

**Ejercicio 1.** Especificar en forma completa el TAD NumeroRacional que incluya al menos las operaciones aritméticas básicas (suma, resta, división, multiplicación)

TAD NR NUMERORACIONAL {

OBS  $m: \mathbb{Z}$

→ siempre genera nuevo

PROC (NEAR (IN  $m: \mathbb{Z}$ ): NR {  $\Rightarrow (\text{IN num: } \mathbb{Z}, \text{ IN den: } \mathbb{Z}) : \text{NR}$   
REQUIERE: {True}}

ASEGURA: {ver.  $m = m$ }

}

Q. Suma(1)

PROC SUMA (IN  $m: \text{NR}$ , IN  $m_2: \text{NR}$ ):  $\text{NR} \{$

REQUIERE: {True} )  $\rightarrow$  Se lo llaman a ver.  $m$ ?

ASEGURA: {ver =  $m + m_2$ }

}

PROC RESTA (IN  $m: \text{NR}$ , IN  $m_2: \text{NR}$ ):  $\text{NR} \{$

REQUIERE: {True}

ASEGURA: {ver =  $m - m_2$ }

}

PROC MULTIPLICACION (in m:NR, in  $m_2$ :NR) : NR {

REQUIERE: {True}

ASEGURA: { $res = m * m_2$ }

}

PROC Division(in m:NR, in  $M_2$ :NR) : NR {

REQUIERE: { $M_2 \neq 0$ }

ASEGURA: { $res = m / M_2$ }

}

}

PREGUNTA: ¿ No hay q decir q M es INVARIABLE,

O sea no se hay q garantizar q seaacional?

¿ tiene sentido q sea un OBS?

**Ejercicio 2.** Especificar TADs para las siguientes figuras geométricas. Tiene que contener las operaciones rotar, trasladar y escalar y una más propuestas por usted.

- a) Rectangulo (2D)
- b) Esfera (3D)



TAD RECTANGULO {

OBS BASE: R

OBS ALTURA: R

OBS ANGULO: R

OBS CENTRO: (R, R)

PROC CREAR (IN BASE: R, IN ALTURA: R, IN ANGULO: R, IN CENTRO: (R, R)) : RECTANGULO {

REQUIERE: {ALTURA ≥ 0 ^ BASE ≥ 0 ^ ANGULO ≥ 0}

ASEGURA: {NEW.BASE = BASE ^ NEW. ALTURA = ALTURA ^ NEW. ANGULO = ANGULO ^  
NEW. CENTRO = CENTRO}

}

PROC ROTAR (INOUT r: RECTANGULO, IN ANGULO: R) {

REQUIERE: {ANGULO ≥ 0 ^ 0 ≤ OLD(r).ANGULO + ANGULO < 2π}

ASEGURA: {r.ANGULO = OLD(r).ANGULO + ANGULO}

ASEGURA: {r.BASE = OLD(r).BASE ^ r.ALTIURA = OLD(r).ALTURA ^  
r.CENTRO = OLD(r).CENTRO}

}

PROC TRASLADAR (INOUT r: RECTANGULO, IN CENTRO: (R, R)) {

REQUIERE: {true}

ASEGURA: {r.CENTRO = CENTRO}

ASEGURA: {r.ANGULO = OLD(r).ANGULO ^ r.BASE = OLD(r).BASE ^ r.ALTIURA = OLD(r).ALTURA}

}

PROC ESCALAR(inout  $r$ :RECTANGULO, in  $Q$ : $\mathbb{R}$ ) {

REQUIERE:  $\{Q \geq 0\}$

ASEGURA:  $\{r.\text{ALTURA} = \text{OLD}(r).\text{ALTURA} * Q \wedge r.\text{BASE} = \text{OLD}(r).\text{BASE} + Q\}$

ASEGURA:  $\{r.\text{CENTRO} = \text{OLD}(r).\text{CENTRO} \wedge r.\text{ANGULO} = \text{OLD}(r).\text{ANGULO}\}$

}

PROC CENTRARENORIGEN(inout  $r$ :RECTANGULO) {

REQUIERE:  $\{\uparrow r \wedge r \neq \emptyset\}$

ASEGURA:  $\{r.\text{CENTRO} = (0,0)\}$

ASEGURA:  $\{r.\text{ALTURA} = \text{OLD}(r).\text{ALTURA} \wedge r.\text{ANGULO} = \text{OLD}(r).\text{ANGULO} \wedge r.\text{BASE} = \text{OLD}(r).\text{BASE}\}$

}

}



TAD ESFERA {

OBS CENTRO:  $(R, R, R)$

OBS RADIO:  $R$

PROC CREARESFERA(in CENTRO:  $(R, R, R)$ , in RADIO:  $\mathbb{R}$ ): ESFERA {

REQUIERE:  $\{\text{RADIO} > 0\}$

ASEGURA:  $\{\text{new.CENTRO} = \text{CENTRO} \wedge \text{new.RADIO} = \text{RADIO}\}$

}

RARI.

}

**Ejercicio 3.** Especifique el TAD DobleCola<T>, en el que los elementos pueden insertarse al principio o al final y se eliminan por el medio. Ejemplo:

```
c := DobleCola<int>.NuevaDobleCola()

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 = <> ○
```

## TAD DOBLECOLA {

OBS ELEMNS:  $\text{Alg} < \mathbb{T} >$

PROC CREARDOBLECOLA(): DOBLECOLA {

REQUIERE: {TRUE} → "Mueve lista vacía"

ASEGURA: {new\_ELEMNS = []}

}

VÁLIDO? Se lo hacen sumatoria?

AUX TAMAÑO (DC: DOBLECOLA):  $TL = |\text{DC}|$

PROC ENCOLARADELANTE (INOUT DC: DOBLECOLA, IN E:  $\mathbb{T}$ ) {

REQUIERE: {TRUE}

ASEGURA: { $|r.\text{ELEMNS}| = |\text{OLD}(r).\text{ELEMNS}| + 1$ }

ASEGURA: { $r.\text{ELEMNS}[0] = E$ } ≡  $\text{METAX}(r.\text{ELEMNS}, 0, E)$

ASEGURA: { $(\forall i: TL)(1 \leq i < |r.\text{ELEMNS}| \rightarrow r.\text{ELEMNS}[i] = \text{OLD}(r).\text{ELEMNS}[i-1])$ }

}



PROC ENCOLARATRAS (INOUT DC: DOBLECOLA, IN E:  $\mathbb{T}$ ) {

REQUIERE: {TRUE}

ASEGURA: { $|r.\text{ELEMNS}| = |\text{OLD}(r).\text{ELEMNS}| + 1$ }

FORMA UEN

ASEGURA: { $r.\text{ELEMNS}[|\text{OLD}(r).\text{ELEMNS}|] = E$ }

PENDIENTE Y ESTÁ EN ULT ÍNDICE



ASEGURA:  $\{ E \in \Gamma. \text{ELEMS} \wedge (\exists i: \mathbb{Z})(0 \leq i < |\Gamma. \text{ELEMS}| \wedge \Gamma. \text{ELEMS}[i] = E \rightarrow i = |\Gamma. \text{ELEMS}| - 1) \}$

ASEGURA:  $\{ (\forall i: \mathbb{Z})(0 \leq i < |\Gamma. \text{ELEMS}| - 1 \rightarrow \Gamma. \text{ELEMS}[i] = \text{OLD}(\Gamma). \text{ELEMS}[i]) \}$

}

PROC DESENCOLAR (INOUT DC: DOBLECOLA): T {

REQUIERE:  $\{ |\text{OLD}(DC). \text{ELEMS}| > 0 \}$

ASEGURA:  $\{ |\text{DC}. \text{ELEMS}| = |\text{OLD}(DC). \text{ELEMS}| - 1 \}$

ASEGURA:  $\{ \text{IF } (|\text{OLD}(DC). \text{ELEMS}| = 1) \text{ THEN } \text{new} = \text{OLD}(DC). \text{ELEMS}[0] \}$

ELSE  $(\exists i: \mathbb{Z})(0 \leq i < \frac{|\text{OLD}(DC). \text{ELEMS}|}{2}, i = \frac{|\text{OLD}(DC). \text{ELEMS}| - 1}{2} \wedge \text{new} = \text{OLD}(DC). \text{ELEMS}[i]) \}$

}

}

$[1, 2, 3]$  de rango 2:  $\text{PERO } 3/2 = 1.5 \Rightarrow \text{IND } 1$

$[1, 3]$  de rango 1:  $\text{PERO } 2/2 = 1 \Rightarrow \text{IND } 1 \text{ MAL, IND } 1 = 3$

$[1, 2] \Rightarrow$  si  $|\text{DC}. \text{ELEMS}|$  es PAR,  $4/2 \Rightarrow \text{IND } 2$

¿Para lograr  $0 \leq i < |n|/2$ ?

$[1, \cancel{2}, 3] \Rightarrow 0 \leq i < 2$

$[1, \cancel{2}, 3, 4] \Rightarrow 0 \leq i < 2$

$[\cancel{1}, 3] \Rightarrow 0 \leq i < 1$

$[1, 2, \cancel{3}, 4, 5] \Rightarrow 0 \leq i < 3$

$[1, 2, \cancel{3}, 4, 5, 6] \Rightarrow 0 \leq i < 3$

¿Debo usar head, tail, size?

Ejercicio 4. Especifique el TAD DiccionarioConHistoria. El mismo guarda, para cada clave, todos los valores que se asociaron con la misma a lo largo del tiempo (en orden).

¿Una lista?

Ejercicio 5. Modifique el TAD ColaDePrioridad<T> para que, si hay muchos valores iguales al máximo, la operación desapilarMax los desapile a todos.

TAD Añadir

```

TAD ColaPrioridad<T> {
    obs d: dict<T, ℝ>

    proc ColaPrioridadVacia(): ColaPrioridad<T>
        asegura {res.d = {}}

    proc vacia(in c: ColaPrioridad<T>): bool
        asegura {res = true  $\leftrightarrow$  c.d = {}}

    proc apilar(inout c: ColaPrioridad<T>, e: T, in pri: ℝ)
        requiere {c = C0}
        requiere {e  $\notin$  c.d}
        asegura {c.d = setKey(C0.d, e, pri)}

    proc desapilarMax(inout c: ColaPrioridad<T>): T
        requiere {c = C0}
        requiere {c.d  $\neq$  {}}
        asegura {c.d = delKey(C0.d, res)}
        asegura {tienePriMax(C0.d, res)}

    proc cambiarPrioridad(inout c: ColaPrioridad<T>, e: T, in priℝ)
        requiere {c = C0}
        requiere {e  $\in$  c.d}
        asegura {c.d = setKey(C0.d, e, pri)}

    pred tienePriMax(d: dict<T, ℝ>, e: T)
        {e  $\in$  d  $\wedge_L$  ( $\forall e' : T$ ) (e'  $\in$  d  $\rightarrow_L$  d[e]  $\geq$  d[e'])}
}

```

BUSQUEDA EN DICT: KEY EN DICT Y DESPUES

$(k \neq k')$  (nunca nesecario) de seguir dict[key]

**Ejercicio 6.** Especifique los TADs indicados a continuación pero utilizando los observadores propuestos:

- a) Diccionario<K,V> observado con conjunto (de tuplas)
- b) Conjunto<T> observado con funciones
- c) Pila<T> observado con diccionarios
- d) Punto observado con coordenadas polares

a)  $\{( ("A", 2), ("A", 2) \} \equiv \{ ("A", 2) \}$

$\{ ("A", 2), ("A", 3) \} \equiv \{ ("A", 2) \}$  o sea la

key no debe estar en la mues.

PARA ELIMINAR UN KEY DE LA TAREA.

ME MANDAN SIEMPRE ()

TAD DICCCIONARIO<K,V> {

OBS DATA CONJ<KxV>

OBS KEYS CONJ<K>

PROC CREARDICCIONARIO(): Diccionario<K,V> {

REQUIERE: {true}

ASEGURA: { new.DATA = {} ^ new.KEYS = {} }

}

PRED ESTA (D:DICCIONARIO<K,V>, e: KxV) {

e ∈ D.DATA

}

PRED KEYENUSO (D:DICCIONARIO<K,V>, k:K) {

k ∈ D.KEYS

}

PROC DEFINIR (INOUT D:DICCIONARIO<K,V>, IN e:KxV) {

REQUIERE: { D=D<sub>0</sub> }

REQUIERE: { !KEYENUSO(D<sub>0</sub>, e[0]) }

ASEGURA: { D.DATA = D<sub>0</sub>.DATA ∪ {e} }

ASEGURA: { D.KEYS = D<sub>0</sub>.KEYS ∪ {e[0]} }

ASEGURA: { |D.DATA| = |D<sub>0</sub>.DATA| + 1 ^ |D.KEYS| = |D<sub>0</sub>.KEYS| + 1 }

}

PROC OBTENER (IN D:DICCIONARIO<K,V>, e: KxV) :{KxV}{ } ] MAL

REQUIERE: { |D.DATA| > 0 }

ASEGURA: { e ∈ D.DATA }

UN CONJUNTO

ASEGURA:  $\{ \text{JEN} = D.\text{DATA} \cap \{e\} \}$

ME DIJE SI EL VALOR  
ESTÁ EN D.

PROC ELIMINAR(inout D: Diccionario<k,v>, in e:kxv) {

REQUIERE:  $\{ D = D_0 \}$

REQUIERE:  $\{ \text{ESTA}(D_0, e) \}$

ASEGURA:  $\{ D.\text{DATA} = D_0.\text{DATA} - \{e\} \}$

ASEGURA:  $\{ D.\text{KEYS} = D_0.\text{KEYS} - \{e[0]\} \}$

ASEGURA:  $\{ |D.\text{DATA}| = |D_0.\text{DATA}| - 1 \wedge |D.\text{KEYS}| = |D_0.\text{KEYS}| - 1 \}$

}

PROC TAMAÑO(in D: Diccionario<k,v>):  $\mathbb{N}$  {

REQUIERE:  $\{ \text{TRUE} \}$

ASEGURA:  $\{ \text{res} = |D.\text{DATA}| \}$

}

}

a) Multiconjunto<T>

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:

↳ se elimina el valor una vez  
 $\#CANTAP(VAISE, 1cr) - 1$

```
c := MultiConjunto<int>.nuevoMultiConjunto()
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
```

{1, 2, 1, 3, 4, 1}

AGG: NADA

BONAR: mi libro, Bonita LA

APARICIÓN.

TAD MULTICONJUNTO<T> {

Elijo no me limito por dup?

OBS DATA: CONJ<T> ?

OBS CANTAPARICIONES(e:T):  $\mathbb{N}$

PROC NUEVOMULTICONJUNTO(): Multiconjunto<T> {

REQUIERE: {True}

ASEGURA:  $\{ \text{res. DATA} = \{ \} \wedge (\forall e:T) (\text{RES.CANTAPARICIONES}(e) = 0) \}$

PROC AGREGAR (inout M: MULTICONJUNTO<T>, e:T) {

REQUIERE:  $\{ M = M_0 \}$

ASEGURA:  $\{ M.\text{DATA} = M_0.\text{DATA} \cup \{e\} \}$

ASEGURA:  $\{ M.\text{CANTAPARICIONES}(e) = M_0.\text{CANTAPARICIONES}(e) + 1 \}$

ASEGURA:  $\{ (\forall e':T) (e' \neq e \rightarrow M.\text{CANTAPARICIONES}(e') = M_0.\text{CANTAPARICIONES}(e')) \}$

}

PROC PERTENECIE (in M: MULTICONJUNTO<T>, in e: T): bool {

REQUIERE:  $\{ \text{true} \}$

ASEGURA:  $\{ \text{res} = \text{true} \Leftrightarrow e \in M.\text{DATA} \}$

}

PROC SACAR (inout M: MULTICONJUNTO<T>, in e:T) {

REQUIERE:  $\{ M = M_0 \wedge M_0.\text{CANTAPARICIONES}(e) > 0 \}$

ASEGURA:  $\{ M.\text{DATA} = M_0.\text{DATA} - \{e\} \}$

ASEGURA:  $\{ M.\text{CANTAPARICIONES}(e) = M_0.\text{CANTAPARICIONES}(e) - 1 \}$

ASEGURA:  $\{ (\forall e':T) (e' \neq e \wedge e' \in M_0.\text{DATA} \rightarrow M.\text{CANTAPARICIONES}(e') = M_0.\text{CANTAPARICIONES}(e')) \}$

}

PROC MULTIPLICIDAD (in n: MULTICONJUNTO<T>, in e:T): TL {

REQUIERE:  $\{ \text{true} \}$

ASEGURA:  $\{ \text{res} = n.\text{CANTAPARICIONES}(e) \}$

}

}

**Ejercicio 10.** 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 pasársela como parámetro a los procedimientos o bien puede asumir que existe una función externa *horaActual() : 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)
}
```

2

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.

DEBO GUARDAR EL ORDEN DE CADA CLAVE ALMEJADA  
QUE SE GUARDAN.

SI HAY MÁS VALORES EN ORDEN Q CAPACIDAD

BORRAR EL PRIMEROS DE ORDEN Y EN EL DIC.

Ej: { "A": 1, "B": 2 }

ORDEN: [ "A", "B" ]  $\Rightarrow$  *largo q el orden*  
*importante*

CAPACIDAD = 2

• Q AGG: { "B": 2 }  $\rightarrow$  NUEVO, NO -

$\exists$  KEY con B. ESCRIBIR KEY DICT.

• Q AGG: { "C": 3 } (new C | KEY)  $\notin$  DICT Y

(APACIDAS = |D.DICT|  $\Rightarrow$  BONITO EL NUMERO, despues

que la logitud le ocupa y el valor elementos  
más uno más en DICT mi orden.

Llego, guarda { "C": 3 }.

Del 2do en adelante (orden de agg)

Retorn igual.

|ORDEN| = |DATA|

TAN CACHE FIFO <K,V> {

OBS DATA : Dict<K,V>

OBS ORDEN : List<K>

OBS CAPACIDAD : TL

OBS esclave(e:K) : Bool

→ si yo mire la lista  
+ como hallo?

PROC NUEVO CACHE FIFO (in CAPACIDAD : TL) : CACHE FIFO <K,V> {

REQUIERE: {True}

ASEGURA: {new.DATA = {} }

ASEGURA: {new.ORDEN = [] }

ASEGURA: {new.CAPACIDAD = CAPACIDAD }

}

PROD CACHE VALIDO (C: CACHE FIFO <K,V>, ORDEN: List<K>) {

( $\forall i: TL$ ) ( $0 \leq i < |\text{ORDEN}| \rightarrow \text{ORDEN}[i] \in C.\text{DATA}$ )

}

PROC OBTENER(in C:(CACHEFIFO<K,V>, in K:K) : V {

REQUIERE:  $\{ |C.\text{DATA}| > 0 \}$

ASEGURA:  $\{ \text{res} = C.\text{DATA}[K] \}$

}

PROC ESTA(in C:(CACHEFIFO<K,V>, in K:K) : BOOL {

REQUIERE:  $\{ \text{true} \}$

ASEGURA:  $\{ \text{res} = \text{true} \Leftrightarrow K \in C.\text{DATA} \}$

}

PROC DEFINIR(inout C:(CACHEFIFO<K,V>, in K:K, in V:V) {

REQUIERE:  $\{ C = C_0 \}$

ASEGURA:  $\{ K \notin C_0.\text{DATA} \wedge |C_0.\text{DATA}| < C_0.\text{CAPACIDAD} \rightarrow_L$

$C.\text{DATA} = \text{SETKEY}(C_0.\text{DATA}, K, V) \wedge$

$C.\text{ORDEN} = \text{CONCAT}(C_0.\text{ORDEN}, [K]) \}$

ASEGURA:  $\{ K \in C_0.\text{DATA} \rightarrow_L (\text{data} = C_0.\text{DATA}) \}$  ✓

ASEGURA:  $\{ K \in C_0.\text{ORDEN} \rightarrow_L C_0.\text{ORDEN} = C.\text{ORDEN} \}$

ASEGURA:  $\{ K \notin C_0.\text{DATA} \wedge |C_0.\text{DATA}| + 1 \geq |C_0.\text{CAPACIDAD}| \rightarrow_L$

$C.\text{DATA} = \text{SETKEY}(C_0.\text{DATA}, K, V) \}$

ASEGURA:  $\{ K \notin C_0.\text{ORDEN} \wedge |C_0.\text{ORDEN}| + 1 \geq |C_0.\text{CAPACIDAD}| - 1,$

$C.\text{ORDEN} = \text{CONCAT}(\text{SUBSEQ}(C_0.\text{ORDEN}, 1, |C_0.\text{ORDEN}|),$   
 $[K]) \}$

ASEGURA:  $\{ K \in C_0.\text{ORDEN} \rightarrow C.\text{ORDEN} = C_0.\text{ORDEN} \}$

}

**Ejercicio 5.** Modifique el TAD ColaDePrioridad<T> para que, si hay muchos valores iguales al máximo, la operación desapilarMax los desapile a todos.

TAD COLAPRORIDAD<T> {  
    ↑  
    el tipo}

OBS DATA: Dict<T, R>

PROC CREARNUEVACOLAPRORIDAD(): COLAPRORIDAD<T> {

REQUIERE: {True}

ASEGURA: {new.DATA = {}}

}

PROC AÑADIR(inout C: COLAPRORIDAD<T>, in e: T, in p: R){

REQUIERE: {C<sub>0</sub> = C}

REQUIERE: {e ∉ C<sub>0</sub>.DATA}

REQUIERE: {p > 1}

ASEGURA: {C.DATA = SETKEY(C<sub>0</sub>.DATA, e, p)}

ASEGURA: {|C.DATA| = |C<sub>0</sub>.DATA| + 1} → *cien por cien!*

ASEGURA: { $\forall e' \in C$  | e' ∈ C<sub>0</sub>.DATA  $\wedge e' \neq e \rightarrow C$ .Data[e'] = C<sub>0</sub>.Data[e']}

}

PROC OBTENER(in C: COLAPRORIDAD<T>, e: T): R {

REQUIERE: {e ∈ C.DATA}

REQUIERE: { |C.DATA| > 0 }

ASEGURA: {res = C.DATA[e]}

}

PROC ESTAVACIA(in C: COLAPRORIDAD<T>): TL {

REQUIERE: {True}

ASEGURA: {nro - true  $\Leftrightarrow$  C.DATA = {}}

}

PROC DE SAPRAR MAX (inout C: COLA PRIORIDAD <T>): nro < $\tau$ > {

REQUIERE: {C = C<sub>0</sub>}

REQUIERE: {|C<sub>0</sub>.DATA| > 0}

ASEGURA: {( $\forall e:T$ ) ( $0 \leq i < |nro| \rightarrow$  C.DATA = DELKEY(C.DATA, nro[i])} ] PNEG.

ASEGURA: {SONMAX(nro)}

}

PRED SONMAX (C: COLA PRIORIDAD <T>, nro: nro < $\tau$ >) {

( $\forall i: \mathbb{N}$ ) ( $0 \leq i < |nro| \rightarrow$  esMAX(C, nro[i]))

}

PRED ESMAX (C: COLA PRIORIDAD < $\tau$ >, e: T) {

( $\forall e': T$ ) ( $e' \neq e \wedge e' \in C.DATa \rightarrow$  C.DATA[e]  $\geq$  C.DATA[e'])

}

Ejercicio 11. 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: Z, y: Z>
TAD Robot {
    proc arriba(inout r: Robot)
    proc abajo(inout r: Robot)
    proc izquierda(inout r: Robot)
    proc derecha(inout r: Robot)
    proc masDerecha(in r: Robot): Z
    proc cuantasVecesPaso(in r: Robot, in c: Coord): Z
}
```

llegó (máximo X)

Reflexión: Comida de ensalada que se mueve se lo da yo jv.

COORD: <1, 2>

TAD ROBOT{

TUPA CON  
NOMBRE EN CAMPOS

OBS COORD: STRUCT <X: ?L, Y: ?L>

OBS HISTC: DICT <COORD, ?L>

PROC CREARROBOT(): ROBOT {

REQUIERE: {True}

ASEGURA: {r0. COORD = <x:0, y:0>}

ASEGURA: {r0. HISTC = {(x:0, y:0) : 1}}

}

PROC ARRIBA(,INOUT R:ROBOT){

REQUIERE: {c=c0}

ASEGURA: {c.COORDy = c0.COORDy + 1}

ASEGURA: {c.COORDx = c0.COORDx}

ASEGURA: { $(\forall c: (x: ?L, y: ?L)) (c.x = c_0.COORDx \wedge c.y = c_0.COORDy + 1 \wedge (x: c.x, y: c.y) \notin$

$c_0.HISTC \rightarrow_L (HISTC = SETKEY(c_0.HISTC, (x: c.x, y: c.y), 1))$ }

ASEGURA: { $(\forall c: (x: ?L, y: ?L)) (c.x = c_0.COORDx \wedge c.y = c_0.COORDy + 1 \wedge (x: c.x, y: c.y) \in$

$c_0.HISTC \rightarrow_L C.HISTC = SETKEY(c_0.HISTC, (x: c.x, y: c.y), [c_0.HISTC[(x: c.x, y: c.y)] + 1])$ }

ASEGURA: { $(\forall c: (x: ?L, y: ?L)) ((c.x \neq c_0.COORDx \vee c.y \neq c_0.COORDy) \wedge$

$(x: c.x, y: c.y) \in c_0.HISTC \rightarrow_L (c_0.HISTC =$

DEMASIADO DIFÍCIL, PERO OPTIMO

TAD ROBOT{

TUPA CON  
NOMBRE EN CAMPOS

OBS COORD: STRUCT <X: ?L, Y: ?L>

OBS PASES : SEQ<COORD>

PROC CREARROBOT(): ROBOT {

REQUIERE: { True }

ASEGURA: {  $R_0.$ COORD =  $\langle x:0, y:0 \rangle$  }

ASEGURA: {  $R_0.$ PASES = [ (x:0, y:0) ] }

}

PROC ARRIBA (inout R: ROBOT) {

REQUIERE: {  $R = R_0$  }

ASEGURA: {  $R.$ COORD =  $\langle x:R_0.$ COORD $_x, y:R_0.$ COORD $_y + 1 \rangle$  }

ASEGURA: {  $R.$ PASES = CONCAT(R\_0.PASES, [ (x:R\_0.COORD\_x, y:R\_0.COORD\_y + 1) ]) }

}

PROC ABAJO (inout R: ROBOT) {

REQUIERE: {  $R = R_0$  }

ASEGURA: {  $R.$ COORD =  $\langle x:R_0.$ COORD $_x, y:R_0.$ COORD $_y - 1 \rangle$  }

ASEGURA: {  $R.$ PASES = CONCAT(R\_0.PASES, [ (x:R\_0.COORD\_x, y:R\_0.COORD\_y - 1) ]) }

}

PROC DERECHA (inout R: ROBOT) {

REQUIERE: {  $R = R_0$  }

ASEGURA: {  $R.$ COORD =  $\langle x:R_0.$ COORD $_x + 1, y:R_0.$ COORD $_y \rangle$  }

ASEGURA: {  $R.$ PASES = CONCAT(R\_0.PASES, [ (x:R\_0.COORD\_x + 1, y:R\_0.COORD\_y) ]) }

}

PROC IZQUIERDA (inout R: ROBOT) {

REQUIERE: {  $R = R_0$  }

ASEGURA: {  $R.$ COORD =  $\langle x:R_0.$ COORD $_x - 1, y:R_0.$ COORD $_y \rangle$  }

ASEGURA: {  $R.$ PASES = CONCAT(R\_0.PASES, [ (x:R\_0.COORD\_x - 1, y:R\_0.COORD\_y) ]) }

}

PROC POSICIONACTUAL(IN R:ROBOT). (X:7L,Y:7L)

REQUIERE : { true }

ASEGURA: {  $n = R.COORD$  }

}

IN

5

PROC VECESX(COORDINADA(iu R:ROBOT, C:(X:7L,Y:7L))):7L {

REQUIERE: {  $C \in R.PASES$  }

ASEGURA: {  $n = \sum_{i=0}^{IR.PASES-1} \text{IF}(R.PASES[i] = C) \text{ THEN } 1 \text{ ELSE } 0$  }

}

PROC COORDENADASALADESILLA(IN R:ROBOT):(X:7L,Y:7L) {

REQUIERE: { true }

ASEGURA: {  $(\forall c' : (X:7L, Y:7L)) (0 \leq i < |R.PASES| \wedge R.PASES[i] \neq n) \rightarrow n.x > R.PASES[i].x$  }

}

}

**Ejercicio 10.** 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 comúnmente 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 pasarlo 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)
}
```

- 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 definida más tarde (última vez utilizada). Si una clave ya no es utilizada durante un cierto período de tiempo, se considera obsoleta.

camamente la clave que fue accedida por ultima vez hace mas 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).

C) Un Cache TTL en memoria Funciona Así:

{ "Elem": "A" }

TTL: 300

Está en el horario de operación.

{ "Elem": 123456 }  $\rightarrow$  No es

Cuando se ejecuta la misma acción se malice que

los items no expiran.

Si  $MONACTUAL() - DATA["Elem"] > TTL$ : Expiró.

$$1234 - 1200 > 300 \Rightarrow F$$

$$1502 - 1200 > 300 \Rightarrow T \rightarrow \text{Expiró}$$

TAJ CACHE TTL<K,V> {

OBS DATA: Dict<K,V>

OBS FECHACNEACION: Dict<K,V>

OBS TTL: TL

PRED KEYSINSINCROIA (C: CACHE TTL<K,V>) {

( $\forall k': k' \in C.\text{FECHACNEACION} \rightarrow k' \in C.\text{DATA})$

}

PROC CREA UN NUEVO CACHE TTL <K,V> (IN TTL: /L) - (CACHE TTL <K,V>)

REQUIERE: { TTL > 0 }

ASEGURA: { RES.FECHA(NEACION) = {} }

ASEGURA: { RES.TTL = TTL }

ASEGURA: { RES.DATA = {} }

}

AGREGAR SIN AFECTAR

GUARDAR FECHA

BORRAR EXPIRADOS, DEJAR NO EXPIRADOS

PROC DEFINIR LAYOUT C: CACHE TTL <K,V>, IN K:K, IN V:V {

REQUIERE: { C = CO }

REQUIERE: { K ∈ CO.DATA ∧ K ∉ CO.FECHA(NEACION) }

REQUIERE: { KEYSINSINCRONIA(CO) }

ASEGURA: { C.DATA = SETKEY(CO.DATA, K, V) }

ASEGURA: { C.FECHA(NEACION) = SETKEY(CO.FECHA(NEACION), K, HOMACTUAL()) }

ASEGURA: { (VK':K) | (K' ∈ CO.DATA ∧ K' ≠ K ∧ HOMACTUAL - CO.FECHA(NEACION[K']) > (0.TTL) -> C.DATA = SETKEY(CO.DATA, K')) }

ASEGURA: { (VK':K) | (K' ∈ CO.DATA ∧ K' ≠ K ∧ HOMACTUAL - CO.FECHA(NEACION[K']) > (0.TTL) -> C.FECHA(NEACION) = SETKEY(CO.FECHA(NEACION), K')) }

ASEGURA: { (VK':K) | (K' ∈ CO.DATA ∧ K' ≠ K ∧ HOMACTUAL - CO.FECHA(NEACION[K']) ≤ (0.TTL) -> C.DATA = CO.DATA) }

ASEGURA: { (VK':K) | (K' ∈ CO.DATA ∧ K' ≠ K ∧ HOMACTUAL - CO.FECHA(NEACION[K']) ≤ (0.TTL) -> C.FECHA(NEACION) = CO.FECHA(NEACION) ) }

ASEGURA: { KEYSINSINCRONIA(C) }

}

PROC KEYVALIDA (C: CACHE TTL <K,V>, K:K) {

K ∈ C.DATA ∧ K ∈ C.FECHA(NEACION) ∧

| HOMACTUAL() - C.FECHA(NEACION(K)) | < C.TTL

}

PROC OBTENERL(IN C: (CACHE TTL <K,V>, IN K:K)) : V

REQUIERE: { K ∈ C.DATA ∧ K ∈ C.FECHA(NEACION) }

REQUIERE: { KEYVALIDA(c, k) }  
REQUIERE: { KEYSINSYNCHRONIA(c) }  
ASEGURA: { res = c.DATA[k] }  
}

Proc ESTA (in c: CACHE[TTL<k,v>], in k:k) : bool {  
REQUIERE: { KEYSINSYNCHRONIA(c) }  
ASEGURA: { res = true ( $\Rightarrow$  k  $\in$  c.DATA) }  
}

{

Observación: El CACHE TTL expira solo al ver un  
pase favorable, sin embargo, a la hora de obtener  
ley q voluntad si el cache q expiró o si ésto q  
no expiró.

Qui más, es necesario garantizar la sincronia  
de los keys.

