

Insegnamento di Laboratorio di algoritmi e strutture dati

ADT lista, pila e coda

Roberto Posenato

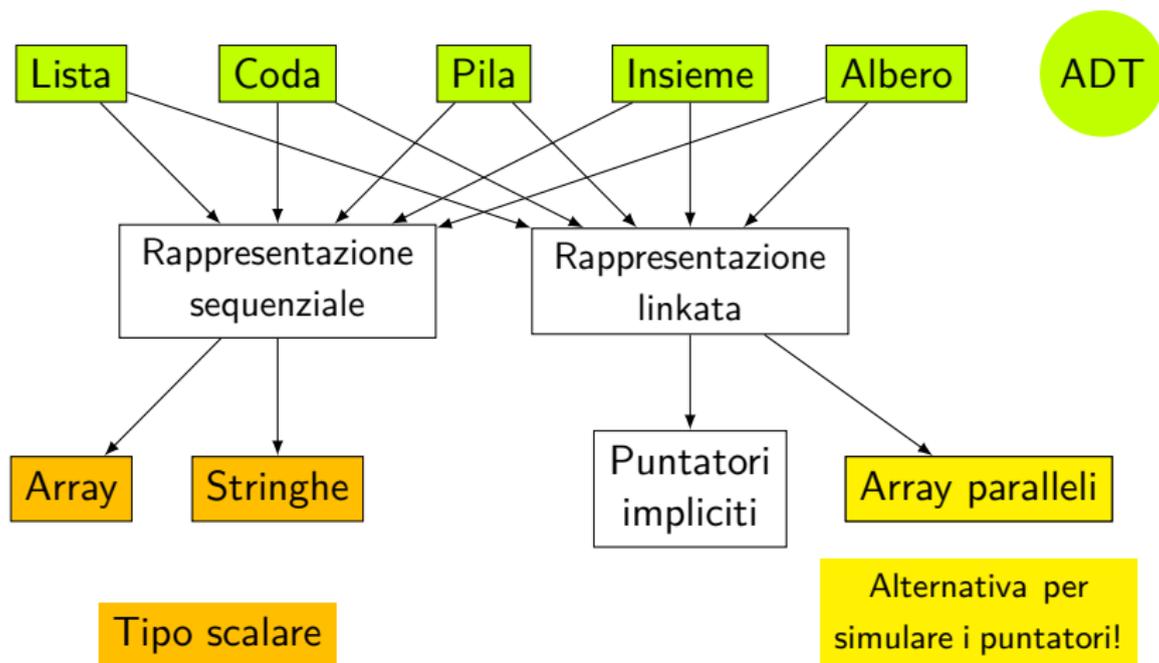
ver. 0.7, 29/01/2008

Sommario

- 1 Schema generale implementazioni
- 2 Lista
- 3 Coda e Pila

ADT Lista, Coda e Pila

Schema generale



Lista

Interfaccia

...continua

```
/**
 * Restituisce l'oggetto di indice k se esiste ,
 * lancia IndexOutOfBoundsException altrimenti.
 * @param k indice elemento ricercato
 * @return oggetto di indice k se  $(0 \leq k < size())$ 
 * @exception IndexOutOfBoundsException se  $k < 0$  o
 *            $k \geq size()$ 
 */
Object get(int k) //Cambio nome?! -> Java best practise!
    throws IndexOutOfBoundsException;
...
```

Esercizio 1

Completare l'interfaccia... osservando anche l'API di
`java.util.List`
Generare il Javadoc.

Lista

Implementazione con array

La struttura dati base è l'array.

Lista implementata con array

```
public class ListaArray implements Lista {
    /**
     * Dimensione iniziale della lista.
     */
    private static int dimensioneIniziale = 20;
    /**
     * Struttura base per la lista.
     */
    private Object[] lista;
    /**
     * Numero di oggetti presenti.
     */
    private int nOggetti;
    ...
}
```

Lista

Implementazione con array

...continua

```
/**
 * Costruttore di default
 */
public ListaArray() {
    lista = new Object[dimensioneIniziale];
    nOggetti = 0;
}
/**
 * @see Lista#isEmpty()
 */
public boolean isEmpty() {
    return nOggetti == 0;
}
...
```

Lista

Implementazione con array

Consideriamo ora un metodo significativo!

Metodo insert(int, Object)

```
/**
 * @see labASD.Lista#insert(int, Object)
 */
public void insert(int k, Object o)
    throws IndexOutOfBoundsException {

    if (k < 0 || k > nOggetti)
        throw new IndexOutOfBoundsException();
    if (nOggetti + 1 <= lista.length) {
        // c'è ancora spazio
        for (int i = nOggetti; i > k; i--)
            lista[i] = lista[i - 1];
        lista[k] = o;
        nOggetti++;
    }
    ...
}
```

Lista

Implementazione con array

continua metodo insert(int, Object)

```
    } else {  
        // è necessario aumentare dimensione array  
        Object[] listaNuova =  
            new Object [lista.length  
                + dimensioneIniziale];  
  
        for (int i = 0; i < k; i++)  
            // copia tutti gli elementi prima di k  
            listaNuova[i] = lista[i];  
  
        //Inserisco oggetto o  
        listaNuova[k] = o;  
        ...
```

Lista

Implementazione con array

```
fine metodo insert(int, Object)
```

```
    for (int i = k; i < nOggetti; i++)  
        // copia tutti gli altri elementi dopo k  
        listaNuova[i + 1] = lista[i];
```

```
    nOggetti++;  
    lista = listaNuova;
```

```
    }
```

```
}
```

Lista

Implementazione con array

metodo remove(int)

```
/**
 * @see labASD.Lista#remove(int)
 */
public Object remove(int k)
    throws IndexOutOfBoundsException {
    if (k < 0 || k >= nOggetti)
        throw new IndexOutOfBoundsException();
    Object eliminato = lista[k];
    nOggetti--; //se è ultimo elemento, questo è sufficiente
    if (k != nOggetti) { //è un elemento non alla fine
        for (; k < nOggetti; k++)
            lista[k] = lista[k + 1];
    }
    return eliminato;
}
```

Lista

Implementazione con puntatori impliciti

La struttura dati base è una classe che rappresenta l'elemento e il collegamento al successivo.

Classe Nodo

```
/** ... */
private static class Node {
    /** ... */
    Object key;

    /** Riferimento al successivo elemento */
    Node next;

    /** ... */
    Node(Object o) {
        key = o;
        next = null;
    }
}
```

Lista

Implementazione con puntatori impliciti

La classe usa la sottoclasse `Nodo` per creare la lista.

Classe `ListaLink`

```
public class ListaLink implements Lista {
    private static class Node { ... };
    /**
     * Testa della lista
     */
    private Node head;

    /** ... */
    private int nOggetti;

    /** ... */
    public ListaLink() {
        head = null;
        nOggetti = 0;
    }
}
```

Lista

Implementazione con puntatori impliciti

Consideriamo ora un metodo significativo!

Metodo insert(int, Object)

```
/**
 * @see Lista#insert(int, java.lang.Object)
 */
public void insert(int k, Object o)
    throws IndexOutOfBoundsException {
    if (k < 0 || k > nOggetti)
        throw new IndexOutOfBoundsException();

    Node nuovo = new Node(o);
    if (nOggetti == 0) {
        // è il primo elemento da inserire
        head = nuovo;
    } else {
```

Lista

Implementazione con puntatori impliciti

continua metodo insert(int, Object)

```
        if (k == 0) { // è da inserire in testa
            nuovo.next = head;
            head = nuovo;
        } else { // è da inserire in mezzo o fine
            Node indice = head;
            // posizionati all'elemento precedente
            for (int i = 0; i < k-1; i++)
                indice = indice.next;
            // al nuovo attacca il successivo del precedente
            nuovo.next = indice.next;
            // al precedente attacca il nuovo
            indice.next = nuovo;
        }
    }
    nOggetti++;
}
```

Lista

Implementazione con puntatori impliciti

Altro metodo...

Metodo remove(int)

```
public Object remove(int k)
    throws IndexOutOfBoundsException {
    if (k < 0 || k >= nOggetti)
        throw new IndexOutOfBoundsException();
    Object eliminato;
    if (nOggetti == 1) {
        eliminato = head.key;
        head = null;
    } else {
        if (k == 0) {
            eliminato = head.key;
            head = head.next;
        } else {
            //è un elemento generico
            Node indice = head;
```

Lista

Implementazione con puntatori impliciti

continua metodo remove(int)

```
//Determina il riferimento all'elemento  
//che precede quello da cancellare  
for (int i = 0; i < k - 1; i++)  
    indice = indice.next;  
  
//Cancella elemento assegnando al successivo  
//del precedente il successivo dell'elemento  
//da cancellare  
eliminato = indice.next.key;  
indice.next = indice.next.next;  
    }  
}  
nOggetti--;  
return eliminato;  
}
```

Lista

Confronto implementazioni

Come si possono “valutare” in modo **semplice** le due implementazioni?

Si deve costruire una classe esterna che manipoli oggetti di tipo Lista, usando le due classi.

Lista

Confronto implementazioni

La classe Test “confronta” le due implementazioni...

Classe Test

```
public class Test {
    /** ... */
    public static void main(String[] args) {
        /**
         * Numero volte che si ripete un test
         */
        final int nTest = 100
        /**
         * Numero elementi da inserire
         */
            , nInserimenti = 10000;
        Lista listaArray = null
            , listaB = null;
        long inizio, fine;
```

Lista

Confronto implementazioni

continua Classe Test

```
// Test sulla listaArray
inizio = System.currentTimeMillis();
//Eseguo lo stesso set di operazioni nTest volte
for (int j=0; j<nTest; j++) {
    listaArray = new ListaArray();

    for (int i=0; i<nInserimenti; i++)
        listaArray.insert(i, new Integer(i));
}
fine = System.currentTimeMillis();
System.out.println(
    "Tempo richiesto (ms) per la lista array:"
    + ((double) (fine - inizio)) / nTest
);
...
```


Lista

Esercizi

Esercizio 2

- 1 Completare la classe `ListaArray`.
- 2 Completare la classe `ListaLink`.
- 3 Completare la classe `Test` scrivendo il codice per il rilevamento tempi per le altre operazioni.
- 4 Eseguire dei rilevamenti tempi variando la variabile `nInserimenti` di `Test` e `dimensioneIniziale` di `ListaArray`.
- 5 Confrontare i costi teorici e i costi rilevati. Hanno influenza i valori delle variabili precedenti? Se sì, cosa si può dedurre?

Lista

Stato dell'arte

Nel linguaggio Java esistono già delle implementazioni dell'ADT Lista.

- `java.util.List` è l'interfaccia;
- `java.util.ArrayList` è l'implementazione con array;
- `java.util.Vector` è un'altra implementazione (migliore?);
- `java.util.LinkedList` è un'implementazione ottimizzata per inserimento e cancellazione in testa o in coda;
- `java.util.AbstractList` è un'implementazione astratta. **perché serve?**
- `java.util.AbstractSequentialList` è un'implementazione astratta. **perché serve?**

Coda e Pila

È una riscrittura dell'interfaccia Lista.

Definizione 3 (Interfaccia Pila)

```
/**
 * Descrive i metodi fondamentali per il tipo Pila.
 */
public interface Pila {
    /** @see Lista#isEmpty() */
    boolean isEmpty();
    /**
     * Toglie e restituisce la cima della pila
     * se esiste, lancia un errore altrimenti.
     * @return oggetto in cima alla pila
     * @exception IllegalStateException
     * se la pila è vuota
     */
    Object pop() throws IllegalStateException;
    ...
}
```

Coda e Pila

Interfaccia Pila

...continua

```
/**
 * Inserisce l'oggetto non nullo in cima alla pila.
 * @param o oggetto da inserire
 * @exception NullPointerException se
 * l'oggetto o è nullo
 */
void push(Object o)
    throws NullPointerException;

/**
 * Ritorna un riferimento alla cima della pila
 * senza alterarla.
 * @return Oggetto in cima alla pila, se esiste
 * @exception IllegalStateException se la pila è vuota
 */
Object top() throws IllegalStateException;
```

Coda e Pila

Implementazione delle pile

- Si possono utilizzare le implementazioni date per le liste, opportunamente aggiustate.
- Per esempio, se si usa `ListaArray`, le operazioni potrebbero essere riscritte come:
 - 1 `pop() = remove(size()-1)`
 - 2 `push(o) = insert(size(), o)`
- E se si usa `ListaLink`? Si possono riusare le mappature del punto precedente?

Coda e Pila

Implementazione delle pile

Esercizio 3

- Scrivere la classe `PilaArraySemplice.java` che implementa l'interfaccia `Pila` estendendo la classe `ListaArray.java`.
- Scrivere la classe `PilaLinkSemplice.java` che implementa l'interfaccia `Pila` estendendo la classe `ListaLink.java`.
- Studiare il paragrafo 12.1 del libro “Dai fondamenti agli oggetti” di Pighizzini & Ferrari.

Coda e Pila

Interfaccia Coda

È una riscrittura dell'interfaccia Lista.

Definizione 4 (Interfaccia Coda)

```
/**
 * Descrive i metodi fondamentali per il tipo Coda.
 */
public interface Coda {
    ...
    /**
     * Toglie il primo elemento della coda se esiste ,
     * lancia un errore altrimenti.
     * @return oggetto nella prima posizione della coda
     * @exception IllegalStateException se la coda è vuota
     */
    Object dequeue() throws
    IllegalStateException;
    ...
}
```

Coda e Pila

Interfaccia Coda

continua...

```
/**
 * Inserisce l'oggetto non nullo in fondo alla coda.
 * @param o oggetto da inserire
 * @exception NullPointerException
 * se l'oggetto o è nullo
 */
void enqueue(Object o)
    throws NullPointerException;
...
```


Coda e Pila

Implementazione della Coda

- L'implementazione della Coda con array/puntatori impliciti può essere costruita in modo più efficiente.
- È necessario scrivere i metodi `dequeue()` e `enqueue()`

Coda e Pila

Implementazione più efficiente della Coda

Un'implementazione più efficiente mantiene riferimento di dove si trova la **testa** e la **fine** della Coda.

Implementazione più efficiente per la Coda con **array**

```
public class CodaArray implements Coda {
    private static int dimIni = 20;

    /**Supporto per la memorizzazione degli elementi.**/
    private Object[] coda;

    /** Indice inizio coda.**/
    private int inizio;

    /** Indice fine coda. */
    private int fine;
    ...
}
```

Coda e Pila

Implementazione più efficiente della Coda

continua

```
/**
 * @see labASD.Coda#enqueue(Object)
 */
public void enqueue(Object o)
    throws NullPointerException {
    // L'array è gestito in modo circolare.
    // Questo facilita il riuso degli elementi dopo
    // inserimenti/cancellazioni.
    if (nOggetti++ < coda.length) {
        coda[fine] = o;
        fine = (fine + 1) % coda.length;
    } else { //è necessario aumentare la dimensione dell'array
        Object[] codaNuova =
            new Object[dimIni+coda.length];
        ...
    }
}
```

Coda e Pila

Implementazione più efficiente della Coda

continua

```
// Copio gli elementi senza riorganizzarli.  
// assert: fine == inizio  
for (int i = 0; i < fine; i++)  
    codaNuova[i] = coda[i];  
for (int i=inizio; i<coda.length;i++)  
    codaNuova[i + dimIni] = coda[i];  
inizio += dimIni;  
codaNuova[fine++] = 0;  
coda = codaNuova;  
}  
}
```

Coda e Pila

Implementazione più efficiente della Coda

Esercizio 4

- 1 Completare la classe precedente `CodaArray`.
- 2 Costruire una classe `Test` e confrontare le prestazioni di questa classe con la classe `CodaArraySemplice` che implementa `Coda` estendendo la classe `ListaArray`.
- 3 che differenze ci sono tra i tempi di `dequeue()` nelle due classi?
- 4 Studiare il paragrafo 12.2 del libro “Dai fondamentali agli oggetti” di Pighizzini & Ferrari.

Coda e Pila

Implementazione più efficiente della Coda

Un'implementazione più efficiente con puntatori impliciti deve mantenere riferimento di dove si trova la **testa** e la **fine** della Coda.

Implementazione più efficiente di tipo **linked** per la Coda

```
public class CodaLink implements Coda {  
  
    private static class Node{...}; //Come in Lista  
  
    /** Riferimento all'inizio della coda. */  
    private Node inizio;  
  
    /** Riferimento alla fine della coda. */  
    private Node fine;  
  
    /** Numero di elementi attualmente presenti. */  
    private int nOggetti;  
    ...  
}
```

Coda e Pila

Implementazione più efficiente della Coda

continua...

```
/**
 * @see labASD.Coda#enqueue(Object)
 */
public void enqueue(Object o)
    throws NullPointerException {
    Node nuovo = new Node(o);
    if (fine == null) { // coda è vuota
        inizio = fine = nuovo;
    } else {
        fine.next = nuovo;
        fine = nuovo;
    }
    nOggetti++;
}
...
```

Coda e Pila

Implementazione più efficiente della Coda

Esercizio 5

- 1 Completare la classe precedente `CodaLink`.
- 2 Costruire una classe `Test` e confrontare le prestazioni di questa classe con la classe `CodaLinkSemplice` che implementa `Coda` estendendo la classe `ListaLink`.
- 3 che differenze ci sono tra i tempi di `enqueue()` nelle due classi?

Coda e Pila

Confronto implementazioni più efficienti per la Coda

Numero operazioni teorico nel caso peggiore

Operazione	Implementazione sequenziale	Implementazione linked
size()	$O(1)$	$O(1)$
enqueue()	$O(n)$	$O(1)$
dequeue()	$O(1)$	$O(1)$
element()	$O(1)$	$O(1)$

Coda e Pila

Stato dell'arte

Nel linguaggio Java esistono già delle implementazioni dell'ADT Coda (Queue) e dell'ADT Pila (Stack).

- `java.util.Stack`, classe completa per l'ADT Pila.
- `java.util.Queue` è l'interfaccia per la Coda.
- Esistono diverse implementazioni dell'interfaccia Queue:
AbstractQueue, ArrayBlockingQueue,
ConcurrentLinkedQueue, DelayQueue,
LinkedBlockingQueue, LinkedList,
PriorityBlockingQueue, PriorityQueue,
SynchronousQueue