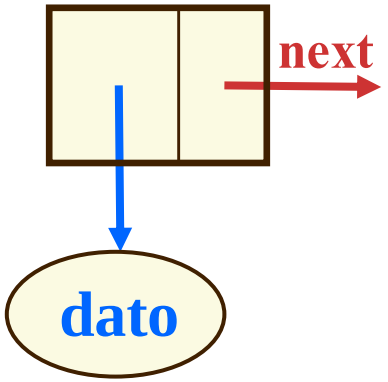
Linked list

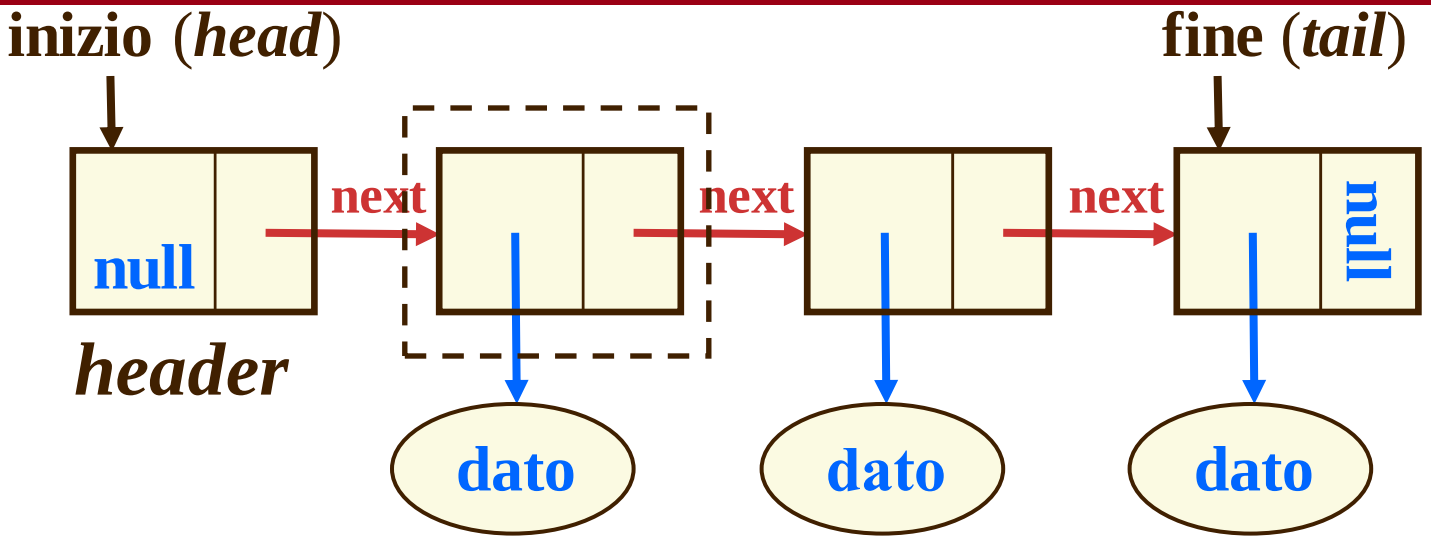
Linked List

- La ***lista concatenata*** (***linked list***) non è un nuovo ADT, ma è una struttura dati alternativa all'array per la realizzazione di ADT



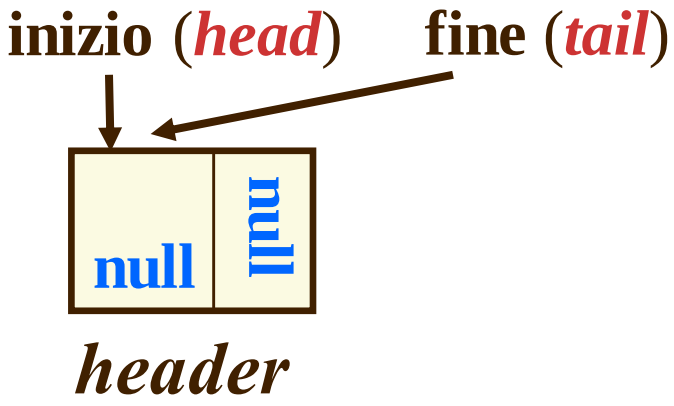
- Una lista concatenata è un insieme **ordinato** di nodi
  - ogni nodo è un oggetto che contiene
    - un riferimento ad un elemento (il dato)
    - un riferimento al nodo successivo nella lista concatenata (next)

Linked List



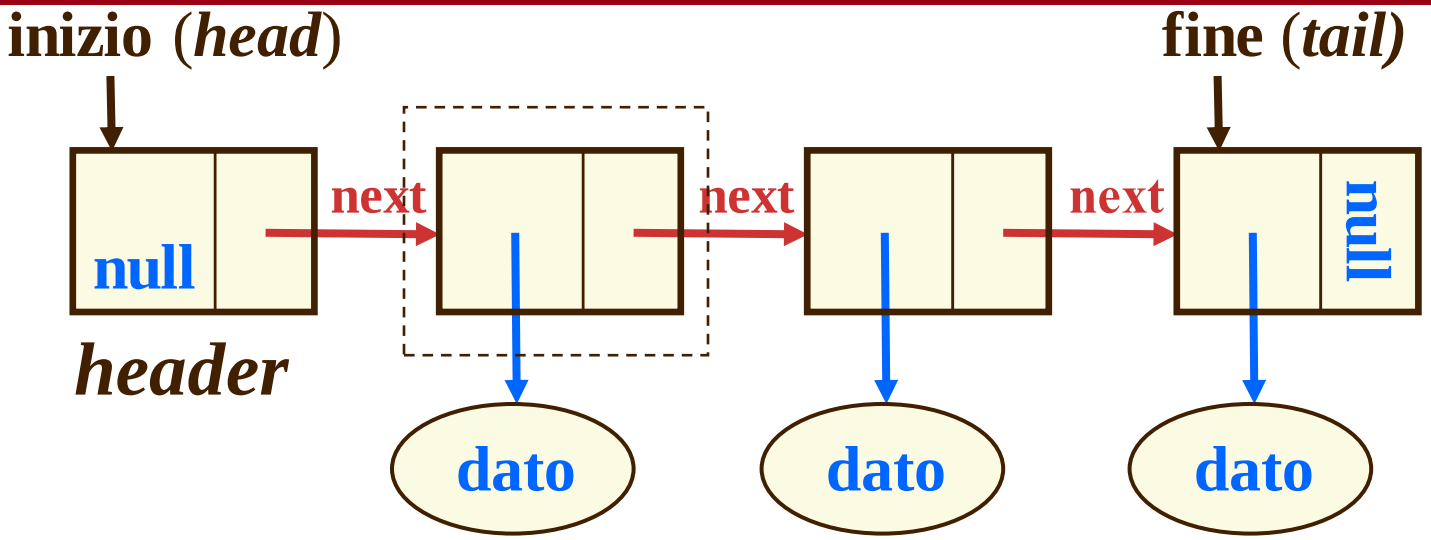
- Per agire sulla lista concatenata è sufficiente memorizzare il ***riferimento al suo primo nodo (head)***
  - è comodo avere anche un riferimento all'ultimo nodo (tail)
- Il campo **next** dell'ultimo nodo contiene **null**
- Vedremo che è comodo avere un primo nodo senza dati, chiamato ***header***

Lista vuota



- Per capire bene il funzionamento della lista concatenata, è necessario avere ben chiara la rappresentazione della **lista vuota**
  - contiene il solo nodo ***header***, che ha **null** in entrambi i suoi campi
  - head e tail puntano entrambi a tale ***header***

Accedere in sequenza ai nodi



- Per accedere in sequenza a tutti i nodi della lista concatenata si parte dal riferimento **inizio** e si seguono i riferimenti contenuti nel campo **next** di ciascun nodo
  - non è possibile scorrere la lista in senso inverso
  - la scansione termina quando si trova il nodo con il valore **null** nel campo **next**

nodo di una Linked list

```
public class ListNode
{   private Object element;
    private ListNode next; //stranezza

    public ListNode(Object e, ListNode n)
    {   element = e; next = n; }

    public ListNode()
    {   element = null; next = null); }

    public Object getElement()
    {   return element; }
    public ListNode getNext()
    {   return next; }

    public void setElement(Object e)
    {   element = e; }
    public void setNext(ListNode n)
    {   next = n; }
}
```

Auto-riferimento

```
public class ListNode
{   ...
    private ListNode next; //stranezza
}
```

- Nella definizione della classe **ListNode** notiamo una "stranezza"
  - la classe definisce e usa ***riferimenti ad oggetti del tipo che sta definendo***
- Ciò è perfettamente lecito e si usa molto spesso quando si rappresentano "strutture a definizione ricorsiva" come la lista concatenata

ListNode: incapsulamento eccessivo?

- A cosa serve l'incapsulamento in classi che hanno lo stato completamente accessibile tramite metodi?
  - ***apparentemente a niente...***
- Supponiamo di essere in fase di debugging e di aver bisogno della visualizzazione di un messaggio ogni volta che viene modificato il valore di una variabile di un nodo
  - se non abbiamo usato l'incapsulamento, occorre aggiungere enunciati in tutti i punti del codice dove vengono usati i nodi…
  - elevata probabilità di errori o dimenticanze

ListNode: incapsulamento eccessivo?

- Se invece usiamo l'incapsulamento
  - è sufficiente inserire l'enunciato di visualizzazione all'interno dei metodi che interessano: ad esempio set()
    - le variabili di esemplare possono essere modificate ***SOLTANTO*** mediante l'invocazione del corrispondente metodo **set()**
  - terminato il debugging, per eliminare le visualizzazioni è sufficiente modificare il solo metodo **set()**, senza modificare di nuovo moltissime linee di codice

linked list
metodi utili

Metodi di Linked List

- I metodi utili per una lista concatenata sono
  - **addFirst** per inserire un oggetto all'inizio della catena
  - **addLast** per inserire un oggetto alla fine della catena
  - **removeFirst** per eliminare il primo oggetto della catena
  - **removeLast** per eliminare l'ultimo oggetto della catena
- Spesso si aggiungono anche i metodi
  - **getFirst** per esaminare il primo oggetto
  - **getLast** per esaminare l'ultimo oggetto
- Non vengono mai restituiti né ricevuti riferimenti ai ***nodi***, ma sempre ai ***dati*** contenuti nei nodi

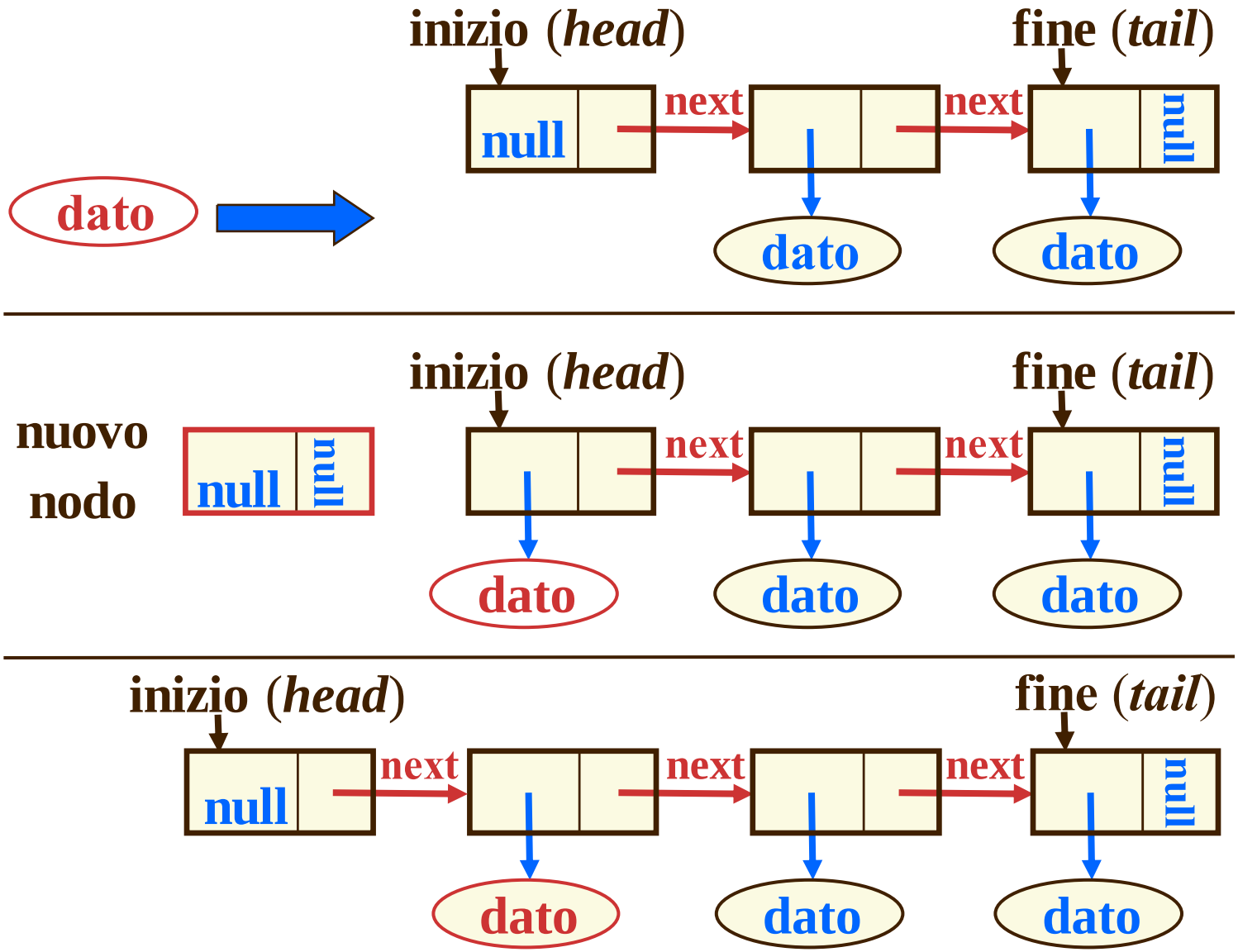
Metodi di Linked List

- Infine, dato che anche la lista concatenata è un contenitore, ci sono i metodi
  - **isEmpty** per sapere se la lista concatenata è vuota
  - **makeEmpty** per rendere vuota la lista concatenata
- Si definisce l'eccezione **EmptyLinkedListException**
- Si noti che, non essendo la lista concatenata un ADT, non viene definita un'interfaccia
  - La lista concatenata non è un ADT perché nella sua definizione abbiamo esplicitamente indicato COME la struttura dati deve essere realizzata, e non semplicemente il suo comportamento

Linked list

```
public class LinkedList implements Container
{  private ListNode head, tail;
   public LinkedList()
   {  makeEmpty(); }

   public void makeEmpty()
   {  head = tail = new ListNode(); }
   public boolean isEmpty()
   {  return (head == tail); }

   public Object getFirst() // operazione O(1)
   {  if (isEmpty())
         throw new EmptyLinkedListException();
      return head.getNext().getElement(); }

   public Object getLast()  // operazione O(1)
   {  if (isEmpty())
         throw new EmptyLinkedListException();
      return tail.getElement();
   }
    ...
}
```

addFirst


inizio (*head*)
fine (*tail*)
null
next
next
null
dato
dato
dato
nuovo
nodo
null
null
inizio (*head*)
fine (*tail*)
next
next
null
dato
dato
dato
inizio (*head*)
fine (*tail*)
null
next
next
next
null
dato
dato
dato

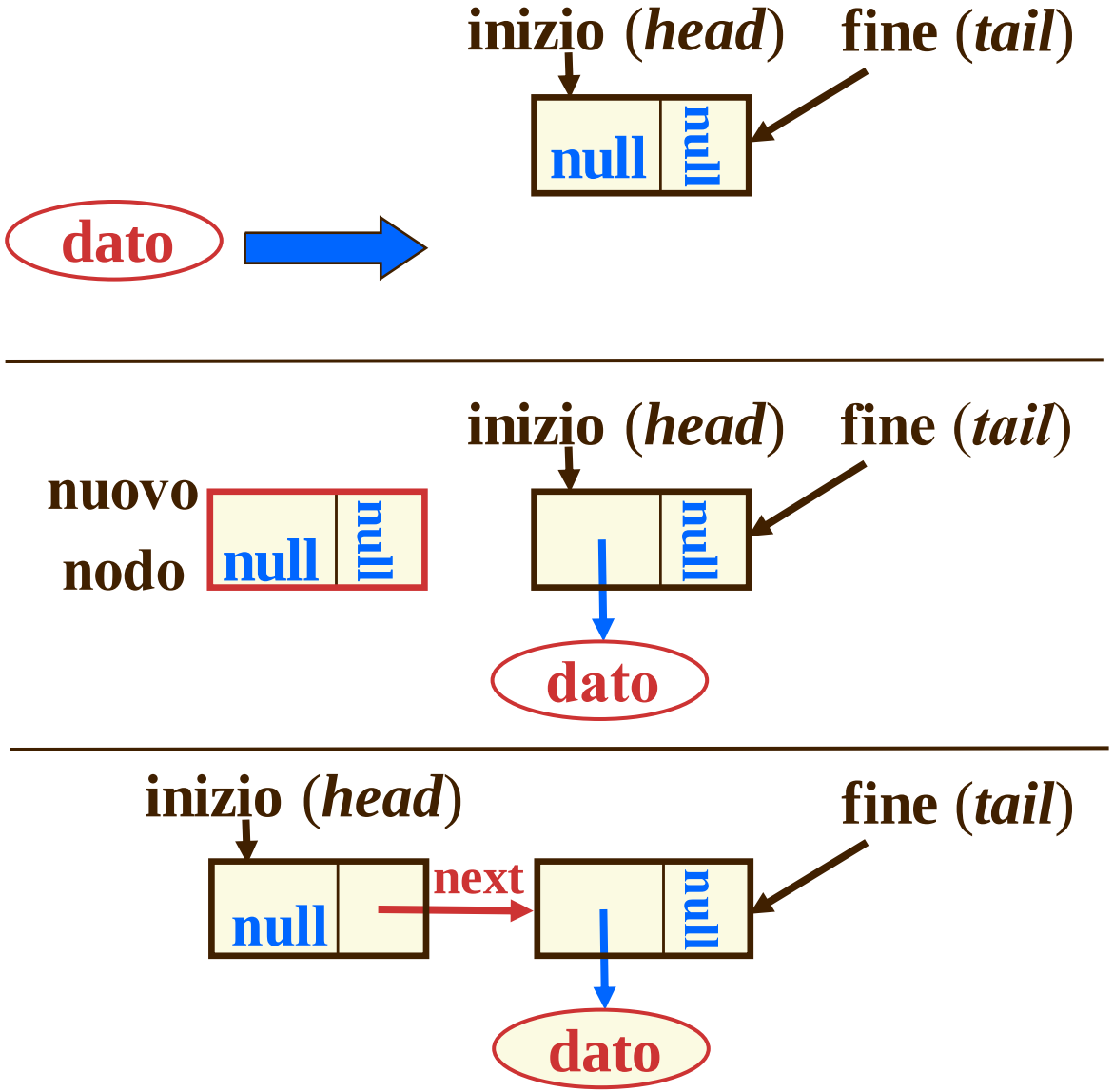
addFirst

```
public class LinkedList ...
{   ...
    public void addFirst(Object e) {
        // inserisco il dato nell'header attuale
        head.setElement(e);
        // creo un nuovo nodo con due riferimenti null
        ListNode n = new ListNode();
        // collego il nuovo nodo all'header attuale
        n.setNext(head);
        // il nuovo nodo diventa l'header della linked list
        head = n;
        // tail non viene modificato
    }
}
```

- Non esiste il problema di "lista concatenata piena"
- L'operazione è **O(1)**

addFirst

- Verifichiamo che tutto sia corretto anche inserendo in una **lista *catenata vuota***
- ***Fare sempre attenzione ai casi limite***



addFirst

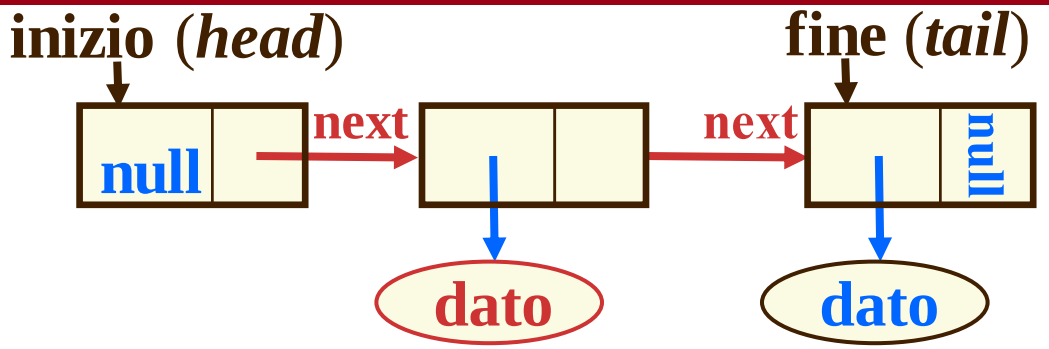
```
public void addFirst(Object e) {
    head.setElement(e);
    ListNode n = new ListNode();
    n.setNext(head);
    head = n;
}
```

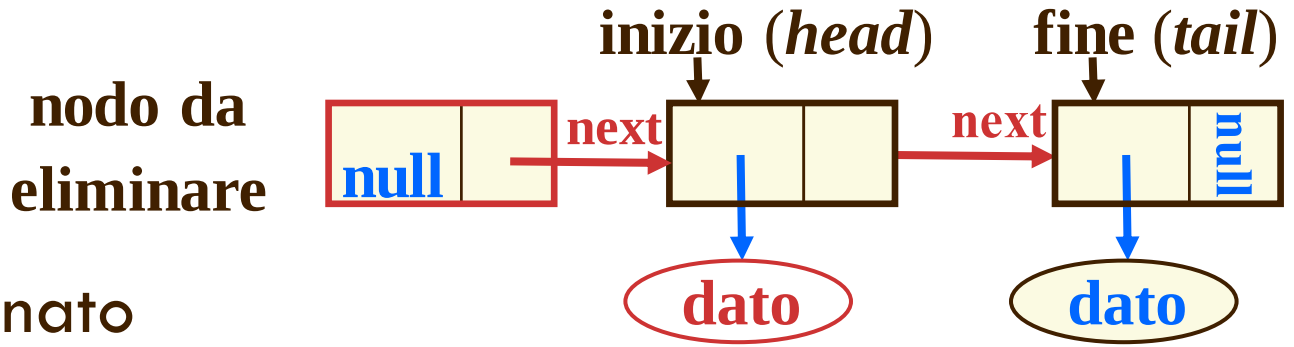
- Il codice di questo metodo si può esprimere anche in modo più conciso

```
public void addFirst(Object e) {
    head.setElement(e);
    // funziona perché prima head viene USATO
    // (a destra) e solo successivamente viene
    // MODIFICATO (a sinistra)
    head = new ListNode(null, head);
}
```

- È più "professionale", anche se meno leggibile

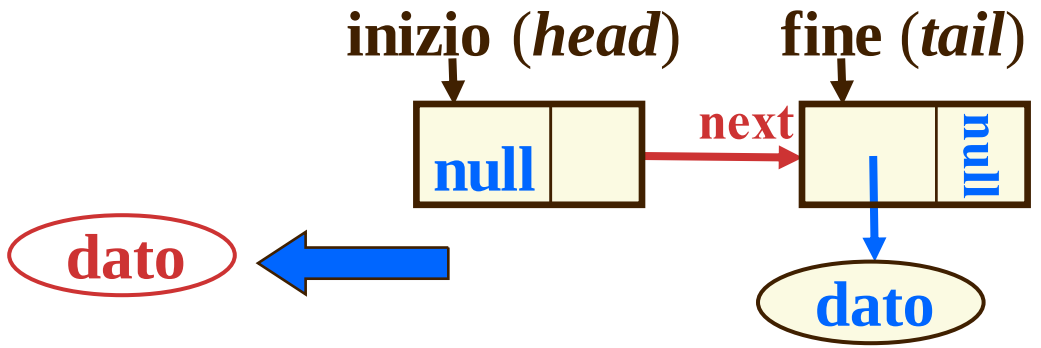
removeFirst





Il nodo viene eliminato

dal garbage collector



Ciclo di vita degli oggetti

- **Un oggetto viene eliminato dalla JVM quando non esiste più alcuna variabile oggetto che faccia riferimento ad esso**
  - La zona di memoria riservata all'oggetto viene "riciclata", cioè resa di nuovo libera, dal ***raccoglitore di rifiuti*** (***garbage collector***) della JVM, che controlla periodicamente e automaticamente se ci sono oggetti da eliminare
- **In Java non si possono eliminare esplicitamente oggetti!**
  - È sufficiente "abbandonarli", cioè cancellare ogni riferimento
- Dopo che un oggetto è stato eliminato, lo spazio di memoria che occupava può essere riutilizzato per altri oggetti

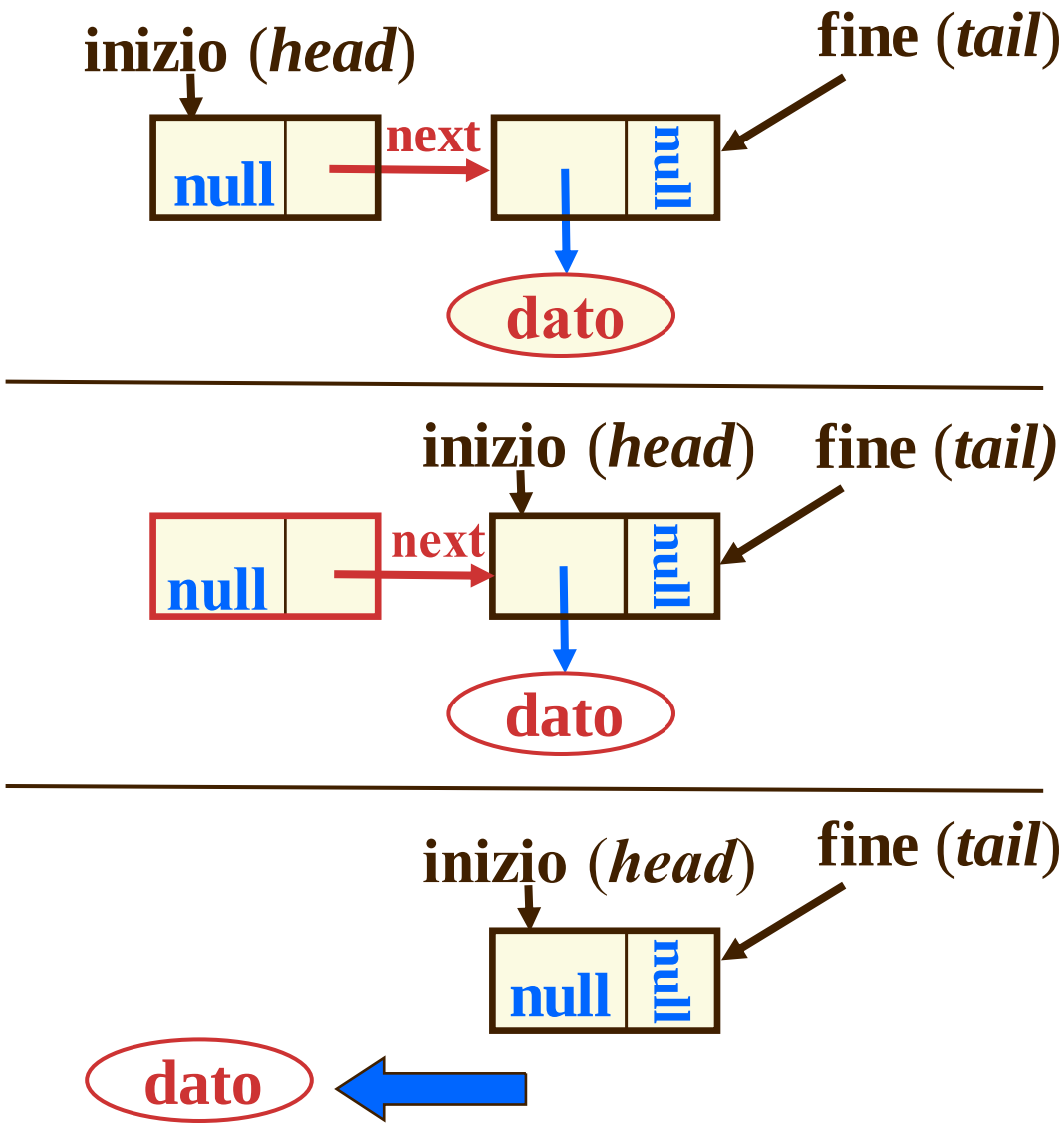
removeFirst

```
public class LinkedList ...
{   ...
    public Object removeFirst() {
        // delega a getFirst il
        // controllo di lista vuota
        Object e = getFirst();
        // aggiorno l'header
        head = head.getNext();
        head.setElement(null);
        return e;
    }
}
```

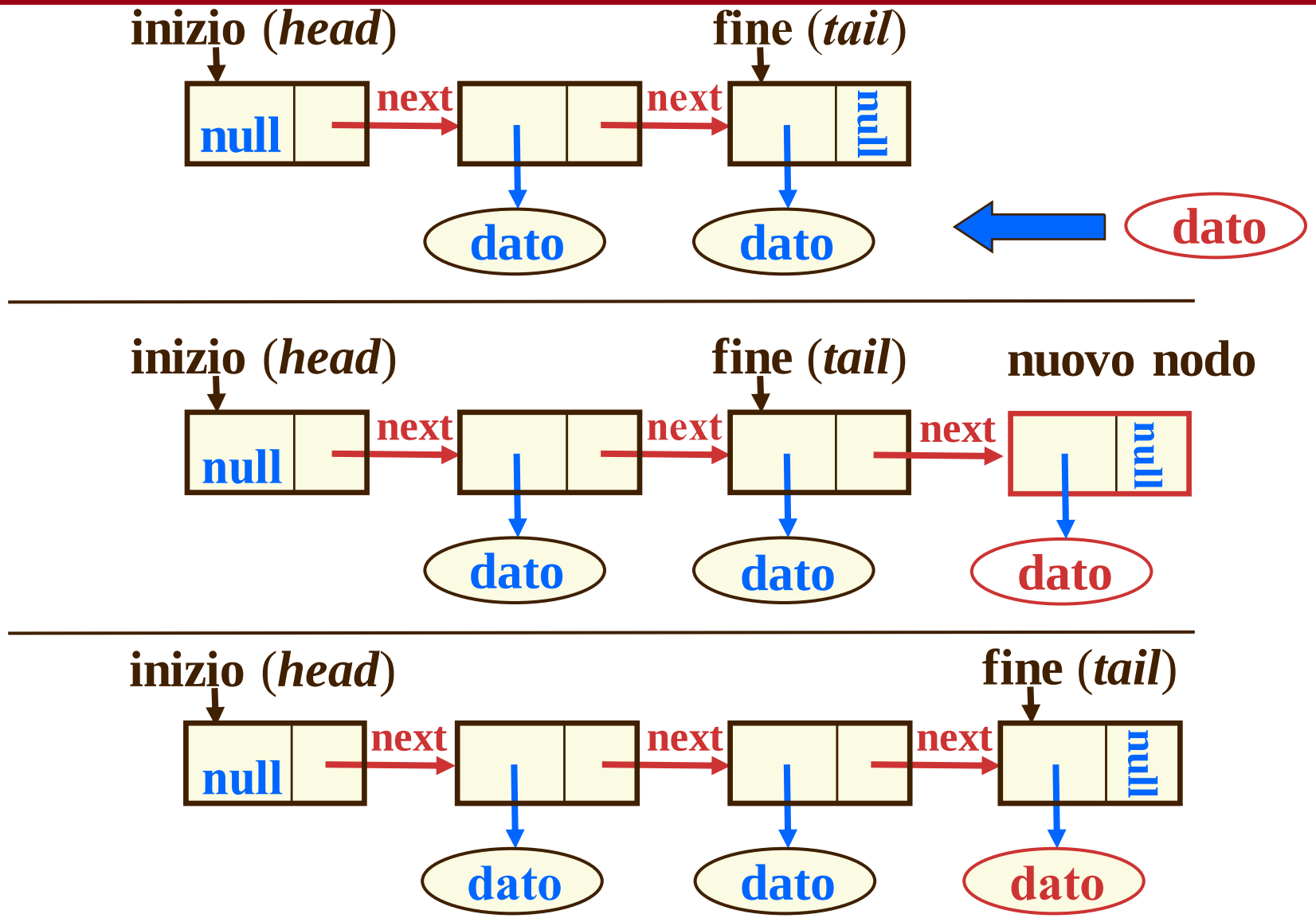
L'operazione è **O(1)**

removeFirst

- Verifichiamo che tutto sia corretto anche rimanendo con una *linked list vuota*
- *Fare sempre attenzione ai casi limite*



addLast


inizio (*head*)
fine (*tail*)
null
next
next
null
dato
dato
dato
inizio (*head*)
fine (*tail*)
nuovo nodo
null
next
next
next
null
dato
dato
dato
inizio (*head*)
fine (*tail*)
null
next
next
next
null
dato
dato
dato

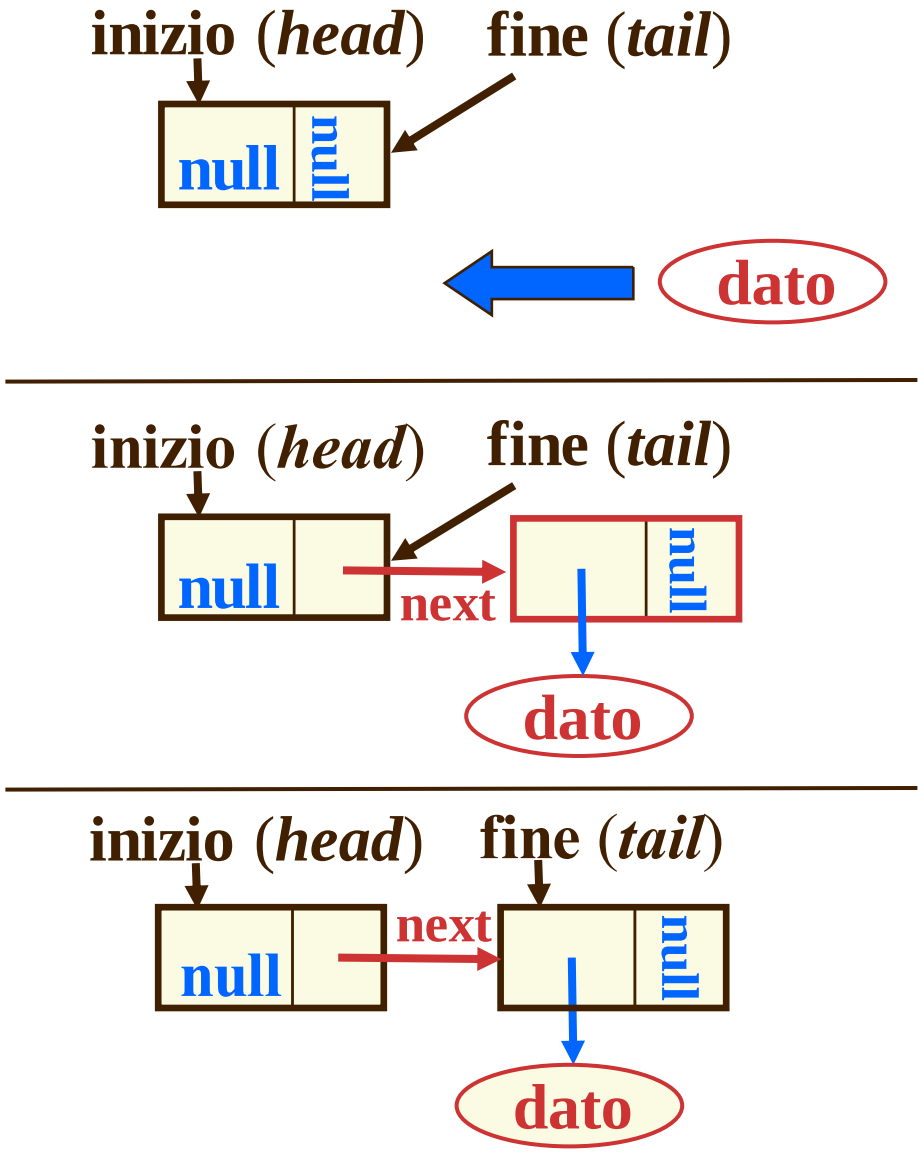
addLast

```
public class LinkedList ...
{   ...
    public void addLast(Object e) {
        tail.setNext(new ListNode(e, null));
        tail = tail.getNext();
    }
}
```

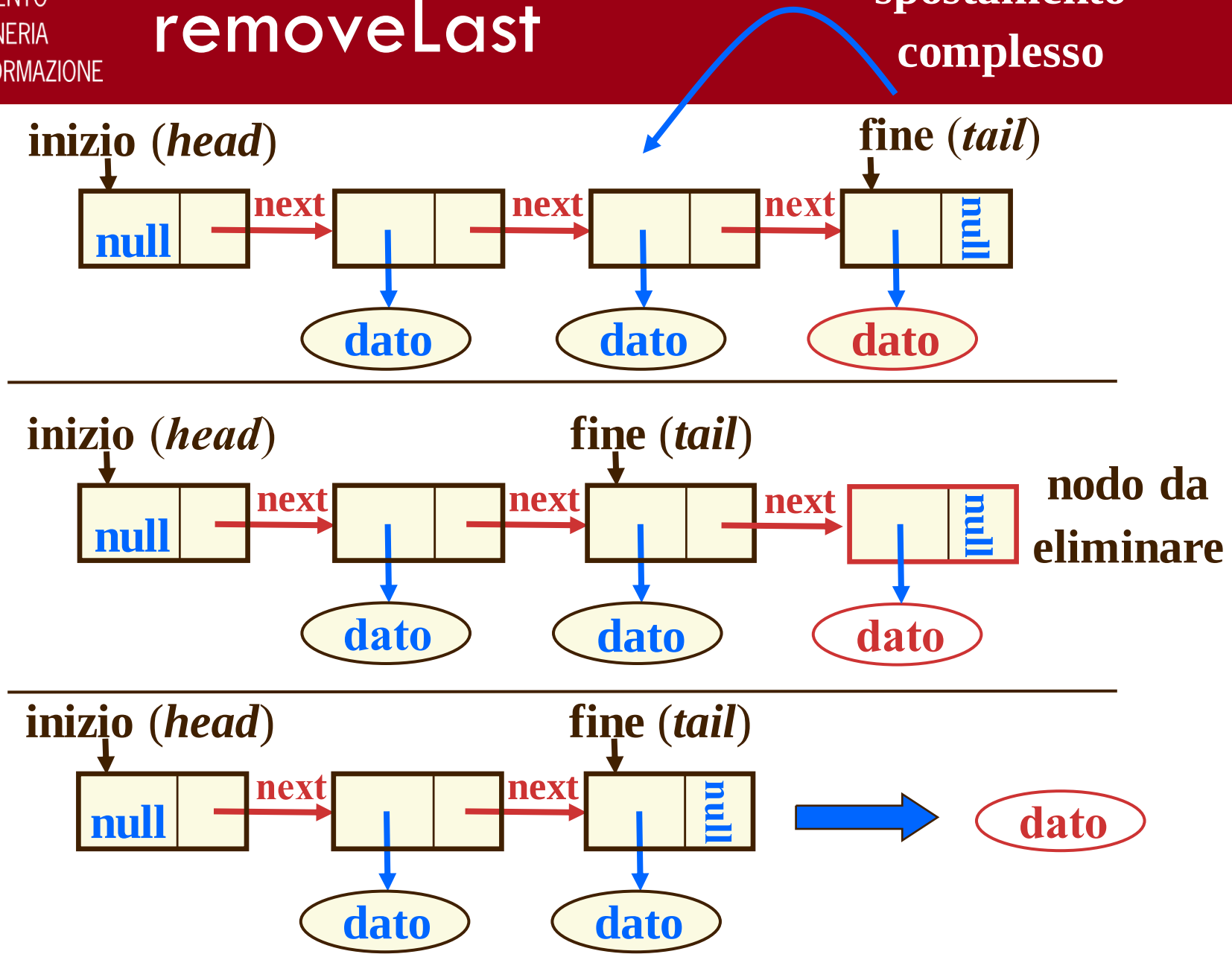
- Non esiste il problema di "lista concatenata piena"
- Anche questa operazione è **O(1)**

addLast

- Verifichiamo che tutto sia corretto anche inserendo in una *lista concatenata vuota*
- *Fare sempre attenzione ai casi limite*



removeLast


spostamento
complesso
inizio (*head*)
fine (*tail*)
null
next
next
next
null
dato
dato
dato
inizio (*head*)
fine (*tail*)
null
next
next
next
null
nodo da
eliminare
dato
dato
dato
inizio (*head*)
fine (*tail*)
null
next
next
null
dato
dato
dato

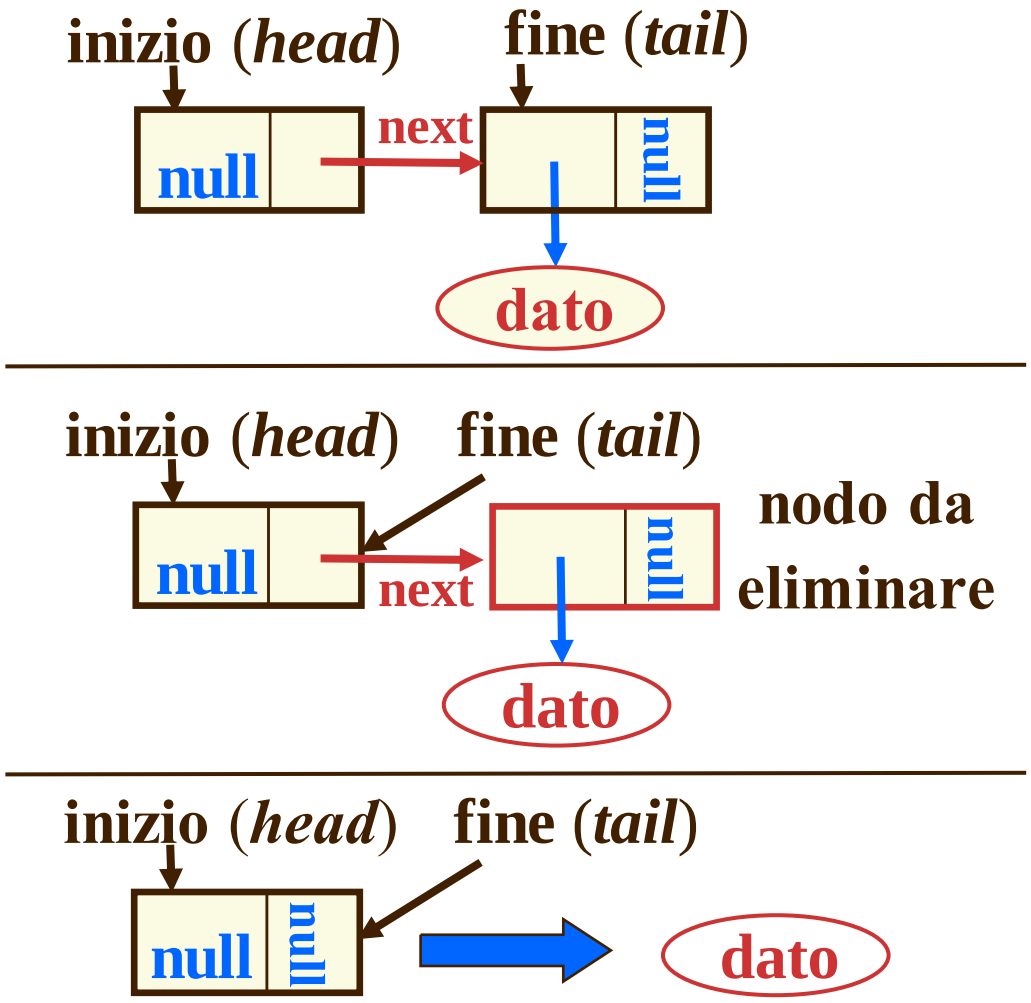
removeLast

```
public class LinkedList ...
{   ...
    public Object removeLast() {
        Object e = getLast();
        // bisogna cercare il penultimo nodo
        // partendo dall'inizio e andando avanti
        // finché non si arriva alla fine della catena
        ListNode temp = head;
        while (temp.getNext() != tail)
            temp = temp.getNext();
        // a questo punto temp si riferisce al
        // penultimo nodo
        tail = temp;
        tail.setNext(null);
        return e;
    }
}
```

"Purtroppo" questa operazione è **O(n)**

removeLast

- Verifichiamo che tutto sia corretto anche rimanendo con una *lista concatenata vuota*
- *Fare sempre attenzione ai casi limite*



Header della linked list

- La presenza del nodo ***header*** nella lista concatenata ne rende più semplici i metodi
  - in questo modo, non è necessario gestire i casi limite in modo diverso dalle situazioni ordinarie
- Senza usare il nodo ***header***, le prestazioni asintotiche rimangono comunque le stesse
- Usando il nodo ***header*** si "spreca" un nodo
  - per valori elevati del numero di dati nella lista concatenata questo spreco, in percentuale, è trascurabile

Prestazioni della Linked List

- Tutte le operazioni sulla ***lista concatenata*** sono **O(1)** tranne **removeLast** che è **O(n)**
  - si potrebbe pensare di tenere un riferimento anche al ***penultimo*** nodo, ma per ***aggiornare*** tale riferimento sarebbe comunque necessario un tempo **O(n)**
- Se si usa una lista concatenata con il solo riferimento **head**, anche **addLast** diventa **O(n)**
  - per questo è utile usare il riferimento **tail**, che migliora le prestazioni di **addLast** senza peggiorare le altre e non richiede molto spazio di memoria

Prestazioni della linked list

- Vantaggi
  - Non esiste il problema di "lista concatenata piena"
    - non bisogna mai "ridimensionare" la catena
    - la JVM lancia l'eccezione **OutOfMemoryError** se viene esaurita la memoria disponibile
  - Non c'è spazio di memoria sprecato (come negli array "riempiti solo in parte")
- Svantaggi
  - un nodo occupa però più spazio di una cella di array, almeno il doppio (contiene due riferimenti anziché uno)

Osservazione

- Osserviamo che la classe ListNode, usata dalla lista concatenata, non viene usata al di fuori della lista concatenata stessa
  - la lista concatenata non restituisce mai riferimenti a ListNode
  - la lista concatenata non riceve mai riferimenti a ListNode

Commento

- Per il principio dell'incapsulamento (***information hiding***) sarebbe preferibile che questa classe e i suoi dettagli non fossero visibili all'esterno della lista concatenata
  - in questo modo una modifica della struttura interna della lista concatenata e/o di **ListNode** non avrebbe ripercussioni sul codice scritto da chi usa la lista concatenata

ListNode come classe interna

- Il linguaggio Java consente **di definire classi all'interno di un'altra classe**
  - tali classi si chiamano **classi interne** (**inner classes**)

- A noi interessa solo il fatto che se una classe interna viene definita **private**
  - essa è accessibile (in tutti i sensi) soltanto all'interno della classe in cui è definita
  - dall'esterno non è nemmeno possibile creare oggetti di tale classe interna

```
public class LinkedList ...
{   ...
    private class ListNode
    {   ... }
}
```

Iteratore in una lista concatenata

Esempio: conteggio di elementi

- Per contare gli elementi presenti in una lista concatenata è necessario scorrere tutta la lista concatenata

```
public class LinkedList ...
{   ...
    public int getSize()
    {   ListNode temp = head.getNext();
        int size = 0;
        while (temp != null)
        {   size++;
            temp = temp.getNext();
        }
        // osservare che size è zero
        // se la lista concatenata è vuota
        //(corretto)
        return size;
    }
}
```

Algoritmi per liste concatenate

- Osserviamo che per eseguire algoritmi sulla lista concatenata è necessario **aggiungere metodi all'interno della classe LinkedList**, che è l'unica ad avere accesso ai nodi della lista concatenata
  - esempio: un metodo che verifichi la presenza di un particolare oggetto nella lista concatenata (algoritmo di ricerca)

- Questo limita molto l'utilizzo della lista concatenata come struttura dati definita una volta per tutte…
  - vogliamo che la lista concatenata fornisca uno **strumento per accedere ordinatamente a tutti i suoi elementi**

Algoritmi per liste concatenate

- L'idea: fornire un metodo **getHead**

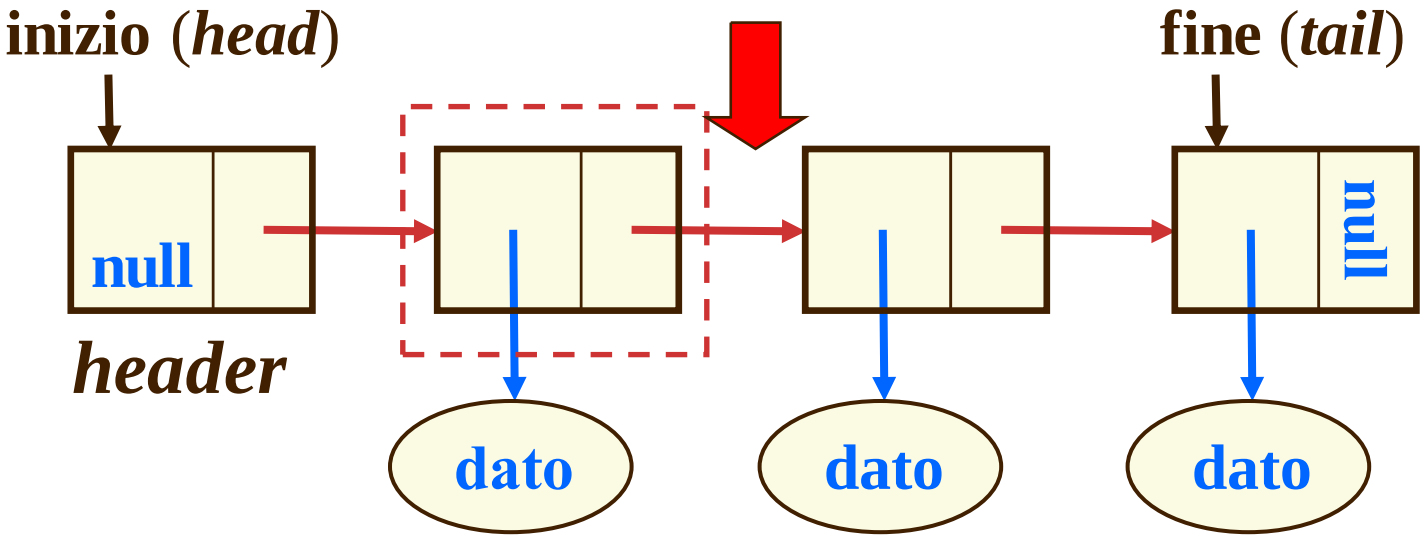
```
public class LinkedList ...
{   ...
    public ListNode getHead()
    {  return head; }
}
```

- Questo approccio non va bene:
  - Se la classe ListNode e' public in LinkedList si viola l'incapsulamento, perché diventa possibile modificare lo stato interno della lista concatenata dall'esterno
  - Se ListNode e' private **non funziona**, perché i nodi sono inacessibili e il metodo e' inutile

Iteratore in una lista concatenata

- La soluzione del problema è quella di fornire all'utilizzatore della lista concatenata uno strumento con cui interagire con la catena per scandire i suoi nodi
- Tale oggetto si chiama **iteratore** e ne definiamo prima di tutto il comportamento astratto (ADT)
  - un iteratore rappresenta in astratto il **concetto di posizione** all'interno di una lista concatenata
  - un iteratore si trova sempre DOPO un nodo e PRIMA del nodo successivo (che può non esistere se l'iteratore si trova dopo l'ultimo nodo)
    - all'inizio l'iteratore si trova dopo il nodo **header**

Iteratore in una lista concatenata



- Un iteratore rappresenta in astratto **il concetto di posizione** all'interno di una catena
  - la posizione è rappresentata concretamente da un riferimento ad un nodo (**il nodo precedente alla posizione dell'iteratore**)

Iteratore in una lista concatenata

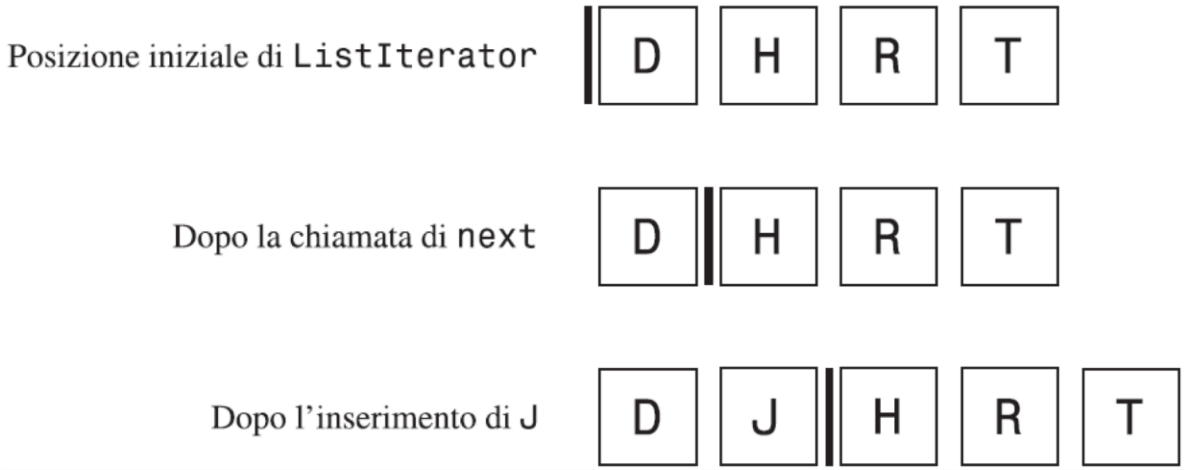
```
public interface ListIterator
{   // Funzionamento del costruttore:
    // quando viene costruito, l'iteratore si
    // trova nella prima posizione,
    // cioè DOPO il nodo header

    // se l'iteratore si trova alla fine della
    // lista concatenata, lancia NoSuchElementException,
    // altrimenti restituisce l'oggetto che si
    // trova nel nodo posto DOPO la posizione
    // attuale e sposta l'iteratore di una
    // posizione in avanti lungo la catena
    Object next();

    // verifica se è possibile invocare next()
    // senza che venga lanciata un'eccezione
    boolean hasNext();
    . . .
}
```

Iteratore in una lista concatenata

```
public interface ListIterator
{
     . . .

     /*   inserisce x prima della posizione attuale
     */
    void add(Object x);

    /* Lancia IllegalStateException se invocato 2 volte
     consecutive altrimenti elimina l'ultimo oggetto
     esaminato da next() o inserito da add()    */
    void remove();

}
```

ADT ListIterator

- Possiamo immaginare un iteratore come un cursore in un elaboratore di testi
  - Un nodo della lista concatenata corrisponde a un carattere
  - L'iteratore si trova sempre "tra due nodi", come un cursore



Iteratore in una lista concatenata

- A questo punto, è sufficiente che la lista concatenata fornisca un metodo per creare un iteratore

```
public class LinkedList ...
{   ...
    public ListIterator getIterator()
    {  return ...; } // dopo vediamo come fare
}
```

- e si può scandire la lista concatenata senza accedere ai nodi

```
LinkedList list = new LinkedList();
...
ListIterator iter = list.getIterator();
while(iter.hasNext())
    System.out.println(iter.next());
```

Iteratore in una Linked List

- Come realizzare il metodo **getIterator** nella lista concatenata?
  - osserviamo che restituisce un riferimento ad una interfaccia, per cui dovrà creare un oggetto di una classe che realizzi tale interfaccia
  - definiamo la classe **LinkedListIterator** che realizza l'interfaccia **ListIterator**

- Gli oggetti di tale classe vengono costruiti soltanto all'interno di **LinkedList** e vengono restituiti all'esterno soltanto tramite riferimenti a **ListIterator**
  - quindi usiamo una classe interna privata

Iteratore in una lista concatenata

- Per un corretto funzionamento dell'iteratore occorre concedere a tale oggetto il pieno accesso alla lista concatenata
  - in particolare, alla sua variabile di esemplare **head**
  - non vogliamo però che l'accesso sia consentito anche ad altre classi
- Questo è consentito dalla definizione di classe interna
  - una **classe interna** può accedere agli elementi **private** della classe in cui è definita
  - essendo tali elementi definiti **private**, l'accesso è impedito alle altre classi

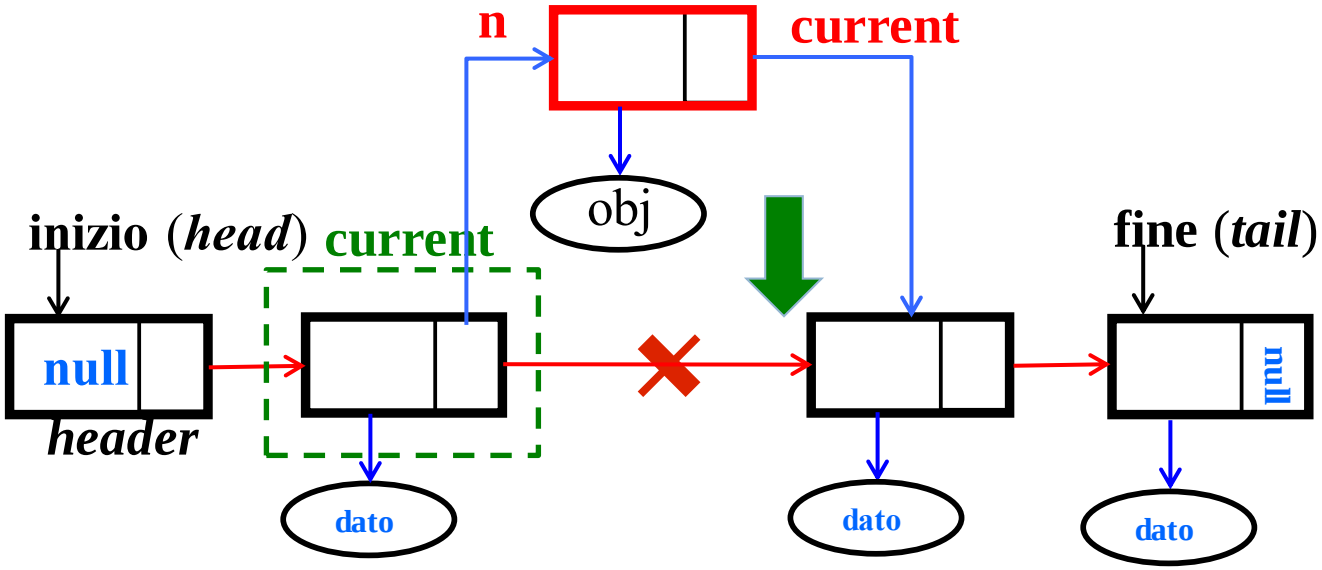
```
public class LinkedList
{   ... // codice di LinkedList come prima
    ... // incluso il codice della classe privata ListNode

    public ListIterator getIterator() // metodo di LinkedList
    {   return new LinkedListIterator(head); }

    private class LinkedListIterator implements ListIterator
    {
        public LinkedListIterator(ListNode head)
        {   current = head;
            previous = null;
        }

       // metodi pubblici (da realizzare)
        public boolean hasNext(){}
        public Object next(){}
        public void add(Object x){}
        public void remove(){}

        // campi di esemplare
        private ListNode current; //nodo che precede pos attuale
        private ListNode previous;// nodo che precede current

    }
}
```

```
public class LinkedList
{   ...

    private class LinkedListIterator implements ListIterator
    {
        . . .
        public boolean hasNext()
        {   return current.getNext() != null;
        }

        public Object next()
        {   if (!hasNext())
                throw new NoSuchElementException();
            previous = current;
            current = current.getNext();
            return current.getElement();
        }

        // metodi pubblici (da realizzare)
        public void add(Object x){}
        public void remove(){}

        // campi di esemplare
        private ListNode current; //nodo che precede pos attuale
        private ListNode previous;// nodo che precede current
    }
}
```

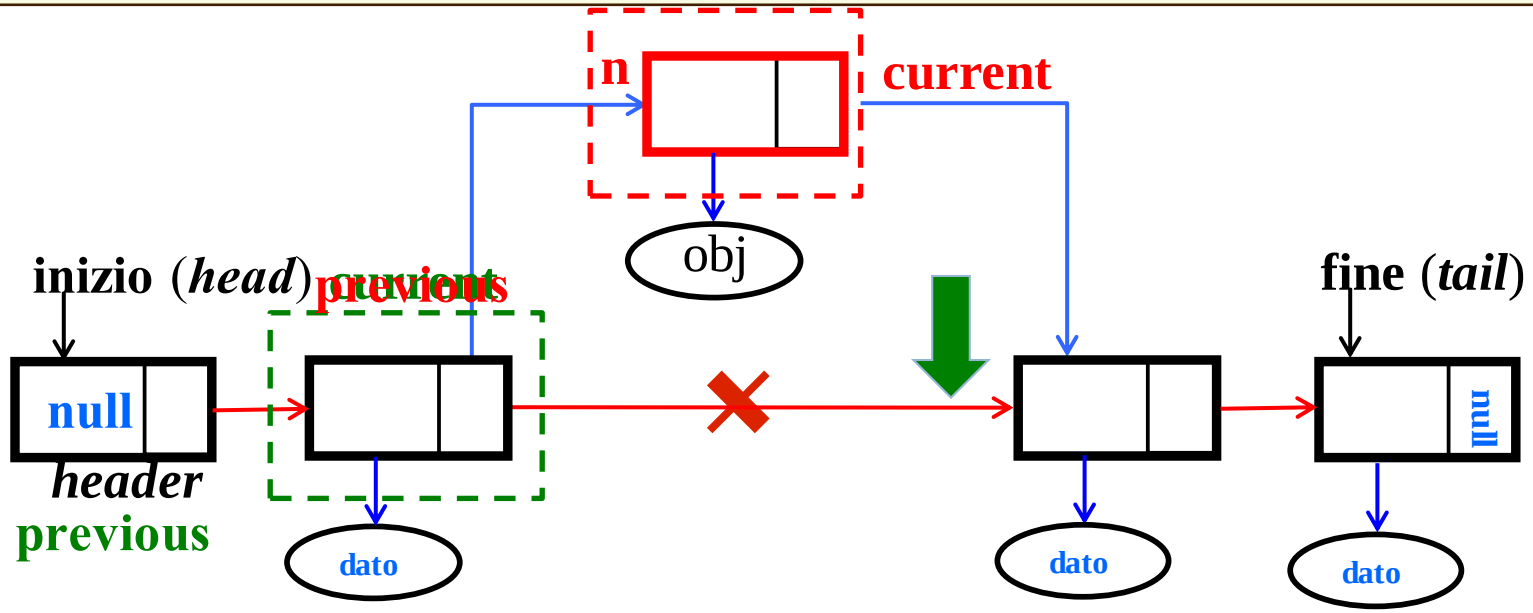
Il metodo add di LinkedListIterator

- Aggiunge il nuovo nodo e avanza di una posizione
- Il nuovo nodo diventa current
- Il vecchio current diventa previous
- Se il nodo viene aggiunto alla fine della lista concatenata, allora bisogna anche aggiornare il riferimento **tail** della lista concatenata
  - **Nota sintattica**: il riferimento **LinkedList.this** punta all'oggetto **LinkedList** all'interno di cui è stato creato l'iterator

```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void add(Object obj)
        {   ListNode n = new ListNode(obj, current.getNext());
            current.setNext(n);
            previous = current;
            current = n; // il nodo prima l'iteratore e' n

            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
        }
        . . .
    }
}
```

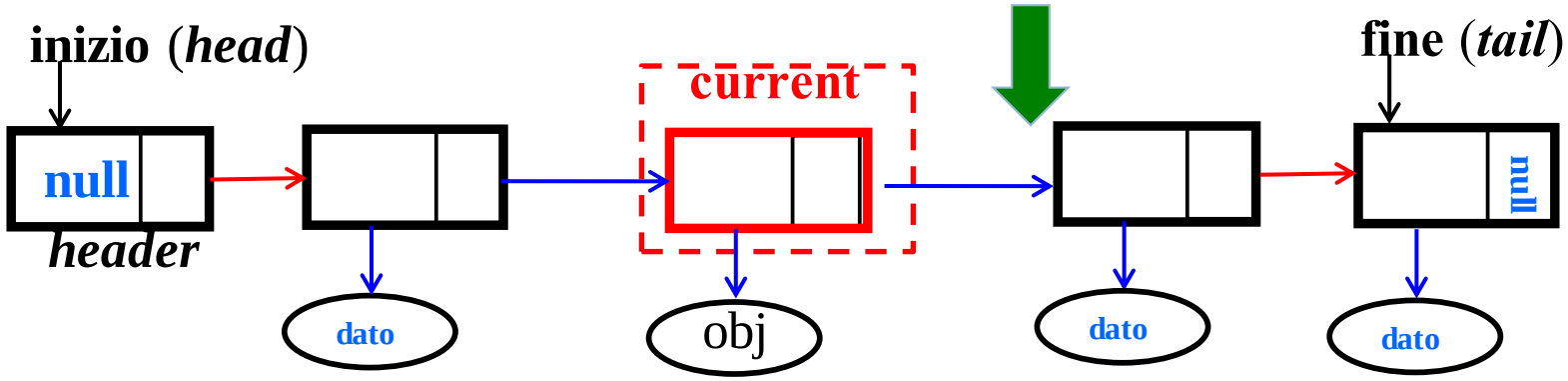


```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void add(Object obj)
        {   ListNode n = new ListNode(obj, current.getNext());
            current.setNext(n);
            previous = current;
            current = n; // il nodo prima l'iteratore e' n

            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
        }
        . . .
    }
}
```

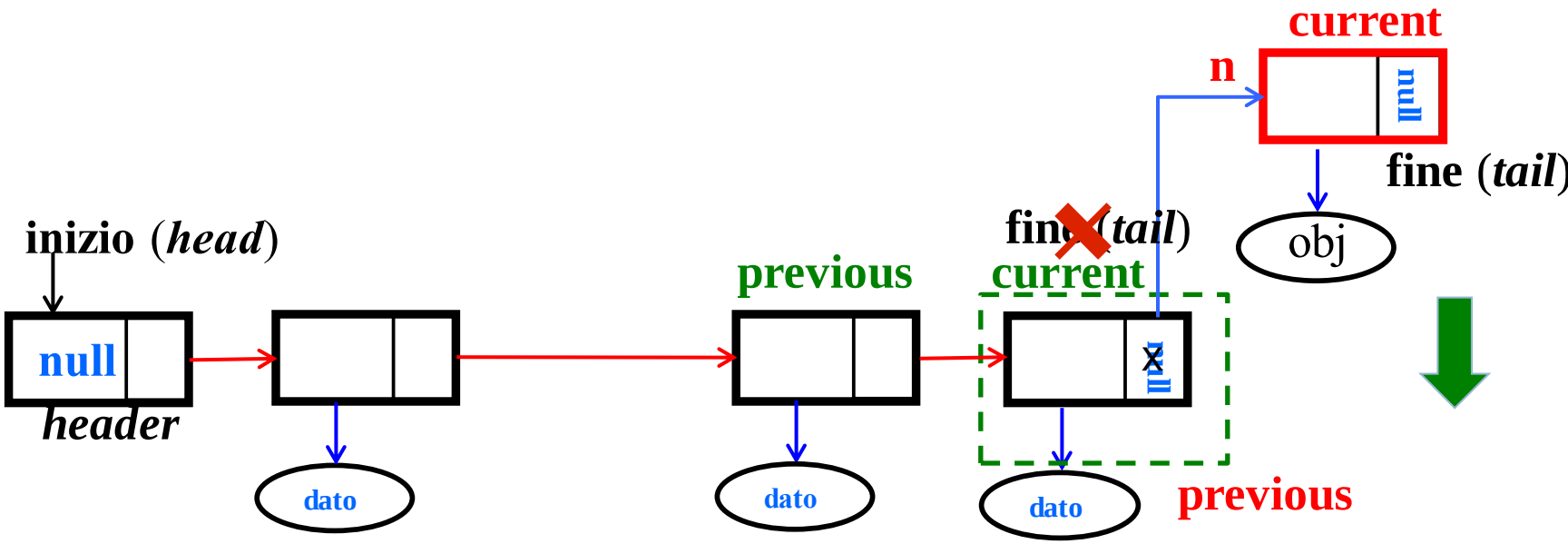


```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void add(Object obj)
        {   ListNode n = new ListNode(obj, current.getNext());
            current.setNext(n);
            previous = current;
            current = n; // il nodo prima l'iteratore e' n

            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
        }
        . . .
    }
}
```

CONFIGURAZIONE FINALE



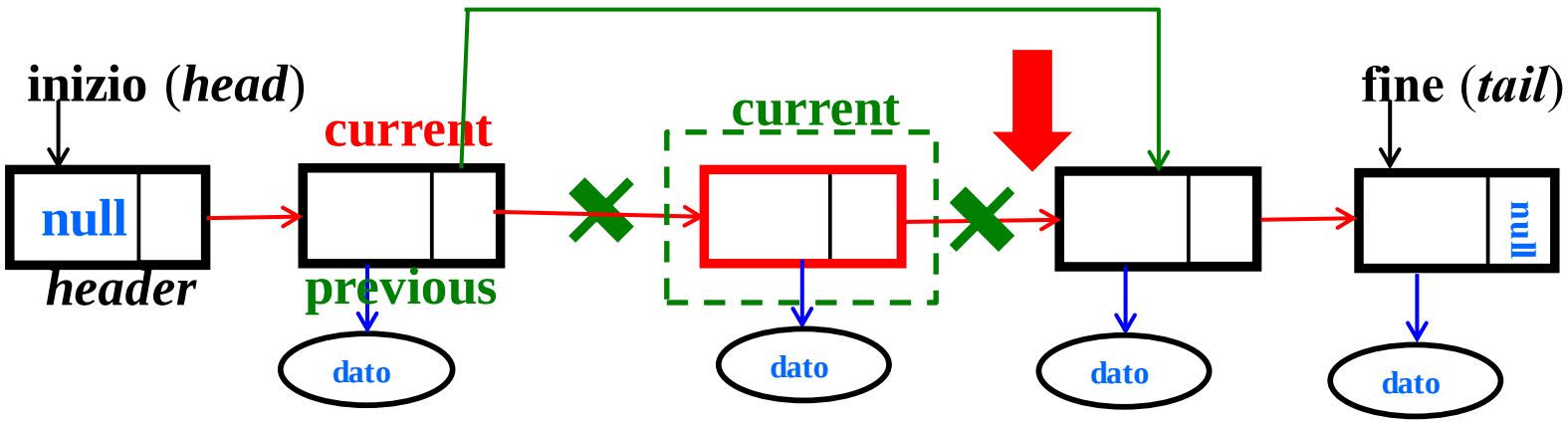
```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void add(Object obj)
        {   ListNode n = new ListNode(obj, current.getNext());
            current.setNext(n);
            previous = current;
            current = n; // il nodo prima l'iteratore e' n

            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
        }
        . . .
    }
}
```



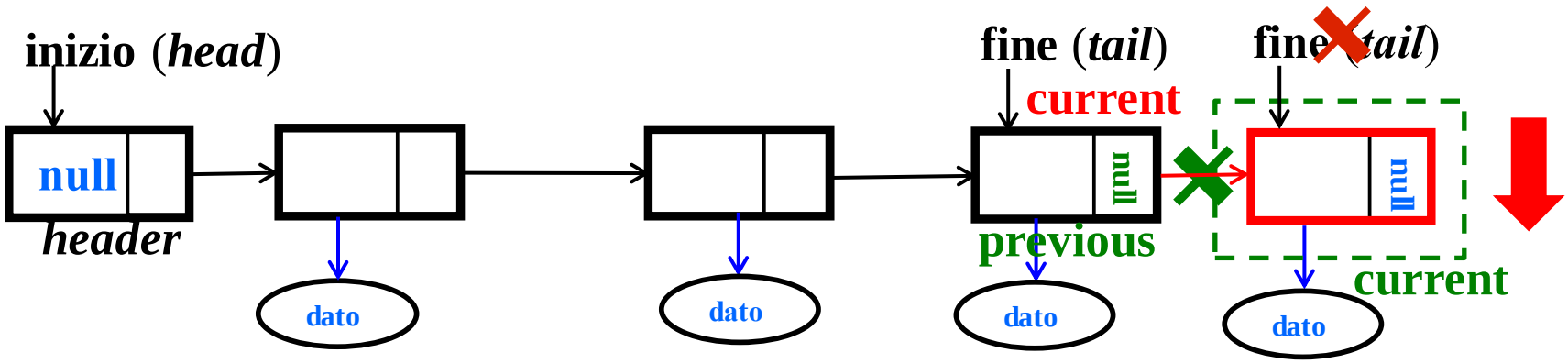
Il metodo remove di LinkedListIterator

- Rimuovendo il nodo current, previous diventa current
- L'iteratore non ha un riferimento al nodo prima di previous
  - Quindi non posso aggiornare correttamente previous dopo la rimozione: gli assegno valore **null**, con la conseguenza che l'iteratore si trova in uno **stato illegale** dopo la rimozione
  - Per questo motivo non si può invocare nuovamente remove. Invocando next o add il valore di **previous** verrà riaggiornato

```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void remove()
        {   if(previous == null)throw IllegalStateException
            previous.setNext(current.getNext());
            current = previous;
            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
            previous = null; // non si puo' fare remove due volte
        }
        . . .
    }
}
```



```
    private class LinkedListIterator implements ListIterator
    {   . . .
        //la posizione dell'iteratore rimane immutata
        public void remove()
        {   if(previous == null)throw IllegalStateException
            previous.setNext(current.getNext());
            current = previous;
            if (current.getNext() == null) // gestione ultimo nodo
                tail = current;
            previous = null; // non si puo' fare remove due volte
        }
        . . .
    }
}
```



Lista concatenata conteggio di elementi

- Possiamo quindi riscrivere il metodo di conteggio degli elementi contenuti in una lista concatenata, ma al di fuori della lista concatenata stessa, in una classe qualsiasi

```
public static int getSize(LinkedList list)
{  ListIterator iter = list.getIterator();
   int size = 0;
   while (iter.hasNext())
   {  size++;
      iter.next(); // ignoro l'oggetto ricevuto
   }
   return size;
}
```

Esercizi:
Utilizzo di liste concatenate

Pila realizzata con lista concatenata

- Una pila può essere realizzata usando una lista concatenata invece di un array
- Si noti che entrambe le estremità di una lista concatenata hanno, prese singolarmente, il comportamento di una pila
  - si può quindi realizzare una pila usando una delle due estremità della lista concatenata
  - è più efficiente usare **l'inizio** della lista concatenata, perché le operazioni su tale estremità sono **O(1)**

Pila realizzata con lista concatenata

```
public class LinkedListStack implements Stack
{  private LinkedList list = new LinkedList();
   public void push(Object obj)
   {  list.addFirst(obj);
   }
   public Object pop()
   {  return list.removeFirst();
   }
   public Object top()
   {  return list.getFirst();
   }
   public void makeEmpty()
   {  list.makeEmpty();
   }
   public boolean isEmpty()
   {  return list.isEmpty();
   }
}
```

Coda realizzata con lista concatenata

- Anche una coda può essere realizzata usando una catena invece di un array
- È sufficiente inserire gli elementi ad un'estremità della catena e rimuoverli dall'altra estremità per ottenere il comportamento di una coda
- Perché tutte le operazioni siano **O(1)** bisogna **inserire alla fine e rimuovere all'inizio**

Coda realizzata con lista concatenata

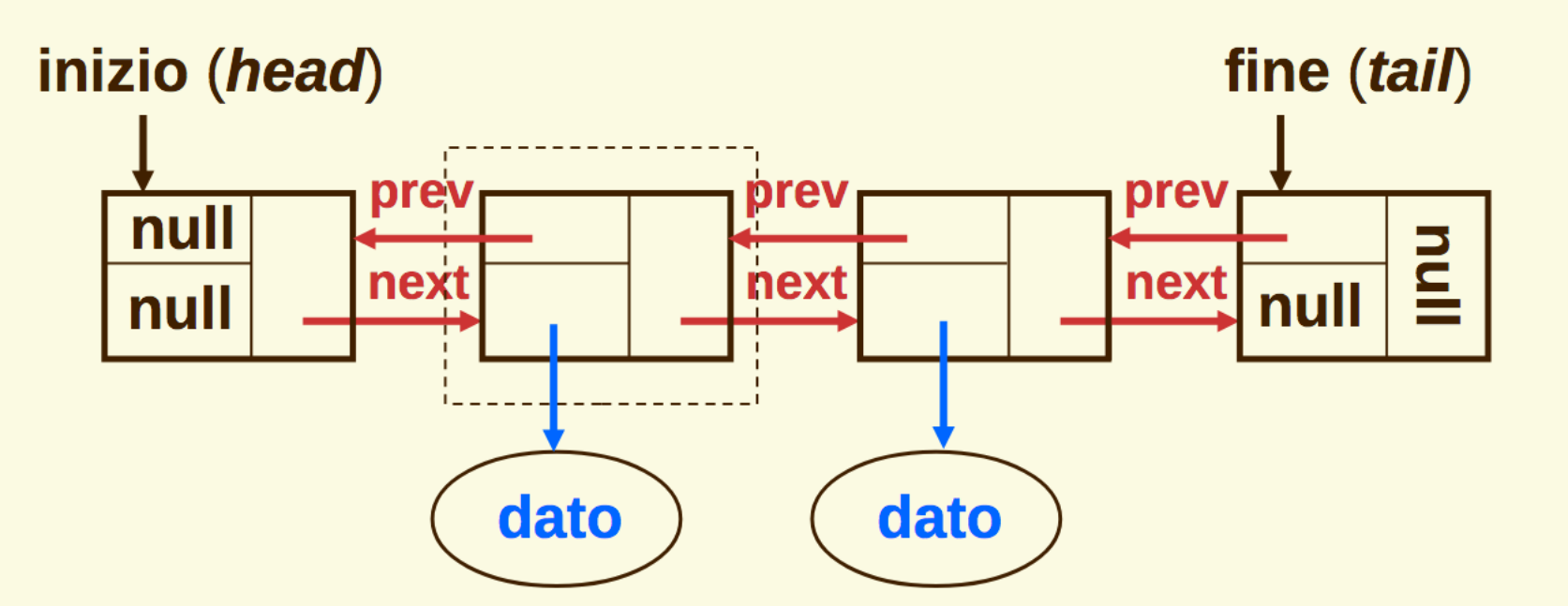
```
public class LinkedListQueue implements Queue
{  private LinkedList list = new LinkedList();
   public void enqueue(Object obj)
   {  list.addLast(obj);

   public Object dequeue()
   {  return list.removeFirst();
   }
   public Object getFront()
   {  return list.getFirst();
   }
   public void makeEmpty()
   {  list.makeEmpty();
   }
   public boolean isEmpty()
   {  return list.isEmpty();
   }
}
```

Doubly linked list

Doubly linked list

- La lista concatenata doppia (lista doppiamente concatenata, doubly linked list) è una struttura dati.
- ogni nodo è un oggetto che contiene
  - un riferimento ad un elemento (il dato)
  - un riferimento al nodo successivo della lista (next)
  - un riferimento al nodo precedente della lista (prev)

Doubly linked list



- Dato che la struttura è ora simmetrica, si usano due nodi che non contengono dati, uno a ciascun estremo della catena

Doubly linked list

- Tutto quanto detto per la lista concatenata (semplice) può essere agevolmente esteso alla catena doppia
- Il metodo removeLast() diventa O(1) come gli altri metodi
- I metodi di inserimento e rimozione si complicano

Hash Table

Funzione di hash

- Una funzione di hash ha
  - come **dominio** l'insieme delle chiavi che identificano univocamente i dati da inserire nella mappa
  - come **codominio** l'insieme degli indici validi per accedere ad elementi della tabella
    - il risultato dell'applicazione della funzione di hash ad una chiave si chiama **chiave ridotta**
- Se manteniamo la limitazione di usare numeri interi come chiavi
  - una semplice funzione di hash è il calcolo del **resto della divisione intera tra la chiave e la dimensione della tabella**

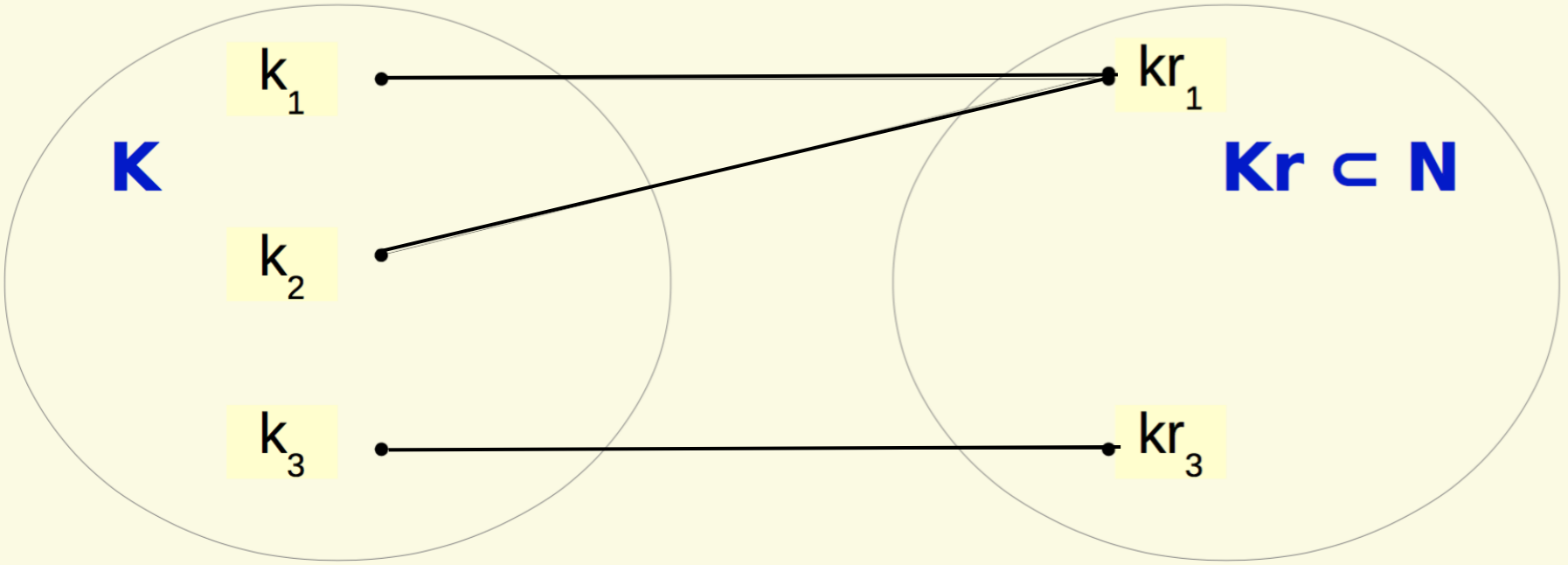
$$Kr = k \% dim$$

Funzione di Hash

Insieme delle chiavi
K

Insieme delle chiavi ridotte
Kr⊂ N

Funzione di hash: F: K -> Kr



Ad ogni chiave e' associata una e una sola chiave ridotta. Piu' chiavi possono essere associate ad una stessa chiave ridotta

Funzione di hash

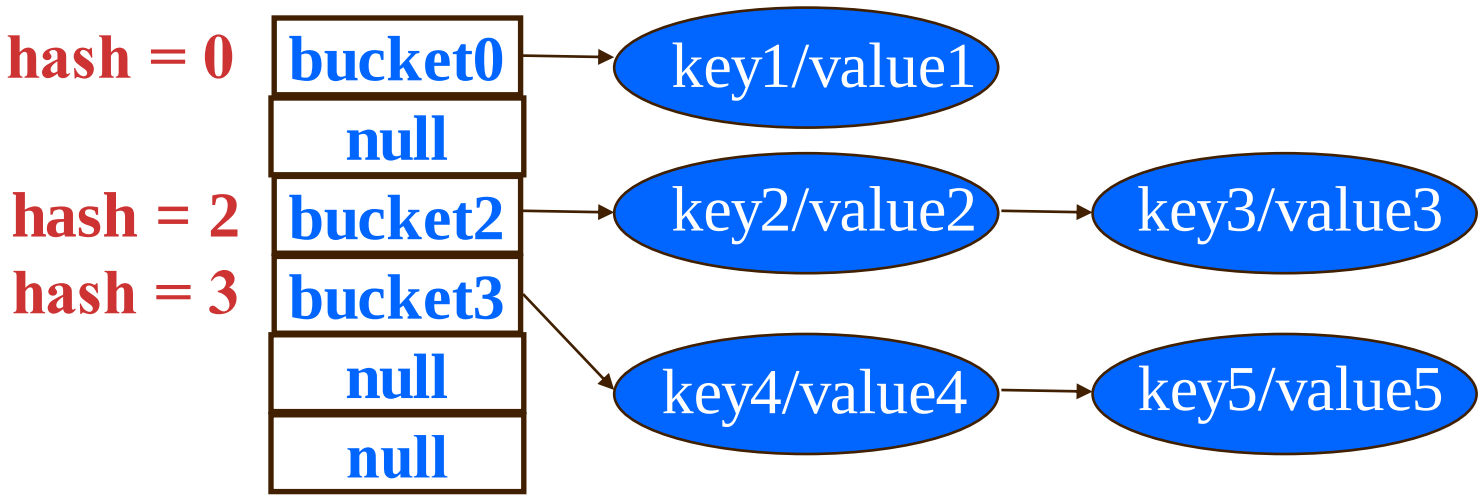
- Per come è definita, la **funzione di hash** è **generalmente non biunivoca**, cioè non è invertibile
  - **chiavi diverse possono avere lo stesso valore per la funzione di hash**
- Per questo si chiama funzione di **hash**
  - è una funzione che "**fa confusione**", nel senso che "mescola" dati diversi…
- In generale, non è possibile definire una funzione di hash biunivoca, perché la dimensione del dominio è maggiore della dimensione del codominio

Risoluzione delle collisioni

- Quando si ha una collisione, bisognerebbe inserire il nuovo valore nella stessa cella della tabella (dell'array) che già contiene un altro valore
  - i due valori hanno chiavi diverse, ma la stessa chiave ridotta
  - **ciascun valore deve essere memorizzato insieme alla sua vera chiave, per poter fare ricerche correttamente**
- Possiamo risolvere il problema **usando una lista concatenata** per ogni cella dell'array
  - Usiamo un array che è un array di riferimenti a liste
  - ciascuna lista contiene le coppie **chiave/valore** che hanno la stessa chiave ridotta

Risoluzione delle collisioni

- Questo sistema di risoluzione delle collisioni si chiama **tabella hash con bucket**
  - un **bucket** è una delle liste associate ad una chiave ridotta



Risoluzione delle collisioni

- Le prestazioni della tabella hash con bucket non sono più, ovviamente, **O(1)** per tutte le operazioni
- **Le prestazioni dipendono fortemente dalle caratteristiche della funzione di hash**
  - **caso peggiore**: la funzione di hash restituisce sempre la stessa chiave ridotta, per ogni chiave possibile
  - tutti i dati vengono inseriti in un'unica lista
    - le prestazioni della tabella hash degenerano in quelle di una lista
    - tutte le operazioni sono **O(n)**

Hash table con bucket

- **caso migliore**: la funzione di hash restituisce chiavi ridotte che si **distribuiscono uniformemente** nella tabella
  - Dato un insieme S di n elementi, la distribuzione uniforme attribuisce a ciascun elemento la stessa probabilità di occorrere
    - Es: in un dato non truccato, ogni faccia ha probabilità 1/6 di occorrere
    - Es: in una moneta non truccata, ogni faccia ha probabilità ½ di occorrere
    - Es: Se la nostra tabella ha 100 posizioni, quindi 100 chiavi ridotte, ciascuna chiave ridotta ha probabilità 1/100 di essere il risultato della funzione di hash

Hash table con bucket

- In questo caso tutte le liste associate ad una chiave ridotta hanno la stessa lunghezza media
  - se **M** è la dimensione della tabella, la lunghezza media di ciascuna lista è **n/M**
    - es: se ho n=100 chiavi e una tabella di dimensione M=20, ciascuna posizione(=chiave ridotta) avrà (circa) 5 elementi (cui corrispondono chiavi diverse trasformate nella stessa chiave ridotta)
  - tutte le operazioni sono **O(n/M)**
- per avere prestazioni **O(1)** occorre dimensionare la tabella in modo che **M** sia dello stesso ordine di grandezza di **n**

Hash table con bucket

- Riassumendo, in una ***hash table con bucket*** si ottengono prestazioni ottimali (tempo-costanti) se
  - la dimensione della tabella è circa uguale al numero di dati che saranno memorizzati nella tabella
    - fattore di riempimento circa unitario
      - così si riduce al minimo anche lo spreco di memoria
  - la funzione di hash genera chiavi ridotte uniformemente distribuite
    - liste di lunghezza quasi uguale alla lunghezza media
    - le liste hanno quasi tutte lunghezza uno!

Hash table con bucket

- Se le chiavi vere sono uniformemente distribuite
  - la funzione di hash che "**calcola il resto della divisione intera**" genera chiavi ridotte uniformemente distribuite

Hash Table con chiave generica

Hash Table con chiave generica

- Rimane da risolvere un solo problema
  - le **chiavi** devono essere **numeri interi** (non negativi)
- Vogliamo cercare di realizzare una tabella hash con bucket che possa gestire coppie chiave/valore in cui **la chiave** non è un numero intero, ma **un dato generico**
  - ad esempio, una stringa
  - applicazione
    - contenitore di oggetti di tipo PersonaFisica
    - chiave: codice fiscale

Hash Table con chiave generica

- Se vogliamo usare **chiavi generiche** (qualsiasi tipo di oggetto, in teoria…)
  - è sufficiente progettare una **adeguata funzione di hash**
- **Se** la funzione di hash ha come dominio l'insieme delle chiavi generiche che interessano e come codominio un sottoinsieme dei numeri interi (cioè se **le chiavi ridotte sono numeri interi**) **allora la tabella hash con bucket funziona correttamente senza modifiche**

Funzione di hash per stringhe

- **Come si può trasformare una stringa in un numero intero?**
- Ricordiamo il significato della notazione posizionale nella rappresentazione di un numero intero

$$434 = 4 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

- Ciascuna cifra rappresenta un numero che dipende
  - dal suo valore intrinseco
  - dalla sua posizione

Funzione di hash per stringhe

- Possiamo usare la stessa convenzione per una stringa, dove ciascun carattere rappresenta un numero che dipende
  - dal suo valore intrinseco come carattere nella codifica Unicode
  - dalla sua posizione nella stringa

- La base del sistema sarà il numero di simboli diversi (come nella base decimale), che per la codifica Unicode è 65536

$$\texttt{"ABC"} \Rightarrow \texttt{'A'} \cdot 65536^2 + \texttt{'B'} \cdot 65536^1 + \texttt{'C'} \cdot 65536^0$$

Funzione hash per oggetti generici

- La classe **Object** mette a disposizione il metodo **hashCode** che restituisce un int con buone proprietà di distribuzione uniforme
- se il metodo **hashCode** non viene ridefinito, viene calcolata una chiave ridotta a partire dall'indirizzo dell'oggetto in memoria
- l'esistenza di questo metodo rende possibile l'utilizzo di qualsiasi oggetto come chiave in una tabella hash
  - invocando **hashCode** si ottiene un valore di tipo int
    - calcolando il resto della divisione intera di tale valore per la dimensione della tabella, si ottiene finalmente la chiave ridotta

```
int hash = obj.hashCode() % DIM;
Attenzione hash □ [-DIM+1, DIM-1]
if (hash < 0) hash = -hash; // modulo di hash
// ora hash e' la chiave ridotta
```

Tabella hash per chiave generica

```
public interface HashTable extends Container
{
   void insert(Object key, Object value);

   void remove(Object key);

   Object find(Object key);
}
```