

Lezione XXIX Lu 26-Nov-2007

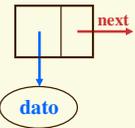
Lista Concatenata (LinkedList)

1

Lista concatenata (linked list)

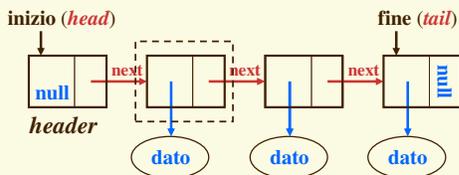
□ La *catena* o *lista concatenata* (*linked list*) è una **struttura dati** alternativa all'array (eventualmente) riempito solo in parte per la realizzazione di classi

- 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*)



2

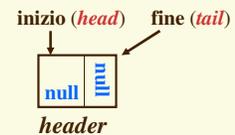
Lista Concatenata



- 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**
 - si dice *lista concatenata con nodo sentinella*
- **head** e' il riferimento al **primo nodo** (header), **tail** all'ultimo nodo

3

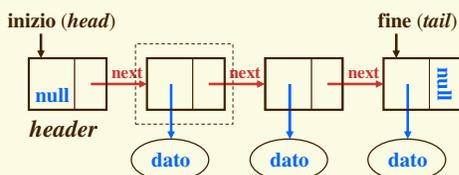
Lista concatenata vuota



- Per capire bene il funzionamento della catena con nodo sentinella, 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**

4

Lista Concatenata



- Per accedere in sequenza a tutti i nodi della catena si parte dal riferimento **head** 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**
- L'Accesso nella lista concatenata è sequenziale

5

Nodo di una Lista Concatenata

```
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

7

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

8

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

9

Lista Concatenata

- 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

10

Lista Concatenata

- 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
 - **size()** che restituisce il numero di elementi nel contenitore. Per ora **non realizziamo** il metodo nella lista concatenata.
- Si definisce l’eccezione **EmptyLinkedListException**

11

Eccezione

EmptyLinkedListException

```
public class EmptyLinkedListException
    extends RuntimeException
{
    public EmptyLinkedListException()
    {
    }

    public EmptyLinkedListException(String err)
    {
        super(err);
    }
}
```

- Estendiamo l’eccezione **java.lang.RuntimeException**, così non siamo costretti a gestirla

12

Interfaccia Container

```
public interface Container
{
    /**
     * verifica se il contenitore e' vuoto
     * @return true se il contenitore e' vuoto,
     * false altrimenti
     */
    boolean isEmpty();

    /**
     * rende vuoto il contenitore
     */
    void makeEmpty();
}
```

Non inseriamo
il metodo size()!

13

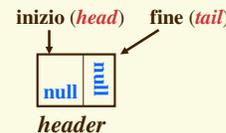
Lista Concatenata

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

    public LinkedList()
    { makeEmpty();
    }

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

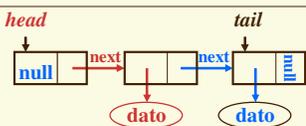
    public boolean isEmpty()
    { return (head == tail);
    }
    ...
}
```



Lista Concatenata

```
public class LinkedList implements Container
{
    ...
    public Object getFirst() // operazione O(1)
    {
        if (isEmpty())
            throw new EmptyLinkedListException();

        return head.getNext().getElement();
    }
    ...
}
```

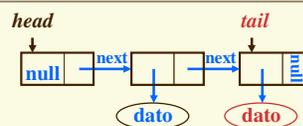


15

Lista Concatenata

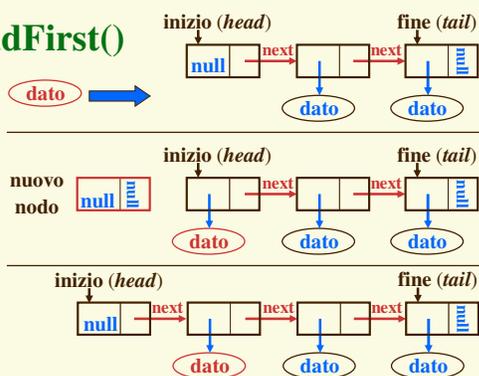
```
public class LinkedList implements Container
{
    ...
    public Object getLast() // operazione O(1)
    {
        if (isEmpty())
            throw new EmptyLinkedListException();

        return tail.getElement();
    }
    ...
}
```



16

addFirst()



17

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)

18

addFirst()

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

19

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

20

removeFirst()

Il nodo viene eliminato dal garbage collector

21

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 $O(1)$

22

removeFirst()

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

23

addLast()

24

addLast()

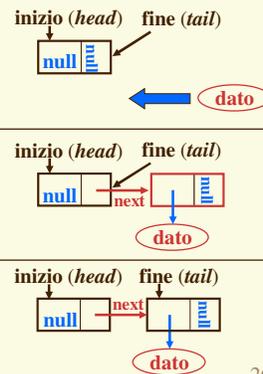
```
public class LinkedList ...
{
    ...
    public void addLast(Object e) {
        tail.setNext(new ListNode(e, null));
        tail = tail.getNext();
    }
}
```

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

25

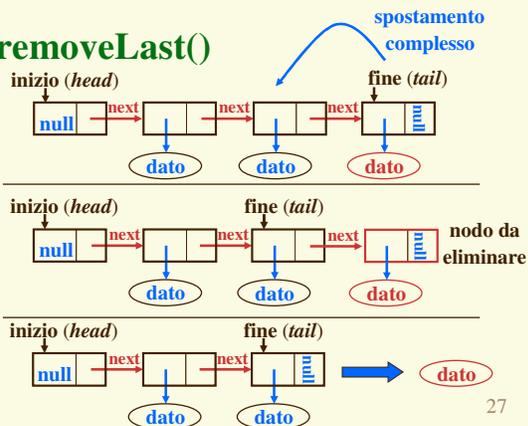
addLast()

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



26

removeLast()



27

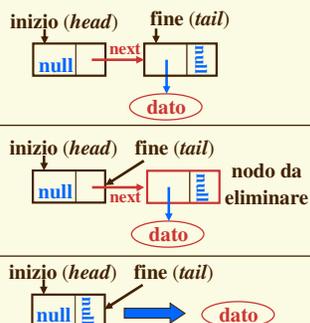
removeLast()

```
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**



29

Header della lista concatenata

- ❑ 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

30

Prestazioni lista concatenata

- ❑ Tutte le operazioni sulla lista *concatenata* sono $O(1)$ tranne `removeLast()` che è $O(n)$
 - si potrebbe pensare di tenere un riferimento anche al *penultimo* nodo, ma per *aggiornare* tale riferimento sarebbe comunque necessario un tempo $O(n)$
- ❑ Se si usa una 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

31

Prestazioni della lista concatenata

- ❑ 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 (*java 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)

32

Riepilogo delle prestazioni della lista concatenata

```
{
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)
}
```

33

Lezione XXX
Ma 27-Nov-2007

Lista Concatenata
Cenni alle Classi Interne

34

Lista Concatenata

```
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)
}
```

35

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

36

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 dentro un'altra classe essa è *accessibile* (in tutti i sensi) *soltanto all'interno della classe* in cui è definita
 - all'esterno non è nemmeno possibile creare oggetti di tale classe interna

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

Lista Concatenata

```
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)

    class ListNode
    {
        . . .
    }
}
```

Lista doppiamente concatenata (doubly linked list)

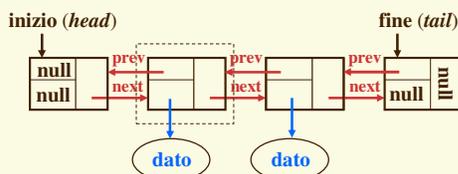
39

Lista Concatenata doppia

- La *catena doppia* (lista doppiamente concatenata, *doubly linked list*) è una struttura dati.
- 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*)

40

Lista Concatenata Doppia



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

41

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
- I metodi di inserimento e rimozione si complicano

42

Tipi di dati astratti e strutture dati

43

Strutture dati

- Una **struttura dati** (*data structure*) è un **modo sistematico** di **organizzare** i dati in un **contenitore** e di **controllare l'accesso** ai dati
- In Java una **struttura dati** viene definita tramite **una classe, ad esempio**
 - la **lista concatenata** è una **struttura dati**
 - è un **contenitore** che **memorizza dati** in modo **sistematico (nei nodi)**
 - l'**accesso** ai dati è **controllato** tramite metodi di accesso (*addFirst(), addLast(), getFirst()*...)

44

Strutture dati

- Anche **java.util.Vector** è un'esempio di **struttura dati**:
 - realizza un array che può crescere. Come negli array, agli elementi si può accedere tramite un **indice**. La dimensione dell'array può aumentare o diminuire, secondo le necessità, per permettere l'inserimento o la rimozione degli elementi dopo che il vettore è stato creato.

45

Tipi di dati astratti

- Un **tipo di dati astratto** (ADT, *Abstract Data Type*) è una rappresentazione **astratta** di una struttura dati, un modello che specifica:
 - il tipo di dati memorizzati
 - le operazioni che si possono eseguire sui dati
 - il tipo delle informazioni necessarie per eseguire le operazioni

46

Tipi di dati astratti

- In Java si definisce un **tipo di dati astratto** con una **interfaccia**
- Come sappiamo, un'interfaccia descrive un **comportamento** che sarà assunto da una classe che realizza l'interfaccia
 - è proprio quello che serve per definire un ADT
- Un ADT definisce **che cosa** si può fare con una struttura dati che realizza l'interfaccia
 - la classe che rappresenta concretamente la struttura dati definisce invece **come** vengono eseguite le operazioni

47

Tipi di dati astratti

- Un **tipo di dati astratto** mette in generale a disposizione metodi per svolgere le seguenti azioni
 - **inserimento** di un elemento
 - **rimozione** di un elemento
 - **ispezione** degli elementi contenuti nella struttura
 - **ricerca** di un elemento all'interno della struttura
- I diversi ADT che vedremo si differenziano per le modalità di funzionamento di queste tre azioni

48

Il pacchetto `java.util`

- Il pacchetto `java.util` della libreria standard contiene molte definizioni di ADT come interfacce e loro realizzazioni (*strutture dati*) come classi
- La nomenclatura e le convenzioni usate in questo pacchetto sono, però, piuttosto diverse da quelle tradizionalmente utilizzate nella teoria dell'informazione (*putroppo e stranamente...*)
- Quindi, proporremo un'esposizione teorica di ADT usando la terminologia tradizionale, senza usare il pacchetto `java.util` della libreria standard

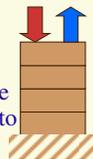
49

ADT Pila (*Stack*)

50

Pila (*stack*)

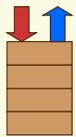
- In una *pila (stack)* gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **LIFO (Last In, First Out)**
 - l'ultimo oggetto inserito è il primo a essere estratto
 - il nome è stato scelto in analogia con una *pila* di piatti
- L'unico oggetto che può essere ispezionato è quello che si trova in cima alla pila
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
 - la JVM usa una pila per memorizzare l'elenco dei metodi in attesa durante l'esecuzione in un dato istante



51

Pila (*stack*)

- I browser per internet memorizzano gli indirizzi dei siti visitati recentemente in una struttura di tipo *pila*. Quando l'utente visita un sito, l'indirizzo è inserito (*push*) nella pila. Il browser permette all'utente di saltare indietro (*pop*) al sito precedente tramite il pulsante "indietro"
- Gli editor forniscono generalmente un meccanismo di "*undo*" che cancella operazioni di modifica recente e ripristina precedenti stati del testo. Questa funzione di "*undo*" è realizzata memorizzando le modifiche in una struttura di tipo pila.



52

Pila (*stack*)

- I metodi che caratterizzano una pila sono
 - `push()` per inserire un oggetto in cima alla pila
 - `pop()` per eliminare l'oggetto che si trova in cima alla pila. Genera l'eccezione `EmptyStackException` se la pila è vuota
 - `top()` per ispezionare l'oggetto che si trova in cima alla pila, senza estrarlo. Genera l'eccezione `EmptyStackException` se la pila è vuota
- Infine, ogni ADT di tipo "contenitore" ha i metodi
 - `isEmpty()` per sapere se il contenitore è vuoto
 - `makeEmpty()` per vuotare il contenitore
 - `size()` per conoscere il numero di elementi contenuti nella struttura dati

53

Estendere un'interfaccia

```
public interface Stack extends Container
{ ... // push, pop, top }
```

- Anche le *interfacce*, come le classi, possono essere "*estese*"
- Un'interfaccia eredita tutti i metodi della sua *super-interfaccia*
- Nel realizzare un'interfaccia *estesa* occorre realizzare anche i metodo della sua *super-interfaccia*

```
public interface Container
{ /**
 * @return true se il contenitore e' vuoto, false
 * altrimenti
 */
 boolean isEmpty();

 /** rende vuoto il contenitore */
 void makeEmpty();

 /** @return il numero di oggetti nel contenitore */
 int size();
}
```

Solita definizione
dell'interfaccia Container

54

EmptyStackException

```
public class EmptyStackException
    extends RuntimeException
{
    public EmptyStackException()
    {
    }

    public EmptyStackException(String err)
    {
        super(err);
    }
}
```

- Estendiamo l'eccezione `java.lang.RuntimeException`, così non siamo costretti a gestirla

55

Pila (stack)

- Definiremo tutti gli ADT in modo che possano genericamente contenere oggetti di tipo **Object**
- in questo modo potremo inserire nel contenitore oggetti di qualsiasi tipo, perché qualsiasi oggetto può essere assegnato a un riferimento di tipo **Object**

56

```
/**
 * Tipo di dati astratto con modalità di accesso LIFO
 * @see Container
 * @see EmptyStackException
 */
public interface Stack extends Container
{
    /**
     * inserisce un elemento in cima alla pila
     * @param obj l'elemento da inserire
     */
    void push(Object obj);

    /**
     * rimuove l'elemento dalla cima della pila
     * @return l'elemento rimosso
     * @throws EmptyStackException se la pila e' vuota
     */
    Object pop() throws EmptyStackException;

    /**
     * ispeziona l'elemento in cima alla pila
     * @return l'elemento in cima alla pila
     * @throws EmptyStackException se la pila e' vuota
     */
    Object top() throws EmptyStackException;
}
```

Osservazioni

```
@see Container
@see EmptyStackException
```

- `@see` è una direttiva per il programma di generazione automatica della documentazione `javadoc`.
- Nella documentazione viene segnalato di consultare l'oggetto che segue la direttiva

58

Osservazioni

```
Object pop() throws EmptyStackException
```

- L'eccezione `EmptyStackException` è a gestione facoltativa, quindi non è obbligatorio segnalare nell'intestazione del metodo
- Spesso lo si fa per segnalare chiaramente che il metodo può lanciare l'eccezione

```
pop
java.lang.Object pop()
    throws EmptyStackException

    rimuove l'elemento dalla cima della pila

Returns:
    l'elemento rimosso

Throws:
    EmptyStackException - se la pila e' vuota
```

59

Utilizzo della pila

- Per evidenziare la potenza della definizione di tipi di dati astratti come interfacce, supponiamo che qualcun altro abbia progettato la seguente classe

```
public class StackX implements Stack
{
    ...
}
```

- Senza sapere come sia realizzata `StackX`, possiamo usarne un esemplare mediante il suo comportamento astratto definito in `Stack`
- Allo stesso modo, possiamo usare un esemplare di un'altra classe `StackY` che realizza `Stack`

60

```
public class StackReverser // UN ESEMPIO
{ public static void main(String[] args)
  { Stack st = new StackX();
    st.push("Pippo");
    st.push("Pluto");
    st.push("Paperino");
    printAndClear(st);
    System.out.println();
    st.push("Pippo");
    st.push("Pluto");
    st.push("Paperino");
    printAndClear(reverseAndClear(st));
  }
  private static Stack reverseAndClear(Stack s)
  { Stack p = new StackY();
    while (!s.isEmpty())
      p.push(s.pop());
    return p;
  }
  private static void printAndClear(Stack s)
  { while (!s.isEmpty())
    System.out.println(s.pop());
  }
}
```

Paperino
Pluto
Pippo

Pippo
Pluto
Paperino

61

Realizzazione della pila

- ❑ Per *realizzare una pila* è facile ed efficiente usare una struttura di tipo *array* "riempito solo in parte"
- ❑ Il solo problema che si pone è *cosa fare quando l'array è pieno* e viene invocato il metodo *push()*
 - la prima soluzione proposta prevede il *lancio di un'eccezione*
 - la seconda soluzione proposta usa il *ridimensionamento dell'array*

62

Eccezioni nella pila

- ❑ Definiamo la classe *FullStackException*, come estensione di *RuntimeException*, in modo che chi usa la pila *non sia obbligato a gestirla*

```
public class FullStackException
  extends RuntimeException
  {
  }
```

Realizzazione minima di un'interfaccia

63

Realizzazione minima

```
public class FullStackException
  extends RuntimeException
  {
  }
```

- ❑ La classe è *vuota*, a cosa serve?
 - serve a definire un nuovo tipo di dato (un'eccezione) che ha le stesse identiche caratteristiche della superclasse da cui viene derivato, ma di cui interessa porre in evidenza il *nome*, che contiene l'informazione di identificazione dell'errore
- ❑ Con le eccezioni si fa talvolta così
 - in realtà la classe non è vuota, perché contiene tutto ciò che eredita dalla sua superclasse

64

```
public class FixedArrayStack implements Stack
{
  private final int CAPACITY = 100;

  // v è un array riempito solo in parte
  protected Object[] v; // considerare
  protected int vSize; // protected
  // uguale a private
  public FixedArrayStack()
  {
    /* per rendere vuota la struttura, invoco
       il metodo makeEmpty; è sempre meglio
       evitare di scrivere codice ripetuto
    */
    makeEmpty();
  }
  ...
}
```

65

```
/** rende vuoto il contenitore */
public void makeEmpty()
{ vSize = 0;
  v = new Object[CAPACITY];
}
/**
  verifica se il contenitore e' vuoto
  @return true se vuota, false altrimenti
*/
public boolean isEmpty()
{ return (vSize == 0);
}
/**
  calcola il numero di elementi nel contenitore
  @return il numero di elementi nel contenitore
*/
public int size()
{ return vSize;
}
...
}
```

CAPACITY
dimensione scelta per il contenitore

dato che Stack estende Container,
occorre realizzare
anche i suoi metodi

66

```
public class FixedArrayStack implements Stack
{
    ...
    public void push(Object obj) throws FullStackException
    { if (vSize >= CAPACITY)
      throw new FullStackException();
      v[vSize++] = obj;
    }
    public Object top() throws EmptyStackException
    { if (isEmpty())
      throw new EmptyStackException();
      return v[vSize - 1];
    }
    public Object pop() throws EmptyStackException
    { Object obj = top(); // lascia a top() l'eccezione
      vSize--;
      v[vSize] = null; //garbage collection
      return obj;
    }
}
```

67

Pila con ridimensionamento

- Definiamo una pila che non generi mai l'eccezione **FullStackException**

```
public class GrowingArrayStack implements Stack
{
    public void push(Object obj)
    { if (vSize >= v.length)
      v = resize(v, 2*vSize);
      v[vSize++] = obj;
    }

    private static Object[] resize(Object[] a, int length)
    {...}
    ... // tutto il resto è identico!
}
```

- Possiamo evitare di riscrivere tutto il codice di **FixedArrayStack** in **GrowingArrayStack**?

68

Pila con ridimensionamento

```
public class GrowingArrayStack extends FixedArrayStack
{
    public void push(Object obj)
    { if (vSize == v.length)
      v = resize(v, 2 * vSize);
      super.push(obj);
    }
    private static Object[] resize(Object[] a, int length)
    {...}
}
```

- Il metodo **push()** sovrascritto deve poter accedere alle variabili di esemplare della superclasse
- Questo è consentito dalla definizione **protected**
 - alle variabili **protected** *si può accedere dalle classi derivate*
 - *ma anche dalle classi dello stesso pacchetto!!!*
- Se le variabili fossero state private, non sarebbe stato possibile ridimensionare dinamicamente nella classe **GrowingArrayStack**

69

Accesso protected

- Il progettista della superclasse decide se rendere accessibile in modo **protected** lo stato della classe (o una sua parte...)
- È una violazione dell'incapsulamento, che avviene in modo consapevole ed esplicito
- Anche i metodi possono essere definiti **protected**
 - possono essere invocati soltanto all'interno della classe in cui sono definiti (come i metodi **private**) e *all'interno delle classi derivate da essa*

*In questo corso si scoraggia l'uso dello specificatore di accesso **protected**!!!*

- Talvolta può, però, essere utile per rendere più semplice (ma meno sicuro) il codice.

70

Prestazioni della pila

- Il tempo di esecuzione di ogni operazione su una **pila realizzata con array di dimensioni fisse** è **costante**, cioè non dipende dalla dimensione **n** della struttura dati stessa (non ci sono cicli...)
 - si noti che *le prestazioni dipendono dalla realizzazione della struttura dati* e non dalla sua interfaccia...
 - per valutare le prestazioni è necessario conoscere il codice che realizza le operazioni!

71

Prestazioni della pila

- Un'operazione eseguita in un tempo costante, cioè in un tempo che non dipende dalle dimensioni del problema, ha un andamento asintotico **O(1)**, perché
 - eventuali costanti moltiplicative vengono trascurate
- Ogni operazione eseguita su **FixedArrayStack** è quindi **O(1)**

72

Prestazioni della pila

- Nella realizzazione *con array ridimensionabile*, l'unica cosa che cambia è l'operazione **push()**
 - “alcune volte” richiede un tempo $O(n)$
 - tale tempo è necessario per copiare tutti gli elementi nel nuovo array, all'interno del metodo **resize()**
 - il ridimensionamento viene fatto ogni n operazioni
 - cerchiamo di valutare il *costo medio* di ciascuna operazione
 - tale metodo di stima si chiama *analisi ammortizzata* delle prestazioni asintotiche

73

Analisi ammortizzata

- Dobbiamo calcolare il valore medio di n operazioni, delle quali
 - $n-1$ richiedono un tempo $O(1)$
 - una richiede un tempo $O(n)$
$$\langle T(n) \rangle = [(n-1) * O(1) + O(n)] / n$$

$$= O(n) / n = O(1)$$
- Distribuendo il tempo speso per il ridimensionamento in parti uguali a tutte le operazioni **push()**, si ottiene quindi ancora $O(1)$

74

Analisi ammortizzata

- Le prestazioni medie di **push()** con ridimensionamento rimangono $O(1)$ per qualsiasi costante moltiplicativa usata per calcolare la nuova dimensione, anche diversa da 2
- Se, invece, si usa una costante *additiva*, cioè la dimensione passa da n a $n+k$, si osserva che su n operazioni di inserimento quelle “lente” sono n/k

$$\langle T(n) \rangle = [(n - n/k) * O(1) + (n/k) * O(n)] / n$$

$$= [O(n) + n * O(n)] / n$$

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

$$= O(1) + O(n) = O(n)$$

75

Lezione XXXI Me 28-Nov-2007

ADT Pila (Stack)

76

Pila realizzata con una catena

- Una pila può essere realizzata efficientemente anche usando una lista concatenata 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 $O(1)$
 - `removeFirst()` è $O(1)$
 - `removeLast()` è $O(n)$

77

```
public class LinkedListStack implements Stack
{ private LinkedList list;
  private int size; // per contare gli elementi

  public LinkedListStack() // O(1)
  { makeEmpty();
  }

  public void makeEmpty() //O(1)
  { list = new LinkedList();
    size = 0;
  }

  public boolean isEmpty() //O(1)
  { return list.isEmpty();
  }

  public int size() // O(1)
  { return size;
  }
  ...
}
```

Pila realizzata con una catena

```
public class LinkedListStack implements Stack
{
    ...
    public void push(Object obj) // O(1)
    {
        list.addFirst(obj);
        size++;
    }

    public Object top() throws EmptyStackException
    {
        if (isEmpty()) // O(1)
            throw new EmptyStackException();
        return list.getFirst();
    }

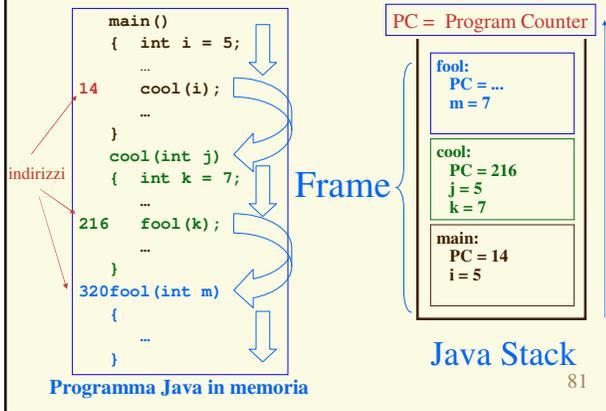
    public Object pop() throws EmptyStackException //O(1)
    {
        Object obj = top(); // lascia a top() il lancio
        list.removeFirst(); // dell'eccezione
        size--;
        return obj;
    }
}
```

Pile nella Java Virtual Machine

- ❑ Ciascun programma java in esecuzione ha una propria pila chiamata **Java Stack** che viene usata per mantenere traccia delle **variabili locali**, dei **parametri formali dei metodi** e di altre importanti informazioni relative ai metodi, man mano che questi sono invocati
- ❑ Più precisamente, durante l'esecuzione di un programma, JVM mantiene uno **stack** i cui elementi sono **descrittori** dello stato corrente dell'invocazione dei metodi (che non sono terminati)
- ❑ I descrittori sono denominati **Frame**. A un certo istante durante l'esecuzione, ciascun metodo sospeso ha un **frame** nel **Java Stack**

80

Java Stack (Runtime Stack)



81

Pile nella Java Virtual Machine

- ❑ Il metodo **fool** è chiamato dal metodo **cool** che a sua volta è stato chiamato dal metodo **main**.
- ❑ Ad ogni invocazione di metodo in run-time viene inserito un frame nello stack
- ❑ Ciascun frame dello stack memorizza i valori del **program counter**, dei **parametri** e delle **variabili locali** di una **invocazione a un metodo**.
- ❑ Quando il metodo chiamato è terminato il frame viene estratto ed eliminato
- ❑ Quando l'invocazione del metodo **fool** termina, l'invocazione del metodo **cool** continuerà la sua esecuzione dall'istruzione di indirizzo 217, ottenuto incrementando il valore del program counter contenuto nel frame del metodo cool

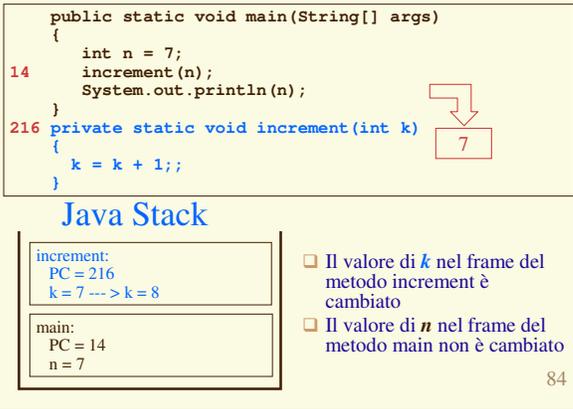
82

Passaggio dei parametri ai metodi

- ❑ Il **java stack** presiede al passaggio dei parametri ai metodi.
- ❑ Il meccanismo usa il passaggio dei **parametri per valore**. Significa che il valore corrente di una variabile (o di un'espressione) è ciò che è passato al metodo.
- ❑ Nel caso di una variabile **x** di un **tipo fondamentale del linguaggio**, come **int** o **double**, il valore corrente di **x** è semplicemente il numero associato a **x**.
- ❑ Quando questo valore è passato a un metodo, esso è assegnato a una variabile locale nel frame del metodo chiamato.
- ❑ Se il metodo chiamato cambia il valore di questa variabile locale, non cambierà il valore della variabile nel metodo chiamante

83

Passaggio dei parametri per valore



84

Passaggio dei parametri per valore

- ❑ Nel caso la variabile x sia una variabile riferimento a un oggetto, il valore corrente di x è l'indirizzo in memoria dell'oggetto.
- ❑ Quando x viene passato come parametro, ciò che è copiato nel frame del metodo chiamato è l'indirizzo dell'oggetto in memoria.
- ❑ Quando questo indirizzo è assegnato a una variabile locale y nel metodo chiamato, y si riferirà allo stesso oggetto a cui si riferisce x.
- ❑ Se il metodo chiamato cambia lo stato interno dell'oggetto a cui y si riferisce, cambia anche lo stato dell'oggetto a cui x si riferisce perché è lo stesso oggetto.

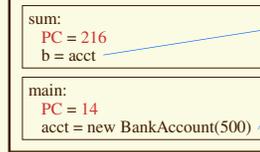
85

Passaggio dei parametri per valore

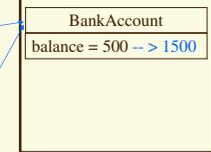
```
public static void main(String[] args)
{
    BankAccount acct = new BankAccount(500);
14    sum(acct);
    System.out.println(acct.getBalance());
}
216 private static void sum(BankAccount b)
{
    b.deposit(1000);
}
```



Java Stack



Java Heap



86

ADT

PILA (STACK)
LIFO

```
public interface Stack extends Container
{
    void push(Object obj);
    Object top() throws EmptyStackException;
    Object pop() throws EmptyStackException;
}
```

Realizzata con Complessita' Temporale

Array (operazioni in coda)	push()	O(1)
Lista concatenata	top()	O(1)
	pop()	O(1)

`java.util.Stack` (con qualche differenza)

87

ADT Coda (Queue)

88

ADT 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

89

ADT 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
 - `size()` per contare gli elementi presenti

90

ADT Coda (queue)

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

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

91

ADT Coda (queue)

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

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

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

ADT 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

93

ADT 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 EmptyQueueException extends RuntimeException
{ }
```

94

```
public class SlowFixedArrayQueue implements Queue
{
  private static final int CAPACITY = 100;
  private Object[] v;
  private int vSize;

  public SlowFixedArrayQueue()
  { makeEmpty();
  }

  public void makeEmpty()
  { v = new Object[CAPACITY];
    vSize = 0;
  }

  public boolean isEmpty()
  { return (vSize == 0);
  }

  public int size()
  { return vSize;
  }
  ...
  // continua
```

Coda realizzata
con un array
a dimensione fissa

```
... // 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(); // genera eccezione
  for (int i = 0; i < vSize - 1; i++)
    v[i] = v[i+1];
  vSize--;
  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

97

Coda (queue) su array con due indici

- Per realizzare una coda più efficiente servono *due indici* anziché uno soltanto
 - `front`: indice che punta al primo elemento nella coda
 - `back`: indice che punta al *primo posto libero dopo l'ultimo elemento* nella coda
 - quando `front` raggiunge `back` l'array è vuoto
 - notare che gli elementi nell'array sono $(back - front)$
- 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 $O(1)$



98

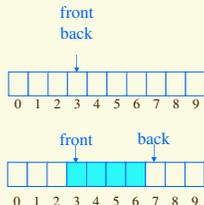
```
public class FixedArrayQueue implements Queue
{
    static final int CAPACITY = 100;
    protected Object[] v;
    protected int front, back;

    public FixedArrayQueue()
    { makeEmpty(); }

    public void makeEmpty()
    { v = new Object[CAPACITY];
      front = back = 0; }

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

    public int size()
    { return back - front; }
} // continua
```



Coda (queue) su array con due indici

```
... // continua
public void enqueue(Object obj)
    throws FullQueueException
{ 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 a vista per la pila, estendendo la classe `FixedArrayQueue` e sovrascrivendo il solo metodo `enqueue()`

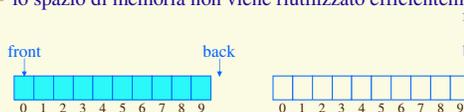
```
public class GrowingArrayQueue extends FixedArrayQueue
{
    public void enqueue(Object obj)
    { if (back >= v.length)
      v = resize(v, 2 * v.length);
      super.enqueue(obj); }

    private static Object[] resize(Object[] a, int length)
    { ... }
}
```

101

Prestazioni della coda realizzata con array

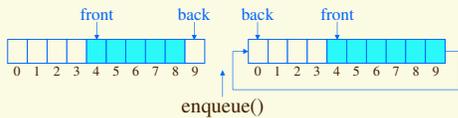
- La realizzazione di una coda con un array e due indici ha la massima efficienza in termini di prestazioni temporali, tutte le operazioni sono $O(1)$, ma ha ancora un punto debole
- Se l'array ha n elementi, proviamo a
 - effettuare n operazioni `enqueue()`
 - e poi
 - effettuare n operazioni `dequeue()`
- Ora la coda è vuota, ma alla successiva operazione `enqueue()` l'array sarà pieno
 - lo spazio di memoria non viene riutilizzato efficientemente



102

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



103

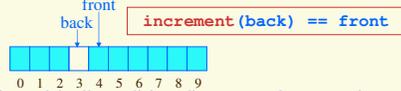
```
public class FixedCircularArrayQueue extends FixedArrayQueue
{ // il metodo increment() fa avanzare un indice di una
  // posizione, tornando all'inizio dell'array se si
  // supera la fine
  protected int increment(int index)
  { return (index + 1) % v.length;
  }
  public void enqueue(Object obj) throws
    FullQueueException
  { if (increment(back) == front)
    throw new FullQueueException();
    v[back] = obj;
    back = increment(back);
  }
  public Object dequeue() throws EmptyQueueException
  { Object obj = getFront();
    v[front] = null; // garbage collector
    front = increment(front);
    return obj;
  }
  public int size()
  { return (v.length - front + back) % v.length; }
}
```

non serve sovrascrivere
getFront() perché non
modifica le variabili back
e front

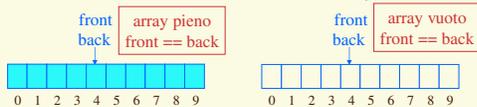
Osservazioni

```
public void enqueue(Object obj)
{ if (increment(back) == front)
  throw new FullQueueException();
  v[back] = obj;
  back = increment(back);
}
```

- L'eccezione viene generata quando nell'array c'è ancora un indice libero



- Questo è necessario perché nella condizione di array completamente pieno avremmo front == back, che è la condizione di array vuoto



105

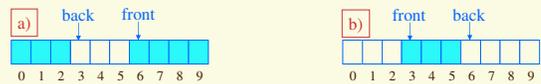
Osservazioni

```
public int size()
{ return (v.length - front + back) % v.length;
}
```

- Se back = front l'espressione ritorna 0; è corretto perché questa è la condizione di array vuoto (si veda la discussione precedente)
- a) se back < front, il numero di elementi è pari a back più gli elementi fra front e la fine dell'array (v.length - front)

$$n = v.length - front + back < v.length \Rightarrow$$

$$\Rightarrow n = (v.length - front + back) \% v.length$$



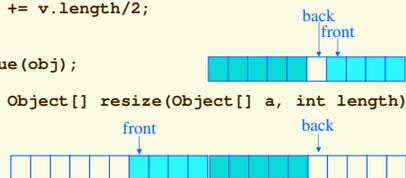
- b) se back > front, allora gli elementi sono back - front

$$n = back - front < v.length \Rightarrow$$

$$\Rightarrow n = (v.length - front + back) \% v.length$$

106

```
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);
  }
  private static Object[] resize(Object[] a, int length)
  {...}
}
```



Coda realizzata con una lista concatenata

- Anche una coda può essere realizzata usando una lista concatenata 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*

108

Coda realizzata con una catena

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

  public LinkedListQueue ()
  { makeEmpty();
  }

  public void makeEmpty()
  { list = new LinkedList();
    size = 0;
  }

  public boolean isEmpty()
  { return list.isEmpty();
  }

  public int size()
  { return size;
  }
  ...
}
```

Coda realizzata con una catena

```
public class LinkedListQueue implements Queue
{ ...
  // O(1)
  public void enqueue(Object obj)
  { list.addLast(obj);
    size++;
  }
  // O(1)
  public Object getFront() throws EmptyQueueException
  { if (isEmpty())
    throw new EmptyQueueException();
    return list.getFirst();
  }
  // O(1)
  public Object dequeue() throws EmptyQueueException
  { Object obj = getFront(); // lancia eccezione
    list.removeFirst();
    size--;
    return obj;
  }
}
```

Estrarre oggetti

- 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 **forzamento** per ottenere un riferimento del tipo originario

```
...
st.push("Hello");
Object obj = st.pop();
char c = obj.charAt(0);
...

...
st.push("Hello");
Object obj = st.pop();
String str = (String)obj;
char c = str.charAt(0);
```

ClassName.java:9: cannot find symbol
symbol : method charAt(int)
location: class java.lang.Object
char c = obj.charAt(0);
1 error

'H'

111

Estrarre oggetti

```
String str = (String)st.pop();
```

- Sappiamo che serve il forzamento 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 **String**?
 - viene lanciata l'eccezione **ClassCastException**
- Possiamo scrivere codice che si comporti in modo più sicuro?

112

Estrarre oggetti

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

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

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

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

113

ADT	CODA (QUEUE) FIFO	
	<pre>public interface Queue extends Container { void enqueue(Object obj); Object getFront() throws EmptyQueueException; Object dequeue() throws EmptyQueueException; }</pre>	
	Realizzata con	Complessita' Temporale
	Array circolare	enqueue() O(1)
		getFront() O(1)
	Lista concatenata	dequeue() O(1)
Non c'è implementazione nella libreria standard interfaccia java.util.Queue (con qualche differenza) ¹⁴		

Lezione XXXII Gi 29-Nov-2007

Strutture dati contenenti oggetti e numeri

115

Strutture dati di oggetti e di numeri

- Abbiamo visto che la pila e la coda come definite sono in grado di gestire dati di qualsiasi tipo, cioè riferimenti a oggetti di qualsiasi tipo (stringhe, conti bancari...)
- Non sono però in grado di gestire dati dei *tipi fondamentali* definiti nel linguaggio Java (**int**, **double**, **char**...)
- tali tipi di dati NON sono oggetti
- Come possiamo gestire una struttura dati di numeri? Ad esempio una pila?

116

Pile di numeri

- Possiamo ridefinire tutto

```
public interface IntStack
    extends Container
{ void push(int obj);
  int top();
  int pop();
}
```

```
public class FixedArrayIntStack implements IntStack
{ protected int[] v; protected int vSize;
  public FixedArrayIntStack()
  { makeEmpty(); }
  public void makeEmpty(){v=new int[100]; vSize =0;}
  public boolean isEmpty(){ return (vSize == 0); }
  public int size() { return vSize; }
  public void push(int obj) throws FullStackException
  { if(vSize == v.length)
    throw new FullStackException();
    v[vSize++] = obj; }
  public int top() throws EmptyStackException
  { if (isEmpty()) throw new EmptyStackException();
    return v[vSize - 1]; }
  public int pop()
  { int obj = top(); vSize--; return obj; }
}
```

Pile di numeri

- La ridefinizione della pila per ogni tipo di dato fondamentale ha alcuni svantaggi
- occorre replicare il codice nove volte (i tipi di dati fondamentali sono otto), con poche modifiche
- non esiste più il tipo di dati astratto **Stack**, ma esisterà **IntStack**, **DoubleStack**, **CharStack**, **ObjectStack**
- Cerchiamo un'alternativa, ponendoci un problema
- è possibile “trasformare” un numero intero (o un altro tipo di dato fondamentale di Java) in un oggetto?
- La risposta è affermativa
- si usano le *classi involucre* (*wrapper*)

118

Classi involucre

- Per dati **int** esiste la *classe involucre* **Integer**
- il costruttore richiede un parametro di tipo **int** e crea un oggetto di tipo **Integer** che contiene il valore intero, “avvolgendolo” con la struttura di un oggetto

```
Integer intObj = new Integer(2);
Object obj = intObj; // lecito
```

- gli oggetti di tipo **Integer** sono *immutabili*
- per conoscere il valore memorizzato all'interno di un oggetto di tipo **Integer** si usa il metodo non statico **intValue**, che restituisce un valore di tipo **int**

```
int x = intObj.intValue(); // x vale 2
```

119

Classi involucre

- Esistono *classi involucre* per tutti i tipi di dati fondamentali di Java, con nomi uguali al nome del tipo fondamentale, ma iniziale maiuscola
- eccezioni: **Integer** e **Character**
- Ogni classe fornisce un metodo per ottenere il valore contenuto al suo interno, con il nome corrispondente al tipo
- es: **booleanValue()**, **charValue()**, **doubleValue()**
- Tutte le classi involucre si trovano nel pacchetto **java.lang** e realizzano l'interfaccia parametrica **Comparable<T>**

120

Classi Involucro (Wrappers)

- ❑ Boolean
- ❑ Byte
- ❑ Character
- ❑ Short
- ❑ Integer
- ❑ Long
- ❑ Float
- ❑ Double

```
// tipo fondamentale del linguaggio
double x = 3.5;

// Oggetto che incapsula un tipo
// fondamentale
Double d = new Double(3.5);

Double d = new Double(3.5);
// d e' un oggetto di classe Double
```

121

Auto-boxing

- ❑ In Java 5.0, se un tipo fondamentale viene assegnato a una variabile della corrispondente classe involucro, viene creato automaticamente un oggetto della classe

```
Double d = 3.5;
// equivale a
Double d = new Double(3.5);
```

- ❑ Questo tipo di conversione automatica si indica col nome di *auto-boxing*
- ❑ Non useremo l'auto-boxing, perché tende a confondere i tipi fondamentali con gli oggetti, mentre la loro natura in java è diversa

122

Algoritmi per liste concatenate

123

Catena: conteggio elementi

- ❑ Per contare gli elementi presenti in una catena è necessario
 - inserire una variabile intera e aggiornarla quando si inseriscono o estraggono elementi
 - oppure scorrere tutta la catena, come nel metodo seguente

```
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;
  }
}
```

124

Algoritmi per Liste Concatenate

- ❑ 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*

125

Algoritmi per Liste Concatenate

- ❑ 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

126

Iteratore in una Lista Concatenata

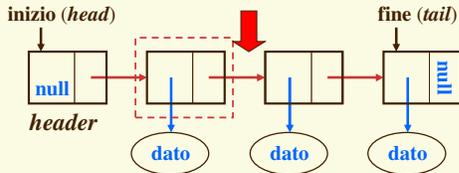
127

Iteratore in una Lista Concatenata

- 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

128

Iteratore in una Lista Concatenata



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

129

Iteratore in una Lista Concatenata

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

  // se l'iteratore si trova alla fine della
  // 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();
}
```

130

Iteratore in una Lista Concatenata

- 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());
```

131

Java.util.Scanner

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

- La classe `java.util.Scanner` realizza l'interfaccia `java.util.Iterator`!

132

Iteratore in una Lista Concatenata

- ❑ 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

133

Iteratore in una Lista Concatenata

- ❑ Per un corretto funzionamento dell'iteratore occorre concedere a tale oggetto il pieno accesso alla catena
 - in particolare, alla sua variabile di esempio `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

134

```
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;
      }
    }
    ...
}
```

135

```
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();
        }
    }
}
```

136

Lista Concatenata : 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;
}
```

137

Lista Concatenata : 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

138

Iteratore in una Lista Concatenata

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

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

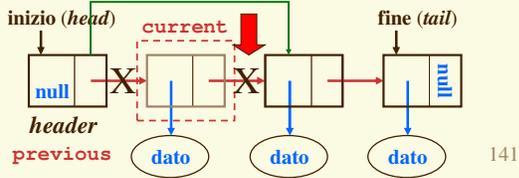
    //elimina l'ultimo nodo esaminato da next()
    //senza modificare la posizione dell'iteratore;
    //può essere invocato solo dopo un'invocazione
    //di next() (lancia IllegalStateException)
    void remove();
}
```

139

```
private class LinkedListIterator . . .
{
    //fa avanzare l'iteratore
    public void add(Object obj)
    {
        ListNode n = new ListNode(obj, current.getNext());
        current.setNext(n);
        current = n;
        previous = null; // non si puo' invocare remove
    }
}
```

140

```
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
    }
}
```



141

Copia da una Lista Concatenata 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());
}
```

142

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

143

Più iteratori sulla stessa lista

```
LinkedList 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*

144

ADT Lista

145

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

146

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)

147

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

148

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
 - catena

149

Liste con rango e posizionali

Da F. Bombi

Modificato da A. Luchetta

150

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**

151

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**

152

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

153

Eliminare i doppi - 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**, **abbiamo quindi una complessità temporale $O(1)$**

154

Classi e interfacce di `java.util`

- Le interfacce `List` e `ListIterator` 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

155

ADT	LISTA
<pre>public interface List extends Container { ListIterator getIterator(); } public interface ListIterator { boolean hasNext(); void add(Object obj); Object next() throws NoSuchElementException; void remove() throws IllegalStateException; }</pre>	
Realizzata con	Complessità Temporale
<code>Array</code>	<code>hasNext ()</code> $O(1)$ $O(1)$
<code>Lista concatenata</code>	<code>next ()</code> $O(1)$ $O(1)$
	<code>add ()</code> $O(n)$ $O(1)$
	<code>remove ()</code> $O(n)$ $O(1)$
Interfaccia <code>java.util.List</code> , <code>Iterator</code> , <code>ListIterator</code> (con qualche differenza)	
Strutture dati: <code>java.util.ArrayList</code> , <code>Vector</code> , <code>LinkedList</code> (con qualche differenza)	

156