UNIVERSITÉ PARIS SUD

Travaux Dirigés d'algorithmique n°5

Cours d'algorithmique

—Licence CFA / Troisième Année—

Listes chaînées

Dans ce TD, nous étudions la manipulation des listes chaînées. Même si de nombreux langages modernes gère automatiquement la mémoire, on écrira, dans un premier temps, explicitement les allocations et désallocations en évitant les fuites de mémoire. En effet, dans de nombreux cas pratiques, la gestion de la mémoire prends une part importante du temps de calcul et doit donc être prise en compte dans les calculs de complexités.

On rappelle les déclarations du type `LinkedList` vu en cours, où `[...]` signale qu'une partie du code à été coupée. Note importante : Dans les déclarations ci-dessous, on ne s'occupe pas des questions d'interfaces, mais seulement d'algorithmique. Ainsi tout est déclaré `public`, ce qui n'est bien évidement pas la bonne manière de faire. C'est un bon exercice de programmation/génie logiciel que de faire le tri entre ce qui est montré et caché...

```
1   public class LinkedList<Elem> {
2       static class RefCell<Elem> {
3           public Cell<Elem> next;
4           public RefCell() [...];
5           public RefCell(Cell<Elem> cell) [...];
6           [...]
7       }
8       static class Cell<Elem> extends RefCell<Elem> {
9           public Elem value;
10          public Cell(Elem value, Cell<Elem> next) [...]
11          public Cell(Elem value) [...]
12          }
13          [...]
14      }
15      public RefCell<Elem> start;
16      public LinkedList() [...]
17      public LinkedList(Elem e) [...]
18      [...]
19  }
```

▸ **Exercice 1. (Constructeurs)**

1. Écrire les constructeurs des classes ci-dessus.

▸ **Exercice 2. (Destruction et copie d'une liste chaînée)**

1. Écrire une méthode `is_empty` qui répond si `this` est vide ou non.
2. Écrire une méthode `clear` qui vide la liste `this`. On écrire deux codes : l'un en Java standard, l'autre en écrivant explicitement les délallocations.
3. Écrire une méthode `clone` qui retourne un clone de `this`.
4. En déduire un constructeur de copie.

▸ **Exercice 3.** Écrire les méthodes suivantes en précisant leurs complexités ?

- insertion en tête ;
- insertion en queue ;
- suppression en tête ;
- suppression en queue ;
- insertion en position i ;
- suppression en position i ;

▸ **Exercice 4.** Concaténation.

1. Écrire une méthode pour concaténer deux listes ; On détaillera l'état des deux listes après le retour de la fonction.

▸ **Exercice 5.** Pour réduire la complexité de l'ajout en queue on peut garder un pointeur sur la dernière cellule de la liste. Dans cet exercice, on travaille en mutatif avec une fausse tête

1. Écrire les déclarations correspondantes et les constructeurs.

Écrire les méthodes suivantes en donnant leurs complexités :

2. test à vide ;
3. insertion en tête ;
4. insertion en queue ;
5. suppression en tête ;
6. suppression en queue ;
7. concaténation ;

Une autre variante consiste à utiliser des listes circulaires où l'on pointe sur la dernière cellule qui elle même pointe sur la première...

▸ **Exercice 6.** Dans l'exercice précédent, la suppression en queue à un coût linéaire. Pour réduire ce coût, on utilise des listes doublements chaînées : chaque cellule garde un pointeur sur l'élément suivant et l'élément précédent. Reprendre l'exercice précédent en utilisant des listes doublement chaînées.