



## Guia 4

2do cuatrimestre 2024

Algoritmos y Estructuras de Datos I

Integrante	LU	Correo electrónico
Federico Barberón	112/24	jfedericobarberonj@gmail.com



**Facultad de Ciencias Exactas y Naturales**

Universidad de Buenos Aires

Ciudad Universitaria - Pabellón I

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar>

# Índice

<b>1. Guía 4</b>	<b>3</b>
1.1. Ejercicio 1 . . . . .	3
1.2. Ejercicio 2 . . . . .	3
1.3. Ejercicio 3 . . . . .	4
1.4. Ejercicio 4 . . . . .	6
1.5. Ejercicio 5 . . . . .	7
1.6. Ejercicio 6 . . . . .	8
1.7. Ejercicio 7 . . . . .	9
1.8. Ejercicio 8 . . . . .	10
1.9. Ejercicio 10 . . . . .	12

# 1. Guía 4

## 1.1. Ejercicio 1

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

```
TAD Racional {
  obs num:  $\mathbb{Z}$ 
  obs den:  $\mathbb{Z}$ 

  proc crear (in num:  $\mathbb{Z}$ , in den:  $\mathbb{Z}$ ) : Racional
    requiere {den  $\neq$  0}
    asegura {res.num = num  $\wedge$  res.den = den}

  proc suma (inout r: Racional , n: Racional )
    requiere {r =  $R_0$ }
    asegura {r.num =  $R_0$ .num * n.den + n.num *  $R_0$ .den  $\wedge$  r.den =  $R_0$ .den * n.den}

  proc resta (inout r: Racional , n: Racional )
    requiere {r =  $R_0$ }
    asegura {r.num =  $R_0$ .num * n.den - n.num *  $R_0$ .den  $\wedge$  r.den =  $R_0$ .den * n.den}

  proc multiplicación (inout r: Racional , n: Racional )
    requiere {r =  $R_0$ }
    asegura {r.num =  $R_0$ .num * n.num  $\wedge$  r.den =  $R_0$ .den * n.den}

  proc división (inout r: Racional , n: Racional )
    requiere {n.num  $\neq$  0  $\wedge$  r =  $R_0$ }
    asegura {r.num =  $R_0$ .num * n.den  $\wedge$  r.den =  $R_0$ .den * n.num}

  proc iguales (in r: Racional , n: Racional ) : Bool
    asegura {res = true  $\leftrightarrow$  r.num * n.den = r.den * n.num}
}
```

## 1.2. Ejercicio 2

Especifique mediante TADs los siguientes elementos geométricos:

1. Punto2D, que representa un punto en el plano. Debe contener las siguientes operaciones:
  - a) *nuevoPunto*: que crea un punto a partir de sus coordenadas  $x$  e  $y$ .
  - b) *mover*: que mueve el punto una determinada distancia sobre los ejes  $x$  e  $y$ .
  - c) *distancia*: que devuelve la distancia entre dos puntos:
  - d) *distanciaAlOrigen*: que devuelve la distancia del punto  $(0,0)$ .
2. Rectangulo2D, que representa un rectángulo en el plano. Debe contener las siguientes operaciones:
  - a) *nuevoRectangulo*: que crea un rectángulo (decida usted cuáles deberían ser los parámetros)
  - b) *mover*: que mueve el rectángulo una determinada distancia en los ejes  $x$  e  $y$ .
  - c) *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.
  - d) *estaContenido*: que dados dos rectángulos, indique si uno está contenido en el otro.

```
TAD Punto2D {
  obs x:  $\mathbb{R}$ 
  obs y:  $\mathbb{R}$ 
```

```

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

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

proc distancia (in p1: Punto2D , in p2: Punto2D ) :  $\mathbb{R}$ 
  asegura  $\{res = dist(p1.x, p1.y, p2.x, p2.y)\}$ 

proc distanciaAlOrigen (in p: Punto2D ) :  $\mathbb{R}$ 
  asegura  $\{res = dist(p1.x, p1.y, 0, 0)\}$ 

aux dist (x1:  $\mathbb{R}$ , y1:  $\mathbb{R}$ , x2:  $\mathbb{R}$ , y2:  $\mathbb{R}$ ) :  $\mathbb{R} = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ ;
}

TAD Rectangulo2D {
  obs centro: Punto2D
  obs largo:  $\mathbb{R}$ 
  obs alto:  $\mathbb{R}$ 

  proc nuevoRectangulo (in c: Punto2D, in l:  $\mathbb{R}$ , in a:  $\mathbb{R}$ ) : Rectangulo2D
    requiere  $\{l, a > 0\}$ 
    asegura  $\{res.centro = centro \wedge res.largo = l \wedge res.alto = a\}$ 

  proc mover (inout r: Rectangulo2D , in dx:  $\mathbb{R}$ , in dy:  $\mathbb{R}$ )
    requiere  $\{r = R_0\}$ 
    asegura  $\{r.largo = R_0.largo \wedge r.alto = R_0.alto\}$ 
    asegura  $\{r.centro.x = R_0.x + dx \wedge r.centro.y = R_0.y + dy\}$ 

  proc escalar (inout r: Rectangulo2D ,  $\alpha$  :  $\mathbb{R}$ )
    requiere  $\{r = R_0 \wedge \alpha \neq 0\}$ 
    asegura  $\{r.centro = R_0.centro\}$ 
    asegura  $\{r.largo = \alpha * R_0.largo \wedge r.alto = \alpha * R_0.alto\}$ 

  proc estaContenido (in r1: Rectangulo2D , in r2: Rectangulo2D ) : Bool
    ESTA MAL
    asegura  $\{res = true \leftrightarrow esIgualOMasChico(r1, r2) \wedge$ 
       $dist(r1.centro.x, r1.centro.y, r2.centro.x, r2.centro.y) \leq distAlVertice(r1)\}$ 

  pred esIgualOMasChico (r1: Rectangulo2D , r2: Rectangulo2D ) {
     $r1.largo \leq r2.largo \wedge r1.alto \leq r2.alto$ 
  }

  aux distAlVertice (r: Rectangulo2D ) :  $\mathbb{R} = \sqrt{\left(\frac{r.largo}{2}\right)^2 + \left(\frac{r.alto}{2}\right)^2}$ ;
}

```

### 1.3. Ejercicio 3

1. Especifique el TAD Cola<T> con las siguientes operaciones:
  - a) *nuevaCola*: que crea una cola vacía.
  - b) *estáVacía*: que devuelve true si la cola no contiene elementos.
  - c) *encolar*: que agrega un elemento al final de la cola:
  - d) *desencolar*: que elimina el primer elemento de la cola y lo devuelve.
2. Especifique el TAD Pila<T> con las siguientes operaciones:
  - a) *nuevaPila*: que crea una pila vacía

- b) *estáVacía*: que devuelve true si la pila no contiene elementos
  - c) *apilar*: que agrega un elemento al tope de la pila
  - d) *desapilar*: que elimina el elemento del tope de la pila y lo devuelve
3. Especifique el TAD  $\text{DobleCola}\langle T \rangle$ , en el que los elementos pueden insertarse al principio o al final y se eliminan por el medio. Debe contener las operaciones *nuevaDobleCola*, *estáVacía*, *encolarAdelante*, *encolarAtrás* y *desencolar*

```

TAD Cola<T> {
  obs data: seq<T>

  proc nuevaCola () : Cola<T>
    asegura {res.data = <>}

  proc estáVacía (in c: Cola<T> ) : Bool
    asegura {res = true ↔ |c.data| = 0}

  proc encolar (inout c: Cola<T> , in e: T)
    requiere {c = C0}
    asegura {c.data = concat(C0.data, <e>)}

  proc desencolar (inout c: Cola<T> ) : T
    requiere {c = C0 ∧ |c.data| > 0}
    asegura {c.data = tail(C0.data) ∧ res = head(C0.data)}
}

```

```

TAD Pila<T> {
  obs data: seq<T>

  proc nuevaPila () : Pila<T>
    asegura {res.data = <>}

  proc estáVacía (in p: Pila<T> ) : Bool
    asegura {res = true ↔ |p.data| = 0}

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

  proc desapilar (inout p: Pila<T> ) : T
    requiere {p = P0 ∧ |p.data| > 0}
    asegura {p.data = tail(P0.data) ∧ res = head(P0.data)}
}

```

```

TAD DobleCola<T> {
  obs data: seq<T>

  proc nuevaDobleCola () : DobleCola<T>
    asegura {res.data = <>}

  proc estáVacía (in c: DobleCola<T> ) : Bool
    asegura {res = true ↔ |c.data| = 0}

  proc encolarAdelante (inout c: DobleCola<T> , in e: T)
    requiere {c = C0}
    asegura {c.data = concat(<e>, C0.data)}

  proc encolarAtrás (inout c: DobleCola<T> , in e: T)
    requiere {c = C0}
    asegura {c.data = concat(C0.data, <e>)}
}

```

```

proc desencolar (inout c: DobleCola(T) ) : T
  requiere {c = C0 ∧ |c.data| > 0}
  asegura {res = C0.data[iMedio(C0.data)] ∧
    c.data = subseq(C0.data, 0, iMedio(C0.data)) ++ subseq(C0.data, iMedio(C0.data) + 1, |C0.data|)}

aux iMedio (s: seq(T)) :  $\mathbb{Z}$  =  $\left\lfloor \frac{|s| - 1}{2} \right\rfloor$ ;
}

```

#### 1.4. Ejercicio 4

1. Especifique el TAD *Diccionario*< *K*, *V* > con las siguientes operaciones:

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

2. Especifique el TAD *DiccionarioConHistoria*< *K*, *V* >. 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.

```

TAD Diccionario<K, V> {
  obs data: dict<K, V>

  proc nuevoDiccionario () : Diccionario<K, V>
    asegura {res.data = {}}

  proc definir (inout d: Diccionario<K, V> , in k: K, in v: V)
    requiere {d = D0}
    asegura {d.data = setKey(D0.data, k, v)}

  proc obtener (in d: Diccionario<K, V> , in k: K) : V
    requiere {k ∈ d.data}
    asegura {res = d.data[k]}

  proc esta (in d: Diccionario<K, V> , in k: K) : Bool
    asegura {res = true ↔ k ∈ d.data}

  proc borrar (inout d: Diccionario<K, V> , in k: K)
    requiere {k ∈ d.data ∧ d = D0}
    asegura {d.data = delKey(D0.data, k)}
}

TAD DiccionarioConHistoria<K, V> {
  obs data: dict<K, seq(V)>

  proc nuevoDiccionario () : DiccionarioConHistoria<K, V>
    asegura {res.data = {}}

  proc definir (inout d: DiccionarioConHistoria<K, V> , in k: K, in v: V)
    requiere {d = D0}
    asegura {
      (k ∈ D0.data → d.data = setKey(D0.data, k, concat(<v>, D0.data[k]))) ∧
      (k ∉ D0.data → d.data = setKey(D0.data, k, <v>))}

  proc obtener (in d: DiccionarioConHistoria<K, V> , in k: K) : V
    requiere {k ∈ d.data}
}

```

```

    asegura {res = d.data[k]}

proc esta (in d: DiccionarioConHistoria $\langle K, V \rangle$  , in k: K) : Bool
    asegura {res = true  $\leftrightarrow k \in d.data$ }

proc borrar (inout d: DiccionarioConHistoria $\langle K, V \rangle$  , in k: k)
    requiere { $k \in d.data \wedge d = D_0$ }
    asegura {
        (cantValores( $D_0, k$ ) = 1  $\rightarrow d.data = \text{delKey}(D_0.data, k)$ )  $\wedge$ 
        (cantValores( $D_0, k$ ) > 1  $\rightarrow d.data = \text{setKey}(D_0.data, k, \text{tail}(D_0.data[k]))$ )
    }
    aux cantValores (d: DiccionarioConHistoria $\langle K, V \rangle$  , in k: K) :  $\mathbb{Z} = |d.data[k]|$  ;
}

```

### 1.5. Ejercicio 5

Especifique los TADs indicados a continuación pero utilizando los observadores propuestos:

- Diccionario $\langle K, V \rangle$  observado con conjunto (de tuplas)
- Conjunto $\langle T \rangle$  observado con funciones
- Pila $\langle T \rangle$  observado con diccionarios
- Punto observado con coordenadas polares

```

TAD Diccionario $\langle K, V \rangle$  {
    obs data: conj $\langle \text{tupla}\langle K, V \rangle \rangle$ 

    proc nuevoDiccionario () : Diccionario $\langle K, V \rangle$ 
        asegura {res.data = {}}

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

    proc obtener (in d: Diccionario $\langle K, V \rangle$  , in k: K) : V
        requiere {perteneceKey( $d, k$ )}
        asegura {( $\forall t : \text{tupla}\langle K, V \rangle$ ) ( $(t \in d.data \wedge t_0 = k) \rightarrow res = t_1$ )}

    proc esta (in d: Diccionario $\langle K, V \rangle$  , in k: K) : Bool
        asegura {res = true  $\leftrightarrow \text{perteneceKey}(d, k)$ }

    proc borrar (inout d: Diccionario $\langle K, V \rangle$  , in k: k)
        requiere {perteneceKey( $d, k$ )  $\wedge d = D_0$ }
        asegura {( $\forall t : \text{tupla}\langle K, V \rangle$ ) ( $t \in d.data \leftrightarrow (t_0 \neq k \wedge t \in D_0.data)$ )}

    pred perteneceKey (d: Diccionario $\langle K, V \rangle$  , in k: K) {
        ( $\exists t : \text{tupla}\langle K, V \rangle$ ) ( $t \in d.data \wedge t_0 = k$ )
    }
}

```

```

TAD Punto {
    HACER! obs radio:  $\mathbb{R}$ 
    obs angle:  $\mathbb{R}$ 

    proc nuevoPunto (in r:  $\mathbb{R}$ , in a:  $\mathbb{R}$ ) : Punto
        asegura {res.radio = r  $\wedge$  res.angle = a}

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

```

```

proc distancia (in p1: Punto , in p2: Punto ) : ℝ
  asegura {res = dist(p1.x, p1.y, p2.x, p2.y)}

proc distanciaAlOrigen (in p: Punto ) : ℝ
  asegura {res = dist(p1.x, p1.y, 0, 0)}
}

```

## 1.6. 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.

### 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.

### a) TAD Multiconjunto $\langle T \rangle$ {

obs bag: dict $\langle T, \mathbb{Z} \rangle$

proc nuevoMulticonjunto () : Multiconjunto $\langle T \rangle$

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

proc pertenece (in m: Multiconjunto $\langle T \rangle$  , in e: T) : Bool

asegura { $res = \text{true} \leftrightarrow e \in m.bag$ }

proc agregar (inout m: Multiconjunto $\langle T \rangle$  , in e: T)

requiere { $m = M_0$ }

asegura { $m.bag = \text{setKey}(M_0.bag, e, \text{IfThenElse}(e \in M_0.bag, M_0.bag[e] + 1, 1))$ }

proc sacar (inout m: Multiconjunto $\langle T \rangle$  , in e: T)

requiere { $m = M_0 \wedge e \in m.bag$ }

asegura { $(M_0.bag[e] = 1 \rightarrow m.bag = \text{delKey}(M_0.bag, e)) \wedge$   
 $(M_0.bag[e] > 1 \rightarrow m.bag = \text{setKey}(M_0.bag, e, M_0.bag[e] - 1))$ }

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

requiere { $e \in m.bag$ }

asegura { $res = m.bag[e]$ }

}



```

b) TAD Multidict<K, V> {
  obs data: dict<K, seq<V>>

  proc nuevoMultidict () : Multidict<K, V>
    asegura {res.data = {}}

  proc definir (inout m: Multidict<K, V> , in k: K, in v: V)
    requiere {m = M0}
    asegura {
      m.data = setKey(M0.data, k, IfThenElse(k ∈ M0.data, concat(<v>, M0.data[k]), <v>))
    }

  proc obtener (in m: Multidict<K, V> , in k: K) : V
    requiere {k ∈ m.data}
    asegura {res = m.data[k]}

  proc pertenece (in m: Multidict<K, V> , in k: K) : Bool
    asegura {res = true ↔ k ∈ m.data}

  proc borrar (inout m: Multidict<K, V> , in k: K) : V
    requiere {m = M0 ∧ k ∈ m.data}
    asegura {
      res = M0.data[k][0] ∧
      (|M0.data[k]| = 1 → m.data = delKey(M0.data, k)) ∧
      (|M0.data[k]| > 1 → m.data = setKey(M0.data, k, tail(M0.data[k])))
    }
}

```

### 1.7. 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).

```

TAD Contadores {
  obs eventos: dict⟨String, ℤ⟩

  proc nuevoContadores (in eventos: seq⟨str⟩) : Contadores
    requiere {sinRepetidos(eventos)}
    asegura {
      (∀e : String) (e ∈ eventos ↔ e ∈ res.eventos)
      (∀e : String) (e ∈ res.eventos →L res.eventos[e] = 0)
    }

  proc incrementar (inout c: Contadores , in e: String)
    requiere {e ∈ c.eventos ∧ c = C0}
    asegura {c.eventos = setKey(C0.eventos, e, C0.eventos[e] + 1)}

  proc cantidadDeVeces (in c: Contadores , in e: String) : ℤ
    requiere {e ∈ c.eventos}
    asegura {res = c.eventos[e]}

  pred sinRepetidos (s: seq⟨T⟩) {
    (∀i, j : ℤ) (0 ≤ i < j < |s| →L s[i] ≠ s[j])
  }
}

```

```

TAD ContadoresConFecha {
  HACER!
  obs eventos: dict⟨String, seq⟨ℤ⟩⟩

  proc nuevoContadores (in eventos: seq⟨str⟩) : ContadoresConFecha
    requiere {sinRepetidos(eventos)}
    asegura {
      (∀e : String) (e ∈ eventos ↔ e ∈ res.eventos)
      (∀e : String) (e ∈ res.eventos →L res.eventos[e] = ⟨⟩)
    }

  proc incrementar (inout c: ContadoresConFecha , in e: String, in fechaActual: ℤ)
    requiere {e ∈ c.eventos ∧ c = C0}
    asegura {c.eventos = setKey(C0.eventos, e, concat(C0.eventos[e], ⟨fechaActual⟩))}

  proc cantidadDeVeces (in c: ContadoresConFecha , in e: String, in fechaEvento: ℤ) : ℤ
    requiere {e ∈ c.eventos}
    asegura {}

  pred sinRepetidos (s: seq⟨T⟩) {
    (∀i, j : ℤ) (0 ≤ i < j < |s| →L s[i] ≠ s[j])
  }
}

```

## 1.8. 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 comunmente 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).

- 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).

TAD  $\text{CacheFIFO}\langle K, V \rangle \{$

obs data:  $\text{dict}\langle K, V \rangle$

obs orden:  $\text{seq}\langle K \rangle$

obs max:  $\mathbb{Z}$

proc nuevaCache (in max:  $\mathbb{Z}$ ) :  $\text{CacheFIFO}\langle K, V \rangle$

requiere  $\{max > 0\}$

asegura  $\{res.data = \{\} \wedge res.orden = \langle \rangle \wedge res.max = max\}$

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

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

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

requiere  $\{k \in c.data\}$

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

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

requiere  $\{c = C_0\}$

asegura {

$c.max = C_0.max$

$(k \in C_0.data \rightarrow c.data = \text{setKey}(C_0, k, v) \wedge \text{mueveAlFinal}(c, C_0, k, v)) \wedge$

$(k \notin C_0.data \rightarrow (c.data = \text{setKey}(\text{delKey}(C_0, C_0.orden[0]), k, v) \wedge$

$\text{poneAlFinal}(c, C_0, k) \wedge \text{sacaElPrimero}(c, C_0)))$

}

pred mueveAlFinal (c:  $\text{CacheFIFO}\langle K, V \rangle$ ,  $C_0$ :  $\text{CacheFIFO}\langle K, V \rangle$ , k: K) {

$|c.orden| = |C_0.orden| \wedge_L (\exists i : \mathbb{Z}) (0 \leq i < |C_0.orden| \wedge_L C_0.orden[i] = k \wedge (\forall j : \mathbb{Z}) (i < j < |C_0.orden| \rightarrow_L$

}

pred poneAlFinal (c:  $\text{CacheFIFO}\langle K, V \rangle$ ,  $C_0$ :  $\text{CacheFIFO}\langle K, V \rangle$ , k: K) {

$c.orden = \text{concat}(C_0.orden, \langle k \rangle)$

}

pred sacaElPrimero (c:  $\text{CacheFIFO}\langle K, V \rangle$ ,  $C_0$ :  $\text{CacheFIFO}\langle K, V \rangle$ ) {

$c = \text{subseq}(C_0, 1, |C_0|)$

}

}

TAD  $\text{CacheLRU}\langle K, V \rangle \{$

**HACER!**

}

TAD  $\text{CacheTTL}\langle K, V \rangle \{$

**HACER!**

}

## 1.9. 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.

- Indique las operaciones (procs) del TAD con todos sus parámetros.
- Describa el TAD en forma completa, indicando sus observadores, los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición.
- ¿Qué cambiaría si supiéramos a priori que cada vez que compramos en el mayorista pagamos exactamente el 10 % más que la vez anterior? Describa los cambios en palabras.

Tomé la decisión de representar una especie sin precio definido poniendo  $precio = -1$

```
TAD Vivero {
  obs plantas: dict⟨String, struct⟨precio : ℝ, stock : ℤ⟩⟩
  obs dinero: ℝ

  proc nuevoVivero (in monto: ℝ) : Vivero
    requiere {monto ≥ 0}
    asegura {res.plantas = {} ∧ res.dinero = monto}

  proc comprar (inout v: Vivero , in especie: String, in cantidad: ℤ, in precios: dict⟨String, ℝ⟩)
    requiere {especie ∈ precios ∧L v.dinero ≥ precios[especie] ∧ v = V0}
    asegura {
      v.dinero = V0.dinero - precios[especie] ∧
      (hayEspecie(V0, especie) →
        v.plantas = actualizarEspecie(V0, especie, precio(V0, especie), stock(V0, especie) + cantidad)) ∧
      (¬hayEspecie(V0, especie) → v.plantas = actualizarEspecie(V0, especie, -1, cantidad))
    }

  proc ponerPrecio (inout v: Vivero , in especie: String, in precio: ℝ)
    requiere {hayEspecie(v, especie) ∧ v = V0 ∧ precio ≥ 0}
    asegura {
      v.dinero = V0.dinero ∧
      v.plantas = actualizarEspecie(V0, especie, precio, stock(V0, especie))
    }

  proc vender (inout v: Vivero , in especie: String)
    requiere {hayEspecie(v, especie) ∧L stock(v, especie) > 0 ∧ precio(v, especie) ≠ -1 ∧ v = V0}
    asegura {
      v.dinero = V0.dinero + precio(V0, especie) ∧
      (stock(V0, especie) = 1 → v.plantas = eliminarEspecie(V0, especie)) ∧
      (stock(V0, especie) > 1 →
        v.plantas = actualizarEspecie(V0, especie, precio(V0, especie), stock(V0, especie) - 1))
    }
}
```

```

proc obtenerBalance (in v: Vivero ) :  $\mathbb{R}$ 
  asegura {res = v.dinero}
aux precio (v: Vivero , especie: String) :  $\mathbb{R}$  = v.plantas[especie].precio ;
aux stock (v: Vivero , especie: String) :  $\mathbb{Z}$  = v.plantas[especie].stock ;
aux actualizarEspecie (v: Vivero , e: String, p:  $\mathbb{R}$ , s:  $\mathbb{Z}$ ) : dict⟨String, struct⟨precio :  $\mathbb{R}$ , stock :  $\mathbb{Z}$ ⟩⟩ =
  setKey(v.plantas, e, {precio : p, stock : s}) ;
aux eliminarEspecie (v: Vivero , e: String) : dict⟨String, struct⟨precio :  $\mathbb{R}$ , stock :  $\mathbb{Z}$ ⟩⟩ =
  delKey(v.plantas, especie) ;
pred hayEspecie (v: Vivero , especie: String) {
  especie ∈ v.plantas
}
}

```