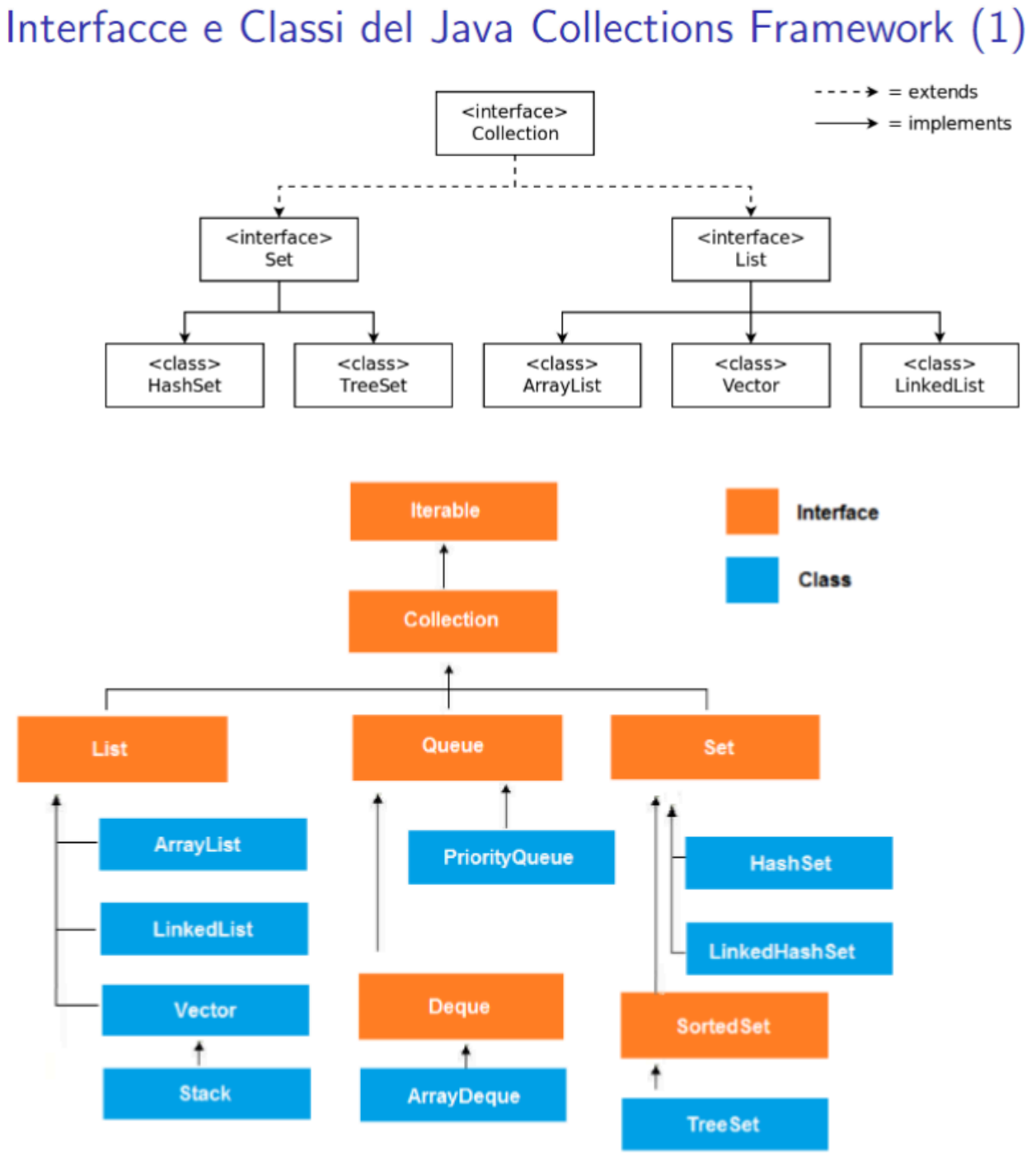
1. Contenitore

```java
public interface Contenitore{
    boolean isEmpty();
    int size();
}
```

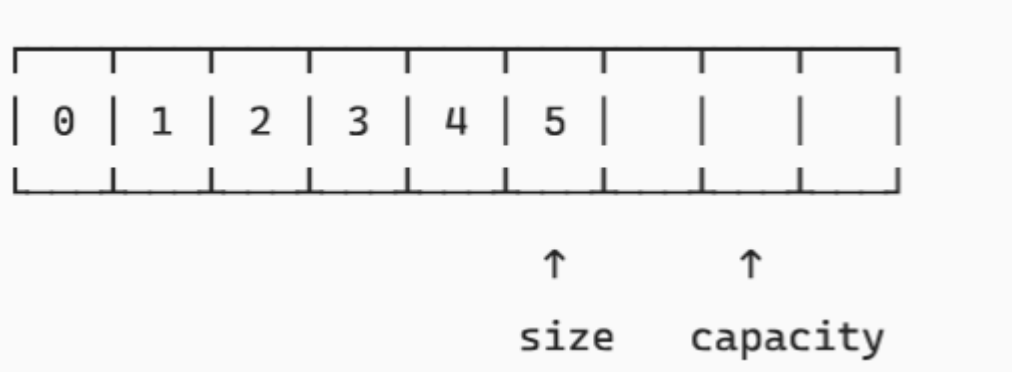


2. ArrayList

- Array che ha una gestione dinamica della memoria

- Tipi generici / Element
- = Lo usi dentro alle strutture dati
- Complessità $O(1)/O(N)$

```
public class ArrayListInteger{
    private int[] elements;
    private int size; // tutti gli elementi
    private int capacity; // gli elementi presenti ora

    // Elementi = Tipo "E" = Element

    // Metodi di accesso generici
    add(int e)
    remove(int index)
    int get(int index)
}

// In altre classi userai proprio
ArrayList<Integer> array = new ArrayList<>();
```



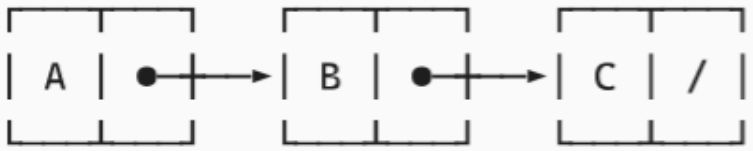
3. LinkedList

- Lista concatenata
  - (Info) -> (Next)
- Può essere trasformata in coda doppia per questo motivo

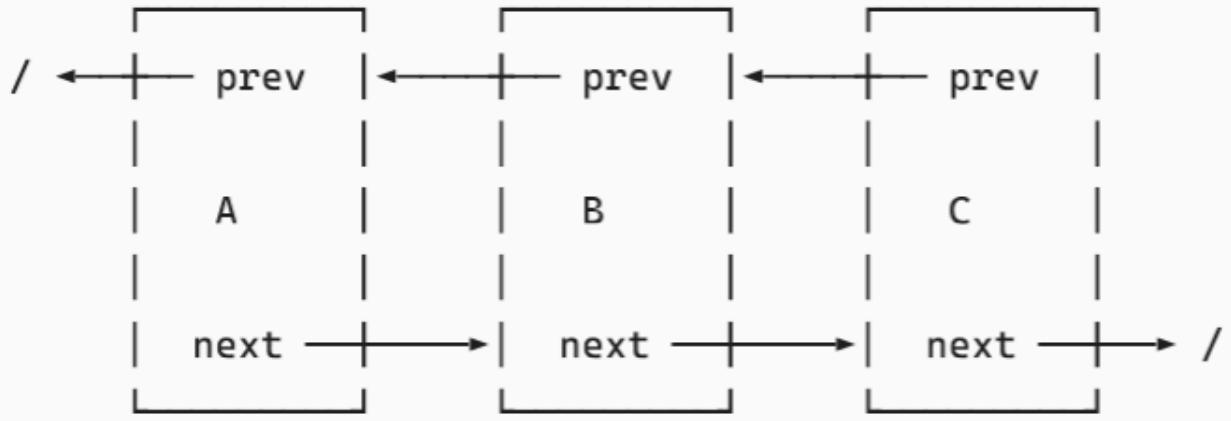
```
import java.util.LinkedList;

public class LinkedListExample{

    LinkedList<String> lista = new LinkedList<>();

    // Metodi di accesso
    add(Element e);
    remove(Element e);

    // Lista = First / Last (simile a coda doppia)
```

```
    addFirst / addLast
    removeFirst / removeLast
}
```

Lista semplicemente concatenata:



Lista doppiamente concatenata:



4. Pila

- Ordine LIFO (Stack)
- Metodi accesso: Push / Pop
- Puntatori: Top / Bottom
- Complessità: $O(log(n))$

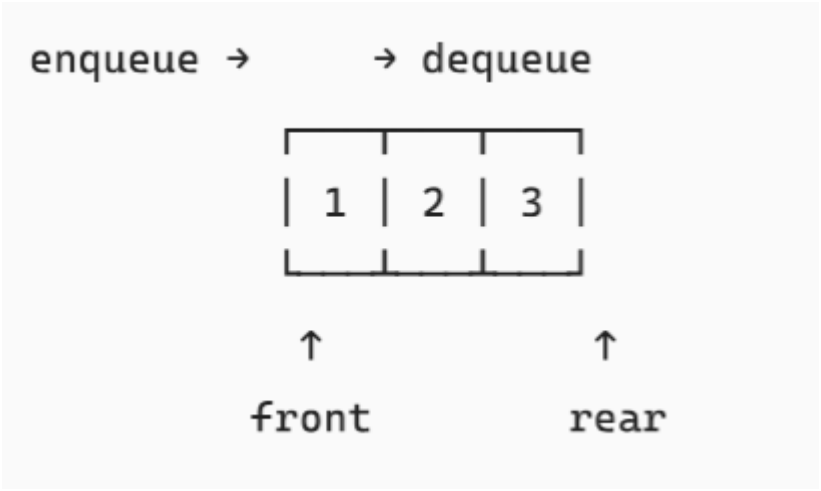
```
public class Pila{
    private ArrayList<T> elements;

    // Metodi di accesso
    void push();
    T pop();
}
```

5. Coda

- Ordine FIFO (Queue)
- Metodi accesso: Enqueue / Dequeue
- Puntatori: Front / Rear (Avanti / Indietro)

- Complessità: $O(log(n))$

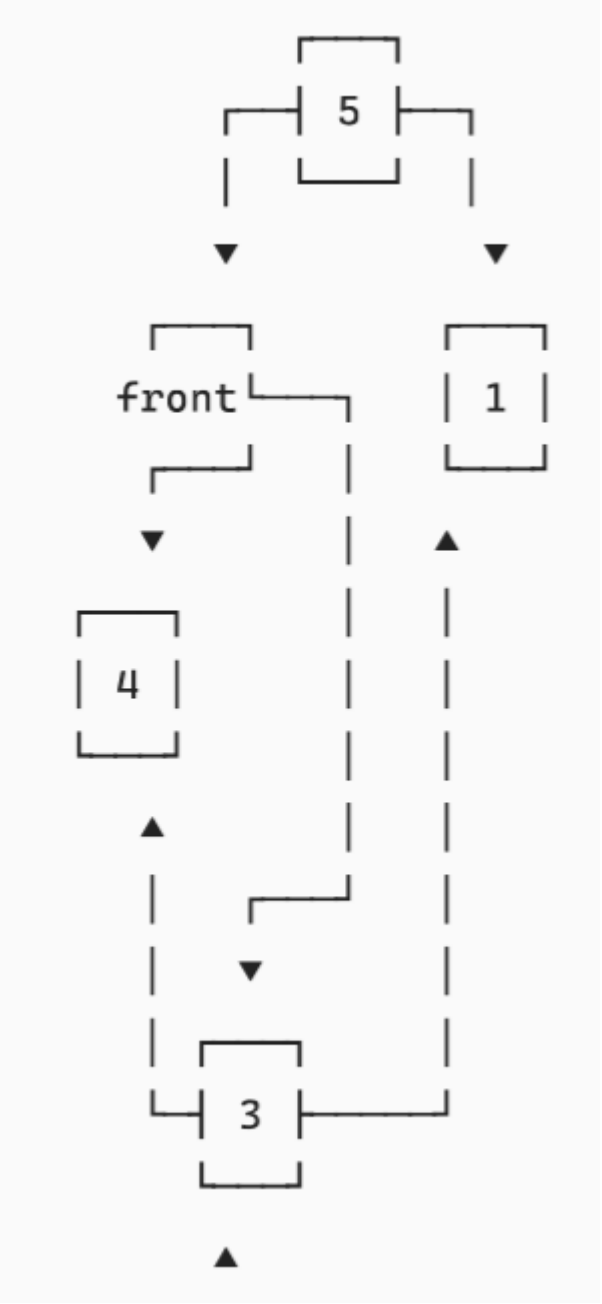
```
public class Coda{
    private ArrayList<T> elements;

    // Metodi di accesso
    void enqueue();
    T dequeue();
}
```



5.1 Coda circolare

- Usiamo "front/rear" in modo intelligente avvolgendosi
- Complessità: $O(n)$ con ArrayList / $O(log(n))$

```
public class CodaCircolare{
    private ArrayList<T> elements;
    private int front;
    private int rear;
    private int capacity;

    // isempty() / size() da contenitore
    // Metodi di accesso
    void enqueue();
    T dequeue();
}
```

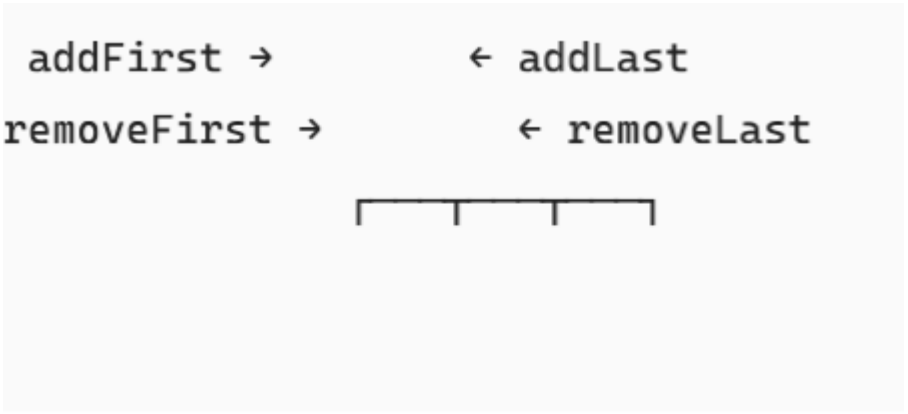


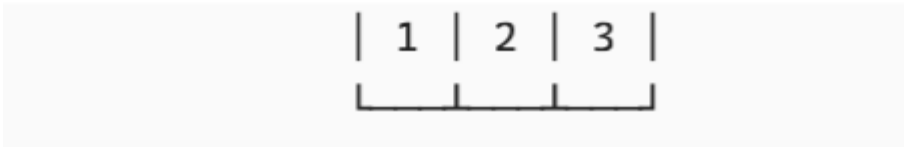
5.2 Coda doppia (Deque)

- Puntatori ad inizio e fine gestito "a coda" -> FIFO

```
public class CodaDoppia{
    private ArrayList<T> elements;
    private int first;
    private int last;
    private int capacity;

    // Metodi di accesso
    void add();
    T remove();

    // Eventuali addFirst / addLast + removeFirst / removeLast
```

```
    // Caso d'uso: isPalindrome (PalindromeChecker)
}
```





5.3 Coda prioritaria (Priority queue)

- Coda FIFO con ogni elemento che ha una descrizione e priorità

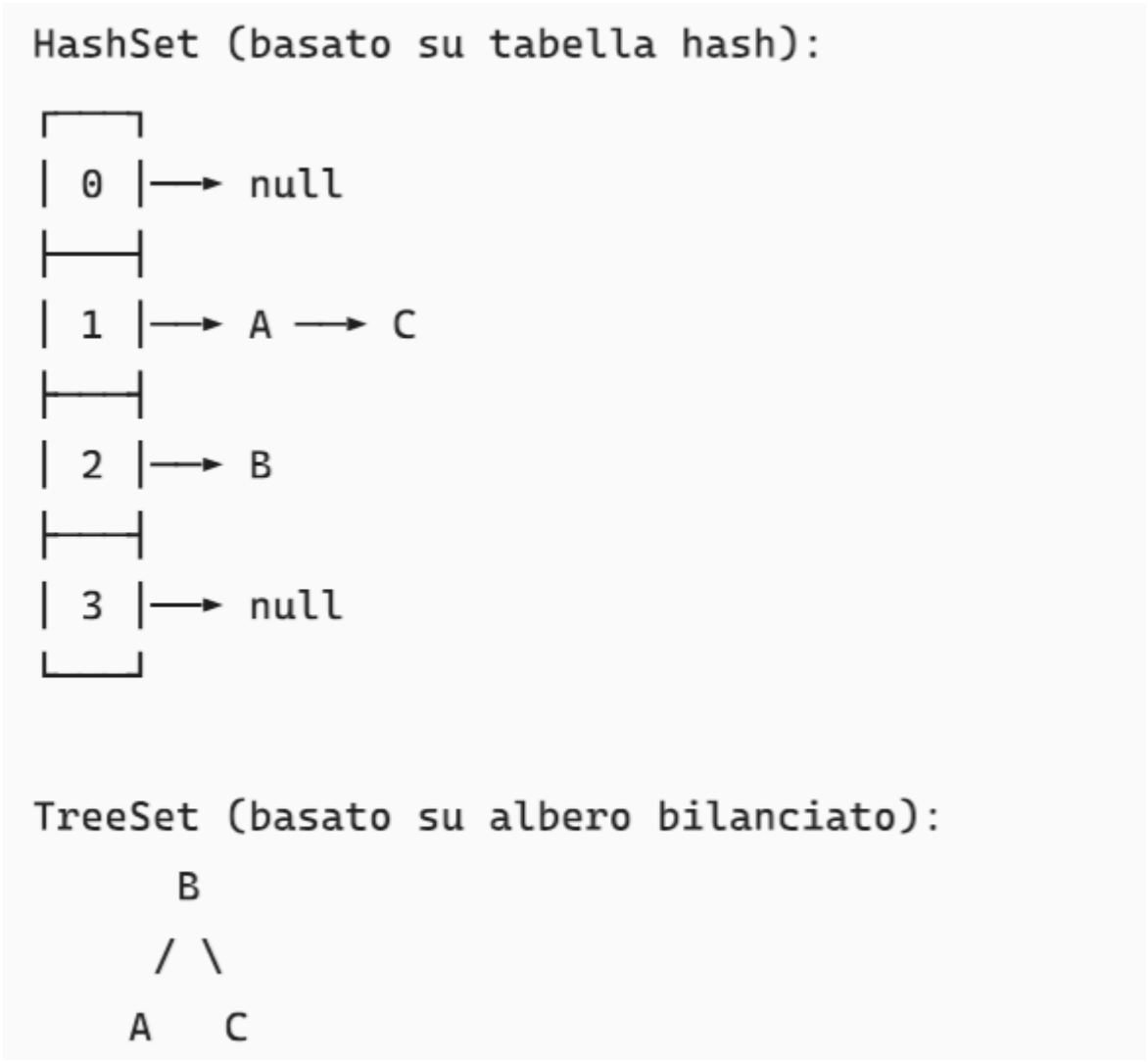
```
public class CodaPrioritaria{
    private String descrizione;
    private int priorita;
    private ArrayList<T> elements;

    // Metodi di accesso (generici)
    // Noi qui ci interessa la priorità, non l'ordine

    // add() / remove() / get()
}
```

```
Coda (minimo in testa):

          1
        /   \
       2     3
      / \
     4   5

Inserimento di un elemento: O(log n)
Estrazione del minimo: O(log n)
```

6. Insiemi (Set)

- Collezione che non contiene elementi duplicati
    - 1. HashSet -> Set con elementi univoci per valore -> Pochi valori ma unici -> $O(1)$
    - 2. TreeSet -> Struttura ad albero (usato se ci serve "avere un ordine migliore" -> $O(log(n))$

```java
public class HashSetExample{

    HashSet<String> set = new HashSet<>();

    void add(Element e);
    void remove();
    // addAll / retainAll / removeAll -> Unione / Intersezione / Differenza
    void contains(Element e);
}
```



7. Mappe (Map)

- Collezioni con elementi nella coppia [chiave; valore]
- Usiamo le chiavi per indicizzare i valori
    - Stesse varianti: HashMap / TreeMap

```
public class HashMapExample{

    HashMap<String> map = new HashMap<>();

    // Metodi di accesso
    void put(Key k, Value v);
    E get(Key k);
    void remove(Key v);
    // contains per key/value

    // Caso d'uso: ContaRicorrenze
}
```

```
Tabella hash con risoluzione delle collisioni per concatenamento:

hash(k1) = hash(k3) = 1
hash(k2) = 2


┌─────┐
│  0  │──► null
├─────┤
│  1  │──► [k1:v1] ──► [k3:v3]
├─────┤
│  2  │──► [k2:v2]
├─────┤
│  3  │──► null
└─────┘
```

Struttura Dati	Accesso	Ricerca	Inserimento	Cancellazione
ArrayList	O(1)	O(n)	O(1)/O(n)	O(n)
Stack (ArrayList)	O(1)	O(n)	O(1)	O(1)
Queue (LinkedList)	O(1)	O(n)	O(1)	O(1)
CodaArrayList	O(1)	O(n)	O(1)	O(n)
CodaArrayListEstremoFissoLiberato	O(1)	O(n)	O(1)	O(1)
Circular Queue	O(1)	O(n)	O(1)	O(1)
Deque (ArrayDeque)	O(1)	O(n)	O(1)	O(1)
Priority Queue	O(1)*	O(n)	O(log n)	O(log n)
LinkedList	O(n)	O(n)	O(1)**	O(1)**
HashSet	N/A	O(1)	O(1)	O(1)
TreeSet	N/A	O(log n)	O(log n)	O(log n)
HashMap	N/A	O(1)	O(1)	O(1)
TreeMap	N/A	O(log n)	O(log n)	O(log n)
Hash Table	N/A	O(1)	O(1)	O(1)