I Structure mathématique

I.1 Généralités

Définition A

Un **arbre** est un ensemble fini A muni d'une relation $\prec$ tel que :

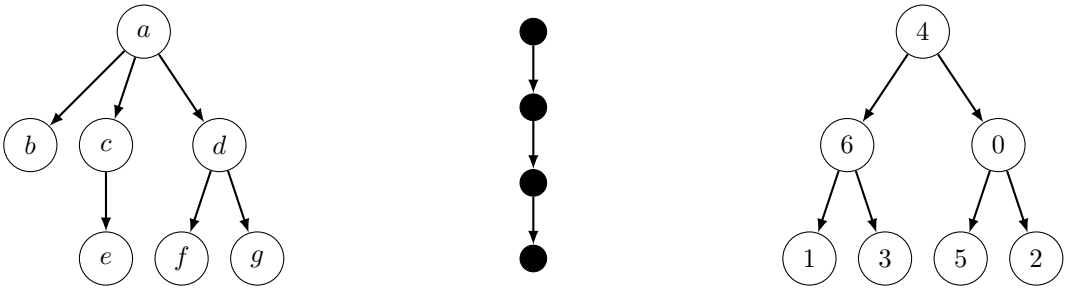
- Il existe $r \in A$, appelé **racine**, tel que pour tout $x \in A$, $r \not\prec x$.
- Pour tout $x \in A$, $x \neq r$, il existe un unique $y \in A$, $y \neq x$, tel que $x \prec y$. On dit que y est le **père** de x ou que x est un **fils** de y.
- Pour tout $x \in A$, $x \neq r$, il existe $y_0 = x, y_1, y_2, \ldots, y_n = r$ tels que $y_0 \prec y_1 \prec \ldots \prec y_n$. La valeur n est appelée **profondeur** de x. Si $i < j$, alors y_i est appelé **descendant** de y_j.

On notera simplement A au lieu de $(A, \prec)$ pour désigner l'arbre.

Exemple B

On représente graphiquement un arbre A de la façon suivante :

- À chaque élément on associe un sommet.
- On trace une flèche entre deux sommets s'ils ont un lien de parenté.
- Les éléments de même profondeur sont représentés au même niveau.
- On peut éventuellement nommer les sommets (en pratique, ça sera généralement le cas).



Définition C

Soit A un arbre.

- Un élément de A est appelé **nœud** de A.
- Un nœud sans fils est appelé **feuille** de A.
- Un nœud qui n'est pas une feuille est appelé **nœud interne**.
- Le nombre de fils d'un nœud x est appelé **arité** de x.
- Un ensemble constitué d'un nœud et de ses descendants est appelé **sous-arbre** de A. S'il n'y a pas d'ambiguïté, on associera généralement un nœud et son sous-arbre associé.

Exemple D

Dans le premier arbre de l'exemple précédent :

- Les feuilles sont les nœuds b, e, f et g. Les nœuds internes sont les nœuds a, c et d.
- L'arité de a est 3. L'arité de d est 2. L'arité de c est 1. L'arité de f est 0.
- L'ensemble formé de d, f et g est un sous-arbre de A de racine d.

Remarque E

On peut définir une feuille comme un nœud d'arité 0.

Définition F

Soit A un arbre. On appelle :

- **taille** de A, notée $|A|$, le cardinal de l'ensemble A.
- **hauteur** de A, notée $h(A)$, la plus grande des profondeurs des éléments de A.
- Sans avoir de noms, on note f_A le nombre de feuilles de A et n_A le nombre de nœuds internes de A.

Exemple G

Le premier arbre de l'exemple précédent a une taille de 7 et une hauteur de 2. Le deuxième arbre a une taille de 4 et une hauteur de 3.

I.2 Arbres binaires

Définition H : Arbre binaire

Soit A un arbre. A est dit binaire si l'arité de chacun de ses nœuds est majorée par 2.

Définition I

Un arbre binaire A est dit **entier** si l'arité de chacun de ses nœuds est soit 0, soit 2.
Il est dit **complet** s'il est entier et que toutes ses feuilles ont même profondeur.

Exemple J

Dans l'exemple précédent, le premier arbre n'est pas binaire. Le deuxième est binaire non entier. Le troisième est binaire entier (et même complet).

Proposition K : Définition inductive

On peut définir inductivement un arbre binaire de la façon suivante en considérant un arbre particulier, l'arbre vide. Un arbre binaire est alors :

- soit l'arbre vide,
- soit un nœud ayant un sous-arbre gauche et un sous-arbre droit.

L'arbre est dit **étiqueté** si, de plus, une valeur est associée à chaque nœud.

Remarque L

Par convention, l'arbre vide est de taille 0 et de hauteur -1.

Proposition M

Soit A un arbre binaire. Alors la taille et la hauteur de A vérifient :

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

Exercice 1

Démontrer la proposition précédente et illustrer les cas d'égalité.

Corolaire N

De même, on a $\log_2(|A| + 1) \leq h(A) \leq |A| - 1$.

Proposition O

Soit A un arbre binaire entier non vide. Alors $n_A = f_A - 1 = \left\lfloor \frac{|A|}{2} \right\rfloor$.

Exercice 2

Démontrer la proposition précédente.

II Implémentation

De par leur définition inductive, les arbres, comme les listes, se prêtent particulièrement bien à la programmation récursive.

On commence par donner une définition générale des arbres. On parlera ici d'arbres étiquetés uniquement (en gardant à l'esprit que les étiquettes sont facultatives).

```
type 'a arbre = Feuille of 'a | Noeud of 'a * 'a arbre list;;
```

Notons que, comme le nombre de fils est arbitraire, on utilise une structure de liste pour la définition inductive.

Exercice 3

1. Écrire une fonction `taille : 'a arbre -> int` qui calcule la taille d'un arbre.
2. Écrire une fonction `affichage : int arbre -> unit` qui affiche toutes les étiquettes d'un arbre (dans l'ordre de son choix).

Dans le cas des arbres binaires, si l'on souhaite utiliser une définition similaire, il faut distinguer en fonction de l'arité d'un nœud interne. On distinguera alors les nœuds simples et les nœuds doubles.

```
type 'a arbre =
   | Feuille of 'a
   | NS of 'a * 'a arbre
   | ND of 'a * 'a arbre * 'a arbre;;
```

Pour limiter le nombre de constructeur, on lui préfèrera la seconde définition inductive, utilisant l'arbre vide.

```
type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre;;
```

En utilisant ce type, la fonction calculant la taille d'un arbre s'écrit classiquement :

```
let rec taille = function
   | Vide -> 0
   | N(x, g, d) -> 1 + taille g + taille d;;
```

Exercice 4

Écrire des fonctions `hauteur`, `feuilles`, `internes`, de signature `'a arbre -> int` calculant les grandeurs associées à un arbre binaire.

Exercice 5

Écrire une fonction `liste_feuilles : 'a arbre -> 'a list` qui renvoie la liste des feuilles de l'arbre, de la gauche vers la droite. On pourra utiliser dans un premier temps l'opérateur `@` de concaténation.

Exercice 6

Écrire une fonction `est_entier : 'a arbre -> bool` qui teste si un arbre est entier ou non.

Exercice 7

Écrire une fonction `maximum : 'a arbre -> 'a` qui renvoie l'élément maximal d'un arbre.

Exercice 8

Écrire une fonction `arbre_alea : int -> int arbre` qui crée un arbre aléatoirement, avec des étiquettes toutes distinctes. On utilisera la fonction `Random.int` telle que `Random.int n` renvoie un entier aléatoire entre 0 et $n-1$. On supposera que les étiquettes d'un sous-arbre gauche seront toujours plus petites que l'étiquette du nœud associé, qui sera plus petite que les étiquettes du sous-arbre droit.

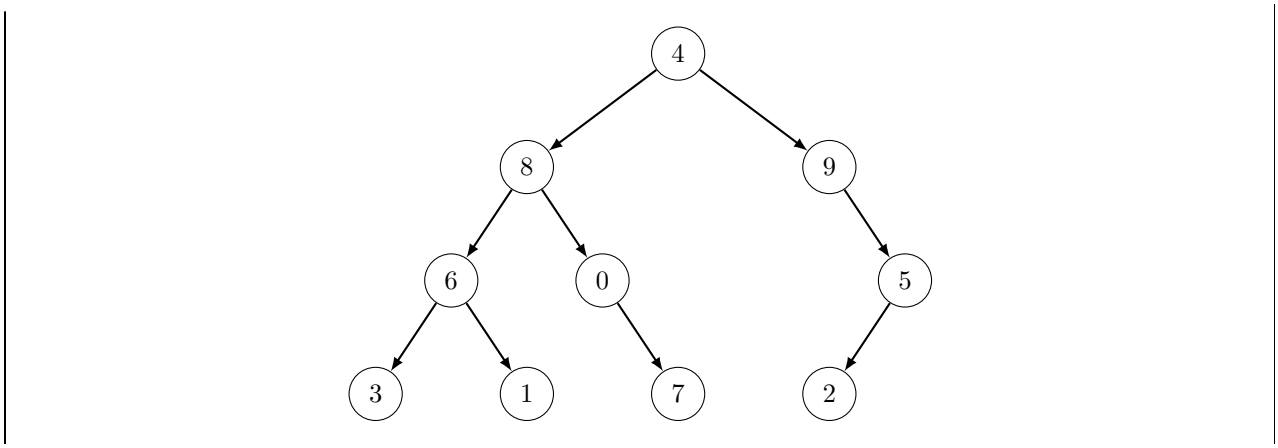
III Parcours d'arbres

Le parcours d'un arbre consiste à lister ses éléments dans un ordre défini à l'avance. Il permet d'effectuer un traitement des données selon certaines règles, comme par exemple le traitement des valeurs par ordre croissant.

III.1 Parcours en profondeur

Dans un parcours en profondeur, on commence par explorer l'arbre gauche dans son ensemble avant de commencer à explorer l'arbre droit. L'exemple ci-dessous illustre le sens de parcours des nœuds.

Exemple P



On distingue trois type de parcours en profondeur, en fonction de l'ordre dans lequel est traité un nœud par rapport à chacun de ses fils :

- Le parcours **préfixe** traite un nœud $N(x,g,d)$ dans l'ordre x puis g puis d. Le parcours préfixe de l'exemple précédent est 4-8-6-3-1-0-7-9-5-2.
- Le parcours **infixe** traite un nœud $N(x,g,d)$ dans l'ordre g puis x puis d. Le parcours infixe de l'exemple précédent est 3-6-1-8-0-7-4-9-2-5.
- Le parcours **postfixe** (ou **suffixe**) traite un nœud $N(x,g,d)$ dans l'ordre g puis d puis x. Le parcours postfixe de l'exemple précédent est 3-1-6-7-0-8-2-5-9-4.

Les fonctions CAML associées (pour des `int arbre`) sont :

```
let rec prefixe = function
   | Vide -> ()
   | N(x, g, d) -> print_int x; prefixe g; prefixe d;;
```

```
let rec infixe = function
   | Vide -> ()
   | N(x, g, d) -> infixe g; print_int x; infixe d;;
```

```
let rec postfixe = function
   | Vide -> ()
   | N(x, g, d) -> postfixe g; postfixe d; print_int x;;
```

Exercice 9

Modifier les fonctions précédentes pour qu'elles soient de signature `'a arbre -> 'a list` et renvoie les listes des éléments de A, dans l'ordre associé. On pourra utiliser dans un premier temps l'opérateur `@` de concaténation, puis écrire la fonction sans utiliser de concaténation.

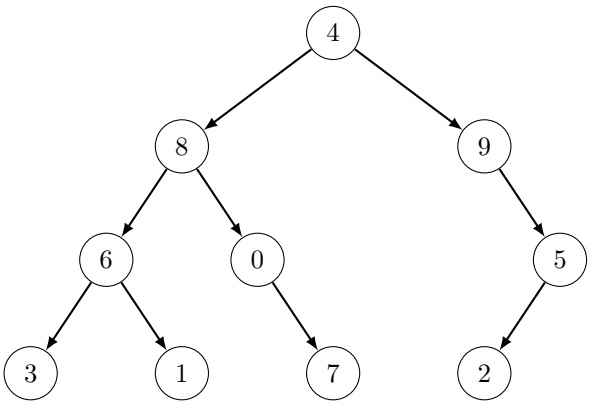
Exercice 10

1. Montrer que la donnée de la liste du parcours préfixe et la liste du parcours infixe permettent de déterminer de manière unique un arbre associé.
2. Écrire une fonction `const_arbre : 'a list -> 'a list -> 'a arbre` qui reconstruit l'arbre associé aux listes du parcours préfixe et du parcours infixe.

III.2 Parcours en largeur

Le parcours en largeur d'un arbre traite les sommets par ordre croissant de profondeur, en commençant du côté gauche. Il nécessite d'utiliser une structure de file pour s'assurer de traiter les sommets dans le bon ordre. L'exemple ci-dessous illustre le sens de parcours :

Exemple Q



La structure de file en Caml est de type `t`, donné par le module `Queue` avec les fonctions suivantes :

- `Queue.create : unit -> 'a t` crée une file vide.
- `Queue.is_empty : 'a t -> bool` teste si la file est vide.
- `Queue.push : 'a -> 'a t -> unit` enfile un élément.
- `Queue.pop : 'a t -> 'a` défile un élément (et renvoie une erreur si la file est vide).

Exercice 11

Écrire une fonction qui affiche les étiquettes dans le sens du parcours en largeur d'un `int arbre`. On commencera par écrire une fonction auxiliaire `traitement` qui gère le traitement d'un nœud et l'ajout éventuel d'éléments à la file.