Laboratoire 10 – Itérateurs

ITI 1521. Introduction à l'informatique II

Semaine du 5 Avril 2021

Non Dû (Aucune soumission n'est exigée) - En démonstration seulement

- ## Objectifs
  • Utiliser les **itérateurs** pour **résoudre** des problèmes
  • **Implémenter** le concept d'itérateur pour une structure de données à base de tableau.

Introduction

Pour ce laboratoire, vous utiliserez une liste chaînée afin de sauvegarder un nombre illimité de bits : des zéros et des uns. Les valeurs à l'intérieur des noeuds sont des entiers (**int**).

Partie 1 : Classe `IterList`

L'implémentation de la classe **`IterList`** doit sauvegarder les bits dans l'ordre «droite à gauche», c.-à-d. le bit le plus à droite en première position, premier noeud.
Par exemple, les bits «11010» doivent être sauvegardés dans une liste tel que le premier noeud contienne 0, le second 1, suivi d'un 0, puis 1, et 1 :
-> 0 -> 1 -> 0 -> 1 -> 1
La classe possède deux constructeurs, un sans argument et un avec un argument ; **public IterList (String str)** qui doit créer une liste représentant la chaîne de 0s et 1s donnée en entrée.
La chaîne **str** ne doit contenir que des 0s et des 1s sinon il faudra lancer l'exception ***IllegalArgumentException***. Le constructeur initialise cette nouvelle liste de bits afin de représenter la valeur de la chaîne. Chaque caractère de la chaîne représente un bit de la liste. Par exemple, étant donné la chaîne «1010111», le constructeur doit initialiser cette liste afin d'y inclure les bits suivants.
-> 1 -> 1 -> 1 -> 0 -> 1 -> 0 -> 1
Si la chaîne est vide, le constructeur doit créer une liste vide - la valeur **null** n'est pas valide.
Le constructeur ne doit pas retirer les zéros de la partie gauche. Par exemple, étant donné «0001» le constructeur doit initialiser cette liste comme suit.
-> 1 -> 0 -> 0 -> 0

Complétez l'implémentation de la classe **`IterList`**, en complétant le constructeur ainsi que les méthodes *addFirst* et *removeFirst*, à la place indiquée ci-contre.

Un programme **`Test1`** vous est fourni ci-joint pour tester.

• Iterator.java
• IterList.java
• *Test1.java*

```
/*Classe IterList. Stores the bits in reverse order!*/
public class IterList {
   // Constants
   public static final int ZERO = 0;
   public static final int ONE = 1;

   // Instance variables
   private Node head;

   // Constructors
   public IterList() {
      head = null;
   }

   public IterList( String str ) {
       À COMPLÉTER
   }

   // Instance methods
   /*
    * add an item at the head of the list
    */
   public void addFirst( int elem ) {
            À COMPLÉTER
   }

    /*
    * remove and return the first item of the list
    */
   public int removeFirst() {
            À COMPLÉTER
   }


  /*
    * iterator
    */
   public Iterator iterator() {
      return new IterListIterator();
   }

    /** toString method **/
   public String toString() {
      String str = "";
      if ( head == null ) {
         str += ZERO;
      }
      else {
         Node p = head;
         while ( p!=null ) {
            str = p.value + str; // reverses the order!
            p = p.next;
         }
      }
      return str;
}
```

```
   // The implementation of the nodes (static nested class)
   private static class Node {
      private int value;
      private Node next;

      private Node( int value, Node next ) {
         this.value = value;
         this.next  = next;
      }
   }

   // The implementation of the iterators (inner class)
   private class IterListIterator implements Iterator {
      private Node current = null;

      private IterListIterator () {
         current = null;
      }

      public boolean hasNext() {
         return ( ( current == null && head != null )   ||
                  ( current != null && current.next != null ) );
      }

      public int next() {
         if ( current == null ) {
            current = head ;
         }
         else {
            current = current.next ;
         }

         if ( current == null ) {
            throw new NoSuchElementException() ;
         }
         return current.value ;
      }

      public void add( int elem ) {
         if ( ( elem != ZERO ) && ( elem != ONE ) ) {
            throw new IllegalArgumentException( Integer.toString( elem ) );
         }

         Node newNode;
         if ( current == null ) {
            head = new Node( elem, head );
            current = head;
         }
         else {
            current.next = new Node( elem, current.next );
            current = current.next;
         }
      }//end of add
   } //end of IterListIterator

} //end of IterList calss
```

Partie 2 : `Iteration`

Implémentez les méthodes suivantes dans la classe **`Test2`** ci-contre. Vos solutions doivent être itératives (utilisent des itérateurs de la partie 1).

2.1. Méthode *complement*: static IterList complement(IterList input)

Écrire une méthode statique itérative retournant une nouvelle liste de bits (**IterList**) qui soit le complément de celle en entrée (input). Le complément de 0 est 1 ; et vice-versa. Le complément d'une liste de bits est une nouvelle liste, de même longueur que l'entrée, telle que chaque bit est le complément du bit à la même position dans la liste d'entrée. Voici 4 exemples.

1011
0100

0
1

01
10

0000111
1111000

2.2. Méthode : static IterList or (IterList list1, IterList list2)

Écrire une méthode de classe retournant le ou des deux listes passées en paramètre. Cette liste a la même longueur que les listes en entrée et chacun des bits de cette liste est le ou des bits à cette position dans les listes en entrée. La méthode lance une exception, ***IllegalArgumentException***, si l'une ou l'autre des deux listes est vide ou si les listes sont de longueur différente.

-Exemple:
list1 = 10001
list2 = 00011
list1 or list2 = 10011

/**********Classe Test2 ************/**

```
public class Test2 {

   public static IterList complement( IterList  input ) {
      IterList result;
      result = new IterList();

      Iterator i = input.iterator();
      Iterator j = result.iterator();
                           À COMPLÉTER
      return result;

   }//end of complement
```

```
public static IterList or ( IterList list1, IterList list2 ) {

    IterList result;
    result = new IterList();

    Iterator i = list1.iterator();
    Iterator j = list2.iterator();

    Iterator k = result.iterator();

    if ( ! i.hasNext() ) {
        throw new IllegalArgumentException( "list1 is empty" );
    }

    if ( ! j.hasNext() ) {
        throw new IllegalArgumentException( "list2 is empty" );
    }


    while ( i.hasNext() ) {
        À COMPLÉTER
    }//end of while

    if ( j.hasNext() ) {
        throw new IllegalArgumentException( " list2 is longer than list1 " );
    }
    return result;

}//end of IterList or



public static void main( String[] args ) {
    IterList list1, list2;
    for ( int i=0; i<args.length; i++ ) {
        list1 = new IterList( args[i] );

        System.out.println( "> " + list1 );
        System.out.println( "< " + complement( list1 ) );
        System.out.println();
    }

    list1 = new IterList( "10001" );
    list2 = new IterList( "00011" );

    System.out.println( "list1 = " + list1 );
    System.out.println( "list2 = " + list2 );
    System.out.println( "list1 or list2 = " + or( list1, list2 ) );
}//end of main

}//end of Test2
```

- Iterator.java
- IterList.java
- Test2.java

Partie 3 : Classe `CircularQueue`

Cette partie porte sur les concepts de file et d'itérateur. La classe **`CircularQueue`** utilise un tableau circulaire de taille fixe afin de sauvegarder les éléments de la file, et possède aussi un itérateur.

```
public interface Iterator <E> {
// Returns the next element in the iteration.
E next ( ) ;
// Returns true if the iteration has more elements.
boolean hasNext ( ) ;
}
```

• Cette implémentation utilise un tableau circulaire de taille fixe ;
• Un objet de la classe **CircularQueue** possède une méthode **iterator** qui retourne un objet **CircularQueueIterator** qui réalise l'interface **Iterator** ;
• Un appel à la méthode **hasNext** d'un itérateur retourne **true** s'il y a encore des éléments à retourner dans cette itération, et **false** sinon.
• Un appel à la méthode **next** d'un itérateur retourne le prochain objet de l'itération. Ainsi, le premier appel retourne l'élément avant (*front*), le second appel retournera l'élément suivant, etc. Éventuellement, un appel à **next** retournera le dernier élément. Maintenant, un appel à **hasNext** retournera **false** et un appel à **next** entraînera une exception de type **NoSuchElementException**.
Complétez l'implémentation de la classe **CircularQueue**, aux places indiquées ci-contre.

Un programme **`Test3`** vous est fourni ci-joint pour tester.

```
/*Classe CircularQueue*/
public class CircularQueue<E> implements Queue<E> {
private static final int DEFAULT_CAPACITY = 100;
private int front, rear, size;
private E[] elems;

public CircularQueue(int capacity) {
elems = (E[]) new Object[capacity];
front = 0;
rear = -1;
size = 0;
}

public boolean isEmpty() {
return (size == 0);
}

public void enqueue(E value) {
rear = (rear+1) % elems.length;
elems[rear] = value;
size++;
}
```

```
public E dequeue() {
E save = elems[ front ];
elems[front] = null;
size--;
front = (front+1) % elems.length;
return save;
}

// CircularQueueIterator class
private _À COMPLÉTER__ CircularQueueIterator implements Iterator<E> {
private __ À COMPLÉTER _ current = À COMPLÉTER ___;

public E next() {
if ( À COMPLÉTER) {
throw new NoSuchElementException();
}
if (current == -1) {
                    current = front;
}
else {
 current = (current + 1) % elems.length;
}

return _ À COMPLÉTER __;
}

public boolean hasNext() {
return size != 0 && (current == -1 || current != rear);
}
} // End of CircularQueueIterator

//iterator method
public _ À COMPLÉTER _ iterator() {
return _ À COMPLÉTER __;
}

} // End of CircularQueue
```

- Queue.java
- Iterator.java
- CircularQueue.java
- Test3.java

Partie 4 : Classe `LinkedList`

Implémentez la méthode d'instance **remove(int fromIndex, int toIndex)** pour la classe **`LinkedList`** ci-contre. Cette méthode retire de cette liste tous les éléments situés dans l'intervalle de positions spécifiées et retourne ces éléments dans une nouvelle liste, dans l'ordre original. Voici les caractéristiques de la classe **`LinkedList`**.

• L'instance débute toujours par un noeud factice. Ce dernier marque le début de la liste. On n'y sauvegarde jamais une valeur. Une liste vide ne comprend que le noeud factice ;
• Les noeuds de la liste sont doublement chaînés ;
• La liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud désigne le noeud factice, la référence **previous** du noeud factice désigne le dernier noeud de la liste. Si la liste est vide, le nœud factice sera à la fois le premier et le dernier noeud de la liste, ses références **previous** et **next** pointeront vers ce noeud unique ;
• Puisqu'on accède facilement au dernier élément de la liste, en effet, c'est le noeud qui précède le nœud factice, l'en-tête de la liste ne possède pas de pointeur arrière.

Exemple : si **list1** désigne une liste contenant les éléments suivants **[a,b,c,d,e,f]**, suite à l'appel de méthode **list2 = list1.remove(2,3)**, la liste désignée par **list1** contient maintenant les éléments suivants **[a,b,e,f]**, et **list2** désigne une liste contenant ces éléments **[c,d]**.

Un programme **`Test4`** vous est fourni ci-joint pour tester.

```
/* Classe LinkedList*/
public class LinkedList<E> {
// static nested class
private static class Node< E > { // noeuds de la liste
private E value;
private Node< E > previous;
private Node< E > next;

private Node(E value, Node< E > previous, Node< E > next) {
this.value = value;
this.previous = previous;
this.next = next;
}
}

//Instance variables
private Node<E> head;
private int size;

// Constructor
public LinkedList() {
head = new Node<E>(null, null, null);
head.next = head.previous = head;
size = 0;
}
```

```
// Instance methods
   public int size() {
      return size;
   }

   /**
    * Returns true if this list is empty
    */
   public boolean isEmpty() {
      return size == 0;
   }

   /**
    * Adds an element at the end of the list.
    */
   public boolean addLast( E elem ) {
      if ( elem == null ) {
         throw new IllegalArgumentException( "null" );
      }

      Node<E> before = head.previous;
      Node<E> after = head;

      before.next = new Node<E>( elem, before, after );
      after.previous = before.next;

      size++;
      return true;
   } // End of addLast

   /**
    * Adds an element at the start of the list.
    */
   public boolean addFirst( E elem ) {
      if ( elem == null ) {
         throw new IllegalArgumentException( "null" );
      }

      Node<E> before = head;
      Node<E> after = head.next;

      before.next = new Node<E>( elem, before, after );
      after.previous = before.next;
      size++;
      return true;
   }// End of addFirst
```

```
    /**
     * Returns the element at the specified position; the first element has the index 0.
     */
    public E get( int index ) {
        if ( index < 0 || index > (size-1) ) {
            throw new IndexOutOfBoundsException( Integer.toString( index ) );
        }

        Node<E> p = head.next;

        for ( int i=0; i<index; i++ ) {
            p = p.next;
        }

        return p.value;
    } // End of get


    /**
     * remove method
     */
    public LinkedList<E> remove(int fromIndex, int toIndex) {
        // À COMPLÉTER...
} // End of remove


    //  toString method
    public String toString() {
        StringBuffer input = new StringBuffer( "[" );
        Node<E> p = head.next;
        while ( p != head ) {
            if ( p != head.next ) {
             input.append( "," );
            }
            input.append( p.value );
            p = p.next;
        }
        input.append( "]" );
        return input.toString();
    }// End of toString

} // End of LinkedList
```

- *LinkedList.java*
- *Test4.java*