Listes doublement chaînées

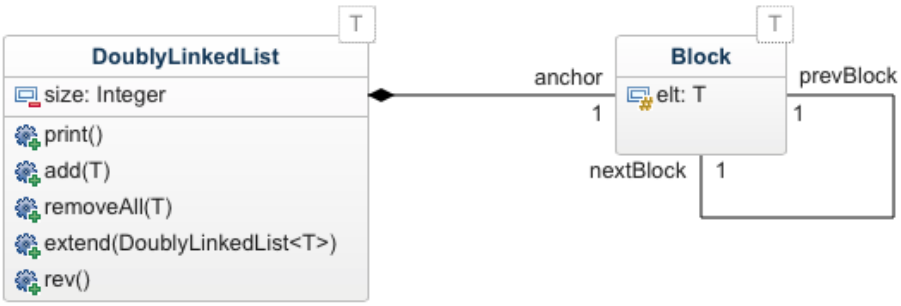
POGL, TP5. Invariants, assertions et lambda-expressions.

Dans ce TP, nous allons réaliser une structure de liste doublement chaînée, et utiliser des assertions pour vérifier que les invariants de cette structure sont toujours valides. Ce sera aussi l'occasion d'utiliser quelques lambda-expressions. Un squelette de code est à télécharger à l'adresse suivante :
`http://www.lri.fr/~blsk/POGL/TDTP/DLL.java`

Listes doublement chaînées

Une liste doublement chaînée est constituée d'une séquence de blocs contenant chacun une valeur (définie), dans laquelle chaque bloc possède un pointeur vers son successeur et un pointeur vers son prédécesseur.

Pour réaliser une telle structure, on propose l'organisation présentée par le diagramme suivant :



Pour que chaque bloc, y compris le premier ou le dernier, ait à la fois un prédécesseur et un successeur on aura recours à l'astuce suivante : chaque liste doublement chaînée contiendra, en plus de ses blocs normaux contenant les éléments de la liste, un bloc spécial appelé *ancre* (`anchor`). L'ancre est le prédécesseur du premier bloc utile, et aussi le successeur du dernier bloc utile. Dans le cas d'une liste vide, l'ancre est son propre prédécesseur et son propre successeur. L'attribut `elt` de l'ancre vaut systématiquement `null`.

1. Donner des diagrammes d'instances pour la liste vide et pour la liste contenant les éléments 4, 2 et 1 dans cet ordre.
2. Dans le squelette de code, déclarer les attributs de la classe `DoublyLinkedList<T>` et sa classe interne `DoublyLinkedList<T>.Block`. Définir un constructeur `DoublyLinkedList()` construisant une liste vide et un constructeur `Block(T elt)` construisant un bloc contenant l'élément `elt`. Définir dans la classe `DoublyLinkedList` une méthode `public void print()` qui affiche dans l'ordre les éléments de la liste. Pour l'instant, on ne s'occupe pas des autres opérations. Dans une méthode `main`, définir deux listes correspondant aux deux exemples précédents et les afficher.
3. Énoncer les invariants de notre structure de liste chaînée. Regarder notamment les aspects suivants :
   — définition ou non-définition des champs `elt`,
   — symétrie des liens prédécesseur-successeur,
   — valeur du champ `size`.
4. Donner des diagrammes d'instances violant chacun de ces invariants.

Vérification des invariants par des assertions

Pour rattraper les erreurs aussi tôt que possible, nous allons écrire du code chargé de vérifier les invariants. Les tests correspondants seront intégrés dans des instructions `assert`, dont on rappelle qu'elles ne sont exécutées que lorsque l'option `-ea` est activée.

Les méthodes suivantes sont à définir dans la classe `DoublyLinkedList`.

5. Définir une méthode `private int countElements()` qui compte le nombre d'éléments de la liste, c'est-à-dire le nombre de blocs utiles. En déduire une première version d'une méthode

`private boolean checkInvariants()` qui teste si la taille de la liste correspond au nombre de ses éléments.

6. Importer l'interface `java.util.function.Predicate`, et se renseigner sur son contenu sur une ressource adéquate.
7. Définir une méthode `private boolean forAllBlocks(Predicate<Block> p)` qui renvoie `true` si tous les blocs de la liste, ancre comprise, vérifient le prédicat `p`, et renvoie `false` sinon. Compléter alors la méthode `checkInvariants()` de manière à ce qu'elle vérifie tous les invariants.
   La méthode `forAllBlocks` peut être basée sur un itérateur comme le `DLLIterator` fourni, ou peut coder elle-même le parcours des blocs. Essayez les deux versions !
8. Vérifier sur les exemples précédents que les instances correctes sont validées et que les instances incorrectes sont rejetées.

Manipulation de listes doublement chaînées

Nous allons maintenant pouvoir définir les opérations des listes doublement chaînées. Vous pourrez inclure des assertions dans chaque opération modifiant une liste pour vous assurer que la modification respecte les invariants.

Les méthodes suivantes sont à nouveau à définir dans la classe `DoublyLinkedList`.

9. Définir une méthode `public void add(T elt)` qui ajoute un élément à la fin de la liste.
10. Voici une proposition de code pour une méthode qui permute les deux premiers éléments de la liste :

```
public void swap() {
    Block pivot = this.anchor.nextBlock.nextBlock.nextBlock;
    this.anchor.nextBlock = this.anchor.nextBlock.nextBlock;
    this.anchor.nextBlock.nextBlock = this.anchor.nextBlock.prevBlock;
    this.anchor.nextBlock.nextBlock.nextBlock = pivot;
}
```

    Tester ce code avec la séquence d'instructions suivante :

```
DoublyLinkedList<Integer> list = new DoublyLinkedList<>();
list.add(1);
list.add(2);
list.print();
list.swap();
list.print();
list.add(3);
list.print();
```

    Que penser de tout cela ?
11. Définir une méthode `public void removeAll(T elt)` qui supprime toutes les occurrences d'un élément.
12. Définir une méthode `public void extend(DoublyLinkedList<T> l)` qui accole une liste à la fin de la liste courante.
13. Définir une méthode `public void rev()` qui renverse la liste (les prédécesseur et successeur de chaque bloc sont inversés).