Algoritmos y Estructuras de Datos

Guía Práctica 8
Rep/Abs y Elección de Estructuras
Segundo Cuatrimestre 2024

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

Invariante de representación y función de abstracción en modelado de problemas

Ejercicio 1. Tenemos un TAD que modela las ventas minoristas de un comercio. Cada venta es individual (una unidad de un producto) y se quieren registrar todas las ventas. El TAD tiene un único observador:

```
Producto es string Monto es ℤ Fecha es ℤ (segundos desde 1/1/1970)
TAD Comercio {
obs ventasPorProducto: dict⟨Producto, seq⟨⟨Fecha, Monto⟩⟩⟩
}
```

ventasPorProducto contiene, para cada producto, una secuencia con todas las ventas que se hicieron de ese producto. Para cada venta, se registra la fecha y el precio. Se puede considerar que todas las fechas son diferentes. Este TAD lo vamos a implementar con la siguiente estructura:

```
Módulo ComercioImpl implementa Comercio <
      var ventas: SecuenciaImpl<tupla<Producto, Fecha, Monto>>
      var totalPorProducto: DiccionarioImpl<Producto, Monto>
      var ultimoPrecio: DiccionarioImpl<Producto, Monto>
>
```

- **ventas** es una implementación de secuencia con todas las ventas realizadas, indicando producto, fecha y monto.
- **totalPorProducto** asocia cada producto con el dinero total obtenido por todas sus ventas.
- **ultimoPrecio** asocia cada producto con el monto de su última venta registrada.

Se pide:

- Escribir en forma coloquial y detallada el invariante de representación y la función de abstracción.
- Escribir ambos en el lenguaje de especificación.

Ejercicio 2. Considere la siguiente especificación de una relación uno/muchos entre alarmas y sensores de una planta industrial: un sensor puede estar asociado a muchas alarmas, y una alarma puede tener muchos sensores asociados.

```
TAD Planta {
obs alarmas: conj⟨Alarma⟩
obs sensores: conj⟨⟨Sensor, Alarma⟩⟩

proc nuevaPlanta () : Planta {
   asegura {res.alarmas = {}}
   asegura {res.sensores = {}}
}

proc agregarAlarma (inout p : Planta, in a : Alarma) {
   requiere {p = P_0}
   requiere {a ∉ p.alarmas}
   asegura {p.alarmas = P_0.alarmas ∪ {a}}
   asegura {p.sensores = P_0.sensores}
}
```

```
    proc agregarSensor (inout p : Planta, in a : Alarma, in s : Sensor)  {
        requiere {p = P_0}
        requiere {ainp.alarmas}
        requiere {⟨s, a⟩ ∉ p.sensores}
        asegura {p.alarmas = P_0.alarmas}
        asegura {p.sensores = P_0.sensores + {⟨s, a⟩}}
    }
}
```

Se decidió utilizar la siguiente estructura como representación, que permite consultar fácilmente tanto en una dirección (sensores de una alarma) como en la contraria (alarmas de un sensor).

```
Módulo PlantaImpl implementa Planta <
      var alarmas: Diccionario<Alarma, Conjunto<Sensor>>
      var sensores: Diccionario<Sensor, Conjunto>Alarma>>
>
```

Se pide:

- Escribir formalmente y en castellano el invariante de representación.
- Escribir la función de abstracción.

Ejercicio 3. Dado el siguiente TAD:

```
Estudiante ES int
TAD Secundario {
obs estudiantes: conj⟨Estudiante⟩
obs faltas: dict⟨Estudiante,ℤ,⟩
obs notas: dict⟨Estudiante, seq⟨ℤ⟩,⟩

proc NuevoSecundario (in es: conj⟨Estudiante⟩) : Secundario  {
    requiere {|es| > 0}
    asegura {res.estudiantes = es}
    asegura {(∀e : Estudiante)(e ∈ es →_L e ∈ res.faltas ∧_L res.faltas[e] = 0)}
    asegura {(∀e : Estudiante)(e ∈ es → e ∈ res.notas ∧_L res.notas[e] = ⟨⟩)}
}

proc RegistrarNota (inout s: Secundario, in e: Estudiante, in nota: ℤ)  {
    requiere {s = S_0}
    requiere {e ∈ s.estudiantes}
    requiere {0 ≤ nota ≤ 10}
    asegura {s.estudiantes = S_0.estudiantes}
    asegura {s.faltas = S_0.faltas}
    asegura {s.notas = setKey(S_0.notas, e, S_0.notas[e] + [nota])}
}

proc RegistrarFalta (inout s: Secundario, in e: Estudiante)  {
    requiere {s = S_0}
    requiere {e ∈ s.estudiantes}
    asegura {s.alumnos = S_0.alumnos}
    asegura {s.faltas = setKey(S_0.faltas, e, old(s).faltas[e] + 1)}
    asegura {s.notas = S_0.notas}
}
}
```

Se propone la siguiente estructura de representación:

```
Módulo SecundarioImpl implementa Secundario <
      var estudiantes: Conjunto<Estudiante>
      var faltas: Diccionario<Estudiante, int>
      var notas: Array<Conj<Estudiante>>
      var notasPorEstudiante: Diccionario<Estudiante, Array<int>>
>
```

Donde:

- En *estudiantes* están todos los estudiantes del colegio secundario
- En *faltas* tenemos para estudiante la cantidad de faltas que tiene hasta el momento
- En *notas* tenemos en la posición i-ésima a los estudiantes que tienen nota i
- En *notasPorEstudiante* tenemos para cada estudiante la cantidad de notas con valor i-ésimo tienen

Se pide:

- Escribir en castellano y formalmente el invariante de representación
- Escribir la función de abstracción

Elección de Estructuras

En esta sección nos concentraremos en la elección de estructuras y los algoritmos asociados. Cuando el enunciado diga "diseñe este módulo", salvo que indique lo contrario, sólo nos interesa:

1. La estructura
2. Su invariante de representación y su función de abstracción
3. Las operaciones con sus parámetros y su complejidad
4. Los algoritmos correspondientes

Ejercicio 4. Se desea diseñar un sistema para registrar las notas de los alumnos en una facultad. Al igual que en Exactas, los alumnos se identifican con un número de LU. A su vez, las materias tienen un nombre, y puede haber una cantidad no acotada de materias. En cada materia, las notas están entre 0 y 10, y se aprueban si la nota es mayor o igual a 7.

```
TAD Sistema {
obs notas: dict⟨materia, dict⟨alumno,ℤ⟩⟩
proc nuevoSistema() : Sistema
proc registrarMateria(inout s : Sistema, in m : materia)
proc registrarNota(inout s : Sistema, in m : materia, in a : alumno, in n : nota)
proc notaDeAlumno(in s : Sistema, in a : alumno, m : materia) : nota
proc cantAlumnosConNota(in s : Sistema, in m : materia, n : nota) : ℤ
proc cantAlumnosAprobados(in s : Sistema, in m : materia) : ℤ
}
```

Dados $m = cantmaterias$ y $n = cantalumnos$ se desea diseñar un módulo con los siguientes requerimientos de complejidad temporal:

- nuevoSistema O(1)
- registrarMateria en O(log m)
- registrarNota en O(log n + log m)
- notaDeAlumno en O(log n + log m)
- cantAlumnosConNota y CantAlumnosAprobados en O(log m)

Ejercicio 5. El TAD *Matriz infinita de booleanos* tiene las siguientes operaciones:

- *Crear*, que crea una matriz donde todos los valores son falsos.
- *Asignar*, que toma una matriz, dos naturales (fila y columna) y un booleano, y asigna a este último en esa coordenada. (Como la matriz es infinita, no hay restricciones sobre la magnitud de fila y columna.)
- *Ver*, que dadas una matriz, una fila y una columna devuelve el valor de esa coordenada. (Idem.)
- *Complementar*, que invierte todos los valores de la matriz.

Elija la estructura y escriba los algoritmos de modo que las operaciones *Crear*, *Ver* y *Complementar* tomen $O(1)$ tiempo en peor caso.

Ejercicio 6. Una matriz finita posee las siguientes operaciones:

- *Crear*, con la cantidad de filas y columnas que albergará la matriz.
- *Definir*, que permite definir el valor para una posición válida.
- *#Filas*, que retorna la cantidad de filas de la matriz.
- *#Columnas*, que retorna la cantidad de columnas de la matriz.
- *Obtener*, que devuelve el valor de una posición válida de la matriz (si nunca se definió la matriz en la posición solicitada devuelve cero).
- *SumarMatrices*, que permite sumar dos matrices de iguales dimensiones.

Dado n y m son la cantidad de elementos no nulos de A y B, respectivamente, diseñe un módulo (elegir una estructura y escribir los algoritmos) para el TAD *MatrizFinita* de modo tal que dadas dos matrices finitas A y B,

(a) *Definir* y *Obtener* aplicadas a A se realicen cada una en $\Theta(n)$ en peor caso, y

(b) *SumarMatrices* aplicada a A y B se realice en $\Theta(n+m)$ en peor caso,

Ejercicio 7. Considere la siguiente especificación

```
Vagón es string
Tren es seq⟨Vagón⟩

TAD PlayaDeManiobras {
obs trenes: seq⟨Tren⟩

proc abrirPlaya (in capacidad : ℤ) : PlayaDeManiobras {
   requiere {capacidad > 1}
   asegura {|ret.trenes| = capacidad ∧ (∀i : ℤ)(0 ≤ i < capacidad) →_L (ret.trenes[i] = [])}
}

proc recibirTren (inout pdm : PlayaDeManiobras, in t : Tren) : ℤ {
   requiere {(∃v : ℤ)(0 ≤ v < |pdm.trenes|) ∧_L (pdm.trenes[v] = []) ∧ pdm = pdm_0}
   requiere {TodosLosVagonesSonDistintos(pdm, t)}
   asegura {(∃v)(0 ≤ v < |pdm_0.trenes|) ∧_L
                        (pdm_0.trenes[v] = [] ∧ pdm.trenes = setAt(pdm_0.trenes, v, t)) ∧ ret = v}
}

proc despacharTren (inout pdm : PlayaDeManiobras, in v) : ℤ {
   requiere {(0 ≤ v < |pdm.trenes|) ∧_L (pdm[v] ≠ []) ∧ pdm = pdm_0}
   asegura {pdm.trenes = setAt(pdm_0.trenes, v, [])}
}

proc unirTrenes (inout pdm : PlayaDeManiobras, in via1 : ℤ, in via2) : ℤ {
   requiere {(0 ≤ via1 < |pdm.trenes|) ∧ 0 ≤ via2 < |pdm.trenes|}
   requiere {pdm.trenes[via1] ≠ [] ∧ pdm.trenes[via2] ≠ []}
   requiere {via1 ≠ via2}
   requiere {pdm = pdm_0}
```

```
        asegura {|pdm| = |pdm0|}
        asegura {pdm.trenes[via1] = concat(pdm0.trenes[via1], pdm0.trenes[via2])}
        asegura {pdm.trenes[via2] = []}
        asegura {(∀v : Z)(0 ≤ v < |pdm.trenes| ∧ v ≠ via1 ∧ v ≠ via2) →L
                                    (pdm.trenes[v] = pdm0.trenes[v])}
    }

    proc moverVagon (inout pdm: PlayaDeManiobras, in vagon: Vagon, in viaOrigen: Z, in viaDestino: Z) {
        requiere {0 ≤ viaOrigen, viaDestino < |pdm.trenes|}
        requiere {vagon ∈ pdm.trenes[viaOrigen]))}
        requiere {pdm = pdm0}
        asegura {|pdm| = |pdm0|}
        asegura {vagon ∉ pdm.trenes[viaOrigen]}
        asegura {vagon ∈ pdm.trenes[viaDestino]}
        asegura {(∀v : Z)(0 ≤ v < |pdm.trenes| ∧ v ≠ viaDestino ∧ vagon ∉ pdm0.trenes[v]) →L
                                    (pdm.trenes[v] = pdm0.trenes[v])}
    }

    pred TodosLosVagonesSonDistintos (pdm: PlayaDeManiobras, t: Tren) {
        (∀vi : Z)(0 ≤ vi < pdm.trenes) →L
            (∀vg : Vagón)(vg ∈ pdm.trenes[vi] → vg ∉ t) ∧ (vg ∈ t → vg ∉ pdm.trenes[vi])
    }
}
```

1. Implementar el `TAD PlayaDeManiobras` usando listas enlazadas y arreglos.
2. Calcular la complejidad de las operaciones en peor caso en función de la cantidad de vías v y el largo del tren más largo t
3. Si la complejidad calculada para las operación `moverVagon()` es mayor a $O(t)$ y/o la de `unirTrenes()` es mayor a $O(1)$, modifique la estructura para lograr estas complejidades.

Ejercicio 8. Se desea diseñar un sistema de estadísticas para la cantidad de personas que ingresan a un banco. Al final del día, un empleado del banco ingresa en el sistema el total de ingresantes para ese día. Se desea saber, en cualquier intervalo de días, la cantidad total de personas que ingresaron al banco. La siguiente es una especificación del problema.

```
    TAD IngresosAlBanco {
    obs totales: seq⟨Z⟩
    proc nuevoIngresos () : IngresosAlBanco {
        asegura {totalDia == []}
    }

    proc registrarNuevoDia (inout i: IngresosAlBanco, in cant: Z) {
        requiere {cant ≥ 0}
        asegura {i.totales == old(i).totales + [cant]}
    }

    proc cantDias (in i: IngresosAlBanco) : : Z {
        asegura {res == |i.totales|}
    }

    proc cantPersonas (in i: IngresosAlBanco, in desde: Z, in hasta: Z) : Z {
        requiere {0 ≤ desde ≤ hasta ≤ |i.totales|}
        asegura {res = Σ_{j=desde}^{hasta} i.totales[j]}
    }
}
```

1. Dar una estructura de representación que permita que la función *cantPersonas* tome $O(1)$.
2. Calcular cómo crece el tamaño de la estructura en función de la cantidad de días que pasaron.

3. Si el cálculo del punto anterior fue una función que no es $O(n)$, piense otra estructura que permita resolver el problema utilizando $O(n)$ memoria.

4. Agregue al diseño del punto anterior una operación *mediana* que devuelva el último (mayor) día d tal que cantPersonas$(i, 1, d)$ $\leq$ cantPersonas$(i, d + 1$,totDias$(i))$, restringiendo la operación a los casos donde dicho día existe.

Ejercicio 9. La Maderera San Blas vende, entre otras cosas, listones de madera. Los compra en aserraderos de la zona, los cepilla y acondiciona, y los vende por menor del largo que el cliente necesite.

Tienen un sistema un poco particular y ciertamente no muy eficiente: Cuando ingresa un pedido, buscan el listón más largo que tiene en el depósito, realizan el corte del tamaño que el cliente pidió, y devuelven el trozo que queda al depósito.

Por otra parte, identifican a cada cliente con un código alfanumérico de 10 dígitos y cuentan con un fichero en el que registran todas las compras que hizo cada cliente (con la fecha de la compra y el tamaño del listón vendido).

Este sería el TAD simplificado del sistema:

```
Cliente es string
TAD Maderera {

proc comprarUnListon (inout m: Maderera, in tamaño: ℤ) {
    // comprar en el aserradero un listón de un determinado tamaño
}

proc venderUnListon (inout m: Maderera, in tamaño: ℤ, in cli: Cliente, in f: Fecha) {
    // vender un listón de un determinado tamaño a un cliente particular en una fecha determinada
}

proc ventasACliente (in m: Maderera, in cli: Cliente) {
    // devolver el conjunto de todas las ventas que se le hicieron a un cliente
    // (para cada venta, se quiere saber la fecha y el tamaño del listón)
}
}
```

Se pide:

- Escriba una estructura que permita realizar las operaciones indicadas con las siguientes complejidades:
  - comprarUnListon en $O(\log(m))$
  - venderUnListon en $O(\log(m))$
  - ventasAClient en $O(1)$

  donde m es la cantidad de pedazos de listón que hay en el depósito
- Escriba el algoritmo para la operación `venderUnListon`

Ejercicio 10. Se quiere implementar el TAD Biblioteca que modela una biblioteca con su colección de libros. Se modela la biblioteca como una sola estantería de capacidad arbitraria, dentro de la cual cada libro ocupa una posición. La biblioteca cuenta con un registro de socios que pueden retirar y devolver libros en cualquier momento. Por restricciones del sistema que se utiliza, un socio no puede registrarse con un nombre de más de 50 caracteres.

Cuando la biblioteca adquiere un nuevo libro o cuando un libro es devuelto, éste es insertado en el primer espacio libre de la estantería. Es decir, si los lugares ocupados son 1, 2, 3, 4 y se presta el libro en la posición 2, al agregar un nuevo libro al catálogo éste será ubicado en la posición 2. Cuando el libro que estaba originalmente en la posición 2 sea devuelto, será ubicado en la primera posición libre, que será la 5.

El TAD tiene las siguientes operaciones, para las que se nos piden las complejidades temporales de peor caso indicadas, donde

- L es la cantidad de libros en la colección
- r es la cantidad de libros que el socio en cuestión tiene retirados
- k la cantidad de posiciones libres en la estantería

- `proc AgregarLibroAlCatálogo`(inout b : *Biblioteca*, in l : *idLibro*)
  - **Requiere**: {l no pertenece a la colección de libros de b}
  - **Descripción**: la biblioteca adquiere un nuevo libro, lo suma a su catálogo y lo pone en la estantería en el primer espacio disponible.
  - **Complejidad**: $O(log(k) + log(L))$
- `proc PedirLibro`(inout b : *Biblioteca*, in l : *idLibro*, in s : *Socio*)
  - **Requiere**: {s es socio de la biblioteca y el libro l no está entre los libros prestados}
  - **Descripción**: el socio pasa a retirar un libro que se retira de la estantería y se acumula en sus libros prestados.
  - **Complejidad**: $O(log(r) + log(k) + log(L))$
- `proc DevolverLibro`(inout b : *Biblioteca*, in l : *idLibro*, in s : *Socio*)
  - **Requiere**: {s es socio de la biblioteca y el libro l está entre sus libros prestados}
  - **Descripción**: el socio pasa a devolver un libro que previamente había tomado prestado. Vuelve a la estantería en el primer espacio disponible.
  - **Complejidad**: $O(log(r) + log(k) + log(L))$
- `proc Prestados`(in b : *Biblioteca*, in s : *Socio*) : Conjunto$\langle$*Libro*$\rangle$
  - **Requiere**: {s es socio de la biblioteca}
  - **Descripción**: este procedimiento retorna los libros que el socio tomó prestados de la biblioteca y aún no devolvió.
  - **Complejidad**: $O(1)$
- `proc UbicaciónDeLibro`(in b : *Biblioteca*, in l : *idLibro*) : *Posicion*
  - **Requiere**: {l pertenece a la colección de libros de b y no está prestado}
  - **Descripción**: obtiene la posición del libro en la estantería.
  - **Complejidad**: $O(log(L))$

Los datos se van a guardar en la siguiente estructura incompleta

```
Socio es string; Libro, Posición es int

Módulo BibliotecaImpl implementa Biblioteca <
      var socios: Diccionario<Socio, Conjunto<Libro>> //Socios y los libros que tienen prestados
      var catálogo: Diccionario<Libro, Posición> //Libros y su posición en la estantería
      var posicionesLibres: Conjunto<Posición> //Posiciones vacías en la estantería
>
```

1. Definir qué estructura hay que implementar los diccionarios y conjuntos para cumplir las complejidades de las operaciones
2. Escribir los algoritmos mostrando cómo se cumplen las cotas de complejidad requeridas.

Ejercicio 11. Se quiere implementar el `TAD Agenda` que modela una agenda semanal donde se registran actividades. Cada actividad tiene un identificador, un horario de inicio y uno de finalización. No puede haber dos actividades con el mismo identificador. Para simplificar, las actividades sólo pueden comenzar y terminar en horarios en punto (por ejemplo, 21:00 hs) y terminan en un horario posterior a su inicio. Tampoco pueden empezar y terminar en días diferentes. Para contar la cantidad de actividades que transcurren en un determinado horario no se debe tener en cuenta aquellas que finalizan en ese momento. En cada actividad se pueden agregar tags que permiten agrupar las distintas actividades por temáticas. Los tags tienen cómo máximo 20 caracteres.

Dadas las siguientes operaciones y de acuerdo a las complejidades temporales de peor caso indicadas, donde

- d es la cantidad de días registrados hasta el momento
- a la cantidad total de actividades

- `proc RegistrarActividad`(inout $ag : Agenda$, in $act : IdActividad$, in $día : Día$, in $inicio : Hora$, in $fin : Hora$)
  1. **Requiere**: la hora de fin sea mayor que la de inicio y la actividad no está actualmente registrada.
  2. **Descripción**: se agrega la actividad a la agenda, marcando en el día indicado la hora de inicio y finalización.
  3. **Complejidad**: $O(\log(a) + \log(d))$
- `proc VerActividad`(in $ag : Agenda$, in $act : IdActividad$) : $struct < día : Día, inicio : Hora, fin : Hora >$
  1. **Requiere**: la actividad debe estar registrada en la agenda.
  2. **Descripción**: se devuelven el día y horario en que se realiza la actividad.
  3. **Complejidad**: $O(\log(a))$.
- `proc AgregarTag`(inout $ag : Agenda$, in $act : IdActividad$, in $t : Tag$)
  1. **Requiere**: la actividad está registrada en la agenda y aún no tiene registrado ese tag.
  2. **Descripción**: se agrega el tag a la actividad indicada.
  3. **Complejidad**: $O(1)$.
- `proc HoraMásOcupada`(in $ag : Agenda$, in $d : Día$) : $Hora$)
  1. **Requiere**: el día indicado tiene al menos una actividad registrada.
  2. **Descripción**: se devuelve la hora que tiene más actividades registradas.
  3. **Complejidad**: $O(\log(d))$.
- `proc ActividadesPorTag`(in $ag : Agenda$, in $t : Tag$) : $Conjunto < IdActividad >$
  1. **Requiere**: true.
  2. **Descripción**: se devuelven las actividades que tienen registrado el tag t.
  3. **Complejidad**: $O(1)$.

1. Definir la estructura de representación del módulo `AgendaImpl`, que provea las operaciones solicitadas.
2. Escribir en castellano el invariante de representación.
3. Escribir los algoritmos de todas las operaciones, explicando cómo se cumplen las complejidades pedidas para cada una.