

# Guia 4 - TAD

April 2024

## 1. Ejercicio 1

```
TAD Racional {
  obs Num: $\mathbb{Z}$ 
  obs Den: $\mathbb{Z}$ 
}

proc nuevoNumero (in x:  $\mathbb{Z}$ , in y: $\mathbb{Z}$ ) : Racional
  requiere  $\{y \neq 0\}$ 
  asegura  $\{res.Num = x \wedge res.Den = y\}$ 

proc suma (inout x:Racional,in y:Racional)
  requiere  $\{x = x0\}$ 
  asegura  $\{x.Num/x.Den = x0.num/x0.Den + y.Num/y.Den\}$ 

proc resta (inout x:Racional,in y:Racional)
  requiere  $\{x = x0\}$ 
  asegura  $\{x.Num/x.Den = x0.num/x0.Den - y.Num/y.Den\}$ 

proc multiplicacion (inout x:Racional,in y:Racional)
  requiere  $\{x = x0\}$ 
  asegura  $\{x.Num/x.Den = x0.Num * y.Num/x0.Den * y.Den\}$ 

proc division (inout x:Racional,in y:Racional)
  requiere  $\{x = x0\}$ 
  asegura  $\{x.Num/x.Den = x0.Num * y.Den/x0.Den * y.Num\}$ 
}
```

## 2. Ejercicio 2

### 2.1. A)

```
TAD Rectangulo {
  obs base: $\mathbb{R}$ 
  obs altura: $seq\langle\mathbb{R}\rangle$ 
  obs angulo: $\mathbb{R}$ 
  obs centro: $seq\langle\mathbb{R}\rangle$ 
}

proc nuevoRectangulo (in b: $\mathbb{R}$ , in h: $\mathbb{R}$ , in angulo: $\mathbb{R}$ , in centro: $seq\langle\mathbb{R}\rangle$ ) : Rectangulo
  requiere  $\{|centro| = 2\}$ 
  asegura  $\{res.base = b \wedge res.altura = h \wedge res.angulo = angulo \bmod 2\pi \wedge res.centro = centro\}$ 

proc rotar (inout r:Rectangulo , in angulo: $\mathbb{R}$ )
  requiere  $\{r = r0\}$ 
```

```

    asegura { $r.base = r0.base \wedge r.altura = r0.altura \wedge r.centro = r0.centro \wedge$ 
 $r.angulo = (r0.angulo + angulo) \bmod 2\pi$ }

proc trasladar (inout r:Rectangulo , in coord:seq( $\mathbb{R}$ ))
    requiere { $r = r0 \wedge |coord| = 2$ }
    asegura { $r.base = r0.base \wedge r.altura = r0.altura \wedge r.angulo = r0.angulo \wedge$ 
 $r.centro = [r0.centro[0] + coord[0], r0.centro[1] + coord[1]]$ }

proc escalar (inout r:Rectangulo, in b: $\mathbb{R}$ , in h: $\mathbb{R}$ )
    requiere { $r = r0 \wedge b > 0 \wedge h > 0$ }
    asegura { $r.base = r0.base * b \wedge r.altura = r0.altura * h \wedge r.centro = r0.centro \wedge$ 
 $r.angulo = r0.angulo$ }

proc area (in r:Rectangulo) :  $\mathbb{R}$ 
    requiere {True}
    asegura { $res = r.base * r.altura$ }

}

```

## 2.2. B)

```

TAD Esfera {
    obs obs radio: $\mathbb{R}$ 
    obs obs centro:seq( $\mathbb{R}$ )
    obs rotacion:seq( $\mathbb{R}$ )

proc nuevaEsfera (in r: $\mathbb{R}$  , c:seq( $\mathbb{R}$ ), in angulos:seq( $\mathbb{R}$ )) : Esfera
    requiere { $r > 0 \wedge |angulos| = 3$ }
    asegura { $res.radio = r \wedge res.centro = c \wedge$ 
 $res.rotacion = [angulos[0] \bmod 2\pi, angulos[1] \bmod 2\pi, angulos[2] \bmod 2\pi]$ }

proc rotar (inout e:Esfera, in x: $\mathbb{R}$ , in y: $\mathbb{R}$ , in z: $\mathbb{R}$ )
    requiere { $e = e0 \wedge |rotacion| = 3$ }
    asegura { $e.radio = e0.radio \wedge e.centro = e0.centro \wedge$ 
 $e.rotacion = [e0.rotacion[0] + x \bmod 2\pi, e0.rotacion[1] + y \bmod 2\pi, e0.rotacion[2] + z \bmod 2\pi]$ }

proc trasladar (inout e:Esfera, in coord:seq( $\mathbb{R}$ ))
    requiere { $e = e0 \wedge |coord| = 3$ }
    asegura { $e.radio = e0.radio \wedge e.rotacion = e0.rotacion \wedge$ 
 $e.centro = [e0.centro[0] + coord[0], e0.centro[1] + coord[1], e0.centro[2] + coord[2]]$ }

proc escalar (inout e:Esfera, in r: $\mathbb{R}$ )
    requiere { $e = e0 \wedge r > 0$ }
    asegura { $e.rotacion = e0.rotacion \wedge e.centro = e0.centro \wedge e.radio = e0.radio * r$ }

}

```

## 3. Ejercicio 3

```

TAD DobleCola< T > {
    obs elems: seq( $T$ )
    obs medio: seq( $T$ )  $\longrightarrow$  devuelve el elemento medio de la cola.

proc nuevaDobleCola () : DobleCola< T >

```

```

    requiere {true}
    asegura {res.elems = {}  $\wedge$  res.medio = {}}

proc encolarAdelante (c:DobleCola< T >, e:T)
    requiere {c = c0}
    asegura {c.elems = concat([e], c0.elems)  $\wedge$  c.medio = c0.medio}

proc encolarAtras (c:DobleCola< T >, e:T)
    requiere {c = c0}
    asegura {c.elems = concat(c0.elems, [e])  $\wedge$  c.medio = c0.medio}

proc desencolar (inout c:DobleCola< T >) : T
    requiere {c = c0  $\wedge$  c.elems  $\neq$  {}}
    asegura {( |c0.elems| = 1  $\wedge$  c.elems = [ ]  $\wedge$  res = c0.elems[0])  $\vee$ 
    |c0.elems| > 1  $\wedge$  c.elems = concat(subseq(c0.elems, 0, c0.medio[0]), subseq(c0.elems, c0.medio[0] +
    1, |c0.elems|)  $\wedge$  res = c0.medio[0])}

```

## 4. Ejercicio 4

```

TAD Dicksionario< K, seq<T> > {
    obs data: dict< K, seq<T> >

proc nuevoDick () : Dicksionario< K, seq<T> >
    requiere {true}
    asegura {res.data = {}}

proc definirHistorial (inout d:Dicksionario< K, seq<T> >, in k:K, in e:T)
    requiere {d = d0}
    asegura {d.data = setKey(d0.data, k, concat(d0.data[k], [e]))}

}

```

## 5. Ejercicio 5

```

TAD ColaDePrioridad<T>{
    obs cola:dict<T,  $\mathbb{R}$ >

proc desapilarMax (inout c:ColaDePrioridad<T>) : <T>
    requiere {c = c0}
    asegura {(  $\forall i : \mathbb{Z}$ ) (0  $\leq i < |res| \longrightarrow_L esPriMax(c0, res[i])$   $\wedge$  res[i]  $\notin$  c.colas  $\wedge$ 
    (( $\forall k : T$ ) (k  $\in$  c.colas  $\longrightarrow_L k \in c0.colas)))$ }

pred esPriMax (c:ColaDePrioridad<T>, k : T){
    k  $\in$  c  $\wedge$  ( $\forall k2 : T$ ) (k2  $\in$  c  $\longrightarrow_L c[k] \geq c[k2]$ )
}
}

```

## 6. Ejercicio 6

### 6.1. A)

TAD Dicksionario {

```

    obs data:conj( $\langle (K, V) \rangle$ )

proc dickcionarioVacio () : Dicksionario
    requiere  $\{True\}$ 
    asegura  $\{res.data = \{\}\}$ 

pred esta (dc:Dicksionario, k:K) {
     $(\exists v : V) ((k, v) \in dc.data)$ 
}

proc definir (inout dc:Dicksionario, in k:K, in v:V)
    requiere  $\{dc = dc0\}$ 
    asegura  $\{(\neg esta(dc0.data, k) \wedge dc.data = dc0.data \cup (k, v)) \vee (esta(dc0.data, k) \wedge (\exists u : V) ((k, u) \in dc0.data \wedge dc.data = (dc0.data - (k, u)) \cup (k, v)))\}$ 

proc obtener (in dc:Dicksionario, in k:K) : V
    requiere  $\{true\}$ 
    asegura  $\{(\exists i : \mathbb{Z}) (0 \leq i < |dc.data| \wedge_L (dc.data[i][0] = k \wedge res = dc.data[i][1]))\}$ 

proc borrar (inout dc:Dicksionario, in k:K)
    requiere  $\{dc = dc0 \wedge esta(dc0.data, k)\}$ 
    asegura  $\{(\exists v : V) ((k, v) \in dc0.data \wedge_L dc.data = dc0.data - (k, v))\}$ 

proc length (in dc:Dicksionario) :  $\mathbb{Z}$ 
    requiere  $\{true\}$ 
    asegura  $\{res = |dc.data|\}$ 

```

## 6.2. B)

[Esta resuelto en la teorica.](#)

## 6.3. C)

```

TAD Pila {
    obs pila:dict( $\mathbb{R}, T$ )

proc nuevaPila () : Pila
    requiere  $\{true\}$ 
    asegura  $\{res.pila = \{\}\}$ 

proc apilar (inout p:Pila, in k:T)
    requiere  $\{p = p0\}$ 
    asegura  $\{p.pila = setKey(p0.pila, |p0.pila|, k)\}$ 

proc desapilar (p:Pila) : T
    requiere  $\{p = p0 \wedge p0.pila \neq \{\}\}$ 

```

asegura  $\{res = p0.pila[|p.pila| - 1] \wedge p.pila = delKey(p0.pila, |p0.pila| - 1)\}$

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

#### 6.4. D)

[Esta resuelto en la teorica.](#)

### 7. Ejercicio 7

#### 7.1. A)

```
TAD Multiconjunto {
  obs apariciones(e: T): $\mathbb{Z}$ 

proc nuevoConjunto () : Multiconjunto[T]
  requiere  $\{true\}$ 
  asegura  $\{(\forall e : T) (res.apariciones(e) = 0)\}$ 

proc agregar (inout m:Multiconjunto, in e:T)
  requiere  $\{m = m0\}$ 
  asegura  $\{(\forall t : T) (t \neq e \longrightarrow_L m.apariciones(t) = m0.apariciones(t) \wedge m.apariciones(e) = m.apariciones(e) + 1)\}$ 

pred pertenece (m:Multiconjunto, e:T) {
   $m.cantApariciones(e) > 0$ 
}

proc sacar (inout m:Multiconjunto,in e:T)
  requiere  $\{m = m0 \wedge e \in m.elems\}$ 
  asegura  $\{(\forall t : T) (t \neq e \longrightarrow_L m.apariciones(t) = m0.apariciones(t) \wedge m.apariciones(e) = m.apariciones(e) - 1)\}$ 

proc multiplicidad (in m:Multiconjunto, in e:T) :  $\mathbb{Z}$ 
  requiere  $\{e \in m\}$ 
  asegura  $\{res = m.apariciones(e)\}$ 
```

#### 7.2. B)

```
TAD Multidick {
  obs data:dict[K,seq⟨T⟩]

proc nuevoDick () : Multidick[K,seq⟨T⟩]
```

```

    requiere {true}
    asegura {res.data = {}}

proc definir (inout m:Multidick, k:K, v:T)
    requiere {m = m0}
    asegura {k ∈ m0.data ∧ m.data = setKey(m0.data, k, concat(m0.data[k], [v]) ∨ k ∉ m0.data ∧
m.data = setKey(m0.data, k, [v])}

proc obtener (in m:Multidick, in k:K) : seq⟨T⟩
    requiere {k ∈ m}
    asegura {res = m.data[k]}

proc borrar (inout m:Multidick, in k:K )
    requiere {m = m0 ∧ k ∈ m.data}
    asegura {m.data[k] = subseq(m0, 0, |m0.data| - 1)}

proc borrarTodo (inout m:Multidick, in k:K)
    requiere {m = m0 ∧ k ∈ m.data}
    asegura {m.data = delKey(m0.data, k)}

```

## 8. Ejercicio 8

```

TAD Contador {
    obs datos: dict[T,ℤ]

proc contadorVacio () : Contador
    requiere {true}
    asegura {res.datos = {}}

proc incrementar (inout c:Contador, in e:T)
    requiere {c = c0 ∧ e ∈ c.datos}
    asegura {c.datos[e] = setKey(c0.datos, e, c0.datos[e] + 1)}

proc cantidad (in c:Contador, in e:T) : ℤ
    requiere {e ∈ c.datos}
    asegura {res = c.datos[e]}

```

## 9. Ejercicio 9

```

obs orden:seq⟨T⟩

proc incrementarSoloAlgunos (inout c:Contador, in porcentaje:ℝ)
    requiere {c = c0}

```

$\text{asegura } \{(\forall i : \mathbb{Z}) (0 \leq i < \text{cuantos}(c0.\text{datos}, \text{porcentaje}) \longrightarrow_L$   
 $c.\text{datos}[c0.\text{orden}[i]] = c0.\text{datos}[\text{orden}[i]] + 1)\}$   
 $\text{asegura } \{((\forall j : \mathbb{Z}) (\text{cuantos}(c0.\text{datos}, \text{porcentaje}) < j < |c0.\text{orden}| \longrightarrow_L$   
 $c.\text{datos}[c0.\text{orden}[j]] = c0.\text{datos}[c0.\text{orden}[j]]))\}$

## 10. Ejercicio 10

### 10.1. A)

TAD CacheFIFO[K,V] {  
  obs cap:ℤ  
  obs claves:seq⟨K⟩  
  obs data:dict[K,V]  
  
proc nuevoCache (cap:ℤ) : CacheFIFO[K,V]  
  asegura {res.cap = cap ∧ res.claves = {} ∧ res.data = {}}  
  
proc esta (in c:CacheFIFO, in k:K) : bool  
  requiere {true}  
  asegura {res = true ⇔ k ∈ c.data}  
  
proc obtener (in c:CacheFIFO, in k:K) : V  
  requiere {k ∈ c.data}  
  asegura {res = c.data[k]}  
  
proc definir (inout c:CacheFIFO, in k:K, in v:V)  
  requiere {c = c0}  
  asegura {k ∈ c0.claves → c.claves = c0.claves ∧ c.data = setKey(c0.data, k, v)}  
  asegura {k ∉ c0.claves ∧ |c0.claves| < cap → c.claves = concat(c0.claves, k) ∧  
  c.data = setKey(c0.data, k, v)}  
  asegura {k ∉ c0.claves ∧ |c0.claves| = cap → c.claves = concat(subseq(c0.claves, 1, |c0.claves|, [k]) ∧  
  c.data = setKey(delKey(c0.data, c0.data[c.claves[0]]), k, v)}  
  asegura {c.cap = c0.cap}

### 10.2. B)

TAD CacheLRU[K,(V,R)] {  
  obs claves:seq⟨K⟩  
  obs data:dict[K,(V,R)]  
  obs cap :ℕ  
  
proc nuevoCache (in cap:ℕ) : CacheLRU[K,(V,R)]  
  requiere {true}  
  asegura {res.data = {} ∧ res.cap = cap ∧ |res.claves| = 0}

```

proc esta (in c:CacheLRU, in k:K) : bool
  requiere {true}
  asegura {res = true  $\iff$  k  $\in$  c.data}

proc obtener (in c:CacheLRU, in k:K) : V
  requiere {k  $\in$  c.data}
  asegura {res = c.data[k]}

proc definir (c:CacheLRU, in k:K, in v:V)
  requiere {c = c0}
  asegura {k  $\in$  c0.claves  $\longrightarrow$  (c.data = setKey(c0.data, k, (v, horaActual()))  $\wedge$  c.claves = c0.claves)}
  si k esta en el diccionario , deajo todo como esta y actualizo su valor con el tiempo en que se agrego.
  asegura {k  $\notin$  c0.claves  $\wedge$  |c0.claves| < c0.cap  $\longrightarrow$  c.data = setKey(c0.data, k, (v, horaActual()))  $\wedge$ 
  c.claves = concat(c0.claves, k)}
  si k no esta en el dic pero todavia tiene capacidad , agrego k a la lista de claves y actualizo su valor
  con el tiempo que entro.
  asegura {k  $\notin$  c0.claves  $\wedge$  |c0.claves| = cap  $\longrightarrow$  (( $\exists i : \mathbb{Z}$ ) (0  $\leq$  i < |c0.claves|  $\wedge$ 
  antiguo(c0.data, c0.claves[i])  $\wedge$  claves(c.claves, c0.claves, i, k)  $\wedge$  actualizo(c.data, c0.data, c0.claves[i], k, v))}
  si k no esta en el dic pero no tengo mas capacidad , saco la clave mas antigua del dic y la lista de
  claves , luego actualizo con la nueva clave k.

pred antiguo (c0:dict[K,(V,R)], k:K) {
  ( $\forall k' : T$ ) (k'  $\in$  c0  $\wedge$  k'  $\neq$  k  $\longrightarrow_L$  c0[k][1] < c0[k'][1])
}

pred claves (c:seq<T>, c0:seq<T>, i:Z, k:T) {
  i = |c0| - 1  $\wedge$  c = concat(subseq(c0, 0, i), [k])  $\vee_L$ 
  c = concat(concat(subseq(c0, 0, i), subseq(c0, i + 1, |c0|), [k]))
}

pred actualizo (c:dict[K,(V,R)], c0:dict[K,(V,R)], ant:K, k:K, v:V) {
  c = setKey(delKey(c0, ant), k, (v, horaActual()))
}

```

### 10.3. C)

```

TAD CacheTTL[K,(V,R)]
  obs data:dict[K,(V,R)]

proc nuevoCache () : CacheTTL[K,(V,R)]
  asegura {res.data = {}}

proc esta (in c:CacheTTL, in k:K) : bool
  requiere {true}
  asegura {res = true  $\iff$  k  $\in$  c.data}

proc obtener (in c:CacheTTL, in k:K) : (V,R)

```



```

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

proc definir (inout c:CacheTTL, in k:K , in v:V,in ttl:R)
    requiere  $\{c = c0 \wedge ttl > 0\}$ 
    asegura  $\{c.data = setKey(c0.data, k, (v, ttl))\}$ 

proc actualizar (inout c:CacheTTL)
    requiere  $\{c = c0\}$ 
    asegura  $\{(\forall k : T) (k \in c0.data \longrightarrow_L \text{if } c0.data[k][1] \leq horaActual() \text{ then } k \notin c.data \text{ else } k \in c.data \text{ fi})} \wedge (\forall k'' : T) (k'' \notin c0.data \longrightarrow_L k'' \notin c.data)\}$ 

```

## 11. Ejercicio 11

```

Coord es struct  $\langle x : \mathbb{Z}, y : \mathbb{Z} \rangle$ 
TAD Robot {
    obs pos:Coord
    obs pasajes:seq $\langle Coord \rangle$ 

proc inicia (in posx:Z, posy:Z) : Robot
    requiere  $\{true\}$ 
    asegura  $\{res.pos = \langle x : posx, y : posy \rangle \wedge res.pasajes = [\langle x : posx, y : posy \rangle]\}$ 

proc arriba (inout r:Robot)
    requiere  $\{r = r0\}$ 
    asegura  $\{r.pos.x = r0.pos.x \wedge r.pos.y = r0.pos.y + 1 \wedge r.pasajes = concat(r0.pasajes, [\langle x : r0.pos.x, y : r0.pos.y + 1 \rangle])\}$ 

proc abajo (inout r:Robot)
    requiere  $\{r = r0\}$ 
    asegura  $\{r.pos.x = r0.pos.x \wedge r.pos.y = r0.pos.y - 1 \wedge r.pasajes = concat(r0.pasajes, [\langle x : r0.pos.x, y : r0.pos.y - 1 \rangle])\}$ 

proc izquierda (inout r:Robot)
    requiere  $\{r = r0\}$ 
    asegura  $\{r.pos.x = r0.pos.x - 1 \wedge r.pos.y = r0.pos.y \wedge r.pasajes = concat(r0.pasajes, [\langle x : r0.pos.x - 1, y : r0.pos.y \rangle])\}$ 

proc derecha (inout r:Robot)
    requiere  $\{r = r0\}$ 
    asegura  $\{r.pos.x = r0.pos.x + 1 \wedge r.pos.y = r0.pos.y \wedge r.pasajes = concat(r0.pasajes, [\langle x : r0.pos.x + 1, y : r0.pos.y \rangle])\}$ 

proc masDerecha (in r:Robot) : Z

```

```

    requiere {true}
    asegura { $(\exists j : \mathbb{Z}) (0 \leq j < |r.pasajes| \wedge r.pasajes[j].x = res \wedge (\forall i : \mathbb{Z}) (0 \leq i < |r.pasajes| \wedge i \neq j \longrightarrow_L r.pasajes[i].x \leq res))$ }

proc cuantasVecesPaso (in r:Robot, in c:Coord) :  $\mathbb{Z}$ 
    requiere {true}
    asegura { $res = \sum_{i=0}^{|r.pasajes|-1} \text{if } r.pasajes[i] = c \text{ then } 1 \text{ else } 0 \text{ fi}$ }

```

## 12. Ejercicio 12

```

TAD Vivero {
    obs stock:dict[K,R]
    obs caja:R
    obs venta:dict[K,R]

proc inicio (recursos:R) : Vivero
    requiere {recursos > 0}
    asegura {res.caja = recursos  $\wedge$  res.stock = {}  $\wedge$  res.venta = {}}

proc stockear (inout viv:Vivero, in k:K, in cant:N, in precio:R) : R
    requiere {viv = viv0  $\wedge$  viv.caja > 0}
    asegura {viv.caja > 0  $\wedge$  viv.caja = viv0.caja - cant * precio}
    asegura {(k  $\in$  viv.stock  $\wedge$  viv.stock = setKey(viv0.stock, k, (viv0.stock[k] + cant)))  $\vee$ 
    (k  $\notin$  viv.stock  $\wedge$  viv.stock = setKey(viv0.stock, k, cant))}
    asegura {viv.venta = viv0.venta}
    asegura {res = viv.caja}

proc actualizar (inout viv:Vivero, in k:K, precioVenta:R) : R
    requiere {viv = viv0  $\wedge$  k  $\in$  viv.stock}
    asegura {viv.caja = viv0.caja  $\wedge$  viv.stock = viv0.stock  $\wedge$ 
    viv.venta = setKey(viv0.venta, k, precioVenta)}
    asegura {res = viv.caja}

proc vender (inout viv:Vivero, in k:K) : R
    requiere {viv = viv0  $\wedge$  k  $\in$  viv.stock  $\wedge$  k  $\in$  viv.venta  $\wedge$  viv.stock[k] > 0}
    asegura {viv.venta = viv0.venta}
    asegura {viv.caja = viv0.caja + viv0.venta[k]}
    asegura {viv.stock = setKey(viv0.stock, k, (viv0.stock[k] - 1))}
    asegura {res = viv.caja}

```