Liste doppie

Liste:

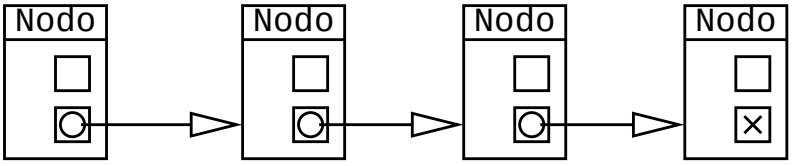
semplici
: ogni elemento contiene un riferimento al successivo

doppie
: ogni elemento contiene un riferimento al successivo e al precedente

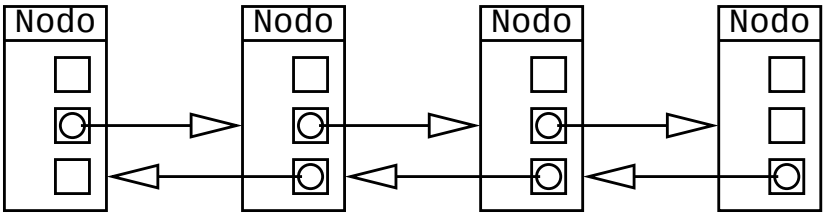
---

Rappresentazione grafica

Lista semplice:



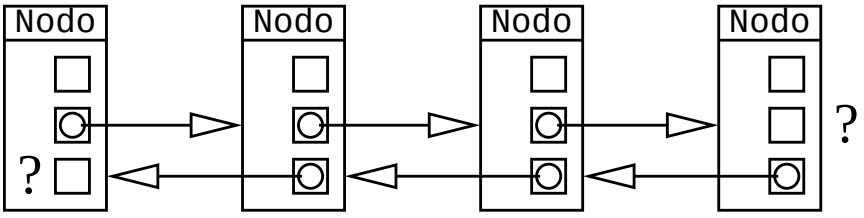
Lista doppia:



Gli elementi della lista (interi se si parla di una lista di interi, `Object` se si realizza un lista di oggetti) vanno nella prima variabile d'instanza di ognuno di questi nodi

```
class Nodo {
  Object info;
  Nodo next;
  Nodo prev;
}
```

Come si conclude la lista

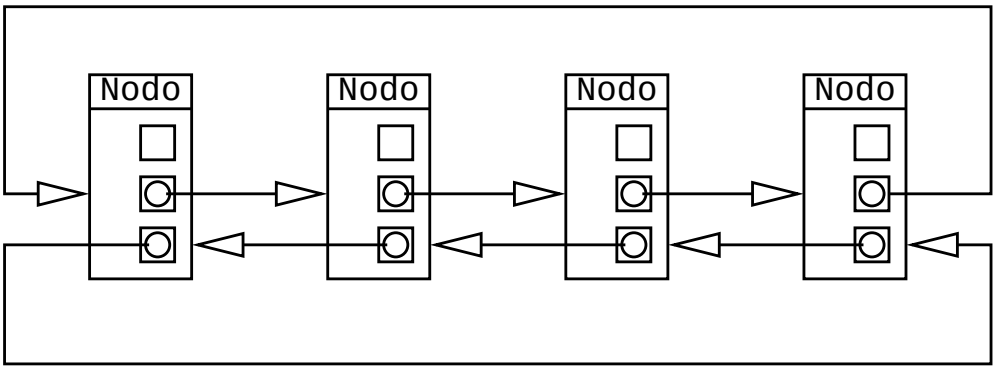


Ho due scelte:

1. l'ultimo riferimento da ogni lato è `null`
2. l'ultimo riferimento punta all'oggetto dalla parte opposta

Lista doppia circolare

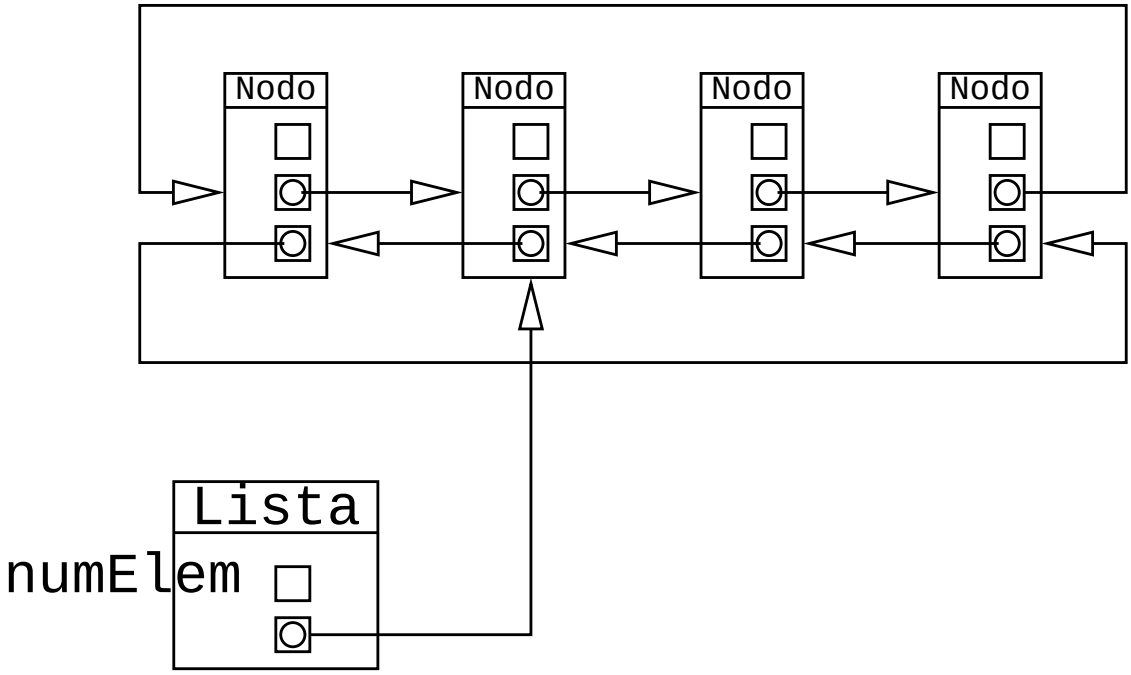
Usiamo la soluzione circolare:



Dato il riferimento a un nodo qualsiasi riesco a scandire la lista

Per comodità, aggiungo anche la dimensione della lista

Oggetto `ListaDoppia`



L'oggetto `Lista` contiene:

un riferimento a un oggetto della catena
    questo permette la scansione della lista
la dimensione della lista
    non è strettamente necessario, però è comodo

L’inizio della catena

Il nodo dova va il puntatore iniziale può essere:

1. il nodo che contiene il primo oggetto della lista
2. un nodo che non contiene oggetti della lista, ma indica solo il punto dove la lista inizia (e finisce)

Nel secondo modo, il nodo di inizio è solo un *segnaposto* (non corrisponde a un elemento della lista)

È come una posizione ‘‘vuota’’ nella lista, e indica un elemento che non c’è

Se ho una variabile di scansione `Nodo n`, posso capire quando sono all’inizio a alla fine della lista `l` facilmente:

primo elemento della lista
    n.prev==l.sentinella
ultimo elemento della lista
    n.next==l.sentinella

Si può pensare al nodo segnaposto come a un elemento che non c’è perchè si trova:

- prima del primo elemento
- dopo l’ultimo elemento

Il numero degli oggetti della lista è pari al numero degli oggetti `Nodo` *meno uno*

Classi

Gli oggetti `Nodo` hanno tre campi: l’oggetto della lista e due riferimenti (avanti e indietro):

```
class Nodo {
  Object info;
  Nodo next;
  Nodo prev;
}
```

Gli oggetti `ListaDoppia` hanno due campi: il numero di elementi e un riferimento a un qualsiasi oggetto dell’anello:

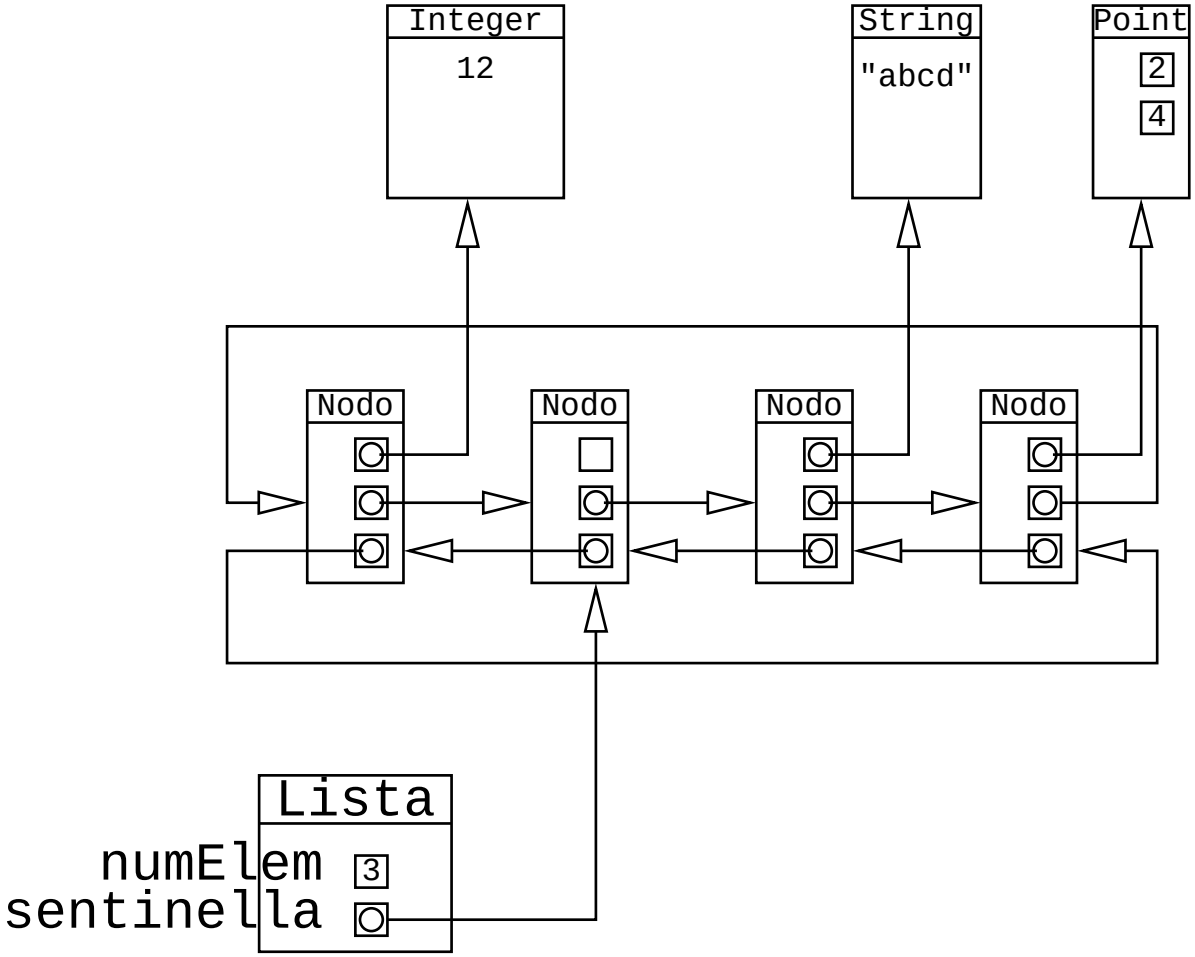
```
class ListaDoppia {
  int numElem;
  Nodo sentinella;
}
```

Nodi e oggetti

Ogni nodo ha un campo `info`

Questo campo `info` contiene un riferimeno a un oggetto della lista

La lista rappresenta la sequenza di oggetti dei campi `info`, non la sequenza di nodi!

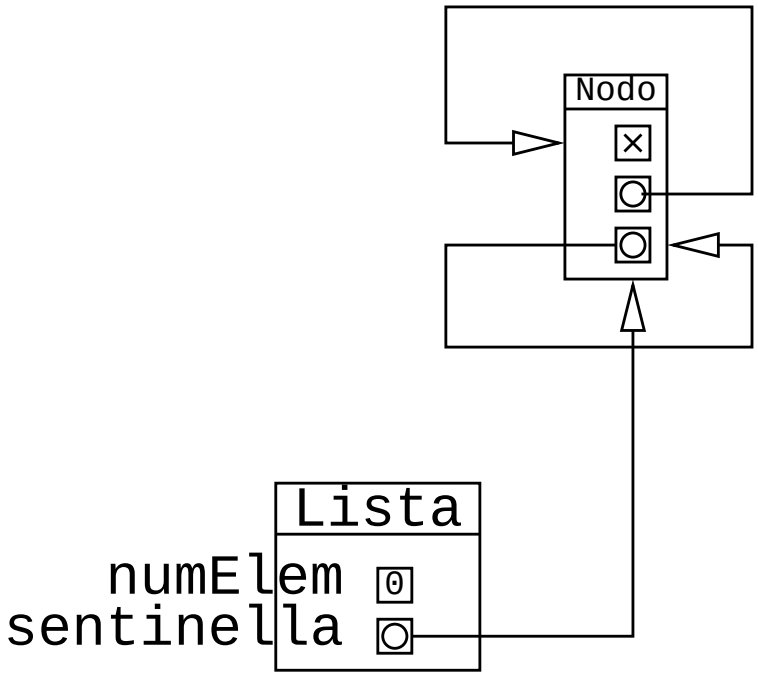


Questa lista rappresenta la sequenza dei tre oggetti `String`, `Point` e `Integer`, non la sequenza degli oggetti `Nodo`

Nelle figure seguenti e successive, si omettono gli oggetti attaccati al campo `info` per semplicità

La lista vuota

Nella lista vuota c’è solo l’elemento segnaposto



Scansione della lista

Si parte dall'oggetto `sentinella`

Si può andare avanti o indietro

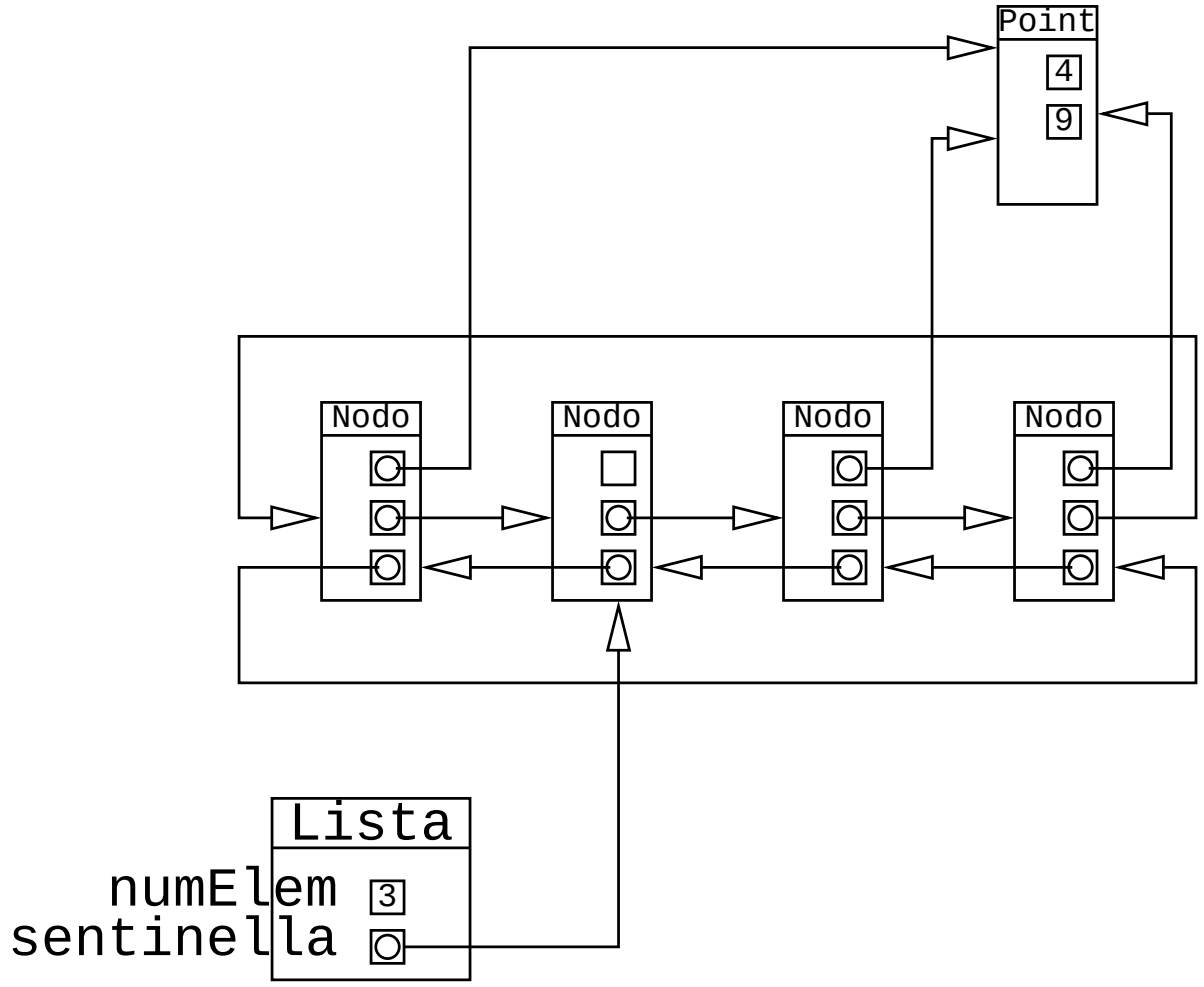
Ci si ferma quando si torna di nuovo a `sentinella`

Cosa succede se la lista contiene due oggetti uguali?

Gli oggetti della lista stanno nel campo `info` dei nodi

Anche se *tutti* gli oggetti della lista sono uguali, gli oggetti `Nodo` sono comunque tutti diversi.

Esempio di lista che contiene tre volte lo stesso `Point`:



Scansione della lista

Questo metodo stampa tutti gli oggetti:

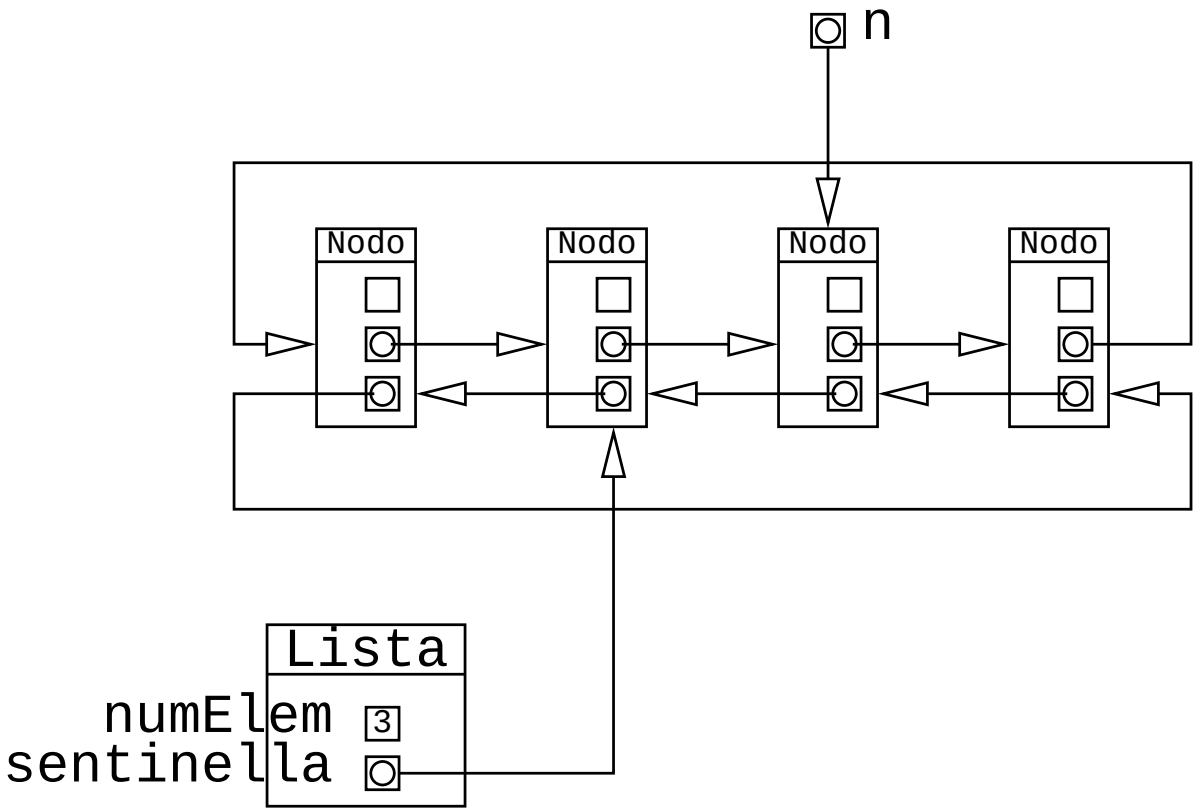
```
public static void stampa(ListaDoppia l) {
  Nodo n;

  if(l==null)
    return;

  n=l.sentinella.next;
  while(n!=l.sentinella) {
    System.out.println(n.info);
    n=n.next;
  }
}
```

Principî:

1. il nodo `sentinella` non rappresenta un oggetto della lista
2. il primo nodo della lista è quello *successivo* alla sentinella
3. si va avanti come al solito (`n=n.next`)
4. quando si arriva di nuovo alla sentinella, vuol dire che la lista è finita

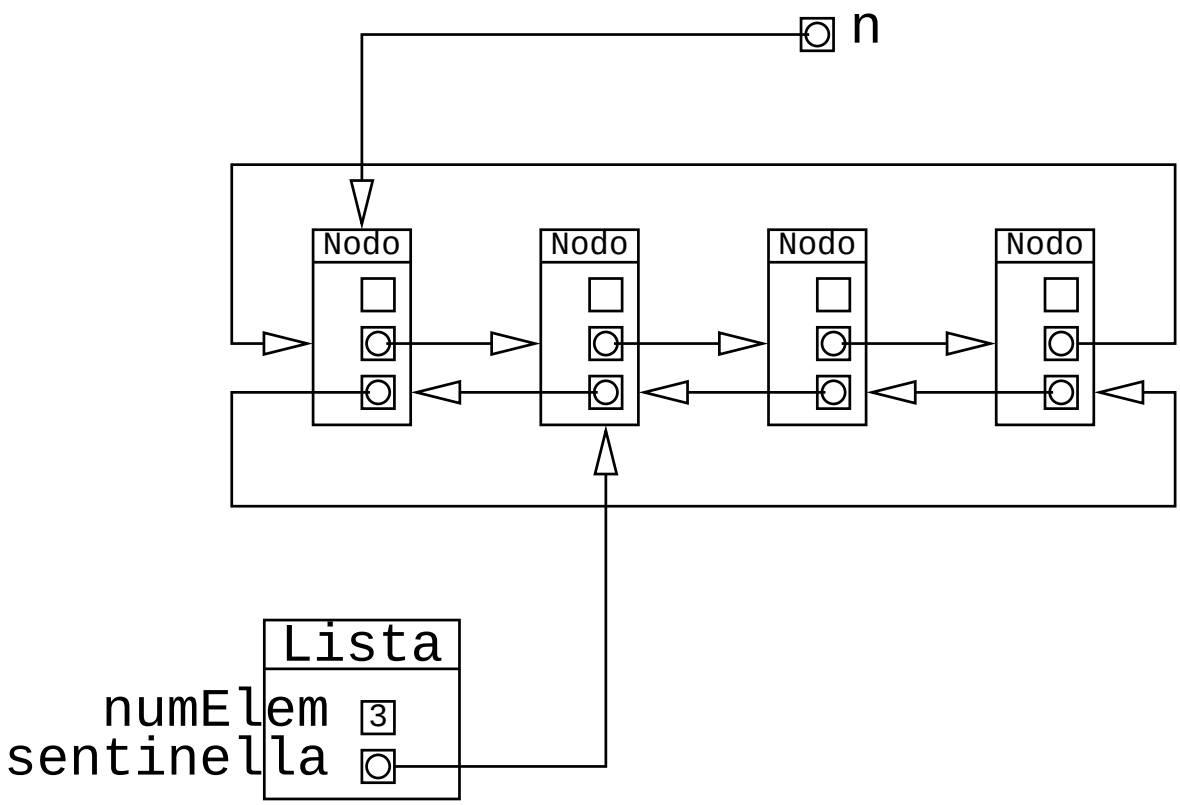
Scansione: grafica

All’inizio della scansione: `n=l.sentinella.next`

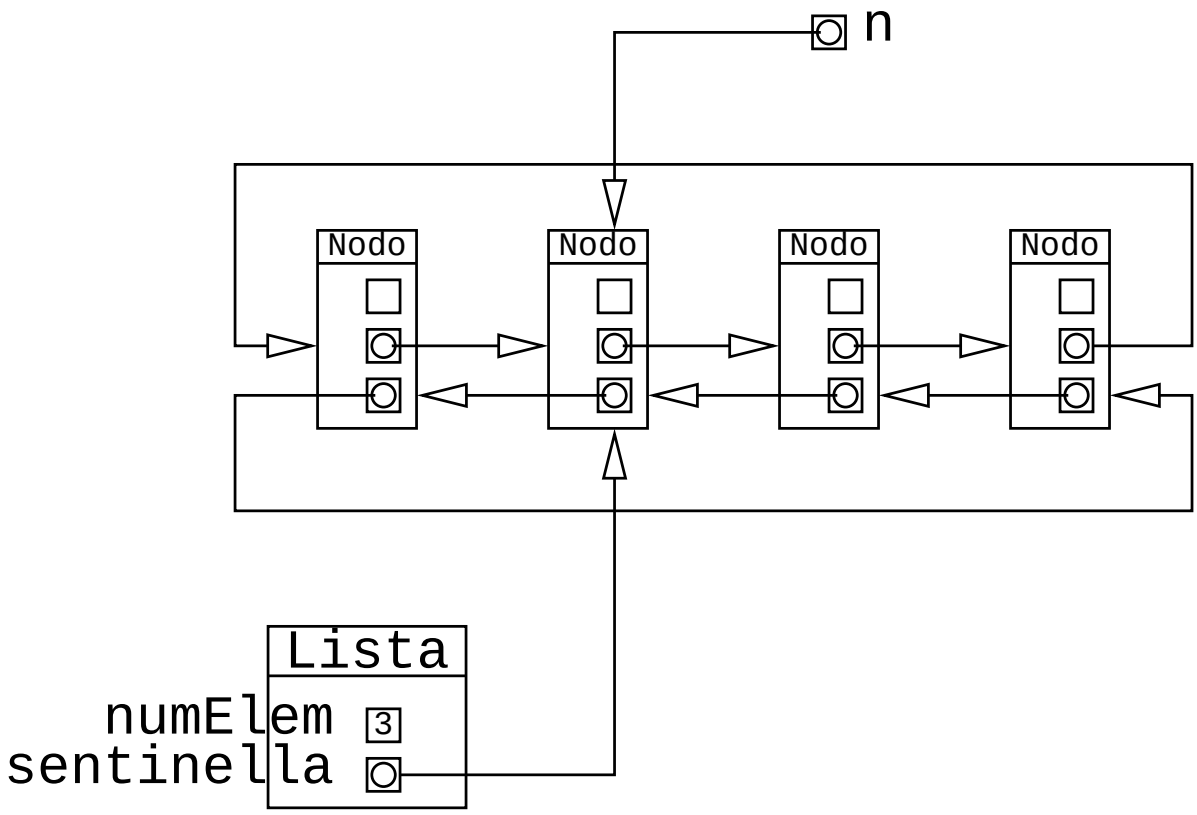


Si va ora avanti (nel disegno, verso destra)

Alla fine:



Al passo successivo, n punta alla sentinella:



Dato che `n==l.sentinella`, la scansione termina (senza stampare l'elemento `n.info`)

Alternativa con `numElem`

Parto da un elemento qualsiasi, e vado avanti per `numElem` volte:

```
public static void stampa(ListaDoppia l) {
  Nodo n;
  int i;

  if(l==null)
    return;

  n=l.sentinella.next;
  for(i=0; i<l.numElem; i++) {
    System.out.println(n.info);
    n=n.next;
  }
}
```

Scansione al contrario

Con le liste doppie, si può fare anche la scansione della lista al contrario:

```
public static void stampa(ListaDoppia l) {
  Nodo n;

  if(l==null)
    return;

  n=l.sentinella.prev;
  for(i=0; i<l.numElem; i++) {
    System.out.println(n.info);
    n=n.prev;
  }
}
```

Basta usare `prev` al posto di `next`

Implementazione di `List`

Vediamo una implementazione dell'interfaccia `List` usando una lista doppia

Dobbiamo implementare tutti metodi dell'interfaccia

Inoltre, va implementata anche una classe che implementa `ListIterator`

Ne vediamo solo alcuni metodi

La classe `Nodo`

Oltre alle componenti, mettiamo anche un costruttore:

```
class Nodo {
  Object info;
  Nodo next;
  Nodo prev;

  Nodo(Object i, Nodo n, Nodo p) {
    this.info=i;
    this.next=n;
    this.prev=p;
  }
}
```

La classe `ListListaDoppia`

Come detto prima, ho due componenti e parecchi metodi:

```
public class ListListaDoppia {
  protected int numElem;
  protected Nodo sentinella;

  // metodi
}
```

Gli iteratori

L'interfaccia `List` ha due metodi che ritornano un iteratore.

- quello ''ereditato'' da `Collection`, che ritorna un `Iterator`
- uno proprio di `List`, che ritorna un `ListIterator`

Dato che `ListIterator` è una sottointerfaccia di `Iterator`, posso implementare anche soltanto un `ListIterator` e ritornare sempre quello

```
class ListIteratorListaDoppia implements ListIterator {
  // metodi di Iterator
  // metodi di ListIterator
}
```

Package

Mettiamo tutto in un package: `listlistadoppia`

Le classi sono dichiarate:

pubbliche:
: soltanto la classe che implementa la lista `ListListaDoppia`

ristrette al package:
: tutte le altre, ossia `Nodo` e `ListIteratorListaDoppia`

Creazione della lista

```
package listlistadoppia;

import java.util.*; // contiene l'interfaccia List

public class ListListaDoppia implements List {
  protected int numElem;
  protected Nodo sentinella;

  public ListListaDoppia() {
    numElem = 0;
    sentinella = new Nodo(null,null,null);
    sentinella.next = sentinella;
    sentinella.prev = sentinella;
  }

  ...
}
```

Viene creata la catena che contiene solo l'elemento sentinella, ossia la lista vuota

---

Lunghezza lista

```
  public int size() {
    return numElem;
  }

  public boolean isEmpty() {
    return sentinella.next == sentinella;
    // oppure return sentinella.prev == sentinella;
    // oppure return numElem == 0;
  }
```

I metodi che aggiungono o tolgono elementi devono modificare `numElem`

Incapsulamento: se `numElem` fosse pubblico, chi usa questa classe potrebbe modificare questo campo (per cui il suo contenuto potrebbe non essere più il numero di elementi della lista)

Metodo `contains`

Si fa la scansione della lista

```
  public boolean contains(Object o) {
    Nodo n;

    n=sentinella.next;
    while(n!=sentinella) {
      if(n.info.equals(o))
        return true;
      n=n.next;
    }

    return false;
  }
```

- si parte dal primo elemento, che è il `next` di `sentinella`
- a ogni passo, si verifica se l'elemento corrente è uguale a quello da cercare
- se lo è, si ritorna `true`
- se si arriva alla fine del ciclo, vuol dire che il nodo non c'è, per cui si ritorna `false`

Soluzione alternativa: metodo ausiliario

Trova il riferimento al nodo che contiene un certo oggetto (se presente);

```
private Nodo trova(Object o) {
  Nodo n = sentinella.next;
  while (n!=sentinella) {
    if (n.info.equals(o))
      return n;
    n = n.next;
  }
  return sentinella;
}
```

Nota: ritorna il `Nodo` in cui è memorizzato l'oggetto da cercare

In altre parole, ritorna il `Nodo` il cui campo `info` contiene un oggetto uguale a quello passato come argomento

È un metodo privato, perchè all'esterno i nodi non si devono vedere (perchè sono parte del modo specifico in cui la classe è stata realizzata)

Metodo `contains`

Basta vedere se l'oggetto c'è:

```
public boolean contains(Object element) {
  return trova(element) != sentinella;
}
```

Aggiunta elemento in coda

Il metodo `add` aggiunge un elemento alla fine della lista

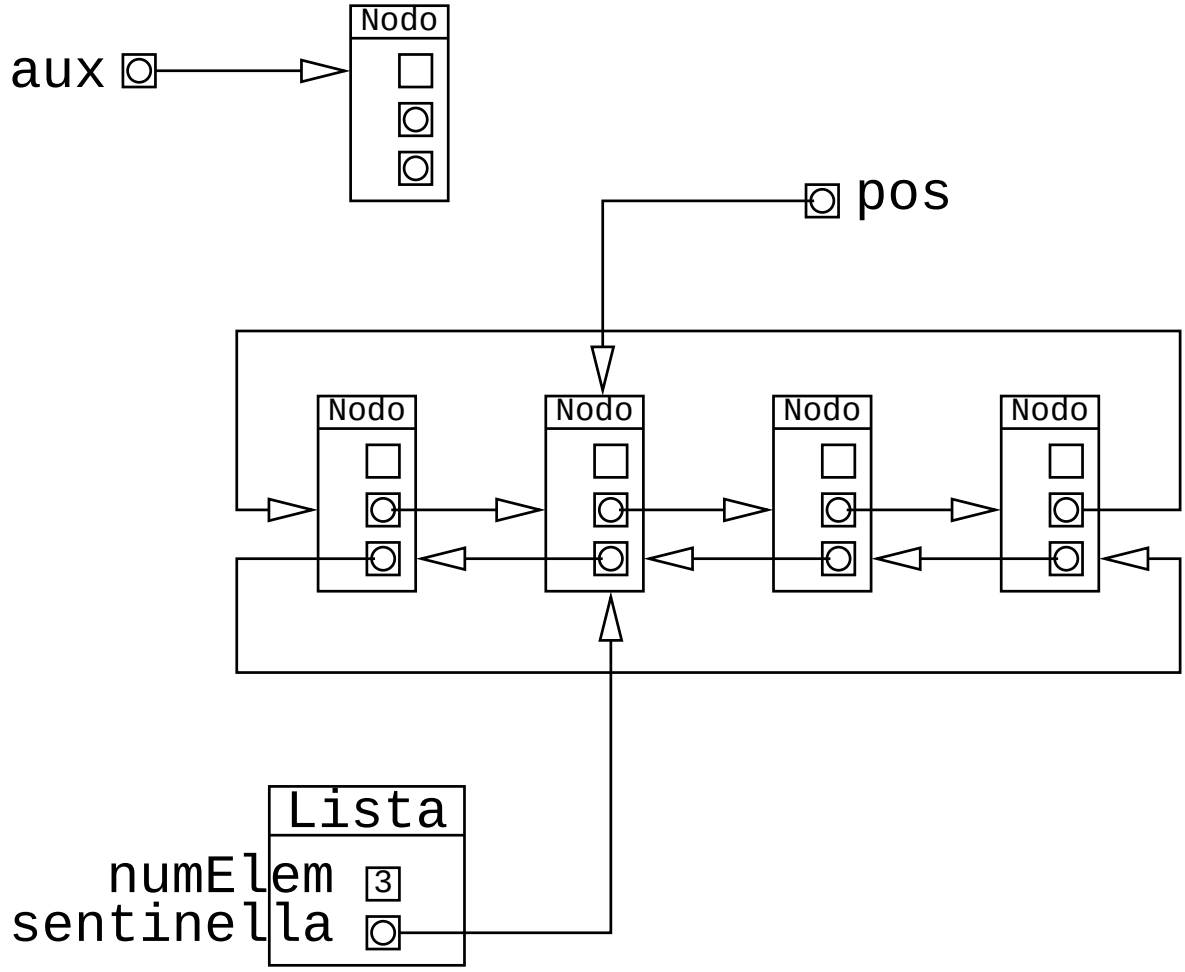
```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(element,pos,pos.prev);
  pos.prev.next = aux;
  pos.prev = aux;
  numElem++;
  return true;
}
```

Vediamo cosa succede un passo per volta

Inizio: creazione elemento

Come prima cosa, si crea l'elemento da aggiungere:

```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(...);
  ...
}
```



Cosa mettere nell'elemento

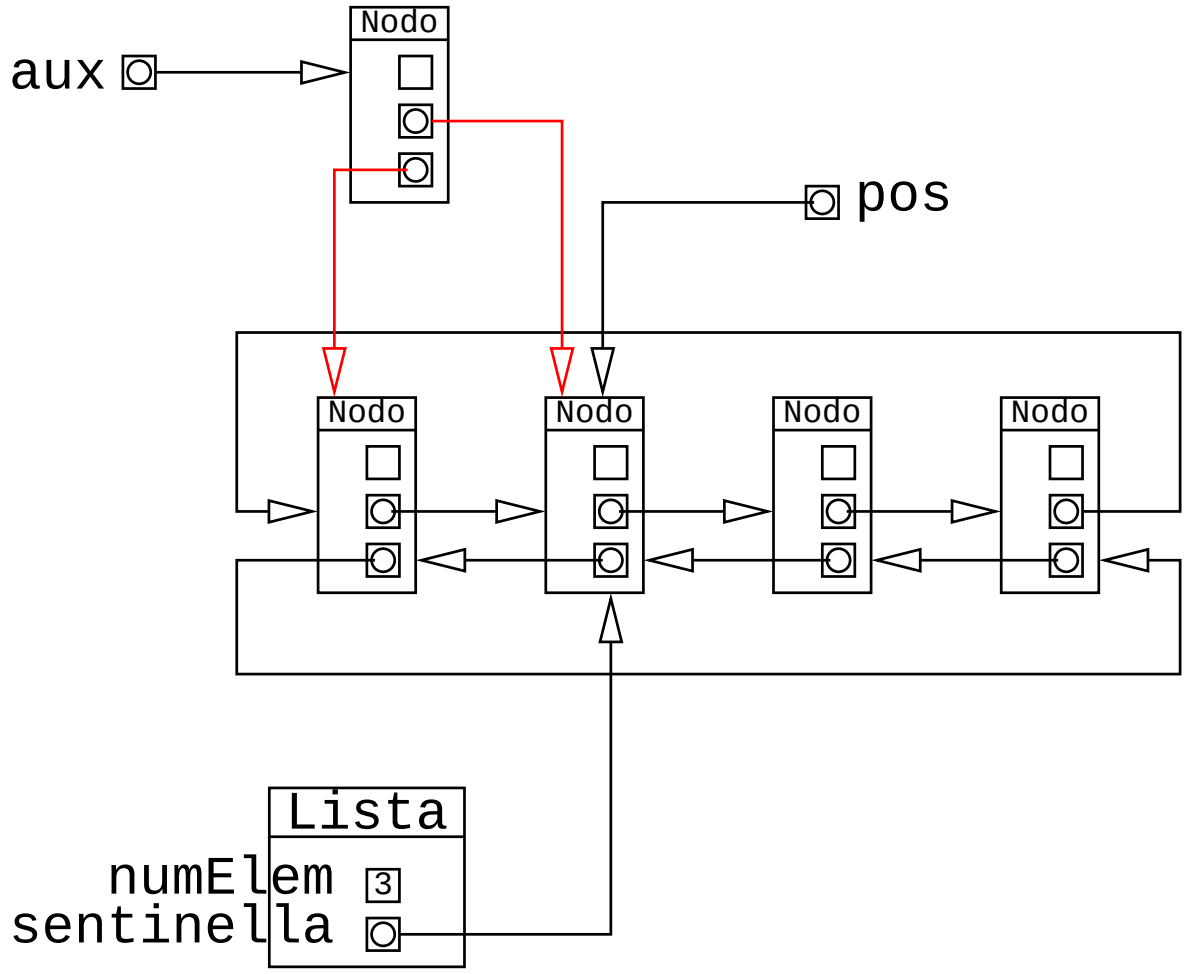
I campi del nuovo elemento devono contenere:

`info`
: l'oggetto da aggiungere

`next`
: dato che questo deve diventare l'ultimo elemento, ci va messo il riferimento alla sentinella

`prev`
: qui ci va messo il riferimento all'elemento precedente, che è quello che prima stava in ultima posizione

```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(element,pos,pos.prev);
  ...
}
```

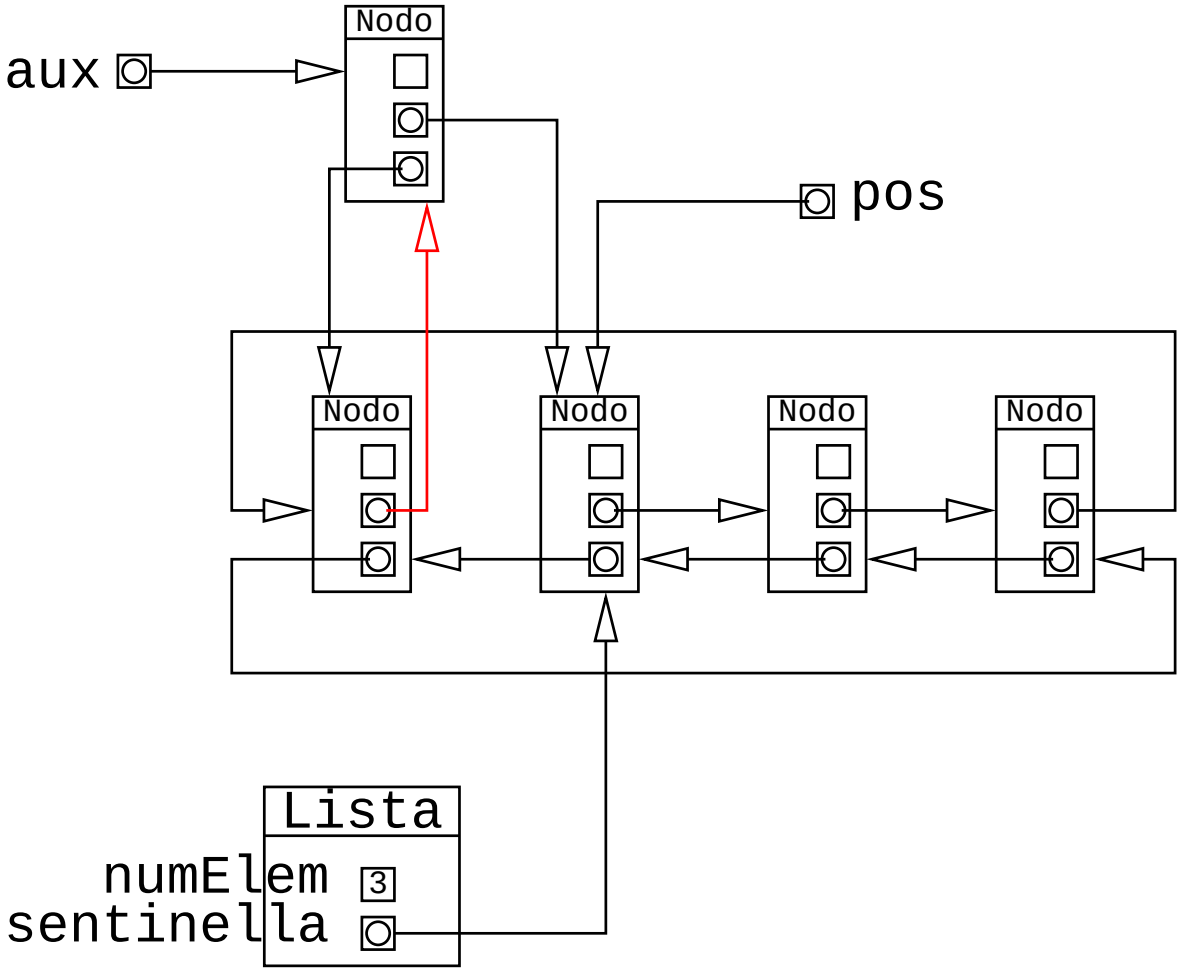


Ultimi collegamenti

Bisogna aggiornare i campi `next` e `prev` dell'ultimo elemento e della sentinella

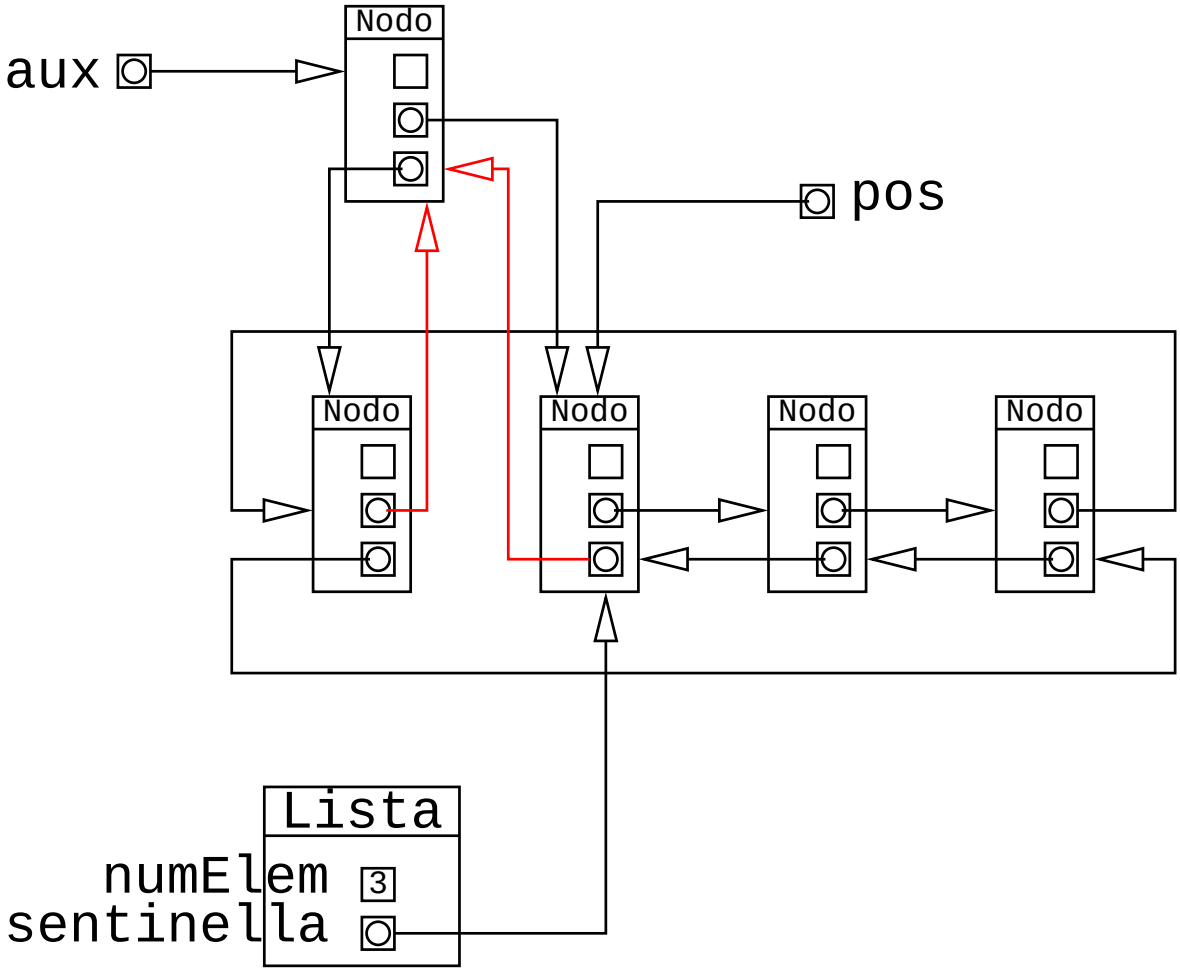
```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(element,pos,pos.prev);
  pos.prev.next = aux;
  ...
}
```

Questo genera:



Ora manca solo un collegamento:

```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(element,pos,pos.prev);
  pos.prev.next = aux;
  pos.prev = aux;
  ...
}
```



Valore di ritorno

Bisogna aggiornare `numElem`

Si ritorna `true` se l'inserimento ha avuto successo

Metodo completo:

```
public boolean add(Object element) {
  Nodo pos = sentinella;
  Nodo aux = new Nodo(element,pos,pos.prev);
  pos.prev.next = aux;
  pos.prev = aux;
  numElem++;
  return true;
}
```

Eliminare un elemento

- si trova la posizione
- si elimina l'elemento

Per il primo passo, possiamo usare il metodo ausiliario `trova`:

```
public boolean remove(Object element) {
  Nodo pos=trova(element);

  if(pos==sentinella)
    return false;

  ...
}
```

Eliminazione elemento

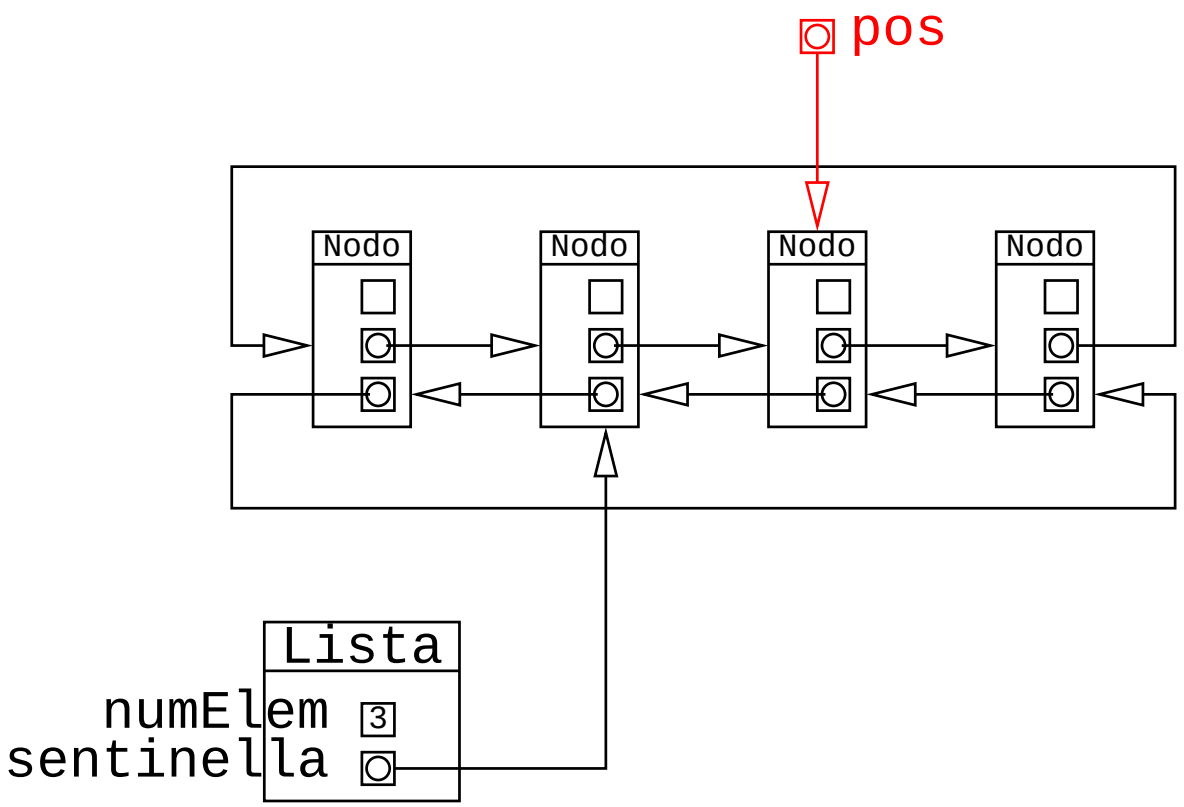
Si tratta di collegare fra di loro l'elemento precedente e successivo:

```
public boolean remove(Object element) {
  Nodo pos=trova(element);

  if(pos==sentinella)
    return false;

  pos.prev.next = pos.next;
  pos.next.prev = pos.prev;
  numElem--;
  return true;
}
```

Un passo per volta

Si trova l'elemento da eliminare

Questo viene fatto dal metodo ausiliario

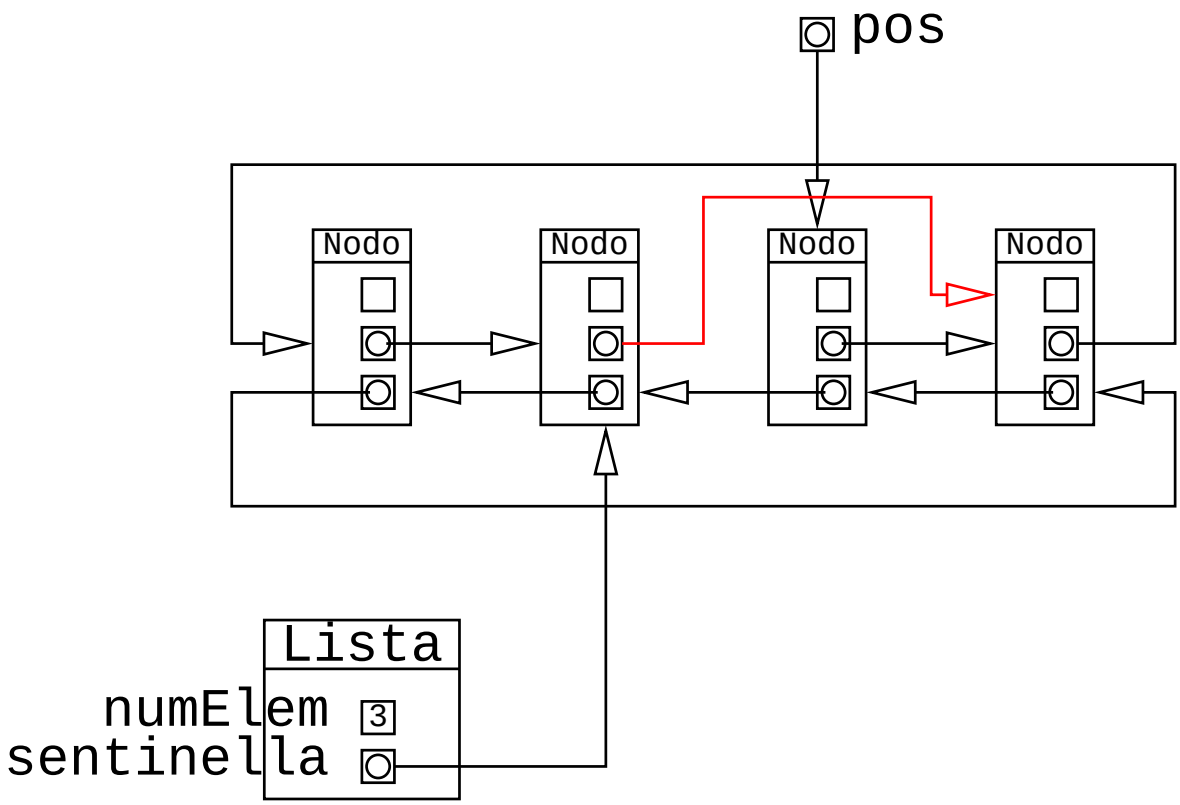


Primo collegamento

Le freccie verso l'elemento da eliminare lo devono ‘‘saltare’’

Prima la freccia orientata verso destra:

```
pos.prev.next = pos.next;
```



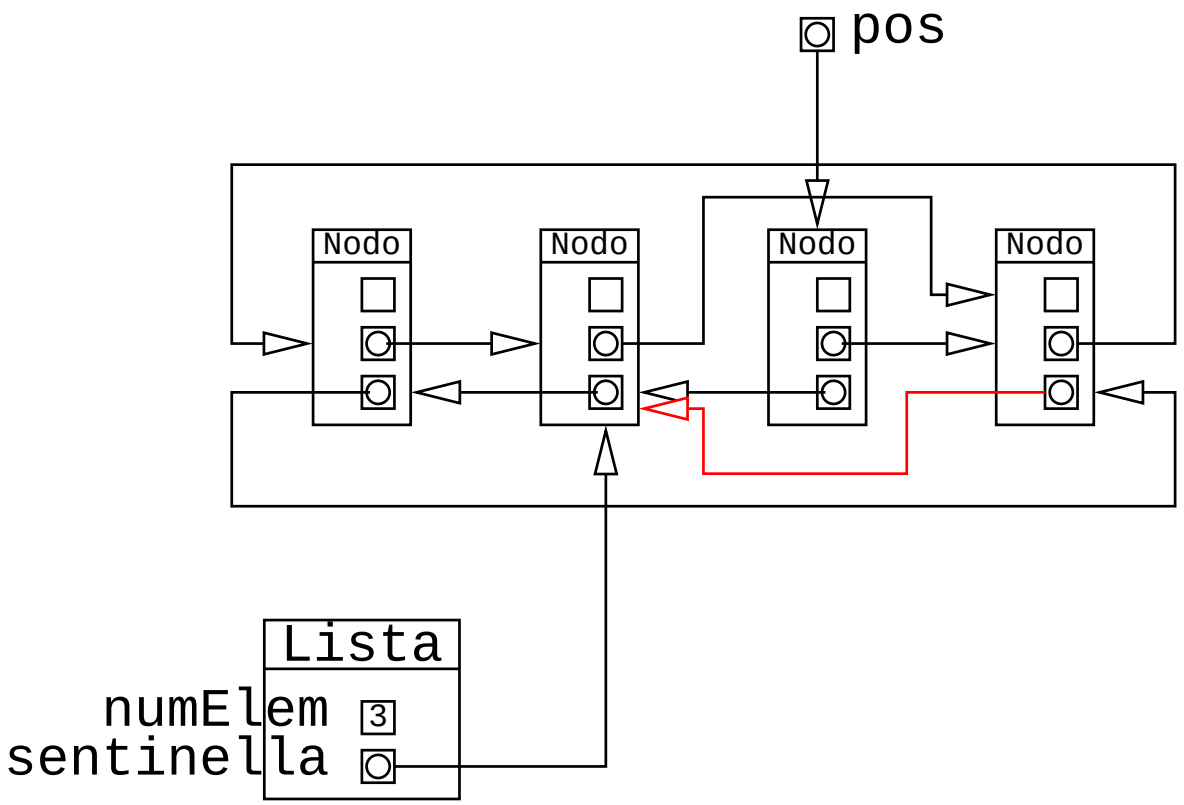
Ora si esegue:

```
pos.next.prev = pos.prev;
```

Secondo collegamento

Lo stesso per l’altra freccia:

```
pos.next.prev = pos.prev;
```

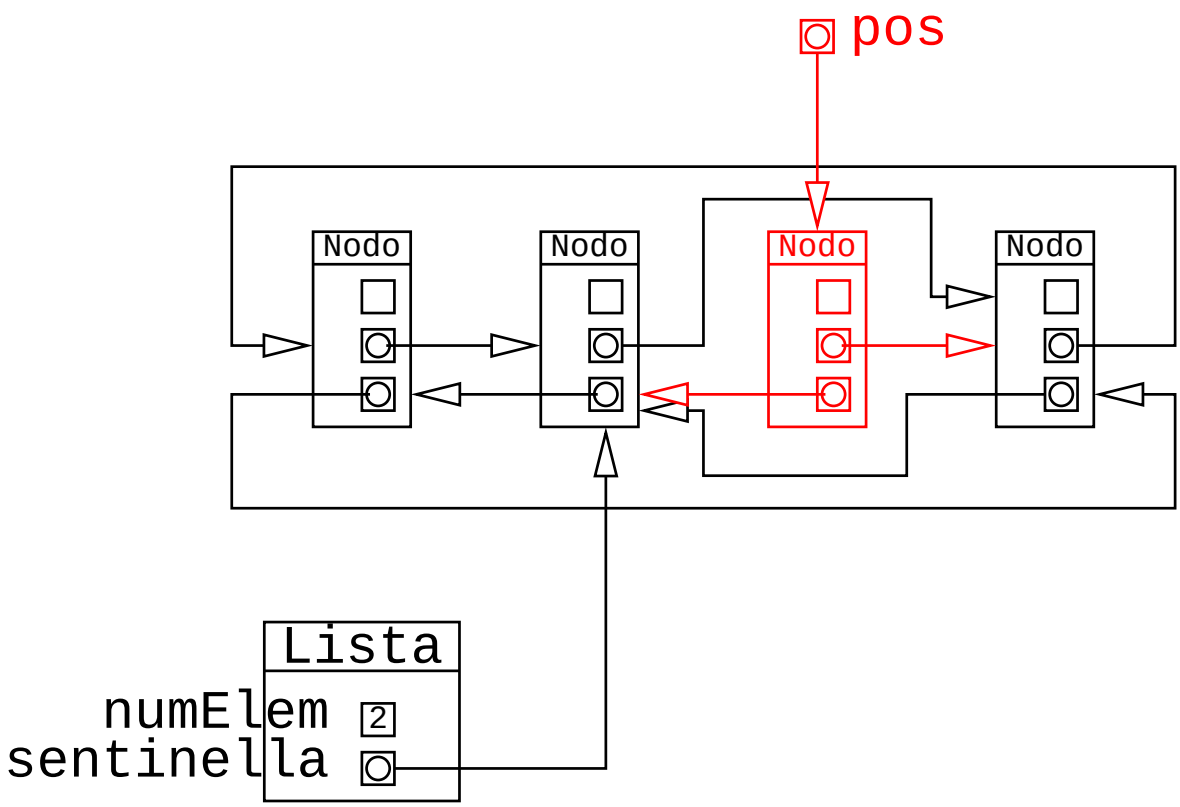


Fine eliminazione

Viene aggiornato il campo `numElem`

Quando il metodo termina, la variabile `pos` viene deallocata

Dato che il nodo da eliminare non ha più riferimenti entranti, viene cancellato



L’iteratore

Notare che non esiste nessun oggetto `Iterator` o `ListIterator`, dal momento che queste sono interfacce

Dobbiamo quindi realizzare una classe che implementa l’interfaccia `Iterator` e `ListIterator`

Lo mettiamo sempre nello stesso package:

```
package listlistadoppia;

import java.util.*; // contiene le interfacce!

class ListIteratorListaDoppia
  implements ListIterator {
  ...
}
```

Diritti di accesso dell’iteratore

La classe e il costruttore hanno diritti di package

Tutti i metodi delle interfacce `Iterator` e `ListIterator` devono essere pubblici perchè questo è il loro modificatore di accesso nelle interfacce

```
class ListIteratorListaDoppia
  implements ListIterator {

  // componenti private

  public boolean hasNext() {
    ...
  }

  public Object next() {
    ...
  }

  public void remove() {
    ...
  }

  // metodi di ListIterator
}
```

Uso dell'iteratore

Dato che `ListIteratorListaDoppia` è ristretto al package, nei programmi esterni al package non posso neanche scrivere il nome di questa classe

Però, se `l` è della classe `ListListaDoppia`, allora `l.iterator()` ritorna un oggetto `ListIteratorListaDoppia`

I programmi esterni al package *possono* usare l'oggetto solo mettendolo in una variabile interfaccia (`Iterator` o `ListIterator`)

A questo punto, possono usare solo i metodi dichiarati nell'interfaccia (`next`, `hasNext`, ecc)

Il metodo `remove`

Questo metodo è definito opzionale nel *contratto* dell'interfaccia `Iterator`

Quindi, va definito, ma può anche soltanto lanciare una eccezione:

```
  public void remove() {
    throw new
      UnsupportedOperationException(
      "remove not supported");
  }
```

Le componenti dell'interfaccia

Dato un `Iterator`, deve essere possibile scandire una lista

Servono quindi:

- la lista da scandire
- il nodo corrente

```
class ListIteratorListaDoppia
  implements ListIterator {

  Nodo pos;
  ListListaDoppia lista;

  ...
```

Il costruttore

Per quello che riguarda gli iteratori, basta questo:

```
  ListIteratorListaDoppia(ListListaDoppia lista) {
    this.lista=lista;
    pos = lista.sentinella;
  }
```

Il metodo `iterator()` della lista

Il metodo `iterator()` della lista è fatto cosí:

```
public class ListListaDoppia {
  ...
  public Iterator iterator() {
    return new ListIteratorListaDoppia(this);
  }
  ...
}
```

Viene ritornato un nuovo iteratore per la lista corrente.

I metodi `next` e `hasNext`

Il `next` ritorna il prossimo elemento:

```
  public Object next() {
    pos=pos.next;
    if(pos==lista.sentinella)
      throw new NoSuchElementException();
    return pos.info;
  }
```

Il metodo `hasNext` vede se c'è un prossimo elemento da guardare:

```
  public boolean hasNext() {
    return pos.next!=lista.sentinella;
  }
```

L'interfaccia `ListIterator`

Questa interfaccia ha due metodi in più per passare all'elemento precedente

Poi ci sono anche i metodi opzionali `add` e `set`

```
public void set(Object o) {
  throw new
    UnsupportedOperationException(
    "set not supported");
}
public void add(Object o) {
  throw new
    UnsupportedOperationException(
    "add not supported");
}
```

Metodi `previous()` e `hasPrevious()`

Servono per andare indietro nella scansione

Attenzione!

```
a=i.next();
b=i.previous();
```

Se si fanno in sequenza queste due invocazioni, viene ritornato lo stesso elemento

---

Implementazione di `previous`

Dato che `pos` indica la posizione dell'ultimo elemento ritornato da `next`, questo è anche quello che va ritornato da `previous`

```
public Object previous() {
  if(pos==list.sentinella)
    throw new NoSuchElementException();
  Object o=pos.info;
  pos=pos.prev;
  return o;
}
```

Nota: è diverso dal metodo `next()`!

`next`
: si porta avanti il puntatore e si ritorna l'elemento su cui si trova

`previous`
: si ritorna l'elemento dove si trova il puntatore *e poi* si manda indietro il puntatore

Poi vediamo il perchè

Il metodo `hasPrevious`

L'unico elemento che non ha il precedente è il primo

```
  public boolean hasPrevious() {
    return pos!=list.sentinella;
}
```

Anche questo è diverso da `hasNext`

Interpretazione di `next` e `previous`

Dal punto di vista della programmazione, un puntatore può solo trovarsi su un elemento della lista, mai in mezzo fra due

Nella *specifica* dei `ListIterator`, viene detto che il cursore **è come** se fosse sempre fra un elemento e l'altro:

Posizioni possibili del cursore:

```
Lista:   Primo Secondo Terzo Quarto
Curs:
iniz    ^
next          ^
next                ^
next                      ^
next                            ^
```

Il cursore è in questo caso un concetto relativo alla specifica

La specifica non dice che il puntatore di scansione deve stare in mezzo fra due elementi (cosa che sarebbe comunque impossibile)

`next` e `previous`

`next`
: ritorna l'elemento dopo il cursore, e lo fa avanzare

`previous`
: ritorna l'elemento prima del cursore, e lo manda indietro di una posizione

Questo riguarda il concetto astratto di cursore

Il puntatore dell'oggetto `ListIterator` deve ‘‘simulare’’ questo ‘‘stare in mezzo’’

Posizione del cursore

È importante perchè:

- se faccio `next` e poi `previous` ritorno sullo stesso elemento

  ```
  Lista:   Primo Secondo Terzo Quarto
                ^
  next                ^
  previos       ^
  ```

  in questo caso, `next` e `previous` ritornano lo stesso elemento

- i metodi opzionali `remove` e `set` lavorano sull'ultimo elemento ritornato dall'ultima invocazione di `next` o `previous`, e non sulla posizione del cursore

```
Lista:   Primo Secondo Terzo Quarto
              ^
next                  ^
```

qui viene canellato il secondo, se invece faccio:

```
Lista:   Primo Secondo Terzo Quarto
                            ^
prev                  ^
```

La posizione del cursore è la stessa, ma viene cancellato il terzo elemento

- il metodo `add` inserisce un elemento nella posizione dove si trova il cursore (che si trova fra due elementi)

Implementazione del cursore

Il cursore fa parte della specifica (a parole) della interfaccia `ListIterator`

Va implementato con un puntatore

Idea: il cursore si trova in mezzo fra il nodo dove c'è il puntatore e il successivo

```
Lista:   Primo Secondo Terzo Quarto
cursore               ^
puntatore         *
```

In questo esempio, il cursore sta fra il secondo e il terzo elemento; il puntatore che rappresenta questa posizione del cursore deve stare sul secondo elemento

Per questo `next` e `previous` sono diversi:

`next`
: per ritornare l'elemento dopo il cursore, bisogna prima avanzare il puntatore

`previous`
: l'elemento prima del cursore è quello dove attualmente si trova il puntatore; va quindi restituito questo elemento, prima di spostare il puntatore indietro

Si poteva anche scegliere di mettere il puntarore sull'elemento *successivo* al cursore

Andavano però cambiati di conseguenza tutti i metodi

Nota su previous

Nella implementazione vista prima, invocare il metodo `previous` come prima istruzione dava errore

Una soluzione possibile è quella di modificare `pos` in modo che inizialmente valga `null`

Alla prima invocazione di `next` o `previous`, viene modificato