Université de Paris
Master 1

Langages à Objets Avancés
Année 2022—2023

TP n°5

Pas de grandes nouveautés pour ce TP, profitez-en pour gagner en confiance en programmation c++. Coder des listes chaînées est un exercice très classique que vous avez déjà certainement fait dans d'autres langages. Testez régulièrement votre travail sur des exemples.

Listes doublement chaînées

On rappelle qu'on implémente les listes en utilisant deux classes : les cellules, et une encapsulation de cellule (ce qui permet de définir la liste vide l'identifier à `nullptr`)

Lorsque la liste est dite doublement chaînée, les cellules sont composées de trois champs : son contenu, un pointeur vers la cellule précédente et un pointeur vers la cellule suivante. Ces pointeurs sont `nullptr` en cas d'absence de précédent ou de suivant. [1]

Ces champs seront évidemment encapsulés et cachés au monde extérieur, qui ne pourra accéder à la liste qu'au travers d'un certain jeu de méthodes garantissant que la liste préserve une structure cohérente.

On se focalisera ici sur les listes chaînées d'entiers.

Exercice 1 [Cellule]

1. Écrire la classe `Cell`.
   Cette classe contient, outre les 3 champs déjà mentionnés, un constructeur adéquat, une méthode `connect` permettant de connecter deux cellules (pensez à modifier le champs `next` de l'une et `previous` de l'autre) et les méthodes `disconnect_next` et `disconnect_previous` (idem : pensez à mettre à jour l'ancienne cellule voisine).
2. Si on veut faire jouer un rôle symétrique aux deux cellules que l'on connecte, en permettant un appel de la forme `Cell::connect(c1, c2)` (au lieu de `c1.connect(c2)`), quelle sera la déclaration correcte de cette méthode ?
3. Faites en sorte que le monde extérieur ne puisse pas modifier des cellules de façon incohérente (notamment, pour toute cellule `c`, il faut que la cellule précédente de la suivante de `c` soit toujours `c`). Pour cela, jouez sur les modificateurs de visibilité (`private`) et ajoutez des accesseurs en lecture seule s'il le faut.

Exercice 2 [Liste]

On écrit maintenant la classe `List` qui doit fournir les méthodes usuelles :

— `int length()` : longueur de la liste ;
— `int get(int idx)` : valeur du `idx`-ième élément de la liste ;
— `int find(int val)` : indice de la valeur `val` si elle existe dans la liste, `-1` sinon ;
— `void set(int idx, int val)` : affecte la valeur `val` à la position `idx` de la liste ;
— `void insert(int idx, int val)` : insère la valeur `val` en position `idx` (et décale les éléments qui suivent) ;
— `void delete(int idx)` : supprime la valeur d'indice `idx` (et décale les éléments qui suivent).

1. Vous pourrez vous interroger sur les raisons qui font que l'on utilise des pointeurs et non des références.

1. Écrivez la classe `List`, munie de champs privés pointant sur la première et la dernière de ses cellules, d'un constructeur instanciant une liste vide, un destructeur qui désalloue les cellules de la liste et les méthodes mentionnées ci-dessus.
2. Testez sur des exemples significatifs : le tri bulle d'une liste, la fusion de 2 listes triées.
3. Ajustez l'encapsulation de la classe `Cell`, afin que seule la classe `List` puisse instancier et manipuler des cellules (qui ne sont qu'un intermédiaire technique pour implémenter une liste chaînée et n'ont pas vocation à être visibles pour les autres classes).

   Indice : il faudra utiliser **private** et **friend**.
4. Ajoutez un constructeur de copie sur les listes. Avez vous pensé à ce qu'il se passe lors de l'affectation entre listes ? (c. à d. lorsque vous écrivez `l1 = l2`)
5. Testez toutes les méthodes ! Comment peut-on faire pour tester les valeurs des champs et méthodes privés, et malgré tout regrouper tous les tests dans une classe séparée ?