Algoritmos y Estructuras de Datos

Guía Práctica 4
Especificación de TADs
Segundo Cuatrimestre 2024

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

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.

Solución

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

`proc nuevoRacional` (in $n : \mathbb{Z}$, in $d : \mathbb{Z}$) : $Racional$ {
  `requiere` $\{d \neq 0\}$
  `asegura` $\{res.num = n \land res.den = d\}$
}

`proc sumame` (inout $r : Racional$, in $p : Racional$) {
  `requiere` $\{r = R_0\}$
  `asegura` $\{r.num = R_0.num \cdot p.den + p.num \cdot R_0.den\}$
}

`proc multiplicame` (inout $r : Racional$, in $p : Racional$) {
  `requiere` $\{r = R_0\}$
  `asegura` $\{r.num = R_0.num \cdot p.num\}$
  `asegura` $\{r.den = R_0.den \cdot p.den\}$
}
...

`proc igual` (in $r_1 : Racional$, in $r_2 : Racional$) : Bool {
  `requiere` $\{true\}$
  `asegura` $\{res = true \leftrightarrow (r_1.num \cdot r_2.den = r_2.num \cdot r_1.den)\}$
}
}

Ejercicio 2. Especifique mediante TADs los siguientes elementos geométricos:

a) `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 $(0, 0)$.

b) `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.

Solución

```
TAD Punto2D {
obs x: ℝ
obs y: ℝ
```

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

`aux distanciaEntrePuntos` $(p1X : \mathbb{R}, p1Y : \mathbb{R}, p2X : \mathbb{R}, p2Y : \mathbb{R}) \ : \ \mathbb{R}\{$
$\sqrt{(p2X - p1X)^2 + (p2Y - p1Y)^2}$
}

`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` $\{True\}$
  `asegura` $\{res = distanciaEntrePuntos(p1.x, p1.y, p2.x, p2.y)\}$
}

`proc distanciaAlOrigen` (in $p : Punto2D$) : $\mathbb{R}$ {
  `requiere` $\{True\}$
  `asegura` $\{res = distanciaEntrePuntos(p.x, p.y, 0, 0)\}$
} }

```
TAD Rectangulo2D {
obs posición: struct {x : ℝ, y : ℝ}
obs tamaño: struct {x : ℝ, y : ℝ}
```

`proc nuevoRectangulo` (in $pos : \langle \mathbb{R}, \mathbb{R} \rangle$, in $tam : \langle \mathbb{R}, \mathbb{R} \rangle$) : Rectangulo2D {
  `requiere` $\{tam.x \geq 0 \land tam.y \geq 0\}$
  `// En realidad esto podría no estar pero para evitar rectangulos invertidos lo pongo.` `asegura` $\{res.posición = pos \land res.tamaño = tam\}$
}

`proc mover` (inout $r : Rectangulo2D$, in $dxy : \langle \mathbb{R}, \mathbb{R} \rangle$) {
  `requiere` $\{r = R_0\}$
  `asegura` $\{r.posicion.x = R_0.posicion.x + dxy.x \land r.posicion.y = R_0.posicion.y + dxy.y\}$
}

`proc escalar` (inout $r : Rectangulo2D$, in $esc : \mathbb{R}$) {
  `requiere` $\{r = R_0\}$
  `asegura` $\{r.tamaño.x = R_0.tamaño.x * esc \land r.tamaño.y = R_0.tamaño.y * esc\}$
  `// Dejamos fijo el vértice de abajo a la izquierda`
}

`proc estáContenido` (in $r1 : Rectangulo2D$, in $r2 : Rectangulo2D$) : Bool {
  `requiere` $\{True\}$
  `asegura` $\{res \iff ($
  $(r1.posicion.x \geq r2.posicion.x \land r1.posicion.y \geq r2.posicion.y \land$
  $r1.tamaño.x < r2.tamaño.x \land r1.tamaño.y < r2.tamaño.y)$
  $\lor$
  $(r1.posicion.x < r2.posicion.x \land r1.posicion.y < r2.posicion.y \land$
  $r1.tamaño.x > r2.tamaño.x \land r1.tamaño.y > r2.tamaño.y)$
  $)\}$
} }

Ejercicio 3.

a) Especifique el TAD Cola$\langle T\rangle$ 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

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

   a) *nuevaPila*: que crea una pila vacía
   b) *estáVacia*: 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

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*, *estáVacía*, *encolarAdelante*, *encolarAtrás* y *desencolar*. 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

TAD Cola<T> $\{$

obs elementos : $seq < T >$

proc nuevaCola () : $Cola < T >$ $\{$
  requiere $\{True\}$
  asegura $\{|res.elementos| = 0\}$
$\}$

proc estaVacía (in $c : Cola < T >$) : Bool $\{$
  requiere $\{true\}$
  asegura $\{res \iff |res.elementos| = 0\}$
$\}$

proc encolar (inout $c : Cola < T >$, in $elem : T$) $\{$
  requiere $\{c = C_0\}$
  asegura $\{|c.elementos| = |C_0.elementos| + 1 \land_L$
  $(\forall i : \mathbb{Z})(0 \leq i < |C_0.elementos| \to_L c.elementos[i] = C_0.elementos[i]) \land c.elementos[|C_0.elementos|] = elem\}$
$\}$

proc desencolar (inout $c : Cola < T >$) : T $\{$
  requiere $\{c = C_0 \land |c.elementos| > 0\}$
  asegura $\{res = C_0.elementos[0] \land |c.elementos| = |C_0.elementos| - 1 \land_L$
  $(\forall i : \mathbb{Z})0 \leq i < |c.elementos| \to_L c.elementos[i] = C_0.elementos[i + 1]\}$
$\}$ $\}$

TAD Pila<T> $\{$

```
obs elementos : seq < T >

proc nuevaPila () : Pila < T > {
    requiere {True}
    asegura {|res.elementos| = 0}
}

proc estaVacía (in c : Pila < T >) : Bool {
    requiere {true}
    asegura {res ⟺ |res.elementos| = 0}
}

proc apilar (inout c : Pila < T >, in elem : T) {
    requiere {p = P_0}
    asegura {|p.elementos| = |P_0.elementos| + 1 ∧_L
    (∀i : ℤ)(0 ≤ i < |P_0.elementos| →_L p.elementos[i + 1] = P_0.elementos[i]) ∧ p.elementos[0] = elem}
}

proc desapilar (inout c : Pila < T >) : T {
    requiere {p = P_0 ∧ |p.elementos| > 0}
    asegura {res = P_0.elementos[0] ∧ |p.elementos| = |P_0.elementos| − 1 ∧_L
    (∀i : ℤ)0 ≤ i < |p.elementos| →_L p.elementos[i] = P_0.elementos[i + 1]}
} }

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

proc nuevaDobleCola () : DobleCola < T > {
    requiere {True}
    asegura {|res.elementos| = 0}
}

proc estaVacía (in c : DobleCola < T >, in elem : T) : 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.elementos| = |C_0.elementos| + 1 ∧_L
    (∀i : ℤ)(0 ≤ i < |C_0.elementos| →_L c.elementos[i] = C_0.elementos[i]) ∧ c.elementos[|C_0.elementos|] = elem}
}

proc encolarAdelante (inout c : DobleCola < T >, in elem : T) {
    requiere {c = C_0}
    asegura {|c.elementos| = |C_0.elementos| + 1 ∧_L
    (∀i : ℤ)(0 ≤ i < |C_0.elementos| →_L c.elementos[i + 1] = C_0.elementos[i]) ∧ c.elementos[0] = elem}
}

proc desencolar (inout c : DobleCola < T >) : T {
    requiere {c = C_0 ∧ |c.elementos| > 0}
    asegura {res = C_0.elementos[|C_0.elementos|/2] ∧ // Division entera
    |c.elementos| = |C_0.elementos| − 1 ∧_L
    (∀i : ℤ)(0 ≤ i < |c.elementos|/2 →_L c.elementos[i] = C_0.elementos[i]) ∧
    (∀i : ℤ)(|c.elementos|/2 ≤ i < |c.elementos| →_L c.elementos[i] = C_0.elementos[i + 1])}
        // Recordando que / es la división entera }  }
```

Ejercicio 4.

a) Especifique el TAD Diccionario$\langle K, V\rangle$ 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

Solución `TAD Diccionario<K, V>` {

`obs claves`: $conj < K >$

`obs valor(k: K)`: V

`proc nuevoDiccionario` () : $Diccionario < K, V >$ {

`requiere` $\{True\}$

`asegura` $\{res.claves = \{\}\}$

// No digo nada de valor porque no importa.

}

`proc definir` (inout $d : Diccionario < K, V >$, in $k : K$, in $v : V$) {

`requiere` $\{d = D_0\}$

`asegura` $\{d.claves = D_0.claves \cup k \land d.valor(k) = v \land valoresIgualesMenosUno(d, D_0, k)\}$

}

`proc obtener` (in $d : Diccionario < K, V >$, in $k : K$) : V {

`requiere` $\{k \in d.claves\}$

`asegura` $\{res = d.valor(k)\}$

}

`proc está` (in $d : Diccionario < K, V >$, in $k : K$) : Bool {

`requiere` $\{True\}$

`asegura` $\{res \iff k \in d.claves\}$

}

`proc borrar` (inout $d : Diccionario < K, V >$, in $k : K$) {

`requiere` $\{d = D_0 \land k \in d.claves\}$

`asegura` $\{d.claves = D_0.claves - \{k\} \land valoresIgualesMenosUno(d, D_0, k)\}$

}

`pred valoresIgualesMenosUno` ($d1 : Diccionario < K, V >, d2 : Diccionario < K, V >, k : K$) {

$(\forall l : K)(l \neq k \to d1.valor(l) = d2.valor(l))$

}

}

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

`TAD` $DiccionarioConHistoria < K, V >$ {

`obs clavesDefinidas`: $conj < K >$

obs clavesBorradas: $conj < K >$
obs historial(k: K): $seq < V >$

proc nuevoDiccionarioConHistoria () : $DiccionarioConHistoria < K, V >$ {
  requiere $\{True\}$
  asegura $\{res.clavesDefinidas = \{\} \land res.clavesBorradas = \{\} \land$
  $(\forall k : K)(res.historial(k) = \langle\rangle)\}$
}

proc definir (inout $d : DiccionarioConHistoria < K, V >$, in $k : K$, in $v : V$) {
  requiere $\{d = D_0\}$
  asegura $\{d.clavesDefinidas = D_0.clavesDefinidas \cup \{k\} \land d.historial(k) = D_0.historial(k) + \langle v\rangle \land$
  $historialesIgualesMenosUno(d, D_0, k) \land d.clavesBorradas = D_0.clavesBorradas - \{k\}\}$
}

// Hay distintas interpretaciones posibles
de como se deberían comportar estas operaciones

proc obtener (in $d : DiccionarioConHistoria < K, V >$, in $k : K$) : $seq < V >$ {
  requiere $\{k \in d.clavesDefinidas \lor k \in d.clavesBorradas\}$
  asegura $\{res = d.historial(k)\}$
}

proc esta (in $d : DiccionarioConHistoria < K, V >$, in $k : K$) : Bool {
  requiere $\{True\}$
  asegura $\{res \iff k \in d.clavesDefinidas\}$
}

proc borrar (inout $d : DiccionarioConHistoria < K, V >$, in $k : K$) {
  requiere $\{d = D_0 \land k \in d.clavesDefinidas\}$
  asegura $\{d.clavesDefinidas = D_0.clavesDefinidas - \{k\} \land$
  $d.clavesBorradas = D_0.clavesBorradas \cup k \land historialesIguales(d, D_0)\}$
}

pred historialesIguales ($d1 : DiccionarioConHistoria < K, V >, d2 : DiccionarioConHistoria < K, V >$) {
  $(\forall k : K)(d1.historial(k) = d2.historial(k))$
}

pred historialesIgualesMenosUno ($d1 : DiccionarioConHistoria < K, V >$,
$d2 : DiccionarioConHistoria < K, V >, k : K$) {
  $(\forall l : K)(l \neq k \to d1.historial(l) = d2.historial(l))$
}
}

Ejercicio 5. Especifique los TADs indicados a continuación pero utilizando los observadores propuestos:

a) Diccionario$\langle K, V\rangle$ observado con conjunto (de tuplas)

Solución

TAD Diccionario<K, V> {

// $_0clave,_1 valor$

obs $tuplas : conj < Tupla < K, V \gg$

proc nuevoDiccionario () : Diccionario¡K,V¿{
  requiere $\{True\}$
  asegura $\{res.tuplas = \{\}\}$
}

proc definir (inout $d : Diccionario < K, V >$, in $k : K, Inv : V$) {
  requiere $\{d = D_0\}$

```
            // Podríamos pedir que k no esté, haría que no se permita redefinir sin borrar
    primero.
```

asegura $\{todosLosMismosMenosUno(d, D_0, k) \land unaSolaConClave(d, k) \land \langle k, v\rangle \in d.tuplas\}$

}

`proc obtener` (in $d : Diccionario < K, V >, in k : K) : \text{V}$ {

requiere $\{(\exists t : Tupla < K, V >)(t_0 = k \land t \in d.tuplas)\}$

asegura $\{(\exists t : Tupla < K, V >)(t_0 = k \land t \in d.tuplas \land res = t_1)\}$

}

`proc esta` (in $d : Diccionario < K, V >$, in $k : K) : \text{Bool}$ {

requiere $\{True\}$

asegura $\{res \iff (\exists t : Tupla < K, V >)(t_0 = k \land t \in d.tuplas)\}$

}

`proc borrar` (inout $d : Diccionario < K, V >$, in $k : K$) {

requiere $\{d = D_0 \land (\exists t : Tupla < K, V >)(t_0 = k \land t \in d.tuplas)\}$

asegura $\{todosLosMismosMenosUno(d, D_0, k) \land (\forall t : Tupla < K, V >)(t_0 = k \to \neg(t \in d.tuplas))\}$

}

`pred todosLosMismosMenosUno` $(d1 : Diccionario < K, V >, d2 : Diccionario < K, V >, k : K)$ {

$(\forall t : Tupla < K, V >)(t_0 \neq k \to (t \in d1.tuplas \iff t \in d2.tuplas))$

}

```
            // NO SE PUEDE USAR ∃!
```

`pred unaSolaConClave` $(d : Diccionario < K, V >, k : K)$ {

$(\exists t : Tupla < K, V >)(t_0 = k \land t \in d.tuplas \land (\forall u : Tupla < K, V >)(u \neq t \land u_0 = k \implies \neg(u \in d.tuplas)))$

}

}

b) Conjunto$\langle T\rangle$ observado con funciones

Solución `TAD Conjunto<T>` {

obs $está(elem : T) : Bool$

```
            // No uso "pertenece"para no repetir nombre con el proc
```

`proc nuevoConjunto` $() : Conjunto < T >$ {

requiere $\{True\}$

asegura $\{noTieneNada(res)\}$

}

`proc vacío` (in $c : Conjunto < T >) : \text{Bool}$ {

requiere $\{True\}$

asegura $\{res \iff noTieneNada(c)\}$

}

`proc pertenece` (in $c : Conjunto < T >$, in $elem : T) : \text{Bool}$ {

requiere $\{True\}$

asegura $\{res \iff c.está(elem)\}$

```
}

proc  agregar (inout c : Conjunto < T >, in elem : T)  {
   requiere {c = C_0}
            // Se podría pedir que el elemento no esté, sería un poco más restrictivo.
   asegura {mismosElementosMenosUno(c, C_0, elem) ∧ c.está(elem)}
}

proc  quitar (inout c : Conjunto < T >, in elem : T)  {
   requiere {c = C_0}
            // Igual que agregar, permito borrar algo que no hay.
   asegura {mismosElementosMenosUno(c, C_0, elem) ∧ ¬c.está(elem)}
}

proc  union (in c1 : Conjunto < T >, in c2 : Conjunto < T >) : Conjunto < T >  {
   requiere {True}
   asegura {(∀elem : T)(enUnión(c1, c2, e) ⟺ res.está(elem))}
}

proc  intersección (in c1 : Conjunto < T >, in c2 : Conjunto < T >) : Conjunto < T >  {
   requiere {True}
   asegura {(∀elem : T)(enIntesección(c1, c2, e) ⟺ res.está(elem))}
}

proc  diferencia (in c1 : Conjunto < T >, in c2 : Conjunto < T >) : Conjunto < T >  {
   requiere {True}
   asegura {(∀elem : T)(enDiferencia(c1, c2, e) ⟺ res.está(elem))}
}

proc  tamaño (in c : Conjunto < T >) : ℤ  {
   requiere {True}
   asegura {esTamanoDeConjunto(c, res)}
}
pred noTieneNada (c : Conjunto < T >) {
 (∀elem : T)(¬c.está(elem))
}
pred mismosElementosMenosUno (c1 : Conjunto < T >, c2 : Conjunto < T >, elem : T) {
 (∀e : T)(e ≠ elem → (c1.está(e) ⟺ c2.está(e)))
}
pred enUnión (c1 : Conjunto < T >, c2 : Conjunto < T >, e : T) {
 c1.está(e) ∨ c2.está(e)
}
pred enIntesección (c1 : Conjunto < T >, c2 : Conjunto < T >, e : T) {
 c1.está(e) ∧ c2.está(e)
}
pred enDiferencia (c1 : Conjunto < T >, c2 : Conjunto < T >, e : T) {
 c1.está(e) ∧ ¬c2.está(e)
}
pred esTamañoDeConjunto (c : Conjunto < T >, t : Z) {
```

$(\exists d : conj < T >)((\forall e : T)((e \in d \iff c.está(e)) \land t = |c|)$
}
}

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

Solución `TAD Pila<T>` {

`obs elemsEnPosicion:` $dict < T, \mathbb{Z} >$

`proc nuevaPila` () : $Pila < T >$ {
  `requiere` $\{True\}$
  `asegura` $\{res.elemsEnPosicion = \{\}\}$
}

`proc estaVacia` (in $p : Pila < T >$) : Bool {
  `requiere` $\{True\}$
  `asegura` $\{res \iff res.elemsEnPosicion = \{\}\}$
}

`proc apilar` (inout $p : Pila < T >$, in $elem : T$) {
  `requiere` $\{p = P_0\}$
  `asegura` $\{(\exists k : \mathbb{Z})(esProximaClave(P_0, k) \land esClavedeTope(p, k) \land p.elemsEnPosicion = setKey(P_0.elemsEnPosicion, k, elem))\}$
  `// SetKey igual a SetAt`
}

`proc desapilar` (inout $p : Pila < T >$) : T {
  `requiere` $\{p = P_0 \land |p.elemsEnPosicion| > 0\}$
  `asegura` $\{res = P_0.elemsEnPosicion[0] \land |p.elemsEnPosicion| = |P_0.elemsEnPosicion| - 1 \land (\forall k : ent)(0 \geq k < p.elemsEnPosicion \rightarrow (k \in p.elemsEnPosicion[k + 1]))\}$
}

`proc tamaño` (in $p : Pila < T >$) : $\mathbb{Z}$ {
  `requiere` $\{True\}$
  `asegura` $\{res = |p.elemsEnPosicion|\}$
}

`pred esClaveDeTope` ($p : Pila < T >, k : \mathbb{Z}$) {
  $(\forall l : \mathbb{Z})((l \in p.elemsEnPosicion \land l \neq k) \rightarrow l < k)$
}

`pred esProximaClave` ($p : Pila < T >, k : \mathbb{Z}$) {
  $(\exists l : \mathbb{Z})(esClaveDeTope(l) \land k = l + 1) \lor (p.elemsEnPosicion = \land k = 0)$
}
}

d) Punto observado con coordenadas polares

Solución `TAD Punto` {

`obs` *angulo* : $\mathbb{R}$
`obs` *radio* : $\mathbb{R}$

```
        // No lo voy a escribir pero hay dos opciones:
        // Exactamente lo mismo que el Punto2D del 2a, pero siempre haciendo las
conversiones con los aux angulo(x, y) y radio(x, y). La interfaz seguiría siendo de
coordenadas cartesianas.
        // Interfaz de coordenadas polares, nos salteamos la conversión. También queda casi
igual que el 2a.
```

}

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
```

Solución `TAD Multiconjunto<T>` {

`obs  multi(e: T):` $\mathbb{Z}$

proc `nuevoMulticonjunto` () : $Multiconjunto < T >$ {
  requiere $\{True\}$
  asegura $\{(\forall e : T)(res.multi(e) = 0)\}$
}

proc vacío (in $c : Multiconjunto < T >$) : Bool {
  requiere $\{True\}$
  asegura $\{res \iff (\forall e : T)(res.multi(e) = 0)\}$
}

proc pertenece (in $c : Multiconjunto < T >$, in $elem : T$) : Bool {
  requiere $\{True\}$
  asegura $\{res \iff c.multi(elem) > 0\}$

```
    }

    proc agregar (inout c : Multiconjunto < T >, in elem : T)  {
        requiere {c = C_0}
        asegura {c.multi(elem) = C_0.multi(elem) + 1 ∧
        multisIgualesMenosUna(c, C_0, elem)}
    }

    proc quitar (inout c : Multiconjunto < T >, in elem : T)  {
        requiere {c = C_0 ∧ c.multi(elem) > 0}
                    // Restrinjo más para trabajar menos.
        asegura {c.multi(elem) = C_0.multi(elem) − 1 ∧ multisIgualesMenosUna(c, C_0, elem)}
    }

    proc union (in c1 : Multiconjunto < T >, in c2 : Multiconjunto < T >) : Multiconjunto < T >  {
        requiere {True}
        asegura {(∀e : T)(res.multi(e) = c1.multi(e) + c2.multi(e))}
    }

    proc intersección (in c1 : Multiconjunto < T >, in c2 : Multiconjunto < T >) : Multiconjunto < T >  {
        requiere {True}
        asegura {(∀e : T)(res.multi(e) = min(c1.multi(e), c2.multi(e)))}
    }

    proc diferencia (in c1 : Multiconjunto < T >, in c2 : Multiconjunto < T >) : Multiconjunto < T >  {
        requiere {True}
        asegura {(∀e : T)(res.multi(e) = max(c1.multi(e) − c2.multi(e), 0))}
    }

    proc tamaño (in c : Multiconjunto < T >) : ℤ  {
        requiere {True}
        asegura {esTamanoDeConjunto(c, res)}
    }
    pred multisIgualesMenosUna (c1 : Multiconjunto < T >, c2 : Multiconjunto < T >, e : T) {
      (∀f : T)(f ≠ e → c1.multi(f) = c2.multi(f))
    }
    pred esTamañoDeIntersección (c : Multiconjunto < T >, t : ℤ) {
      (∃s : seq < T >)(t = |s| ∧ (∀e : T)(#Apariciones(s, e) = c.multi(e)))
    }
}
```

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 `TAD` $Multidict < K, V >$ {

`// Decisiones: Se puede asociar el mismo valor a la misma clave más de una vez, quedan todas guardadas.`

`// Borrar toma un valor especifico para borrar y borra solo uno, si no está no hace nada.`

`obs` $datos : dict < K, seq < V \gg$

`proc nuevoMultidict` $() : Multidict < K, V >$ {

`requiere` $\{True\}$

`asegura` $\{res.datos = \{\}\}$

}

`proc está` (in $d : Multidict < K, V >$, in $k : K$) : Bool {

`requiere` $\{True\}$

`asegura` $\{res \iff pertenece(d, k)\}$

}

`proc obtener` (in $d : Multidict < K, V >$, in $k : K$) : $seq < V >$ {

`requiere` $\{pertenece(d, k)\}$

`asegura` $\{res = d.datos[k]\}$

}

`proc definir` (inout $d : Multidict < K, V >$, in $k : K$, in $v : V$) {

`requiere` $\{d = D_0\}$

`asegura` $\{d.datos = setKey(D_0.datos, k, \mathsf{IfThenElseFi}(k \in D_0.datos, concat(D_0.datos[k], \langle v \rangle), \langle v \rangle))\}$

}

`proc borrar` (inout $d : Multidict < K, V >$, in $k : K$, in $v : V$) {

`requiere` $\{d = D_0 \land pertenece(d, k)\}$

`asegura` $\{(\exists s : seq\langle v \rangle)(|s| = |D_0.datos[k]| - 1 \land$ `// esta comparacion no hace falta con lo de abajo`
$(\forall w : V)(w \neq v \to \#Apariciones(s, w) = \#Apariciones(D_0.datos[k], w)) \land$
$\#Apariciones(s, v) = max(\#Apariciones(D_0.datos[k], v) - 1, 0) \land d.datos = setKey(D_0.datos, k, s))\}$

}

`pred pertenece` $(d : Multidict < K, V >, k : K)$ {

$k \in d.datos \land_L |d.datos[k]| > 0$

}

}

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
`TAD Contadores` {

```
obs   eventos : dict < string, seq < Z ≫
          // 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.
          // seq de fechas, 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,
me parece razonable
```

proc crearContadores (in $eventos : seq < string >$, in $fechaActual : \mathbb{Z}$) : Contadores {
 requiere $\{True\}$
 asegura $\{(\forall e : string)(e \in eventos \iff$
 $(e \in res.eventos \wedge_L |res.eventos[e]| = 1 \wedge_L res.eventos[e][0] = fechaActual))\}$
}

```
          // fObt es la fecha que nos interesa y fAct es la actual
```

proc incrementar (inout $c : Contadores$, in $e : string$, in $fAct : \mathbb{Z}$) {
 requiere $\{c = C_0 \wedge e \in c.eventos \wedge_L fAct > last(c.eventos[e])\}$
 No se puede contar en el pasado
 asegura $\{c.eventos = setKey(C_0.eventos, e, C_0.eventos[e] + < fAct >)\}$
Recordar que + de seq es concat }

proc obtenerEnFecha (in $c : Contadores$, in $e : string$, in $fObt : Z$, in $fAct : \mathbb{Z}$) : $\mathbb{Z}$ {
 requiere $\{e \in c.eventos \wedge_L fObt < fActual\}$
 asegura $\{(\exists i : Z)(0 \leq i < |c.eventos[e]| \wedge_L fObt \leq c.eventos[e][i] \wedge$
 $(i = 0 \vee (i > 0 \wedge_L fObt > c.eventos[e][i-1])) \wedge$
 $res = max(i-1, 0)) \vee (fObt > last(c.eventos[e]) \wedge res = |c.eventos[e]| - 1)\}$
}

```
          // Lo metí acá pero ya lo dejo general como para cualquier ejercicio
```

aux last ($s : seq < T >$) : 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 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).

TAD Cache$\langle K, V\rangle$ {
 proc esta(in c: Cache$\langle K, V\rangle$, in k: K) : Bool
 proc obtener(in c: Cache$\langle K, V\rangle$, in k: K) : V
 proc definir(inout c: Cache$\langle K, V\rangle$, in k: K)
}

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.

Solución

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

obs cap: $\mathbb{Z}$// la capacidad

obs data: dict$\langle K, V\rangle$ // los datos

obs claves: $seq\langle K\rangle$ // las claves, en el orden en que los fueron agregadas

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

asegura $\{res.cap = cap\}$

asegura $\{res.data = \{\}\}$

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

}

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 $\{c = C_0\}$

// claves:

// si la clave ya estaba, no cambia

asegura $\{k \in C_0.claves \to c.claves = C_0.claves\}$

// si la clave no estaba y no se pasa de la capacidad, la agrego al final

asegura $\{k \notin C_0.data \land |C_0.claves| < cap \to c.claves = concat(C_0.claves, \langle k\rangle)\}$

// si la clave no estaba y se pasa de la capacidad, elimino la primera clave y agrego la nueva

asegura $\{k \notin c.data \land |c.claves| \geq cap \to c.claves = concat(subseq(C_0.claves, 1, |C_0.claves|), \langle k\rangle)\}$

// data:

// si la clave ya estaba, data es igual a lo que había antes con el valor v asignado a la clave k

asegura $\{k \in C_0.claves \to c.data = setKey(C_0.data, k, v)\}$

// si la clave no estaba pero no se pasa de la capacidad, data es igual a lo que había antes con el valor v asignado a la clave k

asegura $\{k \notin C_0.data \land |C_0.claves| < cap \to c.data = setKey(C_0.data, k, v)\}$

// si la clave no estaba y se pasa de la capacidad, data es igual a lo que había antes sin la primera clave y con el valor v asignado a la clave k

asegura $\{k \notin C_0.data \land |C_0.claves| \geq cap \to c.data = setKey(delKey(C_0.data, C_0.claves[0]), k, v)\}$

// cap no cambia

asegura $\{c.cap = C_0.cap\}$

}

}

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.

Solución `TAD CacheLRU<K, V>` {

```
        // Es casi lo mismo que FIFO, la unica diferencia es que hay que actualizar la
fecha en obtener.
```

proc obtener (inout $c : CacheLRU < K, V >$, in $k : K$, in $fAct : \mathbb{Z}$) : V {

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

asegura $\{res = C_0.cache[k].valor \land c.capacidad = C_0.capacidad \land c.cache = setKey(C_0.cache, k, \langle fecha : fAct, valor : v\rangle)\}$

} }

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

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

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

obs ttl: $\mathbb{Z}$

pred vencio (c: CacheTTL$\langle K, V\rangle$, k: K) {

$now() - c.data[k]_1 >= c.ttl$

}

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

asegura $\{res.data = \{\}\}$

asegura $\{res.ttl = ttl\}$

}

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

// devuelve true si la clave está en la data y no se venció

asegura $\{res = true \leftrightarrow k \in c.data \land_L \neg vencio(c, k)\}$

}

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

requiere $\{c = C_0\}$

// agrega la nueva asociación clave/valor y el tiempo actual

asegura $\{c.data = setKey(C_0.data, k, \langle v, now()\rangle)\}$

// ttl no cambia

asegura $\{c.ttl = C_0.ttl\}$

}

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

// requiere que la clave esté en la data y que no haya vencido requiere $\{k \in c.data \wedge_L \neg vencio(c,k)\}$

```
// devuelve el dato asociado a la clave
```

asegura $\{res = c.data[k]_0\}$

}

}

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. Indique observadores y precondición/postcondición para cada operación:

Coord es struct $\{x : \mathbb{Z}, y : \mathbb{Z}\}$

TAD Robot {

proc arriba(inout r: $Robot$)

proc abajo(inout r: $Robot$)

proc izquierda(inout r: $Robot$)

proc derecha(inout r: $Robot$)

proc másDerecha(in r: $Robot$) : $\mathbb{Z}$

proc cuantasVecesPaso(in r: $Robot$, in c: $Coord$) : $\mathbb{Z}$

}

Solución

TAD Robot {

obs pos: Coord

obs veces(p: Coord): $\mathbb{Z}$

proc nuevoRobotEn (in p: Coord) : Robot {

requiere $\{True\}$

asegura $\{res.pos = p \wedge res.veces(p) = 1 \wedge (\forall q : Coord)(q \neq p \rightarrow res.veces(q) = 0)\}$

}

proc arriba (inout r: Robot) {

requiere $\{r = R_0\}$

asegura $\{r.pos = posArriba(R_0) \wedge r.veces(posArriba(R_0)) = R_0.veces(posArriba(R_0)) + 1 \wedge$
$(\forall p : Coord)(p \neq posArriba(R_0) \rightarrow r.veces(posArriba(R_0)) = R_0.veces(posArriba(R_0)))\}$

}

```
// Abajo, Izquierda y Derecha son exactamente lo mismo
pero con sus respectivos auxiliares.
```

proc posiciónActual (in r: Robot) : Coord {

requiere $\{True\}$

asegura $\{res = r.pos\}$

}

proc cuantasVecesPasó (in r: Robot, in p: Coord) : $\mathbb{Z}$ {

requiere $\{True\}$

asegura $\{res = r.veces(p)\}$

}

proc másDerecha (in r: Robot) : Coord {

requiere $\{True\}$

asegura $\{r.veces(res) > 0 \wedge (\forall p : Coord)(r.veces(p) > 0 \rightarrow p.x \leq res.x)\}$

}

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{
$\langle x : r.pos.x + 1, y : r.pos.y\rangle$
}
}

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.

Solución
Especie es string
nuevoVivero(in monto: $\mathbb{Z}$): nuevoVivero
comprarPlantas(inout v: Vivero, in especie: Especie, in cantidad: $\mathbb{Z}$, in precioCompraTotal: $\mathbb{Z}$)
cambiarPrecio(inout v: Vivero, in especie: Especie, in nuevoPrecio: $\mathbb{Z}$)
venderPlanta(inout v: Vivero, in especie: Especie)
balance(in v: Vivero): $\mathbb{Z}$

b) 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

c) ¿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.

Solución Por un lado necesitaríamos tener alguna lista de precios iniciales, o suponer que todas empiezan con el mismo valor o incluso que comparten todas un mismo valor. Luego, en cada compra deberíamos actualizar ese valor sumándole el 10 %. En el TAD esto se traduce a agregar un nuevo parámetro en nuevoVivero con los precios iniciales y quitar el último parámetro de comprarPlantas. Necesitaríamos un nuevo observador que siga estos cambios en nuevoVivero y comprarPlantas. Además, comprarPlantas dejaría de especificarse a partir del precio total de la compra y tendría que manejarse con el unitario, sumando cantidad*precioUnitario a los fondos.

Solución

`TAD Vivero` {

`obs balance:` $\mathbb{Z}$
`obs stock:` $\text{dict}\langle string, \mathbb{Z}\rangle$
`obs precio:` $\text{dict}\langle string, \mathbb{Z}\rangle$

`proc nuevoVivero` (in montoInicial: $\mathbb{Z}$) : Vivero {
  `requiere` $\{montoInicial > 0\}$
  `asegura` $\{res.balance = montoInicial \land stock = \{\} \land precio = \{\}\}$
}

`proc comprarPlanta` (inout v: Vivero, in especie: string, in cantidad: $\mathbb{Z}$, in precio: $\mathbb{Z}$) {
  `requiere` $\{v = V_0\}$
  `requiere` $\{cantidad > 0 \land precio > 0 \land v.balance \geq cantidad \times precio\}$
  `asegura` $\{v.balance = V_0.balance - cantidad \times precio\}$
  `asegura` $\{especie \in V_0.stock \rightarrow_L v.stock = setAt(V_0.stock, especie, v.stock[especie] + cantidad)\}$
  `asegura` $\{especie \notin V_0.stock \rightarrow_L v.stock = setAt(V_0.stock, especie, cantidad)\}$
  `asegura` $\{v.precio = V_0.precio \land v.balance = V_0.balance\}$
}

`proc ponerPrecio` (inout v: Vivero, in especie: string, in precio: $\mathbb{Z}$) {
  `requiere` $\{v = V_0\}$
  `requiere` $\{especie \in v.stock \land_L v.stock[especie] > 0\}$
  `asegura` $\{v.precio[especie] = precio\}$
  `asegura` $\{v.stock = V_0.stock \land v.balance = V_0.balance\}$
}

`proc venderPlanta` (inout v: Vivero, in especie: string) {
  `requiere` $\{v = V_0\}$
  `requiere` $\{especie \in v.stock \land_L v.stock[especie] > 0\}$
  `asegura` $\{v.stock[especie] = setKey(V_0.stock, especie, V_0.stock[especie] - 1)\}$
  `asegura` $\{v.balance = V_0.balance + V_0.precio[especie]\}$
  `asegura` $\{v.precio = V_0.precio\}$
}

}

Solución `TAD Vivero` {

`obs` $fondos : \mathbb{Z}$
`obs` $stock(e : Especie) : \mathbb{Z}$
`obs` $precio(e : Especie) : \mathbb{Z}$

`proc nuevoVivero` (in $m : \mathbb{Z}$) : nuevoVivero {
  `requiere` $\{m > 0\}$
  `asegura` $\{res.fondos = m \land (\forall e : Especie)(res.stock(e) = 0 \land res.precio(e) = -1)\}$
}

`proc comprarPlantas` (inout $v : Vivero$, in $e : Especie$, in $c : \mathbb{Z}$, in $p : \mathbb{Z}$) {
  `requiere` $\{v = V_0 \land v.fondos \geq p \land c > 0\}$
  `asegura` $\{v.fondos = V_0.fondos - p \land v.stock(e) = V_0.stock(e) + c \land$
  $mismosPrecios(v, V_0) \land mismoStockMenosUno(v, V_0, e)\}$
}

`proc cambiarPrecio` (inout $v : Vivero$, in $e : Especie$, in $p : \mathbb{Z}$) {
  `requiere` $\{v = V_0 \land p > 0\}$
    `// Se podría pedir que haya stock de e, no me parece que haga falta`
  `asegura` $\{v.fondos = V_0.fondos \land v.precio(e) = p \land$
  $mismoStock(v, V_0) \land mismosPreciosMenosUno(v, V_0, e)\}$
}

proc venderPlanta (inout $v : Vivero$, in $e : Especie$) {
requiere $\{v = v_0 \wedge v.stock(e) > 0 \wedge v.precio(e) > 0\}$
asegura $\{v.fondos = V_0.fondos + v.precio(e) \wedge v.stock(e) = V_0.stock(e) - 1 \wedge$
$mismoStockMenosUno(v, V_0, e) \wedge mismosPrecios(v, V_0)\}$
}

proc balance (in $v : Vivero$) : $\mathbb{Z}$ {
requiere $\{True\}$
asegura $\{res = v.fondos\}$
}

pred mismosPrecios $(v1 : Vivero, v2 : Vivero)$ {
$(\forall e : Especie)(v1.precio(e) = v2.precio(e))$
}
pred mismosPreciosMenosUno $(v1 : Vivero, v2 : Vivero, e : Especie)$ {
$(\forall f : Especie)(f \neq e \rightarrow v1.precio(f) = v2.precio(f))$
}
pred mismoStock $(v1 : Vivero, v2 : Vivero)$ {
$(\forall e : Especie)(v1.stock(e) = v2.stock(e))$
}
pred mismoStockMenosUno $(v1 : Vivero, v2 : Vivero, e : Especie)$ {
$(\forall f : Especie)(f \neq e \rightarrow v1.stock(f) = v2.stock(f))$
}
}

Ejercicio 11. La masividad de la materia Algoritmos y Estructura de Datos (AED) en el primer cuatrimestre de 2024 generó que el primer parcial de la materia se tuviera que tomar en el aula más grande disponible en la Facultad (Magna del Pab2). Esta aula cuenta con 2 puertas, una a cada lado del aula. Debido a la poca previsión de los docentes (que no se previeron organizar un sistema de ingreso ordenado al aula), los y las estudiantes aguardaron el ingreso al aula en ambas puertas. Llegada la hora del inicio del parcial, para poder ordenar el ingreso, uno de los Profesores decidió dejar pasar a los estudiantes de a uno a la vez, alternando un ingreso de cada puerta.

Se desea especificar el TAD `dobleCola<T>` que modele un sistema como el descrito anteriormente, en el que existen dos colas y la salida de las mismas (desencolar) se dá de forma alternada. Para dicha especificación sólo se pueden usar tipos básicos de especificación: Z, R, bool, seq, tupla, conj, dict.

a) Elegir los observadores y especificar los procs necesarios.

b) Especificar formalmente el proc `MudarElemento`, que muda un elemento de cola, sacándolo del lugar en el que esté y encolándolo en la otra cola.

Solución
TAD DobleCola<T> {
obs $izq : seq < T >$
obs $der : seq < T >$
obs $prox : Bool$
// Para el caso del aula tendría sentido que no pudieran haber elementos (estudiantes) repetidos.
// Para que sea una cosa un poco más general no voy a tener eso en cuenta, permito repetidos.
// False es izquierda, True es derecha

proc nuevaDobleCola () : $DobleCola < T >$ {
requiere $\{True\}$
asegura $\{res.izq = \langle\rangle \wedge res.der = \langle\rangle\}$
// No digo nada para prox porque no importa como empieza. }

proc encolarIzquierda (inout $c : DobleCola < T >, in e : T$) {

requiere $\{c = C_0\}$
asegura $\{c.izq = C_0.izq + \langle e \rangle \wedge c.der = C_0.der \wedge (c.prox \iff C_0.prox)\}$
}

proc encolarDerecha (inout $c : DobleCola < T >$, in $e : T$) {
requiere $\{c = C_0\}$
asegura $\{c.izq = C_0.izq \wedge c.der = C_0.der + \langle e \rangle \wedge (c.prox \iff C_0.prox)\}$
}

proc desencolar (inout $c : DobleCola < T >$) : T {
requiere $\{c = C_0 \wedge (|c.izq| > 0 \vee |c.der| > 0)\}$
asegura $\{((C_0.prox \vee |C_0.izq| = 0) \rightarrow res = head(C_0.der) \wedge$
$c.der = tail(C_0.der) \wedge c.izq = C_0.izq \wedge \neg c.prox) \wedge$
$((\neg C_0.prox \vee |C_0.der| = 0) \rightarrow res = head(C_0.izq) \wedge c.izq = tail(C_0.izq) \wedge c.der = C_0.der \wedge c.prox)\}$
}

// Podría comportarse de distintas formas en el caso en que el mismo elemento está más de una vez

proc mudarElemento (inout $c : DobleCola < T >$, in $e : T$) {
requiere $\{c = C_0 \wedge (e \in c.izq \vee e \in c.der)\}$
asegura $\{(izqADer(c, C_0, e) \iff \neg derAIzq(c, C_0, e)) \wedge c.prox = C_0.prox\}$
// $p \iff \neg q$ es un xor, no quiero que se pasen cosas en las dos filas.

// Igual ni hace falta porque izqADer y derAIzq no pueden pasar al mismo tiempo, lo dejo para que se vea lo que se me fue ocurriendo.
}

pred esIndiceDe $(s : seq < T >, e : T, i : Z)$ {
$0 \leq i < |s| \wedge_L s[i] = e$
}

pred izqADer $(c : DobleCola < T >, C_0 : DobleCola < T >, e : T)$ {
$(\exists i : Z)(esIndiceDe(C_0.izq, e, i) \wedge_L c.der = C_0.der + < e > \wedge c.izq = C_0.izq[0..i] + C_0.izq[i + 1..|C_0.izq|])$
}

pred derAIzq $(c : DobleCola < T >, C_0 : DobleCola < T >, e : T)$ {
$(\exists i : Z)(esIndiceDe(C_0.der, e, i) \wedge_L c.izq = C_0.izq + < e > \wedge$
$c.der = C_0.der[0..i] + C_0.der[i + 1..|C_0.der|])$
}
}

Ejercicio 12. 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.

Solución
Los territorios son una lista donde s[i] es la fuerza del batallon enemigo. s[i] = 0 anexado.
nuevoJuego(in $territorios : seq < Z >$, in $dineroInicial : \mathbb{Z}$): Juego
contratarMercenario(inout $juego : Juego$, in $poder : \mathbb{Z}$, in $costo : \mathbb{Z}$, in $batallon : \mathbb{Z}$)
atacarTerritorio(inout $juego : Juego$, in $territorio : \mathbb{Z}$, in $batallon : \mathbb{Z}$)
territoriosAnexados(in $juego : Juego$): $\mathbb{Z}$
dineroDisponible(in $juego : Juego$): $\mathbb{Z}$

b) 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

Solución `TAD Juego` {
`obs` $dinero : \mathbb{Z}$
`obs` $territorios : seq < Z >$
`obs` $batallon(b : \mathbb{Z}) : \mathbb{Z}$
`// Poder de batallon b`

`proc nuevoJuego` (in $t : seq < Z >, ind : \mathbb{Z}$) : Juego {
`requiere` $\{d > 0 \land territoriosInicialesVálidos(t)\}$
`asegura` $\{res.dinero = d \land res.territorio = t \land (\forall b : Z)(res.batallon(b) = 0)\}$
}

`proc contratarMercenario` (inout $j : Juego$, in $p : \mathbb{Z}$, in $c : \mathbb{Z}$, in $b : \mathbb{Z}$) {
`requiere` $\{j = J_0 \land p > 0 \land c > 0 \land j.dinero \geq c\}$
`asegura` $\{j.dinero = J_0.dinero - c \land j.territorios = J_0.territorios \land$
$j.batallon(b) = J_0.batallon(b) + p \land batallonesIgualesMenosUno(j, J_0, b)\}$
}

`proc atacarTerritorio` (inout $j : Juego$, in $t : \mathbb{Z}$, in $b : \mathbb{Z}$) {
`requiere` $\{j = J_0 \land j.batallon(b) > 0 \land 0 \leq t < |j.territorios| \land_L j.territorios[t] > 0 \land j.dinero \geq 500\}$
`asegura` $\{ganó)(j, J_0, t, b) \lor perdió)(j, J_0, t, b)\}$
}

`proc territoriosAnexados` (in $j : Juego$) : $\mathbb{Z}$ {
`requiere` $\{True\}$
`asegura` $\{res = \sum_{i=0}^{|j.territorios|} IfThenElse(j.territorios[i] = 0, 1, 0)\}$
}

`proc dineroDisponible` (in j: Juego) : $\mathbb{Z}$ {
`requiere` $\{True\}$
`asegura` $\{res = j.dinero\}$
}
`pred territoriosInicialesVálidos` ($t : seq < Z >$) {
$|t| > 1 \land (\neg\exists i : Z)(0 \leq i < |t| \land_L t[i] = 0) \land (\forall i : Z)(0 \leq i < |t| \rightarrow_L t[i] \geq 0)$
}
`pred batallonesIguales` ($j1 : Juego, j2 : Juego$) {
$(\forall i : Z)(j1.batallon(i) = j2.batallon(i))$
}
`pred batallonesIgualesMenosUno` ($j1 : Juego, j2 : Juego, b : \mathbb{Z}$) {
$(\forall i : Z)(i \neq b \rightarrow j1.batallon(i) = j2.batallon(i))$
}
`pred ganó` ($j : Juego, J_0 : Juego, t : \mathbb{Z}, b : \mathbb{Z}$) {
$J_0.batallon(b) > J_0.territorios[t] \land j.dinero = J_0.dinero + 1000 \land$
$j.territorios = setAt(J_0.territorios, t, 0) \land batallonesIguales(j, J_0)$
}
`pred perdió` ($j : Juego, J_0 : Juego, t : \mathbb{Z}, b : \mathbb{Z}$) {

$J_0.batallon(b) \leq J_0.territorios[t] \wedge j.dinero = J_0.dinero - 500 \wedge$
$j.territorios = J_0.territorios \wedge batallonesIguales(j, J_0)$
}
}