UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

# Pile e code nel JCF, alberi binari (esercitazione)

PROGETTAZIONE, ALGORITMI E COMPUTABILITÀ
(38090-MOD1)

**Corso di laurea Magistrale in Ingegneria Informatica**

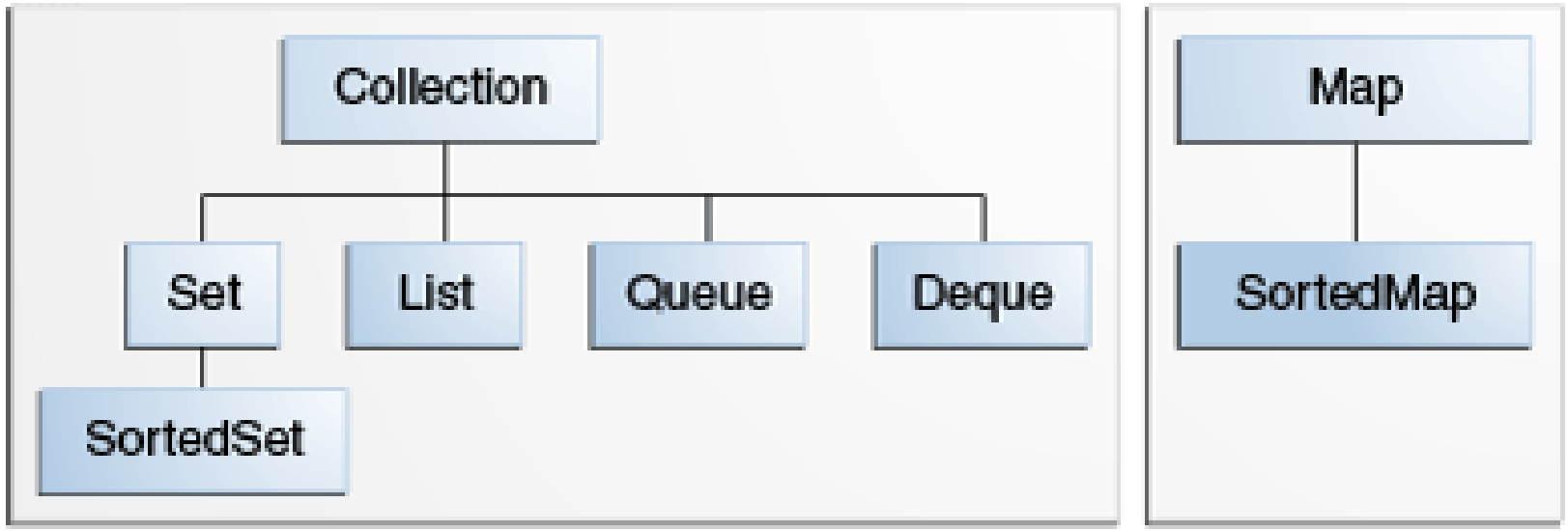
RELATORE
Prof.ssa Patrizia Scandurra

SEDE
DIGIP

# Il Java Collection Framework (JCF)

- **Interfacce e classi del package java.util**
- Forniscono:
  - **ADT** e **strutture dati** per manipolare **collezioni di oggetti**
  - **algoritmi di base** (ad es. come ordinamento e ricerca)
- http://docs.oracle.com/javase/tutorial/collections/index.html

# JCF – le interfacce



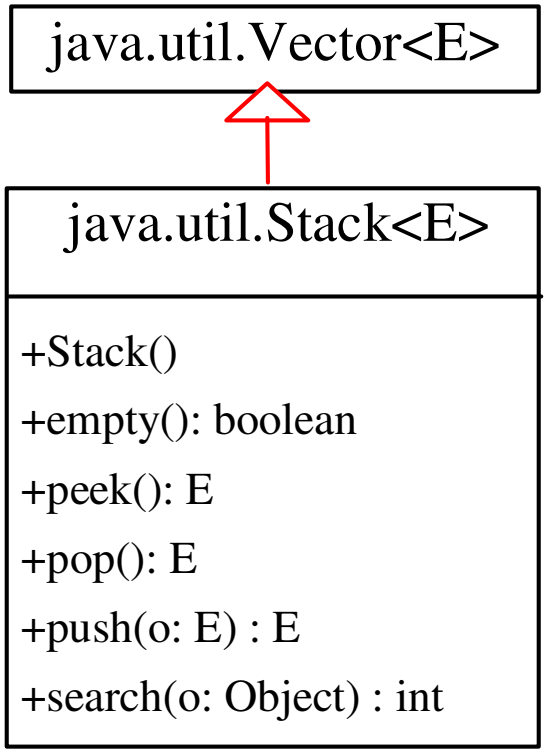

- Formano una gerarchia
- Tutte le interfacce sono generiche. Ad esempio la definizione dell'interfaccia Collection:

  public interface Collection<E>...

# ADT Pila e Coda in JCF

- La classe predefinita **Stack***:*
  - public class Stack<E> extends Vector<E>
- Interfaccia **Queue**:
  - classi JCF che implementano Queque: AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
- Interfaccia **Deque**: usabile sia come pila che come coda
  - public interface Deque<E> extends Queue<E>
  - classi JCF che implementano Deque: ArrayDeque, LinkedBlockingDeque, LinkedList

# La classe **`Stack`**




Creates an empty stack.

Returns true if this stack is empty.

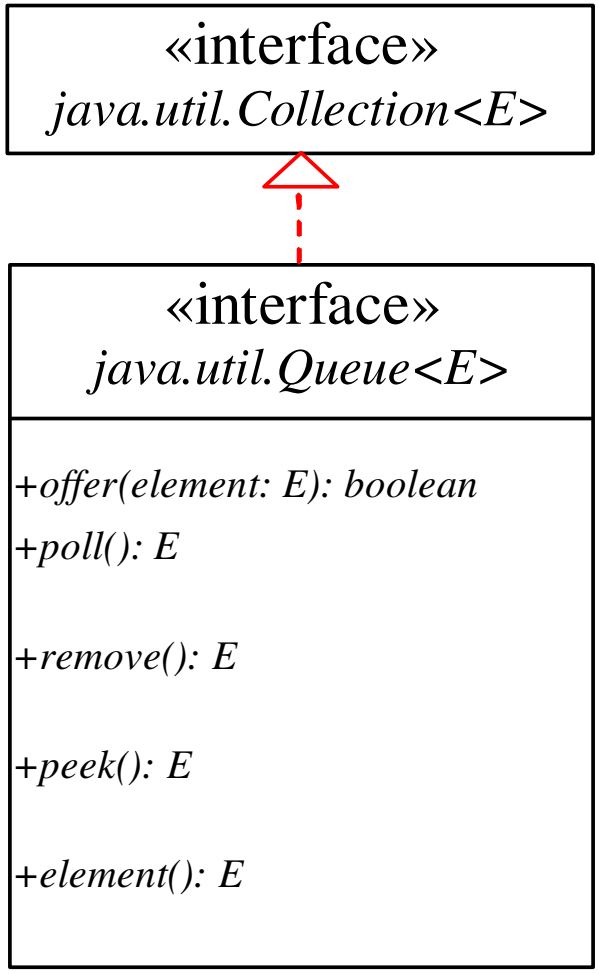
Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

# Interfaccia Queue




rimuovono elemento di testa della coda : poll e remove

restituiscono ma non rimuovono : peek element

Inserts an element to the queue.

Retrieves and removes the head of this queue, or null if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.

# Interfaccia Dequeue

- https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

| | First Element (Head) | | Last Element (Tail) | |
|---|---|---|---|---|
| | *Throws exception* | *Special value* | *Throws exception* | *Special value* |
| **Insert** | `addFirst(e)` | `offerFirst(e)` | `addLast(e)` | `offerLast(e)` |
| **Remove** | `removeFirst()` | `pollFirst()` | `removeLast()` | `pollLast()` |
| **Examine** | `getFirst()` | `peekFirst()` | `getLast()` | `peekLast()` |

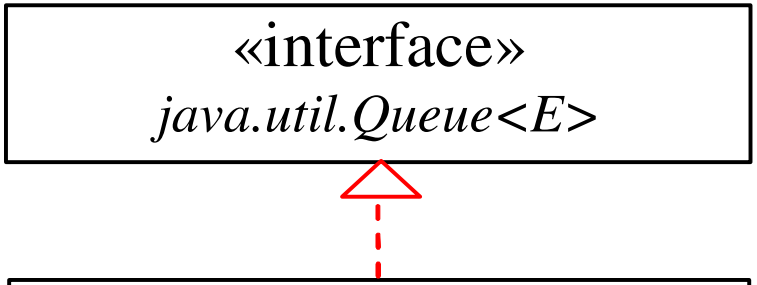
# ADT PILA: Stack vs Dequeue

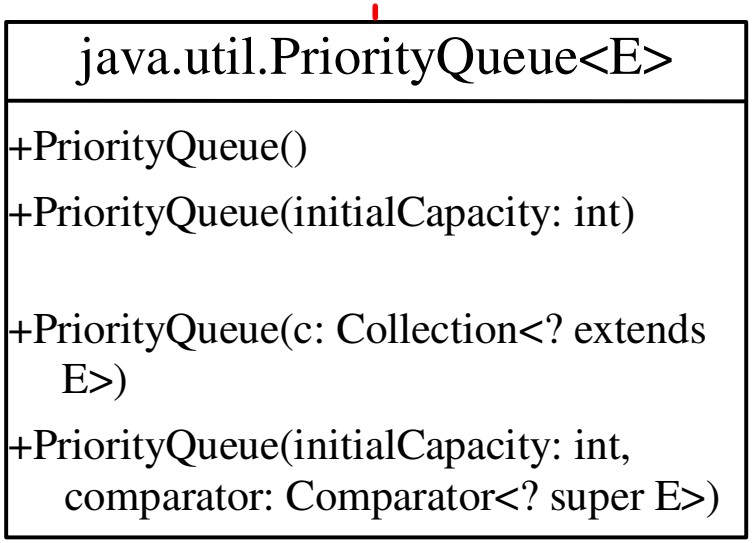
| Stack Method | Equivalent Deque Method |
|---|---|
| push(e) | addFirst(e) |
| pop() | removeFirst() |
| peek() | peekFirst() |

# ADT Coda: Queue vs Dequeue

| Queue Method | Equivalent Deque Method |
|---|---|
| add(e) | addLast(e) |
| offer(e) | offerLast(e) |
| remove() | removeFirst() |
| poll() | pollFirst() |
| element() | getFirst() |
| peek() | peekFirst() |

# La classe PriorityQueue




Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

Creates a priority queue with the specified initial capacity and the comparator.

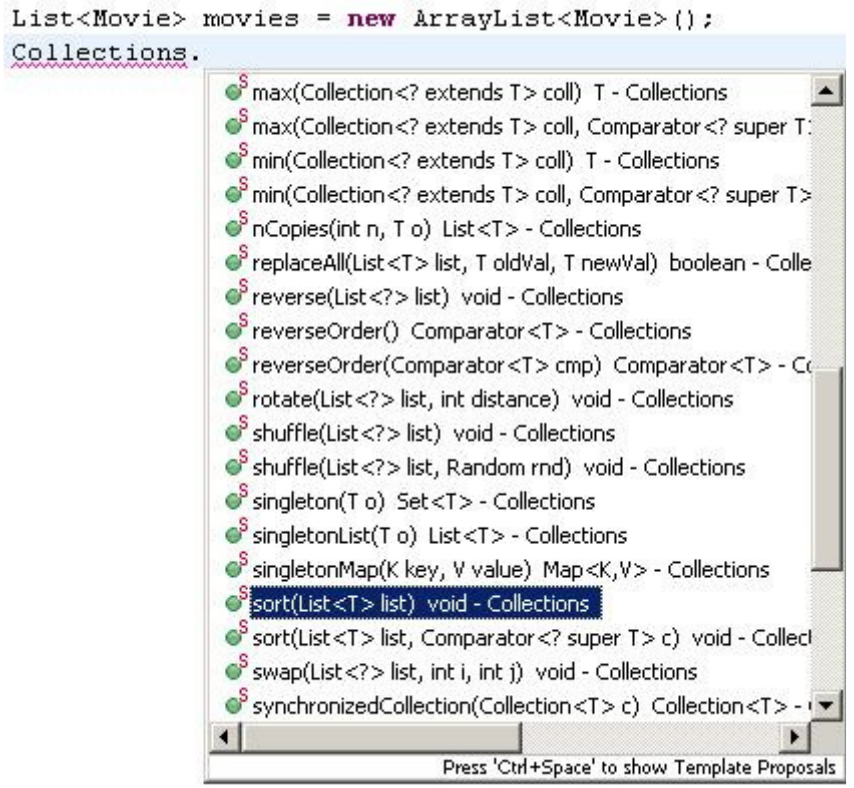
- L'elemento **a priorità max** viene **rimosso per primo**
- La priorità è stabilita esternamente mediante un oggetto Comparator

# JCF - Algoritmi

## `java.util.`**`Collections`**

Algoritmi per diversi tipi di collezioni

- Sorting (e.g. sort)
- Shuffling (e.g. shuffle)
- Routine Data Manipulation
  (e.g. reverse, addAll)
- Searching (e.g. binarySearch)
- Composition (e.g. frequency)
- Finding Extreme Values
  (e.g. max)



# Synchronized Wrapper

- La classe **Collections** contiene dei metodi statici per costruire un **wrapper di sincronizzazione** per una collezione del JCF e renderla *thread-safe*

```
public static  Collection synchronizedCollection(Collection c);
public static  Set synchronizedSet(Set s);
public static  List synchronizedList(List list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static  SortedSet synchronizedSortedSet(SortedSet s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

- In alternativa, usare le classi collezioni del package java.util.concurrent
  - BlockingQueue, BlockingDequeue, ConcurrentMap, ecc..

# Synchronized Wrapper

- Esempio

```
Collection c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    Iterator i = c.iterator(); // Must be in a synchronized block!
    while (i.hasNext())
        foo(i.next());
}
```

# Esercizio

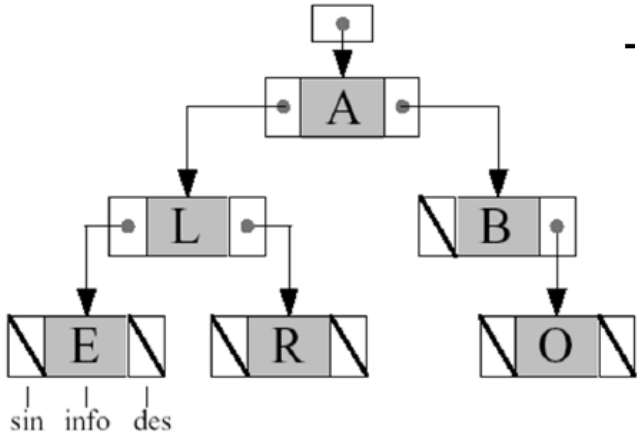
input premendo una pila, e poi faccio travaso

- Invertire l'ordine degli elementi di una pila S usando una coda Q

(Sfruttare gli ADT Stack, Queue, o Dequeue del JCF)

# Alberi binari -- *Descrizione ricorsiva*

- Molti algoritmi su alberi binari possono essere descritti in modo naturale sfruttando la definizione ricorsiva (e quindi la ***strategia divide et impera***):
  - ***Caso base -*** albero vuoto o formato dalla sola radice;
  - ***Clausola ricorsiva -*** *due chiamate ricorsive, una per* ogni sotto-albero
- Esempio: Vedi i metodi ***numNodi()*** e ***numNodi(NodoBinario r)*** nell'implementazione in Java del tipo albero binario
- Nota che la rappresentazione collegata a cui stiamo facendo riferimento è con puntatori diretti ai figli

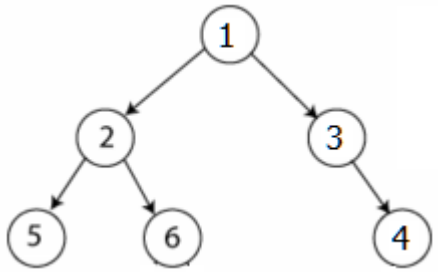


# Esercizi

Dato il codice sorgente su Moodle nella cartella *Codice sorgente/src_alberi,* arricchire l'interfaccia **AlberoBinario** e la classe **AlberoBinarioImpl** con le seguenti operazioni usando (dove possibile) la ricorsione:

1. **public int level(NodoBinario u):** restituisce la profondità (o livello) di un nodo. Ricorda che la profondità della radice è 0 e quella di un nodo diverso dalla radice è quella del padre del nodo, incrementata di 1.
2. **public int altezza():** restituisce l'altezza dell'albero binario.
3. **public in numFoglie():** restituisce il numero di foglie dell'albero binario.
4. **public int numNodiInterni():** restituisce il numero di nodi *interni* dell' albero binario. Ricorda che un nodo interno è un nodo che NON è foglia.
5. **public boolean *equals (Object anotherTree)***: restituisce true se l'albero (*this*) e *anotherTree* (eventualmente vuoti) sono uguali (stessa struttura e medesimi contenuti); false, altrimenti.

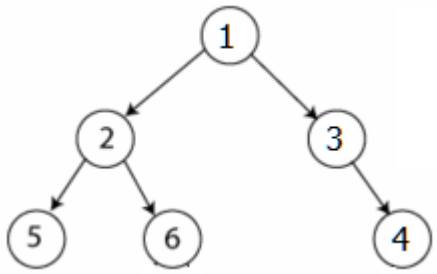
Albero di test costruito da **AlberoBinarioDemo**:



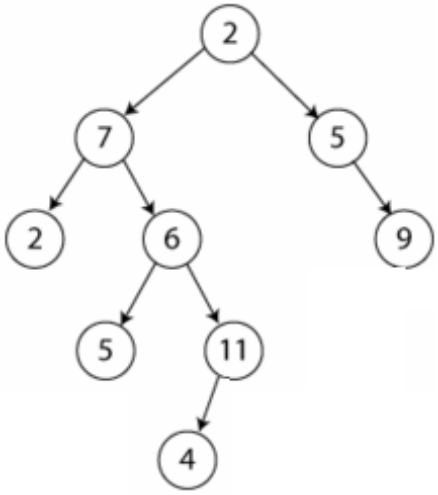
# Esercizi

6. **public void eliminaFoglieUguali():** modifica l'albero eliminando tutte le foglie che hanno un valore uguale al valore del fratello.
7. **public boolean search (Object elem):** stabilisce se un elemento appartiene o meno all'albero binario, attraverso una *visita esaustiva* ricorsiva. Il caso base è banale: se l'albero è vuoto, si restituisce false (zero). La visita viene interrotta non appena si trova l'elemento cercato (la prima occorrenza).
8. **public List nodiCardine ():** Definire un algoritmo ricorsivo che restituisce i nodi cardine di T. In un albero binario T, un nodo *u* è un nodo cardine se e solo se $p_u=h_u$ dove $p_u$ è la profondità di u e $h_u$ è l'altezza dell'albero radicato in *u*.

***(Appello 18 Giugno 2018)***




Nodi cardine: 2, 3




Nodi cardine: 5, 6