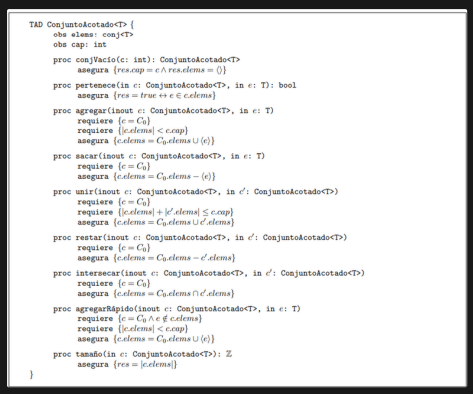
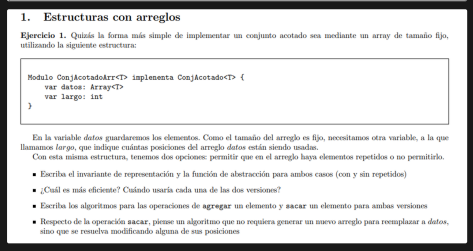
Práctica 5



```
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 intersecar(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|}
}
```



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 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

a)

Sin repetidos:

pred InvRep(c : ConjAcotadoArr<T>){

$$0 \leq c.largo \leq c.datos.Length \land sinRepetidos(c.datos)$$

}

pred sinRepetidos(d : Array<T>)
{

$$(\forall t : T)(t \in d \rightarrow Apariciones(t, d) = 1)$$

}

aux Apariciones(t : T, d : Array<T>) : $\mathbb{Z}$ = $\sum_{i=0}^{d.Length-1}$ if d[i] = t then 1 else 0 fi;

aux FunAbs(cArr : ConjAcotadoArr<T>) : ConjAcotado<T> {

$c : ConjAcotado < T > |$

$c.cap = cArr.datos.Length \ \wedge$

$mismosElementos(cArr.datos[0..cArr.largo], c.elems)$

}

pred mismosElementos(d : Array<T>, e : conj<T>) {

$(\forall t : T)(t \in d \leftrightarrow t \in e)$

}

Con repetidos:

pred InvRep(c : ConjAcotadoArr<T>){

$0 \leq c.largo \leq c.datos.Length$

}

aux FunAbs(cArr : ConjAcotadoArr<T>) : ConjAcotado<T> {

$c : ConjAcotado < T > |$

$c.cap = cArr.datos.Length \ \wedge$

$mismosElementos(cArr.datos[0..cArr.largo], c.elems)$

}

b)

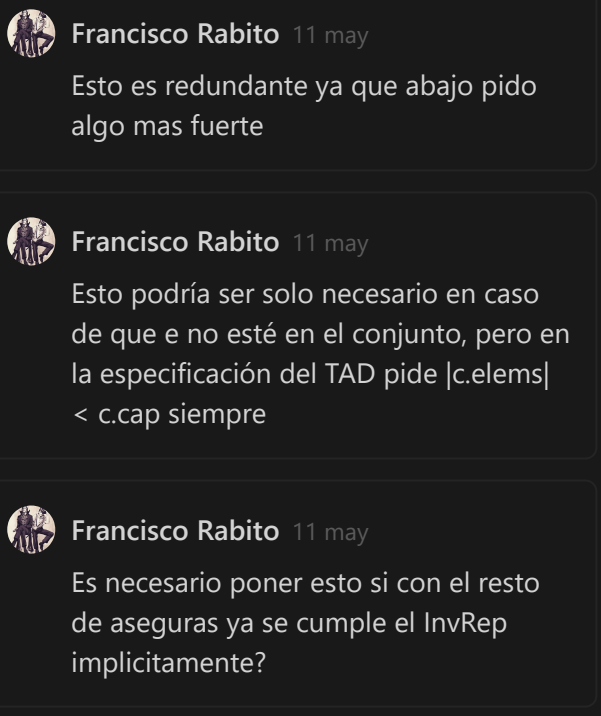
La versión con repetidos tiene la libertad de poder insertar elementos sin verificar si ya estan o no en el arreglo (siempre y cuando haya espacio), pero a la hora de estar lleno tiene que buscar que, por un lado el elemento no esté, y por otro que elementos estan repetidos para saber cual puede quitar y asi insertar uno nuevo.

La versión se tiene que asegurar que cada vez que se quiere insertar un elemento no esté en el array.

c y d)

Es necesario pedir el InvRep en los requiere si se que en ninguno de mis procs puede salir un ConjAcotadoArr<T> que no lo cumpla? Si

Sin repetidos:



proc agregar (inout c : ConjAcotadoArr<T>, in e : T)

requiere{$c = C_0$}

requiere {$InvRep(c)$}

requiere {$0 \leq c.largo < c.datos.Length$}

asegura {$InvRep(c)$}

asegura {$FunAbs(c).elems = FunAbs(C_0).elems \cup \{e\}$}

asegura {$e \notin C_0.datos[0..C_0.largo] \rightarrow c.largo = C_0.largo + 1$}

asegura {$e \in C_0.datos[0..C_0.largo] \rightarrow c.largo = C_0.largo$}

{

```
var i : int i := 0 while
i < c.largo &&
c.datos[i] != e { i := i
+ 1 } if i = c.largo
then c.datos[i] := e
c.largo := c.largo + 1
else skip endif return c
//hay que poner el
return? o es solo cuando
el proc devuelve algo?
```

}

proc sacar(inout c : ConjAcotadoArr<T>, in e : T)

requiere {$c = C_0$}

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {
$FunAbs(c).elems = FunAbs(C_0).elems - \{e\}$}

asegura {$c.largo = C_0.largo - Apariciones(e, C_0.datos[0..C_0.largo])$
}

{

```
var i : int i := 0 while
i < c.largo &&
c.datos[i] != e { i := i
+ 1 } if i != c.largo
then c.datos[i] :=
c.datos[c.largo-1]
c.largo := c.largo - 1
else skip endif
```

}

Con repetidos:

proc agregar (inout c : ConjAcotadoArr<T>, in e : T)

requiere {$c = C_0$}

requiere {$InvRep(c)$}

requiere {$|FunAbs(c).elems| < c.datos.Length$}

asegura {$FunAbs(c).elems = FunAbs(C_0).elems \cup \{e\}$}

asegura {$C_0.largo < C_0.datos.Length \rightarrow c.largo = C_0.largo + 1$}

asegura {$C_0.largo = C_0.datos.Length \rightarrow c.largo = C_0.largo$}

{

```
if c.largo <
c.datos.Length then
c.datos[c.largo] := e
c.largo := c.largo + 1
else var i : int var j :
int var aux: int aux :=
-1 i := 0 while i <
c.largo { j := 0 while j
< c.largo { if i != j &&
c.datos[i] = c.datos[j]
then aux := j else skip
endif j := j + 1 } i :=
i + 1 } c.datos[aux] :=
e //puedo saber que aux
va a ser != -1 porque mi
requiere pide |elems|
<cap endif
```

}

Francisco Rabito 12 may

Es necesario este asegura? o con el de arriba ya estaría diciendo eso basicamente?

proc sacar(inout c : ConjAcotadoArr<T>, in e : T)

requiere {$c = C_0$}

requiere {$InvRep(c)$}

asegura {$FunAbs(c).elems = FunAbs(C_0).elems - \{e\}$}

asegura {$c.largo = C_0.largo - Apariciones(e, C_0.datos[0..C_0.largo])$}

{

```
var i : int var j : int
i := 0 j := c.largo - 1
while j >= i { if
c.datos[j] = e then j :=
j - 1 else if c.datos[i]
= e then c.datos[i] :=
c.datos[j] j := j - 1 i
:= i + 1 else i := i + 1
endif endif } c.largo :=
j + 1
```

}

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`.

Modulo PilaArr<T> implementa Pila<T> {

var datos : Array<T>

var largo : int

}

pred InvRep(p : PilaArr<T>) {

$0 \leq p.largo \leq p.datos.Length$

}

aux FunAbs(pArr : PilaArr<T>) : Pila<T> {

$p : Pila < T > | |p.s| = pArr.largo \ \wedge (\forall i : \mathbb{Z})(0 \leq i < pArr.largo \rightarrow_L p.s[i] = pArr.datos[i])$

}

proc apilar(inout p : PilaArr<T>, in e : T)

requiere $\{p = P_0\}$

requiere $\{InvRep(p)\}$

asegura $\{InvRep(p)\}$

asegura $\{FunAbs(p).s = concat(FunAbs(P_0).s, < e >)\}$

asegura $\{p.largo = P_0.largo + 1\}$

{

Francisco Rabito 12 may (editado)

Como |s| = cantEnUso(enUso) no hace falta pedir la vuelta.
Se puede pedir lo mismo de forma mas sencilla?

```
    if p.largo !=
    p.datos.Length then
    p.datos[p.largo] := e
    else var arr : Array<T>
    var i : int arr := new
    Array<T>(p.largo * 2) i
    := 0 while i < p.largo {
    arr[i] := p.datos[i] i
    := i + 1 } arr[i] := e
    p.datos := arr endif
    p.largo := p.largo + 1
```

}

proc desapilar(inout p : PilaArr<T>) : T

requiere {$p = P_0$}

requiere {$p.largo > 0$}

requiere {$InvRep(p)$}

asegura {$InvRep(p)$}

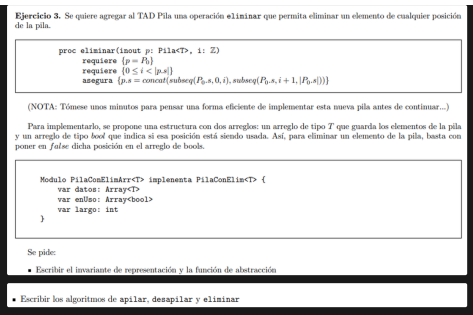
asegura {$FunAbs(p).s = subseq(FunAbs(P_0).s, i, p.largo - 1)$}

asegura {$p.largo = P_0.largo - 1$}

{

```
    p.largo := p.largo - 1
    return p.datos[p.largo]
```

}



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 = P_0}
    requiere {0 ≤ i < |p.s|}
    asegura {p.s = concat(subseq(P_0.s, 0, i), subseq(P_0.s, i + 1, |P_0.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> implementa 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`

a)

pred InvRep(p : PilaConElimArr<T>) {

$$p.datos.Length = p.enUso.Length \land 0 \leq p.largo \leq p.datos.Length$$

}

aux FunAbs(pArr : PilaConElimArr<T>) : PilaConElim<T> {

$$p : PilaConElim < T > |$$

$$|p.s| = cantEnUso(pArr.enUso[0..p.largo]) \ \land_L$$

$$soloEnUsoYEnOrden(p.s, pArr.datos[0..p.largo], pArr.enUso[0..p.largo])$$

}

aux cantEnUso(enUso : Array<Bool>) : $\mathbb{Z}$ = $\sum_{i=0}^{enUso.Length-1}$ if enUso[i] = true then 1 else 0 fi;

Francisco Rabito 12 may

En el segundo subseq se puede romper si te queda que i+1 = |s|?

pred soloEnUsoYEnOrden(s : seq<T>, d : Array<T>, enUso : Array<bool>) {

$(\forall i : \mathbb{Z})(0 \leq i < d.Length \rightarrow_L (enUso[i] = true \rightarrow_L d[i] = s[i - (i - cantEnUso(enUso[0..i]))]))$

}

Francisco Rabito 12 may

Para resolver este ejercicio asumí que en la imágen fin e inicio estaban mal y tendrían que ir al revés. Porque sentía que según la descripción del ejercicio no tiene sentido como está en la imágen

b)

proc apilar(inout p : PilaConElimArr<T>, in e : T)

requiere {$p = P_0$}

requiere {$InvRep(p)$}

asegura {$InvRep(p)$}

asegura {$FunAbs(p).s = concat(FunAbs(P_0).s, < e >)$}

{

Francisco Rabito 12 may

```
    if p.largo !=
    p.datos.Length then
    p.datos[p.largo] := e
    p.enUso[p.largo] := true
    else var arr : Array<T>
    var arrUso : Array<bool>
    var i : int arr := new
    Array<T>(p.largo * 2)
    arrUso := new
    Array<bool>(p.largo * 2)
    i := 0 while i < p.largo
    { arr[i] := p.datos[i]
    arrUso[i] := p.enUso[i]
    i := i + 1 } arr[i] := e
    arrUso[i] := true
    p.datos := arr p.enUso =
    arrUso endif p.largo :=
    p.largo + 1
```

El Array lo hago +1 del cap porque sino no veo manera de distinguir entre un array lleno de uno vacio

}

proc desapilar(inout p : PilaConElimArr<T>) : T

requiere {$p = P_0$}

requiere {$|FunAbs(p).s| > 0$}

requiere {$InvRep(p)$}

asegura {$InvRep(p)$}

asegura {$FunAbs(p).s = subseq(FunAbs(P_0).s, i, |FunAbs(P_0).s| - 1)$}

{

```
    var i : int i :=
    p.largo-1 while
    p.enUso[i] = false { i
    := i - 1 } p.largo := i
    return p.datos[i]
```

}

proc eliminar(inout p : PilaConElimArr<T>, in i : $\mathbb{Z}$)

requiere {$p = P_0$}

requiere {$0 \leq i < |FunAbs(p).s|$}

requiere {$InvRep(p)$}

asegura {$InvRep(p)$}

asegura {$FunAbs(p).s = concat(subseq(FunAbs(P_0).s, 0, i), subseq(FunAbs(P_0).s, i + 1, |FunAbs(P_0).s|))$}

{

```
    var j : int j := 0 while
    j <= i { if p.enUso[j] =
    false then i := i + 1
    else skip endif j := j +
    1 } p.enUso[i] = false
```

}

[2,4,4,2,6]

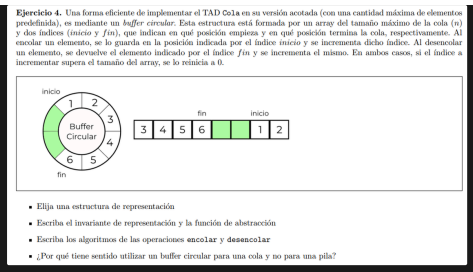
[false,false,true,false,true]

[4,6]

Francisco Rabito 12 may

Esto depende de como se interprete inicio y fin de una Pila. Podría ser al revés

Francisco Rabito 12 may



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?

Asumo que las tuplas son de tipo int. No sabría como decir que un tipo paramétrico sea comparable en orden

a)

```
Modulo ColaAcotadaArr<T>
implementa ColaAcotada<T> {
    var datos : Array<T>
    var inicio : int
    var fin : int
}
```

b)

```
pred InvRep(c :
ColaAcotadaArr<T>) {
    0 ≤ c.inicio ≤
    c.datos.Length ∧ 0 ≤
    c.fin < c.datos.Length
}
```

aux FunAbs(cArr : ColaAcotadaArr<T>) : ColaAcotada<T> {

$c : ColaAcotada < T >$
$|$
$c.cap =$
$cArr.datos.Length -$
$1 \ \wedge$
$cArr.inicio <$
$cArr.fin \rightarrow_L c.s =$
$concat(cArr.datos[cArr.fin..cArr.datos.Length], cArr.datos[0..cArr.inicio]) \ \wedge$

$cArr.inicio \geq$
$cArr.fin \rightarrow_L c.s =$
$cArr.datos[cArr.fin..cArr.inicio]$

}

Francisco Rabito 12 may
Es necesario usar los ifs anidados así? o puedo usar else if?

c)

proc encolar(inout c : ColaAcotadaArr<T>, in e : T)

requiere {$c = C_0$}

requiere {
$FunAbs(c).cap >$
$|FunAbs(c).s|$}

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$FunAbs(c).s =$
$concat(FunAbs(C_0).s, <$
$e >)$}

{

```
    c.datos[inicio] := e
    c.inicio := c.inicio + 1
    if c.inicio =
    c.datos.Length then
    c.inicio := 0 else skip
    endif
```

}

proc desencolar(inout c : ColaAcotadaArr<T>) : T

requiere {$c = C_0$}

requiere {$|FunAbs(c).s| > 0$}

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$FunAbs(c).s = subseq(FunAbs(C_0).s, 1, |FunAbs(C_0).s|)$}

asegura {$res = C_0.datos[C_0.fin]$}

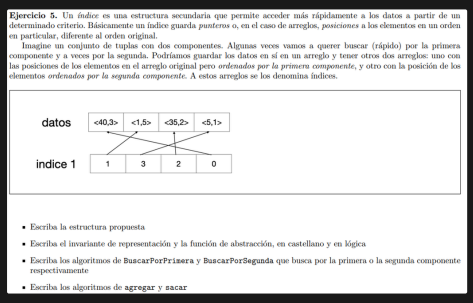
{

```
    var e : T e :=
    c.datos[c.fin] c.fin :=
    c.fin + 1 if c.fin =
    c.datos.Length then
    c.fin := 0 else skip
    endif return e
```

}

d)

Porque al usarlo en una Cola podes estar moviendo el Array de “lugar”, pero sin tener que crear otro. En cambio en una Pila no tendría mucho sentido hacer eso, ya que normalmente inicializarias la variable fin en 0 y se quedaría ahí para siempre, porque ni el metodo apilar ni el desapilar afectarían el fin, solo el inicio. En cambio en una Cola agregas por un lado y sacas por otro, lo cual puede hacer que quieras aumentar de tamaño hacia la derecha o disminuir desde la izquierda, y eso haría que se vaya moviendo de lugar en el Array.



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`

a)

Modulo ConjTuplasArr implementa ConjTuplas {

    var datos : Array<<int,int>>

    var indice0 : Array<int>

    var indice1 : Array<int>

    var largo : int

}

b)

En el Invariante de Representación quiero pedir que el largo esté en rango, que los 3 arrays tengan el mismo tamaño y que los arrays de los indices estén ordenados.

pred InvRep(c : ConjTuplasArr) {

$$0 \leq c.largo \leq c.datos.Length \ \wedge$$

$$c.datos.Length = c.indice0.Length = c.indice1.Length \ \wedge$$

$$tieneIndices(c.indice0[0..c.largo]) \wedge tieneIndices(c.indice1..c.largo) \ \wedge_L$$

$$estaOrdenado(c.indice0[0..c.largo], c.datos[0..c.largo], 0) \ \wedge$$

$$estaOrdenado(c.indice1[0..c.largo], c.datos[0..c.largo], 1)$$

}

pred tieneIndices(arr : Array<int>) {

$$(\forall k : \mathbb{Z})(k \in arr \leftrightarrow 0 \leq k < arr.Length)$$

}

pred estaOrdenado(indice : Array<int>, d : Array<<int,int>>, i : int) {

$$(\forall j : \mathbb{Z})(0 \leq j < d.Length - 1 \rightarrow_L d[indice[j]]_i \leq d[indice[j+1]]_i)$$

}

En la Función de Abstracción simplemente tengo que pedir que los elementos que están en la subseq hasta largo estén en los elementos del conjunto que quiero retornar.

aux FunAbs(cArr : ConjTuplasArr) : ConjTuplas {

$c : ConjTuplas|$

$(\forall t :< int, int >)(t \in c.elems \leftrightarrow t \in cArr.datos[0..c.largo])$

}

c)

proc BuscarPorPrimera(in c : ConjTuplasArr, in e : int) : bool

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$res = true \leftrightarrow (\exists t :< int, int >)(t_0 = e \wedge t \in FunAbs(c))$}

{

Francisco Rabito 12 may

Si esta función tiene aliasing no haría falta el return

```
    if c.datos.Length = 0 || e
    c.datos[c.indice0[0]][0] th
    return false else if c.datc
    = 1 then return
    c.datos[c.indice0[0]][0] =
    if e >=
    c.datos[c.indice0[c.indice0
    1]][0] return
    c.datos[c.indice0[c.indice0
    1]][0] = e else var low : i
    mid : int var high : int lc
    high := c.indice0.Length-1
    low+1 < high &&
    c.datos[indice0[low]][0] !=
    := low + (high-low)/2 if
    c.datos[indice0[mid]][0] <=
    low := mid else high := mic
    return c.datos[indice0[low]
    endif endif endif
```

}

proc BuscarPorSegunda(in c : ConjTuplasArr, in e : int) : bool

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$res = true \leftrightarrow (\exists t :< int, int >)(t_1 = e \wedge t \in FunAbs(c))$}

{

```
    if c.datos.Length = 0 || e
    c.datos[c.indice1[0]][1] th
    return false else if c.datc
    = 1 then return
    c.datos[c.indice1[0]][1] =
    if e >=
    c.datos[c.indice1[c.indice1
    1]][1] return
    c.datos[c.indice1[c.indice1
    1]][1] = e else var low : i
    mid : int var high : int lc
    high := c.indice1.Length-1
    low+1 < high &&
    c.datos[indice1[low]][1] !=
    := low + (high-low)/2 if
    c.datos[indice1[mid]][1] <=
    low := mid else high := mic
    return c.datos[indice1[low]
    endif endif endif
```

}

d)

proc agregar(inout c : ConjTuplasArr, in e : <int,int>)

requiere {$c = C_0$}

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$FunAbs(c).elems = FunAbs(C_0).elems \cup < e >$}

{

Francisco Rabito 13 may

Como debería usar la llamada al proc longitud?

Francisco Rabito 13 may

puedo obtener la posicion i de esta forma? o debería usar obtener()?

Francisco Rabito 17 may

Creo que lo correcto sería usar la funcion setear() o algo similar

```
if c.largo =
c.datos.Length then var d
: Array<<int,int>> var
ind0 : Array<int> var
ind1 : Array<int> var i :
int d := new
Array<<int,int>>(c.largo
* 2) ind0 := new
Array<int>(c.largo * 2)
ind1 := new Array<int>
(c.largo * 2) i := 0
while i < c.largo { d[i]
:= c.datos[i] ind0[i] :=
c.indice0[i] ind1[i] :=
c.indice1[i] i := i + 1 }
c.datos := d c.indice0 :=
ind0 c.indice1 := ind1
else skip endif
c.datos[c.largo] := e
indice0 :=
ordenarIndices(c.indice0,
c.datos, c.largo, e[0])
indice1 :=
ordenarIndices(c.indice1,
c.datos, c.largo, e[1])
c.largo := c.largo + 1
```

```
    function
    ordenarIndices(indice :
    Array<int>, datos :
    Array<<int,int>>, largo
    : int, eX : int) :
    Array<int>{ var arr :
    Array<int> var i : int
    arr := new Array<int>
    (datos.Length) i := 0
    while i < largo &&
    datos[indice[i]][0] <
    eX{ arr[i] := indice0[i]
    i := i + 1 } arr[i] :=
    largo i := i + 1 while i
    < largo + 1{ arr[i] :=
    indice[i-1] i := i + 1 }
    return arr }
```

}

proc sacar(inout c : ConjTuplaArr, in e : <int,int>)

requiere {$c = C_0$}

requiere {$InvRep(c)$}

asegura {$InvRep(c)$}

asegura {$FunAbs(c).elems = FunAbs(C_0).elems - < e >$}

{

```
var i : int var j : int i
:= 0 j := c.largo - 1
while j >= i { if
c.datos[j] = e then j :=
j - 1 else if c.datos[i]
= e then c.datos[i] :=
c.datos[j] j := j - 1 i
:= i + 1 else i := i + 1
endif endif } c.largo :=
j + 1 c.indice0 :=
indiceOrdenado(c.indice0,
c.datos, c.largo, 0)
c.indice1 :=
indiceOrdenado(c.indice1,
c.datos, c.largo, 1)
```

Francisco Rabito 13 may

Debería escribir esto así porque es una implementación Diccionario y no un dict?

Francisco Rabito 17 may

Si, es mas correcto escribirlo asi

```
function indiceOrdenado(indice : Array<int>, datos : Array<<int,int>>, largo : int, k : int) : Array<int>{ var i : int var j : int //acomodo todos los indices que me interesan en el rango hasta largo i := 0 while i < largo{ indice[i] := i } //ordeno los indices hasta largo i := 0 while i < largo{ j := i + 1 while j < largo{ if datos[indice[j]][k] < datos[indice[i]][k] then indice[i] := indice[i] + indice[j] indice[j] := indice[i] - indice[j] indice[i] := indice[i] - indice[j] else skip endif j := j + 1 } i := i + 1 } return indice } swap sin aux a = a + b b = a - b a = a - b
```

}

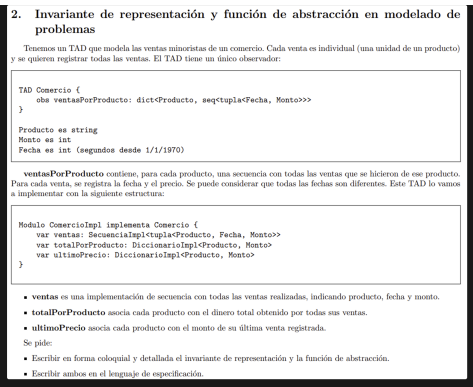
Ejemplo:

largo = 7

datos[k] = [5,4,6,4,7,2,8,x,x]

indices = [0,1,2,3,4,5,6,x,x] (sin ordenar)

indices = [5,3,1,0,2,4,6,x,x] (ordenados)



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.

a)

En el invariante de representación voy a querer decir que todas las fechas de ventas son distintas, que totalPorProducto y ultimoPrecio tienen las mismas claves, que el monto de un producto en totalPorProducto es la sumatoria de los montos en ventas de ese producto y que hay un producto en ventas, si y solo si está en totalPorProducto (por ende también en ultimoPrecio). También quiero pedir que todos los montos y fechas de ventas sean positivos (≥ 0). Aparte tengo que pedir que ultimoPrecio de un producto tenga el monto de ventas con la fecha mas alta.

En la función de abstracción quiero pedir que todos los productos que están en el obs ventasPorProducto están en ventas y viceversa. Por otro lado querré decir que para todos los productos que estén en el obs ventasPorProducto existe una seq tal que están todas las tuplas que contengan el producto en ventas.

b)

Invariante de Representación:

pred InvRep(c : ComercioImpl){

$$todasLasFechasDistintas(ventas) \ \wedge$$

$$todosLasFechasPositivas(ventas) \ \wedge$$

$$todosLosMontosPositivos(ventas) \ \wedge$$

$$totalEsSumaDeMontos(ventas, totalPorProducto) \ \wedge$$

$$ultimoPrecioTieneUltimoMonto(ventas, ultimoPrecio) \ \wedge$$

$$iffEstaEnVentasEstaEnTotalYUltimo(ventas, totalPorProducto, ultimoPre$$

}

```
pred todasLasFechasDistintas(v
:
SecuenciaImpl<tupla<Producto
,Fecha,Monto>>){
    (∀t : tupla <
    Producto, Fecha, Monto >
    )(t ∈ v →
    AparicionesFecha(t_1, v) =
    1)
}


aux AparicionesFecha(f : Fecha,
v :
SecuenciaImpl<tupla<Producto
,Fecha,Monto>>) : ℤ =
∑_{i=0}^{v.longitud()−1} if v[i]_1 = f
then 1 else 0 fi;


pred todasLasFechasPositivas(v
:
SecuenciaImpl<tupla<Producto
,Fecha,Monto>>){
    (∀t : tupla <
    Producto, Fecha, Monto >
    )(t ∈ v → t_1 ≥ 0)
}


pred
todosLosMontosPositivos(v :
SecuenciaImpl<tupla<Producto
,Fecha,Monto>>){
    (∀t : tupla <
    Producto, Fecha, Monto >
    )(t ∈ v → t_2 ≥ 0)
}
```

pred totalEsSumaDeMontos(v : SecuenciaImpl<tupla<Producto,Fecha,Monto>>, total : DiccionarioImpl<Producto,Monto>){

$(\forall p : Producto)(p \in total \rightarrow_L (\exists s : seq < Monto >)(sonMontos(p, v, s) \wedge total[p] = \sum_{i=0}^{|s|} s[i]))$

}

pred sonMontos(p : Producto, v : SecuenciaImpl<tupla<Producto,Fecha,Monto>>, s : seq<Monto>){

$(\forall m : Monto)(AparicionesSeq(m, s) = AparicionesMontoPorProducto(m, v, p))$

}

aux AparicionesMontoPorProducto(m : Monto, v : SecuenciaImpl<tupla<Producto,Fecha,Monto>>, p : Producto) : $\mathbb{Z}$ =

$\sum_{i=0}^{v.longitud()-1}$ if $v[i]_0 = p \wedge v[i]_2 = m$ then 1 else 0 fi;

aux AparicionesSeq(e : T, s : seq<T>) : $\mathbb{Z}$ = $\sum_{i=0}^{|s|-1}$ if s[i] = e