Esercizio
Realizzazione di una pila “sicura”

Realizzazione di una pila “sicura”

- Tutte le strutture dati che abbiamo definito possono contenere oggetti di qualsiasi tipo
- Solitamente vengono utilizzate per contenere oggetti omogenei
  - cioè oggetti che siano tutti dello stesso tipo

  ma in generale questo vincolo non c’è
- Anche quando vengono usate come contenitori di oggetti omogenei, è possibile introdurre oggetti disomogenei, per errore
- Vogliamo realizzare una struttura dati, ad esempio una pila, che possa contenere ***soltanto*** oggetti di un certo tipo

```java
// rivediamo la definizione della pila
public class ArrayStack implements Stack
{  private Object[] v = new Object[100];
   private int vSize = 0;
   public boolean isEmpty() { return vSize == 0; }
   public void makeEmpty() { vSize = 0; }
   public void push(Object obj)
   {  if (vSize == v.length) v = resize(v, 2*v.length);
      v[vSize++] = obj; }
   public Object top()
   {  if (isEmpty()) throw new EmptyStackException();
      return v[vSize - 1]; }
   public Object pop()
   {  Object obj = top(); vSize--; return obj; }

   private Object[] resize(Object[] old, int newLength)
   {  Object[] newArray = new Object[newLength];
      for (int i = 0; i < old.length; i++)
         newArray[i] = old[i];
      return newArray; }
}
```

Realizzazione di una pila “sicura”

- Per realizzare una pila che possa contenere soltanto stringhe, ad esempio
  - si può riscrivere tutto il codice modificando il solo metodo **push**

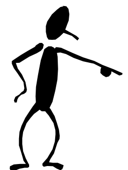
```java
public class StringArrayStack implements Stack
{  ...
   public void push(Object obj)
   {  try
      {  String s = (String) obj;
      } catch (ClassCastException e)
      {  throw new InvalidTypeInStackException();
      }
      if (vSize == v.length)
         v = resize(v, 2*v.length);
      v[vSize++] = obj; }
}
```

L’operatore instanceof

- Usando l’operatore **instanceof** si può scrivere codice più elegante

```
public class StringArrayStack implements Stack
{  ...
   public void push(Object obj)
   {  if (!(obj instanceof String))
         throw new InvalidTypeInStackException();
      if (vSize == v.length)
         v = resize(v, 2*v.length);
      v[vSize++] = obj; }
}
```

L’operatore instanceof



- Sintassi: `variabileOggetto instanceof NomeClasse`
- Scopo: è un operatore booleano che restituisce **true** se e solo se la ***variabileOggetto*** contiene un riferimento ad un oggetto che è un esemplare della classe ***NomeClasse*** (o di una sua sottoclasse)
  - in tal caso l’assegnamento di ***variabileOggetto*** ad una variabile di tipo ***NomeClasse*** **NON** lancia l’eccezione **ClassCastException**
- Nota: il risultato non dipende dal tipo dichiarato per la ***variabileOggetto***, ma dal tipo dell’oggetto a cui la variabile si riferisce effettivamente al momento dell’esecuzione

Realizzazione di una pila "sicura"

- Si potrebbe pensare di evitare il controllo di tipo nel metodo modificando il tipo del parametro

```
public class StringArrayStack implements Stack
{   ... // NON FUNZIONA
    public void push(String obj)
    {   if (vSize == v.length)
            v = resize(v, 2*v.length);
        v[vSize++] = obj; }
}
```

- **Non funziona**: il compilatore segnala che la classe non definisce tutti i metodi richiesti dall'interfaccia **Stack**
  - il metodo **push** ha una firma diversa

Realizzazione di una pila "sicura"

- Un modo più "elegante" sfrutta l'ereditarietà
  - che, come sappiamo, è spesso utile per evitare di ricopiare codice…

```
public class StringArrayStack extends ArrayStack
{   public void push(Object obj)
    {   if (!(obj instanceof String))
            throw new InvalidTypeInStackException();
        super.push(obj);
    }
}
```

- Il metodo **push** viene sovrascritto (si noti la firma identica…), per cui negli esemplari di **StringArrayStack** verrà invocato il metodo qui definito e non quello di **ArrayStack**

Realizzazione di una pila “sicura”

- Vediamo come utilizzare **StringArrayStack**

```
Stack s = new StringArrayStack();
s.push("pippo");
String st = (String) s.pop();
s.push(new Integer(2)); // lancia un’eccezione
```

- Usando **StringArrayStack** è possibile fare il cast esplicito dopo l’estrazione di un elemento con la certezza che andrà a buon fine
- Viene la tentazione di ridefinire il metodo **pop** come segue, per non dover fare il cast

```
{  ...
   public String pop() { ... }
}
```

Realizzazione di una pila “sicura”

- Viene la tentazione di ridefinire il metodo **pop** come segue, per non dover fare il cast

```
{  ...
   public String pop() { ... }
}
```

- Questo di nuovo non funziona, per lo stesso motivo citato in precedenza
  - il compilatore segnala che la classe non definisce tutti i metodi richiesti dall’interfaccia **Stack**
    - il metodo **pop** ha una firma diversa

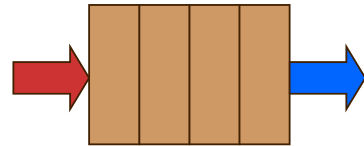
Realizzazione di una pila “sicura”

- Alternativa senza usare l’ereditarietà
  - usiamo una pila come variabile privata di esemplare
  - invece di usare l’ereditarietà, si usa l’incapsulamento

```
public class StringArrayStack implements Stack
{  private Stack s = new ArrayStack();
   public boolean isEmpty() { return s.isEmpty(); }
   public void makeEmpty() { s.makeEmpty(); }
   public void push(Object obj)
   {  if (!(obj instanceof String))
         throw new InvalidTypeInStackException();
      s.push(obj); }
   public Object top() { return s.top(); }
   public Object pop() { return s.pop(); }
}
```

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 ad 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*)

- 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 ad una coda vuota e gestire la situazione di coda piena (segnalando un errore o ridimensionando l’array)
- Definiamo
  - **EmptyQueueException** e **FullQueueException**

```
public class EmptyQueueException extends RuntimeException
{ }

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); }
    public void enqueue(Object obj)
    {   if (vSize == v.length) throw new FullQueueException();
        v[vSize++] = obj; }
    public Object getFront()
    {   if (isEmpty()) throw new EmptyQueueException();
        return v[0]; }
    public Object dequeue()
    {   Object obj = getFront();
        vSize--;
        for (int i = 0; i < vSize; 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 e l'altro indice punta all'ultimo oggetto della coda
- 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); }
    public void enqueue(Object obj)
    {   if (back == v.length) throw new FullQueueException();
        v[back++] = obj; }
    public Object getFront()
    {   if (isEmpty()) throw new EmptyQueueException();
        return v[front]; }
    public Object dequeue()
    {   Object obj = getFront();
        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
    - in realtà una cella dell'array deve rimanere vuota per "motivi tecnici", cioè **back** non può raggiungere **front** (per quale motivo? controllare **isEmpty**…)
  - 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 che cresce

```
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);
    }
}
```

Esercizio: Calcolatrice

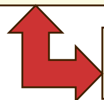
Esercizio: Calcolatrice

- 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 destra
    - estrai dalla pila l'operando sinistro
    - esegui l'operazione
    - inserisci il risultato sulla pila
- 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 ad 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 sinistra 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;
         else
            if (!inBracket) return 2;
            else if(areMatching(bracket, c))
               inBracket = false;
            else return 3;
   }
   if (inBracket) return 4;
   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 sulla 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;
                } catch (EmptyStackException e)
                {   return 2; }
    }
    if (!st.isEmpty()) return 4;
    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
```

Inizializzazione e ambito di visibilità di una variabile

Inizializzazione di una variabile

- Le ***variabili di esemplare*** e le ***variabili statiche***, se non sono inizializzate esplicitamente, vengono ***inizializzate automaticamente*** ad un valore predefinito
  - ***zero*** per le variabili di tipo numerico e carattere
  - **false** per le variabili di tipo booleano
  - **null** per le variabili oggetto

Inizializzazione di una variabile

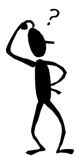
- Le ***variabili parametro*** vengono inizializzate copiando il valore dei parametri effettivi usati nell'invocazione del metodo
- Le ***variabili locali non*** vengono inizializzate automaticamente, ed il compilatore effettua un controllo semantico impedendo che vengano utilizzate prima di aver ricevuto un valore

Ambito di visibilità di una variabile

- L'***ambito di visibilità*** di una variabile indica la parte di codice nel quale è lecito usare la variabile (per leggerne il valore e/o assegnarle un valore)
- Per le ***variabili locali*** e le ***variabili parametro***, l'ambito di visibilità è quello che determina anche il relativo ciclo di vita
- Per le ***variabili statiche*** e le ***variabili di esemplare***, l'ambito di visibilità dipende dalla dichiarazione **public**, **private** o **protected**

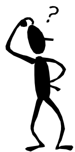
Ambito di visibilità di una variabile

- Se le ***variabili statiche*** e le ***variabili di esemplare*** sono dichiarate
  - **public**, sono visibili in ogni parte del programma
  - **private**, sono visibili soltanto all'interno della classe in cui sono definite
  - **protected**, sono visibili soltanto all'interno della classe in cui sono definite e nelle classi derivate



Ambito di visibilità di una variabile

- È anche possibile dichiarare variabili statiche e variabili di esemplare ***senza indicare uno specificatore di accesso*** (***accesso di default***)
  - sono così visibili anche all’interno di classi che si trovano ***nello stesso package*** (cioè di file sorgenti che si trovano nella stessa cartella)
  - anche le variabili **protected** hanno questa proprietà, oltre a quella già vista



Ambito di visibilità di una variabile

- Conoscere l’***ambito di visibilità*** e il ***ciclo di vita*** di una variabile è molto importante per capire quando e dove è possibile ***usare di nuovo il nome di una variabile che è già stato usato***
- Le regole appena viste consentono di usare, in metodi diversi della stessa classe, ***variabili locali*** o ***variabili parametro*** ***con gli stessi nomi***, senza creare alcun conflitto, perché
  - i rispettivi ambiti di visibilità non sono sovrapposti
- In questi casi, le variabili definite nuovamente non hanno alcuna relazione con le precedenti

Ordinare pile e code

Esercizio: Mergesort per pila

- Proviamo a realizzare l'ordinamento per fusione (Mergesort) applicato ad una pila anziché ad un array
  - ovviamente la pila deve contenere oggetti **Comparable**
- Il primo problema consiste nella suddivisione della pila in due pile di dimensioni circa uguali
  - non conosciamo la dimensione della pila
  - una soluzione semplice consiste nell'estrarre gli elementi dalla pila ed inserirli ***alternativamente*** in due semi-pile
- Ordinare una coda è molto simile…

```
public static void mergeSort(Stack s)
{   if (s == null || s.isEmpty()) return; // caso base
    Object temp = s.pop();
    if (s.isEmpty())
    {   s.push(temp); // altro caso base: dimensione 1
        return; }
    // dividiamo (circa) a meta'
    Stack left = new ArrayStack();
    Stack right = new ArrayStack();
    left.push(temp);
    boolean flag = true;
    while (!s.isEmpty())
    {   if (flag)
            right.push(s.pop());
        else
            left.push(s.pop());
        flag = !flag; // trucco...
    }
    mergeSort(left);
    mergeSort(right);
    // fusione: si osservi che s è rimasta vuota
    merge(s, left, right);
}
```

Fusione di due pile ordinate

```
private static void merge(Stack s, // NON FUNZIONA
                          Stack left, Stack right)
{   while (!left.isEmpty() && !right.isEmpty())
    {   Comparable x = (Comparable) left.top();
        Comparable y = (Comparable) right.top();
        if (x.compareTo(y) < 0)
            s.push(left.pop());
        else
            s.push(right.pop());
    }
    while (!left.isEmpty())
        s.push(left.pop());
    while (!right.isEmpty())
        s.push(right.pop());
    // a questo punto s contiene i dati ordinati,
    // ma in cima alla pila c'è l'elemento maggiore!
}
```

Fusione di due pile ordinate

```
private static void merge(Stack s,
                          Stack left, Stack right)
{   Stack temp = new ArrayStack();
    while (!left.isEmpty() && !right.isEmpty())
    {   Comparable x = (Comparable) left.top();
        Comparable y = (Comparable) right.top();
        if (x.compareTo(y) < 0)
            temp.push(left.pop());
        else temp.push(right.pop());
    }
    while (!left.isEmpty())
        temp.push(left.pop());
    while (!right.isEmpty())
        temp.push(right.pop());
    while (!temp.isEmpty()) // inverte il contenuto
        s.push(temp.pop());
}
```

Esercizio: Mergesort per pila

- Notiamo che, nonostante l'apparente maggiore complessità, le prestazioni asintotiche rimangono **O(n lg n)**
- Ricordando che tutte le operazioni sulle pile sono **O(1)** come gli accessi ad un elemento di un array, l'analisi delle prestazioni dell'algoritmo fornisce ancora

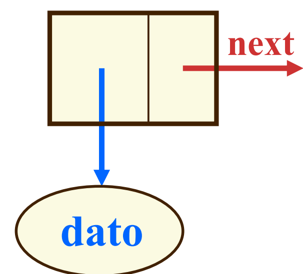
$$T(n) = 2T(n/2) + O(n)$$

da cui si ottiene di nuovo

$$O(n \lg n)$$

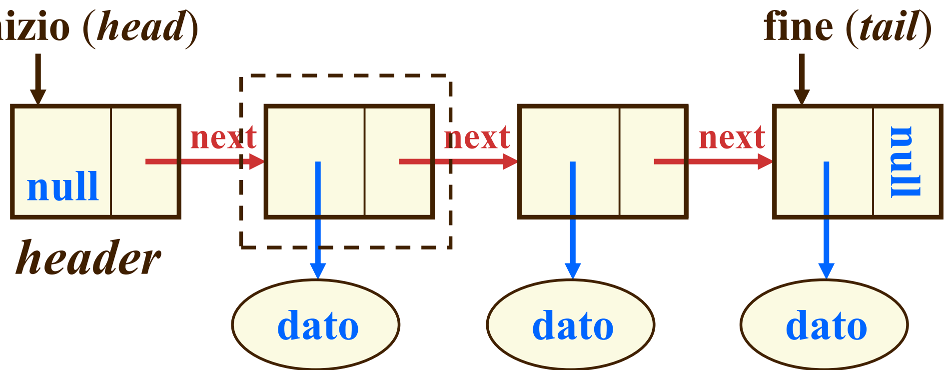
Catena (*linked list*)

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 ad un elemento (***il dato***)
    - un riferimento al nodo ***successivo*** nella catena (**next**)

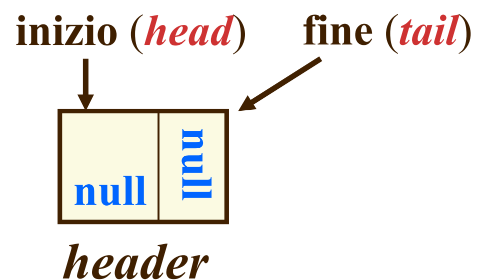
Catena



- Per agire sulla catena è sufficiente memorizzare il ***riferimento al suo primo nodo***
  - è comodo avere anche un riferimento all'ultimo nodo

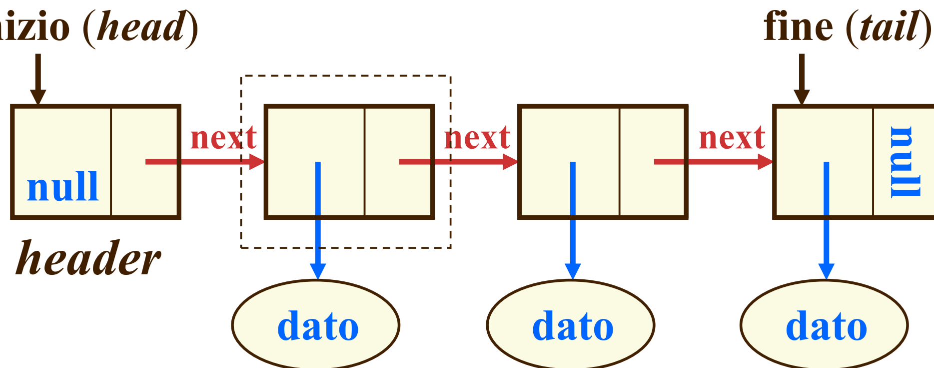
- Il campo **next** dell'ultimo nodo contiene **null**
- Vedremo che è comodo avere un primo nodo senza dati, chiamato ***header***

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()
    {   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 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

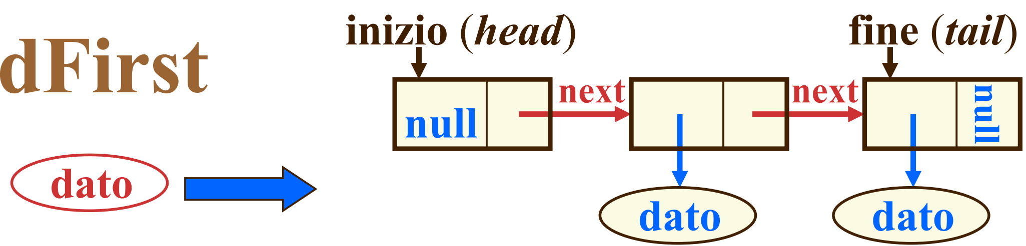
Catena

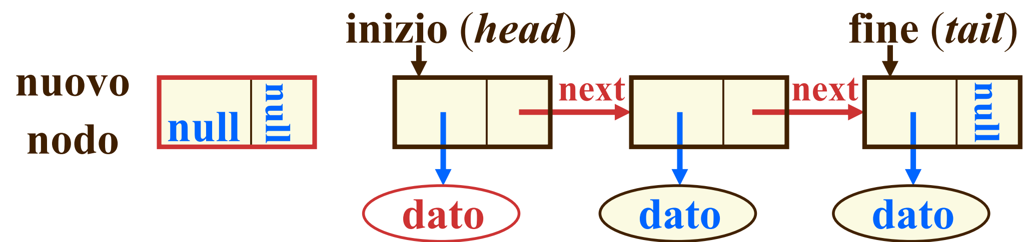
- Infine, dato che anche la catena è un contenitore, ci sono i metodi
  - **isEmpty** per sapere se la catena è vuota
  - **makeEmpty** per rendere vuota la catena
- Si definisce l'eccezione **EmptyLinkedListException**
- Si noti che, non essendo la catena un ADT, non viene definita un'interfaccia
  - 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

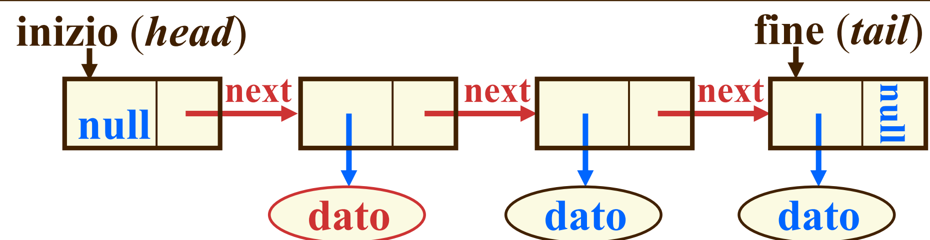
Catena

```
public class LinkedList implements Container
{  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







addFirst

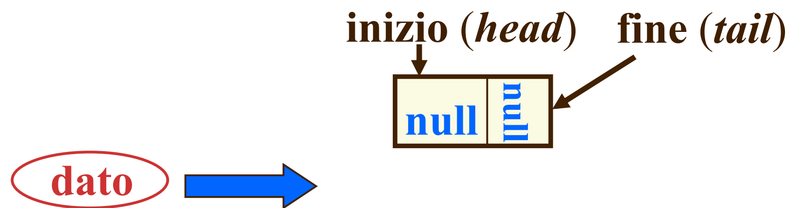
```
public class LinkedList ...
{  ...
   public void addFirst(Object e) {
      // inserisco il dato nell'header attuale
      head.setElement(e);
      // creo un nuovo nodo con due riferimenti null
      ListNode n = new ListNode();
      // collego il nuovo nodo all'header attuale
      n.setNext(head);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

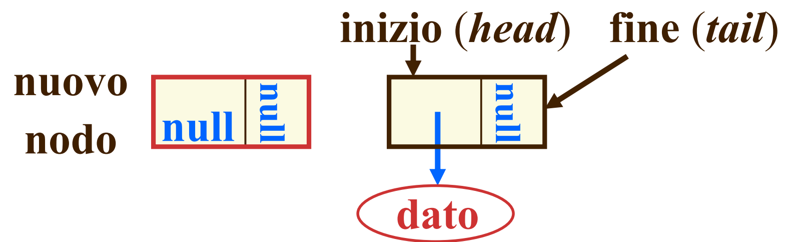
- Non esiste il problema di "catena piena"
- L'operazione è `O(1)`

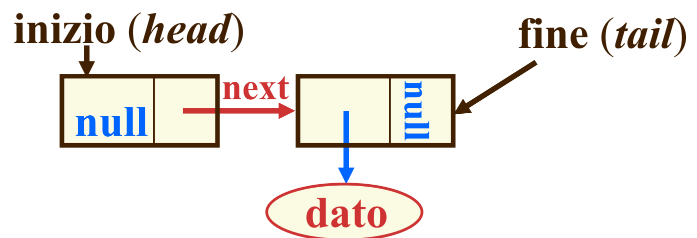
addFirst

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

dato







addFirst

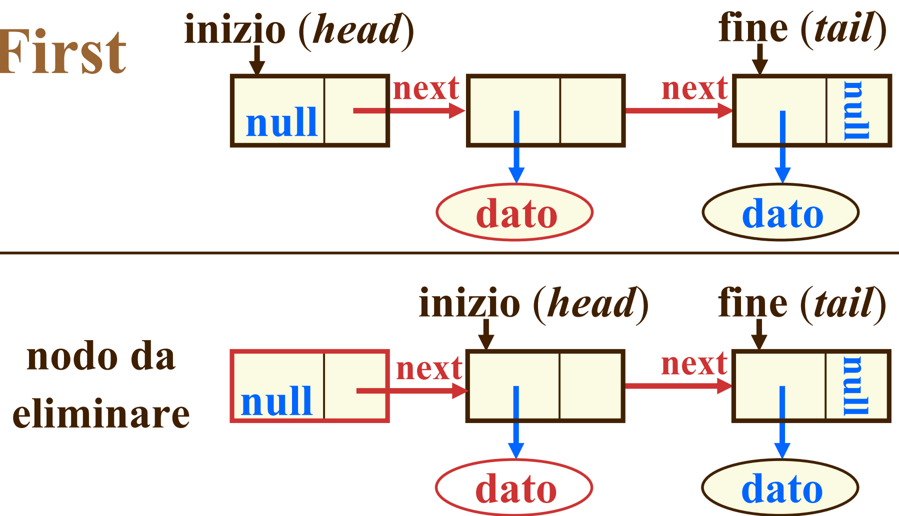
```
public void addFirst(Object e) {
    head.setElement(e);
    ListNode n = new ListNode();
    n.setNext(head);
    head = n;
}
```

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

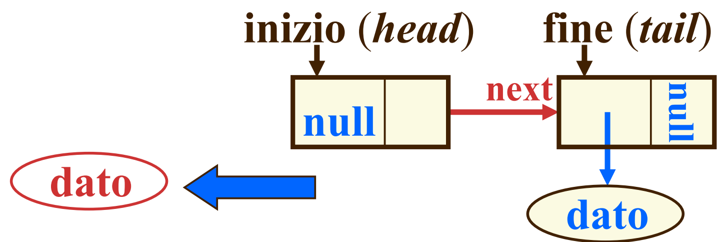
```
public void addFirst(Object e) {
    head.setElement(e);
    head = new ListNode(null, head);
    // funziona perché prima head viene USATO
    // (a destra) e solo successivamente viene
    // MODIFICATO (a sinistra)
}
```

- È più "professionale", anche se meno leggibile

removeFirst



Il nodo viene eliminato dal garbage collector



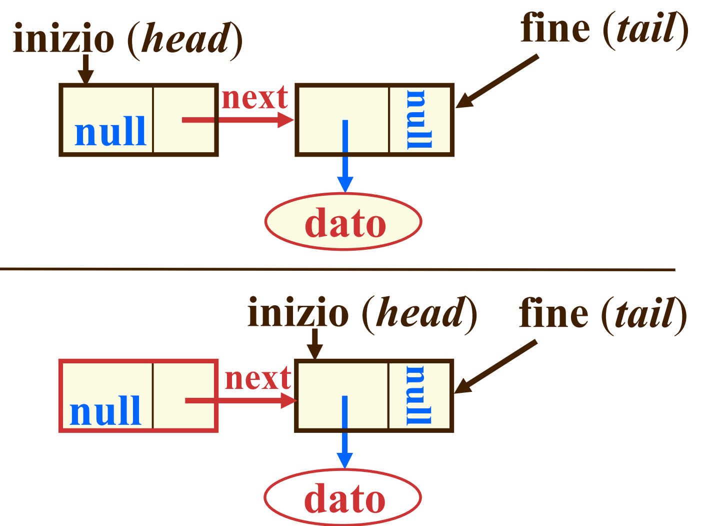
removeFirst

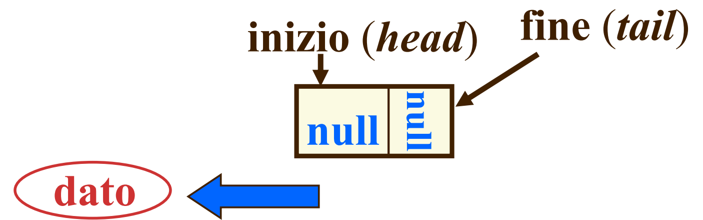
```
public class LinkedList ...
{   ...
    public Object removeFirst() {
        // delega a getFirst il
        // controllo di lista vuota
        Object e = getFirst();
        // aggiorno l'header
        head = head.getNext();
        head.setElement(null);
        return e;
    }
}
```

- L'operazione è O(1)

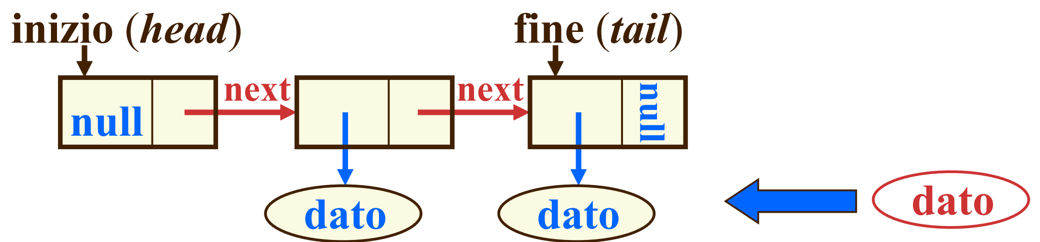
removeFirst

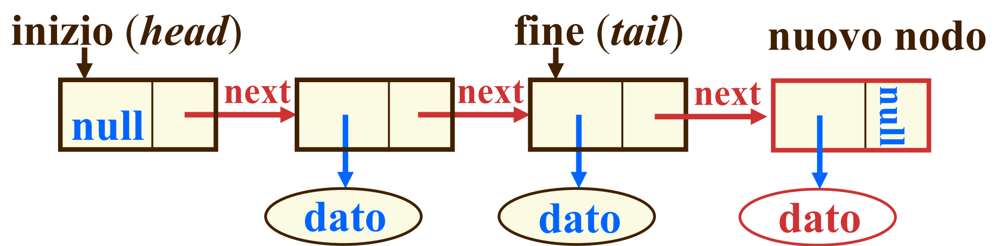
- Verifichiamo che tutto sia corretto anche rimanendo con una ***catena vuota***
- ***Fare sempre attenzione ai casi limite***

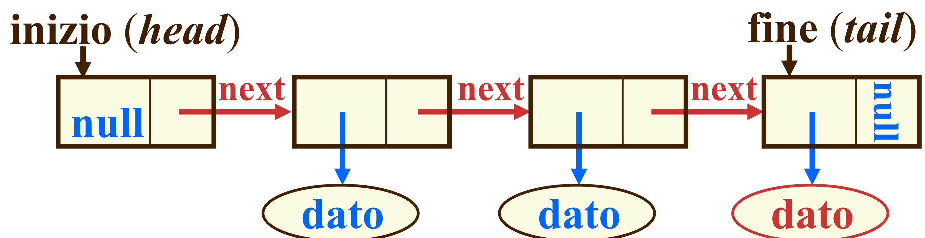




addLast







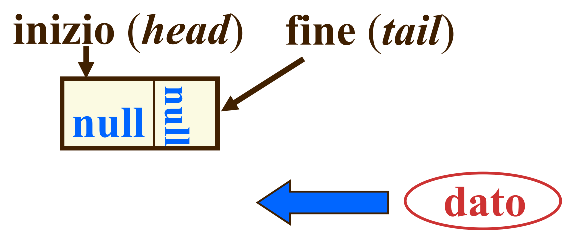
addLast

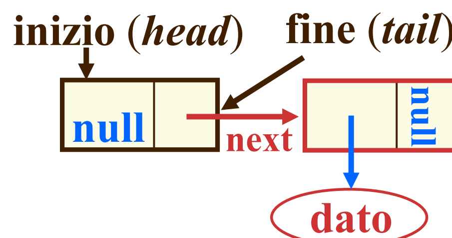
```
public class LinkedList ...
{  ...
   public void addLast(Object e) {
      tail.setNext(new ListNode(e, null));
      tail = tail.getNext();
   }
}
```

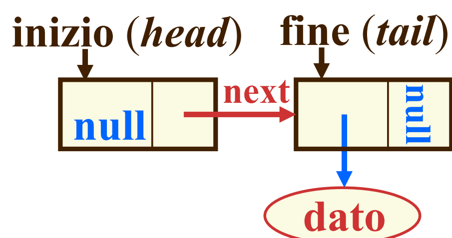
- Non esiste il problema di “catena piena”
- Anche questa operazione è `O(1)`

addLast

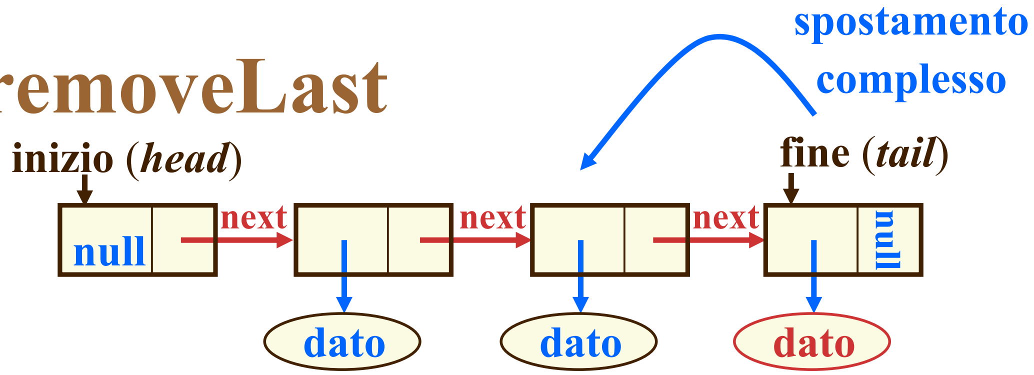
- Verifichiamo che tutto sia corretto anche inserendo in una ***catena vuota***
- ***Fare sempre attenzione ai casi limite***

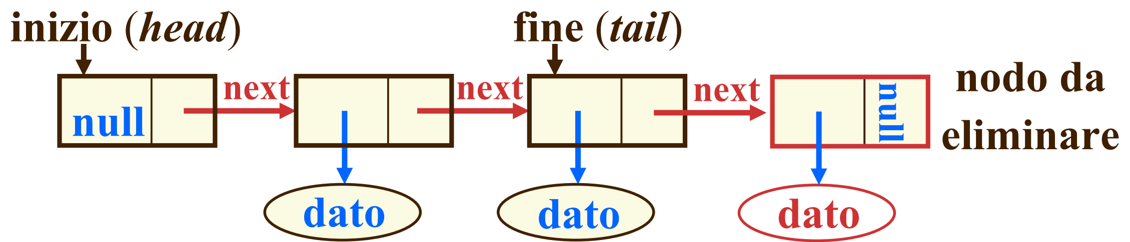


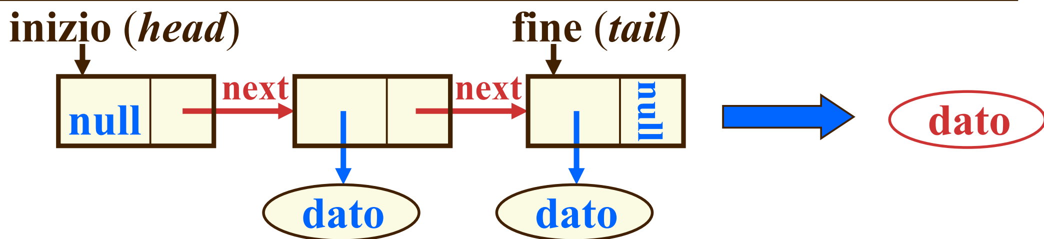




removeLast







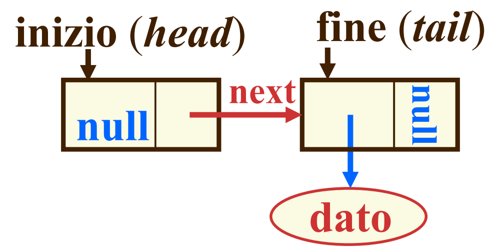
removeLast

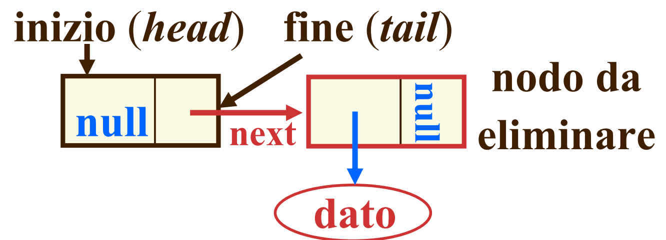
```
public class LinkedList ...
{  ...
   public Object removeLast() {
      Object e = getLast();
      // bisogna cercare il penultimo nodo
      // partendo dall'inizio e andando avanti
      // finché non si arriva alla fine della catena
      ListNode temp = head;
      while (temp.getNext() !=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

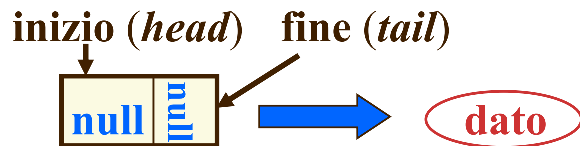
- Purtroppo questa operazione è `O(n)`

removeLast

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

Classi interne

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
    {   ... }
}
```