3 Une première structure de donnée hiérarchique : les arbres

Compétences attendues en fin de chapitre

- Connaître le vocabulaire de base des arbres ;
- savoir définir et manipuler différents types d'arbres selon les propriétés attendues : arbres binaires, binaire strict ou généraux, type différent pour les feuilles et les nœuds, etc. ;
- définir une structure de données associée à un type abstrait ;
- connaître et savoir démontrer les Propriétés 6 et 8 ;
- savoir définir et manipuler des parcours d'arbres en profondeur (préfixe, infixe, postfixe) et en largeur ;
- définir et connaître les propriétés des arbres binaires de recherche simples (sans équilibrage) ;
- implémenter et déterminer la complexité des opérations de base sur les arbres binaires de recherche simples (recherche, insertion, suppression simple) ;
- connaître le principe des arbres rouge-noir, savoir démontrer la Propriété 22, et savoir implémenter les opérations de base sur les arbres rouge-noir (au moins jusqu'à l'insertion) [a].
- Définir la notion de tas-min, implémenter les tas-min et les opérations de base ; connaître la complexité des opérations de base. On complètera cette implémentation pour définir des files de priorité dans le **??**.

Notez que l'on va majoritairement traiter de ces notions en OCaml, le langage étant très adapté à la manipulation de types récursifs. Cependant, on définira également les arbres en C, et on pourra appliquer les mêmes notions et algorithmes en C ou en OCaml.

a. Ce point est sans doute l'un des plus difficiles du programme de MP2I en informatique : la priorité est d'abord de connaître les points précédents, mais un concours d'un niveau un peu relevé pourrait attendre une forme d'aisance de votre part sur ces notions.

I Définitions

Un arbre est une structure hiérarchique, constituée de nœuds potentiellement étiquetés et organisé par des relations de filiation. Commençons par un exemple :

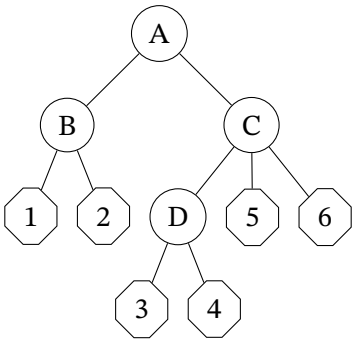
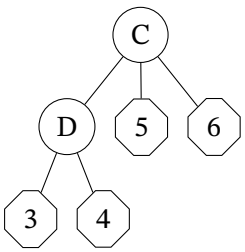


FIGURE 3.1 – Un arbre étiqueté

D'abord, un peu de vocabulaire sur les arbres :

- Un arbre est composé de **nœuds** : ici, il y en a 10.
- Les nœuds peuvent être **étiquetés** par des valeurs. Ici, le nœud le plus haut a pour étiquette *A*. *Souvent, on parlera du nœud A directement, sauf quand plusieurs nœuds portent la même étiquette.*

— Certains de ces nœuds ont des **enfants**, c'est-à-dire des nœuds dont ils sont issus. Par exemple, le nœud C a trois enfants : les nœuds D, 5 et 6. Le nœud 2 n'a pas d'enfants.
— Pour $\mathfrak{n}$ un nœud d'un arbre A, on définit le sous-arbre enraciné en $\mathfrak{n}$ comme l'arbre de racine $\mathfrak{n}$, dans lequel on ne garde que les enfants successifs de $\mathfrak{n}$. Par exemple, l'arbre suivant est le sous-arbre de A enraciné en C :



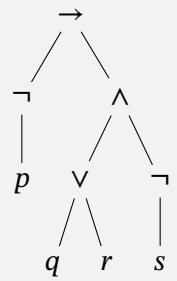
— Dans le sens inverse, un nœud peut avoir un **parent** dont il est le fils. Le père du nœud 2 est B, et A n'a pas de père. Dans un arbre, un seul nœud n'a pas de père : on l'appelle **racine** de l'arbre. Dans l'exemple, la racine de l'arbre est le nœud A. Les autres nœuds ont exactement un père. On parlera aussi d'**ancêtre** pour parler des parents successifs d'un nœud. Enfin, on parlera de **frères** pour deux nœuds ayant le même père.
— L'**arité** d'un nœud est son nombre de fils. Le nœud D a pour arité 2, le nœud 4 a pour arité 0. Un nœud d'arité 0 est appelé une **feuille**, un nœud d'arité non nulle est appelé **nœud interne**.
— La **profondeur** d'un nœud est la longueur du chemin le reliant à la racine, comptée comme le nombre de nœuds visité pour y arriver sans compter le nœud lui-même. La profondeur de la racine est de 0, celle de ses fils est à 1. La profondeur du nœud étiqueté 4 est 3.
— La **hauteur** d'un arbre est la profondeur maximale de ses nœuds. Ici, elle est de 3.

On remarque que l'on a utilisé deux types différents pour les nœuds internes et les feuilles.

On a utilisé des structures séquentielles pour représenter des éléments organisés séquentiellement, c'est-à-dire les uns à la suite des autres. L'intérêt des structures hiérarchiques comme les arbres est de pouvoir représenter des relations de hiérarchie, d'hérédité, et plus généralement des organisations plus complexes que les structures hiérarchiques. Reprenons quelques exemples du début du cours :

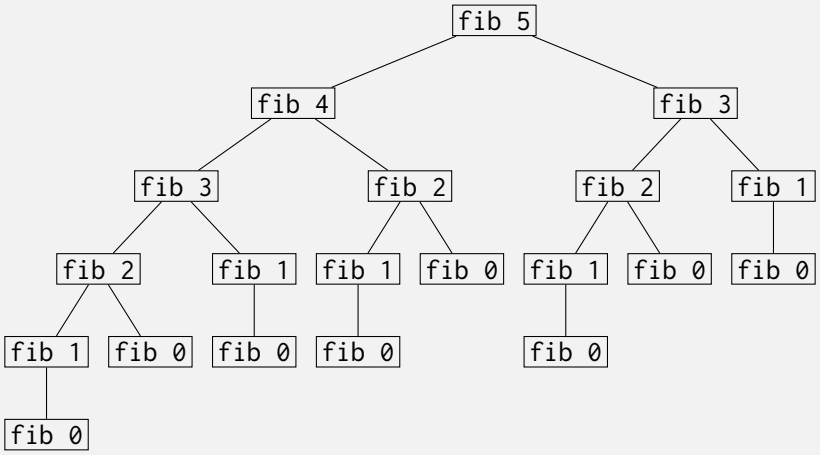
Exemple 1 – Arbre syntaxique de formules de la logique propositionnelle

L'arbre syntaxique de la formule $(\neg p) \to (q \vee r) \wedge \neg s$ est :



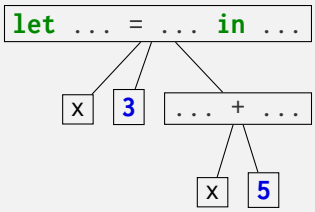
Exemple 2 – Arbre d'appels récursifs de la fonction de Fibonnacci

On peut représenter l'exécution d'un appel à la fonction de Fibonacci comme un arbre où chaque nœud est un appel récursif :



Exemple 3 – Arbre syntaxique abstrait d'un programme

Une arbre syntaxique abstrait d'un programme est une façon de représenter la structure d'un programme à partir de son code source. Notamment, c'est une étape fondamentale de la *compilation* d'un programme, puisque c'est cet arbre qui est utilisé par le compilateur pour déterminer exactement la structure du programme et produire un exécutable y correspondant. Par exemple, pour le programme `let x = 3 in x + 5`, l'arbre syntaxique associé est :



Or, OCaml est un langage particulièrement adapté pour représenter, calculer et analyser de manière riche des arbres, en représentant un constructeur pour les variables, un constructeur pour chaque forme d'expression, un constructeur pour chaque constante littérale, un constructeur pour chaque opérateur / fonction…OCaml est ainsi souvent utilisé pour programmer rapidement un nouveau langage et son compilateur (par exemple, le premier compilateur Rust était écrit en OCaml [a]). Il permet aussi d'analyser rapidement quelle est la portée de chaque variable, où est initialisée une variable donnée dans le programme et évaluer une expression à partir de son arbre syntaxique est évident (pour un langage fonctionnel). Essayez donc comment l'expression suivante est évaluée sans machine :

OCaml
```
let a = 30 in
let a =
  let a = 3 in a * 4
in a + a
```

a. `https://github.com/rust-lang/rust/tree/ef75860a0a72f79f97216f8aaa5b388d98da6480/src/boot` pour un exemple d'à quoi ressemble un compilateur complet écrit en OCaml, notamment `https://github.com/rust-lang/rust/blob/ef75860a0a72f79f97216f8aaa5b388d98da6480/src/boot/fe/ast.ml` qui contient la description de l'arbre syntaxique du langage par des types arborescents en OCaml.

On voit ici que les arbres ont différentes interprétations et notations selon les situations : on fera donc **très attention** à bien lire les énoncés de TD, TP, DS, pour bien comprendre quelle est l'interprétation que l'on donne aux nœuds et aux relations entre eux.

II Définitions possibles des arbres

Les définitions précises des arbres sont multiples et les différences entre elles sont souvent subtiles et peu importantes. Pour les arbres généraux, on OCaml, on utilisera souvent le type concret suivant :

OCaml
```
type ('a, 'b) general_tree =
  | Leaf of 'b
  | Node of 'a * ('a, 'b) general_tree list

let exemple =
  Node ( 'A',
         [
           Node ('B', [ Leaf 1; Leaf 2 ]);
           Node ('C', [ Node ('D', [ Leaf 3; Leaf 4 ]); Leaf 5; Leaf 6 ]);
         ] )
```

On utilisera ce type si on a besoin d'arbres généraux (pas forcément binaires) et que les feuilles aient un type différent des nœuds. Si on considère que `'a = 'b`, on pourra utiliser le type suivant :

OCaml
```
type 'a tree =
  | N of 'a * 'a tree list

(* où une feuille est représentée par : *)
let feuille etiquette = N ( etiquette, [] )
```

Exercice 13

Écrire les fonctions suivantes en OCaml, portant sur les arbres définis comme le type `'a tree` :

1. `etiquette_racine : 'a tree -> 'a` qui à un arbre renvoie l'étiquette de sa racine,
2. `liste_enfants : 'a tree -> 'a tree list` qui à un arbre renvoie la liste des enfants de la racine.
3. `contient : 'a -> 'a tree -> bool` qui à une étiquette et un arbre renvoie si un des nœuds de l'arbre contient l'étiquette.
4. `sous-arbre_opt : 'a -> 'a tree -> 'a tree option` qui à une étiquette x et un arbre a renvoie `Some` d'un sous-arbre de a enraciné en x s'il existe, `None` sinon.
5. `descendant_gauche : 'a tree -> 'a` qui renvoie l'étiquette la plus "à gauche" de l'arbre (on s'arrête quand un nœud n'a plus d'enfants). Idem pour `descendant_droite`.

On utilisera souvent des arbres binaires, que l'on a déjà défini dans l'Exemple 15, Chapitre 6.

Définition 4 – Arbres binaires

L'ensemble des *arbres binaires* est définie par induction structurelle :
— L'arbre vide $\varnothing$ est un arbre binaire.
— Si $\mathcal{A}$ et $\mathcal{B}$ sont deux arbres binaires, alors $Noeud(\mathcal{A}, \mathcal{B})$ est un arbre binaire.

Le type OCaml qui correspondrait à cette définition est :

OCaml
```
type tree =
  | Empty
  | Node of binary_tree * binary_tree
```

Et l'on représente des arbres binaires étiquetés par :

En OCaml, on utilise les types sommes et un constructeur paramétré pour les nœuds.

OCaml
```
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree
```

En C, on utilisera les `struct` et des pointeurs pour représenter les relations hiérarchiques dans un arbre :

C
```
struct tree = {
  char* label;
  tree* left_son;
  tree* right_son;
}
```

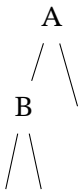
Avec un pointeur `NULL` pour représenter un arbre vide.

Exercice 14 – Programmation sur les arbres binaires

Définir les fonctions suivantes :

OCaml
```
val nombre_noeuds : 'a tree -> int
val hauteur : 'a tree -> int
val liste_sous_arbres : 'a tree -> 'a tree list
```

Ici, il y a une imprécision avec ces implémentations : est-ce que le constructeur `Vide` (ou le pointeur `NULL`) représente l'absence de fils, ou la présence d'un fils vide ? On préférera la première possibilité : un arbre de la forme `Noeud(10, Vide, Vide)` est donc un arbre avec qu'un seul nœud d'arité 0. Ainsi, au lieu d'écrire des arbres binaires avec des fils qui pendent vers des abres vides :



On écrira simplement :

Les arbres binaires définis jusqu'alors sont dits *non stricts* : un nœud peut avoir exactement un fils, qui est son fils gauche ou son fils droit (et l'autre fils est vide).

Définition 5 – Arbre binaire strict

On dit qu'un arbre binaire est *strict* si l'arité de ses nœuds est de 0 ou 2.

Je le rappelle parce que c'est important : un sujet pourra définir différemment les différentes notions présentées jusqu'à maintenant. Il est essentiel de bien lire le sujet qui sera (normalement) suffisamment précis pour reconnaître clairement les notions.

Pour représenter les arbres binaires stricts, on pourra utiliser la notion de *feuille* qui est un arbre composé d'un seul nœud (ou encore que c'est un nœud dont les fils sont vides) :

OCaml
```
type 'a strict_tree =
  | Leaf of 'a
  | Internal_Node of 'a * 'a strict_tree * 'a strict_tree
```

Ce type permet de s'assurer "par conception" que l'on ne construit que des arbres binaires stricts. Cela ne veut pas dire qu'un arbre binaire représenté par un `'a tree` ne peut pas être strict :

Exercice 15 – Traduction arbre binaire strict ↔ arbre binaire non strict

En utilisant les deux types `'a tree` et `'a strict_tree`, écrire les fonctions correspondant aux spécifications suivantes :
- `est_strict : 'a tree -> bool` qui vérifie qu'un arbre binaire est strict
- `tree_to_strict_tree : 'a tree -> 'a strict_tree` qui à un arbre binaire strict de type `'a tree` renvoie le même arbre, mais dans le type spécifique aux arbres binaires stricts `'a strict_tree`. La fonction soulève une erreur si l'arbre en entrée n'est pas strict.
- `strict_tree_to_tree : 'a strict_tree -> 'a tree` qui à un arbre binaire strict de type `'a strict_tree` renvoie le même arbre, mais exprimé dans le type `'a tree`.

Si vous avez besoin de choisir entre toutes les représentations des arbres binaires, en C ou en OCaml, la méthode est la suivante :
- D'abord, si vous êtes au concours à l'écrit ou à l'oral, le concours vous l'imposera probablement et il est de bon ton de ne pas le contredire.
- Si vous êtes au concours et qu'on vous demande de choisir, ou que vous reprenez votre cours après avoir commencé à travailler comme ingénieur 🤩, il y a trois critères :
  1. est-ce que les arbres sont généraux ou binaires ? (ou d'arité bornée ?)
  2. Est-ce que les nœuds sont étiquetés ? Si oui, est-ce que les nœuds internes sont étiquetés différemment des feuilles ?
  3. Est-ce qu'il faut pouvoir représenter un arbre vide, et donc un nœud avec un fils gauche mais pas de fils droit, ou inversement ?

Remarquez aussi qu'il est sans doute plus facile et efficace de manipuler des arbres avec OCaml, grâce à la richesse du filtrage par motif et le fait qu'on n'ait pas à s'embêter avec les pointeurs dans tous les sens.

III Propriétés élémentaires des arbres

Dans cette section, on va voir des égalités classiques sur les arbres : notamment, on va mettre en relation les quantités que l'on a définies jusqu'ici. Chacune de ces démonstrations va faire intervenir des *définitions par induction* des arbres binaires : selon les cas, l'ensemble de base sera l'arbre vide ou les feuilles.

Propriété 6 – Relation entre le nombre de feuilles et de nœuds internes

Dans un arbre binaire strict A, pour $n_{interne}(A)$ le nombre de nœuds internes et $f(A)$ le nombre de feuilles, on a $f(A) = n_{interne}(A) + 1$.

Démonstration. On le montre par induction sur les arbres binaires stricts :
Initialisation : Si A est une feuille, alors on a $f(A) = 1$ et $n_{interne}(A) = 0$, donc $f(A) = n_{interne}(A) + 1$.
Induction : Soit A un arbre de fils gauche B et de fils droit C où la propriété est vraie. Alors on a $f(A) = f(B) + f(C)$ (les feuilles de A sont soit dans le fils gauche soit dans le fils droit) et $n_{interne}(A) = n_{interne}(B) + n_{interne}(C) + 1$ (idem, auquel on ajoute la racine). Par hypothèse d'induction, on a donc :

$$\begin{aligned} f(A) &= n_{interne}(B) + 1 + n_{interne}(C) + 1 \\ &= n_{interne}(A) + 1 \end{aligned}$$

□

Remarque 7 – Sur la notion d'induction structurelle

Dans cette preuve, on utilise la notion d'*induction structurelle* : c'est une notion similaire à l'induction vue dans le Chapitre 6. On montre qu'une propriété est vraie pour toutes les structures obtenues en suivant des règles de constructions, avec un cas de base et l'utilisation de *constructeurs* pour construire depuis des sous-structures une structure plus grande. Elle diffère en fait seulement dans le rôle joué par les *constructeurs*, qui ne sont pas exactement considérés comme des fonctions comme en induction. En pratique, vous pouvez continuer à rédiger les preuves par induction structurelle comme les preuves par induction.

Exercice 16 – Contre-exemple

Proposer un arbre binaire non strict pour lequel la Propriété 6 est fausse. Que peut-on dire quand même ?

Exercice 17 – Généralisation

Pour $k \in \mathbb{N}$, on dit qu'un arbre k-aire est strict si tous ses nœuds sont d'arité nulle ou égale à k. Déterminer une relation similaire à celle de la Propriété 6 pour les arbres k-aires.

Propriété 8 – Le nombre de nœuds est exponentiel en la hauteur

Pour un arbre binaire A, on a les inégalités suivantes :

$$h(A) + 1 \leq n(A) \leq 2^{h(A)+1} - 1$$

Exercice 18 – Démonstration et arbres binaires complets

▶ **Question 1** Démontrer la Propriété 8.

On appelle *arbre binaire complet* un arbre binaire dont toutes les feuilles ont pour profondeur $h(A)$ ou $h(A) - 1$.

▶ **Question 2** Montrer que l'on peut borner plus finement le nombre de nœuds dans le cas où un arbre binaire est complet.

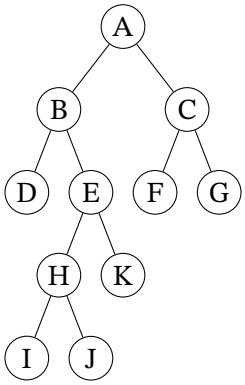
IV Parcours d'arbres

Bon, vous avez maintenant des données que vous avez représentées sous la forme d'un arbre. Dans la suite, on se limite au cas des arbres binaires : on verra en TP comment représenter un arbre général comme un arbre binaire avec les *tries*. La première chose que l'on veut faire, c'est *analyser votre arbre* en regardant dans un certain ordre ce qu'il y a dedans. Pour cela, on utilise des **parcours d'arbres** :

Définition 9 – Parcours d'arbre

Un parcours d'arbre est un algorithme qui parcoure (*ahaha*) dans un certain ordre les nœuds de l'arbre syntaxique.

Le parcours d'un arbre sert notamment à appliquer une certaine opération sur chaque nœud de l'arbre et les différents parcours vont effectuer les opérations dans un ordre différent. Les deux parcours à connaître **par coeur** sont les *parcours en profondeur* et *parcours en largeur* : ils ont en commun qu'on parcoure un arbre de gauche à droite et de haut en bas, mais la priorité entre ces deux ordonnancements diffère. On va l'appliquer sur l'arbre suivant :



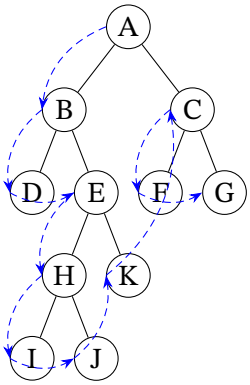
IV.A Parcours en profondeur

L'idée du parcours en profondeur est de descendre le plus profondément possible dans l'arbre, puis de remonter dans l'arbre quand on est arrivé en bas (c'est-à-dire dans une feuille). En anglais, on parle de *depth-first traversal* : un parcours en profondeur d'abord.

En vérité, il y a 3 parcours en profondeur possibles, en fonction de l'ordre des opérations :

— parcours **préfixe** : on traite la racine d'abord, puis on explore le fils gauche, puis le fils droit,
— parcours **infixe** : on explore d'abord le fils gauche, puis on traite la racine, puis on explore le fils droit,
— parcours **suffixe** ou **postfixe** : on explore d'abord le fils gauche, puis on explore le fils droit, puis on traite la racine.

Si l'on applique l'ordre préfixe sur l'arbre exemple, on a :



$[A; B; D; E; H; I; J; K; C; F; G]$

Le parcours infixe visite les nœuds dans l'ordre $[D; B; I; H; J; E; K; A; F; C; G]$, l'ordre suffixe dans l'ordre $[D; I; J; H; K; E; B; F; G; C; A]$.

Vu la description de la stratégie vue plus haut, cette approche est naturellement *récursive*. En considérant que le traitement d'un nœud applique une fonction `f : 'a -> unit` sur son étiquette, ces trois parcours s'écrivent comme suis :

OCaml

```
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree

let exemple =
  Node
    ( 25,
      Node (12, Node (5, Empty, Empty), Node (8, Empty, Empty)),
      Node
        ( 2,
          Node (52, Node (16, Empty, Empty), Node (27, Empty, Empty)),
          Node (10, Empty, Empty) ) )

let rec parcours_prefixe f = function
  | Empty -> ()
  | Node (x, g, d) ->
    f x;
    parcours_prefixe f g;
    parcours_prefixe f d

let rec parcours_infixe f = function
  | Empty -> ()
  | Node (x, g, d) ->
    parcours_infixe f g;
    f x;
    parcours_infixe f d

let rec parcours_suffixe f = function
  | Empty -> ()
  | Node (x, g, d) ->
    parcours_suffixe f g;
    parcours_suffixe f d;
    f x

(* on peut aussi écrire un parcours de manière itérative en utilisant
   la structure de pile : *)

let parcours_prefixe_imp f a =
  let pile = Stack.create () in
  Stack.push a pile;
  while not (Stack.is_empty pile) do
    match Stack.pop pile with
    | Empty -> ()
    | Node (x, g, d) ->
      f x;
      Stack.push d pile;
      Stack.push g pile
  done
```

Exercice 19 – Calculs de complexité

Déterminer la complexité temporelle et spatiale d'un parcours en profondeur récursif. On les exprimera en fonction de la hauteur $h(A)$ et de la taille $n(A)$ de l'arbre A en entrée. On considérera

que la fonction `f` s'exécute en $\mathcal{O}(1)$.

Exercice 20 – Parcours d'arbres généralisés

On reprend le type `'a tree` général où les nœuds peuvent avoir une arité quelconque.

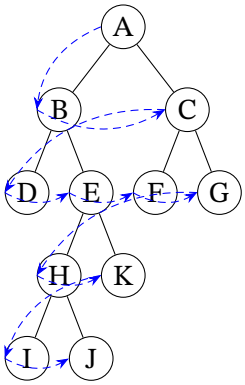
▶ **Question 1** Parmi ces trois parcours, lesquels ont un sens dans les arbres généralisés ?

▶ **Question 2** Les implémenter en OCaml.

IV.B Parcours en largeur

En anglais, on parle de *breadth-first traversal* : un parcours en largeur d'abord. Contrairement au parcours en profondeur, on ne priorise pas les nœuds les plus profonds, mais plutôt les nœuds les plus larges.

Sur le même exemple, cela donne :



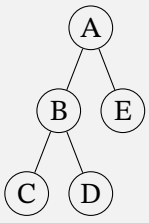
$[A; B; C; D; E; F; G; H; K; I; J]$

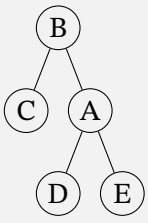
On l'appelle aussi *parcours militaire* : si notre arbre représente la chaîne de commandement d'une armée, le parcours en largeur visite d'abord le général en chef, puis les généraux, puis les colonels, puis les lieutenants, etc.

Remarque 10 – Sur l'unicité des parcours d'arbres

Supposons que l'on a utilisé un parcours pour énumérer et stocker dans une liste les étiquettes des nœuds. À partir de cette liste, est-il possible de retrouver l'arbre ?

La réponse est non, sauf dans certains cas. D'abord, il faut distinguer les nœuds internes des feuilles (les contre-exemples où deux arbres différents donnent le même parcours sont très faciles à trouver sans cette distinction). Ensuite, ça ne fonctionne pas dans l'ordre infixe, par exemple avec les deux arbres suivants :





Exercice 21 – On l'a dit, maintenant on le montre

▶ **Question 1** Écrire des fonctions de type `('a, 'a) strict_tree -> 'a list` qui à un arbre renvoie la liste de ses éléments dans l'ordre de visite des différents parcours vus jusqu'ici.

Il n'est pas possible d'écrire la fonction réciproque ici, comme le montre l'exemple précédent pour le parcours infixe. On va donc introduire un type pour différencier, dans notre liste, les étiquettes

venant d'un nœud interne et celles qui viennent d'un nœud externe.

OCaml
```
type 'a t = | De_feuille of 'a
            | De_noeud_interne of 'a
```

▶ **Question 2** Pour les parcours préfixes et suffixes, écrire la fonction réciproque `('a t) list -> ('a, 'a) strict_tree`.

▶ **Question 3** (Difficile) Réécrire les fonctions de la question 1 pour qu'elles s'exécutent en $\mathcal{O}(n)$.

V Une première application des arbres

Les arbres binaires de recherche sont des arbres binaires particuliers utilisés pour implémenter les structures de données abstraites *ensemble* et *dictionnaire*, que l'on a déjà défini dans le Chapitre 5. On se concentrera d'abord sur les *ensembles*.

Définition 11 – Arbre binaire de recherche

Soit $(X, \leq)$ un ensemble (totalement) ordonné [a]. Les arbres binaires de recherche sur X sont définis pas induction à l'intérieur des arbres binaires étiquetés par X tel que :
- L'arbre vide est un arbre binaire de recherche,
- Un nœud dont la racine est étiquetée par x est un arbre binaire de recherche si et seulement si
  - Son fils gauche et son fils droit sont des arbres binaires de recherche,
  - Pour tout nœud du fils gauche, son étiquette est strictement inférieure à x et pour tout nœud du fils droit, son étiquette est strictement supérieure à x.

a. c'est-à-dire un ensemble muni d'une relation d'ordre tel que deux éléments sont toujours comparables : pour tout $x_1, x_2 \in X$, on a $x_1 \leq x_2$ ou $x_2 \leq x_1$.

Comme d'habitude, on ne représente pas les arbres vides, mais on laisse l'espace concerné vide.

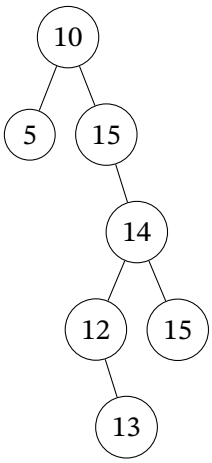
Propriété 12 – Les étiquettes d'un ABR sont distinctes

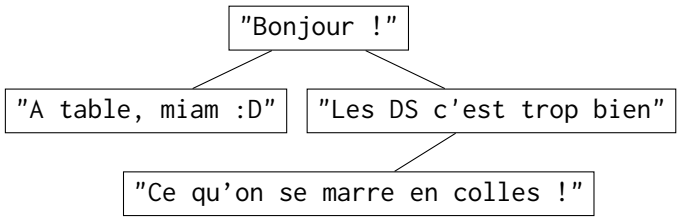
Les étiquettes d'un ABR sont distinctes.

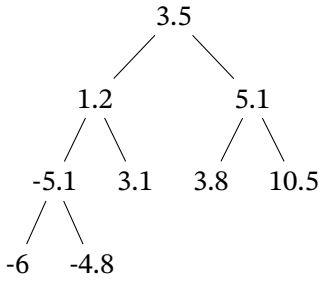
On va donc utiliser les arbres binaires de recherche sur X pour représenter les sous-ensembles de X : on représente un ensemble $Y \subseteq X$ par un arbre binaire de recherche dont l'ensemble des étiquettes est égal à Y.

Exercice 22 – S'il vous plaît, dessine moi un ABR

▶ **Question 1** Déterminer les arbres binaires de recherche parmi les arbres suivants et écrire les ensembles qu'ils représentent.







▶ **Question 2** Proposer des arbres binaires de recherche représentant les ensembles suivants :

$$X_1 = \{-5.5; -100.81; 25.18; 12.3; 50.1; 42.48\}$$
$$X_2 = \{\texttt{'a'}; \texttt{'x'}; \texttt{'z'}; \texttt{'b'}; \texttt{'p'}; \texttt{'f'}\}$$
$$X_3 = \{-10; -9; -8; \dots; 9; 10\}$$

Un arbre binaire de recherche est simplement un arbre binaire (non strict) qui respecte une propriété sur ses étiquettes. Pour le représenter en OCaml, on utilisera donc un type similaire aux arbres binaires (non strict).

OCaml
```
type 'a abr =
  | Vide
  | Noeud of 'a abr * 'a * 'a abr
  (* on met l'étiquette au milieu pour mettre en évidence l'ordre sur les
   ↪ étiquettes *)
```

Exercice 23

Écrire les arbres binaires de recherche de l'Exercice 22 avec le type `abr`.

Exercice 24 – Implémenter les ABR en C

▶ **Question 1** Proposer un type C permettant de représenter les arbres binaires de recherche. Créer une fonction qui, pour deux pointeurs vers des ABR `a` et `b` et une nouvelle clé `x`, crée un nouvel ABR `N(x, a, b)` (on considérera que l'on sait que $\max_{k\in a} k < x < \min_{k\in b}$).

▶ **Question 2** Écrire une fonction qui crée une feuille de cet arbre binaire de recherche, prenant en argument une étiquette `int x`.

▶ **Question 3** Écrire une fonction de prototype `void libere_arbre(struct ABR* a)` qui libère

toute la mémoire utilisée par un arbre binaire de recherche a.

On peut implémenter le type abstrait ensemble en utilisant les propriétés des arbres binaires de recherche : on va le faire en TP.

V.A Arbres binaires de recherche équilibrés — HP option MPSI / MP

La stratégie proposée en TP pour l'insertion et les deux stratégies pour la suppression sont quand même *naïves*. L'idéal serait que ces deux opérations garantissent que l'arbre obtenu après une insertion / suppression soit équilibré :

Définition 13 – Arbres équilibrés

Un arbre est dit *équilibré* quand sa hauteur est logarithmique en sa taille. On dit qu'il est *parfaitement équilibré* toutes les profondeurs des nœuds vides sont égales.

Eh bien ça existe ! On appelle ce genre d'arbres binaires de recherche des arbres autoéquilibrés. On va voir l'exemple des arbres 2–3, qui a l'immense avantage d'être simple à comprendre et des arbres rouge-noir, qui a l'immense avantage d'être au programme de la MP2I.

V.A.1 Arbres 2–3

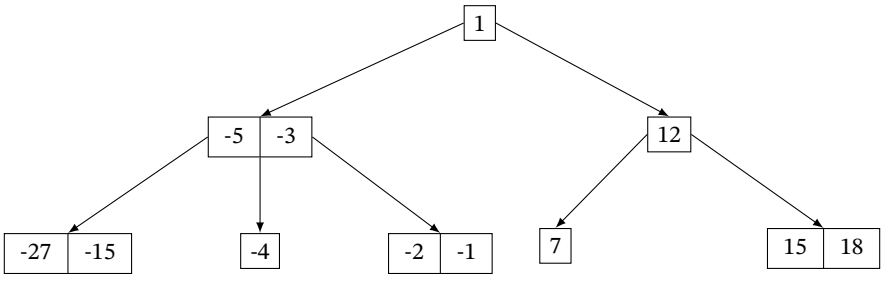
Définition 14 – Arbres 2–3

Un arbre 2–3 et un arbre parfaitement équilibré dont les nœuds sont :
- soit des nœuds binaires $N(g, e, d)$ vérifiant la condition des nœuds des ABR ($\max g \leq e \leq \min d$ et g et d sont des arbres 2–3).
- soit des nœuds ternaires (à trois fils) possédant *deux* étiquettes au lieu d'une seule, de la forme $M(g, e_1, m, e_2, d)$ ayant une propriété similaire : $\max g \leq e_1 \leq \min m \leq \max m \leq e_2 \leq \min d$ et g, m, d sont des arbres 2–3.

On peut l'implémenter en OCaml avec le type concret suivant :

OCaml
```
type 'a a23 =
  | V
  | N of 'a a23 * 'a * 'a a23
  | M of 'a a23 * 'a * 'a a23 * 'a * 'a a23
```

Par exemple, l'arbre suivant est un arbre 2–3 :



Propriété 15 – Un arbre 2–3 est équilibré

La hauteur d'un arbre 2–3 à n nœuds est bornée par $\lfloor \log_2 n \rfloor$.

Démonstration. D'abord, grâce à la définition des arbres 2–3, tous les nœuds vides sont à la profondeur $h(A)+1$. Par une induction immédiate, il y a au moins 2^k nœuds à la profondeur $k \leq h(A)$. On a donc au

moins $2^{h(A)}$ nœuds à la profondeur $h(A)$: notamment, on a $2^{h(A)} \leq n(A)$, c'est-à-dire $h(A) \leq \lfloor \log_2 n \rfloor$. De la même façon, on pourrait montrer que $\lceil \log_3 n \rceil \leq h(A)$. □

La recherche dans un arbre 2–3 suit le même principe que dans les ABR classiques :

OCaml
```
let rec recherche a x = match a with
  | V -> false
  | N (g, e, d) ->
    x = e || (
      if x < e then recherche g x else recherche d x
    )
  | M (g, e1, m, e2, d) ->
    e1 = x || e2 = x || (
      if x < e1 then recherche g x
      else if x < e2 then recherche m x
      else recherche d x
    )
```

En supposant que la complexité de la comparaison entre deux étiquettes se fait en temps constant :

Propriété 16

La fonction `recherche` est correcte et de complexité logarithmique en le nombre d'étiquettes.

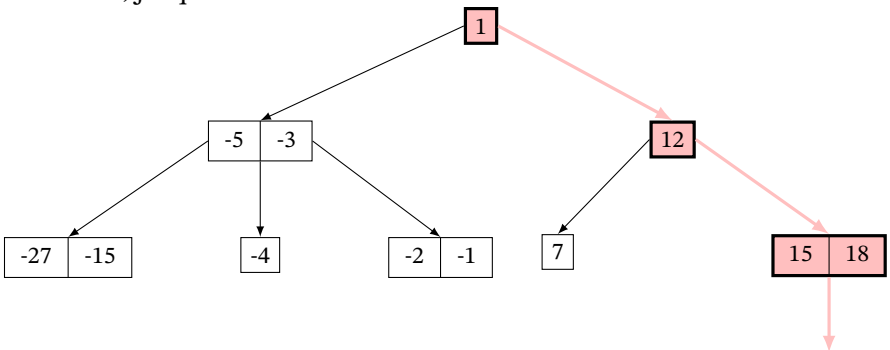
Démonstration. On l'a faite en TP pour les ABR. Ici, c'est la même chose. Pour la complexité, on remarque que l'on fait un nombre au plus constant d'opérations élémentaires pour chaque nœud visité et l'on en visite au plus $h(A) + 1$ (en comptant le vide). □

Parfait ! On a donc inventé une structure de donnée où la recherche est logarithmique de manière garantie (et pas en moyenne comme pour les ABR classiques). Le cours est fini, on peut aller en maths…

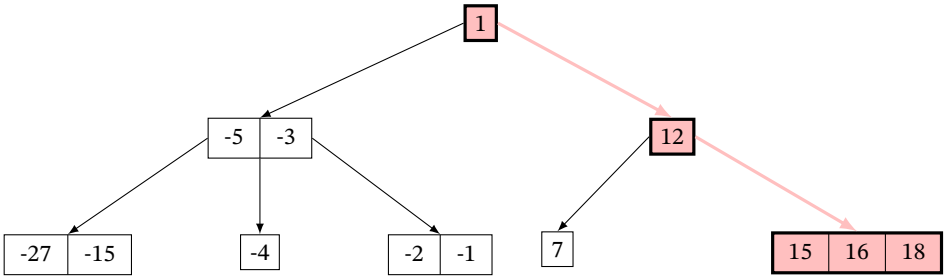
Mais non : il faut encore programmer les fonctions d'insertion et de suppression dans un arbre 2–3. Et quand on y réfléchit, ce n'est pas absolument évident de le faire.

Insertion dans un arbre 2–3 Pour la fonction d'insertion, la stratégie sera la suivante (où l'on insère 16) :

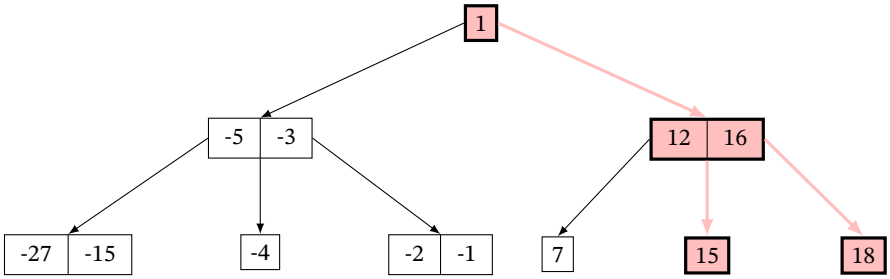
— D'abord, on fait comme pour les ABR classique : on parcourt notre arbre par le même chemin que si l'on faisait la recherche, jusqu'à arriver à un nœud vide.



— Si le dernier nœud était binaire, il suffirait de le transformer en un nœud ternaire, mettre l'ancienne étiquette et celle insérée dans le bon ordre et ça serait terminé. Ici, ce n'est pas le cas : il va donc falloir ruser. D'abord, on va le transformer (dans nos têtes) en un nœud quaternaire (à trois étiquettes) :



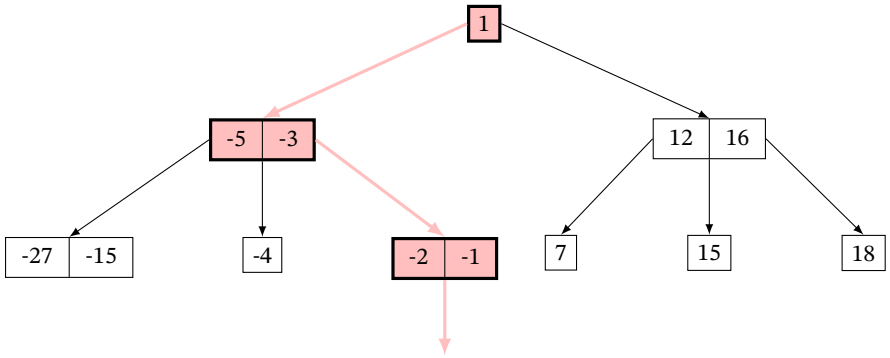
— Ensuite, on se débarrasse du problème en remontant un nœud au-dessus et en y ajoutant un fils au milieu :



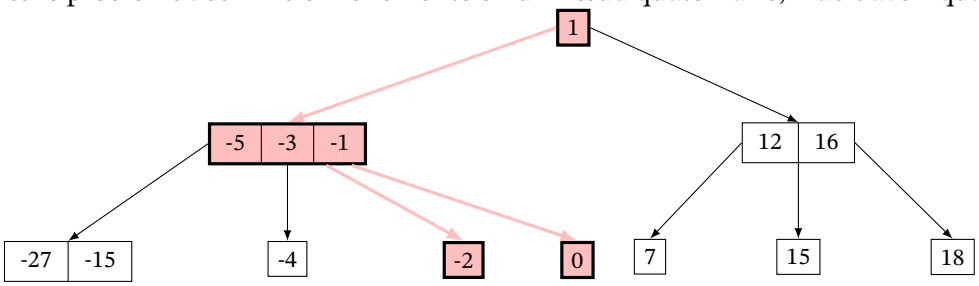
— Et on a fini ! On a bien obtenu un arbre 2–3 correct.

Maintenant, insérons 0 :

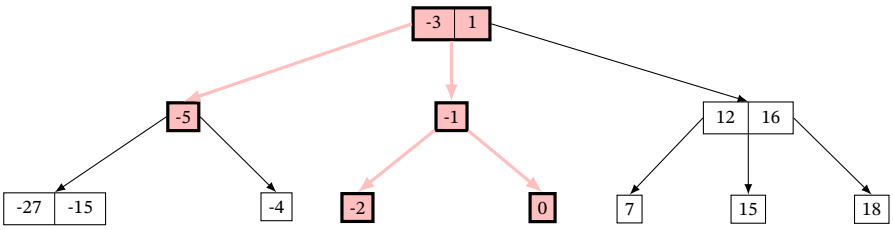
— La première étape est la même : repérer où on l'aurait inséré précédemment, et y créer un nœud quaternaire dans nos têtes.



— On remonte le problème : comme on le remonte en un nœud quaternaire, il doit avoir quatre fils.

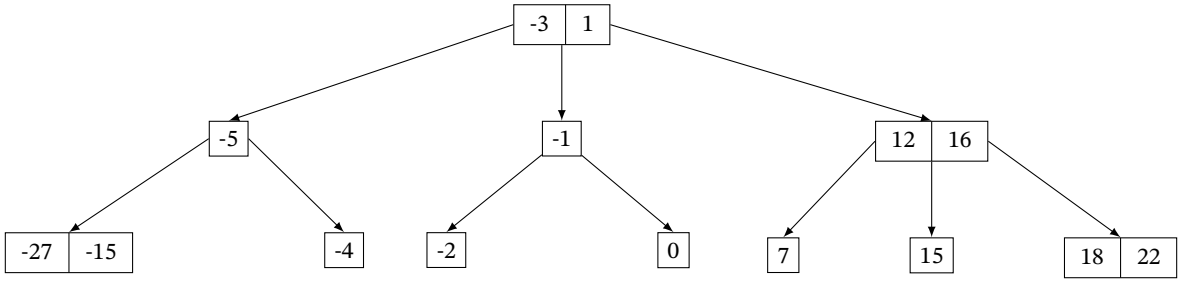


— Ensuite, il faut encore remonter le problème : on va profiter du fait que le nœud de dessus est binaire pour le transformer en ternaire et s'arrêter là.



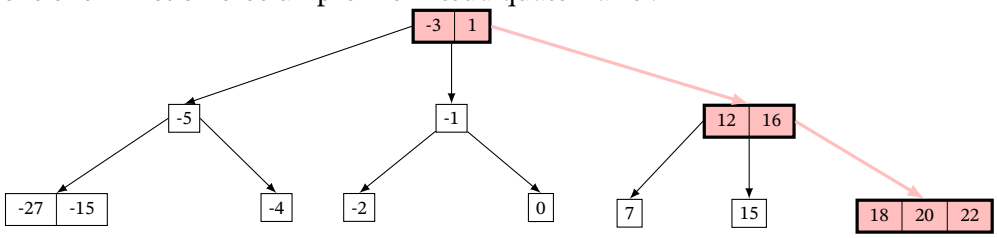
— Et voilà !

Insérons rapidement une nouvelle valeur 22 :

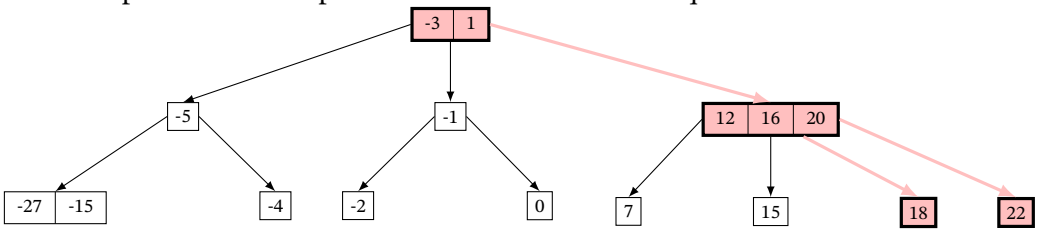


On va maintenant pouvoir considérer le dernier cas, quand tout le chemin parcouru ne passe que par des nœuds ternaires : insérons 20.

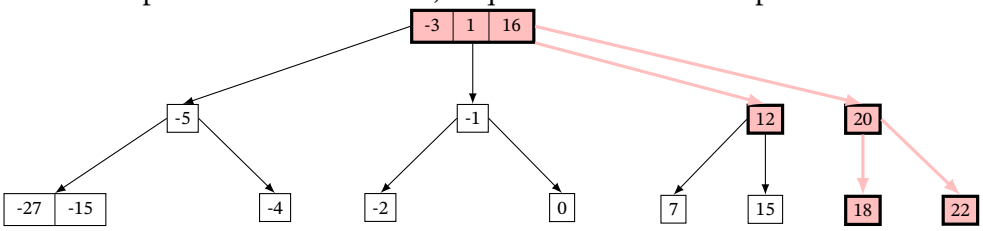
— On repère le chemin et on crée un premier nœud quaternaire :



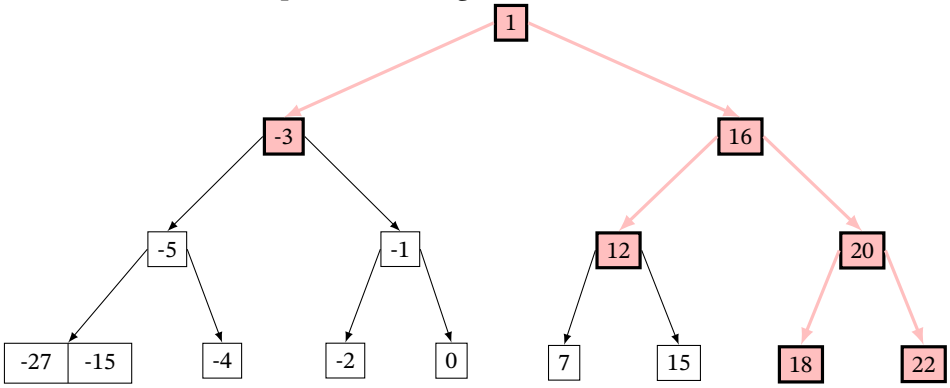
— On remonte une première fois le problème en créant un nœud quaternaire :



— Et on règle encore le problème en remontant, ce qui remonte encore le problème :

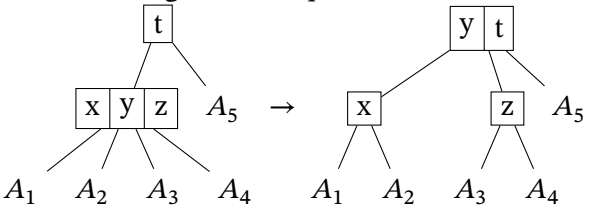


— Ce cas sera le seul augmentant la hauteur de l'arbre : on crée une nouvelle racine. On remarque que *toutes les feuilles vides* ont vu leur profondeur augmenter, donc tout va bien !

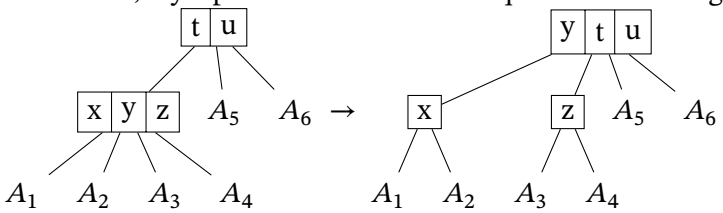


L'algorithme aura donc comme stratégie :

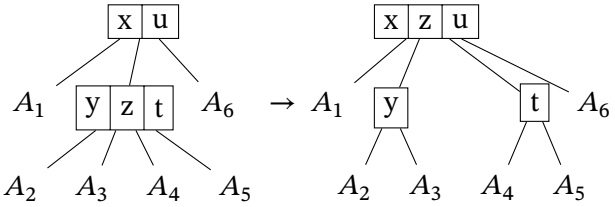
— Si le nœud courant est binaire et le fils gauche est quaternaire :



— La même chose quand le fils droit est quaternaire.
— Si le nœud courant est ternaire, il y a plus de cas. Si le nœud quaternaire est à gauche :



— Idem s'il est à droite.
— S'il est au centre :



Exercice 25

▶ **Question 1** Compléter les transformations quand le fils quaternaire est à droite.

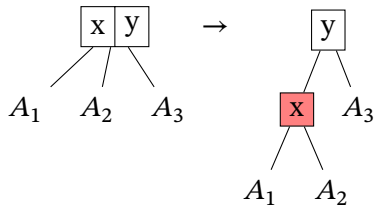
▶ **Question 2** Justifier que les arbres produits par ces transformations sont toujours des arbres 2–3.

▶ **Question 3** L'implémenter (en OCaml, je ne suis pas un monstre).

▶ **Question 4** Réfléchir à la suppression dans les arbres 2–3.

V.A.2 Arbres rouge-noir

Le principe des arbres rouge-noir est de *binariser* les arbres 2–3, qui permet de simplifier (un peu) leur implémentation. L'idée est de séparer les nœuds ternaires en deux :



On marque le nœud contenant l'ancienne étiquette à gauche en rouge, tous les autres nœuds restent colorés en noir. Ainsi, dans cette section, on manipule des *arbres colorés* où chaque nœud possède une étiquette et une couleur rouge ou noire. Si l'on traduit les conditions des arbres 2–3, cela donne :

Définition 17 – Arbre rouge-noir

Un arbre *rouge-noir* (ou *arbre bicolore*) est un arbre binaire de recherche coloré vérifiant les propriétés suivantes :
— Les enfants d'un nœud rouge sont noirs.
— La racine de l'arbre et les nœuds vides sont noirs.
— Tous les chemins reliant la racine à une feuille contiennent le même nombre de nœuds noirs.

La première condition est une condition *locale* fixée sur les arbres bicolores, alors que les deux suivantes forment une condition *globale*.

Remarque 18 – *Left-leaning red-black tree*

En fait, quand on traduit des arbres 2–3 en des arbres rouge-noir, on doit ajouter une condition locale disant que *le fils droit d'un nœud doit être noir*. Pour être plus précis, sans cette condition, on vient de « binariser » des arbres 2–3–4 qui sont une variante des arbres 2–3 dans lesquels les nœuds quaternaires sont autorisés.

On introduit pour les arbres rouges-noirs la notion de hauteur noire :

Définition 19 – Hauteur noire

La hauteur noire d'un arbre est le nombre de nœuds noirs dans un chemin de la racine à un nœud vide (sans compter le nœud vide). *Cette définition n'a de sens que pour les arbres rouges-noirs, pour lequel ce nombre est le même pour tous les chemins de la racine à un nœud vide.*

Remarque 20 – Définition par induction de la hauteur noire

On peut définir la hauteur noire de A par induction sur les arbres colorés :
- Pour $\emptyset$, on a $h_n(\emptyset) = 0$.
- Pour $A = N(c, g, e, d)$, on a $h_n(A) = \max(h_n(g), h_n(d)) + 1$ si la couleur c est noire et $h_n(A) = \max(h_n(g), h_n(d))$ sinon.

Dans un arbre rouge-noir, on aura toujours $h_n(g) = h_n(d)$.

Remarque 21

La condition globale des arbres rouges-noirs peut se réécrire comme suis : pour tout nœud $\mathfrak{n}$ de A, le nombre de nœuds noirs sur un chemin de $\mathfrak{n}$ à l'un de ses descendants vides est le même pour tout chemin. Ainsi, la notion de hauteur noire a aussi un sens dans les sous-arbres d'un arbre rouge-noir.

On les implémentera par les types concrets suivants :

OCaml
```
type couleur = R | B

type 'a arn =
  | V
  | N of couleur * 'a arn * 'a * 'a
   ↪  arn
```

C
```
typedef struct ARN {
  struct ARN *gauche;
  struct ARN *droit;
  int cle;
  bool rouge;
} noeud;
```

Exercice 26

Écrire une fonction en C et en OCaml vérifiant qu'un arbre vérifie les conditions d'un arbre bicolore.

La recherche dans un arbre bicolore se fait de la même manière que dans un ABR, simplement en ignorant la couleur des nœuds. Par exemple, en OCaml :

OCaml
```
let rec recherche cle = function
  | V -> false
  | N (_, g, e, d) ->
    if cle = e then true
    else if cle < e then recherche cle g
    else recherche cle d
```

Puisque les arbres rouge-noir sont une traduction des arbres 2–3, on a la bonne propriété suivante :

Propriété 22 – Recherche dans un arbre bicolore

La fonction `recherche` a une complexité logarithmique en la taille de l'arbre *en supposant que la comparaison entre clés se fait en temps constant*.

Pour le montrer proprement (en classe, je n'en ai parlé qu'à l'oral avec une justification correct, mais peu développée), on peut montrer le lemme suivant :

Lemme 23 – Les arbres rouges-noirs sont équilibrés

Les arbres rouge-noir vérifient l'inégalité suivante :

$$2^{h_n(A)} - 1 \leq n(A)$$

Démonstration. On le montre par induction sur les arbres rouges-noirs [a].

Initialisation : Pour un arbre vide, on a bien $h_n(A) = 0$ donc $2^{h_n(A)} - 1 = n(A) = 0$.

Induction : Supposons que l'arbre A est un sous-arbre d'un arbre rouge-noir non vide, et notons g son fils gauche et d son fils droit. Par la Remarque 21, on a donc deux choix :

— Si A a une racine rouge, $h_n(g) = h_n(d) = h_n(A)$, et l'on a par hypothèse d'induction :

$$\begin{aligned} n(A) &= n(g) + n(d) + 1 \\ &\geq 2^{h_n(g)} - 1 + 2^{h_n(d)} - 1 + 1 \\ &= 2^{h_n(a)+1} - 1 \\ &\geq 2^{h_n(a)} - 1. \end{aligned}$$

— Si A a une racine noire, $h_n(g) = h_n(d) = h_n(A) - 1$ et par hypothèse d'induction :

—

$$\begin{aligned} n(A) &= n(g) + n(d) + 1 \\ &\geq 2^{h_n(g)} - 1 + 2^{h_n(d)} - 1 + 1 \\ &= 2^{h_n(a)} - 1 \end{aligned}$$

□

a. Plus précisément, on le montre sur l'ensemble des arbres colorés vérifiant la seconde propriété globale portant sur les hauteurs des fils des nœuds.

Lemme 24

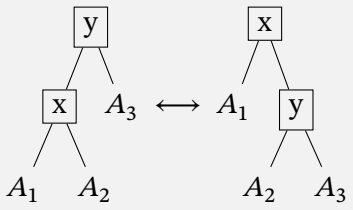
Pour A un arbre rouge-noir, on a $h(A) \leq 2h_n(A)$.

Démonstration. Il suffit de remarque que la hauteur est le nombre de nœuds noirs sur un chemin de la racine à une feuille commence par un nœud noir et que pour chaque nœud rouge, son père est noir. Ainsi, sur tout chemin de la racine (noire) à un arbre vide, pour chaque nœud rouge on passe par un nœud noir, et on a donc au moins autant de nœuds noirs que de nœuds rouges. □

L'insertion suit aussi le même principe que celle dans les arbres 2–3. Dans les arbres bicolores, cela se traduira par des opérations plus "graphiques", comme des rotations d'arbres.

Définition 25 – Rotation d'arbre binaire de recherche

Une rotation dans un arbre binaire de recherche est l'un des deux transformations suivantes (de droite à gauche ou de gauche à droite). Notons qu'elles préservent la propriété d'arbre binaire de recherche.



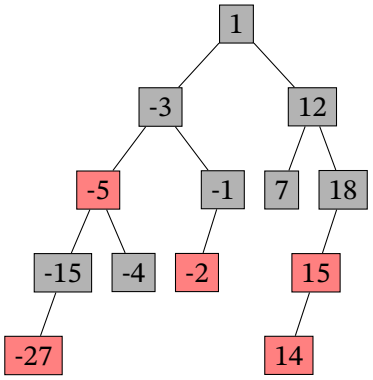
Dans un arbre rouge-noir, une telle rotation échange aussi les couleurs des nœuds x et y. Ainsi, elles préserveront les conditions sur les couleurs des arbres bicolores (mais pas la condition sur le nombre de nœuds noirs de la racine aux nœuds vides !).

Exercice 27 – Rotations en C et en OCaml

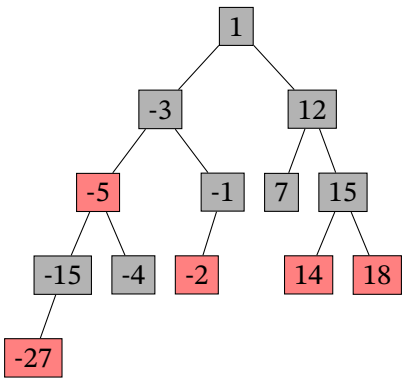
Écrire ces deux opérations (rotation de gauche à droite et rotation de droite à gauche) en C et en OCaml. En C, on modifiera l'arbre, alors qu'en OCaml on en renverra un nouveau.

Ces deux opérations vont permettre d'exprimer plus simplement l'opération d'insertion dans un arbre rouge-noir. La stratégie d'insertion dans les arbres 2–3 se traduit naturellement dans les arbres bicolores : on insère d'abord un nœud rouge en risquant de violer la condition locale, puis on remonte le problème dans l'arbre jusqu'à ne plus la violer (tout en gardant toujours vraie la condition globale).

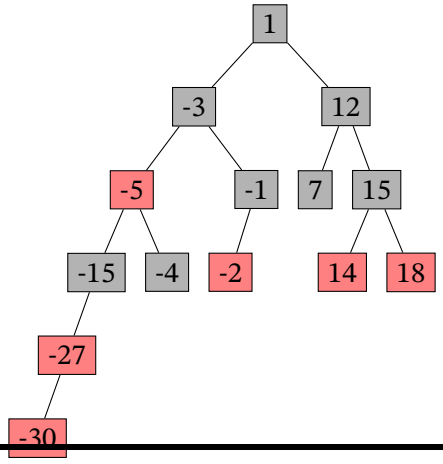
D'abord, on explore notre ABR jusqu'à trouver un endroit où insérer notre nœud. On va l'insérer dans un nœud *rouge*. Reprenons l'exemple des arbres 2–3, et insérons-y 14 :



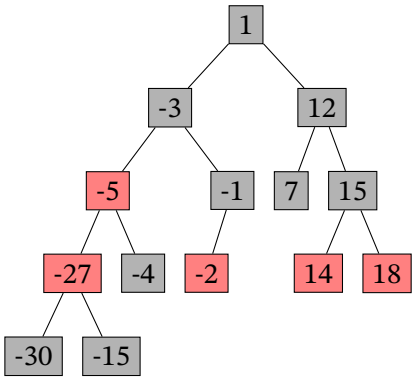
Ensuite, on remarque qu'il y a un problème de définition ici : l'arbre ne respecte plus la condition locale des arbres bicolores. Pour le respecter, on va « rotationner » l'arbre enraciné en 18 :



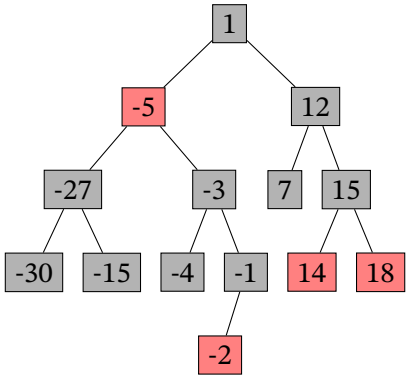
Et c'est gagné ! Insérons maintenant -30 dans cet arbre :



Ici aussi, notre insertion a violé la condition locale. On va devoir faire remonter le problème dans l'arbre. On commence par rotationner l'arbre enraciné en -15 :

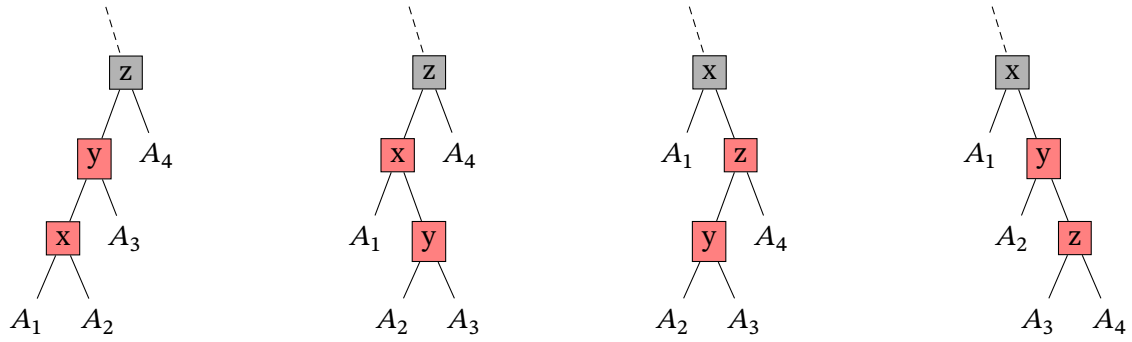


Ici, on a bien fait remonter la violation de la condition locale d'un étage. On va maintenant rotationner l'arbre enraciné en -3 :

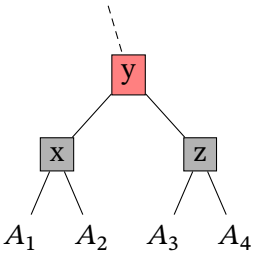


Notez ici que pour chaque insertion, on a fait un choix faisant toujours remonter la violation de la condition locale d'un étage, sans considérer d'autres choix possibles. En effet, on aurait pu tenter d'autres stratégies pour éviter de remonter le problème selon le contenu des sous-arbres « autour » des nœuds rotationnés : pendant l'insertion de -30, on aurait pu par exemple inverser les couleurs de -30, -27 et -15 et on aurait terminé.

Quand on insère un nouveau nœud rouge, on va devoir considérer plusieurs cas pour faire remonter la violation de cette condition locale tout en conservant la condition globale :



Qui se transforme naturellement en :



On a bien fait "remonter" la violation potentielle de la condition locale d'un étage : si le parent de y est noir, on a terminé le travail et les conditions locales et globales sont respectées. Si le parent de y est rouge, on continue de remonter. Enfin, si y est la racine de notre arbre, il suffit de le colorer en noir plutôt qu'en rouge et on obtient bien un arbre rouge-noir correct.

Exercice 28 – Insertion

Implémenter la fonction d'insertion en OCaml. *Pas besoin des fonctions de rotation de l'Exercice 27, c'était juste pour l'exemple.*

Exercice 29 – Implémentons le en C !

En C, l'implémentation serait similaire. Compléter le code suivant, pour que la fonction `nouveau_noeud` crée une nouvelle feuille rouge contenant la valeur `cle` et `insere` qui effectue une insertion (sans faire la dernière étape éventuelle sur la racine) :

C
```c
noeud* nouveau_noeud(int cle) {
  // À compléter
}

noeud* insere_aux(noeud* n, type_donnees k){
  if (n == NULL) { return nouveau_noeud(k); }
  if (k == n->cle) { return n; }
  if (k < n->cle) n->gauche = insere_aux(n->gauche, k);
  if (k > n->cle) n->droit = insere_aux(n->droit, k);

  // À compléter

  return n;
}
```

On pourra utiliser les fonctions de rotation ici.

Exercice 30 – Suppression

Réfléchir à la suppression dans un arbre bicolore. *L'idée est similaire à l'insertion : on a cependant un peu plus de cas à considérer. On pourra découper le code en la suppression du minimum d'un arbre bicolore. On se permettra les mêmes choses que pour l'insertion : on retournera des arbres bicolores, sauf pour la racine qui pourra être rouge et avoir des fils rouges, jusqu'à avoir complètement codé la fonction de suppression.*

Exercice 31 – Dictionnaire et ensemble par arbres bicolores

Implémenter les dictionnaires et ensembles par les arbres binaires de recherche, en adaptant si besoin les fonctions déjà implémentées.

Parmi les implémentations des types abstraits fonctionnels d'ensembles et de dictionnaires, les arbres rouges-noirs sont très souvent utilisés. Une autre variante existe : les arbres AVL, qui sont des arbres binaires de recherche où chaque nœud vérifie que la hauteur de ses deux fils diffère d'au plus 1. Leurs propriétés sont les mêmes que les arbres bicolores : la recherche, insertion et suppression se font en temps logarithmique. En fait, les transformations nécessaires sur les arbres sont exactement celles que l'on a décrit : des rotations dans tous les sens pour préserver l'équilibre des arbres !

Exercice 32

On peut avancer encore un peu dans les propriétés des arbres, en introduisant les *nombres de Catalan*. Ils apparaissent dans divers problèmes de combinatoire et sont définis par récurrence

comme suis :

$$\begin{cases} c_0 = 1 \\ c_{n+1} = \sum_{i=0}^{n} c_i c_{n-i} \end{cases}$$

▶ **Question 1** Montrer qu'il existe exactement c_n arbres binaires stricts avec n nœuds internes. *On ne considère pas des arbres étiquetés, sinon évidemment qu'il y en a une infinité.*

L'objectif est maintenant de montrer que pour tout $n \in \mathbb{N}$, $(n+2)c_{n+1} = 2(2n+1)c_n$. Pour cela, on va montrer qu'il existe une bijection entre :

— le nombre de couples (A, f) avec A un arbre à $n+1$ nœuds internes, et f une feuille de A,
— le nombre de triplets (B, x, e) avec B un arbre à n nœuds internes, x un nœud de B (interne ou feuille) et e une direction (gauche ou droite).

▶ **Question 2** Déterminer ces deux nombres en fonction de la suite de Catalan.

▶ **Question 3** Justifier que la fonction suivante est injective : à un couple (A, f) du premier ensemble, on associe un triplet (B, x, e) du second tel que :

— B est l'arbre a dans lequel on retire la feuille f et son père (en remontant le frère de f).
— x est le nœud remonté,
— e est la direction de la feuille supprimée

▶ **Question 4** Montrer qu'elle est bijective en déterminant sa réciproque.

▶ **Question 5** En déduire que $c_n = \frac{1}{n+1}\binom{2n}{n}$.

VI Structures de tas

Les tas permettent d'implémenter le type abstrait suivant :

Définition 26 – File de priorité

Une file de priorité est un type abstrait (séquentiel, à ranger à côté des types abstraits du Chapitre 5) dont la signature est la suivante (pour sa version impérative) :

Op	Spécification	Type de l'opération
C	Crée une file de priorité vide	`unit -> ('a,'p) file_prio`
A	Vérifie que la file en entrée est vide	`('a,'p) file_prio -> bool`
T	Ajoute un élément x de priorité p	`('a,'p) file_prio -> 'a -> 'p -> unit`
T	Retire et renvoie l'élément de plus petite priorité dans la file.	`('a,'p) file_prio -> 'a * 'p`
T	Diminue la priorité d'un élément de la file.	`('a,'p) file_prio -> 'a -> 'p -> unit`

Exercice 33

▶ **Question 1** Proposer un type abstrait fonctionnel de file de priorité en adaptant les trois transformateurs en des constructeurs.

▶ **Question 2** Implémenter ce type abstrait fonctionnel de file de priorité par les listes associatives `('a * 'p) list`.

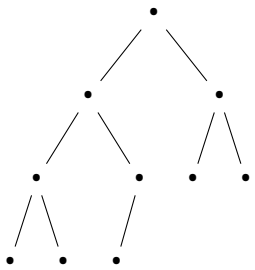
▶ **Question 3** Déterminer la complexité de chacune des opérations.

On pourrait aussi implémenter les files de priorité par des arbres binaires de recherche : en les supposant équilibrés, on obtient une complexité des trois opérations logarithmiques (au lieu de linéaire).

Définition 27

Un *arbre binaire complet* est un arbre binaire dont tous les niveaux sont "remplis", sauf éventuellement le dernier niveau qui est rempli à gauche. Plus formellement, un arbre binaire complet de hauteur h est un arbre binaire tel que :
- Pour tout $0 \leq p < h$, il y a exactement 2^p nœuds à la profondeur p de l'arbre.
- Pour le niveau h, les nœuds sont remplis de gauche à droite.

Dans un arbre binaire complet, sa *forme* est complètement déterminée par son nombre de nœuds. Par exemple, si on vous donne 10 nœuds, le seul arbre binaire complet à 10 nœuds est :



On remarque que dans un arbre binaire complet, toutes les feuilles sont à la profondeur h ou $h-1$. De plus, on a :

Propriété 28

Pour un arbre binaire complet de hauteur h à n nœuds, on a :
- $h = \lfloor \log_2 n \rfloor$
- $2^h \leq n \leq 2^{h+1}$

Notamment, on peut dire que la classe des arbres binaires complets est une classe d'arbres binaires équilibrés (le nombre de nœuds est logarithmique en la hauteur).

Exercice 34

Le démontrer.

Définition 29

Un tas-min (en anglais, *binary heap*) est un arbre binaire complet dont les nœuds (internes ou feuilles) sont étiquetées dans un ensemble totalement ordonné $(E, \leq)$ tel que l'étiquette d'un nœud est toujours inférieure ou égale à celle de ses enfants.
Cette propriété sur les nœuds d'un tas est appelée « propriété de tas-min », qui se vérifie également en montrant par induction que chaque nœud contient une étiquette plus grande que celle de ses enfants, et que chacun de ses enfants vérifie la propriété de tas-min.

Exemple 30

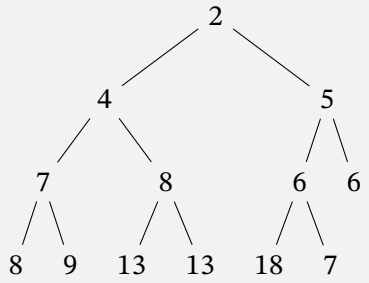


FIGURE 3.2 – Un exemple de tas-min

Remarque 31

On pourrait de même définir la structure de tas-max, dans laquelle l'étiquette d'un nœud est supérieure ou égale à celles de ses enfants.

Exercice 35

En utilisant le type suivant :

```OCaml
type 'a arbre = V | N of 'a * 'a arbre * 'a arbre
```

▶ **Question 1** Écrire une fonction testant la propriété de tas min sur un arbre (on ne teste donc pas qu'il est complet).

★ **Question 2** Écrire une fonction testant si un arbre est un tas min.

Classiquement, un tas n'est pas stocké comme un arbre entier en mémoire : en fait, il suffit d'un tableau de taille n pour stocker un tas à n éléments, comme suis :

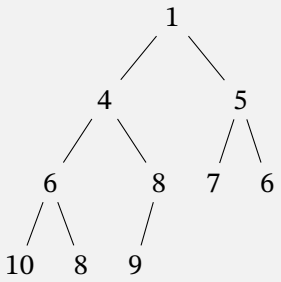
Théorème 32

Soit A un arbre binaire complet à n nœuds étiquetés $x_0, \dots, x_{n-1}$ dans l'ordre du parcours en largeur. Alors la fonction $f : A \mapsto \langle x_0, \dots, x_{n-1} \rangle$ est injective et on a :

1. La racine est à l'indice 0,
2. les nœuds de profondeur i sont numérotés consécutivement (dans l'ordre du parcours en largeur) à partir de $2^i - 1$,
3. le fils gauche du nœud à l'indice k est à l'indice $2k + 1$, son fils droit à l'indice $2k + 2$,
4. le père d'un nœud à l'indice k est à l'indice $\left\lfloor \frac{k-1}{2} \right\rfloor$.

Exemple 33 – Arbre binaire complet comme un tableau

L'arbre binaire complet suivant est représenté par `[| 1; 4; 5; 6; 8; 7; 6; 10; 8; 9 |]` :



Cet arbre binaire complet est bien un tas min.

Démonstration. Supposons que A est un arbre binaire complet à n nœuds et de hauteur h. Montrons par récurrence sur la profondeur des nœuds la seconde propriété :

Initialisation : La racine est bien d'indice 0, le premier nœud visité dans le parcours en largeur (ce qui montre aussi 1.).

Hérédité : Supposons que ce soit vrai pour une profondeur $i < h$. Comme $i < h$, le niveau est plein et contient donc 2^i nœuds. D'après l'hypothèse de récurrence, ils sont numérotés de $2^i - 1$ jusqu'à $2^i - 1 + 2^i - 1 = 2^{i+1} - 2$ et ceux à profondeur $i + 1$ sont donc numérotés à partir de $2^{i+1} - 1$.

On en déduit les deux dernières propriétés : pour un nœud à l'indice $k = 2^i - 1 + k'$ avec $0 \leq k' < 2^i$. Le nœud se trouve donc à la profondeur i et à cette profondeur se trouve k' nœud à sa gauche. Si le nœud à l'indice k admet un fils gauche, alors puisque l'arbre est complet les k' nœuds aux indices $2^i - 1$ à $k - 1$ ont deux fils, il y a $2k'$ nœuds avant le fils gauche de k à la profondeur $i + 1$. Ainsi, l'indice du fils gauche est $2^{i+1} - 1 + 2k' = 2 \times (2^i - 1 + k') + 1 = 2k + 1$. S'il a un fils droit, il est à l'indice suivant $2k + 2$. La quatrième propriété en est une conséquence immédiate. □

Cette représentation a des avantages par rapport à la représentation habituelle des arbres :

— Elle est bien plus compacte en mémoire : une case de tableau contient une étiquette, alors qu'un nœud d'un arbre a besoin de trois cases mémoires (étiquette + 2 pointeurs vers d'éventuels fils), en OCaml il y a aussi une quatrième case identifiant le constructeur.
— Elle impose de représenter des arbres binaires complets. Pas besoin de le vérifier !
— Elle permet plus facilement qu'avec les types habituels des arbres de passer d'un nœud à son père.

Les limites sont, elles aussi, évidentes :

— Ça se généralise mal aux arbres non complets, voire aux arbres non binaires.
— Il est très coûteux en termes de complexité de faire les opérations habituellement immédiates sur les arbres (échanger deux fils d'un arbre, faire une rotation dans un ABR, etc.).

En l'occurrence, les opérations nécessaires pour implémenter un tas min s'adaptent très bien à cette représentation en tableaux. On utilisera, selon les cas, l'un des deux types suivants :

OCaml
```
type 'a tas = {mutable size : int; data : 'a array}

type 'a tas_redimensionnable = {mutable size : int; mutable data : 'a array}
```

Le premier est utilisé quand connaît un majorant du nombre d'éléments qui vont être stockés dans notre tas. La partie du tableau `data` représentant un arbre va alors de `data.(0)` à `data.(size) - 1`, le reste ayant des valeurs quelconques. Le second permet, si le nombre d'éléments à mettre dans le tas dépasse `Array.length data`, on peut modifier ce tableau pour en créer un nouveau plus grand et continuer à y ajouter des éléments (à la manière des tableaux redimensionnables, les complexités *amorties* sur ces structures de données seront les mêmes que les complexités *au pire* sur les tableaux habituels).

Implémentons maintenant le type abstrait de file de priorité (impérative) par les tas représentés par des tableaux. D'abord, le plus facile :

```OCaml
(* renvoie le plus petit élément d'un tas *)
let min_tas t =
  assert(t.size > 0);
  t.data.(0)
```

Pour l'insertion d'une nouvelle valeur x dans un tas, la stratégie sera la suivante :
— On insère d'abord x dans la prochaine case libre, qui est `t.data.(t.size)`. À ce moment précis, la propriété de tas min peut ne plus être vérifiée, puisque le père de `t.size` peut contenir une valeur strictement supérieure à x.
— Si le père de la case `t.size` est strictement supérieure, on échange les deux valeurs et on itère sur ce père.
— Si le père de la case courante est inférieure ou égale à la case courante, on s'arrête et le tableau respecte que la propriété de tas min.

Exercice 36

▶ **Question 1** Justifier sommairement la terminaison de cette stratégie.

▶ **Question 2** L'utiliser pour insérer la valeur 3 dans le tas min de l'Exemple 33.

▶ **Question 3** Insérer 6 dans le tas obtenu.

▶ **Question 4** Déterminer (en justifiant sommairement, je n'attends pas de démonstration rigoureuse) la complexité de l'insertion en fonction de n la taille du tas.

On peut l'implémenter comme suis :

```OCaml
(** insère l'élément [x] dans le tas [t], en supposant qu'il y a la
  place *)
let insere t x =
  assert (t.size < Array.length t.data - 1);
  t.data.(t.size) <- x;
  let i = ref t.size in
  while !i > 0 && t.data.(!i) < t.data.(pere !i) do
    swap t.data !i (pere !i);
    i := pere !i
  done;
  t.size <- t.size + 1
```

L'autre transformateur important dans le type abstrait de file de priorité impérative est l'extraction du minimum : non seulement on veut savoir quelle est la plus petite priorité présente dans le tas (facile, elle est dans la racine), mais en plus on veut la retirer du tas. La stratégie est cette fois-ci :
— D'abord, on note la valeur du minimum `t.data.(0)` quelque part (pour le renvoyer à la fin).
— Ensuite, on choisit la dernière valeur du tas (dernière du parcours en largeur, celle la plus à droite dans le tableau) que l'on place à la racine, et on le *percole* vers le bas : on le compare avec ses deux fils, et :
  — s'il est inférieur ou égal à ses deux fils, on s'arrête
  — sinon, on l'échange avec le plus petit des deux et on recommence.

Exercice 37

▶ **Question 1** Dans l'étape de percolation, pourquoi échange-t-on l'élément à percoler avec le plus petit de ses deux fils ?

▶ **Question 2** Appliquer deux fois la suppression du minimum dans le tas obtenu dans l'Exercice 36 après insertion.

▶ **Question 3** Justifier sommairement de sa complexité.

Ce qui nous donne :

OCaml
```
(* renvoie l'argument du min entre deux cases de tab *)
let argmin2 tab i j = if tab.(i) <= tab.(j) then i else j

(* renvoie l'argument du min entre trois cases de tab *)
let argmin3 tab i j k =
  if tab.(i) <= tab.(j) && tab.(i) <= tab.(k) then i
  else if tab.(j) <= tab.(i) && tab.(j) <= tab.(k) then j
  else k

(* effectue l'opération de percolation vers le bas *)
let rec vers_le_bas t i =
  let dest =
    if fils_droit i < t.size then
      argmin3 t.data i (fils_gauche i) (fils_droit i)
    else if fils_gauche i < t.size then
      argmin2 t.data i (fils_gauche i)
    else i
  in
  if dest <> i then begin
    swap t.data i dest;
    vers_le_bas t dest
  end

let retire_min t =
  assert (t.size >= 0);
  let res_final = t.data.(0) in
  t.size <- t.size - 1;
  t.data.(0) <- t.data.(t.size);
  vers_le_bas t 0;
  res_final
```

Ce qui n'est pas vraiment un code compliqué, juste il faut être attentif à comment on s'organise. On laisse le dernier transformateur de côté pour l'instant.

Exemple 34 – Application : le tri par tas

C'est un des exemples de tri que l'on peut coder en utilisant les bonnes propriétés des tas : la stratégie est la suivante.

— En entrée, on prend un tableau `t`
— On insère ensuite successivement `t.(0)`, `t.(1)`, ... dans le tas jusqu'à ce que `size = Array.length t`.
— On obtient alors un tas : on extrait son minimum n fois et on note les valeurs successives dans un autre tableau, qui sera trié dans l'ordre croissant.

En acceptant de trier notre tableau dans l'ordre décroissant, on obtient même un algorithme en place : on considère le tableau lui-même comme un tas initialement à 0, on insère successivement les valeurs de `t` dans lui-même et on retire les minimums au fur et à mesure en les notant de droite à gauche dans `t` :

OCaml
```
let tri_par_tas tab =
let t = { size = 0; data = tab } in
for k = 0 to Array.length tab - 1 do
  insere t tab.(k)
done;
for k = Array.length tab - 1 downto 0 do
  tab.(k) <- retire_min t
done
```

Chaque tour de la première boucle prend un temps $\mathcal{O}(\log n)$, avec `n = Array.length tab`. On a la même chose pour, la seconde, ce qui donne une complexité totale en $\mathcal{O}(n \log n)$. C'est d'ailleurs le tri implémenté pour la fonction `Array.sort` en OCaml.
La première étape (transformer le tableau en tas) peut en fait être effectuée en $\mathcal{O}(n)$, ce qui ne change pas particulièrement la complexité du tri par tas. La stratégie est de construire le tas de droite à gauche plutôt que de gauche à droite.

Revenons à l'opération de diminution d'une valeur du tas : si l'on sait que cette valeur est à l'indice `k`, il suffit de diminuer cette case et de percoler vers le haut. Et ça marche en $\mathcal{O}(\log n)$!