

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> implemenata 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.

- Escriba el invariante de representación y la función de abstracción para ambos casos (con y sin repetidos)
- ¿Cuál es más eficiente? Cuándo usaría cada una de las dos versiones?
- Escriba los algoritmos para las operaciones de **agregar** un elemento y **sacar** un elemento para ambas versiones
- 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

pred ABS ($C: \text{ConjAcotado}$)

$$C.\text{cap} = \boxed{\text{SmRep}(C')}$$

$\{1, 3\} \leftrightarrow [1, 1, 3, 3]$
 $\text{cap} = 3$ $\text{length} = 4$

Aux ... ($\text{in } C': \dots$):
 $2N \times \text{card Rep. Tc^n} \uparrow$

$$\sum_{i=0}^{|S|} 2^{\text{posiciones}(S[i], S)}$$

$\{1, 3, 2\} \leftrightarrow \{1, 2\}$

COPY = 3

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*.

```
TAD Pila<T> {
    obs s: seq<T>
    proc pilaVacia(): Pila<T>
        asegura {res.s = ()}
```

2

```
proc vacia(in p: Pila<T>): bool
    asegura {res = true  $\leftrightarrow$  p.s = ()}

proc apilar(inout p: Pila<T>, in e: T)
    requiere {p = P0}
    asegura {p.s = concat(P0.s, (e))}

proc desapilar(inout p: Pila<T>): T
    requiere {p = P0}
    requiere {p.s  $\neq$  ()}
    asegura {p.s = subseq(P0.s, 0, |P0.s| - 1)}
    asegura {res = P0.s[|P0.s| - 1]}

proc tope(in p: Pila<T>): T
    requiere {p = P0}
    requiere {p.s  $\neq$  ()}
    asegura {res = P0.s[|P0.s| - 1]}
```

MODULO PILANOACOTADA<T> IMPLEMENTA PILACT> {

\rightarrow LENGTH DEL ARR (ELEMENTOS NO ESTAR OCUPADOS)

VAR DATA: ARRAY<T>

VAR ELEMENTOS: INT
ESP. OCUPADOS

PRED INVREP (P: PILANOACOTADA<T>) {

 P.ELEMENTOS \leq |P.DATA| \wedge |P.DATA| > 0 ✓

}

PRED ABS (P: PILANOACOTADA<T>, P: PILACT>) {

 P'.ELEMENTOS = |P.S| \wedge

 ($\forall e:T$) (e \in P.S \rightarrow $\exists i:\mathbb{N}$) ($0 \leq i < P'.ELEMENTOS \wedge P'.DATA[i] = e$) \wedge

 ($\forall j:\mathbb{N}$) ($0 \leq j < P'.ELEMENTOS \rightarrow \exists i:\mathbb{N}$) ($0 \leq i < P.S \wedge P.S[j] = P'.DATA[i]$) ✓

}

PROC NUEVAPILA (): PILANOACOTADA<T> {

 Var DATA := NEW ARRAY<T>(0)

 Var ELEMENTOS := 0

```

PROC APILAR(inout P:PILANOACOTADA, in e:T) {
    VAR i:int := 0
    VAR NUEVALONG:int = P.ELEMENTOS + 1
    VAR NUEVAPILA := NEW ARRAY<T>(NUEVALONG)
    WHILE(i < P.ELEMENTOS) DO:
        NUEVAPILA[i] := P.DATA[i]
        i := i + 1
    END WHILE
    → Agregar al FINAL
    NUEVAPILA[i] := e
    P.ELEMENTOS := NUEVALONG
    P.DATA := NUEVAPILA
}

```



PROC DESAPILAR (inout P:PILANOACOTADA) : T {

REQUIERE : {P.ELEMENTOS > 0}

↳ Si en INVREP mantiene ciertas DECIR ESTO. Podría ser más, así que lo copiamos así

```

    VAR DESAPILANDO:T = P.DATA[P.DATA.LENGTH-1] → Tomo el ult
    VAR i:int = 0
    VAR NUEVALONG:int = P.ELEMENTOS - 1
    VAR ARR:ARRAY<T> = NEW ARRAY<T>(NUEVALONG)
    WHILE(i < NUEVALONG) DO:
        ARR[i] := P.DATA[i]
        i := i + 1
    END WHILE

```



P.ELEMENTOS := NUEVALONG

P.DATA := ARR

RETURN DESAPILANDO

}

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.

→ area q igual seguré estaría en el arr.

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

EVEM EN TRUE EN USO

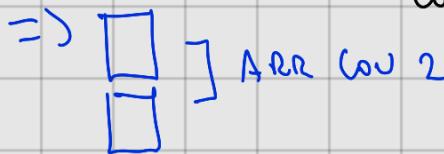
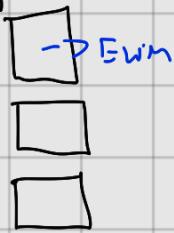
Se pide:

- Escribir el invariante de representación y la función de abstracción

1

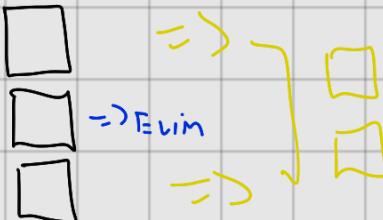
- Escribir los algoritmos de *apilar*, *desapilar* y *eliminar*

1ER)



→ Long & largo en los elem q tiene contado eliminados.

2do)



⇒ O(n)

Como debemos eliminar q obligatoriamente crean otros ARR pues tienen nuevos ARRAYS con LONGITUD - 1, recordale lista original q lleva a guardar el con $j = i \rightarrow$ A EVIM.

TAD PilaConElim<T> { → Mismo q Pila PERO CON EVIM }

OBS S: req < T >

¿ Me debré operar un OBS con EVIM?
Entiendo q no igual q el ultimo obs

Májicor S.

}

[1, 2, 3] largo: 2
[T, F, T]

MÓDULO PILACONELIMARR<T> IMPLEMENTA PILACONELIM<T> {

VAR DATOS: ARRAY<T>



VAR ENUSO ARRAY<BOOL>

VAR LARGO: INT

↳ DATOS & EN USO SON TRUE

PRED INVREP(P: PILACONELIMARR<T>) {
_{|P.DATOS|}

|P.ENUSO| = |P.DATOS| ^ P.LARGO = $\sum_{i=0}^{|P.DATOS|-1}$ if(P.DATOS[i]) THEN 1 ELSE 0 ^ P.LARGO ≥ 0

}

PRED ABS(P': PILACONELIMARR<T>, P: PILACONELIM<T>) {

($\forall i: \pi$) ($0 \leq i < |P'.DATOS| \wedge P'.ENUSO[i] \rightarrow_L P'.DATOS[i] \in P.S$)

($\forall j: \pi$) ($0 \leq j < |P.S| \rightarrow_L (\exists i: \pi) (0 \leq i < |P'.DATOS| \wedge P'.DATOS[i] = P.S[j] \wedge P'.ENUSO[i])$)

}

PROC NUEVAPILACONELIMARR<T>(): PILACONELIMARR<T> {

Ner. LARGO := 0



Ner. DATOS := NEW ARRAY<T>(0)

Ner. ENUSO := NEW ARRAY<BOOL>(0)

}

PROC APIALAR (INOUT P: PILACONELIMARR<T>, e: T) {

VAR i: INT := 0

CUMPU:

VAR DATA: ARRAY<T> = NEW ARRAY<T>(P.DATOS.LENGTH)

INVERP

VAR ACT: ARRAY<BOOL> = NEW ARRAY<BOOL> (P.DATOS.LENGTH)

WHILE ($i < P.DATOS.LENGTH - 1$) DO:

 DATA[i] := P.DATOS[i]

 ACT[i] := P.ENUSO[i]

i := *i* + 1;

END WHILE.

 DATA[i] := e \rightarrow UNO MAS EN P.LARGO

 ACT[i] := $\overline{\text{true}}$

PARA cumplir INVREP.

P.DATOS = DATA

P.ENUSO = ACT

P.LARGO = P.LARGO + 1

}

PROC DESAPILAR (inout P:PILA CON ELIMINAR<T>) : T {

REQUIERE: {P.LARGO > 0}

\hookrightarrow Si en INVREP me tiene que decir esto. PdRKH
que no es, oí q lo copiaron de

VAR i:int := 0

VAR DATA: ARRAY<T> = NEW ARRAY<T> (P.DATA.LENGTH - 1)

VAR ACT: ARRAY<BOOL> = NEW ARRAY<BOOL> (P.DATA.LENGTH - 1)

VAR DESAPILAR: T = P.DATA[P.DATA.LENGTH - 1]

\rightarrow TODOS MENOS ULT.

WHILE ($i < P.DATA.LENGTH - 1$) DO:

 DATA[i] := P.DATOS[i]

 ACT[i] := P.ENUSO[i]

i := *i* + 1;

END WHILE.

\rightarrow ENUSO = $\overline{\text{true}}$

\rightarrow *i* desapilas q no tiene elim, monto el largo.

IF (P.ENUSO[P.DATA.LENGTH - 1] = $\overline{\text{true}}) THEN$

$P.\text{LARGO} := P.\text{LARGO} - 1$
 $P.\text{DATOS} := \text{DATA}$
 $P.\text{ENUSO} := \text{ACT}$

RETURN DESAPILADO
 }

PROC ELIMINAR (inout P : PILA CON ELIMINAR(), i : INT) {

\rightarrow no habrá de $P.\text{ENUSO}$ si en INVREP digo que
 REQUIERE: $\{0 \leq i < |P.\text{DATOS}| \}$ $|P.\text{ENUSO}| = |P.\text{DATOS}|$ ✓
 $P.\text{ENUSO}[i] := \text{folie}$] Cumplir el INVREP.
 $P.\text{LARGO} = P.\text{LARGO} - 1$

}

}

Algunas Observaciones:

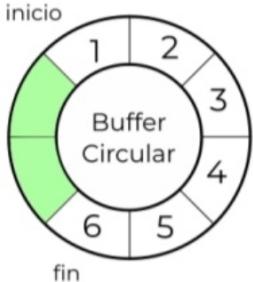
- Ol ver una pila NO tiene sentido operar en el primer folie de ENUSO que encontramos.

$\text{DATA}: [1, 2, 3] \xrightarrow{\text{AGGREGACIÓN}} [4, 2, 3]$ Ahora memoria pila NO cumple
 $\text{ENUSO}: [F, T, T] \quad [T, T, T]$ especificación PILA.

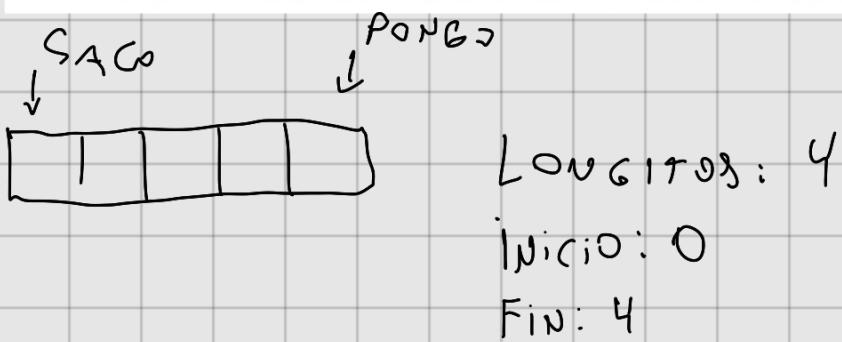
- El uso de largo solo se da para calcular el largo de S en el TAD ORIGINAL.

- Hay q tener cuidado al desapilar q si el elem es el
 (el último logio (folie); el largo q se habia guardado
 anterior x ese elem en la OPERACIÓN 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 inicio y se incrementa dicho índice. Al desencolar un elemento, se devuelve el elemento indicado por el índice fin 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?



Muy q el Punto inicio=FIN

ESTA MÁS ENUNCIADO. INICIO ES FIN Y FIN ES INICIO

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.

INVREP: VIVE EN EL MÓDULO. VQUE SIEMPRE.

- ^{total} En VENTAS, LA FECHA ES MAYOR A 0 Y MONTO > 0.
- En TOTALPORPRODUCTO y ULTIMOPRECIO, EL PRODUCTO DE DICCIONARIOIMPL ESTÁ EN VENTAS, EN PRIMER ELEMENTO DE TUPLA.
- TOTALPORPRODUCTO ES LA CANTIDAD DE DINERO POR PRODUCTO EN TODAS LAS VENTAS, X LOS TORNOS SI HAY UN PRODUCTO X EN LA SECUENCIA DE VENTAS HAY TUPLA DONDE SU PRIMER ELEMENTO SEA X LAS SUMAS DE LOS MONTOS NOS DA LO QUE HAY EN MONTO EN TOTALPORPRODUCTO.
- EN ULTIMOPRECIO, DADO UN PRODUCTO X, EL MONTO QUE APARECE ES EL MISMO QUE ESTÁ EN VENTAS, BUSCANDO LA TUPLA CON EL PRODUCTO X CON LA FECHA MAS GRANDE PARA ESE PRODUCTO X.

Ej: VENTAS: $\{(A, 1, 100), (A, 2, 300), (B, 1, 150)\}$

ULTIMOPRECIO: $\{(A, 300), (B, 150)\}$

TOTALPORPRODUCTO: $\{(A, 400), (B, 150)\}$

ABS: Dado numero hoja del TAB con MJD.

- Si PRODUCTO \in VPP ent^e en alguno \in de VENTAS
- El PRODUCTO DE ALGUNA TUPLA \in SEQ DE VENTAS ESTA EN VPP
- EL PRODUCTO $\{X, \text{SEQTUPLA}(F, M \Rightarrow)\} \in$ VPP ESTA EN VENTAS EN ALGUNA DE LA SEQ (ojo) TUPLA
- LAS TUPLAS EN SEQ DE VENTAS, EL PRIMER \in VEN DE TUPLA ES KEY EN VPP Y LOS DEMAS REGISTROS ESTAN EN ALGUNA SECUENCIA BAJO ESA KEY EN VPP.

INREP:

PRED INREP(C: COMERCIOIMPL){

1. FECHAMONTOSDEPRODUCTOS SON POSITIVOS(C) ^
2. PRODUCTOSNETOTALPORPRODUCTOVALIDOS(C) ^
3. PRODUCTOSDEULTIMOPRECIO VALIDOS(C) ^
4. TOTALPORPRODUCTOES SUMAMONTOS VENTASPROD(C) ^
5. PRODUCTOULTIMOPRECIO TIENE LA MAYOR FECHA(C) ^

}

PRED FECHAMONTOSDEPRODUCTOS SON POSITIVOS(C: COMERCIOIMPL){

$(\forall e: (\text{PRODUCTO}, \text{FECHA}, \text{Monto})) (e \in C.\text{VENTAS} \rightarrow e_1 \wedge e_2 \geq 0)$

}

$\begin{matrix} \swarrow & \searrow \\ \text{FECHA} & \text{MONTO} \end{matrix}$

PRODUCTOS PRODUCTOS DELTORAL PRODUCTO VALIDOS (: CONECCIOMPL) {

($\forall p: \text{PRODUCTO}$) ($p \in C.\text{TOTAL_PRODUCTO} \rightarrow_L$

($\exists i: \langle \text{PRODUCTO}, \text{FECHA}, \text{PRECIO} \rangle$) ($i \in C.\text{VENTAS} \wedge i_0 = p$)))

}

PRODUCTOS PRODUCTOS DELTORAL PRODUCTO VALIDOS (: CONECCIOMPL) {

<P, n>

($\forall p: \text{PRODUCTO}$) ($\vec{i} \in C.\text{ULTIMO_PRECIO} \rightarrow_L$

($\exists i: \langle \text{PRODUCTO}, \text{FECHA}, \text{PRECIO} \rangle$) ($i \in C.\text{VENTAS} \wedge i_0 = p$)))

}

PRODUCTOS TOTAL PRODUCTO ESSUMA MONTOS VENTAS PRDD (: CONECCIOMPL) {

| C.VENTAS | - 1

($\forall p: \text{PRODUCTO}$) ($p \in C.\text{TOTAL_POR_PRODUCTO} \rightarrow_L$ $= \sum_{i=0}^{\text{C.TOTAL_PRODUCTO}[p]} \text{C.VENTAS}[i]_0 = p \text{ THEN}$

$i = 0 \quad \text{C.VENTAS}[i]_2 \text{ ELSE } 0$)

}

PRODUCTOS PRODUCTO ULTIMO PRECIO TIENE LA MAYOR FECHA (: CONECCIOMPL) {

($\forall p: \text{PRODUCTO}$) ($p \in C.\text{ULTIMO_PRECIO} \rightarrow_L (\exists i: \mathbb{Z}) (0 \leq i < |C.\text{VENTAS}| \wedge C.\text{VENTAS}[i]_0 = p \wedge$

($\forall j: \mathbb{Z}) ((0 \leq j < |C.\text{VENTAS}| \wedge C.\text{VENTAS}[j]_0 = C.\text{VENTAS}[i]_0 \wedge i \neq j) \rightarrow_L$

$C.\text{VENTAS}[j]_1 < C.\text{VENTAS}[i]_1) \wedge (\text{ULTIMO_PRECIO}[p]_1 = C.\text{VENTAS}[i]_2)$))

}

PARA TODO SOS PRODUCTOS EN ULTIMOPRECIO, EXISTE

EN VENTAS UNA KEY PARA ESE PRODUCTO Y

PARA LAS ÓRIGENES VENTAS DE ESE PRODUCTO, LA VENTA ES EL
MÁS EN LA MAYOR EN FECHA.

$$\text{Entonces } e_1 = (\cdot \text{VENTAS}[i])_2$$

ABSTRACCIÓN

PRED ABS ($c':\text{ComercioImpl}$, $c:\text{Comercio}$) {

1. PRODUCTOS EN VENTAS ESTA EN VENTAS POR PRODUCTO (c', c) \wedge

2. PRODUCTO VENTAS POR PRODUCTO ESTA EN TUPA VENTAS (c', c) \wedge

3. PFM VENTAS POR PRODUCTO ESTA EN VENTAS (c', c) \wedge

4. PFM VENTAS ESTA EN VPP (c', c)

}

PRED PRODUCTOS EN VENTAS ESTA EN VENTAS POR PRODUCTO ($c:\text{ComercioImpl}$, $c:\text{Comercio}$) {

($\forall e:(p,r,n)$) ($e \in c'.VENTAS \rightarrow_L e_0 \in c.\text{VENTAS POR PRODUCTO}$)

}

PRED PRODUCTO VENTAS POR PRODUCTO ESTA EN TUPA VENTAS ($c':\text{ComercioImpl}$, $c:\text{Comercio}$) {

($\forall p:\text{Producto}$) ($p \in c.\text{VENTAS POR PRODUCTO} \rightarrow_L (\exists i: \gamma_i) (0 \leq i < |c'.VENTAS| \wedge$
 $c'.VENTAS[i]_0 = p)$)

}

PRED PFM VENTAS POR PRODUCTO ESTA EN VENTAS ($c':\text{ComercioImpl}$, $c:\text{Comercio}$) {

($\forall p:\text{Producto}$) ($p \in c.\text{VENTAS POR PRODUCTO} \rightarrow_L$

($\forall i: \gamma_i | (0 \leq i < |c.\text{VENTAS POR PRODUCTO}|) \rightarrow_L$

($\exists j: \gamma_j | (0 \leq j < |c'.VENTAS| \wedge c'.VENTAS[i] = (p, c.\text{VENTAS POR PRODUCTO}[p][i]_0,$
 $c.\text{VENTAS POR PRODUCTO}[p][i]_1))$

}

PRED BE VENTAS EN VPP ($c':\text{ComercioImpl}$, $c:\text{Comercio}$) {

1) UF 7) PF M VENTAS ESTA EN VPP (C : COMERCIOIMPL, C : COMERCIO)

garantiz KEY

$(\forall i: \mathbb{Z}) (\exists j: \mathbb{Z}) (0 \leq i < |C.VENTAS| \rightarrow C.VENTAS[i]_0 \in C.VENTASPORPRODUCTO \wedge 0 \leq j < |C.VENTASPORPRODUCTO[C.VENTAS[i]_0]| \wedge C.VENTASPORPRODUCTO[C.VENTAS[i]_0]_0 = C.VENTAS[i]_1 \wedge C.VENTASPORPRODUCTO[C.VENTAS[i]_0]_1 = C.VENTAS[i]_2)$

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.

1 ALARMA

```
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)}}
}
```

A \xrightarrow{N} S
S $\xrightarrow{1}$ 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.

Leyendo entorno M-N.

INVREP:

• Son keys de SENSORES, EN EL CONJUNTO VALOR TIENEN ELEM'S QUS SON KEY EN ALARMAS.

SENDORES
↑

• Son keys de ALARMAS , EN EL CONJUNTO VALOR TIENEN ELEM'S QUS SON KEY EN SENSORES.

ALARMS
↑

• Son KEYS de SENSORES SON UNICAS .] Entre las implícitas xq

en Diccionario ; no ?

• Los KEYS de ALARMAS SON ÚNICAS

- Si tiene una alarmas, y los sensores no tienen la misma
que tiene la alarmas.

A: { 1, (2, 3) }

S: { 2, (1) }

S: { 3, (1) }

]

OBLIGATORIO

ALARMA, TODOS LOS ELEMENTOS DE LISTA SON KEY EN S Y ALARMA ESTA EN ALGUN INDICE DE VAL.

Lo mismo al revés.

PRED INVREP(P:PLANTAIMPL)

SENDORES EN ALARMAS VALIDOS(P) ^

ALARMAS EN SENDORES VALIDOS(P)



}

PRED SENDORES EN ALARMAS VALIDOS (P:PLANTAIMPL){

($\forall A: ALARMA$) ($A \in P. ALARMAS \rightarrow_L (\forall s: SENSOR) | S \in P. ALARMAS[A] \rightarrow_L S \in P. SENDORES \wedge A \in P. SENDORES[S]$)

}



PRED ALARMAS EN SENDORES VALIDOS (P:PLANTAIMPL){

($\forall s: SENSOR$) ($S \in P. SENDORES \rightarrow_L (\forall A: ALARMA) | A \in SENDORES[S] \rightarrow_L A \in S. ALARMAS \wedge S \in P. ALARMAS[A]$)

}



PRED SENSOER LLEGADA ALARMA (P:PLANTAIMPL){

($\forall A: ALARMA$) | $A \in P. ALARMAS \wedge (\forall s: SENSOR | (S \in P. ALARMAS[A] \rightarrow_L S \in P. SENDORES \wedge A \in P. SENDORES[S]))$

Lo hace así.

ABS:

\Leftrightarrow • TODA KEY DE VAL ALARMAS ESTÁ EN ALARMAS (DE HECHO |ALARMA|=|ALARMAS|)

\Leftarrow • TODA TUPLA EN SENDORES, PRIMER ELEMENTO ESTÁ EN KEY SENDORES Y 2DO EN

(OJAL ALARMA. DE HECHO, $|SENDORES[S]| = |TUPLA(S, A)|$)

KEYS
P mas
cont

=> • TODA KEY DE VAR SENSORES Y CADA ALARMA ESTA CONTENIDA EN SENSORES

PRED ABS(P¹: PLANTAIMPL, P: PLANTA){

ALARMAS EN PLANTA IMP VALIDAS (P¹, P) ^ ALARMAS EN PLANTA VALIDAS (P¹, P) ^

SENDORES EN PLANTA VALIDOS (P¹, P) ^ SENDORES EN PLANTA IMP VALIDOS (P¹, P)

}



PRED ALARMAS EN PLANTA IMP VALIDAS (P: PLANTAIMPL, P: PLANTA){

(VA: ALARMA) (A E P¹. ALARMAS ->_L A E P. ALARMAS)

}



PRED ALARMAS EN PLANTA VALIDAS (P¹: PLANTAIMPL, P: PLANTA){

(VA: ALARMA) | A E P. ALARMAS ->_L A E P¹. ALARMAS

}



PRED SENDORES EN PLANTA VALIDOS (P: PLANTAIMPL, P: PLANTA){

(VT: <SEASON, ALARMA>) | T E P. SENDORES ->_L (T₀ E P¹. SENDORES ^ T₁ E P¹. SENDORES[T₀])

}



PRED SENDORES EN PLANTA IMP VALIDOS (P¹: PLANTAIMPL, P: PLANTA){

(Vs: SENSOR) | S E P¹. SENDORES ->_L (VA: ALARMA) | A E P¹. SENDORES[S] ->_L (S, A) E P. SENDORES)

}



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> implementa 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.

- Escriba el invariant de representación y la función de abstracción para ambos casos (con y sin repetidos)
- ¿Cuál es más eficiente? Cuándo usaría cada una de las dos versiones?
- Escriba los algoritmos para las operaciones de **agregar** un elemento y **sacar** un elemento para ambas versiones
- 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

```
TAD ConjuntoAcotado<T> { -> EVAS = DATOS
    obs elems: conj<T>
    obs cap: int -> CAP MAXIMA (|DATOS|)
    proc conjVacio(c: int): ConjuntoAcotado<T>
        asegura {res.cap = c ^ res.elems = ()}
```

PRED SIN REPETICIONES | l: APPAREL{}{}

$(\forall i : \mathbb{Z})(0 \leq i < |L| \rightarrow (\forall j : \mathbb{Z})(0 \leq j < |L| \wedge L[i] = L[j]) \rightarrow i = j)$

}

SIN REP:

Módulo CONJACOTADOARR< T > IMPLEMENTA CONJACOTADO< T > {

VAR DATOS: ARRAY< T >

VAR LARGO: INT

PROC NUEVOCOTADOARR(): CONJACOTADOARR< T > {

 nro. DATOS := NEW ARRAY(0)

 nro. LARGO := 0

}

PRED INVREP(C: CONJACOTADOARR< T >) {

 SINREPETIDOS(C.DATOS) ^ C.LARGO ≤ |C.DATOS|

}

PRED ABS(C': CONJACOTADOARR< T >, C: CONJACOTADO< T >) {

 C.CAP = |C'.DATOS|

 ^

 ($\forall e : T$) ($e \in C.ELEMS \rightarrow e \in C'.DATOS$)

 ^

 ($\forall i : \mathbb{Z}$) ($0 \leq i < |C'.DATOS| \rightarrow C'.DATOS[i] \in C.ELEMS$)

}

PROC AGREGAR(inout C': CONJACOTADOARR< T >, in e: T) {

VAR i: INT := 0

IF (C'.LARGO = |C'.DATOS|) THEN:

 VAR DATA := NEW ARRAY< T >(C'.LARGO + 1)

 WHILE (i < C'.LARGO) DO:

 DATA[i] := C'.DATOS[i]

```

 $i := i + 1$ 
END WHILE
DATA[C'.LARGO] := e
C'.DATOS = DATA
ELSE
    C'.DATOS[C'.LARGO - 1] = e
ENDIF

C'.LARGO = C'.LARGO + 1
}

```

```

PROC ESTA(IN C': CONSACORTADODARR<T>, IN e:T):bool {
    VAR i:INT := 0
    VAR ESTA:BOOL := FALSE

    WHILE(i < C'.LARGO OR !ESTA) Do:
        IF(C'.DATOS[i] = e) THEN
            ESTA := TRUE
        ENDIF
        i := i + 1
    END WHILE

    RETURN ESTA
}

```

PROC SACAR(inout C': CONSACORTADODARR<T>, in e:T) {

REQUIERE: { C'.ESTA(C', e) }

VAR i:INT := 0

VAR j : INT := 0

VAR l : INT := $C'.LARGO - 1$

VAR ARR: ARRAY<+> = NEW ARRAY<+>(l)

WHILE ($j < l$) DO:

 IF ($C'.DATOS[j] \neq e$) THEN

 ARR[i] = $C'.DATOS[j]$

 i := i + 1

 ENDIF

 j := j + 1

ENDWHILE

$C'.DATOS = ARR$

$C'.LARGO = l$

}

$\{1, 2, 3, 4\}$ \log_2 2

$i := 0 \quad j := 0$

? $i = 2$? No. Ent $ARR[0] = C'.DATOS[0]$ $i := 1 \quad j := 1$

? $i = 2$? Si. Ent $i := 1 \quad j := 2$

? $i = 3$? No. Ent $ARR[1] = C'.DATOS[1]$ $i := 2 \quad j := 3$

? $i = 4$? No. Ent $ARR[2] = C'.DATOS[2]$ $i := 3 \quad j := 4$

FIN

CONREP:

Olaí hay UN PROBLEMA xq en CONJ NO se ocultan repetidos,
entonces se tiene que el TAB CONJ.

Lo tienen como de relaciones DATOS: ARRAY<+> CON

EVENTS: CONJUNTOS.

Pero CAP es TAB en Fija, y largo es IMP TAB pero si
LARGO ≤ CAP tiene que falso ↗:

LARGO: 4 DATOS: [1, 2, 3, 3] → ^{→ ARR} Cíclicamente lleno.
(AP: 4 EVENTS: {1, 2, 3} ↑
_{CONJ.} Cíclicamente no lleno.

Podriamos decir que se repite repeta pero el largo de los
x los NO repetidos, y los DATOS ESTAN EN EVENTS (NO SE
IMPONIA CANT.)

$$\begin{aligned} C'.\text{LARGO} &= \sum \text{DISTINTOS} && \left\{ \begin{array}{l} C.\text{DATOS} \in C.\text{EVENTS} \\ C.\text{EVENTS} \in C'.\text{DATOS} \end{array} \right. \\ C'.\text{AP} &= C.\text{CAP} \end{aligned}$$

PROBLEMA:

Al ofrecer, que LARGO sea lo mas bajo no me sirve xq no
tengo forma de saber si se llena el ARR o no.

MUY DIFÍCIL SIN OTRO UNA.

No se puede probar.

Módulo CONJACOTADODARR ↗ IMPLEMENTA CONJACOTADO ↗

VAR DATOS: ARRAY<T>

VAR LARGO: INT

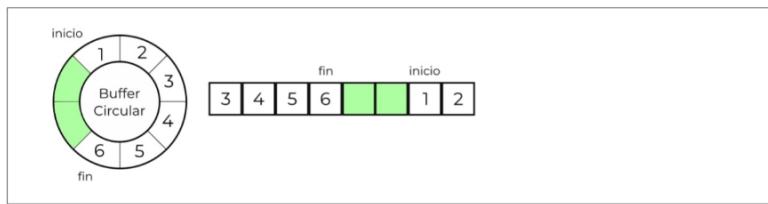
PROC NUEVOCONSACOTADOARR(): CONSACOTADOARR<T> {
 var. DATOS := NEW ARRAY(0)
 var. LARGO := 0
}

PRED INVREP(C: CONSACOTADOARR<T>) {
 C.LARGO ≤ |C.DATOS|
}

PRED ABS(C': CONSACOTADOARR<T>, C: CONSACOTADO<T>) {
 C.CAP = |C'.DATOS|
 ($\forall e: T$) ($e \in C.ELEMS \rightarrow_L e \in C'.DATOS$)
 ($\forall i: \mathbb{N}$) ($0 \leq i < |C'.DATOS| \rightarrow_L C'.DATOS[i] \in C.ELEMS$)
 $|C.ELEMS| = |\text{SINREP}(C'.DATOS)|$
}

}

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?

$$[] \equiv \text{Inici}o = \text{Fin} = 0$$

$$[1] \equiv \text{Fin} = 1, \text{Inici}o = 0$$

$$[1, 2] \equiv \text{Fin} = 2, \text{Inici}o = 0$$

$$[2] \equiv \text{Fin} = 1, \text{Inici}o = 1$$

$[2] \vdash \text{RETURN } 2 \quad \text{FIN} = 2, \text{INICIO} = 2$

INICIO no tiene sentido q sea MAYOR O FIN.

j debiese CREAR un NUEVO ARR o DESENCOLA? xq ly:

$[1, 2] \vdash \text{FIN} = 2, \text{INICIO} = 0$

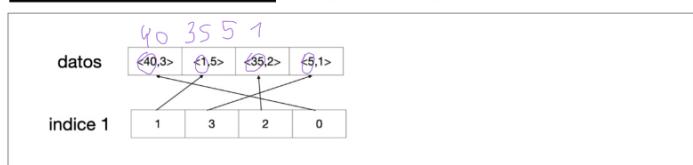
$[2] \vdash \text{FIN} = 2, \text{INICIO} = 1 \rightarrow$ se index xq 1st tiene 1 elem.

si DESENCOLA j q fija (no elem?)

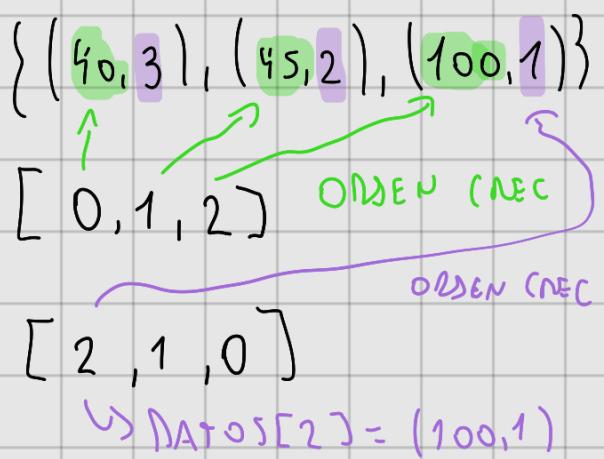
\hookrightarrow NO CAMBIA LONGITUD

Ejercicio 5. Una 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 particular, diferente al orden original.

Imagine un conjunto de tuplas con los componentes. Algunas veces vamos a querer buscar (rápido) por la primera componente y a veces por la segunda. Podríamos guardar los datos en si 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 les denominan 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**



BUSCAR POR SEGUNDA (0) = i=0 \in DATOS [2]

L) OBS: Noch in manchen Industrie-EUEN ist in manchen EU-EN NO COUNTRY.

E.g.: BPS(2) = INDO, DATOS(2) = no coincides

INVEP:

A B S :

1. TODO ES UN DATOS DE INDICES <=> ESTA EN ALGUN INDICE DE DATOS DE INDICESIMPL
 2. TODA IUPA EN ARRDE DATOS DE INDICESIMPL & DATOS DE INDICES <= >
 3. TODOS ELEMENTOS DE IND1 DE INDICES <= > ESTA EN IND1 EN INDICESIMPL <= >
Y EN MISMO INDICES

4. TODO ELEMENTO DE IND₁ DE INDICES<T> ES IND₁ EN INDICES<S>
Y EN MISMO INDICE
5. TODO ELEMENTO DE IND₂ DE INDICES<T> ES IND₂ EN INDICES<T>
Y EN MISMO INDICE
6. TODO ELEMENTO DE IND₂ DE INDICES<T> ES IND₂ EN INDICES<S>
Y EN MISMO INDICE

TAD INDICES<T> {

OBS DATOS: CONJ<T x T>

OBS IND₁: $\text{seq } \langle T \rangle$ (Primeros comp.)

OBS IND₂: $\text{seq } \langle T \rangle$

(Segundos comp.)

}

MÓDULO INDICES<T> IMPLEMENTA INDICES<T> {

\rightarrow si T es INT, Ordenar Tupla con ins

VAR DATOS: ARRAY<TUP<T,T>>

VAR IND₁: ARRAY<T>

VAR IND₂: ARRAY<T>

VAR LARGO: INT

PROC NUEVOINDICES<T>(): INDICES<T> {

Var. DATOS := NEW ARRAY<T>(0)

Var. IND₁ := NEW ARRAY<T>(0)

Var. IND₂ := NEW ARRAY<T>(0)

Var. LARGO := 0

}

PRED INUREP(i: INDICES<T>){

1. |i.DATOS| = |i.IND₁| = |i.IND₂| \wedge i.LARGO \leq i.DATOS \wedge

CADA V_i \in 0 A LARGO
SE D_i HAGA
PRIMEROS DE
VUELTA

INM SON PRIMEROS COMPONENTES VALIDOS(i)

\wedge

IND₂ SON SEGUNDOS COMPONENTES VALIDOS(i)

PRED IND₁ SON PRIMEROS COMPONENTES VALIDOS (in: INDICES<T>){

$(\forall i: TL)(0 \leq i < |\text{in.IND}_1|-1 \Rightarrow \text{in.IND}_1[i] < |\text{in.IND}_1|) \wedge \text{in.DATOS}[\text{in.IND}_1[i]]_0 < \text{in.DATOS}[\text{in.IND}_1[i+1]]_0$

} VAL ES INDIVIDOS
2. VAL ES OBJETOS
3.
PRED IND2SONSEGUNDOSCOMPONENTESVAJINOS (in: INDICESIMPL<?>)
 $(\forall i : \mathbb{Z})(0 \leq i < |in.IND2| \rightarrow_L \underbrace{in.IND2[i] < |in.DATOS|}_{\text{VAL ES INDIVIDOS}} \wedge \underbrace{\text{in.DATOS}[in.IND2[i]], <\text{INDATOS}[in.IND2[i+1]]}_{\text{VAL ES (OBJETOS}}}$
 } 2. 4.

PRED ABS (im: INDICESIMPL<?>, in: INDICES<?>) {

$|in.IND1| = |im'.IND1|$ } *1 ^
 $|in.IND2| = |im'.IND2|$ } ^
 $|in.DATOS| = |im'.DATOS|$ } ^
 DATOSINDICESESTANENINDICESIMPL(im', im) ^
 DATOSINDICESIMPLESTANENINDICES(im', im) ^
 IND1 INDICESIMPL IGUAL A INDICES (In', In) ^
 IND2 INDICESIMPL IGUAL A INDICES (Im', Im) ^
 } ^

PRED DATOSINDICESESTANENINDICESIMPL (im': INDICESIMPL<?>, in: INDICES<?>) {

$(\forall e : \langle \tau, \tau \rangle)(e \in |im'.DATOS| \rightarrow_L (\exists i : \mathbb{Z})(0 \leq i < |im'.DATOS| \wedge im'.DATOS[i] = e))$
 }

PRED DATOSINDICESIMPLESTANENINDICES (in': INDICESIMPL<?>, in: INDICES<?>) {

$(\forall i : \mathbb{Z})(0 \leq i < |in.DATOS| \rightarrow_L in.DATOS[i] \in |in'.DATOS|)$
 }

PRED IND1 INDICESIMPL IGUAL A INDICES (in': INDICESIMPL<?>, in: INDICES<?>) {

$(\forall i : \mathbb{Z})(0 \leq i < |in.IND1| \rightarrow_L in.IND1[i] = im'.IND1[i])$
 }

PRED IND2 INDICESIMPL IGUAL A INDICES (in': INDICESIMPL<?>, in: INDICES<?>) {

$(\forall i : \mathbb{Z})(0 \leq i < |in.IND2| \rightarrow_L in.IND2[i] = im'.IND2[i])$
 }

NOTA: Lo hace verificando que ambos tienen los IND1 e IND2 iguales *1.

¿En qué caso? Ya que la primera de tener 2 A de Ojo q tiene misma ESTRUCTURA y misma CANT.

IND VACIOS en IND1.

↗

PROC BUSCARPORPRIMERA (IN IM: INDICESIMPRES, IN IND1: T) : TUPLA(T,T) {

VAR REN: TUPLA(T,T) := NULL

VAR i: INT := 0

VAR ENCONTRADO: BOOL := false

WHILE (i < im.LARGO || !ENCONTRADO) DO:

IF (IND = = i) THEN

REN = IM.DATOS[im.IND[i]]

ENDIF

i := i + 1

END WHILE

RETURN REN

}

IND VACIOS en IND2.

↗

PROC BUSCARPORSEGUNDA (IN IM: INDICESIMPRES, IN IND2: T) : TUPLA(T,T) {

VAR REN: TUPLA(T,T) := NULL

VAR i: INT := 0

VAR ENCONTRADO: BOOL := false

WHILE (i < im.LARGO || !ENCONTRADO) DO:

IF (IND = = i) THEN

REN = IM.DATOS[im.IND2[i]]

ENDIF

i := i + 1

END WHILE

RETURN res
}

OBS: Pueden generarse el buscar, y mandar como parámetro solo el bucle para no tener 2 func iguales por BUCLE

PROC AGREGAR(indout IN: INDICESIMP(τ), IN e: TUPACT, τ)

D: $\{(40, 3), (45, 2), (100, 1)\}$ AGG (50, 2)

IND1: [0, 1, 2]

IND2: [2 1 0]

D: $\{(40, 3), (45, 2), (100, 1), (50, 2)\}$

IND1: [0, 1, 2, 3]

IND2: [2, 1, 3, 0]

REORDENAR AL SACAR Y AGREGAR 

PREGUNTA: 4 y 5 son realmente omisiones y difíciles.

