Programmation Fonctionnelle : TD5

Université de Tours

`Département informatique de Blois`

Arbres binaire

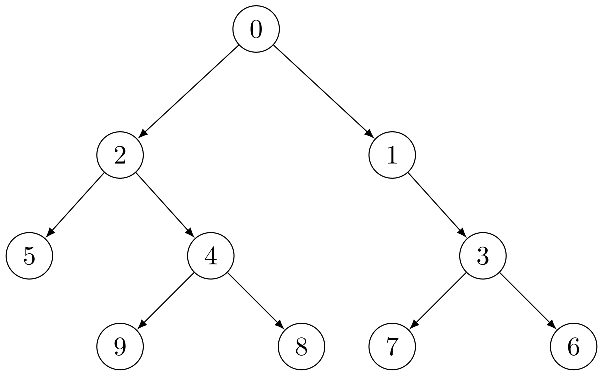
*
* *

Appropriation du cours

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

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

Créer l'arbre suivant :

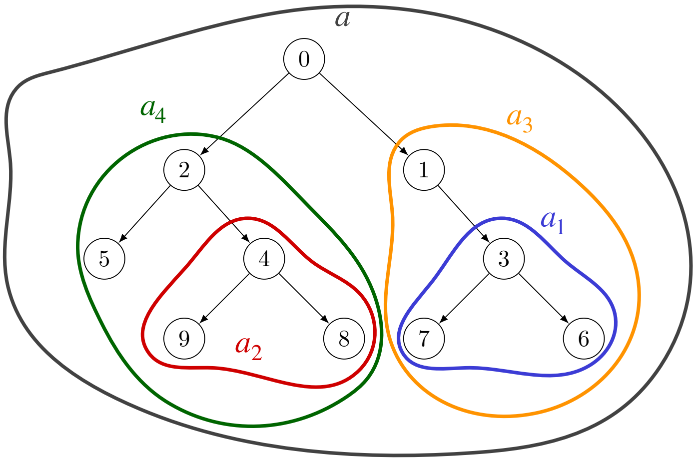


puis, ré-implémenter les fonctions du cours suivantes du cours :

- `taille a` qui retourne le nombre de noeuds d'un arbre binaire a.
- `hauteur a` qui retourne le nombre d'étages d'un arbre binaire a.
- `recherche e a` qui teste si l'élément e appartient à l'arbre a.
- `est_equilibre a` qui teste si un arbre binaire a maintient une profondeur équilibrée entre ses branches, c'est-à-dire que $\forall n = (g, x, d) \in a, |hauteur(g) - hauteur(d)| \leq 1$.

  Où g et d sont respectivement l'arbre gauche et l'arbre droit du noeud n et x l'étiquette du noeud n. (Cf. correction fin de la séance de cours).

```
let a1 = Noeud (Feuille 7, 3, Feuille 6);;
let a2 = Noeud (Feuille 9, 4, Feuille 8);;
let a3 = Noeud (Vide, 1, a1);;
let a4 = Noeud (Feuille 5, 2, a2);;
let a = Noeud (a4, 0, a3);;
```



$$taille : \begin{cases} \texttt{arbreBin} & \to \texttt{int} \\ a & \mapsto \text{nb noeuds de } a \end{cases}$$

```
let rec taille a = match a with
  Vide -> 0
| Feuille _ -> 1
| Noeud (g,x,d) -> 1 + taille g + taille d;;
```

$$hauteur : \begin{cases} \texttt{arbreBin} & \to \texttt{int} \\ a & \mapsto \text{nb étages de } a \end{cases}$$

```
let rec hauteur a = match a with
  Vide -> 0
| Feuille _ -> 1
| Noeud (g,x,d) -> 1 + max (hauteur g) (hauteur d);;
```

$$recherche : \begin{cases} \texttt{arbreBin} \times \texttt{'a} & \to \{true, false\} \\ a \times e & \mapsto \begin{cases} true & \text{si } e \in a \\ false & \text{sinon} \end{cases} \end{cases}$$

```
let rec appartient a e = match a with
  Vide -> false
| Feuille f -> e = f
| Noeud (g,x,d) -> if (e = x) then true
                   else (appartient g e) || (appartient d e);;
```

$$est\_equilibre : \begin{cases} \texttt{arbreBin} & \to \{true, false\} \\ a & \mapsto \begin{cases} true & \text{si } \forall n = (g,x,d) \in a, |hauteur(g) - hauteur(d)| \leq 1 \\ false & \text{sinon} \end{cases} \end{cases}$$

```
let rec est_equilibre a = match a with
  Vide -> true
| Feuille f -> true
```

```
  | Noeud (g,x,d) -> if(abs((hauteur g) - (hauteur d)) <= 1)
                        then (est_equilibre g) && (est_equilibre d)
                     else false;;
```

Problème 1

1. Écrire la spécification et le code d'une fonction `somme a` qui retourne la somme des étiquettes d'un arbre binaire a d'entiers.

$$somme : \begin{cases} \texttt{arbreBin} < \texttt{int} > & \rightarrow \texttt{int} \\ a & \mapsto \text{somme des éléments de } a \end{cases}$$

```
let rec somme a = match a with
  Vide -> 0
| Feuille f -> f
| Noeud (g,x,d) -> x + somme g + somme d;;
```

2. Ecrivez une fonction `max_a a` qui calcule la plus grande étiquette présente dans un arbre binaire a d'entiers.

$$max\_a : \begin{cases} \texttt{arbreBin} < \texttt{int} > & \rightarrow \texttt{int} \\ a & \mapsto \text{Plus grand élément de } a \end{cases}$$

```
(* On précise ici que la valeur Feuille of 'a est utilisée pour les feuilles
des arbres.
La déclaration Noeud (Vide, x, Vide) est prohibée. *)

let rec max_a a = match a with
  Feuille f -> f
| Noeud (Vide,x,d) -> max x (max_a d)
| Noeud (g,x,Vide) -> max x (max_a g)
| Noeud (g,x,d) -> max x (max (max_a g) (max_a d))
| Vide -> failwith "Aucun arbre défini";;
```

Problème 2

Comme vous l'avez peut-être remarqué, il n'est pas toujours simple de ne pas disposer d'une vue globale du contenu de notre arbre. Sans pour autant disposer d'un mode de visualisation graphique, les méthodes de parcours d'arbre permettent d'obtenir le contenu de l'arbre selon différents ordres :

1. On donne l'algorithme en pseudo-code du parcours de l'ordre *préfixe* suivant :

Algorithme 1 : $parcoursPrefixe(a)$

Data : `Arbre binaire` a

begin

$n \leftarrow (g, x, d)$ // $n = root(a)$

$print(x)$

$parcoursPrefixe(g)$

$parcoursPrefixe(d)$

Ainsi, on liste chaque sommet la première fois qu'on le rencontre dans la balade de l'arbre.

Écrire le code Ocaml correspondant au parcours préfixe d'un arbre binaire a.

```
let rec prefixe print = function
  Vide          -> ()
| Feuille f     -> print f
| Noeud (g,x,d) -> print x;
                   prefixe print g;
                   prefixe print d;;
```

2. On donne l'algorithme en pseudo-code du parcours de l'ordre *postfixe* suivant :

Algorithme 2 : $parcoursPostfixe(a)$

Data : `Arbre binaire` a

begin

$n \leftarrow (g, x, d)$ // $n = root(a)$

$parcoursPostfixe(g)$

$parcoursPostfixe(d)$

$print(x)$

On liste ici chaque sommet la dernière fois qu'on le rencontre.

Écrire le code Ocaml correspondant au parcours postfixe d'un arbre binaire a.

```
let rec postfixe print = function
  Vide          -> ()
| Feuille f     -> print f
| Noeud (g,x,d) -> postfixe print g;
                   postfixe print d;
                   print x;;
```

3. On donne l'algorithme en pseudo-code du parcours de l'ordre *infixe* suivant :

Algorithme 3 : $parcoursInfixe(a)$

Data : `Arbre binaire` a

begin

$n \leftarrow (g, x, d)$ // $n = root(a)$

$parcoursInfixe(g)$

$print(x)$

$parcoursInfixe(d)$

Ce qui donne un parcours tel que l'on liste chaque sommet ayant un fils gauche la seconde fois qu'on le voit et chaque sommet sans fils gauche la première fois qu'on le voit

Écrire le code Ocaml correspondant au parcours infixe d'un arbre binaire a.

```
let rec infixe print = function
  Vide          -> ()
| Feuille f     -> print f
| Noeud (g,x,d) -> infixe print g;
                   print x;
                   infixe print d;;
```

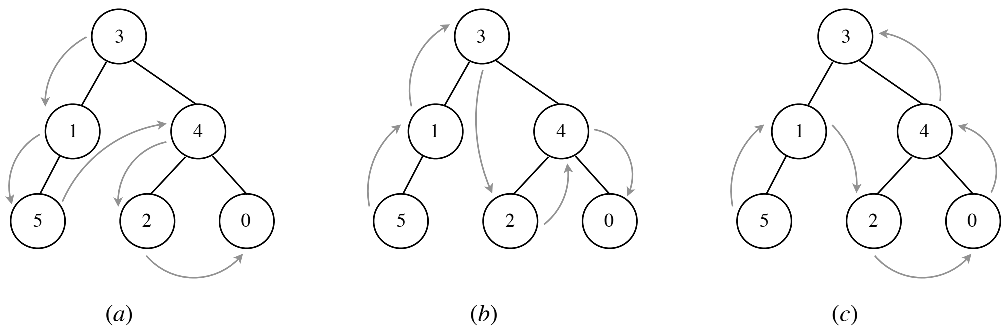


Figure 1: Parcours d'arbre (a) Préfixe (b) Infixe (c) Postfixe

```
let a1 = Noeud (Feuille 2, 4, Feuille 0);;
let a2 = Noeud (Feuille 5, 1, Vide);;
let a = Noeud (a2, 3, a1);;

(* Parcours de l'arbre binaire a ci-dessus *)
let parcours f = f (Printf.printf "%d ") a; print_newline ();;
parcours prefixe;;
  3 1 5 4 2 0
  - : unit = ()
parcours infixe;;
  5 1 3 2 4 0
  - : unit = ()
parcours postfixe;;
  5 1 2 0 4 3
  - : unit = ()
```

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 x et celles du sous-arbre droit d sont strictement supérieures.

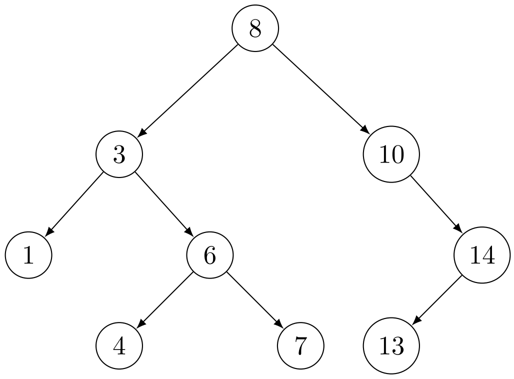


Figure 2: Exemple d'arbre binaire de recherche

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.

```
let a1 = Noeud (Feuille 13, 14, Vide);;
let a2 = Noeud (Feuille 4, 6, Feuille 7);;
let a3 = Noeud (Vide, 10, a1);;
let a4 = Noeud (Feuille 1, 3, a2);;
let a = Noeud (a4, 8, a3);;
```

1. Écrire la spécification et le code d'une fonction `est_de_recherche a` qui teste si l'arbre binaire a respecte la propriété des arbre binaire de recherche.

$$verifie : \begin{cases} '\mathtt{a} \times \mathcal{R} \times \mathtt{arbreBin} & \rightarrow \{true, false\} \\ e, \varphi, a & \mapsto \begin{cases} true & \text{si } \forall x \in a, \varphi(e, a) \equiv true \\ false & \text{sinon} \end{cases} \end{cases}$$

Où $\mathcal{R}$ est l'ensemble des relations de la forme $\varphi : '\mathtt{a} \times' \mathtt{b} \rightarrow \{true, false\}$.

```
let rec verifie e phi a = match a with
  Vide -> true
| Feuille f -> phi e f
| Noeud (g,x,d) -> (phi e x) && (verifie e phi g) && (verifie e phi d);;
```

$$est\_de\_recherche : \begin{cases} \mathtt{arbreBin} & \rightarrow \{true, false\} \\ a & \mapsto \begin{cases} true & \text{si } a \text{ est un arbre binaire de recherche} \\ false & \text{sinon} \end{cases} \end{cases}$$

```
  let est_de_recherche a = match a with
  Vide -> true
| Feuille _ -> true
| Noeud (g,x,d) -> (verifie x (>) g) && (verifie x (<) d);;
```

2. Écrire le code d'une fonction `recherche2 e a` qui teste si l'élément e appartient à l'arbre binaire de recherche a. Quelle est la complexité de cette méthode par rapport à la première du cours que vous avez écrite précédemment ?

```
let rec recherche2 e a = match a with
  Vide -> false
| Feuille f -> f = e
| Noeud (g,x,d) -> if (x = e) then true else
                   if(e < x) then recherche2 e g else recherche2 e d;;
```

La complexité moyenne de cette méthode est en $\log_2(n)$ (sous l'hypothèse que l'arbre est équilibré) car à chaque appel, on élague la moitié de l'arbre ce qui permet de réduire l'exploration.

Cette tehnique est assez similaire à la recherche par dichotomie.

Ce type de recherche est très utilisée dans les bases de données (Index B-tree) et en optimisation.

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

```
let rec add e a = match a with
  Vide -> Feuille e
| Feuille f -> if e < f then Noeud (Feuille e, f, Vide)
               else Noeud (Vide, f, Feuille e)
| Noeud(g,x,d) -> if(e < x) then Noeud(add e g, x, d)
               else Noeud(g, x, add e d);;
```

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

```
let rec remove e a = match a with
  Vide -> Vide
| Feuille f -> if e = f then Vide else Feuille f
| Noeud(g,x,d) -> if(e < x) then Noeud(remove e g, x, d)
               else if (e > x) then Noeud(g, x, remove e d)
               else (* x = e *)
               if(g = Vide) then d else
               if(d = Vide) then g else
               let y = max_a g in Noeud(remove y g, y, d);;
```