Chapitre 5

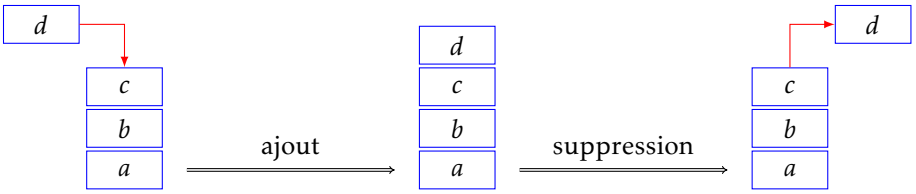
option informatique

Piles et files

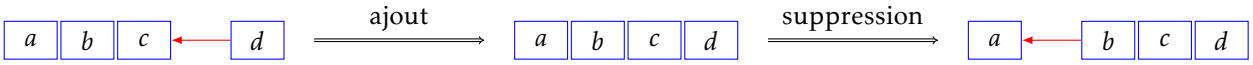
1. Introduction

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures de données fondamentales en informatique. Ce sont toutes deux des structures linéaires dynamiques, qui se distinguent par les conditions d'ajout et d'accès aux éléments :

– Les piles sont fondées sur le principe du « dernier arrivé, premier sorti » ; on les dit de type LIFO (*Last In, First Out*). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.



– Les files sont fondées sur le principe du « premier arrivé, premier sorti » ; on les dit de type FIFO (*First In, First Out*). C'est le principe de la file d'attente devant un guichet.



Ces deux types de données sont fondamentaux en informatique ; une pile est par exemple utilisée par le compilateur pour gérer l'exécution d'une fonction, et une file est utilisée pour gérer les requêtes sur un réseau. Les exemples sont multiples, nous en verrons nous-mêmes quelques-uns.

Primitives

Ces deux structures de données ont des spécifications très semblables. Toutes deux nécessitent :

– un constructeur permettant de créer une pile ou une file vide ;

– une fonction d'ajout d'un élément ;

– une fonction de suppression d'un élément et son renvoi ;

– une fonction permettant de tester si la pile ou la file est vide.

Notons que par essence ce sont des structures de données *mutables*.

Nous discuterons à la fin de ce document des possibilités d'implémentation en Caml de ces structures de données et de ces primitives ; dans l'immédiat, nous allons utiliser deux modules de la bibliothèque standard : `stack` et `queue`, qui permettent de gérer piles et files en Caml.

Attention, les fonctions qui font partie de ces modules ne sont pas accessibles directement ; pour les utiliser, deux possibilités vous sont offertes : ou bien préfixer les noms des fonctions du nom du module d'où elles proviennent (`stack__` ou `queue__`), ou bien importer l'ensemble des fonctions d'un de ces modules à l'aide de la commande `#open "stack"` ou `#open "queue"` (notez bien que le caractère # fait partie de l'instruction), ce qui permet d'éviter d'avoir à écrire le préfixe.

- **Les primitives du module stack**

Les piles d'éléments de type *'a* ont pour type *'a t*.
La fonction **new**, de type *unit -> '_a t*, crée une nouvelle pile, vide au départ.
La fonction **push**, de type *'a -> 'a t -> unit*, permet d'empiler un élément au sommet d'une pile.
La fonction **pop**, de type *'a t -> 'a*, élimine le sommet de la pile et le renvoie.

```
# #open "stack" ;;
# let pile = new () ;;
pile : '_a t = <abstr>
# for i = 1 to 5 do push i pile done ;;
- : unit = ()
# for i = 1 to 5 do print_int (pop pile) done ;;
54321- : unit = ()
```

Il n'existe pas de fonction permettant de déterminer si une pile est vide, mais l'exception **Empty** est déclenchée lorsqu'on tente d'appliquer **pop** à une pile vide.
La raison de cette absence provient de la possibilité en Caml de *rattraper une exception*, à l'aide de la syntaxe : **try expr with filtrage**. Cette construction retourne la valeur de **expr** si aucune exception n'est déclenchée ; dans le cas contraire, la valeur exceptionnelle est filtrée par les motifs du filtrage.

Ainsi, il est facile de définir une fonction qui détermine si une pile est vide :

```
let est_vide p =
  try let x = pop p in push x p ; false with Empty -> true ;;
```

On s'inspirera de ce modèle à chaque fois qu'il faudra déterminer si une pile est vide.

- **Les primitives du module queue**

Les files d'éléments de type *'a* ont pour type *'a t*.
La fonction **new**, de type *unit -> '_a t*, crée une nouvelle file, vide au départ.
La fonction **add**, de type *'a -> 'a t -> unit*, permet d'ajouter un élément en queue de file.
La fonction **take**, de type *'a t -> 'a*, élimine la tête de la file et le renvoie.

```
# #open "queue" ;;
# let file = new () ;;
file : '_a t = <abstr>
# for i = 1 to 5 do add i file done ;;
- : unit = ()
# for i = 1 to 5 do print_int (take file) done ;;
12345- : unit = ()
```

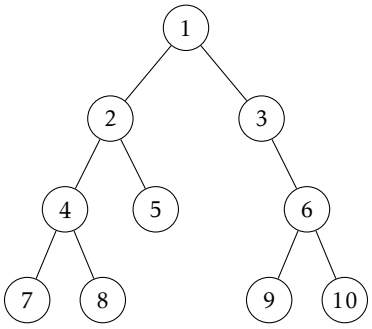
On peut ajouter la fonction **peek**, qui renvoie la tête de file, *mais sans modifier cette dernière*. Cette fonction est utile pour déterminer si une file est vide, sans la modifier dans le cas contraire :

```
let est_vide f =
  try peek f ; false with Empty -> true ;;
```

2. Exemples d'application des listes et des piles

2.1 Parcours hiérarchique d'un arbre

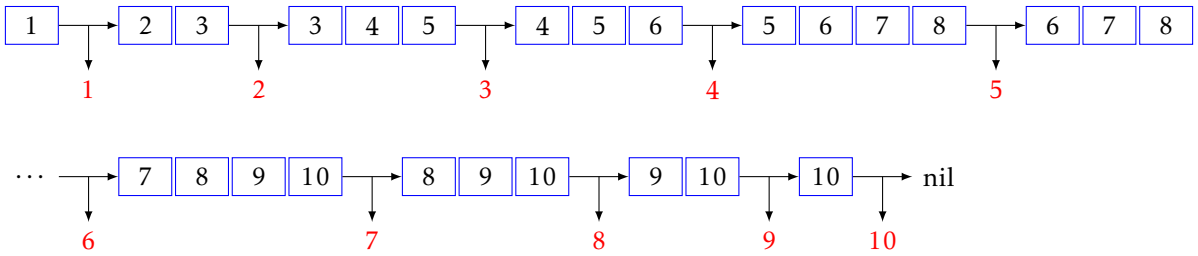
Revenons maintenant à un problème que nous avions laissé en suspend : le parcours en largeur d'un arbre. Il s'agit de parcourir les nœuds et feuilles de l'arbre en les classant par profondeur croissante (et en général de la gauche vers la droite pour une profondeur donnée). Dans l'exemple qui suit, il correspond à l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :



La solution de ce problème consiste à utiliser une file dans laquelle figurera au départ le racine de l'arbre, et à adopter la règle suivante :

1. la tête de liste est traitée ;
2. les fils gauche et droit de cet élément, s'ils sont présents dans l'arbre, sont ajoutés en queue de liste ;

et à procéder ainsi jusqu'à exhaustion de la file.
Dans l'exemple ci-dessus, la file va évoluer de la façon suivante :



Lorsqu'on adopte le type suivant pour définir un arbre :

```
type 'a btree = Nil | Node of 'a * 'a btree * 'a btree ;;
```

la fonction de parcours hiérarchique prend la forme suivante (dans le cas d'une étiquette de type *int*) :

```
let parcours arbre =
  let file = new () in
  add arbre file ;
  let rec aux () = match (take file) with
    | Nil                       -> aux ()
    | Node (r, fils_g, fils_d) -> print_int r ;
                                  add fils_g file ;
                                  add fils_d file ;
                                  aux ()
  in try aux () with Empty -> () ;;
```

Nous verrons l'année prochaine que ce type de parcours est un cas particulier du *parcours en largeur* d'un graphe.

2.2 Expressions algébriques postfixées

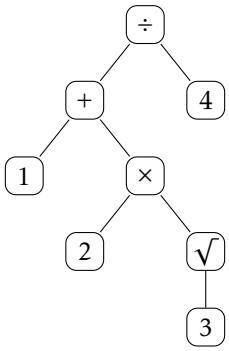
• Représentations d'une expression mathématique

Suivant l'usage courant, une expression mathématique est représentée sous sa forme *infixe* : les opérateurs binaires se placent entre leurs deux arguments. Ceci nécessite l'usage de parenthèses pour préciser la hiérarchie des calculs : $(1+2)\times 3$ est différent de $1+(2\times 3)$.

Par exemple, la formule $\dfrac{1+2\sqrt{3}}{4}$ est représentée par la liste de caractères : (1 + (2 × (√ 3))) ÷ 4.

Une expression est dite *postfixée* lorsque les opérateurs suivent immédiatement la liste de leur(s) argument(s). Par exemple, cette même formule sera représentée sous forme postfixée par la liste : 1 2 3 √ × + 4 ÷, ce qui doit être compris comme : (1 (2 (3 √) ×) +) 4 ÷, mais l'usage des parenthèses est ici superflu puisque l'ordre dans lequel les opérations apparaissent est l'ordre dans lequel elles doivent être effectuées.

De même, une expression est dite *préfixée* lorsque les opérateurs précèdent immédiatement la liste de leurs arguments. Ainsi, la formule donnée en exemple sera représentée sous forme préfixée par : $\div + 1 \times 2 \sqrt{} 3\ 4$, ce qui doit être compris comme : $\div \Big(+ 1 \big(\times 2 (\sqrt{} 3) \big) \Big) 4$, mais là encore, les parenthèses sont inutiles.
On peut noter que l'expression postfixée d'une formule mathématique correspond au parcours en profondeur postfixe de l'arbre associé à cette expression et l'expression préfixée au parcours en profondeur suffixe :



• Évaluation d'une expression postfixée

La notation préfixe, inventée par le mathématicien polonais Lukasiewicz, est également appelée la notation polonaise ; par opposition, la notation postfixe porte le nom de *notation polonaise inverse* (ou encore RPN, pour *Reverse Polish Notation*). Elle est utilisée dans certains langages de programmation ainsi que pour certaines calculatrices, notamment celles de la marque Hewlett-Packard. Nous allons voir comment, à l'aide d'une pile, évaluer une expression algébrique postfixée.

On commence par définir le type suivant :

```
type lexeme = Nombre of float
            | Op_binaire of float -> float -> float
            | Op_unaire of float -> float ;;
```

et on suppose avoir en notre possession un *analyseur lexical*, c'est à dire une fonction **analyseur** de type *string -> lexeme list* qui à une chaîne de caractères associe la liste des lexèmes qui la composent. Voici son rôle sur un exemple :

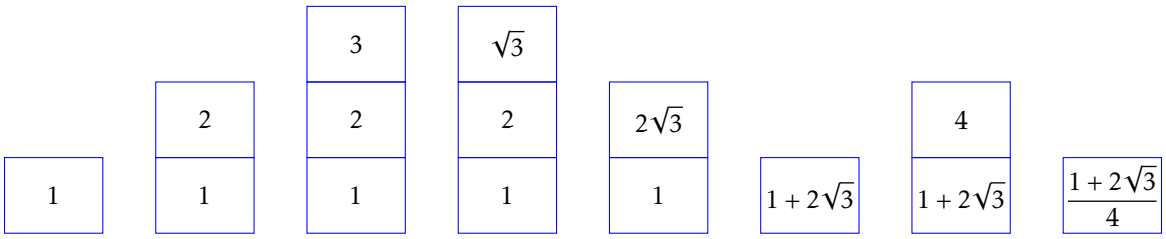
```
# analyseur "1 2 3 sqrt * + 4 /" ;;
-: lexeme list = [ Nombre 1. ; Nombre 2. ; Nombre 3. ; Op_unaire
sqrt ; Op_binaire mult_float ; Op_binaire add_float ; Nombre 4. ;
Op_binaire div_float ]
```

Nous n'aborderons pas ici le problème de la définition d'un tel analyseur lexical.

L'évaluation d'une expression postfixée se déroule en parcourant la liste des lexèmes et en suivant les règles suivantes :

- si la tête de la liste est un nombre a, on l'empile ;
- si la tête est un opérateur unaire f, on dépile le sommet a et on empile $f(a)$;
- si la tête est un opérateur binaire f, on dépile les deux éléments a et b les plus hauts, et on empile $f(a,b)$.

Par exemple, la pile associée à l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$ va évoluer comme suit :



La fonction correspondante est la suivante :

```
let evalue lst =
  let pile = new () in
  let rec aux = function
    | []                -> pop pile
    | (Nombre a)::q     -> push a pile ; aux q
    | (Op_unaire f)::q  -> let a = pop pile in
                           push (f a) pile ; aux q
    | (Op_binaire f)::q -> let b = pop pile in let a = pop pile in
                           push (f a b) pile ; aux q
  in aux lst ;;
```

Il reste à combiner la fonction d'évaluation avec l'analyseur lexical pour obtenir la fonction souhaitée :

```
# evalue (analyseur "1 2 3 sqrt * + 4 /") ;;
- : float = 1.11602540378
```

Remarque. Que se passe-t-il dans le cas d'une expression syntaxiquement incorrecte ? Deux cas de figure peuvent se présenter : ou bien on tente au cours du traitement de dépiler une pile vide, auquel cas l'exception **`Empty`** sera déclenchée, ou bien la pile contient plus d'un élément à la fin du traitement.
Cette dernière erreur n'étant pas détectée par notre fonction actuelle, nous allons la modifier en commençant par créer une nouvelle exception :

```
# exception Syntax_Error ;;
Exception Syntax_Error defined.
```

L'instruction **`raise`** permet ensuite de lever une exception. Combiné au mécanisme de rattrapage d'une exception, ceci nous conduit à modifier notre fonction d'évaluation comme suit :

```
let evalue lst =
  let pile = new () in
  let rec aux = function
    | []                -> let rep = pop pile in
                           try pop pile ; raise Syntax_Error with Empty -> rep
    | (Nombre a)::q     -> push a pile ; aux q
    | (Op_unaire f)::q  -> let a = pop pile in
                           push (f a) pile ; aux q
    | (Op_binaire f)::q -> let b = pop pile in let a = pop pile in
                           push (f a b) pile ; aux q
  in try aux lst with Empty -> raise Syntax_Error ;;
```

Deux exemples d'expressions syntaxiquement incorrectes, la première car la pile finale contient deux éléments, la seconde car on tente de dépiler une pile vide :

```
# evalue (analyseur "0 1 2 3 sqrt * + 4 /") ;;
Uncaught exception: Syntax_Error
# evalue (analyseur "2 3 sqrt * + 4 /") ;;
Uncaught exception: Syntax_Error
```

3. Implémentation des files et des piles

3.1 Implémentation d'une pile

Définir une structure de donnée apte à représenter une pile nécessite de pouvoir accéder en temps constant au sommet de la pile, à la fois pour ajouter et pour ôter cet élément (principe LIFO). Les listes apparaissent évidemment comme de bons candidats, si ce n'est qu'étant une structure de nature impérative, les piles exigent d'en faire des objets mutables. Voila pourquoi on définit le type suivant :

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

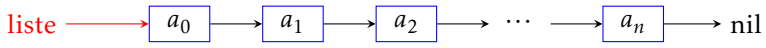
Une fois ceci fait, il nous reste à définir les opérations primitives ; ce n'est guère difficile :

```
exception Empty ;;

let new () = {Liste = []} ;;

let push x p = p.Liste <- x::p.Liste ;;

let pop p = match p.Liste with
  | []   -> raise Empty
  | t::q -> (p.Liste <- q ; t) ;;
```

3.2 Implémentation d'une file

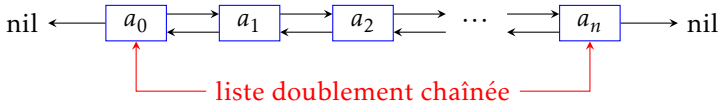
Définir une structure de données concrète apte à représenter une file est plus délicat car il est cette fois nécessaire de pouvoir accéder en temps constant aussi bien à la tête et à la queue de la file (principe FIFO). Nous allons décrire une solution, basée sur la notion de liste *doublement chaînée*.

Revenons un instant sur la description d'une liste : chaque élément de celle-ci connaît son successeur, mais pas son prédécesseur. On peut donc la représenter de la façon suivante :



et il est facile d'imaginer que les cellules dessinées représentent des emplacements en mémoire et les flèches des pointeurs vers l'adresse de ces emplacements.

Les listes doublement chaînées sont tout simplement des listes pour lesquelles chaque cellule pointe non seulement vers la cellule suivante mais aussi vers la cellule précédente :



et pour lesquelles on accède en temps constant aussi bien au premier et au dernier élément de cette liste.

Nous allons les définir de la façon suivante :

```
type 'a cell = {Valeur : 'a ;
                mutable Avant : 'a liste ;
                mutable Apres : 'a liste}
and 'a liste = Nil | Cellule of 'a cell ;;
```

Notez que les pointeurs **Avant** et **Apres** doivent être des objets mutables pour pouvoir les modifier en cas d'ajout ou de suppression d'un élément de la liste.

Une file n'est alors qu'un enregistrement mutable qui pointe vers la tête et la queue d'une liste doublement chaînée :

```
type 'a file = {mutable Tete : 'a liste ;
                mutable Queue : 'a liste} ;;
```

Il reste à définir les opérations primitives :

```
exception Empty ;;

let new () = {Tete = Nil; Queue = Nil} ;;

let add x f =
  let c = Cellule {Valeur = x; Avant = f.Queue; Apres = Nil} in
    match f.Queue with
      | Nil       -> (f.Tete <- c ; f.Queue <- c)
      | Cellule d -> (d.Apres <- c ; f.Queue <- c) ;;

let take f = match f.Tete with
  | Nil -> raise Empty
  | Cellule c -> match c.Apres with
      | Nil       -> (f.Tete <- Nil ; f.Queue <- Nil ; c.Valeur)
      | Cellule d -> (d.Avant <- Nil ; f.Tete <- c.Apres ; c.Valeur) ;;

let peek f = match f.Tete with
  | Nil       -> raise Empty
  | Cellule c -> c.Valeur ;;
```

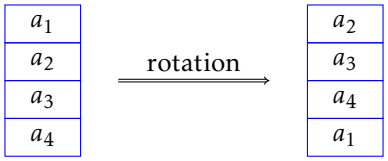
4. Exercices

Exercice 1 On dispose d'une pile d'assiettes bleues ou rouges numérotées disposées dans le désordre. Comment procéder pour former une pile dans laquelle les assiettes bleues sont situées sous les assiettes rouges, mais en faisant en sorte que pour chacune des deux couleurs l'ordre relatif ne soit pas modifié ? (Autrement dit, si l'assiette bleue i est située sous l'assiette bleue j dans la pile initiale, ce sera toujours le cas dans la pile finale.)
On définit les types :

```
type couleur = Bleue | Rouge and assiette == couleur * int ;;
```

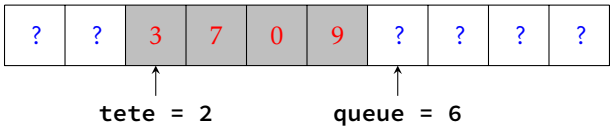
Rédiger une fonction de type *assiette t –> unit* qui range une pile d'assiette en disposant les assiettes bleues sous les assiettes rouges.

Exercice 2 Écrire une fonction de type *'a t –> unit* qui réalise la rotation d'une pile, c'est à dire que l'élément au sommet se retrouve tout en bas de la pile (on pourra utiliser une pile auxiliaire) :

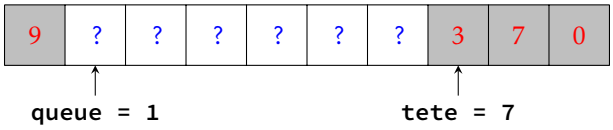


Écrire ensuite une fonction effectuant la rotation inverse.

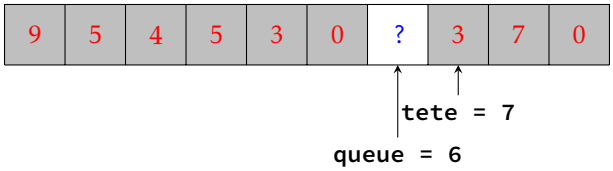
Exercice 3 On se propose ici de décrire une implémentation d'une file à l'aide d'un tableau de taille n. Une file sera constituée d'éléments contigus de ce dernier ; il suffira donc pour représenter la liste de gérer deux indices : l'indice **tete** marquera le début de la file, et **queue** en marquera sa fin.
Voici par exemple une représentation de la liste $(3, 7, 0, 9)$ au sein d'un tableau de 10 cases :



Attention, on utilise un tableau circulaire, il est donc tout à fait possible d'avoir **queue < tete**, comme par exemple dans la représentation suivante de cette même liste :



Une conséquence de ce choix est que l'égalité **tete = queue** ne permet pas de distinguer entre la file vide et la file pleine ; pour palier à ce problème, on s'interdit de remplir complètement le tableau : ainsi, on décrète qu'une file qui contient $n-1$ éléments est pleine, situation caractérisée par l'égalité : **(queue + 1) mod n = tete**.
Un exemple de liste pleine :



Ceci nous conduit à définir deux exceptions :

```
exception Empty ;;
exception Full ;;
```

ainsi que le type :

```
type 'a file = {N : int ;
                Tab : 'a vect ;
                mutable Tete : int ;
                mutable Queue : int} ;;
```

Définir les fonctions primitives **new**, **add** et **take** associées à ces définitions.

Exercice 4 Nous avons montré comment utiliser une file pour effectuer le parcours en largeur d'un arbre binaire. Qu'obtient-on si on utilise une pile à la place de la file ?

Exercice 5 Les nombres de Hamming sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5 :

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, …

Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (songez que le 1999e entier de Hamming est égal à 8 100 000 000 et le 2000e à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).
On adopte donc la démarche suivante : on utilise trois files f_2, f_3, f_5 contenant initialement le nombre 1, et on suit la démarche suivante :

(i) on détermine le plus petit des trois têtes de file, que l'on note k et que l'on imprime à l'écran ;

(ii) on retire cet élément des files où il se trouve ;

(iii) on insère en queue des files f_2, f_3 et f_5 les entiers $2k$, $3k$ et $5k$.

Vous l'avez compris : cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.

a) Rédiger une fonction Caml permettant l'affichage des n premiers nombres de Hamming.

b) L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files. Modifier votre fonction pour que cela ne soit plus le cas.

Exercice 6 On dit qu'une permutation $(a_1 \ \ a_2 \ \ \cdots \ \ a_n)$ de $(1 \ \ 2 \ \ \cdots \ \ n)$ peut être engendrée par une pile lorsque il est possible, à partir de la séquence d'entrée $(1 \ \ 2 \ \ \cdots \ \ n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1 \ \ a_2 \ \ \cdots \ \ a_n)$ en utilisant les opérations suivantes :

- empiler l'élément suivant de la séquence d'entrée ;
- ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2 \ \ 3 \ \ 1)$ est engendrée par la suite d'opérations : EEDEDD.

a) Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?

$$(3\ \ 1\ \ 2)\quad (3\ \ 4\ \ 2\ \ 1)\quad (4\ \ 5\ \ 3\ \ 7\ \ 2\ \ 1\ \ 6)\quad (3\ \ 5\ \ 7\ \ 6\ \ 8\ \ 4\ \ 9\ \ 2\ \ 10\ \ 1)$$

b) Montrer que s'il existe un triplet $(i,j,k) \in [\![1,n]\!]^3$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation $(a_1\ \ a_2\ \ \cdots\ \ a_n)$ n'est pas engendrable par une pile.

c) Écrire une fonction Caml déterminant si une permutation peut être engendrée par une pile. Dans le cas d'une réponse positive, la fonction affichera la suite d'opérations permettant de la produire. Les permutations seront représentées par le type *int list*.

d) Montrer enfin que toute permutation peut être engendrée à l'aide de deux piles, et rédiger la fonction Caml correspondante.