Structures de données, I

I. Structure de données abstraites

En informatique, une structure de données abstraites est un type de données que l'on spécifie non pas en en donnant une implémentation, mais en indiquant les opérations que l'on veut pour les manipuler, ainsi que le comportement de ces opérations.

Prenons l'exemple d'un tableau. Les opérations que l'on veut avoir sont :

- une fonction d'**initialisation**, avec la taille requise et, éventuellement, une valeur d'initialisation ;
- le calcul de sa **longueur** ;
- l'**accés** à la valeur dans telle case ;
- la **modification** de la valeur dans telle case.

On peut spécifier aussi comme se comportent les différentes opérations vis-à-vis des autres. Ainsi,

- étant donné un tableau t, si l'on écrit la valeur x dans la case i, alors tant que l'on ne modifie pas la valeur de cette case, la valeur lue dans cette case sera x ;
- l'accés à la valeur présente dans une case ne modifie pas le tableau ;
- la modification de la valeur dans une case ne modifie pas le contenu des autres cases.

Ce qu'il faut retenir, c'est que l'on ne spécifie pas la façon dont le tableau est implémenté (le niveau concret), mais **on spécifie son comportement**, à l'aide d'une **interface** : c'est une structure de données abstraites.

1. Un exemple, le tableau

Un type `'a tableau` peut être spécifié en OCaml ainsi :

- `init : int -> 'a -> 'a tableau`
- `longueur : 'a tableau -> int`
- `acces : 'a tableau -> int -> 'a`
- `modif : 'a tableau -> int -> 'a -> unit`

Ces fonctions sont disponibles dans le module `Array` sous un autre nom (et on a droit à des notations spécifiques pour l'accés et la modification) :

```
# Array.make ;;
- : int -> 'a -> 'a array = <fun>
# Array.length ;;
```

```
- : 'a array -> int = <fun>
# Array.get ;;
- : 'a array -> int -> 'a = <fun>
# Array.set ;;
- : 'a array -> int -> 'a -> unit = <fun>
```

2. Structures de données persistentes et impératives

Une distinction importante concernant les structures de données est la suivante : lors d'une modification, que devient la structure ? On a principalement deux comportements possibles :

- ⋆ soit les deux états (avant et après modification) coexistent. La modification de la structure va donc conduire à définir une nouvelle version de la structure, sans supprimer la version précédente. On parle de structure de données **persistante**, ou immuable. Elles sont fréquentes en programmation fonctionnelle ;
- ⋆ soit le nouvel état « écrase » l'ancien état qui n'est plus accessible. On parle de structure de données modifiables.

Il est souvent possible de définir les deux versions d'une même structure de données abstraite. Pour les différentier, un bon indicateur est de voir si une fonction qui modifie la structure de données renvoie quelque chose (du type de la structure) ou rien.

Dans l'exemple précédent, la fonction de modification a pour type :

```
modif : 'a tableau -> int -> 'a -> unit
```

On a donc clairement affaire à une implémentation impérative.

II. Quelques structures usuelles

Nous allons étudier un ensemble de structures de données qui répondent tous à une spécification similaire : en plus d'une fonction de construction, on a principalement deux fonctions :

- ⋆ Une fonction d'ajout de type `'a t -> 'a -> unit`,
- ⋆ Une fonction de retrait `'a t -> 'a`.

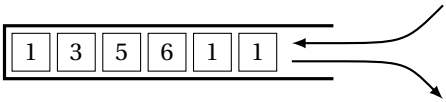
La méthode pour choisir quel élément va être retiré va donner lieu à plusieurs types de structures de données.

On peut souvent considérer aussi :

- ⋆ Une fonction qui indique si la structure est vide, de type `'a t -> bool`,
- ⋆ Une fonction qui renvoie l'élément `'a t -> 'a` qui sera retiré.

1. Piles

Commençons par une structure où l'élément retiré est toujours l'élément ajouté le plus récemment. On appelle cela une *pile*, et on peut se représenter une pile d'assiettes.



Les piles sont implémentées dans le module **`Stack`** (traduction anglaise de *pile*) :

```
# Stack.create ;;
- : unit -> 'a Stack.t = <fun>
# Stack.push ;;
- : 'a -> 'a Stack.t -> unit = <fun>
# Stack.pop ;;
- : 'a Stack.t -> 'a = <fun>
# Stack.top ;;
- : 'a Stack.t -> 'a = <fun>
# Stack.is_empty ;;
- : 'a Stack.t -> bool = <fun>
```

Il s'agit clairement d'une implémentation impérative. Les fonctions sont toutes en $\Theta(1)$.

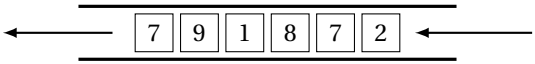
```
# let p = Stack.create () ;;
val p : '_weak1 Stack.t = <abstr>
# Stack.push 12 p ;;
- : unit = ()
# p ;;
- : int Stack.t = <abstr>
# Stack.push 34 p ;;
- : unit = ()
# Stack.top p ;;
- : int = 34
# Stack.pop p ;;
- : int = 34
# Stack.pop p ;;
- : int = 12
# Stack.is_empty p ;;
- : bool = true
```

Une version persistente des piles s'obtient directement en utilisant des listes.

```
let creer () = [] ;;

let ajout a p = a :: p ;;

let retrait p =
  match p with
  | [] -> failwith "retrait : pile vide"
  | a :: p' -> (a, p')
;;

let sommet p =
  match p with
  | [] -> failwith "sommet : pile vide"
  | a :: _ -> a
;;

let est_vide p =
  match p with
  | [] -> true
  | _ -> false
;;
```

2. Files

Commençons par une structure où l'élément retiré est toujours l'élément le plus ancien. On appelle cela une *file*, comme lorsque l'on fait la queue.



Le module **Queue** implémente une version impérative de la file. On notera la présence de synonymes pour coller au plus près à l'interface des piles.

```
# Queue.create ;;
- : unit -> 'a Queue.t = <abstr>
# Queue.add ;;
- : 'a -> 'a Queue.t -> unit = <fun>
# Queue.push ;; (* synonyme de add *)
- : 'a -> 'a Queue.t -> unit = <fun>
# Queue.take ;;
- : 'a Queue.t -> 'a = <fun>
# Queue.pop ;; (* synonyme de take *)
- : 'a Queue.t -> 'a = <fun>
# Queue.peek ;;
- : 'a Queue.t -> 'a = <fun>
```

```
# Queue.top ;; (* synonyme de peek *)
- : 'a Queue.t -> 'a = <fun>
# Queue.is_empty ;;
- : 'a Queue.t -> bool = <fun>
```

À nouveau, les fonctions sont toutes en $\Theta(1)$.

On peut comparer le comportement d'une file avec celui d'une file.

```
# let f = Queue.create () ;;
val f : '_weak2 Queue.t = <abstr>
# Queue.push 12 f ;;
- : unit = ()
# Queue.push 34 f ;;
- : unit = ()
# Queue.pop f ;;
- : int = 12
# Queue.pop f ;;
- : int = 34
# Queue.is_empty ;; f ;;
- : 'a Queue.t -> bool = <fun>
```

3. Files de priorités

Une autre variante de la structure précédente s'obtient lorsque l'on ajoute à chaque élément contenu dans notre structure une **priorité**. Dans ce cas, le retrait supprimera et renverra à chaque fois un élément de priorité maximale.

Nous verrons l'an prochain comment implémenter une telle structure efficacement, et des exemples d'utilisation.

III. Compléments

Voici quelles implémentations des structures de pile et de file.

1. Implémentation impérative d'une pile d'entiers de capacité fixée

À part la création en $\Theta(n)$ avec n la capacité, les opérations sont toutes en temps constant.

```
type pile = { tab : int array; mutable pos : int } ;;

(* val creer_pile : int -> pile *)
let creer_pile n = { tab = Array.make n 0; pos = 0 } ;;

(* val est_vide : pile -> bool *)
let est_vide p = p.pos = 0
```

```
(* val ajout : int -> pile -> unit *)
let ajout a p =
  if p.pos = Array.length p.tab then
    failwith "ajout : pile pleine"
  else begin
    p.tab.(p.pos) <- a;
    p.pos <- p.pos + 1
  end
;;

(* val retrait : pile -> int *)
let retrait p =
  if p.pos = 0 then
    failwith "retrait : pile vide"
  else begin
    p.pos <- p.pos - 1;
    p.tab.(p.pos)
  end
;;

(* val sommet : pile -> int *)
let sommet p =
  if p.pos = 0 then
    failwith "sommet : pile vide"
  else
    p.tab.(p.pos - 1)
;;
```

2. Implémentation impérative des files d'entiers de capacité fixée

Pour distinguer les files pleines des files vides, pour une capacité n, on utilisera un tableau de taille $n+1$. Ainsi le cas d'une file vide correspond à $\texttt{fin} = \texttt{deb}$ et une file pleine à $\texttt{fin} \equiv \texttt{deb} - 1 \left[n+1\right]$, soit respectivement $\texttt{fin} - \texttt{deb} \equiv 0 \left[n+1\right]$ et $\texttt{fin} - \texttt{deb} \equiv n \left[n+1\right]$.

À nouveau, à part la création en $\Theta(n)$ avec n la capacité, les opérations sont toutes en temps constant.

```
type file = {
  tab : int array;
  mutable deb : int;
  mutable fin : int;
}

(* val creer_file : int -> file *)
```

```
let creer_file n =
  {
    (* on a un tableau de taille n + 1
       pour TOUJOURS avoir au moins une case vide. *)
    tab = Array.make (n + 1) 0;
    deb = 0;
    fin = 0;
  }
;;

(* val est_vide : file -> bool *)
let est_vide f = f.fin = f.deb ;;

(* val ajout : int -> file -> unit *)
let ajout a f =
  if (f.fin + 1) mod Array.length f.tab = f.deb then
    failwith "ajout : file pleine"
  else
    f.tab.(f.fin) <- a;
  f.fin <- (f.fin + 1) mod Array.length f.tab
;;

(* val retrait : file -> int *)
let retrait f =
  if f.fin = f.deb then
    failwith "retrait : file vide"
  else
    let elt = f.tab.(f.deb) in
    f.deb <- (f.deb + 1) mod Array.length f.tab;
    elt
;;

(* val sommet : file -> int *)
let sommet f =
  if f.fin = f.deb then
    failwith "retrait : file vide"
  else
    f.tab.(f.deb)
;;
```

3. Implémentation persistente d'une file

On a deux listes, une liste de queue et une liste de tête. On ajoute les éléments dans la queue, on les retire dans la tête. Si celle-ci est vide, on « retourne » la queue en tête.

À cause de cela, l'opération de retrait n'est pas *a priori* en temps constant. On peut montrer cependant que ce retournement n'a pas lieu trop souvent, et qu'en moyenne, on a du temps constant. On parle de complexité *amortie* (lorsque l'on ne raisonne pas sur une opération prise individuellement mais sur une opération considérée au sein d'une suite d'opérations).

```
type 'a file = { tete : 'a list; queue : 'a list }

(* val creer_file : unit -> 'a file *)
let creer_file () = { tete = []; queue = [] }

(* val ajout : 'a -> 'a file -> 'a file *)
let ajout a f = { tete = f.tete; queue = a :: f.queue }

(* val retrait : 'a file -> 'a * 'a file *)
let retrait f =
  match f.tete with
  | a :: t -> (a, { tete = t; queue = f.queue })
  | [] -> (
      match List.rev f.queue with
      | [] -> failwith "retrait : file vide"
      | a :: t -> (a, { tete = t; queue = [] }))
;;
```

Voici une illustration de son comportement.

```
# let f = creer_file () ;;
val f : 'a file = {tete = []; queue = []}
# let f = ajout 4 f ;;
val f : int file = {tete = []; queue = [4]}
# let f = ajout 5 f ;;
val f : int file = {tete = []; queue = [5; 4]}
# let f = ajout 6 f ;;
val f : int file = {tete = []; queue = [6; 5; 4]}
# let v, f = retrait f ;;
val v : int = 4
val f : int file = {tete = [5; 6]; queue = []}
# let f = ajout 7 f ;;
val f : int file = {tete = [5; 6]; queue = [7]}
# let v, f = retrait f ;;
val v : int = 5
val f : int file = {tete = [6]; queue = [7]}
# let f = ajout 8 f ;;
val f : int file = {tete = [6]; queue = [8; 7]}
# let v, f = retrait f ;;
```

```
val v : int = 6
val f : int file = {tete = []; queue = [8; 7]}
# let v, f = retrait f ;;
val v : int = 7
val f : int file = {tete = [8]; queue = []}
```