**Algoritmos y Estructuras de Datos**

Guía Práctica 3
**Especificación de TADs**
Segundo Cuatrimestre 2025

*Departamento de Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*

En esta práctica, la consigna "especificar en forma completa el TAD..." implica:

- Definir las operaciones y sus parámetros
- Definir los observadores
- Especificar Todas las operaciones incluyendo la creación y la igualdad
- Especificar Todos los predicados y auxiliares que sean necesarios

## 3.1. TADs introductorios

**Ejercicio 1.** Especificar en forma completa el TAD NumeroRacional que incluya las operaciones aritméticas básicas (suma, resta, división, multiplicación) y la operación de *igualdad* que dados dos números racionales devuelva verdadero si son iguales.

***Solución***

*– Para este ejercicio pueden asumir que ÷ es división entera.*

TAD NumeroRacional {

obs numerador : $\mathbb{Z}$
obs denominador : $\mathbb{Z}$

proc nuevoNumeroRacional(in $n, d : \mathbb{Z}$) : $NumeroRacional$ {
requiere { $d \neq 0$ }
asegura { $res.numerador = n \land res.denominador = d$ }
}

proc igualdad(in $n_1, n_2 : NumeroRacional$) : bool {
asegura { $res = true \leftrightarrow n_1.numerador \div n_1.denominador = n_2.numerador \div n_2.denominador$ }
}

proc suma(in $n1, n2 : NumeroRacional$) : $NumeroRacional$ {
requiere { $Verdadero$ }
asegura {
$res.numerador = n1.numerador \times n2.denominador + n2.numerador \times n1.denominador \land$
$res.denominador = n1.denominador \times n2.denominador$
}
}

proc resta(in $n_1, n_2 : NumeroRacional$) : $NumeroRacional$ {
requiere { $Verdadero$ }
asegura {
$res.numerador = n_1.numerador \times n_2.denominador - n_2.numerador \times n_1.denominador \land$
$res.denominador = n_1.denominador \times n_2.denominador$
}
}

proc multiplicacion(in $n_1, n_2 : NumeroRacional$) : $NumeroRacional$ {
requiere { $Verdadero$ }
asegura {
$res.numerador = n_1.numerador \times n_2.numerador \land$
$res.denominador = n_1.denominador \times n_2.denominador$
}
}

proc division(in $n_1, n_2 : NumeroRacional$) : $NumeroRacional$ {
requiere { $n_2.numerador \neq 0$ }
asegura {

$res.numerador = n_1.numerador \times n_2.denominador \land$
$res.denominador = n_1.denominador \times n_2.numerador$
}
}
}

**Ejercicio 2.** Especifique mediante TADs los siguientes elementos geométricos:

a) Punto2D, que representa un punto en el plano. Debe contener las siguientes operaciones:

- *nuevoPunto*: que crea un punto a partir de sus coordenadas $x$ e $y$.
- *mover*: que mueve el punto una determinada distancia sobre los ejes $x$ e $y$.
- *distancia*: que devuelve la distancia entre dos puntos.
- *distanciaAlOrigen*: que devuelve la distancia del punto $(0, 0)$.

b) Rectángulo2D, que representa un rectángulo en el plano. Debe contener las siguientes operaciones:

- *nuevoRectángulo*: que crea un rectángulo (decida usted cuáles deberían ser los parámetros).
- *mover*: que mueve el rectángulo una determinada distancia en los ejes $x$ e $y$.
- *escalar*: que escala el rectángulo en un determinado factor. Al escalar un rectángulo un punto del mismo debe quedar fijo. En este caso el punto fijo puede ser el centro del rectángulo o uno de sus vértices.
- *estáContenido*: que dados dos rectángulos, indique si uno está contenido en el otro.

c) Punto2D, con las mismas operaciones pero usando como observadores las coordenadas polares

***Solución***

a) `TAD` Punto2D {

`obs` x : $\mathbb{R}$

`obs` y : $\mathbb{R}$

`proc nuevoPunto`(in $x : \mathbb{R}$, in $y : \mathbb{R}$) : $Punto2D$ {
`requiere` { $Verdadero$ }
`asegura` { $res.x = x \land res.y = y$ }
}

`proc igualdad`(in $P_1 : Punto$, in $P_2 : Punto$) : bool {
`asegura` { $res = \text{true} \leftrightarrow P_1.x = P_2.x \land P_1.y = P_2.y$ }
}

`proc mover`(inout $p : Punto2D$, in $dx : \mathbb{R}$, in $dy : \mathbb{R}$) {
`requiere` { $p = P_0$ }
`asegura` { $p.x = P_0.x + dx \land p.y = P_0.y + dy$ }
}

`proc distancia`(in $p1 : Punto2D$, in $p2 : Punto2D$) : $Punto2D$ {
`requiere` { $Verdadero$ }
`asegura` { $res = distanciaEntrePuntos(p1.x, p1.y, p2.x, p2.y)$ }
}

`aux distanciaEntrePuntos`($x_1 : \mathbb{R}, y_1 : \mathbb{R}, x_2 : \mathbb{R}, y_2 : \mathbb{R}$) : $\mathbb{R} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

`proc distanciaAlOrigen`(in $p : Punto2D$) : $\mathbb{R}$ {

requiere { $Verdadero$ }

asegura { $res = distanciaEntrePuntos(p.x, p.y, 0, 0)$ }

}

}

b) TAD Rectangulo2D {

– *Posición de uno de los vértices*

obs posición : struct $\langle x \rangle \mathbb{R}, y : \mathbb{R}$

– *Posición relativa del vértice opuesto*

obs tamaño : struct $\langle dx \rangle \mathbb{R}, dy : \mathbb{R}$

proc nuevoRectangulo(in $pos$ : tupla $\langle \mathbb{R}, \mathbb{R} \rangle$, in $tam$ : tupla $\langle \mathbb{R}, \mathbb{R} \rangle$) : $Rectangulo2D$ {

requiere { $tam.x \geq 0 \land tam.y \geq 0$ }

asegura { $res.posición = pos \land res.tamaño = tam$ }

}

proc igualdad(in $R_1$ : $Rectangulo2D$, in $R_2$ : $Rectangulo2D$) : bool {

asegura { $res = \text{true} \leftrightarrow R_1.posición = R_2.posición \land R_1.tamaño = R_2.tamaño$ }

}

proc mover(inout $r$ : $Rectangulo2D$, in $dxy$ : tupla $\langle \mathbb{R}\, \mathbb{R} \rangle$) {

requiere { $r = R_0$ }

asegura { $r.posición.x = R_0.posición.x + dxy.x \land r.posición.y = R_0.posición.y + dxy.y$ }

}

proc escalar(inout $r$ : $Rectangulo2D$, in $esc : \mathbb{R}$) {

requiere { $r = R_0$ }

asegura { $r.tamaño.x = R_0.tamaño.x * esc \land r.tamaño.y = R_0.tamaño.y * esc$ }

}

proc estáContenido(in $r1$ : $Rectangulo2D$, in $r2$ : $Rectangulo2D$) : bool {

requiere { $Verdadero$ }

asegura {

$res \leftrightarrow ($
$r_1.posición.x \geq r_2.posición.x \land r_1.posición.y \geq r_2.posición.y \land$
$r_1.tamaño.x < r_2.tamaño.x \land r_1.tamaño.y < r_2.tamaño.y$
$) \lor ($
$r_1.posición.x < r_2.posición.x \land r_1.posición.y < r_2.posición.y \land$
$r_1.tamaño.x > r_2.tamaño.x \land r_1.tamaño.y > r_2.tamaño.y$
$)$

}

}

}

c) TAD Punto2D {

obs r : $\mathbb{R}$

obs $\theta$ : $\mathbb{R}$

proc nuevoPunto(in $x, y : \mathbb{R}$) : $Punto2D$ {

requiere { $Verdadero$ }

asegura { $res.r = radio(x, y) \land esAngulo(res.\theta, x, y)$ }

}

proc igualdad(in $P_1$ : $Punto2D$, in $P_2$ : $Punto2D$) : bool {

asegura { $res = \text{true} \leftrightarrow (P_1.r = P_2.r \wedge (\exists k : \mathbb{Z})\ (P_1.\theta = P_2.\theta + 2k\pi))$ }

}

proc mover(inout $p : Punto2D, dx, dy : \mathbb{R}$) {

requiere { $p = P_0$ }

asegura {

$p.r = radio(coordX(P_0.r, P_0.\theta) + dx, coordY(P_0.r, P_0.\theta) + dy) \wedge$
$esAngulo(p.\theta, coordX(P_0.r, P_0.\theta) + dx, coordY(P_0.r, P_0.\theta) + dy)$

}

}

proc distancia(in $p1, p2 : Punto2D$) : $\mathbb{R}$ {

requiere { $Verdadero$ }

asegura { $res = distanciaEntre(p1, p2)$ }

}

proc distanciaAlOrigen(in $p : Punto2D$) : $\mathbb{R}$ {

requiere { $Verdadero$ }

asegura { $res = p.r$ }

}

aux radio($x, y : \mathbb{R}$) : $\mathbb{R} = \sqrt{x^2 + y^2}$

aux angulo($x, y : \mathbb{R}$) : $\mathbb{R} = atan(y/x)$

pred esAngulo($\theta, x, y : \mathbb{R}$) {

$(x \neq 0 \rightarrow_L \theta = angulo(x, y)) \wedge$
$(x = 0 \rightarrow ((y >= 0 \rightarrow \theta = \pi/2) \wedge (y < 0 \rightarrow \theta = 3 * \pi/2)))$

}

aux coordX($r, \theta : \mathbb{R}$) : $\mathbb{R} = r * cos(\theta)$

aux coordY($r, \theta : \mathbb{R}$) : $\mathbb{R} = r * sin(\theta)$

aux distanciaEntre($p1, p2 : Punto2D$) : $\mathbb{R} =$
$\sqrt{(coordX(p2.r, p2.\theta) - coordX(p1.r, p1.\theta))^2 + (coordY(p2.r, p2.\theta) - coordY(p1.r, p1.\theta))^2}$

}

## 3.2. TADs para colecciones

**Ejercicio 3.**

a) Especifique el TAD Cola$\langle T \rangle$ con las siguientes operaciones:

- *nuevaCola* que crea una cola vacía
- *estáVacía* que devuelve true si la cola no contiene elementos
- *encolar* que agrega un elemento al final de la cola
- *desencolar* que elimina el primer elemento de la cola y lo devuelve

b) Especifique el TAD Pila$\langle T \rangle$ con las siguientes operaciones

- *nuevaPila* que crea una pila vacía
- *estáVacia* que devuelve true si la pila no contiene elementos
- *apilar* que agrega un elemento al tope de la pila
- *desapilar* que elimina el elemento del tope de la pila y lo devuelve

c) Especifique el TAD DobleCola$\langle T \rangle$, en el que los elementos pueden insertarse al principio o al final y se eliminan por el medio. Debe contener las operaciones

- *nuevaDobleCola* que crea una cola vacía
- *estáVacía* que devuelve true si la cola no contiene elementos
- *encolarAtrás* que agrega un elemento al final de la cola
- *encolarAdelante* que agrega un elemento al principio de la cola
- *desencolar* que elimina el primer elemento de la cola y lo devuelve

Ejemplo:

```
c := new NuevaDobleCola<T>()

encolarAdelante(c, 1)    // c = <1>
encolarAdelante(c, 2)    // c = <2,1>
encolarAtrás(c, 3)    // c = <2,1,3>

desencolar(c)      // devuelve 1, c = <2,3>
desencolar(c)      // devuelve 2, c = <3>
desencolar(c)      // devuelve 3, c = <>
```

***Solución***

a) TAD Cola$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc nuevaCola() : Cola$\langle T \rangle$ {
  requiere { $Verdadero$ }
  asegura { $|res.elems| = 0$ }
}

proc igualdad(in $c_1, c_2$ : Cola$\langle T \rangle$) : bool {
  asegura { $res \leftrightarrow c_1.elems = c_2.elems$ }
}

proc estaVacía(in $c$ : Cola$\langle T \rangle$) : bool {
  requiere { $Verdadero$ }
  asegura { $res \leftrightarrow |res.elems| = 0$ }
}

proc encolar(inout $c$ : Cola$\langle T \rangle$, in $e : T$) {
  requiere { $c = C_0$ }
  asegura { $c.elems = C_0.elems +\!\!+ \langle e \rangle$ }
}

proc desencolar(inout $c$ : Cola$\langle T \rangle$) : $T$ {
  requiere { $c = C_0 \wedge |c.elems| > 0$ }
  asegura { $res = C_0.elems[0] \wedge sacoElPrimero(C_0, c)$ }
}

pred sacoElPrimero($C_0, c$ : Cola$\langle T \rangle$) {

$$|c.elems| = |C_0.elems| - 1 \wedge_L$$
$$(\forall i : \mathbb{Z})\ (0 \leq i < |c.elems| \rightarrow_L c.elems[i] = C_0.elems[i+1])$$

}

}

b) TAD Pila$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc nuevaPila() : Pila$\langle T \rangle$ {

requiere { $Verdadero$ }

asegura { $|res.elems| = 0$ }

}

proc igualdad(in $p_1, p_2$ : Pila$\langle T \rangle$) : bool {

asegura { $res \leftrightarrow p_1.elems = p_2.elems$ }

}

proc estaVacía(in $c$ : Pila$\langle T \rangle$) : bool {

requiere { $Verdadero$ }

asegura { $res \leftrightarrow |res.elems| = 0$ }

}

proc apilar(inout $c$ : Pila$\langle T \rangle$, in $e : T$) {

requiere { $p = P_0$ }

asegura { $agregadoAdelante(P_0, p, e)$ }

}

proc desapilar(inout $c$ : Pila$\langle T \rangle$) : $T$ {

requiere { $p = P_0 \land |p.elems| > 0$ }

asegura { $res = P_0.elems[0] \land sacoElPrimero(P_0, p)$ }

}

pred agregadoAdelante($P_0, p$ : Pila$\langle T \rangle, e : T$) {

$p.elems[0] = e \land |p.elems| = |P_0.elems| + 1 \land_L$
$(\forall i : \mathbb{Z})\ (1 \leq i < |p.elems| \rightarrow_L p.elems[i] = P_0.elems[i-1])$

}

pred sacoElPrimero($P_0, p$ : Pila$\langle T \rangle$) {

$|p.elems| = |P_0.elems| - 1 \land_L$
$(\forall i : \mathbb{Z})\ (0 \leq i < |p.elems| \rightarrow_L p.elems[i] = P_0.elems[i+1])$

}

}

c) TAD DobleCola$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc nuevaDobleCola() : DobleCola$\langle T \rangle$ {

requiere { $Verdadero$ }

asegura { $|res.elementos| = 0$ }

}

proc igualdad(in $c_1, c_2$ : DobleCola$\langle T \rangle$) : bool {

asegura { $res \leftrightarrow c_1.elems = c_2.elems$ }

}

proc estaVacía(in $c$ : DobleCola$\langle T \rangle$) : bool {

```
        requiere { c = C_0 }
        asegura { res ↔ |res.elementos| = 0 }
    }

    proc encolarAtrás(inout c : DobleCola⟨T⟩, in elem : T) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems ++ ⟨e⟩ }
    }

    proc encolarAdelante(inout c : DobleCola⟨T⟩, in elem : T) {
        requiere { p = P_0 }
        asegura { agregadoAdelante(C_0, c, e) }
    }

    – Tomo ÷ como división entera
    proc desencolar(inout c : DobleCola⟨T⟩) : T {
        requiere { c = C_0 ∧ |c.elementos| > 0 }
        asegura { res = C_0.elementos[|C_0.elementos| ÷ 2] ∧ sacoElDelMedio(C_0, c) }
    }

    pred agregadoAdelante(C_0, c : DobleCola⟨T⟩, e : T)  {
         c.elems[0] = e ∧ |c.elems| = |C_0.elems| + 1 ∧_L
              (∀i : ℤ) (0 ≤ i < |c.elems| →_L c.elems[i] = C_0.elems[i − 1])
    }

    pred sacoElDelmedio(C_0, c : DobleCola⟨T⟩)  {
         |c.elems| = |C_0.elems| − 1 ∧_L (
              (∀i : ℤ) (0 ≤ i < |c.elems| ÷ 2 →_L c.elems[i] = C_0.elems[i]) ∧
              (∀i : ℤ) (|c.elems| ÷ 2 ≤ i < |c.elems| →_L c.elems[i] = C_0.elems[i + 1])
         )
    }
}
```

**Ejercicio 4.**

a) Especifique el TAD Diccionario$\langle K, V\rangle$ con las siguientes operaciones:

- *nuevoDiccionario*: que crea un diccionario vacío
- *definir*: que agrega un par clave-valor al diccionario
- *obtener*: que devuelve el valor asociado a una clave
- *está*: que devuelve true si la clave está en el diccionario
- *borrar*: que elimina una clave del diccionario

b) Especifique el TAD DiccionarioConHistoria$\langle K, V\rangle$. El mismo permite consultar, para cada clave, todos los valores que se asociaron con la misma a lo largo del tiempo (en orden). Se debe poder hacer dicha consulta aún si la clave fue borrada.

***Solución***

```
a) TAD Diccionario⟨K, V⟩ {
    obs elems : dicc⟨K, V⟩
    proc nuevoDiccionario() : Diccionario⟨K, V⟩ {
```

requiere { $Verdadero$ }
asegura { $res.elems = \langle\rangle$ }
}

proc igualdad(in $d_1, d_2$ : Diccionario$\langle K, V\rangle$) : bool {
asegura { $res \leftrightarrow d_1.elems = d_2.elems$ }
}

proc definir(inout $d$ : Diccionario$\langle K, V\rangle$, in $k : K$, in $v : V$) {
requiere { $d = D_0$ }
asegura { $d.elems = setKey(D_0.elems, k, v)$ }
}

proc obtener(in $d$ : Diccionario$\langle K, V\rangle$, in $k : K$) : $V$ {
requiere { $k \in d.elems$ }
asegura { $res = d.elems[k]$ }
}

proc esta(in $d$ : Diccionario$\langle K, V\rangle$, in $k : K$) : bool {
requiere { $Verdadero$ }
asegura { $res = true \leftrightarrow k \in d.elems$ }
}

proc borrar(inout $d$ : Diccionario$\langle K, V\rangle$, in $k : K$) {
requiere { $d = D_0 \land k \in d.elems$ }
asegura { $d.elems = delKey(D_0.d, k)$ }
}

}

b) TAD DiccionarioConHistoria$\langle K, V\rangle$ {

obs elems : dicc$\langle K, V\rangle$
obs historial : dicc$\langle K,$ seq$\langle V\rangle\rangle$
proc nuevoDiccionarioConHistoria() : DiccionarioConHistoria$\langle K, V\rangle$ {
requiere { $Verdadero$ }
asegura { $res.elems = \langle\rangle \land res.historial = \langle\rangle$ }
}

proc igualdad(in $d_1, d_2$ : DiccionarioConHistoria$\langle K, V\rangle$) : bool {
requiere { $Verdadero$ }
asegura { $d_1.elems = d_2.elems \land d_1.historial = d_2.historial$ }
}

proc definir(inout $d$ : DiccionarioConHistoria$\langle K, V\rangle$, in $k : K$, in $v : V$) {
requiere { $d = D_0$ }
asegura { $d.elems = setKey(D_0, k, v) \land definidoEnHistorial(D_0, d, k, v)$ }
}

pred definidoEnHistorial($D_0, d$ : DiccionarioConHistoria$\langle K, V\rangle, k : K, v : V$) {
$(k \in$ claves$(D_0.historial) \land d.historial = setKey(D_0, k, D_0.historial[k] +\!\!+ \langle v\rangle))$
$\lor$
$(k \notin$ claves$(D_0.historial) \land d.historial = setKey(D_0, k, \langle v\rangle))$
}

proc está(in $d$ : DiccionarioConHistoria$\langle K, V\rangle$, in $k : K$) : bool {

```
        requiere { Verdadero }
        asegura { res ↔ k ∈ claves(d.elems) }
    }

    proc obtener(in d : DiccionarioConHistoria⟨K, V⟩, in k : K) : V {
        requiere { k ∈ claves(d.elems) }
        asegura { res = d.elems[k] }
    }

    proc borrar(inout d : DiccionarioConHistoria⟨K, V⟩, in k : K) {
        requiere { d = D_0 ∧ k ∈ claves(d.elems) }
        asegura { d.elems = delKey(D_0, k) ∧ d.historial = D_0.historial }
    }

    proc obtenerHistorial(in d : DiccionarioConHistoria⟨K, V⟩, in k : K) : seq⟨V⟩ {
        requiere { k ∈ claves(d.historial) }
        asegura { res = d.historial[k] }
    }

}
```

**Ejercicio 5.** Especifique los TADs indicados a continuación utilizando los observadores propuestos:[1]

a) Conjunto$\langle T \rangle$ con las operaciones

- *nuevoConjunto*: que crea un conjunto vacío
- *agregar*: que agrega un elemento al conjunto
- *pertenece*: que devuelve true si un elemento pertenece al conjunto
- *eliminar*: que elimina un elemento del conjunto

observado con

i) conj$\langle T \rangle$
ii) seq$\langle T \rangle$

b) Diccionario$\langle K, V \rangle$ observado con

i) conj$\langle$tupla $\langle K, V \rangle\rangle$
ii) conj$\langle$struct $\langle clave : K, valor : V \rangle\rangle$
iii) seq$\langle$struct $\langle clave : K, valor : V \rangle\rangle$

c) Pila$\langle T \rangle$ observado con

i) seq$\langle T \rangle$
ii) dicc$\langle T, \mathbb{Z} \rangle$
iii) conj$\langle$struct $\langle valor : T, posicion : \mathbb{Z} \rangle\rangle$

Preste especial atención al proc de igualdad para estos TADs.

***Solución***

a) i) `TAD` Conjunto$\langle T \rangle$ {

[1] Asumimos que algunas de estas especificaciones son las que ya se usaron en los ejercicios anteriores de esta sección

obs elems : conj$\langle T \rangle$

proc conjuntoVacio() : Conjunto$\langle T \rangle$ {
  requiere { $Verdadero$ }
  asegura { $|res.elems| = 0$ }
}

proc igualdad(in $c_1, c_2$ : Conjunto$\langle T \rangle$) : bool {
  requiere { $Verdadero$ }
  asegura { $res = true \leftrightarrow c_1.elems = c_2.elems$ }
}

proc agregar(inout $c$ : Conjunto$\langle T \rangle$, in $e : T$) {
  requiere { $c = C_0$ }
  asegura { $c.elems = C_0.elems \cup \langle e \rangle$ }
}

proc pertenece(in $c$ : Conjunto$\langle T \rangle$, in $e : T$) : bool {
  requiere { $Verdadero$ }
  asegura { $res = true \leftrightarrow e \in c.elems$ }
}

proc eliminar(inout $c$ : Conjunto$\langle T \rangle$, in $e : T$) {
  requiere { $c = C_0$ }
  asegura { $c = C_0 - \langle e \rangle$ – *Diferencia de conjuntos* }
}

}

ii) TAD Conjunto$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc conjuntoVacio() : Conjunto$\langle T \rangle$ {
  requiere { $Verdadero$ }
  asegura { $|res.elems| = 0$ }
}

proc igualdad(in $c_1, c_2$ : Conjunto$\langle T \rangle$) : bool {
  requiere { $Verdadero$ }
  asegura { $res = true \leftrightarrow (\forall e : T)\ (e \in c_1.elems \leftrightarrow e \in c_2.elems)$ }
}

proc agregar(inout $c$ : Conjunto$\langle T \rangle$, in $e : T$) {
  requiere { $c = C_0$ }
  asegura {
    $e \in c.elems \land$
    $siguenLosMismos(C_0, c) \land$
    $soloSeAgregó(C_0, c, e)$
  }
}

pred siguenLosMismos($C_0, c$ : Conjunto$\langle T \rangle$) {
  $(\forall t : T)\ (t \in C_0.elems \rightarrow t \in c.elems)$
}

pred soloSeAgregó($C_0, c$ : Conjunto$\langle T \rangle, e : T$) {
  $(\forall t : T)\ (t \in c.elems \land t \neq e \rightarrow t \in C_0.elems)$
}

proc pertenece(in $c$ : Conjunto$\langle T \rangle$, in $e : T$) : bool {
  requiere { $Verdadero$ }
  asegura { $res = true \leftrightarrow e \in c.elems$ }

```
    }
    proc eliminar(inout c : Conjunto⟨T⟩, in e : T) {
        requiere { c = C_0 }
        asegura {
            ¬(e ∈ c.elems) ∧
            noSaquéOtros(C_0, c, e)
        }
    }
    pred noSaquéOtros(C_0, c : Conjunto⟨T⟩, e : T) {
        (∀t : T) (t ≠ e → (t ∈ C_0.elems ↔ t ∈ c.elems))
    }
}
```

b) i)

```
TAD Diccionario⟨K, V⟩ {
    – t_0 clave, t_1 valor
    obs elems : conj⟨tupla ⟨K, V⟩⟩
    proc nuevoDiccionario() : Diccionario⟨K, V⟩ {
        requiere { Verdadero }
        asegura { res.elems = ⟨⟩ }
    }
    proc igualdad(in d_1, d_2 : Diccionario⟨K, V⟩) : bool {
        asegura {
            res = true ↔
            (∀t : tupla ⟨K, V⟩) (t ∈ d_1.elems ↔ t ∈ d_2.elems)
        }
    }
    proc definir(inout d : Diccionario⟨K, V⟩, in k : K, in v : V) {
        requiere { d = D_0 }
        asegura {
            ⟨k, v⟩ ∈ d.elems ∧
            noCambianOtras(D_0, d, k) ∧_L
            cantApariciones(d, k) = 1
        }
    }
    pred noCambiaronOtras(D_0, d : Diccionario⟨K, V⟩, k : K) {
        (∀t : tupla ⟨K, V⟩) (t_0 ≠ k → (t ∈ D_0.elems ↔ t ∈ d.elems))
    }
    aux cantApariciones(d : Diccionario⟨K, V⟩, k : K) : ℤ =
        Σ_{t:tupla ⟨K,V⟩} IfThenElse(t_0 = k, 1, 0)
    proc obtener(in d : Diccionario⟨K, V⟩, ink : K) : V {
        requiere { (∃t : tupla ⟨K, V⟩) (t ∈ d.elems ∧ t_0 = k) }
        asegura { (∃t : tupla ⟨K, V⟩) (t ∈ d.elems ∧ t_0 = k ∧ res = t_1) }
    }
    proc está(in d : Diccionario⟨K, V⟩, in k : K) : bool {
        requiere { Verdadero }
        asegura { res ↔ (∃t : tupla ⟨K, V⟩) (t ∈ d.elems ∧ t_0 = k) }
    }
    proc borrar(inout d : Diccionario⟨K, V⟩, in k : K) {
        requiere { d = D_0 ∧ (∃t : tupla ⟨K, V⟩) (t ∈ d.elems ∧ t_0 = k∧) }
        asegura { ⟨k, v⟩ ∉ d.elems ∧ noCambianOtras(D_0, d, k) }
```

}

}

ii) Es igual pero más legible :)

iii) Dado que nuestra especificación asegura que no haya más de una tupla (o struct) representando la misma clave, no nos afecta el hecho de que una secuencia tenga repetidos. Además como toda nuestra especificación es en términos de $\in$ y $\exists$, no nos afectaría el orden Por eso, usando la notación $\in$ para secuencias podemos "reutilizar" toda la especificación. Podríamos escribirla usando índices, en cuyo caso hay cambios de notación, pero no hay cambios de lógica.

c) i) Ya lo hicimos (ejercicio 3)

ii) Nótese que este observador no permite apilar más de una vez el mismo $T$

```
TAD Pila⟨T⟩ {
    obs elemsEnPosicion : dicc⟨T, ℤ⟩
    proc nuevaPila() : Pila⟨T⟩ {
        requiere { Verdadero }
        asegura { res.elemsEnPosicion = ⟨⟩ }
    }
    proc igualdad(in p1, p2 : Pila⟨T⟩) : bool {
        asegura { res ↔ p1.elemsEnPosicion = p2.elemsEnPosicion }
    }
    proc estaVacía(in p : Pila⟨T⟩) : bool {
        requiere { Verdadero }
        asegura { res ↔ res.elemsEnPosicion = ⟨⟩ }
    }
    proc apilar(inout p : Pila⟨T⟩, in e : T) {
        requiere { p = P0 }
        asegura { apilado(P0, p, e) }
    }
    proc desapilar(inout p : Pila⟨T⟩) : T {
        requiere { p = P0 ∧ |p.elemsEnPosicion| > 0 }
        asegura { esTope(P0, res) ∧ desapilado(P0, p) }
    }
    pred apilado(P0, p : Pila⟨T⟩, e : T) {
        (∃i : ℤ) (esProximaPosicion(P0, k) ∧ p.elemsEnPosicion = setKey(P0.elemsEnPosicion, e, i))
    }
    pred esProximaPosicion(p : Pila⟨T⟩, i : ℤ) {
        (p.elemsEnPosicion = ⟨⟩ ∧ i = 0) ∨L ((∃j : ℤ) (esPosicionDeTope(p, j) ∧ i = j + 1))
    }
    pred esPosicionDeTope(p : Pila⟨T⟩, i : ℤ) {
        (∀t : T) (t ∈ p.elemsEnPosicion →L p.elemsInPosicion[t] ≤ i)
    }
    pred esTope(p : Pila⟨T⟩, res : T) {
        res ∈ claves(p.elemsEnPosicion) ∧L
            (∃i : ℤ) (esPosicionDeTope(p, i) ∧ p.elemsEnPosicion[res] = i)
    }
    pred desapilado(P0, p : Pila⟨T⟩) {
        (∃t : T) (
            t ∈ claves(P0.elemsEnPosicion) ∧
            esPosicionDeTope(P0, P0.elemsEnPosicion[t]) ∧
            p.elemsEnPosicion = delKey(P0.elemsEnPosicion, t)
        )
```

}

}

iii) `TAD` Pila$\langle T \rangle$ {

`obs` elemsEnPosicion : conj$\langle$struct $\langle valor \rangle T, posicion : \mathbb{Z} \rangle$

`proc nuevaPila`() : Pila$\langle T \rangle$ {
`requiere` { $Verdadero$ }
`asegura` { $res.elemsEnPosicion = \langle\rangle$ }
}

`proc igualdad`(in $p_1, p_2$ : Pila$\langle T \rangle$) : bool {
`asegura` { $res \leftrightarrow p_1.elemsEnPosicion = p_2.elemsEnPosicion$ }
}

`proc estaVacía`(in $p$ : Pila$\langle T \rangle$) : bool {
`requiere` { $Verdadero$ }
`asegura` { $res \leftrightarrow res.elemsEnPosicion = \langle\rangle$ }
}

`proc apilar`(inout $p$ : Pila$\langle T \rangle$, in $e : T$) {
`requiere` { $p = P_0$ }
`asegura` { $apilado(P_0, p, e)$ }
}

`proc desapilar`(inout $p$ : Pila$\langle T \rangle$) : $T$ {
`requiere` { $p = P_0 \wedge |p.elemsEnPosicion| > 0$ }
`asegura` { $esTope(P_0, res) \wedge desapilado(P_0, p)$ }
}

`pred apilado`($P_0, p$ : Pila$\langle T \rangle, e : T$) {
$(\exists i : \mathbb{Z})$ (
$esProximaPosicion(P_0, k) \wedge p.elemsEnPosicion = P_0 \cup \langle\langle valor : e, posicion : i \rangle\rangle$
)
}

`pred esProximaPosicion`($p$ : Pila$\langle T \rangle, i : \mathbb{Z}$) {
$(p.elemsEnPosicion = \langle\rangle \wedge i = 0) \vee_L ((\exists j : \mathbb{Z})\ (esPosicionDeTope(p, j) \wedge i = j + 1))$
}

`pred esPosicionDeTope`($p$ : Pila$\langle T \rangle, i : \mathbb{Z}$) {
$(\forall s$ : struct $\langle valor : T, posicion : \mathbb{Z} \rangle)\ (s \in p.elemsEnPosicion \rightarrow_L s.posicion \leq i)$
}

`pred esTope`($p$ : Pila$\langle T \rangle, res : T$) {
$(\exists s$ : struct $\langle valor : T, posicion : \mathbb{Z} \rangle)$ (
$s \in p.elemsEnPosicion \wedge_L res = s.valor \wedge esPosicionDeTope(p, s.posicion)$
)
}

`pred desapilado`($P_0, p$ : Pila$\langle T \rangle$) {
$(\exists s$ : struct $\langle valor : T, posicion : \mathbb{Z} \rangle)$ (
$s \in P_0.elemsEnPosicion \wedge$
$esPosicionDeTope(P_0, s.posicion) \wedge$
$p.elemsEnPosicion = P_0.elemsEnPosicion - \langle t \rangle$
)
}

}

**Ejercicio 6.** Especificar TADs para las siguientes estructuras:

a) Multiconjunto$\langle T \rangle$
También conocido como `multiset` o `bag`. Es igual a un conjunto pero con duplicados: cada elemento puede agregarse múltiples veces. Tiene las mismas operaciones que el TAD `Conjunto`, más una operación que indica la multiplicidad de un elemento (la cantidad de veces que ese elemento se encuentra en la estructura). Nótese que si un elemento es eliminado del multiconjunto, se reduce en 1 la multiplicidad.

Ejemplo:

```
c := new MultiConjunto<int>()

agregar(c, 1)
agregar(c, 1)
pertenece(c, 1)   // devuelve true
multiplicidad(c, 1)  // devuelve 2

sacar(c, 1)
pertenece(c, 1)   // devuelve true
multiplicidad(c, 1)  // devuelve 1
```

b) Multidict$\langle K, V \rangle$
Misma idea pero para diccionarios: Cada clave puede estar asociada con múltiples valores. Los valores se definen de a uno (indicando una clave y un valor), pero la operación `obtener` debe devolver todos los valores asociados a una determinada clave.

Nota: En este ejercicio deberá tomar algunas decisiones. ¿Se pueden asociar múltiples veces un mismo valor con una clave? ¿Qué pasa en ese caso? Qué parámetros tiene y cómo se comporta la operación `borrar`? Imagine un caso de uso para esta estructura y utilice su sentido común para tomar estas decisiones.

***Solución***

a) `TAD` MultiConjunto$\langle T \rangle$ {

`obs` elemsConMult : dicc$\langle T, \mathbb{Z} \rangle$

`proc conjuntoVacio`() : MultiConjunto$\langle T \rangle$ {

`requiere` { $Verdadero$ }

`asegura` { $|res.elemsConMult| = 0$ }

}

`proc igualdad`(in $c_1, c_2$ : MultiConjunto$\langle T \rangle$) : bool {

`requiere` { $Verdadero$ }

`asegura` { $res = true \leftrightarrow c_1.elemsConMult = c_2.elemsConMult$ }

}

`proc agregar`(inout $c$ : MultiConjunto$\langle T \rangle$, in $e : T$) {

`requiere` { $c = C_0$ }

`asegura` {

$(e \in \text{claves}(C_0.elemsConMult) \wedge$
$c.elemsConMult = setKey(C_0.elemsConMult, e, C_0.elemsConMult[e] + 1))$
$\vee$
$(e \notin \text{claves}(C_0.elemsConMult) \wedge c.elemsConMult = setKey(C_0.elemsConMult, e, 1))$

}

}

`proc pertenece`(in $c$ : MultiConjunto$\langle T \rangle$, in $e : T$) : bool {

requiere { $Verdadero$ }

asegura { $res = true \leftrightarrow e \in c.elemsConMult$ }

}

proc eliminar(inout $c$ : MultiConjunto$\langle T \rangle$, in $e : T$) {

requiere { $c = C_0$ }

asegura {

$e \notin C_0.elemsConMult \land c.elemsConMult = C_0.elemsConMult$

$\lor$

$e \in C_0.elemsConMult \land_L ($

$C_0.elemsConMult[e] > 1 \land$

$c.elemsConMult = setKey(C_0.elemsConMult, e, C_0.elemsConMult[e] - 1)$

$\lor$

$C_0.elemsConMult[e] = 1 \land$

$c.elemsConMult = delKey(C_0.elemsConMult, e)$

$)$

}

}

proc multiplicidad(in $c$ : MultiConjunto$\langle T \rangle$, in $e : T$) : $\mathbb{Z}$ {

requiere { $Verdadero$ }

asegura {

$e \in$ claves$(c.elemsConMult) \land res = c.elemsConMult[e]$

$\lor$

$e \notin$ claves$(c.elemsConMult) \land res = 0$

}

}

}

b) TAD MultiDic$\langle K, V \rangle$ {

obs elemsConMult : dicc$\langle K,$ conj$\langle V \rangle\rangle$

proc nuevoDiccionario() : MultiDic$\langle K, V \rangle$ {

requiere { $Verdadero$ }

asegura { $res.elemsConMult = \langle\rangle$ }

}

proc igualdad(in $d_1, d_2$ : MultiDic$\langle K, V \rangle$) : bool {

asegura {

$res = true \leftrightarrow d_1.elemsConMult = d_2.elemsConMult$

}

}

*– Se aceptan varias veces el mismo valor como parámetro, pero no se asocia repetido*

proc definir(inout $d$ : MultiDic$\langle K, V \rangle$, in $k : K$, in $v : V$) {

requiere { $d = D_0$ }

asegura {

$k \in D_0.elemsConMult \land$

$d.elemsConMult = setKey(D_0.elemsConMult, k, D_0.elemsConMult[k] \cup \langle v \rangle)$

$\lor$

$k \notin$ claves$(D_0.elemsConMult) \land$

$d.elemsConMult = setKey(D_0.elemsConMult, k, \langle v \rangle)$

}

```
    }

    proc obtener(in d : MultiDic⟨K, V⟩, ink : K) : conj⟨V⟩ {
        requiere { k ∈ d.elemsConMult }
        asegura { res = d.elemsConMult[k] }
    }

    proc está(in d : MultiDic⟨K, V⟩, in k : K) : bool {
        requiere { Verdadero }
        asegura { res ↔ k ∈ d.elemsConMult }
    }

    – Borra un valor de una clave, si es el único borra la clave
    proc borrar(inout d : MultiDic⟨K, V⟩, in k : K, in v : V) {
        requiere { d = D0 }
        asegura {
            (k ∉ D0.elemsConMult ∨ v ∉ D0.elemsConMult[k]) ∧
                d.elemsConMult = D0.elemsConMult
            ∨
            (k ∈ D0.elemsConMult ∧ v ∈ D0.elemsConMult[k]) ∧L (
                |D0.elemsConMult[k]| > 1 ∧
                    d.elemsConMult = setKey(D0.elemsConMult, k, D0.elemsConMult[k] − ⟨v⟩)
                ∨
                D0.elemsConMult[k] = 1 ∧
                    d.elemsConMult = delKey(D0.elemsConMult, k)
            )
        }
    }

}
```

## 3.3. TADs surtidos

**Ejercicio 7.** Especifique el TAD `Contadores` que, dada una lista de eventos, permite contar la cantidad de veces que se produjo cada uno de ellos. La lista de eventos es fija. El TAD debe tener una operación para incrementar el contador asociado a un evento y una operación para conocer el valor actual del contador para un evento.

- Modifique el TAD para que sea posible preguntar el valor del contador *en un determinado momento del pasado*. Si necesita conocer la fecha y hora actual, puede pasarla como parámetro a los procedimientos. Asuma que las fechas son números enteros (por ejemplo, la cantidad de segundos desde el 1 de enero de 1970).

***Solución***

```
Fecha ES ℤ
Evento ES string
– Hago directamente la versión modificada. La no modificada sería casi igual pero el valor de las claves
–       sería un entero. Incrementar le suma 1 a ese entero y obtener lo devuelve.
TAD Contadores {
    – El valor del contador es la longitud - 1 (el último índice)
    – Esta decisión implica que no se puede contar más de una vez en el mismo momento
    obs eventos : dicc⟨string, seq⟨Fecha⟩⟩
    proc crearContadores(in eventos : seq⟨Evento⟩, in fechaActual : Fecha) : Contadores {
        requiere { Verdadero }
        asegura { (∀e : Evento) (e ∈ eventos ↔ (e ∈ res.eventos ∧L |res.eventos[e]| = 0)) }
```

```
    }
    proc igualdad(in C_1 : Contadores, in C_2 : Contadores) : bool {
        asegura { res = true ↔ C_1.eventos = C_2.eventos }
    }
    proc incrementar(inout c : Contadores, in e : Evento, in fechaActual : Fecha) {
        requiere {
            c = C_0 ∧
            e ∈ c.eventos ∧_L (|c.eventos[e]| = 0 ∨ fechaActual > ultimo(c.eventos[e]))
        }
        asegura { c.eventos = setKey(C_0.eventos, e, C_0.eventos[e] ++ ⟨fechaAct⟩) }
    }
    proc obtenerEnFecha(in c : Contadores, in e : Evento, in fechaBuscada : Fecha, in fechaActual : Fecha) : ℤ {
        requiere { fechaBuscada < fechaActual ∧ e ∈ c.eventos ∧_L |c.eventos[e]| > 0 }
        asegura { esLaCuentaHastaLaFecha(c, e, res, fechaBuscada) }
    }
    pred esLaCuentaHastaLaFecha(c : Contadores, e : Evento, res : ℤ, fechaBuscada : Fecha)  {
         ((|c.eventos[e]| = 0 ∨_L fechaBuscada < ultimo(c.eventos[e])) ∧ res = 0)
         ∨_L
         (∃i : ℤ) (
              1 ≤ i < |c.eventos[e]| ∧_L
                   (c.eventos[e][i − 1] ≥ fechaBuscada < c.eventos[e][i] ∧ res = i)
         )
    }
    aux ultimo(s : seq⟨T⟩) : T = s[|s| − 1]
}
```

**Ejercicio 8.** Un `caché` es una capa de almacenamiento de datos de alta velocidad que almacena un subconjunto de datos, normalmente transitorios, de modo que las solicitudes futuras de dichos datos se atienden con mayor rapidez que si se debe acceder a los datos desde la ubicación de almacenamiento principal. El almacenamiento en caché permite reutilizar de forma eficaz los datos recuperados o procesados anteriormente.

Esta estructura tiene una interface de diccionario: guarda valores asociados a claves, con la diferencia de que los datos almacenados en un cache pueden *desaparecer* en cualquier momento, en función de diferentes criterios.

Especificar caches genéricos (con claves de tipo K y valores de tipo V) que respeten las operaciones indicadas y las siguientes políticas de borrado automático. Si necesita conocer la fecha y hora actual, puede pasarla como parámetro a los procedimientos o bien puede asumir que existe una función externa $horaActual() : \mathbb{Z}$ que retorna la fecha y hora actual. Asuma que las fechas son números enteros (por ejemplo, la cantidad de segundos desde el 1 de enero de 1970).

```
TAD Cache⟨K, V⟩ {
    proc esta(in c : Cache⟨K, V⟩, in k : K) : bool
    proc obtener(in c : Cache⟨K, V⟩, in k : K) : V
    proc definir(inout c : Cache⟨K, V⟩, in k : K, in v : V)
}
```

a) `FIFO` o `first-in-first-out`:
   El cache tiene una capacidad máxima (máximo número de claves). Si se alcanza esa capacidad máxima se borra automáticamente la clave que fue definida por primera vez hace más tiempo.

b) `LRU` o `last-recently-used`:
   El cache tiene una capacidad máxima (máximo número de claves). Si se alcanza esa capacidad máxima se borra automáticamente la clave que fue accedida por última vez hace más tiempo. Si no fue accedida nunca, se considera el momento en que fue agregada.

c) `TTL` o `time-to-live`:
   El cache tiene asociado un máximo tiempo de vida para sus elementos. Los elementos se borran automáticamente cuando se alcanza el tiempo de vida (contando desde que fueron agregados por última vez).

***Solución***

a) TAD CacheFIFO$\langle K, V \rangle$ {

obs capacidad : $\mathbb{Z}$

obs data : dicc$\langle K, V \rangle$

– *Claves en el orden en que los fueron agregadas*

obs claves : seq$\langle K \rangle$

proc nuevoCache(in $cap : \mathbb{Z}$) : CacheFIFO$\langle K, V \rangle$ {

requiere { $cap > 0$ }

asegura { $res.capacidad = cap \land res.data = \langle\rangle \land res.claves = \langle\rangle$ }

}

proc igualdad($C_1, C_2$ : CacheFIFO$\langle K, V \rangle$) : bool {

asegura {

$C_1.capacidad = C_2.capacidad$
$C_1.data = C_2.data$
$C_1.claves = C_2.claves$

}

}

proc esta(in $c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$) : bool {

asegura { $res = true \leftrightarrow k \in c.data$ }

}

proc obtener(in $c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$) : $V$ {

requiere { $k \in c.data$ }

asegura { $res = c.data[k]$ }

}

proc definir(inout $c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$, in $v : V$) {

requiere { $c = C_0$ }

asegura {

$c.capacidad = C_0.capacidad \land ($
$k \in C_0.data \land redefinida(C_0, c, k, v) \lor_L$
$|C_0.data| < c.capacidad \land claveAgregada(C_0, c, k, v) \lor_L$
$|C_0.data| = c.capacidad \land claveAgregadaFIFO(C_0, c, k, v)$
$)$

}

}

pred redefinida($C_0, c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$, in $v : V$) {

$c.data = setKey(C_0.data, k, v) \land c.claves = C_0.claves$

}

pred claveAgregada($C_0, c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$, in $v : V$) {

$c.data = setKey(C_0.data, k, v) \land c.claves = C_0.data.claves +\!+ \langle k \rangle$

}

pred claveAgregadaFIFO($C_0, c$ : CacheFIFO$\langle K, V \rangle$, in $k : K$, in $v : V$) {

$c.data = setKey(delKey(C_0.data, C_0.data.claves[0]), k, v) \land$
$c.claves = \text{tail}(C_0.data.claves) +\!+ \langle k \rangle$

}

}

b) TAD CacheLRU$\langle K, V \rangle$ {

obs capacidad : $\mathbb{Z}$

obs data : dicc$\langle K, V \rangle$

*– Claves y su ultima hora de acceso*

obs ultimosAccesos : dicc$\langle K, \mathbb{Z} \rangle$

proc nuevoCache(in $cap : \mathbb{Z}$) : CacheLRU$\langle K, V \rangle$ {

requiere { $cap > 0$ }

asegura { $res.capacidad = cap \land res.data = \langle\rangle \land res.ultimosAccesos = \langle\rangle$ }

}

proc igualdad($C_1, C_2$ : CacheLRU$\langle K, V \rangle$) : bool {

asegura {

$C_1.capacidad = C_2.capacidad$

$C_1.data = C_2.data$

$C_1.ultimosAccesos = C_2.ultimosAccesos$

}

}

proc esta(in $c$ : CacheLRU$\langle K, V \rangle$, in $k : K$) : bool {

asegura { $res = true \leftrightarrow k \in c.data$ }

}

proc obtener(inout $c$ : CacheLRU$\langle K, V \rangle$, in $k : K$) : $V$ {

requiere { $k \in c.data \land c = C_0$ }

asegura {

$res = c.data[k] \land$

$c.capacidad = C_0.capacidad \land$

$c.data = C_0.capacidad \land$

$c.ultimosAccesos = setKey(C_0.ultimosAccesos, k, horaActual())$

}

}

proc definir(inout $c$ : CacheLRU$\langle K, V \rangle$, in $k : K$, in $v : V$) {

requiere { $c = C_0$ }

asegura {

$c.capacidad = C_0.capacidad \land ($

$k \in C_0.data \land redefinida(C_0, c, k, v) \lor_L$

$|C_0.data| < c.capacidad \land claveAgregada(C_0, c, k, v) \lor_L$

$|C_0.data| = c.capacidad \land claveAgregadaLRU(C_0, c, k, v)$

$)$

}

}

pred redefinida($C_0, c$ : CacheLRU$\langle K, V \rangle$, in $k : K$, in $v : V$) {

$c.data = setKey(C_0.data, k, v) \land c.ultimosAccesos = setKey(C_0.data.ultimosAccesos, k, horaActual())$

}

pred claveAgregada($C_0, c$ : CacheLRU$\langle K, V \rangle$, in $k : K$, in $v : V$) {

$c.data = setKey(C_0.data, k, v) \land c.ultimosAccesos = setKey(C_0.data.ultimosAccesos, k, horaActual())$

}

pred claveAgregadaLRU($C_0, c$ : CacheFIFO$\langle K, V\rangle$, in $k : K$, in $v : V$) {

$(\exists kLRU : K)$ (

$esClaveLRU(C_0, kLRU) \wedge$ (

$c.data = setKey(delKey(C_0.data), kLRU, k, v) \wedge$

$c.ultimosAccesos = setKey(delKey(C_0.data.ultimosAccesos, kLRU), k, horaActual())$

)

)

}

pred esClaveLRU($C_0$ : CacheFIFO$\langle K, V\rangle$, in $kLRU : K$) {

$kLRU \in C_0.ultimosAccesos \wedge_L (\forall k : K)\ (k \in C_0.ultimosAccesos \rightarrow_L C_0.ultimosAccesos[k] \leq C_0.ultimosAccesos[kLRU])$

}

}

c) TAD CacheTTL$\langle K, V\rangle$ {

obs TTL : $\mathbb{Z}$

obs data : dicc$\langle K, V\rangle$

– *Claves y la hora que fueron agregadas*

obs horaAgregada : dicc$\langle K, \mathbb{Z}\rangle$

proc nuevoCache(in $TTL : \mathbb{Z}$) : CacheTTL$\langle K, V\rangle$ {

requiere { $TTL > 0$ }

asegura { $res.TTL = TTL \wedge res.data = \langle\rangle \wedge res.horaAgregada = \langle\rangle$ }

}

proc igualdad($C_1, C_2$ : CacheTTL$\langle K, V\rangle$) : bool {

asegura {

$C_1.TTL = C_2.TTL$

$C_1.data = C_2.data$

$C_1.horaAgregada = C_2.horaAgregada$

}

}

proc esta(in $c$ : CacheTTL$\langle K, V\rangle$, in $k : K$) : bool {

asegura { $res = true \leftrightarrow k \in c.data \wedge_L vigente(c, k)$ }

}

proc obtener(in $c$ : CacheTTL$\langle K, V\rangle$, in $k : K$) : $V$ {

requiere { $k \in c.data \wedge_L vigente(c, k)$ }

asegura { $res = c.data[k]$ }

}

proc definir(inout $c$ : CacheTTL$\langle K, V\rangle$, in $k : K$, in $v : V$) {

requiere { $c = C_0$ }

asegura {

$c.TTL = C_0.TTL \wedge$

$c.data = setKey(C_0.data, k, v) \wedge c.horaAgregada = setKey(C_0.data, k, horaActual())$

}

}

pred vigente($c$ : CacheTTL$\langle K, V\rangle, k : K$) { $horaActual() - c.horaAgregada[k] \leq c.TTL$ }

}

**Ejercicio 9.** Especifique tipos para un robot que realiza un camino a través de un plano de coordenadas cartesianas (enteras), es decir, tiene operaciones para ubicarse en un coordenada, avanzar hacia arriba, hacia abajo, hacia la derecha y hacia la izquierda, preguntar por la posición actual, saber cuántas veces pasó por una coordenada dada y saber cuál es la coordenada más a la derecha por dónde pasó.

Coord ES struct $\langle x : \mathbb{Z}, y : \mathbb{Z} \rangle$

TAD Robot {

proc ubicarse(inout $r : Robot$, in $c : Coord$)
proc arriba(inout $r : Robot$)
proc abajo(inout $r : Robot$)
proc izquierda(inout $r : Robot$)
proc derecha(inout $r : Robot$)
proc coordActual(in $r : Robot$) : $Coord$
proc másDerecha(in $r : Robot$) : $\mathbb{Z}$
proc cuantasVecesPaso(in $r : Robot$, in $c : Coord$) : $\mathbb{Z}$

}

***Solución***

Coord ES struct $\langle x : \mathbb{Z}, y : \mathbb{Z} \rangle$

TAD Robot {

obs pos : $Coord$
obs veces : dicc$\langle Coord, \mathbb{Z} \rangle$
proc nuevoRobotEn(in $p : Coord$) : $Robot$ {
requiere { $Verdadero$ }
asegura { $res.pos = p \land |res.veces| = 1 \land res.veces[p] = 1$ }
}

– ***abajo***, ***izquierda*** y ***derecha*** *son exactamente lo mismo pero con sus respectivos auxiliares.*
proc arriba(inout $r : Robot$) {
requiere { $r = R_0$ }
asegura { $r.pos = posArriba(R_0) \land r.veces = setKey(R_0, posArriba(R_0), R_0.veces[posArriba(R_0)] + 1)$ }
}

proc posiciónActual(in $r : Robot$) : $Coord$ {
requiere { $Verdadero$ }
asegura { $res = r.pos$ }
}

proc cuantasVecesPasó(in $r : Robot$, in $p : Coord$) : $\mathbb{Z}$ {
requiere { $Verdadero$ }
asegura { $res = \mathsf{IfThenElse}(p \in r.veces, r.veces[p], 0)$ }
}

proc másDerecha(in $r : Robot$) : $Coord$ {
requiere { $Verdadero$ }
asegura {
$res \in r.veces \land (\forall p : Coord)\ (p\ in r.veces \rightarrow_L p.x \leq res.x)$
}
}

aux posArriba($r : Robot$) : $Coord = \langle x : r.pos.x, y : r.pos.y + 1 \rangle$
aux posAbajo($r : Robot$) : $Coord = \langle x : r.pos.x, y : r.pos.y - 1 \rangle$
aux posIzquierda($r : Robot$) : $Coord = \langle x : r.pos.x - 1, y : r.pos.y \rangle$
aux posDerecha($r : Robot$) : $Coord = \langle x : r.pos.x + 1, y : r.pos.y \rangle$
}

## 3.4. Ejercicios de parciales

**Ejercicio 10.** Queremos modelar el funcionamiento de un vivero. El vivero arranca su actividad sin ninguna planta y con un monto inicial de dinero.

Las plantas las compramos en un mayorista que nos vende la cantidad que deseemos pero solamente de a una especie por vez. Como vivimos en un país con inflación, cada vez que vamos a comprar tenemos un precio diferente para la misma planta. Para poder comprar plantas tenemos que tener suficiente dinero disponible, ya que el mayorista no acepta fiarnos.

A cada planta le ponemos un precio de venta por unidad. Ese precio tenemos que poder cambiarlo todas las veces que necesitemos. Para simplificar el problema, asumimos que las plantas las vendemos de a un ejemplar (cada venta involucra un solo ejemplar de una única especie). Por supuesto que para poder hacer una venta tenemos que tener *stock* de esa planta y tenemos que haberle fijado un precio previamente. Además, se quiere saber en todo momento cuál es el balance de caja, es decir, el dinero que tiene disponible el vivero.

a) Indique las operaciones (procs) del TAD con todos sus parámetros.

b) Describa el TAD en forma completa, indicando sus observadores y los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición.

c) ¿Qué cambiaría si supiéramos a priori que cada vez que compramos en el mayorista pagamos exactamente el 10 % más que la vez anterior?

***Solución***

a) y b)

Planta `ES` $\mathbb{Z}$

Dinero `ES` $\mathbb{R}$

`TAD` Vivero {

`obs` balance : $Dinero$

– *Este es el precio de venta al público, que es independiente del precio mayorista*

`obs` stock : dicc$\langle Planta, \text{struct } \langle cantidad : \mathbb{Z}, precio : Dinero \rangle \rangle$

`proc abrirVivero`(in $montoInicial : Dinero$) : $Vivero$ {

`requiere` { $montoInicial > 0$ }

`asegura` { $res.balance = montoIncial \land res.stock = \langle\rangle$ }

}

`proc igualdad`(in $V_1 : Vivero$, in $V_2 : Vivero$) : bool {

`asegura` { $res = \text{true} \leftrightarrow V_1.balance = V_2.balance \land V_1.stock = V_2.stock$ }

}

– *Decidimos fijar el precio inicial con el precio mayorista, pero podríamos definirlo en 0 y sería válido*

– *Usar algún valor especial (negativo por ejemplo) para indicar "el precio no está definido" y luego*

– *usarlo en el* requiere *de* ***vender()*** *es una solución demasiado implementativa*

`proc compraMayorista`(inout $v : Vivero$, in $p : Planta$, in $cantidad : \mathbb{Z}$, in $precio : Dinero$) {

`requiere` { $v = V_0 \land cantidad > 0 \land precio > 0 \land v.balance \geq cantidad \times precio$ }

`asegura` {

$v.balance = V_0.balance - cantidad \times precio \land ($
$seAgregoMercaderiaNueva(V_0, v, p, cantidad, precio) \lor$
$seAgregoStock(V_0, v, p, cantidad, precio)$
$)$

}

}

`pred seAgregoMercaderiaNueva`($V_0, v : Vivero, p : Planta, cantidad : \mathbb{Z}, precio : Dinero$) {

$p \notin V_0.stock \land v.stock = setKey(V_0.stock, p, \langle cantidad, precio \rangle)$

}

`pred seAgregoStock`($V_0, v : Vivero, p : Planta, cantidad : \mathbb{Z}, precio : Dinero$) {

$p \in V_0.stock \land$
$v.stock = setKey(V_0.stock, p, \langle V_0.stock[p].cantidad + cantidad, V_0.stock[p].precio \rangle)$

}

```
proc ponerPrecio(inout v : Vivero, in p : Planta, in precio : Dinero) {
    requiere { v = V_0 ∧ p ∈ v.stock ∧ precio > 0 }
    asegura {
        v.balance = V_0.balance ∧
        v.stock = setKey(V_0.stock, p, ⟨V_0.stock[p].cantidad, precio⟩)
    }
}

proc vender(inout v : Vivero, in p : Planta) {
    requiere { v = V_0 ∧ p ∈ v.stock ∧_L v.stock[p] > 0 ∧ v.stock[p].cantidad > 0 }
    asegura {
        v.balance = V_0.balance + V_0.stock[p].precio ∧
        v.stock = setKey(V_0.stock, p, ⟨V_0.stock[p].cantidad − 1, V_0.stock[p].precio⟩)
    }
}

proc balance(in v : Vivero) : Dinero {
    requiere { Verdadero }
    asegura { res = v.balance }
}
}
```

c) Si sabemos por cuánto cambia el precio mayorista en cada compra, sólo nos importa el precio inicial, y de ahí en más tenemos que calcular el precio en función del precio anterior.

Concretamente necesitamos:

- O bien agregar el precio mayorista al observador *stock* o bien agregar un nuevo observador con un diccionario que para cada Planta nos diga su precio mayorista
- Modificamos `encabezadoProc`compraMayorista:
  - En el caso de una planta nueva
    - Definimos el precio mayorista inicial, además del precio de venta
    - El resto de la compra queda igual
  - En el caso de reponer stock
    - Actualizamos el precio de compra mayorista (multiplicando por 0.1)
    - Calculamos el balance en función de ese precio

**Ejercicio 11.** Se desea modelar mediante un TAD un videojuego de guerra desde el punto de vista de un único jugador. En el videojuego es posible ir a las tabernas y contratar mercenarios. Al contratarlo se nos informa el indicador de poder que tiene y el costo que tienen sus servicios. El poder de un mercenario siempre es positivo, sino nadie querría contratarlo. Los mercenarios no aceptan una promesa de pago, por lo que el jugador deberá tener el dinero suficiente para pagarle. El jugador puede juntar la cantidad de mercenarios que desee para poder formar batallones. El poder de los batallones es igual a la suma del poder de cada uno de los mercenarios que lo componen. Cada mercenario puede pertenecer a un solo batallón

El jugador comienza con un monto de dinero inicial determinado por el juego. A su vez, comienza con un sólo territorio bajo su dominio. El objetivo del juego es conquistar la mayor cantidad de territorios posible para dominar el continente. Para ello, el jugador puede tomar uno de sus batallones y atacar un territorio enemigo. Al momento de atacar se conoce la fuerza del batallón enemigo. El jugador resulta vencedor si tiene más poder que el enemigo, en ese caso se anexa el territorio y se ganan 1000 monedas. Caso contrario, se debe pagar por la derrota una suma de 500 monedas. El jugador no puede ir a pelear si no tiene dinero para financiar su derrota.

Además, se desea saber en todo momento la cantidad de territorios anexados y el dinero disponible.

Se pide:

a) Indique las operaciones (procs) del TAD con todos sus parámetros.

b) Describa el TAD en forma completa, indicando sus observadores y los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición.

***Solución***

Mercenario, Batallón, Poder `ES` $\mathbb{Z}$

`TAD` Juego {

`obs` dinero : $\mathbb{Z}$

`obs` mercenearios : dicc$\langle Mercenario, Poder\rangle$

`obs` batallones : dicc$\langle Batallón,$ conj$\langle Mercenario\rangle\rangle$

`obs` territorios : $\mathbb{Z}$

`proc iniciarJuego`(in $dineroInicial : \mathbb{Z}$) : $Juego$ {

`requiere` { $dineroInicial > 0$ }

`asegura` {

$res.dinero = dineroInicial \land$

$res.mercenarios = \langle\rangle \land$

$res.batallones = \langle\rangle \land$

$res.territorios = 1$

}

}

`proc igualdad`(in $J_1 : Juego$, in $J_2 : Juego$) : bool {

`asegura` { $res =$ true $\leftrightarrow$

$J_1.dinero = J_2.dinero \land$

$J_1.mercenarios = J_2.mercenarios \land$

$J_1.batallones = J_2.batallones \land$

$J_1.territorios = J_2.territorios$

}

}

`proc contratarGuerrero`(inout $j : Juego$, in $m : Mercenario$, in $p : Poder$, in $costo : \mathbb{Z}$) {

`requiere` {

$j = J_0 \land$

$m \notin j.mercenarios \land$

$costo \leq j.dinero \land$

$poder > 0$

}

`asegura` {

$j.dinero = J_0.dinero - costo \land$

$j.mercenarios = setKey(J_0.mercenarios, m, p) \land$

$j.batallones = J_0.batallones \land$

$j.territorios = J_0.territorios$

}

}

`proc formarBatallon`(inout $j : Juego$, in $b : Batallón$, in $ms :$ conj$\langle Mercenario\rangle$) {

`requiere` {

$j = J_0 \land$

$b \notin j.batallones \land$

$todosMercenariosContratados(j, ms)$

}

`asegura` {

$j.dinero = J_0.dinero \land$

$j.mercenarios = J_0.mercenarios \land$

$j.batallones = setKey(J_0.batallones, b, ms) \land$

$j.territorios = J_0.territorios$

}

}

```
proc lucharPorTerritorio(inout j : Juego, in b : Batallón, in poderEnemigo : Poder) {
    requiere {
        j = J_0 ∧
        b ∈ j.batallones ∧
        j.dinero > 500 ∧
        poderEnemigo > 0
    }
    asegura {
        j.mercenarios = J_0.mercenarios ∧
        j.batallones = J_0.batallones ∧
        (
            poderBatallon(j, b) ≥ poderEnemigo ∧
                j.dinero = J_0.dinero + 1000 ∧ j.territorios = J_0.territorios + 1
        ) ∨ (
            poderBatallon(j, b) < poderEnemigo ∧ j.dinero = J_0.dinero − 500
        )
    }
}

proc dineroDisponible(in j : Juego) : Dinero {
    requiere { Verdadero }
    asegura { res = j.dinero }
}

proc territoriosAnexados(in j : Juego) : ℤ {
    requiere { Verdadero }
    asegura { res = j.territorios }
}

pred todosMercenariosContratados(j : Juego, ms : conj⟨Mercenario⟩) { ms ⊆ claves(j.mercenarios) }

aux poderBatallon(j : Juego, b : Batallón) : ℤ =
        Σ_{m∈j.batallones[b]}  j.mercenarios[m]
}
```

**Ejercicio 12.** Se quiere organizar un viaje de estudios entre un grupo de estudiantes. Al definir el viaje se sabe qué personas viajan y los costos de los pasajes, así como el costo del combustible para un auto. Como el costo de pasajes es más alto que el del combustible, se quiere que todos viajen en auto, pero inicialmente no se sabe de cuántos se dispondrá.

A medida que se van confirmando autos disponibles se podrá asignar pasajeros a cada uno. Por razones de seguridad no se pueden asignar más de cuatro pasajeros a cada auto. Naturalmente el dueño de cada auto debe quedar asignado al suyo y nadie puede estar asignado a más de un auto. Una persona puede haber sido asignada a un auto pero luego confirmar el propio. Si pasa eso, el auto en el que iba a viajar esa persona antes de conseguir el suyo queda sin pasajeros para poder repartir todas esas personas.

Se quiere poder saber el costo total del viaje, es decir, el costo en pasajes sumado al costo en combustible para todos los autos.

a) Indique las operaciones (procs) del TAD con todos sus parámetros.

b) Describa el TAD en forma completa, indicando sus observadores y los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición.

**Ejemplo**

Condiciones iniciales:

- **Pasajeros**: Juan, Pedro, María, José, Laura, Carla, Manuel, Claudia
- **Costo de combustible**: $40.000 por auto
- **Costo por pasaje**: $30.000 por persona

Costo total actual: $8 \times \$30.000 = \$240.000$

- Juan consigue su auto
- Se le asignan María, José, Laura y Manuel como pasajeros

Costo total actual: $1 \times \$40.000 + 3 \times \$30.000 = \$130.000$

---

- José consigue su auto
- Se asignan a María, Manuel, Laura y Claudia al auto de Juan
- Se asignan a Pedro y Carla al auto de José

Costo total actual: $2 \times \$40.000 = \$80.000$

***Solución***

Persona, Auto `ES` $\mathbb{Z}$
`TAD` ViajeDeEstudios {
  `obs` viajeros : conj$\langle Persona \rangle$
  – *Autos y sus dueños*
  `obs` autos : dicc$\langle Auto, Persona \rangle$
  – *Autos y quiénes viajan en ese auto*
  `obs` asignaciones : dicc$\langle Auto,$ conj$\langle Persona \rangle \rangle$
  `obs` costoPasaje : $\mathbb{Z}$
  `obs` costoCombustible : $\mathbb{Z}$

  `proc nuevoViaje`(in $viajeros$ : conj$\langle Persona \rangle$, in $costoPasaje, costoCombustible : \mathbb{Z}$) : $ViajeDeEstudios$ {
    `requiere` { $|viajeros| > 0 \land costoPasaje > 0 \land costoCombustible > 0$ }
    `asegura` {
      $res.viajeros = viajeros \land$
      $res.costoPasaje = costoPasaje \land$
      $res.costoCombustible = costoCombustible \land$
      $res.autos = \langle\rangle \land$
      $res.asignaciones = \langle\rangle$
    }
  }

  `proc igualdad`(in $V_1 : ViajeDeEstudios$, in $V_2 : ViajeDeEstudios$) : bool {
    `asegura` { $res =$ true $\leftrightarrow$
      $V_1.viajeros = V_2.viajeros \land$
      $V_1.costoPasaje = V_2.costoPasaje \land$
      $V_1.costoCombustible = V_2.costoCombustible \land$
      $V_1.autos = V_2.autos \land$
      $V_1.asignaciones = V_2.asignaciones$
    }
  }

  `proc confirmarAuto`(inout $v : ViajeDeEstudios$, in $a : Auto$, in $dueño : Persona$) {
    `requiere` { $v = V_0 \land dueño$in $v.viajeros \land a \notin v.autos$ }
    `asegura` {
      $v.viajeros = V_0.viajeros \land$
      $v.costoPasaje = V_0.costoPasaje \land$
      $v.costoCombustible = V_0.costoCombustible \land$
      $v.autos = setKey(V_0.autos, a, dueño) \land$
      (
        $estaAsignado(V_0, dueño) \land seVacioElAutoEnQueViaja(V_0, v, dueño)$
      ) $\lor$ (
        $\neg estaAsignado(V_0, dueño) \land v.asignaciones = V_0.asignaciones$
      )
    }
  }

```
proc llenarAuto(inout v : ViajeDeEstudios, in a : Auto, in asignadosconj⟨Persona⟩) {
    requiere {
        v = V_0 ∧
        a ∈ v.autos ∧ a ∉ v.asignaciones
        |asignados| < 5 ∧
        asignados ⊆ v.viajeros ∧
        v.autos[a] ∈ asignados
        ningunoEstaAsignado(v, asignados)
    }
    asegura {
        v.viajeros = V_0.viajeros ∧
        v.costoPasaje = V_0.costoPasaje ∧
        v.costoCombustible = V_0.costoCombustible ∧
        v.autos = v.autos ∧
        v.asignaciones = setKey(V_0.asignaciones, a, asignados)
    }
}

proc calcularCostoTotal(in v : ViajeDeEstudios) : ℤ {
    requiere { Verdadero }
    asegura { res = costoTotalNafta(v) + costoTotalPasajes(v) }
}

pred estaAsignado(v : ViajeDeEstudios, p : Persona) {
    (∃a : Auto) (a ∈ v.asignaciones ∧_L p ∈ v.asignaciones[a])
}

pred seVacioElAutoEnQueViaja(V_0, v : ViajeDeEstudios, p : Persona) {
    (∃a : Auto) (a ∈ V_0.asignaciones ∧_L p ∈ V_0.asignaciones[a] ∧
        v.asignaciones = delKey(V_0.asignaciones, a))
}

pred ningunoEstaAsignado(v : ViajeDeEstudios, as : conj⟨Persona⟩) {
    (∀p : Persona) (p ∈ as → ¬estaAsignado(v, p))
}

aux costoTotalNafta(v : ViajeDeEstudios) : ℤ = |v.autos| × v.costoCombustible

aux costoTotalPasajes(v : ViajeDeEstudios) : ℤ =
    Σ_{p∈v.viajeros} IfThenElse(estaAsignado(v, p), 0, v.costoPasaje)
}
```