Programación II. Algoritmos y Estructuras de datos II. 2024

Programación II. Algoritmos y EDD II. 2024

Agenda 25 de Marzo 2024:

- TDA COLA. Especificación
  - Implementación estructura estática
- TDA COLA CON PRIORIDAD. Especificación
  - Implementación estructura estática
- TDA CONJUNTO. Especificación
  - Implementación estructura estática

TDA Cola

TDA COLA

Una *cola* es otra estructura habitual en la vida real. Por ejemplo, cuando se espera ante una ventanilla para ser atendidos. Normalmente, la primera persona que llegó es la primera que será atendida.

Una cola nos permite almacenar datos. Sólo podemos ingresar y extraer un dato por vez. Los datos se ordenan por su orden de llegada: el primer dato accesible es siempre el primero que entró.

TDA COLA

Una *cola* es otra estructura habitual en la vida real. Por ejemplo, cuando se espera ante una ventanilla para ser atendidos. Normalmente, la primera persona que llegó es la primera que será atendida.

Una cola nos permite almacenar datos. Sólo podemos ingresar y extraer un dato por vez. Los datos se ordenan por su orden de llegada: el primer dato accesible es siempre el primero que entró.

Las operaciones que necesitaremos son: agregar y eliminar datos de la cola (que llamaremos posteriormente *Acolar* y *Desacolar*), consultar el valor del primer elemento (que llamaremos *Primero*) y consultar si la cola está o no vacía (a esta consulta la llamaremos *ColaVacia*.)

TDA COLA

Una *cola* es otra estructura habitual en la vida real. Por ejemplo, cuando se espera ante una ventanilla para ser atendidos. Normalmente, la primera persona que llegó es la primera que será atendida.

Una cola nos permite almacenar datos. Sólo podemos ingresar y extraer un dato por vez. Los datos se ordenan por su orden de llegada: el primer dato accesible es siempre el primero que entró.

Las operaciones que necesitaremos son: agregar y eliminar datos de la cola (que llamaremos posteriormente *Acolar* y *Desacolar*), consultar el valor del primer elemento (que llamaremos *Primero*) y consultar si la cola está o no vacía (a esta consulta la llamaremos *ColaVacia*.)

A estas operaciones agregaremos la inicialización de una cola, que llamaremos *InicializarCola*.

TDA COLA

Una *cola* es otra estructura habitual en la vida real. Por ejemplo, cuando se espera ante una ventanilla para ser atendidos. Normalmente, la primera persona que llegó es la primera que será atendida.

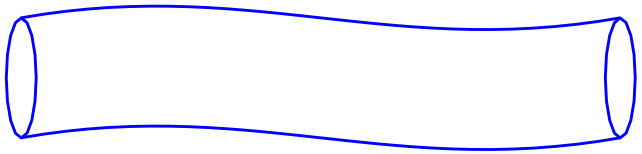
Una cola nos permite almacenar datos. Sólo podemos ingresar y extraer un dato por vez. Los datos se ordenan por su orden de llegada: el primer dato accesible es siempre el primero que entró.

Las operaciones que necesitaremos son: agregar y eliminar datos de la cola (que llamaremos posteriormente *Acolar* y *Desacolar*), consultar el valor del primer elemento (que llamaremos *Primero*) y consultar si la cola está o no vacía (a esta consulta la llamaremos *ColaVacia*.)

A estas operaciones agregaremos la inicialización de una cola, que llamaremos *InicializarCola*.

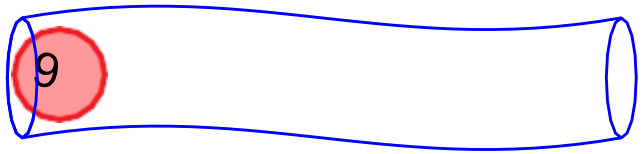
Una cola es lo que se suele llamar una estructura FIFO (del inglés First In, First Out.)

TDA COLA



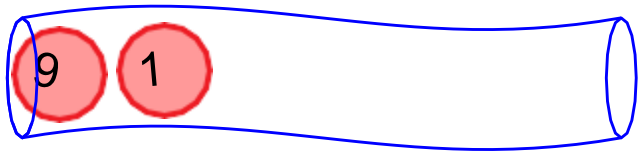
ColaVacia() = true

TDA COLA



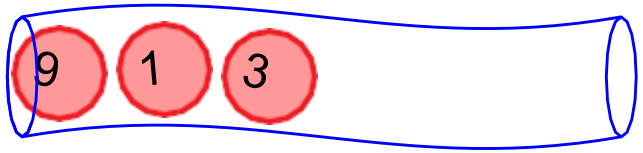

```
Acolar(9)
ColaVacia() = false
Primero() = 9
```

TDA COLA



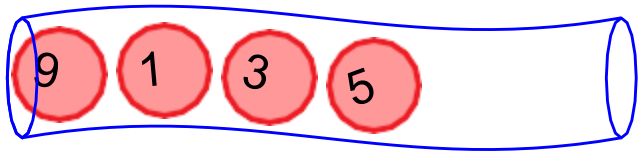

```
Acolar(1)
ColaVacia() = false
Primero() = 9
```

TDA COLA



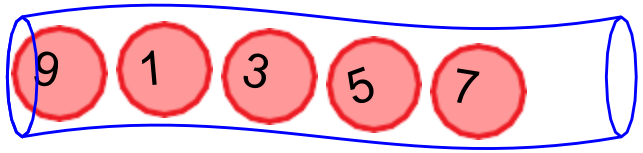

```
Acolar(3)
ColaVacia() = false
Primero() = 9
```

TDA COLA



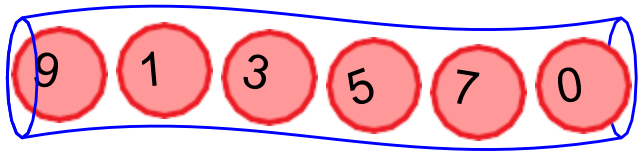

```
Acolar(5)
ColaVacia() = false
Primero() = 9
```

TDA COLA



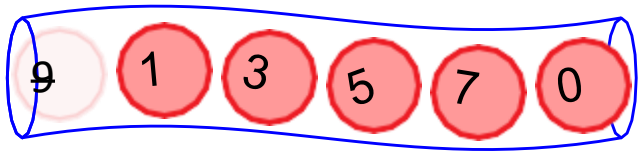

```
Acolar(7)
ColaVacia() = false
Primero() = 9
```

TDA COLA




```
Acolar(0)
ColaVacia() = false
Primero() = 9
```

TDA COLA




```
Desacolar()
ColaVacia() = false
Primero() = 1
```

TDA COLA. Especificación

Recordemos que una cola permite almacenar un conjunto de valores, recuperarlos y eliminarlos. Su particularidad es que el elemento que se recupera o se elimina es siempre el más el primero que ingresó (es una estructura FIFO.)

Adicionalmente, podemos saber cuál es el elemento más reciente y si la cola está vacía o no.

TDA COLA. Especificación

Recordemos que una cola permite almacenar un conjunto de valores, recuperarlos y eliminarlos. Su particularidad es que el elemento que se recupera o se elimina es siempre el más el primero que ingresó (es una estructura FIFO.)

Adicionalmente, podemos saber cuál es el elemento más reciente y si la cola está vacía o no.

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

TDA COLA. Especificación

Recordemos que una cola permite almacenar un conjunto de valores, recuperarlos y eliminarlos. Su particularidad es que el elemento que se recupera o se elimina es siempre el más el primero que ingresó (es una estructura FIFO.)

Adicionalmente, podemos saber cuál es el elemento más reciente y si la cola está vacía o no.

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

Acolar, que permite agregar un elemento a una cola inicializada.

TDA COLA. Especificación

Recordemos que una cola permite almacenar un conjunto de valores, recuperarlos y eliminarlos. Su particularidad es que el elemento que se recupera o se elimina es siempre el más el primero que ingresó (es una estructura FIFO.)

Adicionalmente, podemos saber cuál es el elemento más reciente y si la cola está vacía o no.

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

Acolar , que permite agregar un elemento a una cola inicializada.

Desacolar , que permite eliminar el ~~último~~ primero elemento que ingresó a una cola inicializada y no vacía.

TDA COLA. Especificación

Recordemos que una cola permite almacenar un conjunto de valores, recuperarlos y eliminarlos. Su particularidad es que el elemento que se recupera o se elimina es siempre el más el primero que ingresó (es una estructura FIFO.)

Adicionalmente, podemos saber cuál es el elemento más reciente y si la cola está vacía o no.

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

Acolar, que permite agregar un elemento a una cola inicializada.

Desacolar, que permite eliminar el último elemento que ingresó a una cola inicializada y no vacía.

Primero, que permite conocer el último elemento que ingresó a una cola inicializada y no vacía.

primer

TDA COLA. Especificación

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

Acolar, que permite agregar un elemento a una cola inicializada.

Desacolar, que permite eliminar el último elemento que ingresó a una cola inicializada y no vacía.

Primero, que permite conocer el último elemento que ingresó a una cola inicializada y no vacía.

ColaVacia, que permite saber si una cola inicializada está vacía o no.

TDA COLA. Especificación

Los métodos son (razonando en término de constructoras básicas, observadoras y modificadoras):

Acolar, que permite agregar un elemento a una cola inicializada.

Desacolar, que permite eliminar el último elemento que ingresó a una cola inicializada y no vacía.

Primero, que permite conocer el último elemento que ingresó a una cola inicializada y no vacía.

ColaVacia, que permite saber si una cola inicializada está vacía o no.

InicializarCola, que permite inicializar la estructura de la cola.

TDA COLA. Interfaz

```
1.  public interface ColaTDA; {
2.      void  InicializarCola();          // sin precondiciones
3.      void  Acolar(int x);// cola inicializada
4.      void  Desacolar();// cola inicializada y no vacía
5.      boolean  ColaVacia();          // cola inicializada
6.      int Primero();     // cola inicializada y no vacía
7.  }
```

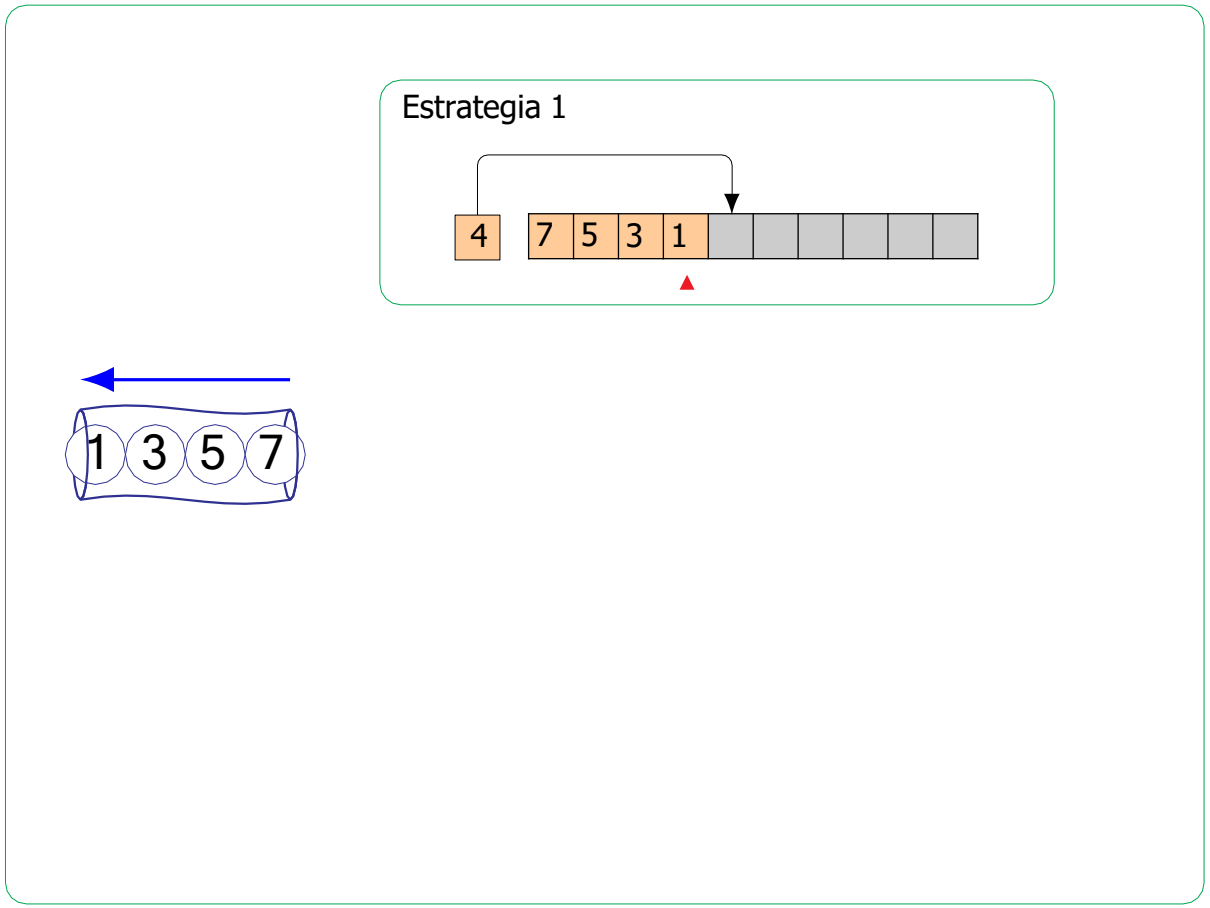
Ejemplo: pasaje de los elementos de una cola *Origen* a otra cola *Destino*.

```
1.  public void PasarCola (ColaTDA Origen, ColaTDA Destino) {
2.      while (!Origen.ColaVacia()) {
3.          Destino.Acolar(Origen.Primero());
4.          Origen.Desacolar();
5.      }
6.  }
```

Estrategias de implementación

- Primera: se tienen un **arreglo** y una **variable** entera, que indica la cantidad de elementos de la cola. Cuando se agrega un nuevo elemento, se lo coloca en la posición inicial del arreglo después de haber desplazado los restantes elementos hacia la derecha. Luego de esto, la variable y se incrementa en uno; cuando se elimina un elemento, se decrementa simplemente la variable en uno.

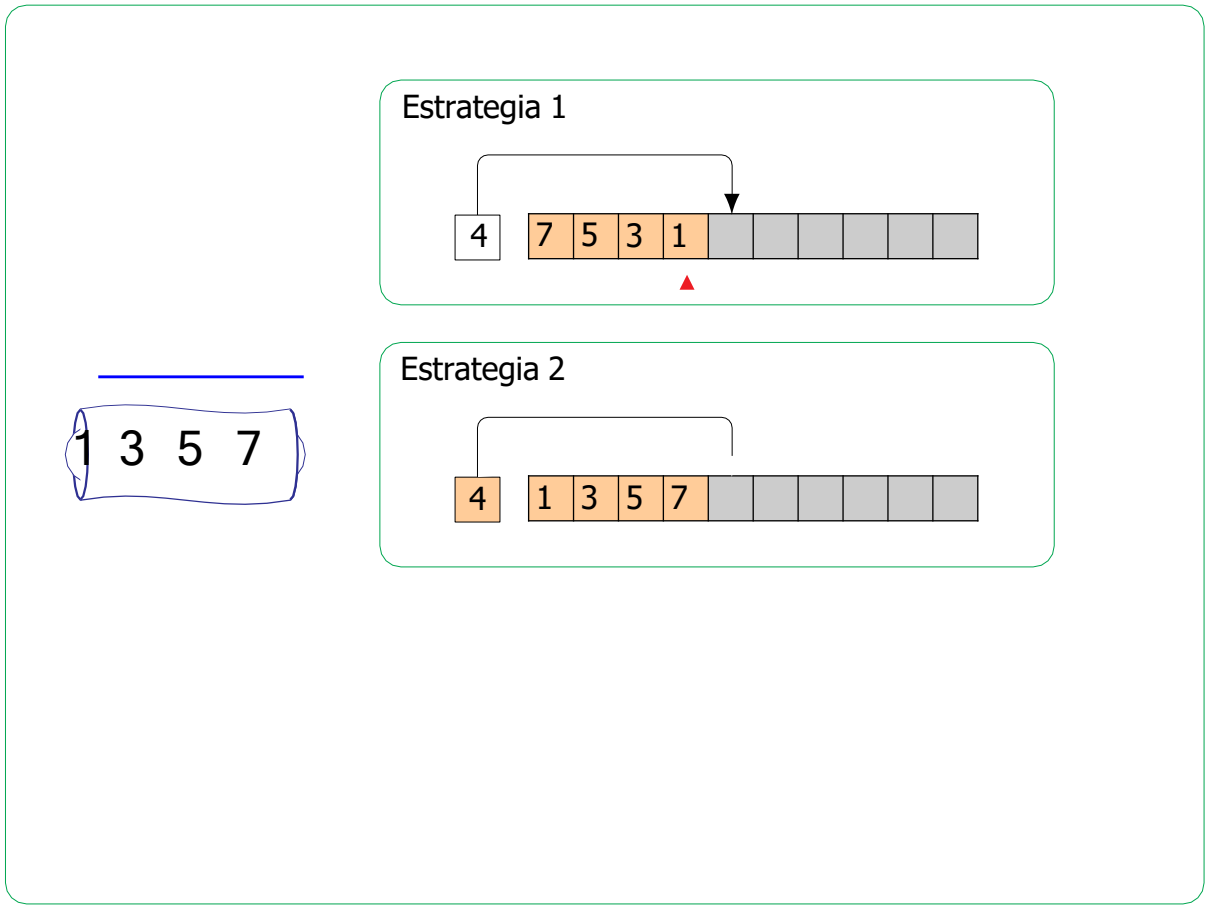
Las representaciones de una cola. Estrategia 1



Estrategias de implementación

- Primera: se tienen un **arreglo** y una **variable** entera, que indica la cantidad de elementos de la cola. Cuando se **agrega** un nuevo elemento, se lo coloca en la posición inicial del arreglo después de haber desplazado los restantes elementos hacia la derecha. Luego de esto, la variable y se incrementa en uno; cuando se **elimina** un elemento, se decrementa simplemente la variable en uno.
- Segunda, se tienen también un **arreglo** y una **variable** entera, que indica la cantidad de elementos de la cola. Cuando se **agrega** un nuevo elemento, se lo coloca en la posición indicada por la variable y ésta se incrementa en uno. Para **eliminar** un elemento, se elimina el de la primera posición y se desplazan todos los otros elementos hacia la izquierda una posición. La variable se decrementa en uno.

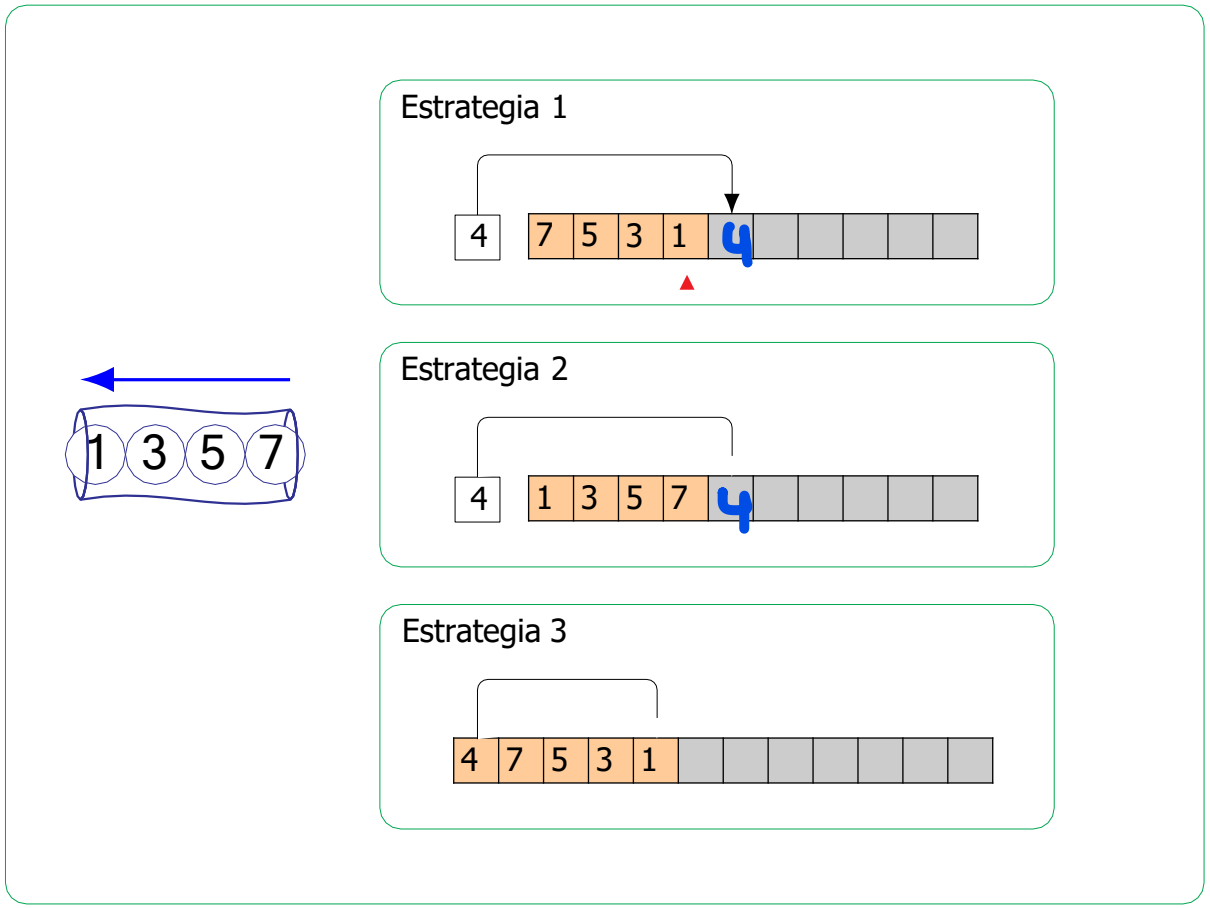
Las representaciones de una cola. Estrategia 2



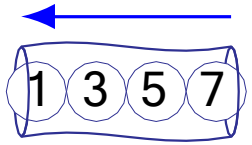
Estrategias de implementación

- Primera, se tienen un **arreglo** y una **variable** entera, que indica la cantidad de elementos de la cola. Cuando se **agrega** un nuevo elemento, se lo coloca en la posición inicial del arreglo después de haber desplazado los restantes elementos hacia la derecha. Luego de esto, la variable y se incrementa en uno; cuando se **elimina** un elemento, se decrementa simplemente la variable en uno.
- Segunda, se tienen también un **arreglo** y una **variable** entera, que indica la cantidad de elementos de la cola. Cuando se **agrega** un nuevo elemento, se lo coloca en la posición indicada por la variable y ésta se incrementa en uno. Para **eliminar** un elemento, se elimina el de la primera posición y se desplazan todos los otros elementos hacia la izquierda una posición. La variable se decrementa en uno.
- Tercera, similar a la primera, pero en lugar de utilizar una variable entera separada, se utiliza la primera posición del arreglo para indicar la cantidad de elementos.

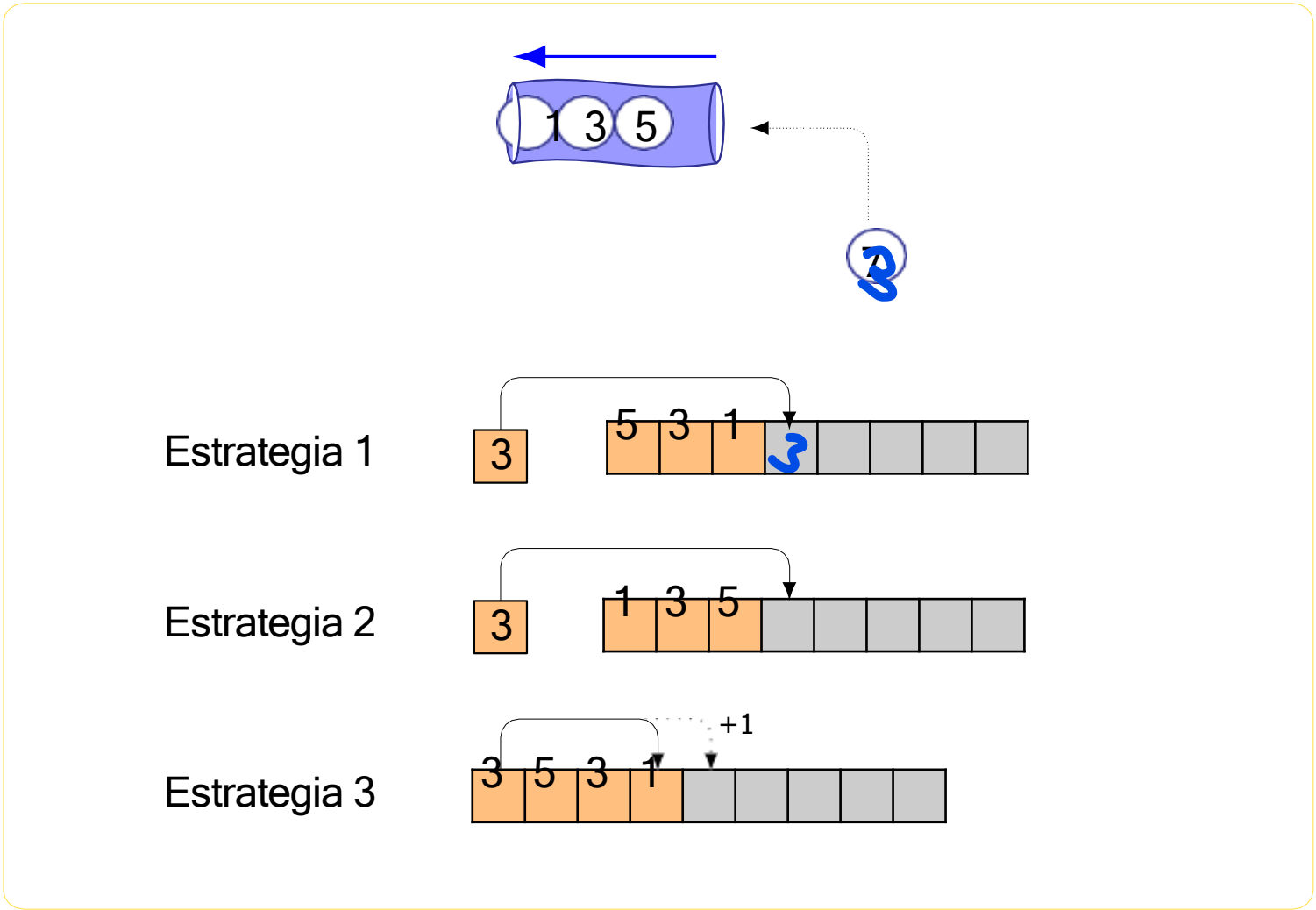
Las representaciones de una cola. Estrategia 2



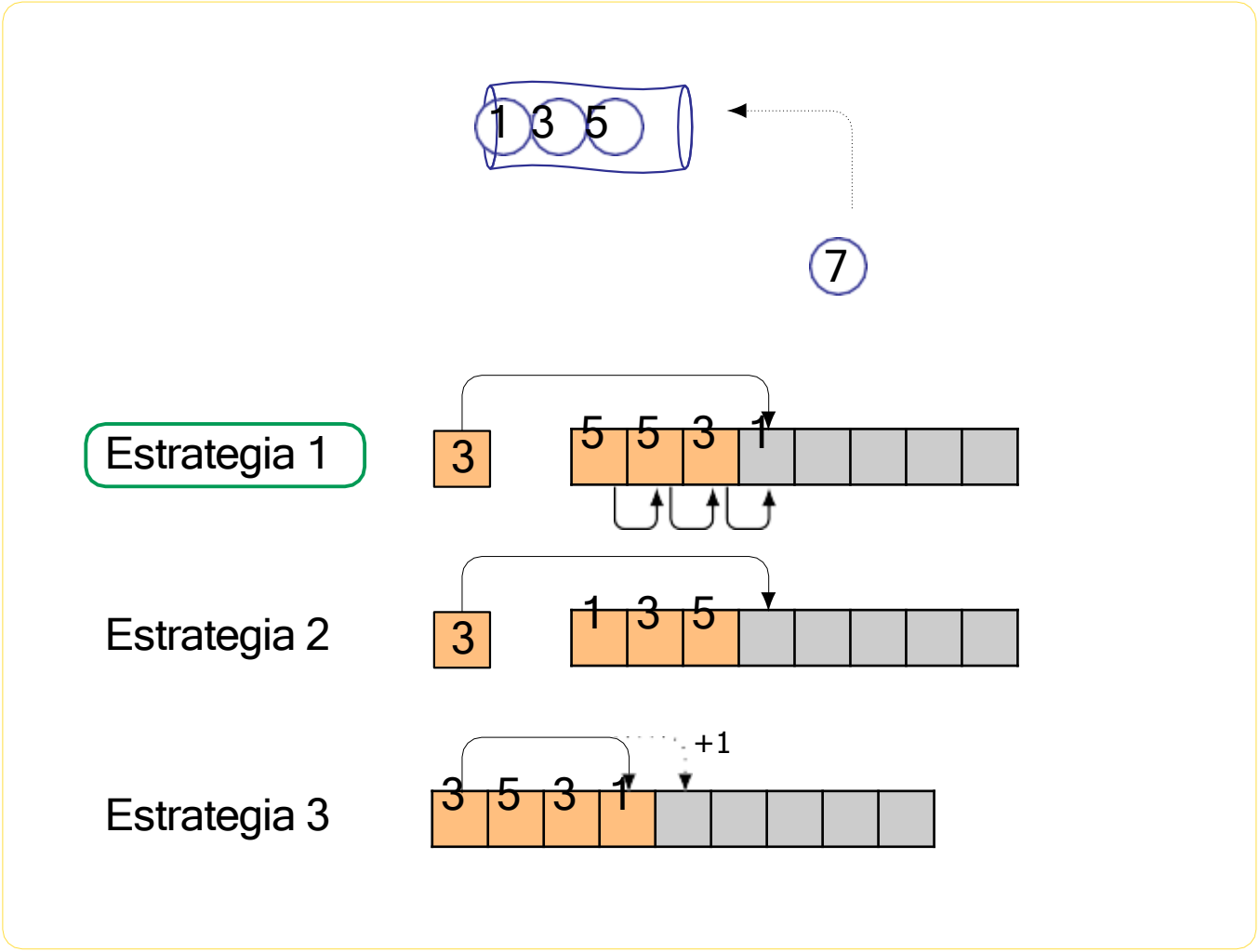
Las representaciones de una cola. Estrategias



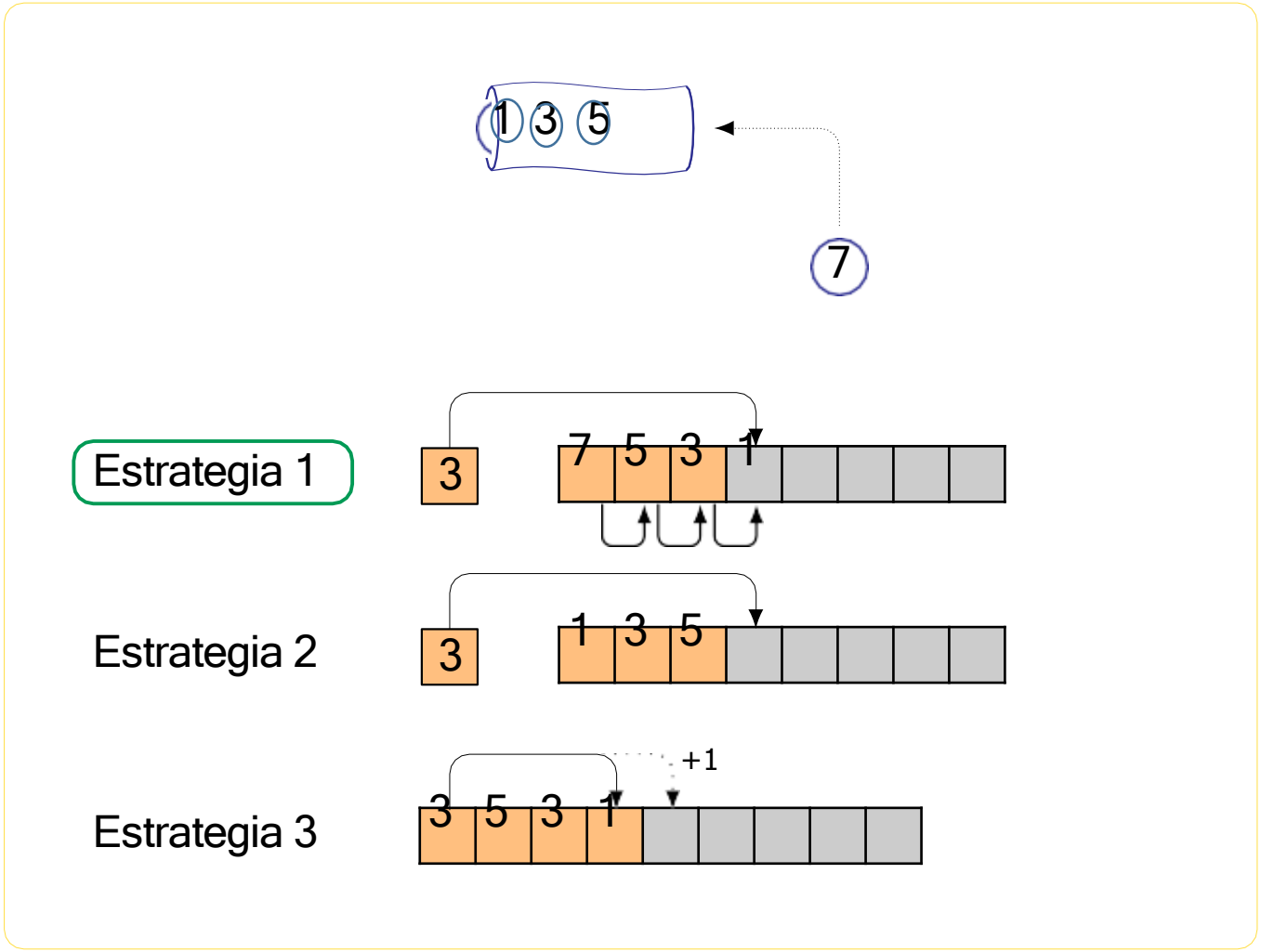
Las representaciones de una cola. Estrategias Acolar



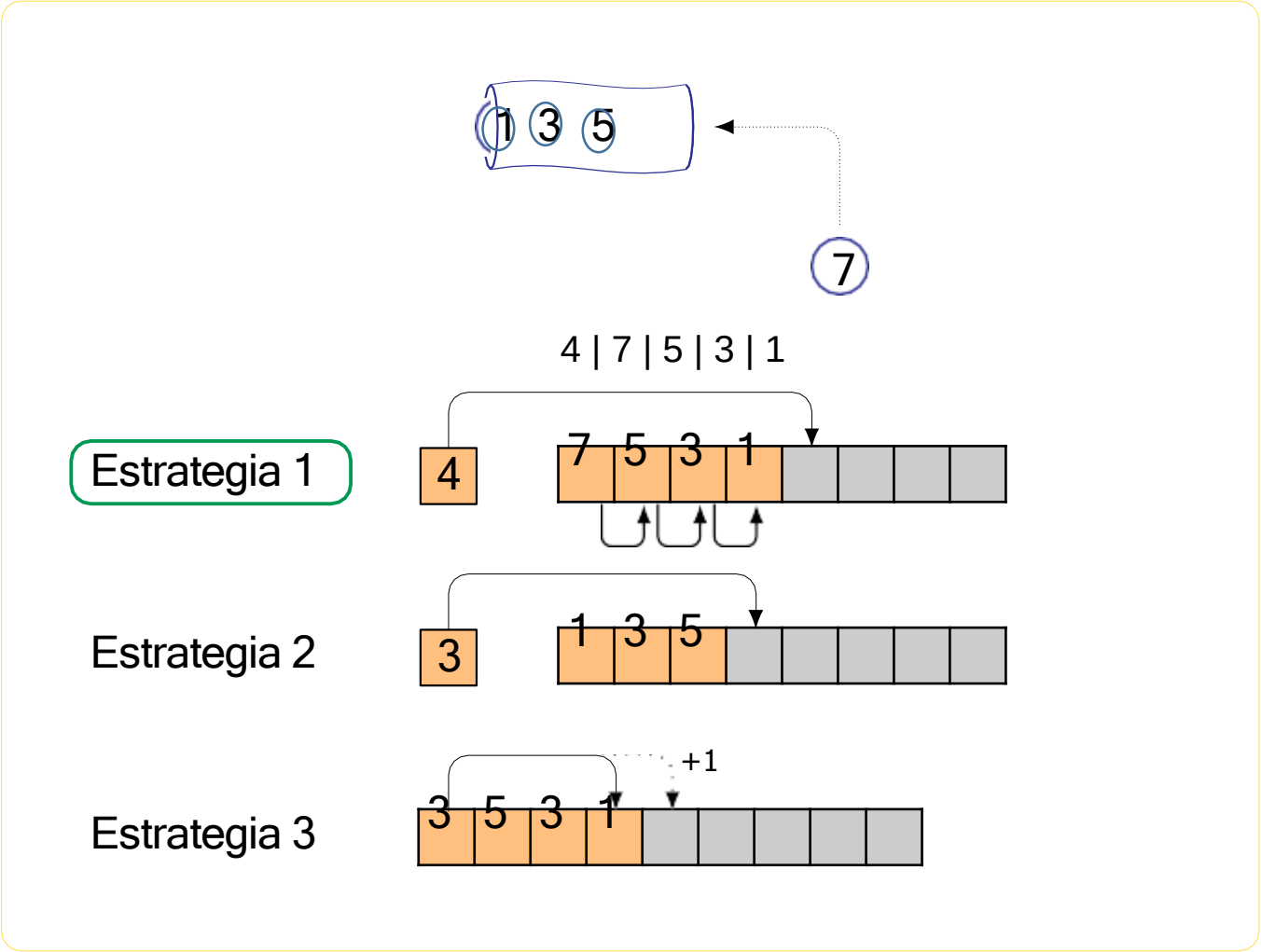
Las representaciones de una cola. Estrategias Acolar



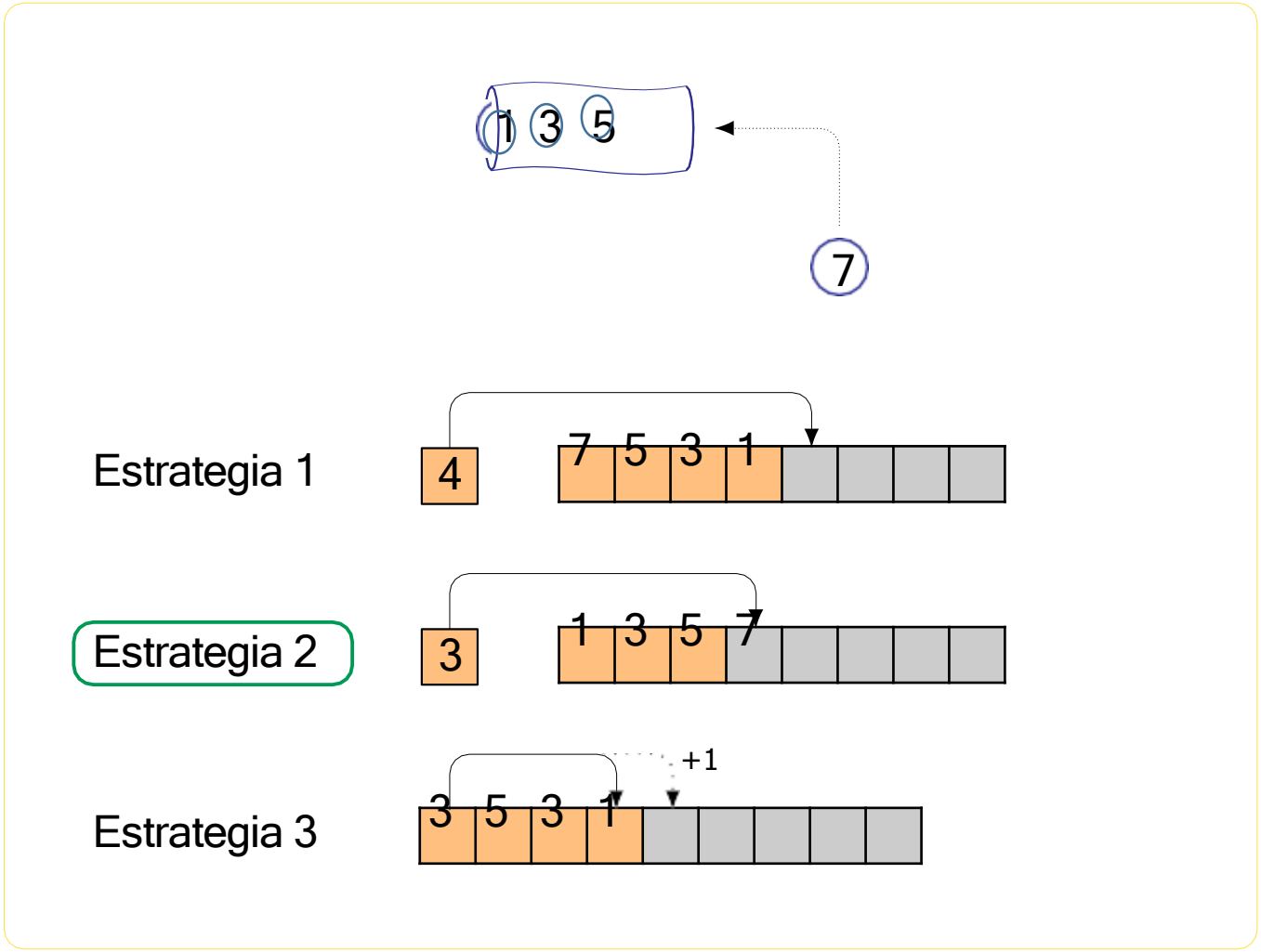
Las representaciones de una cola. Estrategias Acolar



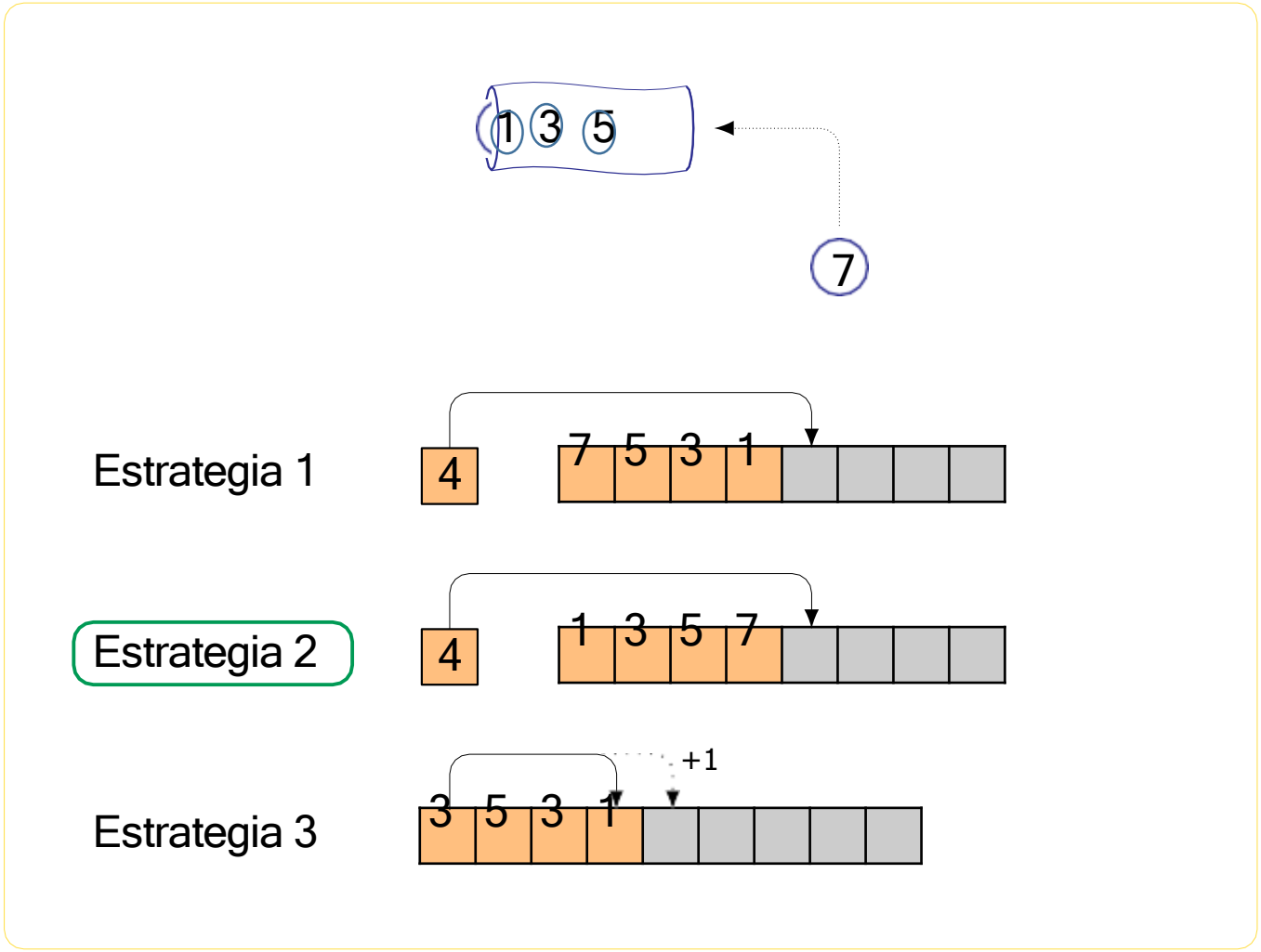
Las representaciones de una cola. Estrategias Acolar



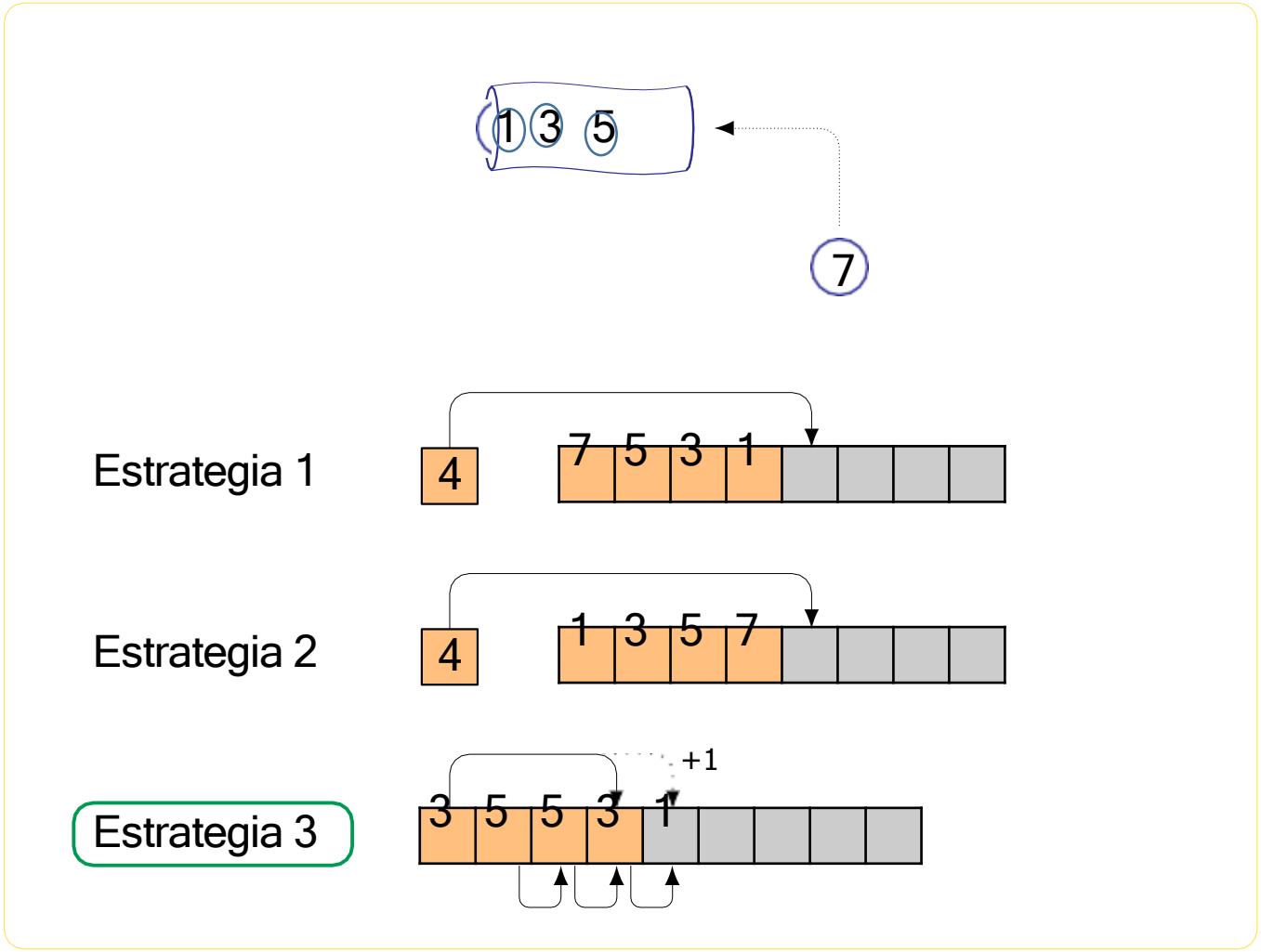
Las representaciones de una cola. Estrategias Acolar



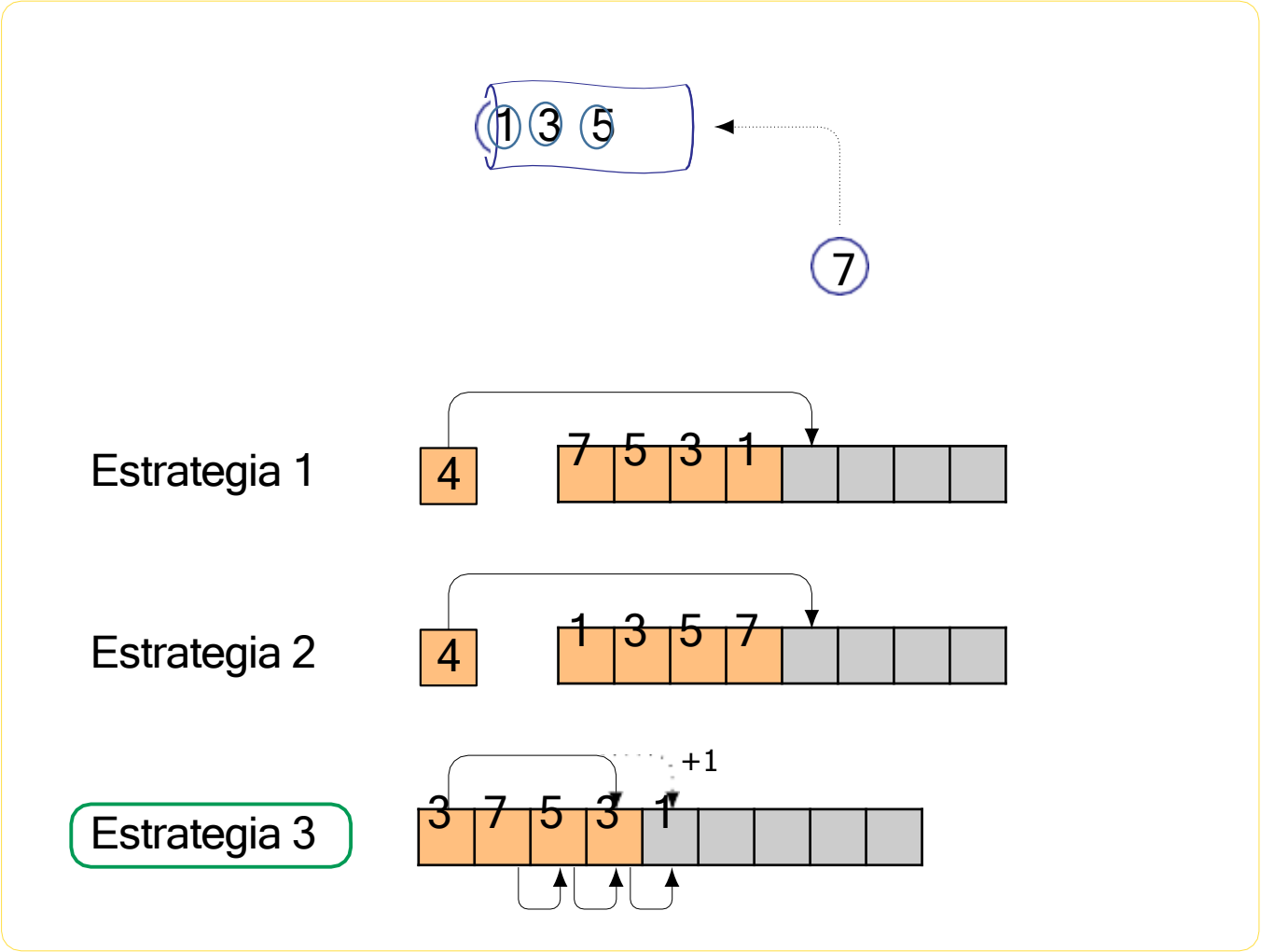
Las representaciones de una cola. Estrategias Acolar



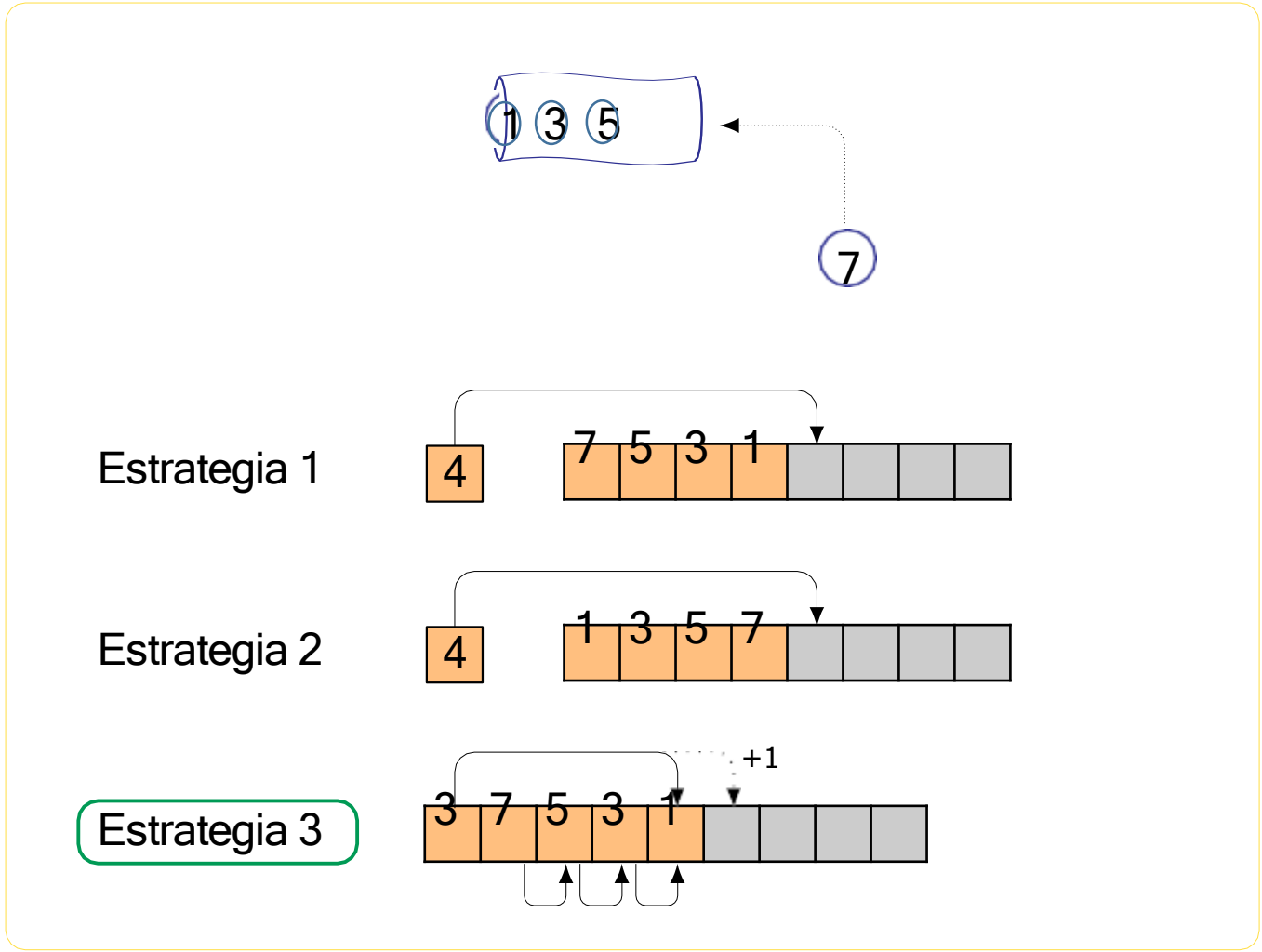
Las representaciones de una cola. Estrategias Acolar



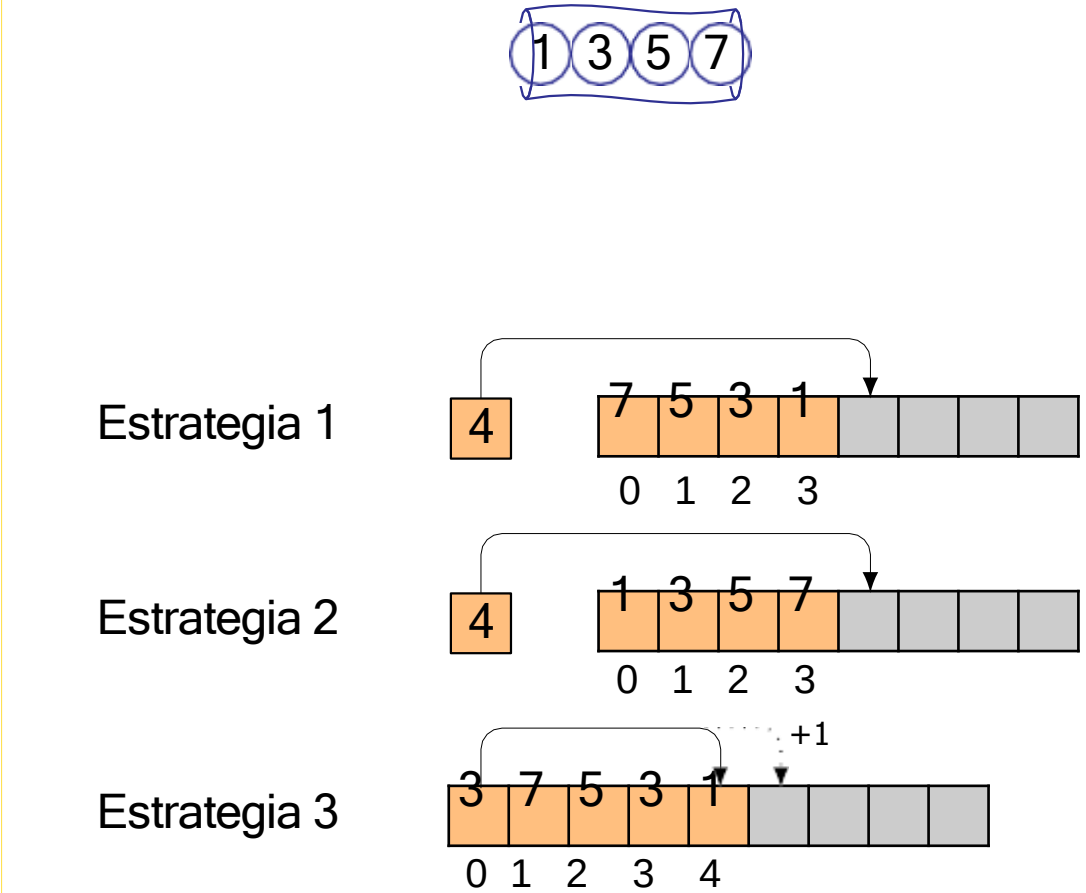
Las representaciones de una cola. Estrategias Acolar



Las representaciones de una cola. Estrategias



Las representaciones de una cola. Estrategias



TDA Cola. Implementación

La eliminación de un elemento del vector arr se representa dejándolo afuera de la parte del arreglo delimitada por la variable inx; a los efectos prácticos, cualquier elemento arr[i] situado en una posición $i \geq inx$ no existe más en la cola.

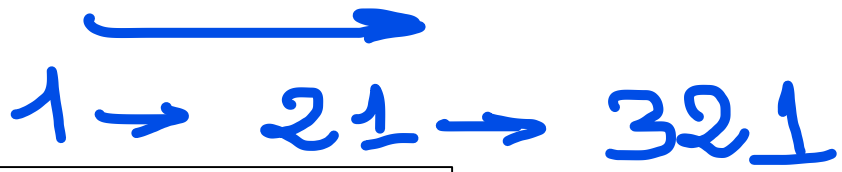
TDA Cola. Implementación

La eliminación de un elemento del vector arr se representa dejándolo afuera de la parte del arreglo delimitada por la variable inx; a los efectos prácticos, cualquier elemento arr[i] situado en una posición i≥ inx no existe mas en la cola

Convención de colores habitual:

Los métodos públicos, que pueden ser invocados desde fuera de la implementación, están en azul.
Los comentarios están en verde.
Los métodos y estructuras de datos no accesibles desde fuera de la implementación están en rojo.

Implementación. Estrategia 1




```
1.    public class ColaPU implements ColaTDA {
2.        int[] arr;                          // arreglo que contiene la información
3.        int inx;                            // cantidad de elementos en la cola
4.        public void InicializarCola() {
5.            arr = new int[100];
6.            inx = 0;
7.        }
8.        public void Acolar (int x) {
9.            for (int i=inx-1; i >= 0; i--) {
10.               arr[i+1] = arr[i];
11.           }
12.           arr[0] = x;
13.           inx++
14.       }
15.       public void Desacolar() {
16.           inx--;                          → desacolo el último
17.       }
18.       public boolean ColaVacia() {
19.           return  (inx == 0);
20.       }
21.       public int Primero() {
22.           return  arr[inx-1];             → el primero es el último en realidad
23.       }
21.   }
```

Implementación. Estrategia 2

```
1.    public class ColaPI implements ColaTDA {
2.        int [] arr;                          // arreglo que contiene la información
3.        int inx;                             // cantidad de elementos en la cola
4.        public void InicializarCola() {
5.            arr = new int[100];
6.            inx = 0;
7.        }
8.        public void Acolar(int x) {
9.            arr[inx] = x;
10.           inx++;
11.       }
12.       public void Desacolar() {
13.           for (int i = 0; i < inx-1; i++) {
14.               arr[i] = arr[i+1];
15.           }
16.           inx--;
16.       }
17.       public boolean ColaVacia() {
18.           return  (inx == 0);
24.       }
25.       public int Primero() {
26.           return  arr[0];
27.       }
   }
```

compila

el primero (pos 0) es el primero

Implementación. Estrategia 3

```
public class ColaPU2 implements ColaTDA {
    int[] arr;                                  // arreglo con toda la información
        public void InicializarCola() {
            arr = new int[100];
            arr[0] = 0;
        }
      } public void Acolar (int x);
            for (int i=arr[0]; i > 0; i--) {
                arr[i+1] = arr[i];
            }
            arr[1] = x;
            arr[0]++;
        }
        public void Desacolar() {
            arr[0]--;
        }
        public boolean ColaVacia() {
            return (arr[0] == 0);
        }
        public int Primero() {
            return  arr[arr[0]];
        }
```

TDA Cola con prioridad

TDA Cola con prioridad

Una cola de prioridad es un tipo de datos abstracto similar a una cola normal en la que cada elemento tiene además una *prioridad* asociada. En una cola de prioridad, un elemento con prioridad alta queda antes que un elemento con prioridad baja.

TDA Cola con prioridad

Una cola de prioridad es un tipo de datos abstracto similar a una cola normal en la que cada elemento tiene además una *prioridad* asociada. En una cola de prioridad, un elemento con prioridad alta queda antes que un elemento con prioridad baja.

En el caso de elementos con la misma prioridad, no definimos la semántica, con lo que el implementador tiene libertad de decisión.

TDA Cola con prioridad

Una cola de prioridad es un tipo de datos abstracto similar a una cola normal en la que cada elemento tiene además una *prioridad* asociada. En una cola de prioridad, un elemento con prioridad alta queda antes que un elemento con prioridad baja.

En el caso de elementos con la misma prioridad, no definimos la semántica, con lo que el implementador tiene libertad de decisión.

Las operaciones (Constructoras/Observadoras/Modificadoras) : agregar y eliminar datos de la cola (que llamaremos posteriormente AcolarPrioridad y Desacolar), consultar el valor y la prioridad del elemento más prioritario (que llamaremos Primero y Prioridad respectivamente) y consultar si la cola está o no vacía (a esta consulta la llamaremos ColaVacia.)

TDA Cola con prioridad

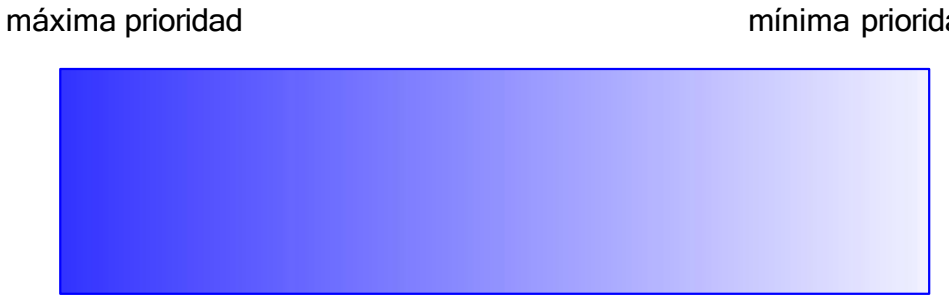
Una cola de prioridad es un tipo de datos abstracto similar a una cola normal en la que cada elemento tiene además una *prioridad* asociada. En una cola de prioridad, un elemento con prioridad alta queda antes que un elemento con prioridad baja.

En el caso de elementos con la misma prioridad, no definimos la semántica, con lo que el implementador tiene libertad de decisión.

Las operaciones (Constructoras/Observadoras/Modificadoras) : agregar y eliminar datos de la cola (que llamaremos posteriormente AcolarPrioridad y Desacolar), consultar el valor y la prioridad del elemento más prioritario (que llamaremos Primero y Prioridad respectivamente) y consultar si la cola está o no vacía (a esta consulta la llamaremos ColaVacia.)

A estas operaciones agregaremos la inicialización de una cola con prioridad, que llamaremos InicializarCola.

TDA Cola con prioridad

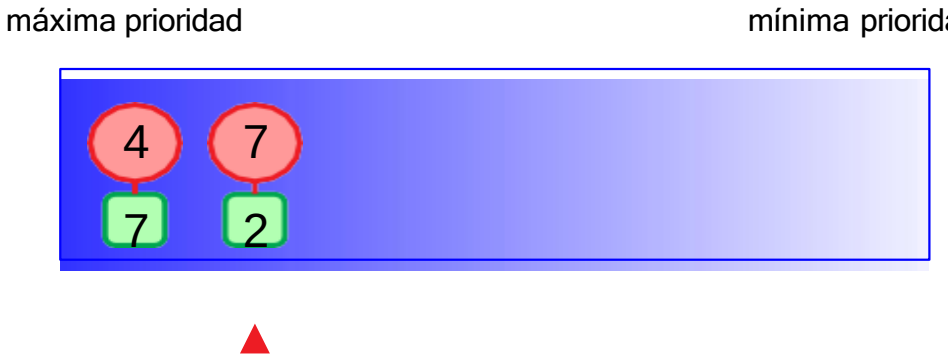


`ColaVacia() = true`

TDA Cola con prioridad

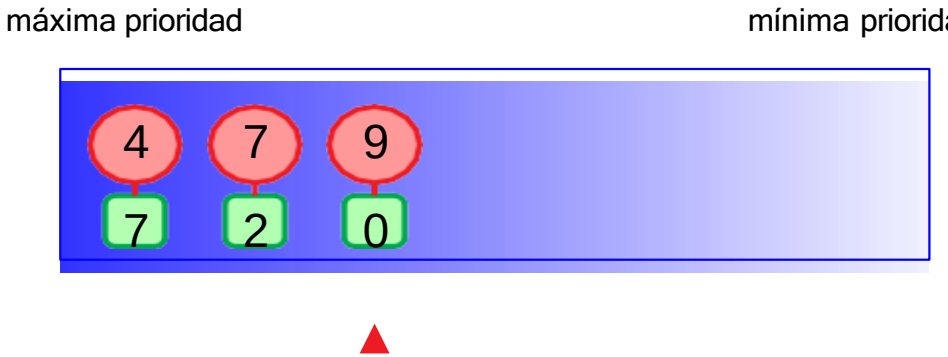

```
AcolarPrioridad(4,7)
ColaVacia() = false
Primero() = 4
Prioridad() = 7
```

TDA Cola con prioridad



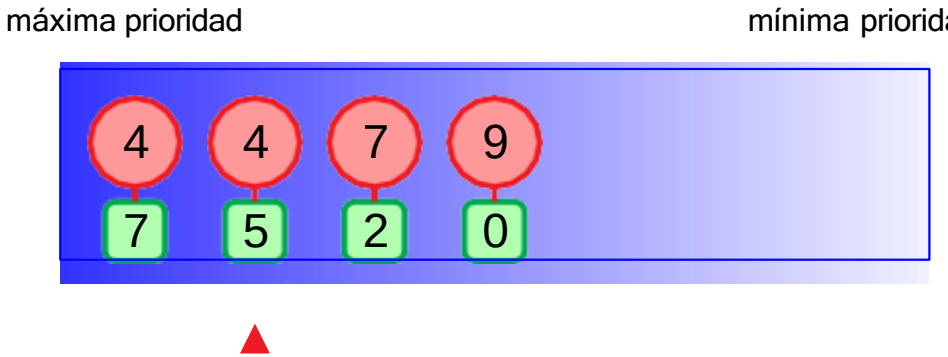

```
AcolarPrioridad(7,2)
ColaVacia() = false
Primero() = 4
Prioridad() = 7
```

TDA Cola con prioridad



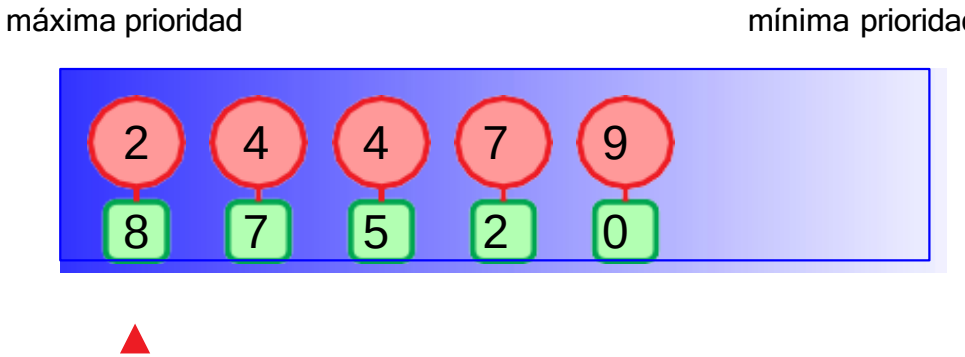

```
AcolarPrioridad(9,0)
ColaVacia() = false
Primero() = 4
Prioridad() = 7
```

TDA Cola con prioridad



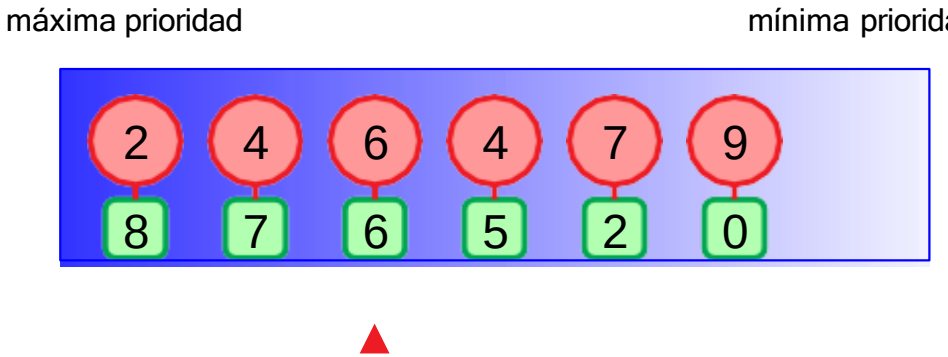

```
AcolarPrioridad(4,5)
ColaVacia() = false
Primero() = 4
Prioridad() = 7
```

TDA Cola con prioridad



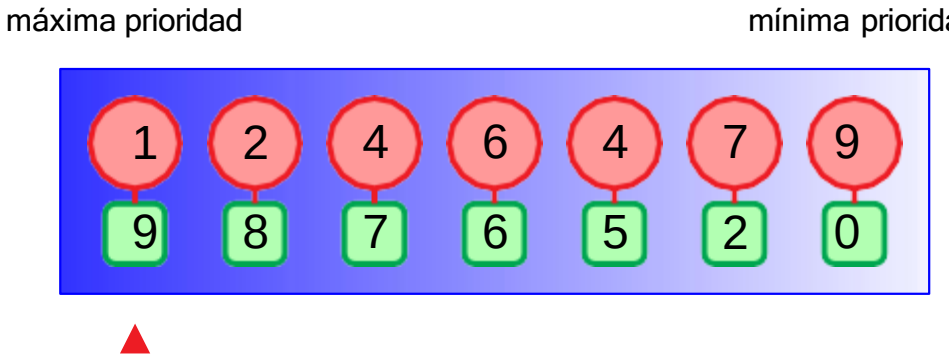

```
AcolarPrioridad(2,8)
ColaVacia() = false
Primero() = 2
Prioridad() = 8
```

TDA Cola con prioridad



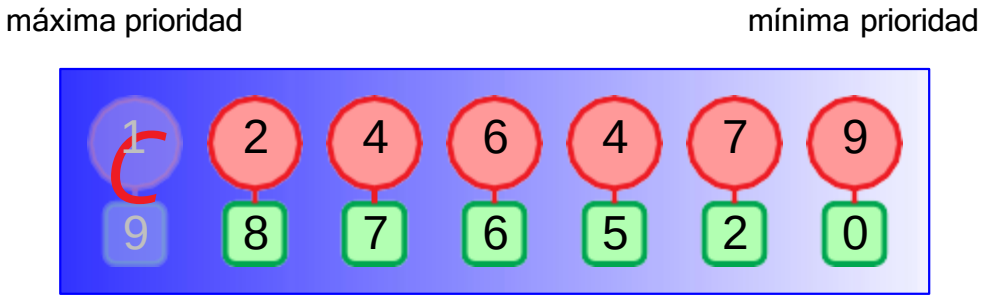

```
AcolarPrioridad(6,6)
ColaVacia() = false
Primero() = 2
Prioridad() = 8
```

TDA Cola con prioridad




```
AcolarPrioridad(1,9)
ColaVacia() = false
Primero() = 1
Prioridad() = 9
```

TDA Cola con prioridad




```
Desacolar()
ColaVacia() = false
Primero() = 2
Prioridad() = 8
```

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

Desacolar , que permite eliminar el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

Desacolar , que permite eliminar el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

Primero, que permite conocer el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

Desacolar , que permite eliminar el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

Primero, que permite conocer el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

*P*rioridad, que permite conocer la prioridad del elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

Desacolar , que permite eliminar el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

Primero, que permite conocer el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

*P*rioridad, que permite conocer la prioridad del elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

ColaVacia, que permite saber si una cola con prioridad inicializada está vacía o no

TDA Cola con prioridad

Una cola con prioridad permite almacenar, recuperar y eliminar valores. El elemento que se recupera o se elimina es siempre el de mayor prioridad, sin importar cuándo ingresó. Se puede saber cuál es el elemento de mayor prioridad, cuál es su prioridad y si la cola está vacía o no.

Los métodos (Constructoras/Observadoras/Modificadoras) son:

AcolarPrioridad, que permite agregar un elemento con su prioridad a una cola con prioridad inicializada.

Desacolar , que permite eliminar el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

Primero, que permite conocer el elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

*P*rioridad, que permite conocer la prioridad del elemento de mayor prioridad de una cola con prioridad inicializada y no vacía.

ColaVacia, que permite saber si una cola con prioridad inicializada está vacía o no

InicializarCola, que permite inicializar la estructura

TDA Cola con prioridad. Interfaz

```
1.  public interface ColaTDA; {
2.      void InicializarCola();        // sin precondiciones
3.      void AcolarPrioridad(int x);        // cola inicializada
4.      void Desacolar ();                  // cola inicializada y no vacía
5.      boolean ColaVacia();                // cola inicializada
6.      int Primero();                      // cola inicializada y no vacía
7.      int Prioridad();                    // cola inicializada y no vacía
8.  }
```

TDA Cola con prioridad. Interfaz

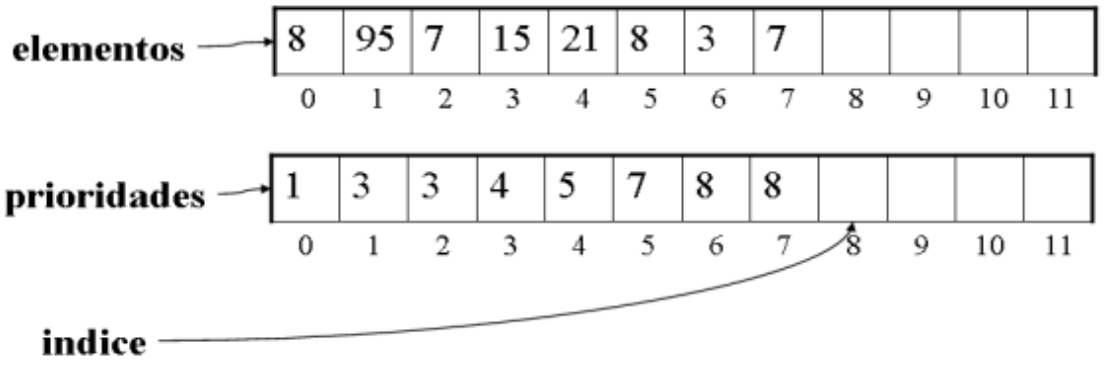
```
1.  public interface ColaTDA; {
2.      void InicializarCola();              // sin precondiciones
3.      void AcolarPrioridad(int x);         // cola inicializada
4.      void Desacolar();                    // cola inicializada y no vacía
5.      boolean ColaVacia();                 // cola inicializada
6.      int Primero();                       // cola inicializada y no vacía
7.      int Prioridad();                     // cola inicializada y no vacía
8.  }
```

Ejemplo: pasaje de los valores de una cola con prioridad *Origen* a una cola normal *Valores* y de las prioridades correspondientes a una cola normal *Prioridades*.

```
1.  public void Pasaje (ColaPrioridadTDA Origen, ColaTDA Valores,
    ColaTDA Prioridades) {
2.      while (!Origen.ColaVacia()) {
3.          Valores.Acolar(Origen.Primero());
4.          Prioridades.Acolar(Origen.Prioridad());
5.          Origen.Desacolar();
6.      }
7.  }
```

Estrategias de implementación 1

- Consideramos como estructuras dos arreglos, en uno de ellos tendremos los elementos y en el otro la prioridad de cada uno de esos elementos. Como veremos se tendrán que mantener ambos arreglos sincronizados en cuanto a que para una posición dada se tendrá en un arreglo el valor del elemento y en el otro la prioridad que le corresponde a ese elemento
- Como en todos los casos que hemos trabajado con arreglos tendremos una variable en donde llevemos la cantidad de elementos que se tienen.
- El elemento con mayor prioridad estará en la posición anterior a la marcada por índice



Implementación 1

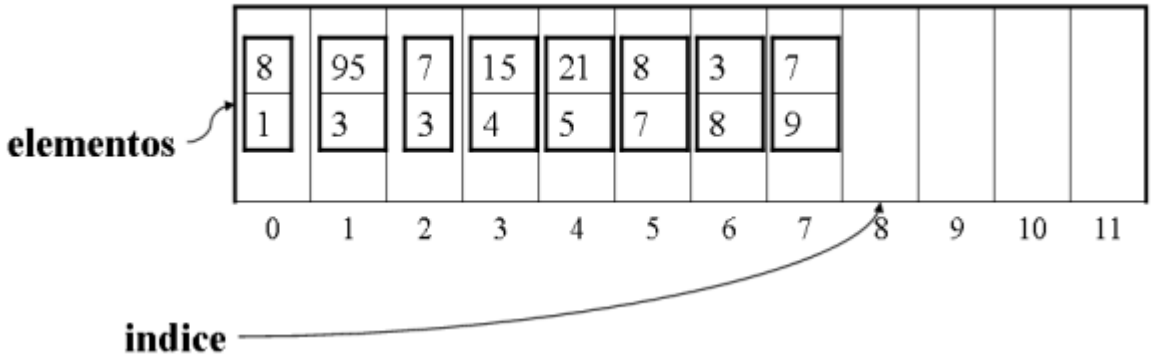
```
public class ColaPrioridadTDA implements ColaPrioridadTDA{
    in t [] elementos;
    in t [] prioridades;
    int indice;
    public void InicializarCola () {
        indice = 0;
        elementos = new in t [100];
        prioridades = new in t [100];
        }
    public void AcolarPrioridad ( in t x, in t prioridad){
        // desplaza a derecha los elementos de la cola mientras
        // estos tengan mayor o igual prioridad que la de x
        in t j = indice;
        f o r ( ; j >0 && prioridades[j -1] >= prioridad; j -- ){
        elementos[j] = elementos[j -1];
        prioridades[ j] = prioridades[j -1];
        }
        elementos[ j] = x ;
        prioridades[j] = prioridad;
        indice++;
        }
```

Implementación 1

```
public class ColaPrioridadTDA implements ColaPrioridadTDA{
    public void Desacolar() {
        indice - -;
    }
    public int Primero() {
        return elementos[ indice -1];
    }
    public boolean ColaVacia (){
        return ( indice == 0);
    }
    public int Prioridad() {
        return prioridades[ indice -1];
    }
    }
```

Estrategias de implementación 2

Considerar un arreglo cuyos elementos representan una estructura (por ser en Java una clase) que contiene el valor y la prioridad asociada a ese valor.



Implementación 2

```
public class ColaPrioridadAO implements ColaPrioridadTDA {

    class Elemento{
        int valor;
        int prioridad;
    }

    Elemento[] elementos;
    int indice;

    public void InicializarCola(){
        indice = 0;
        elementos = new Elemento[100];
    }

    public void AcolarPrioridad(int x, int prioridad){
        int j = indice;

        //desplaza a derecha los elementos de la cola mientras
        //estos tengan mayor o igual prioridad que la de x
        for (; j>0 && elementos[j-1].prioridad>=prioridad; j--){
            elementos[j] = elementos[j-1];
        }
        elementos[j]= new Elemento();
        elementos[j].valor=x;
        elementos[j].prioridad = prioridad;
        indice++;
    }
```

Implementación 2

```
public void Desacolar(){
    elementos[indice - 1]= null;
    indice--;
}

public int Primero(){
    return elementos[indice-1].valor;
}

public boolean ColaVacia(){
    return (indice == 0);
}

public int Prioridad(){
    return elementos[indice-1].prioridad;
}
```

TDA Conjunto

TDA Conjunto

Un *conjunto* es una colección de elementos en la que **no existen duplicados**.

No existe ninguna relación posicional concreta entre los elementos de un conjunto (contrariamente a una cola, en la que el elementos más reciente tiene prioridad o una cola, en la que el elemento más antiguo tiene prioridad.)

TDA Conjunto

Un *conjunto* es una colección de elementos en la que **no existen duplicados**.

No existe ninguna relación posicional concreta entre los elementos de un conjunto (contrariamente a una cola, en la que el elementos más reciente tiene prioridad o una cola, en la que el elemento más antiguo tiene prioridad.)

Un conjunto puede verse entonces como una caja o una bolsa de elementos.

TDA Conjunto

Un *conjunto* es una colección de elementos en la que **no existen duplicados**.

No existe ninguna relación posicional concreta entre los elementos de un conjunto (contrariamente a una cola, en la que el elementos más reciente tiene prioridad o una cola, en la que el elemento más antiguo tiene prioridad.)

Un conjunto puede verse entonces como una caja o una bolsa de elementos.

Las operaciones (CB/O/M) son: agregar y eliminar datos del conjunto (que llamaremos posteriormente Agregar y Sacar), elegir un elemento arbitrario del conjunto (que llamaremos Elegir), consultar si el conjunto está o no vacío (a esta consulta la llamaremos ConjuntoVacio) y si un elemento dado pertenece o no al conjunto (que llamaremos Pertenece.)

TDA Conjunto

Un *conjunto* es una colección de elementos en la que **no existen duplicados**.

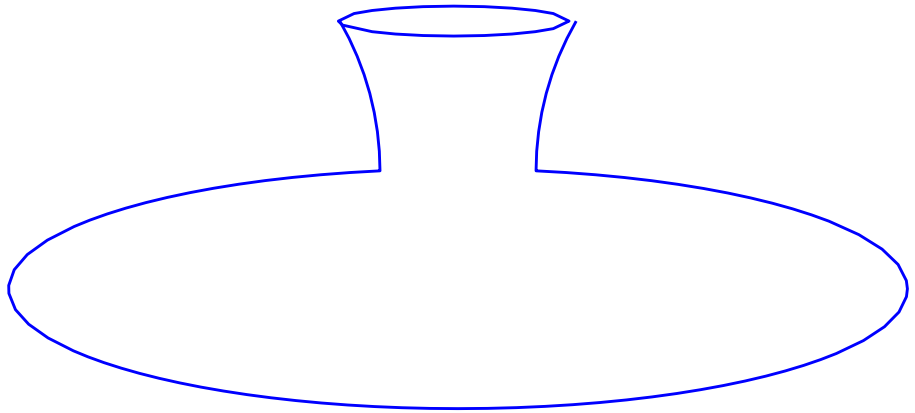
No existe ninguna relación posicional concreta entre los elementos de un conjunto (contrariamente a una cola, en la que el elementos más reciente tiene prioridad o una cola, en la que el elemento más antiguo tiene prioridad.)

Un conjunto puede verse entonces como una caja o una bolsa de elementos.

Las operaciones (CB/O/M) son: agregar y eliminar datos del conjunto (que llamaremos posteriormente Agregar y Sacar), elegir un elemento arbitrario del conjunto (que llamaremos Elegir), consultar si el conjunto está o no vacío (a esta consulta la llamaremos ConjuntoVacio) y si un elemento dado pertenece o no al conjunto (que llamaremos Pertenece.)

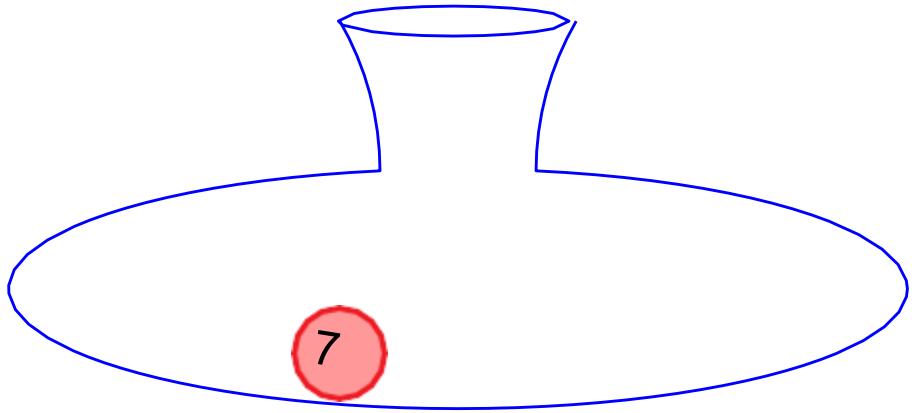
A estas operaciones agregaremos la inicialización de un conjunto, que llamaremos InicializarConjunto.

TDA Conjunto



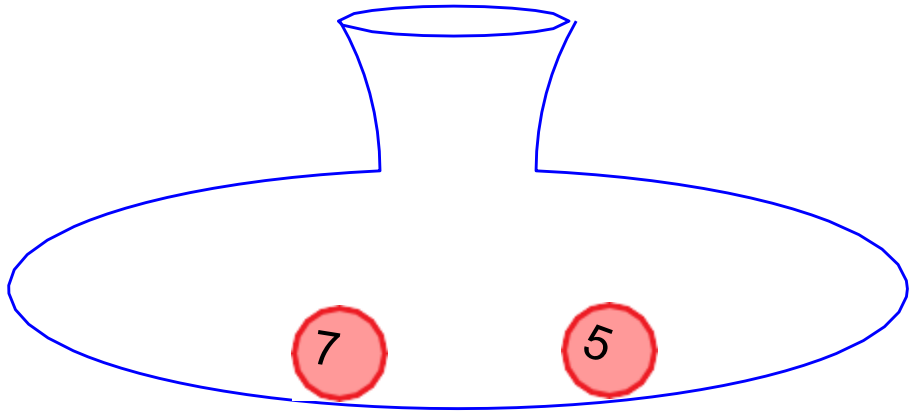
```
ConjuntoVacio() = true
Pertenece(4) = false
Pertenece(5) = false
```

TDA Conjunto



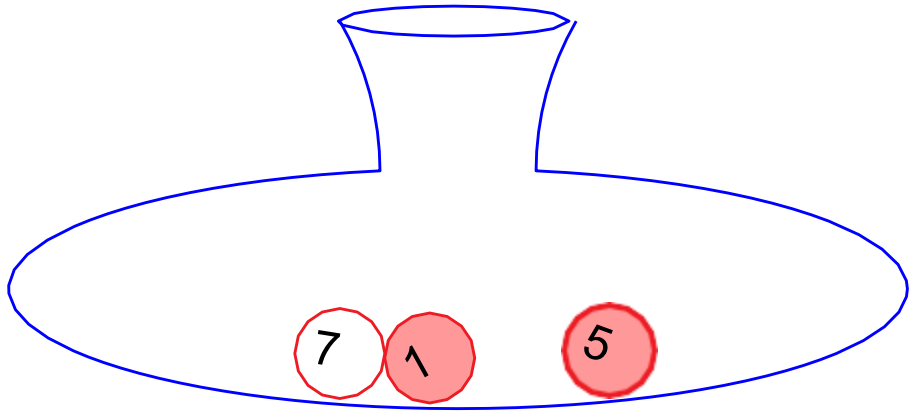
Agregar(7)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = false

TDA Conjunto



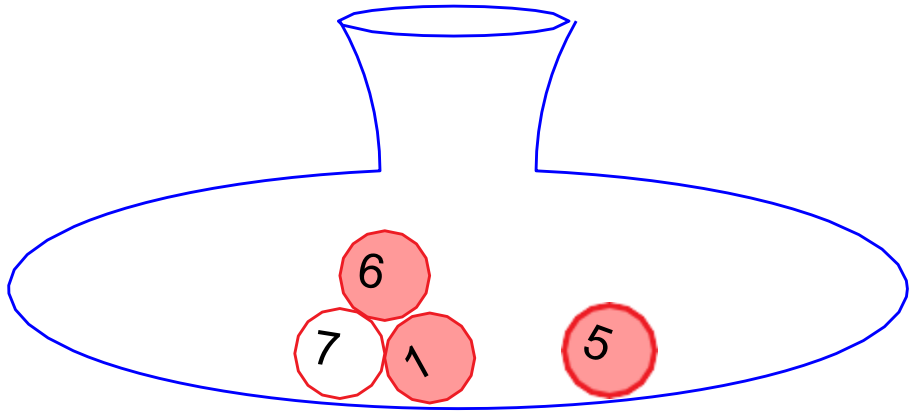
Agregar(5)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



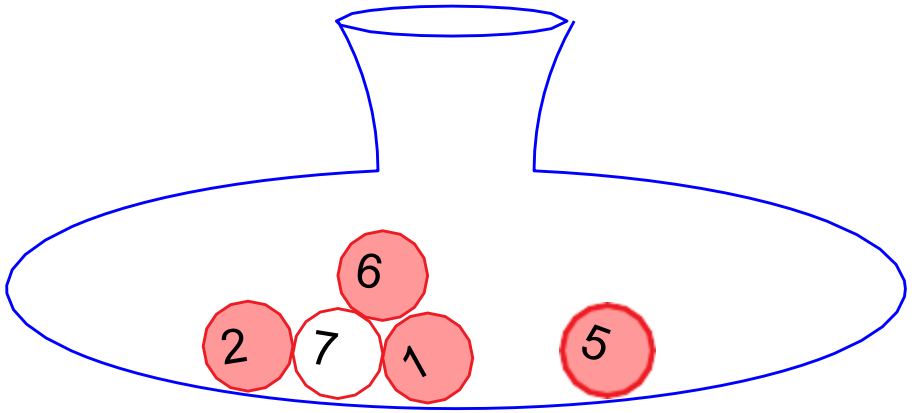
Agregar(1)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



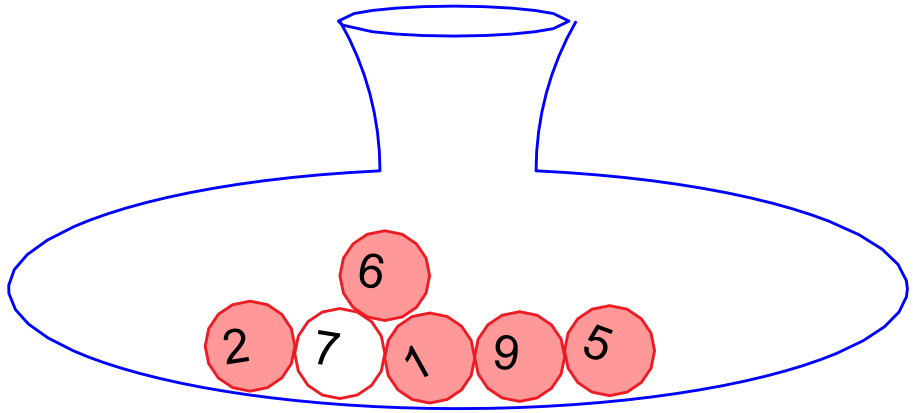
Agregar(6)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



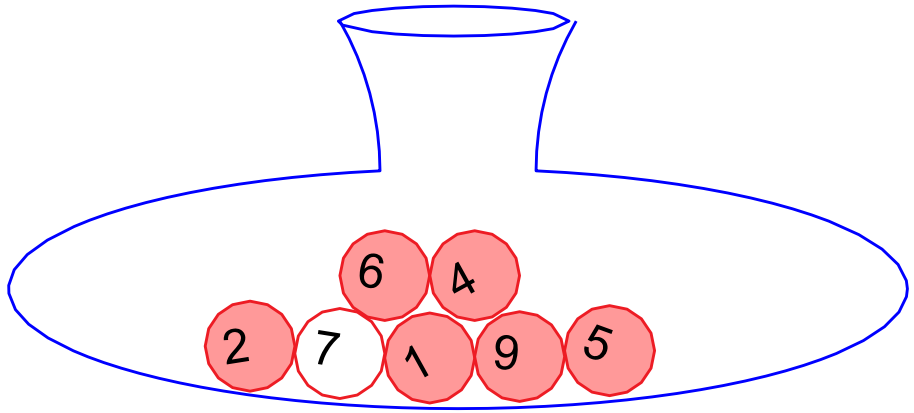
Agregar(2)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



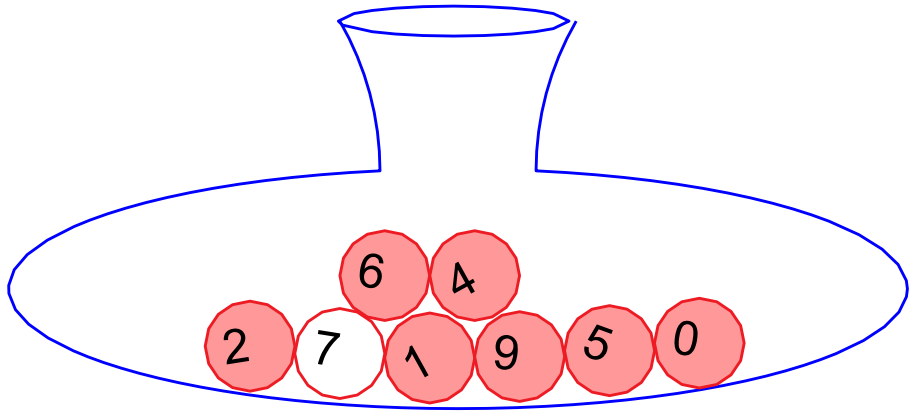
Agregar(9)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



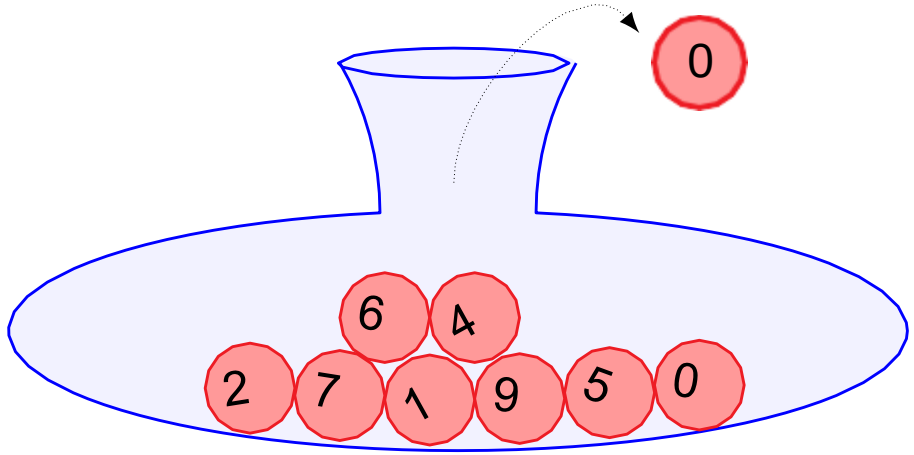
Agregar(4)
ConjuntoVacio() = false
Pertenece(4) = true
Pertenece(5) = true

TDA Conjunto



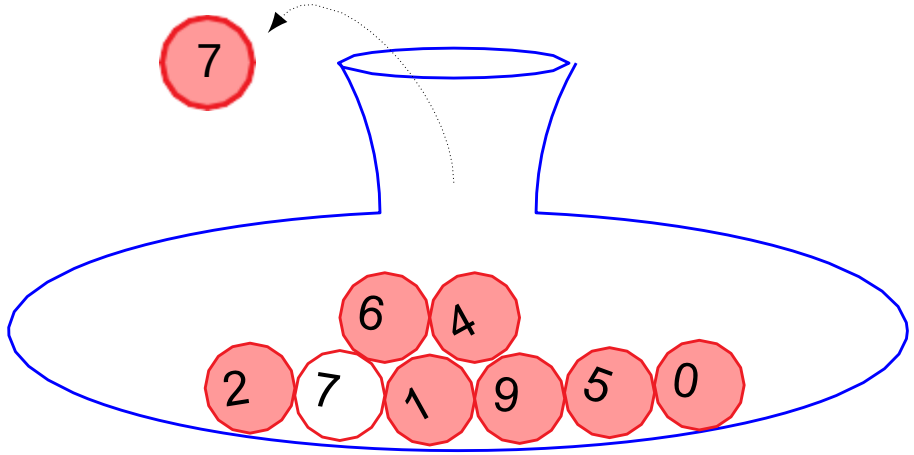
Agregar(0)
ConjuntoVacio() = false
Pertenece(4) = true
Pertenece(5) = true

TDA Conjunto



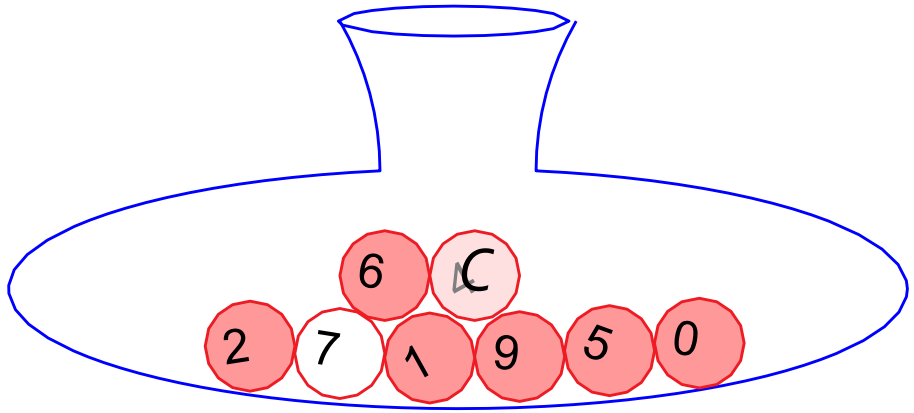

```
Elegir()
ConjuntoVacio() = false
Pertenece(4) = true
Pertenece(5) = true
```

TDA Conjunto



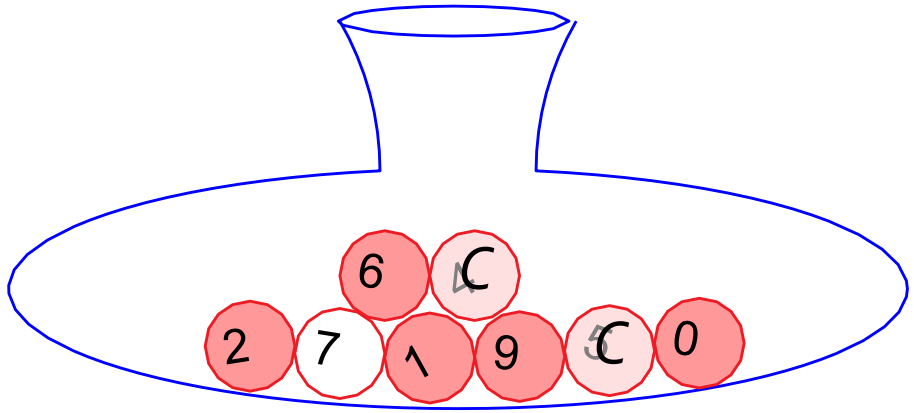

```
Elegir()
ConjuntoVacio() = false
Pertenece(4) = true
Pertenece(5) = true
```

TDA Conjunto



Sacar(4)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = true

TDA Conjunto



Sacar(5)
ConjuntoVacio() = false
Pertenece(4) = false
Pertenece(5) = false

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío
Sacar, que permite eliminar un elemento de un conjunto no vacío.

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío
Sacar, que permite eliminar un elemento de un conjunto no vacío.
Elegir , que devuelve un elemento arbitrario de un conjunto no vacío.

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío
Sacar, que permite eliminar un elemento de un conjunto no vacío.
Elegir , que devuelve un elemento arbitrario de un conjunto no vacío.
Pertenece, que nos dice si un elemento está en un conjunto.

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**

Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío
Sacar, que permite eliminar un elemento de un conjunto no vacío.
Elegir , que devuelve un elemento arbitrario de un conjunto no vacío.
Pertenece, que nos dice si un elemento está en un conjunto.
ConjuntoVacio, que indica si el conjunto tiene elementos o no.

TDA Conjunto

Observación preliminar: puesto que la estructura no tiene un orden, cuando **recuperamos** un dato, la estructura nos devuelve **uno cualquiera** de los que pertenecen a ella.

Por la misma razón, cuando queremos **eliminar** un valor debemos **indicarle cuál.**
Los métodos son (CB/O/M):

Agregar, que permite agregar un elemento de un conjunto no vacío
Sacar, que permite eliminar un elemento de un conjunto no vacío.
Elegir , que devuelve un elemento arbitrario de un conjunto no vacío.
Pertenece, que nos dice si un elemento está en un conjunto.
ConjuntoVacio, que indica si el conjunto tiene elementos o no.
InicializarConjunto, que inicializa el conjunto.

TDA Conjunto. Interfaz

```
1. public interface ConjuntoTDA; {
2.     void InicializarConjunto(); // sin precondiciones
3.     void Agregar(int x);        // conjunto inicializado
4.     int Elegir();               // conjunto inicializado y no vacío
5.     boolean ConjuntoVacio(); // conjunto inicializado
6.     void Sacar();                    // conjunto inicializado
7.     boolean Pertenece();             // cola inicializado
8. }
```

TDA Conjunto. Interfaz

```
1.    public interface ConjuntoTDA; {
2.        void InicializarConjunto();        // sin precondiciones
3.        void Agregar(int x);               // conjunto inicializado
4.        int Elegir();                      // conjunto inicializado y no vacío
5.        boolean ConjuntoVacio();           // conjunto inicializado
6.        void Sacar();                      // conjunto inicializado
7.        boolean Pertenece();               // cola inicializado
8.    }
```

Ejemplo: determinación de si un conjunto *Conjunto1* incluye a otro conjunto *Conjunto2*.

```
1.    public boolean Incluye (ConjuntoTDA Conjunto1, ConjuntoTDA Conjunto2) {
2.        boolean incluye = true;
3.        while (!Conjunto2.ConjuntoVacio() && incluye) {
4.            int x = Conjunto2.Elegir());
5.            if (!Conjunto1.Pertenece(x)) {
6.                incluye = false;
7.            } else {
8.                Conjunto2 .Sacar(x );
9.            }
10.       }
11.       return incluye
12.   }
```

TDA Conjunto. Implementación

A partir de la utilización de un arreglo.

Los elementos del conjunto no tienen un orden: cuando se elimina un elemento se puede colocar el último elemento que se encuentra en el arreglo en la posición del elemento a eliminar.

Ventaja: se evita el desplazamiento de todos los elementos.

TDA Conjunto. Implementación

A partir de la utilización de un arreglo.

Para implementar el método **Elegir**, dado que la especificación no indica cuál es el elemento a recuperar, vamos a recuperar el elemento que está en la primera posición del arreglo.

TDA Conjunto. Implementación

```
public class ConjuntoTA implements ConjuntoTDA{
    int[] a;
    int cant;

    public void Agregar(int x) {
        if (!this.Pertenece(x)){
            a[cant] = x;
            cant++;
        }
    }

    public boolean ConjuntoVacio() {
        return cant == 0;
    }

    public int Elegir() {
        return a[cant - 1];
    }

    public void InicializarConjunto() {
        a = new int[100];
        cant = 0;
    }
```

TDA Conjunto. Implementación

```
public boolean Pertenece(int x) {
    int i = 0;
    while (i<cant && a[i]!=x)
        i++;
    return ( i < cant);

}

public void Sacar(int x) {
    int i = 0;
    while (i<cant && a[i]!=x)
        i++;

    if (i < cant){
        a[i] = a[cant-1];
        cant--;
```

TP Nro 2

- Para los ejercicios propuestos, trabajar en forma grupal o individual

- Implemetar en JAVA los TDA

    - Cola
    - Cola con prioridad
    - Conjunto