DS3 MP2I : piles et files ; complexité

Aucun appareil électronique n'est autorisé.

Solution. □

1 Structure de file immuable

Avant de répondre aux questions de complexité, consulter la section 4.1.
Les fonctions du module `List` sont interdites.

Question 1.

Écrire la fonction `rev : 'a list -> 'a list` qui inverse une liste.

Solution. Code

```
let rev l =
  let rec aux acc l =
    match l with
    | [] -> acc
    | x :: r -> aux (x :: acc) r
  in
  aux [] l
```

□

Question 2.

Donner précisément le nombre d'ajouts en tête de liste (par `::`) effectués pour l'appel `rev q` si la liste `q` est de longueur n.

Solution. Exactement n ajouts avec `::`. En effet, on a une relation de récurrence de la forme

$$C_n = C_{n-1} + 1 \text{ si } n > 0 \text{ et } C_0 = 0$$

si C_n compte le nombre d'ajouts en tête de liste pour un appel à `rev` avec une liste de longueur n.

□

Nous implémentons une structure de file immuable. Dans un fichier d'interface **files.mli** [1] écrivons :

```
  type 'a queue
  val empty : 'a queue
  val is_empty: 'a queue -> bool
  val enqueue: 'a queue -> 'a -> 'a queue
  val dequeue: 'a queue -> 'a * 'a queue
```

1. Les fichiers **.ml/.mli** jouent un rôle analogue aux fichiers **.c/.h** que l'on trouve en C.

Nous voulons implanter ces primitives. Pour réaliser une file immuable on se sert de deux piles immuables (on utilise des listes OCaml pour cela). Dans la première pile (`rear`), on ajoute les éléments qui entrent dans la file ; dans la seconde (`front`), on retire les éléments qui sortent de la file. Lorsque la seconde pile est épuisée, on y déplace tous les éléments de la première pile *en les inversant*.

On pose donc tout d'abord :

```
type 'a queue = {
  front: 'a list; (*liste où on retire les éléments*)
  rear:  'a list; (*liste où on ajoute les éléments*)
}
```

Par exemple, la file qui contient les éléments $1, 2$ dans cet ordre (1 est le prochain élément défilé, 2 est le dernier arrivé) peut être représentée par

```
let f12 = {front = []; rear = [2;1]};;
let f12 = {front = [1]; rear = [2]};;
let f12 = {front = [1;2]; rear = []};;
```

Question 3.

Avec notre structure de données `queue` de combien de façons différentes peut-on représenter une file qui contient les éléments $1, 2, \ldots, n$ dans cet ordre ?

Solution. De $n + 1$ façons. □

Question 4.

Déclarer la variable globale `empty : 'a queue` qui représente une file vide.

Question 5.

Écrire la fonction `is_empty : 'a queue -> bool`. On veillera à obtenir la complexité temporelle la plus petite possible.

```
# let q = empty in is_empty q;;
- : bool = true
# let  q = {front = [1;2;3] ; rear = [6;5;4]} in is_empty q;;
- : bool = false
```

Question 6.

Écrire la fonction `enqueue : 'a queue -> 'a -> 'a queue` telle que `enqueue q x` ajoute l'élément `x` à la file `q`. Comme il n'y a pas d'effet de bord (structure immuable) on crée une nouvelle file.

On veillera à obtenir la complexité temporelle la plus petite possible.

Voici un exemple d'utilisation :

```
 # let q = {front = [1;2;3] ; rear = [6;5;4]} in
    enqueue q 7;;
- : int queue = {front = [1; 2; 3]; rear = [7; 6; 5; 4]}
```

Question 7.

Écrire la fonction `dequeue : 'a queue -> 'a * 'a queue` telle que `dequeue q` renvoie un couple dont le premier élément est celui qui a été retiré à `q` et le second est une nouvelle file qui correspond à ce qu'il reste de `q` après ce retrait.

Dans le cas où la file est vide une exception (de votre choix) est soulevée. Sinon, le principe est le suivant : si la liste `front` n'est pas vide on retire simplement son premier élément, mais si elle est vide, l'inverse de la liste `rear` prend la place de `front` et c'est à cet inverse qu'on retire un élément.

Voici un exemple d'exécution :

```
# let q = {front = [1;2;3] ; rear = [6;5;4]} in
    dequeue q;;
- : int * int queue = (1, {front = [2; 3]; rear = [6; 5; 4]})
# let q = {front = [] ; rear = [6;5;4;3;2;1]} in
    dequeue q;;
- : int * int queue = (1, {front = [2; 3; 4; 5; 6]; rear = []})
```

Solution.

```
let empty = {front = [] ; rear = [];};;

let is_empty q = q.front = [] && q.rear = [];;

let is_empty q = q = empty;;(*V2*)

let enqueue q x =
  {front = q.front ; rear = x::q.rear};;

let dequeue q =
  match q.front with
  | x::t -> x , {front = t; rear = q.rear}
  | [] -> let inv = List.rev q.rear in
    match inv with
    | [] -> invalid_arg "dequeue"
    | x::t -> x, {front = t ; rear = []};;
```

□

Question 8.

On considère dans cette question que les files possèdent n éléments.

1. Quelle est précisément le plus petit nombre d'ajouts en tête de liste `::` pour l'appel `dequeue q` si `q` contient n éléments ?
2. Identifier le pire cas pour l'appel `dequeue q`. Et donner précisément le plus grand nombre d'ajouts en tête de liste `::` si `q` contient n éléments.
3. On suppose que toutes les représentations d'une file donnée par un objet du type `queue` sont équiprobables (on rappelle que plusieurs représentations sont possibles pour une même file).
   On note X la variable aléatoire qui donne la longueur du membre `q.rear` si `q` est un objet de type `queue` représentant une file de longueur n. On note c la fonction telle que $c(k)$ donne le nombre d'opérations `::` pour un appel `enqueue q` si `q.rear` est de longueur k.
   Calculer proprement (c'est à dire comme l'espérance $\mathbb{E}(c(X))$ la complexité en moyenne de l'appel `dequeue q` pour une file q de n éléments tous distincts.

Solution.

1. La complexité temporelle au mieux de l'appel `dequeue(q)` est $O(1)$ (cas où `q.front` est non vide). 0 ajout
2. La complexité au pire est $O(n)$ (cas où `q.front` est vide) car il faut d'abord inverser la liste (les opérations de filtrage et de réécriture qui s'ensuivent sont en $O(1)$).
   n ajouts

3. Pour la complexité moyenne, qui est un calcul d'espérance, il faut identifier une variable aléatoire. Celle de l'énoncé est convenable : X donne la longueur de la liste `q.rear`. Pour une file donnée $a_1, \dots, a_n$ de longueur n et dont les éléments sont tous distincts, il y a $n+1$ façon de représenter cette file (voir question 3).

   On a alors par équiprobabilité (puisque le sujet l'indique) : $\mathbb{P}(X = i) = \dfrac{1}{n+1}$ si $i \in [\![0, n]\!]$ et $\mathbb{P}(X = i) = 0$ sinon. De plus, $c(i) = 0$ si $i < n$ (pas d'ajout en tête de liste par `::`) et $c(i) = n$ sinon (exactement n ajouts pour le renversement).
   Par th. du transfert

$$\mathbb{E}(c(X)) = \sum_{x=0}^{n} c(x)\mathbb{P}(X = x) = \frac{1}{n+1}c(n) + \sum_{x=0}^{n-1} \underbrace{c(x)}_{=0} \mathbb{P}(X = x) = \frac{n}{n+1} + 0 \underset{n\to+\infty}{\longrightarrow} 1$$

   Donc complexité $O(1)$ en moyenne.

□

Il ressort de calculs précédents que tout est réuni pour se lancer dans un calcul de complexité amortie. On étudie donc, en partant de la file vide, le coût d'une séquence d'opérations (enfilement ou défilement) où chaque opération s'applique à la file obtenue avec l'opération précédente.

Pour une file `q`, on définit son potentiel ainsi :

$$\Phi(\texttt{q}) \overset{\text{def}}{=} \text{la longueur de la liste } \texttt{q.rear}$$

Le *coût amorti* a d'une opération est donc la complexité réelle c de cette opération plus la différence des potentiels après et avant l'opération.

Soit une file `q` dont la longueur `q.rear` est ℓ. On compte le nombre d'opérations `::`.

Question 9.

On considère une opération d'enfilement sur `q`. Montrer que le coût amorti (nombre d'opérations de construction `::`) est constant.

Solution. La complexité réelle est 1 (une seule construction). Et `q.rear` grossit de 1 élément. On a donc

$$\begin{aligned} a &= 1 + \Delta_\Phi \\ &= 1 + (\ell + 1) - \ell \\ &= 2 \end{aligned}$$

□

Question 10.

On considère une opération de défilement sur `q`. Montrer que le coût amorti est constant.

Solution. Il faut distinguer :

Si `q.front` n'est pas vide. Dans ce cas, on retire le premier élément de `q.front` : $c = 0$ puisqu'il n'y a pas de d'opération `::`. Et on ne touche pas à la longueur de `q.rear` qui reste à ℓ.

$$\begin{aligned} a &= 0 + \Delta_\Phi \\ &= 0 + \ell - \ell \\ &= 0. \end{aligned}$$

Si `q.front` est vide Dans le cas où `q.rear` est vide aussi, une exception est soulevée et aucune liste n'est modifiée. On se retrouve dans le cas précédent.

Sinon, on inverse `q.rear`. Donc $c = \ell$ (nombre d'opérations du renversement). La longueur de `q.rear` passe de ℓ à 0.

$$\begin{aligned} a &= \ell + 0 + \Delta_\Phi \\ &= (\ell) + 0 - \ell \\ &= 0. \end{aligned}$$

□

Question 11.

En déduire la complexité amortie d'une opération dans une séquence d'opérations (enfilement/défilement) commençant sur une file ayant un potentiel nul.

Solution. D'après le corollaire du th. d'amortissement, puisque le coût amorti est majoré par une constante (il vaut 0 ou 2), on considère qu'une séquence de n opérations d'enfilement/défilement a un coût amorti *pour chaque opération* en $O(1)$. □

2 Structure de pile mutable

On se donne les structures suivantes :

```c
typedef struct node {
    struct node *prev;
    struct node *next;
    DATA data;
} node_t;

typedef struct dlist {
    node_t *head;
    node_t *tail;
    size_t  size;
} dlist_t;
```

Elles implémentent la notion de *liste doublement chaînée*. La constante de préprocessing `DATA` est définie à la compilation avec l'option `gcc ... -DDATA=float`. Une valeur par défaut pour cette constante peut par exemple être fixée ainsi :

```c
#ifndef DATA
#define DATA int
#endif
```

Question 12.

Écrire la fonction

```c
dlist_t* dlist_empty(void)
```

Elle renvoie un pointeur sur une liste doublement chaînée vide. Le champ `size` vaut zéro, les pointeurs internes `head` et `tail` sont nuls.

FF j'avais écris les pointeurs internes `next` et `prev` sont nuls : cela a perturbé certains

Solution. Code

```c
dlist_t* dlist_empty(void)
{
    dlist_t *l = malloc(1 * sizeof *l);
    if (l == NULL) return NULL;
    //on peut utiliser calloc (1 , sizeof *l)
    l->head = NULL;
    l->tail = NULL;
    l->size = 0;
    return l;
}
```

ou

```c
dlist_t* dlist_empty(void){
   return calloc(1,sizeof(dliste_t)) ;
    }

```

□

Question 13.

Écrire la fonction

```c
bool dlist_push_front(dlist_t *l, DATA x)
```

qui ajoute la donnée `x` en tête de liste. Le bouléen renvoyé indique que tout s'est bien passé (`true`) ou qu'un problème d'allocation a été rencontré lors de la création du nouveau maillon (`false`).

Solution. Code

```c
static node_t *node_new(DATA x)
{
  node_t *p = (node_t *)calloc(1,sizeof(node_t));
    if (p == NULL) return NULL;
    p->data = x;// prev et next sont NULL
    return p;
}

/* ===== Ajouts ===== */
bool dlist_push_front(dlist_t *l, DATA x)
{
  assert(l != NULL);
  node_t *p = node_new(x);
  if (p == NULL) return false;

  p->next = l->head;


  if (l->head != NULL) {
    l->head->prev = p;
  } else {
    /* liste vide : head et tail deviennent p */
    l->tail = p;
  }
  l->head = p;
  l->size++;
  return true;
}
```

□

Question 14.

Écrire la fonction

```c
bool dlist_push_back(dlist_t *l, DATA x)
```

qui ajoute la donnée `x` en queue de liste. Le bouléen renvoyé indique que tout s'est bien passé (`true`) ou qu'un problème d'allocation a été rencontré lors de la création du nouveau maillon (`false`).

Solution. Code

```c
bool dlist_push_back(dlist_t *l, DATA x)
{
    node_t *p = node_new(x);
    if (p == NULL) return false;

    p->prev = l->tail;


    if (l->tail != NULL) {
        l->tail->next = p;
    } else {
        /* liste vide */
        l->head = p;
    }
    l->tail = p;
    l->size++;
    return true;
}
```

□

Question 15.

Écrire la fonction

```c
bool dlist_pop_front(dlist_t *l, DATA *out)
```

qui retire le maillon de tête de la liste `l` (la liste est modifiée). La valeur `false` est renvoyée si ce retrait est impossible, sinon on renvoie `true`. Si le pointeur `out` n'est pas vide, on le fait pointer sur la valeur du maillon retiré.

Solution. Code

```
bool dlist_pop_front(dlist_t *l, DATA *out)
{
  assert(l!=NULL);
  if (l->head == NULL) return false;

  node_t *p = l->head;
  if (out != NULL) *out = p->data;//l->head->data

  l->head = l->head->next;// p->next;
  if (l->head != NULL) {
    l->head->prev = NULL;
  } else {
    /* la liste devient vide */
    l->tail = NULL;
  }

  free(p);
  l->size--;
  return true;
}
```

□

Question 16.

Écrire la fonction

```
Code

bool dlist_pop_back(dlist_t *l, DATA *out)
```

qui retire le maillon de queue de la liste `l` (la liste est modifiée). La valeur `false` est renvoyée si ce retrait est impossible (liste vide), sinon on renvoie `true`. Si le pointeur `out` n'est pas vide, on le fait pointer sur la valeur du maillon retiré.

Solution. Code

```
bool dlist_pop_back(dlist_t *l, DATA *out)
{
  assert(l!=NULL);
  if (l->tail == NULL) return false;

  node_t *p = l->tail;
  if (out != NULL) *out = p->data;

  l->tail = p->prev;
  if (l->tail != NULL) {
    l->tail->next = NULL;
  } else {
    /* la liste devient vide */
    l->head = NULL;
  }

  free(p);
  l->size--;
  return true;
}
```

□

Question 17.

Écrire la fonction

```
size_t dlist_length(const dlist_t *l)
```

qui donne le nombre de maillons en $O(1)$.

Solution. Code

```
size_t dlist_length(const dlist_t *l)
{
    /* on a un champ size, donc O(1) */
    return l->size;
}
```

□

Question 18.

Écrire la fonction

```
void dlist_clear(dlist_t *l)
```

qui nettoie le pointeur `l` : elle libère tous les maillons de la liste, et met à jour les 3 champs de `l` de sorte que la liste pointée devienne vide. Le pointeur `l` n'est pas libéré.

Solution. Code

```
void dlist_clear(dlist_t *l)
{
    node_t *p = l->head;
    while (p != NULL) {
        node_t *nxt = p->next;
        free(p);
        p = nxt;
    }
    l->head = NULL;
    l->tail = NULL;
    l->size = 0;
}
```

ou

```
void dlist_clear(dlist_t *l)
{
    while (dlist_pop_front(l, NULL)) { }
}
```

plus court ! □

Question 19.

Écrire une fonction

```
void dlist_rev(dlist_t *l);
```

qui inverse la liste pointée par `l`. La liste $1, 2, 3$ devient $3, 2, 1$.

FF on ne construit pas une nouvelle liste, la liste en entrée doit être modifiée.

Solution. Code

```
1
2  void dlist_rev(dlist_t *l){
3    //une liste vide ou singleton est son propre inverse
4    if (dlist_length(l)<=1) return;
5    node_t *p = l->tail;
6    node_t *tmp;
7    while (p != NULL){
8      tmp = p->prev;
9      p->prev = p->next;
10     p->next = tmp;
11     p = tmp;
12   }
13   tmp = l->head;
14   l->head = l->tail;
15   l->tail = tmp;
16 }
```

On évite les codes comme

```
1 for (size_t i = 0, n = l->size; i < n; i++) {
2     dlist_push_back(l, l->head->data);
3     dlist_pop_front(l, NULL);
4 }
```

C'est une mauvaise idée ici (alloc/free inutiles, copie de DATA), alors que linversion des pointeurs est propre et efficace. □

3 Quick Select de Hoare

De nombreuses applications requièrent le calcul de l'élément de rang k (dans l'ordre croissant et en commençant à zéro) d'une liste d'objets. **Pour tous les calculs de complexité à venir, on suppose que la longueur de la liste à étudier est** n.

Question 20.

Expliquer en quelques mots comment obtenir l'élément de rang r par application d'un tri. Puis donner la complexité temporelle au pire qu'on peut raisonnablement espérer avec cette méthode SI LES COMPARAISONS SONT EN O(1).

Solution. On fait un tri (fusion) en $O(n \log n)$ puis une recherche de l'élément de rang k en $O(\min(k, n))$. Comme $k < n$ (pas de programmation défensive) on a un $O(n \log n)$.

Les comparaisons sont comptées en $O(1)$ dans cette analyse. Pourquoi $O(1)$? Le sujet n'est pas assez explicité à ce propos, j'ai été large dans la correction. □

Le *Quick Select de Hoare* ne fait pas de tri préalable et s'inspire du tri rapide. Il s'appuie lui aussi sur une fonction de partitionnement.

Question 21.

Écrire la fonction en récursion terminale

```
partition
(leq : 'a -> 'a -> bool)
(l : 'a list)
(x : 'a) :
'a list * 'a list * int
```

La fonction renvoie la liste des éléments de `l` plus petit que le pivot `x` (au sens de la fonction de comparaison `leq`) [2], la liste des éléments plus grands et le nombre d'éléments plus petits ou égaux au pivot. Aucune fonction auxiliaire de calcul de longueur de liste n'est autorisé : le triplet renvoyé doit s'obtenir en un seul parcours.

```
# let l = [5; 1;8;9;2;1;0;10] in partition (<=) l 6;;
- : int list * int list * int = ([5; 1; 2; 1; 0], [8; 9; 10], 5)
```

Solution. Code

```
let partition leq (l : 'a list) (x : 'a)
    : 'a list * 'a list * int =
  let rec aux l acc_le acc_gt n_le =
    match l with
    | [] -> (List.rev acc_le, List.rev acc_gt, n_le)
    | y :: q ->
       if  leq y x
       then aux q (y :: acc_le) acc_gt (n_le + 1)
       else aux q acc_le (y :: acc_gt) n_le
  in
  aux l [] [] 0;;
```

□

2. leq signifie « less or equal ».

Question 22.

Quelle est le nombre d'appels à `leq` dans le calcul `partition leq l x` ? On ne demande pas une analyse détaillée, seule la réponse importe.

Solution. n fois □

On cherche le k-ième plus petit élément (en comptant les positions à partir de zéro) d'une liste ℓ de longueur n. L'algorithme procède récursivement ainsi :

- On isole le premier élément p de la liste (le pivot) et on partitionne le reste de la liste (donc sans p) au moyen de ce pivot. On récupère un triplet g, d, m avec g, d deux listes telles que $|g| + |d| = n - 1$ et $m = |g|$.
- Si $k < m$ alors on cherche l'élément en position k dans g ;
- Si $k = m$, le pivot est l'élément en position k ;
- Si $k > m$, l'élément en position k de ℓ dans d. Mais la liste parcourue n'est plus de taille n : il faut changer k (voir question suivante).

Question 23.

Avec les conventions précédentes, lorsque $k > m$, l'élément de ℓ en position k est aussi l'élément de d en position k'. Exprimer k' à l'aide de k.

Solution. Pour comprendre, prenons $k = 5$ et la liste $\ell = 1, 2, 3, 6, 7, 8, 9, 10$. Alors $g = 2, 3$ et $d = 6, 7, 8, 9, 10$. La longueur $m = 2$.
On cherche l'élément en position 5 de ℓ : c'est 8.
Et 8 est en position $k' = 2$ dans d, c'est à dire $k - (m + 1)$.
Ainsi $k' = k - (m + 1)$ □

Question 24.

Écrire la fonction

```
quickselect (leq :'a -> 'a -> bool) (l : 'a list) (k : int) : 'a
```

qui réalise le Quick Select et renvoie la valeur de l'élément en position k de ℓ (lorsque ℓ est trié par l'ordre induit par `leq`). Si $k < 0$ ou $k > n - 1$, une exception est soulevée mais aucune fonction de calcul de longueur ne doit être appliquée.

```
# let l = [5; 1;8;9;2;1;0;10] in partition  (<=) l 15;;
- : int list * int list * int = ([5; 1; 8; 9; 2; 1; 0; 10], [], 8)
# let x = quickselect (<=) [7; 2; 9; 2; 5; 1] 20;;
Exception: Invalid_argument "quickselect: liste vide".
```

Solution. Code

```
let quickselect leq (l : 'a list) (k : int) : 'a =
  let rec loop l k =
    match l with
    | [] -> invalid_arg "quickselect: liste vide"
    | p :: q ->
        let (le, gt, m) = partition leq q p in
        if k < m then
          loop le k
        else if k = m then
          p
        else
          loop gt (k - (m + 1))
```

```
  in
  if k < 0 then invalid_arg "quickselect: k negatif";
  loop l k;;

let x = quickselect (<=) [7; 2; 9; 2; 5; 1] 3;;
(* liste triée: [1;2;2;5;7;9], k=3 -> 5 *)
```

□

Question 25.

Établir la terminaison de l'appel `quickselect (<=) l k` pour $|\ell| = n$.

Solution. La liste passée en paramètre de l'appel interne est strictement pus courte que celle de l'appel externe. □

On peut établir que la complexité moyenne du Quick Select est un $O(n)$ pour des listes dont tous les éléments sont distincts.

4 Appendice

4.1 À propos du nombre de concaténations

Les questions relatives à la complexité en moyenne et amortie dans la première partie portent sur le nombre d'applications de l'opérateur `::` effectuées pour l'ensemble de la récursion. On rappelle à ce propos que :

— la production de la liste vide se fait sans appel au constructeur `::` ;

— la production d'une liste comme `[1;2]` se fait en 2 applications du constructeur `:,:` (mêmes si elles sont cachées). En effet `[1;2]` est du sucre syntaxique pour exprimer `1::(2::[])`.

Attention : **On ne compte que les constructions de listes, c'est-à-dire les applications du constructeur (`::`) apparaissant dans les expressions, et non celles intervenant dans les motifs de filtrage.**

4.2 Espérance

On rappelle le théorème du transfert :

Théorème 4.1. *Soit X une variable aléatoire finie prenant les valeurs $\{x_1, \dots, x_n\}$ et f une fonction à valeurs réelles dénie sur $\{x_1, \dots, x_n\}$. Alors la variable aléatoire $f(X)$ a une espérance finie et*

$$\mathbb{E}(f(X)) = \sum_{i=1}^{n} f(x_i)\mathbb{P}(x = x_i).$$

4.3 Enregistrements immuables en OCaml

En OCaml, un *enregistrement* est un type composé de plusieurs champs nommés. Par défaut, les champs sont **immuables** : une fois la valeur créée, on ne peut pas modifier un champ.

Déclaration d'un type enregistrement On déclare un type `point` représentant un point du plan à deux coordonnées `x` et `y` de type `int` :

```
type point = {
  x : int;
  y : int;
}
```

Chaque champ est défini par :

— son nom (`x`, `y`),

— son type (`int`).

Création d'un enregistrement On crée une valeur de type `point` en donnant une valeur à chaque champ :

```
let p1 : point = { x = 3; y = 5 }
```

L'ordre des champs n'a pas d'importance :

```
let p2 = { y = 0; x = 1 }
```

Accès aux champs L'accès à un champ se fait avec l'opérateur `.` :

```
let abs_x = p1.x
let abs_y = p1.y
```

On peut utiliser les champs dans des expressions :

```
let distance_origine p =
  abs p.x + abs p.y
```

Immuabilité des champs Les champs étant immuables, l'instruction suivante est **interdite** :

```
p1.x <- 10    (* ERREUR *)
```

Pour « modifier » un point, on doit en créer un nouveau :

```
let p3 = { x = 10; y = p1.y}
```

ou (plus concis mais moins clair) :

```
let p3 = { p1 with x = 10 }
```

Ici, `p3` est un nouveau point ; `p1` reste inchangé.