Dans ce TP, on manipulera des arbres binaires stricts de type « `'a abs` », des arbres binaires de type « `'a bin` » et des arbres généraux (c'est à dire avec des noeuds d'arité quelconque) de type « `'a gen` ». Notez que l'arbre vide peut être représenté avec le type « `'a bin` », mais pas avec « `'a abs` » et « `'a gen` ».

```
type 'a abs =
  | F_abs of 'a
  | N_abs of 'a * 'a abs * 'a abs;;
```

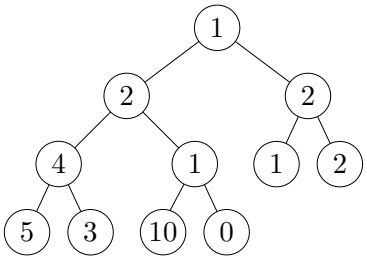
```
type 'a bin =
  | Vide
  | N_bin of 'a * 'a bin * 'a bin;;
```

```
type 'a gen = N_gen of 'a * 'a gen list;;
```

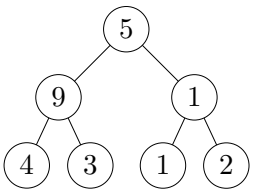
Exercice 1. Arbres binaires stricts

1. Rappeler la définition d'arbre "binaire" et d'arbre "binaire strict".

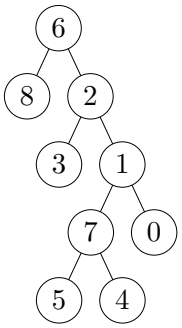
Soient `abs_A1`, `abs_A2` et `abs_A3` les arbres suivants :



`abs_A1`



`abs_A2`

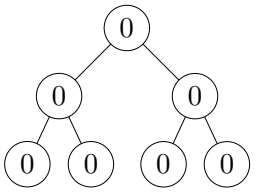


`abs_A3`

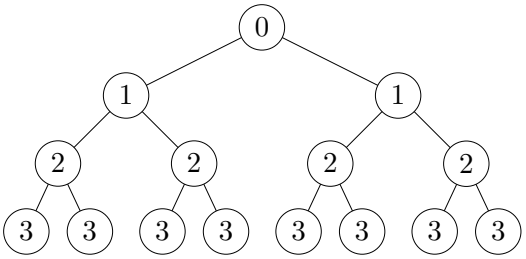
2. Définir `abs_A1`, `abs_A2` et `abs_A3` en OCaml. Les variables créées devront être de type « `int abs` ».
3. Écrire une fonction de type « `'a abs -> int` » qui renvoie le nombre de noeuds d'un arbre. Quelle est la complexité de la fonction ?

Arbre	`abs_A1`	`abs_A2`	`abs_A3`
Hauteur	3	2	4
Est complet ?	Non	Oui	Non
`fils_droit`	`[2; 1; 3; 0; 2]`	`[1; 3; 2]`	`[2; 1; 0; 4]`

4. Écrire une fonction de type « `'a abs -> int` » qui renvoie la hauteur d'un arbre. Voir les exemples dans le tableau ci-dessus. Quelle est la complexité de la fonction ?
5. Écrire une fonction de type « `'a abs -> bool` » qui indique si l'arbre donné en paramètre est complet. Le temps d'exécution devra être linéaire en le nombre de noeuds. Voir les exemples dans le tableau ci-dessus.
6. Écrire une fonction « `suppr_etiq: 'a abs -> int abs` » qui prend en entrée un arbre `abs_A`, et renvoie un arbre ayant le même squelette que `abs_A` mais où toutes les étiquettes ont été remplacées par 0. Par exemple, « `suppr_etiq abs_A2` » renvoie l'arbre ci-contre. Quelle est la complexité de la fonction ?



7. Écrire une fonction « `make_complet: int -> int abs` » qui prend en entrée un entier $n \in \mathbb{N}$ et renvoie un arbre complet de hauteur n où chaque feuille est étiquetée avec sa profondeur. Par exemple, « `make_complet 3` » renvoie l'arbre ci-contre. Quelle est la complexité de la fonction ?



8. Écrire une fonction « `fils_droit: 'a abs -> 'a list` » qui renvoie la liste des noeuds qui sont le fils droit de leur père (l'ordre des éléments de la liste n'a pas d'importance). Voir les exemples dans le tableau ci-dessus. Quelle est la complexité de la fonction ?
9. Écrire une fonction de type « `int abs -> unit` » qui affiche un arbre horizontalement dans la console. Si besoin, on pourra lire les indications ci-dessous. Par exemple, pour `abs_A1`, `abs_A2` et `abs_A3`, on obtient :

```
      /--2
   /--2
   |  \--1
---1
   |     /--0
   |  /--1
   |  |  \--10
   \--2
      |  /--3
      \--4
         \--5
```

```
      /--2
   /--1
   |  \--1
---5
   |  /--3
   \--9
      \--4
```

```
         /--0
      /--1
      |  |  /--4
      |  \--7
      |     \--5
   /--2
   |  \--3
---6
   \--8
```

Indications (essayez de répondre à la question sans lire ce qui suit). Pour chaque noeud u de l'arbre, notons s_u la chaîne de caractères à afficher juste avant le `"---"` ou le `"/--"` ou le `"\--"` associé à u. Par exemple dans `abs_A3` :

u	6	2	3	7	4
s_u	`""`	`"  "`	`"	"`	`"	"`	`"		"`

(a) Afin de comprendre comment évolue s_u, dessiner l'arbre `abs_A1` en étiquetant chaque noeud u par s_u.

(b) Pour afficher un arbre A_0, on écrit une fonction intermédiaire ayant pour argument :

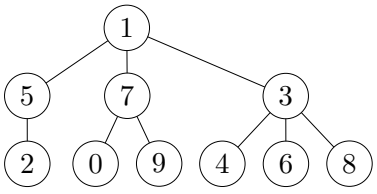
→ Un arbre A qui est un sous-arbre de A_0. Dans la suite, on note u la racine de A.

→ La chaîne de caractères s_u.

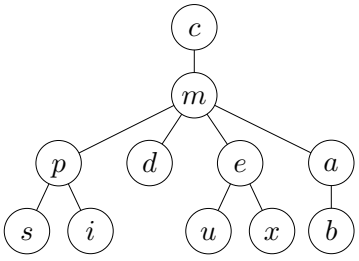
→ Un entier $t \in \{0, 1, 2\}$ tel que $t = 0$ si u est la racine de A_0, $t = 1$ si u est un fils gauche dans A_0 et $t = 2$ si u est un fils droit dans A_0.

Exercice 2. Arbres avec noeuds d'arité quelconque

Soient `gen_A1` et `gen_A2` les arbres suivants :

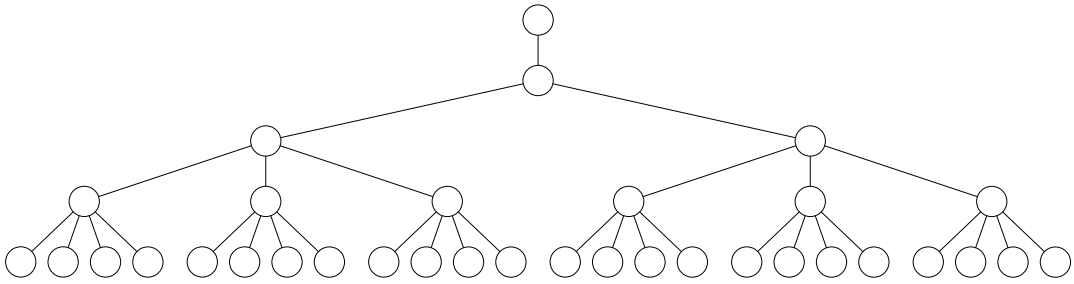


`gen_A1`



`gen_A2`

1. Définir les arbres `gen_A1` et `gen_A2` en OCaml.
2. Écrire une fonction de type « `int -> unit gen` » qui prend en entrée un entier $h \in \mathbb{N}$, et renvoie un arbre de hauteur h tel que chaque noeud de profondeur $p \in [\![0, h-1]\!]$ possède $p+1$ fils. Par exemple, avec $h = 4$, votre fonction doit renvoyer l'arbre :



Quelle est la complexité de la fonction ?

3. Écrire une fonction « `trouver_chemin 'a gen -> 'a -> 'a list` » qui prend en entrée un arbre et une étiquette `etiq`, et renvoie la liste représentant le chemin de la racine au noeud d'étiquette `etiq`. Si l'étiquette n'appartient pas à l'arbre, votre fonction renverra `[]`. Par exemple :

Arbre	`gen_A1`	`gen_A1`	`gen_A1`	`gen_A2`	`gen_A2`
`etiq`	`9`	`3`	`10`	`'c'`	`'u'`
`fils_droit`	`[1; 7; 9]`	`[1; 3]`	`[]`	`['c']`	`['c'; 'm'; 'e'; 'u']`

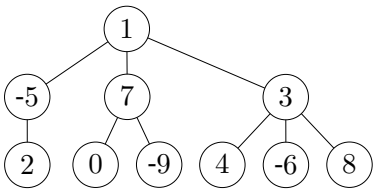
Quelle est la complexité de la fonction ?

Exercice 3. Fonctionnelles itératives sur les arbres.

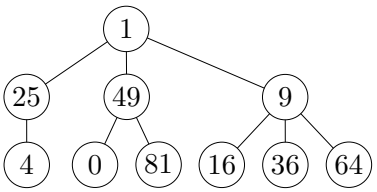
La fonction « `List.map: ('a -> 'b) -> 'a list -> 'b list` » du module `List` prend en entrée une fonction « `f: 'a -> 'b` » ainsi qu'une liste `[a1; a2; ...; an]`, et renvoie `[f a1; f a2; ...; f an]`. Par exemple :

« `List.map (fun x -> x*x) [1; 2; 3; 4]` » vaut `[1; 4; 9; 16]`.

On souhaite écrire une fonction « `arb_map: ('a -> 'b) -> 'a gen -> 'b gen` » en suivant le même principe. La fonction `arb_map` prend donc en entrée « `f: 'a -> 'b` » ainsi qu'un arbre `gen_A`, et applique la fonction `f` sur chacune des étiquettes de `gen_A`. Par exemple, à partir de `f` : $x \mapsto x^2$ et `gen_A1`, on obtient `gen_A2` :

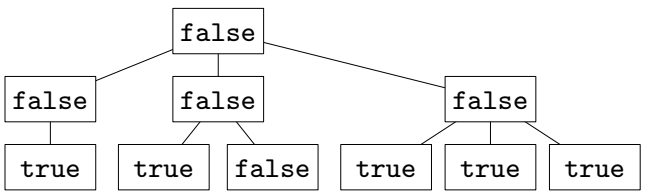


`gen_A1`



`gen_A2`

1. (a) Écrire la fonction « `arb_map: ('a -> 'b) -> 'a gen -> 'b gen` ».
   (b) En déduire une fonction « `paire: int gen -> bool gen` » qui prend en entrée un arbre `gen_A` et renvoie un arbre `gen_B` dont les étiquettes indiquent si l'étiquette correspondante dans `gen_A` est paire ou non. Par exemple, à partir de `gen_A1`, on obtient :



La fonction « `List.iter: ('a -> unit) -> 'a list -> unit` » du module `List` prend en entrée une fonction « `f: 'a -> unit` » ainsi qu'une liste `[a1; a2; ...; an]`, et exécute la suite d'instructions « `f a1; f a2; ...; f an` ». Par exemple :

« `List.iter print_int [1; 4; 9; 16]` » affiche `14916`.

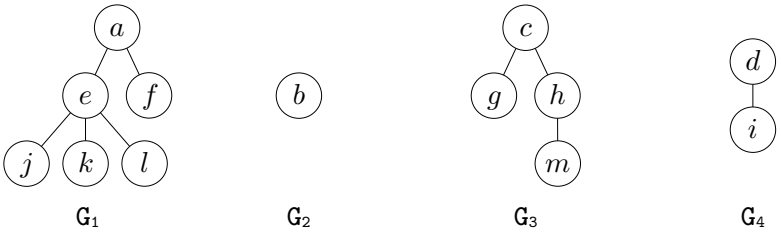
On souhaite écrire une fonction « `arb_iter: ('a -> unit) -> 'a gen -> unit` » en suivant le même principe. La fonction `arb_iter` prend donc en entrée « `f: 'a -> unit` » ainsi qu'un arbre `gen_A`, et exécute la suite d'instructions « `f` $\texttt{e}_1$`; f` $\texttt{e}_2$`; ...; f` $\texttt{e}_n$ » où les $\texttt{e}_i$ sont les étiquettes de `gen_A` données dans un ordre arbitraire.

2. (a) Écrire la fonction « `arb_iter: ('a -> unit) -> 'a gen -> unit` ».
   (b) En déduire une fonction « `somme: int gen -> int` » qui renvoie la somme des étiquettes d'un arbre.

Exercice 4. Bijection entre arbres binaires et forêts d'arité quelconque

Une ***forêt d'arité quelconque*** (appelée plus simplement ***forêt*** dans la suite) est une liste dont les éléments sont des arbres d'arité quelconque. En OCaml, les forêts sont représentées par le type « `'a gen list` ». On s'intéresse à une bijection φ entre l'ensemble des forêts et l'ensemble des arbres binaires.

Soit `li` une forêt de la forme `li = [`$\texttt{G}_1$`; `$\texttt{G}_2$`; ...; `$\texttt{G}_n$`]` où les $\texttt{G}_i$ sont de type « `'a gen` ». Pour illustrer les explications qui suivent on utilisera l'exemple $\texttt{li}_0$ `= [`$\texttt{G}_1$`; `$\texttt{G}_2$`; `$\texttt{G}_3$`; `$\texttt{G}_4$`]` avec les arbres suivants :



Soient u et v deux noeuds présents dans les arbres de `li`. On dit que "v est le frère cadet de u" si l'une des deux conditions suivantes est vérifiée :

→ u est la racine de l'un des $\texttt{G}_i$ et v est la racine de $\texttt{G}_{i+1}$.

→ u et v ont le même père et v est le noeud situé directement à droite de u.

Par exemple dans $\texttt{li}_0$, voici tous les couples (e_u, e_v) où e_u, e_v sont les étiquettes de noeuds u, v tels que v est le frère cadet de u : (a, b), (b, c), (c, d), (e, f), (g, h), (j, k) et (k, l).

Soit u un noeud présent dans les arbres de `li`. On dit que "u est un frère aîné" s'il n'est le frère cadet d'aucun autre noeud. Dans $\texttt{li}_0$, les étiquettes des frères aînés sont donc : a, e, g, i, j et m.

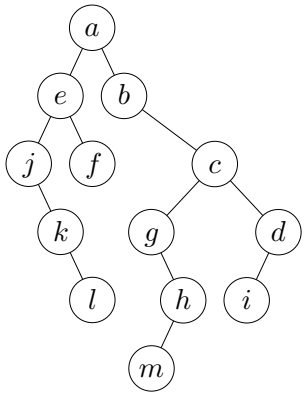
Soit `li` une forêt de type « `'a gen list` », alors l'image de `li` par la bijection φ est l'arbre `bin_B` de type « `'a bin` » tel que :

→ `bin_B` et `li` contiennent le même nombre de noeuds et les mêmes étiquettes. À chaque noeud u de `li` correspond donc un noeud de `bin_B` noté u' dans la suite (u et u' ont la même étiquette).

→ Soit u un noeud de `li`. Si u n'a pas de fils dans `li`, alors u' n'a pas de fils gauche dans `bin_B`. Sinon, soit v le frère aîné parmi les fils de u ; alors le fils gauche de u' est v'.

→ Soit u un noeud de `li`. Si u n'a pas de frère cadet dans `li`, alors u' n'a pas de fils droit dans `bin_B`. Sinon, soit v le frère cadet de u ; alors le fils droit de u' est v'.

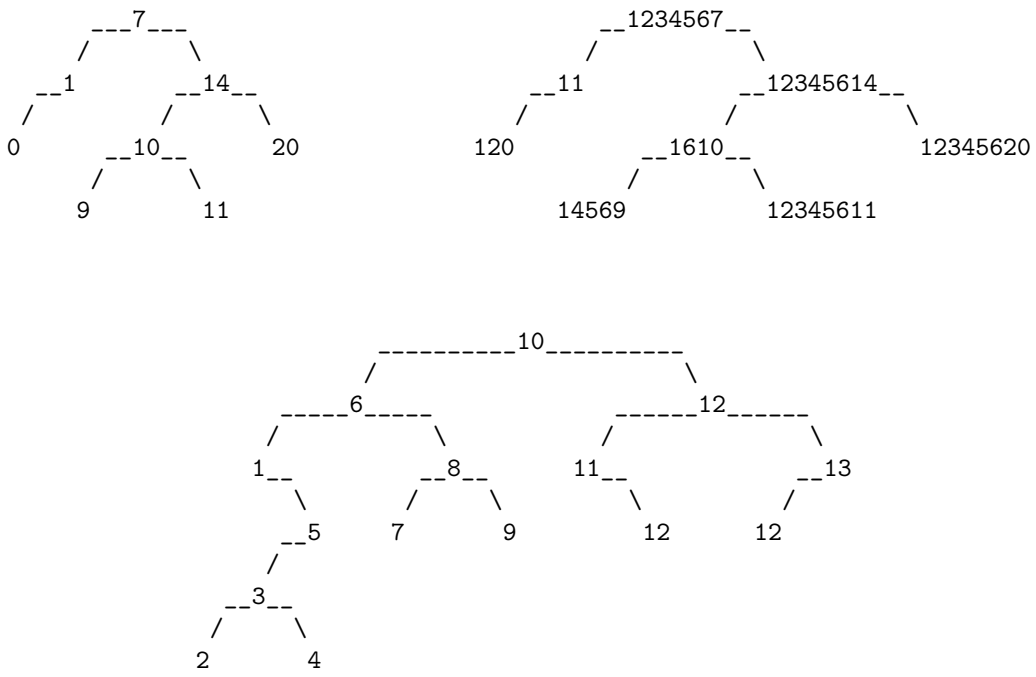
Par exemple, l'image de $\texttt{li}_0$ par φ est donnée ci-dessous :



1. Écrire une fonction « `bin_of_gen: 'a gen list -> 'a bin` » qui transforme une forêt en un arbre binaire en suivant la bijection φ. Notez que malgré la longueur de l'énoncé, la fonction `bin_of_gen` est très simple (environ 3 lignes).
2. Écrire une fonction « `gen_of_bin: 'a bin -> 'a gen list` » qui réalise la transformation réciproque de la fonction `bin_of_gen`. Même remarque que pour la question précédente.

Exercice 5. Affichage d'arbres dans la console

Écrire une fonction de type « `int bin -> unit` » qui affiche un arbre binaire verticalement dans la console :



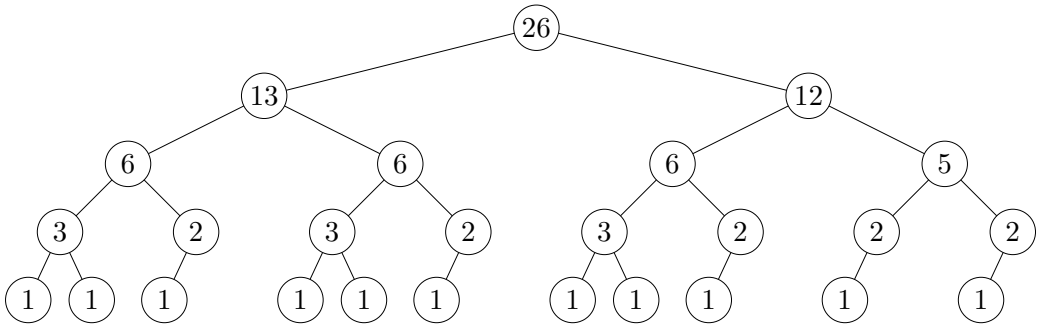
Exercices à rendre au plus tard le 15/05/2025 à 20h

Exercice 6. Recherche d'une étiquette dans un arbre binaire

Écrire une fonction « `bin_mem: 'a -> 'a bin -> bool` » qui prend en entrée « `x: 'a` » ainsi que « `a: 'a bin` », et indique si `x` est une étiquette de `a`.

Exercice 7. Arbres binaires parfaitement équilibrés

Soit A un arbre binaire. On dit que A est ***parfaitement équilibré*** si pour tout noeud u de A, si on note n_G (resp. n_D) le nombre de noeuds dans le sous-arbre gauche (resp. droit) de u, alors $n_G \in \{n_D, n_D + 1\}$. On admet que pour tout $n \in \mathbb{N}$, il existe un unique arbre parfaitement équilibré avec n noeuds. Par exemple, pour $n = 26$:



Sur ce schéma, l'étiquette d'un noeud u est le nombre de noeuds dans le sous-arbre dont la racine est u.

1. Écrire une fonction « `parf_eq: int -> int bin` » qui prend en entrée $n \in \mathbb{N}$ et renvoie un arbre parfaitement équilibré avec n noeuds.

Exercice 8. Arbres d'arité quelconque

Pour les tests, on pourra utiliser les arbres de l'exercice 2.

Lorsqu'on écrit une fonction qui prend en entrée un arbre « `gen_A: 'a gen` », une astuce classique consiste à commencer par écrire une fonction `aux` qui prend en entrée une liste de type « `'a gen list` », puis à appliquer `aux` sur `[gen_A]`. Par exemple, pour calculer la somme des étiquettes d'un arbre, on écrit d'abord une fonction `aux` qui prend en entrée une liste d'arbres $\texttt{li} = [\texttt{G}_1; \texttt{G}_2; \ldots; \texttt{G}_n]$, et renvoie $\texttt{s}_1 + \texttt{s}_2 + \ldots + \texttt{s}_n$ où $\texttt{s}_i$ est la somme des étiquettes de $\texttt{G}_i$.

```
let somme (gen_A: int gen): int =
  let rec aux (li: int gen list): int = match li with
    | [] -> 0
    | (N_gen (e, fils)) :: q -> e + (aux fils) + (aux q)
  in
  aux [gen_A];;
```

1. Écrire une fonction `nb_noeuds` qui calcule le nombre de noeuds dans un arbre de type « `'a gen` ». Vous devez définir une fonction auxiliaire en précisant ce qu'elle renvoie.
2. Écrire une fonction `hauteur` qui calcule la hauteur d'un arbre de type « `'a gen` ». Vous devez définir une fonction auxiliaire en précisant ce qu'elle renvoie.
3. Écrire une fonction `est_strict` qui indique si l'arbre « `gen_A: 'a gen` » donné en entrée est un arbre binaire strict (c'est à dire si tous les noeuds sont d'arité 0 ou 2). Vous devez définir une fonction auxiliaire en précisant ce qu'elle renvoie.

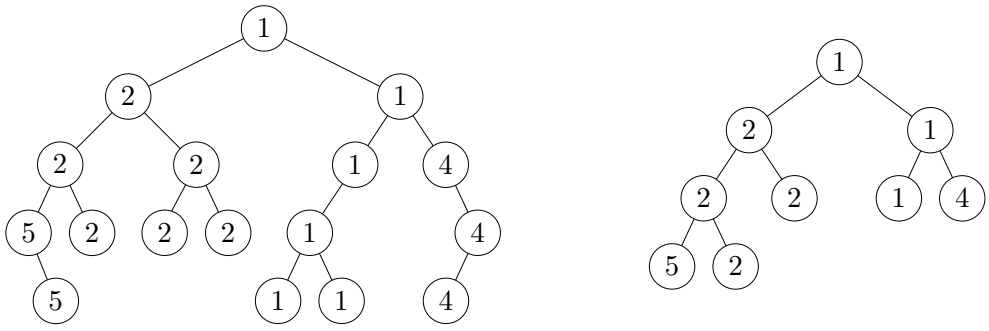
Exercice 9. Réduction d'arbres binaires (facultatif)

Soit A un arbre, u un noeud interne de A et A_u le sous-arbre de A dont la racine est u. On suppose que toutes les étiquettes de A_u sont égales à la même valeur x. Soit B l'arbre obtenu à partir de A en remplaçant A_u par un unique noeud étiqueté par x. Cette opération qui permet de passer de A à B s'appelle une ***réduction***. On dit qu'un arbre R est ***l'arbre réduit*** associé à A si :

→ On peut passer de A à R par une suite de réductions.

→ R ne peut plus être réduit.

Dans la suite, on admet que l'arbre réduit associé à A est unique. Par exemple :



A R

1. Écrire une fonction « `reduire: 'a bin -> 'a bin` » qui prend en entrée un arbre A et renvoie l'arbre réduit associé. Votre fonction devra être de complexité linéaire en le nombre de noeuds dans A.