Structures séquentielles

Tableau en OCaml

Exercice 1 : Exercice

Écrire une fonction `appartient : 'a -> 'a array -> bool` telle que `appartient x t` renvoie `true` si et seulement si l'élément `x` apparaît dans le tableau `t`.

Exercice 2 : Exercice

Écrire une fonction `croissant : 'a array -> bool` qui détermine si le tableau passé en argument est croissant (au sens large). Si possible, on arrêtera le parcours dès que la réponse est connue.

Exercice 3 : Exercice

Pour une séquence $s = s_0, \dots, s_{n-1}$, on définit le *miroir* de s par $\overline{s} = s_{n-1}, \dots, s_0$. Une séquence est un *palindrome* si $s = \overline{s}$. Écrire une fonction `est_palindrome : 'a array -> bool` qui détermine si le tableau passé en argument est un palindrome. On essaiera de limiter au maximum le nombre de comparaisons effectuées.

```
# est_palindrome [|2; 1; 1; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 2; 2|];;
- : bool = false
```

Exercice 4 : Exercice

Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que, après l'appel `swap t i j`, les éléments de `t` d'indices `i` et `j` aient été échangés.

Exercice 5 : Exercice

Écrire une fonction `renverse : 'a array -> unit` telle que, si l'on a initialement `t = [| t_0; ...; t_{k-1} |]`, alors, après l'appel `renverse t`, on a `[| t_{k-1}; ...; t_0 |]`. On travaillera « en place » sans créer de tableau auxiliaire, et l'on fera bien attention à ce que la fonction soit correcte indépendamment de la parité de la longueur du tableau considéré.

```
# let t = [|4; 1; 2; 5|];;
val t : int array = [|4; 1; 2; 5|]
# renverse t;;
- : unit = ()
# t;;
- : int array = [|5; 2; 1; 4|]
```

Exercice 6 : Exercice

1. Écrire une fonction `cherche_somme (t : int array) (s : int) : (int * int) option` ayant la spécification suivante :
   — si la fonction renvoie `Some (i, j)`, alors on a $0 \leqslant i \leqslant j < |t|$ et `t.(i) + t.(j) = s` ;
   — si la fonction renvoie `None`, alors il n'existe pas de couple (i, j) vérifiant $0 \leqslant i \leqslant j < |t|$ et `t.(i) + t.(j) = s`.

```
# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 11;;
- : (int * int) option = Some (6, 7)
(* Some (1, 6) ou Some (2, 8) conviennent également. *)

# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 14;;
- : (int * int) option = Some (5, 5)

# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 19;;
- : (int * int) option = None
```

2. On suppose maintenant que le tableau `t` est trié par ordre croissant. Écrire une fonction `cherche_somme_croissant` ayant la même spécification que `cherche_somme` mais de complexité linéaire en la taille du tableau.

```
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 13;;
- : (int * int) option = Some (0, 8)
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 17;;
- : (int * int) option = Some (3, 8)
```

Exercice 7 : Exercice

Écrire une fonction `indices_mini (t : int array) : int list` renvoyant la liste des indices i tels que `t.(i)` $=$ min `t`. On commencera par écrire la version la plus naturelle puis on pourra essayer d'écrire une version ne faisant qu'un seul parcours du tableau `t`.

```
# indices_mini [|10; 2; 7; 1; 7; 1; 3; 7|];;
- : int list = [3; 5]
```

Exercice 8 : Exercice

Écrire une fonction `filtre (pred : 'a -> bool) (t : 'a array) -> 'a list` qui renvoie la liste des `t.(i)` tels que `pred t.(i) = true`, classée dans l'ordre des i croissants.

```
# filtre (fun n -> n mod 2 = 0) [|1; 4; 2; 5; 7; 8; 7; 10; 4; 5|];;
- : int list = [4; 2; 8; 10; 4]
```

Exercice 9 : Section équilibrée

On considère un tableau de booléens t de longueur n. Pour $0 \leqslant a \leqslant b \leqslant n$, on note $t[a,b[$ la séquence $t_a, \ldots, t_{b-1}$ (elle est vide si $a = b$). La longueur de $t[a,b[$ est définie comme $b - a$. On définit alors

— vrai(a,b) comme le nombre de `true` dans la section $t[a,b[$

$$\text{vrai}(a,b) = \text{Card}\{i \in [\![a,b[\![\mid \texttt{t.(i) = true}\}.$$

— faux(a,b) comme le nombre de `false` dans la section $t[a,b[$

$$\text{faux}(a,b) = \text{Card}\{i \in [\![a,b[\![\mid \texttt{t.(i) = false}\}.$$

— $\delta(a,b)$ comme l'écart entre les deux

$$\delta(a,b) = \text{vrai}(a,b) - \text{faux}(a,b).$$

Une section $t[a,b[$ sera dite *équilibrée* lorsque $\delta(a,b) = 0$, c'est-à-dire lorsqu'elle contient autant de `true` que de `false`. Le but de l'exercice est de déterminer efficacement la longueur maximale d'une section équilibrée d'un tableau t.

1. Écrire une fonction `est_equilibree (t : bool array) (a : int) (b : int) : bool` qui détermine si la section $t[a,b[$ est équilibrée.
2. Écrire une fonction `max_equilibree (t : bool array) : int` qui calcule la longueur maximale d'une section équilibrée de `t` en testant toutes les possibilités. Quelle est la complexité de cette fonction ?
3. Écrire une fonction `max_depuis (t : bool array) (a : int) : int` qui renvoie la longueur maximale d'une section équilibrée de t de la forme $t[a,b[$ (où a est fixé car donné en argument). Cette fonction doit avoir une complexité linéaire en $|t|$.
4. En déduire une version plus efficace de `max_equilibree` et déterminer sa complexité.
5. Trouver une relation entre $\delta(a,b)$, $\delta(0,a)$ et $\delta(0,b)$ et en déduire une version de `max_equilibree` de complexité linéaire en la taille de t.

1 Pile

Exercice 10 : Pile en OCaml

1. Rappeler la signature du type abstrait de pile impérative.
2. Implémenter en OCaml ce type abstrait `'a pile`.
3. Implémenter les fonctions suivantes sur les piles, en utilisant uniquement les fonctions de la signature.
   — `efface (p : 'a pile) : unit`, qui efface le contenu de la pile `p`.
   — `echange (p : 'a pile) : unit`, qui échange les deux premiers éléments de la pile `p`. On s'assurera que la pile contient au moins 2 éléments.
   — `rotation (p : 'a pile) : unit`, qui place le premier élément de la pile `p` tout au fond de `p`.
   — `taille (p : 'a pile) : unit`, qui renvoie le nombre d'éléments dans la pile `p`. La pile doit être inchangée après l'exécution de la fonction.
   — `retourner (p : 'a pile) : unit`, qui renverse la pile `p`. On pourra s'aider d'une liste classique.
   — `copier (p : 'a pile) : unit` qui crée et renvoie une copie de la pile `p`. La pile d'origine doit être inchangée après l'exécution de la fonction.

Exercice 11 : Le lièvre et la tortue

1. Implémenter en C le type liste chainée permettant de stocker des valeurs entières. On devra pouvoir l'utiliser de la façon suivante :

```c
// Création de la liste [ 1; 2; 3 ]
cell *lst1 = cons(1, cons(2, cons(3, NULL)));
// Accès à l'élément suivant
cell *lst2 = lst1->next;
// Accès à la valeur
printf("%d", lst2->value);
```

2. Implémenter une fonction `void delete (cell *lst)` permettant de libérer la totalité de la mémoire allouée par la liste `lst`.

Rien n'empêche de modifier un champ `next` pour le faire revenir sur un autre chainon. Par exemple :

```c
cell *lst3 = lst2->next;
// lst2 pointe vers l'élément "2", lst3 pointe vers l'élément "3"
lst3->next = lst2;
```

3. Représenter par un schéma l'état de la mémoire.
4. Est-ce que votre fonction de libération de la mémoire fonctionne sur cette liste ?

On va s'intéresser à la détection de cycle dans une liste chainée. On veut savoir si une liste est ou non cyclique à partir d'un certain rang. Pour cela, on veut écrire une fonction `bool is_cyclic(cell *lst)` qui renvoie `true` si un cycle est détecté, et `false` sinon. On va utiliser l'algorithme du lièvre et de la tortue. L'idée est la suivante : on parcourt la liste avec deux pointeurs. À chaque itération, le premier pointeur avance d'un élément (c'est la tortue), et le second pointeur avance de deux éléments à la fois (c'est le lièvre). Si le lièvre atteint la fin de la liste (`NULL`), il n'y a pas de cycle. Si la tortue rattrape le lièvre après le départ, alors il y a un cycle.

5. Illustrer une exécution de l'algorithme sur un exemple.
6. Implémenter cet algorithme en C.
7. Justifier la correction et la terminaison de cet algorithme.

2 File

Exercice 12 : Comptine

1. Rappeler la signature d'une file impérative abstraite.
2. En OCaml, implémenter un type `'a file` qui contiendra au cours de son existence au plus n éléments. On utilisera pour cela un tableau circulaire.

Une comptine est une chanson enfantine permettant de désigner une personne avec un semblant de hasard. Les comptines « plouf plouf » ou « am stram gram » en sont des exemples célèbres. Le principe est le suivant : des enfants se mettent en cercle et l'un d'entre eux récite la comptine en pointant du doigt successivement chaque enfant du cercle. À la fin de la comptine, le dernier enfant désigné est éliminé. Le processus recommence alors avec les enfants restants,

jusqu'à ce qu'il n'en reste plus qu'un. Plus généralement, à partir d'une liste d'enfants de longueur n et d'une comptine de longueur k, on cherche à déterminer l'enfant restant à la fin du processus. En mathématiques, ce problème est connu sous le nom de *problème de Josèphe*.

3. Illustrer l'exécution du processus pour $n = 5$ enfants `["Léa"; "Hugo"; "Zoé"; "Bilal"; "Liv"]` et une comptine de longueur $k = 8$. Le premier chant commencera en pointant `"Léa"`.
4. Écrire une fonction

```
elimine : (enfants : 'a file) (k : int) (f : 'a -> unit) : unit
```

exécutant ce processus en commençant le chant de la première comptine en pointant l'enfant en tête de la file. On exécutera la fonction `f` pour chaque enfant éliminé. Une fois la fonction exécutée, la file `enfants` sera vide. Utiliser ensuite cette fonction pour afficher dans l'ordre les noms des enfants éliminés présents dans une file de type `string file`.

Exercice 13 : Fenêtre glissante

Soit a un tableau de $n \in \mathbb{N}$ éléments de type `int` et $k \in [\![1, n]\!]$. L'objectif de cet exercice est de créer un tableau b de taille $m \coloneqq n + 1 - k$ tel que pour tout $i \in [\![0, m[\![$, b_i soit le minimum des a_j pour $j \in [\![i, i + k[\![$.

1. Implémenter une version naïve de cet algorithme

```
int *minimum_sliding(int *a, int n, int k)
```

Quelle est la complexité de cette fonction ?

Dans la suite, on va implémenter un algorithme utilisant une file à deux bouts $\mathcal{F}$ (*deque* ou *double-ended queue* en anglais) qui va contenir des indices d'éléments de a. Pour cela, nous utilserons une implémentation basée sur un tableau circulaire.

```
struct deque {
  T *data;
  int left;
  int size;
  int capacity;
};

typedef struct deque deque;
```

Une file à deux bouts possède les opérations suivantes :
— `deque *new(int r)` : Crée une file vide à deux bouts avec une capacité de r éléments.
— `void add_right(deque *f, int x)` : Ajoute l'élément x à la droite de $\mathcal{F}$.
— `void add_left(deque *f, int x)` : Ajoute l'élément x à la gauche de $\mathcal{F}$.
— `int peek_right(deque *f)` : Renvoie l'élément à la droite de $\mathcal{F}$.
— `int peek_left(deque *f)` : Renvoie l'élément à la gauche de $\mathcal{F}$.
— `int pop_right(deque *f)` : Renvoie et supprime l'élément à la droite de $\mathcal{F}$.
— `int pop_left(deque *f)` : Renvoie et supprime l'élément à la gauche de $\mathcal{F}$.
— `bool is_empty(deque *f)` : Détermine si $\mathcal{F}$ est vide.
— `void delete(deque *f)` : Libère la mémoire de $\mathcal{F}$.

2. Implémenter les fonctions ci-dessus.
3. Pour tout couple d'entier (p, q) tel que $0 \leqslant p \leqslant q \leqslant n$, on considère l'invariant $\mathcal{I}_{p,q}$ suivant :
   — Les éléments de $\mathcal{F}$ sont, de la gauche vers la droite, $s_1, \ldots, s_r$.
   — $p \leqslant s_1 < s_2 < \cdots < s_r < q$
   — $\forall s \in [\![p, q[\![\,, \quad s \in \mathcal{F} \iff [\forall j \in [\![s, q[\![\,, \quad a_s \leqslant a_j]$

   (a) En supposant que $\mathcal{I}_{i,i+k}$ est vrai, que vaut b_i ?
   (b) Si $\mathcal{I}_{p,q}$ est vrai, comment mettre à jour $\mathcal{F}$ pour rendre $\mathcal{I}_{p+1,q}$ vrai ?
   (c) Si $\mathcal{I}_{p,q}$ est vrai, comment mettre à jour $\mathcal{F}$ pour rendre $\mathcal{I}_{p,q+1}$ vrai ?
4. En déduire une fonction `int *minimum_sliding(int *a, int n, int k)`
5. Quelle est la complexité de `minimum_sliding` ?

3 File de priorité

4 Ensemble et dictionnaire

Exercice 14 : Collisions

On considère la fonction de hachage suivante sur les chaines de caractères :

```c
uint64_t hash(char *s) {
    uint64_t h = 0;
    for (int k = 0; s[k] != '\0'; k++) {
        h = 31 * h + s[k];
    }
    return h;
}
```

1. Sans les expliciter, montrer qu'il existe deux chaines distinctes de 15 caractères, pris parmi les 26 lettres de l'alphabet, qui donnent la même valeur pour la fonction `hash` ci-dessus.
2. Expliciter deux chaines de longueur 2, distinctes, formées uniquement de caractères alphabétiques dans `A-Z` et `a-z` (52 caractères au total) qui ont la même valeur pour la fonction `hash`.
3. En déduire une façon simple de construire un nombre arbitraire de chaines de caractères ayant la même valeur pour la fonction `hash`.

Exercice 15 : Filtre de Bloom

Un *filtre de Bloom* est une structure de données qui réalise un ensemble et fournit deux opérations : ajouter un élément et tester la présence d'un élément. Cette dernière opération doit donner un résultat correct pour les éléments qui ont déjà été ajoutés à l'ensemble mais elle peut donner un résultat incorrect pour les autres éléments. Un filtre de Bloom utilise un tableau de m booléens et k fonctions de hachage $h_1, \dots, h_k$ qui envoient les éléments sur $[\![0, m[\![$. Quand on ajoute un élément x, on met à `true` tous les booléens aux indices $h_1(x), \dots, h_k(x)$. Quand on teste la présence de x, on renvoie `true` si et seulement si tous les booléens aux indices $h_1(x), \dots, h_k(x)$ sont à `true`.

1. Proposer une implémentation en C d'un filtre de Bloom pour des chaines de caractères, où les paramètres k et m sont passés comme arguments au constructeur. Pour la fonction de hachage h_i, on pourra utiliser la fonction suivante où r_i est un entier tiré au hasard au moment de la construction.

   ```c
   uint64_t hash(uint64_t r, char *s) {
       uint64_t h = 0;
       for (int k = 0; s[k] != '\0'; k++) {
           h = r * h + s[k];
       }
       return h;
   }
   ```

2. Tester empiriquement l'efficacité d'un tel filtre, pour différentes valeurs des paramètres k et m. Par exemple, ajouter tous les mots du dictionnaire dans un filtre, puis, pour chaque mot w du dictionnaire, teste la présence du mot wc où c est un caractère qui n'apparait dans aucun mot, par exemple un caractère non alphabétique comme `'\n'`. Compter les faux positifs et commenter le résultat.