Algoritmos y Estructuras de Datos

Guía Práctica 4
Especificación de TADs

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

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

```
TAD Racional {
      obs num: ℤ
      obs den: ℤ

      proc nuevoRacional(in n: ℤ, in d: ℤ): Racional
            requiere {d ≠ 0}
            asegura {res.num = n ∧ res.den = d}

      proc sumame(inout r: Racional, in p: Racional)
            requiere {r = R0}
            asegura {r.num = R0.num · p.den + p.num · R0.den}
            asegura {r.den = R0.den · p.den}

      proc multiplicame(inout r: Racional, in p: Racional)
            requiere {r = R0}
            asegura {r.num = R0.num · p.num}
            asegura {r.den = R0.den · p.den}
...

      pred igualdad(r1: Racional, r2: Racional)
         {r1.num · r2.den = r2.num · r1.den}
}
```

Ejercicio 2. Especifique los TADs indicados a continuación pero utilizando los observadores propuestos:

a) `Diccionario<K,V>` observado con funciones

```
TAD Diccionario<K,V> {
      obs esClave(k: K): bool
      obs valor(k: K): V

      proc nuevoDiccionario(): Diccionario<K,V>
            asegura {(∀k : K)(res.esClave(k) = false)}

      proc esta(in d: Diccionario<K,V>, in k: K): bool
            asegura {res = true ↔ d.esClave(k)}

      proc obtener(in d: Diccionario<K,V>, in k: K): V
            requiere {d.esClave(k)}
            asegura {res = d.valor(k)}

      proc definir(inout d: Diccionario<K,V>, in k: K, in v: V)
            requiere {d = D0}
            // la clave k está y tiene el valor v
            asegura {d.esClave(k) = true ∧L d.valor(k) = v}
            // los que estaban (que no son k) siguen estando y tienen el mismo valor
```

```
        asegura {(∀k' : K)(D0.esClave(k') = true ∧ k' ≠ k → d.esClave(k') = true ∧L d.valor(k') =
D0.valor(k'))}
        // todo lo que está y no es k, estaba
        asegura {(∀k' : K)(d.esClave(k') = true ∧ k' ≠ k → D0.esClave(k') = true)}
...

    pred igualdad(d1: Diccionario<K,V>, d2: Diccionario<K,V>)
        {(∀k : K)(d1.esClave(k) = d2.esClave(k) ∧ d1.esClave(k) →L d1.valor(k) = d2.valor(k))}
}
```

Ejercicio 3. 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<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, 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.

```
TAD CacheFIFO<K,V> {
    obs cap: int // la capacidad
    obs data: dict<K,V>// los datos
    obs claves: seq<K>// las claves, en el orden en que los fueron agregadas

    proc nuevoCache(in cap: ℤ): CacheFIFO<K,V>
        asegura {res.cap = cap}
        asegura {res.data = {}}
        asegura {res.claves = ⟨⟩}

    proc esta(in c: CacheFIFO<K,V>, in k: K): bool
        asegura {res = true ↔ k ∈ c.data}

    proc obtener(in c: CacheFIFO<K,V>, in k: K): V
        requiere {k ∈ c.data}
        asegura {res = c.data[k]}

    proc definir(inout c: CacheFIFO<K,V>, in k: K, in v: V)
        requiere {c = C0}
        // claves:
        // si la clave ya estaba, no cambia
```

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

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

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

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

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

// data:

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

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

// 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 $\{k \notin C_0.data \wedge |C_0.claves| < cap \rightarrow c.data = setKey(C_0.data, k, v)\}$

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

// cap no cambia

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

}