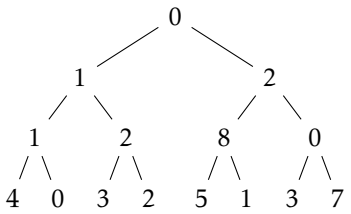


Figure 3 – Un exemple d'arbre binaire complet : $|A| = 15$ et $h(A) = 3$.

Cependant, la plupart des arbres binaires que l'on rencontre dans la pratique ne sont pas complets, cette notion étant trop restrictive ; c'est pourquoi on privilégie certaines catégories d'arbres qui garantissent l'équilibrage : arbres rouge-noir [4], arbres AVL, etc.

Par exemple, la catégorie des arbres AVL introduit la notion de *déséquilibre* : le déséquilibre d'un arbre $A = (F_g, x, F_d)$ est égal à l'entier $h(F_g) - h(F_d)$. On adopte alors la définition :

Définition. — *Un arbre binaire* A *est un arbre* AVL [5] *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.

Autrement dit, un arbre AVL est un arbre dans lequel le déséquilibre de chaque sous-arbre est égal à -1, 0 ou 1.

Théorème. — *Tout arbre AVL est équilibré.*

4. Voir l'exercice 5 à ce sujet.
5. D'après le nom de leurs inventeurs Adelson-Velsky et Landis.

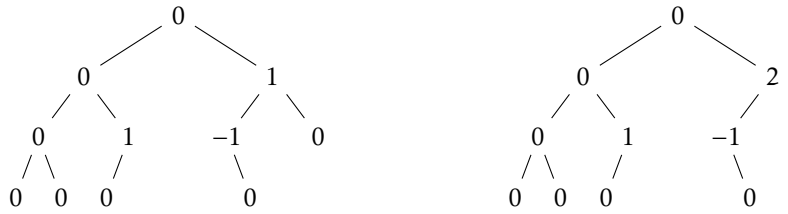


Figure 4 – L'arbre de gauche est un AVL, pas celui de droite [6].

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

- Si A = 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. On en déduit que A contient au moins $f_{h-1} + f_{h-2} + 1 = f_h + 1$ nœuds.

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

2. Arbres binaires de recherche

Un arbre binaire de recherche (en abrégé : ABR) permet l'implémentation sous forme d'arbre binaire de certaines structures de données stockant des éléments formés d'une *clé* et d'une *valeur*, tels les dictionnaires [7]. Nous allons donc considérer un ensemble ordonné de clés C ainsi qu'un ensemble de valeurs V, et utiliser des arbres binaires étiquetés par $E = C \times V$.
Les arbre binaire de recherche supportent nombre d'opérations qu'on utilise dans les structures de données, en particulier :

- la *recherche* d'un valeur associée à une clé donnée ;
- la recherche de la valeur associée à la clé *maximale* (ou *minimale*) ;
- la recherche du *successeur* d'une clé c, c'est à dire la valeur associée à la plus petite des clés strictement supérieures à c ;
- la recherche du *prédécesseur* d'une clé ;
- et bien sûr l'*insertion* ou la *suppression* d'un nouveau couple clé/valeur.

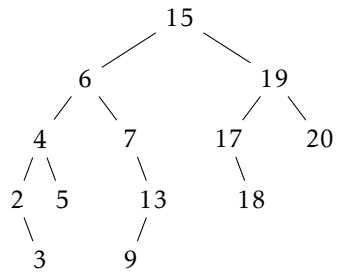


Figure 5 – Un exemple d'arbre binaire de recherche (seules les clés ont été représentées)

Définition. — *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 [8]) à* c *;*
- *toute clé de* F_d *est supérieure (ou égale) à* c.

6. Les arbres sont étiquetés par leur déséquilibre.
7. Voir le cours de première année.
8. Il serait préférable d'imposer aux clés présentes d'être deux à deux distinctes, mais cela n'est pas toujours vérifié dans la pratique.

Autrement dit, A est un arbre binaire de recherche lorsque tout nœud de A 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.

Dans la suite du cours, on utilisera le type :

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

et les arbres binaires de recherche seront représentés par le type *('a, 'b) data arbre*.

2.1 Parcours infixe d'un arbre binaire de recherche

La propriété des arbres binaires de recherche permet d'afficher toutes les valeurs de l'arbre par ordre croissant de clé à l'aide d'un *parcours infixe*. Rappelons rapidement le principe de l'exploration en profondeur d'un arbre (F_g, x, F_d) : chaque sous-arbre est exploré dans son entier avant d'explorer le second sous-arbre. Le parcours infixe consiste à parcourir l'arbre dans l'ordre : $F_g \longrightarrow x \longrightarrow F_d$.
Si **`traitement`** est une fonction de type *'b -> unit*, le parcours infixe d'un arbre binaire de recherche prendra la forme suivante :

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

À l'évidence, le coût temporel de ce parcours est un $\Theta(n)$ lorsque $n = |A|$.

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

Preuve. On procède par induction :

- si A = 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. Sachant que toute clé de F_g est inférieure à la clé de x et toute clé de F_d supérieure à cette dernière, le parcours $F_g \longrightarrow x \longrightarrow F_d$ est bien effectué par ordre croissant de clé.

□

2.2 Requêtes dans un arbre binaire de recherche

Recherche d'une clé
Une des opérations les plus courantes dans un arbre binaire de recherche $A = (F_g, (c, v), F_d)$ est la recherche d'une valeur associée à une clé particulière k. La démarche est évidente :

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

Dans le cas où les clés ne sont pas toutes distinctes on retournera la valeur associée à la première clé égale à k rencontrée.

```
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 ;;
```

Par exemple, la recherche de la valeur associée à la clé $k = 13$ dans l'arbre de la figure 5 conduit à suivre le chemin : $15 \to 6 \to 7 \to 13$ à partir de la racine.

Recherche de la clé minimale / maximale
La recherche de la valeur associée à 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 ;;
```

La fonction retournant la valeur associée à la clé maximale est symétrique :

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

Dans l'exemple de la figure 5 on obtient la clé minimale 2 en suivant les fils gauches à partir de la racine, et la clé maximale 20 en suivant les fils droits.

Recherche du prédécesseur / successeur
$k \in \mathrm{C}$ étant donné, il s'agit cette fois de retourner la valeur associée à la plus grande des clés c contenues dans l'arbre et vérifiant : $c < k$ (respectivement $c > k$).

```
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 ;;
```

```
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 ;;
```

Par exemple, le successeur de 13 dans l'ABR représenté figure 5 est la racine 15 puisqu'il n'y a pas de successeur possible dans le fils gauche.

• Complexité temporelle

Toutes ces fonctions ont à l'évidence un coût temporel en $\mathrm{O}(h(\mathrm{A}))$, ce qui explique tout l'intérêt qu'il peut y avoir à ce que l'arbre binaire de recherche soit équilibré : dans un arbre binaire quelconque d'ordre $n = |\mathrm{A}|$, on peut affirmer que le coût d'une requête est un $\mathrm{O}(n)$; dans le cas d'un arbre de recherche équilibré, on peut assurer un coût en $\mathrm{O}(\log n)$.

2.3 Insertion et suppression

• Insertion et suppression dans un arbre binaire de recherche quelconque

Il existe deux techniques d'insertion d'une nouvelle clé dans un arbre binaire de recherche : au niveau des feuilles ou au niveau de la racine.
La première est très semblable à celle employée pour rechercher un élément. Pour insérer le couple $(k,u) \in \mathrm{C}\times\mathrm{V}$ dans l'arbre A on procède ainsi :

- si $\mathrm{A} = \mathsf{nil}$, on retourne l'arbre $(\mathsf{nil},(k,u),\mathsf{nil})$;
- si $\mathrm{A} = (\mathrm{F}_g,(c,v),\mathrm{F}_d)$ alors :
  - si $c = k$ on remplace [9] 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.

9. Lorsqu'on autorise la présence de clés identiques dans un ABR, on souhaite que la fonction **recherche** renvoie la valeur associée à la clé la plus récemment introduite, autrement dit que celle-ci se trouve à la profondeur minimale.

```
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) ;;
```

L'autre possibilité consiste à insérer le nouvel élément à la racine puis à partitionner l'arbre de manière à placer 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) ;;
```

On trouvera figure 6 un exemple d'insertion d'une clé par ces deux techniques dans l'ABR représenté figure 5.

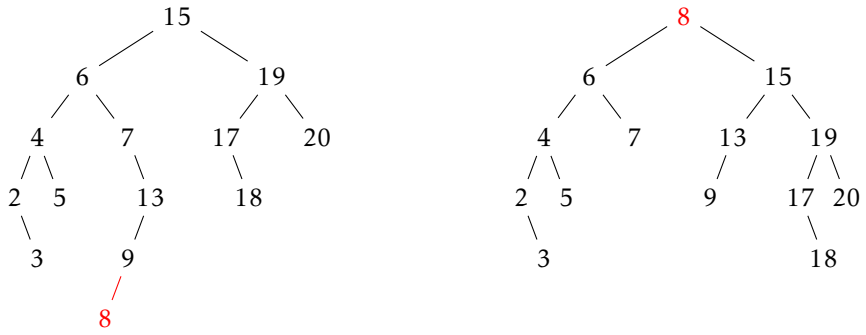


FIGURE 6 – Insertion au niveau des feuilles et au niveau de la racine d'une clé de valeur 8.

Ces fonctions d'insertion ont toutes deux un coût en $O(h(A))$.

La suppression d'une clé k dans un arbre binaire de recherche consiste à supprimer le couple (k, v) situé à la profondeur minimale dans l'arbre $A = (F_g, (c, v), F_d)$. 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 un élément m de clé minimale pour obtenir F'_d et on retourne (F_g, m, F'_d).

Nous aurons besoin d'une fonction qui prend en argument un arbre A et retourne le couple (m, A') formé d'un élément m de clé minimale et de l'arbre $A' = A \setminus \{m\}$:

```
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) ;;
```

La suppression d'une clé k se réalise alors ainsi :

```
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) ;;
```

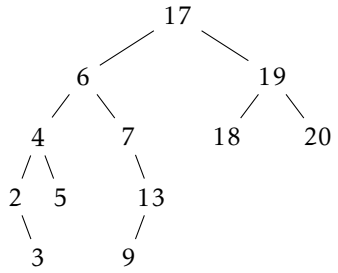


Figure 7 – Suppression de la clé de valeur 15 dans l'ABR de la figure 5

La fonction de suppression a elle aussi un coût temporel en $O(h(A))$ puisque chaque appel récursif s'effectue sur un arbre de hauteur inférieur au précédent.

• Le problème du déséquilibre

Nous avons vu que toutes les fonctions de requêtes, d'insertion et de suppression dans un arbre de recherche ont un coût temporel 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.

Malheureusement, les fonctions d'insertion et de suppression que nous avons écrites peuvent conduire à des arbres très déséquilibrés, par exemple des arbres peignes dans le cas où les clés sont insérées par ordre croissant ou décroissant.

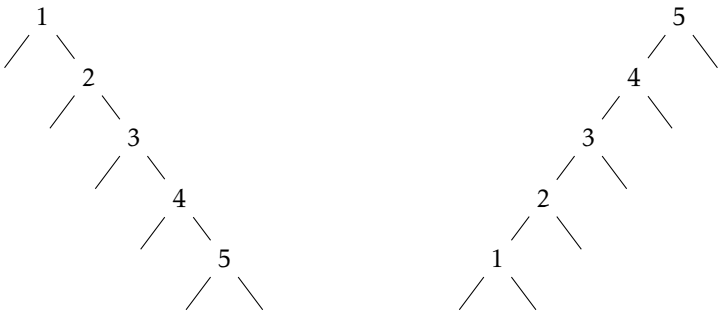
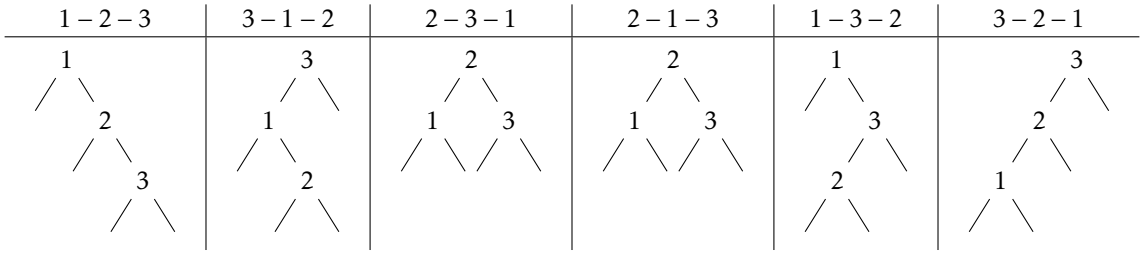


Figure 8 – Insertion des clés 1, 2, 3, 4, 5 au niveau des feuilles ou au niveau de la racine.

Cependant, on peut démontrer (mais c'est difficile) que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable. Plus précisément, si un arbre binaire de recherche est créé en insérant n clés distinctes au niveau des feuilles dans un ordre aléatoire (chacune des $n!$ permutations étant équiprobable), il existe une constante c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c \log n$.
Par exemple, les six arbres que l'on peut obtenir en insérant dans un ordre arbitraire les trois entiers 1, 2 et 3 sont les suivants :



On peut constater que cette manière de faire diffère de celle consistant à supposer que chaque arbre binaire de recherche à n nœuds est équiprobable : l'arbre binaire complet a la probabilité 1/3 d'apparaitre, contre 1/6 pour les quatre autres [10].

10. La hauteur moyenne d'un arbre binaire de recherche de taille n s'ils sont tous équiprobables est équivalente à $2\sqrt{\pi n}$.

• Insertion et suppression dans un arbre binaire équilibré

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. (l'utilisation d'arbres AVL pour représenter des arbres binaires de recherche est étudiée dans l'exercice 9).

2.4 Arbres binaires de recherche 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.

Il a été montré qu'une *table de hachage* permet la réalisation d'une structure impérative de dictionnaire, avec les coûts suivants :

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

La structure d'arbre binaire de recherche 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.

3. Tas binaire

3.1 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.

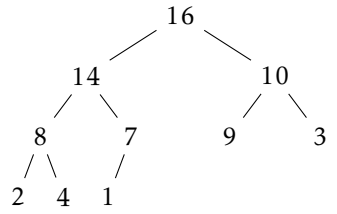


FIGURE 9 – Un exemple de tas-max.

Si A est un arbre parfait, alors $1 + 2 + \cdots + 2^{h(A)-1} < |A| \leqslant 1 + 2 + \cdots + 2^{h(A)}$ donc $2^{h(A)} \leqslant |A| < 2^{h(A)+1}$. La hauteur d'un arbre parfait à n nœuds est égale à $\lfloor \log n \rfloor$; c'est un arbre équilibré.

On appelle *tas-max* un arbre parfait étiqueté par un ensemble ordonné E tel que l'étiquette de chaque nœud autre que la racine soit inférieure ou égale à l'étiquette de son père.
Bien entendu, 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.

Implémentation d'un tas
Comme tout arbre binaire, un tas peut être représenté par le type *'a arbre*, mais une autre représentation est possible en utilisant un vecteur de type *'a vect*. Rappelons que la numérotation Sosa-Stradonitz des nœuds d'un arbre binaire se définit ainsi :

- 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 $2k+1$.

De ceci il résulte immédiatement que le père d'un nœud de numéro k porte le numéro $\left\lfloor \frac{k}{2} \right\rfloor$.
Il est donc possible d'établir une correspondance entre l'indice $k \geqslant 1$ d'un tableau et la numérotation d'un nœud. Dans le cas d'un arbre binaire quelconque, cette représentation ne présente en général pas d'intérêt car le nombre de cases inutilisées peut être très important (dans le cas extrême d'un arbre peigne gauche, seules les cases correspondant aux puissances de 2 sont utilisées, ce qui nécessiterait un coût spatial exponentiel pour le représenter). En revanche, pour un arbre parfait, seules les cases numérotées entre 1 et n sont utilisées pour représenter un arbre de taille n.

16	14	10	8	7	9	3	2	4	1

Figure 10 – Une représentation tabulaire du tas-max de la figure 9.

On notera que les tableaux Caml sont indexés à partir du rang 0, ce qui nous laisse deux possibilités : ou bien ne pas utiliser la case d'indice 0 et conserver la numérotation présentée ci-dessus, ou bien décaler la numérotation d'un cran. C'est ce que nous allons faire dans la suite de ce chapitre, en adoptant désormais la convention suivante :

- 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 $\left\lfloor \frac{k-1}{2} \right\rfloor$.

3.2 Opérations usuelles sur les tas

Dans ce qui suit, nous ne prendrons en compte que des tas-max, mais il va de soit que toutes les fonctions que nous écrirons s'adaptent sans difficulté au cas des tas-min.

• Préservation de la structure de tas

Une des opérations les plus fréquentes sur les tas consiste à reconstituer la structure de tas après qu'un élément ait 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 ; dans le cas contraire, il se peut que cet élément ait à « descendre » dans l'arbre.

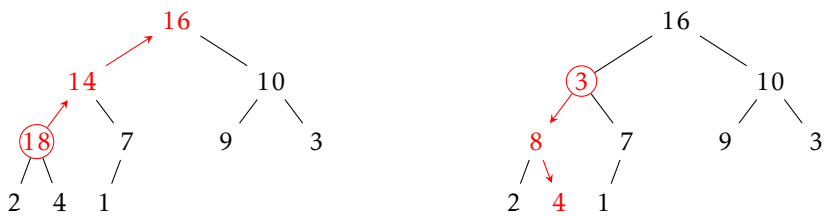


Figure 11 – À gauche, l'élément entouré doit monter dans le tas-max ; à droite il doit descendre.

La montée est plus simple : il suffit de permuter l'élément modifié avec son père tant que la structure de tas n'est pas reconstituée :

```
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 ;;
```

Pour la descente, c'est un peu plus délicat : il faut permuter l'élément modifié avec le plus grand de ses fils, s'il en existe. On introduit un paramètre supplémentaire qui indique l'indice de la première case non utilisée dans la représentation du tas (si le tableau **t** est plein, on aura donc **n = vect_length t**) [11].
Trois cas sont possibles : si $2i+2 < n$, t_i possède deux fils t_{2i+1} et t_{2i+2} ; si $n = 2i+2$, t_i possède un seul fils t_{2i+1} ; si $2i+1 > n$ t_i ne possède pas de 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 de chacune des ces deux fonctions dans un tas T est un $O(h(T))$ donc un $O(\log n)$, l'entier n désignant la taille du tas.

• construction d'un tas

Il y a deux façons de convertir un tableau t en un tas-max. La première consiste à maintenir 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** :

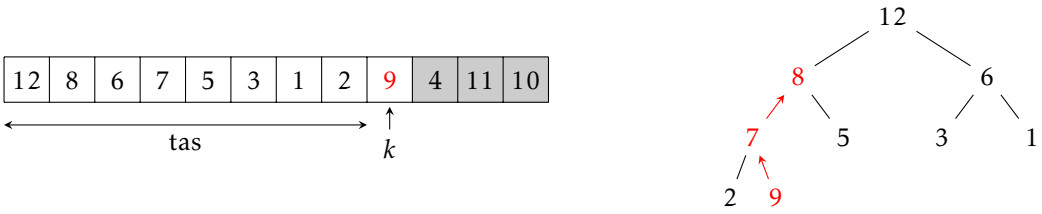


FIGURE 12 – Une étape de la construction d'un tas par remontée des nœuds.

```
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 (c'est le cas par exemple lorsque le tableau est initialement trié par ordre croissant). Dans ce cas, le nombre de permutations effectuées est égal à la somme des profondeurs de chacun des nœuds à l'exception de la racine. Si on note $p = \lfloor \log n \rfloor$ la hauteur du tas, le nombre de permutations est donc égal à :

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

On peut faire mieux en observant (voir l'exercice 11) que la deuxième moitié du tableau t correspond aux feuilles de l'arbre et que chacune de ces feuilles est un tas à un élément. Il suffit donc de faire descendre l'élément t_k pour unir peu à peu ces différents tas, de manière à préserver l'invariant : « chaque nœud de $t[k..n-1]$ est la racine d'un tas-max ».

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

11. L'intérêt de ce paramètre n'apparaitra que dans la section 3.3.

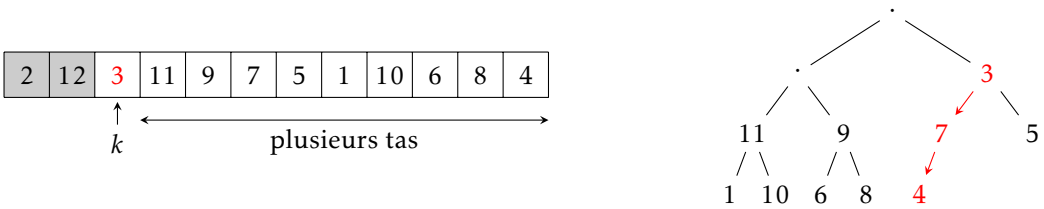


FIGURE 13 – Une étape de la construction d'un tas par descente des pères.

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 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ (voir l'exercice 11) donc le nombre total d'échanges est majoré par :

$$\sum_{h=1}^{p} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \leqslant \frac{p(p+1)}{2} + n \sum_{h=1}^{p} \frac{h}{2^{h+1}} = \Theta(n) \quad \text{car } p = \lfloor \log n \rfloor \text{ et la série } \sum \frac{h}{2^{h+1}} \text{ converge.}$$

Cette deuxième méthode permet donc de construire un tas-max en temps linéaire.

3.3 Tri par tas

La notion de tas conduit naturellement à un algorithme de tri appelé le tri par tas (*heap sort* en anglais) : pour trier un tableau t, on commence par le transformer en tas-max à l'aide de la fonction **cree_tas**. À l'issue de cette première étape, l'élément maximal se trouve à la racine t_0 et on peut le placer dans sa position finale en le permutant avec t_{n-1}. Il faut ensuite reformer le tas entre les indices 0 et $n-2$ en faisant descendre t_{n-1} à l'aide de la fonction **descend**. Il reste à réitérer ce processus pour le tas-max de taille $n-1$, jusqu'à arriver à un tas de taille 1.

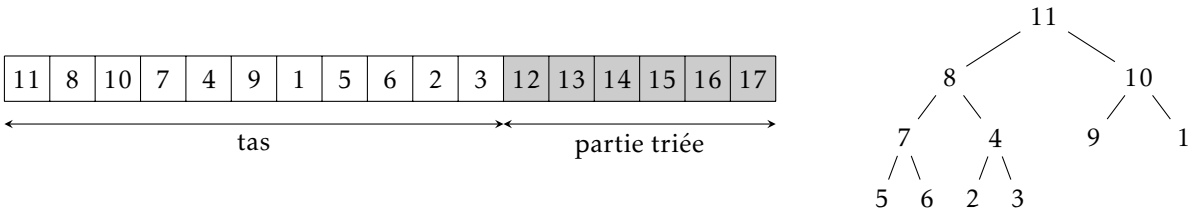


FIGURE 14 – Un tableau en cours de tri.

```
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 ;;
```

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

Complexité temporelle
Puisque la fonction **cree_tas** a un coût temporel en $O(n)$ et que chacune des appels à la fonction **descend** un coût en $O(\log n)$, le coût total de cette méthode de tri est un $O(n \log n)$.

3.4 Files de priorité

Une des applications des tas est la mise en œuvre des files de priorités, qui permettent de gérer un ensemble V de valeurs associées chacune à une priorité, élément d'un ensemble totalement ordonné.
Par exemple, une file de priorité permet de planifier les tâches d'un ordinateur : la file gère les travaux en attente, avec leurs priorités relatives. Lorsqu'une tâche est terminée ou interrompue, l'ordinateur exécute la tâche de plus forte priorité parmi les travaux en attente.

Une file de priorité supporte en général les opérations suivantes :

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

Nous définissons donc le type :

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

Puisque le tas est représenté par un vecteur, il est nécessaire de choisir la taille maximale du tas au moment de sa création, et puisqu'il est destiné à évoluer en taille, il faut garder trace de l'indice de la première case disponible. On utilise donc le type suivant :

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

Le champ mutable `N` désigne l'indice de la première case disponible, ce qui permettra de faire grandir le tas.

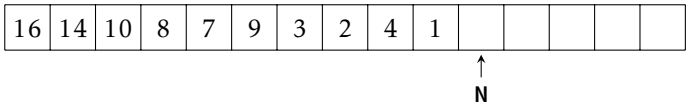


FIGURE 15 – Une représentation dynamique du tas-max de la figure 9.

On définit deux exceptions :

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

et on modifie légèrement les définitions des fonctions **monte** et **descend** pour tenir compte du type de données utilisé (il s'agit maintenant de comparer les priorités des éléments du tableau) :

```
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 ;;
```

La création d'une file de priorité vide nécessite de fixer au départ la taille maximale du tas et d'attribuer une valeur arbitraire aux éléments du tableau pour fixer le type :

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

L'insertion consiste à placer le nouvel élément en dernière position puis à le faire remonter :

```
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 ;;
```

Pour le retrait de la valeur de priorité maximale, on permute le dernier élément du tas avec la racine puis on le fait descendre :

```
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 ;;
```

Enfin, après avoir accru la valeur d'une clé, il suffit de la faire remonter pour reformer le tas :

```
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 ;;
```

4. Exercices

4.1 Arbres binaires

Exercice 1 Définir une fonction Caml qui prend en arguments un entier p et un arbre binaire A et qui retourne la liste des sous-arbres non vides dont la racine est à la profondeur p dans A.
Lorsque A est un arbre binaire complet de hauteur p, combien d'insertions en tête de liste votre fonction effectue-t-elle ?

Exercice 2 Un arbre binaire A est *complet* lorsque ses fils gauche et droit sont complets et de même hauteur. Pour déterminer si un arbre binaire est complet, on propose la fonction suivante :

```
let rec est_complet = function
  | Nil               -> true
  | Noeud (fg, _, fd) -> est_complet fg && est_complet fd
                         && hauteur fg = hauteur fd ;;
```

Évaluer en fonction de $n = |A|$ le coût temporel de cette fonction. Peut-on faire mieux ?

Exercice 3 Définir une fonction Caml qui prend en argument un entier n et qui retourne le squelette d'un arbre binaire complet de taille n s'il en existe, et déclenche une exception sinon. On utilisera le type :

```
type squelette = Nil | Noeud of squelette * squelette ;;
```

Exercice 4 Un arbre de Fibonacci d'ordre p est :

- l'arbre vide si $p = 0$;
- un arbre réduit à une feuille si $p = 1$;
- un arbre dont le fils gauche est un arbre de Fibonacci d'ordre $p-1$ et le fils droit un arbre de Fibonacci d'ordre $p-2$ si $p \geqslant 2$.

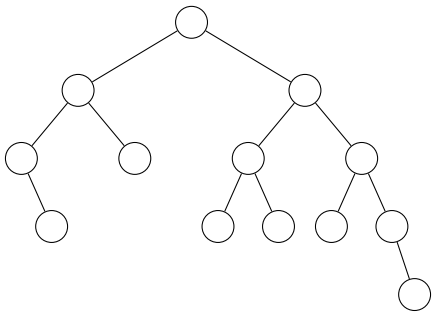
Dessiner le squelette d'un arbre de Fibonacci d'ordre 5. Combien un arbre de Fibonacci possède-t-il de feuilles ? Calculer le déséquilibre en tout nœud d'un arbre de Fibonacci.

Exercice 5 Définir une fonction Caml qui détermine avec une complexité en $O(|A|)$ si un arbre binaire A est un arbre AVL.

Exercice 6 On appelle arbre *rouge-noir* un arbre binaire comportant un champ supplémentaire par nœud : sa *couleur*, qui peut être rouge ou noire, et qui vérifie les conditions suivantes :

- la racine est noire ;
- le parent d'un nœud rouge est noir ;
- pour chaque nœud, tous les chemins le reliant à des feuilles (`nil`) contiennent le même nombre de nœuds noirs.

a) Montrer que l'arbre ci-dessous peut être muni d'une coloration rouge-noir.



b) Donner un exemple d'arbre qui ne puisse pas être muni d'une coloration rouge-noir.

c) Étant donné un arbre rouge-noir A, on note $b(A)$ le nombre de nœuds noirs que contient chacun des chemins d'un nil à la racine (indépendant du choix de la feuille par définition).

Montrer que $b(A) \leqslant h(A)+1 \leqslant 2b(A)$ et que $|A| \geqslant 2^{b(A)}-1$, et en déduire que les arbres rouge-noir sont équilibrés.

d) On définit le type :

```
type couleur = Rouge | Noir ;;
```

Définir en Caml une fonction qui détermine si un élément de type *couleur arbre* est un arbre rouge-noir. L'algorithme choisi devra être de coût linéaire vis-à-vis de $|A|$.

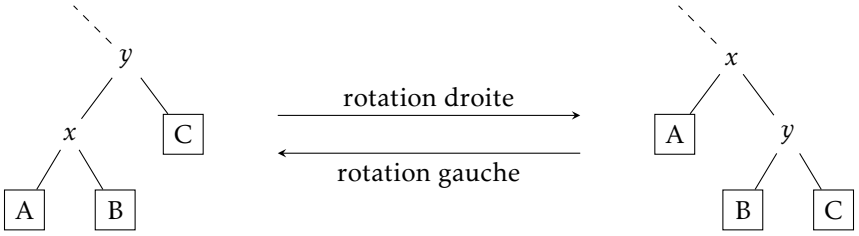
4.2 Arbres binaires de recherche

Exercice 7 Ecrire une fonction Caml de type *('a, 'b) data arbre -> bool* qui détermine si un arbre passé en paramètre est un ABR.

Exercice 8 On peut trier un tableau de n nombres en commençant par les insérer un par un dans un arbre binaire de recherche puis en les ordonnant suivant un parcours infixe. Rédiger la fonction Caml correspondante puis étudier les temps d'exécution de cet algorithme dans le pire et le meilleur des cas.

Exercice 9 Proposer plusieurs solutions pour fusionner deux arbres binaires de recherche A_1 et A_2 et rédiger en Caml celle qui vous parait la meilleure.

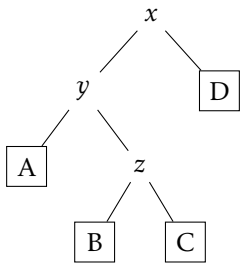
Exercice 10 Soit A un arbre binaire de recherche et y l'un de ses nœuds. Une *rotation droite* autour de y consiste à faire descendre ce nœud et à faire remonter sont fils gauche x sans invalider l'ordre des éléments :



L'opération inverse s'appelle *rotation gauche* autour du sommet x.

a) Montrer qu'une rotation préserve la structure d'arbre binaire de recherche et peut se réaliser à coût constant lorsqu'elle est effectuée autour de la racine (on écrira explicitement les deux fonctions **rotd** et **rotg** correspondantes, de type *'a arbre -> 'a arbre*).

b) Appliquer une rotation gauche autour de y puis une rotation droite autour de x dans l'arbre suivant :



c) On considère désormais un arbre binaire de recherche qui respecte la condition AVL, à savoir que le déséquilibre de chaque nœud est égal à -1, 0 ou 1, et on considère une insertion qui a conduit à la création d'une feuille f. Seuls les ancêtres de f ont vu leur déséquilibre modifié ; on suppose que l'un au moins ne respecte plus la condition AVL et on note y le premier de ceux-là.

(i) Montrer que $\mathrm{eq}(y) = \pm 2$. Sans perte de généralité on suppose désormais $\mathrm{eq}(y) = 2$ et on note x le fils gauche de y.

(ii) Montrer que si $\mathrm{eq}(x) = 0$ ou $\mathrm{eq}(x) = 1$ une rotation suffit pour retrouver la condition AVL.

(iii) Montrer que si $\mathrm{eq}(x) = -1$, deux rotations suffisent pour retrouver la condition AVL.

4.3 Tas binaires

Exercice 11

a) Définir une fonction Caml **`tab_of_arbre`** qui prend en argument un arbre (parfait) de type *int arbre* et qui retourne sa représentation sous la forme d'un tableau de type *int vect*.

b) Définir la fonction inverse **`arbre_of_tab`**.

Exercice 12

a) Montrer que dans un tas représenté par un tableau Caml à n cases les feuilles correspondent aux indices supérieurs ou égaux à $\left\lfloor \frac{n}{2} \right\rfloor$.

b) Montrer que dans un tas de taille n le nombre de noeuds de hauteur h est au plus égal à $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Exercice 13 Rédiger en Caml une fonction qui retourne l'élément maximal d'un tas-min.

Exercice 14 Un arbre *binomial* d'ordre p est :

- un arbre réduit à une feuille si $p = 0$;
- un arbre qui admet p fils qui sont respectivement des arbres binomiaux d'ordre $p-1, p-2, \ldots, 1, 0$.

a) Dessiner le squelette d'un arbre binomial d'ordre 4, puis exprimer en fonction de p :

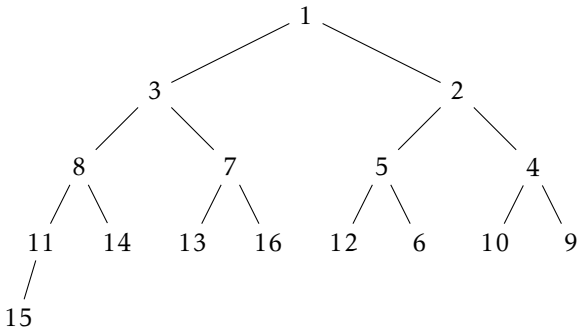
- la taille d'un arbre binomial d'ordre p ;
- la hauteur d'un arbre binomial d'ordre p ;
- le nombre de nœuds de profondeur k d'un arbre binomial d'ordre p.

Un *tas binomial* est une arbre binomial qui a une structure de tas : l'information portée par un nœud est (dans le cas d'un tas-min) inférieure ou égale à l'information portée par chacun de ses fils. Nous allons étudier un algorithme de conversion d'un tas binaire de taille 2^p en un tas binomial d'ordre p sans faire aucune comparaison.

b) Un tas binaire A de taille 2^p ($p \geqslant 1$) est constitué d'une racine r, d'un sous-arbre gauche F_g qui est un tas de taille 2^{p-1} et d'un sous-arbre droit F_d qui est un tas de taille $2^{p-1} - 1$. Décrire un algorithme qui transforme en un tas binaire de taille 2^{p-1} le sous-arbre droit F_d et la racine r. Précisez le nombre d'opérations effectuées.

c) Les deux tas binaires obtenus, F_g et le nouveau tas créé à la question précédente, sont transformés récursivement en tas binomiaux d'ordre $p-1$. Expliquer comment les réunir pour former un tas binomial d'ordre p.

d) Illustrer cet algorithme en transformant en tas binomial le tas binaire suivant :



e) Calculer le nombre total d'échanges dans un tas binaire ainsi que le nombre de concaténations de tas binomiaux effectués lors de la conversion d'un tas binaire de taille 2^p. Sachant que ces opérations peuvent être réalisées à coût constant, quel est le coût total de l'algorithme ?