Ejercicios introductorios de Elección de Estructuras

Román Gorojovsky & Camilo Semeria

4 de Junio de 2025

Enunciados

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.

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 que deberíamos tener

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

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

Ejercicio 8. 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`

Soluciones

Matriz Infinita

Cuando nos piden hacer una matriz y nos piden que la operación *Ver* sea $O(1)$ lo primero que debería ocurrírsenos es usar arreglos para acceder en esa complejidad a la íesima posición de cada dimensión. Considerando que las matrices que vamos a usar son de tamaño arbitrario, tiene sentido usar vectores, es decir, arreglos redimensionables, en vez de usar directamente arreglos, ya que la complejidad de agregar nuevos elementos cuando superamos la capacidad inicial es asintóticamente $O(1)$[1]

Entonces proponemos esta primer estructura:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <
    var matriz: Vector<Vector<bool>>
>
```

Veamos informalmente si con esta estructura podríamos cumplir las complejidades que nos piden

[1] Los detalles de esto están en las clases teóricas y también en el apunte de módulos básicos

- proc Ver(in $M : MIB$, in $f : \mathbb{Z}$, in $c : \mathbb{Z}$) : *bool*
  Es $O(1)$ porque, en principio, son dos accesos a vector.

- proc Crear() : MIB
  ¿Es $O(1)$? ¿Qué debería hacer el algoritmo de esta operación? El estado inicial de una matriz de booleanos es que todos los valores estén en *False*, es decir que `ver(M, f, c)` debería devolver **False** para cualquier valor. Entonces, para crear en $O(1)$, podemos inicializar con un vector vacío y en el algoritmo de `Ver()` devolver false para cualquier posición mientras no se asigne **`true`** en algún lado. De hecho, vamos a ver en el algoritmo completo que esto es cierto para cualquier f y c que sean mayores que las dimensiones mayores que asignadas hasta este momento.

  Con este cambio, el proc `ver` pasa a ser:

  - Si f y c son menores que los tamaños de los arreglos, devolver el valor de la posición
  - Si no, devolver `False`

- proc Asignar(inout $M : MIB$, in $f : \mathbb{Z}$, in $c\mathbb{Z}$, in $bbool$)
  No nos piden nada sobre la complejidad de esta operación, pero de todos modos veamos qué complejidad estaría teniendo. El peor caso es cuando tanto f como c sean mayores a las dimensiones asignadas previamente, en cuyo caso hay que recorrer todas las filas existentes, pedir memoria para extender los vectores y luego pedir las filas que falten y crear los vectores de tamaño c en cada posición. Esta operación es $O(f \times c)$, aunque como estamos trabajando con vectores en vez de arreglos no hay que copiar todos los datos al extender los que ya existen.

- proc Complementar(inout $MMIB$)
  Acá tenemos un problema: con la estructura que tenemos la única forma de implementar esta operación es recorriendo todos nuestros vectores de vectores y cambiando el valor en cada posición, lo que claramente no es $O(1)$.

¿Cómo resolvemos esto? Lo que podemos hacer es tener dos vectores de vectores, uno con los valores como entran y otro con los valores invertidos, más un booleano para definir de cuál leer. Entonces el proc `complementar` todo lo que hace es invertir ese booleano:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <

    var matriz: Vector<Vector<bool>>

    var matrizComplementada: Vector<Vector<bool>>

    var devolverComplementada: bool

>
```

Veamos de nuevo cómo serían informalmente los algoritmos y si cumple las complejidades

- proc Ver(in $M : MIB$, in $f : \mathbb{Z}$, in $c : \mathbb{Z}$) : *bool*
  Ahora es:

  - Si f y c son menores que los tamaños de los arreglos, devolver el valor de la posición en la matriz determinada por `devolverComplementada`
  - Si no, devolver `False` o `True` según corresponda

  Todas operaciones en $O(1)$

- proc Crear() : MIB
  Sigue prácticamente igual, $O(1)$.

- proc asignar(inout $M : MIB$, in $f : \mathbb{Z}$, in $c\mathbb{Z}$, in $bbool$)
  Ahora hay que cargar los datos en dos matrices separadas, que es dos veces la operación anterior así que el orden sigue siendo el mismo.

- proc Complementar(inout $MMIB$)
  Ahora esto es simplemente invertir un booleano, que es claramente $O(1)$

Con esto tenemos una solución correcta del ejercicio, pero no es la ideal. Las dos matrices están duplicando información, lo que deberíamos controlar en el Invariante de Representación, pero además en este caso duplicar la información es redundante. Podríamos guardar todo en una sola matriz y según el booleano decidir si complementar el valor que hay en la matriz antes de devolverlo en `ver` o el que entra en `asignar`. El módulo quedaría:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <
      var matriz: Vector<Vector<bool>>
      var devolverComplementada: bool
>
```

Y ahora sólo falta escribir los algoritmos:

```
function Crear : MIB
    MIB res := new MIB
    res.matriz = new Vector<Vector<bool>>
    # El booleano se inicializa solo en False
    return res
end function

function Ver(inout M: MIB, in f: int, in c: int) : bool
    bool res := False
    if EnRango(M, f, c) then
        res := M.matriz[f][c]
    end if
    if M.devolverComplementada then
        res := ¬ res
    end if
    return res
end function

function Asignar(inout M: MIB, in f: int, in c: int, in b: bool)
    bool b_a_guardar = b
    if M.devolverComplementada then
        b_a_guardar := ¬ b_a_guardar
    end if
    if b_a_guardar = True ∧ !EnRango(M, f, c) then
        # Pedir memoria para los vectores según sea necesario
    end if
    M.matriz[f][c] := b_a_guardar
end function

function Complementar(inout M: MIB)
    M.devolverComplementada := ¬ M.devolverComplementada
end function

function EnRango(inout M: MIB, in f: int, in c: int)
    return f < longitud(M.matriz) ∧ c < longitud(M.matriz[0])
end function
```

Maderera San Blas

Para implementar un TAD cumpliendo complejidades que nos indican un buen método es, primero entender qué información necesitamos guardar (generalmente va a ser la misma que está en los observadores del TAD) y después recorriendo las operaciones y tratando de entender qué restricciones nos están imponiendo a cómo guardar esta información.

En el caso de la maderera vamos a tener que guardar dos cosas:

- Los listones a vender
- Los clientes junto con la información de cada compra que hizo (que desde el punto de la maderera se llaman ventas)

Las operaciones que nos piden son las siguientes, desglosadas en todo lo que hace cada operación sobre nuestra estructura:

- comprarUnListon en $O(\log(m))$
  - Agrega un listón de un tamaño L a la colección de listones
- venderUnListon en $O(\log(m))$
  - Saca el listón más largo del depósito
  - Guarda el listón sobrante si corresponde
  - Registra la venta
- ventasACliente en $O(1)$
  - Devuelve todas las ventas hechas al cliente

Vemos que la complejidad de nuestras operaciones sólo depende de la cantidad de listones existentes y no depende ni de la cantidad de clientes registrados ni de las ventas que se hicieron. Sin embargo la cantidad de clientes es arbitraria, debería ser imposible tener una complejidad constante para buscar la información de uno de ellos. Por suerte tenemos un dato más: Los clientes se identifican con un código alfanumérico de tamaño fijo. Esto nos da la pista de que podemos implementar el fichero con un diccionario digital (o diccionario sobre *Trie*) de clientes que tiene el conjunto de ventas como significado, y obtener los datos en $O(1)$ como nos piden.

¿Cómo resolvemos las otras operaciones? Para guardar los listones en $O(log(m))$ podemos usar un conjunto logarítmico (sobre AVL) ordenando los listones por algún criterio, por ejemplo por tamaño. De hecho para representar los listones lo único que nos importa es el tamaño así que podemos representarlos directamente como `int`, y guardarlos en un conjunto.

Con estos dos criterios tendríamos esta primera versión del Módulo

`Liston` es `int`
`Cliente` es `string`

```
Módulo MadereraImpl implementa Maderera <
    var listones: ConjLog<Liston>
    var clientes: DiccDigital<Cliente, Conjunto<Tupla<Fecha, Liston>>>
>
```

Y podemos revisar si cumple las complejidades pedidas:

- comprarUnListon en $O(\log(m))$
  - Agrega un listón de un tamaño L al conjunto logarítimico de listones en $O(log(m))$ ✓

- venderUnListon en $O(\log(m))$
  - Saca el listón más largo del depósito – Buscarlo entre todos los elementos en $O(m)$ ✗
  - Guarda el listón sobrante si corresponde en $O(log(m))$ ✓
  - Registra la venta – Esto debería ser $O(1)$. Acceder a las ventas del cliente es $O(1)$, pero necesitamos que agregar la venta también lo sea ✗
- ventasACliente en $O(1)$
  - Devuelve todas las ventas hechas al cliente en $O(1)$ ✓

Tenemos dos problemas a resolver. Por un lado agregar las ventas al conjunto de cada cliente. Podemos asumir que todas las ventas son diferentes, entonces podemos usar la operación **agregarRapido**, que es $O(1)$ si usamos un Conjunto Lineal, es decir, sobre una lista enlazada.

Para resolver el problema de obtener el listón más grande, podemos usar una cola de prioridad, que nos permite:

- Obtener el valor máximo en $O(1)$
- Desencolarlo en $O(log(m))$
- Encolar uno nuevo en $O(log(m))$

La nueva estructura sería

```
Liston es int
Cliente es string
  Módulo MadereraImpl implementa Maderera <
      var listones: ColaPrioridadMax<Liston>
      var clientes: DiccTrie<Cliente, ConjLineal<Tupla<Fecha, Liston>>>
  >
```

Y ahora

- comprarUnListon en $O(\log(m))$
  - Agrega un listón de un tamaño L al conjunto logarítimico de listones en $O(log(m))$ ✓
- venderUnListon en $O(\log(m))$
  - Saca el listón más largo del depósito – Buscarlo entre todos los elementos en $O(m)$ ✓
  - Guarda el listón sobrante si corresponde en $O(log(m))$ ✓
  - Registra la venta – Acceder a las ventas del cliente es $O(1)$ y agregar la nueva también ✓
- ventasACliente en $O(1)$
  - Devuelve todas las ventas hechas al cliente en $O(1)$ ✓

Ahora sólo falta escribir los algoritmos

```
function CompararUnListon(inout M: Maderera, in l: Liston)
    res := M.listones.encolar(l)                                              ▷ O(log(m))
end function
function VenderUnListon(inout M: Maderera, in tamaño: Liston, in c: Cliente, in f: Fecha)
    int maxTamañoDisponible := M.listones.desencolar()                        ▷ O(log(m))
    int remanente := maxTamañoDisponible - tamaño                             ▷ O(1)
    if remanente > 0 then
        M.listones.encolar(remanente)                                         ▷ O(log(m))
    end if
    Tupla<Fecha, Liston> nuevaVenta := <f, tamaño>
    if ! M.clientes.está(c) then                                              ▷ O(1)
        M.clientes.definir(c, new ConjLineal<Tupla<Fecha, Liston>>)           ▷ O(1)
    end if
    ConjLineal<Tupla<Fecha, Liston>> ventasCliente := M.clientes.obtener(c)   ▷ O(1)
    ventasCliente.agregarRapido(nuevaVenta)                                   ▷ O(1)
end function
function VentasACliente(inout M: Maderera, in c: Cliente)Conjunto<Tupla<Fecha, Liston>>
    res := M.clientes.obtener(c)                                              ▷ O(1)
end function
```