TD structures de données : corrigé
MP2I informatique

I Centrale 2016 : implémentation impérative d'une file par un tableau

On veut implémenter une file d'attente à l'aide d'un vecteur circulaire. On définit pour cela un type particulier nommé `file` par

```
type 'a file={tab:'a array ; mutable deb: int ; mutable fin: int ; mutable vide: bool}
```

`deb` indique l'indice du premier élément dans la file et `fin` l'indice qui suit celui du dernier élément de la file, `vide` indiquant si la file est vide. Les éléments sont rangés depuis la case `deb` jusqu'à la case précédent `fin` en repartant à la case 0 quand on arrive au bout du vecteur (cf exemple). Ainsi, on peut très bien avoir l'indice `fin` plus petit que l'indice `deb`. Par exemple, la file figure 5 contient les éléments 4, 0, 1, 12 et 8, dans cet ordre, avec `fin=2` et `deb=9`.

		fin ↓							deb ↓		
12	8	7	2	5	3	1	16	3	4	0	1

Figure 5 Un exemple de file où `fin` < `deb`

On rappelle qu'un champ mutable peut voir sa valeur modifiée. Par exemple, la syntaxe `f.deb <- 0` affecte la valeur 0 au champ `deb` de la file `f`.

1) Écrire une fonction `ajoute` de signature `'a file -> 'a -> unit` telle que `ajoute f x` ajoute `x` à la fin de la file d'attente `f`. Si c'est impossible, la fonction devra renvoyer un message d'erreur, en utilisant l'instruction `failwith "File pleine"`.

2) Écrire une fonction `retire` de signature `'a file -> 'a` telle que `retire f` retire l'élément en tête de la file d'attente et le renvoie. Si c'est impossible, la fonction devra renvoyer un message d'erreur.

3) Quelle est la complexité de ces fonctions ?

▶ Les fonctions suivantes sont bien sûr en O(1):

```
let ajoute f e =
  if f.deb = f.fin && not f.vide then failwith "File pleine"
  else (f.tab.(f.fin) <- e;
        f.fin <- (f.fin + 1) mod Array.length f.tab;
        f.vide <- false);;

let retire f =
  if f.vide then failwith "File vide"
  else let res = f.tab.(f.deb) in
       (f.deb <- (f.deb + 1) mod Array.length f.tab;
       f.vide <- f.deb = f.fin;
       res);;
```

II Dérécursivation

Il est possible de transformer n'importe qu'elle fonction récursive `f` par un code impératif (qui n'utilise pas de récursivité). Pour cela on peut simuler les appels récursifs de `f` en utilisant une pile `p`[1] telle que:

- chaque élément de `p` est un n-uplet qui correspond aux arguments d'un appel récursif sur `f`
- tant que `p` n'est pas vide, on dépile l'élément `(a1, ..., an)` du dessus et on fait la même chose que l'appel `f(a1, ..., an)` aurait fait (en empilant dans `p` au lieu d'effectuer des appels récursifs)

On utilisera le module Stack de OCaml, dont on donne une partie de la documentation officielle (trouvable sur internet):

```
val create : unit -> 'a t
```

Return a new stack, initially empty.

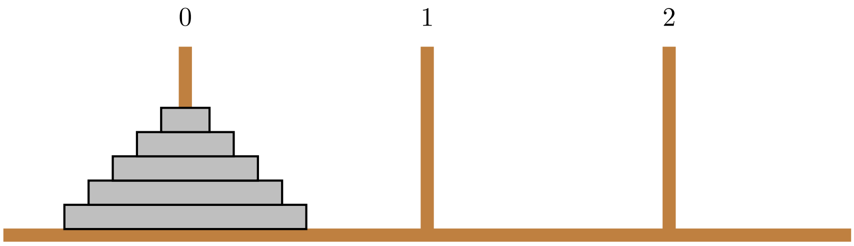
```
val push : 'a -> 'a t -> unit
```

`push x s` adds the element `x` at the top of stack `s`.

```
val pop : 'a t -> 'a
```

`pop s` removes and returns the topmost element in stack `s`, or raises `Stack.Empty` if the stack is empty.

Le but de cet exercice et de dérécursiver l'algorithme de résolution des tours de Hanoï.



On rappelle que le problème des tours de Hanoï consiste à déplacer les n disques de la tige 0 sur la tige 2, avec les règles suivantes:

- On ne peut déplacer qu'un disque à la fois (celui du dessus).
- Il est interdit de poser un disque sur un disque de taille inférieure.

1. Se souvenir de la solution récursive des tours de Hanoï.
2. Écrire une fonction sans récursivité `hanoi : int -> unit` telle que `hanoi n` affiche la suite de mouvements nécessaire à la résolution des tours de Hanoï avec n disques.
   ► On utilise une pile `p` dont chaque élément est un 3-uplet: le nombre k de disques à déplacer, depuis le piquet deb vers le piquet fin:

```
let hanoi n =
  let p = Stack.create () in
  Stack.push (n, 0, 2) p;
  while not (Stack.is_empty p) do
    let (k, deb, fin) = Stack.pop p in
    if k = 1 then ( print_int deb;
                    print_string " ";
                    print_int fin;
                    print_string "\n" )
    else let mid = 3 - deb - fin in
         ( Stack.push (k - 1, mid, fin) p;
           Stack.push (1, deb, fin) p;
           Stack.push (k - 1, deb, mid) p )
  done;;
```

3. Quel est le nombre de mouvements utilisé par cet algorithme? (Est-ce optimal?)
   ► Soit $C(n)$ le nombre de mouvements utilisé par notre algorithme (les algorithmes récursif et itératif effectue le même nombre de mouvements).
   Comme déplacer n disques demande de déplacer deux fois $n-1$ disques et 1 disque, $C(n) = 2C(n-1) + 1$. De plus il est clair que $C(1) = 1$.
   Soit $u_n = 2^n - 1$. Alors $u_1 = 2 - 1 = 1$ et $2u_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1 = u_n$. Comme u_n vérifie la même équation de récurrence que $C(n)$ avec la même condition initiale, $u_n = C(n) = 2^n - 1$, $\forall n$.

[1] C'est ce que fait votre ordinateur lorsque vous programmez une fonction récursive