Programmation Fonctionnelle: Arbres

Adrien Durier

13 septembre 2023

Arbres Binaires

Arbres binaires

Définition du type des arbres binaires ayant une donnée de type 'a en chaque noeud :

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

```
# let a =
   Noeud(10,
        Noeud(2,Noeud(8,Vide,Vide),Vide),
        Noeud(5,Noeud(11,Vide,Vide),Noeud(3,Vide,Vide)))
val a : arbre =   Noeud (10, ... , ...)
```

Arbres binaires : Taille

Nombre de noeuds d'un arbre :

```
# let rec taille a =
    match a with
      | Vide -> 0
      | Noeud(_, g, d) -> 1 + taille g + taille d
val taille : 'a arbre -> int
```

Arbres binaires : Taille

Nombre de noeuds d'un arbre :

```
# let rec taille a =
    match a with
      | Vide -> 0
      | Noeud(_, g, d) -> 1 + taille g + taille d
val taille : 'a arbre -> int
```

```
# taille a;;
- : int = 6
```

Arbres binaires : Profondeur

Longueur de la plus grande branche d'un arbre :

```
# let rec profondeur a =
   match a with
     | Vide -> 0
     | Noeud(_, g, d) -> 1 + max (profondeur g) (profondeur d)
- val profondeur : 'a arbre -> int
```

Arbres binaires : Miroir

Miroir d'un arbre binaire :

```
# let rec miroir a =
   match a with
     | Vide -> Vide
     | Noeud(r,g,d) -> Noeud(r, miroir d, miroir g)
- val miroir : 'a arbre -> 'a arbre
```

Arbres binaires : Miroir

Miroir d'un arbre binaire :

```
# let rec miroir a =
   match a with
     | Vide -> Vide
     | Noeud(r,g,d) -> Noeud(r, miroir d, miroir g)
- val miroir : 'a arbre -> 'a arbre
```

```
# miroir a;;
- : int arbre =
  Noeud (10,
    Noeud (5, Noeud (3, Vide, Vide), Noeud (11, Vide, Vide)),
    Noeud (2, Vide, Noeud (8, Vide, Vide)))
```

Arbres binaires : Recherche

Recherche d'un élément dans un arbre :

```
let rec recherche e a =
  match a with
    | Vide -> false
    | Noeud(x, g, d) ->
        x=e || recherche e g || recherche e d
```

- Temps de recherche dans le pire des cas : $O(n)$.
- Il faut des hypothèses plus fortes sur la structure de l'arbre afin d'obtenir une meilleure complexité.

Arbres n-aires

Arbres *n*-aires

- Les arbres *n*-aires ont un nombre arbitraire de sous-arbres
- Définition du type des arbres *n*-aires :

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

Arbres *n*-aires

- Les arbres *n*-aires ont un nombre arbitraire de sous-arbres
- Définition du type des arbres *n*-aires :

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

```
# let a =
   Noeud(10,[ Noeud(2,[]);
              Noeud(5,[Noeud(11,[]);Noeud(3,[])]);
              Noeud(8,[])]);;
val a : int arbre =  Noeud (10,[...])
```

Arbres n-aires : Taille et profondeur

Taille et profondeur des arbres n-aires :

```
# let rec taille a =
   match a with
     | Vide -> 0
     | Noeud(_, l) ->
        1 + List.fold_left (fun acc x -> acc + taille x) 0 l
val taille : 'a arbre -> int
```

Arbres n-aires : Taille et profondeur

Taille et profondeur des arbres n-aires :

```
# let rec taille a =
   match a with
     | Vide -> 0
     | Noeud(_, l) ->
        1 + List.fold_left (fun acc x -> acc + taille x) 0 l
val taille : 'a arbre -> int
```

```
# let rec profondeur a =
  match a with
    | Vide -> 0
    | Noeud(_, l) ->
       1 + List.fold_left
             (fun acc x -> max acc (profondeur x)) 0 l
val profondeur : 'a arbre -> int
```

Arbres *n*-aires : Ensemble des valeurs

Éléments d'un arbre n-aire

```
# let liste_arbre a =
   let rec liste_rec acc a =
     match a with
      | Vide -> acc
      | Noeud(r, l) -> List.fold_left liste_rec (r::acc) l
   in
   liste_rec [] a
val liste_arbre : 'a arbre -> 'a list
```

```
# liste_arbre a;;
- : int list = [10; 2; 5; 11; 3; 8]
```