

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

dos num: \mathbb{Z}

dos den: \mathbb{Z}

recordemos qué es un
número racional: una fracción
 $\frac{a}{b} \rightarrow$ numerador
denominador ↙ b

proc nuevoRacional (in N: \mathbb{Z} , in D: \mathbb{Z}): Racional {

requiere { $D \neq 0$ }

asegura { res.num = N \wedge res.den = D }

}

podemos hacer que devuelva uno nuevo o
hacer la suma sobre uno de los dos ↗

proc Suma (in R1: Racional, in R2: Racional): Racional {

requiere { True }

asegura { (res.num = R1.num * R2.den + R2.num * R1.den
 \wedge res.den = R1.den * R2.den) }

}

Hagámoslo ahora de la otra forma...

proc resta (inout A: Racional, in B: Racional) {

requiere { A = A0 }

asegura { (A.num = A0.num * B.den - B.num * A0.den) \wedge
(A.den = A0.den * B.den) }

}

proc division (inout A: rational, in B: rational) {
 requiere $\{(A = A_0) \wedge (B \cdot \text{num} \neq 0)\}$
 asegura $\{(A \cdot \text{num} = A_0 \cdot \text{num} * B \cdot \text{den}) \wedge$
 $(A \cdot \text{den} = A_0 \cdot \text{den} * B \cdot \text{num})\}$
}

proc multi (inout A: rational, in B: rational) {
 requiere $\{A = A_0\}$
 asegura $\{(A \cdot \text{num} = A_0 \cdot \text{num} * B \cdot \text{num}) \wedge$
 $(A \cdot \text{den} = A_0 \cdot \text{den} * B \cdot \text{den})\}$
}

Segun lo charlado en clase, veremos dos maneras de especificar el proc igual. Pero primero pensemos casos en que la igualdad deberia cumplirse:

$$\frac{1}{5} = \frac{1}{5} \quad \frac{1}{2} = \frac{2}{4} \quad \frac{-8}{24} = \frac{-1}{3} \quad \frac{3}{3} = \frac{5}{5}$$

$$\frac{0}{7} = \frac{0}{13}$$

proc igual (in A: rational, in B: rational): Bool {
 requiere $\{\text{True}\}$
 asegura $\{\text{res} = \text{true} \leftrightarrow ((\exists k : \mathbb{Z})(((A \cdot \text{num} * k = B \cdot \text{num}) \wedge (A \cdot \text{den} * k = B \cdot \text{den})) \vee ((A \cdot \text{num} = B \cdot \text{num} * k) \wedge (A \cdot \text{den} = B \cdot \text{den} * k))) \vee ((\exists p, j : \mathbb{Z})(A \cdot \text{num} * p = B \cdot \text{num} * j) \wedge (A \cdot \text{den} * p = B \cdot \text{den} * j)))\}$
}

Esta solucion efectivamente cubre todas las opciones, pero es algo tediosa veamos otra forma...

proc igual (in A : racional , in B: racional): Bool {

 requiere { True }

 asegura { res = true \leftrightarrow (A . num * B . den =
 A . den * B . num) }

} funciones ya que $\frac{a}{b} = \frac{c}{d} \Rightarrow a \cdot d = b \cdot c$

para concluir, pensemos un poco nuestra elección de los observadores. Por qué no usamos uno solo que represente un número racional y ya?

Bueno, notemos como tanto las operaciones, como las condiciones que tuvimos que estipular para que un número racional sea válido (ej: que el denominador no sea 0) fueron mucho más simples pensando al número racional como 2 enteros, numerador y denominador. De haberlo hecho con un único observador se nos hubiera dificultado más ya que en especificación no tenemos el conjunto \mathbb{Q} (racionales) por lo que deberíamos haberlo definido a partir de \mathbb{IR} y termina siendo muy complejo. Por eso la importancia de elegir bien nuestros observadores! ★

Ejercicio 4.

a) 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) *está*: que devuelve true si la clave está en el diccionario
- e) *borrar*: que elimina una clave del diccionario

¿Qué es lo que tenemos que tener en cuenta para especificar un diccionario?

comencemos por pensar las características de un diccionario:

- tienen claves únicas (k) con valores asociados (v)
- son mutables, se pueden agregar, eliminar y modificar elementos
- los métodos serán: *nuevoDiccionario*, *definir*, *obtener*, *está* y *borrar*

ojo! en los diccionarios definir contempla agregar y modificar, a esto se lo llama: *setKey*.

en este caso, ¿cuáles se les ocurre que podrían ser los observadores?

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

obs data: Dict $\langle K, V \rangle$

proc nuevoDiccionario (): Diccionario $\langle K, V \rangle$ {

requiere { True }

asegura { res.data = {} }

}

como deberían ser los parámetros de entrada para definir?
sabemos que deberán venir una key y un valor
a asociar, y el diccionario? Un inout! ya que
definir no debería devolver nada.

proc definir (inout D: Diccionario $\langle K, V \rangle$, in K: K, in V: V) {

requiere { D = Do }

asegura { D.data = SetKey(Do.data, K, V) }

}

proc obtener (in D: Diccionario $\langle K, V \rangle$, in K: K): V {

requiere { pertenece(D, K) } → es comodo tener

asegura { res = D.data[K] } una auxiliar no?

podríamos usar ésta?

No! proc dentro de otro no se puede
(pero si podemos un Pred dentro de un Proc)

proc esta (in D: Diccionario $\langle K, V \rangle$, in K: K): Bool {

requiere { True }

asegura { res = true \leftrightarrow pertenece(D, K) }

}

```
proc eliminar (inout D:Diccionario<K,V>, in k:k) {  
    requiere { D = D0 }  
    asegura { D . data = delkey ( D0 . data , k ) }  
}
```

```
pred pertenece ( in D:Diccionario, in k:k ) {  
    k ∈ D . data  
}
```

y listo! Notemos lo cómodo de elegir un buen tipo para nuestro observador :)

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 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 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< $K, V$ > {
    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:  $K$ ) ← falta un in v:  $V$ 
}
```

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.

pensemos en las cosas a considerar...

1. esta será una especificación relativamente sencilla y funcionará para determinar si es un hit or miss
2. obtener tiene que tener en cuenta que esté en el cache para devolver el valor
3. definir tiene que considerar el espacio del cache (no excederlo)

primero pensemos los observadores, nos dicen que tiene una interface basada en un diccionario, entonces sabemos que uno será dato de tipo dict. ¿Qué más? pensemos en el tema del espacio y el método para agregar nuevos elementos si la capacidad está al tope (FIFO). Un observador podría ser capacidad, tal que mantengamos su cambio de estado y una secuencia que determine el orden en el que se agregaron las cosas.

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

obs data: Dict $\langle K, V \rangle$ —————> →

obs cap : \mathbb{N}

obs ordenK: Seq $\langle \mathbb{N} \rangle$

recordemos que

al utilizar dict

podremos hacer uso
de setkey y del key

qui es lo que definimos
en nuestro

TAD diccionario
como definir y
eliminar

proc nuevoCache (c: \mathbb{N}): Cache $\langle K, V \rangle$ {

requiere {True}

asegura {res.data = c}

asegura {res.cap = 0}

asegura {res.ordenK = <>}

}

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

requiere {True}

asegura {res = true \leftrightarrow pertenece (c, k)}

}

en este caso, asumimos que al obtener el valor
de un elemento el orden no cambia.

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

requiere {pertenece (c, k)}

asegura {res = c.data [k]}

}

para pensar: ¿que habria que cambiar para que se
pueda especificar que al obtener un valor cambie el
orden?

Ahora, podríamos pensar que casos hay que considerar:

1. Ya existe la clave en el cache
2. Nueva clave y no se pasa la capacidad
3. Nueva clave y sí se pasa la capacidad

Ojo! recordemos que tenemos 2 estructuras, datos y ordenk, por lo que vamos a tener que modificar ambas:

proc definir (inout c: Cache<K,V>, in k: K, in v: V) {
requiere { c = co }

CASO 1:

asegura { pertenece (c, k) \rightarrow (c.data = setKey(co.data, k, v) \wedge ordenActualizado(c, k)) }

CASO 2:

asegura { (\neg pertenece (c, k) \wedge |co.ordenk| < co.cap) \rightarrow
(c.data = setKey(co.data, k, v) \wedge
c.ordenk = concat(co.ordenk, <k>)) }

CASO 3:

asegura { (\neg pertenece (c, k) \wedge |co.ordenk| \geq co.cap) \rightarrow
(c.data = setKey(delKey(co.data,
co.ordenk[|co.ordenk|-1]), k, v) \wedge
c.ordenk = concat(subseq(co.ordenk, 1, |co.ordenk|-1), <k>)) }

asegura { c.cap = co.cap } \rightarrow importante para
los inout no
olvidarse de
especificar el estado de
todos los observadores

}

```

pred pertenece (in c:Cache<k,v>, in k: k) {
    { (k ∈ c.data ∧ (∃ i: ℤ) (0 ≤ i < |c.ordenK| →
    { c.ordenK[i] = k)) )
}

```

Si bien es una única línea, al ser constantemente usado es común tenerlo como pred

```

pred ordenActualizado ( inout c: Cache<k,v>, in k:k)
{ (forall i,j: ℤ) (0 ≤ i,j < |c.ordenK| ∧ i < j ∧ c.ordenK[i] ≠ k ∧
c.ordenK[j] ≠ k → (exists p,q: ℤ) (0 ≤ p,q < |c.ordenK|-1 ∧
p < q ∧ c.ordenK[i] = c.ordenK[p] ∧
c.ordenK[j] = c.ordenK[q])) ∧ c.ordenK[|c.ordenK|-1] = k
}

```

ojo! Esta es una forma generalizable que no funciona con elementos repetidos (en este caso no hay problema porque son keys pero tengan cuidado al reutilizarlo)

ahora sí! terminamos ☺

y recuerden: no hay una única forma de especificar TADs!