

CODE (QUEUE)

CARATTERISTICHE GENERALI

- Comportamento definito **FIFO**
 - **First In, First Out**
 - primo inserito è il primo ad essere estratto
 - può essere ispezionato solo il primo oggetto della coda

UTILIZZO DELLE CODE

- Simulazione sportello bancario
 - clienti in coda
- File da stampare vengono inseriti in una coda di stampa
- Task scheduler

INTERFACCIA

- Definisce le operazioni:
 - **enqueue**: inserisce oggetto nella coda
 - **dequeue**: elimina dalla coda oggetto inserito per primo
 - **getFront**: esamina primo oggetto senza estrarlo
- operazioni base del ADT "container"

```
public interface Queue extends Container {  
    void enqueue(Object obj);  
    Object dequeue();  
    Object getFront();  
}
```

REALIZZAZIONE DELLA CODA

- Definiamo due nuove **eccezioni**
 - `class EmptyQueueException extends RuntimeException`
 - `class FullQueueException extends RuntimeException`

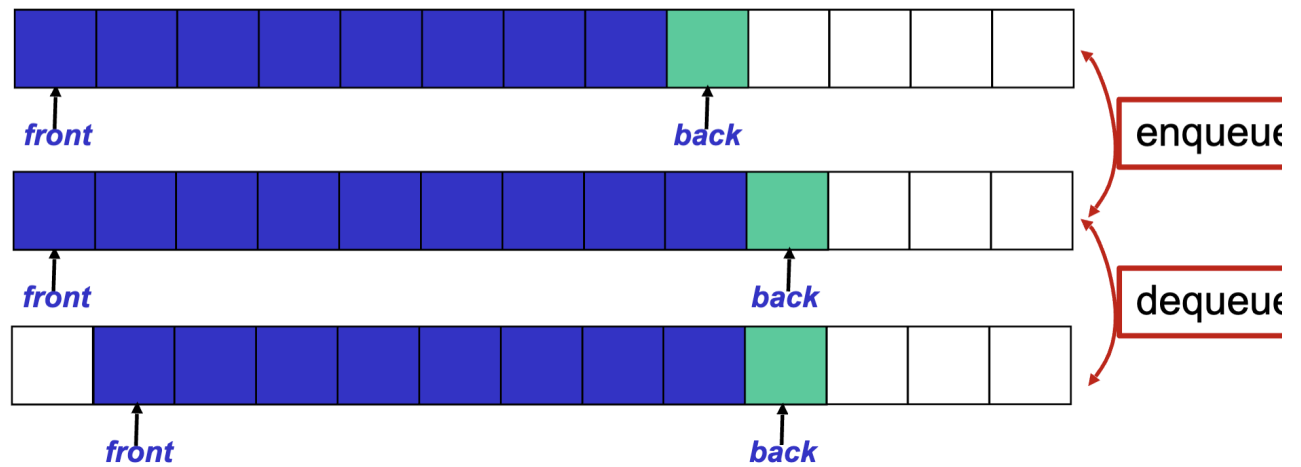
IMPLEMENTAZIONE CON ARRAY RIEMPITO SOLO IN PARTE

```
1 public class SlowFixedArrayQueue implements Queue {
2     protected Object[] array;
3     protected int arraySize;
4     public static final int INIT_SIZE = 100;
5
6     public SlowFixedArrayQueue() {
7         array = new Object[INIT_SIZE];
8         makeEmpty();
9     }
10
11     public boolean isEmpty() {
12         return arraySize == 0;
13     }
14
15     public void makeEmpty() {
16         arraySize = 0;
17     }
18
19     public void enqueue(Object obj) {
20         if (arraySize == array.length)
21             throw new FullQueueException();
22         array[arraySize++] = obj;
23     }
24
25     public Object dequeue() {
26         Object obj = getFront();
27         arraySize--;
28         for (int i = 0; i < arraySize; i++) {
29             array[i] = array[i + 1];
30         }
31         return obj;
32     }
33
34     public Object getFront() {
35         if (isEmpty())
36             throw new EmptyQueueException();
37         return array[0];
38     }
39 }
40
```

- Il metodo `dequeue` ha complessità $O(n)$
 - non è molto efficiente

IMPLEMENTAZIONE CON "ARRAY RIEMPITO SOLO NELLA PARTE CENTRALE"

- Si usano due indici
 - **front**: indica primo elemento nella coda
 - **back**: indica primo posto libero dopo ultimo elemento nella coda
 - numeri di elementi: $\text{back} - \text{front}$



```

1  public class FixedArrayQueue implements Queue {
2      protected Object[] array;
3      protected int front, back;
4      public static final int INIT_SIZE = 100;
5
6      public FixedArrayQueue() {
7          array = new Object[INIT_SIZE];
8          makeEmpty();
9      }
10
11     public boolean isEmpty() {
12         return back == front;
13     }
14
15     public void makeEmpty() {
16         back = front = 0;
17     }
18
19     public void enqueue(Object obj) {
20         if (back == array.length)
21             throw new FullQueueException();
22         array[back++] = obj;
23     }
24
25     public Object dequeue() {
26         Object obj = getFront();
27         front++;
28         return obj;
29     }
30
31     public Object getFront() {
32         if (isEmpty())
33             throw new EmptyQueueException();
34         return array[front];
35     }
36 }
37

```

- Il metodo `dequeue` ha ora efficienza $O(1)$

IMPLEMENTAZIONE CON ARRAY A RIDIMENSIONAMENTO DINAMICO

```
public void enqueue(Object obj) {  
    if (back == array.length)  
        array = resize(2 * array.length);  
    array[back++] = obj;  
}
```

- Tutte le operazioni continuano ad avere efficienza: $O(1)$

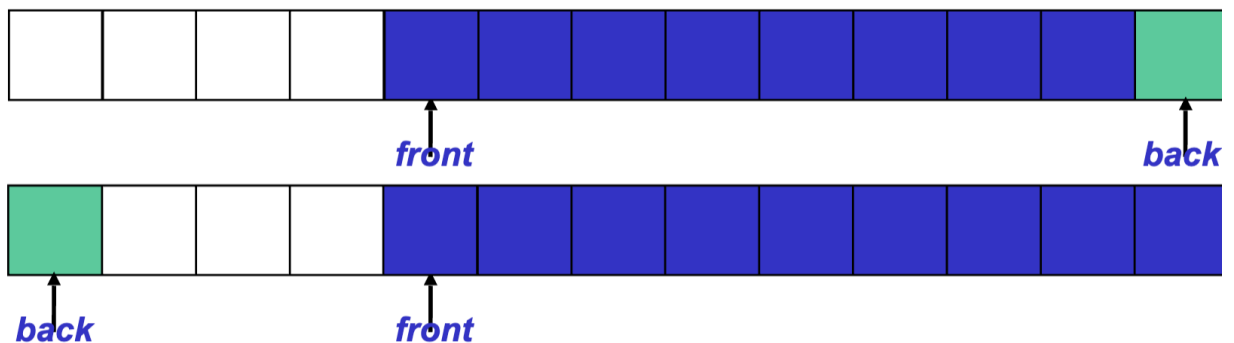
⚡ Attenzione

Si rischia, se aumenta molto indice front, di avere un array quasi totalmente vuoto

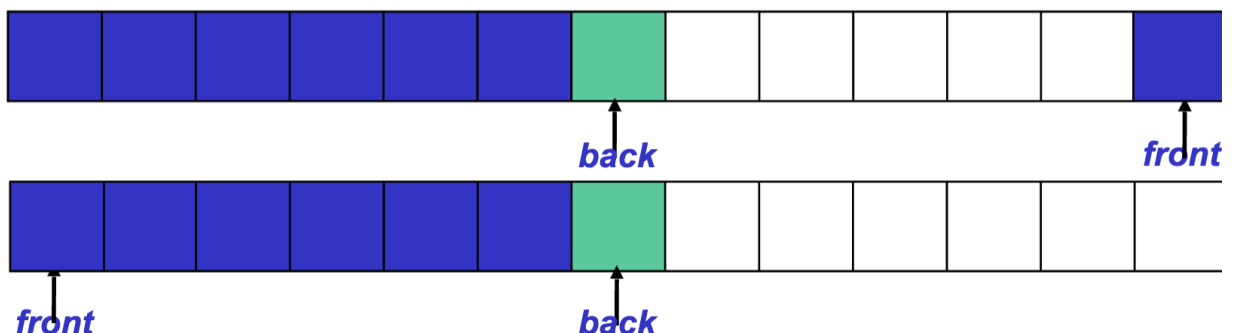
CODA CON ARRAY CIRCOLARE

- Array in cui, quando è pieno, si re-inizia ad inserire elementi dal primo elemento
- È pieno solo quando `front == back`

- Incremento dell'indice **back** da $n-1$ a 0



- Incremento dell'indice **front** da $n-1$ a 0

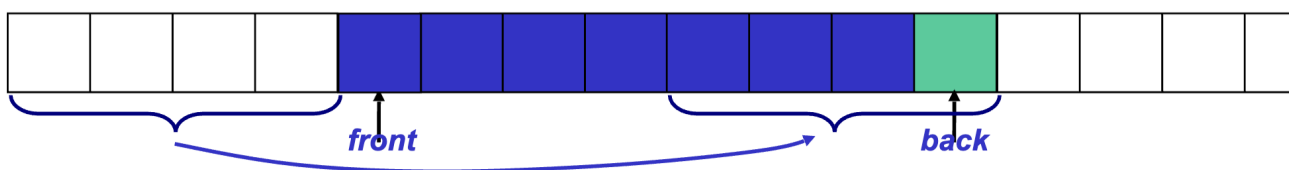


IMPLEMENTAZIONE CON ARRAY STATICI

```
1 public class FixedCircularArrayQueue extends FixedArrayQueue {
2     protected int increment(int index) {
3         return (index + 1) % array.length;
4     }
5
6     public void enqueue(Object obj) {
7         if (increment(back) == front)
8             throw new FullQueueException();
9         array[back] = obj;
10        back = increment(back);
11    }
12
13    public Object dequeue() {
14        Object obj = getFront();
15        front = increment(front);
16        return obj;
17    }
18 }
19
```

RIDIMENSIONARE CON ARRAY CIRCOLARE

- Potrebbe dare array pieno anche quando l'indice `back` non è l'ultimo elemento
- Sposto gli elementi prima di `front` nella seconda metà dell'array

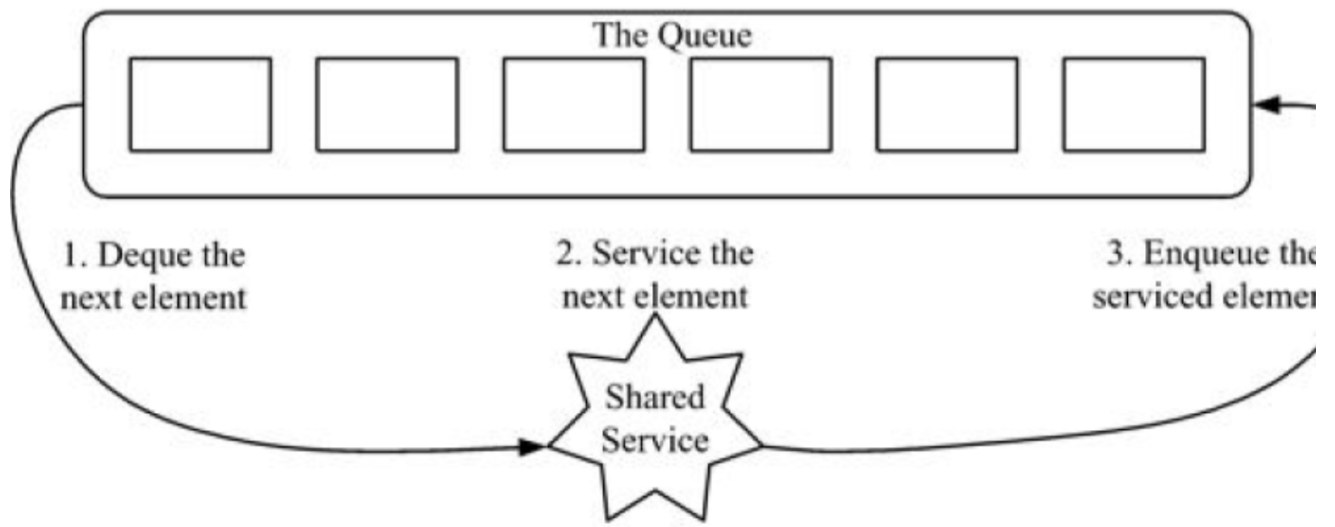


```
public void enqueue(Object obj) {
    if (increment(back) == front) {
        array = resize(2 * array.length);
        if (back < front) {
            System.arraycopy(array, 0, array, array.length / 2, back);
            back += array.length / 2;
        }
    }
    array[back] = obj;
    back = increment(back);
}
```

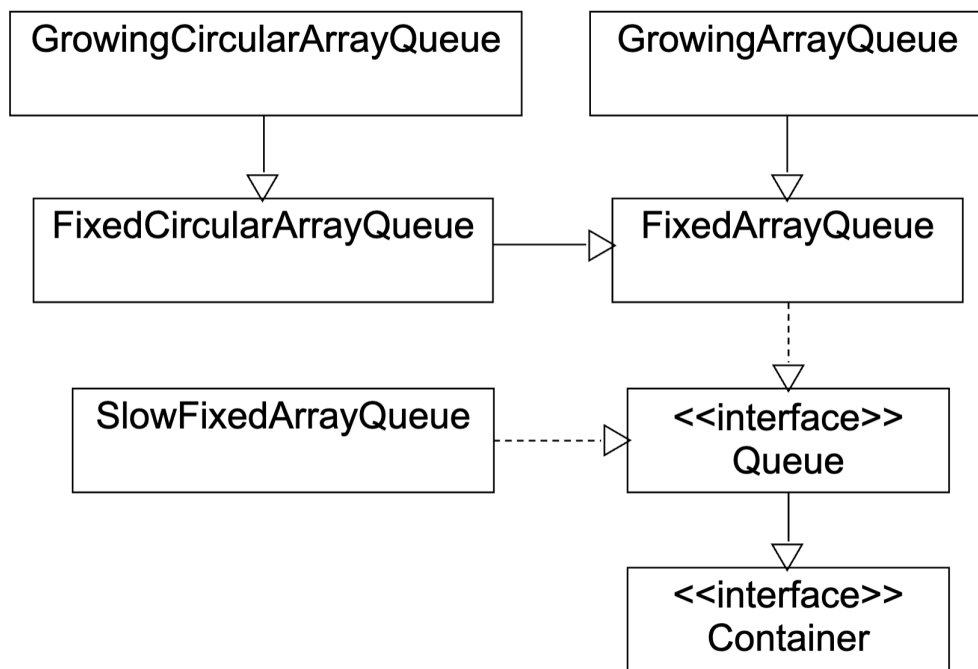
CODA CIRCOLARE

- Politica **round robin**

- la CPU esegue una porzione del processo che ha atteso più a lungo e poi viene re-inserito nell'ultima posizione



GERARCHIA DI CLASSI E INTERFACCE



CODA DOPPIA (DEQUE)

- Elementi possono venire inseriti ed estratti ai due estremi

INTERFACCIA

```
public interface Deque extends Container {  
    void addFirst(Object obj);  
    void addLast(Object obj);  
  
    Object removeFirst();  
    Object removeLast();  
  
    Object getFirst();  
    Object getLast();  
  
    int size();  
}
```

- Può essere realizzata con **array circolare ridimensionabile**
- Tutti i metodi hanno prestazioni $O(1)$

ANALISI PRESTAZIONI

Prestazioni in sintesi:

- enqueue: $O(1)$
- dequeue: $O(1)$ (usando 2 indici) - $O(n)$ (usando 1 indice)
- getFront: $O(1)$