Piles et files

1 Piles

1.1 Définition

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 :

— 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.

1.2 Implémentation 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 :

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

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

On définit ensuite les opérations primitives.

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

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

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

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

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

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

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

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

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

1.3 Implémentation 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.

```
type 'a pile = {
    mutable hauteur : int ;
    pile : 'a array
  };;
```

```
type 'a pile = { mutable hauteur : int; pile : 'a 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.

```
let creer_pile c i =  {
    hauteur = 0 ;
    pile = Array.make c i
  };;
```

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

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

```
let est_vide p = p.hauteur = 0;;
```

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

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

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

```
let depile p =
  if est_vide p
  then failwith "Pile vide"
  else
    begin
      p.hauteur <- p.hauteur - 1 ;
      p.pile.(p.hauteur)
    end
;;
```

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

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.

1.4 Module `Stack`

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

```
let maPile = Stack.create();;
```

```
val maPile : '_weak1 Stack.t = <abstr>
```

```
Stack.push 0 maPile;;
maPile;;
```

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

```
for i = 1 to 6 do
  Stack.push i maPile
done;;
```

```
- : unit = ()
```

```
while not (Stack.is_empty maPile) do
  print_int (Stack.pop maPile);
  print_char ' '
done;;
```

```
- : unit = ()
```

2 Files

2.1 Définition

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.

2.2 Implémentation 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.

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

```
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 :

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

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

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

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

```
let enfile f x =  f.entree <- x::f.entree;;
```

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

```
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
;;
```

```
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, 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 $O(1)$. On dit que la complexité *amortie* de l'opération *défiler* est $O(1)$.

2.3 Implémentation 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).

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

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

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

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

```
let est_vide f = f.longueur = 0 ;;
```

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

```
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
;;
```

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

```
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
;;
```

```
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.

2.4 Module `Queue`

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

```
let maFile = Queue.create ();;
```

```
val maFile : '_weak2 Queue.t = <abstr>
```

```
for i = 1 to 15 do
  Queue.add i maFile
done;;
```

```
- : unit = ()
```

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

```
6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- : unit = ()
```

3 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 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).

4 Exercice

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.