Algorithmique avancée - TD2

—o000o——o000o—

Listes chaînées : copie par adresse ou valeur, récursivité

1 Ajoutez les tous !

On souhaite pouvoir ajouter tous les éléments d'une liste L_2 aux éléments déjà présents dans une liste L_1 selon différentes méthodes. Pour cela, on va étendre les méthodes disponibles pour la classe `SList<T>` en créant une nouvelle classe `TD2<T>` qui hérite de `SList<T>` et lui ajouter les méthodes décrites dans les questions ci-après.

1. Écrire une méthode nommée `addAll(TD2<T> L)` qui permet d'ajouter les éléments de la liste L **à la fin** de ceux déjà présents dans la liste courante. Vous pouvez ou non utiliser les méthodes existantes dans `SList<T>` pour répondre à la question. Vous écrirez une version qui copie la référence de la liste `L` et une version qui construit une copie des valeurs de la liste `L`.

2. Écrire ensuite une méthode itérative nommée `addAllIndex(TD2<T> L, int index)` qui permet d'insérer les éléments de la liste L **à l'indice** *index*.

3. Proposer une méthode `static` et **récursive** nommée `concat(TD2<T> L1, TD2<T> L2)` qui construit une nouvelle liste contenant les éléments de `L1` et `L2`. Pour écrire une méthode récursive, la technique consiste à écrire une méthode privée `static` qui travaille sur `Node<T>`. Attention, dans le cas des méthodes `static`, pour que le type générique `T` soit reconnu, il faut faire une méthode générique également (voir le programme du 1er semestre).

Solution :

```
1  package td.td2;
2
3  import list.SList;
4  import list.Node;
5
6  public class TD2_1<T> extends SList<T> {
7
8      /**
9       * Q1. add all elements from L to the current list
10      * @param L
11      */
12     // version utilisant les methodes de l'interface LList<T
       >, pas de copie des valeurs
```

```
13      public void addAll(TD2_1<T> L){
14          if (head == null) {
15              head = L.head;
16              last = L.last;
17              size = L.size;
18          }
19          else {
20              last.next = L.head;
21              last = L.last;
22              size += L.size;
23          }
24      }
25
26      // version qui ajoute une copie des valeurs
27      public void addAllCopy(TD2_1<T> L){
28          Node<T> p = L.head;
29          while (p != null){
30              Node<T> tmp = new Node(p.value);
31              if (last == null){
32                  head = tmp;
33                  last = head;
34                  size = 1;
35              } else {
36                  last.next = tmp;
37                  last = last.next;
38                  size ++;
39              }
40              p = p.next;
41          }
42      }
43
44      private static <T> Node<T> concatNode(Node<T> L1, Node<T>
            L2){
45          if (L1 == null) return L2;
46          else if (L2 == null) return L1;
47          else {
48              return new Node(L1.value, concatNode(L1.next, L2)
                    );
49          }
50      }
51
52      public static <T> TD2_1<T> concat(TD2_1<T> L1, TD2_1<T>
            L2){
53          TD2_1<T> L = new TD2_1<>();
```

```
54          L.head = concatNode(L1.head, L2.head);
55          return L;
56      }
57
58      public void insert(TD2_1<T> L, int index){
59          if (index == 0){
60              // insert at beginning
61              L.last.next = head;
62              head = L.head;
63          } else if (index > size){
64              // insert in the end
65              last.next = L.head;
66              last = L.last;
67          } else {
68              // insert at position index
69              int i = 0;
70              Node<T> p = head;
71              while (i < index-1){
72                  p = p.next;
73                  i = i + 1;
74              }
75              Node<T> tmp = p.next; // memorize p.next
76              p.next = L.head;
77              L.last.next = tmp;
78          }
79          size += L.size;
80      }
81
82      public static void main(String[] args) {
83          TD2_1<Integer> L = new TD2_1<>();
84          L.add(1); L.add(2); L.add(3);
85          TD2_1<Integer> M = new TD2_1<>();
86          M.add(4); M.add(5);
87
88          System.out.println(L);
89          System.out.println(M);
90
91          TD2_1<Integer> tmp = new TD2_1<>();
92          //tmp.addAll(L);
93          //tmp.addAllCopy(M);
94          //tmp.insert(M, 1);
95          tmp = TD2_1.concat(M,L);
96          System.out.println(tmp);
97      }
```

```
98
99 }
```

java/TD2_1.java

2 Palindrome

On s'intéresse désormais à déterminer si une liste chaînée ne contenant que des caractères de type `Character` représente un palindrome. Le problème est que traditionnellement avec les tableaux nous pouvons utiliser la taille du tableau et les indices pour déterminer rapidement s'il représente un palindrome. Avec les listes chaînées, la méthode consiste à créer une chaîne inverse qui possède les éléments dans l'ordre inverse et de tester l'égalité de la chaîne initiale et de la chaîne inverse. Comme pour l'exercice précédent, on hérite de la classe `SList<T>` une classe `TD2_2` qui ne contient que des `Character` pour travailler.

1. Écrire une méthode `equals(TD2_2 L)` qui retourne `true` si et seulement si la liste courante et L contiennent des éléments de même valeurs et contiennent le même nombre d'éléments.

2. Proposer une méthode itérative `reverse()` qui modifie la liste courante en plaçant toutes ses valeurs en ordre inverse. Vous pourrez utiliser le constructeur de `Node` pour cela et ensuite changer l'adresse du nœud de départ de la liste avec le champ `head`.

3. Proposer une méthode récursive `reverseRec()` qui fait le même traitement en suivant la méthode étudiée à l'exercice précédent.

4. Écrire enfin la méthode `palindrome()` qui indique si une liste de type `TD2_2` représente un palindrome ou pas. Vous pourrez utiliser les méthodes précédentes pour cela.

Solution :

```java
1  package td.td2;
2
3  import list.SList;
4  import list.Node;
5
6  import java.util.Objects;
7
8  public class TD2_2 extends SList<Character> {
9
10     // Exercice 2 - PALINDROME
11     public boolean equals(TD2_2 L){
```

```
12          if (this.isEmpty()) return L.isEmpty();
13          else {
14              Node<Character> p = head;
15              Node<Character> q = L.head;
16              boolean res = true;
17              while (p != null && q != null && res) {
18                  res = Objects.equals(p.value, q.value);
19                  p = p.next;
20                  q = q.next;
21              }
22              return res;
23          }
24      }
25
26      public void reverse() {
27          if (!this.isEmpty()) {
28              Node<Character> p = head;
29              Node<Character> res = new Node<>(p.value);
30              while (p.next != null) {
31                  p = p.next;
32                  res = new Node<>(p.value, res);
33              }
34              head = res;
35          }
36      }
37
38      private Node<Character> reverseRecNode(Node<Character> L)
            {
39          if (L == null || L.next == null) return L;
40          else {
41              // at lest 2 elements in L
42              Node<Character> res = reverseRecNode(L.next);
43              Node<Character> p = res;
44              while (p.next != null) {
45                  p = p.next;
46              }
47              p.next  = new Node<>(L.value);
48              return res;
49          }
50      }
51
52      public void reverseRec() {
53          head = reverseRecNode(head);
54      }
```

```java

    // palindrome!
    public boolean isPalindrome() {
        // create inverse list
        TD2_2 inverse = new TD2_2();
        inverse.head = reverseRecNode(this.head);
        return equals(inverse);
    }


    public static void main(String[] args) {
        TD2_2 mot1 = new TD2_2();
        mot1.add('s'); mot1.add('o'); mot1.add('s');

        System.out.println(mot1.isPalindrome());
    }
}
```

java/TD2_2.java