TP 10 : Arbres en OCaml et en C

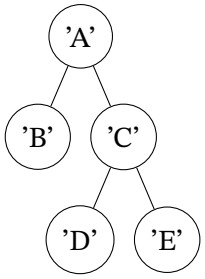
On commence par manipuler simplement des arbres en OCaml. On va commencer par les arbres binaires non stricts, puis on pourra passer aux arbres binaires stricts et on finit par les arbres généraux.

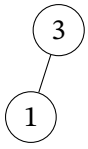
Exercice 1 – Arbres binaires non stricts en OCaml

On va utiliser le type suivant :

```OCaml
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre
```

▶ **Question 1** Représenter les arbres suivants avec ce type :





- un arbre vide,
- un arbre de hauteur 3 à 4 nœuds,
- un arbre de hauteur 3 à 15 nœuds (réfléchir une seconde avant de faire un exemple à 15 lignes),
- ★ un arbre de hauteur 11 à 4095 nœuds (réfléchir une seconde avant de faire un exemple à 4095 lignes),

▶ **Question 2** Tester vos fonctions `hauteur taille liste_sous_arbres` sur ces exemples et vérifier que le résultat correspond à vos attentes.

▶ **Question 3** Écrire une fonction `est_feuille : 'a arbre -> bool` vérifiant si un nœud est une feuille. *On s'interdira d'utiliser cette fonction dans la suite, mais on notera et on chérira la beauté du filtrage par motif que l'on y utilise.*

▶ **Question 4** Écrire une fonction `nb_feuilles : 'a arbre -> int` qui renvoie le nombre de feuilles d'un arbre.

▶ **Question 5** Écrire une fonction `somme_etiquettes : int arbre -> int` qui renvoie la somme des étiquettes des nœuds de l'arbre.

▶ **Question 6** Écrire une fonction `somme_feuilles : int arbre -> int` qui renvoie la somme des étiquettes de toutes les feuilles.

▶ **Question 7** Écrire une fonction `parcours : 'a arbre -> 'a list` qui renvoie la liste des étiquettes d'un arbre, dans l'ordre que vous voulez, mais une et une seule fois chacune. On pourra utiliser l'opérateur `@`.

★ **Question 8** Écrire une fonction `max_min_arbre : int arbre -> int * int` qui renvoie le minimum et le maximum des étiquettes d'un arbre. Chaque nœud ne devra être parcouru qu'une seule fois. On lèvera une exception si l'arbre est vide. Attention : il ne doit y avoir, lors d'un appel de la fonction, qu'au plus un appel récursif sur chaque fils.

★ **Question 9** Une branche est un chemin de la racine vers une feuille. Le poids d'une branche est la somme des étiquettes des nœuds qui la composent. Écrire une fonction `max_somme_branche : int arbre -> int` qui renvoie le poids de la branche de poids maximal.

On considère maintenant le type :

```OCaml
type 'a arbre_strict =
  | Feuille of 'a
  | Noeud_Interne of 'a arbre_strict * 'a * 'a arbre_strict
```

On remarque qu'il n'y a plus la possibilité de définir un arbre vide. Une feuille est maintenant un arbre qui est une `Feuille`. La taille d'un arbre est le nombre total de nœuds (nœuds internes et feuilles). Les autres définitions sont inchangées.

▶ **Question 10** Certains des arbres définis à la section précédente ne peuvent plus l'être avec ce nouveau type. Lesquels ? Pourquoi ?

★ **Question 11** Reprendre toutes les questions de la section précédente avec ce nouveau type.

★ **Question 12** *Quand on aura parlé de parcours d'arbres*, implémenter les parcours d'arbres pour ce nouveau type.

Exercice 2 – Arbres généraux et forêts

Pour ne pas m'embêter, je réutilise le nom de type `arbre` *ici. Pour être plus propre, je vous conseille de commencer un nouveau fichier ici.*
Dans cet exercice, on s'intéresse aux nœuds d'arité quelconque, avec un point de vue légèrement différent du cours. Pour cela, on introduit la notion de *forêt* :

Définition 1

Une *forêt* est un ensemble fini (éventuellement vide) d'arbres non vides sans nœuds communs.

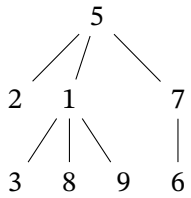
On peut donc voir les fils d'un nœud d'un arbre général comme une forêt. En OCaml on peut aussi définir les arbres généraux ainsi :

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

On peut vouloir donner un nom plus explicite à une liste d'arbres qui est une forêt ! Mais alors pour définir une forêt, il faut savoir définir un arbre et réciproquement…De la même manière que l'on peut définir des fonctions mutuellement récursives, on peut définir des types mutuellement récursifs :

OCaml
```
type 'a arbre = Vide | Noeud of 'a * 'a foret
  and 'a foret = 'a arbre list
(* comme au DS ! *)
```

▶ **Question 1** Implémenter l'arbre suivant :



Les notions de taille et profondeur existent toujours sur les arbres généraux. On peut les programmer en OCaml à l'aide de deux fonctions mutuellement récursives :

```OCaml
let rec taille arbre =
  match arbre with
  | Vide -> 0
  | Noeud (_, fils) -> 1 + taille_foret fils
and taille_foret foret =
  match foret with
  | [] -> 0
  | arbre :: autres -> (taille arbre) + (taille_foret autres)
```

▶ **Question 2** Écrire la fonction `hauteur : 'a arbre -> int`. On pourra définir la fonction mutuellement récursive `max_hauteur_foret : 'a arbre list -> int`.

▶ **Question 3** Écrire une fonction `degre : 'a arbre -> int` qui donne le degré d'un arbre, c'est-à-dire le plus grand degré de ses nœuds.

Si on suppose que l'on n'a pas besoin de l'arbre vide (ce qui est assez naturel dans certaines études) on peut se passer du constructeur de type. Pour bien insister sur le fait que l'on manipule un arbre et non un couple quelconque, on propose le type suivant :

```OCaml
type 'a arbre_general = {etiquette : 'a; fils : 'a arbre_general list}
```

On aurait pu aussi utiliser un type somme avec un unique constructeur paramétré, comme on l'a mentionné dans le cours.

▶ **Question 4** Réécrire les fonctions `taille` et `hauteur` avec ce nouveau type.

▶ **Question 5** Peut-on écrire une fonction qui renverse l'ordre des fils de chaque nœud d'un arbre (du *même* arbre, sans en recréer un nouveau)? Pourquoi?

▶ **Question 6** Écrire une fonction `nb_feuilles : 'a arbre_general -> int` qui renvoie le nombre de feuilles d'un arbre.

▶ **Question 7** Écrire une fonction `max_etiquettes : 'a arbre_general -> 'a` qui renvoie le maximum des étiquettes des nœuds de l'arbre.

▶ **Question 8** Adapter la fonction `max_somme_branche : int arbre_general -> int` qui renvoie le poids de la branche de poids maximal. *Une branche est un chemin allant de la racine jusqu'à une feuille.*

Exercice 3 – Manipulation d'arbres binaires en C

Pour rappel, on peut manipuler des arbres binaires en C grâce un type enregistrement de la forme :

```C
typedef struct arbre {
  int valeur;
  struct arbre* gauche;
  struct arbre* droit;
} arbre;
```

Comme pour les listes chaînées, on manipulera toujours des arbres sous forme de pointeur de type `arbre*`. Ainsi, un arbre vide sera représenté par `NULL`. Pour commencer à tester ses fonctions, on peut retirer temporairement l'option `-fsanitize=address` de la commande de compilation.

▶ **Question 1** Quels arbres sont représentables par ce type?

▶ **Question 2** Écrire une fonction `arbre* construire_arbre(int valeur, arbre* gauche,`

`arbre* droit)` qui renvoie un arbre binaire de racine `valeur` et de fils gauche gauche et droit `droit`.

▶ **Question 3** Écrire une fonction `int hauteur(arbre* a)` qui renvoie la hauteur de l'arbre `t`. De même, écrire une fonction `int taille(arbre* a)` qui renvoie le nombre de nœuds de l'arbre `t`.

▶ **Question 4** Écrire une fonction `int est_feuille(arbre* a)` qui renvoie 1 si `t` est une feuille et 0 sinon.

▶ **Question 5** Écrire une fonction `int nb_feuilles(arbre* a)` qui renvoie le nombre de feuilles de l'arbre `t`.

▶ **Question 6** Écrire une fonction `int somme_feuilles(arbre* a)` qui renvoie la somme des étiquettes des feuilles de l'arbre `t`.

▶ **Question 7** Écrire une fonction `bool est_strict(arbre* a)` qui renvoie `true` si `t` est un arbre binaire strict et `false` sinon.

▶ **Question 8** Écrire une fonction `bool est_dans_arbre(int valeur, arbre* a)` qui renvoie `true` si `valeur` est dans l'arbre `t` et `false` sinon.

Évidemment, l'intérêt d'une telle structure de donnée est de pouvoir travailler en place, contrairement aux types que l'on utilise habituellement sur les arbres en OCaml. Par exemple :

▶ **Question 9** Écrire une fonction `void ajoute_valeur(int valeur, arbre* a)` qui ajoute `valeur` à l'étiquette de chaque nœud de l'arbre `t`.

▶ **Question 10** Écrire une fonction `void miroir(arbre* a)` qui modifie en place l'arbre `t` en échangeant les fils gauches et droits de chaque nœud.

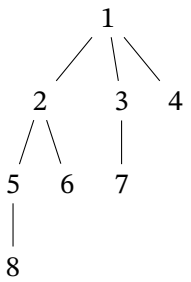
▶ **Question 11** Écrire une fonction `void liberer(arbre* a)` qui libère la mémoire allouée pour l'arbre `t`.

Exercice 4 – Représentation des arbres généraux en C

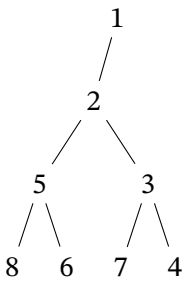
Évidemment, on peut adapter la représentation des arbres en C en adaptant le type de liste chaînée vu en cours et TP en C. Il existe une autre représentation plus efficace en mémoire qui consiste à transformer un arbre général en un arbre binaire. Pour cela, on utilise le type suivant :

C
```c
typedef struct arbre_general {
  int valeur;
  struct arbre_general* fils;
  struct arbre_general* frere;
} arbre_general;
```

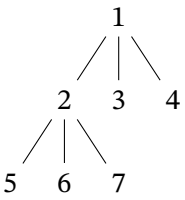
Chaque nœud de l'arbre général est représenté par un nœud de l'arbre binaire. Le fils gauche de ce nœud est son premier fils dans l'arbre général (`NULL` s'il n'a aucun fils), et son fils droit est son frère dans l'arbre général [a]. Par exemple, on aura :



représenté par l'arbre binaire :



▶ **Question 1** Tracer la représentation en arbre binaire de l'arbre général suivant :



▶ **Question 2** Écrire une fonction `arbre_general* feuille(int valeur)` qui renvoie un arbre général de racine `valeur` et sans fils.

▶ **Question 3** Écrire une fonction `void ajoute_frere(arbre_general* a, arbre_general* fils)` ajoutant comme premier fils de `a` l'arbre général `fils`. On vérifiera que `fils` n'a pas de frère avant de l'ajouter.

▶ **Question 4** Parmi les fonctions écrites pour les arbres binaires dans l'Exercice 3, lesquelles sont directement adaptables pour ce type d'arbres généraux ?

★ **Question 5** Implémenter les autres fonctions de l'Exercice 3 pour ce type d'arbres.

★ **Question 6** Écrire une fonction `arbre_general* convertir(arbre* a)` qui renvoie l'arbre général correspondant à l'arbre binaire `t`.

a. Cette représentation est appelée *LCRS* en anglais, pour *Left-Child Right-Sibling*.

Exercice 5 – ★ Arbres à trois pointeurs

Dans les Exercices 3 et 4, on a vu deux représentations des arbres en C identiques en mémoire (une valeur et deux pointeurs par noeud) permettant, selon leur interprétation, de représenter les arbres binaires ou les arbres généraux. Dans les deux cas, on peut compléter les types avec un troisième pointeur indiquant l'adresse du père du noeud en mémoire (le père de la racine étant `NULL`). On obtient alors les types suivants pour les arbres binaires :

C
```c
typedef struct arbre_avec_pere {
  int valeur;
  struct arbre_avec_pere* gauche;
  struct arbre_avec_pere* droit;
  struct arbre_avec_pere* pere;
} arbre_avec_pere;
```

et pour les arbres généraux :

```c
typedef struct arbre_general_avec_pere {
  int valeur;
  struct arbre_general_avec_pere* fils;
  struct arbre_general_avec_pere* frere;
  struct arbre_general_avec_pere* pere;
} arbre_general_avec_pere;
```

L'un des avantages de cette représentation est de permettre de parcourir l'arbre purement itérativement, sans utiliser de pile.

▶ **Question 1** Écrire une fonction `int taille(arbre_avec_pere* t)` qui renvoie la taille de l'arbre `t` en espace constant.

▶ **Question 2** Écrire une fonction `int hauteur(arbre_avec_pere* t)` qui renvoie la hauteur de l'arbre `t` en espace constant.

▶ **Question 3** Écrire une fonction `void parcours_prefixe(arbre_avec_pere* t)` qui affiche les étiquettes des nœuds de l'arbre `t` en parcours préfixe en espace constant.

Évidemment, comparé à la représentation à deux pointeurs par nœud, ce pointeur supplémentaire par nœud peut être considéré comme un espace auxiliaire en $\mathcal{O}(n(A))$, ce qui est identique à la complexité mémoire de la pile d'appel utilisée pour les parcours récursifs. La constante associée sera cependant plus faible. De plus, cette représentation permet de manipuler des nœuds à l'intérieur d'un arbre sans oublier leur contexte (le fait qu'ils ne sont qu'un sous-noeud d'un plus grand arbre), alors que les représentations précédentes permettaient de raisonner sur chaque nœud comme un sous-arbre enraciné en ce nœud.