TP9: Tableaux redimensionnables

Dans ce TP on propose d'implémenter une structure de *tableaux redimensionnables* (ou "vector"). Il s'agit d'une structure de données *impérative*, stockant une séquence d'éléments, et supportant les opérations suivantes en temps constant (amorti) :

- lecture et écriture d'un élément à une position donnée
- insertion d'un nouvel élément à la fin
- suppression du dernier élément

On implémente cette structure grâce à un tableau, initialement rempli de "valeurs de remplissage", et que l'on remplit progressivement. Si celui-ci vient à être totalement rempli, alors on crée un nouveau tableau avec plus de place.

Définition de type

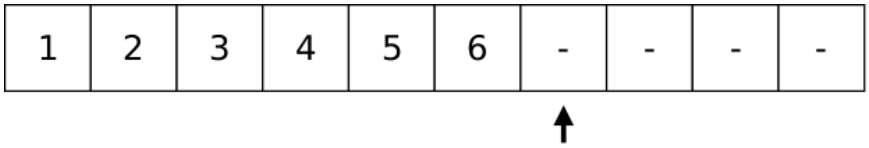

Figure 1: Schéma de la structure

Par exemple, le tableau redimensionnable ci-dessus contient la séquence `1, 2, 3, 4, 5, 6`. Les cases affichant `-` contiennent une valeur de remplissage. On stocke également la position de la prochaine case disponible (indiquée par la flèche).

Il est alors facile de rajouter un nouvel élément à la séquence : on peut le placer dans la case disponible à droite du `6`, et décaler la position sur la prochaine case disponible d'un cran vers la droite.

La définition de type correspondante est un type enregistrement :

```
type 'a vector = {
  mutable contents : 'a array;
  default : 'a;
  mutable size : int;
}
```

`contents` est le tableau stockant les éléments, `default` est la valeur de remplissage, et `size` le nombre d'éléments stockés dans le tableau (sans compter les valeurs de remplissage).

Note : si `t` est un tableau redimensionnable de type `'a vector`, les positions valides dans ce tableau sont donc les entiers compris entre 0 et `t.size - 1`. On ne considérera ici que des tableaux redimensionnables `t` pour lesquels `t.size <= Array.length t.contents`.

Compilation

Dans ce TP, on va compiler notre programme à la main.

Question 0. Dans un répertoire `PF5-TP9/`, créer un fichier `vector.ml`, et y recopier la définition du type `'a vector` décrit plus haut. Dans un terminal, vérifier que ça compile en tapant la commande : `ocamlc -c vector.ml`.

Pour ce TP, vous écrirez toutes les fonctions demandées dans le fichier `vector.ml`. On ne fournit pas de tests automatisés : vous devrez tester vos fonctions à la main, soit en utilisant le *toplevel* d'OCaml, soit en écrivant vos tests dans un autre fichier `test_vector.ml` que vous pouvez ensuite compiler pour créer un exécutable :

```
ocamlc -c test_vector.ml
ocamlc -o test vector.cmo test_vector.cmo
./test
```

Opérations de base

Pour l'instant, on ne s'occupe pas de redimensionner le tableau.

Question 1. Écrire une fonction `create : int -> 'a -> 'a vector` renvoyant un tableau redimensionnable vide. Son premier paramètre est la taille à utiliser pour le tableau `contents`. Son deuxième paramètre est la valeur de remplissage à utiliser.

Astuce : penser à utiliser `Array.make`.

Question 2. Écrire une fonction `of_list : 'a list -> 'a -> int -> 'a vector` permettant de créer un tableau redimensionnable à partir des éléments contenus dans une liste.

`of_list l default capacity` doit retourner un tableau redimensionnable contenant les éléments de `l`, avec un tableau sous-jacent `contents` de taille `capacity` et `default` comme valeur de remplissage.

Question 3. Écrire une fonction `get : 'a vector -> int -> 'a`. `get v i` doit renvoyer l'élément stocké à la position `i` dans le tableau `v`. Elle doit échouer en lançant l'exception `Invalid_argument "get"` si la position est invalide (en particulier, il n'est pas acceptable de renvoyer une valeur de remplissage).

Question 4. Écrire une fonction `set : 'a vector -> int -> 'a -> unit`. `set v i x` doit stocker l'élément `x` à la position `i` dans `v`. Elle doit échouer en lançant l'exception `Invalid_argument "set"` si la position est invalide.

Question 5. Écrire une fonction `equal : ('a -> 'a -> bool) -> 'a vector -> 'a vector -> bool`, permettant de comparer deux tableaux redimensionnables.

`equal eq v1 v2` doit retourner `true` si `v1` et `v2` contiennent les mêmes éléments dans le même ordre, et `false` sinon. Pour comparer les éléments contenus dans

le tableau, on utilisera la fonction `eq` fournie en argument.

Question 6. Écrire une fonction `clear : 'a vector -> unit` qui enlève tous les éléments contenus dans le tableau (en réécrivant des valeurs de remplissage à la place).

Astuce : penser à utiliser `Array.fill`.

Ajout progressif d'éléments

Question 7. Écrire une fonction `push_back : 'a vector -> 'a -> unit`, ajoutant un élément à la fin du tableau. `push_back v x` doit stocker l'élément `x` à la suite des éléments déjà contenus dans le tableau `v`.

Pour l'instant, si le tableau est plein, on échouera en lançant une exception `Failure "push_back"`

Question 8. Écrire une fonction `pop_back : 'a vector -> 'a option`, enlevant le dernier élément du tableau. Si `v` est non vide, en nommant `x` le dernier élément, alors `pop_back v` doit enlever ce dernier élément et retourner `Some x`. Si le tableau est vide, la fonction doit retourner `None`.

Note : il est important de réécrire la valeur de remplissage dans la case de tableau ainsi libérée.

Question 9. Écrire une fonction `append : 'a vector -> 'a vector -> unit` permettant de copier les éléments d'un tableau dans un autre.

`append v1 v2` doit copier les éléments de `v2` dans le tableau `v1`. Pour l'instant, s'il n'y a pas assez de place dans `v1`, on échouera en lançant une exception `Failure "append"`.

Astuce : penser à utiliser `Array.blit`.

Redimensionnement

On veut maintenant faire grandir automatiquement le tableau `contents` contenu dans l'enregistrement, afin qu'il soit toujours possible de rajouter un élément avec `push_back` ou transférer des éléments grâce à `append`.

Question 10. Écrire une fonction `resize : 'a vector -> int -> unit`.

`resize v new_size` doit modifier `v` pour contenir un nouveau tableau `contents` de la taille indiquée `new_size`, et contenant toujours les mêmes éléments.

La fonction doit échouer en lançant une exception `Invalid_argument "resize"` si la taille `new_size` indiquée est trop petite.

Question 11. Modifier `push_back` pour automatiquement redimensionner le tableau si celui-ci est plein, *en doublant sa taille.*

Question 12. Il pourrait arriver que l'on fasse grandir le tableau `contents` en faisant une suite de `push_back`, puis de tous les ejecter avec `pop_back`. Afin de ne pas consommer trop de mémoire inutilement dans ce cas, modifier `pop_back` pour réduire de moitié la taille du tableau `contents` lorsque le nombre d'éléments descend en dessous du *quart* de sa taille.

Question 13. Modifier `append` pour automatiquement redimensionner le tableau si celui-ci n'est pas assez grand pour contenir tous les éléments.

Itération

On veut fournir un moyen de parcourir facilement les éléments contenus dans le tableau redimensionnable.

Question 14. Écrire une fonction `iter : ('a -> int -> unit) -> 'a vector -> unit`. `iter f v` doit appeler la fonction `f` sur chaque élément contenu dans le tableau ainsi que sa position.

Question 15. On veut maintenant permettre à la fonction `f` d'arrêter le parcours. Modifier `iter` pour stopper le parcours lorsque la fonction `f` lance l'exception `Break`.

Question 16. En utilisant la fonction iter, écrire une fonction `print_int_vector : int vector -> unit` qui affiche le contenu d'un vecteur d'entiers (sans les valeurs de remplissage !).

Question 17. Un utilisateur étourdi a écrit le code suivant dans son fichier `test_vector.ml` :

```
let () =
  let t = Vector.create 2 0 in
  Vector.push_back t 1;
  Vector.push_back t 2;
  Vector.push_back t 3;
  t.size <- 10;
  Vector.print_int_vector t
```

Que se passe-t-il lorsqu'on exécute ce bout de code ? Pourquoi ? Écrire un fichier d'interface `vector.mli` afin de rendre abstrait le type `'a vector`. Essayer à nouveau de compiler le code ci-dessus, et constater qu'il ne compile plus.