Algorithmique
TP 2 : Structures de données linéaires

Avertissement : *les bibliothèques standards de la plupart des langages de programmation (dont Java) viennent avec leurs propres implémentations de listes, qu'il conviendra d'utiliser dans la grande majorité des cas, plutôt que toute implémentation « maison » moins optimale (cela est vrai pour la plupart des structures de données et algorithmes de base).*
Cela étant dit, implémenter sa propre liste est un exercice intéressant et très souvent proposé dans différents TP d'informatique. C'est le cas de celui-ci !

1 Écrire sa propre liste

On considère dans cet exercice qu'il n'existe pas d'API Java pour les listes acycliques [1] et qu'il faut implémenter une telle structure de données. Pour simplifier la programmation, on considère uniquement les listes contenant comme données des entiers (classe `Integer`).

1. Dans le fichier `CellInteger.java`, définir une classe publique `CellInteger` qui représente une cellule de liste d'entiers. Cette liste doit avoir deux attributs privés : `data` de type `Integer` (qui représente un élément de contenu de la liste) et `next` de type `CellInteger` (qui contient une référence vers la cellule suivante dans la liste, ou `null` s'il s'agit de la dernière cellule). *Note :* Dans la plupart des implémentations existantes, les listes sont doublement chaînées afin d'obtenir des opérations sur les listes plus efficaces.
2. Définir un constructeur sans paramètre de `CellInteger` qui initialise le contenu de la cellule à `0` et un constructeur qui prend comme argument la valeur du contenu. Dans les deux constructeurs, l'attribut `next` vaut `null`.
3. Définir les méthodes d'accès en lecture et écriture aux attributs de la classe : `Integer getData()`, `CellInteger getNext()`, `void setData(Integer d)` et `void setNext(CellInteger n)`.
4. Définir une méthode `String toString()` qui retourne une chaîne représentant le contenu de la cellule.
5. Définir la méthode `main` qui alloue deux cellules de liste, les initialise avec les valeurs 3 et 5 et affiche leur contenu.
6. Dans le fichier `ListInteger.java`, définir une classe publique `ListInteger` qui représente une liste d'entiers. Cette classe a un attribut privé `head` de type `CellInteger`.
7. Définir un constructeur sans paramètre de `ListInteger` qui initialise la liste à vide.
8. Définir une méthode `void addFirst(Integer d)` qui ajoute une cellule contenant `d` au début de la liste.
9. Définir une méthode `String toString()` qui retourne une chaîne représentant (dans l'ordre) les données contenues dans toutes les cellules de la liste.

1. Cette API est, en réalité, disponible à l'adresse `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html`

10. Définir une méthode `main` qui alloue une liste, y ajoute deux éléments et l'affiche. Cette méthode vous servira à tester les méthodes suivante.
11. Définir une méthode `void add(Integer d)` qui ajoute l'entier `d` à la fin de la liste.
12. Définir une méthode `Integer element()` qui renvoie le premier entier de la liste ou `null` si la liste est vide.
13. Définir une méthode `int size()` qui renvoie la taille de la liste.
14. Définir une méthode `boolean contains(Integer d)` qui renvoie `true` si et seulement si la liste contient une cellule dont la donnée est égale à `d`.
15. Définir une méthode `boolean remove(Integer d)` qui enlève la première cellule qui contient l'entier `d` ; la méthode renvoie `true` si la liste a changé suite à cette opération.
16. Définir une méthode `Integer get(int i)` qui renvoie le `i`ème entier dans la liste ou `null` si la taille de la liste est plus petite que `i`.
17. Définir une méthode `Integer set(int i, Integer d)` qui affecte le `i`ème entier dans la liste à `d` ; elle renvoie l'ancienne valeur de la cellule à cette position ou `null` si la liste n'a pas de `i`ème élément.
18. Définir une méthode `Integer removeFirst()` qui enlève la cellule en tête de liste et renvoie sa donnée comme résultat.
19. Tester dans le `main` toutes les méthodes implémentées.

2 Notation polonaise inverse ✍

La notation polonaise inverse sert a écrire les expressions arithmétiques sans parenthèses. Dans cette notation on écrit d'abord deux opérandes, puis l'opérateur : ainsi, l'expression 10 4 - signifie 10 - 4. On peut aussi enchainer plusieurs opérations : par exemple, l'expression 10 4 - 7 * signifie (10 - 4) * 7. Cet exercice a comme objectif de vous faire utiliser l'interface `Deque`[2] pour coder la fonction `eval` qui permet de calculer la valeur d'une expression en notation polonaise inverse.

1. Dans le fichier `Polonaise.java`, définir l'attribut privé `expression` de type `String[]` qui représente une expression en notation polonaise inverse. Par exemple, si l'expression est `7 3 + 6 4 * - 2 /` alors l'attribut privé expression sera le tableau de `String` suivant `"7", "3", "+", "6", "4", "*", "-", "2", "/"`.
2. Définir un constructeur de `Polonaise` qui initialise l'expression en utilisant une chaîne de caractères. Vous pouvez utiliser l'appel `chaine.split("␣")` (méthode de la classe `String`) pour diviser la chaîne de caractères `chaine` en le tableau des mots qu'elle contient, délimités par des espaces.
3. Définir une méthode `String toString()` qui retourne l'expression courante sous la forme d'une chaîne de caractères.
4. Définir la méthode `Integer eval()` qui évalue l'expression courante et renvoie sa valeur si elle est correcte ou `null` sinon.

2. Cette API est disponible à l'adresse `https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html`.

5. Verifier que votre méthode `eval` passe tous les tests fournis.

Quelques informations sur comment utiliser l'interface `Deque` :

— Un objet de type `Deque<ElementType>` est une structure dans laquelle on peut ajouter et retirer des éléments qui sont des objets de type `ElementType` au début ou à la fin (ce qui permet d'implémenter des files, des piles, des queues...).

— Un moyen d'initialiser une instance de `Deque<Integer>` (c'est-à-dire, un objet de ce type-là) :
`Deque<Integer> deque = new ArrayDeque<>();`.

— Les fonctions usuelles de manipulation de cette structure sont dans la doc. En particulier, on a les méthodes `addFirst(Type x)` et `addLast(Type x)` qui ajoutent un élément en début et en fin. Les méthodes pour retirer un élément sont `removeFirst()` et `removeLast()`, elles renvoient l'élément retiré. Ces fonctions peuvent renvoyer des exceptions (par exemple, quand on veut retirer un élément dans un `Deque` vide).