Fondamenti di Informatica

Giuseppe Manco, Luigi Pontieri

Liste

Lezione 10

Settembre 2001

Tipi di dato e strutture dati

- Tipo di dato
  - (nei linguaggi di programmazione) valore che una variabile può assumere
  - (modello matematico) una collezione di valori sui quali sono ammesse certe operazioni
- Siamo interessati alla descrizione di un modello matematico in Java
  - Per fare questo ci aiutano gli oggetti
    - Una classe rappresenta un tipo di dato
- Una struttura dati è un tipo di dato
  - Non è importante il tipo degli elementi che la compongono
  - È importante l'organizzazione interna degli elementi

Esempi: tipi di dato

- Interi
  - La collezione dei numeri interi con le operazioni definite su di essa è un tipo di dato

$$(Z,+,\times)$$

  - Esiste il corrispettivo in Java
    - Il tipo **`int`**
    - Le operazioni tradizionali sugli interi

```
int a;
a = 1*(2+5);
…
```

Esempi: Strutture dati

- Array
  - Un array è una sequenza di elementi dello stesso tipo
  - Sugli elementi si possono effettuare operazione di lettura e scrittura
    - `Crea : Int × TypeSpec → Array`
    - `Leggi : Array × Int →tipoelem`
    - `Scrivi : Array × Int × tipoelem → Array`
- Implementazione in Java
  - Creazione

```
String[] animali = {"leone","tigre","orso"};
/* animali è un array di 3 elementi di tipo String */
```

  - Lettura

```
String nome;
…
/* Legge il 2° nome dall'array */
Nome = animali[2]
/* ora Nome contiene il valore "tigre" */
```

  - Scrittura

```
String nome = new String("canguro");
…
animali[2] = nome
/* L'array animali alla 12ª posizione assume il valore "canguro" */
```

Cos'è quindi una Struttura dati?

- Un modo sistematico di organizzare i dati
- Un insieme di operatori che permettono di manipolare elementi della struttura o di aggregare elementi per costruire altri agglomerati
  - Rappresenta fondamentalmente raggruppamenti di elementi
    - Insiemi
    - Non necessariamente gli elementi sono dello stesso tipo
    - È importante l'organizzazione degli elementi
- Caratterizzazione di una struttura dati
  - Lineari
    - Gli agglomerati sono formati da dati disposti in sequenza
  - Non lineari
  - A dimensione fissa
    - Il numero di elementi rimane costante nel tempo
  - A dimensione variabile
    - Il numero di elementi può variare
- **Come si caratterizza la struttura array?**
  - Lineare
    - È possibile individuare un ordine negli elementi
  - A dimensione fissa
    - La dimensione viene stabilita in fase di creazione e non può più essere modificata

Strutture dati in Java

- Alcune strutture sono predefinite in Java
  - Si identificano con tipi di dato
    - Array
    - Stringhe
- Come possiamo **creare** (e utilizzare) nuove strutture dati?
  - Con gli oggetti
    - I membri di un oggetto permettono di rappresentare i dati della struttura dati e la loro organizzazione interna
    - I metodi permettono di rappresentare gli operatori che manipolano gli elementi della struttura
- Un oggetto è quindi una struttura dati?
  - No! Un oggetto rappresenta una componente del programma
    - Le strutture dati possono essere componenti del programma, e quindi possono essere specificate tramite oggetti e classi
  - Esistono oggetti che non rappresentano strutture dati
    - Sapete darne un esempio?

Un esempio di Struttura dati complessa

- Supponiamo di voler estendere la struttura array
  - Vogliamo una struttura che aggrega elementi dello stesso tipo
    - Che permetta accesso in lettura e scrittura in base alla posizione
    - Che sia a dimensione variabile
      - Consente di aggiungere un elemento in fondo
      - Consente di eliminare un elemento dal fondo
  - Chiamiamo la nuova struttura **`ArrayDinamico`**
    - Per semplicità, assumiamo che gli elementi siano di tipo **`int`**

Un array dinamico

```
public class ArrayDinamico {
```

- Struttura dati lineare
  - Collezione di elementi di tipo intero
  - Comprende due operatori per l'accesso agli elementi

```
public int get(int pos) {…};
/* restituisce l'elemento alla posizione pos */
public void put(int pos, int elem) {…};
/* inserisce l'elemento elem nella posizione pos */
```

- A dimensione variabile
  - Può essere creato con una dimensione iniziale

```
public ArrayDinamico(int size) {…};
/* crea un array vuoto di dimensione size*/
```

  - È possibile conoscere la sua posizione in qualsiasi momento

```
public int size() {…};
/* restituisce la dimensione dell'array */
```

  - La sua dimensione può essere modificata aggiungendo ed eliminando elementi

```
public void add(int elem) {…};
/* aggiunge l'intero elem in fondo all'array */
public void remove() {…}
/* rimuove l'ultimo elemento dall'array */
…
}
```

Elementi di un array dinamico

- L'accesso agli elementi deve essere fatto in tempo costante
  - Utilizziamo un contenitore per gli oggetti

```
private int[] members;
/* contenitore per gli elementi dell'array */
```

  - Il costruttore inizializza il contenitore

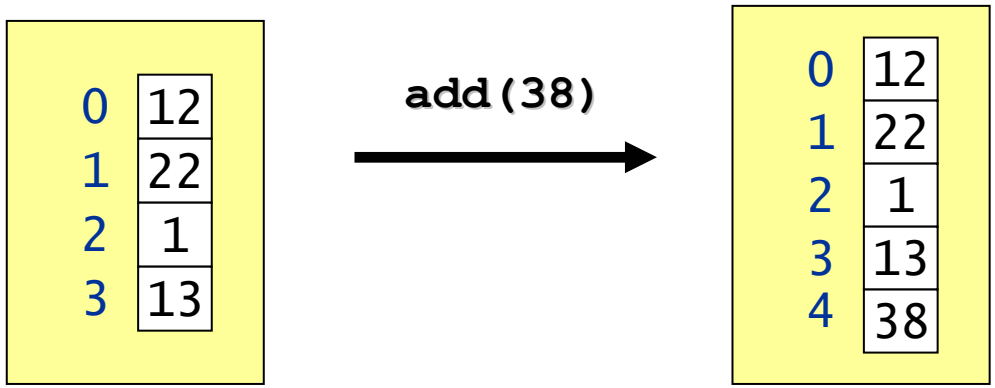
```
public ArrayDinamico(int size) {
    members = new int[size];
};
```

  - Le operazioni di lettura/scrittura corrispondono a quelle tradizionali

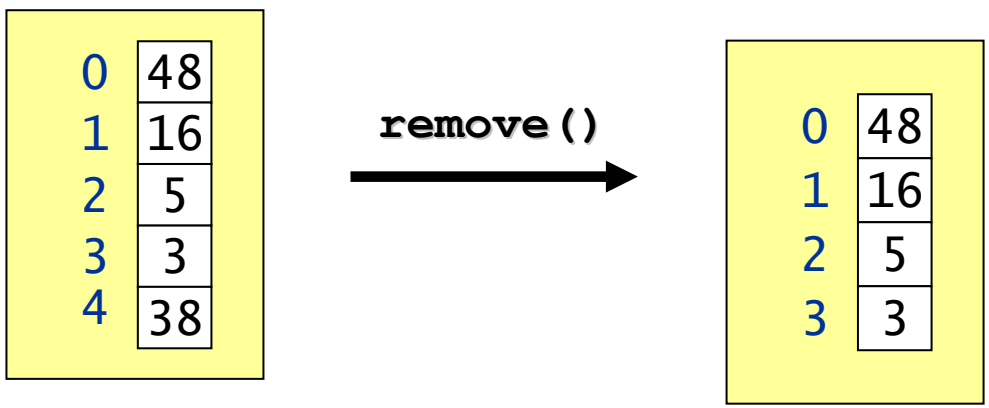
```
public int get(int pos){
    return members[pos];
    /* può generare una ArrayIndexOutOfBoundsException */
};
public void put(int pos,int elem){
    members[pos] = elem;
    /* può generare una ArrayIndexOutOfBoundsException */
};
```

Operazioni su un array dinamico: inserimento e rimozione

- Inserimento in coda



- Rimozione dalla coda



Inserimento

- Creiamo un array temporaneo con una capacità più ampia
- Riversiamo gli elementi dell'array contenitore all'interno di questo array
- Aggiungiamo in fondo all'array temporaneo l'elemento da inserire
- L'array temporaneo diventa il nuovo array contenitore

```
public void add(int elem) {
    int size = members.length;
    int[] temp = new int[size+1];

    for (int i = 0; i < size; i++)
                temp[i] = members[i];

    /* aggiunge il nuovo elemento nell'ultima posizione */
    temp[size] = elem;

    /* l'array temporaneo diventa il nuovo array */
    members = temp;
}
```

Rimozione

- Creiamo un array temporaneo con una capacità ridotta
- Riversiamo gli elementi dell'array contenitore (tranne che l'ultimo) all'interno di questo array
- L'array temporaneo diventa il nuovo array contenitore

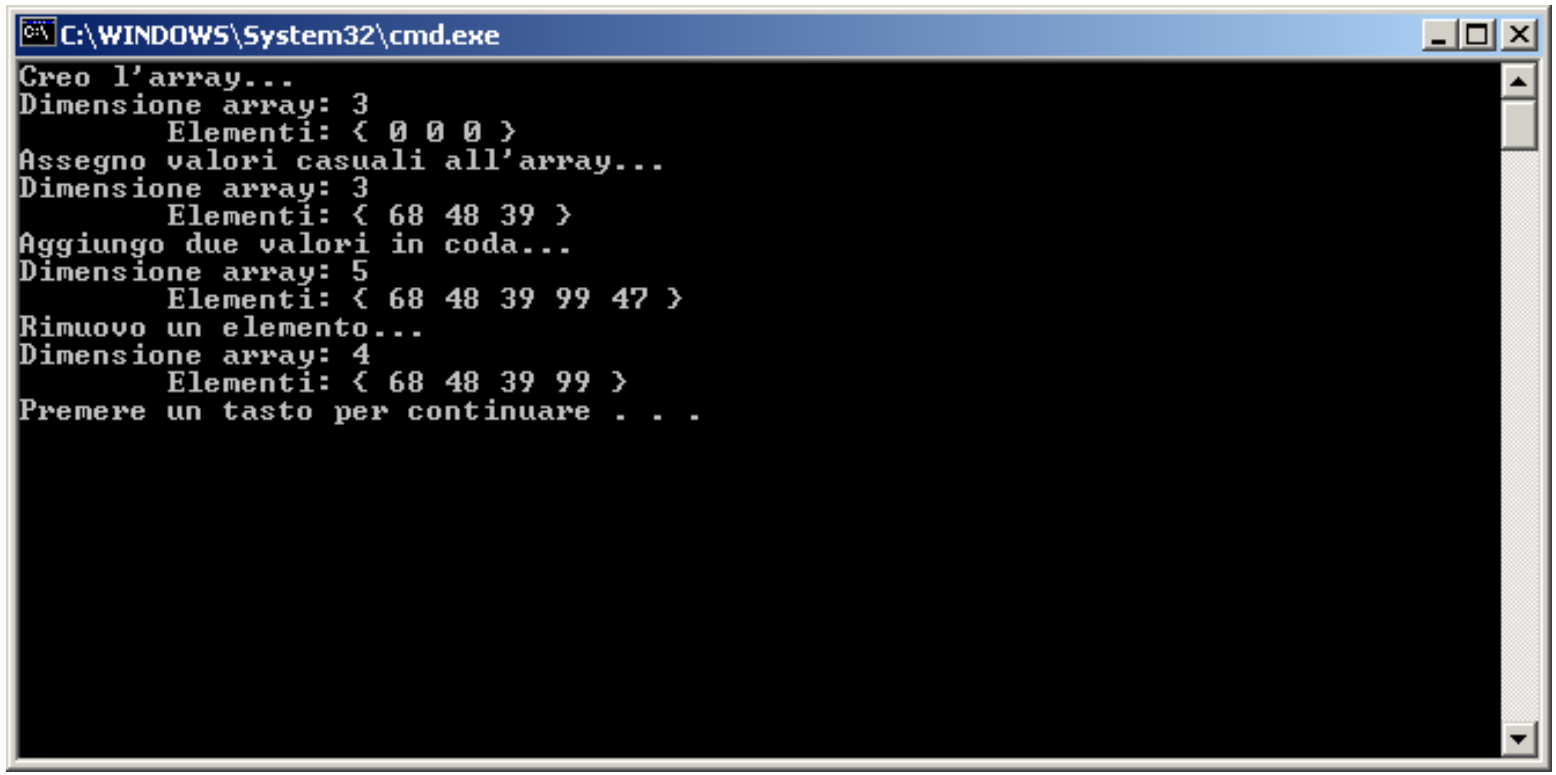
```
public void remove() {
        int size = members.length-1;
        int[] temp = new int[size];

        for (int i = 0; i < size; i++)
              temp[i] = members[i];

        members = temp;
}
```

Testing della struttura dati

- Aggiungiamo all'oggetto un metodo main per testarlo
- Utilizziamo un oggetto della classe Random per ottenere valori casuali
  - Il metodo nextInt(int n) restituisce un valore casuale tra 0 e n

```
public static void main(String args[]){
    ArrayDinamico array;      // Utilizzo una variabile locale
    int i;
    Random rnd = new Random();  // Inizializzo il generatore di numeri
                                // random
    System.out.println("Creo l'array...");
    array = new ArrayDinamico(3); // Creo l'array con una dimensione
                                  // iniziale di 3
    array.print();
    System.out.println("Assegno valori casuali all'array...");
    for (i = 0; i < array.length(); i++)
            array.put(i,rnd.nextInt(100));
    array.print();
    System.out.println("Aggiungo due valori in coda...");
    array.add(rnd.nextInt(100));  // aggiungo un elemento all'array
    array.add(rnd.nextInt(100));  // aggiungo un elemento all'array
    array.print();
    System.out.println("Rimuovo un elemento...");
    array.remove();             // Rimuovo un elemento dall'array
    array.print();
}
```

Testing della struttura dati



Quali sono i problemi di questa progettazione?

- Le operazioni di inserzione e di rimozione sono troppo costose
  - Ad ogni rimozione e inserimento, l'intero array viene ricopiato
  - Il problema principale consiste nell'organizzazione interna dei dati
    - L'array "logico" corrisponde ad un array "fisico"
    - La corrispondenza viene mantenuta ad ogni istante a livello fisico
  - È veramente necessaria tale organizzazione?
    - No. Possiamo ridefinire la corrispondenza in modo tale che non ci sia l'esigenza di ridimensionare l'array tutte le volte

Array Dinamico (2): Riorganizzazione interna

- Separiamo dimensione e capacità
  - La dimensione dell'array è il numero di elementi contenuti nell'array
  - La capacità è il numero di elementi che possono essere contenuti
    - Nell'implementazione precedente, dimensione e capacità corrispondevano (per lo meno nelle due operazioni di inserzione e rimozione)
  - L'inserimento e la rimozione modificano la dimensione, non la capacità
- Il contenitore rimane un array "statico"
  - La dimensione dell'array contenitore corrisponde alla capacità dell'array dinamico
  - Il contenitore va cambiato solo quando un inserimento pregiudica la sua capacità

Array Dinamico (2): Elementi

- Array contenitore

```
int[] members;
```

- Dimensione dell'array

```
int size;
```

  - Per tenere traccia del numero di elementi dell'array

- Capacità dell'array
  - Possiamo definire una dimensione iniziale di default

```
Static final int DEFAULT_CAPACITY = 10;
```

  - La capacità deve sempre essere maggiore o uguale alla dimensione

```
public ArrayDinamico2() {
    members = new int[DEFAULT_CAPACITY];
    size = 0;
}
public ArrayDinamico2(int capacity) {
    if (capacity < DEFAULT_CAPACITY)
        members = new int[DEFAULT_CAPACITY];
    else
        members = new int[capacity];
    size = 0;
}
```

Array Dinamico (2): operatori

- Accesso diretto agli elementi

```
public int get(int pos) {…};
public void put(int pos, int elem) {…};
```

  - Esiste una precondizione: non possiamo accedere ad un elemento in una determinata posizione se la posizione è superiore alla posizione massima consentita

```
public int get(int pos) throws Exception {
    if (pos < size)
            return members[pos];
    else
            throw new IndexOutOfBoundsException();
}
```

Array Dinamico (2): inserzione e rimozione

- Operando fondamentalmente sull'attributo size
  - size indica la posizione del primo elemento libero
    - Rimozione: decrementa size
    - Inserzione: inserisci l'elemento alla posizione size
  - Quanto può variare la struttura?
    - Possiamo rimuovere al più tutti gli elementi
      - Se il numero di elementi presenti è molto maggiore della capacità, l'array è sovradimensionato
    - Possiamo aggiungere al più tanti elementi per quanto ampia è la capacità
      - Se vogliamo aggiungere più elementi di quanto sia la capacità consentita, l'array è sottodimensionato
    - Soluzione: ridimensionamento
      - Imponendo che la capacità sia sempre superiore di un certo fattore alla dimensione del vettore

```
Static final int CAPACITY_FACTOR = 2;
```

Inserzione

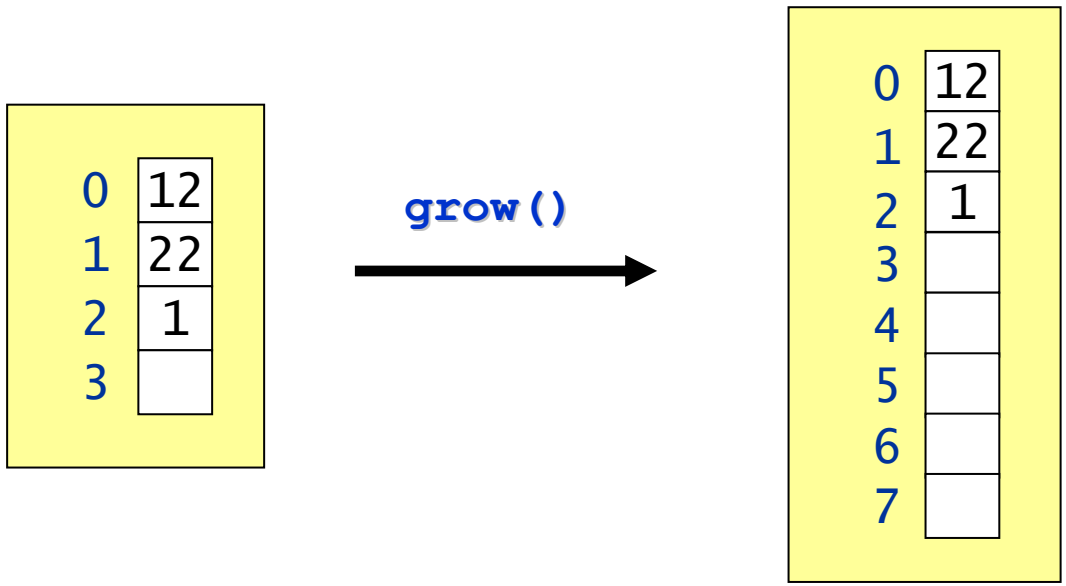
- Aggiunta
  - Il metodo **`grow()`** aumenta la capacità dell'array del fattore **`CAPACITY_FACTOR`**

```
public void add(int elem) {
        if ( size + 1 > members.length )
                grow();
        members[size] = elem;
        size++;
    }
}
```

- Rimozione
  - Il metodo **`shrink()`** riduce la capacità dell'array del fattore **`CAPACITY_FACTOR`**

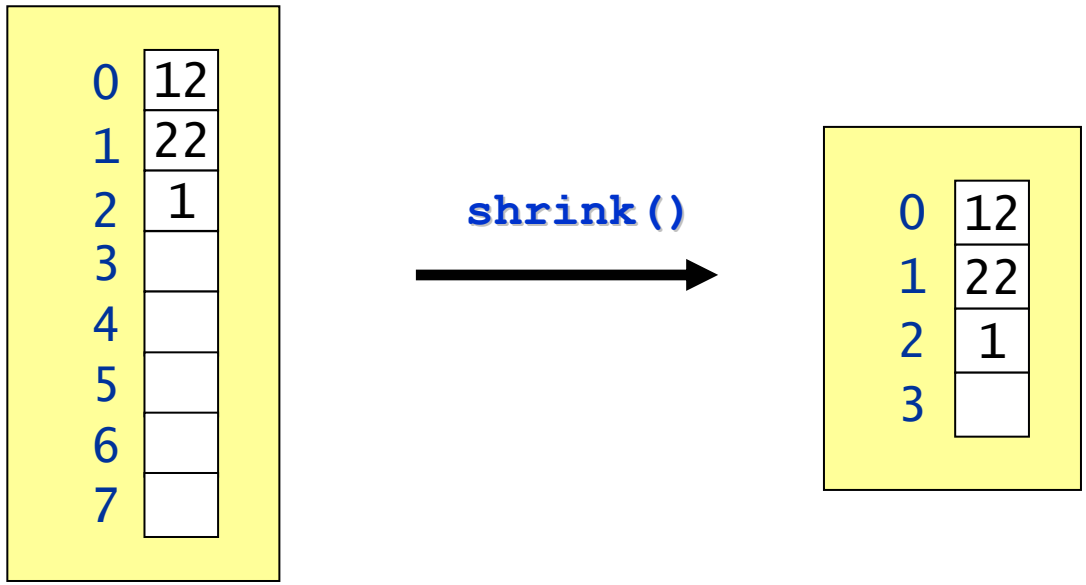
```
public void remove(int elem) {
    size--;
    if ( size < members.length/ CAPACITY_FACTOR )
                shrink();
    }
}
```

Ridimensionamento




```
private void grow() {
    int capacity = members.length;
    int[] temp = new int[capacity*CAPACITY_FACTOR];

    for (int i = 0; i < size; i++)
            temp[i] = members[i];

    members = temp;
}
```

Ridimensionamento

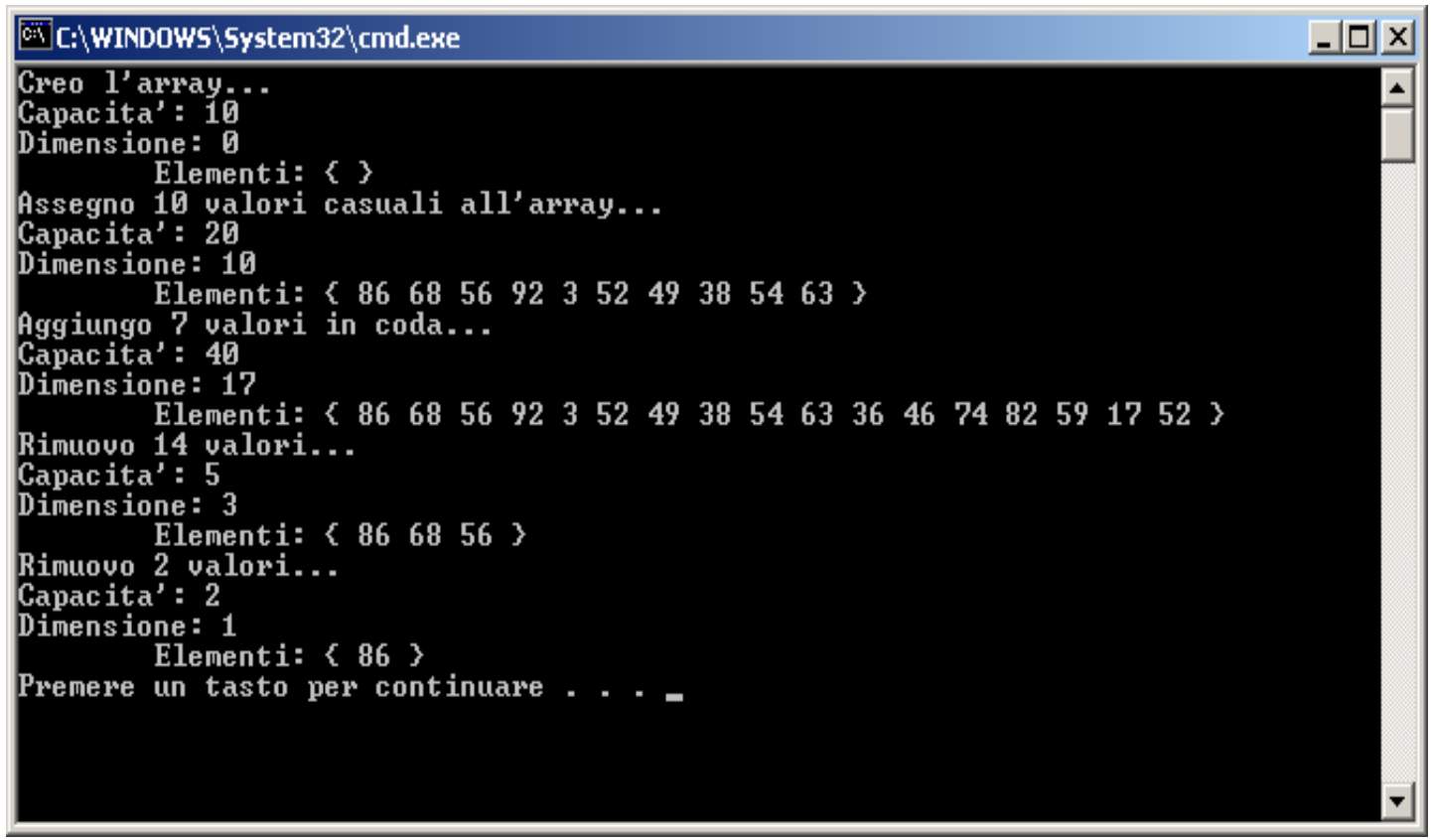



```
private void shrink() {
    int capacity = members.length/CAPACITY_FACTOR;
    if (capacity != 0) {
            int[] temp = new int[capacity];
            for (int i = 0; i < size; i++)
                    temp[i] = members[i];

            members = temp;
    }
}
```

Testing della struttura dati

```
public static void main(String args[]){
     ArrayDinamico2 array;
     int i;
     Random rnd = new Random();
     System.out.println("Creo l'array...");
     array = new ArrayDinamico2(10);
     array.print();
     System.out.println("Assegno 10 valori casuali all'array...");
     for (i = 0; i < 10; i++)
              array.add(rnd.nextInt(100));
     array.print();
     System.out.println("Aggiungo 7 valori in coda...");
     for (i = 0; i < 7; i++)
              array.add(rnd.nextInt(100));
     array.print();
     System.out.println("Rimuovo 14 valori...");
     for (i = 0; i < 14; i++)
              array.remove();
     array.print();
     System.out.println("Rimuovo 2 valori...");
     for (i = 0; i < 2; i++)
              array.remove();
     array.print();
}
```

Testing della Struttura dati



Esercizio: Aggiungiamo dinamicità

- La struttura dati descritta precedentemente permette inserire e/o eliminare elementi solo da una posizione prefissata
  - La fine dell'array contenitore, ovvero l'ultimo elemento inserito
- La situazione ideale sarebbe quella di poter aggiungere e/o rimuovere elementi in qualsiasi posizione
  - Implementando i seguenti metodi

    `remove(int pos)`
    - che rimuove un elemento dalla posizione pos

    `add(int elem, int pos)`
    - che inserisce un elemento nelle posizione pos
- Come possiamo implementare questi metodi?
  - Quali modifiche vanno fatte all'organizzazione della struttura dati **`ArrayDinamico`**?
  - Come problemi comporta l'implementazione di tali metodi?

Esercizio: Oggetti generici nell'array dinamico

- La struttura descritta precedentemente stabilisce il tipo di default
  - L'array è sempre di valori interi
- Possiamo definire una struttura generica, che sia in grado di implementare un array dinamico di qualsiasi tipo di elemento?
  - Possiamo utilizzare il tipo **`Object`**
    - **`Object`** è un tipo generico da cui ogni oggetto eredita
  - L'array contenitore diventa quindi un array di riferimenti a **`Object`**

```
private Object[] members;
```

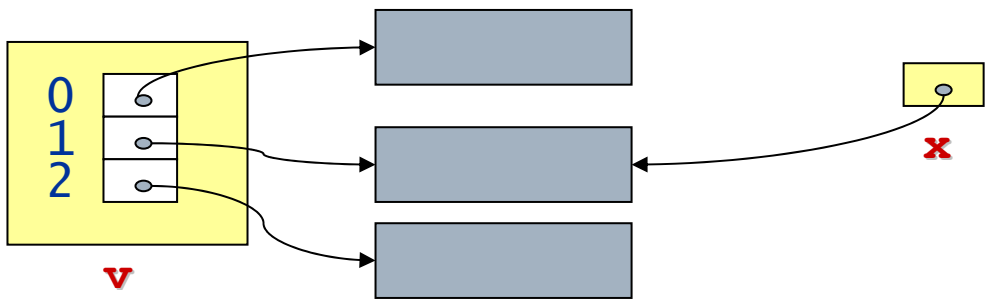
  - Problema: come garantire che tutti gli oggetti di **`members`** abbiano lo stesso tipo?
    - Raccogliendo informazioni sulla classe di appartenenza!

Array di oggetti: il problema degli oggetti condivisi

- Le operazioni di accesso della classe ArrayDinamico creano condivisione di oggetti, poiché essa utilizza un array di riferimenti agli elementi
- In generale, dato un array **v** di oggetti di classe **T** ed un riferimento **x** di tipo **T**
  - dopo l'istruzione

    ```
    x = v[i];
    ```

    x e *members[i]* si riferiscono allo stesso elemento



  - Problema (potenziale): ogni metodo di modifica richiamato su x agisce sull'oggetto riferito da members[i], e viceversa
  - NB: la condivisione di oggetti non è dannosa di per sé, purché il programmatore sia cosciente delle sue implicazioni

Array di oggetti: accesso senza condivisione di memoria

- Se di vuole evitare la condivisione di oggetti è necessario creare una copia dell'oggetto da restituire o da inserire
  - Nella classe ***T*** deve essere definito un metodo d'istanza che ritorna un riferimento ad una copia dell'oggetto su cui è invocato

```
public T copia()
```

- Nella classe ArrayDinamico potrebbero essere definite le seguenti funzioni di accesso

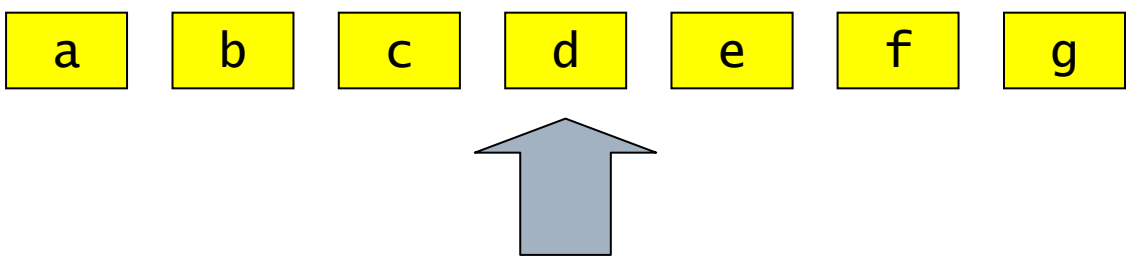
```
public int getCopy(int pos){
        return members[pos].copia();
};
public void putCopy(int pos, T elem){
        members[pos] = elem.copia();
};
public void addCopy(T elem ) {
        T [] temp = new T [members.length+1];
        for (int i = 0; i < members.length; i++)
                temp[i] = members[i];
        temp[members.length] = elem.copia();
        members = temp;
}
```

Una struttura dati predefinita: la classe Vector

- La classe Vector implementa un array dinamico di oggetti
  - Contiene componenti che possono essere accedute usando un indice intero
  - La capacità del vettore può aumentare e diminuire secondo necessità, in modo da permettere l'inserzione e l'eliminazione di elementi.
- Descritto nella documentazione Java
  - Implementa tutte le operazioni che abbiamo trattato
  - Struttura dati standard, importabile tramite il package **`java.util`**

Sequenze ordinate per posizione

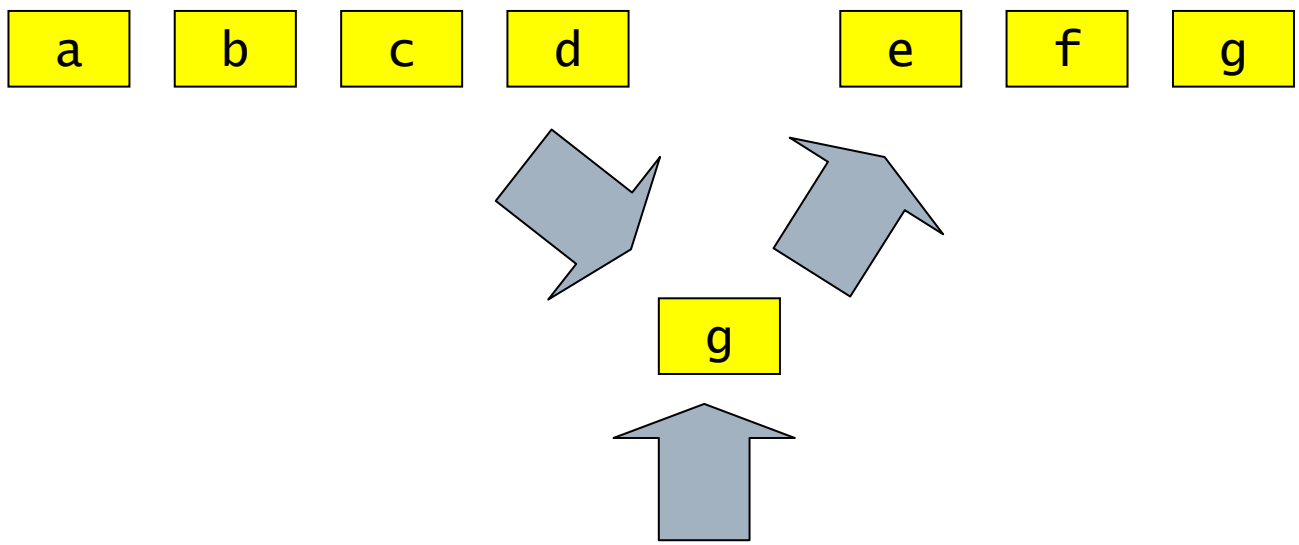
- Un vettore permette l'accesso diretto
  - In molti casi, l'unico accesso possibile è di tipo sequenziale
    - Per accedere ad a_i, devo prima accedere ad a_{i-1}
    - Esempio: l'elenco delle persone che possono accedere ad un servizio
      - L'accesso è sequenziale
    - È importante la posizione:



  - a viene prima di b che viene prima di c che …

Sequenze ordinate per posizione

- Aspetti rilevanti
  - Dinamicità
    - L'implementazione con vettori renderebbe una sequenza altamente dinamica molto inefficiente



Sequenze ordinate per posizione

- Aspetti non rilevanti
  - Accesso diretto
    - Il reperimento di un'informazione può essere fatta in maniera sequenziale
- Un esempio: l'insieme dei collaboratori di un'azienda
  - Un'azienda di volantinaggio ha un insieme di collaboratori ai quali assegna mansione di distribuzione volantini
    - L'elenco dei collaboratori dell'azienda varia molto spesso
    - Ad ogni collaboratore viene assegnato un compenso mensile fissato
  - Operazioni
    - Aggiunta collaboratore
    - Rimozione collaboratore
    - Aggiornamento compensi per ogni collaboratore
      - L'aggiornamento viene fatto periodicamente per tutti i collaboratori!

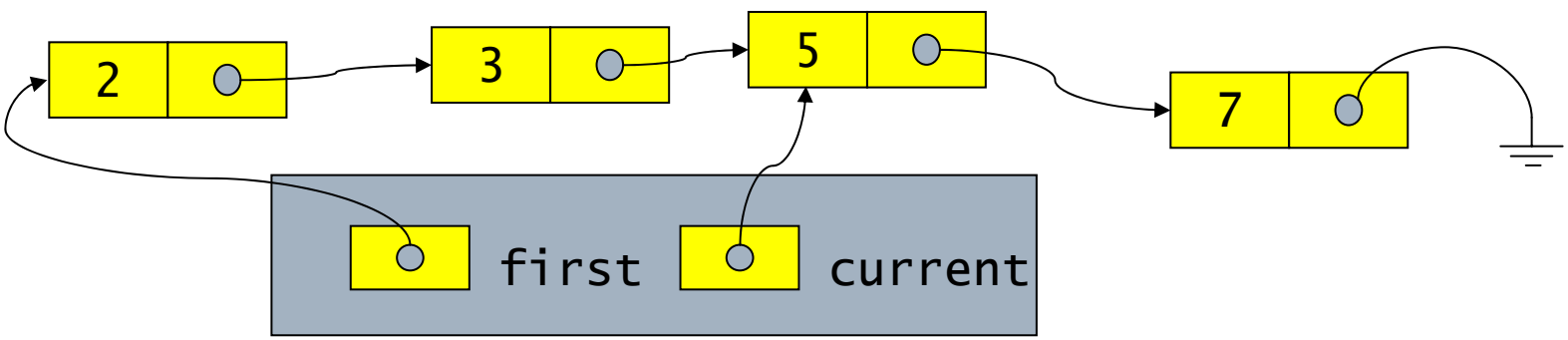
Realizzazione di Sequenze

- Implementazione tramite array
  - Vantaggi: accesso diretto
    - È veramente utile?
      - La scansione viene fatta una volta per tutte
  - Svantaggi: la dinamicità e' molto costosa

- Quale soluzione alternativa possiamo adottare?
  - Vogliamo una struttura che consenta di scandire gli elementi
  - E che al tempo stesso mi permetta operazioni dinamiche in maniera molto efficiente
    - Aggiunta in una posizione
    - Rimozione da una posizione
  - Quale efficienza?
    - Scansione lineare
    - Update costante

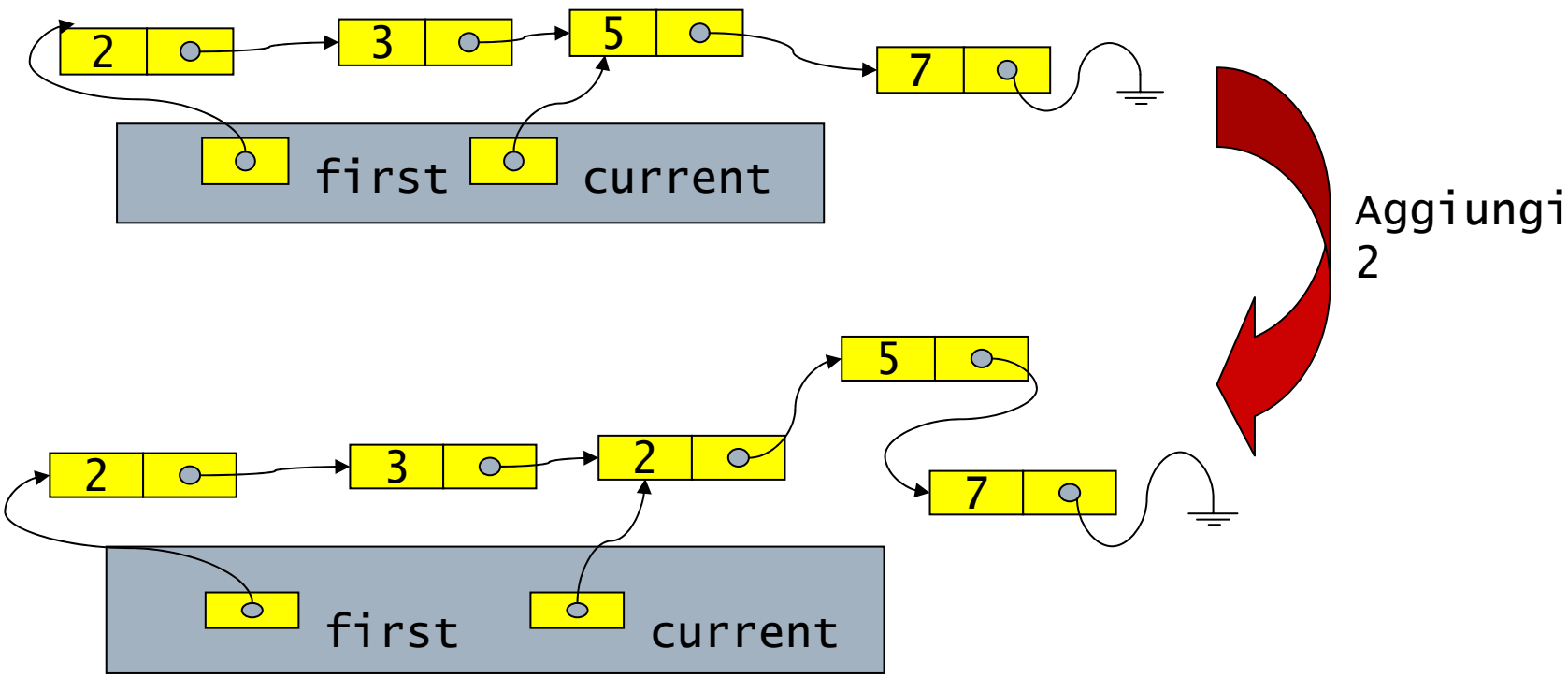
Liste

- Una lista è un insieme di elementi ordinati per posizione
  - Ogni elemento è collegato all'elemento che lo segue
  - Non è possibile accedere ad un elemento senza avere acceduto l'elemento che lo precede
    - Due operazioni (statiche)
      - Accedi all'elemento corrente
      - Avanza al prossimo elemento
  - Due punti d'accesso
    - Il primo elemento
    - L'elemento corrente



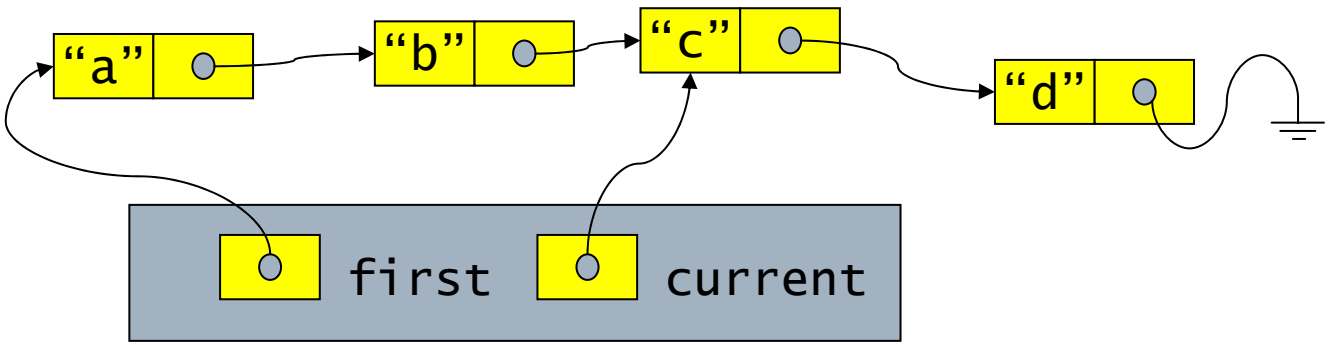
Liste

- Aggiungi un elemento alla posizione corrente



Liste: aspetti salienti

- Parametricità
  - Il tipo di dato astratto lista (rappresentazione+operazioni) può essere definito a prescindere dal tipo degli elementi
    - Esempio: liste di interi, di stringhe, di vettori, …



- Estendibilità
  - Una “porzione” di lista è una lista (sotto-lista)
    - Operazioni tipiche: merge di due liste, identificazione di una sottolista, …

Implementazione di una Lista

- La lista è un tipo di dato astratto
  - Ha un comportamento
  - Ha una interfaccia
  - Ha uno stato

- La sua implementazione deve permettere la realizzazione di tutte le sue peculiarità
  - Rappresentare una sequenza di elementi
  - Rappresentare operazioni su questa sequenza

Implementazione per collegamenti

- Lista linkata
  - Un elemento della lista è rappresentato mediante un nodo
    - Memorizza il valore contenuto in una determinata posizione
    - Memorizza il collegamento al prossimo elemento
  - La lista mantiene i riferimenti a due elementi notevoli
    - Il primo elemento
    - L'elemento corrente

Nodo

```
class Nodo{
    private int info;       // eventualmente altri tipi
    private Nodo next;

    public Nodo (int val){
         info = val;
         this.next = null;
    }
    public Nodo getNext(){
         return this.next;
    }
    public int getVal(){
         return info;      // eventualmente info.copy()
    }
    public void setNext(Nodo n){
         if (n != null)
                  n.next = this.next;
         next = n;
    }
}
```

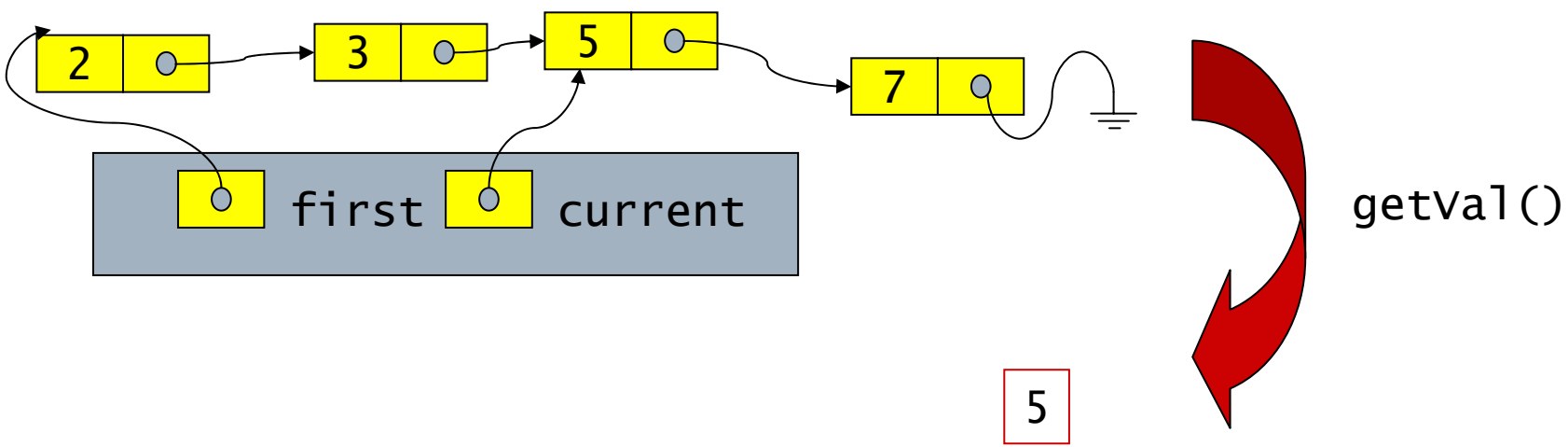
Lista

```
class Lista{
   private Nodo first;
   private Nodo current;

   public Lista (){
       first = null;
       current = null;
   }
```

- Due campi: first e current
  - First riferisce il primo elemento
  - Current riferisce l'elemento corrente

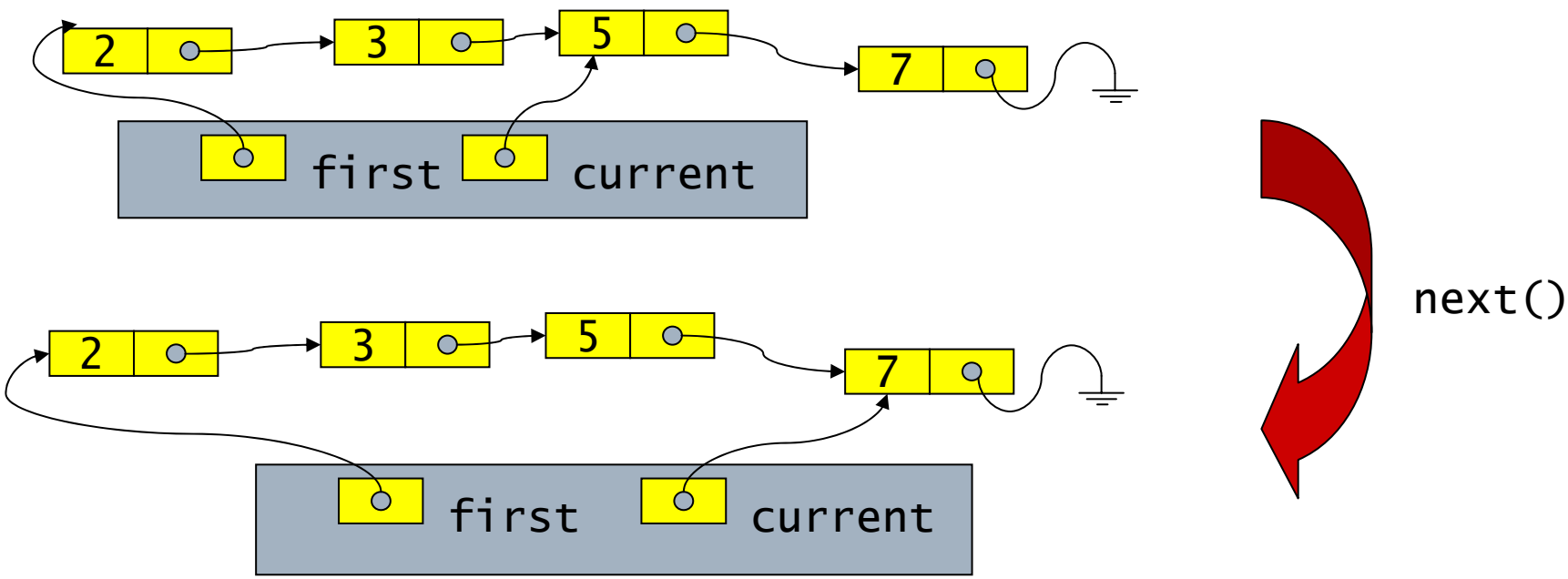
Accesso

```
public boolean hasCurrent (){
     return (current != null);
}

public int getVal() throws Exception
{
     if (hasCurrent())
               return current.getVal();
     else
               throw new Exception("no current element");
}
```



Avanzamento

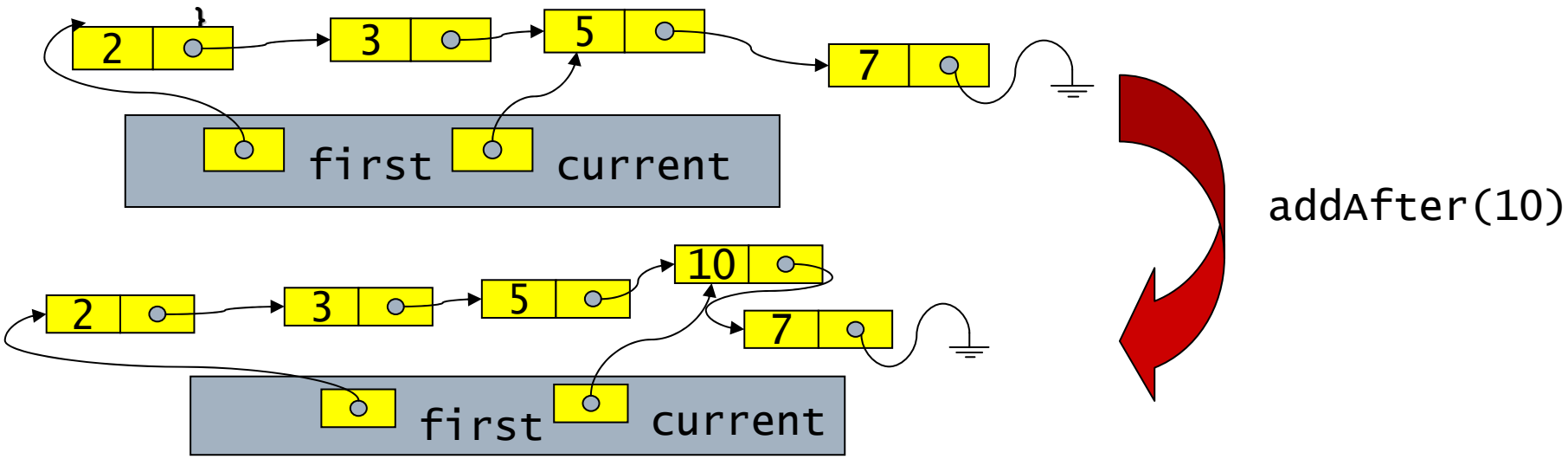
```
public void next (){
     if (current != null)
             current = current.next();
}
```



Dinamicità: Aggiunta di un nuovo elemento

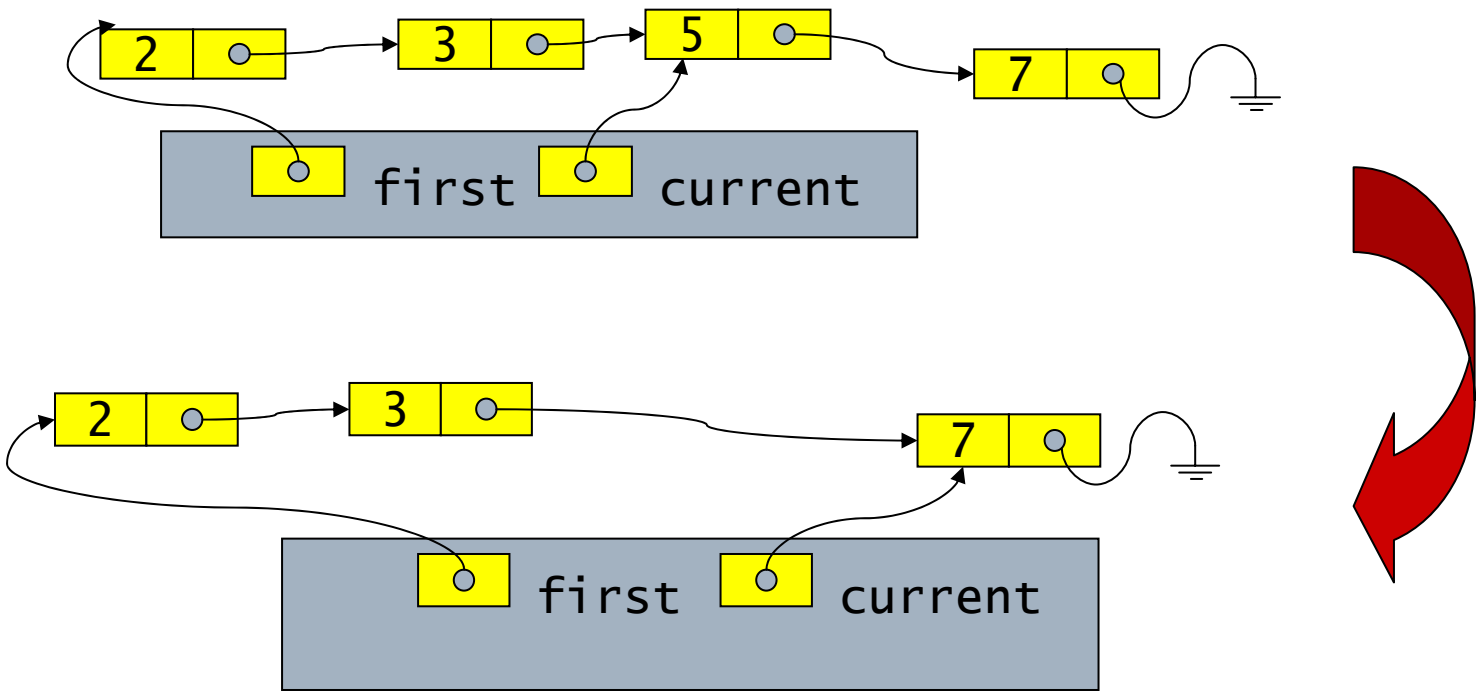
- AddAfter inserisce un elemento dopo il corrente

```
public void addAfter (int val){
    if (first == null){
            first = new Nodo(val);
            current = first;
    }
    else {
            Nodo n = new Nodo(val);
            current.setNext(n);
            current = current.getNext();
    }
```



Dinamicità: Rimozione

- Distrugge l'elemento corrente



Rimozione

- L'elemento corrente non è più un nodo della lista
  - non più è riferito da nessun riferimento e sarà deallocato dal garbage collector

```
public void remove (){
  if (current != null)
      if (first != current) {
          Nodo pr;
          for (pr=first; pr.getNext()!=current; pr=pr.getNext());
          pr.setNext(current.getNext());
      }
      else
          first = current.getNext();
      current = current.getNext();
   }
}
```

Inefficienza della realizzazione

- Inefficienza
  - Nella rimozione l'accesso all'elemento da rimuovere non è costante (dipende dal numero di elementi che lo precedono)
    - E' necessario accedere all'elemento che precede l'elemento corrente
  - Anche per inserire un elemento prima del corrente l'accesso non è costante
- Soluzione
  - Introduciamo nell'oggetto lista un riferimento all'elemento che precede il corrente

```
class Lista{
   private Nodo first;
   private Nodo current;
   private Nodo previous;

   public Lista (){
         first = null;
         current = null;
         previous = null;
   }
         ...
}
```

Inserimento

- E' possibile inserire in tempo costante un elemento prima dell'elemento corrente (il nuovo elemento diventa corrente)

```
public void add (int val){
    if (first == null){
            first = new Nodo(val);
            current = first;
    }
    else {
            Nodo n = new Nodo(val);
            n.setNext(current);
            if (previous != null)
                    previous.setNext(n);
            else
                    first = n;
            current = n;
    }
}
```

Rimozione

- L'elemento successivo al corrente diventa il nuovo corrente

```
public void remove () {
    if (current != null){
            if (previous != null)
                    previous.setNext(current.getNext());
            else
                    first = current.getNext();
            current = current.getNext();
    }
}
```

Altre operazioni

- Posizionamento del cursore sul primo elemento della lista
  - può essere utilizzato per effettuare una nuova scansione della lista

```
public void reset(){
        current = first;
        previous = null;
}
```

- Indicazione se la lista è vuota o meno

```
public boolean empty(){
        return (first == null);
}
```

Altre operazioni utili

- Conversione della lista in stringa (contenuto della lista)

```
public String toString(){
        String strRet = "";

        for (Nodo cur=first; cur!= null; cur=cur.getNext()){
                if (cur = = current)
                        strRet += "[" + cur.getVal() + "] ";
                else
                        strRet += cur.getVal() + " ";
        }
        return strRet;
}
```

- Inserimento/estrazione di un elemento all'inizio della lista ("in testa")
- Inserimento/estrazione di un elemento alla fine della lista ("in coda")

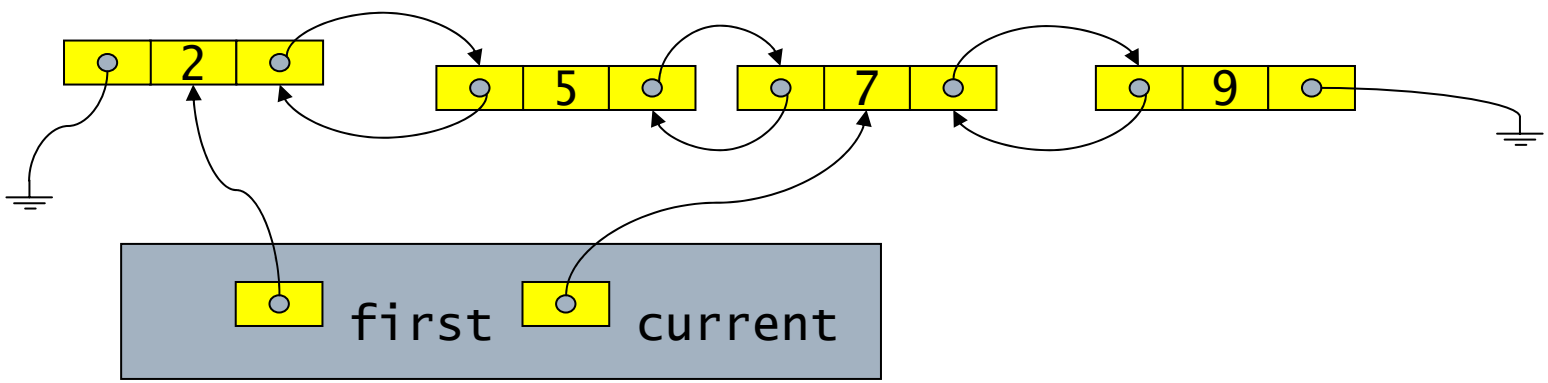
Test della Struttura

```
public static void main(String[] args){
    Lista l = new Lista();
    l.add(1);      l.add(2);      l.add(4);
    l.add(5);      l.remove();    l.add(6);
    l.next();      l.next();      l.remove();
    l.remove();    l.add(34);     l.reset();
    l.remove();    l.add(5);      l.add(4);
    l.add(12);     l.next();      l.next();
    l.next();      l.remove();

}
```

- Esercizio
  - Tracciare lo stato della lista dopo ogni operazione, specificando qual è l'elemento corrente e qual è l'elemento precedente

Liste doppiamente linkate

- La lista presentata finora può scorrere in una sola direzione
- Modificare la semantica di un nodo:
  - Un nodo può puntare sia all'elemento successivo che al precedente
  - Non è più necessario conservare nella lista il riferimento all'elemento che precede il corrente
  - Possiamo muoverci in entrambe le direzioni



Liste doppiamente linkate: Nodo

```
class Nodo{
    private int info;
    private Nodo next;
    private Nodo prev;
    public Nodo (int val){
          info = val;
          next = null;
          prev = null;
    }
    public Nodo getNext(){
          return next;
    }
    public void setNext(Nodo n){
          next = n;
    }
    public Nodo getPrevious(){
          return prev;
    }
    public void setPrevious(Nodo n){
          prev = n;
    }
    public int getVal(){
          return info;
    }
```

Liste doppiamente linkate: Struttura

- Introduciamo un riferimento per avere un accesso costante all'ultimo elemento della lista
  - per inserire o estrarre un elemento in coda non è necessario scandire tutti gli elementi che lo precedono nella lista

```
class DLista{
        private Nodo first;
        private Nodo last;
        private Nodo current;
        public DLista(){
                first = null;
                current = null;
                last = null;
        }
        ...
```

Liste doppiamente linkate: Operazioni

```
// ritorna true se la lista è vuota, false altrimenti
public boolean empty(){
    return (first == null);
}

// ritorna true se esiste un elemento corrente, false
// altrimenti
public boolean hasCurrent(){
    return current != null;
}

// ritorna il valore dell'elemento corrente
public int getCurrent(){
    return current.getVal();
}
```

Liste doppiamente linkate: Spostamento del cursore

```
// rende corrente il primo elemento
public void reset(){
    current = first;
}
// rende corrente l'ultimo elemento
public void gotoLast(){
    current = last;
}
// rende corrente l'elemento che precede il corrente
public void previous(){
    current = current.getPrevious();
}
// rende corrente l'elemento che segue il corrente
public void next(){
    current = current.getNext();
}
```

Liste doppiamente linkate: Aggiunta

```
public void addBefore(int val){       // inserisce un elemento
        if ( empty() )                // prima del corrente
                last = current = first = new Nodo(val);
        else {
                Nodo prec, n = new Nodo(val);
                if ( current != null ) {
                        prec = current.getPrevious();
                        current.setPrevious(n);
                }
                else
                        prec = last;
                n.setPrevious(prec);
                if (prec != null)
                        prec.setNext(n);
                else
                        first = n;
                n.setNext(current);
                current = n;
        }
    }
```

Liste doppiamente linkate: Rimozione

- Rimozione dell'elemento corrente
  - diventa corrente il successivo, se esiste
  - diventa corrente il precedente, altrimenti

```
public void remove(){
     if ( !empty() ){
             Nodo prev = current.getPrevious(),
             nxt = current.getNext();
             if (prev != null)
                     prev.setNext(nxt);
             else
                     first = nxt;
             if (nxt != null)
                     nxt.setPrev(prev);
             else
                     last = prev;
             current = nxt;
     }
}
```

Test della Struttura

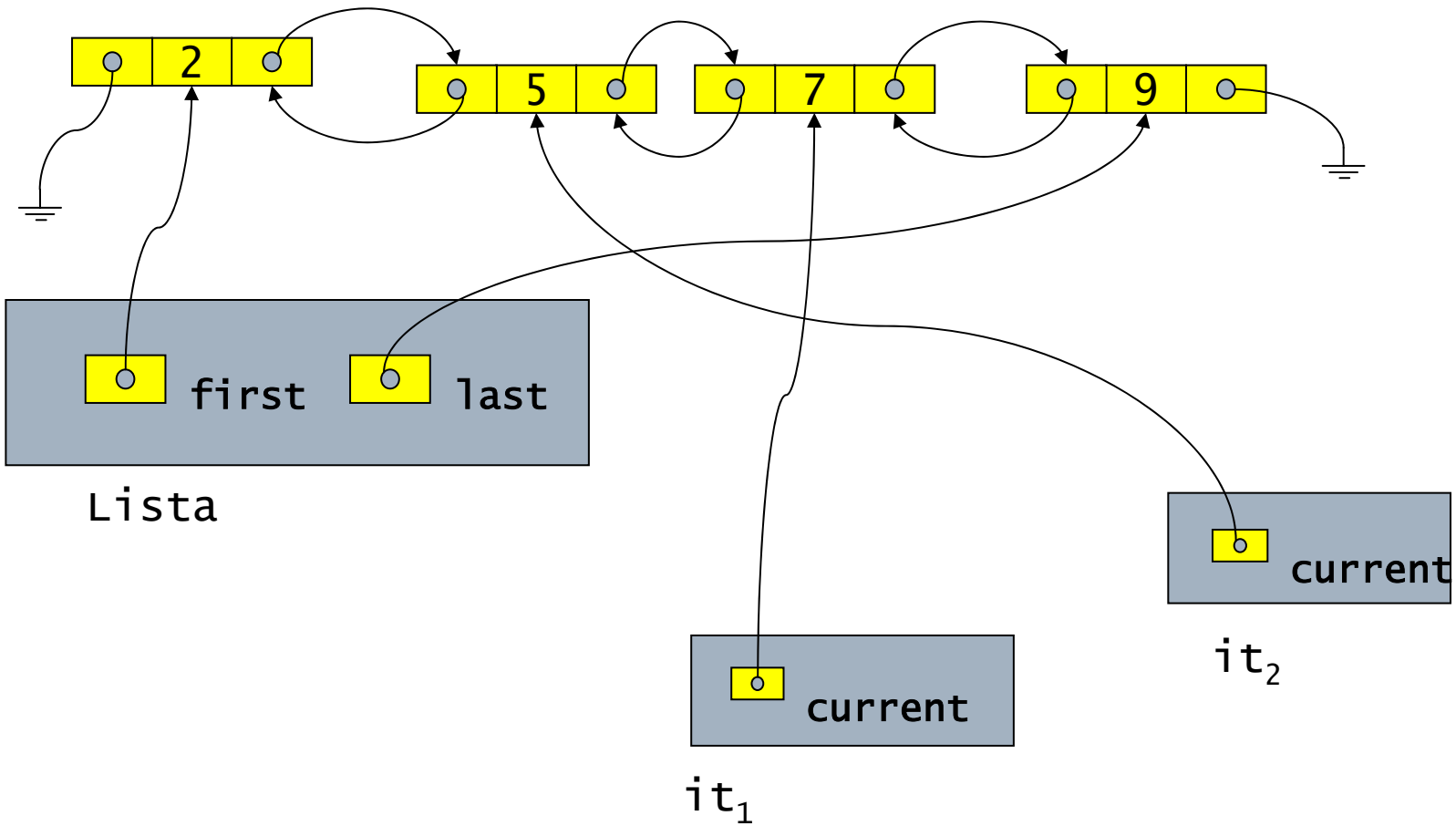
```
public static void main(String[] args){
   DLista l = new DLista();
   l.add(1);   l.add(2);   l.add(4);
   l.add(5);   l.remove(); l.add(6);
   l.next();   l.next();   l.remove();
   l.remove(); l.add(34);  l.reset();
   l.remove(); l.add(5);   l.add(4);
   l.add(12);  l.next();   l.next();
   l.next();   l.remove();

}
```

- Esercizio
  - Tracciare lo stato della lista dopo ogni operazione, specificando qual è l'elemento corrente

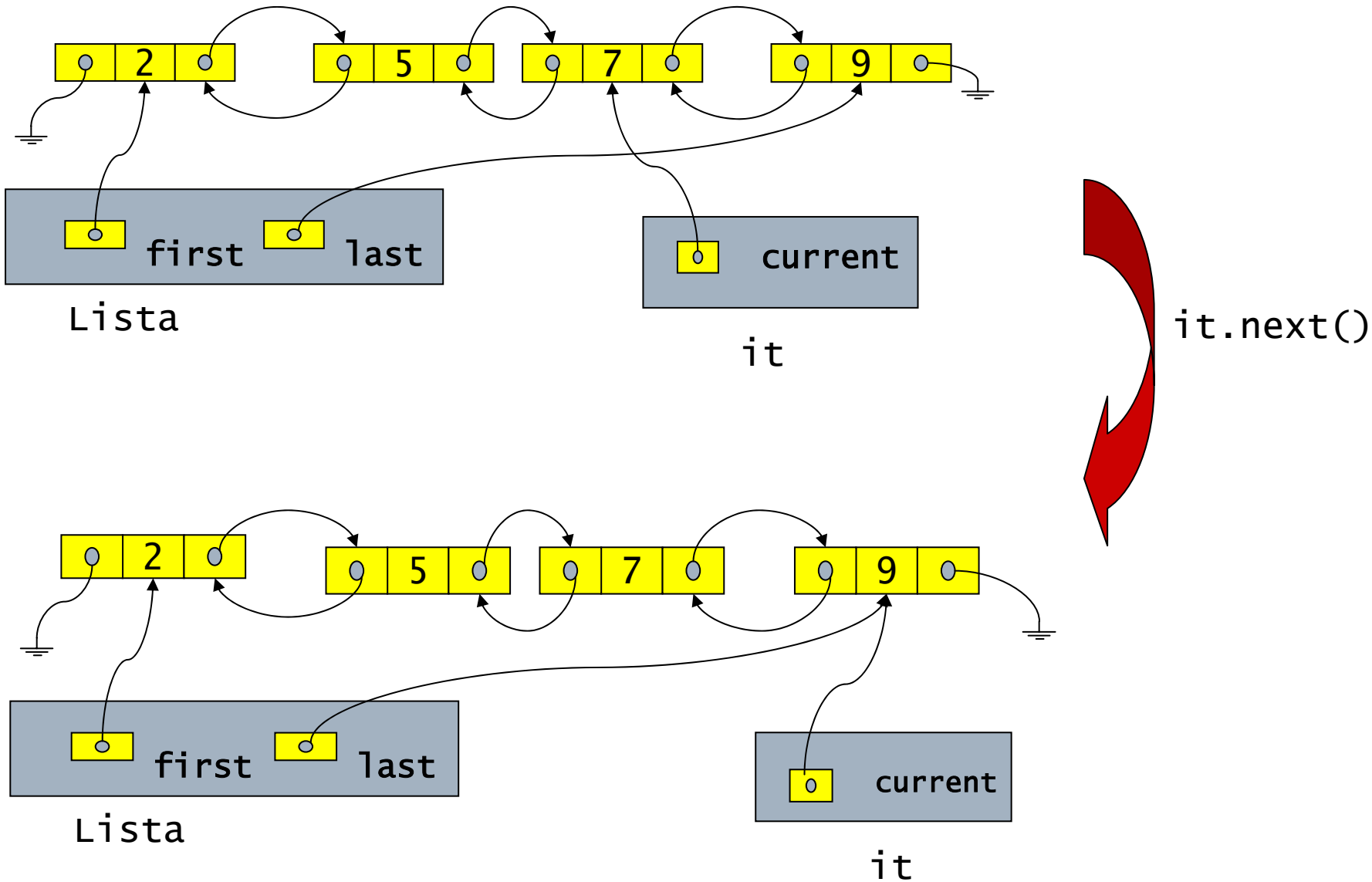
Iteratori

- Il problema con le implementazioni precedenti sta nel fatto che la lista e la sua scansione sono un'unica cosa
  - Non è possibile definire più scansioni contemporanee della lista
  - Non è possibile accedere a più elementi della stessa lista contemporaneamente
- Soluzione: introduciamo gli iteratori
  - Un iteratore tiene traccia dell'elemento corrente in maniera separata dalla lista
    - Si tratta di una classe separata dalla classe lista, che tuttavia punta ad un elemento della lista
    - Simile all'indice di un array
      - Possiamo spostarlo avanti e indietro, e accedere agli elementi che punta
        - Tuttavia ammette solo spostamenti sequenziali

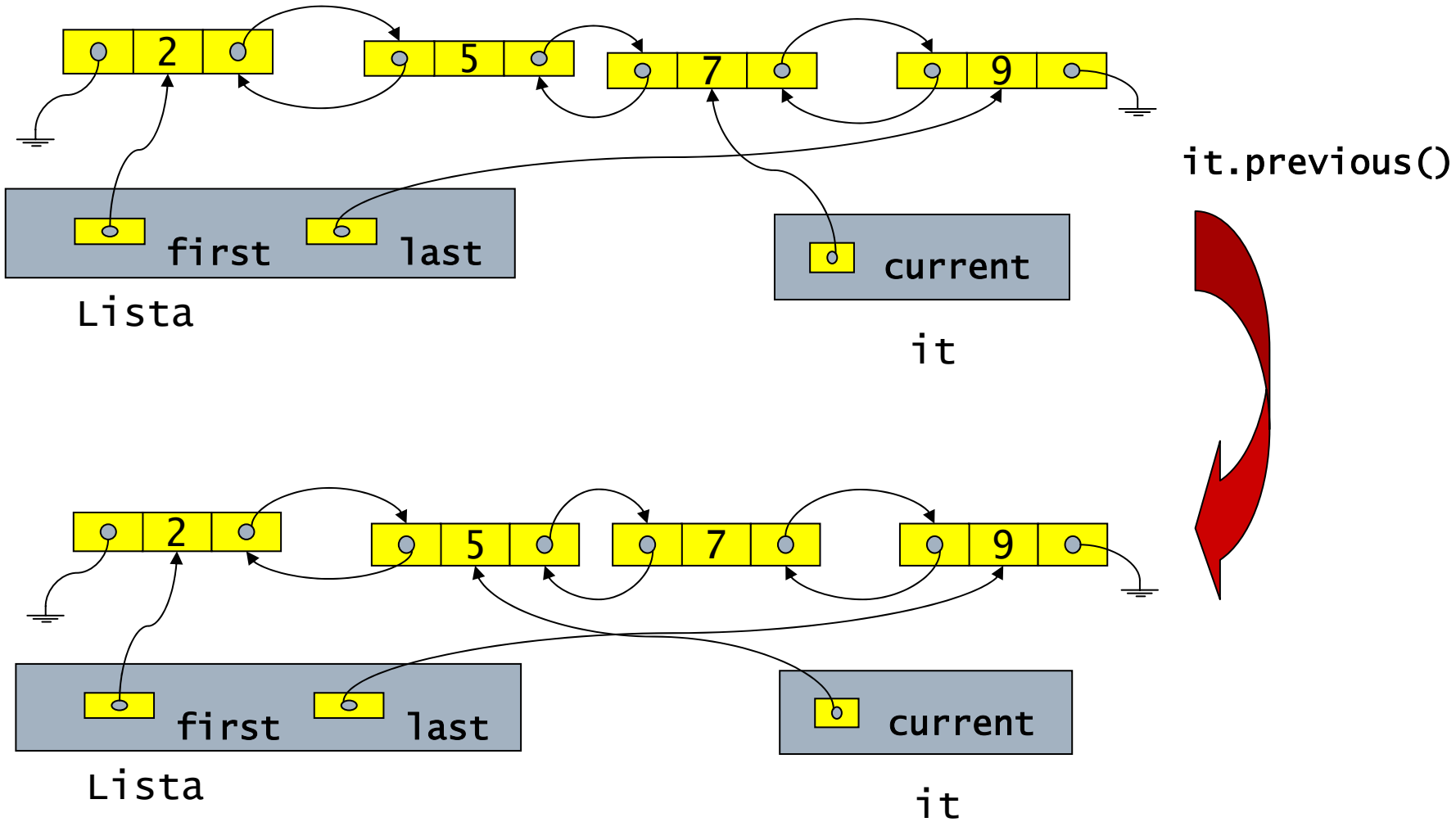
Iteratori



Iteratori


2
5
7
9
first
last
Lista
current
it
it.next()
2
5
7
9
first
last
Lista
current
it

Iteratori


2
5
7
9
first
last
Lista
current
it
it.previous()
2
5
7
9
first
last
Lista
current
it

Liste Linkate: Implementazione di Riferimento

- La classe **`LinkedList`**
  - **`LinkedList()`** Costruisce una lista vuota
  - **`boolean empty()`** restituisce **`true`** se la lista è vuota
  - **`int size()`** restituisce la dimensione della lista
  - **`ListIterator listIterator()`** restituisce un iteratore alla lista, posizionato sul primo elemento della lista
  - **`int get(ListIterator it)`** restituisce il valore dell'elemento puntato da **`it`**
  - **`void remove(ListIterator it)`** elimina l'elemento puntato da **`it`** e posiziona **`it`** sull'elemento successivo
  - **`void insert(ListIterator it, int x)`** inserisce un elemento con valore **`x`** come precedente di quello puntato da **`it`** (se **`it`** non punta ad alcun elemento, il nuovo inserito in coda). L'iteratore punterà al nuovo elemento inserito.
  - **`void addFirst(int e)`** Inserisce un elemento **`e`** in prima posizione
  - **`void addLast(int e)`** Inserisce un elemento **`e`** in ultima posizione
  - **`void removeFirst()`** Elimina il primo elemento
  - **`void removeLast()`** Elimina l'ultimo elemento

Iteratori: Implementazione di Riferimento

- La classe **`ListIterator`**
  - **`boolean hasCurrent()`** restituisce **`true`** se esiste un elemento corrente
  - **`int next()`** sposta il cursore corrente sull'elemento successivo
  - **`int previous()`** sposta il cursore corrente sull'elemento precedente
  - **`void begin()`** sposta il cursore all'inizio della lista
  - **`void end()`** sposta il cursore alla fine della lista
  - **`ListIterator(LinkedList l)`** crea un iteratore sulla lista l, posizionato sul primo elemento di l
  - **`ListIterator copy()`** crea una copia dell'iteratore

Esempi di utilizzo di Liste: Sequenze Ordinate

- Inserimento degli elementi in una lista in modo che costituiscano una sequenza ordinata (ordinamento crescente)

```
public static void inserisci(int x, LinkedList l) {
          ListIterator it = l.listIterator();

          for(it.begin(); it.hasCurrent() && l.get(it)< x;
                it.next());

          l.insert(it,x);

}
```

Esempi di utilizzo di Liste: Ricerca di un elemento

- Ricerca di un elemento con un valore dato in una lista ordinata
  - la ricerca può terminare se si trova un elemento uguale o maggiore
  - *se l'elemento è nella lista il metodo ritorna true e posiziona il cursore (current) sull'elemento trovato,*
  - *se l'elemento con valore x non è presente nella lista il metodo ritorna false e posiziona il cursore:*
    - *sul primo elemento maggiore di quello trovato, se esiste tale elemento*
    - *sul fine lista (current==null), se tutti gli elementi sono minori di x*

```
public static boolean ricerca ( Lista l, int x ) {
    ListIterator it = l.listIterator();
    for (it.begin(); it.hasCurrent && (l.get(it) < x); it.next())
            ; // il corpo del for è l'istruzione nulla
     return ( it.hasCurrent() && l.get(it) == x );
}
```

Esercizi su Liste

- Verificare se due liste contengono la stessa sequenza di elementi
- Verificare se due liste contengono lo stesso insieme di elementi
- Invertire una lista
- Costruire una lista ordinata senza duplicati contenente gli elementi di una lista data