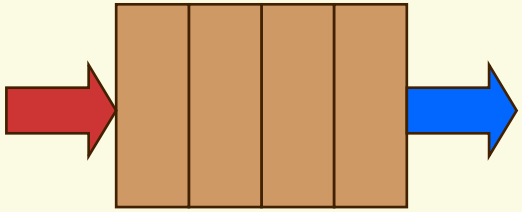
Lezione XXVII
Lu 14-Nov-2005

ADT Coda
(Queue)

Coda (*queue*)



- In una ***coda*** (***queue***) gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **FIFO** (***First In, First Out***)
  - il primo oggetto inserito è il primo a essere estratto
  - il nome è stato scelto in analogia con persone in ***coda***
- L'unico oggetto che può essere ispezionato è quello che verrebbe estratto
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
  - la simulazione del funzionamento di uno sportello bancario con più clienti che arrivano in momenti diversi userà una coda per rispettare la priorità di servizio

Coda (*queue*)

- I metodi che caratterizzano una coda sono
  - **enqueue()** per inserire un oggetto nella coda
  - **dequeue()** per esaminare ed eliminare dalla coda l'oggetto che vi si trova da più tempo
  - **getFront()** per esaminare l'oggetto che verrebbe eliminato da **dequeue()**, senza estrarlo
- Infine, ogni ADT di tipo "contenitore" ha i metodi
  - **isEmpty()** per sapere se il contenitore è vuoto
  - **makeEmpty()** per vuotare il contenitore

Coda (*queue*)

```
public interface Queue extends Container
{  void enqueue(Object obj);
   Object dequeue();
   Object getFront();
}
```

- Si notino le similitudini con la pila
  - **enqueue()** corrisponde a **push()**
  - **dequeue()** corrisponde a **pop()**
  - **getFront()** corrisponde a **top()**

Coda (*queue*)

```
public interface Queue extends Container
{  /**
    inserisce un elemento all'ultimo posto della coda
    @param obj l'elemento da inserire
   */
   void enqueue(Object obj);

   /**
    rimuove l'elemento in testa alla coda
    @return l'elemento rimosso
    @throws QueueEmptyException se la coda e' vuota
   */
   Object dequeue() throws EmptyQueueException;

   /**
    restituisce l'elemento in testa alla coda
    @return l'elemento in testa alla coda
    @throws EmptyQueueException se la coda e' vuota
   */
   Object getFront() throws EmptyQueueException;
}
```

Coda (*queue*)

- Per ***realizzare una coda*** si può usare una struttura di tipo ***array*** "riempito solo in parte", in modo simile a quanto fatto per realizzare una pila
- Mentre nella pila si inseriva e si estraeva allo stesso estremo dell'array (l'estremo "destro"), qui dobbiamo inserire ed estrarre ai due diversi estremi
  - decidiamo di inserire a destra ed estrarre a sinistra

Coda (*queue*)

- Come per la pila, anche per la coda bisognerà segnalare l'errore di accesso a una coda vuota e gestire la situazione di coda piena (segnalando un errore o ridimensionando l'array)
- Definiamo
  - **EmptyQueueException** e **FullQueueException**

```
public class FullQueueException extends
    RuntimeException
{ }
```

```
public class SlowFixedArrayQueue implements Queue
{
  private Object[] v;
  private int vSize;

  public SlowFixedArrayQueue()
  { v = new Object[100];
    makeEmpty();
  }

  public void makeEmpty()
  { vSize = 0;
  }

  public boolean isEmpty()
  { return (vSize == 0);
  }
  …// continua
```

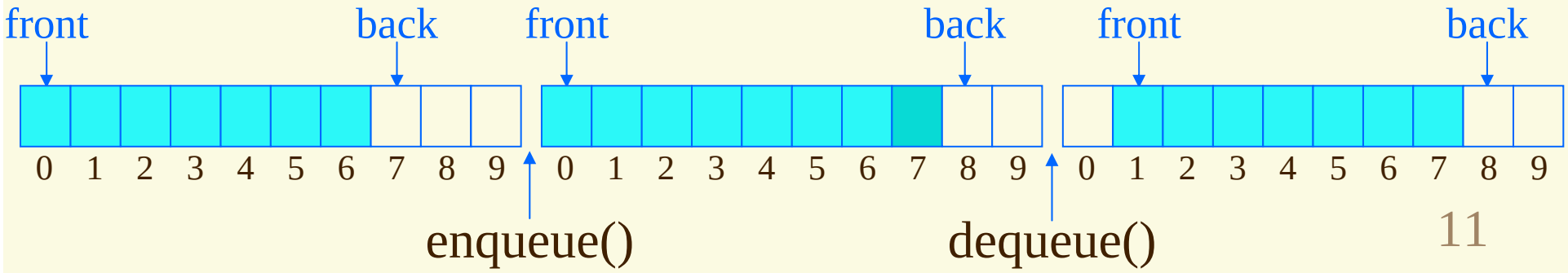
```
  … // continua
  public void enqueue(Object obj)
  { if (vSize ==v.length)
       throw new FullQueueException();
    v[vSize++] = obj;
  }

  public Object getFront() throws EmptyQueueException
  { if (isEmpty())
       throw new EmptyQueueException();
    return v[0];
  }

  public Object dequeue() throws EmptyQueueException
  {
    Object obj = getFront();
    vSize--;
    for (int i = 0; i < vSize-1; i++)
       v[i] = v[i+1];
    return obj;
  }
}
```

Coda (*queue*)

- Questa semplice realizzazione con array, che abbiamo visto essere molto efficiente per la pila, è al contrario assai inefficiente per la coda
  - il metodo **dequeue()** è `O(n)`, perché bisogna spostare tutti gli oggetti della coda per fare in modo che l'array rimanga "compatto"
  - la differenza rispetto alla pila è dovuta al fatto che nella coda gli inserimenti e le rimozioni avvengono alle due estremità diverse dell'array, mentre nella pila avvengono alla stessa estremità

Coda (*queue*)

- Per realizzare una coda più efficiente servono ***due indici*** anziché uno soltanto
  - un indice punta al primo oggetto della coda (front) e l'altro indice punta al primo posto libero dopo l'ultimo oggetto della coda (back)
- In questo modo, aggiornando opportunamente gli indici, si ottiene la realizzazione di una coda con un "***array riempito solo nella parte centrale***" in cui tutte le operazioni sono **`O(1)`**
  - la gestione dell'array pieno ha le due solite soluzioni, ridimensionamento o eccezione



```
public class FixedArrayQueue implements Queue
{
  protected Object[] v;
  protected int front, back;

  public FixedArrayQueue()
  {
    v = new Object[100];
    makeEmpty();
  }

  public void makeEmpty()
  {
    front = back = 0;
  }

  public boolean isEmpty()
  { return (back == front);
  }
  // continua
```

```
… // continua
  public void enqueue(Object obj)
  { if (back == v.length)
        throw new FullQueueException();
    v[back++] = obj;
  }

  public Object getFront() throws EmptyQueueException
  { if (isEmpty())
        throw new EmptyQueueException();
    return v[front];
  }

  public Object dequeue() throws EmptyQueueException
  { Object obj = getFront();
    v[front] = null;  //garbage collector
    front++;
    return obj; }
}
```

Coda (*queue*)

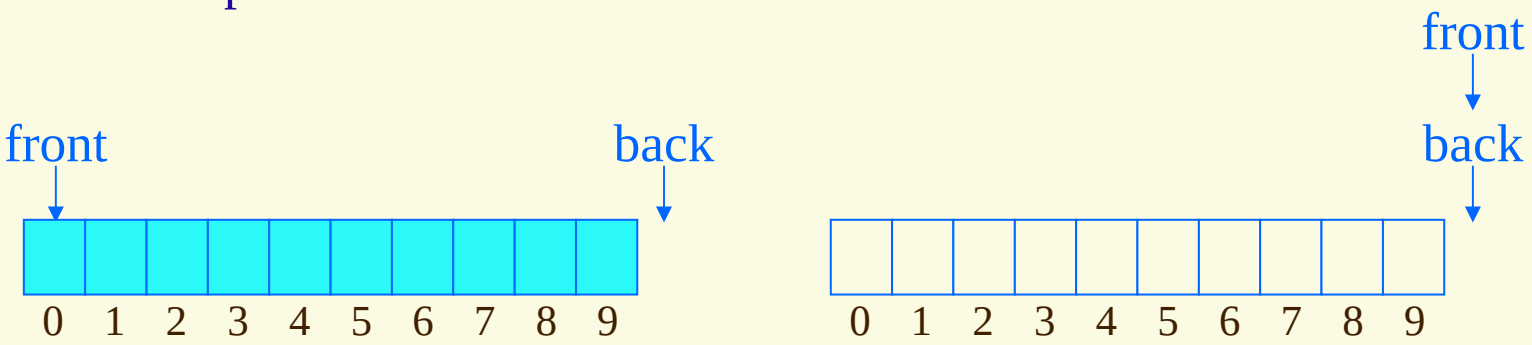
- Per rendere la coda ridimensionabile, usiamo la stessa strategia vista per la pila, estendendo la classe **FixedArrayQueue** e sovrascrivendo il solo metodo **enqueue()**

```
public class GrowingArrayQueue
       extends FixedArrayQueue
{  public void enqueue(Object obj)
   {  if (back == v.length)
          v = resize(v, 2*v.length);
      super.enqueue(obj);
   }
}
```

Prestazioni della coda

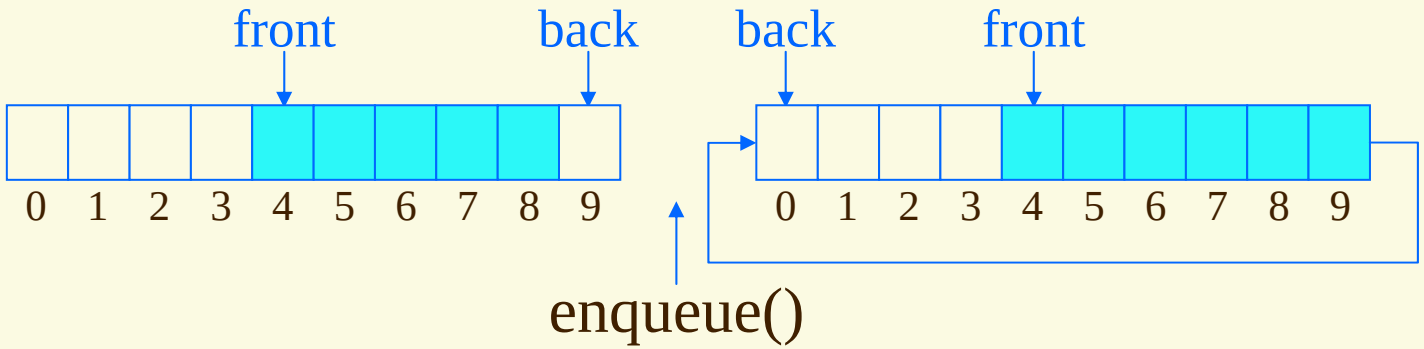
- La realizzazione di una coda con un array e due indici ha la massima efficienza in termini di prestazioni temporali, tutte le operazioni sono **`O(1)`**, ma ha ancora un punto debole
- Se l'array ha N elementi, proviamo a
  - effettuare N operazioni **enqueue()**

  e poi
  - effettuare N operazioni **dequeue()**
- Ora ***la coda è vuota***, ma alla successiva operazione **enqueue()** ***l'array sarà pieno***
  - lo spazio di memoria non viene riutilizzato



Coda con array circolare

- Per risolvere quest'ultimo problema si usa una tecnica detta "***array circolare***"
  - i due indici, dopo essere giunti alla fine dell'array, possono ritornare all'inizio se si sono liberate delle posizioni
  - in questo modo l'array risulta pieno solo se la coda ha effettivamente un numero di oggetti uguale alla dimensione dell'array
  - le prestazioni temporali rimangono identiche

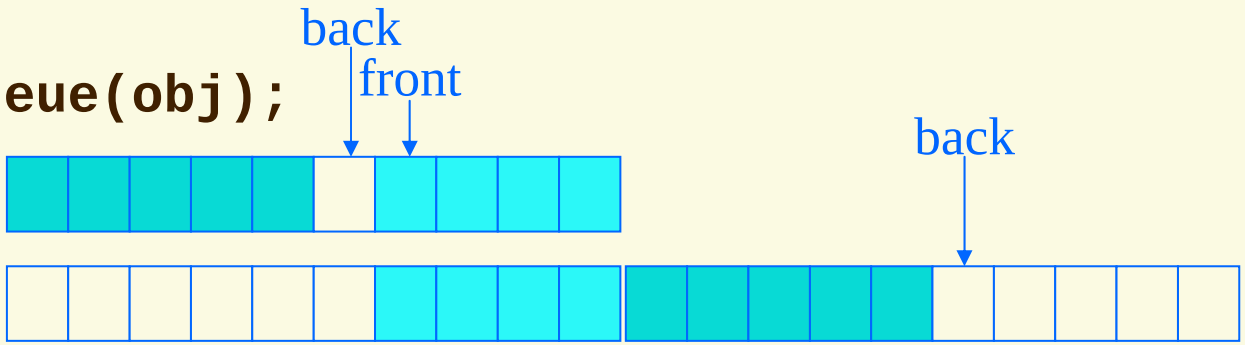


Coda con array circolare

```
public class FixedCircularArrayQueue extends
   FixedArrayQueue
{  // il metodo increment() fa avanzare un indice di una
   // posizione, tornando all'inizio dell'array se si
   // supera la fine
   protected int increment(int index)
   {  return (index + 1) % v.length;
   }
   public void enqueue(Object obj)
   {  if (increment(back) == front)
         throw new FullQueueException();
      v[back] = obj;
      back = increment(back);
   }
   public Object dequeue()
   {  Object obj = getFront();
      front = increment(front);
      return obj;
   }
   // non serve sovrascrivere getFront() perché non
   // modifica le variabili back e front
}
```

Coda con array circolare

```
public class GrowingCircularArrayQueue
      extends FixedCircularArrayQueue
{ public void enqueue(Object obj)
  { if (increment(back) == front)
    { v = resize(v, 2*v.length);
      // se si ridimensiona l'array e la zona utile
      // della coda si trova attorno alla sua fine,
      // la seconda metà del nuovo array rimane vuota
      // e provoca un malfunzionamento della coda,
      // che si risolve spostandovi la parte della
      // coda che si trova all'inizio dell'array
      if (back < front)
       {  System.arraycopy(v, 0, v, v.length/2, back);
            back += v.length/2;
       }
    }
    super.enqueue(obj);
  }
}
```



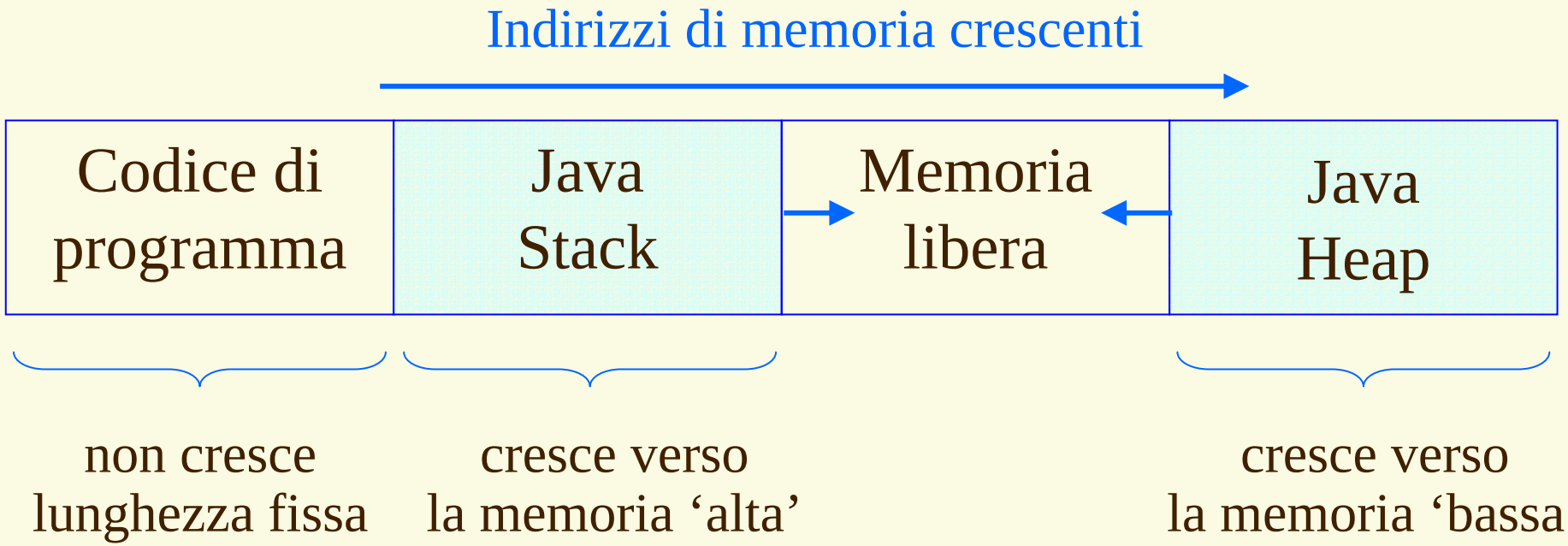
Allocazione della memoria in Java

- Il java Stack non è l'unico tipo di memoria disponibile per i programmi java.
- Durante l'esecuzione dei metodi di un programma vengono creati dinamicamente oggetti (allocazione dinamica) usando lo speciale operatore new di java
  - **`BankAccount acct = new BankAccount()`**

  crea dinamicamente un oggetto di classe BankAccount
- Questo oggetto continua a esistere anche quando l'invocazione del metodo è terminata e quindi non può essere allocato nel Java stack
- Per l'allocazione dinamica Java usa memoria da un'altra area denominata **java heap**
- La memoria nello **java heap** è suddivisa in blocchi che sono formati da locazioni contigue (come negli array) e possono essere di lunghezza fissa o variabile
- I blocchi non usati sono mantenuti in una coda di blocchi

Allocazione della memoria in Java

- Schema della disposizione degli indirizzi di memoria nella Java Virtual Machine



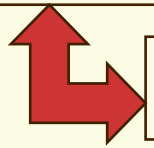
Esercizio: Calcolatrice

- Vogliamo risolvere il problema di calcolare il risultato di un'espressione aritmetica (ricevuta come **String**) contenente somme, sottrazioni, moltiplicazioni e divisioni
- Se l'espressione usa la classica notazione (detta ***infissa***) in cui i due operandi di un'operazione si trovano ai due lati dell'operatore, l'ordine di esecuzione delle operazioni è determinato dalle regole di precedenza tra gli operatori e da eventuali parentesi
- Scrivere un programma per tale compito è piuttosto complesso, mentre è molto più facile calcolare espressioni che usano una diversa notazione

Notazione postfissa

- Un’espressione aritmetica può anche usare la notazione ***postfissa***, detta anche ***notazione polacca inversa*** (**RPN**, ***Reverse Polish Notation***)

```
7 1 2 + 4 * 5 6 + - /
```



```
7/[(1+2)*4 - (5+6)]
```

- In tale notazione non sono ammesse parentesi (né sarebbero necessarie)
- I due operandi di ciascun operatore si trovano alla sua sinistra

Notazione postfissa

- Esiste un semplice algoritmo che usa una pila per valutare un'espressione in notazione postfissa
- Finché l'espressione non è terminata
  - leggi da sinistra il primo simbolo o valore non letto
  - se è un valore, inseriscilo sulla pila
  - altrimenti (è un operatore…)
    - estrai dalla pila l'operando destro
    - estrai dalla pila l'operando sinistro
    - esegui l'operazione
    - inserisci il risultato sulla pila
- Alla fine, se la pila contiene più di un valore, l'espressione contiene un errore
- L'unico valore presente sulla pila è il risultato

```
public static double evaluateRPN(String s)
{ Stack st = new GrowingArrayStack();
  StringTokenizer tk = new StringTokenizer(s);
  while (tk.hasMoreTokens())
  {  String x = tk.nextToken();
     try
     {  Double.parseDouble(x);
        // è un valore numerico
        st.push(x);
     } catch (NumberFormatException e)
     {  // è un operatore
        double r = evalOperator(x,
          (String)st.pop(), (String)st.pop());
        st.push(Double.toString(r));
     }
  }
  double r= Double.parseDouble((String)st.pop());
  if (!st.isEmpty()) throw new RuntimeException();
   return r;
}
```

```
private static double evalOperator(String op,
                                   String right,
                                   String left)
{  double opLeft = Double.parseDouble(left);
   double opRight = Double.parseDouble(right);
   double result;

   if (op.equals("+"))
      result = opLeft + opRight;
   else if (op.equals("-"))
      result = opLeft - opRight;
   else if (op.equals("*"))
      result = opLeft * opRight;
   else if (op.equals("/"))
      result = opLeft / opRight;
   else throw new RuntimeException();

   return result;
}
```

Esercizio:
Controllo di parentesi

Esercizio: Controllo parentesi

- Vogliamo risolvere il problema di verificare se in un'espressione algebrica (ricevuta come **String**) le parentesi tonde, quadre e graffe sono utilizzate in maniera corretta
- In particolare, vogliamo verificare che a ogni parentesi aperta corrisponda una parentesi chiusa dello stesso tipo
- Risolviamo prima il problema nel caso semplice in cui non siano ammesse parentesi annidate

Esercizio: Algoritmo

- Inizializza la variabile booleana **x** a **false** (vale **true** quando ci si trova all'interno di una coppia di parentesi)
- Finché la stringa non è finita
  - leggi nella stringa il carattere più a destra non ancora letto
  - se è una parentesi
    - se è una parentesi aperta
      - se **x** è **false** poni **x** = **true** e memorizza il tipo di parentesi
      - altrimenti errore (parentesi annidate...)
    - altrimenti (è una parentesi chiusa…)
      - se **x** è **false** errore (parentesi chiusa senza aperta...)
      - altrimenti se corrisponde a quella memorizzata poni **x** = **false** (la parentesi è stata chiusa…)
      - altrimenti errore (parentesi non corrispondenti)
- Se **x** è **true**, errore (parentesi aperta senza chiusa)

```
public static int checkWithoutNesting(String s)
{
   boolean inBracket = false;
   char bracket = '0'; // un valore qualsiasi
   for (int i = 0; i < s.length(); i++)
   {  char c = s.charAt(i);
      if (isBracket(c))
         if (isOpeningBracket(c))
            if (!inBracket)
            {  inBracket = true;
               bracket = c;
            }
            else return 1; //Errore
         else
            if (!inBracket) return 2; //Errore
            else if(areMatching(bracket, c))
               inBracket = false;
            else return 3; //Errore
   }
   if (inBracket) return 4; //Errore
   return 0; // OK }
```

```
private static boolean isOpeningBracket(char c)
{  return c == '(' || c == '[' || c == '{';
}
private static boolean isClosingBracket(char c)
{  return c == ')' || c == ']' || c == '}';
}
private static boolean isBracket(char c)
{  return isOpeningBracket(c)
          || isClosingBracket(c);
}
private static boolean areMatching(char c1,
   char c2)
{  return c1 == '(' && c2 == ')' ||
          c1 == '[' && c2 == ']' ||
          c1 == '{' && c2 == '}';
}
```

Esercizio: Controllo parentesi

- Cerchiamo di risolvere il caso più generale, in cui le parentesi di vario tipo possono essere annidate

```
a + [ c + (g + h) + (f + z) ]
```

- In questo caso non è più sufficiente memorizzare il tipo dell'ultima parentesi che è stata aperta, perché ci possono essere più parentesi aperte che sono in attesa di essere chiuse
  - quando si chiude una parentesi, bisogna controllare se corrisponde al tipo della parentesi in attesa che è stata aperta ***più recentemente***

Esercizio: Controllo parentesi

- Possiamo quindi risolvere il problema usando una pila
- Effettuando una scansione della stringa da sinistra a destra
  - inseriamo nella pila le parentesi aperte
  - quando troviamo una parentesi chiusa, estraiamo una parentesi dalla pila (che sarà quindi l'ultima ad esservi stata inserita) e controlliamo che i tipi corrispondano, segnalando un errore in caso contrario
    - se ci troviamo a dover estrarre da una pila vuota, segnaliamo l'errore (parentesi chiusa senza aperta)
  - se al termine della stringa la pila non è vuota, segnaliamo l'errore (parentesi aperta senza chiusa)

```
public static int checkWithNesting(String s)
{  Stack st = new GrowingArrayStack();
   for (int i = 0; i < s.length(); i++)
   {  char c = s.charAt(i);
      if (isBracket(c))
         if (isOpeningBracket(c))
            st.push(new Character(c));
         else
            try
            {  Object obj = st.pop();
               Character ch = (Character)obj;
               char cc = ch.charValue();
               if (!areMatching(cc, c))
                  return 3; //Errore
            } catch (EmptyStackException e)
            {  return 2; } //Errore
   }
   if (!st.isEmpty()) return 4; //Errore
   return 0;
}
```

Estrarre oggetti da una struttura dati

- Abbiamo visto che le strutture dati generiche, definite in termini di **Object**, sono molto comode perché possono contenere oggetti di qualsiasi tipo
- Sono però un po' scomode nel momento in cui effettuiamo l'estrazione (o l'ispezione) di oggetti in esse contenuti
  - viene sempre restituito un riferimento di tipo **Object**, indipendentemente dal tipo di oggetto effettivamente restituito
  - si usa un cast per ottenere un riferimento del tipo originario

```
Object obj = st.pop();
Character ch = (Character)obj;
```

Estrarre oggetti da una struttura dati

```
Character ch = (Character)st.pop();
```

- Sappiamo che serve il cast perché l'operazione di assegnamento è potenzialmente pericolosa
- Il programmatore si assume la responsabilità di inserire nella struttura dati oggetti del tipo corretto
- Cosa succede se è stato inserito un oggetto che NON sia di tipo **Character**?
  - viene lanciata l'eccezione **ClassCastException**
- Possiamo scrivere codice che si comporti in modo più sicuro?

Estrarre oggetti da una struttura dati

- Ricordiamo che le eccezioni la cui gestione non è obbligatoria, come **ClassCastException**, possono comunque essere gestite!

```
try
{  Character ch = (Character)st.pop();
} catch (ClassCastException e)
{  // gestione dell'errore
}
```

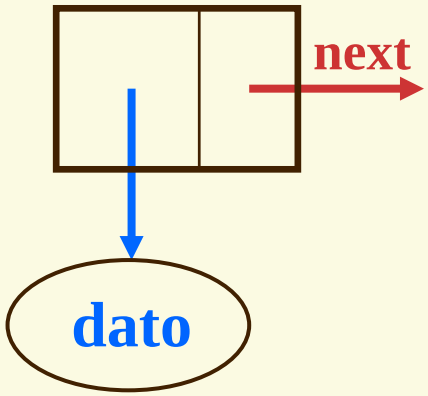
- In alternativa si può usare l'operatore **instanceof**

```
Object obj = st.pop();
if (obj instanceof Character)
   Character ch = (Character)obj;
else
   // gestione dell'errore
```

Lezione XXVIII
Ma 15-Nov-2005

Lista Concatenata
(Linked List)

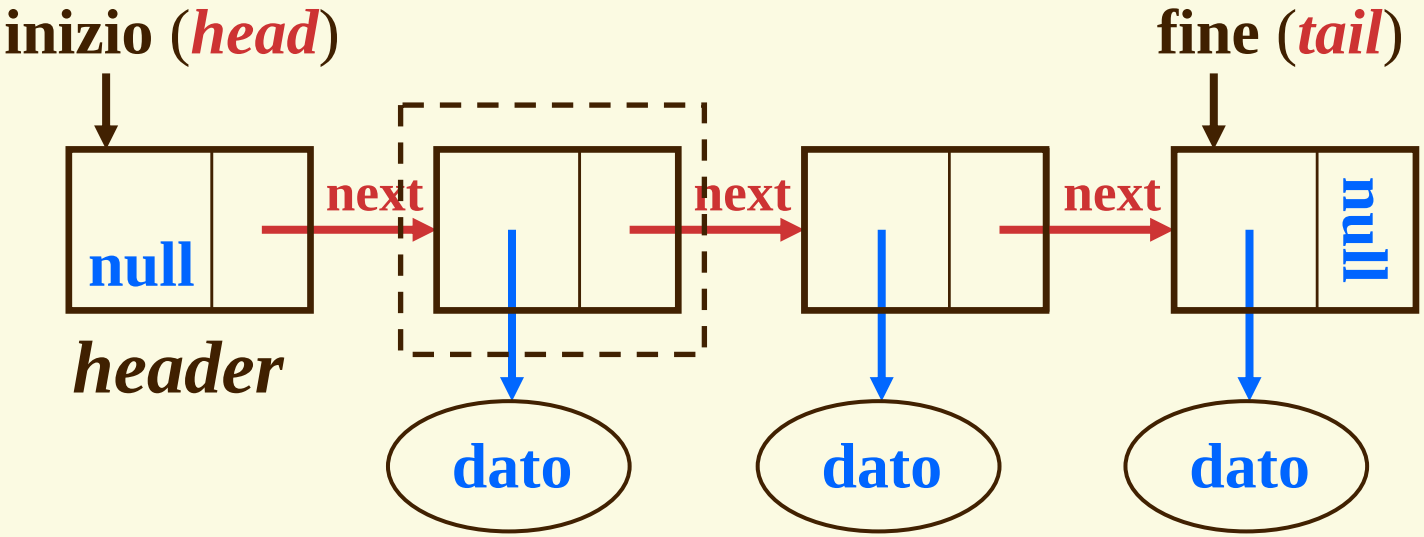
Catena (*linked list*)



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

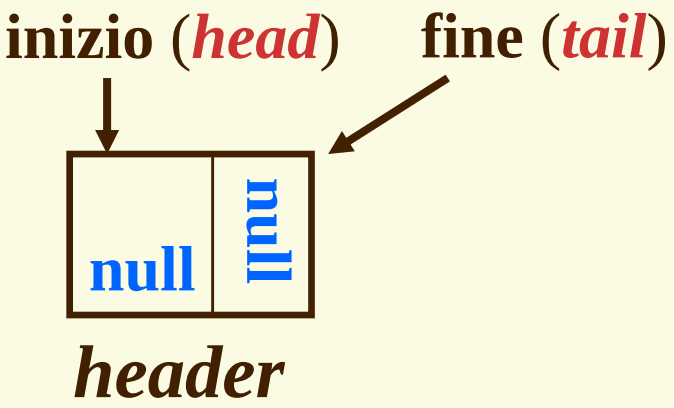
- Una catena è un insieme ***ordinato*** di ***nodi***
  - ogni nodo è un oggetto che contiene
    - un riferimento a un elemento (***il dato***)
    - un riferimento al nodo ***successivo*** nella catena (**next**)

Catena



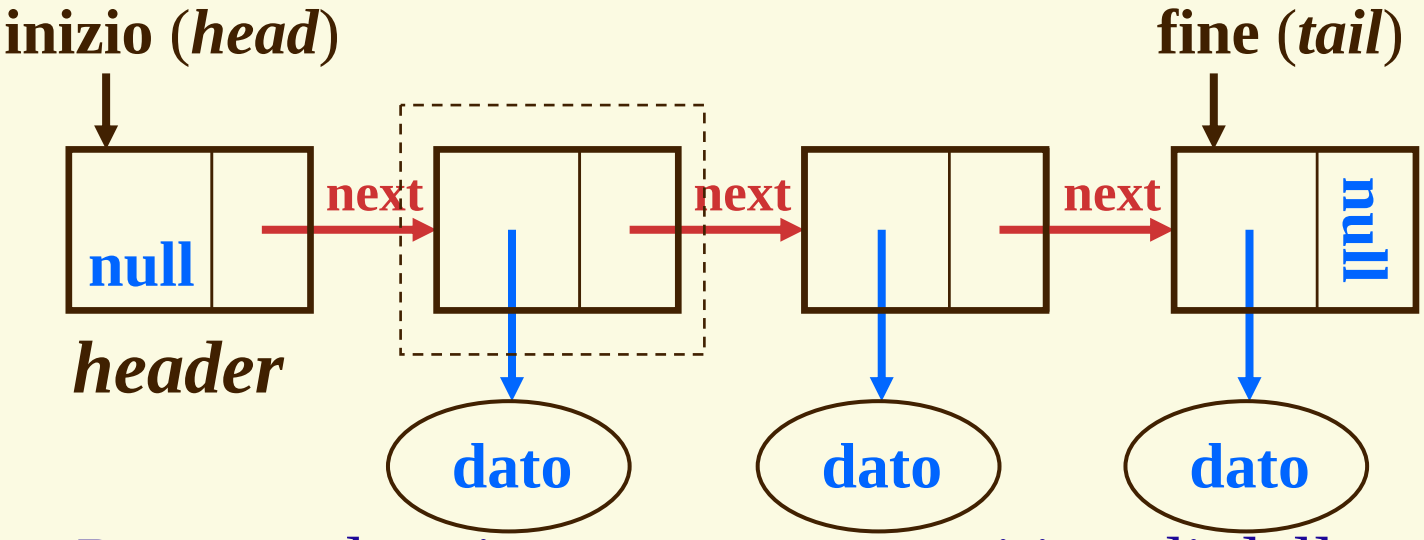
- Per agire sulla catena è sufficiente memorizzare il ***riferimento al suo primo nodo***
  - è comodo avere anche un riferimento all'ultimo nodo
- Il campo **next** dell'ultimo nodo contiene **null**
- Vedremo che è comodo avere un primo nodo senza dati, chiamato ***header***

Catena vuota



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

Catena



- Per accedere in sequenza a tutti i nodi della catena 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 catena

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

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

   public ListNode()
   {  this(null, 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 a oggetti del tipo che sta definendo***
- Ciò è perfettamente lecito e ***si usa molto spesso*** quando si rappresentano “strutture a definizione ricorsiva” come la catena

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

Incapsulamento eccessivo?

- Se invece usiamo l'incapsulamento
  - è sufficiente inserire l'enunciato di visualizzazione all'interno dei metodi **set()** che interessano
  - 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

Catena

- I metodi utili per una catena 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
- Si osservi che non vengono mai restituiti né ricevuti riferimenti ai ***nodi***, ma sempre ai ***dati*** contenuti nei nodi

Catena

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

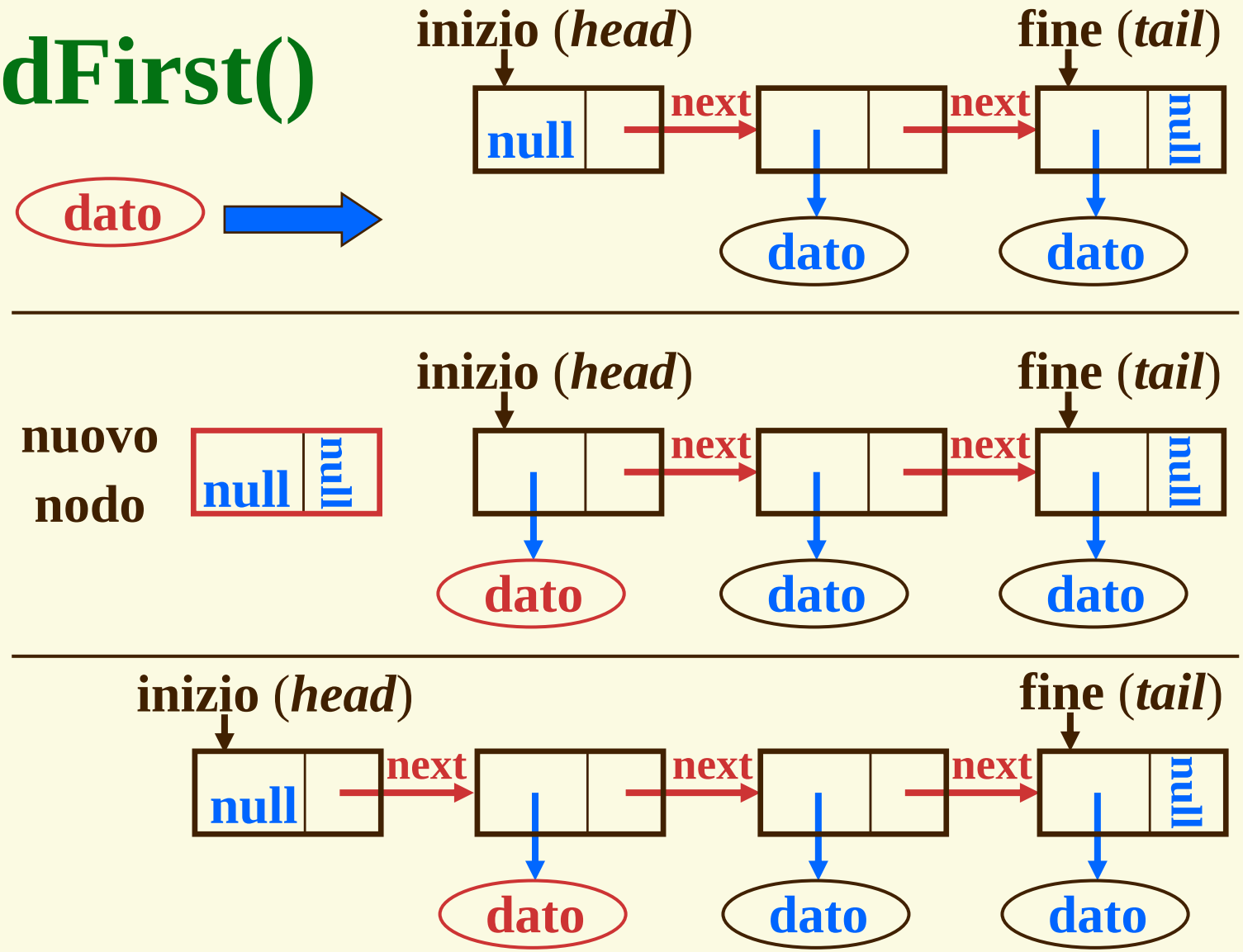
Catena

```
public class LinkedList implements Container
{
   // parte privata
   private ListNode head, tail;

   public LinkedList()
   {  makeEmpty();
   }

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

   public boolean isEmpty()
   {  return (head == tail);
   }
...
}
```

Catena

```
public class LinkedList implements Container
{
   ...
   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

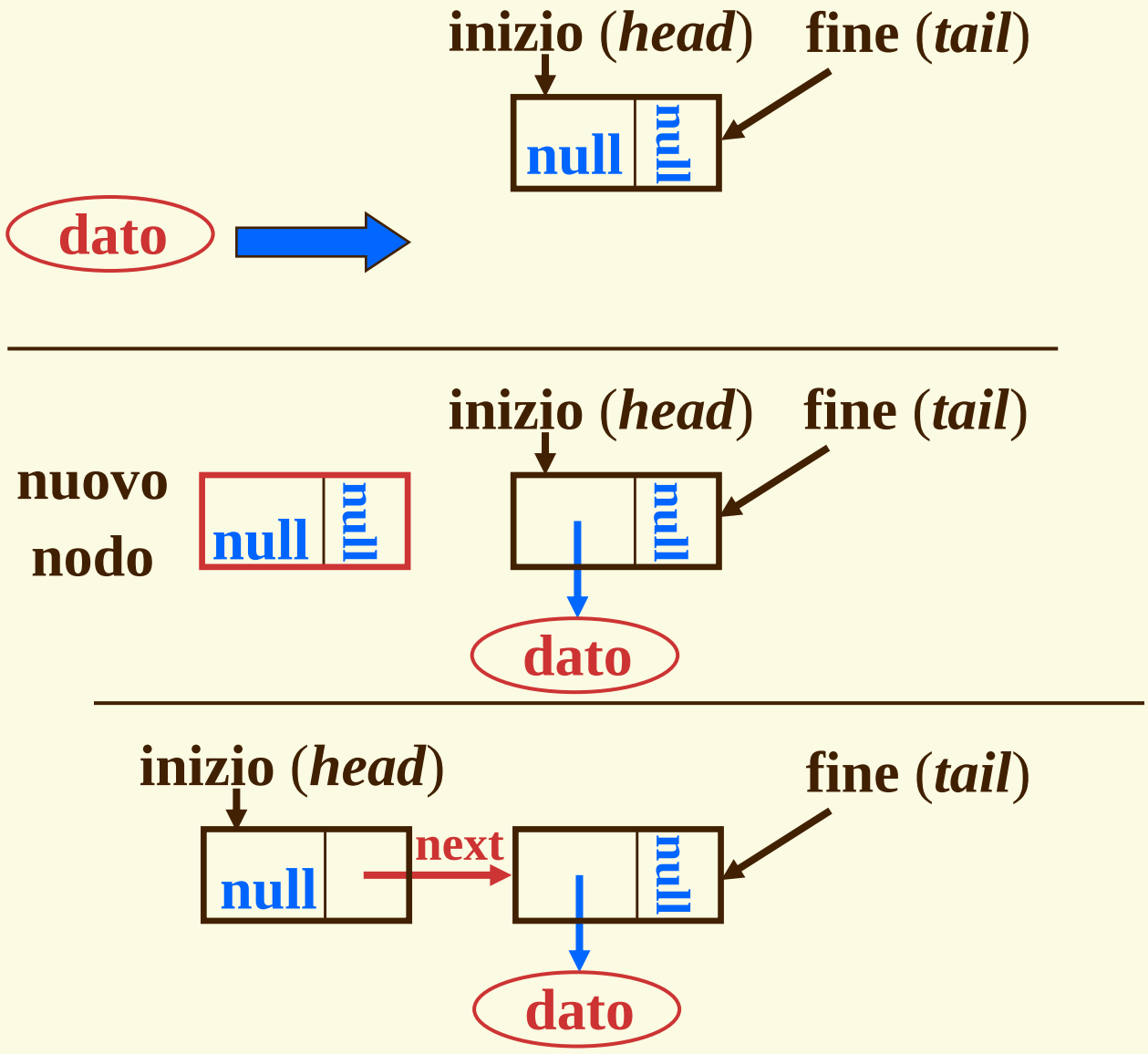
addFirst()

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

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

addFirst()

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



addFirst()

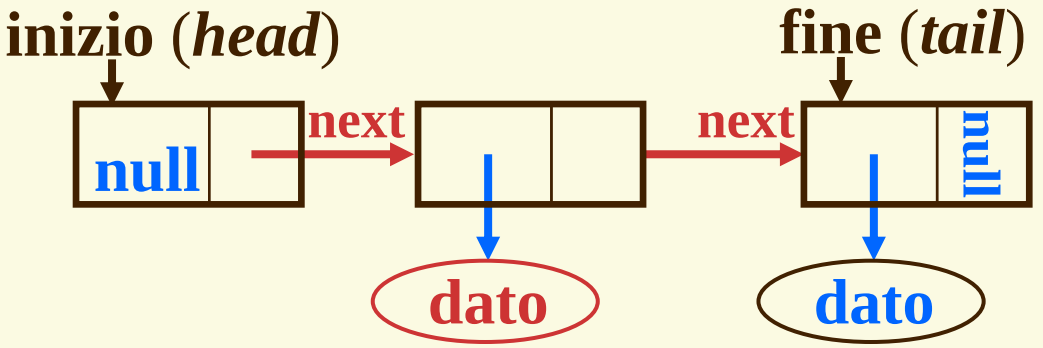
```
public void addFirst(Object e) {
   head.setElement(e);
   ListNode n = new ListNode();
   n.setNext(head);
   head = n;
}
```

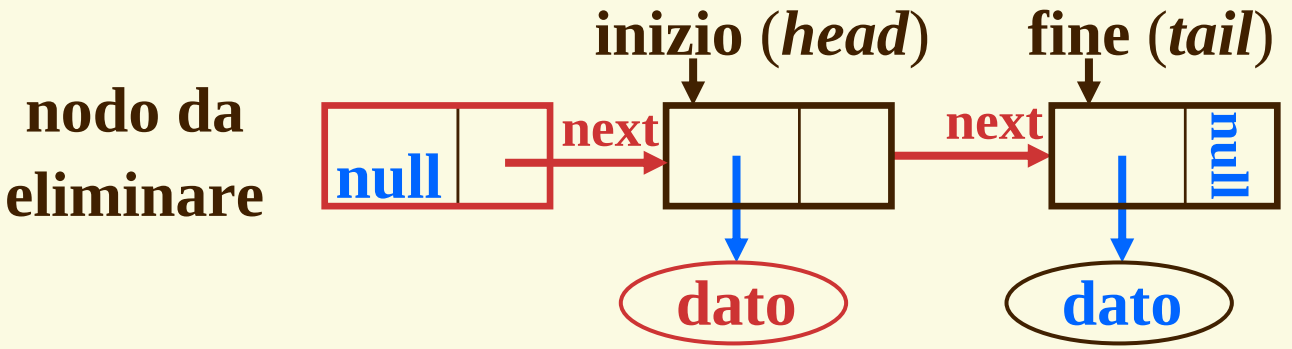
- Il codice di questo metodo si può esprimere anche in modo più conciso

```
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);
}
```

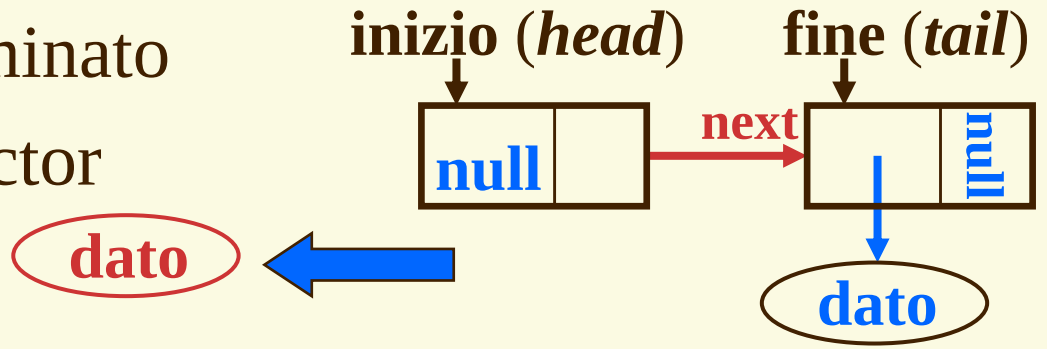
- È più “professionale”, anche se meno leggibile

removeFirst()





Il nodo viene eliminato dal garbage collector



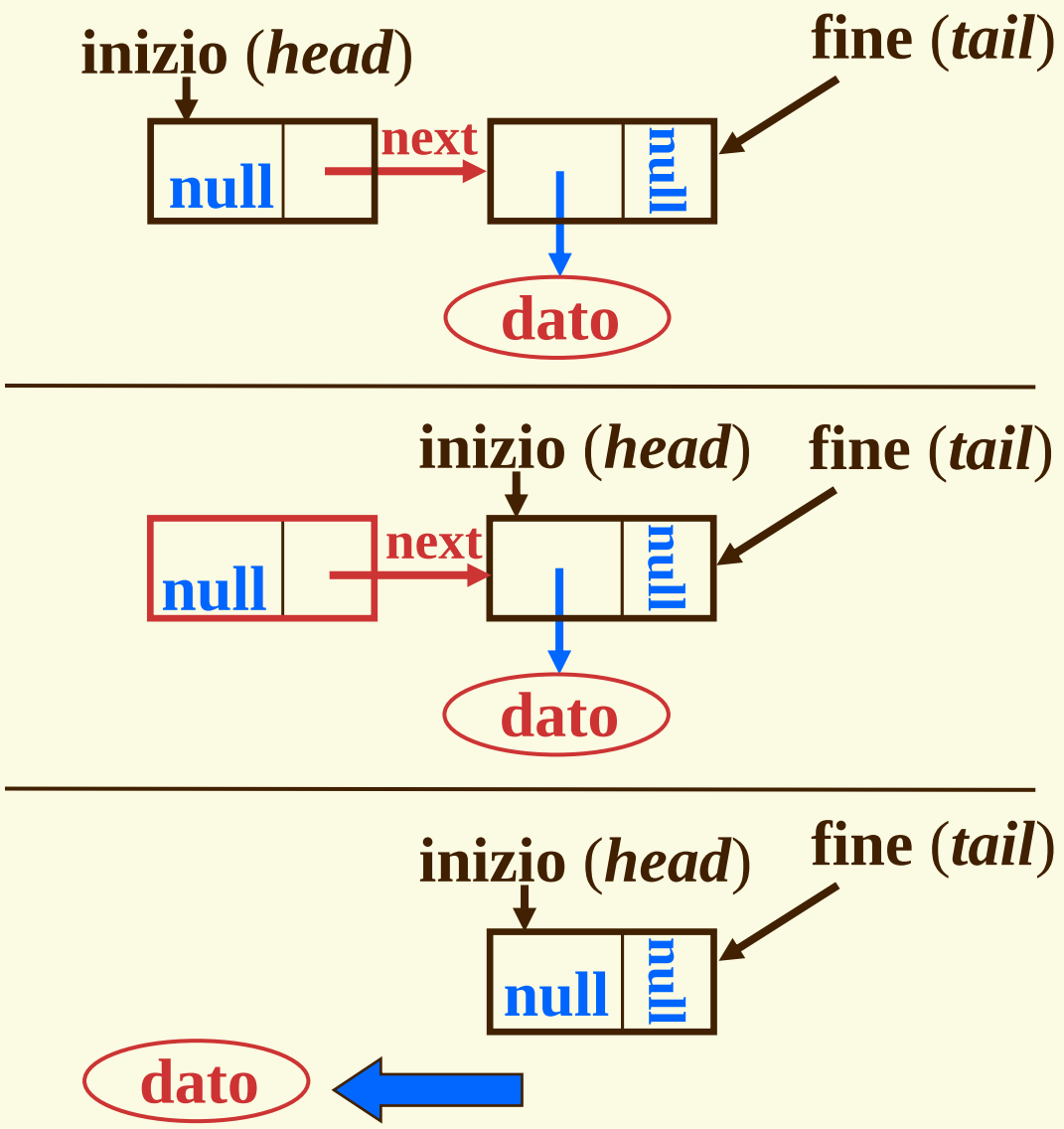
removeFirst()

```
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;
   }
}
```

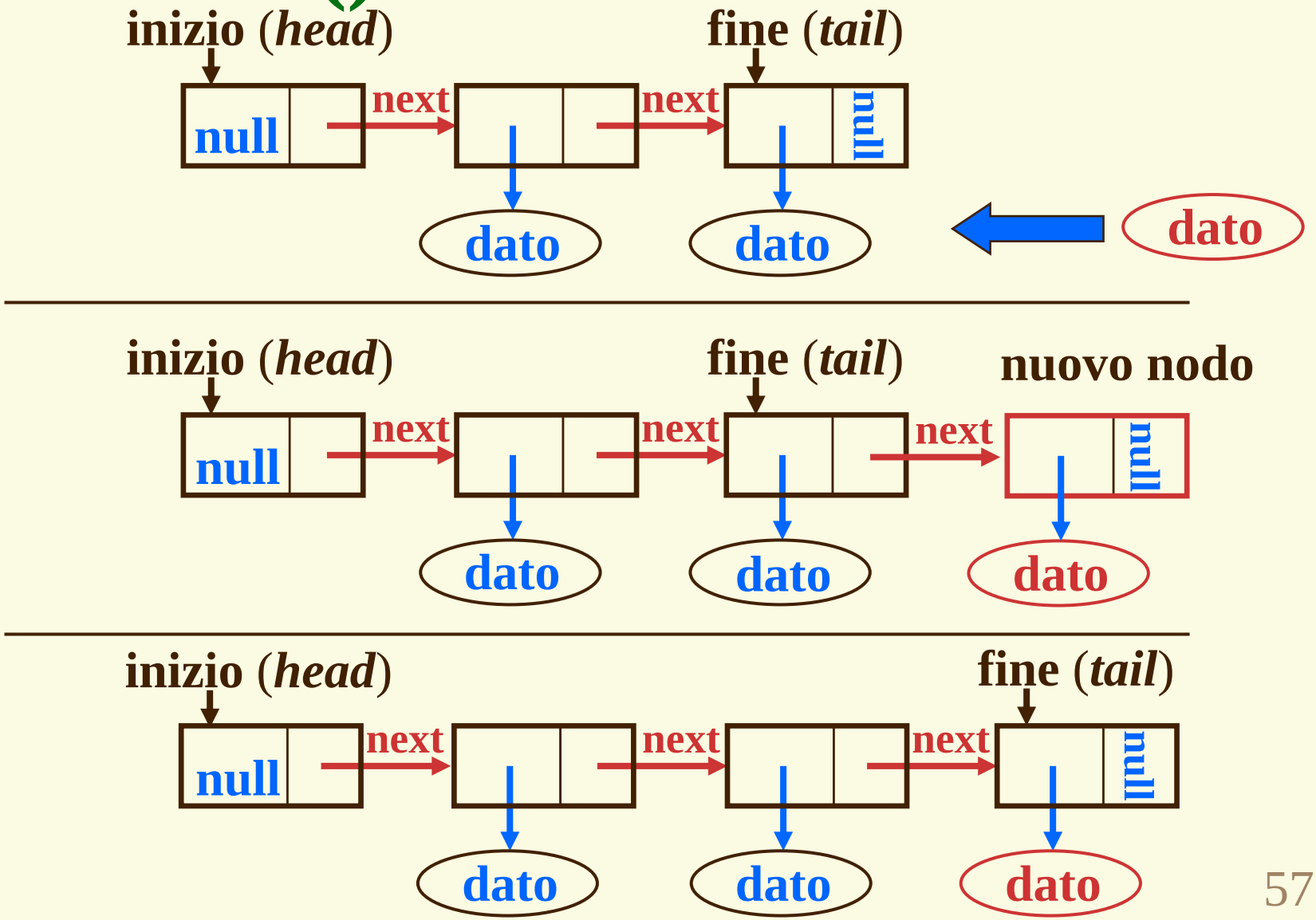
- L'operazione è **O(1)**

removeFirst()

- Verifichiamo che tutto sia corretto anche rimanendo con una *catena 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

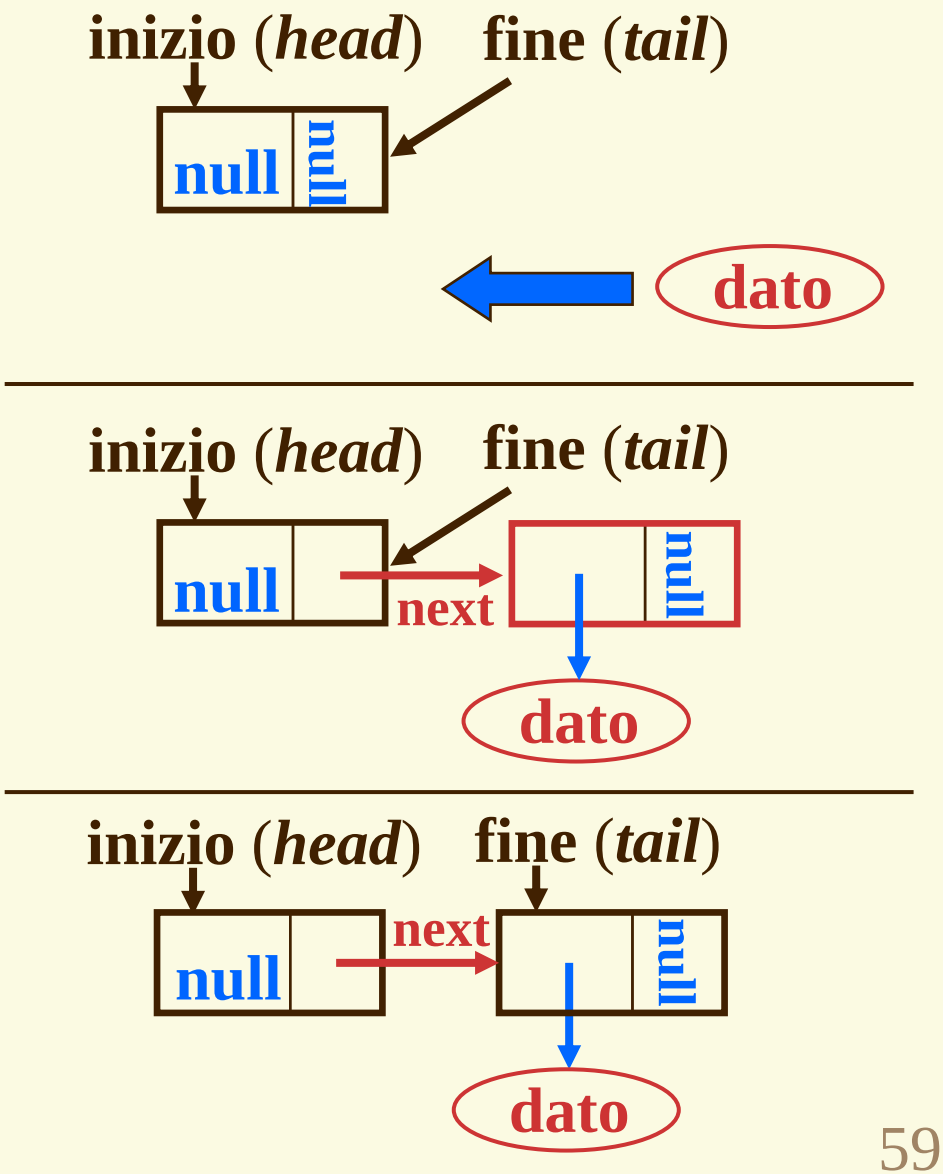
addLast()

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

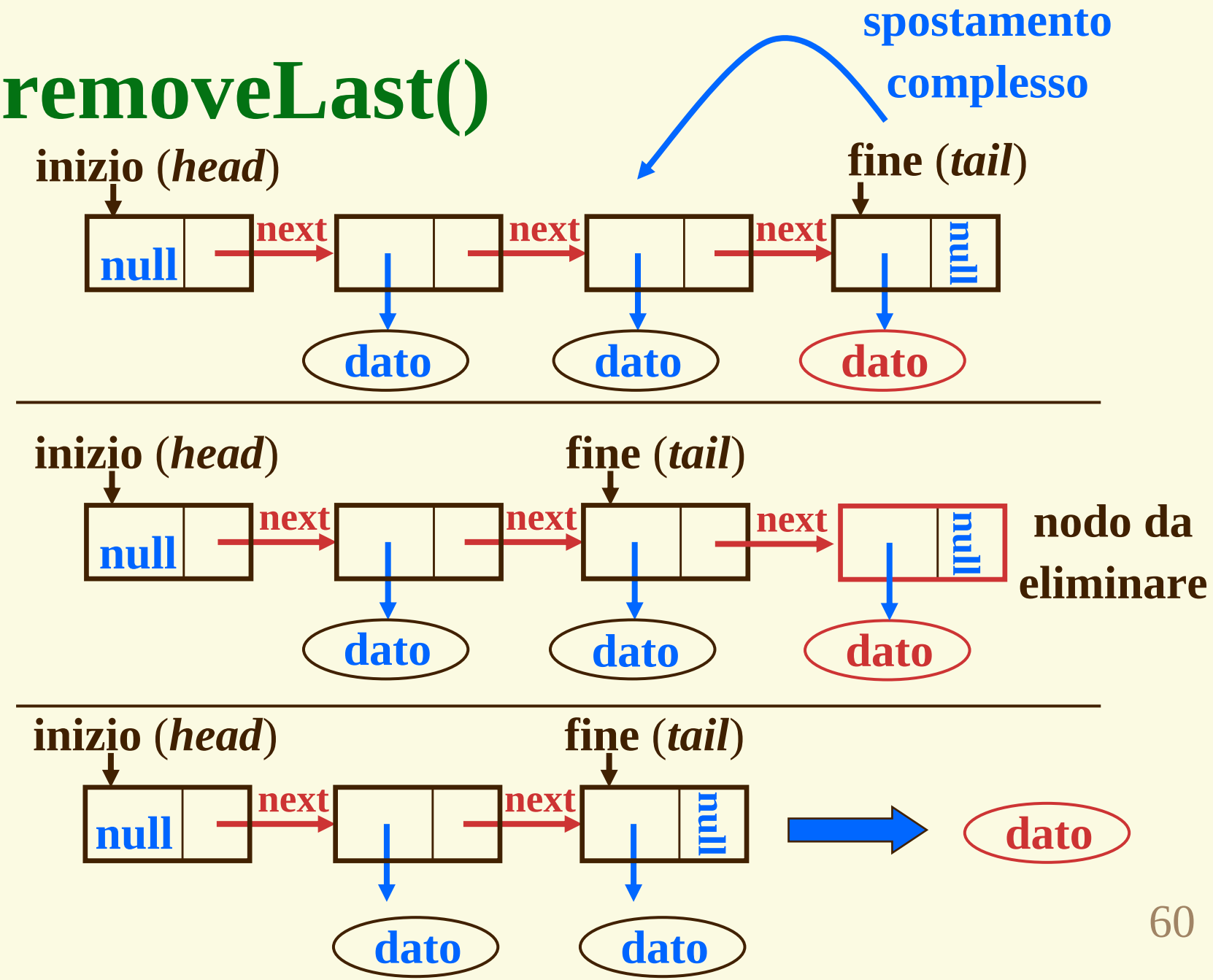
- Non esiste il problema di “catena piena”
- Anche questa operazione è **O(1)**

addLast()

- Verifichiamo che tutto sia corretto anche inserendo in una *catena 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

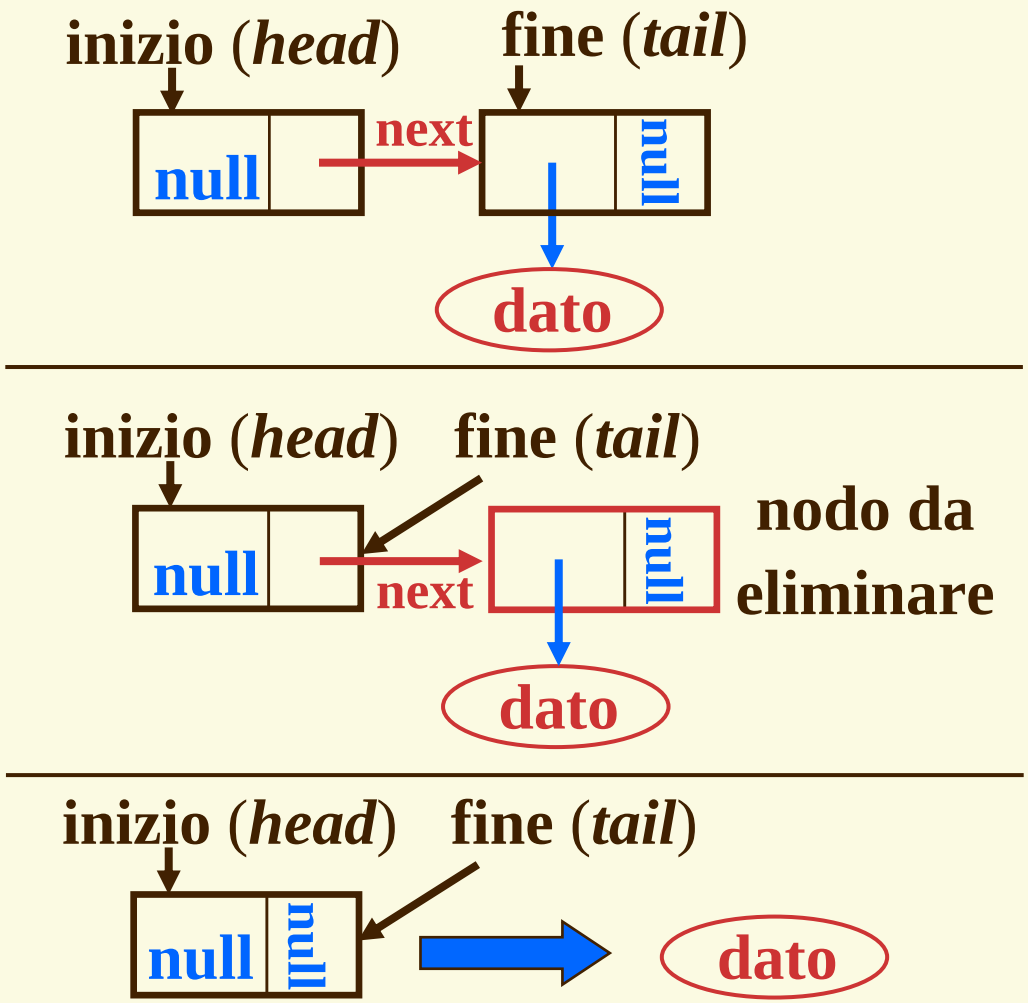
removeLast()

```
public class LinkedList ...
{  ...
   public Object removeLast() {
      Object e = getLast();
      // bisogna cercare il penultimo nodo
      // partendo dall'inizio e 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;
   }
}
```

Operazione **O(n)**

removeLast()

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



Header della catena

- La presenza del nodo ***header*** nella catena rende più semplici i metodi della catena stessa
  - 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 catena questo spreco, in percentuale, è trascurabile

Prestazioni della catena

- Tutte le operazioni sulla ***catena*** 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 catena 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 catena

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

Catena

```
public class LinkedList
{
   private ListNode head, tail;

   public LinkedList(){… }
   public void makeEmpty(){ }  // O(1)
   public boolean isEmpty(){ } // O(1)
   public Object getFirst(){ } // O(1)
   public Object getLast() { } // O(1)
   public void addFirst(Object obj) { } // O(1)
   public Object removeFirst() { }      // O(1)
   public void addLast(Object obj) { }  // O(1)
   public Object removeLast(){ }        // O(n)
}
```

Classi interne

- Osserviamo che la classe **ListNode**, usata dalla catena, non viene usata al di fuori della catena stessa
  - la catena non restituisce mai riferimenti a **ListNode**
  - la catena non riceve mai riferimenti a **ListNode**
- Per il principio dell'incapsulamento (***information hiding***) sarebbe preferibile che questa classe e i suoi dettagli non fossero visibili all'esterno della catena
  - in questo modo una modifica della struttura interna della catena e/o di **ListNode** non avrebbe ripercussioni sul codice scritto da chi usa la catena

Classi interne

- Il linguaggio Java consente di ***definire classi all'interno di un'altra classe***
  - tali classi si chiamano ***classi interne*** (***inner classes***)
- L'argomento è molto vasto
- 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
   {  ... }
}
```

Catena

```
public class LinkedList
{  private ListNode head, tail;

   public LinkedList(){ }
   public void makeEmpty(){ }  // O(1)
   public boolean isEmpty(){ } // O(1)
   public Object getFirst(){ } // O(1)
   public Object getLast() { } // O(1)
   public void addFirst(Object obj) { } // O(1)
   public Object removeFirst() { }      // O(1)
   public void addLast(Object obj) { }  // O(1)
   public Object removeLast(){ }        // O(n)

   private class ListNode
   {
   . . .
   }
}
```

Utilizzo di catene

Pila realizzata con una catena

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

Pila realizzata con una catena

```
public class LinkedListStack implements Stack
{  private LinkedList list;

   public LinkedlinkStack()
   {  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 una catena

- Anche una coda può essere realizzata usando una catena invece di un array
- È sufficiente inserire gli elementi a un'estremità della catena e rimuoverli all'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 una catena

```
public class LinkedListQueue implements Queue
{  private LinkedList list;

   public LinkedlinkQueue()
   {  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();
   }
}
```

Algoritmi per catene

Catena: conteggio elementi

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

```
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 catena è vuota (corretto)
      return size;
   }
}
```

Algoritmi per catene

- Osserviamo che per eseguire algoritmi sulla catena è necessario ***aggiungere metodi all'interno della classe*** **LinkedList**, che è l'unica ad avere accesso ai nodi della catena
  - ad esempio, un metodo che verifichi la presenza di un particolare oggetto nella catena (algoritmo di ricerca)
- Questo limita molto l'utilizzo della catena come struttura dati definita una volta per tutte…
  - vogliamo che la catena fornisca ***uno strumento per accedere ordinatamente a tutti i suoi elementi***

Algoritmi per catene

- L'idea più semplice è quella di fornire un metodo **getHead()**

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

  ma questo viola completamente l'incapsulamento, perché diventa possibile modificare direttamente lo stato interno della catena, anche in modo da non farla più funzionare correttamente
- Fortunatamente **non funziona**, perché **ListNode** è una classe interna

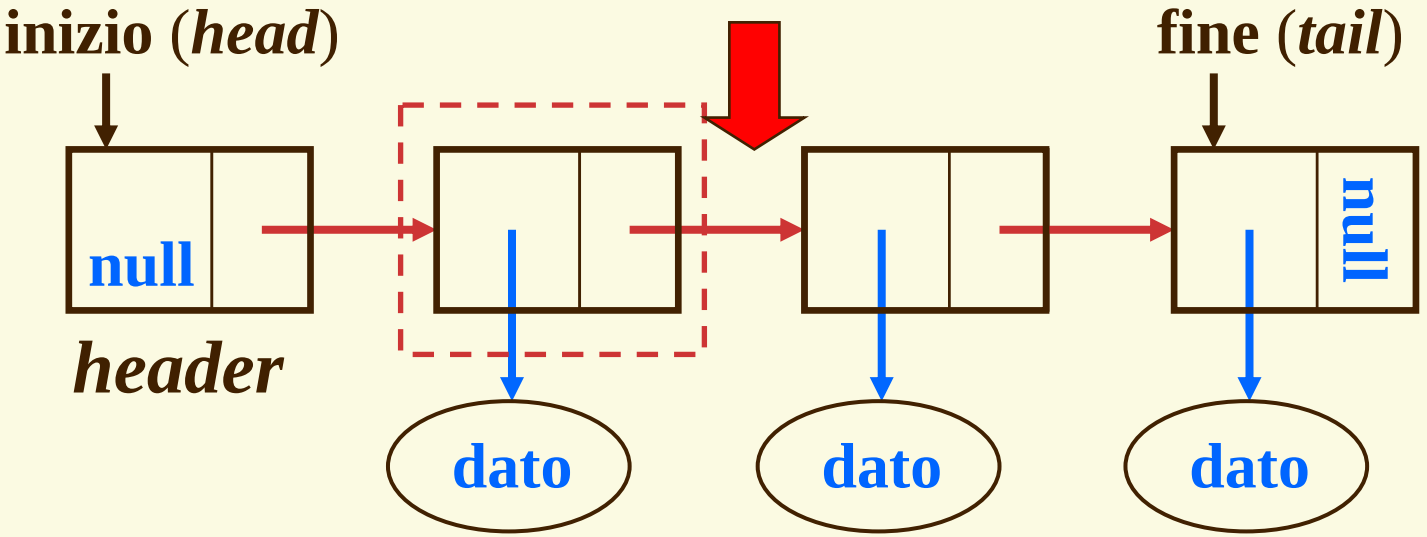
Lezione XXIX
Me 16-Nov-2005

Iteratore in una Lista Concatenata

Iteratore in una catena

- La soluzione del problema della scansione della lista senza interagire con i nodi è quella di fornire all'utilizzatore della catena 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
  - un iteratore rappresenta in astratto ***il concetto di posizione*** all'interno di una catena
  - 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 catena



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

Iteratore in una catena

```
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
   // catena, 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 catena

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

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

e si può scandire la catena senza accedere ai nodi

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

Iteratore in una catena

- Come realizzare il metodo **getIterator()** nella catena?
  - 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 **ListIterator**
- Gli oggetti di tale classe vengono costruiti soltanto all'interno di **LinkedList** e vengono restituiti all'esterno soltanto tramite riferimenti a **ListIterator**
  - quindi possiamo usare una classe interna

Iteratore in una catena

- Per un corretto funzionamento dell'iteratore occorre concedere a tale oggetto il pieno accesso alla catena
  - 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 ...
{  ...
   public ListIterator getIterator()
   {
      return new LinkedListIterator(head);
   }
   private class LinkedListIterator
      implements ListIterator
   {  // nodo che precede la posizione attuale
      // (non è mai null)
      private ListNode current;
      // nodo precedente
      private ListNode previous;

      public LinkedListIterator(ListNode h)
      {
         current = h; previous = null;
      }
      ...
```

```
import java.util.NoSuchElementException
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();
      }
   }
}
```

Catena: conteggio elementi

- Possiamo quindi riscrivere il metodo di conteggio degli elementi contenuti in una catena, ma al di fuori della catena 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;
}
```

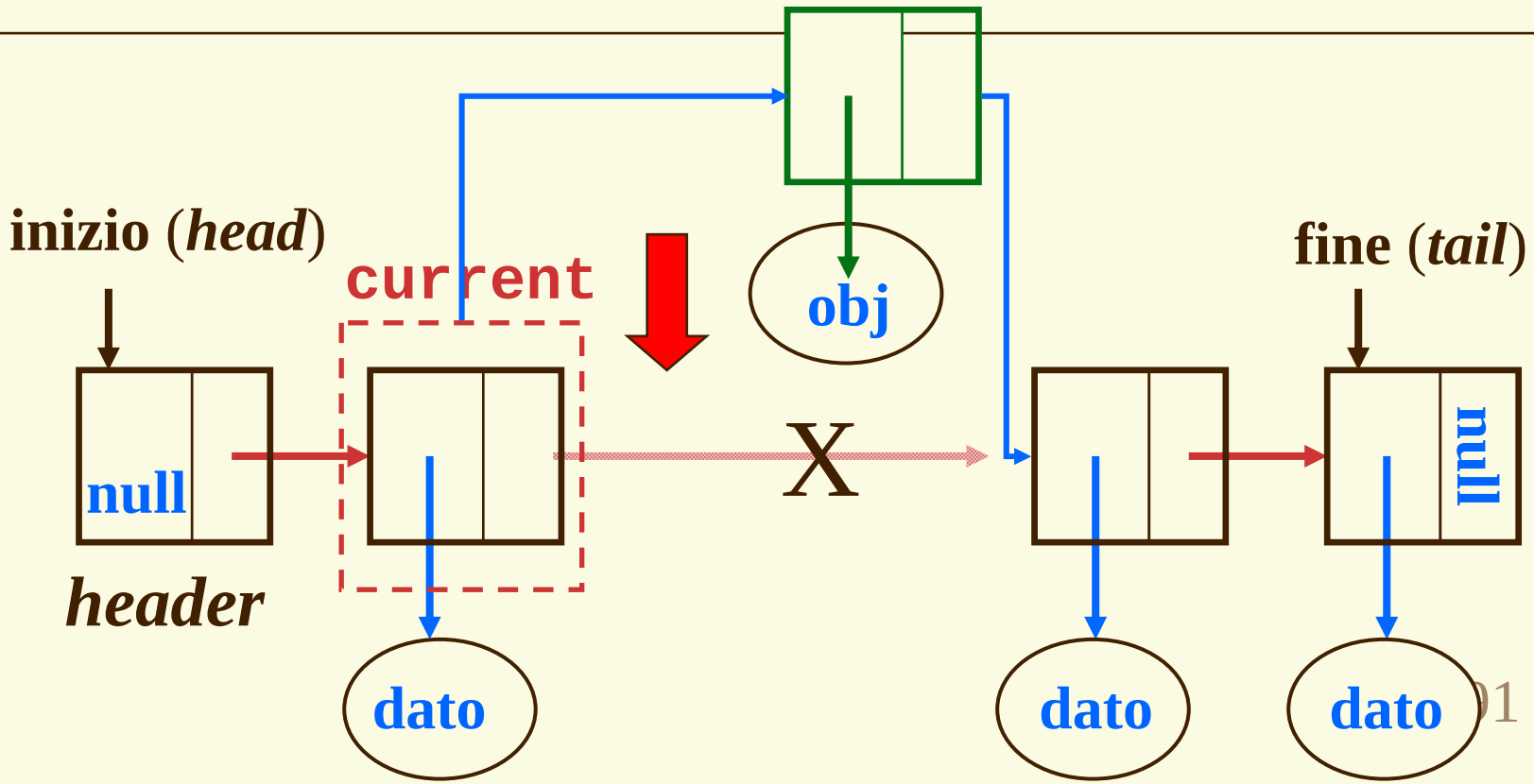
Catena: inserimento e rimozione

- Abbiamo visto l'inserimento e la rimozione di un elemento all'inizio e alla fine della catena
  - addFirst(), addlast()
- Vogliamo estendere le modalità di funzionamento della catena per poter inserire e rimuovere elementi in qualsiasi punto della catena stessa
  - abbiamo di nuovo il problema di ***rappresentare il concetto di posizione***, la posizione in cui inserire il nuovo nodo nella catena o da cui rimuovere il nodo
- Usiamo di nuovo l'***iteratore***
  - dobbiamo però estenderne le funzionalità

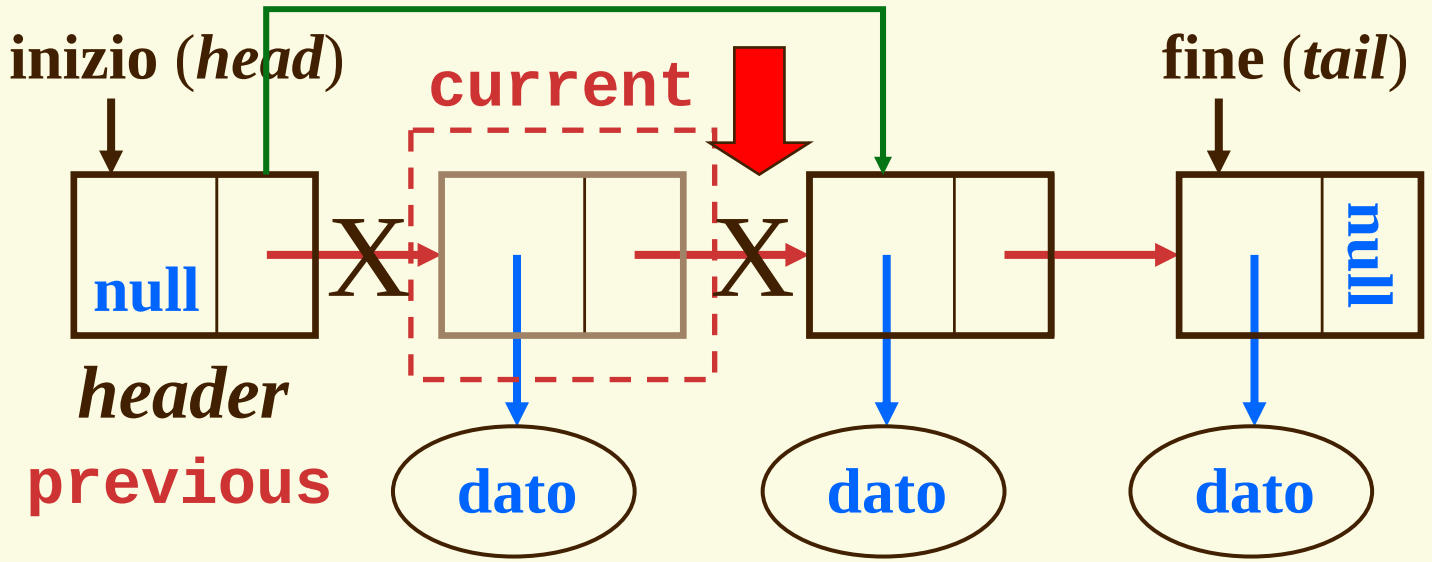
Iteratore in una catena

```
public interface ListIterator
{ boolean hasNext();
  Object next();

  // inserisce l'oggetto x in un nuovo nodo
  // che si aggiunge alla catena PRIMA della
  // posizione attuale,
  // senza modificare la posizione dell'iteratore
  void add(Object x);

  //elimina l'ultimo nodo esaminato da next()
  //senza modificare la posizione dell'iteratore;
  //può essere invocato solo dopo un'invocazione
  //di next() (lancia IllegalStateException)
  void remove();
}
```

```
private class LinkedListIterator . . .
{ private ListNode previous;
  //non fa avanzare l'iteratore
  //diverso da C. S. Horstmann pp 610
  public void add(Object obj)
  { ListNode n = new ListNode(obj, current.getNext());
    current.setNext(n);
    previous = null;
  }
}
```



```
  private class LinkedListIterator . . .
  { private ListNode previous;
  ...
    public void remove()
    {
       if (previous == null)
            throw new IllegalStateException();
       previous.setNext(current.getNext());
       current = previous;
       previous = null; // non si puo' chiamare
    }                   // remove() due volte
  }                     // di seguito
}
```



Copia da una catena all'altra

```
//Copia da una lista a un'altra
public static void copy(LinkedList from,
   LinkedList to)
{
  //le due liste sono lo stesso oggetto
  if(from == to) return;

  to.makeEmpty(); //Vuoto la seconda
  ListIterator fromItr = from.getIterator();
  ListIterator toItr = to.getIterator();
  while (fromItr.hasNext())
  {
     toItr.add(fromItr.next());
     toItr.next();
  }
}
```

Più iteratori sulla stessa lista

- Se vengono creati ***più iteratori che agiscono sulla stessa lista***
  - cosa perfettamente lecita, invocando più volte il metodo **getIterator()**

  ciascuno di essi mantiene il proprio stato, cioè memorizza la propria posizione nella lista
- Ciascun iteratore può muoversi nella lista indipendentemente dagli altri
  - occorre però usare qualche cautela quando si usano gli iteratori per ***modificare*** la lista

Più iteratori sulla stessa lista

```
List list = new LinkedList()
ListIterator iter1 = list.getIterator();
iter1.add(new Integer(1));
ListIterator iter2 = list.getIterator();
// iter2 punta al primo elemento in lista, 1
ListIterator iter3 = list.getIterator();
iter3.add(new Integer(2));
// il primo elemento della lista è diventato 2
System.out.println(iter2.next()); // 1
// iter2 non funziona correttamente!!
```

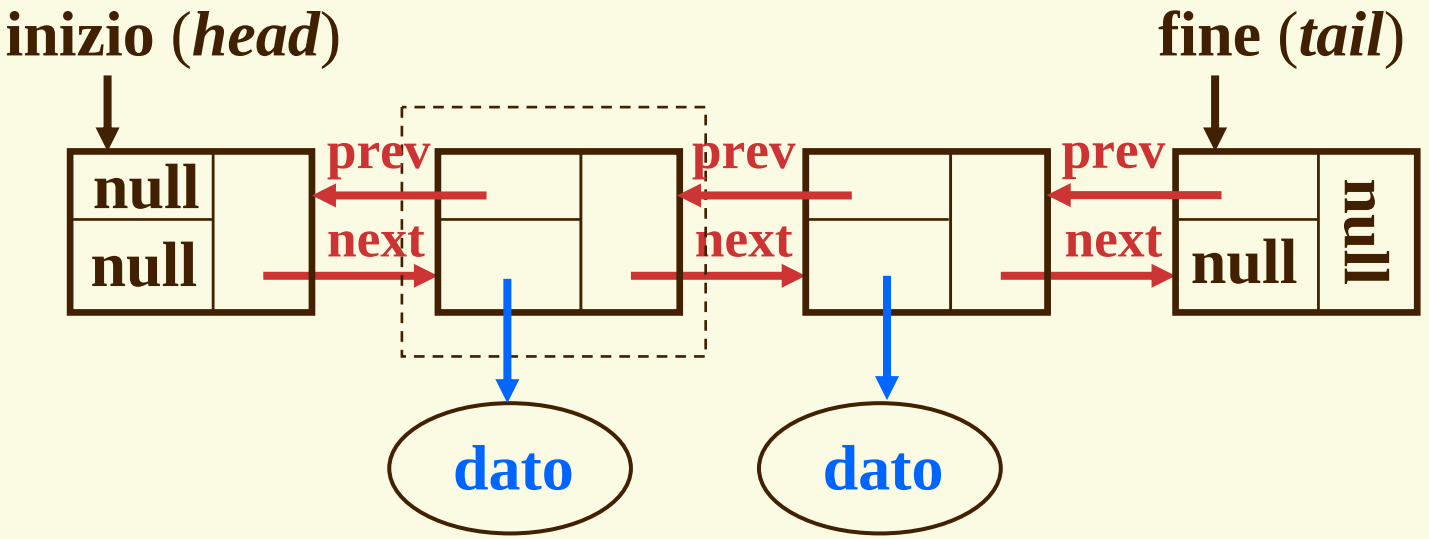
- L'argomento è molto complesso, ma viene qui soltanto accennato per invitare alla cautela nell'uso di più iteratori contemporaneamente
  - ***nessun problema se si usano solo in lettura***

Catena doppia
(*doubly linked list*)

Catena doppia

- La ***catena doppia*** (lista doppiamente concatenata, ***doubly linked list***) non è un nuovo ADT, ma è una struttura dati per la realizzazione di ADT
- Una catena doppia 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 della lista (**next**)
    - un riferimento al nodo precedente della lista (**prev**)

Catena doppia



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

Catena doppia

- Tutto quanto detto per la catena (semplice) può essere agevolmente esteso alla catena doppia
- Il metodo **removeLast()** diventa **O(1)** come gli altri metodi
- L'iteratore per la catena doppia avrà anche un metodo per retrocedere di una posizione, oltre a quello per avanzare
  - i metodi di inserimento e rimozione si complicano

ADT Lista

Lista

- Dopo aver introdotto l'***iteratore*** completo per una catena, possiamo osservare che l'interfaccia **ListIterator**, oltre a consentire la manipolazione di una catena, definisce anche il comportamento astratto di un contenitore in cui
  - i dati sono disposti in sequenza (cioè per ogni dato è definito un precedente e un successivo)
  - nuovi dati possono essere inseriti in ogni punto della sequenza
  - dati possono essere rimossi da qualsiasi punto della sequenza

Lista

- Un contenitore avente un tale comportamento può essere molto utile, per cui si definisce un tipo di dati astratto, detto ***lista***, con la seguente interfaccia

```
public interface List extends Container
{  ListIterator getIterator();
}
```

- Attenzione a non confondere la ***lista*** con la ***lista concatenata*** (o catena)

Lista

- A questo punto potremmo ridefinire la catena

```
public class LinkedList implements List
{   ...
}
```

ma si noti che non è necessario realizzare una lista mediante una catena, perché ***nella definizione della lista non vengono menzionati i nodi***

- è infatti possibile definire una lista che usa un array come struttura di memorizzazione dei dati

Dati in sequenza

- Abbiamo quindi visto diversi tipi di contenitori per dati in sequenza, rappresentati dagli ***ADT***
  - pila
  - coda
  - lista con iteratore
- Per realizzare questi ADT, abbiamo usato diverse ***strutture dati***
  - array
  - Lista concatenata

Liste con rango e posizionali

Da F. Bombi

Modificato da A. Luchetta

Rango e posizione

- Continuiamo a parlare delle liste aggiungendo qualche particolare
- Le operazioni più generali che vogliamo effettuare su di una lista riguardano l'inserimento e l'eliminazione di un nuovo elemento in una posizione qualsiasi
- La posizione di un elemento nella lista può essere indicata in modo assoluto attraverso il ***rango*** di un elemento
- Il rango è un ***indice*** che assume il valore ***0*** (zero) per l'elemento in testa alla lista e assume il valore ***i+1*** per l'elemento che segue l'elemento di rango ***i***

Una lista con rango = vettore

- Conveniamo di chiamare ***vettore*** una lista con rango
- Il ***vettore*** è una generalizzazione del concetto di array in quanto:
  - Ha una lunghezza variabile
  - È possibile aggiungere e togliere elementi in qualsiasi posizione del vettore
  - È possibile accedere (in lettura e scrittura) al valore di un elemento noto il suo rango
- Un modo naturale per rappresentare un vettore è quello di utilizzare un ***array parzialmente riempito***
- In `java.util` esiste la classe `Vector` caratterizzata da una ricca dotazione di funzionalità che realizza una ***lista con rango***

Limiti nell'uso di un vettore

- La realizzazione di un ***vettore*** utilizzando un ***array*** soffre di un limite
- Mentre le operazioni di ***accesso*** ad un elemento dato il rango (`get` e `set`) richiedono un tempo ***O(1)***, le operazioni di ***inserimento*** e di ***eliminazione*** di un elemento dato il rango (`add` e `remove`) richiedono, in media, un tempo ***O(n)***
- Perché ci preoccupiamo di questo?
- Rispondiamo con un esempio: vogliamo eliminare gli elementi ripetuti da una lista con il seguente algoritmo:
  - Consideriamo gli elementi della lista dal primo al penultimo
  - Per ogni elemento consideriamo gli elementi che lo seguono, se troviamo un elemento uguale all'elemento corrente lo eliminiamo dalla lista

Eliminare i doppioni - analisi

- Se la lista contiene inizialmente ***n*** elementi al primo passo effettuiamo ***n-1*** operazioni, al successivo ***n-2***, al passo i-esimo ***n-i*** operazioni
- L'algoritmo richiede $O(n^2)$ passi e ***quindi*** ha una complessità temporale $O(n^2)$ (?!?)
- Mentre la prima affermazione è vera la seconda è ***sbagliata*** perché, nel caso si debba eliminare un elemento, l'operazione elementare non richiede un tempo costante (indipendente dalla taglia del problema) ma un tempo $O(n)$ e quindi l'algoritmo avrà una complessità temporale $O(n^3)$
- La conclusione è dunque: vorremmo una rappresentazione della lista per la quale le ***operazioni elementari richiedano sempre un tempo costante, abbiano quindi una complessità temporale O(1)***

Classi e interfacce di `java.util`

- Le interfacce introdotte sono versioni ridotte a scopo didattico di interfacce e classi di `java.util` quali
  - **`List`**
  - **`LinkedList`**
  - **`Iterator`**
  - **`ListIterator`**
- Quando si debbano risolvere problemi reali si dovrà fare ricorso alle classi di libreria, in modo analogo a quanto suggerito con riferimento alla classe Vector discussa per la realizzazione di liste con rango
- Notare in particolare che il codice presentato a lezione è molto debole in presenza di errori nell'uso dell'iteratore

Uso di array bidimensionali

Uso di array bidimensionali

- Problema
  - stampare una tabella con i valori delle potenze $\mathbf{x}^{\mathbf{y}}$, per ogni valore di **x** tra 1 e 4 e per ogni valore di **y** tra 1 e 5

```
1     1     1     1     1
2     4     8    16    32
3     9    27    81   243
4    16    64   256  1024
```

e cerchiamo di risolverlo in modo generale, usando un array bidimensionale

```
/**
 visualizza una tabella con i valori delle potenze "x
 alla y", con x e y che variano indipendentemente tra 1
 e un valore massimo assegnato dall'utente. I dati
 relativi a ciascun valore di x compaiono su una riga,
 con y crescente da sx a dx e x dall'alto in basso.
*/
import java.util.Scanner;

public class TableOfPowers
{
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);
    System.out.println(
      "Calcolo dei valori di x alla y");
    System.out.print("Valore massimo di x: ");
    int maxX = in.nextInt();
    System.out.print("Valore massimo di y: ");
    int maxY = in.nextInt(); // (continua)
```

Uso di array bidimensionali

```
int maxValue =
      (int)Math.round(Math.pow(maxX, maxY));

    int columnWidth = 1 +
      Integer.toString(maxValue).length();

    int[][] powers = genPowers(maxX, maxY);
    printPowers(powers, columnWidth);

}

 // (continua)
```

- Notare l'utilizzo di metodi **private** per la scomposizione di un problema in sottoproblemi più semplici
  - in genere non serve preoccuparsi di pre-condizioni perché il metodo viene invocato da chi l'ha scritto

Uso di array bidimensionali

```
(continua)
  /**
     Genera un array bidimensionale con i
     valori delle potenze di x alla y
  */
  private static int[][] genPowers(int x,int y)
  {
     int[][] powers = new int[x][y];
     for (int i = 0; i < x; i++)
       for (int j = 0; j < y; j++)
         powers[i][j] =
           (int)Math.round(Math.pow(i+1,j+1));
      return powers;
   }
(continua)
```

Uso di array bidimensionali

```
  /** (continua)
    Visualizza un array bidimensionale di
    numeri interi con colonne di larghezza
    fissa e valori allineati a destra.
  */
  private static void printPowers(int[][] v, int width)
  {
    for (int i = 0; i < v.length; i++)
    {
      for (int j = 0; j < v[0].length; j++)
      {
        String s = Integer.toString(v[i][j]);
        while (s.length() < width)
          s = " " + s;

        System.out.print(s);
      }
      System.out.println(); //vado a capo
    }
  }
} //chiude la classe TableOfPowers
```

Esempio di programma ricorsivo

Esercizio: Permutazioni

- Vogliamo risolvere il problema di determinare tutte le possibili permutazioni dei caratteri presenti in una stringa
  - facciamo l'ipotesi che non ci siano caratteri ripetuti
- Ricordiamo dalla matematica combinatoria che il numero di permutazioni di ***N*** simboli è ***N!***
- Esempio: **ABC**
  - **ABC ACB BAC BCA CAB CBA**

Esercizio: Permutazioni

- Esiste un semplice algoritmo ricorsivo per trovare le permutazioni di una stringa di lunghezza N
- Se N vale 1, l'unica permutazione è la stringa stessa
- Altrimenti
  - estrai il primo carattere dalla stringa
  - calcola le (N-1)! permutazioni della stringa rimanente
  - per ognuna delle permutazioni (incomplete) ottenute
    - genera N permutazioni (complete) inserendo il carattere precedentemente estratto in ognuna delle posizioni possibili nella permutazione incompleta

Esercizio: Permutazioni

- Analizziamo meglio questo punto
  - calcola le (N-1)! permutazioni della stringa rimanente
  - per ognuna delle permutazioni (incomplete) ottenute
    - genera N permutazioni (complete) inserendo il carattere precedentemente estratto in ognuna delle posizioni possibili nella permutazione incompleta
- Le posizioni in cui si può inserire un carattere in una delle permutazioni incomplete, che ha dimensione N-1, sono le N-2 posizioni che si trovano tra due caratteri, più la posizione iniziale e la posizione finale
  - sono quindi N posizioni diverse

```
public static String[] permutations(String p)
{  // gestiamo i casi base
   if (p == null || p.length() == 0)
      return new String[0]; // oppure return null
   if (p.length() == 1) return new String[] {p};
   // isoliamo il primo carattere
   String c = p.substring(0,1);
   // passo ricorsivo: permutazioni incomplete
   String[] cc = permutations(p.substring(1));
   // numero di permutazioni da generare
   String[] r = new String[p.length() * cc.length];
   for (int i = 0; i < p.length(); i++)
      for (int j = 0; j < cc.length; j++)
      {  String s = cc[j];
         String sLeft = s.substring(0,i);
         String sRight = s.substring(i);
         r[i*cc.length+j] = sLeft + c + sRight;
      }
   return r;
}
```

Esempio: Permutazioni

- Alcuni commenti generali sulla gestione delle condizioni “eccezionali”
  - il metodo riceve un riferimento a una stringa: tale riferimento può essere **null** oppure la stringa può essere vuota (cioè avere lunghezza zero)
    - in entrambi i casi il metodo ricorsivo non funzionerebbe correttamente, quindi li inseriamo come casi base della ricorsione
  - cosa restituiamo?
    - quando un metodo deve restituire un oggetto, nei casi in cui riceve un parametro non corretto di solito restituisce **null**

```
if (p == null || p.length() == 0)
   return null;
```

Esempio: Permutazioni

```
if (p == null || p.length() == 0)
   return new String[0];
```

- In questo caso il metodo deve restituire un oggetto di tipo un po' particolare, in quanto è un array
- Sarebbe comunque corretto restituire **null**, ma di solito si preferisce restituire un array di lunghezza zero (perfettamente lecito), in modo che il metodo invocante riceva un array valido, seppure vuoto
- In questo modo, il codice seguente funziona…

```
String[] x = permutations("");
for (int i = 0; i < x.length; i++)
   System.out.println(x[i]);
```

Esempio: Permutazioni

```
if (p == null || p.length() == 0)
   return new String[0];
```

- Notiamo che anche ***l'ordine in cui vengono valutate le due condizioni è molto importante***
- Se **p** è **null**, la prima condizione è vera e per la strategia di cortocircuito nella valutazione dell'operatore **||** ***la seconda condizione non viene valutata***
  - se venisse valutata, verrebbe lanciata l'eccezione **NullPointerException** !!

Esempio: Permutazioni

```
// isoliamo il primo carattere
String c = p.substring(0,1);
// passo ricorsivo
String[] cc = permutations(p.substring(1));
// numero di permutazioni da generare
String[] r = new String[p.length() *
   cc.length];
```

- Per calcolare la dimensione del vettore che conterrà le permutazioni, sfruttiamo le informazioni ottenute dall'invocazione ricorsiva
  - il numero di permutazioni è uguale alla dimensione della stringa moltiplicata per il numero di permutazioni (incomplete) già generate
    - n = p.length(), (n-1) ! =cc.length
    - p.length * cc.length = n * (n-1) !

Esempio: Permutazioni

```
for (int i = 0; i < p.length(); i++)
   for (int j = 0; j < cc.length; j++)
   {  String s = cc[j];
      String sLeft = s.substring(0,i);
      String sRight = s.substring(i);
      r[i*cc.length+j] = sLeft + c + sRight;
   }
```

- Per completare le permutazioni inseriamo il carattere **c** in tutte le posizioni possibili in ciascuna permutazione incompleta **s**
- Per ogni **s**, calcoliamo le sottostringhe che verranno concatenate a sinistra e a destra di **c**
- Sfruttiamo il fatto che il metodo **substring()** gestisce in modo congruente le situazioni "anomale"

Esempio: Permutazioni

```
String sLeft = s.substring(0,i);
```

- Quando **i** vale **0**, **substring** restituisce una stringa vuota, che è proprio ciò che vogliamo

```
String sRight = s.substring(i);
```

- Quando **i** vale **p.length( )-1** (suo valore massimo), allora **i** è anche uguale a **s.length( )**
  - in questo caso particolare, **substring()** non lancia eccezione, ma restituisce una stringa vuota, che è proprio ciò che vogliamo

Esempio: Permutazioni

```
r[i*cc.length+j] = sLeft + c + sRight;
```

- Analizziamo meglio questa espressione dell'indice
- Globalmente, tale indice deve andare da `0` a `p.length() * cc.length` (escluso)
- Verifichiamo innanzitutto i limiti
  - per `i = 0` e `j = 0`, l'indice vale `0`
  - per `i = p.length()-1` e `j = cc.length-1`, l'indice vale

`(p.length()-1)*cc.length + cc.length – 1`
`= p.length()*cc.length – 1`

(come volevamo)

Esercizio: Permutazioni

```
r[i*cc.length+j] = sLeft + c + sRight;
```

- Alla prima iterazione di **i**, l'indice varia tra **0** e **cc.length-1** (perché **i** vale 0)
- Alla seconda iterazione di **i**, l'indice varia tra **1*cc.length+0 = cc.length** e **1*cc.length+cc.length-1 = 2*cc.length-1**
- Si osserva quindi che gli indici vengono generati consecutivamente, senza nessun valore mancante e senza nessun valore ripetuto

Ma 22-Nov-2005

FI1: ore 14:15 - 16:15
MAT_A: ore 16:15 - 18:15

Lezione XXX
Gi 17-Nov-2005

ADT Dizionario

Dizionario

- Un ***dizionario*** è un ADT con le seguenti proprietà
  - è un contenitore
  - consente l'inserimento di coppie di dati di tipo
    - ***chiave*** / ***valore (attributo)***

    con la **chiave** che deve essere ***confrontabile*** e ***unica*** nell'insieme dei dati memorizzati
    - non possono esistere nel dizionario due valori con identica chiave
  - consente di effettuare in modo efficiente la ricerca e la rimozione di valori ***usando la chiave come identificatore***

Dizionario

- L’analogia con il dizionario di uso comune è molto forte
- In un comune dizionario
  - le chiavi sono le singole parole
  - il valore corrispondente a una chiave è la definizione della parola nel dizionario
  - tutte le chiavi sono distinte
  - a ogni chiave è associato uno e un solo valore
  - la ricerca di un valore avviene tramite la sua chiave

Dizionario

```
public interface Dictionary extends Container
{  // l'inserimento va sempre a buon fine;
   // se la chiave non esiste, la coppia
   // key/value viene aggiunta al dizionario;
   // se la chiave esiste già, il valore ad
   // essa associato viene sovrascritto con
   // il nuovo valore
   void insert(Comparable key, Object value);

   // la rimozione della chiave rimuove anche
   // il corrispondente valore
   void remove(Comparable key);

   // la ricerca per chiave restituisce
   // soltanto il valore ad essa associato
   Object find(Comparable key);
}
```

Dizionario in un array

- Un ***dizionario*** può anche essere realizzato con un ***array***
  - ogni cella dell'array contiene un riferimento ad una coppia chiave/valore
    - un oggetto di tipo **Pair**
- Ci sono due strategie possibili
  - mantenere le chiavi ***ordinate*** nell'array
  - mantenere le chiavi ***non ordinate*** nell'array

Dizionario in un array ordinato

- Se le **n** chiavi vengono conservate ***ordinate*** nell'array
  - la **ricerca** ha prestazioni **O(log n)**
    - si può usare la ***ricerca per bisezione***
  - l'**inserimento** ha prestazioni **O(n)**
    - Si verifica la presenza della chiave (ricerca binaria)
    - si usa l'ordinamento per ***inserzione in un array ordinato***
    - con altre strategie, occorre invece ordinare l'intero array, con prestazioni, in generale, **O(n log n)**
  - la **rimozione** ha prestazioni **O(n)**
    - bisogna fare una ricerca binaria
    - spostare ***mediamente*** **n/2** elementi per mantenere l'ordinamento

Dizionario in un array non ordinato

- Se le **n** chiavi vengono conservate ***disordinate*** nell'array
  - la **ricerca** ha prestazioni **O(n)**
    - Si deve usare la ***ricerca lineare***
  - l'**inserimento** ha prestazioni **O(n)**
    - Si verifica la presenza della chiave (ricerca lineare)
    - è sufficiente inserire il nuovo elemento nell'ultima posizione dell'array
  - la **rimozione** ha prestazioni **O(n)**
    - bisogna fare una ricerca, e poi spostare nella posizione trovata l'ultimo elemento dell'array, perché l'ordinamento non interessa

Prestazioni di un dizionario

Dizionario	array ordinato	array non ordinato
ricerca	O(lg n)	O(n)
inserimento	O(n)	O(n)
rimozione	O(n)	O(n)

Prestazioni di un dizionario

- La scelta di una realizzazione piuttosto di un'altra dipende dall'utilizzo tipico del dizionario nell'applicazione
- Ad esempio
  - se nel dizionario si fanno frequenti inserimenti e sporadiche ricerche e rimozioni
    - la scelta più opportuna è l'array non ordinato
  - se il dizionario viene costruito una volta per tutte, poi viene usato per fare soltanto ricerche
    - la scelta più opportuna è l'array ordinato

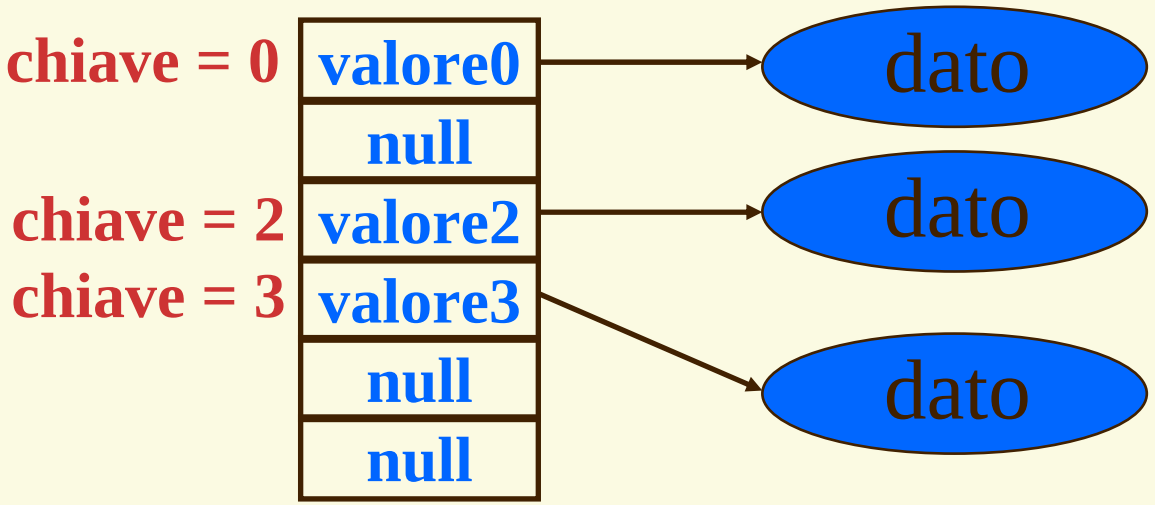
Chiavi numeriche

- Se imponiamo una restrizione al campo di applicazione di un dizionario
  - supponiamo che le chiavi siano ***numeri interi*** appartenenti a un intervallo noto a priori

  allora si può realizzare molto semplicemente un dizionario con prestazioni **O(1)** per tutte le operazioni
- Si usa un array che contiene soltanto i riferimenti ai valori, usando ***le chiavi come indici nell'array***
  - le celle dell'array che hanno come indice una chiave che non appartiene al dizionario hanno il valore **null**

Tabella

- Un dizionario con chiavi numeriche intere viene detto ***tabella*** o tavola (***table***)
- L'analogia è la seguente
  - un valore è una riga nella tabella
  - le righe sono numerate usando le chiavi
  - alcune righe possono essere ***vuote*** (senza valore)



Tabella

- Definiamo il tipo di dati astratto **Table** con un comportamento identico al dizionario
  - l'unica sostanziale differenza è che le chiavi non sono riferimenti a oggetti di tipo **Comparable**, ma sono ***numeri interi*** (che evidentemente sono confrontabili)

```
public interface Table extends Container
{  void insert(int key, Object value);
   void remove(int key);
   Object find(int key);
}
```

Tabella

```
public class ArrayTable100 implements Table
{  private Object[] v;
   private int count; //count rende isEmpty O(1)

   public ArrayTable100()
   {  makeEmpty(); }

   public void makeEmpty()
   {  count = 0;
      v = new Object[100];
   }
   public boolean isEmpty()
   {  return (count == 0); }

   private void check(int key)
   {  if (key < 0 || key >= v.length)
        throw new
           InvalidPositionTableException();
   }
    ...
```

```
public class ArrayTable100 implements Table
{  ...
   public void insert(int key, Object value)
   {  check(key);
      if (v[key] == null)
         count++;

      v[key] = value;
   }

   public void remove(int key)
   {  check(key);
      if (v[key] != null)
      {  count--;
         v[key] = null;
      }
   }

   public Object find(int key)
   {  check(key);
      return v[key];
   }
}
```

Tabella

- La tabella potrebbe avere ***dimensione variabile***, cioè utilizzare un ***array di dimensione crescente*** quando sia necessario
  - l'operazione di ***inserimento*** richiede, però, un tempo **O(n)** ogni volta che è necessario un ridimensionamento
  - in questo caso non si può utilizzare l'analisi ammortizzata perché non si può prevedere quali siano le posizioni richieste dall'utente
  - non è più vero che il ridimensionamento avviene "una volta ogni tanto", può avvenire anche tutte le volte
  - le prestazioni nel caso peggiore sono quindi **O(n)**

Tabella

- La tabella non utilizza la memoria in modo efficiente
  - l'occupazione di memoria richiesta per contenere **n** dati non dipende da **n** in modo lineare
    - come invece avviene per tutti gli altri ADT

    ma dipende dal contenuto informativo presente nei dati
    - in particolare, dal valore della chiave massima
- Può essere necessario un array di milioni di elementi per contenere poche decine di dati
  - si definisce ***fattore di riempimento*** (***load factor***) della tabella il numero di dati contenuti nella tabella diviso per la dimensione della tabella stessa

Tabella

- La tabella è un dizionario con prestazioni ottime
  - tutte le operazioni sono **O(1)**

  ma con le seguenti limitazioni
  - le ***chiavi*** devono essere ***numeri interi*** (non negativi)
    - in realtà si possono usare anche chiavi negative, sottraendo ad ogni chiave il valore dell'estremo inferiore dell'intervallo di variabilità
  - ***l'intervallo di variabilità delle chiavi deve essere noto a priori***
    - per dimensionare la tabella
  - se il fattore di riempimento è molto basso, si ha un grande ***spreco di memoria***
    - ciò avviene se le chiavi sono molto "disperse" nel loro insieme di variabilità

Tabella

- Cerchiamo di eliminare una delle limitazioni
  - ***l'intervallo di variabilità delle chiavi deve essere noto a priori***
    - per dimensionare la tabella
- Risolviamo il problema in modo diverso
  - fissiamo la dimensione della tabella in modo arbitrario
    - in questo modo si definisce di conseguenza l'intervallo di variabilità delle chiavi utilizzabili
  - per usare chiavi esterne all'intervallo, si usa una ***funzione di trasformazione delle chiavi***
    - ***funzione di hash***

Funzione di hash

- Una funzione di hash ha
  - come ***dominio*** l'insieme delle chiavi che identificano univocamente i dati da inserire nel dizionario
  - come ***codominio*** l'insieme degli indici validi per accedere a elementi della tabella
    - il risultato dell'applicazione della funzione di hash a 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 k e la dimensione della tabella n***

$$k_r = k \,\%\, n$$

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

Funzione di hash

- Il fatto che la funzione di hash non sia univoca genera un problema nuovo
  - inserendo un valore nella tabella, può darsi che la sua chiave ridotta sia uguale a quella di un valore già presente nella tabella e avente una ***diversa*** chiave, ma la stessa chiave ridotta
  - usando l'algoritmo già visto per la tabella, il nuovo valore andrebbe a sostituire il vecchio
    - questo non è corretto perché i due valori hanno, in realtà, chiavi diverse
  - questo fenomeno si chiama ***collisione*** nella tabella e si può risolvere in molti modi diversi
    - una tabella che usa chiavi ridotte si chiama ***tabella hash*** (***hash table***)

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 per ogni cella dell'array
  - l'array 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 a una chiave ridotta

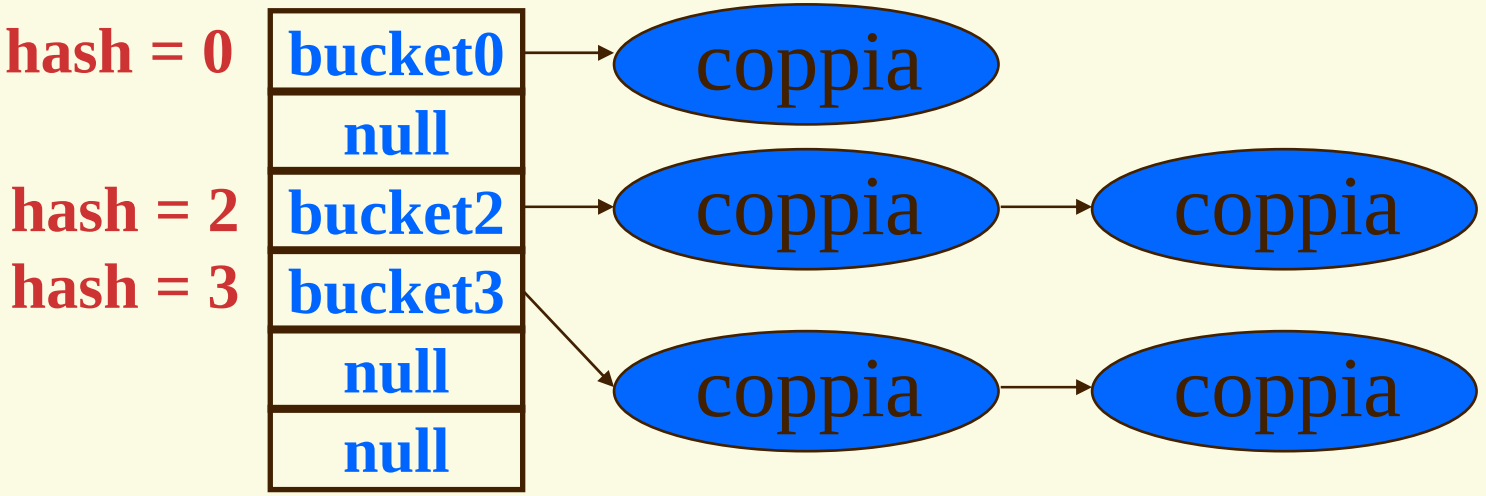


Tabella hash con bucket

- 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)**

Tabella hash con bucket

- **caso migliore**: la funzione di hash restituisce chiavi ridotte che si distribuiscono uniformemente nella tabella
  - tutte le liste hanno la stessa lunghezza media
    - se **M** è la dimensione della tabella
      - la lunghezza media di ciascuna lista è **n/M**
    - 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**

Tabella hash con bucket

- Riassumendo, in una ***tabella hash 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!
- Se le chiavi vere sono uniformemente distribuite
  - la funzione di hash che "***calcola il resto della divisione intera***" genera chiavi ridotte uniformemente distribuite

Tabella hash 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***

Tabella hash 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 chiave generica

- *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
- 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

Funzione di hash per chiave generica

- 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

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

- In Java è molto semplice, perché un **char** è anche un numero intero...

Funzione di hash per stringa

"ABC" ⇒ 'A'·65536^2 + 'B'·65536^1 + 'C'·65536^0

```
public int hashCode()
//ritorna un intero
{
   int final BASE = 65536;
   int h = 0;
   for (int i = 0; i < length(); i++)
      h = BASE * h + charAt(i);

   return h;
}
```

e l'overflow?
Una volta tanto e' utile!

Funzione di hash per stringa

- La libreria standard usa per le stringhe la seguente funzione di hash

```
public int hashCode()
{
  final int HASH_MULTIPLIER = 31;
  int h = 0
  for (int i = 0; i < length(); i++)
    h = HASH_MULTIPLIER * h + charAt(i);
  return h;
}
```

- Usiamo un numero primo come moltiplicatore di hash perche' si ritiene che distribuisca meglio i valori.

Attenzione l'hash code puo' essere negativo a causa dell'overfolw!

Funzione di hash per chiave generica

- Come fare 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() % MAX;// hash ∈ [-MAX+1, MAX-1]

Funzione di hash per chiave generica

- Per calcolare la funzione di hash di un oggetto che abbia due variabili di istanza:

```
String nome; double d;
```

- Si fa generalmente come segue

```
public int hashCode()
{
  final int HASH_MULTIPLIER = 29;
  int h1 = nome.hashCode();
  int h2 = (new Double(d)).hashCode();
  int h = HASH_MULTIPLIER * h1 + h2;
  return h;
}
```

- HASH_MULTIPLIER e' come al solito un numero primo
- Se ci sono variabili di istanza intere, si usa generalmente il numero intero nella generazione del hashCode

Tabella hash per chiave generica

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