DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

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

1. Guia 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` $\{res = \text{true} \leftrightarrow r.num * n.den = r.den * n.num\}$

}

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` $\{res.x = x \land res.y = y\}$

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

`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` $\{res.centro = centro \land res.largo = l \land res.alto = a\}$

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

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

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

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

`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 DobleCola<T>, 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: $seq\langle T\rangle$

proc nuevaCola () : $Cola\langle T\rangle$

asegura $\{res.data = \langle\,\rangle\}$

proc estáVacía (in c: $Cola\langle T\rangle$) : Bool

asegura $\{res = \text{true} \leftrightarrow |c.data| = 0\}$

proc encolar (inout c: $Cola\langle T\rangle$, in e: T)

requiere $\{c = C_0\}$

asegura $\{c.data = \mathsf{concat}(C_0.data, \langle e\rangle)\}$

proc desencolar (inout c: $Cola\langle T\rangle$) : T

requiere $\{c = C_0 \wedge |c.data| > 0\}$

asegura $\{c.data = \mathsf{tail}(C_0.data) \wedge res = \mathsf{head}(C_0.data)\}$

}

TAD Pila<T> {

obs data: $seq\langle T\rangle$

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

asegura $\{res.data = \langle\,\rangle\}$

proc estáVacía (in p: $Pila\langle T\rangle$) : Bool

asegura $\{res = \text{true} \leftrightarrow |p.data| = 0\}$

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

requiere $\{p = P_0\}$

asegura $\{p.data = \mathsf{concat}(\langle e\rangle, P_0.data)\}$

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

requiere $\{p = P_0 \wedge |p.data| > 0\}$

asegura $\{p.data = \mathsf{tail}(P_0.data) \wedge res = \mathsf{head}(P_0.data)\}$

}

TAD DobleCola<T> {

obs data: $seq\langle T\rangle$

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

asegura $\{res.data = \langle\,\rangle\}$

proc estáVacía (in c: $DobleCola\langle T\rangle$) : Bool

asegura $\{res = \text{true} \leftrightarrow |c.data| = 0\}$

proc encolarAdelante (inout c: $DobleCola\langle T\rangle$, in e: T)

requiere $\{c = C_0\}$

asegura $\{c.data = \mathsf{concat}(\langle e\rangle, C_0.data)\}$

proc encolarAtrás (inout c: $DobleCola\langle T\rangle$, in e: T)

requiere $\{c = C_0\}$

asegura $\{c.data = \mathsf{concat}(C_0.data, \langle e\rangle)\}$

proc desencolar (inout c: $DobleCola\langle T\rangle$) : T
requiere $\{c = C_0 \land |c.data| > 0\}$
asegura $\{res = C_0.data[iMedio(C_0.data)] \land$
$c.data = \text{subseq}(C_0.data, 0, iMedio(C_0.data)) +\!+ \text{subseq}(C_0.data, iMedio(C_0.data) + 1, |C_0.data|)\}$

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

}

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 dell tiempo (en orden). Se debe poder hacer dicha consulta aún si la clave fue borrada.

TAD Diccionario<K, V> {
obs data: $\text{dict}\langle K, V\rangle$

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 $\{d.data = \text{setKey}(D_0.data, k, v)\}$

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

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

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

}

TAD DiccionarioConHistoria<K, V> {
obs data: $\text{dict}\langle K, seq\langle V\rangle\rangle$

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

proc definir (inout d: $DiccionarioConHistoria\langle K, V\rangle$, in k: K, in v: V)
requiere $\{d = D_0\}$
asegura {
$(k \in D_0.data \rightarrow d.data = \text{setKey}(D_0.data, k, \text{concat}(\langle v\rangle, D_0.data[k]))) \land$
$(k \notin D_0.data \rightarrow d.data = \text{setKey}(D_0.data, k, \langle v\rangle))\}$

proc obtener (in d: $DiccionarioConHistoria\langle K, V\rangle$, in k: K) : V
requiere $\{k \in d.data\}$

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

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

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:

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

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

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

d) Punto observado con coordenadas polares

TAD Diccionario<K,V> {
obs data: conj$\langle$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 $\{res = \text{true} \leftrightarrow perteneceKey(d, k)\}$

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: $Punto$) : $\mathbb{R}$
asegura $\{res = dist(p1.x, p1.y, p2.x, p2.y)\}$

proc distanciaAlOrigen (in p: $Punto$) : $\mathbb{R}$
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<T> {

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

proc sacar (inout m: $Multiconjunto\langle T \rangle$, in e: T)
requiere $\{m = M_0 \wedge e \in m.bag\}$
asegura $\{(M_0.bag[e] = 1 \rightarrow m.bag = \text{delKey}(M_0.bag, e)) \wedge$
$(M_0.bag[e] > 1 \rightarrow m.bag = \text{setKey}(M_0.bag, e, M_0[e] - 1))\}$

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: $\mathsf{dict}\langle K, seq\langle V\rangle\rangle$

proc nuevoMultidict () : $Multidict\langle K, V\rangle$

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

proc definir (inout m: $Multidict\langle K, V\rangle$, in k: K, in v: V)

requiere $\{m = M_0\}$

asegura {

$m.data = \mathsf{setKey}(M_0.data, k, \mathtt{IfThenElse}(k \in M_0.data,\ \mathsf{concat}(\langle v\rangle, M_0.data[k]),\ \langle v\rangle))$

}

proc obtener (in m: $Multidict\langle K, V\rangle$, in k: K) : V

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

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

proc pertenece (in m: $Multidict\langle K, V\rangle$, in k: K) : Bool

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

proc borrar (inout m: $Multidict\langle K, V\rangle$, in k: K) : V

requiere $\{m = M_0 \land k \in m.data\}$

asegura {

$res = M_0.data[k][0] \land$
$(|M_0.data[k]| = 1 \rightarrow m.data = \mathsf{delKey}(M_0.data, k)) \land$
$(|M_0.data[k]| > 1 \rightarrow m.data = \mathsf{setKey}(M_0.data, k, \mathsf{tail}(M_0.data[k])))$

}

}

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, $\mathbb{Z}$⟩

proc nuevoContadores (in eventos: $seq\langle str\rangle$) : $Contadores$

requiere {$sinRepetidos(eventos)$}

asegura {

($\forall e$: String) ($e \in eventos \leftrightarrow e \in res.eventos$)

($\forall e$: String) ($e \in res.eventos \rightarrow_L res.eventos[e] = 0$)

}

proc incrementar (inout c: $Contadores$, in e: String)

requiere {$e \in c.eventos \wedge c = C_0$}

asegura {$c.eventos = \text{setKey}(C_0.eventos, e, C_0.eventos[e] + 1)$}

proc cantidadDeVeces (in c: $Contadores$, in e: String) : $\mathbb{Z}$

requiere {$e \in c.eventos$}

asegura {$res = c.eventos[e]$}

pred sinRepetidos (s: $seq\langle T\rangle$) {

($\forall i, j : \mathbb{Z}$) ($0 \leq i < j < |s| \rightarrow_L s[i] \neq s[j]$)

}

}

TAD ContadoresConFecha {

HACER!

obs eventos: dict⟨String, $seq\langle\mathbb{Z}\rangle$⟩

proc nuevoContadores (in eventos: $seq\langle str\rangle$) : $ContadoresConFecha$

requiere {$sinRepetidos(eventos)$}

asegura {

($\forall e$: String) ($e \in eventos \leftrightarrow e \in res.eventos$)

($\forall e$: String) ($e \in res.eventos \rightarrow_L res.eventos[e] = \langle\rangle$)

}

proc incrementar (inout c: $ContadoresConFecha$, in e: String, in fechaActual: $\mathbb{Z}$)

requiere {$e \in c.eventos \wedge c = C_0$}

asegura {$c.eventos = \text{setKey}(C_0.eventos, e, \text{concat}(C_0.eventos[e], \langle fechaActual\rangle))$}

proc cantidadDeVeces (in c: $ContadoresConFecha$, in e: String, in fechaEvento: $\mathbb{Z}$) : $\mathbb{Z}$

requiere {$e \in c.eventos$}

asegura {}

pred sinRepetidos (s: $seq\langle T\rangle$) {

($\forall i, j : \mathbb{Z}$) ($0 \leq i < j < |s| \rightarrow_L s[i] \neq s[j]$)

}

}

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-fist-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 CacheFIFO<K, V> {
obs data: dict$\langle K, V\rangle$
obs orden: $seq\langle K\rangle$
obs max: $\mathbb{Z}$

proc nuevaCache (in max: $\mathbb{Z}$) : $CacheFIFO\langle K, V\rangle$
requiere $\{max > 0\}$
asegura $\{res.data = \{\} \land res.orden = \langle\,\rangle \land res.max = max\}$

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

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

proc definir (inout c: $CacheFIFO\langle K, V\rangle$, in k: K, in v: V)
requiere $\{c = C_0\}$
asegura {
$c.max = C_0.max$
$(k \in C_0.data \rightarrow c.data = \text{setKey}(C_0, k, v) \land mueveAlFinal(c, C_0, k, v)) \land$
$(k \notin C_0.data \rightarrow (c.data = \text{setKey}(\text{delKey}(C_0, C_0.orden[0]), k, v) \land$
$poneAlFinal(c, C_0, k) \land sacaElPrimero(c, C_0)))$
}

pred mueveAlFinal (c: $CacheFIFO\langle K, V\rangle$, C_0: $CacheFIFO\langle K, V\rangle$, k: K) {
$|c.orden| = |C_0.orden| \land_L (\exists i : \mathbb{Z})\ (0 \leq i < |C_0.orden| \land_L C_0.orden[i] = k \land (\forall j : \mathbb{Z})\ (i < j < |C_0.orden| \rightarrow_L$
}
pred poneAlFinal (c: $CacheFIFO\langle K, V\rangle$, C_0: $CacheFIFO\langle K, V\rangle$, k: K) {
$c.orden = \text{concat}(C_0.orden, \langle k\rangle)$
}
pred sacaElPrimero (c: $CacheFIFO\langle K, V\rangle$, C_0: $CacheFIFO\langle K, V\rangle$) {
$c = \text{subseq}(C_0, 1, |C_0|)$
}
}

TAD CacheLRU<K, V> {
HACER!
}

TAD CacheTTL<K, V> {
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.

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

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.

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

`TAD Vivero` {

obs plantas: dict⟨String, struct⟨$precio : \mathbb{R}, stock : \mathbb{Z}$⟩⟩

obs dinero: $\mathbb{R}$

`proc nuevoVivero` (in monto: $\mathbb{R}$) : $Vivero$

`requiere` $\{monto \geq 0\}$

`asegura` $\{res.plantas = \{\} \land res.dinero = monto\}$

`proc comprar` (inout v: $Vivero$, in especie: String, in cantidad: $\mathbb{Z}$, in precios: dict⟨String, $\mathbb{R}$⟩)

`requiere` $\{especie \in precios \land_L v.dinero \geq precios[especie] \land v = V_0\}$

`asegura` {

$v.dinero = V_0.dinero - precios[especie] \land$

$(hayEspecie(V_0, especie) \rightarrow$

$v.plantas = actualizarEspecie(V_0, especie, precio(V_0, especie), stock(V_0, especie) + cantidad)) \land$

$(\neg hayEspecie(V_0, especie) \rightarrow v.plantas = actualizarEspecie(V_0, especie, -1, cantidad))$

}

`proc ponerPrecio` (inout v: $Vivero$, in especie: String, in precio: $\mathbb{R}$)

`requiere` $\{hayEspecie(v, especie) \land v = V_0 \land precio \geq 0\}$

`asegura` {

$v.dinero = V_0.dinero \land$

$v.plantas = actualizarEspecie(V_0, especie, precio, stock(V_0, especie))$

}

`proc vender` (inout v: $Vivero$, in especie: String)

`requiere` $\{hayEspecie(v, especie) \land_L stock(v, especie) > 0 \land precio(v, especie) \neq -1 \land v = V_0\}$

`asegura` {

$v.dinero = V_0.dinero + precio(V_0, especie) \land$

$(stock(V_0, especie) = 1 \rightarrow v.plantas = eliminarEspecie(V_0, especie)) \land$

$(stock(V_0, especie) > 1 \rightarrow$

$v.plantas = actualizarEspecie(V_0, especie, precio(V_0, especie), stock(V_0, especie) - 1))$

}

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

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

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

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

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

Dinero es $\mathbb{R}$
Territorio es String
Poder es $\mathbb{Z}$
Mercenario es struct⟨*nombre* : String, *poder* : *Poder*⟩
Taberna es dict⟨String, struct⟨*mercenario* : *Mercenario*, *precio* : *Dinero*⟩⟩
Batallon es conj⟨String⟩

(DUDO que este bien hecho el TAD)

TAD Jugador {

obs nombre: String

obs territorios: conj$\langle Territorio \rangle$

obs mercenarios: conj$\langle Mercenario \rangle$

obs dinero: $Dinero$

obs batallones: dict$\langle$String, $Batallon\rangle$

proc nuevoJugador (in n: String) : $Jugador$

asegura $\{res.nombre = n\}$

asegura $\{|res.territorios| = 1 \wedge_L (\forall j : Jugador)\ (j.nombre \neq res.nombre \rightarrow_L res.territorios[0] \notin j.territorios)\}$

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

asegura $\{res.dinero = 10000\}$

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

proc contratarMercenario (inout j: $Jugador$, in n: String, in t: Taberna)

requiere $\{tieneMercenariosValidos(t)\}$

requiere $\{n \in t \wedge_L j.dinero \geq t[n].precio\}$

requiere $\{j = J_0\}$

asegura $\{j.nombre = J_0.nombre\}$

asegura $\{j.territorios = J_0.territorios\}$

asegura $\{j.batallones = J_0.batallones\}$

asegura $\{j.dinero = J_0.dinero - t[n].precio\}$

asegura $\{j.mercenarios = J_0.mercenarios \cup t[n].mercenario\}$

pred tieneMercenariosValidos (t: Taberna) {

$(\forall n : \text{String})\ (n \in t \rightarrow_L t[n].mercenario.nombre = n \wedge t[n].mercenario.poder > 0 \wedge t[n].precio > 0)$

}

proc formarBatallon (inout j: $Jugador$, in n: String, in ln: conj$\langle$String$\rangle$)

requiere $\{n \notin j.batallones \wedge tengaMercenarios(j, ln) \wedge mercenariosDisponibles(j, ln)\}$

requiere $\{j = J_0\}$

asegura $\{j.nombre = J_0.nombre\}$

asegura $\{j.territorios = J_0.territorios\}$

asegura $\{j.mercenarios = J_0.mercenarios\}$

asegura $\{j.dinero = J_0.dinero\}$

asegura $\{j.batallones = \text{setKey}(J_0.batallones, n, ln)\}$

pred tengaMercenarios (j: $Jugador$, ln: conj$\langle$String$\rangle$) {

$(\forall s : \text{String})\ (s \in ln \rightarrow (\exists m : Mercenario)\ (m \in j.mercenarios \wedge m.nombre = s))$

}

pred mercenariosDisponibles (j: $Jugador$, ln: conj$\langle$String$\rangle$) {

$(\forall s : \text{String})\ (s \in ln \rightarrow \neg(\exists b : \text{String})\ (b \in j.batallones \wedge_L s \in j.batallones[b]))$

}

proc atacar (inout j: $Jugador$, in t: Territorio, in b: Batallon, in pE: Poder) : Tipo de res
requiere $\{j.dinero \geq 500 \land t \notin j.territorios \land b \in j.batallones\}$
requiere $\{j = J_0\}$
asegura $\{j.nombre = J_0.nombre\}$
asegura $\{j.mercenarios = J_0.mercenarios\}$
asegura $\{j.batallones = J_0.batallones\}$
asegura {
$(tieneBatallonMasFuerte(j, b, pE) \rightarrow j.dinero = J_0.dinero + 1000 \land j.territorios = J_0.territorios \cup t)$
$\land (\neg tieneBatallonMasFuerte(j, b, pE) \rightarrow j.dinero = J_0.dinero - 500)$
}
pred tieneBatallonMasFuerte (j: $Jugador$, b: Batallon, pE: Poder) {
$(\exists s : seq\langle Poder\rangle)$ (
$|s| = |b| \land (\forall m : Mercenario)\ ((m \in j.mercenarios \land m.nombre \in b) \rightarrow m.poder \in s) \land$
$pE < \sum_{i=0}^{|s|-1} s[i]$
)
}

proc cantidadTerritorios (in j: $Jugador$) : $\mathbb{Z}$
asegura $\{res = |j.territorios|\}$

proc dineroDisponible (in j: $Jugador$) : Dinero
asegura $\{res = j.dinero\}$

}