Realizzazione della coda

Coda (queue)

```
public interface Queue extends Container{

   // inserisce un elemento in coda
   void enqueue(Object obj);

   // estrae il primo elemento dalla coda
    Object dequeue();

   // restituisce il primo elemento della coda
   // senza rimuoverlo
    Object getFront();
}
```

- Si notino le similitudini con la pila
  - **enqueue** corrisponde a **push**
  - **dequeue** corrisponde a **pop**
  - **getFront** corrisponde a **top**

Coda (queue)

- Per ***realizzare una coda*** si può usare una struttura di tipo ***array*** "riempito solo in parte", in modo simile a quanto fatto per realizzare una pila
- Mentre nella pila si inseriva e si estraeva allo stesso estremo dell'array (l'estremo "destro"),
  qui dobbiamo inserire ed estrarre ai due diversi estremi
  - decidiamo di
    - inserire a destra (mantenendo un indice)
    - estrarre a sinistra (posizione 0)

Coda (queue)

- Come per la pila, anche per la coda bisognerà segnalare l'errore di accesso ad una coda vuota e gestire la situazione di coda piena (segnalando un errore o ridimensionando l'array)
- Definiamo
  - **EmptyQueueException** e **FullQueueException**

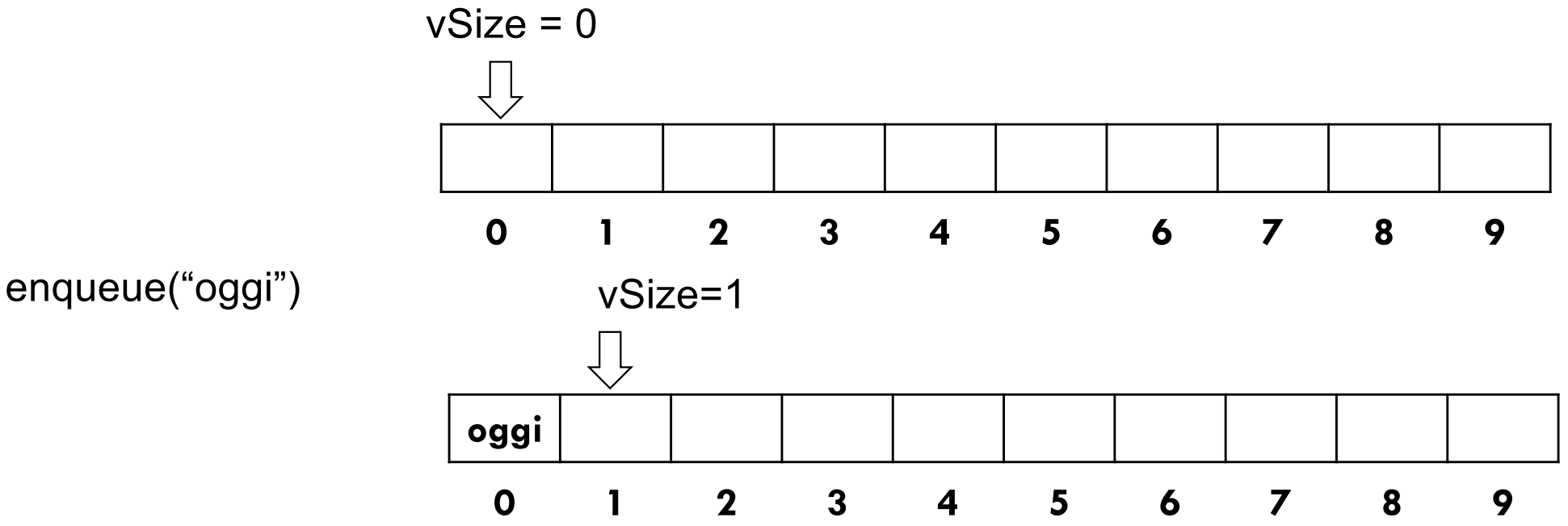
```
public class EmptyQueueException extends RuntimeException
{ }

public class FullQueueException extends RuntimeException
{ }
```

La classe SlowFixedArrayQueue

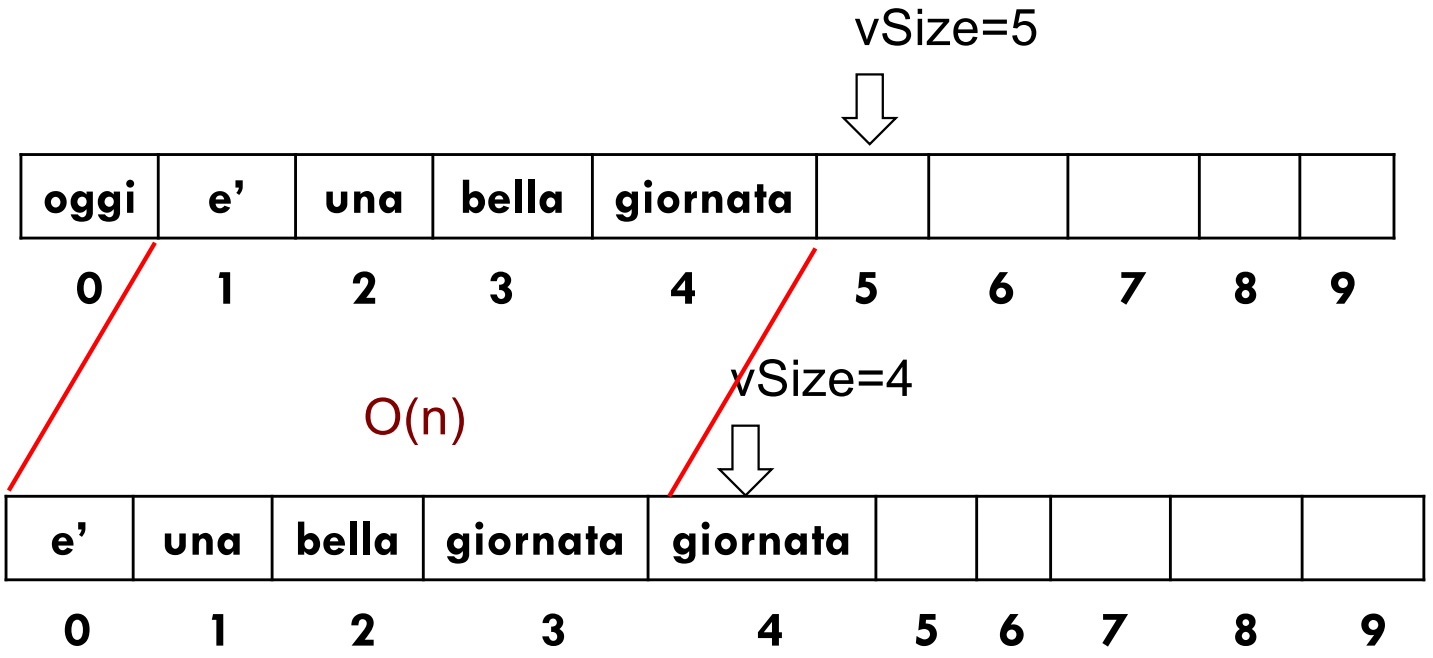
```
public class SlowFixedArrayQueue implements Queue
{  private Object[] v;
   private int vSize;
   public SlowFixedArrayQueue()
   {  v = new Object[100]; makeEmpty(); }
   public void makeEmpty()
   {  vSize = 0; }
   public boolean isEmpty()
   {  return (vSize == 0); }
   public void enqueue(Object obj)
   {  if (vSize == v.length) throw new FullQueueException();
      v[vSize++] = obj; }
   public Object getFront()
   {  if (isEmpty()) throw new EmptyQueueException();
      return v[0]; }
   public Object dequeue()
   {  Object obj = getFront();
      vSize--;
      for (int i = 0; i < vSize; i++) v[i] = v[i+1];
      return obj; }
}
```

Esempio


vSize = 0
0 1 2 3 4 5 6 7 8 9
enqueue("oggi")
vSize=1
oggi
0 1 2 3 4 5 6 7 8 9
enqueue("e'")
enqueue("una")
enqueue("bella")
enqueue("giornata")
vSize=5
oggi e' una bella giornata
0 1 2 3 4 5 6 7 8 9

Esempio

oggi ⟸ getFront()



oggi ⟸ dequeue()

enqueue("ciao")

Migliorare il metodo deque

- Questa semplice realizzazione con array, che abbiamo visto essere molto efficiente per la pila, è al contrario assai inefficiente per la coda
  - il metodo **dequeue** è **O(n)**, perché bisogna spostare tutti gli oggetti della coda per fare in modo che l'array rimanga "compatto"
  - la differenza rispetto alla pila è dovuta al fatto che nella coda gli inserimenti e le rimozioni avvengono alle due estremità diverse dell'array, mentre nella pila avvengono alla stessa estremità

Migliorare il metodo deque

- Per realizzare una coda più efficiente servono ***due indici*** anziché uno soltanto
  - un indice punta al primo oggetto della coda
  - l'altro indice punta alla prima cella libera dopo l'ultimo oggetto della coda
- In questo modo, aggiornando opportunamente gli indici, si ottiene la realizzazione di una coda con un "***array riempito solo nella parte centrale***" in cui tutte le operazioni sono **O(1)**
  - la gestione dell'array pieno ha le due solite soluzioni, ridimensionamento o eccezione

Array riempito nella parte centrale

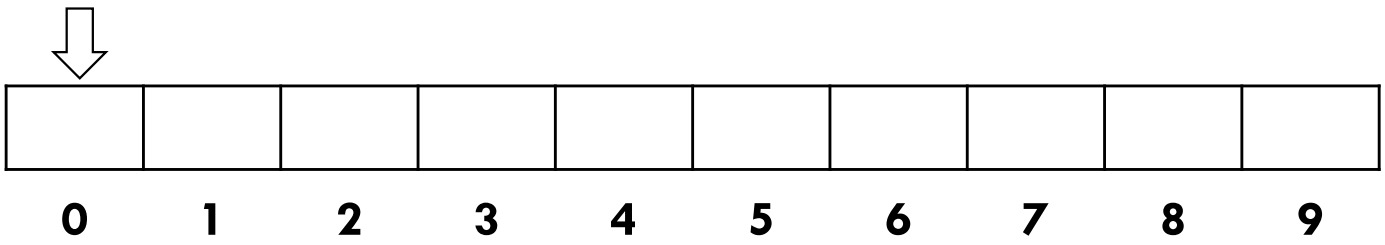
- Si usano **due indici** anziché uno soltanto
  - Indice **front**: punta al primo elemento nella coda
  - Indice **back**: punta al primo posto libero dopo l'ultimo elemento nella coda
  - Il numero di elementi è **(back – front)**, in particolare quando **front == back** l'array è vuoto.

La classe FixedArrayQueue

```
public class FixedArrayQueue implements Queue
{  protected Object[] v;
   protected int front, back;
   public FixedArrayQueue()
   {  v = new Object[100]; makeEmpty(); }
   public void makeEmpty()
   {  front = back = 0; }
   public boolean isEmpty()
   {  return (back == front);
   public void enqueue(Object obj)
   {  if (back == v.length) throw new FullQueueException();
      v[back++] = obj; }
   public Object getFront()
   {  if (isEmpty()) throw new EmptyQueueException();
      return v[front]; }
   public Object dequeue()
   {  Object obj = getFront();
      front++;
      return obj; }
}
```

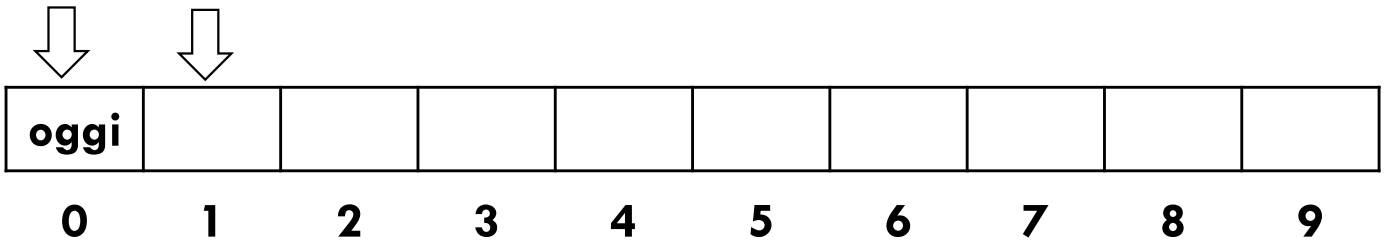
Esempio

front = back = 0



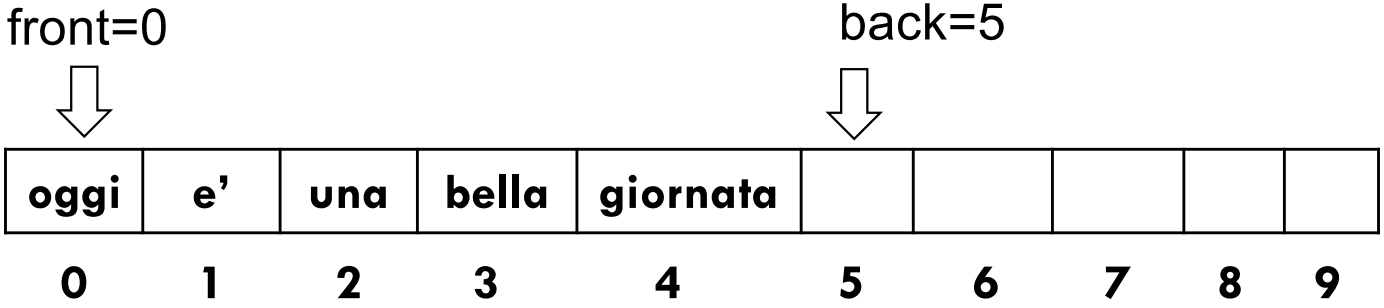
enqueue("oggi")

front=0 back=1



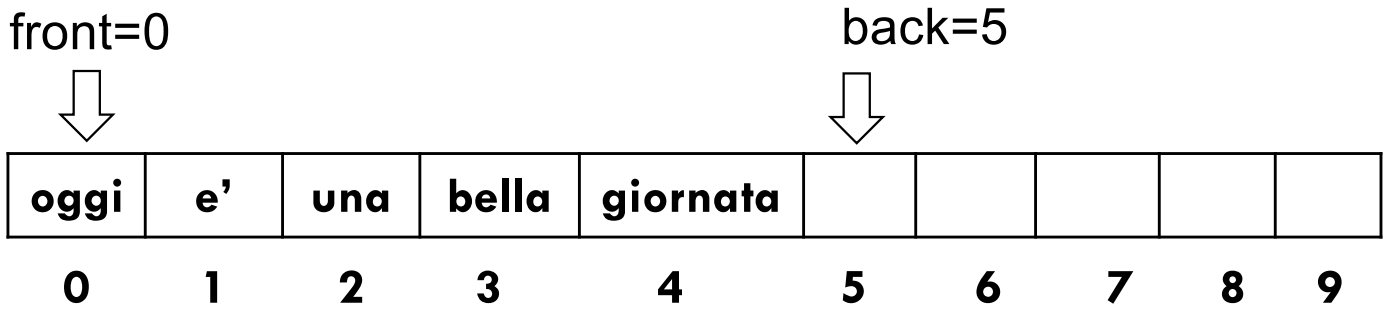
enqueue("e'")
enqueue("una")
enqueue("bella")
enqueue("giornata")

front=0 back=5



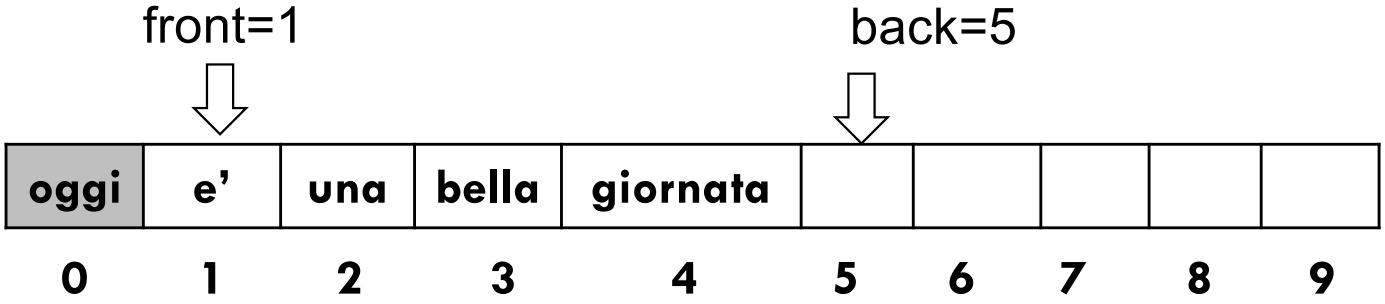
Esempio

oggi ⟸ getFront()



oggi ⟸ dequeue()

Questa cella non e' piu' accessibile (front va solo avanti)



enqueue("ciao")

e' ⟸ getFront()

Coda ridimensionabile

- Per rendere la coda ridimensionabile, usiamo la stessa strategia vista per la pila, estendendo la classe **FixedArrayQueue** e sovrascrivendo il solo metodo **enqueue**

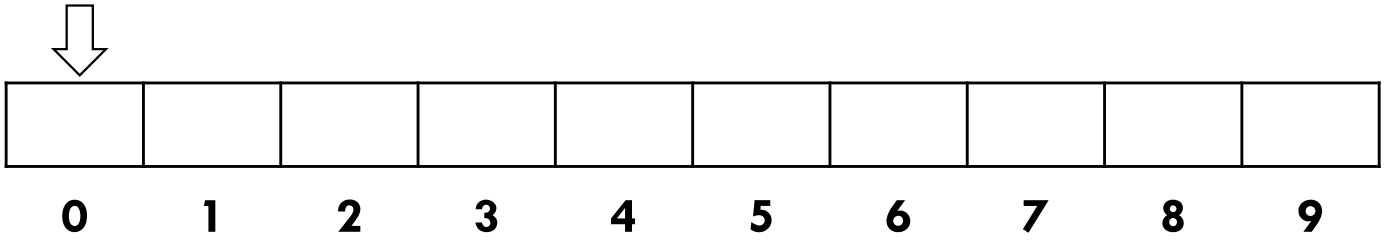
```
public class GrowingArrayQueue
                              extends FixedArrayQueue{
   public void enqueue(Object obj){
      if (back == v.length){
          v = resize(v, 2*v.length);
      }
      super.enqueue(obj);
    }
}
```

Prestazioni

- La realizzazione di una coda con un array e due indici ha la massima efficienza in termini di prestazioni temporali, tutte le operazioni sono **O(1)**, ma ha ancora un punto debole
- Se l'array ha N elementi, proviamo a
  - effettuare N operazioni **enqueue**
- e poi
  - effettuare N operazioni **dequeue**
- Ora ***la coda è vuota***, ma alla successiva operazione **enqueue** ***l'array sarà pieno***
  - lo spazio di memoria non viene riutilizzato

Esempio

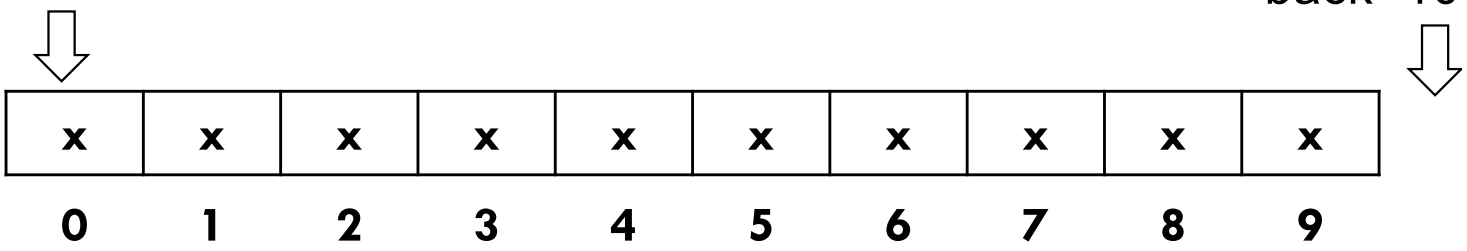
front = back = 0



Effettuiamo n (=10) operazioni enqueue("x")

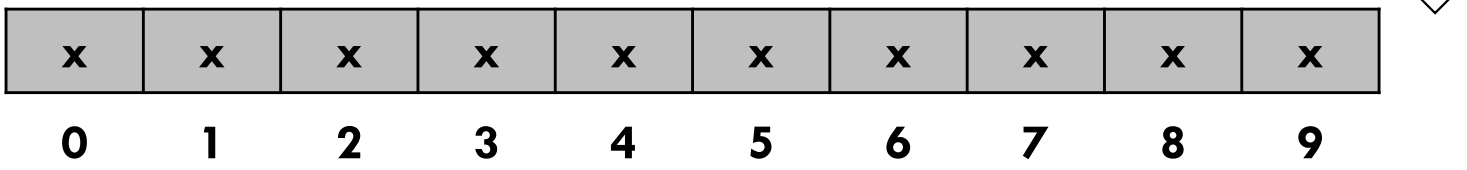
front=0

back=10



Effettuiamo n (=10) operazioni dequeue()

front=back=10

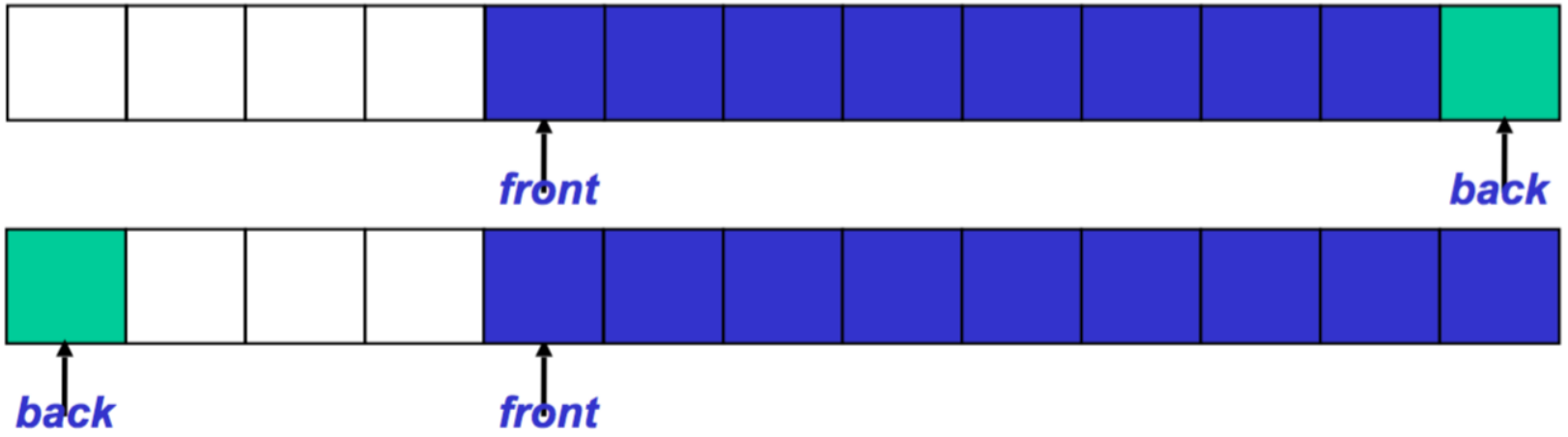


Coda con array circolare

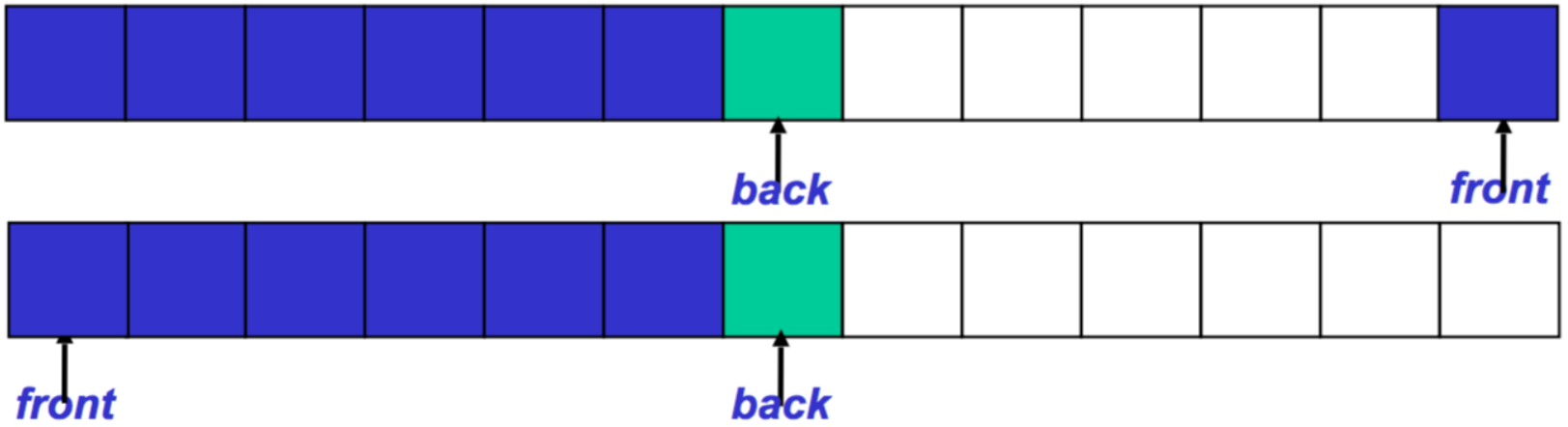
- Per risolvere quest'ultimo problema si usa una tecnica detta "***array circolare***"
  - i due indici, dopo essere giunti alla fine dell'array, possono ritornare all'inizio se si sono liberate delle posizioni
- L'array circolare è pieno quando la coda contiene un numero di oggetti **uguale** a **n–1** (e non **n**).
  - Si "spreca" quindi un elemento dell'array: ciò è necessario per distinguere la condizione di coda vuota (**front==back**) dalla condizione di coda piena
- le prestazioni temporali rimangono identiche

Array circolare

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



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

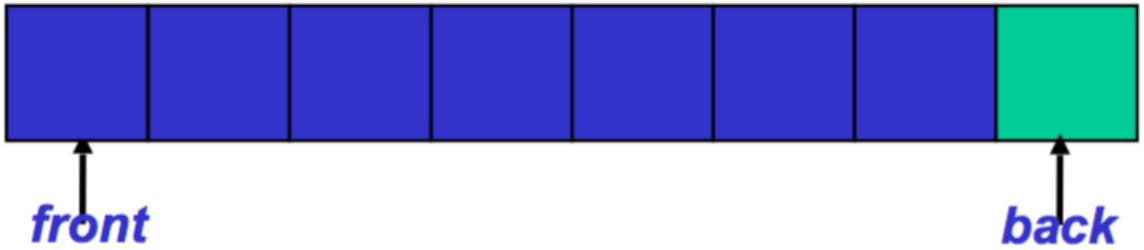


La classe FixedCircularArrayQueue

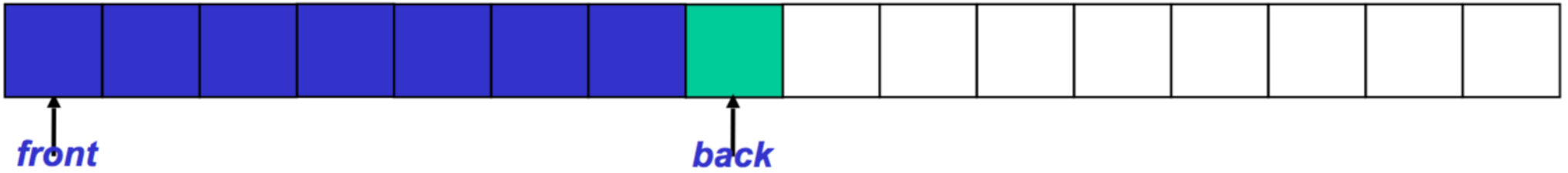
```
public class FixedCircularArrayQueue extends FixedArrayQueue
{  // il metodo increment fa avanzare un indice di una
   // posizione, tornando all'inizio dell'array se si supera
   // la fine
   protected int increment(int index)
   {  return (index + 1) % v.length;
   }
   public void enqueue(Object obj)  // SOVRASCRITTO
   {  if (increment(back) == front)
         throw new FullQueueException();
      v[back] = obj;
      back = increment(back);
   }
   public Object dequeue() // SOVRASCRITTO
   {  Object obj = getFront();
      front = increment(front);
      return obj;
   }
   // non serve sovrascrivere getFront perché non modifica
   // le variabili back e front
}
```

Ridimensionare un array circolare

- Vogliamo estendere **FixedCircularArrayQueue** in maniera tale che l'array contenente i dati possa essere ridimensionato quando la coda è piena
  - Effettuiamo un **resize** come su un array ordinario

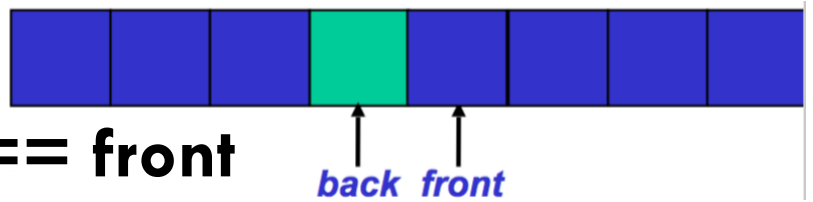


- Se **front = 0** e **back = n-1** la condizione di array pieno equivale alla condizione **increment(back) == front**
  - L'operazione di **resize** ha l'effetto desiderato:

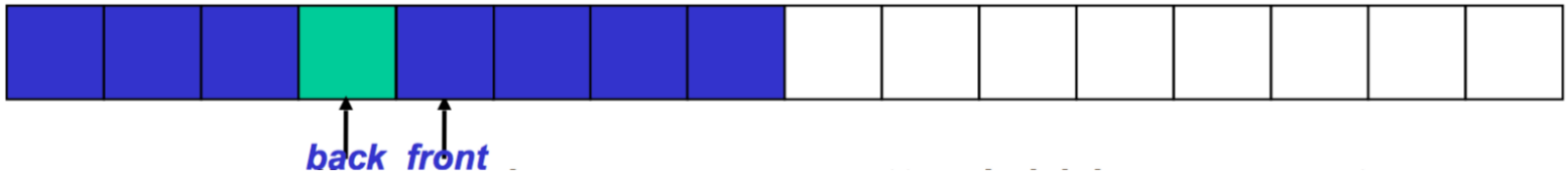


Ridimensionare un array circolare

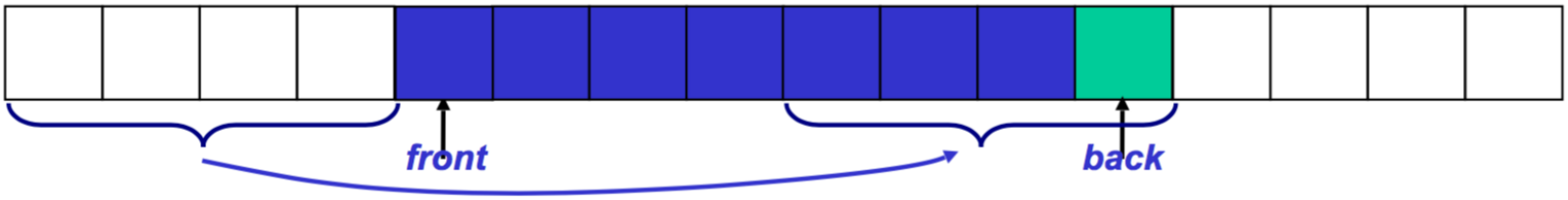
- In generale però la zona utile della coda è **attorno** alla sua fine (ovvero **back < front**): c'è un problema in più
  - La condizione di array pieno equivale sempre a **increment(back) == front**



  - Raddoppiamo la dimensione:



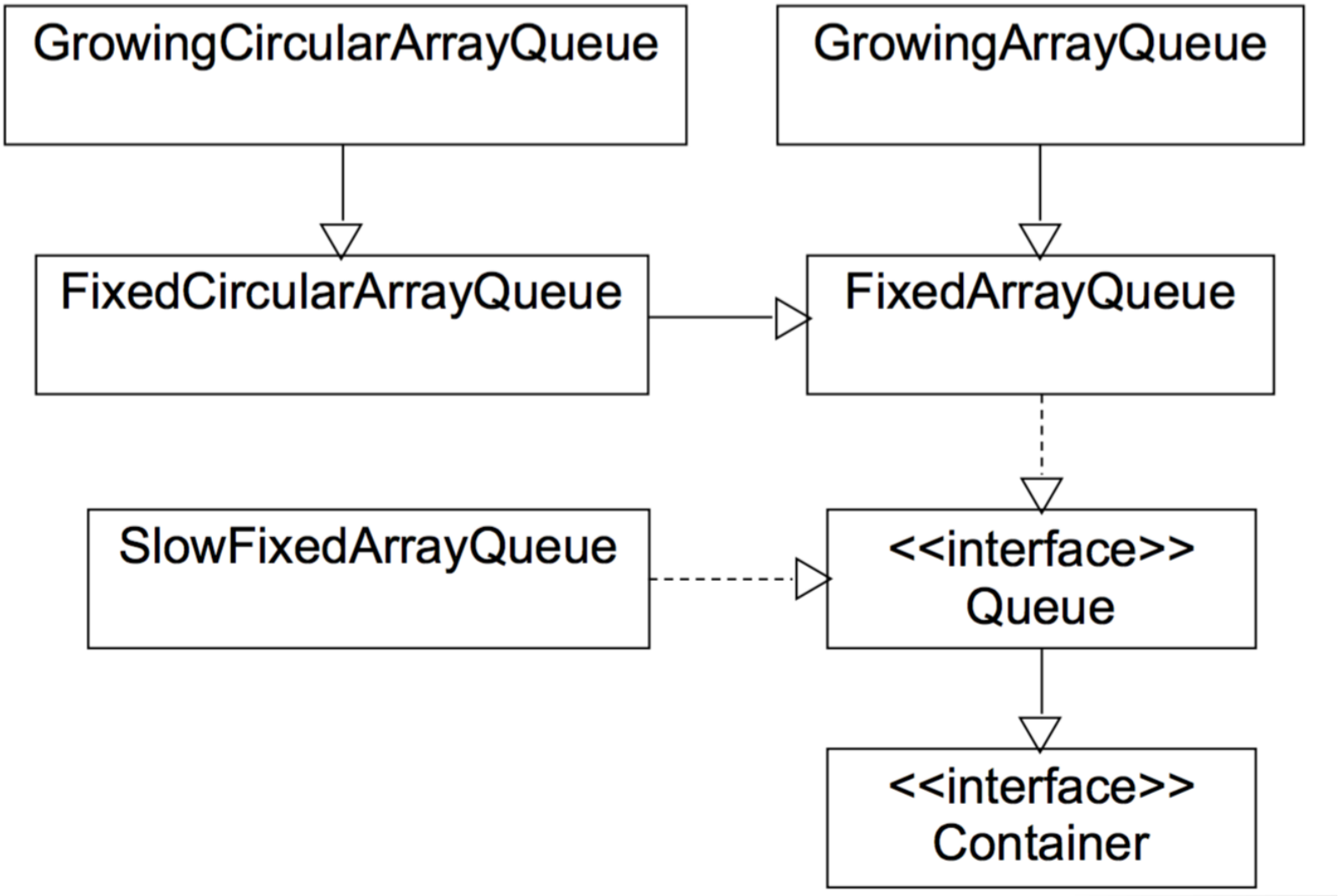
  - Affinché' l'array rimanga compatto dobbiamo spostare nella seconda metà dell'array la prima parte della coda:



La classe GrowingCircularArrayQueue

```
public class GrowingCircularArrayQueue
      extends FixedCircularArrayQueue
{  public void enqueue(Object obj)
   {  if (increment(back) == front)
      {  v = resize(v, 2*v.length);
         // se si ridimensiona l'array e la zona utile
         // della coda si trova attorno alla sua fine,
         // la seconda metà del nuovo array rimane vuota
         // e provoca un malfunzionamento della coda,
         // che si risolve spostandovi la parte della
         // coda che si trova all'inizio dell'array
         if (back < front)
         {  System.arraycopy(v, 0, v, v.length/2, back);
            back += v.length/2;
         }
      }
      super.enqueue(obj);
   }
}
```

Gerarchia di classi e interfacce



Schema riassuntivo delle prestazioni

	slowFixedArray	FixedArray	GrowingArray	FixedCircArray	GrowCircArray
enqueue	O(1)	O(1)	O(1)*	O(1)	O(1)*
getFront	O(1)	O(1)	O(1)	O(1)	O(1)
dequeue	O(n)	O(1)	O(1)	O(1)	O(1)

* Costo medio

Realizzazione di un Insieme (Set)

Insieme (set)

- Il tipo di dati astratto "**insieme**" (**set**) è un contenitore (eventualmente vuoto) di oggetti **distinti** (cioè non contiene duplicati)
  - senza alcun particolare ordinamento
  - senza memoria dell'ordine temporale in cui gli oggetti vengono inseriti od estratti
  - si comporta come un insieme matematico

Set

```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

- Le operazioni consentite sull'insieme sono
  - **inserimento** di un oggetto
    - fallisce silenziosamente se l'oggetto è già presente
  - **verifica della presenza** di un oggetto
  - **ispezione di tutti** gli oggetti
    - metodo che restituisce un array di riferimenti agli oggetti contenuti nell'insieme, senza alcun requisito di ordinamento di tale array (anche perché i dati non sono ordinabili…)

Set

```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

- **Non** esiste un'operazione di **rimozione**
  - si usa la sottrazione tra insiemi

Operazioni sugli insiemi

- Per due insiemi A e B, si definiscono le operazioni
  - **unione**, A ∪ B
    - appartengono all'unione di due insiemi tutti e soli gli oggetti che appartengono ad almeno uno dei due insiemi
  - **intersezione**, A ∩ B
    - appartengono all'intersezione di due insiemi tutti e soli gli oggetti che appartengono ad entrambi gli insiemi
  - **sottrazione**, A - B (oppure anche A \ B)
    - appartengono all'insieme sottrazione A-B tutti e soli gli oggetti che appartengono all'insieme A e non appartengono all'insieme B
    - non è necessario che B sia un sottoinsieme di A

Realizzazione con array non ordinato

- Quando si scrive una classe, è comodo scrivere prima le firme di tutti i metodi, con un corpo "***vuoto***"

```
public class ArraySet implements Set
{  public void makeEmpty() { }
   public boolean isEmpty() { return true; }
   public Object[] toArray() { return null; }
   public boolean contains(Object x) { return false; }
   public void add(Object x) { }
}
```

- Invece di compilare tutto alla fine, si compila ogni volta che si scrive il corpo di un metodo
  - in questo modo, si evita di trovarsi nella situazione in cui il compilatore segnala molti errori

Insieme con array non ordinato

```
public class ArraySet implements Set
{
   private final static int INITIAL_CAPACITY = 1;
   private Object[] v;
   private int vSize;

   public ArraySet()
   {  v = new Object[INITIAL_CAPACITY];
      makeEmpty();
   }

   public void makeEmpty(){ vSize = 0; }
   public boolean isEmpty() { return vSize == 0; }
   public int size(){ return vSize;}

   public Object[] toArray()      // O(n)
   {  Object[] x = new Object[vSize];
      System.arraycopy(v, 0, x, 0, vSize);
      return x;
   }
   . . .
}
```

Insieme con array non ordinato

```
public class ArraySet implements Set
{
   . . .
   public boolean contains(Object x)   // O(n)
   {  for (int i = 0; i < vSize; i++)
         if (v[i].equals(x)) return true;
      return false;
   }

   public void add(Object x) // O(n)  (usa contains)
   {
      if (contains(x)) return; // esce silenziosamente
      if (vSize == v.length)
         v = resize(v, 2*vSize);
      v[vSize++] = x;
   }

   private static Object[] resize(Object[] v, int n)
   { ... }
}
```

Operazioni su insiemi: unione

```
public static Set union(Set s1, Set s2)
{   Set x = new ArraySet();
    // inseriamo gli elementi del primo insieme
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    // inseriamo tutti gli elementi del
    // secondo insieme, sfruttando le
    // proprietà di add (niente duplicati...)
    v = s2.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    return x;
}
```

- Se **contains** è **O(n)** (e quindi lo è anche **add**) questa operazione è **$O(n^2)$**

Operazioni su insiemi: intersezione

```
public static Set intersection(Set s1, Set s2)
{
    Set x = new ArraySet();

    Object[] v = s1.toArray();

    for (int i = 0; i < v.length; i++){
        if (s2.contains(v[i])){    // O(n)
            x.add(v[i]);
        }
    }
    return x;
}                                          // O(n^2)
```

Operazioni su insiemi: sottrazione

```
public static Set subtract(Set s1, Set s2)
{
   Set x = new ArraySet();

   Object[] v = s1.toArray();

   for (int i = 0; i < v.length; i++){
      if (!s2.contains(v[i])){      // O(n)
         x.add(v[i]);
      }
   }

   return x;
}                                    // O(n²)
```

Set con array non ordinato

- Riassumendo, realizzando un Insieme (Set) con un array non ordinato
  - le prestazioni di tutte le primitive dell'insieme sono **O(n)**
  - le prestazioni di tutte le operazioni che agiscono su due insiemi sono **$O(n^2)$**

Insieme di dati ordinabili

- Realizziamo l'interfaccia **SortedSet** usando un array ordinato
  - dovremo definire due metodi **add()**, uno dei quali ***impedisce l'inserimento di dati non ordinabili***

```
public interface SortedSet extends Set
{   void add(Comparable obj);
    Comparable[] toSortedArray();
}
```

```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

Insieme con array ordinato

```
public class ArraySortedSet implements SortedSet
{
   private static final int INITIAL_CAPACITY = 1;
   private Comparable[] v;
   private int vSize;

   public ArraySortedSet()
   {  v = new Comparable[INITIAL_CAPACITY];
      makeEmpty();
   }

   public void makeEmpty(){  vSize = 0; }

   public boolean isEmpty(){  return vSize == 0; }

...

}
```

Insieme con array ordinato

```
public class ArraySortedSet implements SortedSet
{
    ...

    public Comparable[] toSortedArray() // O(n)
    {  Comparable[] x = new Comparable[vSize];
       System.arraycopy(v, 0, x, 0, vSize);
       return x;
    }

    public Object[] toArray()
    {  return toSortedArray();
    }

    ...
}
```

Insieme con array ordinato

```
public class ArraySortedSet implements SortedSet
{   ...
    public boolean contains(Object x)  // O(log n)
    {   // si può fare una ricerca binaria
        // qui ci va il codice ...
    }
    public void add(Object x) // non deve essere usato!
    {   throw new IllegalArgumentException(); }

    public void add(Comparable x)       // O(n)
    {   if (contains(x))
            return;
        if (vSize == v.length)
            v = resize(v, 2*vSize);
        v[vSize++] = x;
        // usiamo insertion sort che è O(n)
        // perché inseriamo in un array ordinato
        // qui ci va il codice ...
    }
}
```

Operazioni su insiemi ordinati

- Gli algoritmi già visti per le operazioni sugli insiemi generici possono essere utilizzati senza alcuna modifica anche per l'insieme ordinato, realizzato con un array ordinato
  - infatti, un **SortedSet** è anche un **Set**
  - la complessità di tutti gli algoritmi (unione, intersezione e sottrazione) rimane $O(n^2)$, perché il metodo **add** è rimasto $O(n)$
- Senza sfruttare le informazioni sull'ordinamento dell'insieme, non è possibile ottenere prestazioni migliori...

Operazioni su insiemi ordinati

- Usare l'incapsulamento ad oltranza può essere sconveniente…

- In questo caso, sapendo che
  - l'array ottenuto con il metodo **toSortedArray** è ordinato
  - l'inserimento nell'insieme avviene nel metodo **add()** con l'algoritmo di ordinamento per inserzione in un array ordinato

  è possibile scrivere versioni più efficienti dei metodi già visti

Operazioni su insiemi ordinati: unione

- Abbiamo due insiemi rappresentati come array ordinati… vogliamo ottenere un array ordinato che contenga gli elementi di entrambi

- Ci ricorda qualcosa?

Operazioni su insiemi: unione

- Per realizzare l'***unione***, osserviamo che il problema è molto simile alla ***fusione di due array ordinati***
  - come abbiamo visto in **MergeSort**, questo algoritmo di fusione è **O(n)**

- L'unica differenza consiste nella contemporanea eliminazione (cioè nel non inserimento…) di eventuali oggetti duplicati
  - un oggetto presente in entrambi gli insiemi dovrà essere presente una sola volta nell'insieme unione

Operazioni su insiemi: unione

- Effettuando la fusione dei due array ordinati secondo l'algoritmo visto in **MergeSort**, gli oggetti vengono via via inseriti nell'insieme unione che si va costruendo

- Se gli inserimenti avvengono con oggetti che risultano già ***in ordine crescente***, l'ordinamento per inserzione in un array ordinato che viene usato dal metodo **add()** ha prestazioni **O(1)** per ogni inserimento (l'elemento è sempre aggiunto in ultima posizione)!
  - il metodo **add()** ha quindi prestazioni **O(log n)**
    - perché invoca **contains()** che è **O(log n)** (ricerca in un array ordinato)
  - se gli inserimenti non avvengono con oggetti in ordine crescente, il metodo **add()** ha prestazioni medie **O(n)**

Operazioni su insiemi: unione

```
public static SortedSet union(SortedSet s1,SortedSet s2)
{
   SortedSet x = new ArraySortedSet();
   Comparable[] v1 = s1.toSortedArray();
   Comparable[] v2 = s2.toSortedArray();

   int i = 0, j = 0;
   while (i < v1.length && j < v2.length)
   {
      if (v1[i].compareTo(v2[j]) < 0)
         x.add(v1[i++]);    // contains:O(log n); ins:O(1)

      else if (v1[i].compareTo(v2[j]) > 0)
               x.add(v2[j++]);
            else // sono uguali: avanzo entrambi gli indici
            {  x.add(v1[i++]);
               j++;
            }
   }
  . . . // continua
} // prestazioni O(n log n) anziché quadratiche
```

Operazioni su insiemi: unione

```
public static SortedSet union(SortedSet s1, SortedSet s2)
{
    . . .
   while (i < v1.length)
      x.add(v1[i++]);
   while (j < v2.length)
      x.add(v2[j++]);
   return x;

} // prestazioni O(n log n) anziché quadratiche
```

Operazioni su insiemi: intersezione

```
public static SortedSet intersection(SortedSet s1,
                                     SortedSet s2)
{  SortedSet x = new ArraySortedSet();
   Comparable[] v1 = s1.toSortedArray();
   Comparable[] v2 = s2.toSortedArray();

   for (int i = 0, j = 0; i < v1.length; i++)
   {
      while (j < v2.length && v1[i].compareTo(v2[j]) > 0){
         j++;
      }
      if (j == v2.length){
         break;
      }
      if (v1[i].compareTo(v2[j]) == 0)
      {  x.add(v1[i]); j++;}
   }
   return x;
} // prestazioni O(n log n) anziché quadratiche
```

Operazioni su insiemi: sottrazione

```
public static SortedSet subtract(SortedSet s1, SortedSet s2)
{  SortedSet x = new ArraySortedSet();
   Comparable[] v1 = s1.toSortedArray();
   Comparable[] v2 = s2.toSortedArray();
   int i,j;
   for (i = 0, j = 0; i < v1.length; i++)
   {  while (j < v2.length && v1[i].compareTo(v2[j]) > 0){
         j++;
      }
      if (j == v2.length){
         break;
      }
      if (v1[i].compareTo(v2[j]) != 0){
         x.add(v1[i]);
      }
   }
   while (i < v1.length){
      x.add(v1[i++]);
   }
   return x;
} // prestazioni O(n log n) anziché quadratiche
```

ADT Tabella
(Table)

Chiavi numeriche

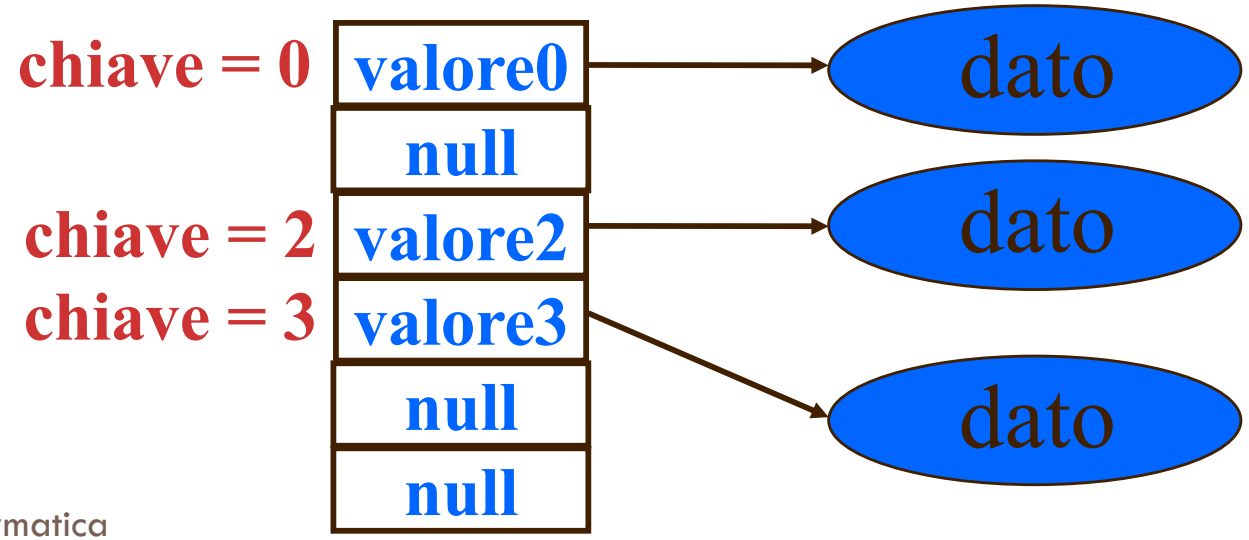
- Imponiamo una restrizione al campo di applicazione di una mappa
  - Supponiamo che le chiavi siano **numeri interi** in un intervallo noto a priori

  allora si può realizzare una mappa con

  prestazioni **O(1)** per tutte le operazioni
- Si usa un array che contiene soltanto i riferimenti ai valori, usando **le chiavi come indici nell'array**
  - Celle dell'array che hanno come indice una chiave che non appartiene alla mappa hanno valore **null**

Tabella

- Una mappa con chiavi numeriche intere viene detta **tabella** (**table**)
- L'analogia è la seguente
  - un valore è una riga nella tabella
  - le righe sono numerate usando le chiavi
  - alcune righe possono essere vuote (senza valore)



Tabella

- Definiamo il tipo di dati astratto **Table** con un comportamento identico alla mappa
  - l'unica sostanziale differenza è che le chiavi non sono riferimenti ad oggetti di tipo **Comparable**, ma sono **numeri interi** (che evidentemente sono confrontabili)

```
public interface Table extends Container
{   void insert(int key, Object value);
    void remove(int key);
    Object find(int key);
}
```

Realizzazione di una Table

- Realizziamo una classe che implementa l'interfaccia Table
  - Utilizziamo un array
  - Fissiamo la taglia a 100 elementi

Tabella

```
public class ArrayTable100 implements Table
{  private Object[] v;
   private int count; // count rende isEmpty O(1)

   public ArrayTable100()
   {  v = new Object[100];
      makeEmpty();
   }
   public void makeEmpty()
   {  count = 0;
      for(int i=0; i < v.length; i++)
      {  v[i] = null; }
   }
   public boolean isEmpty()
   {  return (count == 0);}

   private void check(int key)
   {  if (key < 0 || key >= v.length)
        throw new InvalidPositionTableException();
   }
   ...
}
```

```
public class ArrayTable100 implements Table
{  ...
   public void insert(int key, Object value)
   {  check(key);
      if (v[key] == null)
      {  count++;
         v[key] = value;
      }
      else v[key] = value;
   }
   public void remove(int key)
   {  check(key);
      if (v[key] != null)
      {  count--;
         v[key] = null;
      }
   }
   public Object find(int key)
   {  check(key);
      return v[key];
   }
}
```

Tabella

- La tabella non utilizza la memoria in modo efficiente
  - l'occupazione di memoria richiesta per contenere n dati non dipende da n in modo lineare come invece avviene per tutti gli altri ADT) ma dipende dal contenuto informativo presente nei dati
    - in particolare, dal valore della chiave massima
- Può essere necessario un array di milioni di elementi per contenere poche decine di dati
  - si definisce **fattore di riempimento** (**load factor**) della tabella il numero di dati contenuti nella tabella diviso per la dimensione della tabella stessa

Tabella

- La tabella è un dizionario con prestazioni ottime
  - tutte le operazioni sono **O(1)**
- ma con le seguenti limitazioni
  - le **chiavi** devono essere **numeri interi** (non negativi)
    - in realtà si possono usare anche chiavi negative, sottraendo ad ogni chiave il valore dell'estremo inferiore dell'intervallo di variabilità
  - **l'intervallo di variabilità delle chiavi deve essere noto a priori**
    - per dimensionare la tabella (ed avere inserimento O(1))
  - se il fattore di riempimento è molto basso, si ha un grande **spreco di memoria**
    - ciò avviene se le chiavi sono molto "disperse" nel loro insieme di variabilità

Tabella con array ridimensionabile

- La tabella potrebbe avere **dimensione variabile**, cioè utilizzare un **array di dimensione crescente** quando sia necessario:
  - l'operazione di **inserimento** richiede, però, un tempo **O(n)** ogni volta che è necessario un ridimensionamento
  - in questo caso non si può utilizzare l'analisi ammortizzata perché non si può prevedere quali siano le posizioni richieste dall'utente
  - non è più vero che il ridimensionamento avviene "una volta ogni tanto", può avvenire anche tutte le volte
- le prestazioni nel caso peggiore sono quindi **O(n)**