Chapitre 7

Arbres

1 Notion mathématique d'arbre

1.1 Définitions

Notations. Soit $(A, \leq)$ un ensemble partiellement ordonné. Pour tout a appartenant à A, nous notons :

- $\text{Maj}(a) = \{x \in A \mid a \leq x\}$, ensemble des majorants de a,
- $\text{Maj}^+(a) = \{x \in A \mid a \leq x \text{ et } x \neq a\}$, ensemble des majorants stricts de a,
- $\text{Min}(a) = \{x \in A \mid x \leq a\}$, ensemble des minorants de a,
- $\text{Min}^+(a) = \{x \in A \mid x \leq a \text{ et } x \neq a\}$, ensemble des minorants stricts de a.

Définition.

- Une **forêt** est un ensemble fini non vide partiellement ordonné $(A, \leq)$ tel que pour tout a appartenant à A, $\text{Min}(a)$ est une partie totalement ordonnée de A.
- Un **arbre** est une foret $(A, \leq)$ tel que A possède un plus petit élément $\min(A)$ appelé **racine de** A.

Remarque. Suivant les situations, nous pouvons considérer que l'ensemble vide est ou non un arbre.

Exemple. $A = [\![0, 10]\!]$ et nous décidons de noter pour a et b appartenant à A :

$\Leftrightarrow$ a divise b $\Leftrightarrow$ $a < b$.

1. Proposer une "structure arborescente" de l'ensemble $(A, |)$.

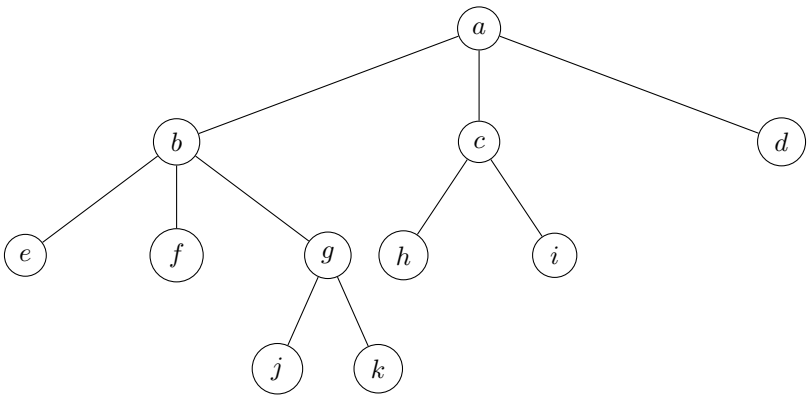
2. Est-ce que $(A, |)$ est une arbre ? Justifier.

Définition.

Soit $(A, \leq)$ un arbre non vide.

- On appelle **sommet** tout élément de A.
- On appelle **feuille** tout élément maximal de A, c'est-à-dire un élément a de A tel qu'aucun élément de A ne majore strictement a.
- On appelle **noeud** tout élément non-maximal de A.

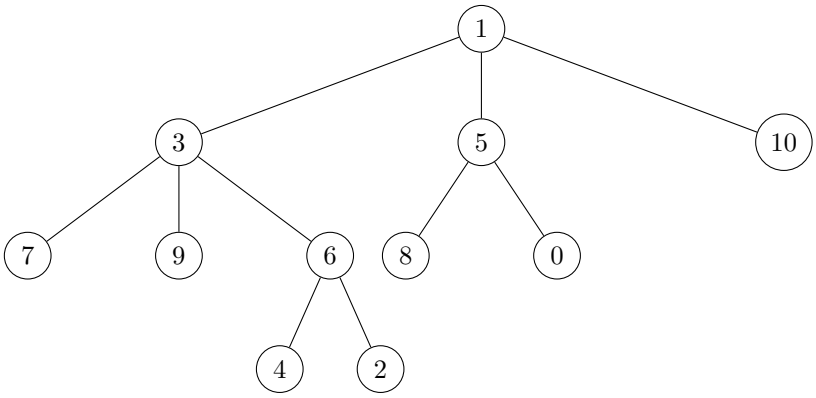
Exemple. Considérons un arbre A sous forme arborescente :



Notons $<$ la relation de précédence. Par exemple : $a < b$, $g < j$, $c < i$...
L'ordre partiel sur A est la clôture réflexive et transitive $\leq$ de $<$. Par exemple : $a \leq b \leq g \leq k$.
Les sommets de A sont $a, b, c, d, e, f, g, h, i, j, k$. Les feuilles de A sont les sommets e, f, j, k, h, i, d et les noeuds de A sont les sommets a, b, c, g.

Remarque. Il est possible de stocker des informations soit au niveau des noeuds, des feuilles ou des sommets.

- Lorsque les informations sur les noeuds et les feuilles sont de même type, nous parlons d'**arbre homogène**. Par exemple :



- Lorsque les informations sur les noeuds et les feuilles ne sont pas de même type, nous parlons d'**arbre hétérogène**. Par exemple, pour représenter la formule mathématique $\sin(3.0 + 4.0) \times \ln(2.0 \times \pi)$:

Définition.

Soit $(A, \leq)$ un arbre ayant au moins deux sommets et a un sommet de A distinct de la racine.
Comme $\text{Min}^+(a)$ est une partie non vide de A et totalement ordonnée, on dit que :

- un **ascendant** de a est un élément de $\text{Min}(a)$,
- un **ascendant strict** de a est un élément de $\text{Min}^+(a)$,
- le **père** de a est $\max\left(\text{Min}^+(a)\right)$.

Définition.

Soit $(A, \leq)$ un arbre ayant au moins deux sommets et a un sommet de A qui ne soit pas une feuille.
Comme $\text{Maj}^+(a)$ est une partie non vide de A et partiellement ordonnée, on dit que :

- un **descendant** de a est un élément de $\text{Maj}(a)$,
- un **descendant strict** de a est un élément de $\text{Maj}^+(a)$,
- un **fils** de a est un élément minimal de $\text{Maj}^+(a)$.

Attention.

Un sommet a distinct de la racine admet un **unique père**.
Un sommet a qui n'est pas une feuille peut admettre **plusieurs fils**.

Propriété 1 (Lien entre père et fils)

(1) Tout sommet autre que la racine est le fils de son père.

(2) Tout sommet autre qu'une feuille est le père de chacun de ses fils.

Définition.

Soit $(A, \leq)$ un arbre ayant au moins deux sommets.

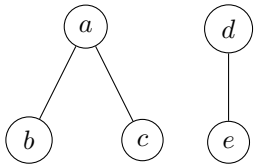
- Une **arête** de A est un couple de sommets en relation père-fils.
- Un **chemin** entre deux sommets a et b de A est une séquence $s_0, s_1, \ldots, s_n$ de sommets de A tels que :

$$s_0 = a, \quad s_n = b, \quad \text{et} \quad \forall i \in [\![0, n-1]\!],\ (s_i, s_{i+1}) \text{ est une arête}$$

  On dit que n est la longueur du chemin.
- Un **chemin direct** entre deux sommets a et b de A est un chemin entre a et b ne passant pas deux fois par le même sommet. Pour l'obtenir, on remonte en fait à l'ascendant commun le plus proche et c'est le chemin le plus court.

Remarques.

1. On peut définir de même la notion de chemin pour une forêt $(A, \leq)$.
2. Comme on l'a déjà dit, il existe toujours un chemin entre deux sommets d'un arbre. Ce n'est pas vrai pour une forêt. Par exemple, pour la forêt



il n'y a pas de chemin entre les sommets b et e.

Définition.

Soit $(A, \leq)$ une forêt non vide.
Une **composante connexe** de A est un sous-ensemble de sommets B de A tel que, pour tout $a, b \in B$, il existe un chemin entre a et b.

Propriété 2 (Composantes connexes d'une forêt)

Soit $(A, \leq)$ une forêt non vide.
Chaque **composante connexe** de A, munie de la relation d'ordre partielle induite par $\leq$, est un **arbre**.

Définition.

Soit $(A, \leq)$ un arbre ayant au moins deux sommets a et b.

- On dit que l'arbre $\text{Maj}(a)$ est un **sous-arbre** de A de racine a.
- On dit que $\text{Maj}^+(a)$ est la **forêt** formée des branches issues de a.

Définition.

Soit $(A, \leq)$ un arbre non vide.

- La **hauteur** de A est la longueur maximale d'un chemin direct allant de la racine à une feuille.
- La **profondeur** d'un sommet a de A est la longueur du chemin direct allant de la racine à a.

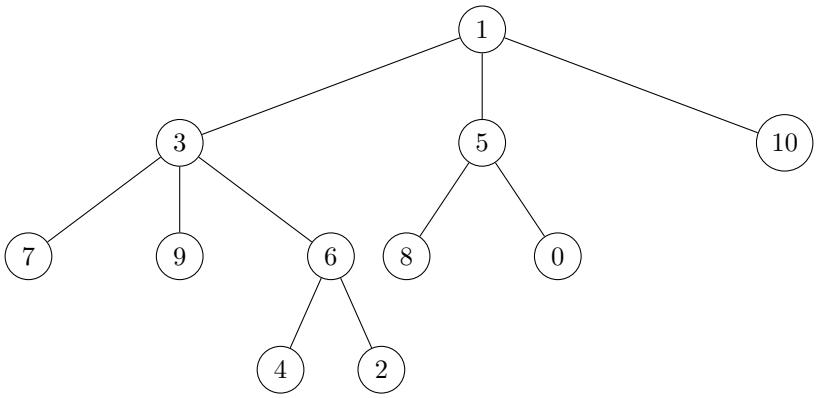
Remarque. La hauteur de l'arbre vide n'est pas définie.

Définition.

Soit $(A, \leq)$ un arbre non vide et $n \in \mathbb{N}^*$.

- L'**arité** d'un sommet a de A est le nombre de fils de a. Les feuilles sont d'arité nulle et les noeuds sont d'arité non nulle.
- Les sommets d'arité 1 sont dits **unaires**, les sommets d'arité 2 sont dits **binaires**.
- Un **arbre n-aire** est un arbre dont tous les noeuds sont d'arité inférieure ou égale à n.
- Un **arbre n-aire entier** est un arbre dont tous les noeuds sont d'arité égale à n .

Remarque. Il est possible de référencer la position des sommets d'un arbre A par des mots dont les lettres appartiennent à l'alphabet $[\![1, M]\!]$ où M désigne le maximum des arités des noeuds de A. Par exemple, dans l'arbre



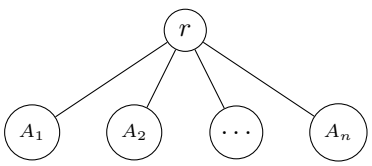
6 est en position 1|3, 0 est en position 2|2, 10 est en position 3...

1.2 Définitions récursives

Définition.

Soit Σ un ensemble.
On peut définir de manière récursive des arbres sur l'ensemble Σ :

- L'arbre vide est un arbre sur Σ.
- r étant un élément de Σ d'arité n (avec $n \in \mathbb{N}^*$) et $A_1, A_2, \ldots, A_n$ étant n arbres sur Σ, alors



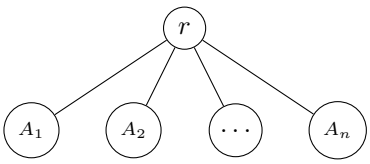
est un arbre sur Σ .

Remarque. Une liste est un arbre unaire entier (représentation horizontale d'une liste chaînée).

Définition.

Soit $(A, \leq)$ un arbre.
On peut définir de manière récursive la hauteur $h(A)$ de A :

- Si A est vide, $h(A)$ n'est pas définie.
- Si A est un arbre réduit à la racine, $h(A) = 0$.
- Sinon, A est de la forme :



où r est une racine d'arité non nulle n, et alors : $h(A) = 1 + \max(h(A_1), \ldots, h(A_n))$.

Remarque. De sa définition récursive, les objets définis sur un arbre et les preuves sur les arbres se feront par induction structurelle.

1.3 Combinatoire sur les arbres

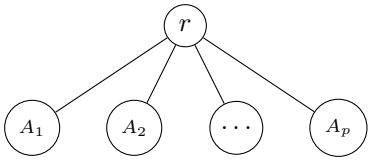
Propriété 3 (Lien entre le nombre de sommets et le nombre d'arêtes)

Soit n un entier naturel non nul.
Tout arbre non vide à n sommets possède $n - 1$ arêtes.

Preuve. Par induction structurelle à partir de la définition récursive d'arbre.

Cas de base : Soit A un arbre à 1 sommet, alors A possède 0 arête.

Etape d'induction : On considère un arbre

$A =$ 

où r est la racine d'arité p et $A_1, A_2, \ldots, A_p$ sont p arbres non vides vérifiant la propriété.
Notons, pour tout arbre B, $N_s(B)$ le nombre de sommets de B et $N_a(B)$ le nombre d'arêtes de B. Alors, en utilisant l'hypothèse d'induction structurelle à la deuxième égalité :

$$\begin{aligned} N_a(A) &= N_a(A_1) + \ldots + N_a(A_p) + p \\ &= N_s(A_1) - 1 + \ldots + N_s(A_p) - 1 + p \\ &= N_s(A_1) + \ldots + N_s(A_p) \\ &= N_s(A) - 1. \end{aligned}$$

Par induction structurelle, le résultat est vrai. □

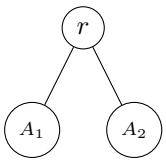
Propriété 4 (Lien entre le nombre de noeuds et de feuilles d'un arbre binaire entier)

Soit n un entier naturel.
Tout arbre binaire entier non vide à n noeuds possède $n + 1$ feuilles.

Preuve. Par induction structurelle à partir de la définition récursive d'arbre.

Cas de base : Soit A un arbre binaire entier à 0 noeud. A possède 1 feuille (la racine).

Etape d'induction : On considère l'arbre binaire entier

$A =$ 

où r est la racine d'arité 2 et A_1, A_2 sont 2 arbres binaires entiers non vides vérifiant la propriété.
Notons, pour tout arbre B, $N_f(B)$ le nombre de feuilles de B et $N_n(B)$ le nombre de noeuds de B. Alors, en utilisant l'hypothèse d'induction structurelle à la deuxième égalité :

$$N_f(A) = N_f(A_1) + N_f(A_2) = N_n(A_1) + 1 + N_n(A_2) + 1 = N_n(A) + 1.$$

Par induction structurelle, le résultat est vrai. □

Propriété 5 (Lien entre le nombre de noeuds et la hauteur d'un arbre binaire)

Soient n et h deux entiers naturels.
Tout arbre binaire non vide à n noeuds et de hauteur h vérifie :

$$h \leq n \leq 2^h - 1 \qquad \text{et donc} \qquad \lceil \log_2(n+1) \rceil \leq h \leq n,$$

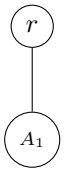
où $\lceil x \rceil$ est la partie entière supérieure d'un réel x.

Preuve. Par induction structurelle à partir de la définition récursive d'arbre.

Cas de base : Soit A un arbre binaire à 0 noeud. A est de hauteur 0 et on a bien la relation demandé.

Etape d'induction : On considère un arbre binaire A à n noeuds. Notons, pour tout arbre B, $N_n(B)$ le nombre de noeuds de B et $h(B)$ sa hauteur. On a deux cas :

- Soit A est de la forme



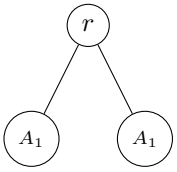
où r est la racine d'arité 1 et A_1 est un arbre binaire non vide vérifiant la propriété. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leq 1 + N_n(A_1) = N_n(A)$$

et

$$N_n(A) = N_n(A_1) + 1 \leq 2^{h(A_1)} - 1 + 1 = 2^{h(A_1)} \leq 2^{h(A_1)} + \underbrace{2^{h(A_1)} - 1}_{0\leq} = 2^{h(A_1)+1} - 1 = 2^{h(A)} - 1.$$

- Soit A est de la forme



où r est la racine d'arité 2 et A_1, A_2 sont des arbres binaires non vides vérifiant la propriété. Supposons par exemple $h(A_1) \geq h(A_2)$. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leq 1 + N_n(A_1) \leq 1 + N_n(A_1) + N_n(A_2) = N_n(A)$$

et

$$N_n(A) = 1 + N_n(A_1) + N_n(A_2) \leq 1 + 2^{h(A_1)} - 1 + 2^{h(A_2)} - 1 \leq 2^{h(A_1)} + 2^{h(A_1)} - 1 = 2^{h(A)} - 1.$$

Par induction structurelle, le résultat est vrai. □

Remarque. La complexité de nombreux algorithmes sur les arbres se calcule en fonction de la hauteur de ces arbres, d'où l'idée d'avoir la hauteur la plus petite possible.
Dans le cas des arbres binaires :

- configuration la pire : $n = h$, nous sommes en fait ramenés à une liste,
- configuration la meilleure : $n = 2^h - 1$, configuration d'arbre binaire "complet", configuration idéale mais réalisable que pour certaines valeurs de n.

2 Notion informatique d'arbre

2.1 Arbres binaires

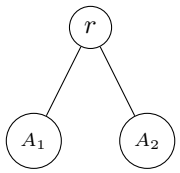
Structure de données

On utilise la définition inductive suivante :

Définition.

Un arbre binaire homogène d'éléments de type a est une structure de données qui est :

- soit l'arbre `Vide`;
- soit un triplet A composé d'un élément r de type a et de deux autres arbres binaires homogènes A_1 et A_2 d'éléments de type a :

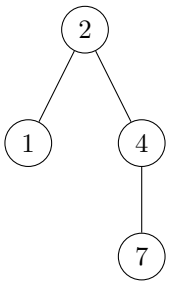
$A =$ 

L'élément r est la racine de A, A_1 est le "fils gauche de A", A_2 le "fils droit de A". Une feuille est alors un noeud dont les deux fils sont des arbres vides.

Une implémentation en OCaml simple d'un type d'arbre binaire homogène est le type personnalisé suivant :

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

Exemple. Définir l'arbre suivant sur OCaml :




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

Opérations sur les arbres binaires

Schématiquement, les fonctions sont de la forme :

```
let rec f a = match a with
  | Vide -> ...
  | N(_, _, _) -> ...
;;
```

Exercices.

1. Définir une fonction en OCaml qui calcule le nombre de sommets d'un arbre binaire homogène :

```
let rec sommets a = match a with
  | Vide -> 0
  | N(_, g, d) -> 1 + sommets g + sommets d
;;
```

2. Définir une fonction en OCaml qui calcule le nombre de noeuds d'un arbre binaire homogène :

```
let rec noeuds a = match a with
  | Vide -> 0
  | N(_, Vide, Vide) -> 0
  | N(_, g, d) -> 1 + noeuds g + noeuds d
;;
```

3. Définir une fonction en OCaml qui calcule le nombre de feuilles d'un arbre binaire homogène :

```
let rec feuilles a = match a with
  | Vide -> 0
  | N(_, Vide, Vide) -> 1
  | N(_, g, d) -> feuilles g + feuilles d
;;
```

4. Définir une fonction en OCaml qui calcule la hauteur d'un arbre binaire homogène :

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

5. Définir une fonction en OCaml qui détermine l'étiquette maximale dans un arbre binaire homogène :

```
let rec maximum a = match a with
  | Vide -> failwith "Arbre vide"
  | N(r, Vide, Vide) -> r
  | N(r, g, Vide) -> max r (maximum g)
  | N(r, Vide, d) -> max r (maximum d)
  | N(r, g, d) -> max r (max (maximum g) (maximum d))
;;
```

6. Définir une fonction en OCaml qui teste si un arbre binaire homogène est entier ou non :

```
let rec entier a = match a with
  | Vide -> true
  | N(_, Vide, Vide) -> true
  | N(_, Vide, _) -> false
  | N(_, _, Vide) -> false
  | N(_, g, d) -> entier g && entier d
;;
```

Cas des arbres binaires entiers homogènes

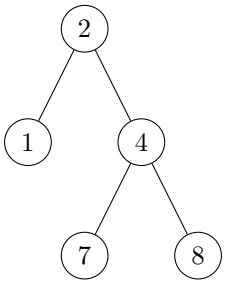
Deux possibilités pour les définir :

- Soit tous les noeuds ont des étiquettes de même type et les feuilles sont des arbres vides. Cela revient à utiliser le type défini précédemment pour les arbres binaires.

- Soit tous les sommets ont des étiquettes de même type. Dans ce cas, on suppose qu'il n'y a pas d'arbre vide et on doit distinguer les noeuds des feuilles. Voici l'implémentation en OCaml :

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

Exemple. Définir l'arbre suivant sur OCaml en utilisant les deux méthodes décrites ci-dessus :




```
let a = N(2, N(1, Vide, Vide), N(4, N(7, Vide, Vide), N(8, Vide, Vide))) ;;

let a = N(2, F 1, N(4, F 7, F 8)) ;;
```

Remarque. La plupart des fonctions sur les arbres binaires homogènes s'adaptent au cas des arbres binaires entiers homogènes.

Cas des arbres binaires entiers hétérogènes

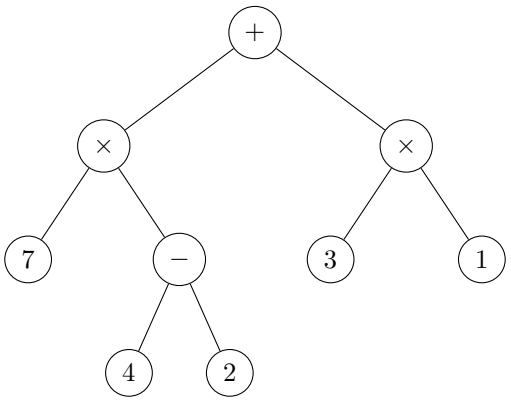
Il est également possible de définir des arbres binaires entiers hétérogènes en différenciant le type des étiquettes des noeuds et celles des feuilles. On utilise un type union avec argument polymorphe et récursif :

```
type ('f,'n) arbre =
  | F of 'f
  | N of 'n * (('f,'n) arbre) * (('f,'n) arbre)
;;
```

Exemple. Déterminer la représentation arborescente de la formule

$$(7 \times (4 - 2)) + (3 \times 1)$$

et définir alors l'arbre binaire entier hétérogène associé.




```
let a = N((+), N((*), F 7, N((-), F 4, F 2)), N((*), F 3, F 1)) ;;
```

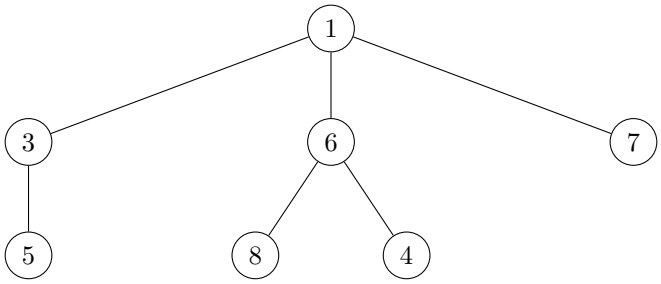
2.2 Cas général

Structure de données

Dans le cas général, le nombre de fils maximal d'un noeud n'est a priori pas connu. Pour implémenter cette structure, on utilise des listes :

```
type 'a arbre = N of 'a * ('a arbre list) ;;
```

Exemple. Définir l'arbre suivant sur OCaml :




```
let a = N(1, [N(3, [N(5, [])]); N(6, [N(8, []); N(4, [])]); N(7, [])]);;
```

Opérations sur les arbres

Schématiquement, les fonctions sont de la forme :

```
let rec f a = match a with
  | N(_, []) -> ...
  | N(_, b::q) -> ...
;;
```

Exercices.

1. Définir une fonction en OCaml qui calcule le nombre de sommets d'un arbre homogène :

```
let rec sommets a = match a with
  | N(_, []) -> 1
  | N(x, b::q) -> sommets b + sommets (N(x, q))
;;
```

2. Définir une fonction en OCaml qui calcule la hauteur d'un arbre homogène :

```
let rec hauteur a = match a with
  | N(_, []) -> 0
  | N(x, b::q) -> max (1 + hauteur b) (hauteur (N(x, q)))
;;
```

Cas des arbres hétérogènes

Il est également possible de différencier le type des étiquettes des noeuds internes et celles des feuilles :

```
type ('f,'n) arbre =
  | F of 'f
  | N of 'n * (('f,'n) arbre list)
;;
```

3 Parcours d'arbres binaires

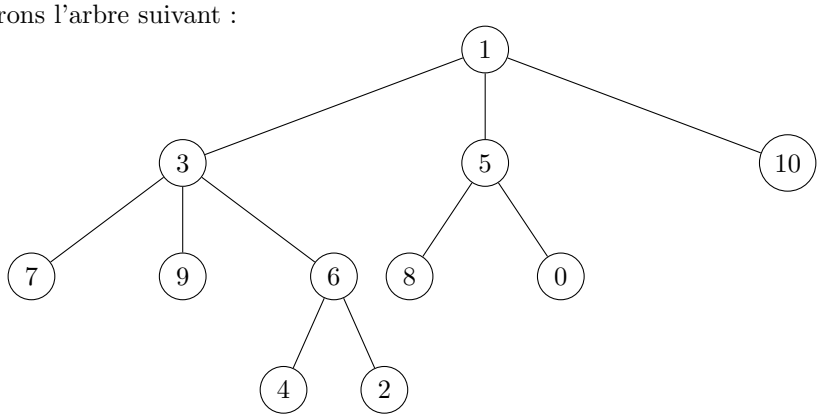
Un parcours d'un arbre est une visite de ses sommets. Il permet d'effectuer un traitement des sommets (affichage, calcul,...) dans un certain ordre choisi à l'avance.

3.1 Parcours en profondeur

Pour visiter un arbre en profondeur, on procède récursivement en visitant entièrement les sous-arbres de gauche à droite. Chaque sommet est ainsi visité un nombre de fois égal à son arité +1. Il y a alors deux possibilités :

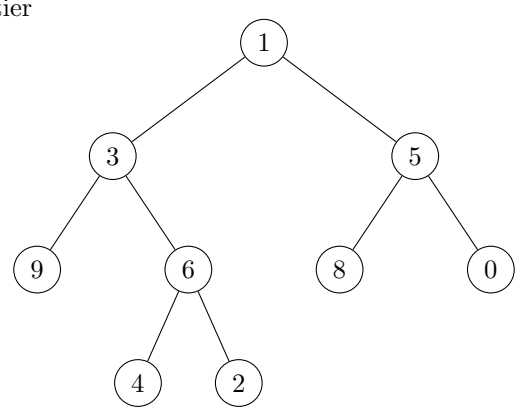
- Traiter chaque sommet à la première visite : c'est le **parcours en profondeur en mode préfixe**.
- Traiter chaque sommet à la dernière visite : c'est le **parcours en profondeur en mode postfixe**.

Exemple. Considérons l'arbre suivant :



Le parcours en profondeur en mode préfixe est $[1; 3; 7; 9; 6; 4; 2; 5; 8; 0; 10]$.
Le parcours en profondeur en mode postfixe est $[7; 9; 4; 2; 6; 3; 8; 0; 5; 10; 1]$.

Remarque. Dans le cas des arbres binaires entiers, chaque sommet est visité 3 fois et on peut ainsi décider d'effectuer le traitement à la deuxième visite : c'est le **parcours en profondeur en mode infixe**. Par exemple, pour l'arbre binaire entier



le parcours en profondeur en mode infixe est $[9; 3; 4; 6; 2; 1; 8; 5; 0]$.

Exercices. On travaille avec le type `'a arbre` suivant :

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

1. Définir une fonction en OCaml qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en profondeur en mode préfixe d'un arbre binaire entier homogène :

```
let rec prefixe a = match a with
  | Vide -> []
  | N(r, g, d) -> (r :: prefixe g) @ (prefixe d)
;;
```

2. Définir une fonction en OCaml qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en profondeur en mode postfixe d'un arbre binaire entier homogène :

```
let rec postfixe a = match a with
  | Vide -> []
  | N(r, g, d) -> (postfixe g) @ ((postfixe d) @ [r])
;;
```

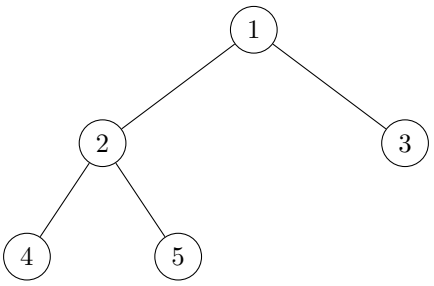
3. Définir une fonction en OCaml qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en profondeur en mode infixe d'un arbre binaire entier homogène :

```
let rec infixe a = match a with
  | Vide -> []
  | N(r, g, d) -> ((infixe g) @ [r]) @ (infixe d)
;;
```

Propriété 6 (Parcours préfixe et postfixe)

Un parcours préfixe (respectivement postfixe) d'un arbre binaire entier où l'on a distingué les feuilles des noeuds correspond à un unique arbre binaire entier.

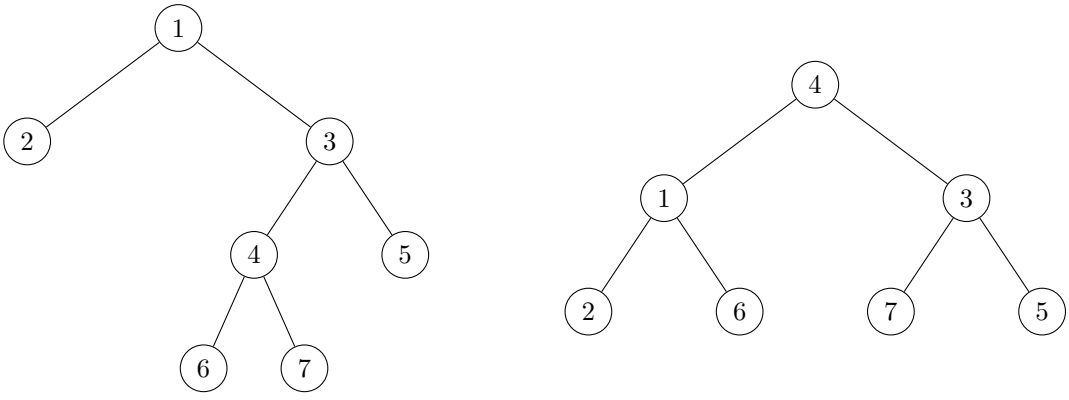
Exemple. L'arbre binaire entier associé au parcours préfixe $[1; 2; \mathbf{4}; \mathbf{5}; \mathbf{3}]$, où les feuilles sont indiquées en gras, est :



Remarque. Cette propriété justifie l'utilisation de l'écriture de Lukasiewicz (notation polonaise inverse) pour les expressions arithmétiques ou les expressions logiques.

Attention.

Le parcours infixe ne décrit pas de manière unique l'arbre. Par exemple, le parcours infixe $[\mathbf{2}; 1; \mathbf{6}; 4; \mathbf{7}; 3; \mathbf{5}]$ avec les feuilles en gras peut correspondre aux deux arbres distincts suivants :



Exercices. On commence par définir le type `'a sommet list` pour décrire la liste des sommets où l'on a distingué les feuilles des noeuds :

```
type 'a sommet =
  | SF of 'a
  | SN of 'a
;;
```

On travaille avec le type `'a arbre` suivant :

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

1. Définir une fonction en OCaml qui à partir de la liste des sommets parcourus lors d'un parcours en profondeur préfixe, reconstitue l'arbre binaire entier homogène associé :

```
let reconstruire_préfixe l =
  let rec aux l = match l with
    | [] -> failwith "Liste vide"
    | SF x :: q -> F x, q
    | SN x :: q -> let g, qg = aux q in
                   let d, qd = aux qg in
                     N(x, g, d), qd in
  fst (aux l)
;;
```

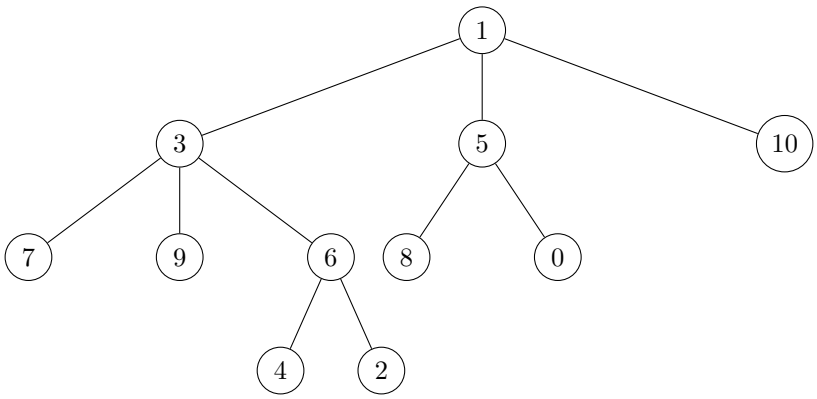
2. Définir une fonction en OCaml qui à partir de la liste des sommets parcourus lors d'un parcours en profondeur postfixe, reconstitue l'arbre binaire entier homogène associé :

```
let reconstruire_postfixe l =
  let rec aux p l = match p, l with
    | [a], [] -> a
    | d :: g :: q, SN x :: r -> aux (N(x, g, d) :: q) r
    | _, SF x :: r -> aux (F x :: p) r in
  aux [] l
;;
```

3.2 Parcours en largeur

Le **parcours en largeur** d'un arbre correspond à la visite des sommets profondeur par profondeur : la racine, puis les sommets de profondeur 1 (de gauche à droite), puis les sommets de profondeur 2 (de gauche à droite), et ainsi de suite.

Exemple. Considérons l'arbre suivant :



Le parcours en largeur est $[1; 3; 5; 10; 7; 9; 6; 8; 0; 4; 2]$.

Exercices. On travaille avec le type `'a arbre` suivant :

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

1. Pour déterminer la liste du parcours en largeur d'un arbre, on va utiliser des listes d'arbres (c'est-à-dire des forêts). Définir deux fonctions en OCaml qui donnent la liste des racines et la liste des fils (sous-arbres) des racines d'une forêt d'arbres binaires entiers homogènes non vides :

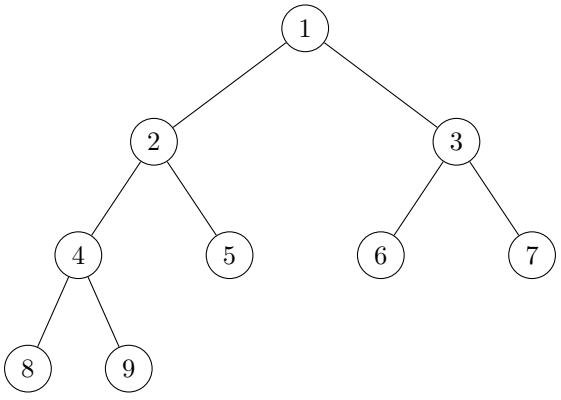
```
let rec liste_racines l = match l with
  | [] -> []
  | N(x, _, _) :: q -> x :: listes_racines q
;;

let rec liste_fils l = match l with
  | [] -> []
  | N(x, Vide, Vide) :: q -> liste_fils q
  | N(x, g, d) :: q -> g :: d :: liste_fils q
;;
```

2. Définir une fonction en OCaml qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en largeur d'un arbre binaire entier homogène :

```
let parcours_largeur a =
  let rec aux l = match l with
    | [] -> []
    | _ -> (liste_racines l) @ aux (liste_fils l) in
  aux [a]
;;
```

Remarque. Dans le cas d'un arbre binaire entier, en numérotant les sommets dans l'ordre de visite du parcours en largeur,



nous pouvons remarquer que :

- numéro du fils gauche = (numéro du sommet) $\times 2$,
- numéro du fils droit = (numéro du sommet) $\times 2 + 1$,
- numéro du père = $\left\lfloor \dfrac{\text{(numéro du sommet)}}{2} \right\rfloor$.

Un tel arbre est avantageusement codé en tableau :

		père		sommet		fg	fd	
		$n/2$		n		$2n$	$2n+1$	

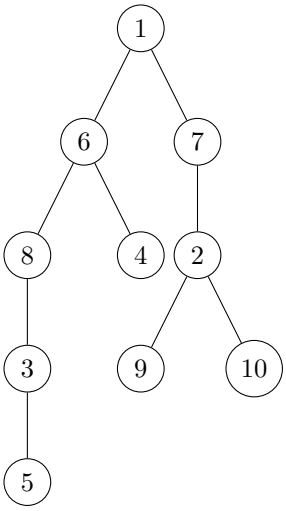
4 Exercices

Exercice 1 (Arbres binaires homogènes)

Dans cet exercice, nous choisissons d'implémenter en OCaml le type "arbre binaire homogène" par le type personnalisé suivant :

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

1. Définir en OCaml l'arbre `a_ex` suivant :



2. Écrire une fonction `branchedroite` qui prend un arbre binaire et qui renvoie la liste des étiquettes de la branche le plus à droite.

   Par exemple, appliquée sur l'arbre `a_ex`, la fonction devra renvoyer

   ```
   - :  int list =[1;7;2;10]
   ```

3. (a) Écrire une fonction `feuilles` qui prend un arbre binaire et qui renvoie la liste de ses feuilles. Par exemple, appliquée sur l'arbre `a_ex`, la fonction devra renvoyer

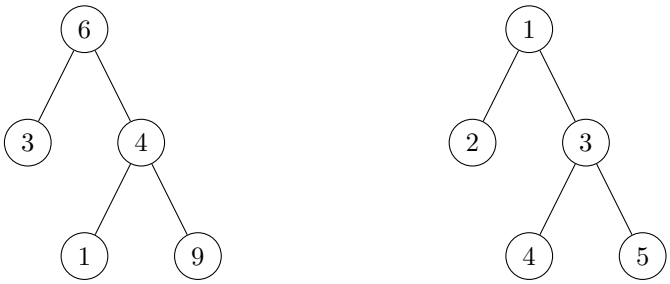
   ```
   - :  int list =[5;4;9;10]
   ```

   (b) Écrire une version récursive terminale de la fonction `feuilles`.

   On utilisera une fonction auxiliaire récursive terminale qui prend en entrée l'arbre ainsi qu'un accumulateur contenant la liste des feuilles déjà traitées.

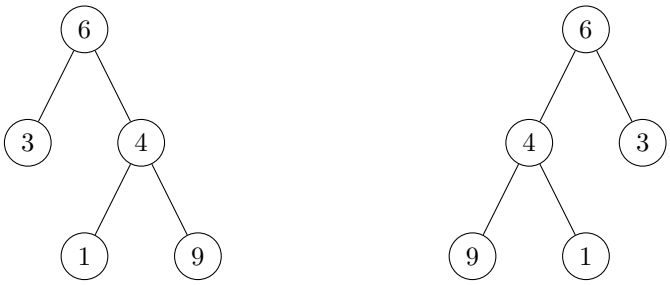
4. Écrire une fonction `squelette` qui teste l'égalité du "squelette" (structure sans étiquettes) de deux arbres binaires.

   Par exemple, les deux arbres suivants ont le même squelette :



5. Écrire une fonction donnant le miroir d'un arbre donné (au sens d'une symétrie par rapport à un "axe vertical").

   Par exemple, voici un arbre et son miroir :



6. La numérotation Sosa d'un arbre binaire est définie comme suit :
   - Le numéro de la racine est 1 ;
   - Si k est le numéro d'un noeud, alors son fils gauche (respectivement droit) a le numéro $2k$ (respectivement $2k+1$).

   (a) Écrire une fonction `sosa` qui reconstruit l'arbre donné en argument en remplaçant l'étiquette de chaque noeud par sa numérotation Sosa.

   (b) Représenter graphiquement l'arbre renvoyé par l'instruction `sosa a_ex;;`.

7. Écrire une fonction `test_complet` qui vérifie si un arbre binaire est complet.

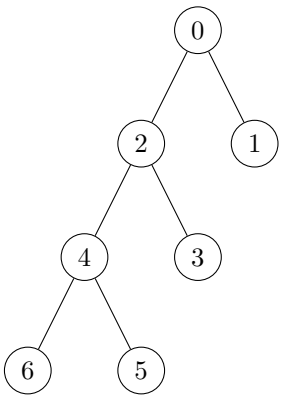
   On pourra utiliser une fonction récursive auxiliaire qui renvoie un couple (b, h) où b est est un booléen qui ne vaut `true` que si l'arbre est complet et h est la hauteur de l'arbre.

   Si l'arbre est vide, on renverra un message d'erreur.

8. Écrire une fonction créant un arbre binaire complet d'une hauteur donnée.

   Les nœuds seront notés selon la numérotation Sosa.

9. On appelle "peigne" un arbre binaire entier dont tous les fils droits sont des feuilles :



   Écrire une fonction créant un peigne d'une hauteur donnée.

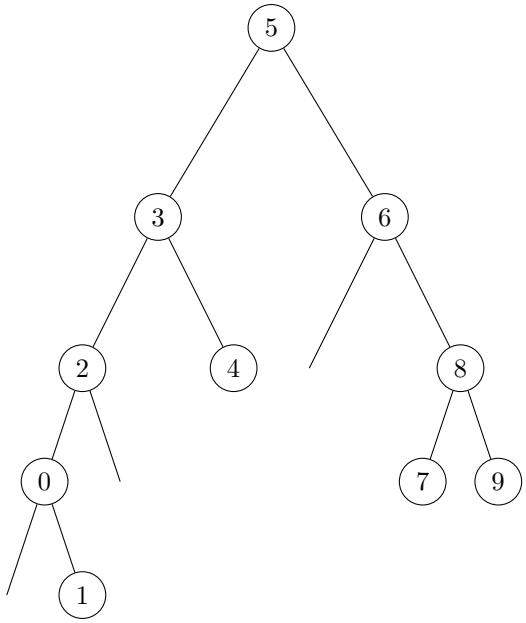
   On s'efforcera de distribuer les étiquettes de façon injective.

10. Écrire une fonction `arbre_alea : int -> int arbre` qui, étant donné n, crée un arbre binaire à n nœuds 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 noeud associé, qui sera plus petite que les étiquettes du sous-arbre droit (on dit que c'est un **arbre binaire de recherche**).

L'appel de `arbre_alea 10` pourra par exemple renvoyer :

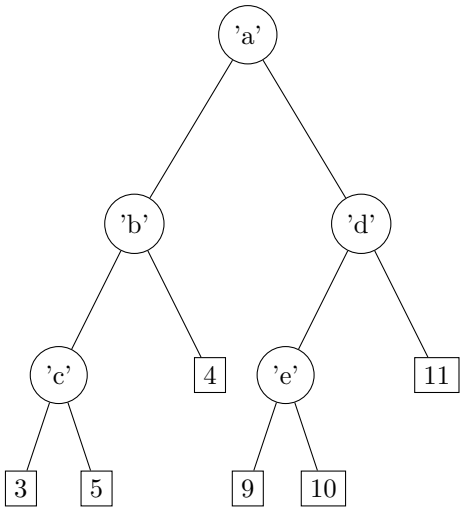


Exercice 2 (Arbres binaires entiers hétérogènes)

Dans cet exercice, nous choisissons d'implémenter en OCaml le type "arbre binaire entier hétérogène" par le type personnalisé suivant :

```
type ('f,'n) arbre_binaire =
  | Feuille of 'f
  | Noeud of 'n * ('f,'n) arbre_binaire * ('f,'n) arbre_binaire ;;
```

1. Définir en OCaml l'arbre `a_ex` suivant:



2. Écrire une fonction OCaml `hauteur` qui détermine la hauteur d'un arbre binaire entier hétérogène.

3. Écrire une fonction OCaml `liste_a_profondeur` qui prend en arguments un arbre binaire entier hétérogène `a` et un entier n et qui renvoie la liste (éventuellement vide) de tous les sous-arbres de `a` dont la racine est à la profondeur n dans `a`.

4. Un arbre sera dit *équilibré* quand ses feuilles se répartissent sur au plus deux niveaux seulement. Ce sera le critère essentiel d'efficacité des algorithmes de recherche à l'aide de structures d'arbres.

   Écrire une fonction OCaml `est_equilibre` qui prend en argument un arbre binaire entier hétérogène et qui dit si oui ou non il est équilibré.

Exercice 3 (Reconstitution d'arbres binaires)

Dans cet exercice, nous choisissons d'implémenter en OCaml le type "arbre binaire homogène" par le type personnalisé suivant :

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

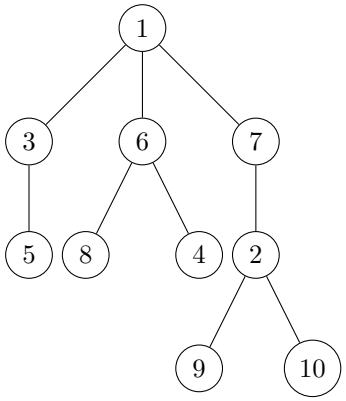
1. Trouver l'arbre binaire homogène ayant pour parcours préfixe la liste `[2; 4; 1; 5; 3]` et pour parcours infixe la liste `[1; 4; 5; 2; 3]`.
2. Montrer par récurrence forte sur la taille des listes que la donnée de la liste du parcours préfixe et la liste du parcours infixe permettent de déterminer de manière unique un arbre binaire homogène dont les étiquettes des sommets sont deux à deux distinctes.
3. (a) Écrire une fonction `indice` qui permet de déterminer la position d'un élément dans une liste.
   (b) Écrire une fonction d'entête `let partition l n` qui permet de séparer une liste ℓ en deux sous-listes ℓ_1 et ℓ_2 telle que la taille de ℓ_1 est composée des n premiers éléments de ℓ.
   (c) À l'aide des fonctions précédentes, écrire une fonction `const_arbre : 'a list -> 'a list -> 'a arbre` qui reconstruit l'arbre binaire associé aux listes du parcours préfixe et du parcours infixe.

Exercice 4 (Arbres homogènes quelconques)

Dans cet exercice, nous choisissons d'implémenter en OCaml le type "arbre homogène" par le type personnalisé suivant :

```
type 'a arbre = N of 'a * 'a arbre  list ;;
```

1. Définir en OCaml l'arbre `a_ex` suivant :



2. Écrire une fonction `nb_feuilles` qui renvoie le nombre de feuilles d'un arbre homogène quelconque.
3. Écrire une fonction `chemin` qui prend en arguments une étiquette `x` et un arbre homogène quelconque `a` et qui renvoie la liste des ancêtres de `x` de la racine de `a` jusqu'au noeud étiqueté par `x`.

   On pourra parcourir l'arbre en profondeur (pour un noeud donné, on visite d'abord les fils puis les frères) en définissant une fonction auxiliaire `aux` qui prend en entrée la liste des arbres encore à visiter.
