Miércoles 17/09/2014

ESTRUCTURAS DE DATOS : Interfaces, data, Arboles

1. Modelo Puro (mencion de Haskell)
    - **Tipos Algebráicos**
        - Constructores (Pattern Matching)
        - Listas, árboles <- RECURSIÓN
    - **Tipos Abstractos de Datos (TADs)**
        - Interfaz (roles)
        - Eficiencia
            - Modelo:
                - Peor caso
                - Costo amortizado
            - Medidas: (proporcionales a la cantidad de datos, n)
                - Constante O(1)
                - Lineal O(n)
                - Cuadrática O(n)
2. Modelo Destructivo
3. Temas adicionales

Usuario programador	Diseñador	Implementador

`module Usuario where`
`  import qualified QueueM as Q`
`lenQ :: Q.Queue a -> Int`
`lenQ q = if Q.isEmptyQ q`
`  then 0`
`  else 1 + lenQ (Q.dequeue q)`
	`emptyQ  :: Queue a`
	`queue  :: Queue a -> a -> Queue a`
	`isEmptyQ  :: Queue a -> Bool`
	`dequeue  :: Queue a -> Queue a`
	`firstQ  :: Queue a -> a`
		`module QueueM (Queue, emptyQ, isEmptyQ, queue, dequeue, firs`
		`data Queue a = Q[a]`
		
		`emptyQ  = Q[]`
		`queue  =  -- ...`
		`isEmptyQ q  = isNil ( dameTuRep q )`
		`dequeue q  =  -- ...`
		`firstQ  =  -- ...`
		
		`dameTuRep :: Queue a -> [a]`
		`dameTuRep (Q xs) = xs d`

Eficiencia

Medimos la eficiencia en términos de cantidad de operaciones a realizar para resolver un problema, no en términos de tiempo.

Modelos de eficiencia

- Peor caso: medimos considerando cual es la mayor cantidad de operaciones que se van a realizar.
- Costo Amortizado: Cada cierta cantidad de operaciones de poco costo, se realiza una operación más costosa.

Medidas

Hacemos una estimacion proporcional a la cantidad de datos que vamos a trabajar. Esas proporciones son:

- **Constante** : O(1) (se lee "O de 1", o "de orden 1"). Lleva una cantidad constante de operaciones solucionar el problema. O sea, no importa cuantos datos haya, la cantidad de operaciones que toma resolver el problema es SIEMPRE LA MISMA.
- **Lineal**: O(n) . Lleva tantas operaciones como elementos sobre los que hay que trabajar hayan. Ej: si tengo 1000 personas en una lista y la tengo que recorrer me lleva aproximadamente 1000 operaciones.
- **Cuadrática**: O(n^2). Cuando hay que recorrer n veces (no es exacto, a veces pueden ser menos de n veces), cada uno de los n datos que tenemos.

¿Cómo evaluar cual alternativa nos conviene elegir a la hora de implementar?

Suponemos la siguiente implementación de Queue

```
    emptyQ      = Q[]                                       -- O(1)
    queue q x   = Q ( agregarAlPrincipio x (dameTuRep q) )  -- O(1)
    isEmptyQ q  = null (dameTuRep q)                        -- O(1)
    dequeue q   = Q ( sacarElUltimo (dameTuRep q) )         -- O(n)
    firstQ q    = dameElUltimo (dameTuRep q)                -- O(n)

    agregarAlPrincipio :: a -> [a] -> [a]
    agregarAlPrincipio x xs  = x:xs

    sacarElUltimo :: [a] -> [a]   -- O(n)
    sacarElUltimo [x]    = []
    sacarElUltimo (x:xs) = x:(sacarElUltimo xs)

    dameElUltimo :: [a] -> a
    dameElUltimo [x]    = x
    dameElUltimo (x:xs) = dameElUltimo xs
```

Ahora medimos la eficiencia de lenQ con la implementación de Queue que acabamos de implementar

```
module Usuario where
    import qualified QueueM as Q

    lenQ :: Q.Queue a -> Int
    lenQ q = if Q.isEmptyQ q
             then 0
             else 1 + lenQ (Q.dequeue q)    -- la recursión es O(n)
                      -----------
                          O(n)              -- dequeue es O(n)

    -- lenQ termina siendo O(n^2)
```

lenQ termina siendo cuadrática porque al recorrer toda la cola con recursión, recorro todos los elementos de la cola; pero como en cada operación de recursión llamo a dequeue y el dequeue implementado arriba es O(n) entonces termino recorriendo n^2 veces la lista que representa a la Queue (o sea, O(n) operaciones para cada dequeue, y se hacen n de esos, uno para cada elemento de la cola).

Ahora arreglamos la implementación de Queue para mejorar la eficiencia de lenQ

Para mejorar la eficiencia de lenQ tengo que hacer que dequeue sea O(1)

```
emptyQ      = Q []                                      -- O(1)
queue q x   = Q ( agregarAlFinal x (dameTuRep q) )      -- O(n)
isEmptyQ q  = null (dameTuRep q)                        -- O(1)
dequeue q   = Q ( sacarElPrimero (dameTuRep q) )        -- O(1)
firstQ q    = dameElPrimero (dameTuRep q)               -- O(1)

agregarAlFinal a -> [a] -> [a]
agregarAlFinal x xs = xs ++ [x]

sacarElPrimero :: [a] -> [a]
sacarElPrimero (x:xs) = xs

dameElPrimero :: [a] -> a
dameElPrimero (x:xs) = x
```

La elección de costos que hagamos siempre va a depender de las necesidades que tengamnos.
Ahora podemos suponer que el usuario le dice al diseñador que necesita una operación lenQ porque su programa es ineficiente. Entonces el diseñador, que tiene permisos de modificar la interfaz, acepta y agrega `lenQ :: Queue a -> Int` a la misma.

Ahora el implementador puede crear una nueva Queue, con una nueva representación que le permita recordar la longitud sin necesidad de calcularla cada vez. El nuevo tipo ya no va a ser Queue, sino QueueLong

```
module QueueLong(Queue, emptyQ, isEmptyQ, queue, dequeue, firstQ, lenQ) where

data Queue a = Q2 [a] Int
    {- INV. de REP: Q xs h
                          ,  h = length xs
     -}

emptyQ              = Q2 [] 0
            -- donde 0 es la longitud de una cola vacía justamente

queue (Q2 xs h) x   = Q2 (agregarR xs x) (h+1)
            {-
             -- al desarmar la representación de Queue con Pattern Matching
             -- obtengo xs , que es la lista de elementos y h que es el largo
             -- de la cola actual. queue devuelve una nueva Queue donde ademas
             -- de agregarse un elemento en la lista de la representación, se
             -- sumó 1 a la longitud
            -}

isEmptyQ (Q2 xs h)  = null xs
dequeue (Q2 xs h)   = sacarUnoYRearmar xs h
firstQ (Q2 xs h)    = head xs
lenQ (Q2 xs h)      = h

sacarUnoYRearmar [] _ = error "No hay elementos para sacar"
sacarUnoYRearmar xs h = Q2 (tail xs) (h-1)

agregarR xs x = x ++ [x]
```

Invariantes de Representación: son condiciones que los datos tienen que cumplir para que yo pueda "poner la tapa" a la representación (aplicar la función de abstracción). Para que los datos tengan sentido.

Ejemplos de Costo Amortizado

Se plantea la necesidad de implementar una Queue que cuyas operaciones `queue` y `dequeue` sean eficientes. O sea, agregar adelante y sacar de adelante.
Con la representación que teníamos hasta ahora no podíamos hacer esto. Por lo tanto proponemos una nueva representación de cola que permita llevar esto a cabo.
Proponemos una cola que este representada por 2 listas, una lista "frontal", donde está la gente que está adelante en la cola, y la segunda lista donde está la gente que va llegando, es la lista de atrás. La lista frontal va a tener los elementos ordenados de manera que pueda OBTENER EL PRIMER ELEMENTO en O(1), mientras que la lista de atrás va a tener sus elementos ordenados de manera que pueda AGREGAR UN ELEMENTO en O(1). Ej:

```
[ "Primero" , "Segundo" , "Tercero" ]  -- lista frontal
[ "Sexto"   , "Quinto"  , "Cuarto"  ]  -- lista de atrás
```

Esta representación va a ser muy eficiente tanto para agregar como para obtener el primer elemento. PERO, cada tanto vamos a tener una operación de costo alto porque cuando se vacíe la lista frontal, voy a tener que "recargarla" moviendo los elementos de la lista de atrás a la lista frontal, al revés de como están.

Implementación de la Queue con 2 listas

```
data Queue a = Q3 [a] [a]
        {- INV. de REP: Q fs bs
                          , fs es vacío => bs es vacío (si hay gente en la cola, la parte de adelante, el "frente", no está vacío)
         -}

emptyQ              = Q3 [] []
queue (Q3 fs bs) x  = armarCola fs (x:bs)
isEmptyQ (Q3 fs bs) = null fs
dequeue (Q3 fs bs)  =
firstQ (Q3 fs bs)   = head fs
```

Se plantea un ejercicio entregable: Implementar `armarCola :: [a] -> [a] -> Queue a` y `dequeue :: Queue a -> Queue a`:

```

-- PROP: arma una cola GARANTIZANDO mantener el invariante de representación
armarCola :: [a] -> [a] -> Queue a
armarCola [] bs = Q3 (reverse bs) []
armarCola fs bs = Q3 fs bs

dequeue :: Queue a -> Queue a
dequeue (Q3 fs bs) = armarCola (tail fs) bs  -- puedo hacer (tail fs) porque dequeue es parcial en emptyQ
```

Análisis de costos

```
emptyQ              = Q3 [] []                  -- O(1)
queue (Q3 fs bs) x  = armarCola fs (x:bs)       -- O(n) (en peor caso)
isEmptyQ (Q3 fs bs) = null fs                   -- O(1)
dequeue (Q3 fs bs)  = armarCola (tail fs) bs    -- O(n) (en peor caso)
firstQ (Q3 fs bs)   = head fs                   -- O(1)
```

Analizo un caso en particular . Consideramos la siguiente secuencia de operaciones de `queue` y `dequeue`

```
emptyQ ->   queue       ->  queue       ->  queue       ->  queue       ->  dequeue     ->  queue
[] []       [1] []          [1] [2]         [1] [2,3]       [1] [2,3,4]     [2,3,4] []      [2,3,4] [5]
            1:fs            2:bs            3:bs            4:bs            reverse         5:bs        -- operacion
 O(1)       O(1)            O(1)            O(1)            O(1)            O(n)            O(1)        -- costo
```

Como se puede ver, si bien `dequeue` es O(n) en peor caso, a la hora de usarlo vemos que este O(n) solo ocurre una vez cada tanto. Por eso decimos que esta implementación de la función `dequeue` no es O(n) realmente, sino **O(1) AMORTIZADO**, porque "cada tanto" `dequeue` cuesta O(n).

Introducción a Diccionarios (Mappings o asociaciones)

Un diccionario es un tipo de datos donde puedo asociar ***Claves*** con ***Valores***. Por ejemplo un diccionario en la vida real: tenemos palabras asociadas a sus definiciones. Uno busca una palabra en el diccionario y al encontrar la palabra obtiene la definición. Pero para buscarla necesitamos esa clave: la palabra. Ej:

Diccionarios
    Palabra -> Definición
Agenda
    Nombre -> Nro de Telefono
Tesauro (diccionario de sinónimos)
    Palabra -> [Sinónimo]
Memoria
    Variable -> Dato

Definimos la interfaz

```
module MapM (Map, emptyM, assocM, lookupM, deleteM, domM)
    where
data  Map    k        v    = M [(k,v)]
    --    (clave)  (valor)

-- PROP: Crea un Diccionario Vacío
emptyM :: Map k v

-- PROP: Agrega una entrada al diccionario que para la clave k, le asocia el valor v. Si la clave ya estaba, CAMBIA su valor
assocM :: Map k v -> k -> v -> Map k v

-- PROP: Busca en el diccionario una entrada cuya clave sea k y devuelve (si hay) su valor.
-- Obs:  El tipo que devuelve lookupM es un Maybe v. Maybe es un tipo algebráico que devuelve "tal vez" un valor, o si no está, "nada
--       (el tipo de dato Maybe se explica en detalle mas abajo)
lookupM :: Map k v -> k -> Maybe v

-- PROP: Borra de un diccionario dado la entrada cuya clave es k. y Devuelve un diccionario sin esa entrada.
deleteM :: Map k v -> k -> Map k v

-- PROP: Devuelve el Set de elementos de tipo k que es el conjunto de TODAS las claves que hay en el Diccionario dado.
domM :: Map k v -> Set k
```

Implementación de Map k v

```
emptyM              = M []
assocM (M kvs)  k v = M (agregar (k,v) kvs)

-- COMPLETAR en la práctica
```

Tipo algebraico Maybe

```
data Maybe a = Just a | Nothing
```

El tipo Maybe es un tipo que se usa para denotar la POSIBILIDAD de que haya un valor de un tipo *a* o la alternativa de que no lo haya. Por ejemplo, una función como lookupM que busca un dato en un diccionario según su clave; puede no encontrar una entrada para esa clave. Dada esta posibilidad, en vez de lanzar un "error", lookupM puede devolver Nothing (Nada) denotando que no se encontró un valor para esa clave en el diccionario. Si en cambio sí se encontró puede devolver un valor de tipo Maybe que indique el valor encontrado (Just valor) que luego podrá descomponerse con pattern matching para extraer el valor deseado.

```
-- EJ:
-- PROP: devolver un string que diga informe en palabras claras si se encontró o no una palabra en el diccionario
mensajeEncontradoONo :: k -> Maybe v -> String
mensajeEncontradoONo Nothing  = "No se encontró " ++ k ++ " en el diccionario"
mensajeEncontradoONo (Just v) = "El significado de " ++ k ++ " es " ++ v

mensajeEncontradoONo "recursión" (lookupM miDiccionario "recursión")
```