

Exemples du chapitre 6

Structures de données et algorithmes

Sommaire

6.1	Quelques structures de données courantes	53
6.1.1	Piles	53
6.1.2	Files (FIFO)	54
6.1.3	Listes doublement chaînées circulaires	55
6.1.4	Ensembles et tables d'association	55
6.1.5	Mémoïsation	55
6.2	Représentation des graphes	56
6.2.1	Arbres : exemple des parcours	56
6.2.2	Liste de couples	57
6.2.3	Matrices d'adjacence	57
6.2.4	Listes d'adjacence	57
6.2.5	Structures de données cycliques	57
6.3	Itérateurs	58
6.3.1	Boucles et fonctions d'itération	58
6.3.2	Itérateurs sur les listes	59
6.3.3	Récurtivité terminale	60
6.3.4	Itérateurs sur les arbres	61

6.1 Quelques structures de données courantes

6.1.1 Piles

Interface

```
type 'a t
exception Empty
val empty : 'a t
val is_empty : 'a t -> bool
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> ('a * 'a t)
(** raises [Empty] if the stack is empty *)
```

Implémentation

```
type 'a t = 'a list
exception Empty
let empty = []
```

```

let push a p = a::p
let pop = function
| [] -> raise Empty
| a::l -> (a, l)

```

6.1.2 Files (FIFO)

1^{re} version (fonctionnelle, coût élevé)

Interface

```

type 'a t
exception Empty
val empty : 'a t
val is_empty : 'a t -> bool
val add : 'a -> 'a t -> 'a t
val remove : 'a t -> ('a * 'a t)
(** raises [Empty] if the queue is empty *)

```

Implémentation

```

type 'a t = 'a list
let empty = []
let is_empty f = f = []
let add a f = f@[a]
let remove = function
| [] -> raise Empty
| a::l -> (a, l)

```

2^e version (impérative)

Interface

```

type 'a t
exception Empty
val create : unit -> 'a t
val add : 'a -> 'a t -> unit
val remove : 'a t -> 'a
(** raises [Empty] if the queue is empty *)

```

Implémentation

```

type 'a content = Vide | Cons of 'a * 'a content ref;;

type 'a t = {mutable first: 'a content; mutable last : 'a content};;

exception Empty

let create () = {first = Vide; last = Vide};;

let add a f =
  match f.last with
  | Vide -> (* Dans ce cas F.first doit être Vide aussi *)
    f.first <- Cons (a, ref Vide);
    f.last <- f.first;
  | Cons (_, r) ->
    r := Cons (a, ref Vide);
    f.last <- !r

let remove f = match f.first with
| Vide -> raise Empty
| Cons (a, r) -> f.first <- !r ; a

let is_empty f = f.first = Vide

```

3^e version (fonctionnelle, plus efficace)

Interface : la même que la première version

Implémentation

```

type 'a t = 'a list * 'a list

exception Empty

let empty = ([], [])

let add x (l1, l2) = (x::l1, l2)

let remove (l1, l2) = match l2 with
| a::l -> (a, (l1, l))
| [] -> match List.rev l1 with
| [] -> raise Empty
| a::l -> (a, ([], l));;

```

6.1.3 Listes doublement chaînées circulaires

Interface

```

type 'a t
val create : 'a -> 'a t
val add : 'a -> 'a t -> unit
val succ : 'a t -> 'a t
val prev : 'a t -> 'a t

```

Implémentation

```

type 'a t =
  {value: 'a;
   mutable succ: 'a t;
   mutable prev: 'a t}

let create a = let rec v = {value = a; succ = v; prev = v} in v;;

let add x r =
  let n = {value = x; prev = r.prev; succ = r } in
  n.succ.prev <- n;
  n.prev.succ <- n

let succ l = l.succ

let prev l = l.prev

```

Et listes doublement chaînées non circulaires ? (exercice)

6.1.4 Ensembles et tables d'association

Listes d'association

Arbres binaires équilibrés

Tables de hashage

Fonctions

Ensembles

6.1.5 Mémoïsation

```

# let memoise f =
  let module Table =
    Hashtbl.Make(struct
      type t = string
      let equal x y = compare x y = 0
      let hash = Hashtbl.hash
      end)
  in
  let table = Table.create 100 in
  fun key ->
    try
      Table.find table key
    with Not_found ->
      let valeur = f key in
      Table.add table key valeur;
      valeur;;
val memoise : (string -> 'a) -> string -> 'a = <fun>
# let f x = for i = 1 to 100000000 do () done; x;;
val f : 'a -> 'a = <fun>
# let g = memoise f;;
val g : string -> string = <fun>
# g "a";; (* c'est long *)
- : string = "a"
# g "a";; (* c'est rapide *)
- : string = "a"
# g "a";; (* c'est rapide *)
- : string = "a"
# g "b";; (* c'est long *)
- : string = "b"
# g "b";; (* c'est rapide *)
- : string = "b"

```

6.2 Représentation des graphes

6.2.1 Arbres : exemple des parcours

```

# type 'a arbre = 'a * 'a arbre list;;
Characters 4-34:
  type 'a arbre = 'a * 'a arbre list;;
  ~~~~~
The type abbreviation arbre is cyclic
# type 'a arbre = Noeud of 'a * 'a arbre list;;
type 'a arbre = Noeud of 'a * 'a arbre list
# let a = Noeud ("a", [Noeud ("b", [Noeud ("d", [])]); Noeud ("c", [])]);
val a : string arbre =
  Noeud ("a", [Noeud ("b", [Noeud ("d", [])]); Noeud ("c", [])])
# let rec parcours_prof = function
  | Noeud (a, l) -> print_string a; List.iter parcours_prof l;;
val parcours_prof : string arbre -> unit = <fun>
# let rec parcours_prof (Noeud (a, l)) =
  print_string a; List.iter parcours_prof l;;
val parcours_prof : string arbre -> unit = <fun>
# parcours_prof a;;
abcd- : unit = ()
# let parcours_largeur arbre =
  let rec aux = function
  | [] -> ()
  | Noeud (a, voisins)::l ->
    print_string a;
    aux (l@voisins)
  in
  aux [arbre];;
val parcours_largeur : string arbre -> unit = <fun>
# parcours_largeur a;;
abcd- : unit = ()

```



```

    [Graphe ("d",
      [Graphe ("d",
        [Graphe ("d",
          ...
        ]
      ]
    )
  ]
)
val n5 : string graphe =
  Graphe ("e",
    [Graphe ("b",
      [Graphe ("a",
        [Graphe ("b",
          [Graphe ("a",
            ...
          ]
        ]
      ]
    )
  ]
)

```

```

# let rec a = a + 1;;
Characters 12-17:
  let rec a = a + 1;;
          ^^^^^

```

This kind of expression is not allowed as right-hand side of 'let rec'

6.3 Itérateurs

6.3.1 Boucles et fonctions d'itération

Application répétée bornée (paramètres $(0, 3, instr)$) :

```

# let t = [|0; 4; 8; 12|];;
val t : int array = [|0; 4; 8; 12|]
# for i = 0 to 3 do
  t.(i) <- t.(i) + 1
done;;
- : unit = ()
# t;;
- : int array = [|1; 5; 9; 13|]

```

Réduction bornée (paramètres $(0, 3, 0, +, t)$)

```

# let t = [| 0; 4; 8; 12 |];;
val t : int array = [|0; 4; 8; 12|]
# for i = 1 to 3 do
  acc := !acc + t.(i)
done;;
- : unit = ()
# t;;
- : int array = [|0; 4; 8; 12|]

```

Itération non bornée (paramètres $(booleen, instr)$)

```

# let x = ref 5;;
val x : int ref = {contents = 5}
# let y = ref 13;;
val y : int ref = {contents = 13}
# while !y > !x do
  y := !y - a
done;;
- : unit = ()
# !y;;
- : int = 4

```

Avec des fonctions :

```

# let do_for deb fin instr =
  for i = deb to fin do instr i done;;
val do_for : int -> int -> (int -> 'a) -> unit = <fun>
# let do_while b i = while b () do i () done;;
val do_while : (unit -> bool) -> (unit -> 'a) -> unit = <fun>

# let acc = ref 0;;
val acc : int ref = {contents = 0}
# do_for 0 3 (fun i -> acc := !acc + t.(i));;

```

```

- : unit = ()
# !acc;;
- : int = 32

# let x = ref 5;;
val x : int ref = {contents = 5}
# let y = ref 13;;
val y : int ref = {contents = 13}
# do_while (fun () -> !y > !x) (fun () -> y := !y - a);;
- : unit = ()
# !y;;
- : int = 4

# let rec do_for deb fin instr = if deb <= fin then begin
  instr deb;
  do_for (deb + 1) fin instr
end;;
val do_for : int -> int -> (int -> 'a) -> unit = <fun>
# let rec do_while b instr =
  if b() then begin
    instr ();
    do_while b instr
  end;;
val do_while : (unit -> bool) -> (unit -> 'a) -> unit = <fun>

```

6.3.2 Itérateurs sur les listes

map

```

# let rec inc = function
| [] -> []
| x::l -> (x+1)::inc l;;
val inc : int list -> int list = <fun>

# let rec map f = function
| [] -> []
| x::l -> (f x)::map f l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# inc [1;2;3];;
- : int list = [2; 3; 4]
# map succ [1;2;3];;
- : int list = [2; 3; 4]
# let inc = map succ;;
val inc : int list -> int list = <fun>

```

reduce

```

# let rec sigma = function
| [] -> 0
| x::l -> x + sigma l;;
val sigma : int list -> int = <fun>

# let rec reduce f e = function
| [] -> e
| x::l -> f x (reduce f e l);;
val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>

# let sigma = reduce (+) 0;;
val sigma : int list -> int = <fun>

# let length = function
| [] -> 0
| x::l -> 1 + length l;;

```

```

val length : int list -> int = <fun>

# let length = reduce (fun x y -> y + 1) 0;;
val length : int list -> int = <fun>

# let copy = reduce (:) [];;
Characters 19-21:
  let copy = reduce (:) [];;
                ^
Syntax error
# let copy = reduce (fun a l -> a::l) [];;
val copy : 'a list -> 'a list = <fun>
# let copy l = reduce (fun a l -> a::l) [] l;;
val copy : 'a list -> 'a list = <fun>

# let rec reverse = function (* version pas efficace *)
  | [] -> []
  | x::l -> (reverse l)@[x];;
val reverse : 'a list -> 'a list = <fun>
# let reverse = reduce (fun x l -> l@[x]) [];;
val reverse : 'a list -> 'a list = <fun>
# let reverse l = reduce (fun x l -> l@[x]) [] l;;
val reverse : 'a list -> 'a list = <fun>

# let rec append l1 l2 = match l1 with
  | [] -> l2
  | x::l -> x::(append l l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
# let rec append l2 l1 = match l1 with (* j'inverse juste les paramètres *)
  | [] -> l2
  | x::l -> x::(append l2 l);;
val append : 'a list -> 'a list -> 'a list = <fun>
# let append l2 = reduce (fun x l -> x::l) l2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# let append = reduce (fun x l -> x::l);;
val append : 'a list -> 'a list -> 'a list = <fun>

# let map f = reduce (fun x l -> (f x)::l) [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map succ [4;7;99];;
- : int list = [5; 8; 100]

```

fold

```

# let rec fold_left f a = function
  | [] -> a
  | x::l -> fold_left f (f a x) l;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
# let rec fold_right f l a = match l with (* quasimentt reduce *)
  | [] -> a
  | x::l -> f x (fold_right f l a);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
# let sigma liste = List.fold_right (+) liste 0;;
# let sigma2 = List.fold_left (+) 0;;
# let append l1 = List.fold_right (fun a l -> a::l) l1;;
# let rev liste = List.fold_left (fun l a -> a::l) [] liste;;
# let allassoc3 a l =
  List.fold_left (fun ll (x,y) -> if x=a then y::ll else ll) [] l;;

```

6.3.3 Récursivité terminale

Taille de la pile

Quelle version est la meilleure ?

```

# let rec somme1 p n = if n = p then p else n + (somme1 p (n-1));;

```

```

val somme1 : int -> int -> int = <fun>

# let somme2 p n =
  let rec aux n = if n = p then p else n + (aux (n-1)) in
  aux n;;
val somme2 : int -> int -> int = <fun>

# let rec somme3 p n =
  let rec aux n temp = if n < p then temp else aux (n-1) (temp+n)
  in aux n 0;;
val somme3 : int -> int -> int = <fun>

```

Itérateurs récursifs terminaux

Laquelle des deux?

```

# let rec iter f n x =
  if n = 0 then x
  else f (iter f (n-1) x);;
val iter : ('a -> 'a) -> int -> 'a -> 'a = <fun>

# let rec iter f n x =
  if n = 0 then x
  else iter f (n-1) (f x);;
val iter : ('a -> 'a) -> int -> 'a -> 'a = <fun>

```

6.3.4 Itérateurs sur les arbres

```

type 'a barbre = Feuille | Noeud of 'a * 'a barbre * 'a barbre;;

let rec reduce f arbre v = match arbre with
  Feuille -> v
  | Noeud (x,g,d) -> f x (reduce f g v) (reduce f d v);;

let hauteur a = reduce (fun _ hg hd -> 1+(max hg hd)) a 0;;

let nbnoeuds a = reduce (fun _ nbg nbd -> 1 + nbg + nbd) a 0;;

let miroir a = reduce (fun a mg md -> Noeud(a,md,mg)) a Feuille;;

let recherche v a = reduce (fun a dansg dansd -> dansd || dansg || a=v) a false;;

```