ESERCITAZIONE: ITERATORE PER LE LISTE ORDINATE DI INTERI

Nuova specifica di `OrderedIntList`

```
public class OrderedIntList {
 public OrderedIntList ()
 public void addEl (int el) throws
    DuplicateException
 public void remEl (int el) throws
    NotFoundException
 public boolean isIn (int el)
 public boolean isEmpty ()
 public int least () throws EmptyException
 public Iterator smallToBig ()
  // EFFECTS: ritorna un generatore che produrrà gli
  // elementi di this (come Integers), in ordine
  // crescente
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
 public boolean repOk ()
 public String toString ()}
```

- senza l'iteratore, non ci sarebbe nessuna operazione per accedere gli elementi

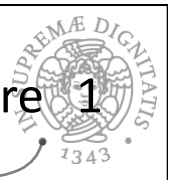
Il generatore

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public Iterator smallToBig ()
  // EFFECTS: ritorna un generatore che produrrà gli
  // elementi di this (come Integers), in ordine
  // crescente
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
 {return new OLGen(this, count());}

 private int count () {
  if (vuota) return 0;
  return 1 + prima.count() + dopo.count(); }
```

- al generatore viene passato il numero di elementi della lista
  - calcolato dal metodo privato `count`

Implementazione del generatore 1



```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
 // la funzione di astrazione (ricorsiva!)
 // α(c) = se c.cnt = 0 allora [], altrimenti
 // se il numero di elementi di α(c.figlio) è c.cnt
 // allora α(c.figlio),
 // altrimenti α(c.figlio) + [Integer(c.me.val)] +
 //    α(OLGen(c.dopo))
```

Implementazione del generatore 2

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
  // l'invariante di rappresentazione
  // I(c) = c.cnt = 0 oppure
  // (c.cnt > 0 e c.me != null e c.figlio != null e
  // (c.cnt = c.figlio.cnt + 1 oppure
  // c.cnt = c.figlio.cnt + c.me.dopo.count() + 1))
```


Implementazione del generatore 3

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo

  OLGen (OrderedIntList o, int n) {
    // REQUIRES: o != null
   cnt = n;
   if (cnt > 0) { me = o;
     figlio = new OLGen(o.prima, o.prima.count()); }
```

- anche il costruttore è ricorsivo

Implementazione del generatore 3.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
 // I(c) = c.cnt = 0 oppure
 // (c.cnt > 0 e c.me != null e c.figlio != null e
 // (c.cnt = c.figlio.cnt + 1 oppure
 // c.cnt = c.figlio.cnt + c.me.dopo.count() + 1))
  OLGen (OrderedIntList o, int n) {
    // REQUIRES: o != null
   cnt = n;
   if (cnt > 0) { me = o;
     figlio = new OLGen(o.prima, o.prima.count()); }
```

- *cnt = 0 oppure*
- *cnt > 0 e me != null (clausola REQUIRES) e*
- *figlio != null (chiamata ricorsiva) e*
- *disgiunzione sui count (proprietà di count)*

Implementazione del generatore 4

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo

  public boolean hasNext () {
   return cnt > 0; }
```

- si noti l'uso di `cnt` per rendere efficiente anche questo metodo

Implementazione del generatore 5

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
  public Object next () throws NoSuchElementException {
   if (cnt == 0) throw new
     NoSuchElementException("OrderedIntList.smallToBig") ;
   cnt --;
   try { return figlio.next(); }
   catch (NoSuchElementException e) {}
   // se arriva qui ha finito tutti quelli prima
   figlio = new OLGen(me.dopo, cnt);
   return new Integer(me.val); }
```