ESTRUCTURAS DE DATOS Y ALGORITMOS II

Bautista José Peirone

2023

Resumen

Contents

Introducción

Estructuras de Datos y Algoritmos II es una materia del primer cuatrimestre del 3er año de la Licenciatura en Ciencias de la Computación de la FCEIA - UNR. Este apunte aborda los temas dados a lo largo del cursado de 2023, plan de estudios 2010.

En la asignatura se tratan diversos temas, desde el análisis asintótico, pasando por la programación funcional y las estructuras de datos persistentes, hasta llegar a estudios de trabajo y profundidad de algoritmos. La asignatura se divide en 5 unidades, algunas con un enfoque más en el análisis matemático, y otras cuyo hincapié reside en pseudocódigo para hacer pruebas y análisis de programas.

El lenguaje de programación usado en la materia es Haskell, un lenguaje funcional puro, sobre el cual implementaremos diversos programas haciendo uso de características como memory sharing, pattern matching, currification, entre otras características del lenguaje. El pseudocódigo dado en la materia presenta una gran similitud a Haskell, y muchas veces en la práctica variamos entre uno y otro según sea conveniente. En este apunte se usan ambos, debido a que dependiendo del contexto y el tema suele ser conveniente uno u otro.

1 Unidad I - Análisis Asintótico y Modelo de Costos

1.1 Motivaciones

Para poder dar un análisis riguroso de un programa en términos de energía, tiempo y/o memoria, debemos establecer un método formal para poder medir estos. No es suficiente con medir estas magnitudes físicas, ya que estos factores dependerán de tanto el hardware que los corre como el lenguaje de programación sobre el cual el programa está escrito. Esto nos daría lugar a comparaciones injustas e incorrectas, por lo que, como herramienta matemática se decide usar el análisis asintótico.

1.2 Análisis asintótico

El análisis asintótico es un método para analizar y comparar las velocidades de crecimientos de funciones.

Notación O
Sean $f, g : \mathbb{N} \to \mathbb{R}$, decimos que f tiene orden de crecimiento $O(g)$, y se simboliza $f \in O(g)$, si:

$$\exists n_0 \in \mathbb{N}\, \exists c \in \mathbb{R}^+ : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)$$

La notación O da una cota superior para el crecimiento de la función f.

Notación Ω
Sean $f, g : \mathbb{N} \to \mathbb{R}$, decimos que f tiene orden de crecimiento $\Omega(g)$, y se simboliza $f \in \Omega(g)$, si:

$$\exists n_0 \in \mathbb{N}\, \exists c \in \mathbb{R}^+ : \forall n \geq n_0, 0 \leq cg(n) \leq f(n)$$

La notación Ω da una cota inferior para el crecimiento de la función f.

Notación Θ
Sean $f, g : \mathbb{N} \to \mathbb{R}$, decimos que f tiene orden de crecimiento $\theta(g)$, y se simboliza $f \in \Theta(g)$, si: $f \in O(g)$ y $g \in O(f)$.
La notación Θ da una cota ajustada para el crecimiento de la función f.

NOTA: Si bien $O(g(n)) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists n_0 \in \mathbb{N}\, \exists c \in \mathbb{R}^+ : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$, a menudo usaremos la misma notación como una expresión algebraica, ej. $T(n) = T(n-1) + O(n)$, para denotar cualquier función en este conjunto. En el caso del ejemplo, $T(n)$ es una función con una llamada recursiva acompañada por un termino lineal.
Lo mismo aplica para la notación Ω y Θ.

1.3 Modelo de costos

Podemos ahora hablar de un modelo de costos, esto es, un modelo que formalice los costos de un programa o algoritmo. Para esto, nos centramos en dos costos en particular:

1. Trabajo (W)
   El trabajo representa el costo secuencial de un programa, esto es, el costo de si se ejecuta con un solo procesador. El trabajo también puede ser pensado como el computo total que hace el programa.

2. Profundidad (S)
   La profundidad representa el costo paralelo de un programa, esto es, el costo de si se ejecuta con infinitos procesadores. Si bien esto es un requerimiento imposible de cumplir, en realidad nos basta con que haya disponibilidad de procesadores siempre que se necesite. La profundidad puede pensarse como la máxima cantidad de computo que se hará en un rama de ejecución del programa. La profundidad también puede entenderse como una medida de la dependencia entre los cómputos del programa, ya que cuantos mas alta es esta dependencia, menos cálculos pueden realizarse en simultáneo, dando una mayor profundidad.

Ejemplo: Se tienen n números a ser sumados ¿Cómo se puede resolver? La idea más directa y sencilla es sumar los primeros $n-1$ números recursivamente, y a ese resultado sumarle el último. Esto nos daría que $W(n) \in O(n)$ y $S(n) \in O(n)$, ya que realmente no podemos dividir en ningún momento el computo en varios procesadores.

Esto se puede mejorar si decidimos dividir la lista de números a la mitad (una mitad con un elemento extra en caso de n impar), realizar la suma de cada mitad en paralelo y luego sumar ambos valores. Esto, como ya lo veremos más adelante, nos resulta en $W(n) \in O(n)$ y $S(n) \in O(\log n)$.

NOTA: La idea tras esta ultima implementación cumple con la cota para la profundidad siempre y cuando la estructura de datos donde se almacenan los números cumpla algunas condiciones sobre los tiempos para dividir la estructura a la mitad. Esto se verá en la última unidad.

Introducimos ahora el concepto de paralelismo.

$$P = \frac{W}{S}$$

El paralelismo es una medida de cuantos procesadores se pueden usar de forma eficiente. En el ejemplo anterior, para la suma de numeros:

$$P \leq \frac{kn}{k' \log n} \in O\left(\frac{n}{\log n}\right)$$

El criterio que usaremos para elegir entre varias implementaciones de un algoritmo será escoger aquella que mayor paralelismo brinde.

1.4 Scheduler

¿Qué es un scheduler? Un scheduler es una pieza de software del sistema operativo encargada de repartir y asignar a cada proceso el hardware correspondiente para poder correr en una computadora. Más concretamente, asigna a cada tarea pendiente un procesador para que este se encargue de llevarla a cabo. Si bien esto no es en sí parte de la asignatura, nos basaremos en ciertos principios de los schedulers de nuestros sistemas operativos.

Decimos que un scheduler es voraz cuando, cada vez que haya un procesador libre y una tarea pendiente, el scheduler inmediatamente asigna dicha tarea al procesador. Cuando un scheduler es voraz, se cumple lo que se conoce como el Principio del Scheduler Voraz (Brent), que dice que en una maquina con un scheduler voraz, el tiempo T de ejecución de un programa con trabajo W, profundidad S y p procesadores cumple:

$$T \leq \frac{W}{p} + S$$

Esto es considerado una buena cota para el tiempo, y además, podemos reformular la desigualdad para obtener:

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

Entonces, si $p \ll P$, al tener $\frac{p}{P} \to 0$ obtenemos una cota óptima para el tiempo.
Nuevamente, estas cotas valen bajo los supuestos de un scheduler voraz y de baja latencia de comunicación entre procesadores o en red, los cuales son temas que no entran en el foco de la materia, por lo que no les daremos importancia.

1.5 Pseudocódigo y costos de programas

Como se mencionó anteriormente, se usará un modelo de costos basado en el análisis asintótico para expresar los costos de nuestros programas, ya que solo nos interesan cotas para estos y no los valores exactos. Además, queremos que nuestro modelo se abstraiga del hardware y lenguaje sobre los cuales funcionan nuestros programas. Para esto usaremos un pseudocódigo, y daremos junto con él la forma en que calcularemos los costos de un programa escrito en este pseudocódigo.

Nuestro pseudocódigo se conformará de la siguiente manera:

1. Constantes: `0`, `1`, `2`, `True`, `False`, `[]`, `'a'`, `'b'`
2. Operadores: `+`, `*`, `&&`, `<`, `if then else`
3. Pares ordinarios y paralelos: `(3, 4)`, `(3 + 4 || 5 + 6)`

4. Expresiones `let` (nombres locales): `let x = 3 + 4 in x + x`

5. Secuencias: `[ x * 2 | x <- xs ]`

Además, indicaremos con `f: a` que el tipo de `f` es `a`. Al usar Haskell el símbolo para el operador de paralelización será `|||`.

Una vez definido esto, damos el cálculo de trabajo y profundidad:

$$
\begin{aligned}
W(c) &= 1, \text{con } c \text{ constante} \\
W(op\, e) &= 1 + W(e) \\
W((e_1, e_2)) &= 1 + W(e_1) + W(e_2) \\
W((e_1 \,||\, e_2)) &= 1 + W(e_1) + W(e_2) \\
W(\texttt{let } x = e1 \texttt{ in } e2) &= 1 + W(e1) + W(e2(x \mapsto \text{Eval}(e1))) \\
W([f(x) \,|\, x \leftarrow xs]) &= 1 + \sum_{x \in xs} W(f(x))
\end{aligned}
$$

$$
\begin{aligned}
S(c) &= 1, \text{con } c \text{ constante} \\
S(op\, e) &= 1 + S(e) \\
S((e_1, e_2)) &= 1 + S(e_1) + S(e_2) \\
S((e_1 \,||\, e_2)) &= 1 + \max\left(S(e_1), S(e_2)\right) \\
S(\texttt{let } x = e1 \texttt{ in } e2) &= 1 + S(e1) + S(e2(x \mapsto \text{Eval}(e1))) \\
S([f(x) \,|\, x \leftarrow xs]) &= 1 + \max_{x \in xs} S(f(x))
\end{aligned}
$$

NOTA: $e_2(x \mapsto \text{Eval}(e_1))$ representa reemplazar todas las ocurrencias de x en e_2 por la evaluación de la expresión e_1.

1.6 Divide and Conquer

Divide and Conquer es una estrategia de resolución de problemas muy útil cuando se tienen problemas cuya solución se puede plantear en terminos de una solución del mismo problema pero de tamaño más chico. Por su naturaleza, Divide and Conquer es fácil de plantear mediante recursión, y se compone de dos partes:

1. Caso base: El tamaño del problema es chico, y se puede dar una solución particular y específica para esta instancia del problema

2. Caso recursivo: Se divide el problema en subproblemas, se resuelve cada problema recursivamente, y por último se combinan todas las soluciones en una solución al problema general.

Bajo esta estructura, podemos plantear el trabajo y profundidad de una función que use Divide and Conquer para un problema de tamaño n:

$$W(n) = W_{\text{dividir}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combinar}}(n)$$

$$S(n) = S_{\text{dividir}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combinar}}(n)$$

Ejemplo: El algoritmo de ordenación mergesort es un ejemplo de aplicación de la estrategia Divide and Conquer.

```
msort : [Int] -> [Int]
msort [] = []
msort [x]= [x]
msort xs = let
             (ls ,rs ) = split xs
             (ls',rs') = (msort ls || msort rs)
           in
             merge (ls',rs')

split : [Int] -> ([Int],[Int])
split [] = ([],[])
split [x] = ([x],[])
split (x:y:zs) = let
                   (xs,ys) = split zs
                 in
                   (x:xs,y:ys)

merge : ([Int],[Int]) -> [Int]
merge ( [], ys) = ys
merge ( xs, []) = xs
merge (x:xs,y:ys) = if x <= y
                    then x:merge(xs,y:ys)
                    else y:merge(x:xs,ys)
```

Del anterior código podemos plantear las recurrencias de trabajo, donde n es la longitud de la lista:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = W_{split}(n) + 2W_{msort}(n/2) + W_{merge}(n) + c_2, n > 1$$

$$W_{split}(0) = c_3$$
$$W_{split}(1) = c_4$$
$$W_{split}(n) = W_{split}(n-2) + c_5, n > 1$$

$$W_{merge}(0) = c_6$$
$$W_{merge}(n) = W_{merge}(n-1) + c_7, n > 1$$

y en este último caso, n es la suma de las longitudes de las listas.

Podemos deducir, y veremos adelante como mostrarlo formalmente, que $W_{merge}(n), W_{split}(n) \in O(n)$, y se obtiene, aunque más difícil de ver, que $W_{msort}(n) \in O(n \log n)$.

De forma similar, las recurrencias de profundidades quedan:

$$S_{msort}(0) = k_0$$
$$S_{msort}(1) = k_1$$
$$S_{msort}(n) = S_{split}(n) + S_{msort}(n/2) + S_{merge}(n) + k_2, n > 1$$

$$S_{split}(0) = k_3$$
$$S_{split}(1) = k_4$$
$$S_{split}(n) = S_{split}(n-2) + k_5, n > 1$$

$$S_{merge}(0) = k_6$$
$$S_{merge}(n) = S_{merge}(n-1) + k_7, n > 1$$

Y concluimos que $S_{merge}(n), S_{split}(n) \in O(n)$, y luego $S_{msort}(n) \in O(n)$. Más adelante veremos como mejorar la profundidad del mergesort con el uso de otras estructuras de datos.

2 Unidad II - Recurrencias

En la Unidad I planteamos los costos de funciones en términos de recurrencias, ¿pero qué son estas? Una recurrencia es una función definida en términos de sí misma, osea, una función definida recursivamente. Lo ideal sería poder obtener una ley exacta para una recurrencia, por ejemplo, dada la siguiente recurrencia definida para todo $n \in \mathbb{N}$:

$$f(0) = 1,\ f(n) = 2f(n-1),\ \text{si} n > 1$$

podemos ver fácilmente que $f(n) = 2^n\ \forall n \in \mathbb{N}$.

Sin embargo, esto no siempre es tan fácil, y la mayoría de las veces no llegaremos a una expresión cerrada para una recurrencia. Esto no es problema, ya que nuestro objetivo es dar cotas para estas recurrencias, y a continuación se explican algunos métodos para poder hacer esto.

2.1 Método de substitución

Este método sirve para probar que una recurrencia cumple una cota. Osea que debemos tener alguna idea de una cota que pueda valer, ya sea por intuición, adivinando, o mediante el árbol de recurrencias (próximo apartado).

Una vez que se tiene una cota, la idea es probar que la cota se verifica mediante el uso de inducción matemática sobre los números naturales. Veamos este método aplicado en el ejemplo de la recurrencia de trabajo de mergesort. Recordemos

$$\begin{aligned} W(0) &= c_0 \\ W(1) &= c_1 \\ W(n) &= 2W(\lfloor n/2 \rfloor) + c_2 n, n > 1 \end{aligned}$$

Supongamos que sospechamos que vale $W(n) \in O(n \log n)$. Para probarlo, por definición de O tenemos que probar que $\exists n_0 \in \mathbb{N}$, $\exists c \in \mathbb{R}^+ | \forall n \geq n_0, 0 \leq W(n) \leq cn \log n$

Hacemos entonces la prueba:

Caso inductivo: Tomamos $\exists c \in \mathbb{R}^+$, $0 \leq W(k) \leq ck \log k$, $k < n$ como la Hipótesis Inductiva.

$$\begin{aligned} W(n) &= 2W(\lfloor n/2 \rfloor) + c_2 n \\ &\leq 2c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + c_2 n && \text{(H.I.)} \\ &\leq 2c \frac{n}{2} \log\left(\frac{n}{2}\right) + c_2 n \\ &= cn(\log n - \log 2) + c_2 n \\ &= cn \log n + (c_2 n - cn) \\ &= cn \log n && \text{si } c \geq c_2 \end{aligned}$$

Luego, si tomamos $c \geq c_2$ podemos ver que el caso inductivo vale.

Caso base: Queremos ver ahora si $W(0) \leq c\,0\log 0$ y $W(1) \leq c\log 1$ Sin embargo $c\,0\log 0$ no está definido y $c\log 1 = 0$, luego no vale para estos casos base. Recordemos que la notación O solo nos pide que la desigualdad comience a valer a partir de un n_0, luego podemos probar a partir de 2. Notemos que todos los casos recursivos se reducirán en algun punto a $W(2)$ o a $W(3)$, luego escogiendo c tal que la desigualdad vale en estos dos casos tendremos la prueba hecha.
$W(2) = 2W(1) + 2c_2 = 2(c_1 + c_2) \leq c\,2\log 2 = 2c$
$W(3) = 2W(1) + 3c_2 = 2c_1 + 3c_2 \leq c\,3\log 3$
Finalmente, y para concluir, hemos obtenido que si $c \geq c_2$, $2c \geq 2(c_1 + c_2)$ y $3c\log 3 \geq 2c_1 + 3c_2$, la prueba vale, luego tomando $c \geq \max\left\{c_2, c_1 + c_2, \frac{2c_1+3c_2}{3\log 3}\right\}$ concluimos que $\forall n \geq 2,\ W(n) \leq cn\log n \implies W(n) \in O(n\log n)$ □

Muchas veces, para poder terminar la prueba, requeriremos de desigualdades que nos den cotas más ajustadas, y la hipótesis inductiva es candidata a proporcionarnos estas mejores cotas, a cambio de usar una hipótesis inductiva más específica. Para esto, en las pruebas en este método podemos agregar a la función con que acotamos términos de orden menor. Por ejemplo, para la prueba de recién podríamos haber usado funciones como $cn\log n + mn$, $cn\log n - h$, o $cn\log n - mn + h$.

2.2 Árboles de recurrencias

La técnica del árbol de recurrencias es útil para hallar estimaciones o posibles cotas a una recurrencia. Este método consiste en dibujar un árbol, donde en el nodo raíz se tiene el costo de la recurrencia en n, y una rama por cada llamada recursiva. Cada rama de una llamada recursiva a su vez se expande en su propio árbol, y se procede así hasta llegar a un caso base, en el cual se deja de expandir el árbol. Por cada nivel del árbol se suma el total de operaciones, y estas a su vez se suman para dar el costo total de la recurrencia.

En la figura 1 se ve en un ejemplo el árbol de recurrencia de $T(n) = 2T(n/3) + cn^2$. La suma de todos los niveles luego es

$$\begin{aligned}
&cn^2 + \frac{2}{9}cn^2 + \frac{4}{27}cn^2 + \ldots + \left(\frac{2}{9}\right)^{\log_3 n-1} cn^2 + n^{\log_3 2}\cdot T(1)\\
&= cn^2 \sum_{k=0}^{\log_3 n-1} \left(\frac{2}{9}\right)^k + n^{\log_3 2}\cdot T(1)\\
&= cn^2 \frac{(2/9)^{\log_3 n} - 1}{(2/9) - 1} + n^{\log_3 2}\cdot T(1) \in \Theta(n^2)
\end{aligned}$$

ya que $\frac{(2/9)^{\log_3 n}-1}{(2/9)-1} \in O(1)$.

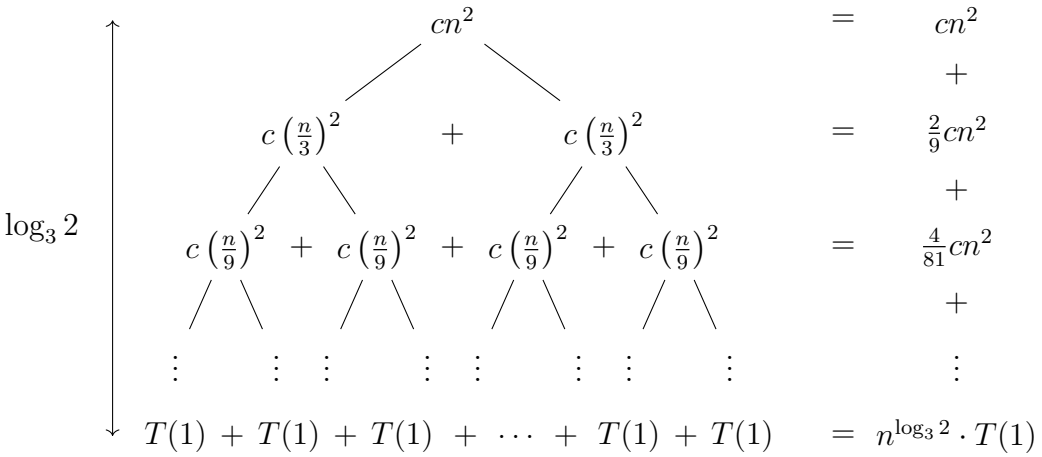


Figura 1: Árbol de recurrencia para $T(n) = 2T(n/3) + cn^2$

Este método no lo usaremos por lo general para demostrar una cota de una recurrencia, sino más bien para obtener cierta confianza en que dicha cota puede valer.

2.3 Funciones suaves y regla de suavidad

Si recurrimos al planteo que hicimos en la primera unidad sobre el trabajo de mergesort, podremos notar que se omitieron $\lfloor\rfloor$ y $\lceil\rceil$, siendo que ambos corresponden ya que n podría ser impar, quedando una mitad de la lista de longitud $\lfloor n/2 \rfloor$ y la otra de longitud $\lceil n/2 \rceil$. ¿Hay algún motivo por el cuál hayamos podido hacer esto? Si la respuesta es sí, ¿cuando se pueden hacer estas aproximaciones y/o simplificaciones?

Para dar solución a esta pregunta, se darán algunas definiciones a continuación:
Función eventualmente no decreciente
Una función $f : \mathbb{N} \to \mathbb{R}^+$ es eventualmente no decreciente sii

$$\exists N \in \mathbb{N}.\, f(n) \leq f(n+1) \forall n \geq N$$

Función b-suave (b-smooth)
Una función $f : \mathbb{N} \to \mathbb{R}^+$ es b-suave si

1. f es eventualmente no decreciente
2. $f(bn) \in O\left(f(n)\right)$

Función suave
Una función es suave si es b-suave para todo $b \in \mathbb{N}$

Propiedad de suavidad: Si f es b-suave para $b \geq 2$, entonces es suave.
Demostración: Supongamos que f es b-suave para $b \geq 2$, entonces sabemos que f es eventualmente no decreciente y además existen n_0 y k tales que para $n \geq n_0$, $f(bn) \leq kf(n)$.

Sea $c \in \mathbb{N}$, nos interesa ver que f es c-suave. Claramente f es eventualemente no decreciente, y además si $c \leq b$, $f(cn) \leq f(bn) \leq kf(n)$, luego $f(cn) \in O(f(n))$. Ahora, para el caso $c > b$, sea $m = \min\{j \in \mathbb{N} \mid b^j \geq c\}$, luego para n suficientemente grande se tiene $f(cn) \leq f(b^m n) \leq f(b(b^{m-1}n)) \leq kf(b^{m-1}n) \leq \cdots \leq k^m f(n)$, y surge de aquí inmediato que $f(cn) \in O(f(n))$, y concluimos que f es c-suave.
$\therefore f$ es suave para todo $c \implies f$ es suave. □

Ejemplos: n^2, n^k (para todo k), $n \log n$ son funciones suaves.
Por otro lado, $n^{\log n}, 2^n, n!$ son funciones no suaves.

Regla de suavidad:

Sea f suave y g eventualmente no decreciente, entonces para $b \geq 2$:
$g(b^k) \in \Theta\left(f(b^k)\right) \implies g(n) \in \Theta(f(n))$

Podemos ahora volver al caso del mergesort. La verdadera recurrencia que surge del trabajo del mergesort es, dada su definición en pseudocódigo:

$$W_{msort}(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 & \text{si } n = 1 \\ W_{msort}(\lfloor n/2 \rfloor) + W_{msort}(\lceil n/2 \rceil) + W_{split}(n) + W_{merge}(n) & \text{en otro caso} \end{cases}$$

Notemos que si fuese el caso en que $n = 2^k$, $k \in \mathbb{N}$, entonces sucedería que $W_{msort}(n) = W_{msort}(\lfloor n/2 \rfloor) + W_{msort}(\lceil n/2 \rceil) + W_{split}(n) + W_{merge}(n) = 2W_{msort}(n/2) + W_{split}(n) + W_{merge}(n) = 2W_{msort}(n/2) + O(n) \in O(n \log n)$. Luego, como $n \log n$ es suave y $W_{msort}(n)$ es eventualmente no decreciente (por el Lema que sigue), por la regla de suavidad podemos concluir que $W_{msort}(n) \in O(n \log n)$ para $n \in \mathbb{N}$.

Lema: $W_{msort}(n)$ es eventualmente no decreciente
Demostración: Probaremos por inducción sobre los naturales.
Suponemos como hipótesis inductiva que vale $W_{msort}(k) \leq W_{msort}(k+1)$, $k+1 \leq n$. Luego,

$$\begin{aligned} W_{msort}(n) &= W_{msort}(\lfloor n/2 \rfloor) + W_{msort}(\lceil n/2 \rceil) + O(n) \\ &\leq W_{msort}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + W_{msort}(\lceil n/2 \rceil) + O(n) \qquad (1) \\ &\leq W_{msort}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + W_{msort}\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + O(n) \qquad (2) \\ &= W_{msort}(n+1) \end{aligned}$$

donde (1) corresponde con, $\lfloor n/2 \rfloor = \left\lfloor \frac{n+1}{2} \right\rfloor$ o $\lfloor n/2 \rfloor + 1 = \left\lfloor \frac{n+1}{2} \right\rfloor$, luego $W_{msort}(\lfloor n/2 \rfloor) = W_{msort}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$ o , por la H.I., $W_{msort}(\lfloor n/2 \rfloor) \leq W_{msort}(\lfloor n/2 \rfloor + 1) = W_{msort}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$. En

cualquier caso $W_{msort}(\lfloor n/2 \rfloor) \leq W_{msort}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$.
La justificación de (2) es equivalente pero para $\lceil\ \rceil$.

Ahora, como caso base, vemos que $W_{msort}(2) \geq W_{msort}(1)$, luego podemos concluir que $W_{msort}(n)$ es eventualmente no decreciente. □

La regla de suavidad da entonces respuesta a por qué pudimos simplificar la recurrencia de mergesort en la primera unidad, olvidándonos tanto de techos como pisos en esta. En general, podremos hacer esta excepción, y a menudo no los escribiremos para hacer mas general una recurrencia dada. Además, las constantes de los casos bases tampoco suelen afectar el resultado, por lo tanto también se omitirán.
Con estas reglas de notación, la recurrencia de trabajo de mergesort puede expresarse de forma simple como:

$$W(n) = 2W(n/2) + W_{split}(n) + W_{merge}(n)$$

2.4 Teorema Maestro

Existen miles de tipos de recurrencias, y cada una presenta una sus propias complicaciones a la hora de acotarlas. Sin embargo, existe un tipo particular de recurrencias de interés especial para nosotros, y son aquellas recurrencias de trabajo y profundidad que surgen de los algoritmos Divide and Conquer debido al importante uso que tiene esta estrategia. Debido a la forma que presentan los programas con dicha estrategia, las recurrencias que se suelen obtener ya sea para trabajo o profundidad tienen la siguiente forma:

$$T(n) = aT(n/b) + f(n)$$

Existe un teorema que nos permite fácilmente encontrar cotas Θ para algunas (no todas) las recurrencias de esta forma. Dicho teorema es llamado el Teorema Maestro, y plantea que para una recurrencia $T(n)$ como la anterior planteada, para $a \geq 1$ y $b > 1$, entonces

$$T(n) \in \begin{cases} \Theta(n^{\lg_b a}) & \text{si } \exists \epsilon > 0,\ f(n) \in O\left(n^{(\lg_b a)-\epsilon}\right) \\ \Theta(n^{\lg_b a} \lg n) & \text{si } f(n) \in \Theta(n^{\lg_b a}) \\ \Theta(f(n)) & \text{si } \exists \epsilon > 0,\ f(n) \in \Omega(n^{(\lg_b a)+\epsilon}) \wedge \exists N \in \mathbb{N},\ c < 1 \\ & \qquad \forall n \geq N,\ af(n/b) \leq cf(n) \end{cases}$$

Cabe aclarar que este teorema **NO** cubre todas las posibilidades, luego puede haber casos donde no se logre aplicar este teorema.
La idea detrás de lo que el teorema expresa es que aquel término que domine de entre $n^{\lg_b a}$ y $f(n)$ es quien determina el orden asintótico de la recurrencia. Más aun, no solo debe ser una más grande que la otra, sino que deben ser polinomialmente más grandes.

3 Unidad III - Haskell y Estructuras Persistentes

Haskell es un lenguaje de programación funcional puro basado en el λ-cálculo, y no en modelos de máquinas.
Un lenguaje funcional puro es aquel que no posee efectos colaterales, esto es, una llamada a una función con los mismos argumentos producirá siempre el mismo resultado. Esto se logra ocultando al programador el estado de memoria del programa en todo momento, y dando lugar a programar mediante la evaluación pura y exclusivamente de expresiones. Además, entre otra de las características notables de Haskell encontramos que todos los valores con que se trabaja en este son inmutables, lo cual da gran importancia a lo que se conoce como estructuras persistentes, abordado más adelante en esta unidad.

La unidad 3 cuenta con una introducción a la sintaxis básica de Haskell, su sistema de tipos y ejemplos de uso del lenguaje. Dichos temas no se tratan en este resumen, ya que tienen un enfoque más orientado a la práctica, y además están suficientemente explayados en las presentaciones de la materia.

3.1 Funcional vs. Imperativo

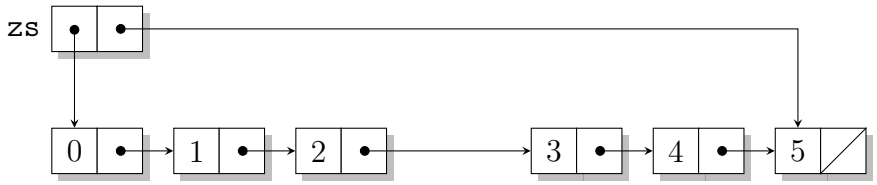
Gran parte de los algoritmos que se usan tradicionalmente están pensados para estructuras efímeras. ¿Qué es una estructura efímera? Una estructura efímera es aquella que soporta una sola versión de sí misma, y donde los cambios a dicha estructura son destructivos. Consideremos una implementación de listas enlazadas en C, un lenguaje imperativo. Si queremos modificar el primer elemento de la lista, es tan sencillo como acceder a la región de memoria de dicho nodo y modificarlo, pero la estructura original se habrá perdido debido al cambio. Estas estructuras están fuertemente relacionadas a programación secuencial e imperativa, sin embargo no es este tipo de programación la que queremos usar.
La programación funcional hace uso de otro modelo de estructuras, las estructuras persistentes o inmutables. Estas estructuras son muy ventajosas en cuanto a paralelización respecta ya que podemos estar seguros de que dicha estructura no sufrirá cambios, además de que pueden soportar varias versiones de si mismas en simultaneo.
El uso de estas estructuras tiene dos desventajas principales, en primer lugar, todos los algoritmos previamente diseñados deben ser adaptados a este nuevo tipo de estructuras (en caso de ser posible), y en segundo lugar, el uso de estas puede incurrir en un overhead de memoria o incluso cotas asintóticas que no podremos alcanzar.
En un lenguaje puro, todas las estructuras deberían ser inmutables, y la idea general es que al realizar un cambio sobre estas, la memoria sea duplicada y sea esta copia la que es modificada. Para evitar grandes desperdicios de memoria, lenguajes como Haskell implementan algo llamado memory sharing, lo cual permite reusar partes de una estructura en sus versiones modificadas, y así evitar la redundancia en copiado de memoria. En casos como estos, el garbage collection (manejo automático de la memoria) es esencial, ya que será dificil para

el programador determinar cuando un dato puede ser descartado de la memoria, además de que provee un nivel más de abstracción.

El caso de las listas enlazadas es un buen ejemplo nuevamente, para, por ejemplo, la operación de concatenación.

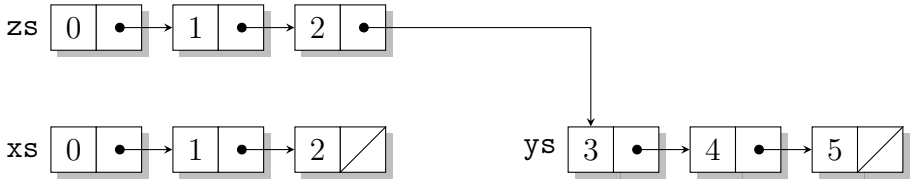
1. Listas enlazadas efímeras:
   En listas efímeras, dadas las listas `xs` y `ys`, tenemos que `zs = xs ++ ys` tiene un costo $O(1)$, pero destruye las primeras dos listas.

   Luego de la operación pasa a ser:



2. Listas enlazadas persistentes:
   Por otro lado, en el caso de las listas persistentes, `zs = xs ++ ys` tiene un costo $O(|xs|)$, y requiere duplicar la lista `xs`, pero tras la operación las tres listas son validas.

   Y luego de la operación tenemos:



Usaremos memory sharing y las estructuras de datos persistentes a lo largo de todos los algoritmos que se verán.

3.2 Arboles binarios

Un árbol binario es un arbol en el que cada nodo tiene exactamente dos hijos. En Haskell, un árbol binario puede ser representado mediante el tipo de dato

```
data Bin a = Hoja | Nodo (Bin a) a (Bin a)
```

Algunos ejemplos de funciones elementales para árboles binarios son:

```
-- Determina si un dato está en un árbol binario
member :: Eq a => a -> Bin a -> Bool
member a Hoja = False
member a (Nodo l b r) = (a == b) || (member a l) || (member a r)

-- Recorrido inorder de un arbol binario
inorder :: Bin a -> [a]
inorder Hoja = [ ]
inorder (Nodo l a r) = inorder l ++ [a] ++ inorder r
```

Usando este código, $W_{member}(a, t)$, $S_{member}(a, t) \in O(n)$ donde n es la cantidad de nodos de t. Usando el operador de paralelismo se podrían paralelizar ambas llamadas recursivas, obteniendo $S_{member}(a, t) \in O(h)$ donde h es la altura de t. El motivo por el cual no se usa es porque todavía no lo hemos introducido.

Una posible variación a los arboles binarios son los arboles binarios de busqueda (BST). Un árbol binario de búsqueda es un árbol binario t tal que

- t es una hoja
- t es un Nodo l a r, y se cumple que:
  - l y r son árboles binarios de búsqueda
  - si y es una clave en algún nodo de l entonces $y \leq a$.
  - si y es una clave en algún nodo de r entonces $a < y$.

Algunas implementaciones de funciones comunes para este tipo son:

```
-- Determina si un dato está en un BST
member :: Ord a => a -> Bin a -> Bool
member a Hoja = False
member a (Nodo l b r) | a == b = True
                      | a < b = member a l
                      | a > b = member a r
```

```haskell
-- Mínimo valor en un BST no vacío
minimum :: Bin a -> a
minimum (Nodo Hoja a r) = a
minimum (Nodo l a r) = minimum l

-- Máximo valor en un BST no vacío
maximum :: Bin a -> a
maximum (Nodo l a Hoja) = a
maximum (Nodo l a r) = maximum r

-- Determina si un arbol es BST
checkBST :: Bin a -> Bool
checkBST Hoja = True
checkBST (Nodo l x r) = (checkBST l) && (checkBST r) && lineq && rineq
                      where lineq = case l of
                                      Leaf -> True
                                      maximumBST l -> y <= x
                            rineq = case r of
                                      Leaf -> True
                                      minimumBST r -> x <= z
```

Teorema
Si t es un BST, entonces `member`, `minimum` y `maximum` son de orden $O(h)$, donde h es la altura de t.

En el mejor de los casos estas operaciones tendrá orden $O(\lg n)$, pero en el peor caso serán $O(n)$. Es por esto que nos interesa utilizar otra versión de arboles binarios de búsqueda, que nos pueda garantizar que su altura será razonable. Para esto existen los arboles balanceados, y a continuación se presenta un tipo de estos.

3.3 Red-Black Trees

La estructura Red-Black Tree es un árbol binario de búsqueda autobalanceado, osea, que implementa mecanismos para controlar el crecimiento en su altura al realizar operaciones sobre este. Cada nodo en un Red-Black Tree cuenta con un dato añadido, un color que puede ser rojo o negro, y en la estructura deben satisfacerse las siguientes invariantes:

INV1: Ningún nodo de color rojo tiene un hijo rojo

INV2: Todos los caminos de la raíz a una hoja tienen la misma cantidad de hijos negros. Esta cantidad es denominada ”altura negra”.

1) En un Red-Black Tree, el camino más largo de la raíz a una hoja tendrá longitud a lo sumo del doble de la longitud del camino más corto de la raíz a una hoja.
Para razonar esto, pensemos que en el peor caso el camino más corto estará formado por todos nodos negros y el más largo intercalará entre negro y rojo. Si no se cumpliese esto, el camino más largo tendría mas nodos negros o un par de nodos padre hijo ambos de color rojo.

2) Sea h la altura de un Red-Black Tree y n su cantidad de nodos, entonces $h \in O(\lg n)$. Esto es, un árbol Red-Black Tree es un árbol balanceado.
Esta propiedad no es tan directa de ver. Sea k la altura negra del árbol. Entonces, todos los caminos tienen k nodos negros, y en el mejor de los casos no habría nodos rojos, dando lugar a la desigualdad $n \geq 2^k - 1$. Luego, $\lg(n+1) \geq k$. Ahora, por Propiedad (1), si h es la altura del árbol, $h \leq 2k$, luego $h \leq 2k \leq 2\lg(n+1) \implies h \in O(\lg n)$. □

Definimos `data Color = R | B`, y representamos este tipo de árboles con el tipo:

```
data RBT a = E | T Color (RBT a) a (RBT a)
```

La idea para insertar en este tipo de árboles es insertar en el lugar que corresponde (mismo procedimiento que insertar en un BST), y siempre al nuevo nodo se le asigna el color rojo para preservar la segunda invariante. Ahora, ¿qué sucede si el padre del nodo que acabamos de insertar es de color rojo? Eso violaría la primer invariante. Como el árbol que había antes cumplía con la invariante, tenemos que los últimos 3 nodos en el camino tienen colores B-R-R, y la idea será reestructurarlo en términos de un nodo rojo con hijos negros, y propagar el problema para arriba. Las posibles violaciones a esta invariante, y como corregirlas, están representadas en la figura 2.
Por último, a la raíz le daremos el color negro resolviendo el problema. Este cambio aumenta en uno la cantidad de nodos negros de cada camino, osea que la segunda invariante se mantiene y la altura negra aumenta en uno.

A continuación damos la implementación de la función de insertar y balanceo, siguiendo los pasos detallados anteriormente:

```
insert :: Ord a => a -> RBT a -> RBT a
insert x t = makeBlack (ins x t)
      where ins x E = T R E x E
            ins x (T c l y r) | x < y = balance c (ins x l) y r
                              | x > y = balance c l y (ins x r)
                              | otherwise = T c l y r
            makeBlack E = E
            makeBlack (T _ l x r) = T B l x r
```

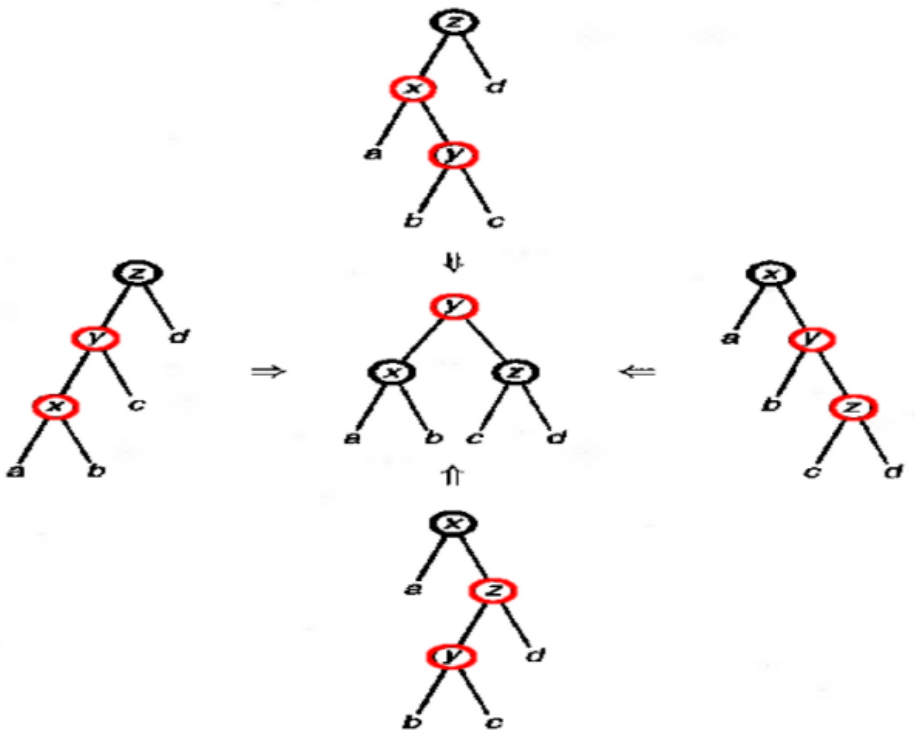


Figura 2: Casos para balanceo al insertar en un Red-Black Tree

```
balance :: Color -> RBT a -> a -> RBT a
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance c l a r = T c l a r
```

Vemos que $W_{balance} \in O(1)$, luego $W_{insert} \in O(\lg n)$

Por último, existen varios tipos de arboles autobalanceados, como lo son los AVL o B-Tree, sin embargo los Red-Black Tree presentan una fácil implementación gracias al Pattern Matching de Haskell. En la práctica también se encuentra otro tipo de árboles, los 1-2-3 Tree, que de hecho son isomorfos a los Red-Black Tree.

3.4 Heaps

Los heaps son árboles que permiten un acceso eficiente al mínimo elemento de este (o máximo en el caso de Max Heaps). La invariante en este tipo de estructuras es que cada nodo tiene un dato menor al almacenado en sus hijos, y por ende el mínimo dato en la estructura es

aquel almacenado en la raíz.
Hay varias versiones de heaps con distintos invariantes además de las tradicionales: Leftist, Binomial, Splay y Pairing Heaps. En cualquier caso, la idea es que todas estas deben tener operaciones `insert`, `findMin` y `deleteMin` eficientes, y tal vez hasta `merge`.

Los Pairing Heaps, se encuentran en la práctica. La idea general de estos es generalizar la idea del Heap tradicional, permitiendo a un nodo tener varios hijos, todos siendo a su vez Pairing Heaps. Esta variante también es relativamente fácil de implementar en Haskell.

Estudiaremos e implementaremos la estructura Leftist Heap, que presenta un invariante que hará sencillo y eficiente la implementación de `merge`, a partir de la cual definimos las demás.

3.5 Leftist Heaps

La estructura Leftist Heap es una variante de la estructura Heap que resulta fácil de implementar de manera persistente. Se definen dos términos para trabajar con estas estructuras, espina derecha (la cual representa el camino simple desde la raíz hasta el nodo vacío más a la derecha) y rango, definido como la longitud de la espina derecha.
La invariante que mantiene esta estructura es: el rango de cualquier hijo izquierdo es mayor o igual que el rango de su hermano derecho. Como consecuencia de esta y la invariante de Heap, se derivan las siguientes propiedades.

`Propiedades:`

1. La espina derecha es el camino mas corto a un nodo vacío.
2. Los elementos de la espina derecha están ordenados.
3. El rango es a lo sumo $\lg(n+1)$, con n la cantidad de nodos.

La ultima propiedad es importante y no tan directa de ver. La idea es entender al rango no sólo como la longitud de la espina derecha, sino también como la longitud del camino más corto de la raíz a una hoja (similar al concepto de altura pero para el camino mas corto). Si no fuese así, los hijos podrían intercambiarse de lugar para que dicho camino quede en la espina derecha, obteniendo un menor rango, lo cual sería contradictorio.
Surge luego que si el camino más corto desde la raíz hasta una hoja es r, entonces el árbol es completo hasta la profundidad r (de no serlo se encontraría un camino más corto). Ser completo hasta la profundidad r nos implica $n \geq 2^r - 1 \implies \lg(n+1) \geq r$, y como $r \in \mathbb{N}$ tenemos $r \leq \lfloor \lg(n+1) \rfloor$.

Ahora, en esta estructura la operación más importante es `merge`. A partir de esta podemos definir el resto de operaciones con complejidades logarítmicas. En el siguiente fragmento de código damos la representación de este tipo de datos y la implementación de esta función.

```
type Rank = Int
data Heap a = E | N Rank a (Heap a) (Heap a)
```

```haskell
merge :: Ord a => Heap a -> Heap a -> Heap a
merge h1 E = h1
merge E h2 = h2
merge h1@(N x a1 b1) h2@(N y a2 b2) =
        if x < y then makeH x a1 (merge b1 h2)
                 else makeH y a2 (merge h1 b2)

rank :: Heap a -> Rank
rank E = 0
rank (N r ) = r

makeH :: a -> Heap a -> Heap a -> Heap a
makeH x a b = if rank a > rank b then N (rank b + 1) x a b
                                 else N (rank a + 1) x b a
```

La idea general es mezlcar las ramas derechas de los dos heaps según corresponda, esto es, ver quien tiene el menor dato en la raíz, si el primero, entonces mezclamos su hijo derecho con el segundo heap, y sino mezclamos el hijo derecho del segundo heap con el primer heap. Una vez hecho esto, la función auxiliar `makeH` se encarga de colocar los dos hijos correspondientes según quien tenga el menor rango. Tenemos de aquí que $W_{makeH}, W_{rank} \in O(1)$, y junto con la propiedad de que el rango es a lo sumo logarítmico, se deduce $W_{merge} \in O(\lg n)$, donde n es el máximo entre la cantidad de nodos de ambos heaps.

NOTA: Los leftist heap **NO** son arboles balanceados. Lo importante en esta estructura es que siempre conocemos el camino más corto a la raíz y de esta manera usar este camino para hacer la recursión de `merge` lo más corta posible.

Con esta función, las demás tienen una definición sencilla, y es fácil ver que $W_{insert}, W_{deleteMin} \in O(\lg n)$, $W_{findMin} \in O(1)$.

```haskell
insert :: Ord a => a -> Heap a -> Heap a
insert x h = merge (N 1 x E E) h

findMin :: Ord a => Heap a -> a
findMin (N x a b) = x

deleteMin :: Ord a => Heap a -> Heap a
deleteMin (N x a b) = merge a b
```

4 Unidad IV - TADs y Especificaciones

Al usar estructuras de datos, en general no nos interesa la implementación interna de esta. Suelen existir varias formas distintas de implementar algunas estructuras, y para preservar un buen diseño de software lo ideal es encapsular el comportamiento de tales en un conjunto de operaciones y reglas que estas deben seguir. A esta clase de tipos de datos los llamamos Tipos Abstractos de Datos (TADs). Un TAD consta de un tipo de dato que presenta una interfaz de uso propio, osea, funciones que se pueden usar para construir, modificar, combinar, visualizar, etc., el tipo, sin tener que conocer su implementación interna.
Sin embargo, una interfaz de este tipo puede resultar inútil si no tenemos garantías sobre lo que hacen, o sobre si su comportamiento es el esperado. Esta es la utilidad de las especificaciones. Una especificación consta de una serie de "leyes" que las operaciones expuestas por el TAD deben cumplir. Existen diversas formas de expresar tales leyes, y en la asignatura se estudian dos: especificación algebraica y especificación por modelos.

Como ejemplo simple, se tienen las colas. El TAD de cola consiste entonces en:

1. Nombre del tipo: Cola
2. Operaciones
   tad $Cola\,(A : Set)$ **where**
   import $Bool$
   `vacia` : $Cola\,A$
   `poner` : $A \to Cola\,A \to Cola\,A$
   `primero`: $Cola\,A \to A$
   `sacar` : $Cola\,A \to Cola\,A$
   `esVacia`: $Cola\,A \to Bool$
3. Especificación de operaciones: Estas se dan a lo largo de las siguientes secciones, a medida que se introduce cada tipo de especificación.

Cuando definimos un TAD, generalmente exigimos ciertas propiedades de los datos, como orden parcial, igualdad, etc. para poder usar tales dentro de la estructura. Estos requerimientos los indicamos escribiendo $Cola\,(A : Set)$ para dar a entender que $Cola$ es un tipo de dato abstracto para cualquier tipo A que se pueda pensar como un conjunto o colección de elementos. Si, por ejemplo, necesitamos que estos datos sean ordenados, lo simbolizaríamos como $A : Ordered\,Set$. Estas son las únicas dos restricciones que impondremos sobre los datos al trabajar con TADs en esta materia.

4.1 Especificación algebraica

La especificación algebraica consiste en expresar el comportamiento de las operaciones de un TAD mediante ecuaciones, en particular relacionar comportamientos entre una o más

operaciones de la interfaz.

NOTA: Las especificaciones algebraicas se escribirán en pseudocódigo. Muchas veces estas serán sintácticamente idénticas a una posible implementación, pero no se debe confundirlas ya que son esencialmente distintas. El = en implementaciones representa una definición, pero en las especificaciones algebraicas representa una igualdad.

En una especificación algebraica entonces solo aparecen operaciones de TADs junto con variables libres cuantificadas universalmente. Entonces, por ejemplo, si se quiere especificar el comportamiento de `size`, lo siguiente es incorrecto:

$$\texttt{size x} = \begin{cases} 0 & \text{si } \texttt{esVacia x} \\ 1 + \texttt{size (sacar x)} & \text{en otro caso} \end{cases}$$

mientras que lo correcto es hacerlo como:

$$\texttt{size x} = \textbf{if } \texttt{esVacia x} \textbf{ then } 0 \textbf{ else } 1 + \texttt{size (sacar x)}$$

En este ejemplo, 0, 1 y + forman todos parte del TAD Nat, y el operador `if then else` forma parte del TAD Bool.

A continuación, se da una especificación algebraica para el TAD Cola:

```
esVacia vacia                 = True
esVacia (poner x q)           = False
primero (poner x vacia)       = x
primero (poner x (poner y q)) = primero (poner y q)
sacar (poner x vacia)         = vacia
sacar (poner x (poner y q))   = poner x (sacar (poner y q))
```

Notemos que las especificaciones no tienen por qué cubrir todos los casos, ya que por ejemplo aquí ninguna ecuación nos dice que debería suceder con `primero vacía`. Los casos que no sean cubiertos por ninguna ecuación de la especificación los llamaremos **comportamiento indefinido**.

4.2 Especificación por modelos

La especificación por modelos de un TAD consiste en darle una interpretación semántica mediante un modelo al tipo de dato. La idea es dar una interpretación a cada operación de la interfaz, y una asignación a cada instancia del tipo de dato con un elemento particular del elemento del universo. Esto es equivalente a decir:

- Si `x` es una instancia del tipo, $[\![\texttt{x}]\!]$ es un elemento en el universo del modelo.
- Si `op` es una función del tipo, entonces en el modelo se tiene una función $[\![\texttt{op}]\!]$.

Y diremos que una implementación se ajusta a un modelo si para todo `x` y `op`:

$$[\![\texttt{op x}]\!] = [\![\texttt{op}]\!][\![\texttt{x}]\!]$$

Para el caso de las colas, una especificación posible por modelos es usando secuencias: Una instancia de cola se representará mediante una secuencia finita $\langle x_1, x_2, ..., x_n\rangle$ y las operaciones las interpretamos como:

$$\begin{aligned}
&\texttt{vacia} && = \langle\rangle \\
&\texttt{poner}\, x\, \langle x_1, x_2, ..., x_n\rangle && = \langle x, x_1, x_2, ..., x_n\rangle \\
&\texttt{sacar}\, \langle x_1, x_2, ..., x_n\rangle && = \langle x_1, x_2, ..., x_{n-1}\rangle \\
&\texttt{primero}\, \langle x_1, x_2, ..., x_n\rangle && = x_n \\
&\texttt{esVacia}\, \langle x_1, x_2, ..., x_n\rangle && = \texttt{True si } n = 0 \\
&\texttt{esVacia}\, \langle x_1, x_2, ..., x_n\rangle && = \texttt{False en otro caso}
\end{aligned}$$

4.3 Implementaciones y Costos

En esta sección se cubren distintas posibles implementaciones para el TAD cola, y compararemos las complejidades de sus operaciones.

Listas enlazadas con inicio de cola al principio
En esta implementación la cola crece hacia el final de la lista.

```
vacia = []

esVacia [] = True
esVacia _  = False

poner x xs = xs ++ [x]

primero = head

sacar = tail
```

Listas enlazadas con inicio de cola al final
En esta implementación la cola crece hacia el comienzo de la lista.

```
vacia = []

esVacia [] = True
esVacia _  = False

poner = (:)

primero    [x] = x
primero (x:xs) = primero xs

sacar [x] = []
sacar (x:xs) = x : sacar xs
```

Par de listas
En esta implementación una cola se representa con `(xs, ys)`, y el orden de los elementos está dado por `xs ++ reverse ys`. La idea es que tanto el primero como el último de la cola siempre están en el inicio de alguna de las dos listas.
Esta estructura además llevará la invariante: si `xs` vacía, `ys` también lo es.

```
vacia = ([], [])

esVacia ([], ys) = True
esVacia    _     = False

poner a ([], []) = ([a], [])
poner a (xs, ys) = (xs, a:ys)

primero (x:xs, ys) = x

sacar ([x], ys) = (reverse ys, [])
sacar (x:xs, ys) = (xs, ys)
```

4.4 Razonamiento sobre Programas e Inducción Estructural

Mediante los Tipos Abstractos de Datos presentamos una entidad con ciertas operaciones, y con las especificaciones podemos establecer reglas que queremos que tales operaciones cumplan. Ahora, dada una implementación concreta ¿cómo sabemos que cumple con las especificaciones?. Queremos conocer una forma de razonar sobre programas para poder determinar si verifica o no las ecuaciones presentadas mediante la especificación.
Los lenguajes funcionales (o declarativos en general) son ideales para esto, ya que se abstraen

[1] $W_{sacar}(\texttt{(xs,ys)}) \in O(|\texttt{ys}|)$, pero es $O(1)$ amortizado. Este concepto no se ve en la asignatura.

Implementación	Listas (inicio en cabeza)	Listas (final en cabeza)	Par de listas
W_{vacia}	$O(1)$	$O(1)$	$O(1)$
$W_{esVacia}(\texttt{q})$	$O(1)$	$O(1)$	$O(1)$
$W_{poner}(\texttt{q})$	$O(\lvert\texttt{q}\rvert)$	$O(1)$	$O(1)$
$W_{primero}(\texttt{q})$	$O(1)$	$O(\lvert\texttt{q}\rvert)$	$O(1)$
$W_{sacar}(\texttt{q})$	$O(1)$	$O(\lvert\texttt{q}\rvert)$	$O(1)$[1]

Tabla 1: Tabla de costos para TAD de colas según implementación

del "cómo hacer" y se enfocan en el "qué hacer". En un lenguaje funcional no tendremos que analizar instrucciones como `int x = 1;`, ni tendremos que analizar funciones que tengan efectos secundarios. En C, una estructura de datos podría ser modificada en cualquier momento (por otros hilos de ejecución, efectos secundarios de una función, etc.), mientras que en Haskell nuestras estructuras son inmutables. Además, los lenguajes funcionales tienen una tendencia muy fuerte al uso de recursión, la cual esta fuertemente ligada al Principio de Inducción Matemática. Más aún, el sistema de tipos estático y fuerte de Haskell nos asegurará a tiempo de compilación que los tipos de la implementación son correctos, lo cual nos da una gran ayuda de base. Queda como tarea del programador verificar que su implementación cumple tal especificación.

Para decidir si dos funciones `f, g` : $A \to B$ son equivalentes solo nos interesa comparar los resultados de ambas, y no la implementación o funcionamiento interno de cada una. De aquí surge el **Principio de Extensionalidad**, que sostiene:

$$\texttt{f = g} \iff \forall \texttt{x : A. f x = g x}$$

Razonamiento ecuacional: La combinación entre la programación funcional y la sintaxis de Haskell (o nuestro pseudocódigo en cuestión) nos permiten escribir pruebas de programas como si fueran ecuaciones algebraicas. Por ejemplo, dada la siguiente definición de la función, probamos que `reverse [x] = [x]`:

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

```
 reverse [x]
= Def. [x]
 reverse (x:[])
= reverse . 2
 reverse [] ++ [x]
= reverse . 1
 [] ++ [x]
= Def ++
 [x]
```

NOTA: Al razonar sobre programas, se usa = y no ==, ya que el segundo refiere simplemente a un operador en el lenguaje.

Descomposición en casos y/o patrones: Una forma de prueba común en matemática es descomponer en casos algún objeto para poder aplicar argumentos más específicos en el razonamiento. Lo más parecido a casos que hemos usado nosotros son los patrones, y es por esto que será importante hacer pruebas identificando los patrones que cumpla nuestra estructura, ya que a cada patrón suele corresponderle una distinta definición de una función dada.

Hay que tener precauciones a la hora de hacer esto. Si se considera el programa 3a, podemos notar que el orden es importante. El segundo patrón solo es aplicable si el primero no lo es, luego es equivalente a `n /= 0`.

```
esCero :: Int -> Bool
esCero 0 = True
esCero n = False
```

(a) Función definida por patrones en orden

```
esCero': : Int -> Bool
esCero' 0 = True
esCero' n | n /= 0 = False
```

(b) Función definida por patrones disjuntos

Figura 3: Funciones equivalentes con distinto uso de patrones

Por este motivo, es más práctico y sencillo usar patrones disjuntos como lo hace el fragmento 3b para que así no sea necesario tener en cuenta el orden de las ecuaciones.

Inducción sobre $\mathbb{N}$: Otra gran técnica de prueba es la inducción sobre los números naturales. El Principio de Inducción Matemática nos permite desarrollar argumentos sobre razonamientos de infinitos de manera finita. Se da una breve definición a modo de referencia de este:

Principio de Inducción Matemática
Sea $P(n)$ un predicado sobre los naturales. Si valen:

- Caso base: $P(0)$
- Case inductivo: $\forall m \in \mathbb{N}, P(m) \implies P(m+1)$ (A $P(m)$ lo llamamos Hipótesis Inductiva)

Entonces vale $\forall n \in \mathbb{N}. P(n)$

Otro tipo de inducción sobre los naturales, aunque equivalente en cuanto a poder de prueba se refiere, es la inducción fuerte, la cual toma una hipótesis inductiva más general.

Principio de Inducción Fuerte
Sea $P(n)$ un predicado sobre los naturales. Si cuando vale $\forall m \leq n$, $P(m)$ se tiene $P(n+1)$, entonces $\forall n \in \mathbb{N}$, $P(n)$

Ahora, la inducción sobre naturales puede quedar corta para nuestro caso, ya que en general trabajaremos con estructuras más complejas. Sería interesante entonces poder usar esta inducción para realizar inducción sobre otros conjuntos. A modo de ejemplo, esto podría ser inducción sobre la longitud de una lista, la altura o cantidad de nodos de un árbol, etc.
Para poder transformar una propiedad de A a $\mathbb{N}$, dada una función $\texttt{f} : A \to \mathbb{N}$ y un predicado sobre A, se construye:

$$Q(n) \equiv \forall \texttt{a} : A. \texttt{ f a} = n \implies P(\texttt{a})$$

Si bien esto es útil, estamos haciendo una prueba indirecta sobre A ya que recurrimos a la inducción en $\mathbb{N}$. Pero podría ocurrir que alguna propiedad no sea fácilmente expresable en términos de una $\texttt{f} : A \to \mathbb{N}$. El siguiente método trabaja directamente sobre A:

Principio de Inducción Estructural
Dada una propiedad P sobre un tipo de datos algebraico T, si valen:

- Caso base: $P(t)$ vale para todo t que sea un constructor no recursivo
- Case inductivo: Sea t un constructor recursivo con instancias recursivas $t_1, t_2, \ldots, t_k$, si $P(t_i)$, $1 \leq i \leq k$ entonces $P(t)$.

entonces vale $\forall t : T. P(t)$.
NOTA: La inducción estructural tiene una variante que es análoga a la inducción fuerte en naturales. En esta, se asume como hipótesis inductiva que vale $P(t')$ para todo t' contenido en la estructura t, y no solo en los hijos.

Probaremos que `reverse (xs ++ ys) = reverse ys ++ reverse xs`.
Para el tipo de listas de Haskell, el Principio de Inducción Estructural nos dice: Si P es una propiedad sobre listas, para probar $\forall \texttt{xs} :: [\texttt{a}]\ P(\texttt{xs})$ probamos $P([\,])$ y $P(\texttt{xs}) \implies P(\texttt{x:xs})$
Haremos inducción sobre el primer argumento.

Caso base: `xs = []`

$$\begin{aligned} \texttt{reverse (xs ++ ys)} &= \texttt{reverse ([] ++ ys)} \\ &= \texttt{reverse ys} \\ &= \texttt{reverse ys ++ []} \\ &= \texttt{reverse ys ++ reverse []} \\ &= \texttt{reverse ys ++ reverse xs} \end{aligned}$$

Caso inductivo: Tomamos `reverse (xs ++ ys) = reverse ys ++ reverse xs` como hipótesis inductiva, y probamos para `(x:xs)`:

$$\begin{aligned} \texttt{reverse ((x:xs) ++ ys)} &= \texttt{reverse (x:(xs ++ ys))} \\ &= \texttt{reverse (xs ++ ys) ++ [x]} \\ &= \texttt{(reverse ys ++ reverse xs) ++ [x]} && \text{(H.I.)} \\ &= \texttt{reverse ys ++ (reverse xs ++ [x])} && \text{asoc. de ++} \\ &= \texttt{reverse ys ++ reverse (x:xs)} \end{aligned}$$

$\therefore \forall$ `xs, ys :: [a], reverse (xs ++ ys) = reverse ys ++ reverse xs`

5 Unidad V - Estructuras Paralelizables y Secuencias

Si recordamos la implementación del algoritmos `msort` sobre listas en la primera unidad, se llegó a que las cotas de su trabajo y profundidad eran $W_{msort}(n) \in O(n \lg n)$ y $S_{msort}(n) \in O(n)$. Si recordamos la definición de paralelismo, $P = \lg n$ para este algoritmo, lo cual nos dice que muy pocos procesadores pueden usarse de forma eficiente. El problema aquí es la profundidad, ya que el trabajo no puede ser mejorado (se puede demostrar que cualquier algoritmo de ordenamiento basado en comparaciones es al menos $n \lg n$ en trabajo). Si analizamos la recurrencia de la profundidad de `msort`, $S_{msort}(n) = S_{split}(n) + S_{msort}(n/2) + S_{merge}(n) + k$ notaremos que son `merge` y `split` quienes incurren un término lineal en la recurrencia. ¿Se pueden entonces mejorar las profundidades de ambas?
La respuesta es sí y no. Sí se pueden mejorar las profundidades de las tres funciones, pero no trabajando sobre listas. Esto es porque las listas son estructuras inherentemente no paralelas. Su modelo de construcción es claramente secuencial, nodo tras nodo hasta llegar al final. La idea para mejorar la profundidad de la implementación de mergesort y cualquier algoritmo en general será buscar una estructura de datos que admita altos niveles de paralelización.

5.1 Implementación paralela de mergesort

Para este problema trabajaremos con árboles binarios de la forma
`data BT a = Empty | Node (BT a) a (BT a)`. Como primera ventaja de esta estructura destacamos $W_{split}, S_{split} \in O(1)$ (basta con hacer pattern matching), y ahora nos interesa definir S_{merge} tal que sea mejor que $O(n)$.
Ahora, para hacer un merge de dos árboles ordenados `t1` y `t2` tomaremos la raíz del primero, llamémosle x, y construiremos un nuevo árbol con raíz x tal que los hijos sean llamadas recursivas paralelas a `merge`. En el hijo izquierdo irá el merge del árbol de valores menores a x en `t1` y `t2`. Realizaremos lo mismo pero con los árboles de elementos mayores a x para el hijo derecho. Los árboles con valores menores y mayores a x en `t1` son su hijo izquierdo y derecho resp. ya que `t1` está ordenado. Para calcular estos árboles para `t2` usaremos una función auxiliar `splitAt :: Ord a => BT a -> a -> (BT a, BT a)` la cual los calculará en tiempo proporcional a la altura de `t2`. Con esto aclarado, la implementación es la siguiente:

```
splitAt :: Ord a => BT a -> a -> (BT a, BT a)
splitAt Empty _ = (Empty, Empty)
splitAt (Node l y r) x = if y <= x then let (small, big) = splitAt r x
                                        in (Node l y small, big)
                                   else let (small, big) = splitAt l x
                                        in (small, Node big y r)

merge :: Ord a => BT a -> BT a -> BT a
merge t Empty = t
merge Empty t = t
```

```
merge (Node l1 x r1) t = let (l2, r2) = splitAt t x
                             (l', r') = merge l1 l2 ||| merge r1 r2
                         in Node l' x r'

msort :: BT a -> BT a
msort Empty = Empty
msort (Node l x r) = let (l', r') = msort l ||| msort r
                     in rebalance $ merge (merge l' r') (Node Empty x Empty)
```

Notemos que a la implementación se agregó una función `rebalance`. No cubrimos la implementación de esta función ya que requeriría entrar muy en detalle en la implementación (se pide como ejercicio en la práctica de la materia).

Ahora, $S_{splitAt}(h) = S_{splitAt}(h-1) + k \implies S_{splitAt}(h) \in O(h)$, donde h es la altura del árbol. Por otro lado, si h_1 y h_2 son las alturas de los arboles argumento a `merge`, y h_{21}, h_{22} son las alturas de los árboles resultantes de la llamada a `splitAt`, tendremos:

$$\begin{aligned} S_{merge}(h_1, h_2) &\leq S_{splitAt}(h_2) + \max\left(S_{merge}(h_1 - 1,\ h_{21}), S_{merge}(h_1 - 1,\ h_{22})\right) + c \\ &\leq S_{splitAt}(h_2) + \max\left(S_{merge}(h_1 - 1,\ h_2), S_{merge}(h_1 - 1,\ h_2)\right) + c \\ &\qquad\qquad \text{ya que } h_{21}, h_{22} \leq h_2 \\ &= S_{splitAt}(h_2) + S_{merge}(h_1 - 1,\ h_2) + c \\ &= S_{merge}(h_1 - 1,\ h_2) + O(h_2) \in O(h_1 \cdot h_2) \end{aligned}$$

Y de aquí surge que para un árbol balanceado con n nodos:

$$\begin{aligned} S_{msort}(n) &\leq \max\left(S_{msort}\left(\frac{n}{2}\right), S_{msort}\left(\frac{n}{2}\right)\right) + S_{merge}(\lg n,\ \lg n) + S_{merge}(2\lg n,\ 1) + s \\ &= S_{msort}\left(\frac{n}{2}\right) + k_1 \lg^2 n + k_2 \lg n + s \\ &= S_{msort}\left(\frac{n}{2}\right) + O(\lg^2 n) \in O(\lg^3 n) \end{aligned}$$

5.2 Funciones de alto orden en árboles

Trabajaremos ahora sobre otra variante de árboles binarios

```
data T a = Empty | Leaf a | Node (T a) (T a)
```

Asumiremos que trabajamos con árboles balanceados y que usaremos funciones $W_{\mathtt{f}}, S_{\mathtt{f}} \in O(1)$. Con las siguientes funciones de alto orden se abstraen los patrones más interesantes y usados en árboles, suficientes para diseñar soluciones a múltiples problemas.

```
mapT :: (a -> b) -> T a -> T b
mapT f Empty = Empty
```

```
mapT f (Leaf x) = Leaf (f x)
mapT f (Node l r) = let (l', r') = mapT f l ||| mapT f r
                    in Node l' r'

reduceT :: (a -> a -> a) -> a -> T a -> a
reduceT _ e Empty = e
reduceT _ _ (Leaf x) = x
reduceT f e (Node l r) = let (l', r') = reduceT f e l ||| reduceT f e r
                         in f l' r'
```

Bajo las hipótesis antes planteadas logramos $W_{mapT}, W_{reduceT} \in O(n)$ y $S_{mapT}, S_{reduceT} \in O(\lg n)$.

Varias veces requeriremos hacer un reduce seguido de un map. Componer ambas funciones resulta no ser lo más eficiente, ya que el árbol creado por `mapT` es inmediatamente consumido por `reduceT`. Luego, es útil contar con una función `mapreduceT` que cumpla la especificación `mapreduceT f g e t = reduceT g e (mapT f t)`.

```
mapreduceT :: (a -> b) -> (b -> b -> b) -> b -> T a -> b
mapreduceT _ _ e Empty = e
mapreduceT f _ _ (Leaf x) = f x
mapreduceT f g e (Node l r) = let (l', r') = mapreduceT f g e l
                                             |||
                                             mapreduceT f g e r
                              in g l' r'
```

5.3 Secuencias

Seq es un TAD que usaremos para representar secuencias de cualquier tipo de elementos. Una secuencia s se denotará $\langle s_0, s_1, \ldots, s_{n-1} \rangle$, y $|s| = n$ es la longitud de la secuencia. Si I es una secuencia estrictamente creciente de índices válidos, entonces la secuencia $s'_i = s_{I_i}$ es una subsecuencia de s.
A continuación estan listadas las operaciones de la interfaz *Seq*.

- `empty` : *Seq* `a`
  Representa la secuencia $\langle\rangle$
- `singleton` : `a` $\to$ *Seq* `a`
  `singleton` x representa la secuencia $\langle x \rangle$
- `length` : *Seq* `a` $\to \mathbb{N}$
  `length` s evalúa a $|s|$

- `nth` : Seq `a` $\to \mathbb{N} \to$ `a`
  Si i es un índice válido, `nth` $s\, i$ evalúa a s_i

- `toSeq` : `[a]` $\to$ Seq `a`
  Convierte una lista en una secuencia, con el orden dado por la lista.

- `tabulate` : `(`$\mathbb{N} \to$ `a)` $\to \mathbb{N} \to$ Seq `a`
  `tabulate` $f\, n$ evalúa a la secuencia $\langle f(0),\, f(1), \ldots,\, f(n-1)\rangle$

- `map` : `(a` $\to$ `b)` $\to$ Seq `a` $\to$ Seq `b`
  `map` $f\, s$ evalúa a la secuencia $\langle f(s_0),\, f(s_1), \ldots,\, f(s_{|s|-1})\rangle$

- `filter` : `(a` $\to$ $Bool$`)` $\to$ Seq `a` $\to$ Seq `a`
  `filter` $p\, s$ evalúa a la subsecuencia de todos los elementos tales que ps_i `== True`.

- `append` : Seq `a` $\to$ Seq `a` $\to$ Seq `a`
  `append` $s\, t$ evalúa a $\langle s_0,\, s_1, \ldots,\, s_{|s|-1},\, t_0,\, t_1, \ldots,\, t_{|t|-1}\rangle$

- `take` : Seq `a` $\to \mathbb{N} \to$ Seq `a`
  `take` $s\, k$ evalúa a $\langle s_0,\, s_1, \ldots,\, s_{min(k,|s|)-1}\rangle$

- `drop` : Seq `a` $\to \mathbb{N} \to$ Seq `a`
  `drop` $s\, k$ evalúa a $\langle s_k,\, s_{k+1}, \ldots,\, s_{|s|-1}\rangle$

- `foldr` : `(a` $\to$ `b` $\to$ `b)` $\to$ `b` $\to$ Seq `a` $\to$ `b`
  `foldr` $\oplus\, e\, s$ evalúa a $(s_0 \oplus (s_1 \oplus (\ldots (s_{|s|-1} \oplus e) \ldots)))$

- `foldl` : `(b` $\to$ `a` $\to$ `b)` $\to$ `b` $\to$ Seq `a` $\to$ `b`.
  `foldl` $\oplus\, e\, s$ evalúa a $(\ldots ((e \oplus s_0) \oplus s_1) \oplus \ldots) \oplus\ s_{|s|-1}$

- `showt` : Seq `a` $\to$ $TreeView$ `a`
  donde `data TreeView a = EMPTY | ELF a | NODE (Seq a) (Seq a)`.
  La idea es que `showt` s vale:
  - `EMPTY` si $|s| = 0$
  - `ELF` s_0 si $|s| = 1$
  - `NODE (take` $\frac{|s|}{2}$`) (drop` $\frac{|s|}{2}$`)` si $|s| > 1$

- `reduce` : `(a` $\to$ `a` $\to$ `a)` $\to$ `a` $\to$ Seq `a` $\to$ `a`

- `scan` : `(a` $\to$ `a` $\to$ `a)` $\to$ `a` $\to$ Seq `a` $\to$ `(`Seq `a` $\times$ `a)`

- `collect` : Seq `a` $\times$ `b` $\to$ Seq `(a` $\times$ Seq `b)`

5.4 Operación `reduce`

Una de las operaciones más importantes en secuencias es `reduce`. Esta operación es similar a `foldr` y `foldl` pero con otro orden de reducción. Si $\oplus$ es asociativa el orden de reducción no importa, y si además `e` es el neutro de $\oplus$ entonces se tiene `reduce` $\oplus$ `e s` = `foldr` $\oplus$ `e s` = `foldl` $\oplus$ `e s`. Para el caso general buscamos un orden de reducción que nos permita la mayor paralelización posible. Tal orden de reducción lo podemos expresar para un árbol como:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
toTree : Seq a → Tree a
toTree s = case |s| of
    1 → (Leaf s0)
    n → Node (toTree (take s pp)) (toTree (drop s pp))
    where pp = 2 ↑ ilg (n - 1)

reduceT : (a → a → a) → Tree a → a
reduceT ⊕ (Leaf x) = x
reduceT ⊕ (Node l r) = (reduceT ⊕ l) ⊕ (reduceT ⊕ r)
```

La estructura general de tal reducción es operar entre pares adyacentes (excepto tal vez el último si la longitud de s es impar), y sobre esa nueva secuencia de elementos reducidos realizar una nueva reducción. Entonces, una especificación para `reduce` es:

```
reduce ⊕ e s = if length s == 0 then e else e ⊕ (reduceT ⊕ (toTree s))
```

NOTA: El código anterior busca dar una especificación para el orden de reducción, **NO** es una implementación. Implementar `reduce` de tal forma resultaría extremadamente ineficiente.

Planteamos el caso de una secuencia s de longitud 7 para visualizar su árbol de reducción.

El árbol de la figura 4 muestra el orden de reducción que se obtiene al aplicar `reduce` sobre `s`. Luego, `reduce` $\oplus\, e\, s = e \oplus (((s_0 \oplus s_1) \oplus (s_2 \oplus s_3)) \oplus ((s_4 \oplus s_5) \oplus s_6))$

El trabajo y profundidad de `reduce` dependerán de la estructura usada para implementar el TAD secuencias, del trabajo y profundidad del operador de reducción y además del orden de reducción usado. El orden de reducción dado minimiza la profundidad del algoritmo, y por ende usamos siempre este en la asignatura.

En la primera unidad se trató la estrategia Divide&Conquer, la cual sigue un patrón claro y simple de resolución de problemas. Para las secuencias esta estrategia puede pensarse como:

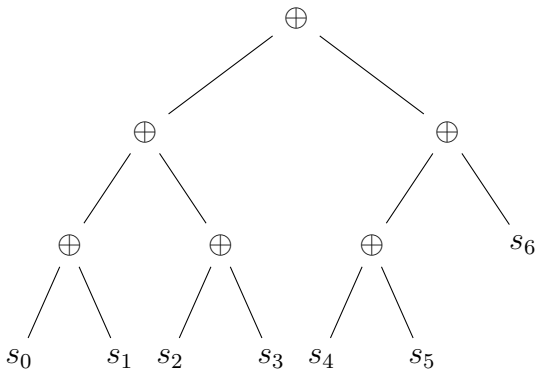


Figura 4: Árbol de reducción para secuencia de longitud 7

```
dyc s = case showt s of
  EMPTY → val
  ELT v → base v
  NODE l r → let (l',r') = (dyc l || dyc r) in combine l' r'
```

donde `val` es el valor para la secuencia vacía, `base` es una función que se aplica sobre cada caso base, y `combine` es la función que combina dos soluciones recursivas. Esto es equivalente a `reduce combine val (map base s)`.
NOTA: El algoritmo de `reduce` combina siempre al final la solución final con `val`, luego ambas implementaciones pueden no ser equivalentes, sin embargo lo usual es que `val` sea el elemento neutro del operador de combinado, en cuyo caso son equivalentes.

5.5 Operación `scan`

Otra operación importante en secuencias es la operación `scan`. Este operador tiene cierta relación con `reduce` y la asociatividad del operador de reducción. Si $\oplus$ es asociativa, imponemos la especificación

$$\texttt{scan}\ \oplus\ \texttt{e s} = \texttt{(tabulate (}\lambda\texttt{i}\rightarrow\texttt{reduce}\ \oplus\ \texttt{e (take s i)) |s|, reduce}\ \oplus\ \texttt{e s)}$$

La idea entonces es que el comportamiento de `scan` para operadores asociativos sea evaluar a la secuencia de reducciones de los prefijos de `s` junto con la reducción completa de esta. Nuevamente, si bien esto funciona como implementación, lo tomamos solo como una especificación ya que el objetivo es implementarlo con una mejor profundidad.
Lo que haremos será implementar este algoritmo con un orden de reducción que nos permita la mayor paralelización posible a cambio de perder utilidad con varios operadores que no son asociativos. Esto es ya que el orden de reducción no solo deja de ser sencillo, sino que además cambiará para cada elemento de la secuencia, luego es prácticamente imposible hallar un operador no asociativo que sea útil con estas reducciones.

Para implementar `scan` usaremos una estrategia similar a Divide&Conquer que usa una técnica llamada contracción y expansión. La contracción es idéntica a la contracción de `reduce` que se basa en operar elementos adyacentes, y sobre esta secuencia contraída aplicar recursivamente la función. Ahora, teniendo tanto esta secuencia como la secuencia de elementos original podemos rellenar cada hueco restante con como máximo una aplicación del operador. Lo bueno de esta reducción es que cada contracción reduce la secuencia a la mitad. Es cierto que la expansión también tomará una cantidad de operaciones proporcional a la longitud de la secuencia, pero todos estos cálculos son independientes, luego con una buena estructura se puede lograr la expansión con profundidad $O(1)$. Esta estructura pueden ser los arreglos persistentes, como se ven en la última sección.

Para explicar el algoritmo usaremos la secuencia $s = \langle x_0,\, x_1,\, x_2\, x_3\rangle$ y un operador asociativo $\oplus$. La contracción de s nos da la secuencia $\langle x_0 \oplus x_1,\, x_2 \oplus x_3\rangle$ y entonces $s' = \texttt{scan}\oplus\ e\,\langle x_0 \oplus x_1,\, x_2 \oplus x_3\rangle = (\langle e, e \oplus x_0 \oplus x_1\rangle,\, e \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$. Queremos con estas dos secuencias poder construir $r = \texttt{scan}\ \oplus\ e\, s$

$$\begin{aligned} s =& \langle x_0,\, x_1,\, x_2\, x_3\rangle \\ s' =& (\langle e,\quad e \oplus x_0 \oplus x_1 \quad\rangle,\, e \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3) \\ r =& (\langle e,\, e \oplus x_0,\quad e \oplus x_0 \oplus x_1,\, e \oplus x_0 \oplus x_1 \oplus x_2\rangle,\, e \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3) \end{aligned}$$

Luego la reducción total se obtiene de la llamada recursiva, y la secuencia de adentro cumple:

- $r_i = s'_{i/2}$ si i es par
- $r_i = s'_{\lfloor i/2\rfloor} \oplus s_{i-1}$ si i es impar

5.6 Operación `collect`

Para `collect` requerimos un tipo ordenado `a` que actuará de llave, y un tipo cualquiera `b` que actúa de valor. La idea de esta operación es transformar una secuencia en una secuencia estructurada como un diccionario. Esta última debe estar ordenada por las llaves. Un ejemplo de uso sería:
`collect` ⟨`(3, "EDyAII"), (1, "Prog1"), (2, "Prog2"), (2, "EDyAI"), (3, "SOI")`⟩
`=` ⟨`(1,` ⟨`"Prog1"`⟩`), (2,` ⟨`"Prog2", "EDyAI"`⟩`), (3,` ⟨`"EDyAII", "SOI"`⟩`)`⟩

Esta operación se puede implementar en dos pasos:

1. Ordenar la secuencia original según las claves
2. Juntar todos los valores de claves iguales

Si W_c y S_c son el trabajo y profundidad de la comparación de claves respectivamente, `sort` puede implementarse con $W(n) \in O\left(W_c \cdot n \lg n\right)$ y $S(n) \in O\left(S_c \cdot \lg^2 n\right)$, mientras que juntar

las claves es $W(n) \in O(n)$ y $S(n) \in O(\lg n)$, luego el costo de `collect` es dominado por el ordenamiento.
Esta operación es muy usada en un patrón de alto orden llamado `mapCollectReduce` que se basa en un map a pares key-value, un collect de todos y finalmente una serie de reducciones. Esta operación suele tener la siguiente estructura, donde `apv` y `red` son generalmente funciones secuenciales:

```
mapCollectReduce :: (Seq s, Ord b) =>
    (s a -> s (b, c)) -> ((b, s c) -> (b, d)) -> s (s a) -> s (b, d)
mapCollectReduce apv red s = let pairs = join (map apv s)
                                 groups = collect pairs
                             in map red groups
```

5.7 Arreglos persistentes

Los arreglos persistentes son la variante funcional a los arreglos efímeros comunes en lenguajes imperativos. La mayor ventaja que presenta esta estructura en Haskell es su altísimo grado de paralelización, motivo por el cuál usaremos esta estructura para implementar `reduce` y `scan`.

Las siguientes dos definiciones ayudan al análisis de costos de `reduce` y `scan`, ya que como se mencionó, estos costos dependen del orden de reducción hechos.

$\mathcal{O}_r(\oplus, e, s) = \{$aplicaciones de $\oplus$ en el árbol de reducción$\}$ y de forma similar $\mathcal{O}_s(\oplus, e, s)$ es el conjunto de aplicaciones de $\oplus$ en `scan` $\oplus$ `e s`.

Las implementación de las funciones de alto orden en arreglos tienen los siguientes costos:

Operación	W	S
`tabulate` $f\ n$	$O\left(\sum_{i=0}^{n-1} W(f\ i)\right)$	$O\left(\max_{i=0}^{n-1} S(f\ i)\right)$
`map` $f\ s$	$O\left(\sum_{x \in s} W(f\ x)\right)$	$O\left(\max_{x \in s} S(f\ x)\right)$
`filter` $p\ s$	$O\left(\sum_{x \in s} W(p\ x)\right)$	$O\left(\lg \lvert s\rvert + \max_{x \in s} S(p\ x)\right)$
`reduce` $\oplus\ e\ s$	$O\left(\sum_{(x \oplus y) \in \mathcal{O}_r(\oplus, e, s)} W(x \oplus y)\right)$	$O\left(\lg \lvert s\rvert \max_{(x \oplus y) \in \mathcal{O}_r(\oplus, e, s)} S(x \oplus y)\right)$
`scan` $\oplus\ e\ s$	$O\left(\sum_{(x \oplus y) \in \mathcal{O}_s(\oplus, e, s)} W(x \oplus y)\right)$	$O\left(\lg \lvert s\rvert \max_{(x \oplus y) \in \mathcal{O}_s(\oplus, e, s)} S(x \oplus y)\right)$

Tabla 2: Cotas asintóticas O de funciones de alto orden en arreglos persistentes

6 Bibliografía

Slides de la cátedra, Año 2023.

Introduction to Algorithms, 3rd Edition, 2009, Thomas H. Cormen

Purely Functional Data Structures, Chris Okasaki