Corrector: Fermín Schlottmann

Algoritmos y Estructuras de Datos
Segundo Parcial – Sábado 25 de Noviembre de 2023

#Orden	Libreta	Apellido y Nombre	E1	E2	E3	Nota Final
153	88/23	OTAZUA ARCE, MATEO	40	27	30	97

- Es posible tener una hoja (2 carillas), escrita a mano, con los anotaciones que se deseen, además de los apuntes de la cátedra.
- Cada ejercicio debe entregarse en **hojas separadas**.
- Incluir en cada hoja el número de orden asignado, número de libreta, número de hoja, apellido y nombre.
- El parcial se aprueba con 60 puntos. Para promocionar es necesario tener al menos 70 y ningún ejercicio con 0 puntos (en ambos parciales).

E1. Elección de estructuras (40 pts)

Se quiere implementar el TAD BIBLIOTECA que modela una biblioteca con su colección de libros. Por el momento la biblioteca cuenta con una sola estantería, 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.

Dadas las siguientes operaciones y de acuerdo a 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 y k la cantidad de posiciones libres en la estantería, respectivamente:

- `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<Libro>`
  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: $O(log(L))$

Se pide:

a) Plantear la estructura de representación del módulo `BibliotecaImpl`, que provea las operaciones mencionadas. Se debe explicar detalladamente qué información se guarda en cada parte, las relaciones entre ellas, y cómo se aseguraría que la información registrada es consistente.

b) Justificar cómo se cumplen las complejidades pedidas con esta estructura por cada operación. Indicar suposiciones sobre la implementación de las estructuras usadas y aclaraciones sobre aliasing.

c) Escribir los algoritmos de `AgregarLibroAlCatálogo` y `DevolverLibro`, justificando detalladamente que se cumplen las cotas de complejidad requeridas.

E2. Invariante de representación y función de abstracción (30 pts)

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:

```
TAD Comercio {
    obs ventasPorProducto: dict<Producto, seq<tupla<Fecha, Monto>>>
}

Producto es string
Monto es int
Fecha es int (segundos desde 1/1/1970)
```

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:

```
Modulo ComercioImpl implementa Comercio {
    var ventas: Secuencia<tupla<Producto, Fecha, Monto>>
    var totalPorProducto: Diccionario<Producto, Monto>
    var ultimoPrecio: Diccionario<Producto, Monto>
}
```

- **ventas** es una 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:

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

E3. Sorting (30 pts)

Se nos pide ayudar a un herborista que quiere poder organizar sus ingredientes para determinar qué hierbas le conviene recolectar. Para ello cuenta con su propio inventario. Como no es una persona muy organizada, puede tener distintas hierbas del mismo tipo en distintas alacenas o cofres. Luego de realizar una inspección de su lugar de trabajo, nos entrega una secuencia de n tuplas que constan de una hierba, identificada por su nombre, y la cantidad que se encontró. El nombre de cada hierba tiene como máximo 100 caracteres, de acuerdo al estándar de la Organización Mundial de Herboristas. El herborista cuenta a su vez con su libro de creaciones, que le permite saber en cuántas recetas se utiliza cada hierba.

Se necesita saber cuáles son las hierbas que se usan en más creaciones y, en caso de empate, deberían aparecer primero aquellos de las que tiene menos reservas. La complejidad esperada en el *peor caso* es de $O(n+h\ log(h))$, donde h es la cantidad de hierbas distintas con las que cuenta el herborista.

```
proc Recolectar(in s:Vector<tupla<string,int>>, in u:Diccionario<string,int>):Vector<string>
```

Ejemplo:

```
stock = [ ("Diente de León", 10), ("Menta", 4), ("Margarita", 13),
          ("Lavanda", 12), ("Diente de León", 5), ("Margarita", 6) ]

usos = {"Diente de León": 5, "Menta": 1, "Margarita": 3, "Lavanda": 5}

Recolectar(stock, usos) = ["Lavanda", "Diente de León", "Margarita", "Menta"]
```

Los primeros son la lavanda y el diente de león porque ambos tiene 5 usos, pero aparece primera la lavanda porque hay menos stock. En tercer lugar tenemos la margarita que tiene 3 usos. Finalmente, en último lugar está la menta, que tiene un solo uso.

a) Se pide escribir el algoritmo de `Recolectar`. Justificar **detalladamente** la complejidad y escribir todas las suposiciones sobre las implementaciones de las estructuras usadas, entre otras.
b) ¿Cuál sería el *mejor caso* para este problema? ¿Cuál sería la cota de complejidad más ajustada?

Ejercicio 1

a) modulo BibliotecaImpl implementa Biblioteca{

Posición es int
idLibro es int
Socio es string

var socios : DiccDig<Socio, ConjLog<idLibro>>
var catálogo : DiccLog<idLibro, Posición>
var estantería : ConjLog<Posición>
...
}

En la variable "socios", que es un diccionario, las claves son los socios, y los valores son conjuntos de libros retirados por cada uno. Como "Socio" tiene alfabeto y longitud acotadas, puedo acceder en O(1).

En la variable "catálogo", que es un diccionario, las claves son los libros, y los valores su respectiva posición. Cuando un libro fue retirado, su posición es -1.

En la variable "estantería", que es un conjunto, están las posiciones libres de la estantería. Cuando un libro es retirado, la posición que ocupaba se agrega a este conjunto. Cuando un libro es devuelto o agregado, tomará como posición el mínimo de este conjunto y se eliminará de este. Si se vacía el conjunto, se agregará la Posición catálogo.tamaño()+1. Asumimos la estantería como infinita (nunca se llena). Esto ConjLog va guardando el mínimo para poder acceder a él mediante el proceso ".mínimo()".
Podrían ser mejores nombres "retiradosPorSocio", "posiciónDeLibro" y "posicionesLibresEnEstantería", pero elegí estos para simplificar la notación en papel.

b) proc agregarLibro (b:Biblioteca, l:idLibro)
- toma el mínimo de "estantería" y lo elimina en O(log(K))
- agrega el libro a "catálogo" como clave, con el mínimo tomado como valor, en O(log(L))

complejidad total: O(log(K) + log(L))

proc pedirLibro (b:Biblioteca, l:idLibro, s:Socio)
- agrega el libro al conjunto de libros retirados por el socio en O(log(r)), (accede al conjunto en O(1))
- toma la posición que ocupaba y la cambia a -1 en O(log(L))
- agrega la posición liberada al conjunto de estantería en O(log(K))

complejidad total: O(log(r) + log(K) + log(L))

proc devolverLibro (b:Biblioteca, l:idLibro, s:Socio)
- elimina el libro del conjunto de libros retirados por el socio en O(log(r)), (accede al conjunto en O(1))
- toma el mínimo de "estantería" y lo borra en O(log(K))
- modifica la posición del libro en "catálogo" al mínimo tomado, en O(log(L))

complejidad total: O(log(r) + log(K) + log(L))

proc prestados (b:Biblioteca, s:Socio) : Conjunto<idLibro>
- toma el conjunto de libros retirados por el socio en O(1) y lo retorna (con aliasing)

complejidad total: O(1)

proc ubicaciónDeLibro (b:Biblioteca, l:idLibro) : Posición
- toma la posición del libro en "catálogo" en O(log(L)) y retorna ese valor (sin aliasing)

complejidad total: O(log(L))

c) proc agregarLibroAlCatálogo(b: BibliotecaImpl, l: idLibro) {

```
    int posición := b.estantería.mínimo();   // O(1), justificado en punto a
    b.estantería.sacar(posición);   // O(log(k))
    if (b.estantería.tamaño() == 0) then   // O(1)
        b.estantería.agregar(b.catálogo.tamaño() + 2);   // O(1), porque estaría vacío
    else                                                  // y es +2 porque no agregué el libro al catálogo todavía
        skip;
    endif;
    b.catálogo.definir(l, posición);   // O(log(L))
}
```

complejidad total: $O(\log(k) + \log(L))$

```
proc devolverLibro(b: BibliotecaImpl, l: idLibro, s: Socio) {
    b.socios.obtener(s).borrar(l);   // O(1) + O(log(r))
    int posición := b.estantería.mínimo();   // O(1)
    b.estantería.sacar(posición);   // O(log(k))
    if (b.estantería.tamaño == 0) then   // O(1)
        b.estantería.agregar(b.catálogo.tamaño()+1);   // O(1)
    else
        skip;
    endif;
    b.catálogo.definir(l, posición);   // O(log(L))
}
```

= agregarLibroAlCat(b, l).

complejidad total: $O(\log(r) + \log(k) + \log(L))$

Ejercicio 2

a) pred InvRep (c: ComercioImpl) {

- todo producto en "ventas" está en "totalPorProducto" y "últimoPrecio" y viceversa.
- para todo producto en "totalPorProducto", su valor es la sumatoria de todos los montos asociados al producto en "ventas".
- para todo producto en "últimoPrecio", su valor es el monto asociado al producto en "ventas" en la tupla con mayor Fecha.
- todos los ints tipo "Monto" son mayores o iguales a 0
- los ints tipo "Fecha" están entre 0 y los segundos que hayan pasado hasta AHORA desde 01/01/1970

}

Falta pedir que los productos no sean string vacío y que no haya fechas repetidas. Mal

proc FuncAbs (c: ComercioImpl) : Comercio {

res : Comercio | - todos los productos en "res.ventasPorProducto" están en "c.totalPorProducto" y viceversa
- para todo producto en "res.ventasPorProducto", todas las tuplas de la secuencia asociada tienen tupla equivalente en "c.ventas", y en "c.ventas" no hay más tuplas asociadas al producto que estas.

}

b) pred InvRep (c: ComercioImpl) {

forall p : Producto ::

(p in c.totalPorProducto $\longleftrightarrow$ p in c.últimoPrecio) $\wedge$

(p in c.totalPorProducto $\longleftrightarrow$ $(\exists p : int)(0 \leq i < |c.ventas| \wedge_L c.ventas[i]_0 = p)$ $\wedge$

(p in c.totalPorProducto $\rightarrow$ c.totalPorProducto[p] = sumCorrespondiente (c.ventas, p)) $\wedge$

(p in c.últimoPrecio $\rightarrow$ c.últimoPrecio[p] = ultPrecioCorrespondiente (c.ventas, p))

$\wedge$

forall i : int ::

$0 \leq i < |c.ventas| \rightarrow c.ventas[i]_2 \geq 0 \wedge 0 \leq c.ventas[i]_1 <$ now() No está definido

}

aux sumCorrespondiente (ventas : seq<<Producto, Fecha, Monto>>, p : Producto) =

$$\sum_{i=0}^{|ventas|-1} \text{if } (ventas[i]_0 = p) \text{ then } (ventas[i]_2) \text{ else } (0) \text{ endif}$$

aux ultPrecioCorrespondiente (ventas : seq<<Producto, Fecha, Monto>>, p : Producto) = → Mal definido. Debería haber usado un predicado

res : int | $(\exists i : int)(0 \leq i < |ventas| \wedge_L$ ~~$(\forall j : int)(ventas$~~ $ventas[i]_0 = p \wedge (\forall j : int)(ventas[j]_0 = p \rightarrow ventas[i]_1 \geq ventas[j]_1)$ $\wedge$ res = $ventas[i]_2$?

proc FuncAbs (c: Comercio Impl) : Comercio {

res: Comercio | forall p: Producto ::
p in res.ventasPorProducto <-> p in c.totalPorProducto ∧

p.in res.ventas -> forall v: <Fecha, Monto> ::
v in res.ventasPorProducto[p] <->
(∃ i: int) (0 ≤ i < |c.ventas| ∧_L c.ventas[i]_0 = p ∧ c.ventas[i]_1 = V_0 ∧ c.ventas[i]_2 = V

}

Ejercicio 3

a) proc recolectar (stock: Vector<<Hierba, Cantidad>>, usos: DiccDig<Hierba, Cantidad>): Vector<Hierba> {

- armo un DiccDig<Hierba, Cantidad> "stockTotal". // O(1)
- armo un arreglo "res" del tamaño de "usos", asumiendo que tiene solo // O(h) las hierbas que nos interesan.

O(n) (
- recorro "stock".
  - si no está la hierba en "stockTotal", la agrego con la cantidad correspondiente y la agrego a "res" // O(1)
  - si está, le sumo la cantidad correspondiente // O(1)

O(h * log(h)) (
- hago mergesort en "res" usando los DiccDig "stockTotal" y "usos", teniendo el siguiente criterio de comparación:

$$(x > y) \leftrightarrow ((usos[x] > usos[y]) \vee (usos[x] == usos[y] \wedge stockTotal[x] < stockTotal[y]))$$

- devuelvo "res" ya ordenado

}

// aclaración: como "stockTotal" y "usos" son diccionarios digitales, y sus claves tienen longitud acotada, sus operaciones se realizan en O(1)

b) El mejor caso sería que en "stock" vengan las hierbas ya juntas y ordenadas. En ese caso $h = n$, y el mergesort se ahorraría la mitad de las comparaciones, pero aún así, la complejidad sería $\Theta(h + h * \log(h))$, lo que equivale a $\Theta(h * \log(h))$.

Esto podría cambiar si antes del mergesort se ve si ya está ordenado en $O(h)$, y en tal caso saltar el mergesort, dejando la complejidad en $\Theta(h)$. (total)