



## 1. Estructuras con arreglos

**Ejercicio 1.** Quizás la forma más simple de implementar un conjunto acotado sea mediante un array de tamaño fijo, utilizando la siguiente estructura:

```
Modulo ConjAcotadoArr<T> implemента ConjAcotado<T> {
    var datos: Array<T>
    var largo: int
}
```

En la variable *datos* guardaremos los elementos. Como el tamaño del arreglo es fijo, necesitamos otra variable, a la que llamamos *largo*, que indique cuántas posiciones del arreglo *datos* están siendo usadas.

Con esta misma estructura, tenemos dos opciones: permitir que en el arreglo haya elementos repetidos o no permitirlo.

- ¿Cuál es más eficiente? Cuándo usaría cada una de las dos versiones?
- Escriba el invariante de representación y la función de abstracción para **el caso de sin repetidos**.
- Escriba los algoritmos para las operaciones de **agregar** un elemento y **sacar** un elemento (para la versión de sin repetidos)
- Respecto de la operación **sacar**, piense un algoritmo que no requiera generar un nuevo arreglo para reemplazar a *datos*, sino que se resuelva modificando alguna de sus posiciones
- **Intente escribir el invariante de representación y la operación de agregar para el caso de con repetidos. Qué dificultades encuentra? Indique alguna manera de abordarlas.**

**Ejercicio 2.** Cómo implementaría una pila *no acotada* (sin capacidad máxima) utilizando arreglos? Escriba la estructura propuesta, el invariante de representación, la función de abstracción y las operaciones **apilar** y **desapilar**.

**Ejercicio 3.** Se quiere agregar al TAD Pila una operación **eliminar** que permita eliminar un elemento de cualquier posición de la pila.

```
proc eliminar(inout p: Pila<T>, i: ℤ)
    requiere {p = P0}
    requiere {0 ≤ i < |p.s|}
    asegura {p.s = concat(subseq(P0.s, 0, i), subseq(P0.s, i + 1, |P0.s|))}
```

(NOTA: Tómese unos minutos para pensar una forma eficiente de implementar esta nueva pila antes de continuar...)

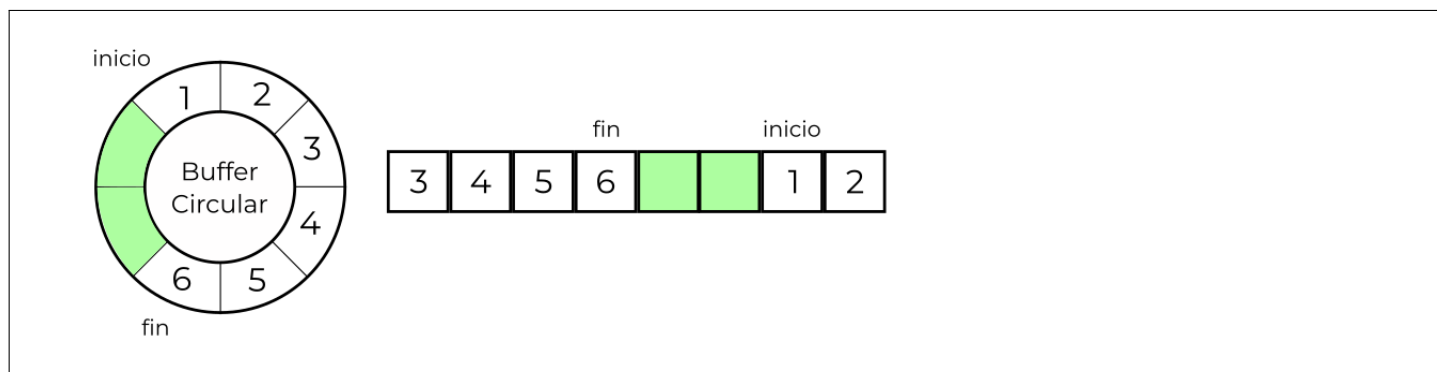
Para implementarlo, se propone una estructura con dos arreglos: un arreglo de tipo *T* que guarda los elementos de la pila y un arreglo de tipo *bool* que indica si esa posición está siendo usada. Así, para eliminar un elemento de la pila, basta con poner en *false* dicha posición en el arreglo de bools.

```
Modulo PilaConElimArr<T> implemента PilaConElim<T> {
    var datos: Array<T>
    var enUso: Array<bool>
    var largo: int
}
```

Se pide:

- Escribir el invariante de representación y la función de abstracción
- Escribir los algoritmos de **apilar**, **desapilar** y **eliminar**

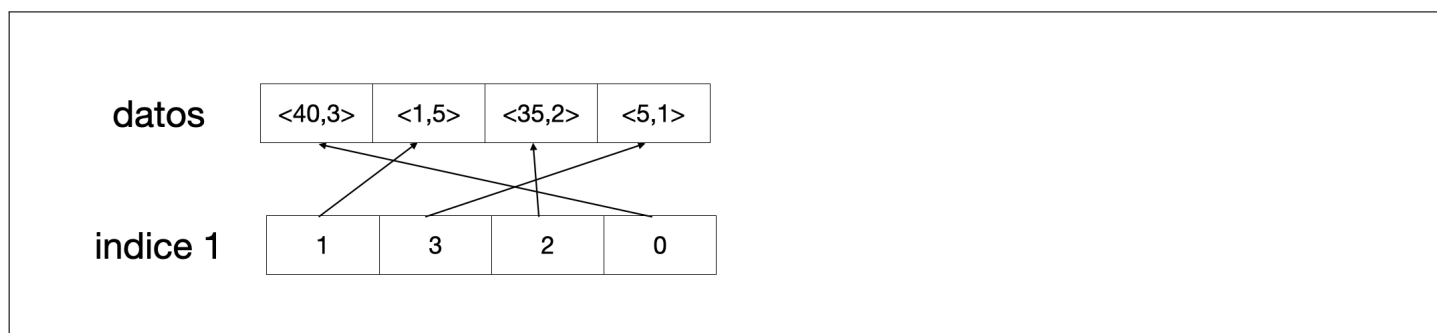
**Ejercicio 4.** Una forma eficiente de implementar el TAD Cola en su versión acotada (con una cantidad máxima de elementos predefinida), es mediante un *buffer circular*. Esta estructura está formada por un array del tamaño máximo de la cola ( $n$ ) y dos índices (*inicio* y *fin*), que indican en qué posición empieza y en qué posición termina la cola, respectivamente. Al encolar un elemento, se lo guarda en la posición indicada por el índice *fin* y se incrementa dicho índice. Al desencolar un elemento, se devuelve el elemento indicado por el índice *inicio* y se incrementa el mismo. En ambos casos, si el índice a incrementar supera el tamaño del array, se lo reinicia a 0.



- Elija una estructura de representación
- Escriba el invariante de representación y la función de abstracción
- Escriba los algoritmos de las operaciones **encolar** y **desencolar**
- ¿Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

**Ejercicio 5.** Un *índice* es una estructura secundaria que permite acceder más rápidamente a los datos a partir de un determinado criterio. Básicamente un índice guarda *punteros* o, en el caso de arreglos, *posiciones* a los elementos en un orden en particular, diferente al orden original.

Imagine un conjunto de tuplas con dos componentes. Algunas veces vamos a querer buscar (rápido) por la primera componente y a veces por la segunda. Podríamos guardar los datos en sí en un arreglo y tener otros dos arreglos: uno con las posiciones de los elementos en el arreglo original pero *ordenados por la primera componente*, y otro con la posición de los elementos *ordenados por la segunda componente*. A estos arreglos se los denomina índices.



- Escriba la estructura propuesta
- Escriba el invariante de representación y la función de abstracción, en castellano y en lógica
- Escriba los algoritmos de **BuscarPorPrimera** y **BuscarPorSegunda** que busca por la primera o la segunda componente respectivamente
- Escriba los algoritmos de **agregar** y **sacar**

## 2. Invariante de representación y función de abstracción en modelado de problemas

Tenemos un TAD que modela las ventas minoristas de un comercio. Cada venta es individual (una unidad de un producto) y se quieren registrar todas las ventas. El TAD tiene un único observador:

```
TAD Comercio {  
  obs ventasPorProducto: dict<Producto, seq<tupla<Fecha, Monto>>>  
}  
  
Producto es string  
Monto es int  
Fecha es int (segundos desde 1/1/1970)
```

**ventasPorProducto** contiene, para cada producto, una secuencia con todas las ventas que se hicieron de ese producto. Para cada venta, se registra la fecha y el precio. Se puede considerar que todas las fechas son diferentes. Este TAD lo vamos a implementar con la siguiente estructura:

```
Modulo ComercioImpl implementa Comercio {  
  var ventas: SecuenciaImpl<tupla<Producto, Fecha, Monto>>  
  var totalPorProducto: DiccionarioImpl<Producto, Monto>  
  var ultimoPrecio: DiccionarioImpl<Producto, Monto>  
}
```

- **ventas** es una implementación de secuencia con todas las ventas realizadas, indicando producto, fecha y monto.
- **totalPorProducto** asocia cada producto con el dinero total obtenido por todas sus ventas.
- **ultimoPrecio** asocia cada producto con el monto de su última venta registrada.

Se pide:

- Escribir en forma coloquial y detallada el invariante de representación y la función de abstracción.
- Escribir ambos en el lenguaje de especificación.

**Ejercicio 6.** Considere la siguiente especificación de una relación uno/muchos entre alarmas y sensores de una planta industrial: un sensor puede estar asociado a muchas alarmas, y una alarma puede tener muchos sensores asociados.

```
TAD Planta {  
  obs alarmas: conj<Alarma>  
  obs sensores: conj<tupla<Sensor, Alarma>>  
  
  proc nuevaPlanta(): Planta  
    asegura {res.alarmas = {}}  
    asegura {res.sensores = {}}  
  
  proc agregarAlarma(inout p: Planta, in a: Alarma)  
    requiere {p = P0}  
    requiere {a ∉ p.alarmas}  
    asegura {p.alarmas = P0.alarmas ∪ {a}}  
    asegura {p.sensores = P0.sensores}  
  
  proc agregarSensor(inout p: Planta, in a: Alarma, in s: Sensor)  
    requiere {p = P0}  
    requiere {ainp.alarmas}  
    requiere {{s, a} ∉ p.sensores}  
    asegura {p.alarmas = P0.alarmas}  
    asegura {p.sensores = P0.sensores + {{s, a}}}  
}
```

Se decidió utilizar la siguiente estructura como representación, que permite consultar fácilmente tanto en una dirección (sensores de una alarma) como en la contraria (alarmas de un sensor).

```
modulo PlantaImpl implementa Planta {
  var alarmas: Diccionario<Alarma, Conjunto<Sensor>>
  var Sensores: Diccionario<Sensor, Conjunto<Alarma>>
}
```

Se pide:

- Escribir formalmente y en castellano el invariante de representación.
- Escribir la función de abstracción.

## 2.1. Anexo: TADs acotados

Todos los TADs básicos van a contar con una versión acotada, es decir, que acepta un número limitado de elementos. A modo de ejemplo, esta sería la versión acotada del TAD Conjunto:

```
TAD ConjuntoAcotado<T> {
  obs elems: conj<T>
  obs cap: int

  proc conjVacio(c: int): ConjuntoAcotado<T>
    asegura {res.cap = c ∧ res.elems = ⟨⟩}

  proc pertenece(in c: ConjuntoAcotado<T>, in e: T): bool
    asegura {res = true ↔ e ∈ c.elems}

  proc agregar(inout c: ConjuntoAcotado<T>, in e: T)
    requiere {c = C0}
    requiere {|c.elems| < c.cap}
    asegura {c.elems = C0.elems ∪ {e}}

  proc sacar(inout c: ConjuntoAcotado<T>, in e: T)
    requiere {c = C0}
    asegura {c.elems = C0.elems - {e}}

  proc unir(inout c: ConjuntoAcotado<T>, in c': ConjuntoAcotado<T>)
    requiere {c = C0}
    requiere {|c.elems| + |c'.elems| ≤ c.cap}
    asegura {c.elems = C0.elems ∪ c'.elems}

  proc restar(inout c: ConjuntoAcotado<T>, in c': ConjuntoAcotado<T>)
    requiere {c = C0}
    asegura {c.elems = C0.elems - c'.elems}

  proc intersectar(inout c: ConjuntoAcotado<T>, in c': ConjuntoAcotado<T>)
    requiere {c = C0}
    asegura {c.elems = C0.elems ∩ c'.elems}

  proc agregarRápido(inout c: ConjuntoAcotado<T>, in e: T)
    requiere {c = C0 ∧ e ∉ c.elems}
    requiere {|c.elems| < c.cap}
    asegura {c.elems = C0.elems ∪ {e}}

  proc tamaño(in c: ConjuntoAcotado<T>): ℤ
    asegura {res = |c.elems|}
}
```