Estructuras de datos y algoritmos II

27 de febrero de 2019

Índice general

Capítulo 1

Análisis de algoritmos

1.1. Introducción

Supongamos que las computadoras fueran infinitamente rápidas y completamente gratuitas. Dados dos algoritmos que resuelven correctamente el mismo problema: ¿Existe alguna razón para utilizar uno en vez de otro? Mas allá de la facilidad de implementación y claridad del código, no habría mas razones para optar por alguno.

Desafortunadamente las computadoras no son infinitamente rápidas y tampoco son gratuitas, por lo tanto, el tiempo computacional es un recurso acotado que debe ser aprovechado sabiamente y el análisis de eficiencia de algoritmos es la herramienta para realizar dicha tarea.

¿Como puede compararse la eficiencia de dos algoritmos? Un primer enfoque podría ser, correr el algoritmo para diferentes tamaños de entrada y comparar los tiempos de ejecución. Sin embargo esta idea tiene algunos inconvenientes:

- Podría ocurrir que para algunos tamaños de entrada sea mas rápido un algoritmo y para otros tamaños sea mas eficiente el otro.
- Algunos algoritmos podrían tener diferentes tiempos de ejecución dependiendo de la arquitectura en donde se ejecuten.

Por ejemplo supongamos que un algoritmo para ordenar una lista de n elementos tarda $2 \cdot n^2 + 1$ segundos y otro algoritmo tarda $7 \cdot n \cdot \log_2(n) + 5$ segundos.

- Para ordenar una lista de 4 elementos el primer algoritmo tarda $2 \cdot 4 \cdot 4 + 1 = 33$ y el segundo $7 \cdot 4 \cdot 2 + 5 = 61$. Aparentemente el primer algoritmo es dos veces mas rápido que el segundo.
- Para ordenar una lista de 1024 elementos el primer algoritmo tarda $2 \cdot 1024 \cdot 1024 + 1 = 2097153$ y el segundo $7 \cdot 1024 \cdot 10 + 5 = 7168$. Aparentemente el segundo algoritmo es 290 veces mas rápido que el primero.

Para entradas chicas el primer algoritmo es ligeramente mejor, pero para entradas grandes el segundo algoritmo es mucho mejor. Sin importar que tan grande sea el factor constante que multiplica al factor variable, siempre habrá un tamaño de entrada a partir del cual el segundo algoritmo es mejor.

El análisis asintótico de funciones nos permite hacer la comparación que necesitamos. Es decir, analizamos el comportamiento de las funciones a medida que las entradas tienden a infinito.

1.2. Notación asintótica

La notación asintótica nos comparar el orden de crecimiento de dos funciones de manera sencilla (evitando el calculo de limites).

1.2.1. Notación O

Sean $f, g : \mathbb{N} \to \mathbb{R}$, decimos que f tiene orden de crecimiento $O(g)$ (y lo notamos $f \in O(g)$) si existen constantes $c \in \mathbb{R}$ y $n_0 \in \mathbb{N}$ tales que:

$$\forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

Si f es una función que indica el tiempo que tarda un algoritmo en retornar el resultado en relación al tamaño de la entrada, $f \in O(g)$ nos indica que para entradas lo suficientemente grandes (a partir de n_0), el peor comportamiento que podemos esperar es el de g multiplicada por alguna constante.

A modo de ejemplo, consideremos un típico algoritmo que determine si un elemento se encuentra en una lista simplemente enlazada. Para una lista de n elementos, en el peor caso este algoritmo realizara n comparaciones. Si cada comparación tarda 2 segundos y retornar un valor tarda 1 segundo, la función que expresa el tiempo que tarda el algoritmo para retornar el resultado es $f(n) = 2n + 1$. Si tomamos $n_0 = 1$, podemos ver que $f \in O(n)$ de la siguiente manera:

$$0 \leq 1 \leq n \iff 2n \leq \underbrace{2n+1}_{f(n)} \leq \underbrace{3n}_{cg(n)}$$

Un análisis mas profundo nos revela que $f \in O(g)$ independientemente del coeficiente lineal y el termino independiente, por lo cual podemos comparar dos algoritmos sin importar cuanto tarda cada comparación. Ademas, aunque para entradas chicas un algoritmo $f'(n) \in O(n^2)$ puede resolver el problema en menos tiempo, para entradas grandes f siempre será mejor.

Notese que el elemento buscado no siempre va a estar al final de la lista, es por eso que también nos interesa preguntarnos cual es la complejidad en el mejor de los casos.

1.2.2. Notación Ω

Sean $f, g : \mathbb{N} \to \mathbb{R}$, decimos que f tiene orden de crecimiento $\Omega(g)$ (y lo notamos $f \in \Omega(g)$) si existen constantes $c \in \mathbb{R}$ y $n_0 \in \mathbb{N}$ tales que:

$$\forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$$

Es decir que en el mejor de los casos, podemos esperar que f se comporte como g. En nuestro ejemplo de buscar un elemento en una lista enlazada, si el elemento buscado es el primero el algoritmo tardara $f(n) = 2 + 1$. Podemos ver que $f \in \Omega(1)$ de la siguiente forma:

$$0 \leq \underbrace{3 \cdot 1}_{cg(n)} \leq \underbrace{2+1}_{f(n)}$$

1.2.3. Notación Θ

Decimos que f tiene orden de crecimiento $\Theta(g)$ (y lo notamos $f \in \Theta(g)$) si $f \in O(g)$ y $f \in \Omega(g)$.

La notación Theta expresa una cota justa del comportamiento asintótico de una función, es decir que podemos acotar inferior y superiormente por la misma función.

1.2.4. Propiedades

COMPLETAR.

1.2.5. Conclusión

En definitiva la notación asintótica nos permite comparar algoritmos para entradas grandes en el peor y mejor de los casos. Hay que tener en cuenta que dos funciones con el mismo orden de crecimiento no son igualmente eficientes, por ejemplo $3n^2 + 2n + 7$ y $n^2 + n + 2$ tiene ambas orden $O(n^2)$ pero sin embargo la segunda es mas eficiente.

Finalmente terminamos con una lista de las complejidades mas frecuentes en orden creciente:

- 1.
- $\log_2(n)$.
- n.
- $n \cdot \log_2(n)$.
- n^2.
- n^k.
- 2^n.
- k^n.
- $n!$.

1.3. Modelo de costos

Para especificar cual es el costo de un algoritmo para una entrada determinada es necesario establecer un modelo de costos.

Para ello definiremos el trabajo (costo del algoritmo con un solo procesador) y profundidad (costo del algoritmos con infinitos procesadores) basados en un lenguaje.

1.3.1. Trabajo

- $W(c) = 1$.
- $W(op\ e) = 1 + W(e)$.
- $W(e_1, e_2) = 1 + W(e_1) + W(e_2)$.
- $W(e_1 ||| e_2) = 1 + W(e_1) + W(e_2)$.
- $W(\text{let } x = e_1 \text{ in } e_2) = 1 + W(e_1) + W(e_2[x := Eval(e_1)])$.
- $W(\{f(x) / x \in A\}) = 1 + \sum_{x \in A} W(f(x))$.

A modo de ejemplo, analicemos el trabajo de un algoritmo que calcula la longitud de una lista simplemente enlazada:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Para una lista de longitud 0 (primer caso) el algoritmo retorna una expresión constante (0), es decir que su trabajo será $W(0) = 1$.

Para una lista de longitud n (segundo caso) el algoritmo opera (suma) sobre dos expresiones (1 y length xs), por lo que su trabajo será:

$$W(n) = 1 + W(1, length\ xs) = 1 + 1 + W(n-1)$$

Si continuamos desarrollando la expresión observaremos que sumaremos $1+1$ por cada elemento de la lista, es decir que su trabajo será $W(n) = 2n$.

Para poder realizar este análisis fue indispensable notar que el llamado recursivo se realiza sobre una lista de $n-1$ elementos.

1.3.2. Profundidad

- $S(c) = 1$.
- $S(op\ e) = 1 + S(e)$.
- $S(e_1, e_2) = 1 + S(e_1) + S(e_2)$.
- $S(e_1 ||| e_2) = 1 + \text{máx}\{S(e_1), S(e_2)\}$.
- $S(\text{let } x = e_1 \text{ in } e_2) = 1 + S(e_1) + S(e_2[x := Eval(e_1)])$.
- $S(\{f(x)/x \in A\}) = 1 + \underset{x \in A}{\text{máx}} S(f(x))$.

Calculemos la profundidad de un algoritmo que determina la pertenencia de un elemento a un árbol binario completo con información en las hojas:

```
member x E          = False
member x (Leaf y)   = if x == y then True else False
member x (Node l r) = let (a,b) = member x l ||| member x r
                      in a || b
```

Para un árbol vacío (0 elementos) su profundidad es la de una constante ($False$) por lo tanto tiene profundidad $S(0) = 1$.

Para un árbol con una sola hoja (1 elemento) su profundidad es la de operar (comparar) dos expresiones (x e y), operar (decidir) según su resultado y retornar una constante ($True$ o $False$). Es decir: $S(1) = 1 + 1 + 1 = 3$.

Finalmente para un árbol completo de n elementos su profundidad es:

$$S(n) = 1 + \text{máx}\{member\ x\ l, member\ x\ r\} = 1 + S(n/2)$$

Si continuamos desarrollando esa suma observaremos que sumamos una unidad, $\log_2(n)$ veces, y una vez mas al llegar a una hoja, es decir $S(n) = \log_2(n) + 1$.

Vale la pena observar que en un árbol binario completo cada rama contiene la mitad de los elementos.

1.3.3. Paralelismo

El paralelismo es una medida de la cantidad de procesadores que pueden utilizarse de forma eficiente. Dado un algoritmo con trabajo en el orden de $O(f(n))$ y profundidad en $O(g(n))$ calculamos el paralelismo como:

$$P = \frac{f(n)}{g(n)}$$

A la hora de buscar algoritmos eficientes, buscamos aquellos que tengan el mejor trabajo y entre todos ellos, elegimos el de mayor paralelismo.

En una maquina con p procesadores, un algoritmo con trabajo W y profundidad S puede correr en un tiempo

$$T < \frac{W}{p} + S = \frac{W}{p} + \frac{W}{P} = \frac{W}{p}\left(1 + \frac{p}{P}\right)$$

suponiendo que el repartidor de tareas cada vez que haya un procesador libre y tareas pendientes por ejecutar, las asigne inmediatamente; y que los costos de comunicación (latencia y ancho de banda) sean bajos.

Si P es considerablemente mas grande que p, entonces la cota es prácticamente optima.

1.4. Resolución de recurrencias

Al plantear el trabajo y profundidad de algoritmos recursivos surgen expresiones recursivas que será necesario resolver. En esta sección se estudiaran varias técnicas.

1.4.1. Método inductivo (substitución)

Una forma de resolver la complejidad de un trabajo expresado en forma recursiva es intentar adivinar cual puede ser la solución, y luego demostrarlo por inducción.

Por ejemplo, supongamos un algoritmo que para una entrada de 0 elementos tiene trabajo $W(0) = c_1$, para una entrada de un elemento $W(1) = c_2$ y para n elementos $W(n) = 2 \cdot W(\lfloor n/2 \rfloor) + c_3 n$.

Para facilitar el análisis haremos caso omiso de la función piso, y mas adelante estudiaremos en que situaciones es correcto hacer esta omisión.

Si observamos la recurrencia podemos intuir que sumaremos $c_3 n$, $\log_2(n)$ veces, por lo que adivinamos que $W(n) \in O(n \cdot \log_2(n))$. A continuación realizamos la prueba:

- Caso base $n = 2$: $W(n) = 2 \cdot W(1) + 2c_3 = 2 \cdot c_2 + 2c_3 = 2(c_2 + c_3) = nc \cdot 1 = nc \cdot \log_2(n) \in O(n \cdot \log_2(n))$.
- Caso inductivo. Supongamos que para $3 \leq n < k$ resulta que $W(n) \in O(n \cdot \log_2(n))$ es decir, $0 \leq 2 \cdot W(n/2) + c_3 n \leq cn \cdot \log_2(n)$. En particular también

vale para $k/2$, luego:

$$W(k) = 2 \cdot W(k/2) + c_3 k \leq 2 \cdot [c(k/2) \cdot \log_2(k/2)] + c_3 k = ck \cdot \log_2(k/2) =$$

$$= ck \cdot \log_2(k) - ck \cdot \log_2(2) + c_3 k = ck \cdot \log_2(k) - ck + c_3 k \leq ck \cdot \log_2(k)$$

La ultima desigualdad se cumple si tomamos $c \geq c_3$.

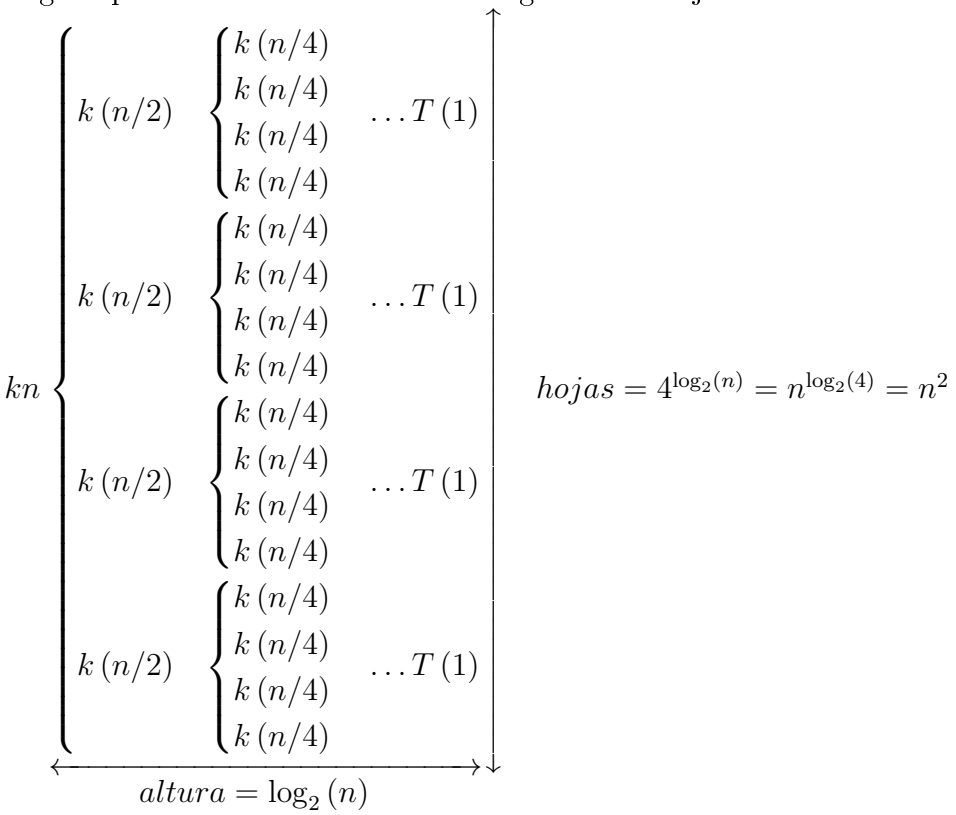
1.4.2. Arboles de recurrencia

Una técnica general para resolver recurrencias son lo árboles de recurrencia. Hacemos un árbol que en su raíz tiene el costo no recursivo para la entrada inicial y cada rama indica cada una de las llamadas recursivas. Luego expandimos las ramas correspondientes a las llamadas recursivas hasta llegar a las hojas.

Por ejemplo para $f(n) = 4f(n/2) + kn$ el primer paso nos da un arbol así:

$$kn \begin{cases} f(n/2) \\ f(n/2) \\ f(n/2) \\ f(n/2) \end{cases}$$

Luego expandimos cada rama hasta llegar a una hoja:



Finalmente sumamos cada nivel, y el costo total es la suma de todos los niveles:

$$T(n) = kn + 4k(n/2) + 4^2 k(n/4) + \ldots + n^2 T(1) = \sum_{i=0}^{\log_2(n)-1} \frac{4^i kn}{2^i} + n^2 T(1) =$$

$$= kn \sum_{i=0}^{\log_2(n)-1} 2^i + n^2 T(1) = kn \frac{1 - 2^{\log_2(n)}}{1-2} + n^2 T(1) = -kn(1-n) + n^2 T(1) =$$

$$= kn^2 - kn + n^2 T(1) \in \Theta\left(n^2\right)$$

Si somos prolijos, el método nos da una solución exacta. Si no, al menos tenemos un candidato para probar por inducción.

1.4.3. Regla de suavidad

En los análisis de algoritmos, frecuentemente aparecen funciones piso y techo: $\lfloor x/b \rfloor$ y $\lceil x/b \rceil$. Si en vez de considerar x cualesquiera solamente consideramos potencias de b, entonces podemos omitir dichas funciones de nuestro análisis pues para $n = b^k$ resulta $n/b = \lfloor x/b \rfloor = \lceil x/b \rceil$.

Sin embargo el resultado al que llegamos es solamente el comportamiento de la función para este tipo particular de entradas, pero podría ocurrir que para otro tipo de entradas no lo sea.

La regla de suavidad nos dice que si f es una función suave y g eventualmente no decreciente, si $b \geq 2$ entonces:

$$g\left(b^k\right) \in \Theta\left(f\left(b^k\right)\right) \Rightarrow g\left(n\right) \in \Theta\left(f\left(n\right)\right)$$

Es decir, si para nuestro análisis consideramos exclusivamente a potencias de b, podemos omitir pisos y techos; y si el resultado al que llegamos es una función suave, entonces nuestro análisis es correcto para cualquier entrada.

1.4.4. Teorema maestro

Las recurrencias que aparecen en los algoritmos «Divide & Conquer» usualmente tienen la forma $T\left(n\right) = aT\left(n/b\right) + f\left(n\right)$. En general podemos resolver este tipo de recurrencias mediante el siguiente teorema: Dados $a \geq 1$ y $b > 1$, entonces

$$T\left(n\right) \in \begin{cases} \Theta\left(n^{\log_b(a)}\right) & \text{si } \exists \epsilon > 0 / f\left(n\right) \in O\left(n^{\log_b(a-\epsilon)}\right) \\ \Theta\left(n^{\log_b(a)} \cdot \log_2\left(n\right)\right) & \text{si } f\left(n\right) \in \Theta\left(n^{\log_b(a)}\right) \\ \Theta\left(f\left(n\right)\right) & \text{si } \exists \epsilon > 0 / f\left(n\right) \in \Omega\left(n^{\log_b(a+\epsilon)}\right) \\ & \text{y } \exists c < 1, n_0 \in \mathbb{N} / \forall n > n_0 : af\left(n/b\right) \leq cf\left(n\right) \end{cases}$$

Si bien los tres casos son disjuntos, hay que tener en cuenta que no cubren todas las posibilidades.

1.4.5. Propiedades útiles

COMPLETAR

Capítulo 2

Tipos abstractos de datos

2.1. Introducción

La idea de un tito abstracto de datos es abstraer detalles de implementación. Es una descripción que debe indicar como se comporta la estructura sin indicar como esta programada. Dicha tarea sea trabajo del implementador, que debe proveer funciones que respeten el comportamiento esperado. El usuario es alguien que utiliza la abstracción asumiendo unicamente el comportamiento que el TAD describe.

Por ejemplo, una cola es una estructura en la cual podemos agregar elementos, quitar el primero de ellos, podemos preguntar si esta vacía y ademas existe una relación entre el orden en el que se agregan y quitan elementos. Como programadores podemos implementar este tipo con listas, arboles, arreglos o cualquier otro tipo de estructura; pero como usuarios no tenemos ninguna idea de como esta programada, solamente sabemos que respeta los puntos mencionados anteriormente.

Un TAD se compone de dos partes: por un lado debe proveer una lista de operaciones que soporta, junto con los tipos esperados; y por el otro una especificación algebraica o matemática sobre como deben comportarse.

Por ejemplo el TAD cola soporta las siguientes operaciones:

```
TAD Cola (a :: Set) where
  import Bool
  Vacia   :: Cola a
  Poner   :: a -> Cola a -> Cola a
  primero :: Cola a -> a
  sacar   :: Cola a -> Cola a
  esVacia :: Cola a -> Bool
```

2.2. Especificación

2.2.1. Especificación algebraica

Una especificación algebraica describe operaciones y ecuaciones entre operaciones. En general en una especificación algebraica los constructores de un TAD no se describen, pues solo cumplen la función de construir valores; pero en algunos casos los valores se construyen de acuerdo a algún comportamiento especifico.

Continuando con nuestro ejemplo de la cola, podemos describir el comportamiento deseado de la siguiente manera:

```
primero (poner x Vacia) = x
primero (poner x (poner y c)) = primero (poner y c)
sacar   (poner x Vacia) = Vacia
sacar   (poner x (poner y c)) = poner x (sacar (poner y c))
esVacia Vacia = True
esVacia (poner x c) = False
```

2.2.2. Especificación con modelo matemático

Una especificación matemática toma un modelo matemático conocido y describe los constructores y operaciones en términos del modelo elegido.

Si tomamos como modelos a las secuencias, podemos especificar las operaciones del TAD cola de la siguiente manera:

$$
\begin{array}{lll}
\texttt{vacia} & \equiv \langle\rangle & \\
\texttt{poner} & x\ \langle x_1, \ldots, x_n\rangle & \equiv \langle x, x_1, \ldots, x_n\rangle \\
\texttt{sacar} & \langle x_1, \ldots, x_n\rangle & \equiv \langle x_1, \ldots, x_{n-1}\rangle \\
\texttt{primero} & \langle x_1, \ldots, x_n\rangle & \equiv x_n \\
\texttt{esVacia} & \langle x_1, \ldots, x_n\rangle & \equiv \begin{cases} True & n = 0 \\ False & n > 0 \end{cases}
\end{array}
$$

2.3. Implementación

Es importante no confundir la especificación con la implementación. Tanto en la especificación algebraica como en la matemática, solo hemos descrito el comportamiento de las operaciones. Cada TAD admite diferentes implementaciones y cada una de ellas tendrá diferentes costos. Dependiendo del uso que le de el usuario al TAD, puede convenir una implementación u otra, por lo tanto es importante tener una especificación de costo para cada implementación.

Veremos a continuación dos implementaciones diferentes del TAD cola: una con listas y otra con pares de listas.

```
type Cola a = [a]
vacia = []
poner = (:)
primero [x] = x
primero (_:xs) = primero xs
sacar [x] = []
sacar (x:xs) = x : (sacar xs)
esVacia = null
```

- $W_{vacia} = S_{vacia} = \Theta(1)$.
- $W_{poner} = S_{poner} = \Theta(1)$.
- $W_{primero} = S_{primero} = \Theta(n)$.
- $W_{sacar} = S_{sacar} = \Theta(n)$.
- $W_{esVacia} = S_{esVacia} = \Theta(1)$.

Si implementamos colas con un par de listas `(xs,ys)` donde el orden de los elementos es `xs ++ reverse ys` y respetamos como invariante que si xs es vacía también lo es ys, podemos implementar colas de la siguiente manera:

```
type Cola a = ([a],[a])
vacia = ([],[])
poner x (ys,zs) = validar (ys, x:zs)
primero (x:_,_) = x
sacar (_:xs,ys) = validar (xs,ys)
esVacia = null . fst
validar (xs,ys) | null xs   = (reverse ys, [])
                | otherwise = (xs,ys)
```

- $W_{vacia} = S_{vacia} = \Theta(1)$.
- $W_{poner} = S_{poner} = \Theta(1)$.
- $W_{primero} = S_{primero} = \Theta(1)$.
- $W_{sacar} = S_{sacar} = O(n)$.
- $W_{esVacia} = S_{esVacia} = \Theta(1)$.

2.4. Verificación

Una implementación de un TAD es correcta si implementa las operaciones indicadas y dichas operaciones verifican la especificación.

En Haskell, el sistema te tipos asegura que los tipos de las operaciones son correctos, pero la verificación de la especificación debe hacerla manualmente el programador.

Para la primer implementación de colas, la verificación es la siguiente:

- `primero (poner x vacia)`

  $\equiv \langle def.vacia \rangle$

  `primero (poner x [])`

  $\equiv \langle def.poner \rangle$

  `primero (x:[])`

  $\equiv \langle def.primero.1 \rangle$

  `x`

- `primero (poner x (poner y c))`

  $\equiv \langle def.poner \rangle$

  `primero (x : (poner y c))`

  $\equiv \langle def.primero.2 \rangle$

  `primero (poner y c)`

- `esVacia vacia`

  $\equiv \langle def.vacia \rangle$

  `esVacia []`

  $\equiv \langle def.esVacia \rangle$

  `null []`

  $\equiv \langle def.null.1 \rangle$

  `True`

- `esVacia (poner x c)`

  $\equiv \langle def.poner \rangle$

  `esVacia (x:c)`

  $\equiv \langle def.esVacia \rangle$

  `null (x:c)`

  $\equiv \langle def.null.2 \rangle$

  `False`

- `sacar (poner x vacia)`

  $\equiv \langle def.vacia \rangle$

  `sacar (poner x [])`

  $\equiv \langle def.poner \rangle$

  `sacar [x]`

  $\equiv \langle def.sacar.1 \rangle$

  `[]`

  $\equiv \langle def.vacia \rangle$

  `vacia`

- `sacar (poner x (poner y c))`

  $\equiv \langle def.poner \rangle$

  `sacar (x:(poner y c))`

  $\equiv \langle def.sacar.2 \rangle$

  `x:(sacar (poner y c))`

  $\equiv \langle def.poner \rangle$

  `poner x (sacar (poner y c))`

2.5. Inducción

Los programas funcionales interesantes usan recursivo. Para poder probar propiedades acerca de programas recursivos usualmente es necesario utilizar el principio de inducción.

Para un tipo de datos recursivo, si lográramos demostrar que el hecho de que una propiedad valga en ciertos valores implica que también debe valer para los valores que podemos construir a partir de ellos y ademas; la propiedad también vale para cualquier constructor no recursivo, entonces naturalmente dicha propiedad debe valer para todos los valores de ese tipo de datos.

Por ejemplo el principio de inducción para listas es el siguiente:

Sea P una propiedad definida sobre listas,

- Si la propiedad vale en la lista vacía y ademas;
- el hecho de que la propiedad valga en una lista xs implica que también vale en $x : xs$

entonces la propiedad vale para cualquier lista.

Para arboles generales definidos como `data AGT a = Node a [AGT a]` el principio es como sigue:

Sea P una propiedad definida sobre arboles generales,

- Si la propiedad vale en `Node x []` y ademas;
- el hecho de que la propiedad valga para cada árbol general en xs implica que también vale en `Node x xs`

entonces la propiedad vale para cualquier árbol general.

A modo de ejemplo, para el tipo de datos `data BIN = Null | Leaf | Node BIN BIN` probaremos que `cantleaf t` $\leq$ `cantnode t + 1`

- Caso base `Null`:

  `canleaf Null`

  $\equiv \langle def.cantleaf.1 \rangle$

  `0`

  $\leq$

  `1`

  $=$

  `0+1`

  $\equiv \langle def.cantnode.2 \rangle$

  `cantnode Null + 1`

- Caso base `Leaf`:

  `canleaf Leaf`

  $\equiv \langle def.cantleaf.1 \rangle$

  `1`

  $\leq$

  `1`

  $=$

  `0+1`

  $\equiv \langle def.cantnode.2 \rangle$

  `cantnode Leaf + 1`

- Caso inductivo `Node u v`:

  `canleaf (Node u v)`

  $\equiv \langle def.cantleaf.3 \rangle$

  `cantleaf u + cantleaf v`

  $\leq \langle H.I. \rangle$

  `cantnode u + 1 + cantnode v + 1`

  $\equiv \langle def.cantnode.1 \rangle$

  `cantnode (Node u v) + 1`

Capítulo 3

Estructuras de datos

3.1. Árbol binario

Un árbol binario es una estructura donde cada nodo tiene exactamente dos hijos. En Haskell podemos representar dicho árbol con la siguiente definición: `data Bin a = E | N (Bin a) a (Bin a)`. Esto nos permite en cada paso de la recursión, dividir los datos en dos partes. Si no lo hacemos de forma inteligente, la estructura no nos provee ninguna ventaja.

Una función que determina si un elemento se encuentra en el árbol es como sigue:

```
member x E          = False
member x (N l y r) | x == y = True
                   | x /= y = let (a,b) member x l ||| member x r
                              in a || b
```

Si decidimos guardar los elementos en una sola rama del árbol, vemos que $W_{member}(n) \in O(n)$ y $S_{member}(n) \in O(n)$. Esto es igual que en una lista simplemente enlazada. Lo que ocurre aquí es que en cada paso de la recursion, estamos trabajando con un árbol de un solo elemento menos.

Una idea mas interesante es mantener el árbol balanceado, de manera tal de que en cada paso de la recursion trabajemos con un árbol de la mitad de elementos. En este caso logramos $W_{member}(n) \in O(n)$ y $S_{member}(n) \in O(\log_2(n))$, aunque para insertar elementos de forma eficiente necesitamos algún cambio en la estructura.

3.2. Árbol binario de búsqueda

Un árbol binario de búsqueda es un árbol binario que o bien es una hoja, o bien es un árbol donde la rama izquierda es un BST cuyos valores son menores o iguales al del árbol y los rama derecha es un BST cuyos valores son mayores.

Podemos reimplementar nuestra función para que sea mas eficiente, si tenemos en cuenta el invariante:

```
member x E          = False
member x (N l y r) | x == y = True
                   | x <  y = member x l
                   | x >  y = member x r
```

Insertar elementos en este tipo de arboles es sencillo: simplemente basta comparar el elemento a insertar con el nodo actual, y luego se inserta recursivamente en alguna de las ramas, hasta llegar a una hoja.

Para borrar elementos tenemos que prestar mas atención. En primer lugar recorremos el árbol hasta encontrarlo:

```
delete _ E          = E
delete x t@(N l y r) | x <  y = N (delete x l) y r
                     | x >  y = N l y (delete x r)
                     | x == y = delete' t where
```

Si el nodo donde esta el elemento no tiene ramas, devolvemos el árbol vacío, si tiene una sola rama la devolvemos:

```
delete _ E          = E
delete x t@(N l y r) | x <  y = N (delete x l) y r
                     | x >  y = N l y (delete x r)
                     | x == y = delete' t where
                       delete' (N E _ E) = E
                       delete' (N l _ E) = l
                       delete' (N E _ r) = r
```

Finalmente si nada de lo anterior ocurre no podemos dejar un hueco en el árbol: debemos buscar alguno de sus hijos que nos sirva como reemplazo y cambiarlo de lugar. Para no violar el invariante debemos buscar un elemento que sea mayor que todos los de la izquierda y menor que todos los de la derecha. Tanto el mayor de la izquierda como el menor de la derecha cumplen los requisitos:

```
delete _ E          = E
delete x t@(N l y r) | x <  y = N (delete x l) y r
                     | x >  y = N l y (delete x r)
                     | x == y = delete' t where
                       delete' (N E _ E) = E
                       delete' (N l _ E) = l
                       delete' (N E _ r) = r
                       delete' (N l _ r) = let z = minimum r
                                           in N l z (delete z r)
```

Aun cuando en un caso promedio hayamos mejorado la eficiencia, la complejidad sigue dependiendo del balance del árbol. Es por eso que para garantizar la mejoría, utilizamos otra estructura que nos facilita el balanceo al insertar.

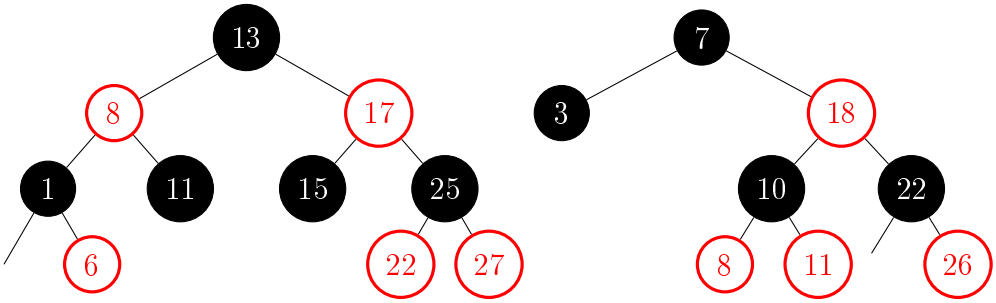
3.3. Árboles Red-Black

Un árbol Red-Black es un árbol binario de búsqueda con nodos «coloreados» que cumple los siguientes invariantes:

1. La raíz es negra.
2. Ningún nodo rojo tiene hijos rojos.
3. Todos los caminos de la raíz a una hoja tienen el mismo numero de nodos negros (altura negra).

En un RBT, el camino mas largo es a lo sumo el doble que el camino mas corto. El camino más corto posible tiene todos sus nodos negros, y el más largo alterna entre nodos rojos y negros. Esto significa que la altura está siempre en $O(\log_2(n))$.

Veamos algunos ejemplos:

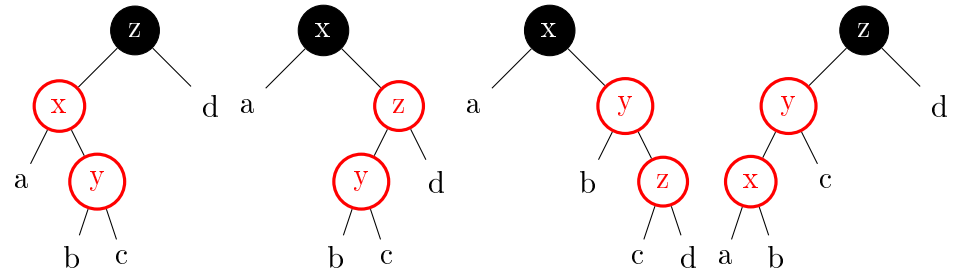


El código para buscar un elemento es el mismo que para arboles binarios de búsqueda, simplemente ignoramos el color.

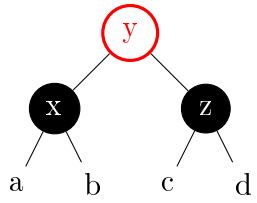
A la hora de insertar un nuevo elemento lo insertaremos siempre de color rojo, de esta manera nos garantizamos que seguimos cumpliendo el invariante 3. Sin embargo, si el nodo que insertamos tiene un padre rojo, estaríamos violando el invariante 2. Es por ello que luego de insertarlo debemos rebalancear el árbol. Luego del rebalanceo la raíz podría quedar roja, por lo tanto hay que volver a pintarla.

```
insert x = makeBlack (ins x t) where
  ins x E           = N R E x E
  ins x (N c l y r) | x <  y = balance c (ins x l) y r
                    | x >  y = balance c l y (ins x r)
                    | x == y = N c l y r
```

Pensemos ahora como hacer el rebalanceo. La única violación puede ocurrir en el nodo que acabamos de insertar y ocurrirá en un camino Black-Red-Red:



En todos los casos la solución es la misma: reescribir el nodo como un padre rojo con dos hijos negros:



La implementación en Haskell es trivial:

```
balance B (N R (N R a x b) y c) z d = N R (N B a x b) y (N B c z d)
balance B (N R a x (N R b y c)) z d = N R (N B a x b) y (N B c z d)
balance B a x (N R (N R b y c) z d) = N R (N B a x b) y (N B c z d)
balance B a x (N R b y (N R c z d)) = N R (N B a x b) y (N B c z d)
balance c l a r = N c l a r
```

Con esta estructura hemos conseguido $W_{insert} \in O(\log_2(n))$.

3.4. Leftist Heaps

Los heaps (o montículos) son árboles que permiten un acceso eficiente al mínimo elemento. Mantienen el invariante de que todo nodo es menor a todos los valores de sus hijos, por lo tanto el mínimo siempre esta en la raíz.

Un leftist heap es un heap donde el rango de cualquier hijo izquierdo es mayor o igual que el de su hermano derecho, siendo el rango de un heap la longitud del camino hacia la derecha hasta un nodo vacío (espina).

Como consecuencia del invariante la espina derecha es el camino mas corto a un nodo vacío, y ademas la longitud de la espina derecha es a lo sumo $\log_2(n+1)$.

La operación mas importante en un leftist heap es fusionarlos. Para ello, si alguno de los heaps es vacío, simplemente devolvemos el otro.

De lo contrario observamos cual de las raíces es la menor y fusionamos su rama derecha con el otro heap para mantener el invariante.

De esta manera obtenemos dos subheaps que ubicaremos a izquierda o derecha según su rango:

```
merge h1 E = h1
merge E h2 = h2
merge h1@(N _ l1 x r1) h2@(N _ l2 y r2) | x <= y = makeH l1 x (merge r1 h2)
                                       | x >  y = makeH l2 y (merge h1 r2)

makeH r a b | rank a >= rank b = N (rank b + 1) r a b
            | otherwise        = N (rank a + 1) r b a
```

Como la espina derecha es a lo sumo logarítmica, $W_{merge} \in O(\log_2(n))$.

3.5. Secuencias