Dans ce TP vous allez implémenter les listes immutables d'entiers présentées en cours, ainsi que des méthodes pour les manipuler. Lorsque le choix est possible, vous privilégierez un algorithme récursif à un algorithme itératif.

Exercice 1

1. Écrivez une classe `IntFList` ayant les attributs suivants :
   — `public static final IntFList NULL_INTLIST = new IntFList();`
   — `private final boolean empty;`
   — `private final int first;`
   — `private final IntFList rest.`

   Pourquoi tous ces attributs sont déclarées `final` ?
2. Redéfinissez le constructeur sans arguments qui construit une liste vide.
3. Ajoutez un constructeur à deux arguments qui prends un entier `e` et une liste `ll`, et construit la liste ayant `e` comme premier élément et `ll` comme queue.

Exercice 2

Implémentez les opérations de base sur les listes `IntFList`. Dans une classe `Test` vérifiez leur comportement.

1. `public boolean isNull()` : vérifie si une liste est vide.
2. `public int head()` : rend le premier élément de la liste.
3. `public IntFList tail()` : donne le reste de la liste.
4. `public IntFList cons(int e)` : définition dynamique du "cons". Ajoute l'élément `e` en tête de liste.
5. `public IntFList add(int e)` : ajoute l'élément `e` à la fin de la liste.
6. `public int length()` : rend le nombre d'éléments de la liste (c'est-à-dire, sa longueur).
7. `public int sum()` : rend la somme de tous les éléments de la liste. Si la liste est vide, la somme doit donner 0.
8. `publicboolean int ordered()` : donne `true` si les éléments de la liste sont triée en ordre croissant, `false` sinon.
9. `public int listRef(int k)` : récupère le k-ième élément de la liste, donne -1 s'il n'existe pas. Attention : la numération commence à 0, donc `ll.listRef(0)` rend la tête de la liste.
10. `public boolean contains(int e)` : donne `true` si l'élement `e` appartient à la liste, `false` sinon.
11. `public IntFList remove(int e)` : si l'élément `e` appartient à la liste, sa première occurrence est supprimé. Si l'élément n'appartient pas, alors la liste reste inchangée.
12. `public IntFList remove_last(int e)` : si l'élément `e` appartient à la liste, sa dernière occurrence est supprimé. Si l'élément n'appartient pas, alors la liste reste inchangée.
13. `public String toString()` : pretty printing de la liste. Le symbole '`(`' dénotera le début de la liste, le symbole '`)`' sa fin, les éléments de la liste seront séparés par des virgules. Par exemple, la liste contenant les éléments 1,2,3,4 (en cet ordre) sera représenté par `(1,2,3,4)`. Suggestion : utiliser une méthode auxiliaire `toStringRec()` qui renvois la chaine de caractère avec les virgules sans parenthèse. Elle sera appelé dans `toString()`.

Exercice 3

Implémentez les opérations suivantes sur les listes et ajoutez des tests pour vérifier leur comportement.

1. `public IntFList append(IntFList il)` où `l1`, `l2` sont deux listes. La méthode `l1.append(l2)` crée une liste constituée de la juxtaposition de `l1` et `l2`. Par exemple, si `l1 = (1,2)` et `l2 = (3,4)` alors `l1.append(l2)` renvoie `(1,2,3,4)` et `l2.append(l1)` renvoie `(3,4,1,2)`.
2. `public IntFList reverse()` : méthode dynamique qui calcule l'inverse de la liste. Par exemple, si `l = (1,2,3,4)` est une liste, alors `l.reverse() = (4,3,2,1)`. Suggestion : utiliser une méthode auxiliaire `reverseRec(IntFList acc)` qui sera appelée par `reverse()` sur la liste vide. L'argument `acc` pourra être ensuite utilisé pour accumuler la liste inversée. Dans le cas de base, il suffira de rendre `acc` comme valeur de retour.
3. `public boolean equals(IntFList il)` : implémente l'égalité structurelle entre listes. La méthode `l.equals(il)` donne `true` si et seulement si les listes `l` et `il` contiennent les mêmes éléments, dans le même ordre.
4. `public static IntFList range(int inf, int sup)` : si inf <= sup alors cette méthode (statique) construit la liste qui représente l'intervalle [inf, sup].