SOMMARIO

- Coda (queue):
  - Specifica: interfaccia.
  - Implementazione:
    - Strutture indicizzate (*array*):
      - Array di dimensione variabile.
      - Array circolari.
    - Strutture collegate (*nodi*).
  - Prestazioni.

QUEUE

- *Una coda è una sequenza $<a_1,\ldots,a_n>$ di elementi dello stesso tipo, della quale si distinguono due elementi, il primo e l'ultimo inserito (a_1 e a_n, denominati la testa e il fondo della coda). La modalità di accesso è "primo elemento inserito, primo elemento rimosso" (FIFO: first in, first out).*

- È una linea d'attesa che cresce aggiungendo elementi in fondo e si accorcia rimuovendo elementi dall'inizio.
- Una coda può essere descritta in termini di operazioni che ne modificano lo stato o che ne verificano lo stato: pertanto è un ADT.

QUEUE : specifica

- Una possibile (e minima) interfaccia è la seguente:

```
interface IQueue
{
    void Enqueue(object x);
    object Dequeue();
    object Front();
    bool IsEmpty();
    void Clear();
}
```

Inserisce l'oggetto `x` in fondo alla coda, dopo l'ultimo elemento.

Rimuove e restituisce il primo oggetto della coda.

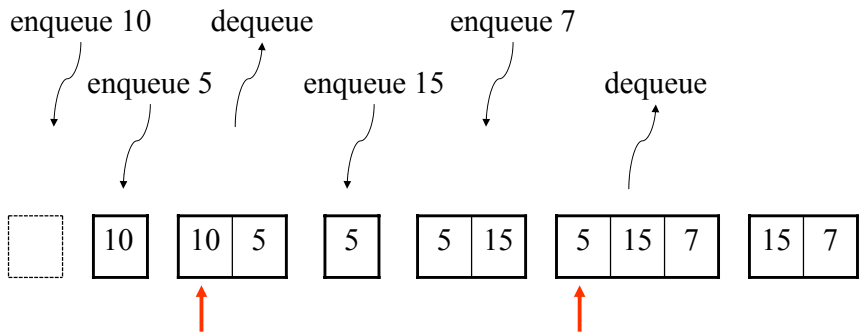
Restituisce il primo oggetto della coda senza rimuoverlo.

Verifica se la coda è logicamente vuota.

Svuota la coda.

QUEUE

- Vediamo graficamente alcune operazioni su una coda.



QUEUE: esempio

```
IQueue q = new ArrayQueue(3);

Console.WriteLine(q.IsEmpty());
Console.Write("------------\n");

for (int i = 0; i < 7; i++)
    q.Enqueue(i);
Console.WriteLine(q.IsEmpty());
Console.Write("------------\n");

for (int i = 0; i < 7; i++)
    Console.Write(q.Dequeue() + " ");
Console.WriteLine("\n"+q.IsEmpty());
Console.Write("------------\n");

try{
    q.Dequeue();
}
catch (Exception e){
    Console.WriteLine(e.Message);
}
```

Una possibile uscita

```
True
------------
False
------------
0 1 2 3 4 5 6
True
------------
Coda vuota.
```


QUEUE: implementazione

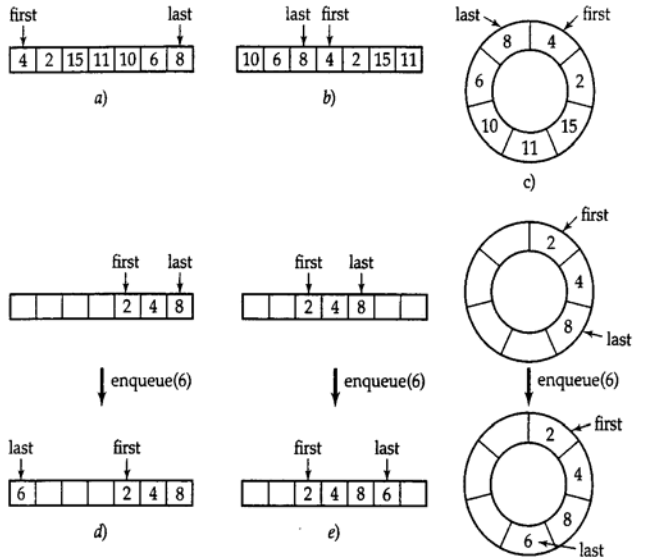
- Si consideri ora la *realizzazione* della coda. Si sono usate le operazioni espresse dall'interfaccia, ma devono essere implementate, cioè realizzate come metodi che operano sui dati della coda.
- L'implementazione può essere realizzata sfruttando due diversi tipi di strutture dati: strutture indicizzate (*array*) o strutture collegate (*nodi*).
- Analogamente a quanto visto per le pile, l'implementazione con array consiste nell'usare un *array flessibile*, cioè un array che può *modificare dinamicamente* le sue *dimensioni*, perchè, in generale, non è noto a priori il numero di oggetti che una coda deve contenere.

QUEUE: implementazione

- In particolare, nella struttura dati coda gli elementi vengono aggiunti alla fine e estratti dall'inizio, pertanto possono esserci *celle libere all'inizio* dell'array. Per non essere sprecate devono essere usate per accodare nuovi elementi. Quindi la fine della coda può trovarsi all'inizio dell'array. Tale situazione è illustrata dall'*array circolare*.

- Vediamo una rappresentazione grafica.

QUEUE: array circolare



QUEUE: array circolare

- Tuttavia si opera su array non circolari, quindi i metodi `Enqueue()` e `Dequeue()` devono realizzare la possibilità di "girare" intorno all'array, cioè di riusare le celle libere, quando si aggiunge o rimuove un elemento.

- Si realizza una struttura dati complessa utilizzandone una primitiva, l'array.
- Il pregio di tale implementazione è il *basso costo computazionale* per inserimenti ed estrazioni, *O(1)*, mentre il punto critico riguarda la *gestione della memoria*.

QUEUE : prestazioni

- Nota sull'implementazione di un *array circolare*:
  - Per semplificare l'implementazione si può pensare di non realizzare un array circolare: ad ogni `Dequeue()` tutti gli elementi presenti vengono fatti "traslare" di una posizione indietro in modo da avere sempre la posizione iniziale nella prima cella dell'array.
  - Tuttavia tale soluzione porta ad avere un costo lineare *O(n)* nel caso medio per il metodo `Dequeue()`.

QUEUE : implementazione con array

```
class ArrayQueue : IQueue
{
    int first, last, isize;
    object[] data;

    public ArrayQueue(int dim)
    {
        isize = dim;
        data = new object[isize];
        first = last = -1;
    }

    public bool IsEmpty()
    {
       return first == -1;
    }

   …
```

I dati della coda sono contenuti in un array di `object`.

QUEUE : implementazione con array

```
public void Enqueue(object x)
{
    if (first == 0 && last == data.Length - 1 ||
                                        first == last + 1)
        RaddoppiaArray();

    if (last == data.Length - 1 || last == -1)
    {
        data[0] = x;
        last = 0;
        if (first == -1)
            first = 0;
    }
    else
        data[++last] = x;
}
   …
```

Array circolare.

QUEUE : implementazione con array

```
public object Dequeue()
{
    if (IsEmpty())
        throw new Exception("Coda vuota.");

    int el_num;
    if (first <= last)
        el_num = last - first + 1;
    else
        el_num = data.Length - first + last + 1;
    if ((el_num < data.Length / 4) && (el_num >= isize / 4))
        DimezzaArray(el_num);


    object tmp = data[first];
    if (first == last)
        last = first = -1;
    else if (first == data.Length - 1)
        first = 0;
    else
        first++;
    return tmp;}
```

...

Numero di elementi della coda.

Array circolare.

QUEUE : implementazione con array

```
public object Front()
{
    if (IsEmpty())
        throw new Exception("Coda vuota.");
    return data[first];
}


public void Clear()
{
    last = first = -1;
}
```

...

QUEUE : implementazione con array

- Il controllo su `el_num<data.Length/4` serve per evitare di invocare un dimezzamento della dimensione dell'array non appena si è eseguito un raddoppio della sua dimensione.
- Il controllo su `el_num>=isize/4` assicura di non scendere mai sotto la dimensione iniziale della coda.
- I metodi privati `RaddoppiaArray()` e `DimezzaArray()` implementano il concetto di array flessibile.
- Si sono inseriti dei controlli per implementare il concetto di array circolare.

QUEUE : implementazione con array

```
private void RaddoppiaArray()
{
    object[] newdata = new object[2 * data.Length];

    for (int i = first, j = 0; j < data.Length;
                    j++, i++, i = i % data.Length)
    {
        newdata[j] = data[i];
    }
    first = 0;
    last = data.Length - 1;
    data = newdata;
}
```

...

QUEUE : implementazione con array

```
private void DimezzaArray(int el_num)
{
    object[] newdata = new object[data.Length / 2];

    for (int i = first, j = 0; j < el_num; j++,
                          i++, i = i % data.Length)
    {
        newdata[j] = data[i];
    }
    first = 0;
    last = el_num - 1;
    data = newdata;
}
```


QUEUE : prestazioni

- Le operazioni di accodamento ed estrazione vengono eseguite a tempo costante *O(1)*, essendo realizzate con un array.
- Come per le pile, l'inserimento di un elemento in una *coda piena* richiede l'assegnazione di maggiore memoria e gli elementi dell'array pieno sono copiati in quello nuovo. Quindi inserire elementi nel caso peggiore richiede un costo *O(n)*. Tuttavia si può pensare di avere ancora *costo* costante in *media*: tale operazione avviene in media dopo *n* inserimenti, quindi il *costo distribuito* per ogni operazione è *O(1)*. Lo stesso vale per l'estrazione di un elemento.
- Inoltre vi sono dei controlli (che hanno un costo) per implementare il concetto di array circolare.

QUEUE : considerazioni

- L'uso della struttura dati primitiva array, come rappresentazione interna dei dati della coda, evidenzia due aspetti:
  - La semplicità ed efficienza dell'uso degli indici.
  - Il costo che si paga nella gestione della memoria per ottenere un "array ridimensionabile" e "circolare".
- Inoltre è evidente il vantaggio di usare un ADT, che nasconde i problemi dell'implementazione all'utilizzatore.

QUEUE : implementazione con nodi

- Una implementazione alternativa consiste nell'uso di *strutture collegate*: i dati sono memorizzati in nodi, ogni nodo ha un riferimento al precedente.
- In questo caso non vi sono problemi di gestione della memoria: si alloca solo il numero necessario di nodi e risulta semplice inserire in "coda" ed estrarre dalla "testa". Ogni operazione ha costo *O(1)*.
- Ogni inserimento provoca la creazione di un oggetto nodo. Ogni estrazione rilascia un oggetto nodo.

QUEUE : considerazioni

- Entrambe le implementazioni hanno metodi con costo *O(1)*, quindi è necessario eseguire un *confronto* sui *tempi di esecuzione* delle due implementazioni (per esempio, effettuando 10^7 chiamate ai metodi `Enqueue()` e `Dequeue()`):
  - Se non è necessario risparmiare memoria e si conosce la massima dimensione della coda, allora l'implementazione con array è più veloce (per esempio, *ArrayQueue = 9.063E-001 sec* e *NodeQueue = 1.734E+000 sec*).
  - Se è importante risparmiare memoria e non si conosce il numero di oggetti da inserire nella coda, allora l'implementazione con nodi è più veloce (per esempio, *ArrayQueue = 2.563E+000 sec* e *NodeQueue = 1.734E+000 sec*).

PRIORITY QUEUE

- Nelle *code prioritarie* gli elementi vengono estratti in base alla loro priorità e alla loro attuale posizione in coda.
- Per esempio, in una *coda di processi* può essere necessario che per il corretto funzionamento del sistema il processo P_2 venga eseguito prima del processo P_1, anche se P_1 è stato inserito per primo nella coda.
- Il problema di una coda prioritaria è trovare realizzazioni efficienti che consentano accodamenti ed estrazioni veloci.

PRIORITY QUEUE

- Vi sono diverse soluzioni: in generale, c'è la necessità di mantenere due strutture dati (per es., una ordinata secondo l'ordine di ingresso nella coda e l'altra ordinata secondo la priorità).

- Dal punto di vista della complessità computazionale si ottengono prestazioni da $O(n)$ a $O(n^{1/2})$ in relazione alle diverse implementazioni.