Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

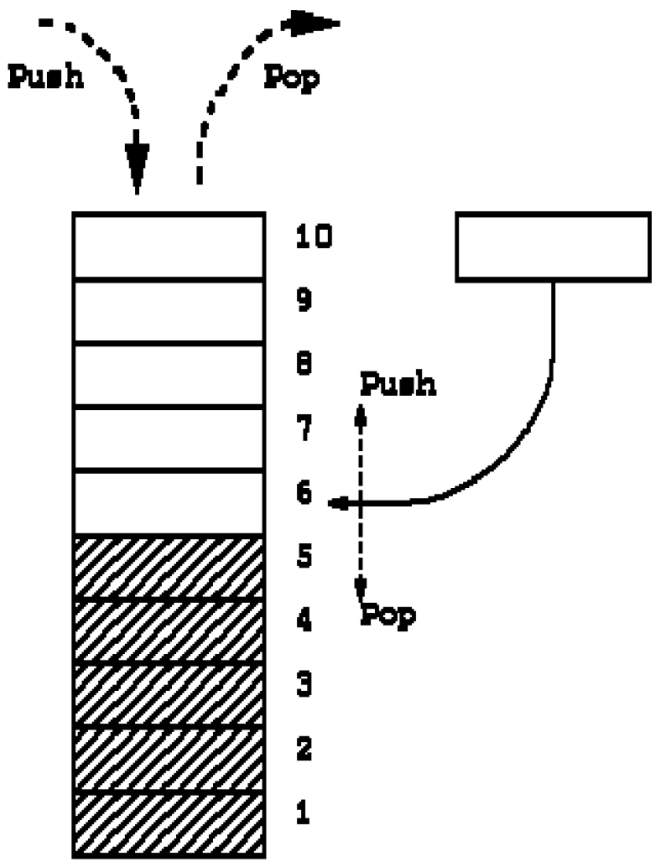
STACK and QUEUE

Tipo di dato Pila (Stack)

- Uno **stack** è una collezione di elementi dello stesso tipo che supporta le seguenti operazioni:

  `push, pop, peek o top, isEmpty, [ isFull ]`

- Disciplina di accesso **LIFO – last in first out**: l'accesso agli elementi avviene secondo l'ordine inverso di inserimento



Tipo di dato Stack

tipo `Stack`:
dati: una sequenza S di n elementi.
operazioni:

`isEmpty()` → *result*
  restituisce `true` se S è vuota, e `false` altrimenti

`push`(*elem e*)
  aggiunge *e* come ultimo elemento di S

`peek()` → *elem // altrimenti riferita come top()*
  restituisce l'ultimo elemento di S (senza eliminarlo da S)

`pop()` → *elem*
  elimina da S l'ultimo elemento e lo restituisce

Tipo di dato Pila

- Il termine stack viene usato in informatica in modo più specifico in diversi contesti:
  - lo stack è un elemento dell'architettura dei moderni processori, e fornisce il supporto fondamentale per l'**implementazione del concetto di subroutine** (vedi call stack, ricorsione)
  - le macchine virtuali di quasi tutti i linguaggi di programmazione ad alto livello usano uno **stack dei record di attivazione** per implementare il concetto di subroutine (generalmente, ma non necessariamente, basandosi sullo stack del processore)
  - la memoria degli automi a pila dell'informatica teorica è uno stack

Tipo di dato Pila

Esempi

- Verificare il bilanciamento delle parentesi in espressioni e programmi

```
abc{defg{ijk}{l{mn}}op}qr   (true)
abc{def}}{ghij{kl}m         (false)
abc{def}{ghij{kl}m          (false)
```

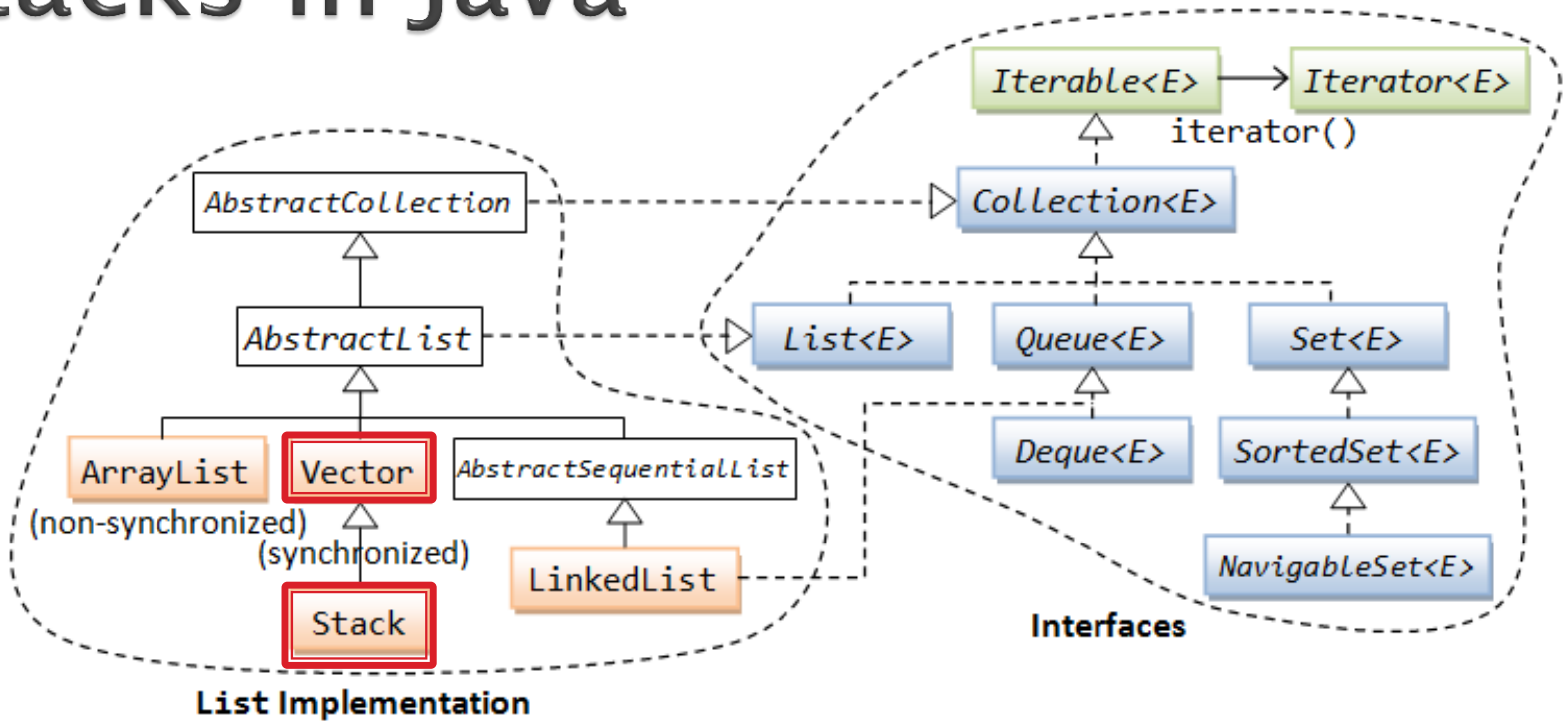
- Riconoscere stringhe palindrome

```
abcdcba
```

- Valutare espressioni postfisse

```
2 3 4 + *
```

Stacks in Java



- La classe `stack` nel package `java.util` dovrebbe essere evitata poiché è una sottoclasse di **`Vector`** e perciò consente l'esecuzione di operazioni non-stack (**`Rif. StackExample.java`**)

Vector<E> vs ArrayList<E> (cenno)

- Le classi generiche **Vector<E>** e **ArrayList<E>** sono sostanzialmente equivalenti, ma:
  - I metodi di **Vector<E>** sono **sincronizzati**, mentre quelli di **ArrayList<E>** non lo sono. Quindi se il programma è **concorrente** (cioè usa il **multi-threading** di Java) è opportuno usare **Vector<E>**, altrimenti converrebbe **ArrayList<E>** perché più efficiente.
  - **Vector<E>** fornisce, con opportuni metodi e costruttori, un controllo maggiore sulla **capacità**, cioè la dimensione dell'array soggiacente.
  - Per motivi storici, **Vector<E>** fornisce più metodi con nomi diversi per manipolare gli elementi di un vettore.

Vector<E> vs ArrayList<E>

- I costruttori di **`Vector<E>`** permettono di specificare la capacità iniziale del vettore (**`initialCapacity`**) e il valore da usare per aumentare la capacità (**`capacityIncrement`**) quando necessario.
  - Se (**`capacityIncrement`** == **0**), il nuovo array avrà capacità doppia rispetto all'attuale.
- I costruttori di **`ArrayList<E>`** permettono di specificare solo la capacità iniziale del vettore.

Vector<E> vs ArrayList<E>

- **Vector<E>**

```
/* crea un vettore vuoto, con i parametri specificati */
Vector (int initialCapacity, int capacityIncrement)
/* default: capacityIncrement==0 */
Vector (int initialCapacity)
/* default: initialCapacity==10 e capacityIncrement==0 */
Vector ()
```

- **ArrayList<E>**

```
/* crea un vettore con la capacità iniziale indicata */
ArrayList (int initialCapacity)
/* crea un vettore vuoto; */
ArrayList ()
```

L'interfaccia Stack

- La seguente interfaccia definisce le operazioni di interesse per uno stack

  Rif. Stack.java

```
public interface Stack<T> {
  void push(T item);
  T pop();
  T peek();
  int size();
  boolean isEmpty();
}
```

Implementazioni

- Implementazione semplice basata su LinkedList: lo stack “delega” banalmente alla lista!

  Rif. SimpleStack.java
- Implementazione basata su array (tecnica del raddoppiamento-dimezzamento)

  Rif. ArrayStack.java
- Implementazione basata su liste collegate semplici (migliorare con gestione esplicita della size!)

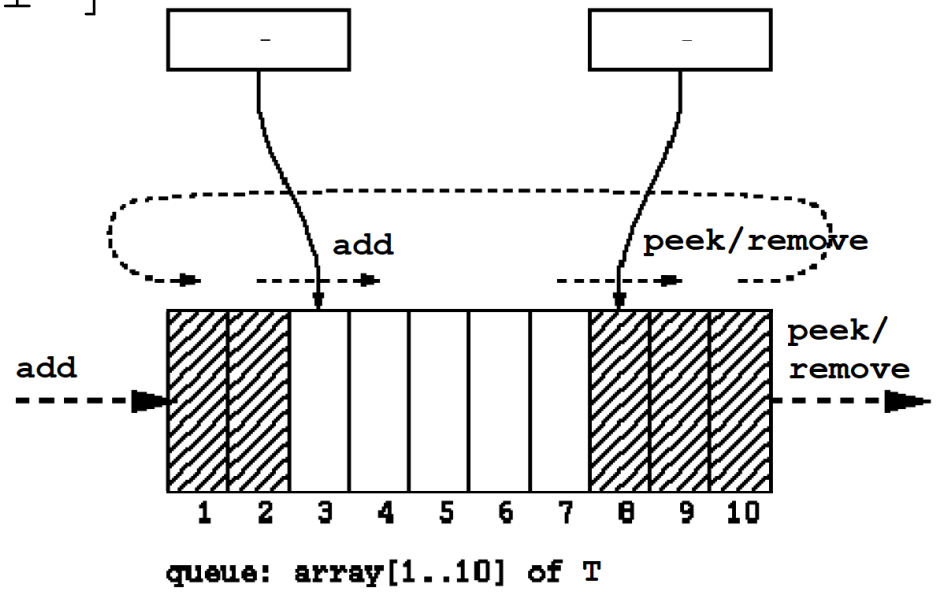
  Rif. LinkedStack.java

Tipo di dato Coda

- Una **coda** è una collezione di elementi dello stesso tipo che supporta le seguenti operazioni tipiche:
  - `Enqueue (add), dequeue (remove), peek, isEmpty, [ isFull ]`

- Disciplina di accesso

FIFO – first in first out: l'accesso agli elementi avviene secondo l'ordine di inserimento



Tipo di dato Coda

- Numerose applicazioni delle code in computer science/engineering:
  - **accesso a risorse condivise in mutua esclusione** (coda di accesso alla CPU, spooling di stampa, …)
  - **code di pacchetti** nei dispositivi di rete per **l'instradamento (router)**

Tipo di dato Queue

tipo `Queue`:
dati: una sequenza S di n elementi.
operazioni:

`isEmpty()` → *result*
  restituisce `true` se S è vuota, e `false` altrimenti

`add`(*elem e*)
  aggiunge *e* come ultimo elemento di S

`peek()` → *elem // altrimenti riferita come first()*
  restituisce il primo elemento di S (senza eliminarlo da S)

`remove()` → *elem*
  elimina da S il primo elemento e lo restituisce

L'interfaccia Coda

- La seguente interfaccia definisce le operazioni di interesse di una coda

  Rif. Queue.java

```
public interface Queue<T> {
  public boolean isEmpty();
  public boolean add (T e);
  public T peek();
  public T remove();
  public int size();
}
```

Implementazioni

- Implementazione semplice basata su LinkedList: la coda “delega” banalmente alla lista!

  Rif. SimpleQueue.java
- Implementazione basata su array a dimensione fissa (buffer circolare)

  Rif. BoundedQueue.java
- Implementazione basata su liste collegate semplici

  Rif. LinkedQueue.java

L'interfaccia Queue<E> (JCF)

```
public interface Queue<E> extends Collection<E> {
  boolean add(E e)
  E element();
  boolean offer(E e);
  E peek();
  E poll();
  E remove();
}
```

- Le classi `LinkedList<E>` e `PriorityQueue<E>` implementano l'interfaccia `Queue<E>`

L'interfaccia Queue<E> (JCF)

Ogni metodo esiste in due forme:

1. Una solleva un'eccezione se l'operazione fallisce
2. L'altra restituisce un valore speciale se l'operazione fallisce

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()