Un esempio:
le liste ordinate di interi

Un nuovo esempio completo: le liste ordinate

- ✓ `OrderedIntList`
- ✓ lista ordinata di interi
  - modificabile

Specifica di `OrderedIntList` 1

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 // costruttore
 public OrderedIntList ()
   // EFFECTS: inizializza this alla lista vuota
 // metodi
 public void addEl (int el) throws
    DuplicateException
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
 public void remEl (int el) throws
    NotFoundException
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
```

Specifica di `OrderedIntList` 2

```
 public boolean isIn (int el)
  // EFFECTS: se el appartiene a this ritorna
  // true, altrimenti false
 public boolean isEmpty ()
  // EFFECTS: se this è vuoto ritorna true, altrimenti
  // false
 public int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 public boolean repOk ()
 public String toString ()
}
```

Specifica di `OrderedIntList` 3

```
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 boolean repOk ()
 public String toString ()}
```

- ✓ notare che non esiste nessuna operazione per accedere gli elementi
  - si risolverà in seguito con l'introduzione di un iteratore
- ✓ `DuplicateException` e `NotFoundException` checked

Implementazione di `OrderedIntList` 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;
```

- ✓ la rep contiene
  - una variabile `boolean` che ci dice se la lista è vuota
  - la variabile intera che contiene l'eventuale valore dell'elemento
  - due (puntatori a) `OrderedIntList`s che contengono la lista di quelli minori e quelli maggiori, rispettivamente
- ✓ implementazione ricorsiva

Implementazione di `OrderedIntList` 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;
 // la funzione di astrazione (ricorsiva!)
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 // l'invariante di rappresentazione (ricorsivo!)
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
```

- ✓ l'invariante utilizza metodi esistenti
  - `isEmpty` e `least`
- ✓ + `max`

Implementazione di `OrderedIntList` 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;
 // costruttore
 public OrderedIntList ()
   // EFFECTS: inizializza this alla lista vuota
   { vuota = true; }
```

- ✓ il costruttore inizializza solo la variabile `vuota`

Implementazione di `OrderedIntList` 3.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public OrderedIntList ()
   // EFFECTS: inizializza this alla lista vuota
   { vuota = true; }
```

- ✓ l'implementazione del costruttore
  - soddisfa l'invariante (`c.vuota = true`)
  - verifica la propria specifica (`α(c) = []`)

Implementazione di `OrderedIntList` 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;
 public void addEl (int el) throws DuplicateException
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   if (el == val) throw new
       DuplicateException("OrderedIntList.addEl");
   if (el < val) prima.addEl(el);
      else dopo.addEl(el); }
```

- ✓ propaga automaticamente l'eventuale eccezione sollevata dalle chiamate ricorsive

Implementazione di `OrderedIntList` 4.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public void addEl (int el) throws DuplicateException
   {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   ...
```

`prima != null e dopo != null` (calcolati dal costruttore)

`I(prima) e I(dopo)` (calcolati dal costruttore)

le implicazioni sono vere perché la premessa è falsa

Implementazione di `OrderedIntList` 4.2

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public void addEl (int el) throws DuplicateException
    ...
    if (el < val) prima.addEl(el);
       else dopo.addEl(el); }
```

- this non è vuoto

`prima != null e dopo != null`

`I(prima) e I(dopo)` (calcolati da una chiamata ricorsiva)

- ramo then: il nuovo massimo di `prima` è (induttivamente) minore di `val`
- ramo else: il nuovo minimo di `dopo` è (induttivamente) maggiore di `val`

Implementazione di `OrderedIntList` 4.3

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   ...
```

- ✓ $\alpha(c_{pre}) = []$
- ✓ $\alpha(c.prima) = []$
- ✓ $\alpha(c.dopo) = []$
- ✓ $[c.val] = [el]$
- ✓ $\alpha(c) = [el]$

Implementazione di `OrderedIntList` 4.4

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  ...
   if (el == val) throw new
       DuplicateException("OrderedIntList.addEl");
```

- se ci sono elementi duplicati solleva l'eccezione, eventualmente propagando eccezioni sollevate dalle chiamate ricorsive (vedi dopo)

Implementazione di `OrderedIntList` 4.5

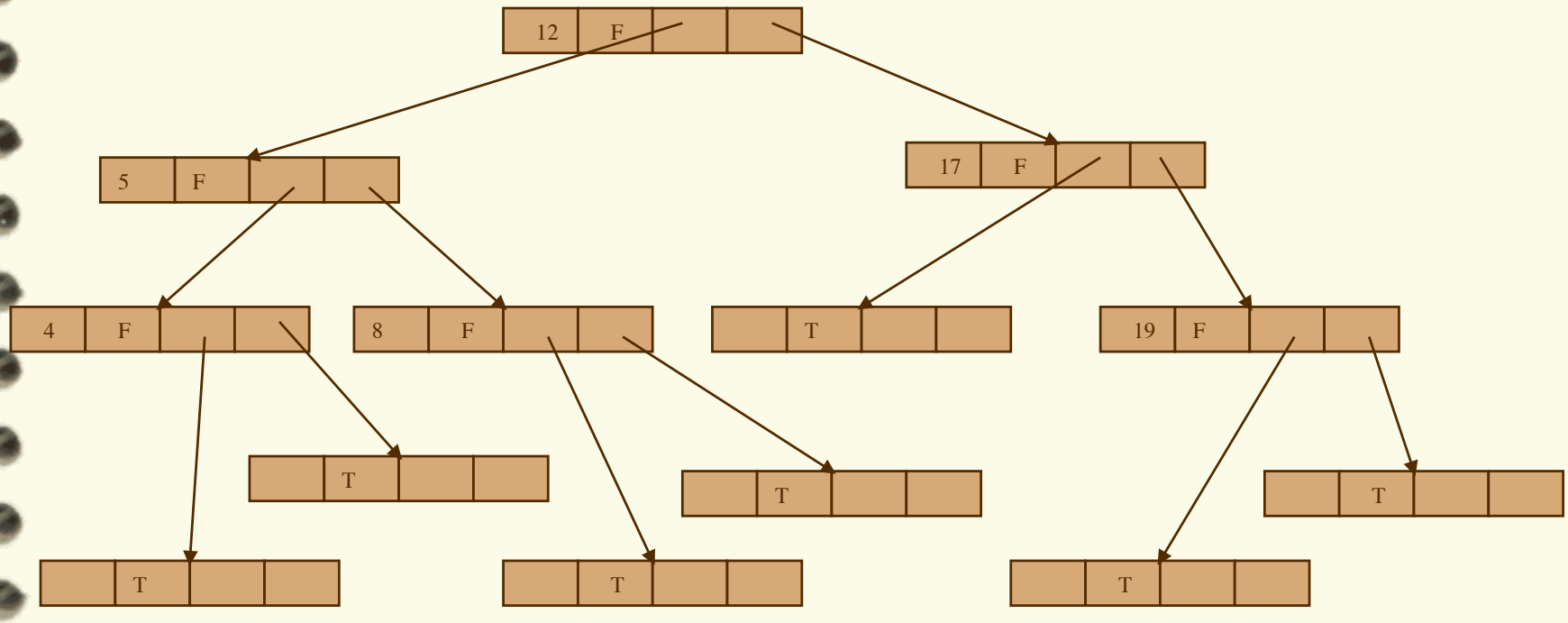
```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  ...
   if (el < val) prima.addEl(el);
      else dopo.addEl(el); }
```

- ✓ $\alpha(c_{pre}) = \alpha(c.prima_{pre}) + [c.val] + \alpha(c.dopo_{pre})$
- ✓ se `el < val` la chiamata ricorsiva solleva l'eccezione oppure produce
- ✓ $\alpha(c.prima) = aggiunge\ el\ a\ prima_{pre}$
- ✓ $\alpha(c.dopo) = \alpha(c.dopo_{pre})$
- ✓ $\alpha(c) = aggiunge\ el\ a\ c_{pre}$

Come è fatta una `OrderedIntList`

- vediamo la lista prodotta dalla sequenza di comandi

```
OrderedIntList ls = new OrderedIntList();
ls.addEl(12); ls.addEl(5); ls.addEl(17);
ls.addEl(4);ls.addEl(8); ls.addEl(19);
```



Implementazione di `OrderedIntList` 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;
 public void remEl (int el) throws NotFoundException
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
  {if (vuota) throw new
       NotFoundException("OrderedIntList.remEl");
   if (el == val)
    try { val = dopo.least(); dopo.remEl(val); }
    catch (EmptyException e)
     { vuota = prima.vuota; val = prima.val;
      dopo = prima.dopo; prima = prima.prima; return;}
   else if (el < val) prima.remEl(el);
      else dopo.remEl(el); }
```

Implementazione di `OrderedIntList` 6

```
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;
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public void remEl (int el) throws NotFoundException
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
  {if (vuota) throw new
       NotFoundException("OrderedIntList.remEl");
   if (el == val)
    try { val = dopo.least(); dopo.remEl(val); }
    catch (EmptyException e)
     { vuota = prima.vuota; val = prima.val;
      dopo = prima.dopo; prima = prima.prima; return;}
   else if (el < val) prima.remEl(el);
      else dopo.remEl(el); }
```

- ✓ per la prossima esercitazione portare la dimostrazione che remEl preserva l'invariante

Implementazione di `OrderedIntList` 7

```
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;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)

 public void remEl (int el) throws NotFoundException
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
  {if (vuota) throw new
       NotFoundException("OrderedIntList.remEl");
   if (el == val)
    try { val = dopo.least(); dopo.remEl(val); }
    catch (EmptyException e)
     { vuota = prima.vuota; val = prima.val;
      dopo = prima.dopo; prima = prima.prima; return;}
   else if (el < val) prima.remEl(el);
      else dopo.remEl(el); }
```

- ✓ per la prossima esercitazione portare la dimostrazione che l'implementazione di remEl soddisfa la specifica

Implementazione di `OrderedIntList` 8

```
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 boolean isIn (int el)
   // EFFECTS: se el appartiene a this ritorna
  // true, altrimenti false
  {if (vuota) return false;
   if (el == val) return true;
   if (el < val) return prima.isIn(el); else return
       dopo.isIn(el);  }
 public boolean isEmpty ()
  // EFFECTS: se this è vuoto ritorna true, altrimenti false
  {return vuota; }
```

Implementazione di `OrderedIntList` 9

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public boolean isIn (int el)
   // EFFECTS: se el appartiene a this ritorna
   // true, altrimenti false
   {if (vuota) return false;
    if (el == val) return true;
    if (el < val) return prima.isIn(el); else return
        dopo.isIn(el);  }
 public boolean isEmpty ()
   // EFFECTS: se this è vuoto ritorna true, altrimenti false
   {return vuota; }
```

- ✓ dimostrazioni di correttezza da preparare per la prossima esercitazione

Implementazione di `OrderedIntList` 10

```
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 int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 {if (vuota) throw new
    EmptyException("OrderedIntList.least");
  try { return prima.least(); }
  catch (EmptyException e) {return val;} }
```

Implementazione di `OrderedIntList` II

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 {if (vuota) throw new
    EmptyException("OrderedIntList.least");
  try { return prima.least(); }
  catch (EmptyException e) {return val;} }
```

- ✓ dimostrazione di correttezza da preparare per la prossima esercitazione