VII - Piles et files

3 mars 2024

1 Types de données abstraits

Nous avons déjà croisé deux manières de stocker et d'accéder à des données : les listes et les tableaux. Nous avons vu qu'elles présentent des caractéristiques différentes. Les listes sont persistantes alors que les tableaux sont mutables. La longueur d'une liste peut être modifiée, pas celle d'un tableau. Chacun de ces types de données permet de faire une ou plusieurs opérations précises en temps constant : les listes permettent l'ajout ou la suppression d'un élément en temps constant ; les tableaux permettent l'accès à un élément et sa modification en temps constant. Ainsi, suivant l'usage que l'on veut faire de ces données, on choisira l'un ou l'autre de ces types de données.

Il existe d'autres types de données abstraits, ayant chacun leurs spécificités. Dans ce chapitre nous allons en voir trois : les piles, les files, et les files de priorité.

Nous verrons quelles sont leur *fonctions primitives*, c'est-à-dire les manipulations qui peuvent être réalisées en temps constant. Ensuite nous *implémenterons* ces types, c'est-à-dire que nous passerons de la théorie à la pratique en écrivant ces fonctions dans le langage utilisé, OCaml pour ce cours.

2 Piles

2.1 Définition de pile

Une *pile* (*stack* en anglais) est une structure de données linéaire dynamique, dans laquelle l'insertion ou la suppression d'un élément s'effectue toujours à partir de la même extrémité, appelée *sommet* de la pile. On parle de structure de type *LIFO* (*Last In, First Out*).

Cette structure nécessite :

- un constructeur permettant de créer une pile vide ;
- une fonction permettant d'*empiler* (*push*) un élément, c'est-à-dire d'ajouter un élément dans la pile (en OCaml les éléments empilés seront tos de même type) ;
- une fonction permettant de *dépiler* (*pop*) un élément, c'est-à-dire de supprimer l'élément au sommet de la pile et de le renvoyer ;
- une fonction permettant de tester si la pile est vide.

Ces quatre fonctions sont appelées les *primitives* de la structure.

2.2 Implémentation d'une pile à l'aide d'une liste

La structure de pile nécessite de pouvoir accéder en temps constant au sommet de la pile pour l'ajout ou la suppression d'un élément. La structure de liste semble plutôt bien adaptée, mais il

s'agit d'une structure persistante, alors que la structure de liste est une structure impérative, donc nécessite d'avoir des objets mutables. On définit le type suivant :

```
[1] : type 'a pile = {
        mutable pile  : 'a list
      } ; ;
```

```
[1] : type 'a pile = { mutable pile  : 'a list ; }
```

On définit ensuite les opérations primitives.

```
[2] : let creer_pile () =  {pile = []} ; ;
```

```
[2] : val creer_pile  : unit -> 'a pile = <fun>
```

```
[3] : let ma_pile = creer_pile () ; ;
```

```
[3] : val ma_pile  : '_weak1 pile = {pile = []}
```

```
[4] : let est_vide p = p.pile = []  ; ;
```

```
[4] : val est_vide  : 'a pile -> bool = <fun>
```

```
[5] : est_vide ma_pile ; ;
```

```
[5] : -  : bool = true
```

```
[6] : let empile x p =
        p.pile <- x : :p.pile
      ; ;
```

```
[6] : val empile  : 'a -> 'a pile -> unit = <fun>
```

```
[7] : empile 'l' ma_pile ; ;
```

```
[7] : -  : unit = ()
```

```
[8] : ma_pile ; ;
```

```
[8] : -  : char pile = {pile = ['l']}
```

```
[9] : empile 14 ma_pile ; ;
```

```
Line 1, characters 10-17 :
Error : This expression has type char pile
        but an expression was expected of type int pile
        Type char is not compatible with type int
```

```
[10] : let depile p =
         match p.pile with
         | [] -> failwith "Pile vide"
         | t : :q -> p.pile <- q  ; t
```

```
 ; ;
```

```
[10] : val depile  : 'a pile -> 'a = <fun>
```

```
[11] : depile ma_pile ; ;
```

```
[11] : -  : char = 'l'
```

```
[12] : ma_pile ; ;
```

```
[12] : -  : char pile = {pile = []}
```

```
[13] : let pp = creer_pile () ; ;
       for i = 1 to 10 do empile i pp done ;
       while not (est_vide pp)
       do print_int (depile pp)  ; print_newline()
       done ; ;
```

```
10
9
8
7
6
5
4
3
2
1
```

```
[13] : val pp  : '_weak2 pile = {pile = []}
       -  : unit = ()
```

Toutes ces opérations s'effectuent en temps constant $O(1)$.

2.3 Implémentation d'une pile à l'aide d'un tableau

On stocke les éléments de la pile dans un tableau, en gérant une variable pour la hauteur de la pile. Cette donnée peut éventuellement être stockée en premier élément du tableau.

```
[14] : type 'a pile = {
           mutable hauteur  : int  ;
           pile  : 'a option array
         } ; ;
```

```
[14] : type 'a pile = { mutable hauteur  : int ; pile  : 'a option array ; }
```

La fonction de création de pile doit prendre en paramètre une taille maximale (ou *capacité*) de la pile qui permettra d'initialiser le tableau, ainsi qu'une "valeur initiale" dont le seul intérêt sera d'initialiser le tableau avec des valeurs du bon type.

```
[15] : let creer_pile c =  {
         hauteur = 0  ;
         pile = Array.make c None
       } ; ;
```

```
[15] : val creer_pile  : int -> 'a pile = <fun>
```

```
[16] : let ma_pile = creer_pile 5 ; ;
```

```
[16] : val ma_pile  : '_weak3 pile =
         {hauteur = 0 ; pile = [|None ; None ; None ; None ; None|]}
```

Pour gérer les erreurs lorsqu'on souhaite empiler dans une pile pleine ou dépiler une pile vide, on peut créer deux exceptions.

```
[17] : let est_vide p = p.hauteur = 0 ; ;
```

```
[17] : val est_vide  : 'a pile -> bool = <fun>
```

```
[18] : let empile x p =
         if p.hauteur = Array.length p.pile
         then failwith "Pile pleine"
         else
           begin
             p.pile.(p.hauteur) <- Some x  ;
             p.hauteur <- p.hauteur + 1
           end
       ; ;
```

```
[18] : val empile  : 'a -> 'a pile -> unit = <fun>
```

```
[19] : empile 1 ma_pile ; ;
```

```
[19] : -  : unit = ()
```

```
[20] : for i = 2 to 5 do empile i ma_pile done ; ;
       ma_pile ; ;
```

```
[20] : -  : unit = ()
       -  : int pile =
       {hauteur = 5 ; pile = [|Some 1 ; Some 2 ; Some 3 ; Some 4 ; Some 5|]}
```

```
[21] : empile 6 ma_pile ; ;
```

```
[21] : Exception : Failure "Pile pleine".
```

```
[22] : let depile p =
         if est_vide p
         then failwith "Pile vide"
         else
           begin
```

```
      p.hauteur <- p.hauteur - 1  ;
      match p.pile.(p.hauteur) with
      | Some x -> x
      | None -> failwith "Non possible"
    end
;;
```

[22] :
```
val depile  : 'a pile -> 'a = <fun>
```

[23] :
```
depile ma_pile ;;
```

[23] :
```
-  : int = 5
```

[24] :
```
ma_pile ;;
```

[24] :
```
-  : int pile =
{hauteur = 4 ; pile = [|Some 1 ; Some 2 ; Some 3 ; Some 4 ; Some 5|]}
```

[ ] :
```

```

Toutes ces opérations s'effectuent en temps constant $O(1)$, sauf la création de la pile qui s'effectue en $O(c)$ où c est la capacité de la pile.

L'inconvénient de cette implémentation est qu'elle fixe dès la création de la pile la taille de celle-ci ; dans le cas où cette taille a été sous-estimée, la pile déborde ; si elle est sur-estimée, on réserve de l'espace en mémoire qui ne sera jamais utilisé. Les débordements peuvent éventuellement être gérés en réallouant le tableau si besoin, par exemple en doublant sa capacité si la pile est pleine. Mais cette opération a un coût, en $O(n)$ si la pile est de taille n. Néanmoins, ce cas ne se produit pas trop souvent. En effet, lorsqu'il y a n éléments dans la pile et que le tableau est plein, le coût est $O(n)$, mais les n prochaines opérations d'empilement seront réalisées en temps constant. Par conséquent, pour ces n opérations, le coût total est $O(n)$ donc le coût moyen est $O(1)$. On dit que la complexité *amortie* de l'opération *empiler* est $O(1)$.

2.4 Module `Stack`

OCaml dispose d'un module appelé `Stack` permettant de créer et de manipuler des piles :

[25] :
```
let maPile = Stack.create() ;;
```

[25] :
```
val maPile  : '_weak4 Stack.t = <abstr>
```

[26] :
```
Stack.push 0 maPile ;;
maPile ;;
```

[26] :
```
-  : unit = ()
-  : int Stack.t = <abstr>
```

[27] :
```
for i = 1 to 6 do
  Stack.push i maPile
done ;;
```

```
[27] : -  : unit = ()
```

```
[28] : while not (Stack.is_empty maPile) do
         print_int (Stack.pop maPile) ;
         print_char ' '
       done ;
       print_newline ()
```

```
6 5 4 3 2 1 0
```

```
[28] : -  : unit = ()
```

3 Files

3.1 Définition d'une file

Une *file* (*queue* en anglais) est aussi une structure de données linéaire dynamique, mais l'insertion d'un élément s'effectue d'un côté de la file alors que la suppression s'effectue de l'autre côté. On parle de structure de type *FIFO* (*First In, First Out*).

Cette structure nécessite :

- un constructeur permettant de créer une pile vide ;
- une fonction permettant d'*enfiler* (*push/add*) un élément, c'est-à-dire d'ajouter un élément à la fin de la file ;
- une fonction permettant de *défiler* (*pop/take*) un élément, c'est-à-dire de supprimer l'élément au début de la file et de le renvoyer ;
- une fonction permettant de tester si la file est vide.

3.2 Implémentation d'une file à l'aide de deux listes

Il est plus délicat d'implémenter la structure de file, car il faudrait pouvoir accéder en temps constant aux deux extrémités de la file.

L'implémentation proposée dans ce paragraphe n'atteint pas tout à fait cet objectif. On va ici utiliser deux listes : une première liste utilisée pour insérer des éléments, la seconde pour en enlever.

```
[29] : type 'a file = {
           mutable entree  : 'a list  ;
           mutable sortie  : 'a list
         } ; ;
```

```
[29] : type 'a file = { mutable entree  : 'a list ; mutable sortie  : 'a list ; }
```

Seule l'opération *défiler* pose des difficultés. En effet, si la liste de sortie est vide, il faut basculer les éléments de la liste d'entrée dans la liste de sortie :

```
[30] : let creer_file () = {entree = [] ; sortie = []} ; ;
```

```
[30] : val creer_file  : unit -> 'a file = <fun>
```

```
[31] : let est_vide f = f.entree = [] && f.sortie = [] ; ;

[31] : val est_vide  : 'a file -> bool = <fun>

[32] : let enfile f x =  f.entree <- x : :f.entree ; ;

[32] : val enfile  : 'a file -> 'a -> unit = <fun>

[33] : let miroir lst =
         let rec aux lst acc =
           match lst with
           | [] -> acc
           | t : :q -> aux q (t : :acc)
         in
         aux lst [] ; ;

       let rec defile f =
         match f.sortie with
         | t : :q -> f.sortie <- q  ; t
         | [] ->
            match f.entree with
            | [] -> failwith "File Vide"
            | _ -> f.sortie <- miroir f.entree  ;
                   f.entree <- [] ;
                   defile f
        ; ;

[33] : val miroir  : 'a list -> 'a list = <fun>
       val defile  : 'a file -> 'a = <fun>
```

Toutes les opérations s'effectuent en temps constant, sauf l'opération *défiler* lorsque la liste de sortie est vide. Dans ce cas, la complexité est linéaire par rapport au nombre d'éléments de la file.

Néanmoins, là encore ce cas ne se produit pas trop souvent. En effet, lorsqu'il y a n éléments dans la file et que la liste de sortie est vide, le coût est $O(n)$, mais les $n-1$ prochaines opérations *défiler* seront réalisées en temps constant. Par conséquent, pour ces n opérations, le coût total est $O(n)$ donc le coût moyen est $O(1)$. La complexité amortie de l'opération *défiler* est $O(1)$.

3.3 Implémentation d'une file à l'aide d'un tableau

Pour implémenter une file à l'aide d'un tableau, on considère le tableau des éléments comme circulaire : on utilise deux indices qui indiquent les positions de la tête de file (le prochain à sortir) et de la queue de file (la position du prochain élément inséré).

```
[34] : type 'a file =  {
           mutable longueur  : int  ;
           mutable debut  : int  ;
           mutable fin  : int  ;
           file  : 'a array } ; ;
```

```
[34] : type 'a file = {
         mutable longueur  : int ;
         mutable debut  : int ;
         mutable fin  : int ;
         file  : 'a array ;
       }
```

```
[35] : let creer_file n i =  {
           longueur = 0  ;
           debut = 0 ;
           fin = 0  ;
           file = Array.make n i
         } ; ;
```

```
[35] : val creer_file  : int -> 'a -> 'a file = <fun>
```

```
[36] : let est_vide f = f.longueur = 0  ; ;
```

```
[36] : val est_vide  : 'a file -> bool = <fun>
```

```
[37] : let enfile x f =
         let n = Array.length f.file in
         if f.longueur = n
         then failwith "File pleine"
         else
           begin
             f.file.(f.fin) <- x  ;
             f.fin <- (f.fin + 1) mod n  ;
             f.longueur <- f.longueur + 1
           end
       ; ;
```

```
[37] : val enfile  : 'a -> 'a file -> unit = <fun>
```

```
[38] : let defile f =
         let n = Array.length f.file in
         if est_vide f
         then failwith "File vide"
         else
           begin
             let x = f.file.(f.debut) in
             f.debut <- (f.debut + 1) mod n  ;
             f.longueur <- f.longueur - 1 ;
             x
           end
       ; ;
```

```
[38] : val defile  : 'a file -> 'a = <fun>
```

Toutes ces opérations s'effectuent en temps constant $O(1)$, sauf la création de la file qui s'effectue en $O(c)$ où c est la capacité de la pile.

3.4 Module `Queue`

Ocaml dispose d'un module appelé `Queue` permettant de créer et de manipuler des files :

```
[39] : let maFile = Queue.create () ; ;
```

```
[39] : val maFile  : '_weak5 Queue.t = <abstr>
```

```
[40] : for i = 1 to 15 do
         Queue.add i maFile
       done ; ;
```

```
[40] : -  : unit = ()
```

```
[41] : while not (Queue.is_empty maFile) do
         print_int (Queue.take maFile) ;
         print_char ' '
       done ;
       print_newline() ; ;
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
[41] : -  : unit = ()
```

4 Files de priorité

Une file de priorité suit le même principe qu'une file, mais chaque élément de la file est associé à une priorité, en général représentée par un entier. La file de priorité stocke donc des couples (e, p) où e est un élément et p un entier (la priorité).

Le prochain élément à sortir de la file est celui qui a la plus grande priorité (dans le cas d'une file de priorité max ; on parle sinon de file de priorité min).

Les opérations primitives sur les files de priorité sont les suivantes :

- création d'une file de priorité vide ;
- test permettant de savoir si une file de priorité est vide ;
- suppression et renvoi de l'élément de plus grande priorité d'une file non vide ;
- ajout d'un élément avec une priorité donnée ;
- éventuellement modification de la priorité d'un élément (il faut alors imposer que les éléments de la file soient distincts).

5 Exercice

5.1 Évaluation d'expressions arithmétiques postfixées

On écrit habituellement les expressions arithmétiques sous forme *infixe*, en faisant figurer les opérateurs entre leur deux opérandes. Néanmoins, cette notation est ambiguë si on ne définit pas les priorités entre opérateurs : $1 + 2 \times 3$ peut représenter $(1 + 2) \times 3$ ou $1 + (2 \times 3)$. Il faut alors introduire des règles de priorité ou des parenthèses.

La notation *postfixée* consiste à écrire d'abord les opérandes, puis leur opérateur ; par exemple $3+4$ s'écrit «3 4 +» ; $(2+4)\times 3$ s'écrit «2 4 + 3 ×». L'avantage de cette notation est que les expressions sont alors non ambiguës : pas besoin de parenthèses ni de règles de priorité.

L'évaluation d'une telle expression est réalisée à l'aide d'une pile. On lit l'expression de gauche à droite : lorsqu'on lit un entier, on l'empile ; lorsqu'on lit un opérateur, on dépile deux éléments (ses deux opérandes), on effectue l'opération et on empile le résultat.

A la fin de l'évaluation, la pile ne contient plus qu'un élément, qui est le résultat de son évaluation.

On va représenter une expression sous la forme d'une liste d'opérateurs et d'entiers.

— Définir un type `lexeme` permettant de représenter soit un opérateur binaire, soit un entier.
— Écrire une fonction `evalue : lexeme list -> int` qui prend en argument une expression sous forme de liste et qui renvoie le résultat de cette expression.