

DM5 MP2I : piles et files + une touche de OCaml

Une partie des fonctions à écrire pour ce devoir fera l'objet d'évaluation dans le DS3 du lundi 12 janvier. Il est donc recommandé de les travailler sérieusement.

Le devoir, à faire seul ou en binôme, est à rendre pour le dimanche 11 janvier 23h45. le rendu comporte les fichiers **projet.c** et **transpose.ml** et rien d'autre.

1 Piles et files

Objectif : listes doublement chaînées et adaptation aux piles et files. Le Makefile et le fichier de tests sont un peu compliqués : il n'est pas demandé de les comprendre en détails ni de savoir les refaire sans aide. Il faut écrire le fichier **projet.c**.

1.1 Présentation

Les prototypes sont décrits dans le fichier **projet.h**. Un Makefile pour la partie en C est fourni :

- **make** pour tout compiler (ne pas oublier de définir la variable de Makefile **TYPE** pour obtenir des listes d'entiers, de flottants ou de chaînes de caractères) ;
- **make clean** pour nettoyer ;
- **make run** pour exécuter.

Un fichier de tests **tests.c** est donné.

Les différentes sorties attendues selon qu'on a appelé le Makefile avec la définition de variable **TYPE=int**, **TYPE=float** ou **TYPE=string** correspondent aux 3 fichiers **compilation_avec*.txt**

La fonction **print_one** affiche un objet de type **DATA** (ce type est fixé à la compilation). Son code est adapté au moment du préprocessing selon qu'on a défini la variable de Makefile comme **TYPE=int**, **TYPE=float** ou **TYPE=string**. On peut dans la suite passer indifféremment **print_one**, **&print_one** ou ***print_one** en paramètre d'une fonction, le résultat sera le même puisque cela désigne la même fonction.

La structure de liste doublement chaînée comporte trois champs :

- **tail** et **head** (pointeurs vers le dernier et le premier maillon) ;
- **size** qui renseigne en $O(1)$ sur la longueur de la liste et qu'il faut mettre à jour à chaque retrait/ajout.

Les maillons ont également 3 champs : un pour la valeur, un pointeur sur le maillon suivant et un autre sur le précédent. Le pointeur **pred** (resp. **next**) du premier (resp. du dernier) maillon vaut **NULL**.

Dans le cas d'une compilation avec l'option **string**, les chaînes de caractères sont des chaînes littérales non allouées dynamiquement (il n'est pas nécessaire de les libérer ni les allouer).

1.2 Codes à écrire

Question 1.

Écrire la fonction

```
1 dlist_t* dlist_empty(void)
```

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

Question 2.

Écrire la fonction

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

qui ajoute la donnée `x` en tête de liste. Le booléen 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`).

Question 3.

Écrire la fonction

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

qui ajoute la donnée `x` en queue de liste. Le booléen 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`).

Question 4.

Écrire la fonction

```
1 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.

Question 5.

Écrire la fonction

```
1 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, sinon on renvoie `true`. Si le pointeur `out` n'est pas vide, on le fait pointer sur la valeur du maillon retiré.

Question 6.

Écrire la fonction

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

qui donne le nombre d'emaillons en $O(1)$.

Question 7.

Écrire la fonction

```
1 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`. Le pointeur `l` n'est pas libéré.

Question 8.

Écrire la fonction

```
1 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`. Le pointeur `l` n'est pas libéré.

Pour l'affichage, il faut se rappeler qu'on passe en paramètre de `dlist_print` un pointeur sur la fonction `print_one`. Or, en C, l'opérateur d'appel `(...)` force la déréréférence automatique d'un pointeur sur fonction. Autrement dit, ces deux écritures sont équivalentes :

```
1 print_one(p->data);
2 (*print_one)(p->data);
```

Question 9.

Écrire la fonction

```
1 void dlist_print(const dlist_t *l, print_data_fn print_one, const char *sep)
```

Elle affiche le contenu de la liste pointée par `l` de la tête vers la queue. La fonction `print_one` affiche un objet de type `DATA` (cf. `tests.c`, pour voir comment s'adapte - pendant la compilation - `print_one` au type `DATA` véritablement choisi). Le séparateur `sep` (typiquement une virgule, un point-virgule ou une flèche) indique comment séparer l'affichage de la valeur d'un maillon de celle du maillon suivant.

Question 10.

Notre structure de liste peut implémenter les celles de piles ou de files. Écrire les 4 fonctions :

```
1 /* ===== API pile ===== */
2 bool stack_push(dlist_t *l, DATA x)
3 bool stack_pop(dlist_t *l, DATA *out)
4 /* ===== API file ===== */
5 bool queue_enqueue(dlist_t *l, DATA x)
6 bool queue_dequeue(dlist_t *l, DATA *out)
```

Le travail à faire pour ces 4 fonctions n'est pas énorme !

2 Transposée

Un peu de OCaml dans un fichier `transpose.ml`.

Question 11.

Écrire une fonction `encapsule (l: 'a list) : 'a list list` qui encapsule chaque élément de `l`.

```
1 | # encapsule [1;2;3];;
2 | - : int list list = [[1]; [2]; [3]]
```

Question 12.

Écrire une fonction `ajoute_col (l:'a list) (m: 'a list list) : 'a list list` qui ajoute un élément de `l` à chacune des lignes de la « matrice » `m`.

Une exception est soulevée si les dimensions ne sont pas compatibles.

```
1 | # let l = [-1;-2;-3] and
2 |     m = [[1;2];[3;4];[5;6]] in
3 |     ajoute_col l m;;
4 | - : int list list = [[-1; 1; 2]; [-2; 3; 4]; [-3; 5; 6]]
```

Question 13.

Écrire une fonction `transpose (m:'a list list) : 'a list list` qui renvoie la transposée de la matrice `m`.

```
1 | # let m = [[1;2];[3;4];[5;6]] in
2 |   transpose m;;
3 | - : int list list = [[1; 3; 5]; [2; 4; 6]]
```