Algorithmique avancée - TD5

—o000o——o000o—

Implémentation des itérateurs pour listes chaînées

Préambule

Les itérateurs sont des objets qui permettent un parcours simplifié et sécurisé d'une structure de type liste. En Java classique, ils sont accessibles depuis l'interface `Iterator<T>` et notamment la classe `listIterator<T>` qui instancie cette interface. L'idée de cette séance est de dériver notre classe `SList<T>` et de l'enrichir avec cette interface `Iterator<T>`. Pour cela, nous allons créer une nouvelle interface `IList<T>` qui étend `LList` pour avoir l'ensemble des méthodes des listes et également `Iterable<T>` pour avoir accès aux itérateurs.

Principe général Un **itérateur** donne accès à 2 méthodes principales :

1. `boolean hasNext()` qui retourne `true` si et seulement si la structure de liste possède un élément suivant.
2. `T next()` qui retourne la valeur suivante dans la liste et avance d'une position.

Réalisation pour réaliser cet **itérateur**, la solution la plus simple consiste à ajouter le code suivant à votre classe :

```
public Iterator<T> iterator() {
        return new DummyIterator<>(this);
}
public class DummyIterator<T> implements Iterator<T> {
         // a completer
}
```

Problèmes et solutions cette implémentation pose plusieurs problèmes :

- il faut que le premier appel à `hasNext()` vérifie que la liste chaînée n'est pas vide
- il faut que le premier appel à `next()` retourne le 1er élément de la liste en cours de parcours

Pour résoudre ces problèmes :

- un itérateur sera représenté sous la forme d'un nouveau pointeur (en plus des pointeurs `head` et `last`) nommé `current` et ajouté à la classe principale,

- ce pointeur pourra être utilisé pour paramétrer le comportement de `hasNext()` et `next()`
- comme un nouvel itérateur sera créé à chaque parcours, il faut que le constructeur de `DummyIterator` ré-initialise le pointeur courant.

1 Implémentation des méthodes obligatoires

Question 1 Créez une interface `IList<T>` et la classe `ISList<T>` qui dérive de `SList<T>` et implémente l'interface `IList<T>`

Question 2 Ajoutez le champ `Node<T> current` à votre classe `ISList<T>`.

Question 3 Complétez les méthodes `boolean hasNext()` et `T next()` sur la base des explications précédentes dans la classe interne `DummyIterator`.

2 Ré-écriture des méthodes de la classe `SList`

Question 4 Réécrivez les méthodes suivantes en les précédant de la mention `@Override` :

- `boolean contains(T elem)`
- `int indexOf(T elem)`
- `T get(int index)`
- `void set(T elem, int index)`

Solution :

```java
package list;

public interface IList<T> extends LList<T>, Iterable<T> {

}
```

java/IList.java

Solution :

```java
package list;

import java.util.Iterator;
```

```java
import java.util.function.Consumer;

public class DummyIteratorExt<T> implements Iterator<T> {

    private ISList2<T> internal;

    public DummyIteratorExt(ISList2<T> internal) {
        this.internal = internal;
        this.internal.current = this.internal.head;
    }

    @Override
    public boolean hasNext(){
        return this.internal.current != null;
    }

    @Override
    public T next() {
      if (this.internal.current != null){
            T tmp = this.internal.current.value;
            this.internal.current = this.internal.current.
               next;
            return tmp;
      } else return null;
    }

    @Override
    public void listIterator() {
        current = internal.head;
    }
}
```

java/DummyIteratorExt.java

Solution :

```java
package list;

import java.util.Iterator;
import java.util.Objects;

public class ISList2<T> extends SList<T> implements IList<T>
   {

```

```
8
9       protected Node<T> current;
10
11      /**
12       * Create an empty list
13       */
14      public ISList2(){
15          super();
16      }
17
18      public ISList2(T value) {
19          super(value, null);
20      }
21
22      public ISList2(T value, ISList2<T> tail) {
23          super(value, tail);
24      }
25
26
27      /**
28       * Add an element to the end of the list
29       * @param elem
30       */
31      @Override
32      public void add(T elem) {
33          if (isEmpty()){
34              this.head = new Node<T>(elem);
35              this.last = head;
36              size = 1;
37          } else {
38              last.next = new Node<T>(elem);
39              last = last.next;
40              size +=1;
41          }
42      }
43
44      @Override
45      public void add(T elem, int index) {
46          Node<T> p = this.head;
47          Node<T> tmp = new Node<T>(elem);
48          if (index == 0) {
49              tmp.next = head;
50              head = tmp;
51          } else {
```

```
52              int i = 1;
53              while (p.next != null && i < index){
54                  p = p.next;
55                  i++;
56              }
57              tmp.next = p.next;
58              p.next = tmp;
59          }
60      }
61
62      /**
63       * Remove all elements of the list
64       */
65      @Override
66      public void clear() {
67          head = null;
68          last = null;
69          size = 0;
70      }
71
72      /**
73       * Note: rewriting contains with iterator
74       * @param elem
75       * @return
76       */
77      @Override
78      public boolean contains(T elem) {
79          Iterator<T> it = iterator();            // reset
               iterator
80          while(it.hasNext()){
81              if (Objects.equals(elem, it.next())) return true;
82          }
83          return false;
84      }
85
86      @Override
87      public boolean isEmpty() {
88          return head == null;
89      }
90
91      // Note: rewriting indexOf with iterator
92      @Override
93      public int indexOf(T elem) {
94          Iterator<T> it = iterator();            // reset
```

```
        iterator
95          int index = -1;
96          while (it.hasNext()){
97              index ++;
98              if (Objects.equals(it.next(), elem)) return index
                    ;
99          }
100         return index;
101     }
102
103
104     @Override
105     public void remove(int index) {
106         if (index < size){
107             if (index == 0) this.head = this.head.next;
108             else {
109                 int cpt = 0;
110                 Node<T> p = head;
111                 while (cpt < index - 1) {
112                     p = p.next;
113                     cpt++;
114                 }
115                 p.next = p.next.next;
116             }
117             size --;
118         }
119     }
120
121     // Note: rewriting get with iterator
122     @Override
123     public T get(int index) {
124         Iterator<T> it = iterator();            // reset
            iterator
125         int cpt = 0;
126         T res = null;
127         while (it.hasNext() && cpt <= index){
128             res = it.next();
129             cpt ++;
130         }
131         if (cpt < index) return null;
132         else return res;
133     }
134
135     // Note: rewriting set with iterator
```

```java
    @Override
    public void set(T elem, int index){
        Iterator<T> it = iterator();            // reset iterator
        int cpt = 0;
        T tmp = null;
        while (it.hasNext() && cpt < index){ // notice < strict to stop before as next will advance the pointer
            tmp = it.next();
            cpt ++;
        }
        if (cpt == index) current.value = elem;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer("(");
        Iterator<T> it = iterator();
        while (it.hasNext()){
            sb.append(it.next());
            if (it.hasNext()) sb.append(",");
        }
        sb.append(")");
        return sb.toString();
    }

    @Override
    public Iterator<T> iterator() {
        return new DummyIteratorExt<>(this);
    }
}
```

java/ISList2.java