DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Guia 8

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 8

1.1. 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 $\mathbb{Z}$
Fecha es $\mathbb{Z}$(segundos desde 1/1/1970)

```
TAD Comercio {
    obs ventasPorProducto: dict⟨Producto, seq⟨tupla⟨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 invrep y la func abs.
- Escribir ambos en el lenguaje de especificación.

InvRep:

- Las claves de totalPorProducto son iguales a las claves de ultimoPrecio.
- Una clave está en totalPorProducto si y solo si está como primer componente de algún elemento de ventas.
- La segunda componente de todos los elementos de ventas son distintas.
- La tercera componente de todos los elementos de ventas son positivas.
- El valor de todas las claves de totalPorProducto es igual a la suma de la tercer componente de todos los elementos de ventas que tengan como primer componente esa misma clave.
- El valor de todas las claves de ultimoPrecio es igual a la tercer componente del elemento de ventas que su primer componente es esa misma clave y la segunda componente es el mayor de todos los elementos de ventas con esa misma clave.

FuncAbs:

- Una clave pertenece a ventasPorProducto si y solo si pertenece a totalPorProducto
- El valor de una clave c perteneciente a ventasPorProducto es una secuencia de tuplas con la segunda y tercer componente de todos los elementos de ventas que tienen como primer componenete a c

```
pred clavesIguales (d1: dict⟨Producto, Monto⟩, d2: dict⟨Producto, Monto⟩) {
    (∀c : Producto) (c ∈ d1 ⟺ c ∈ d2)
}
pred todosFechasDistintas (v: seq⟨tupla⟨Producto, Fecha, Monto⟩⟩) {
    (∀i, j : ℤ) (0 ≤ i < j < v.length →_L v[i].fecha ≠ v[j].fecha)
}
pred todosMontosPositivos (v: seq⟨tupla⟨Producto, Fecha, Monto⟩⟩) {
    (∀i : ℤ) (0 ≤ i < v.length →_L v[i].monto ≥ 0)
}
pred esVentaMasRecienteDeProd (ventas: seq⟨tupla⟨Producto, Fecha, Monto⟩⟩, v: tupla⟨Producto, Fecha, Monto⟩)
{
    v ∈ ventas ∧ (∀v2 : tupla⟨Producto, Fecha, Monto⟩) (v1 ∈ ventas → v1.fecha ≤ v.fecha)
}
aux sumaMontosDeProducto (v: seq⟨tupla⟨Producto, Fecha, Monto⟩⟩, p: Producto) : ℤ =
    Σ_{i=0}^{v.length−1} IfThenElse(v[i].producto = p, v[i].monto, 0);

pred InvRep (c: ComercioImpl) {
    clavesIguales(c.totalPorProducto.data, c.ultimoPrecio.data) ∧
    (∀p : Producto) (
        p ∈ c.totalPorProducto.data ⟺ (∃v : tupla⟨Producto, Fecha, Monto⟩) (
            v ∈ c.ventas.s ∧ v.producto = p
        )
    ) ∧
    todosFechasDistintas(c.ventas.s) ∧ todosMontosPositivos(c.ventas.s) ∧
    (∀p : Producto) (
        p ∈ c.totalPorProducto.data → c.totalPorProducto.data[p] = sumaMontosDeProducto(c.ventas.s, p)
    ) ∧
    (∀p : Producto) (p ∈ c.ultimoPrecio.data → (∃v : tupla⟨Producto, Fecha, Monto⟩) (
        v ∈ c.ventas.s ∧ v.producto = p ∧ v.monto = c.ultimoPrecio.data[p] ∧
        esVentaMasRecienteDeProd(c.ventas.data, v)
    ))
}
aux ventasDeProd (v: seq⟨tupla⟨Producto, Fecha, Monto⟩⟩, p: Producto,
                  list: seq⟨tupla⟨Fecha, Monto⟩⟩) : seq⟨tupla⟨Fecha, Monto⟩⟩ =
    IfThenElse(v.length == 0, list,
        IfThenElse(v[0].producto = p,
            ventasPorProd(tail(v), p, list ++ ⟨v[0].fecha, v[0].monto⟩),
            ventasPorProducto(tail(v), p, list)));

pred predAbs (c1: ComercioImpl, c2: Comercio) {
    clavesIguales(c1.totalPorProducto.data, c2.ventasPorProducto) ∧
    (∀p : Producto) (
        p ∈ c2.ventasPorProducto → c2.ventasPorProducto[p] = ventasPorProd(c1.ventas.s, p, [])
    )
}
```

1.2. 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 {a ∈ p.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 invrep.
- Escribir la func Abs.

InvRep:
Opción 1:

- Todos los elementos de los valores de todas las claves de *alarmas* son claves de *sensores*
- Todos los elementos de los valores de todas las claves de *sensores* son claves de *alarmas*

Opción 2:

- Para toda clave k en *alarmas*, todo elemento e en $alarmas[k]$ es clave de *sensores*
- Para toda clave k en *sensores*, todo elemento e en $sensores[k]$ es clave de *alarmas*

Func Abs:
Sea $p1 : PlantaImpl, p2 : Planta$

- Una alarma a es una clave de $p1.alarmas$ si y solo si a pertenece a $p2.alarmas$
- Un par de sensor y alarma $\langle$s, a$\rangle$ pertenece a $p2.sensores$ si y solo si s pertenece a $p1.sensores$ y a pertenece a $p1.sensores[s]$

1.3. Ejercicio 3

HACER!

1.4. 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, in m: materia) : nota

    proc cantAlumnosConNota (in s: Sistema , in m: materia, in 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 $O(\log m)$
- registrarNota $O(\log n + \log m)$
- notaDeAlumno $O(\log n + \log m)$
- cantAlumnosConNota y cantAlumnosAprobados $O(\log m)$

Materia es String
Alumno es String
Nota es int

```
Módulo SistemaImpl implementa Sistema {
    var notas: DiccionarioLog⟨Materia, DiccionarioLog⟨Alumno, Nota⟩⟩
    var cantAlumnosPorNota: DiccionarioLog⟨Materia, Array⟨int⟩⟩

    pred esNota (n: ℤ) {
        0 ≤ n ≤ 10
    }
    pred InvRep (s: SistemaImpl ) {
        (∀m : Materia) (m ∈ s.notas.data ⟺ m ∈ s.notas.cantAlumnosPorNota.data) ∧
        (∀m : Materia) (m ∈ s.notas.data → esNota(s.notas.data[m])) ∧
        (∀m : Materia) (
            m ∈ s.cantAlumnosPorNota.data → s.cantAlumnosPorNota.data[m].length = 11 ∧
            (∀n : ℤ) (esNota(n) →_L (?))
        )
    }
    pred predAbs (s1: SistemaImpl , s2: Sistema ) {
        HACER!
```

```
}
proc nuevoSistema () : SistemaImpl
      Complejidad: O(1)

    res.notas := diccionarioVacio();
    res.cantAlumnosPorNota := diccionarioVacio();

    return res;

proc registrarMateria (inout s: SistemaImpl , in m: Materia)
      Complejidad: O(log m)

    var alumnos: DiccionarioLog < Alumno, Nota >;
    var notas: Array < int >;
    var i := 0;

    alumnos := diccionarioVacio();
    notas := new Array < int > (11);

    while i < 11 do
        notas[i] := 0;
        i := i + 1;
    endwhile

    s.notas.definir(m, alumnos);
    s.cantAlumnosPorNota.definir(m, notas);

proc registrarNota (inout s: SistemaImpl , in m: Materia, in a: Alumno, in n: Nota)
      Complejidad: O(log n + log m)

    var notasDeMateria: DiccionarioLog < Materia, Array < int >>;

    s.notas.obtener(m).definir(a, n);

    notasDeMateria := s.cantAlumnosPorNota.obtener(m);
    notasDeMateria[n] := notasDeMateria[n] + 1;

proc notaDeAlumno (in s: SistemaImpl , in a: Alumno, in m: Materia) : Nota
      Complejidad: O(log n + log m)

    return s.notas.obtener(m).obtener(a);

proc cantAlumnosConNota (in s: SistemaImpl , in m: Materia, in n: Nota) : int
      Complejidad: O(log m)

    return s.cantAlumnosPorNota.obtener(m)[n];

proc cantAlumnosAprobados (in s: SistemaImpl , in m: Materia) : int
      Complejidad: O(log m)

    var cant: int;
    var nota: int;
    var notasDeMateria: DiccionarioLog < Materia, Array < int >>;
    cant := 0;
    nota := 7;
    notasDeMateria := s.cantAlumnosPorNota.obtener(m);

    while nota < 11 do
```

```
            cant := cant + notasDeMateria[nota];
            nota := nota + 1;
        endwhile

        return cant;

}
```

1.5. 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 sobore 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.

Ejemplo de uso del módulo:

```
MatrizInfinita M := Crear()
bool b1 := Ver(M , 0, 0)
Asignar(M , 1, 3, False)
Asignar(M , 100, 5000, True)
bool b2 := M .Ver(100, 5000)
Complementar( M )
bool b3 := Ver(M , 394, 788)
bool b4 := Ver(M , 100, 5000)
```

Tras lo ue deberíamos tener

```
b1 = False
b2 = True
b3 = True
b4 = False
```

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

```
Módulo MatrizInfinita implementa MatrizInfinitaBooleanos {
    var matriz: Vector⟨Vector⟨bool⟩⟩
    var complementar: bool

    proc Crear () : MatrizInfinita
        res.matriz := vectorVacio();
        res.complementar := false;
        return res;

    proc Asignar (inout m: MatrizInfinita , in f: int, in c: int, in b: Bool)
    HACER!

    proc Ver (in m: MatrizInfinita , in f: int, in c: int) : Bool
        if m.matriz.longitud() == 0 || f >= m.matriz.longitud() || c >= m.matriz[0].longitud() then
            return m.complementar;
        else if m.complementar then
            return !m.matriz[f][c];
        else
            return m.matriz[f][c];
        endif

    proc Complementar (inout m: MatrizInfinita )
        m.complementar := !m.complementar;

}
```

1.6. Ejercicio 6

Una matriz finita posee las siguientes operaciones:

- Crear, con la cnatidad 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 columas 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 dados 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.

HACER!

1.7. 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: $seq\langle\mathbb{Z}\rangle$

`proc nuevoIngresos` () : $IngresosAlBanco$
`asegura` $\{res.totales = []\}$

`proc registrarNuevoDia` (inout i: $IngresosAlBanco$, in cant: $\mathbb{Z}$)
`requiere` $\{cant \geq 0 \wedge i = I_0\}$
`asegura` $\{i.totales = I_0.totales.concat(cant)\}$

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

`proc cantPersonas` (in i: $IngresosAlBanco$, in desde: $\mathbb{Z}$, in hasta: $\mathbb{Z}$) : $\mathbb{Z}$
`requiere` $\{0 \leq desde \leq hasta < |i.totales|\}$
`asegura` $\{res = \sum_{j=desde}^{hasta} i.totales[j]\}$

}

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 $cantPersonas(i, 0, d) \leq cantPersonas(i, d+1, cantDias(i)-1)$, restringiendo la operación a los casos donde dicho día existe.

```
Módulo IngresosAlBancoImpl implementa IngresosAlBanco {
    var totalesAcumulados: Vector⟨int⟩
    Crece proporcionalmente a la cantidad de días que pasaron

    proc nuevoIngresos () : IngresosAlBancoImpl

        res.totales := new Vector < int > ();
        return res;

    proc registrarNuevoDia (inout i: IngresosAlBancoImpl , in cant: int)

        if i.cantDias() == 0 then
            i.totalesAcumulados.add(cant);
        else
            i.totalesAcumulados.add(cant + i.totalesAcumulados[i.cantDias() − 1]);
        endif

    proc cantDias (in i: IngresosAlBancoImpl ) : int

        return i.totalesAcumulados.size();

    proc cantPersonas (in i: IngresosAlBancoImpl , in desde: int, in hasta: int) : int
        Complejidad: O(1)

        if desde == 0 then
            return i[hasta];
        endif

        return i[hasta] − i[desde − 1];

    proc mediana (in i: IngresosAlBancoImpl ) : int

        var mediana: int;
        var j: int;
        mediana := −1;
        j := 0

        while j < i.cantDias() do
            if i.cantPersonas(0, j) <= i.cantPersonas(j + 1, i.cantDias() − 1) && j > mediana then
                mediana := j;
            endif

            j := j + 1;
        endwhile

        return mediana;

}
```

1.8. 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 $O(\log(m))$
  - ventasACliente en $O(1)$

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

```
Módulo MadereraImpl implementa Maderera {
      var listones: ConjuntoLog⟨int⟩
      var ventasPorCliente: DiccDigital⟨Cliente, ListaEnlazada⟨struct⟨tamaño: int, fecha: Fecha⟩⟩⟩
      // Como las claves están acotadas a 10 dígitos, las operaciones son O(1)
      proc venderUnListon (inout m: MadereraImpl , in tamaño: int, in cli: Cliente, in f: Fecha)
            Complejidad: O(log(m))
            var venta: Struct < tamano: int , fecha: Fecha >;
            venta := new Struct < tamano: int , fecha: Fecha >;
            venta.tamano := tamano;
            venta.fecha := f;

            m.listones.sacar(tamano); // O( log m )

            if m.ventasPorCliente.esta(cli) then // O(1)
                m.ventasPorCliente.obtener(cli).agregarAtras(venta); // O(1)
            else
                var ventas: ListaEnlazada< Struct < tamano: int , fecha: Fecha > >;
                ventas := new ListaEnlazada();
                ventas.agregarAtras(venta); // O(1)
                m.ventasPorCliente.definir(cli , ventas); // O(1)
            endif

}
```

1.9. 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 es la cantidad de posiciones libres en la estantería

Socio es String; Libro, Posicion es int

```
Módulo Biblioteca {
    var socios: DiccionarioDigital⟨Socio, ConjuntoLog⟨Libro⟩⟩
    // Socios y los lirbos que tienen prestados

    var catalogo: DiccionarioLog⟨Libro, Posicion⟩
    // Libros y su posición en la estantería

    var posicionesLibres: ColaDePrioridadLog⟨Posicion⟩
    // Posiciones vacías en la estantería // Las posiciones mas chicas tienen mayor prioridad en la cola
```

`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**: $O(\log(k) + \log(L))$

```
var pos: Posicion;

if b.posicionesLibres.size() == 0 then
    pos := b.catalogo.size(); // O(1)
else
    pos := b.posicionesLibres.desencolar(); // O(log(k))
endif // O(log(k))

b.catalogo.definir(l, pos); // O(log(L))
```

Complejidad final: $O(\log(k) + \log(L))$

`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**: $O(\log(r) + \log(k) + \log(L))$

```
var pos: Posicion := b.catalogo.obtener(l); // O(log(L))
b.catalogo.borrar(l); // O(log(L))
b.posicionesLibres.encolar(pos); // O(log(k))
b.socios.obtener(s).agregar(l); // O(1) + O(log(r))
```

Complejidad final: $O(2\log(L) + \log(k) + 1 + \log(r)) = O(\log(r) + \log(k) + \log(L))$

`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**: $O(\log(r) + \log(k) + \log(L))$

```
b.socios.obtener(s).sacar(l); // O(1) + O(log(r))
var pos: Posicion;

if b.posicionesLibres.size() == 0 then
    pos := b.catalogo.size(); // O(1)
else
    pos := b.posicionesLibres.desencolar(); // O(log(k))
endif // O(log(k))

b.catalogo.definir(l, pos); // O(log(L))
```

Complejidad final: $O(1 + \log(r) + \log(k) + \log(L)) = O(\log(r) + \log(k) + \log(L))$

`proc Prestados` (inout 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**: $O(1)$

```
return b.socios.obtener(s); // O(1)
```

`proc UbicacionDeLibro` (inout 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**: $O(\log(L))$

```
return b.coleccion.obtener(l); // O(log(L))
```

}