Implémentations des piles et files

Dans ce document, on propose plusieurs implémentations possibles des structures abstraites de piles et files — persistantes ou impératives — de manière à réaliser concrètement les structures de données associées.

On cherche évidemment principalement des implémentations qui offrent des complexités en temps constant pour toutes les opérations.

1 Implémentation d'une pile persistante

La signature d'une pile persistante est :

OCaml
```
type 'a pstack

exception StackEmpty
exception StackFull

val empty : 'a pstack
val is_empty : 'a pstack -> bool
val is_full : 'a pstack -> bool
val push : 'a -> 'a pstack -> 'a pstack
val pop : 'a pstack -> 'a * 'a pstack
```

En réalité, il est très simple de se convaincre qu'il n'y a aucune différence entre une pile persistante et une liste Caml. Autrement dit une liste Caml implémente déjà la structure abstraite de liste persistante. Il reste simplement à ajuster l'interface.

OCaml
```
type 'a pstack == 'a list
```

OCaml
```
let empty = []
```

OCaml
```
let is_empty p = p = []
```

OCaml
```
let is_full p = false
```

OCaml
```
let push elt p = elt :: p
```

OCaml
```
let pop p =
  match p with
  | [] -> raise StackEmpty
  | head :: tail -> (head, tail)
```

La complexité de toutes ces opérations est bien en $O(1)$ en Caml. Avec cette implémentation l'exception `StackFull` n'est pas utile.

2 Implémentation d'une pile impérative

La signature d'une pile impérative est :

OCaml
```
type 'a mstack

exception StackEmpty
exception StackFull

val empty : unit -> 'a mstack
val is_empty : 'a mstack -> bool
val is_full : 'a mstack -> bool
val push : 'a -> 'a mstack -> unit
val pop : 'a mstack -> 'a
```

Une première idée est de reprendre la structure de pile persistante et de la rendre impérative. On peut par exemple utiliser une référence de liste.

OCaml
```
type 'a mstack == 'a list ref
```

OCaml
```
let empty () = ref []
```

OCaml
```
let is_empty p = !p = []
```

OCaml
```
let is_full p = false
```

OCaml
```
let push elt p =
  p := elt :: !p
```

OCaml
```
let pop p =
  match !p with
  | [] ->
     raise StackEmpty
  | head :: tail ->
     p := tail;
     head
```

La complexité de toutes ces opérations est encore en $O(1)$.

EXERCICE 1

Ce n'est pas très élégant... On propose, ce qui revient certes au même, d'utiliser plutôt le type :

OCaml
```
type 'a mstack = {mutable body : 'a list}
```

Adapter les fonctions précédentes avec cette modification du type.

Une deuxième possibilité est d'utiliser un tableau. L'inconvénient est que la taille de la pile sera alors bornée. La complexité des opérations sera bien en $O(1)$, sauf la création du contenu, qui sera linéaire en la taille maximale de la pile (création du tableau associé). On peut palier ce problème avec des tableaux redimensionnables, qui sont hors programme.

La principale difficulté, puisque l'on cherche ici une implémentation générique, est qu'il s'avère nécessaire d'initialiser un tableau sans connaître a priori le type des élements qu'il va contenir. Pour cela, on se propose d'utiliser une option :

OCaml
```
type 'a option = None | Some of 'a
```

☞ **Remarque 1**

Ce type est déjà défini par défaut en CAML.

OCaml
```
let max_stack_size = 1024
```

OCaml
```
type 'a mstack = {
  content : 'a option array;
  mutable pointer : int
}
```

OCaml
```
let empty () = {
  content = Array.make max_stack_size None;
  pointer = -1
}
```

OCaml
```
let is_empty p = p.pointer = -1
```

OCaml
```
let is_full p = p.pointer = max_stack_size - 1
```

OCaml
```
let push elt p =
  if is_full p then
    raise StackFull
  else begin
    p.pointer <- p.pointer + 1;
    p.content.(p.pointer) <- Some elt
  end
```

OCaml
```
let pop p =
  if is_empty p then
    raise StackEmpty
  else begin
    p.pointer <- p.pointer - 1;
    match p.content.(p.pointer + 1) with
    | Some elt -> elt
    | None -> failwith "Implementation error : pop"
  end
```

On préfère proposer un filtrage exhaustif, même si ce dernier cas ne peut normalement jamais arriver.

EXERCICE 2

L'utilisation d'une option complique sensiblement l'implémentation. Dans le cas où le type est connu à l'avance et si l'on dispose d'une valeur par défaut pour ce type, ceci n'est alors plus nécessaire car on peut initialiser le tableau avec cette valeur. Implémenter, un peu plus simplement, la structure de pile impérative d'entiers suivante :

OCaml
```
type int_stack

exception StackEmpty
exception StackFull

val empty : unit -> int_stack
val is_empty : int_stack -> bool
val is_full : int_stack -> bool
val push : int -> int_stack -> unit
val pop : int_stack -> int
```

On remarque que le paramètre de type a disparu, puisque l'on sait que les éléments sont des entiers.

3 Implémentation d'une file persistante

La signature d'une file persistante est :

OCaml
```
type 'a pqueue

exception QueueEmpty
exception QueueFull

val empty : 'a pqueue
val is_empty : 'a pqueue -> bool
val is_full : 'a pqueue -> bool
val push : 'a -> 'a pqueue -> 'a pqueue
val pop : 'a pqueue -> 'a * 'a pqueue
```

On propose, pour implémenter une file persistante, d'utiliser un couple de listes CAML (ou, ce qui revient au même, deux piles persistantes !). On empile les éléments dans une pile (entrée) et on dépile les éléments dans l'autre (sortie). Lorsque l'on souhaite retirer un élément de la pile de sortie et que celle-ci est vide, on y transvase alors le contenu de la pile d'entrée.

OCaml
```
type 'a pqueue = {input : 'a list; output : 'a list}
```

OCaml
```
let empty = {input = []; output = []}
```

OCaml
```
let is_empty q = q.input = [] && q.output = []
```

OCaml
```
let is_full q = false
```

OCaml
```
let push elt q =
  {input = elt :: q.input; output = q.output}
```

OCaml
```
let rec pop q =
  match (q.input, q.output) with
  | _, head :: tail ->
     head, {input = q.input; output = tail}
  | _ ->
     match List.rev q.input with
     | [] ->
        raise QueueEmpty
     | head :: tail ->
        head, {input = []; output = tail}
```

Toutes les opérations sont bien en temps contant *sauf* lorsque l'on veut défiler un élément et que la pile de sortie est vide, la complexité étant alors linéaire en la taille de la file. Cependant, on peut montrer qu'une suite de p opérations [1] à partir d'une file vide est toujours en $O(p)$. Considérons une suite de p opérations. En comptant à part les éventuels appels à `rev`, toutes les opérations sont de complexité constante (y compris `pop`). La complexité d'une suite de p opérations est donc $O(p)$ plus la complexité cumulée des appels à `rev`. Un élément peut être dans une liste passée en argument à `rev` au plus une fois par ajout dans la file, puisque `rev` ne s'applique que sur des éléments de la première liste et que suite à l'appel à `rev` les éléments se retrouvent dans la deuxième. Le nombre d'éléments pouvant se trouver dans une liste passée en argument à `rev` est donc inférieur à p, puisqu'il y a au plus p ajouts et que l'on commence à partir d'une file vide. La complexité de `rev` étant linéaire, la complexité cumulée des appels à `rev` est en $O(p)$ et il en est de même de la complexité de la suite de p opérations.

4 Implémentation d'une file impérative

La signature d'une file impérative est :

OCaml
```
type 'a mqueue

exception QueueEmpty
exception QueueFull

val empty : unit -> 'a mqueue
val is_empty : 'a mqueue -> bool
val is_full : 'a mqueue -> bool
val push : 'a -> 'a mqueue -> unit
val pop : 'a mqueue -> 'a
```

1. On appelle cela une analyse de complexité amortie.

EXERCICE 3

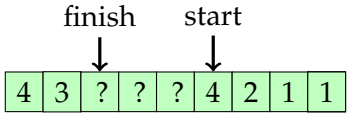
Une première possibilité est d'adapter l'idée que nous avons mise en place pour la structure de file persistante. On définit alors le type d'une file impérative par

OCaml
```
type 'a mqueue = {
  mutable input : 'a list;
  mutable output : 'a list
}
```

Implémenter alors les opérations de file impérative avec ce choix de type.

Une deuxième possibilité est d'utiliser un tableau circulaire, c'est-à-dire qu'une fois arrivé à la fin du tableau, un indice reprend au début.

On utilise deux indices : `finish` qui pointe sur la case ou l'on peut enfiler et `start` qui pointe sur la case du premier élément à défiler [2].



OCaml
```
type 'a mqueue = {
  content : 'a option array;
  mutable finish : int;
  mutable start : int
}
```

On utilise une case en plus pour pouvoir distinguer une file vide d'une file pleine (cas où `finish` et `start` seraient égaux). On aurait aussi pu prendre une variable booléenne (cf Centrale 2017).

2. Il y a bien d'autres manières de faire, par exemple pointer juste avant ou juste après. Il faut savoir s'adapter, cf. encore Centrale 2017.

OCaml
```
let max_queue_size = 1024 + 1
```

On utilise cette fonction pour chercher le représentant d'un indice dans la plage de valeurs $[\![0; \text{max\_queue\_size} - 1]\!]$ après d'éventuels décalages d'indices.

OCaml
```
let repr i = (i + max_queue_size) mod max_queue_size
```

OCaml
```
let empty () =
  {content = Array.make max_queue_size None;
   finish = 0;
   start = 0
  }
```

OCaml
```
let is_empty q = q.finish = q.start
```

OCaml
```
let is_full q = q.finish = repr (q.start - 1)
```

OCaml
```
let push elt q =
  if is_full q then
    raise QueueFull
  else begin
    q.content.(q.finish) <- Some elt;
    q.finish <- repr (q.finish + 1)
  end
```

OCaml
```
let pop q =
  if is_empty q then
    raise QueueEmpty
  else begin
    let elt_option = q.content.(q.start) in
    q.start <- repr (q.start + 1);
    match elt_option with
    | None -> failwith "Implementation error : pop"
    | Some elt -> elt
  end
```

Encore une fois les opérations sont toutes de coût constant sauf la création d'une file vide qui nécessite d'initialiser un tableau.

Exercice 4

Comme dans l'exercice 2, implémenter, plus simplement, la structure de file impérative pour le cas particulier des entiers. Donner la signature correspondante (`type int_queue`).

En TD, la semaine prochaine, nous étudierons une implémentation alternative en utilisant une liste chaînée circulaire.