NOM :

PRÉNOM :

RICM3 - 2016/2017

Algorithmique Programmation Fonctionnelle

Devoir Surveillé :

Des arbres binaires de recherche

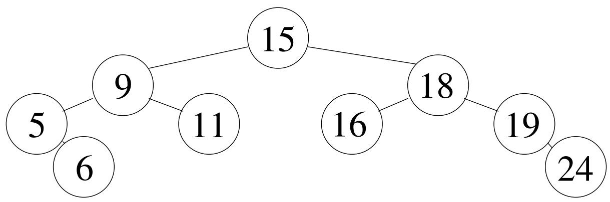
Durée : 1h

Le seul document autorisé est une feuille A4 recto-verso manuscrite de notes personnelles.

Le barème est indicatif. Le nombre de points correspond au nombre de minutes estimé nécessaire pour faire les exercices. Le total des points est de 70 points, dont 10 points de bonus.

La plupart des questions sont indépendantes, et on pourra toujours supposer qu'une fonction demandée en exercice est disponible pour les questions suivantes.

Définition. Un arbre binaire a est un arbre binaire de recherche (en abrégé ABR) si, pour tout nœud n de a, **toutes les étiquettes** des nœuds du **sous-arbre gauche** de n sont **inférieures** à l'étiquette de n, et si toutes les étiquettes des nœuds du sous-arbre droit de n sont supérieures à l'étiquette de n. L'arbre ci-dessous en est un exemple :



Principe d'utilisation. L'avantage de ces arbres est qu'ils permettent d'utiliser presque systématiquement un principe dichotomique. Par exemple, dans l'arbre de la figure précédente, si on cherchait l'entier 11, il ne serait pas nécessaire de le chercher dans le fils droit de 15. On pourrait donc se contenter de le chercher dans le sous-arbre de racine 9, et à nouveau dans celui-ci, il n'est pas nécessaire d'explorer tout l'arbre, etc.

Implémentation. On représente un ABR en OCaml par un type identique à celui des arbres binaires usuels :

```
type abr = V | N of abr × int × abr
```

Ce n'est donc pas le type qui assurera que les arbres manipulés respectent la définition d'ABR, mais la façon dont on les construit.

1 Manipulations de base

Exercice 1 : Recherche d'un élément (5 points)

Écrire une fonction `present : int -> abr -> bool` qui détermine si l'entier reçu en argument est présent ou non dans l'ABR. Votre fonction devra profiter des propriétés des ABR pour effectuer cette recherche le plus efficacement possible.

Corrigé

let rec $present\ e\ a\ =$ match a with
  | $V \ \rightarrow$ false
  | $N(\_, x, \_)$ when $x = e \ \rightarrow$ true
  | $N(g, x, d) \ \rightarrow$ if $e < x$ then $present\ e\ g$
    else $present\ e\ d$

Exercice 2 : Insertion d'un élément (5 points)

Écrire une fonction `inserer : int -> abr -> abr` telle que `inserer e a` renvoie un ABR contenant les mêmes entiers que `a` et l'entier `e` en plus, placé à un endroit qui respecte la définition d'un ABR. On pourra supposer que l'entier `e` n'est pas déjà présent dans `a`.

Indication : Inspirez-vous de la fonction `present` ci-dessus, notamment du cas où l'entier recherché est absent.

Corrigé

let rec $inserer\ e\ a\ =$ match a with
  | $V \ \rightarrow \ N(V, e, V)$
  | $N(g, x, d) \ \rightarrow$ if $e < x$ then $N(inserer\ e\ g, x, d)$
    else $N(g, x, inserer\ e\ d)$

Exercice 3 : Preuve (10 points)

On rappelle que le nombre de noeuds d'un arbre binaire est défini par la fonction

let rec $nbnoeuds\ a\ =$ match a with
  | $V \ \rightarrow \ 0$
  | $N(g, \_, d) \ \rightarrow \ nbnoeuds\ g\ +\ nbnoeuds\ d\ +\ 1$

Démontrer que pour tout arbre binaire de recherche `a` et pour tout entier `e`, on a l'égalité `nbnoeuds (insere e a) = nbnoeuds a + 1`.

Corrigé

On effectue une démonstration par récurrence.

Pour l'arbre vide :

`nbnoeuds (insere e a) = nbnoeuds N(V,e,V) = 1 = nbnoeuds V + 1`.

Supposons cette égalité vérifiée par deux arbres `g` et `d`.

Soient alors deux entiers `e` et `x` tels que `e` < `x` :

$$\begin{array}{rcl} \textit{nbnoeuds}(\textit{insere } e\ N(g,x,d)) & = & \textit{nbnoeuds } N(\textit{insere } e\ g, x, d) \\ & = & \textit{nbnoeuds } (\textit{insere } e\ g) + \textit{nbnoeuds } d + 1 \\ & = & \textit{nbnoeuds } g + 1 + \textit{nbnoeuds } d + 1 \\ & = & \textit{nbnoeuds } N(g,x,d) + 1 \end{array}$$

Le cas où `e` > `x` se traite de façon similaire.

2 Vérifications

Dans toute cette partie, on s'intéresse au problème suivant : étant donné un arbre binaire, a priori quelconque, celui-ci respecte-t-il la définition d'ABR ?

On propose trois méthodes indépendantes, à peu près équivalentes mais utilisant des concepts distincts d'OCaml, pour résoudre ce problème.

Exercice 4 : Vérification par parcours (10 points)

1. *(5 points)* Écrire une fonction `est_triee : int list -> bool` qui détermine si la liste reçue en argument est en ordre croissant.
   On considérera qu'une liste contenant 0 ou 1 élément est triée.

   Corrigé

   ```
   let rec est_triee l = match l with
     | [] | [_] -> true
     | x :: y :: s -> x <= y && est_triee (y :: s)
   ```

2. *(4 points)* Écrire une fonction `parcours : abr -> int list` qui renvoie la liste des entiers contenus dans un ABR. Pour tout nœud de la forme `N(g,x,d)`, la liste renvoyée devra contenir, dans l'ordre, les entiers de `g`, puis `x`, puis enfin les entiers de `d`.

Corrigé

$\mathsf{let\ rec}\ parcours\ a\ = \mathsf{match}\ a\ \mathsf{with}$
$\quad |\ V\ \rightarrow\ [\,]$
$\quad |\ N(g,x,d)\ \rightarrow\ (parcours\ g)@[x]@(parcours\ d)$

3. *(1 point)* Utiliser les deux fonctions précédentes pour écrire une fonction `verif_parcours : abr -> bool` qui vérifie si l'arbre binaire reçu en argument est bien un ABR.

Corrigé

$\mathsf{let}\ verif\ a\ =\ est\_triee\ (parcours\ a)$

Exercice 5 : Vérification par exceptions (15 points)

On définit les exceptions suivantes :

$\mathsf{exception}\ ArbreNonABR$
$\mathsf{exception}\ ArbreVide$

1. *(10 points)* Écrire une fonction `minmax : abr -> int * int` qui :
   - renvoie le couple (valeur minimale, valeur maximale) de l'arbre reçu en argument si celui-ci est effectivement un ABR
   - sinon, lève l'exception `ArbreNonABR`.

Corrigé

$\mathsf{let\ rec}\ minmax\ a\ = \mathsf{match}\ a\ \mathsf{with}$
$\quad |\ V\ \rightarrow\ raise\ ArbreVide$
$\quad |\ N(V,x,d)\ \rightarrow\ \mathsf{let}\ (mi,ma)\ =\ minmax\ d$
$\qquad\qquad \mathsf{in\ if}\ mi < x\ \mathsf{then}\ raise\ ArbreNonABR$
$\qquad\qquad\quad \mathsf{else}\ (x,ma)$
$\quad |\ N(g,x,V)\ \rightarrow\ \mathsf{let}\ (mi,ma)\ =\ minmax\ g$
$\qquad\qquad \mathsf{in\ if}\ ma > x\ \mathsf{then}\ raise\ ArbreNonABR$
$\qquad\qquad\quad \mathsf{else}\ (mi,x)$
$\quad |\ N(g,x,d)\ \rightarrow\ \mathsf{let}\ (mig,mag)\ =\ minmax\ g\ \mathsf{and}\ (mid,mad)\ =\ minmax\ d$
$\qquad\qquad \mathsf{in\ if}\ mag > x\ \vee\ mid < x$
$\qquad\qquad\quad \mathsf{then}\ raise\ ArbreNonABR$
$\qquad\qquad\quad \mathsf{else}\ (mig,mad)$

2. *(5 points)* Utiliser la fonction précédente pour écrire une fonction `verif_exceptions : abr -> bool` qui vérifie si l'arbre binaire reçu en argument est bien un ABR.

Corrigé

let $\mathit{verif\_exceptions}\ a\ =$
  try let _ $=$ $\mathit{minmax}\ a$ in true with
  $|\ \mathit{ArbreNonABR}\ \rightarrow$ false
  $|\ \mathit{ArbreVide}\ \rightarrow$ true

Exercice 6 : Vérification par ordre supérieur (15 points)

1. *(2 points)* Comment écrit-on un prédicat anonyme (sans `let`) de type `int -> bool` qui vérifie si son argument est inférieur à un entier `x` ?

fun $y\ \rightarrow\ y\ <\ x$

2. *(5 points)* Écrire une fonction `pourtous : (int -> bool) -> abr -> bool` telle que `pourtous p a` renvoie `true` si et seulement si tous les entiers de l'ABR `a` vérifient le prédicat `p`.

Corrigé

let rec $\mathit{pourtous}\ p\ a\ =$ match a with
  $|\ V\ \rightarrow$ true
  $|\ N(g, x, d)\ \rightarrow\ p\ x$
    $\wedge\ \mathit{pourtous}\ p\ g\ \wedge\ \mathit{pourtous}\ p\ d$

3. *(8 points)* À l'aide de `pourtous`, écrire une fonction `verif_ordresup : abr -> bool` qui vérifie si l'arbre binaire reçu en argument est bien un ABR.
Notons que cette version de `verif` n'est pas forcément très efficace !

Corrigé

let rec $\mathit{verif\_ordresup}\ a\ =$ match a with
  $|\ V\ \rightarrow$ true
  $|\ N(g,\ x,\ d)\ \rightarrow$
    $\mathit{verif\_ordresup}\ g\ \wedge\ \mathit{verif\_ordresup}\ d\ \wedge$
      $\mathit{pourtous}$ (fun $y\ \rightarrow\ y \leq x$) $g\ \wedge$
      $\mathit{pourtous}$ (fun $y\ \rightarrow\ x < y$) d

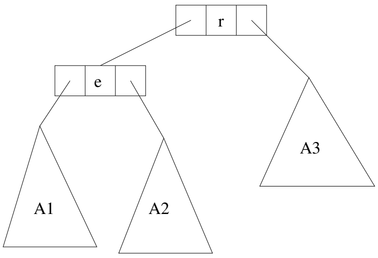
3 Réorganisation

Exercice 7 : Changer de racine (10 points)

On considère un ABR `a` contenant des entiers *tous différents*.

Le but de cet exercice est le suivant : étant donné un entier `e présent dans l'arbre`, créer un ABR `b` dont la racine porte l'entier `e`, et dont les entiers soient les mêmes que ceux de `a`.

1. Supposons par exemple que `a = N(g,r,d)` et que `e` < `r`. On commence par transformer `g` en un arbre dans lequel `e` est la racine, et on est alors ramené à la figure ci-dessous.



$\Rightarrow$

Corrigé

Montrez par un dessin comment transformer cet arbre en un ABR comportant le même ensemble d'étiquettes, et dont la racine soit étiquetée par e.

2. En déduire une fonction `reorg : int -> abr -> abr` qui effectue la manipulation demandée. L'entier donné en argument est la nouvelle racine de l'ABR renvoyé ; vous traiterez le cas où cet élément est absent à l'aide d'une exception de votre choix.

Corrigé

```
exception ElementAbsent

let rec reorg e a = (* ne renvoie jamais un arbre vide *)
  match a with
  | V -> raise ElementAbsent
  | N(g, r, d) when r = e -> a
  | N(g, r, d) -> if e < r then
      let N(gg, _, gd) = reorg e g
      in N(gg, e, N(gd, r, d))
    else
      let N(dg, _, dd) = reorg e d
      in N(N(g, r, dg), e, dd)
```