

TD3 : Arbres binaire de recherche (ABR)

La plupart des structures de données “efficaces” nécessitent la mise en place d’invariants structurels, i.e. des propriétés locales à chaque nœud qui ne peuvent être exprimées par le type seul et qui garantissent globalement l’efficacité des opérations. De telles propriétés sont par exemple :

- des propriétés numériques portant sur la taille, la profondeur, etc.
- des propriétés d’ordre entre éléments.
- des propriétés portant sur des données auxiliaires ajoutées (arbres colorés Rouge-Noir par exemple).

Lorsque les invariants sont suffisamment forts, nous pouvons obtenir une représentation canonique.

Enfin, ces invariants doivent obligatoirement être maintenus par les constructeurs abstraits (à supposer que leurs arguments les respectent : programmation offensive), si l’on veut que les structures créées soient toujours correctes. Les types doivent donc être abstraits/privés, afin que l’utilisateur ne puisse pas manipuler “à la main” les constructeurs “concrets” et casser les invariants.

1 Spécification

On définit le type récursif suivant :

```
type 'a abr = Vide | Noeud of 'a abr * 'a * 'a abr;;
```

Ce type permet de définir des arbres binaires de recherche (en abrégé *abr*). Ces arbres permettent de représenter des multi-ensembles sur lesquels les opérations ensemblistes sont efficaces par rapport aux mêmes opérations sur des structures de données linéaires comme les listes ou les tableaux. Un *abr* est :

- Soit *Vide*;
- Soit un *Noeud(g,n,d)* qui contient un élément *n*, un sous-arbre gauche *g* dont tous les éléments sont strictement inférieurs à *n*, et un sous-arbre droit *d* dont tous les éléments sont supérieurs ou égaux à *n*.

Dans le cas des *abr*, le principe des opérations comme le test d’appartenance, l’ajout ou le retrait d’un élément *e* à un multi-ensemble est le suivant :

- Soit l’arbre est *Vide*, il s’agit alors d’un cas terminal;
- Soit l’arbre est un *Noeud(g,n,d)*, alors si $e < n$ (respectivement $e > n$), on applique récursivement l’opération sur *g* (respectivement *d*). Enfin, le cas $e = n$ dépend de la fonction considérée.

2 Exercices

- ▷ **Exercice 1 (Itérateur)** Nous souhaitons posséder un itérateur structurel simple qui permet d’implanter rapidement de nombreux algorithmes nécessitant un parcours exhaustif des éléments. Cet itérateur est similaire à l’itérateur sur les arbres binaires standards *tree_fold*.

Écrire *abr_fold* : $((b \rightarrow a \rightarrow b \rightarrow b) \rightarrow a \text{ abr} \rightarrow b \rightarrow b)$ qui parcourt tous les éléments d’un *abr* et leur applique une fonction donnée. Le troisième paramètre correspond à la valeur par défaut pour les *abr* vides. Par exemple, l’expression suivante :

```
abr_fold f (Noeud(Noeud(Vide, 1, Vide), 2, Noeud(Vide, 3, Vide))) e
```

s’évalue comme :

```
f (f e 1 e) 2 (f e 3 e)
```

-
- ▷ **Exercice 2 (Test d'appartenance)** *Écrire une fonction qui teste l'appartenance d'un élément à un multi-ensemble représenté par un abr. Quelle est la complexité de cette opération ?*
 - ▷ **Exercice 3 (Ajout)** *Écrire une fonction qui ajoute un élément à un multi-ensemble représenté par un abr. Cette fonction renvoie le multi-ensemble résultat.*
 - ▷ **Exercice 4 (Cardinal)** *Écrire une fonction permettant de calculer le cardinal d'un multi-ensemble représenté par un abr, à l'aide de l'itérateur précédent.*
 - ▷ **Exercice 5 (Aplatissement)** *Écrire une fonction permettant de construire la liste triée des éléments d'un multi-ensemble représenté par un abr, à l'aide de l'itérateur précédent.*
 - ▷ **Exercice 6 (Tri d'une liste)** *Écrire une fonction triant une liste dans l'ordre croissant par l'intermédiaire d'un abr, en utilisant un itérateur sur les listes et l'itérateur sur les abr.*
 - ▷ **Exercice 7 (Retrait de l'élément minimum)** *Afin de faciliter l'écriture de l'opération de retrait, il est utile de définir une opération auxiliaire qui, à partir d'un abr, renvoie le couple formé de l'élément minimum et de l'arbre privé de cet élément. Écrire une fonction réalisant cette opération.*
 - ▷ **Exercice 8 (Retrait)** *Écrire une fonction qui retire un élément à un multi-ensemble représenté par un abr, en utilisant la fonction précédente (dans les cas terminaux uniquement). Cette fonction renvoie le multi-ensemble résultat.*

Arbres binaire de recherche

Exercice 1: (Itérateur)

```
let rec abr.fold f arb e =
  match arb with
  | Vide  $\rightarrow$  e
  | Node (g, n, d)  $\rightarrow$  f (abr.fold f g e) n (abr.fold f d e)
```

Exercice 2: (Test d'appartenance)

```
let rec appartient x abr =
  match abr with
  | Empty  $\rightarrow$  false
  | Node (g, n, d)  $\rightarrow$  if n < x then
                        appartient x d
                      else if n > x then
                        appartient x g
                      else true
  }  $x = n \parallel x > n \&\& \text{appartient } x d$ 
     $\parallel \text{appartient } x g$ .
```

Exercice 3: (Ajout)

```
let rec ajout abr e =
  match abr with
  | Empty  $\rightarrow$  Node (Empty, e, Empty)
  | Node (g, n, d)  $\rightarrow$  if e < n then
                        Node (ajout g e, n, d)
                      else
                        Node (g, n, ajout d e)
```


Exercice 4: (Cardinal)

let rec cardinal abr =

match abr with

| Vide \rightarrow 0

| Node (g, n, d) \rightarrow 1 + cardinal g + cardinal d.

let cardinal abr = abr-fold (fun taille g _ taille d \rightarrow 1 + taille g + taille d) abr 0.

Exercice 5: (Aplatissement)

let aplatissement abr =

abr-fold (fun fg n ld \rightarrow fg @ n :: ld) abr [].

Exercice 6: (Tri d'une liste)

let tri-liste l =

let abr = list-fold-right ajout l Vide in aplatissement abr

avec ajout : 'a \rightarrow 'a abr \rightarrow 'a abr.

Exercice 7: (Retrait de l'élément minimum)

let rec mini abr =

match abr with

| Vide \rightarrow fail with ("Arbre vide").

| Node (Vide, n, d) \rightarrow (d, n).

| Node (g, n, d) \rightarrow (Node (mini g, fst, n, d), (mini d, snd))

Exercice 8: (Retrait).

let rec retrait abr e =

if appartient e abr then

match abr with

| Node (g, n, d) \rightarrow if (n = e) then Node (g, (mini d, snd), (mini d, fst))

else if e < n then Node (retrait g, n, d) else Node (g, n, retrait d e).

else fail with " ".