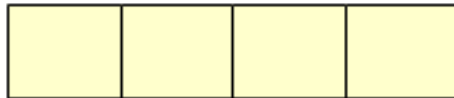

Liste

Dal materiale di Stefano Ferretti
s.ferretti@unibo.it

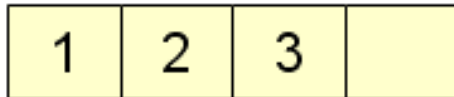
Limitazioni degli array

- Abbiamo visto come definire una sequenza di variabili tramite **array**
- Un array è una struttura dati **statica**
 - La **lunghezza** è specificata al momento della creazione
 - Dopo la creazione **non** può essere più essere modificata
- In molti casi è molto **difficile stimare** la dimensione necessaria al momento della creazione dell'array

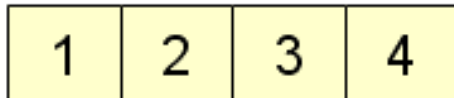
Limitazioni degli array



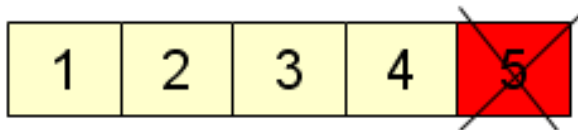
Create an empty integer array.



Fill it partially with some data. The array component without a number indicates allocated but unused space. This is space you could have used for something better.



Add another element. We now have a full array to which we can not add any more elements. We can delete or replace, but we can not add.



If the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

Liste

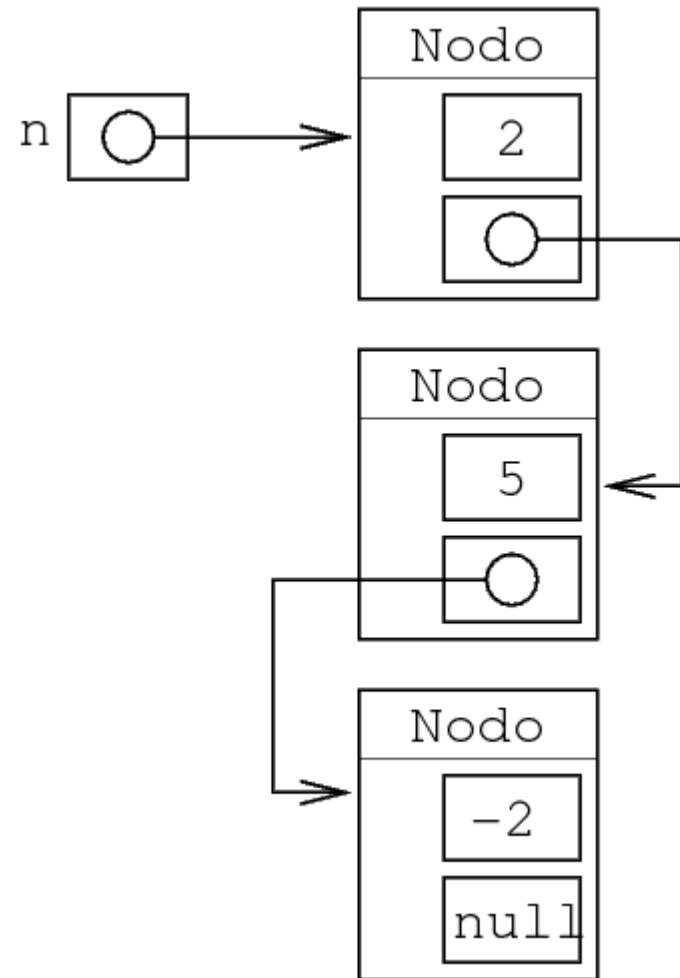
- Una **lista** è una struttura dati **dinamica**
 - Gli elementi sono memorizzati *sequenzialmente*
 - Possono essere aggiunti, modificati o rimossi arbitrariamente



- Ogni **nodo** della lista è solitamente caratterizzato da:
 - Un campo, contenente un **elemento** di un certo tipo
 - In realtà potrei avere più campi, contenenti diversi elementi
 - Uno o due campi contenenti un **riferimento** al nodo successivo o precedente
 - A seconda che la lista sia *singly-linked* o *doubly-linked*

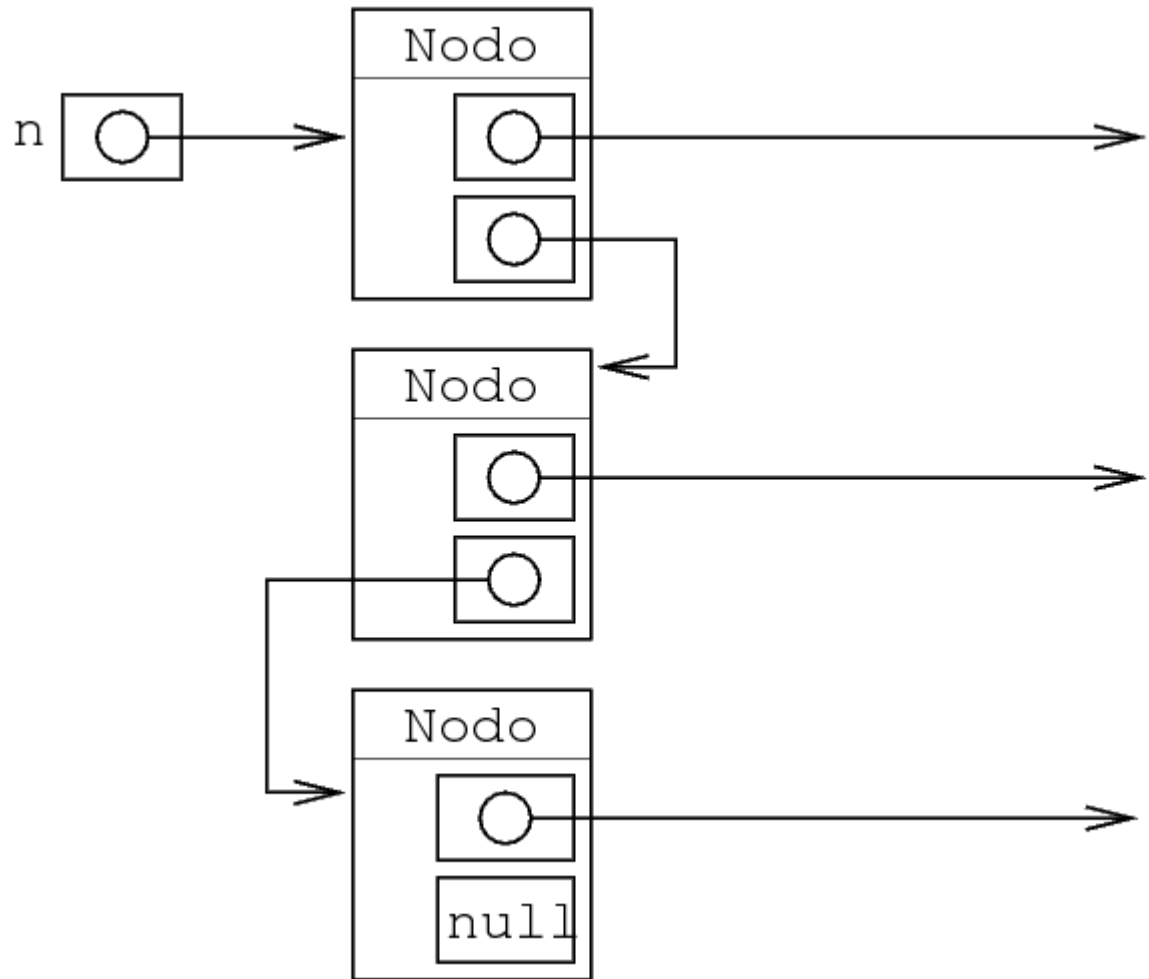
Lista

```
class Nodo {  
    int info;  
    Nodo next;  
}
```



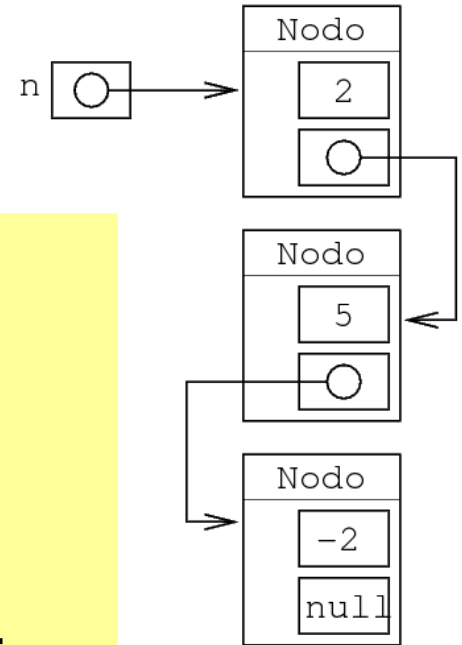
Lista

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



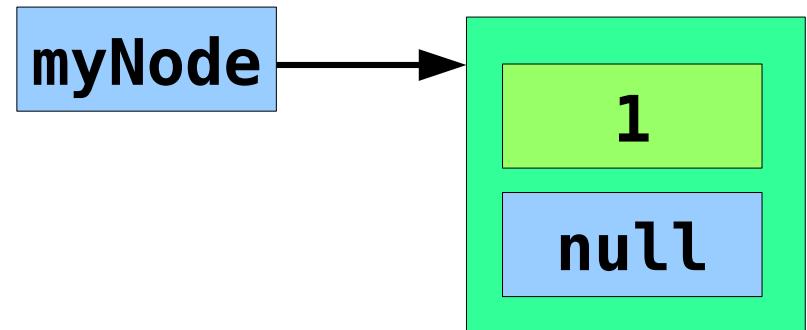
Lista

```
private class Nodo {  
    private int dati;  
    private Nodo next;  
    public Nodo(int valore,  
                Nodo collegamento) {  
        dati = valore;  
        next = collegamento;  
    }  
}
```



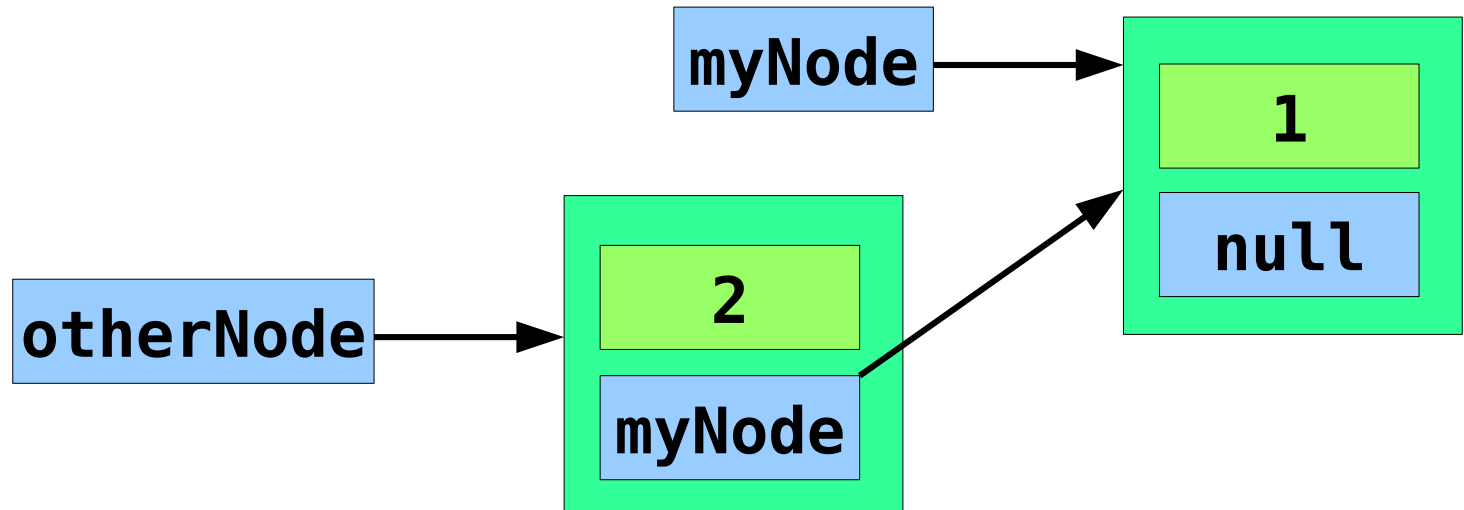
Creare nodi

```
Nodo myNode = new Nodo("1", null);
```



Creare nodi

```
Nodo myNode = new Nodo("1", null);  
Nodo otherNode = new Nodo("2", myNode);
```

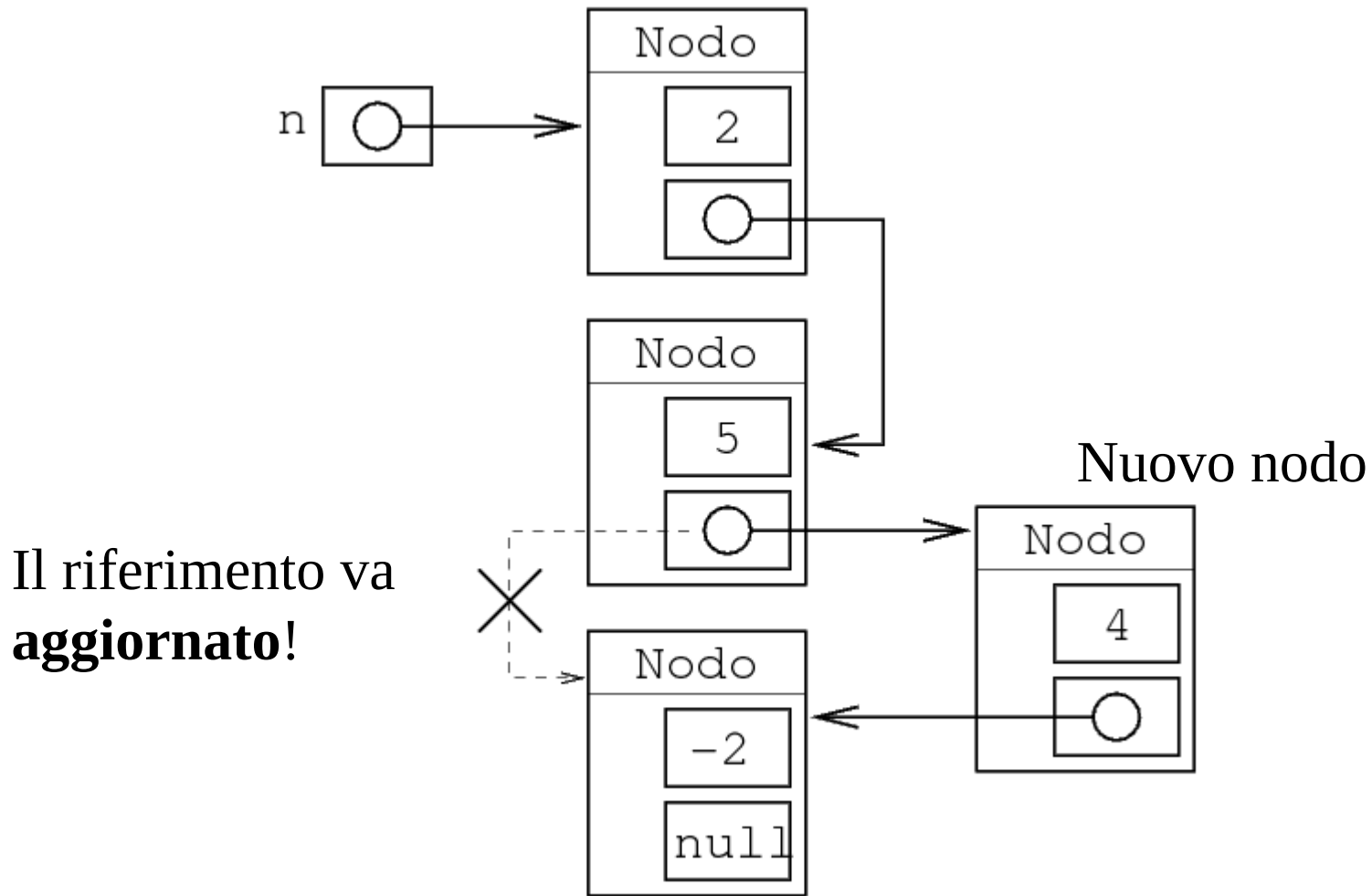


- Se ho N elementi, diventa pesante aggiungere elementi in questo modo...

Operazioni su liste

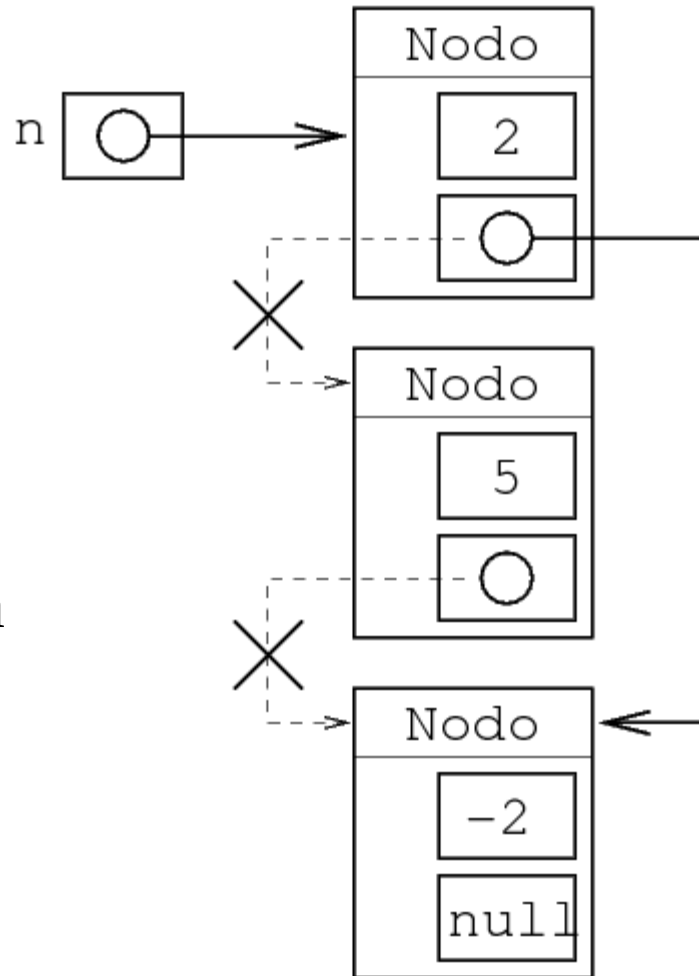
- Tipiche **operazioni** su una lista sono:
 - verificare se è **vuota**
 - trovare la **lunghezza**
 - **ritornare** l'elemento in una certa posizione
 - **modificare** l'elemento in una certa posizione
 - **inserire** un elemento in una certa posizione
 - **eliminare** l'elemento in una certa posizione
- Per alcune di esse usare un array è sicuramente una scelta migliore
 - *Quali?*

Inserimento



Cancellazione

Il riferimento va
aggiornato!



Liste in Java

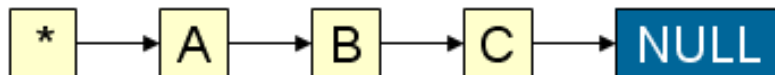
- LinkedList
 - Realizza liste *doubly-linked*
 - Il tipo degli elementi è *parametrico*
- ArrayList
- Vector
- ...

Tipi di liste

A few basic types of Linked Lists

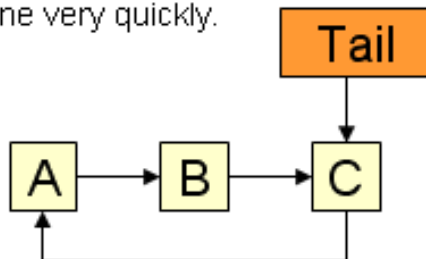
Singly Linked List

Root node links one way through all the nodes. Last node links to null.



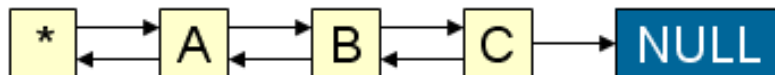
Circular Linked List

Circular linked lists have a reference to one node which is the tail node and all the nodes are linked together in one direction forming a circle. The benefit of using circular lists is that appending to the end can be done very quickly.

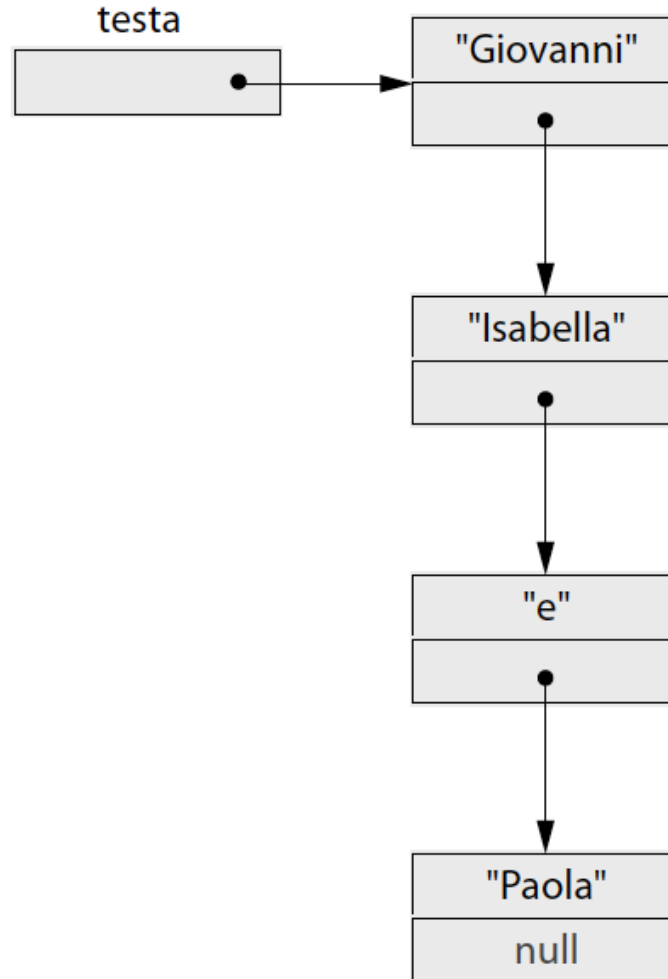


Doubly Linked List

Every node stores a reference to its previous node as well as its next. This is good if you need to move back by a few nodes and don't want to run from the beginning of the list.



Singly-linked list



Implementazione

```
public class NodoLista {  
    private String dati;  
    private NodoLista collegamento;
```

```
    public NodoLista() {  
        collegamento = null;  
        dati = null;  
    }
```

```
    public NodoLista(String valoreDati, NodoLista valoreCollegamento) {  
        dati = valoreDati;  
        collegamento = valoreCollegamento;  
    }
```

```
    public void setDati(String nuoviDati) {  
        dati = nuoviDati;  
    }
```

Lista concatenata di stringhe
(1/2)

Implementazione

Lista concatenata di stringhe
(2/2)

```
public String getDati() {  
    return dati;  
}  
  
public void setCollegamento(NodoLista nuovoCollegamento) {  
    collegamento = nuovoCollegamento;  
}  
  
public NodoLista getCollegamento() {  
    return collegamento;  
}  
}
```

```
public class ListaConcatenataDiStringhe {  
    private NodoLista testa;
```

```
  
    public ListaConcatenataDiStringhe() {  
        testa = null;  
    }  
  
    /**  
    Mostra i dati della lista.  
    */  
    public void mostraLista() {  
        NodoLista posizione = testa;  
        while (posizione != null) {  
            System.out.println(posizione.getDati());  
            posizione = posizione.getCollegamento();  
        }  
    }  
  
    /**  
    Restituisce il numero di nodi che compongono la lista.  
    */  
    public int lunghezza() {  
        int conteggio = 0;  
        NodoLista posizione = testa;  
        while (posizione != null) {  
            conteggio++;  
            posizione = posizione.getCollegamento();  
        }  
        return conteggio;  
    }  
}
```

testa è un rif. al primo
elemento della lista

Potrebbe essere meglio
avere un campo apposito...

Implementazione

```
/**
Aggiunge all'inizio della lista
un nodo contenente datiDaAggiungere.
*/
public void aggiungiNodoInTesta(String datiDaAggiungere) {
    testa = new NodoLista(datiDaAggiungere, testa);
}

/**
Elimina il primo nodo della lista.
*/
public void eliminaNodoDiTesta() {
    if (testa != null)
        testa = testa.getCollegamento();
    else {
        System.out.println("Si sta eliminando da una lista vuota.");
        System.exit(0);
    }
}
```

Implementazione

```
/**
Verifica se elemento è nella lista.
*/
public boolean nellaLista(String elemento) {
    return trova(elemento) != null;
}

// Restituisce un riferimento al primo nodo che contiene elemento.
// Se elemento non è nella lista, restituisce null.
private NodoLista trova(String elemento) {
    boolean trovato = false;
    NodoLista posizione = testa;
    while ((posizione != null) && !trovato) {
        String datiAllaPosizione = posizione.getDati();
        if (datiAllaPosizione.equals(elemento))
            trovato = true;
        else
            posizione = posizione.getCollegamento();
    }
    return posizione;
}
}
```

Serve davvero una variabile “trovato”?

Esercizio

- Il metodo **trova (String)** può essere realizzato da una funzione **ricorsiva**

- Meno efficiente, più compatto ed elegante

$$\text{trova}(x, L) = \begin{cases} \text{null} & \text{se } L = [] \\ \text{rif}(x) & \text{se } x = \text{head}(L) \\ \text{trova}(x, R) & \text{se } x \neq \text{head}(L) \end{cases}$$

- Provate a implementarlo per esercizio

Esempio (1/2)

```
public class ListaConcatenataDiStringheDemo {  
  
    public static void main(String[] args) {  
        ListaConcatenataDiStringhe lista = new ListaConcatenataDiStringhe();  
        lista.aggiungiNodoInTesta("Uno");  
        lista.aggiungiNodoInTesta("Due");  
        lista.aggiungiNodoInTesta("Tre");  
        System.out.println("La lista ha " + lista.lunghezza() + " elementi.");  
        lista.mostraLista();  
  
        if (lista.nellaLista("Tre"))  
            System.out.println("Tre e' sulla lista.");  
        else  
            System.out.println("Tre NON e' sulla lista.");  
        lista.eliminaNodoDiTesta();  
  
        if (lista.nellaLista("Tre"))  
            System.out.println("Tre e' sulla lista.");  
        else  
            System.out.println("Tre NON e' sulla lista.");  
    }  
}
```

Esempio (2/2)

```
        lista.eliminaNodoDiTesta();  
        lista.eliminaNodoDiTesta();  
        System.out.println("Inizio della lista:");  
        lista.mostraLista();  
        System.out.println("Fine della lista.");  
    }  
}
```

Esempio di output

La lista ha 3 elementi.

Tre

Due

Uno

Tre e' sulla lista.

Tre NON e' sulla lista.

Inizio della lista:

Fine della lista.

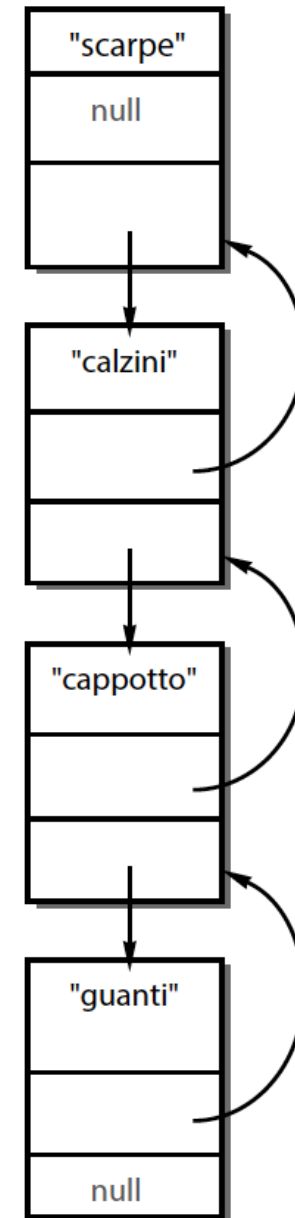
Doubly-linked lists

In una lista **doubly-linked** (doppiamente concatenata) per ogni nodo ci sono 2 rif.:

- Al nodo **precedente** (*se esiste*)
- Al nodo **successivo** (*se esiste*)

Oltre al rif. alla testa della lista (*head*), può essere utile avere un rif. alla **coda** (*tail*)

- Cioè un rif. all'ultimo elemento

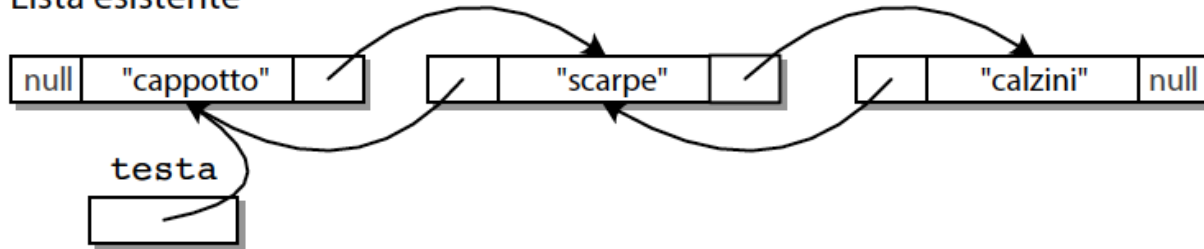


Nodi

```
public class NodoListaDoppia {  
    public String dati;  
    public NodoListaDoppia precedente;  
    public NodoListaDoppia successivo;  
  
    public NodoListaDoppia(String nuoviDati,  
                             NodoListaDoppia nodoPrecedente,  
                             NodoListaDoppia nodoSuccessivo) {  
        dati = nuoviDati;  
        successivo = nodoSuccessivo;  
        precedente = nodoPrecedente;  
    }  
}
```

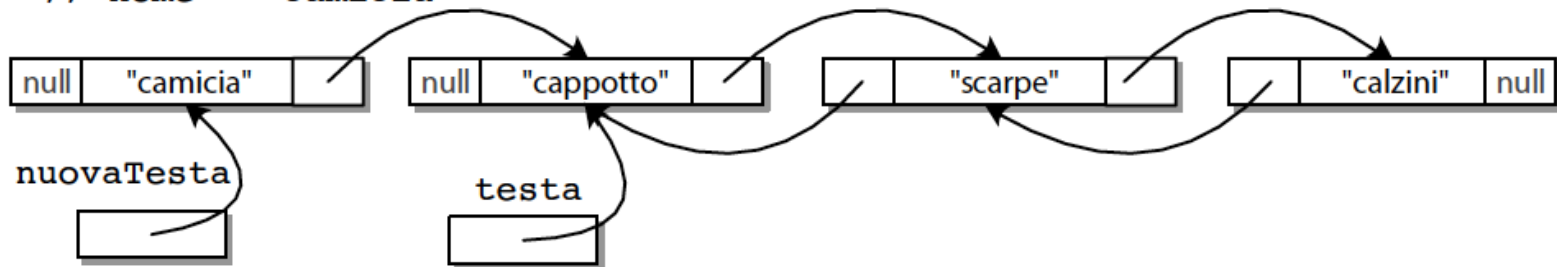
Inserire nodo in testa

1. Lista esistente



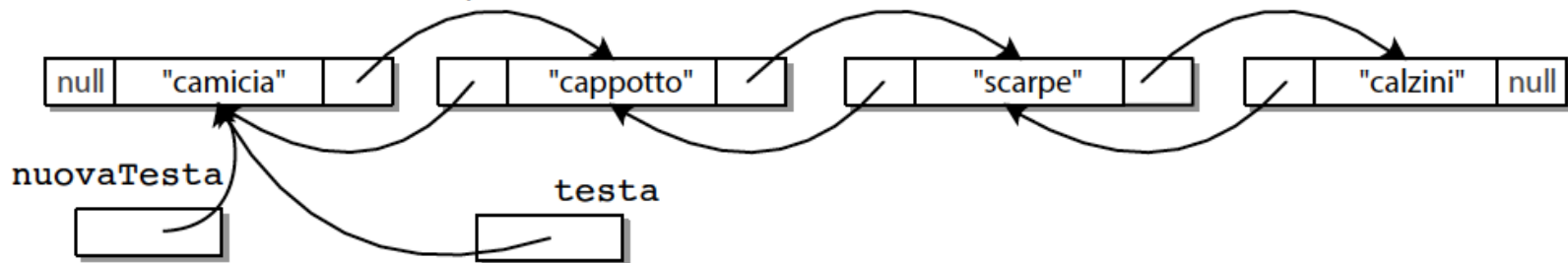
2. Creare un nuovo NodoListaDoppia collegato a "cappotto"

```
NodoListaDoppia nuovaTesta = new NodoListaDoppia(nome, null, testa)  
// nome = "camicia"
```



3. Impostare il collegamento all'indietro e la nuova testa

```
testa.precedente = nuovaTesta;  
testa = nuovaTesta;
```



Esercizi

- Implementare un metodo per **eliminare** un nodo da una lista doubly-linked, scorrendo gli elementi dalla testa alla coda
 - Implementare anche la versione che scorre la lista in senso opposto
- Implementare un metodo per verificare se una lista doubly-linked di interi è **palindroma**
 - Es. [1, 2, 5, 2, 1] è palindroma, [1, 2, 2, -1] no

Pila

```
public class Pila {  
    private NodoLista testa;  
  
    public Pila() {  
        testa = null;  
    }  
  
    /**  
    Questo metodo sostituisce aggiungiNodoInTesta  
    */  
    public void push(String nuoviDati) {  
        testa = new NodoLista(nuoviDati, testa);  
    }  
}
```

La **pila** (*stack*) è una struttura dati che permette di inserire/rimuovere nodi solo in testa

- Logica **LIFO**: *Last In, First Out*

Si può implementare con una lista

Pila

```
/**
Questo metodo sostituisce eliminaNodoDiTesta
e restituisce la stringa in cima alla pila
*/
public String pop() {
    String dati = "";

    if (testa != null) {
        dati = testa.getDati();
        testa = testa.getCollegamento();
    }
    return dati;
}


public boolean vuota() {
    return (testa == null);
}
}
```

Fare una pop() su una pila vuota non è desiderabile...

Pila

```
public class EsempioPila {  
    public static void main(String args[]) {  
        Pila pila = new Pila();  
        pila.push("Alessandro");  
        pila.push("Federico");  
        pila.push("Marta");  
  
        while (!pila.vuota()) {  
            String s = pila.pop();  
            System.out.println(s);  
        }  
    }  
}
```

Gli elementi vengono rimossi dalla pila in ordine inverso rispetto a quello nel quale erano stati inseriti.



Esempio di output

Marta
Federico
Alessandra

Esercizio

- Implementare e testare la struttura dati **coda** (*queue*) simile alla pila ma con logica **FIFO** (*First in, First Out*)
 - L'operazione push inserisce l'elemento in coda
 - L'operazione pop rimuove l'elemento in testa

Riferimenti

- Lucidi del Libro di Riferimento
- <http://java.sun.com/>