Esercizio su Alberi

Specifiche Esercizio Alberi

Si richiede di progettare ed implementare un componenete che gestisce una struttura dati ad albero binario di ricerca per un insieme di stringhe. In particolare, la relazione < tra le stringhe è quella lessicografica. In particolare il componente software deve consentire di

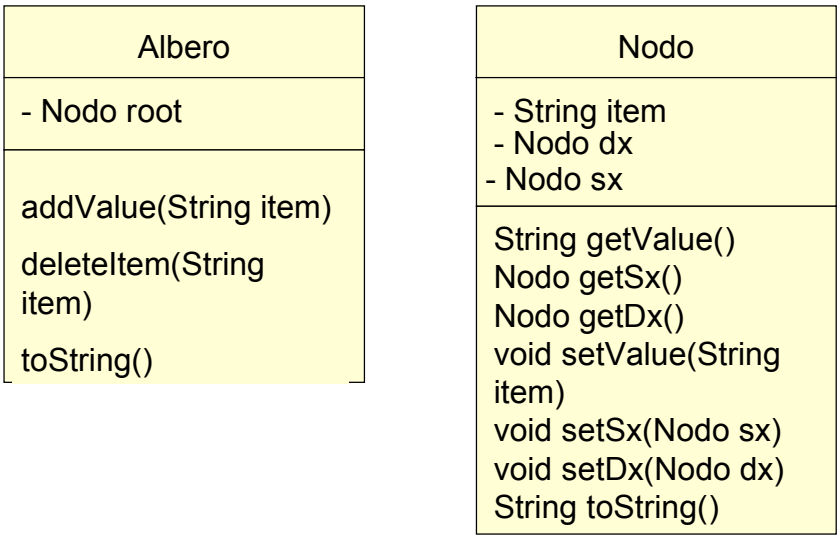
1) Inserire una nuova stringa nell'albero

2) Cancellare una stringa

3) Verificare se una stringa è stata inserita in precedenza

4) Restituire una stringa ottenuta concatenando in ordine alfabetico tutte le stringhe memorizzate nell'albero

Traccia Di Soluzione



Nodo

```
class Nodo {
        String item;
        Nodo dx;
        Nodo sx;

        public Nodo(String item){
         this.item = item;
         dx = null;
         sx = null;
        }
        public Nodo(String item, Nodo sx, Nodo dx){
         this.item = item;
         this.sx = sx;
         this.dx = dx;
        }
```

Nodo

```
public String getValue() { return item; }
        public Nodo getSx() { return sx; }
        public Nodo getDx() { return dx; }

        public void setValue(String item) { this.item = item; }
        public void setSx(Nodo sx) { this.sx = sx; }
        public void setDx(Nodo dx) { this.dx = dx; }


        public String toString() {
              return item;
        }
}
```


Albero Binario: inserimento

```
public class AlberoBinario
{Nodo root;

public AlberoBinario() {root = null;}

public void addValue(String value) {
root = insert(root, value); }

private Nodo insert(Nodo actual, String value){
if (actual == null) return new Nodo(value);
if (actual.getValue().equals(value)) return actual;
if (actual.getValue().compareTo(value)>0)
            actual.setSx(insert(actual.getSx(), value));
else actual.setDx(insert(actual.getDx(), value));
return actual;
}
```

Albero Binario: visita completa dell'albero

```
public String toString()
{return stringa(root);}

private String stringa(Nodo actual){
String app ;
if (actual == null) app = "";
else app =
stringa(actual.getSx())+""+actual.getValue()+"
"+stringa(actual.getDx());

return app;}
```


Albero Binario: ricerca

```
 public boolean isPresente(String ricerca){
      return (cerca(ricerca, root)!=null);
    }

private Nodo cerca(String ricerca, Nodo actual){
Nodo app;

if (actual == null) return null;
if (actual.getValue().equals(ricerca)) return actual;
app = cerca (ricerca, actual.getSx());
if (app!=null) return app;
return cerca (ricerca, actual.getDx());
}
```

Albero Binario: cancellazione

```
public void elimina(String item) {root = cancella(item, root);}
private Nodo cancella(String ricerca, Nodo actual){
         Nodo app;
         Nodo app1;
        if (actual == null) return null;
        if (actual.getValue().equals(ricerca)) {
            if (actual.getSx()==null) return actual.getDx();
            if (actual.getDx()==null) return actual.getSx();
            app = insAlbero(actual.getSx(), actual.getDx());
            return app;}
         app = cancella (ricerca, actual.getSx());
         if (app!=actual.getSx()){
            actual.setSx(app);
            return actual;
         } else {
            app1 = cancella(ricerca, actual.getDx());
            actual.setDx(app1);
       }
        return actual;}
private Nodo insAlbero(Nodo actual, Nodo nuovoRamo){
       if (actual == null) return nuovoRamo;
       actual.setDx(insAlbero(actual.getDx(), nuovoRamo));
       return actual;
    }
```


Liste

STRUTTURE DATI: OLTRE GLI ARRAY

- le strutture dati progettate per ospitare una *collezione di elementi,* sono variazioni di array
- Ma l'array ha dimensione fissa anche in Java
  - determinata *a priori*, in linguaggi statici
  - determinata *al momento della creazione*, se definiti dinamicamente (Java)
- Molti problemi richiedono strutture dati in grado di rappresentare collezioni di elementi *di numero variabile dinamicamente*
- Efficienza nell'uso delle risorse

LISTE

- Una *lista* è una *collezione di elementi* organizzati concettualmente *in sequenza*
- La dimensione della *lista non* è *prefissata*

Progetto in Java

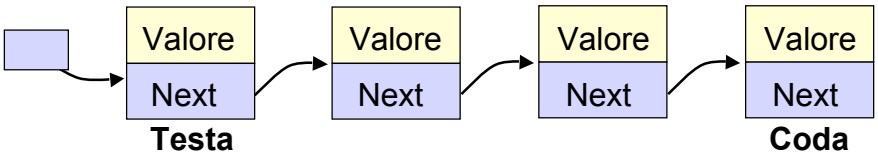
- Uso della semantica per riferimento
- Realizzare delle classi corrispondenti

Decisioni progettuali

- Una lista *come contenitore* o come valore
- Quali operazioni? Quali costruttori?
- Quali operazioni *primitive*?

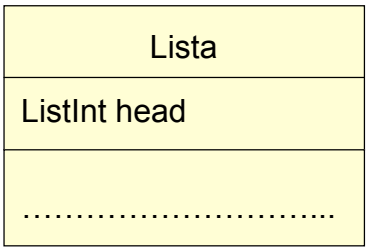
LISTE: RAPPRESENTAZIONE

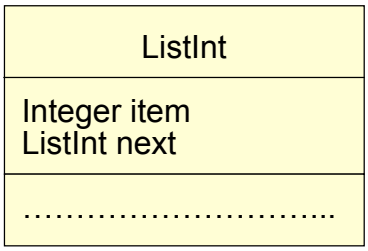
- Una *lista* è spesso rappresentata sotto forma di *sequenza di nodi concatenati*

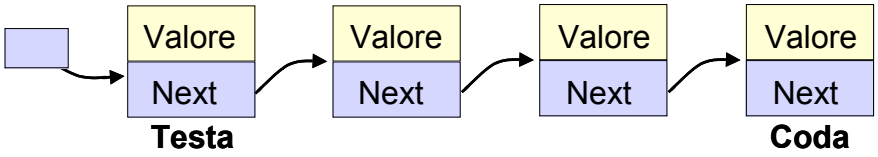


- Ogni *nodo* della lista è fatto di *due parti*: un *valore* e un *riferimento al nodo successivo*
- La *rappresentazione fisica* può essere:
  - basata su puntatori e allocazione dinamica
  - basata su oggetti creati dinamicamente
  - basata su altre rappresentazioni (file, array, …)

Esempio Liste di Interi

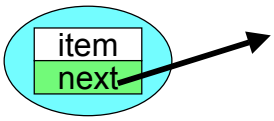






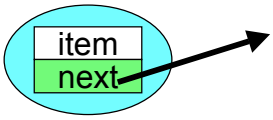
LISTE: funzionalità

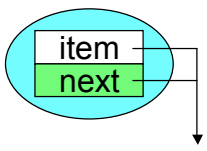
```
public class ListInt {
  private ListInt next;
  private Integer item;

public ListInt() {}
public void  insert (int i) {}
public Integer extract () {}

public boolean isNotEmpty () {}
public void printList () {}
}
```



LISTE: funzionalità

```
public class ListInt {
  private ListInt next;
  private Integer item;

public ListInt() {next = null; item =
  null;}
}
```

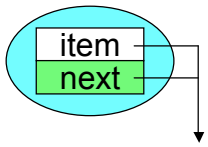


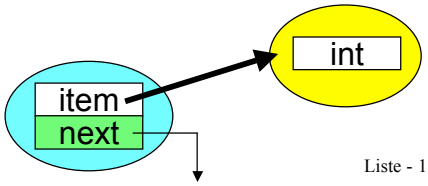


L'elemento viene creato senza riferire a nulla
Si noti che consideriamo di riferire un altro (next) e un wrapper Integer (item)

LISTE: inserimento

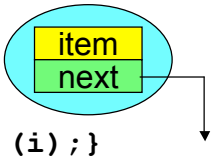
```
public class ListInt {
  private ListInt next;
  private Integer item;
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else {
  ListInt temp = next;
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```

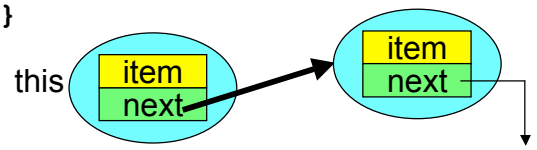




LISTE: inserimento

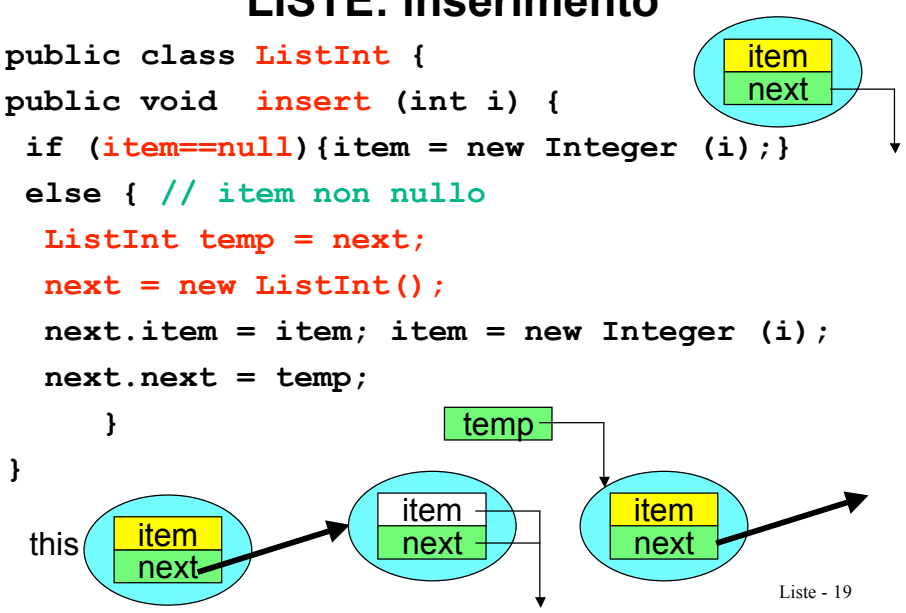
```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next;
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```





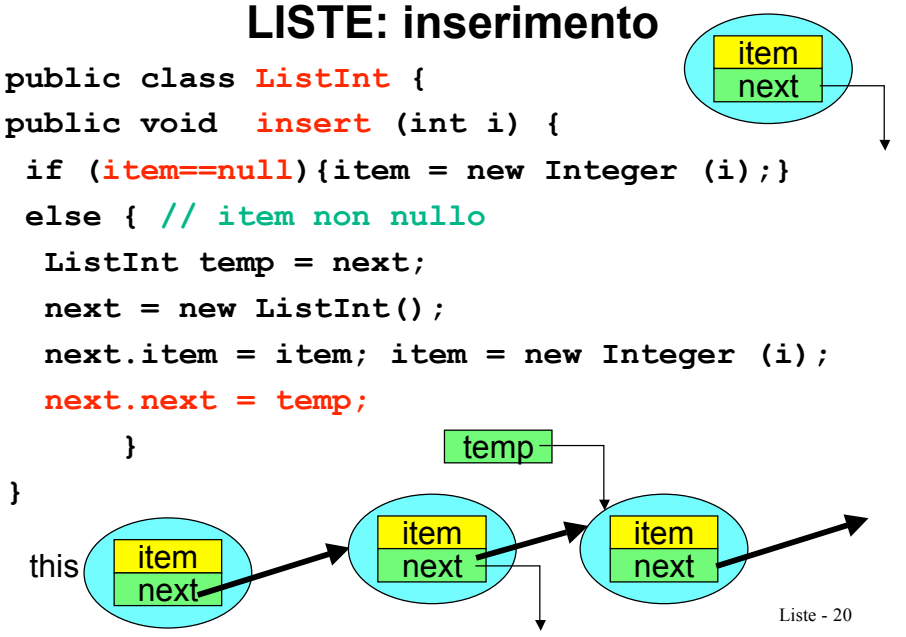
LISTE: inserimento

```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next;
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```



LISTE: inserimento

```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next;
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```



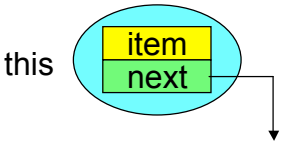
LISTE: estrazione

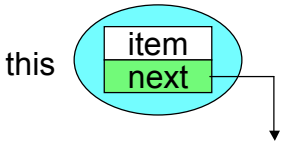
```
public Integer extract () { Integer iI;
 if (item == null) {return null;}
 else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```

Si noti il ruolo di Integer (wrapper) per indicare che non si restituisce alcun valore

LISTE: estrazione

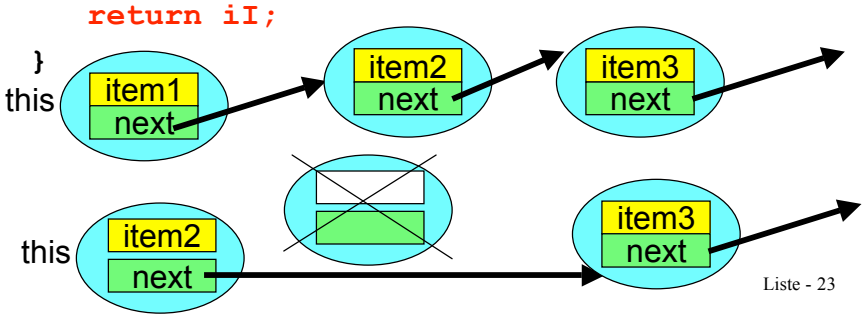
```
public Integer extract () { Integer iI;
 … else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```





LISTE: estrazione

```
public Integer extract () { Integer iI;
 … else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```



LISTE: funzionalità

```
public class ListInt {
 private ListInt next; private Integer
  item;
public ListInt() {...}
public void  insert (int i) {...}
public Integer extract () {...}

public boolean isNotEmpty ()
   {return (item != null);}
public void printList () {}
// stampa la lista in formato esterno
}
```

LISTE: printList

```
public void printList () {
ListInt temp = next; int i = 1;
if (item == null) System.out.println (" Lista vuota ");
else
{System.out.println(" Elem N.1 vale "+ temp.item);
 while (temp != null)
  { i++;
    System. out. println (" Elem N." + i +
                " vale " + temp.item);
    temp = temp. next;
  }
}}
```


LISTE: toString

```
public String toString () {
ListInt temp = next; int i = 1; String s;
if (item == null) return " Lista vuota \n";
else
{s = "\n Lista con elementi " + item;
 while (temp != null)
 { s = s + "\t" + temp.item;
   temp = temp.next;  i++; }
 return  s = s + "\n In totale  sono "
               + i + " elementi\n";
}
}
```

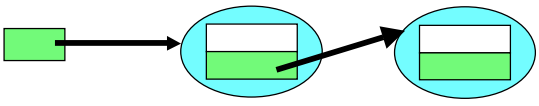
USO da parte di CLIENTI

```
public static void main(String[] args) {
  int i;  Integer iI;
ListInt first = new ListInt ();
first.insert(2); first.insert(6);
first.insert(17);
iI = first.extract();
if (iI != null) { i = iI.intValue(); ...}
first.insert (19);
first.printList ();
while (first.isNotEmpty ())
 { i = first.extract().intValue();}
}
```


PROGETTO DI LISTE IN JAVA

`List`: quale rappresentazione?

- approccio classico:
  - un campo `value` che rappresenta il valore (`Object`)
  - un campo `next` che punta al prossimo nodo (`AnyList`)

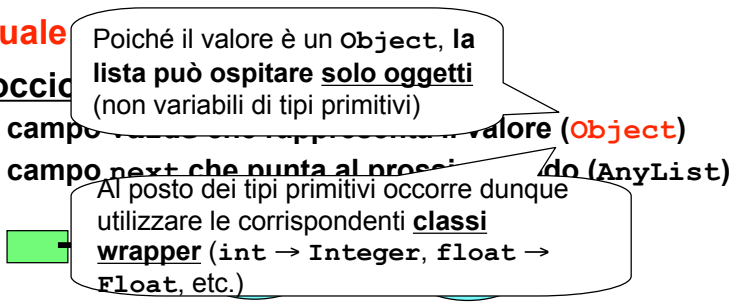


- semplice, ma con alcuni difetti
  - la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)
  - ciò rende difficile avere semantiche di costruzione diverse da quella predefinita (inserimento in testa)

PROGETTO DI LISTE IN JAVA

`List`: quale

- **approccio**
  - **un campo [illegible] valore (`Object`)**
  - **un campo `next` che punta al prossimo (`AnyList`)**



- **semplice, ma con alcuni difetti**
  - **la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)**
  - **ciò rende difficile avere semantiche di costruzione diverse da quella predefinita (inserimento in testa)**

ANALISI CRITICA DEL PROGETTO

L'approccio classico è soddisfacente?

- **non del tutto, in quanto *la lista non esiste come entità esplicita*: coincide con il riferimento al primo nodo**
  - **la classe `List` in realtà rappresenta *il nodo della lista*, non la lista in quanto tale!**
- **ciò rende impossibile avere semantiche di costruzione *diverse da quella predefinita* (inserimento in testa)**
  - **ad esempio, è impossibile derivare da `List` una classe `SortedList` che costruisca *liste ordinate***

ANALISI CRITICA DEL PROGETTO

Perché questo?

- Perché il costruttore di una tale ipotetica classe `SortedList` ***chiamerebbe come prima cosa il costruttore della classe base*** `List`, che crea già un nuovo nodo e lo inserisce in testa!

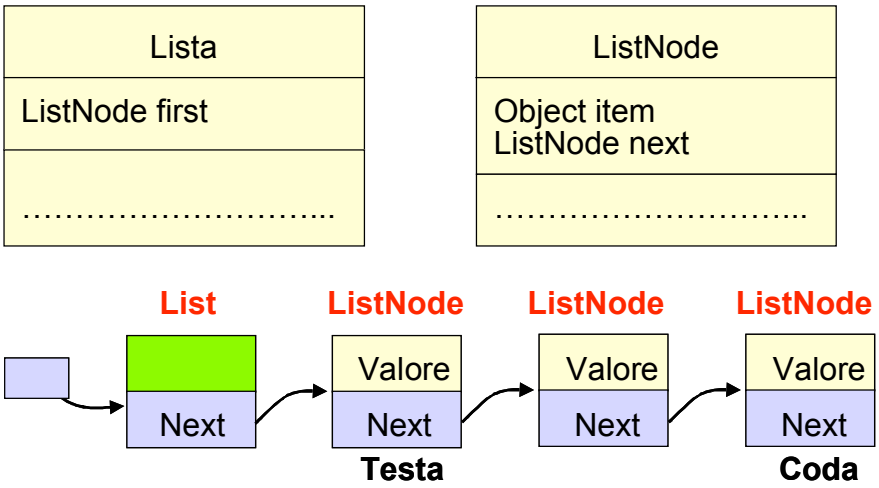
- Nessuno può revocare quel comportamento, ***indipendentemente dal costruttore di*** `SortedList`
- Il cliente riceverebbe ***in ogni caso*** il riferimento al nuovo nodo creato in testa.

REVISIONE DEL PROGETTO

Una diversa rappresentazione

- `List` rappresenta la lista: è un oggetto che contiene *la radice della sequenza di nodi*
  - tale radice è *un riferimento a un oggetto* `ListNode`
- `ListNode` rappresenta il nodo della lista
  - contiene il valore e il riferimento al nodo successivo
  - è quello che prima chiamavamo impropriamente lista

REVISIONE DEL PROGETTO



REVISIONE DEL PROGETTO

Perché questo approccio risolve il problema?

- **Perché ora creare una `List` *non implica più* creare anche un nodo!**
- **Quindi, il costruttore di `List` può gestire direttamente la creazione del nodo, *inserendolo dove ritiene più appropriato***
- **Ridefinendo il proprio costruttore, una sottoclasse `SortedList` può cambiare tale comportamento**
- **Il cliente riceve *in ogni caso* il riferimento alla nuova lista creata, non già a uno specifico nodo!**

Nuovo Progetto

```
public class ListNode implements Iposition {
// Implementazione dell'interfaccia "IPosition"
protected Object item; protected ListNode next;

public ListNode(Object o) { item = o; next=null;}
public Object getItem(){ return item;}

public ListNode getNext() { return next;}
public Object container() {return null; }
// modificare per riferire la lista cui nodo appartiene
public void setNext(ListNode next){ this.next=next;}
}
```


INTERFACCIA per NODO di LISTA

```
public interface IPosition
{
  public Object getItem ();
  public Object container();
}
```

La Interfaccia consente ad un nodo di trovare il prossimo e di riferire una ed una sola lista

INTERFACCIA LISTA

```
public interface IList
{
// Inserimento oggetto a inizio e a fine lista
public void insert(Object o)
public void append(Object o)
// Rimozione primo e ultimo elemento
public Object removeFirst() throws ListEmptyException
public Object removeLast() throws ListEmptyException
// Restituire il primo elemento e ultimo lista
public Object first() throws ListEmptyException
public Object last() throws ListEmptyException
// Numero elementi e predicato di lista vuota
public int size()          public boolean isEmpty()
```


LISTA SEMPLICE (link in avanti)

```
public class SimpleList implements IList
{private int size; private ListNode first;
// Inserimento oggetto a inizio e a fine lista
public void insert(Object o);
public void append(Object o);
// Rimozione primo e ultimo elemento
public Object removeFirst() throws ListEmptyException;
public Object removeLast() throws ListEmptyException;
// Restituire il primo elemento e ultimo lista
public Object first() throws ListEmptyException;
public Object last() throws ListEmptyException;
// Numero elementi e predicato di lista vuota
public int size();          public boolean isEmpty();
```

Inserimento in prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento oggetto all'inizio della lista
public void insert(Object o)
{ ListNode node = new ListNode(o);
  node.setNext(first);
  first = node;
  size++; }
```


Inserimento ultima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento alla fine della lista
public void append(Object o)
{ ListNode newNode = new ListNode(o);
  ListNode node = first;
  while (node.getNext()!= null)
     node= node.getNext();
  node.setNext(newNode);
  size++;
}
```

Estrazione dalla prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Rimozione primo elemento
public Object removeFirst() throws
 ListEmptyException
{ if(isEmpty()) throw new
 ListEmptyException("lista vuota!");
  ListNode node = first;
  first = node.getNext(); size--;
  return node.getItem();
}
```


Estrazione dalla ultima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;
public Object removeLast() throws ListEmptyException
{if (isEmpty())
     throw new ListEmptyException("lista vuota!");
 if (first.getNext() == null)
 {Object o = first. getItem(); first = null; return o;}
 else
 {ListNode node=first.getNext(); ListNode pnode=first;
 while (node.getNext() != null)
       {pnode = node; node= node.getNext();}
 Object o=node.getItem (); pnode.setNext(null); size--
 ;
 return node.getItem();
 }
}
```

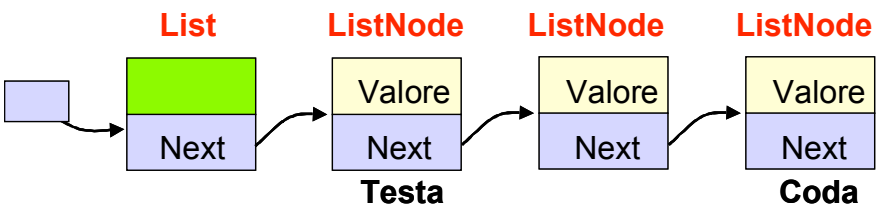
Accessori

```
public Object first() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException(" vuota!");
 return first.getItem();}

public Object last() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException("vuota!");
 ListNode node = first;
 while (node.getNext()!= null) node= node.getNext();
 return node.getItem();
}

public int size(){ return size;}
public boolean isEmpty(){ return first==null;}
```


IL PROGETTO REVISIONATO

Ora è semplice derivare la classe `SortedList`

- una `SortedList` specializza la classe base `List` nel senso che *il suo costruttore inserisce l'oggetto nella posizione opportuna in base all'ordinamento* anziché sempre in test



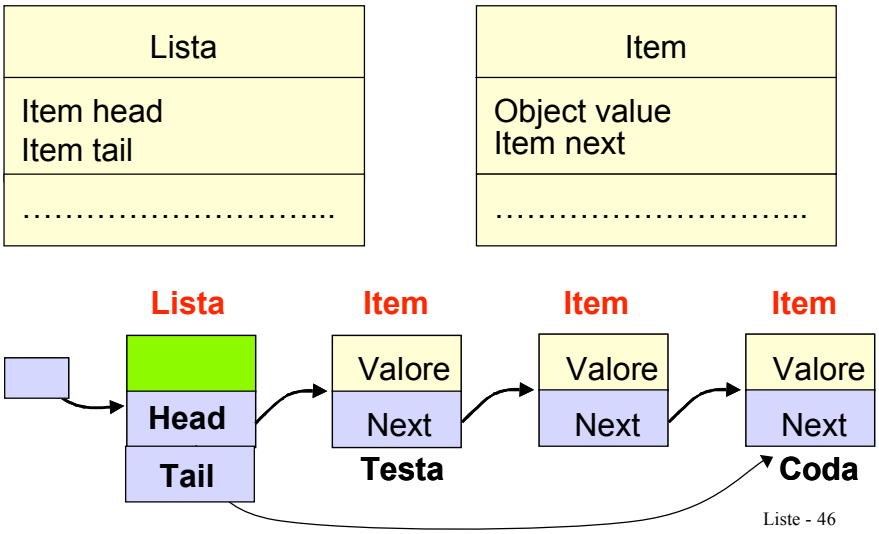
LISTE COME CONTENITORI

- **Approccio imperativo: una lista è una sequenza di contenitori di elementi, di lunghezza non stabilita a priori**

Conseguenze

- **Una lista è un contenitore**
- **Si può *cambiare* una lista esistente**
- **È un approccio efficiente (non si ricostruisce inutilmente ciò che già esiste)**
- **Ma può essere pericoloso in caso di *structure sharing* e *aliasing* (che quindi vanno evitati!)**

Alternativa di Progetto

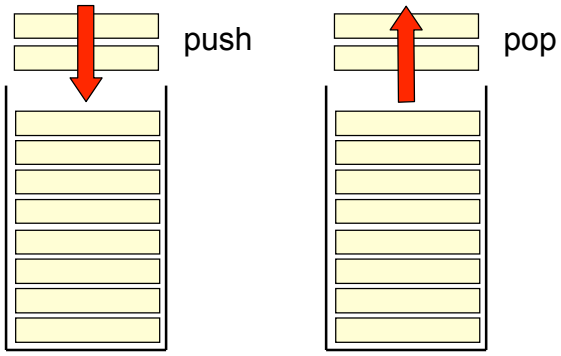


Stack

Stack

- Una lista è una collezione di elementi organizzati logicamente in sequenza.
- La dimensione di una lista non è fissata a priori
- Gli elementi della lista sono aggiunti solo quando necessario
- Accesso **Last In First Out**

Stack



L'ultimo elemento inserito è il primo che esce dallo stack

Stack

Si richiede di sviluppare un componente software **Stack** che implementa i seguenti metodi.

- **push**: inserisce un elemento nello stack
- **pop**: preleva l'ultimo elemento inserito nello stack. Solleva una eccezione nel caso in cui lo stack sia vuoto.
- **top**: restituisce il valore dell'ultimo elemento inserito nello stack ma non lo elimina. Solleva una eccezione nel caso in cui lo stack sia vuoto.

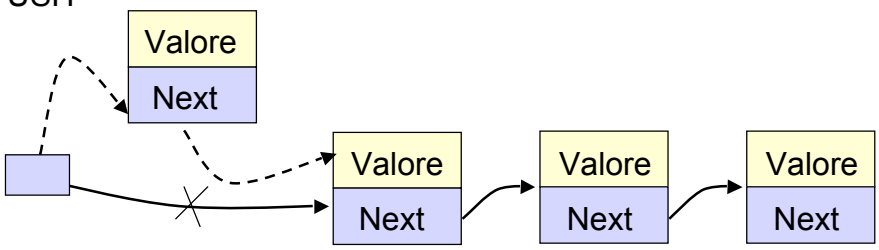
Implementazione

Stack
Item stk
push(Object value) pop() top()

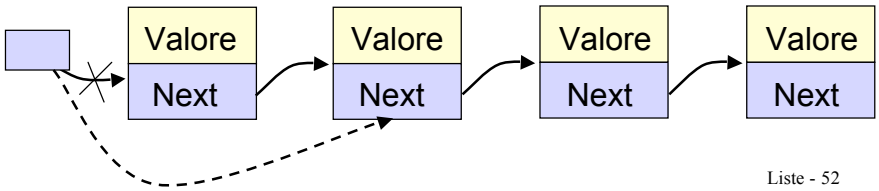
Item
Object value Item next
……..

Implementazione

PUSH



POP



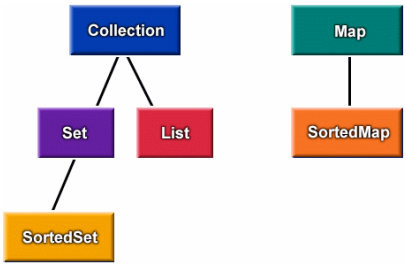
Strutture Dati in Java

STRUTTURE DATI IN JAVA

- **Molto spesso, una computazione si basa su una o più strutture dati, di vario tipo:**
  - **insiemi, multi-insiemi**
  - **code, stack, tabelle**
  - **liste, alberi**
- **Data la loro importanza, Java ne offre un'ampia scelta nella *Java Collection Framework (JCF)***
  - **interfacce che definiscono *insiemi di funzionalità***
  - **classi che ne forniscono *varie implementazioni***
  - **da Java 1.5 in poi: supporto ai TIPI GENERICI**

JAVA COLLECTION FRAMEWORK (package `java.util`)

Interfacce fondamentali



- `Collection`: nessuna ipotesi su elementi duplicati o relazioni d'ordine
- `List`: introduce l'idea di *sequenza*
- `Set`: introduce l'idea di *insieme* di elementi (come tale, senza duplicati)
- `SortedSet`: l'insieme *ordinato*
- `Map`: introduce l'idea di *mappa*, ossia di funzione che associa *chiavi* (identificatori univoci) a *valori*
- `SortedMap`: la mappa *ordinata*

Criteri-guida per la definizione delle interfacce:

- **Minimalità – prevedere solo metodi *davvero basilari…***
- **Efficienza – …o che *migliorino nettamente le prestazioni***

JCF: QUADRO GENERALE

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Implementazioni fondamentali

- **per le liste: `LinkedList`, `Vector`**
- **per gli alberi: `TreeSet`, `TreeMap`**

Legacy implementations

Altri concetti di utilità:

- **Concetto (interfaccia) di *iteratore***
- **Classe factory `Collections`**

Implementano rispettivamente `SortedSet` e `SorteMap`

ITERATORI

Un *iteratore* è una entità capace di *garantire l'attraversamento di una collezione* con una semantica chiara e ben definita *(anche se la collezione venisse modificata)*

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // opzionale
}
```

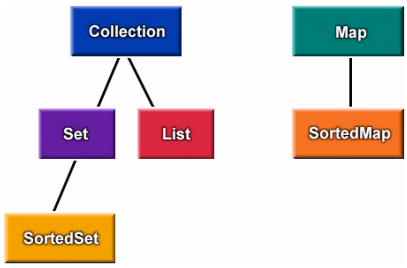
Di fatto, un iteratore offre un metodo **`next`** per esplorare uno ad uno gli elementi della collezione: il metodo **`hasNext`** permette di finire quando non ci sono più elementi.

Per ottenere un iteratore su una qualsiasi collezione, basta chiamare l'apposito metodo `iterator()`

JCF: L'ORDINAMENTO

L'*ordinamento* è presente in *`SortedSet`* e *`SortedMap`*

- **`SortedSet`** è usato tipicamente per elenchi di parole, di iscritti, etc.
- **`SortedMap`** è usata tipicamente per elenchi telefonici, agende e più in generale quando vi sono chiavi univoche associate a valori ordinati.



ESEMPIO: `Set`

Questo permette di scegliere un'altra implementazione senza dover cambiare niente, tranne che nella creazione.

Ad esempio, se volessimo un **elenco ordinato**:

```
...
   Set s = new TreeSet();
   for (int i=0; i<args.length; i
    if (!s.add(args[i]))
...
```

Una classe Factory nasconderebbe questo dettaglio implementativo!

Output:

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
4 parole distinte: [Io, esisto, parlo, sono]
```


ESEMPIO: `Set`

Per elencare tutti gli elementi di una collezione, il modo più rapido e sicuro è *farsi dare un iteratore:*

```
...
 for (Iterator i = s.iterator(); i.hasNext(); ) {
       System.out.print(i.next() + " ");
 }
```

Come già detto, per ottenere un iteratore su una qualsiasi collezione, basta **chiamare l'apposito metodo `iterator`**, usando `hasNext` per controllare il ciclo e `next` per avanzare

Output:

```
>java FindDupsIt Io sono Io esisto Io parlo
Io esisto parlo sono
```

DA `Set` A `List`

`List` introduce un concetto di *sequenza*: definisce un tipo di contenitore che può contenere duplicati.

```
public interface List extends Collection {
  Object get(int index);
  Object set(int index, Object element); // Optional
  void   add(int index, Object element); // Optional
  Object remove(int index);              // Optional
  abstract boolean addAll(int index, Collection c);
  //Optional
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator listIterator();
  ListIterator listIterator(int index);
  List subList(int from, int to);
}
```

La lista è una sequenza → c'è una idea di posizione

Nuovo tipo di iteratore per liste: può anche iniziare da un indice specificato

IMPLEMENTAZIONI di `List`

- ***Fino a JDK 1.4, la forma di lista più usata era `Vector`***
- **Sebbene `Vector` rimanga molto usata, Java 5 promuove l'uso di `List` e delle nuove implementazioni fornite, come `ArrayList` e `LinkedList`**
  - l'interfaccia List offre metodi con nomi più semplici e brevi, e con parametri in un ordine più naturale

ESEMPIO

Se in un array scriviamo

```
a[i] = a[j].mul(a[k]);
```

con un `Vector` scriveremmo:

```
v.setElementAt(v.elementAt(j).mul(v.elementAt(k)),i);
```

mentre con `List` scriviamo, più comodamente:

```
v.set(i, v.get(j).mul(v.get(k)));
```

ESEMPIO: `List`

Questa esercizio scambia due elementi in una lista, *qualunque sia il tipo di tali elementi*

```
static void swap(List a, int i, int j) {
  Object tmp = a.get(i);
  a.set(i, a.get(j)); a.set(j, tmp);
}
```

```
public class EsList {
 public static void main(String args[]){
   List l = new ArrayList();
   for (int i=0; i<args.length; i++) l.add(args[i]);
   System.out.println(l);
   swap(l, 2, 3);
   System.out.println(l);
 }
```

```
java EsList cane gatto pappagallo
  canarino cane canarino pescerosso
[cane, gatto, pappagallo, canarino,
  cane, canarino, pescerosso]
[cane, gatto, canarino, pappagallo,
  cane, canarino, pescerosso]
```

Notare la presenza di elementi duplicati

Da `Iterator` A `ListIterator`

Poiché `List` introduce un concetto di *sequenza*, nasce anche un nuovo tipo di iteratore che la sfrutta.

```
public interface ListIterator extends Iterator {
   boolean hasNext();
   Object next();
   boolean hasPrevious();
   Object previous();
   int nextIndex();
   int previousIndex();
   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

La lista è una sequenza navigabile anche a ritroso

L'iteratore di lista ha un concetto di indice prossimo o precedente

```
ListIterator listIterator();
ListIterator listIterator(int index);
```

Si può farsi dare un iteratore che inizi da un indice specificato

ESEMPIO: `ListIterator`

Schema tipico di iterazione a ritroso:

```
for( ListIterator i = l.listIterator(l.size()) ;
     i.hasPrevious() ; ) {
  MyClass f = (MyClass) i.previous();
  ...
}
```

Indispensabile iniziare dalla fine

```
public class EsList2 {
 public static void main(String args[]){
   List l = new ArrayList();
   for (int i=0; i<args.length; i++) l.add(args[i]);
   for( ListIterator i = l.listIterator(l.size()) ;
        i.hasPrevious() ; )
    System.out.print(i.previous()+" ");
 }
}
```

```
java EsList2 cane gatto
  cane canarino
canarino cane gatto cane
```

LA CLASSE FACTORY `Collections`

`Collections` è una classe factory, che comprende

- **metodi statici per lavorare con collezioni**
- **funzioni che implementano algoritmi utili**

- **`sort(List)`**: ordina una lista con l'algoritmo "merge sort"
- **`shuffle(List)`**: permuta casualmente gli elementi della lista
- **`reverse(List)`**: inverte l'ordine degli elementi della lista
- **`fill(List,Object)`**: rimpie la lista col valore dato
- **`copy(List dest,List src)`**: copia la lista sorgente nell'altra
- **`binarySearch(List,Object)`**: cerca l'elemento dato nella lista ordinata fornita, usando un algoritmo di ricerca binaria.

L'INTERFACCIA `Map`

`Map` introduce l'idea di *mappa come funzione* che associa univocamente *chiavi* a *valori.*

```
public interface Map {
  Object put(Object key, Object value);
  Object get(Object key);
  Object remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();
  // Collection Views
  public Set keySet();
  public Collection values();
  public Set entrySet();
  ...
}
```

Inserisce una coppia (chiave, valore)

Recupera un oggetto data la chiave

VISTE A COLLEZIONE

È possibile ottenere svariate *VISTE* della Map, come:

- la collezione dei valori
- la collezione (set) delle chiavi
- la collezione (set) delle coppie

ESEMPIO: `Map`

Questa esercizio conta le occorrenze delle parole sulla linea di comando.

```
import java.util.*;
public class HashMapFreq {
 public static void main(String args[]) {
   Map m = new HashMap();
   for (int i=0; i<args.length; i++) {
    Integer freq = (Integer) m.get(args[i]);
    m.put(args[i], (freq==null ? new Integer(1) :
                    new Integer(freq.intValue() + 1)));
   }
   System.out.println(m.size()+" parole distinte:");
   System.out.println(m);
 }
}
```

```
>java HashMapFreq cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

I metodi richiedono come parametro un `Object`, ma `int` non lo sarebbe
→ lo si incapsula

UN ESEMPIO DI ELENCHI ORDINATI (1/2)

Una piccola classe **`Persona`** che implementa **`Comparable`**:

```
class Persona implements Comparable {
  private String nome, cognome;
  public Persona(String nome, String cognome) {
    this.nome = nome;   this.cognome = cognome;
  }
  public String nome() {return nome;}
  public String cognome() {return cognome;}
  public String toString() {return nome + " " + cognome;}
  public int compareTo(Object x) {
    Persona p = (Persona) x;
    int confrontoCognomi = cognome.compareTo(p.cognome);
    return (confrontoCognomi!=0 ? confrontoCognomi :
            nome.compareTo(p.nome));
  }
}
```

Confronto lessicografico fra stringhe

UN ESEMPIO DI ELENCHI ORDINATI (2/2)

Un esempio d'uso dei metodi di `Collections`:

```
class NameSort {
  public static void main(String args[]) {
    Persona elencoPersone[] = {
        new Persona("Eugenio", "Bennato"),
        new Persona("Roberto", "Benigni"),
        new Persona("Edoardo", "Bennato"),
        new Persona("Bruno", "Vespa")
    };
    List l = Arrays.asList(elencoPersone);
    Collections.sort(l);
    System.out.println(l);
  }
}
```

Produce una `List` a partire dall'array dato

Ordina tale `List` in senso ascendente

Se il cognome è uguale, valuta il nome

```
>java NameSort
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
```

L'INTERFACCIA `SortedSet`

`SortedSet` introduce l'idea di *insieme ordinato:* nascono quindi concetti di *primo, ultimo, sottoinsieme,* etc

```
public interface SortedSet extends Set {
  SortedSet subSet(Object fromElement, Object toElement);
  SortedSet headSet(Object toElement);
  SortedSet tailSet(Object fromElement);
  Object first();
  Object last();
  Comparator comparator();
}
```

L'eventuale entità (user-defined) che svolge i confronti secondo logiche definite dall'utente

Specializzazioni di funzionamento

- L'iteratore restituito da `iterator()` ora naviga *seguendo l'ordine*
- L'array restituito da `toArray()` contiene gli elementi *nell'ordine*

SortedMap è del tutto analoga

QUALI IMPLEMENTAZIONI USARE?

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Regole generali per Set e Map:

- **se è indispensabile l'ordinamento, `TreeMap` e `TreeSet`** (perché sono le uniche implementazioni di `SortedMap` e `SortedSet`)
- **altrimenti, preferire `HashMap` e `HashSet` che sono *molto più efficienti* (*tempo di esecuzione costante* anziché log(N))**

Regole generali per List:

- **normalmente conviene `ArrayList`, in quanto il tempo di accesso è costante (anziché lineare con la posizione) essendo realizzate su array**
- **preferire invece `LinkedList` se l'operazione più frequente è l'aggiunta in testa o l'eliminazione di elementi in mezzo**