TP #19 — Tableaux dynamiques purement fonctionnels

Le but de ce sujet est d'écrire en OCaml une implémentation purement fonctionnelle efficace d'une structure de données de tableau dynamique.
Le principe de l'implémentation est de stocker les éléments du tableau dans un arbre binaire *quasi-complet*, dans l'ordre d'un parcours en largeur (c'est à dire que le $i$ème élément du tableau est stocké comme l'étiquette du $i$ème nœud à être visité par un parcours en largeur). Ultimement, cela permettra notamment, pour des tableaux de N éléments de :

- créer un nouveau tableau en temps $O(N)$;
- accéder à un élément en temps $O(\log N)$ (plus précisément, accéder à l'élément d'indice i en temps $O(\log i)$) ;
- créer un nouveau tableau avec une nouvelle valeur pour l'élément d'indice i en temps $O(\log N)$ (en fait $O(\log i)$) ;
- créer un nouveau tableau de taille $N+1$ avec un élément supplémentaire en temps $O(\log N)$;
- créer un nouveau tableau de taille $N-1$ obtenu en supprimant le dernier élément en temps $O(\log N)$;
- parcourir tous les éléments du tableau (dans des buts divers) en temps $O(N)$;
- créer un nouveau tableau obtenu en concaténant un tableau de longueur M en temps $O(N+M)$.

Outre son intérêt esthétique & pédagogique, une telle structure de donnée peut être intéressante lorsque l'on souhaite maintenir plusieurs versions d'un même tableau. En comparaison de la structure de données de tableau persistant vue au TP #17, celle proposée ici est légèrement moins efficace pour le coût des accès et de stockage supplémentaire pour chaque version (qui est un $O(\log N)$), mais possède l'avantage comparatif que les accès aux versions historiques ne nécessitent aucune reconstruction et sont tous aussi efficaces que pour n'importe quel tableau.

Consignes relatives aux tests. Chaque fonction principale implémentée devra être accompagnée d'(au moins) une fonction de test dédiée, que vous devrez notamment exécuter à chaque modification du code. Pour cela, vous pouvez par exemple définir une fonction :

```
let _test_all () =
    assert (_test_get ()) ;
    assert (_test_set ()) ;
    (* snip *) ;
    true
```

Types

On utilisera les types suivants pour représenter nos tableaux :

```
type 'a tree = E | N of 'a * 'a tree * 'a tree
type 'a farray = { length : int ; a : 'a tree }
```

Le type `'a farray` stocke le nombre d'éléments du tableau qu'il représente dans son champ `length`, et un arbre binaire (représenté par le type `'a tree`) représentant les éléments dans son champ `a`.
On définit également une `exception IndexOutOfBounds`.

1. Écrivez une fonction OCaml :
`length : 'a farray -> int`
qui s'évalue en la longueur de son argument.

Construction & accès élémentaires

La représentation des éléments de nos tableaux se fait grâce à un arbre binaire *quasi-complet*, que l'on se contentera de définir informellement comme un arbre binaire dont tous les niveaux sont complets sauf éventuellement le dernier, où quand il n'est pas complet le dernier niveau est d'abord rempli « par la gauche ».
L'élément d'indice i du tableau est alors stocké dans l'étiquette du $i$ème nœud d'un tel arbre, pour l'ordre de visite d'un parcours en largeur.
On peut construire efficacement de tels arbres en utilisant une numérotation similaire à la numérotation par mots binaires déjà vue en cours. Cependant, afin de faciliter la représentation concrète des mots binaires par des entiers `int`, celle-ci est modifiée de la façon suivante :

— la racine est numérotée 1
— l'enfant gauche (resp. droit) d'un nœud de numéro ν (se lit « nu ») est numéroté $\nu_g := \nu \cdot 0$ (resp. $\nu_d := \nu \cdot 1$) ; si l'on interprète ν comme un nombre entier représenté en base deux, on a alors que $\nu_g = 2\nu$ et $\nu_d = 2\nu + 1$.

Ceci donne une numérotation injective qui (on l'admettra) numérote bien les nœuds par ordre d'un parcours en largeur. On peut remarquer que les éléments sont ici numérotés à partir de un au lieu de zéro, comme il est plus habituel pour un tableau ; il ne sera cependant pas bien compliqué de modifier les fonctions écrites pour obtenir des tableaux indexés à partir de zéro.
Une fonction de création de tableau initialisé avec un élément constant étant relativement inutile pour un tableau purement fonctionnel, on commence directement par écrire une fonction similaire à `Array.init` :

2. Écrivez une fonction OCaml :
`_init1 : int -> (int -> 'a) -> 'a farray`

telle que `_init1 n f` construit un tableau de longueur `n` représenté par un `'a farray`, dont l'élément d'indice `i` est initialisé à `f i`, avec `i` allant de 1 à `n`.
On a par exemple :

```
utop # _init1 0 Fun.id;;
- : int farray = {length = 0; a = E}
utop # _init1 2 Fun.id;;
- : int farray = {length = 2; a = N (1, N (2, E, E), E)}
utop # _init1 4 Fun.id;;
- : int farray = {length = 4;
                  a = N (1, N (2, N (4, E, E), E),
                            N (3, E, E))}
```

Nous voulons maintenant pouvoir accéder (et éventuellement « modifier ») un élément du tableau en fonction de son indice `i`. Pour cela il peut être pratique de connaître la profondeur à laquelle se trouve la nœud de l'arbre dans lequel l'élément est stocké, qui correspond simplement au logarithme (en base deux) de `i`.

3. Écrivez une fonction OCaml :
`_log2 : int -> int`
qui s'évalue en le logarithme en base deux de son argument (supposé strictement positif).

Il peut également être utile de disposer d'une fonction qui s'évalue en la valeur du nème bit d'un `(i : int)`. Une telle fonction peut s'écrire assez aisément en utilisant les fonctions de division entière, mais aussi en manipulant directement la représentation binaire de `i` ; on donne une telle version ci dessous :

```
let _read_bit i n = (i lsr n) land 1
```

4. Écrivez une fonction OCaml :
`_get1 : 'a farray -> int -> 'a`
telle que `_get1 a i` s'évalue en le `i`ème élément de `a` (en commençant l'indexation à 1), ou lève une exception `IndexOutOfBounds` si un tel élément n'existe pas.
Il « suffit » pour cela de s'appeler récursivement sur le bon sous-arbre, donné par la lecture du second bit de poids le plus fort de l'index en court de considération.
Remarque : il peut être utile d'utiliser une fonction récursive auxiliaire, afin d'éviter des calculs de logarithme inutiles.

5. Écrivez une fonction OCaml :
`_set1 : 'a farray -> int -> 'a -> 'a farray`
telle que `_set1 a i v` lève une exception `IndexOutOfBounds` si `a` ne possède pas d'élément d'indice `i`, et sinon construit un nouveau tableau identique à `a` si ce n'est que son élément d'indice `i` vaut `v`.

Il « suffit » pour cela de construire le résultat à partir d'un sous-arbre inchangé et d'un autre sous-arbre nouvellement construit (récursivement). Notamment, cette fonction devra avoir un coût (temporel) logarithmique en `i`.

N.B. : Le coût (temporel) logarithmique de `_set1` borne également son coût mémoire. Conjugué avec le caractère fonctionnel des tableaux manipulés, ceci a des implications qui ne sont pas forcément intuitives : il est par exemple possible de créer N tableaux (non indépendants mais distincts) de N éléments (possédant donc un total combiné de N^2 éléments) en temps $O(N \log N)$.

6. Écrivez des fonctions OCaml :
   ```
   init : int -> (int -> 'a) -> 'a farray
   get : 'a farray -> int -> 'a
   set : 'a farray -> int -> 'a -> 'a farray
   ```
   similaires aux fonctions précédentes, mais pour des tableaux dont l'indexation commence à zéro au lieu de un.
   Indice : `init` s'obtient par une modification marginale de `_init1`, et `get` et `set` s'expriment trivialement en fonction de `_get1` et `_set1`.

N.B. : La numérotation ainsi obtenue peut être décrite de la façon suivante : la racine a numéro 0, et l'enfant gauche (resp. droit) d'un nœud de numéro v a numéro $2v+1$ (resp. $2v+2$).

À partir de maintenant, nous manipulerons uniquement des tableaux dont les éléments sont indexés à partir de zéro.

Fonctions dynamiques

La représentation choisie fait qu'il est aisé d'implémenter des fonctions `push` et `pop` qui « ajoutent » ou « suppriment » un élément à un tableau en coût logarithmique en sa longueur. Il serait donc dommage de s'en priver !

7. Écrivez une fonction OCaml :
   ```
   pop : 'a farray -> 'a farray
   ```
   qui lève une exception si son argument est de longueur nulle, et sinon s'évalue en un tableau nouvellement construit qui lui est égal si ce n'est qu'il est de longueur inférieure de un et que le dernier élément (d'indice le plus élevé) a été retiré.
   Indice 1 : cette fonction est en fait très similaire à un `set`.
   Indice 2 : comment pouvez-vous aisément déterminer le numéro du nœud qui doit perdre un enfant ?
8. Écrivez une fonction OCaml :
   ```
   push : 'a -> 'a farray -> 'a farray
   ```

qui s'évalue en un tableau nouvellement construit égal à son second argument mais de longueur supérieure de un, où un nouvel élément égal à son premier argument a été ajouté en fin du tableau (à l'indice le plus élevé).
Indice : il suffit d'adapter la fonction précédente.

Parcours désordonnés

Les fonctions `get` et `set` écrites plus haut sont plutôt efficaces (après tout, un coût logarithmique en un nombre d'éléments n'est jamais qu'un coût constant pour une constante inférieure à 64…), mais ne permettent pas d'obtenir un coût *réellement* linéaire si l'on souhaite les utiliser pour parcourir tous les éléments d'un tableau. Il est donc intéressant de proposer des fonctions dédiées, qui peuvent exploiter la représentation interne des tableaux pour effectuer ces parcours plus efficacement.
Nous commencerons par des parcours « désordonnés », en ce qu'ils ne parcourent pas les éléments du tableau par index croissant ; en l'absence d'une telle contrainte, de simples parcours en profondeur des arbres stockant les éléments du tableau suffisent.

9. Écrivez une fonction OCaml :
`for_all : ('a -> bool) -> 'a farray -> bool`
telle que `for_all f a` s'évalue à `true` si `f` s'évalue à `true` en tous les éléments de `a`, et `false` sinon.
Cette fonction devra être paresseuse (c'est à dire interrompre son parcours dès que son résultat est déterminé).

10. Écrivez une fonction OCaml :
`exists : ('a -> bool) -> 'a farray -> bool`
telle que `exists f a` s'évalue à `true` si `a` possède un élément pour lequel `f` s'évalue à `true`, et `false` sinon.
Cette fonction devra également être paresseuse.

11. Écrivez une fonction OCaml :
`map : ('a -> 'b) -> 'a farray -> 'b farray`
telle que `map f a` construit un nouveau tableau `b` obtenu en appliquant `f` aux éléments de `a` (c'est à dire que le `i`ème élément de `b` est égal à `f` appliquée au `i`ème élément de `a`).

Un parcours « dans le désordre » n'implique pas nécessairement la méconnaissance des indices des nœuds parcourus, comme le montrent les fonctions suivantes.

12. Écrivez une fonction OCaml :
`for_alli f a : (int -> 'a -> bool) -> 'a farray -> bool`
similaire à `for_all`, si ce n'est que son premier argument s'applique désormais à la fois à l'indice d'un élément ainsi que sa valeur.
Cette fonction devra toujours être paresseuse.

13. Écrivez de même une fonction OCaml :
    `existsi : (int -> 'a -> bool) -> 'a farray -> bool`
    qui adapte `exists` de façon similaire.
14. Écrivez de même une fonction OCaml :
    `mapi : (int -> 'a -> 'b) -> 'a farray -> 'b farray`
    qui adapte `map` de façon similaire.

Parcours respectant l'ordre

Certains parcours nécessitent fonctionnellement de traiter les éléments du tableau dans l'ordre des indices. Ceci peut être implémenté assez aisément pour notre représentation, puisque cet ordre coïncide avec celui d'un parcours en largeur de l'arbre sous-jacent. Afin de maintenir le caractère purement fonctionnel de notre implémentation, ce parcours en largeur devra (typiquement) utiliser une file fonctionnelle. Nous nous satisferont pour cela de l'implémentation déjà vue en cours d'une file avec deux piles, qui offre des performances suffisantes dans notre cas.
On définit le `type 'a _queue = { out : 'a list; inp : 'a list }`

15. Écrivez des fonctions OCaml :
    `_push : 'a -> 'a _queue -> 'a _queue`
    `_pop_opt : 'a _queue -> ('a * 'a _queue) option`
    qui ensemble implémentent les opérations de file pour le type `'a _queue`.
16. Écrivez une fonction OCaml :
    `to_list : 'a farray -> 'a list`
    telle que `to_list a` construit une `list` de mêmes éléments (avec multiplicité) que `a`, donnés par ordre d'indice croissant.

Afin d'éviter de réécrire à répétition le même code de parcours en largeur pour des applications légèrement différentes, il peut être utile de définir des fonctions d'ordre supérieur « `fold` ».

17. Écrivez une fonction OCaml :
    `fold_right : ('a -> 'b -> 'b) -> 'a farray -> 'b -> 'b`
    telle que `fold_right f a e` s'évalue identiquement à :
    `f (get a 0) (f (get a 1) (... (f (get a (n - 1)) e) ..))`
    pour `n` la longueur de `a`.
    Indice : cette fonction est *exactement* la même que `to_list`, où l'on a substitué la liste vide par `e` et l'application de `::` par celle de `f`.
18. Redéfinissez votre fonction `to_list` par une simple application de `fold_right`.
19. Écrivez une fonction OCaml :
    `fold_left : ('a -> 'b -> 'a) -> 'b farray -> 'a -> 'a`

telle que `fold_left f a e` s'évalue identiquement à :
`f (... (f (f e (get a 0)) (get a 1)) ..) (get a (n - 1))`

20. Écrivez une fonction OCaml
`find_opt : ('a -> bool) -> 'a farray -> 'a option`
telle que `find_opt f a` s'évalue à `None` si `f` s'évalue à `false` pour tous les éléments de `a`, et sinon s'évalue à `Some v` avec `v` la valeur de l'élément de `a` d'indice minimal satisfaisant `f`.
Cette fonction devra être paresseuse.

21. Écrivez une fonction OCaml
`find_map : ('a -> 'b option) -> 'a farray -> 'b option`
telle que `find_map f a` évalue `f` sur les éléments de `a` par ordre d'indice croissant, et s'évalue en le premier résultat de la forme `Some v`, ou `None` s'il n'y en a aucun.
Cette fonction devra être paresseuse.

22. Écrivez une fonction OCaml :
`find_mapi :`
`(int -> 'a -> 'b option) -> 'a farray -> 'b option`
similaire à `find_map`, si ce n'est que le premier argument est maintenant évalué à la fois sur l'index de l'élément avant l'élément lui-même.
Cette fonction doit être paresseuse.

23. Écrivez une fonction OCaml :
`find_index : ('a -> bool) -> 'a farray -> int option`
telle que `find_index f a` s'évalue à `Some i` pour `i` le plus petit index tel que `f (get a i)` s'évalue à `true`, ou `None` s'il n'en existe aucun.
Cette fonction doit être paresseuse.

N.B. : Ces fonctions peuvent s'exprimer simplement en fonction d'un `fold_right` (pour les deux premières ; essayez de le faire ?) ou d'un « `fold_righti` », mais perdent dans ces cas leurs caractéristiques paresseuses. Il est cependant possible de définir des variantes paresseuses des `fold` eux-mêmes pour éviter ce problème, mais cela dépasse le cadre de ce sujet.

Conversion & concaténation

Nous avons déjà écrit une fonction `to_list` qui convertit un tableau en une `'a list`, et souhaitons maintenant implémenter son inverse `of_list`, qui construit un tableau dont les éléments sont donnés sous forme de `'a list`. Ceci est nettement plus complexe à faire en temps linéaire que pour `to_list`, tout du moins si l'on se restreint à un style purement fonctionnel (notamment sans aucune mutabilité). On propose pour cela l'approche suivante : on construit l'arbre (représentant les éléments du tableau) « de bas en haut », en construisant

chaque élément de chaque niveau (en partant des feuilles) en fusionnant deux arbres déjà créés pour un niveau précédent (ou éventuellement des arbres vides). On illustre ceci pas à pas pour la création de l'arbre du tableau dont les éléments sont donnés par la liste `[0; 1; 2; 3; 4; 5]`.
On commence par compter le nombre de nœuds qui se trouveront à chaque niveau de profondeur du résultats, ce qui est facile étant donné son caractère quasi-complet ; on obtient `[3; 2; 1]` (par niveau de profondeur décroissant).
On renverse la liste initiale, puis extrait ses trois premiers éléments (un pour chaque feuille), que l'on utilise chacun comme racine d'un arbre dont les sous-arbres sont donnés par une second entrée initialisée pour l'occasion avec 6 arbres vides, et l'on crée une nouvelle liste avec le résultat :

```
(* « état » initial *)
inp1 : [5; 4; 3; 2; 1; 0]
inp2 : [E; E; E; E; E; E]
out  : []
(* après une étape *)
inp1 : [4; 3; 2; 1; 0]
inp2 : [E; E; E; E]
out  : [N (5, E, E)]
(* « état » final *)
inp1 : [2; 1; 0]
inp2 : []
out  : [N (3, E, E); N (4, E, E); N (5, E, E)]
```

La construction du niveau suivant se fait en extrayant les deux éléments suivants de la liste initiale, que l'on utilise comme racines d'arbres dont les sous-arbres sont calculés en fonction de la sortie de l'étape précédente. Plus précisément, on renverse pour cela les *paires* d'éléments de cette sortie, en ajoutant en tête le bon nombre de paires d'arbres vides (ce qui n'est pas visible dans notre exemple) et un éventuel arbre vide pour accompagner un arbre solitaire (visible ici). Ceci donne :

```
(* « état » initial *)
inp1 : [2; 1; 0]
inp2 : [N (5, E, E); E; N (3, E, E); N (4, E, E)]
out  : []
(* après une étape *)
inp1 : [1; 0]
inp2 : [N (3, E, E); N (4, E, E)]
out  : [N (2, N (5, E, E), E)]
(* après deux étapes *)
inp1 : [0]
inp2 : []
out  : [N (1, N (3, E, E), N (4, E, E));
```

```
        N (2, N (5, E, E), E)]
```

La construction du dernier (premier) niveau se fait de façon similaire :

```
(* « état » initial *)
inp1 : [0]
inp2 : [N (1, N (3, E, E), N (4, E, E));
        N (2, N (5, E, E), E)]
out  : []
(* après l'unique étape *)
inp1 : []
inp2 : []
out  : [N (0,
           N (1, N (3, E, E), N (4, E, E)),
           N (2, N (5, E, E), E))]
```

On peut argumenter le coût linéaire d'une telle approche en remarquant que chaque élément de la liste initiale n'est impliqué que dans un nombre constant d'opérations de liste de coût constant.

24. Écrivez une fonction OCaml :

`of_list : 'a list -> 'a farray`

telle que `of_list x` utilise l'approche esquissée ci-dessus pour construire un tableau dont les éléments sont donnés (dans l'ordre d'indice croissant) par ceux de `x`.

Remarque : on pourra utiliser la fonction `lsl` qui pour un `(i : int)` positif permet de calculer efficacement un `int` égal à 2^i comme `1 lsl i`.

Un intérêt de posséder une fonction `of_list` en temps linéaire est qu'elle permet (dans un usage conjoint avec `to_list`) d'implémenter la concaténation de deux tableaux en temps linéaire.

25. Écrivez une fonction OCaml :

`append : 'a farray -> 'a farray -> 'a farray`

telle que `append a1 a2` construit un nouveau tableau dont les éléments sont ceux de `a1` puis ceux de `a2`, par ordre d'indices respectifs croissant. Cette fonction devra être de coût linéaire en la somme des longueurs de `a1` et `a2`.