Programmation Fonctionnelle : TD5

EFREI-Paris
Arbres binaires

*
* *

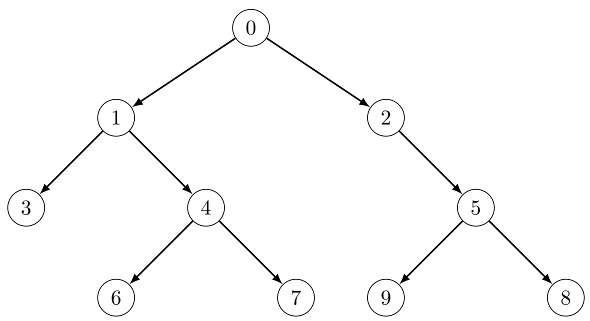
Pour chaque fonction, on donnera à la fois le code Ocaml les spécifications fonctionnelles / signatures nécessaires.

Préalables

On donne la définition du type `arbreBin` suivant en Ocaml:

```
type 'a arbreBin = Vide | Noeud of 'a arbreBin * 'a * 'a arbreBin | Feuille of 'a;;
```

Créer l'arbre binaire a_1 à gauche et dessiner l'arbre binaire a_2 dont le code est donné ci-dessous:



```
let a2 = Noeud(
            Noeud(
                Feuille 2,
                4,
                Feuille 8
            ),
            16,
            Feuille 32
        );;
```

puis implémenter les fonctions suivantes:

- `taille` qui retourne le nombre d'étiquettes, c'est-à-dire le nombre de noeuds et de feuilles, d'un arbre binaire a.

  `taille a1;;` $\Longrightarrow$ `10`

- `hauteur` qui retourne le nombre d'étages d'un arbre binaire a.

  `hauteur a1;;` $\Longrightarrow$ `4`
  `hauteur a2;;` $\Longrightarrow$ `3`

- `recherche` qui teste si un élément $e \in a$.

  `recherche 10 a1;;` $\Longrightarrow$ `false`

- `equilibre` qui teste si un arbre binaire a maintient une profondeur équilibrée entre ses branches. On tolère que la branche la plus à gauche soit 1 étage plus grande que les autres de l'arbre.

  $$\forall n = (g, x, d) \in a, 0 \leq hauteur(g) - hauteur(d) \leq 1$$

  Où g et d sont respectivement l'arbre gauche et l'arbre droit du noeud n, et x l'étiquette de n.

  `equilibre a1;;` $\Longrightarrow$ `false`
  `equilibre a2;;` $\Longrightarrow$ `true`

Problème 1

1. Écrire une fonction `somme` qui retourne la somme d'un arbre binaire a d'entiers. Écrire également une version récursive terminale `somme_term` de cette fonction.

   `somme a1;;` $\Longrightarrow$ `45`

2. Écrire une fonction `max_a` qui retourne l'élément maximal présent dans un arbre binaire a d'entiers.

   `max_a a1;;` $\Longrightarrow$ `9`

3. Écrire une fonction `complet` qui retourne vrai si un arbre binaire a est complet et faux sinon.

   Un arbre binaire est dit *complet* si et seulement si tous ses noeuds internes sont de degré deux.

   `complet a1;;` $\Longrightarrow$ `false`
   `complet a2;;` $\Longrightarrow$ `true`

4. Écrire une fonction `parfait` qui retourne vrai si un arbre binaire a est parfait et faux sinon.

   Un arbre binaire est dit *parfait* si et seulement si il est complet et équilibré.

   `parfait a1;;` $\Longrightarrow$ `false`
   `parfait a2;;` $\Longrightarrow$ `true`

5. Écrire une fonction `miroir` qui retourne l'arbre binaire miroir d'un arbre binaire a.

   `miroir a1;;` $\Longrightarrow$ `Noeud (Noeud (Noeud (Feuille 8, 5, Feuille 9), 2, Vide), 0, Noeud (Noeud (Feuille 7, 4, Feuille 6), 1, Feuille 3))`

Problème 2

Un *parcours* d'arbre binaire est un algorithme qui permet de visiter tous les noeuds d'un arbre.
Il existe cependant plusieurs façons de visiter un arbre, on en distingue ici trois différentes qui sont représentées ci-dessous:

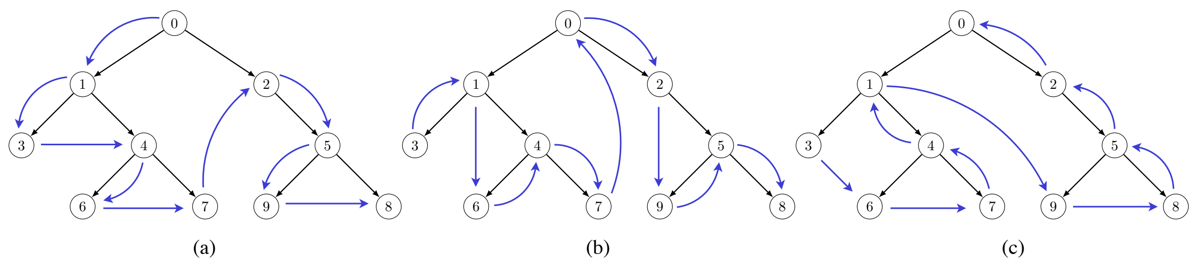


Figure 1: (a) Parcours prefixe (b) Parcours infixe (c) Parcours postfixe

Ces différents parcours permettent, entre autre, de transformer artificiellement un arbre binaire en une liste d'élément. On donne le pseudo-code des trois algorithmes de parcours : *prefixe*, *infixe* et *postfixe*.

Algorithme 1 : $prefixe(a)$

Data : Arbre binaire a

begin
- **switch** a **do**
  - `Vide` $\rightarrow$ `[V]`
  - `Feuille` $f \rightarrow$ `[F` f`]`
  - `Noeud` $(g, x, d) \rightarrow$
    `[N` x`] @`
    $prefixe(g)$ `@`
    $prefixe(d)$;;

Algorithme 2 : $infixe(a)$

Data : Arbre binaire a

begin
- **switch** a **do**
  - `Vide` $\rightarrow$ `[V]`
  - `Feuille` $f \rightarrow$ `[F` f`]`
  - `Noeud` $(g, x, d) \rightarrow$
    $infixe(g)$ `@`
    `[N` x`] @`
    $infixe(d)$;;

Algorithme 3 : $postfixe(a)$

Data : Arbre binaire a

begin
- **switch** a **do**
  - `Vide` $\rightarrow$ `[V]`
  - `Feuille` $f \rightarrow$ `[F` f`]`
  - `Noeud` $(g, x, d) \rightarrow$
    $postfixe(g)$ `@`
    $postfixe(d)$ `@`
    `[N` x`]` ;;

1. On considère le type `elem_arbre` qui indique le type d'une étiquette d'un arbre, `N` pour noeud, `F` pour feuille et `V` pour vide:

   ```
   type 'a elem_arbre = N of 'a | F of 'a | V
   ```

   Implémenter les trois méthodes de parcours ci-dessus.

   `prefixe a1;;` $\Longrightarrow$ `[N 0; N 1; F 3; N 4; F 6; F 7; N 2; V; N 5; F 9; F 8]`

2. Comme on l'a vue, il est possible de ranger un arbre binaire dans une liste. De cette façon, il est tout à fait possible de générer un arbre binaire à partir de la donnée de son parcours préfixe.

   (a) Deux arbres distincts peuvent-ils avoir le même parcours infixe ? Justifier par un contre-exemple dans le cas où l'assertion est fausse ; la démontrer si elle est vraie.

   (b) (FACULTATIF) Écrire une fonction `list_to_arbreBin` qui retourne un arbre binaire issu d'un parcours prefixe stocké dans une liste l.

   ```
   listPrefixe_to_arbreBin [N 16; N 4; F 2; F 8; F 32];;
                ⟹ Noeud (Noeud (Feuille 2, 4, Feuille 8), 16, Feuille 32)
   ```

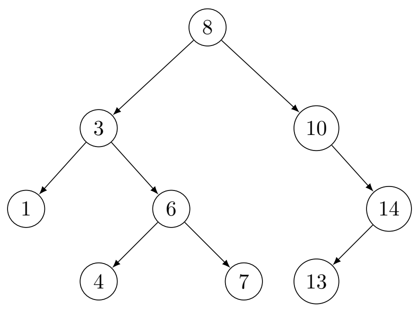
Problème 3

Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante :

Soit a, un arbre binaire, alors:

$$\forall n = (g, x, d) \in a, \forall i \in g, \forall j \in d | i < x \wedge j > x$$

c'est-à-dire que les étiquettes apparaissant dans le sous-arbre gauche g sont strictement inférieures à l'étiquette x et celles du sous-arbre droit d sont strictement supérieures.



On souhaite généralement disposer de trois opérations « primitives » sur de telles structures : la recherche (tester si un élément e est présent ou non dans a), l'ajout d'un élément et la suppression. Ces arbres permettent ainsi de représenter des ensembles en effectuant les opérations ensemblistes usuelles de manière relativement efficace.

1. Implémenter l'arbre binaire de recherche a_3 ci-dessus.

2. Écrire une fonction une fonction `est_de_recherche` qui teste si un arbre binaire a respecte la propriété des arbres binaires de recherche.

   ```
   est_de_recherche a1;; ⟹ false
   est_de_recherche a2;; ⟹ true
   ```

3. Écrire le code d'une fonction `recherche2` qui teste si un élément $e \in a$. Quelle est la complexité de cette méthode par rapport à la première que vous avez écrite dans la partie **Préalables** ?

4. Écrire une fonction `add` qui ajoute un élément e à un arbre binaire de recherche a. On supposera que $e \notin a$.

   ```
   add 1 a2;; ⟹ Noeud (Noeud (Noeud (Feuille 1, 2, Vide), 4, Feuille 8), 16, Feuille 32)
   add 11 a3;;  ⟹ Noeud (Noeud (Feuille 1, 3, Noeud (Feuille 4, 6, Feuille 7)), 8,
                  Noeud (Feuille 9, 10, Noeud (Feuille 13, 14, Vide)))
   ```

5. (Facultatif) Écrire une fonction `remove` qui supprime un élément e à un arbre binaire de recherche a. On supposera que $e \in a$.

   ```
   remove 8 a3 ⟹ Noeud (Noeud (Feuille 1, 3, Noeud (Feuille 4, 6, Vide)), 7,
                 Noeud (Vide, 10, Noeud (Feuille 13, 14, Vide)))
   ```