Cours de programmation séquentielle

Vecteur générique sur une liste chaînée

1 Buts

— Utilisation de liste chaînée.
— Allocation dynamique de mémoire.
— Utilisation de pointeurs de fonctions.
— Utilisation de `void *` pour la généricité.

2 Énoncé

En `C` il existe deux façons de créer des tableaux.

1. Les tableaux statiques, stockés sur la pile, qui se déclarent comme ceci:
```
int size = 10;
int tab[size];
```

2. Les tableaux dynamiques, stockés sur le tas, qui se déclarent comme cela:
```
int size = 10;
int *tab = malloc(size * sizeof(*tab));
// On pourrait aussi écrire sizeof(int)
```

Notez comme dans les deux cas les tableaux sont de type `int` (un entier).

Une fois ces tableaux créés il n'est pas possible de rajouter ou d'enlever des éléments aux tableaux de façon simple. Dans ce travail pratique, il s'agit d'implémenter un type *vecteur* (c'est comme ça qu'ils sont appelés en général). Pour ce faire vous allez implémenter une structure `lst_vector`, qui offre la possibilité d'ajouter/enlever efficacement des éléments sur une liste chaînée. Cette façon de faire est efficace pour l'ajout/extraction d'éléments en début de vecteur, mais moins efficace lorsqu'il s'agit d'accéder un élément particulier en milieu de vecteur (pour le lire, le modifier, l'ajouter). Cette structure aura une interface similaire à ce qui existe dans les librairies standard de la plupart des langages modernes (mais pas en `C`) et sera représentée par une structure de donnée de liste chaînée (ce qui n'est pas le cas pour les langages modernes qui préfèrent les tableaux dynamiques [1]).

2.1 La structure `lst_vector`

Pour ce faire, il faut déclarer une structure `lst_vector`

1. Le but ici est de vous exercer à utiliser une liste chaînée.

```c
typedef struct _element {
   void *data;
   struct _element* next;
} element;

typedef element* lst_vector;
```

Chaque élément de notre liste, contient un pointeur vers les données et un pointeur vers l'élément suivant. Nous voulons que le type des données soit générique, par exemple un `int`, un `char`, etc... Pour avoir cette généricité nous utiliserons le type `void *`. C'est un moyen en `C` pour que les données soient génériques, on peut mettre ce que l'on veut à l'adresse pointée par `data`. Cela comporte évidemment des risques car aucune vérification de type ne peut être faite.

2.2 Les fonctionnalités de `lst_vector`

Puis il faudra implémenter les fonctions suivantes:

1. Une fonction `lst_vector_init()` qui retourne un vecteur vide.
2. Une fonction `lst_vector_length(lst_vector v)` qui retourne la longueur d'un vecteur.
3. Une fonction `lst_vector_push(lst_vector *v, void *data)` qui ajoute `data` au début du vecteur et le retourne.
4. Une fonction `lst_vector_pop(lst_vector *v)` qui retire le premier élément du vecteur passé en argument et retourne les données qu'il contient.
5. Une fonction `lst_vector_set(lst_vector *v, int index, void *data)` qui assigne la `index`-ème valeur du vecteur à la valeur de `data` et retourne le vecteur.
6. Une fonction `lst_vector_get(lst_vector *v, int index)` qui retourne les données contenues dans le `index`-ème élément du vecteur (sans le retirer l'élément du vecteur).
7. Une fonction `lst_vector_remove(lst_vector *v, int index)` qui retire l'`index`-ème élément du vecteur et retourne les données qu'il contient.
8. Une fonction `lst_vector_insert(lst_vector *v, void *data, int index)` qui insère `data` au `index`-ème indice du vecteur et retourne le vecteur.
9. Une fonction `lst_vector_empty(lst_vector *v)` qui vide un vecteur, libère la mémoire et met la tête à `NULL`.

Puis implémenter également deux fonctions un peu plus complexes syntaxiquement.

10. Une fonction `lst_vector_map(lst_vector *v, void *(*f)(void *))` qui itère sur tous les éléments du vecteur `v`, applique la fonction `f` sur les données de chaque élément, et retourne un nouveau vecteur avec le résultat.
11. Une fonction `lst_vector_filter(lst_vector *v, bool (*f)(void *))` applique le prédicat `f` sur toutes les données contenues dans les éléments

d'un vecteur et retourne ceux qui le satisfont dans un nouveau vecteur. Les éléments qui ne satisfont pas le prédicat doivent être libérés.

Afin d'utiliser les fonctions `lst_vector_map()` et `lst_vector_filter()` vous devez écrire deux fonctions. La première, `square`, calculera le carré d'un élément. La seconde, `is_even`, vérifiera si `data` est pair. Vous pouvez implémenter d'autres fonctions si vous le souhaitez.

2.3 La gestion des erreurs

Un certain nombre de fonctions peuvent échouer (`push`[2], `get`, `set`, `pop`, ...). Dans ce cas, il suffit de retourner un vecteur `NULL`. Cette façon de faire ne donne pas une très grande granularité aux erreurs possibles, mais est bien plus simple.

2.4 Tests

Écrivez un petit programme avec des assertions permettant de tester votre code (voir le cours du jour, bonjour).

En fait, si vous le souhaitez, vous pouvez commencer par écrire les tests pour chaque fonction, et petit à petit écrire le code pour que les tests passent! Cela s'appelle le "test-driven-development" en bon français.

2. La seule façon pour que `push` échoue est si le vecteur est `NULL`.