TP8 : Listes doublement chaînées

MP2I Lycée Pierre de Fermat

Vous pouvez télécharger sur Cahier de Prépa une archive pour ce TP.

Nous avons vu en cours et en TP qu'il est possible d'implémenter certaines structures en utilisant des listes chaînées. En réalité, les listes chaînées peuvent elle-même être vues comme des structures de données abstraites !

Dans ce TP, on se propose d'implémenter une structure de liste **doublement** chaînée, où chaque maillon pointe vers les deux maillons adjacents.

Définition 1. Une **liste doublement chaînée**, qu'on appellera **liste** dans ce TP, est une SDA permettant de stocker une suite finie d'éléments. Cette structure utilise une sous-structure, appelée **maillon**. Une liste contient plusieurs maillons mis à la suite, et chaque maillon a une référence vers les deux maillons adjacents. De plus, deux maillons sont particuliers : la tête et la queue.

On propose les opérations suivantes pour les listes :

- Création d'une liste vide
- Déterminer la taille d'une liste
- Ajouter un élément à l'avant de la liste
- Ajouter un élément à l'arrière de la liste
- Récupérer le maillon à l'avant de la liste
- Récupérer le maillon à l'arrière de la liste
- Supprimer un maillon donné
- Insérer un élément après un maillon donné
- Récupérer le maillon suivant d'un maillon donné
- Récupérer le maillon précédent d'un maillon donné
- Récupérer le contenu d'un maillon, c'est à dire la valeur qu'il stocke.

Par exemple, si l'on considère la liste suivante, dessinée de l'avant vers l'arrière :

```
1 <-> 6 <-> 3
```

Si l'on récupère le premier maillon, on obtient un maillon m_0. Si l'on récupère le contenu de ce maillon, on obtient 1. Si l'on récupère le maillon suivant de m_0, on obtient un maillon m_1 dont le contenu est 6. Si l'on supprime m_1, alors la liste devient :

```
1 <-> 3
```

et le maillon m_1 est invalide, on ne peut plus l'utiliser.

Question 1. En partant de la liste $L =$ `1 <-> 6 <-> 3`, exécutez le pseudo-code suivant :

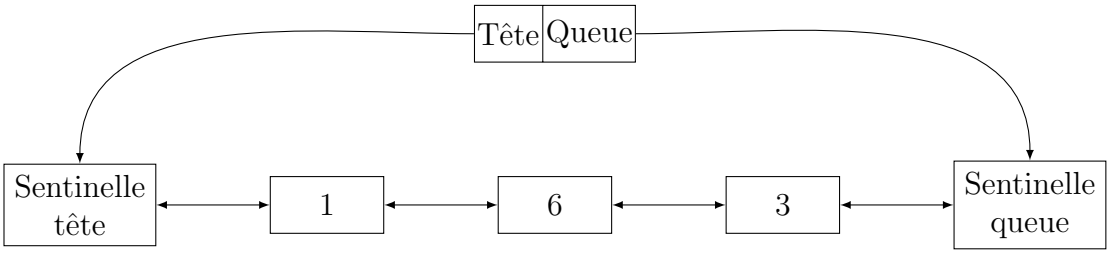
1 $M_0 \leftarrow$ maillon à l'avant de L;
2 Ajouter 8 à l'avant de L;
3 $M_1 \leftarrow$ maillon précédent de M_0;
4 Insérer 4 après M_1;
5 Supprimer M_1;
6 Insérer 2 après M_0;

Nous allons implémenter cette structure en utilisant des structures concrètes très proches de celles utilisées pour les implémentations des piles/files. On se propose d'étudier une variante : les **listes à sentinelles**. Les sentinelles sont deux maillons ne contenant pas de données mais étant toujours présents, un à la tête, un à la queue. **Les sentinelles ne comptent pas comme des maillons de la structure abstraite !**

Par exemple, on considère la liste (abstraite) suivante :

```
1 <-> 6 <-> 3
```

Dans l'implémentation concrète, on aura le schéma suivant en mémoire :



En particulier, pour une liste vide, la structure concrète aura tout de même deux maillons, les sentinelles, qui pointeront l'une vers l'autre.

On utilisera les structures suivantes en C :

```c
struct maillon {
  int data;
  struct maillon* suiv; // de la tête vers la queue
  struct maillon* prec; // de la queue vers la tête
};

typedef struct maillon maillon_t;

struct liste {
  maillon_t* tete;
  maillon_t* queue;
};

typedef struct liste liste_t;
```

Question 2. Récupérez dans l'archive du TP les fichiers `liste.h` et `liste.c`. Ils contiennent pour l'instant les déclarations et définitions des types structs, ainsi qu'un constructeur, une fonction d'affichage et une fonction de libération de mémoire. Lisez le code pour vous familiariser avec la structure.

Question 3. Créez un fichier `main.c` contenant deux fonctions : `int main()` et `void test()`. La fonction `test` contiendra des tests pour toutes les opérations que l'on implémentera, et la fonction `main` appellera juste la fonction test.

Question 4. Pour commencer les tests, créez une liste vide, affichez-la, et libérez-la immédiatement.

Sentinelles On remarque que si `m` est un vrai maillon, i.e. pas une sentinelle, alors ni son suivant ni son précédent n'est `NULL`. On remarque également que même si la tête et la queue de la SDA changent, dans la structure concrète, les sentinelles seront toujours à la queue et à la tête.

Question 5. Dans `liste.c`, écrivez une fonction `bool est_sentinelle` prenant en entrée un maillon et déterminant si c'est une sentinelle.

Vous pourrez utiliser la fonction précédente dans les assertions des prochaines fonctions : lorsqu'un maillon n'est **pas** une sentinelle, ses maillons voisins ne sont pas NULL, on peut donc les déréférencer.

Nous allons maintenant implémenter toutes les opérations de la SDA. A chaque fois, on écrira la spécification dans `liste.h`, puis on rajoutera des tests dans `main.c`, et enfin on implémentera l'opération dans `liste.c`.

Ajouts On commence par l'ajout à l'avant / à l'arrière.

Question 6. Faites un dessin pour voir comment les différents pointeurs changent au cours d'un ajout à l'avant d'une liste, et déduisez-en la fonction C correspondante.

Question 7. Par symétrie, écrivez une fonction pour l'ajout à l'arrière. Testez-bien les deux fonctions.

Maillons et parcours

Question 8. Implémentez l'opération permettant de récupérer le contenu d'un maillon. On aura comme précondition que le maillon n'est ni nul, ni une sentinelle.

Question 9. Implémentez l'opération permettant de récupérer le maillon à l'avant de la liste. Si la liste est vide, la valeur renvoyée sera `NULL`. Attention, cette fonction doit renvoyer un maillon du point de vue de la SDA, et donc ne doit jamais renvoyer une sentinelle.

Question 10. Par symétrie, implémentez l'opération permettant de récupérer le maillon à l'arrière.

Question 11. Implémentez les fonctions permettant de récupérer le maillon suivant et le maillon précédent d'un maillon. Le maillon en entrée ne devra pas être une sentinelle.

Insertion et suppression On implémente maintenant l'opération d'insertion. Cette opération prend en entrée un maillon M et un élément x, et crée un nouveau maillon entre M et $\mathbf{suiv}(M)$ contenant x. Si M est la queue de la liste, alors l'opération revient à ajouter l'élément en queue de liste.

Question 12. Faites un schéma pour représenter l'insertion d'un élément après un maillon, afin de repérer les différents liens à modifier. N'oubliez pas que même si $\mathbf{suiv}(M)$ peut être `NULL` du point de vue de la SDA, le maillon après lequel on insère n'est pas `NULL` du point de vue de la SDC, car l'utilisateur n'est pas sensé appeler la fonction d'insertion sur une sentinelle (il n'est même pas sensé avoir accès aux sentinelles).

Question 13. Implémentez l'insertion, la fonction doit être en complexité $\mathcal{O}(1)$. N'oubliez pas de bien tester !

Question 14. De même, implémenter la suppression. En précondition de cette fonction, on aura que le maillon à supprimer est un vrai maillon, et pas une sentinelle.

Utilité des sentinelles

Question 15. Réécrivez les fonctions d'ajout à l'avant/arrière en une seule ligne, en utilisant la fonction d'insertion.

Question 16. Afin de vous convaincre encore plus de l'utilité des sentinelles, essayez d'écrire les fonctions d'insertion et de suppression sans présupposer que les maillons en entrée ne sont pas des sentinelles. Une fois que vous êtes convaincu/e du gain de simplicité, dites "merci les sentinelles" et passez à la partie suivante

Assurez-vous que votre SDA fonctionne correctement jusqu'ici, avant de passer à la suite.

Taille Pour implémenter le calcul de la taille, une première idée est d'utiliser les opérations définies précédemment pour parcourir la liste :

Algorithme 1 : Taille

Entrée(s) : L liste
Sortie(s) : Nombre d'éléments dans L

```
1 c ← 0// compte le nombre d'éléments
2 M ← tête de L;
3 tant que M n'est pas NULL faire
4 |   c++;
5 |   M ← suivant(M);
6 retourner M
```

Le problème est que cette implémentation est très coûteuse : $\mathcal{O}(n)$. Pour améliorer cela, on propose d'ajouter à la structure concrète un attribut donnant la taille. A chaque ajout / insertion / suppression de maillon, on incrémente ou décrémente l'attribut. La fonction **taille** peut alors s'implémenter en $\mathcal{O}(1)$. Cependant, lorsque l'on insère un maillon, on précise le maillon après lequel insérer, mais on n'a aucun moyen d'accéder à la liste elle-même. Nous allons donc également rajouter un attribut sur les maillons : chaque maillon possède un pointeur `.mere` vers la liste le contenant.

Question 17. Mettre en place les modifications proposées et implémenter l'opération de taille en $\mathcal{O}(1)$.

Utilisation de la SDA Tout le code suivant est à faire dans `main.c`, vous ne pouvez donc utiliser que les opérations de la SDA, présentes dans le fichier liste.h, vous n'avez pas le droit de faire référence aux attributs de la structure concrète.

Question 18. Écrire une fonction permettant de rechercher un élément donné dans une liste chaînée. Cette fonction prendra en entrée la liste ainsi que l'élément à rechercher, et renverra un booléen indiquant la présence ou non de l'élément.

Question 19. Écrire une fonction `liste_t * tri_X( liste_t * l)` qui vide intégralement `l`, et en renvoie une version triée. Vous pouvez procéder par tri insertion, sélection, rapide ou fusion (et vous devez donc nommer la fonction de la manière adaptée).

(Facultatif) Application : dictionnaire

Nous allons utiliser la SDA de liste que nous venons de créer pour implémenter une autre SDA : les dictionnaires.

Définition 2. Soient K, V deux ensembles. Un dictionnaire est une SDA stockant des couples $(k, v) \in K \times V$. Les éléments de K sont appelés "clés" et ceux de V "valeurs". Pour (k, v) un couple dans D, on dit que la valeur v est associée à la clé k. Un dictionnaire est tel que pour toute clé $k_0 \in K$, au plus un couple (k, v) du dictionnaire est tel que $k = k_0$.

Un dictionnaire peut donc être vu comme une fonction au sens mathématique : à certains éléments de K, on associe des éléments de V selon une relation fonctionnelle : chaque élément de k est associé à au plus un élément de V.

Les opérations sont :

— Créer un dictionnaire vide

— Déterminer si une clé est dans un dictionnaire

— Récupérer la valeur associée à une clé donnée

— Modifier la valeur associée à une clé donnée

— Supprimer une clé, c'est à dire supprimer le couple contenant cette clé.

Par exemple, le dictionnaire suivant associe à certains prénoms un nom :

```
Caroline  -> Goutelard
Guillaume -> Rousseau
Olivier   -> Ginoux
```

Si l'on modifie la valeur associée à `Olivier` en `Giroud`, le dictionnaire devient :

```
Caroline  -> Goutelard
Guillaume -> Rousseau
Olivier   -> Giroud
```

Si l'on modifie la valeur associée à `Alexandre` en `Grothendieck`, le dictionnaire devient :

```
Caroline  -> Goutelard
Guillaume -> Rousseau
Olivier   -> Giroud
Alexandre -> Grothendieck
```

Nous allons implémenter un dictionnaire où les clés et les valeurs sont des chaînes de caractères `char*`. Pour cela, nous allons utiliser une liste. Cette liste stockera des couples (k, v), et on s'assurera toujours de n'avoir qu'une seule occurrence de chaque clé.

Question 20. Créez un nouveau dossier, et copiez-y les fichiers `liste.h` et `liste.c`. Copiez-y également les fichiers `dict.h` et `dict.c` qui se trouvent dans l'archive du TP.

Regardez le contenu des deux fichiers de l'archive : le .h contient les déclarations des différentes opérations des dictionnaires, et le .c contient du code pour démarrer, notamment la structure suivante :

```
1 struct cle_valeur {
2   char* cle;
3   char* valeur;
4 };
5 typedef struct cle_valeur cv_t;
```

Question 21. Modifiez le code des listes pour que les éléments stockés ne soient plus des `int` mais des `cv_t*`.

Question 22. Lisez le code de la fonction `bool in(dict_t* d, char* k)`. Cette fonction utilise des opérations sur les listes, mais n'utilise pas les noms de fonctions que vous avez mis dans votre header des listes. Remplacez les noms de fonction pour pouvoir utiliser votre implémentation des listes avec ce code.

Question 23. Sur le même modèle que la fonction précédente, implémenter la fonction `char* get(dict_t* d, char* k)` qui renvoie la valeur correspondant à une clé. Si k n'est pas dans d, on renverra `NULL`.

Question 24. Implémenter la fonction `void set(dict_t* d, char* k, char* v)` qui associe v à k dans d. Attention à ne pas créer de doublons si k est déjà dans d!

Application : doublons Les dictionnaires permettent de détecter les doublons dans un tableau :

Algorithme 2 : Doublons

Entrée(s) : T tableau de taille n
Sortie(s) : Oui si T contient un doublon, i.e. deux indices $i \neq j$ tels que $T[i] = T[j]$,
Non sinon

```
1 D ← dictionnaire vide;
  // Invariant: D a pour clé les T[j] avec j ∈ [[0, i − 1]]
  // Invariant: Il n'y a aucun doublons parmis les cases T[0], ...T[i − 1]
2 pour i = 0 à n − 1 faire
3     si T[i] est une clé de D alors
          // Il existe j ∈ [[0, i − 1]] tel que T[j] = T[i]
4         retourner Oui
5     D[T[i]] = 1;
  // Il n'y a aucun doublon dans tout le tableau
6 retourner Non
```

Question 25. Implémenter et tester cet algorithme.