

Algorítmica

Conrado Martínez
U. Politècnica Catalunya

4 de mayo de 2020





Queda prohibida la reproducción parcial o la modificación, por cualesquiera medios, de estas transparencias sin el consentimiento expreso de los autores. Puede accederse a la copia principal de este documento electrónico en la web, enlazarse, realizarse cuántas copias se desee y difundirlas públicamente a través de la web u otros medios, siempre y cuando no se altere su contenido en absoluto.

El consentimiento para la reproducción, modificación o uso total o parcial de estas transparencias con fines exclusivamente docentes está automáticamente garantizado, tanto en la Univ. Politècnica de Catalunya (UPC) como en cualquier otra institución de enseñanza, superior o no, de cualquier país y siempre y cuando: a) se respete el debido crédito a los autores; b) los derechos de copia, reproducción, modificación y uso total o parcial aplicables a los materiales derivados sean los mismos aquí expresados. Por favor, envie un e-mail a [alg -at- lsi.upc.edu](mailto:alg-at-lsi.upc.edu) si está interesado en la reproducción, modificación o uso total o parcial de estas transparencias con cualquier fin, incluídos los fines docentes.

Syllabus

- Part 1: Repaso de Conceptos Algorítmicos
- Part 2: Algoritmos Voraces
- Part 3: Programación Dinámica
- Part 4: Flujos sobre Redes y Programación Lineal
- Part 5: Estructuras de Datos Avanzadas

Parte I

Repaso de Conceptos Algorítmicos

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- Tablas de Hash
- Colas de prioridad
- Grafos y Recorridos

- Eficiencia de un algoritmo = consumo de recursos de cómputo: tiempo de ejecución y espacio de memoria
- Análisis de algoritmos → Propiedades sobre la eficiencia de algoritmos
 - Comparar soluciones algorítmicas alternativas
 - Predecir los recursos que usará un algoritmo o ED
 - Mejorar los algoritmos o EDs existentes y guiar el diseño de nuevos algoritmos

En general, dado un algoritmo A cuyo conjunto de entradas es \mathcal{A} su **eficiencia** o **coste** (en tiempo, en espacio, en número de operaciones de E/S, etc.) es una función T de \mathcal{A} en \mathbb{N} (o \mathbb{Q} o \mathbb{R} , según el caso):

$$\begin{aligned} T : \mathcal{A} &\rightarrow \mathbb{N} \\ \alpha &\rightarrow T(\alpha) \end{aligned}$$

Ahora bien, caracterizar la función T puede ser muy complicado y además proporciona información inmanejable, difícilmente utilizable en la práctica.

Sea \mathcal{A}_n el conjunto de entradas de tamaño n y $T_n : \mathcal{A}_n \rightarrow \mathbb{N}$ la función T restringida a \mathcal{A}_n .

- *Coste en caso mejor:*

$$T_{\text{mejor}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Coste en caso peor:*

$$T_{\text{peor}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Coste promedio:*

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha) T_n(\alpha) \\ &= \sum_{k \geq 0} k \Pr(T_n = k). \end{aligned}$$

- 1 Para todo $n \geq 0$ y para cualquier $\alpha \in \mathcal{A}_n$

$$T_{\text{mejor}}(n) \leq T_n(\alpha) \leq T_{\text{peor}}(n).$$

- 2 Para todo $n \geq 0$

$$T_{\text{mejor}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{peor}}(n).$$

Estudiaremos generalmente sólo el coste en caso peor:

- ① Proporciona garantías sobre la eficiencia del algoritmo, el coste **nunca** excederá el coste en caso peor
- ② Es más fácil de calcular que el coste promedio

Una característica esencial del coste (en caso peor, en caso mejor, promedio) es su **tasa de crecimiento**

Ejemplo

- ① Funciones lineales: $f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$
- ② Funciones cuadráticas:

$$q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$$

Se dice que las funciones lineales y las cuadráticas tienen tasas de crecimiento distintas. También se dice que son de **órdenes de magnitud** distintos.

| $\log_2 n$ | n | $n \log_2 n$ | n^2 | n^3 | 2^n |
|------------|------|--------------|-------|--------|----------------------|
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4096 | 262144 |
| 5 | 32 | 160 | 1024 | 32768 | $6,87 \cdot 10^{10}$ |
| 6 | 64 | 384 | 4096 | 262144 | $4,72 \cdot 10^{21}$ |
| ... | | | | | |
| ℓ | N | L | C | Q | E |
| $\ell + 1$ | $2N$ | $2(L + N)$ | $4C$ | $8Q$ | E^2 |

Los factores constantes y los términos de orden inferior son irrelevantes desde el punto de vista de la tasa de crecimiento:
p.e. $30n^2 + \sqrt{n}$ tiene la misma tasa de crecimiento que
 $2n^2 + 10n \Rightarrow$ notación asintótica

Definición

Dada una función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ la clase $\mathcal{O}(f)$ (O-grande de f) es

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

En palabras, una función g está en $\mathcal{O}(f)$ si existe una constante c tal que $g < c \cdot f$ para toda n a partir de un cierto punto (n_0).

Aunque $\mathcal{O}(f)$ es un conjunto de funciones por tradición se escribe a veces $g = \mathcal{O}(f)$ en vez de $g \in \mathcal{O}(f)$. Sin embargo, $\mathcal{O}(f) = g$ no tiene sentido.

Propiedades básicas de la notación \mathcal{O} :

- ① Si $\lim_{n \rightarrow \infty} g(n)/f(n) < +\infty$ entonces $g = \mathcal{O}(f)$
- ② Es reflexiva: para toda función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, $f = \mathcal{O}(f)$
- ③ Es transitiva: si $f = \mathcal{O}(g)$ y $g = \mathcal{O}(h)$ entonces $f = \mathcal{O}(h)$
- ④ Para toda constante $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

Los factores constantes no son relevantes en la notación asintótica y los omitiremos sistemáticamente: p.e. hablaremos de $\mathcal{O}(n)$ y no de $\mathcal{O}(4 \cdot n)$; no expresaremos la base de logaritmos ($\mathcal{O}(\log n)$), ya que podemos pasar de una base a otra multiplicando por el factor apropiado:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

Otras notaciones asintóticas son Ω (omega) y Θ (zeta). La primera define un conjunto de funciones acotada inferiormente por una dada:

$$\Omega(f) = \{g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c > 0 \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

La notación Ω es reflexiva y transitiva; si $\lim_{n \rightarrow \infty} g(n)/f(n) > 0$ entonces $g = \Omega(f)$. Por otra parte, si $f = \mathcal{O}(g)$ entonces $g = \Omega(f)$ y viceversa.

Se dice que $\mathcal{O}(f)$ es la clase de las funciones que crecen no más rápido que f . Análogamente, $\Omega(f)$ es la clase de las funciones que crecen no más despacio que f .

Finalmente,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

es la clase de la funciones con la misma tasa de crecimiento que f .

La notación Θ es reflexiva y transitiva, como las otras. Es además simétrica: $f = \Theta(g)$ si y sólo si $g = \Theta(f)$. Si $\lim_{n \rightarrow \infty} g(n)/f(n) = c$ donde $0 < c < \infty$ entonces $g = \Theta(f)$.

Propiedades adicionales de las notaciones asintóticas (las inclusiones son estrictas):

- ① Para cualesquiera constantes $\alpha < \beta$, si f es una función creciente entonces $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$.
- ② Para cualesquiera constantes a y $b > 0$, si f es creciente, $\mathcal{O}((\log f)^a) \subset \mathcal{O}(f^b)$.
- ③ Para cualquier constante $c > 1$, si f es creciente, $\mathcal{O}(f) \subset \mathcal{O}(c^f)$.

Los operadores convencionales (sumas, restas, divisiones, etc.) sobre clases de funciones definidas mediante una notación asintótica se extienden de la siguiente manera:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

donde A y B son conjuntos de funciones. Expresiones de la forma $f \otimes A$ donde f es una función se entenderá como $\{f\} \otimes A$.

Este convenio nos permite escribir de manera cómoda expresiones como $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, ó $\Theta(1) + \mathcal{O}(1/n)$.

Regla de las sumas:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Regla de los productos:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Reglas similares se cumplen para las notaciones \mathcal{O} y Ω .

Análisis de algoritmos iterativos

- ➊ El coste de una operación elemental es $\Theta(1)$.
- ➋ Si el coste de un fragmento S_1 es f y el de S_2 es g entonces el coste de $S_1; S_2$ es $f + g$.
- ➌ Si el coste de S_1 es f , el de S_2 es g y el coste de evaluar B es h entonces el coste en caso peor de

```
if  $B$  then  $S_1$ 
else  $S_2$ 
end if
```

es $\max\{f + h, g + h\}$.

- 4 Si el coste de S durante la i -ésima iteración es f_i , el coste de evaluar B es h_i y el número de iteraciones es g entonces el coste T de

while B **do**

S

end while

es

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

Si $f = \max\{f_i + h_i\}$ entonces $T = \mathcal{O}(f \cdot g)$.

Análisis de algoritmos recursivos

El coste (en caso peor, medio, ...) de un algoritmo recursivo $T(n)$ satisface, dada la naturaleza del algoritmo, una ecuación **recurrente**: esto es, $T(n)$ dependerá del valor de T para tamaños menores. Frecuentemente, la recurrencia adopta una de las dos siguientes formas:

$$T(n) = a \cdot T(n - c) + g(n),$$
$$T(n) = a \cdot T(n/b) + g(n).$$

La primera corresponde a algoritmos que tiene una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño $n - c$, donde c es una constante. La segunda corresponde a algoritmos que tienen una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño (aproximadamente) n/b , donde $b > 1$.

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n - c) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $c \geq 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$. Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1. \end{cases}$$

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $b > 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.
Sea $\alpha = \log_b a$. Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k. \end{cases}$$

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- **Divide y vencerás**
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- Tablas de Hash
- Colas de prioridad
- Grafos y Recorridos

Introducción

El principio básico de **divide y vencerás** (en inglés, *divide and conquer*; en catalán, *dividir per conquerir*) es muy simple:

- ① Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.
- ② En caso contrario, se *divide* o *fragmenta* el ejemplar dado en subejemplares x_1, \dots, x_k y se resuelve, independiente y recursivamente, el problema para cada uno de ellos.
- ③ Las soluciones obtenidas y_1, \dots, y_k se *combinan* para obtener la solución al ejemplar original.

- El esquema de divide y vencerás expresado en pseudocódigo tiene el siguiente aspecto:

```
procedure DIVIDE_Y_VENCERAS( $x$ )
    if  $x$  es simple then
        return SOLUCION_DIRECTA( $x$ )
    else
         $\langle x_1, x_2, \dots, x_k \rangle := \text{DIVIDE}(x)$ 
        for :=1 to  $k$  do
             $y_i := \text{DIVIDE\_Y\_VENCERAS}(x_i)$ 
        end for
        return COMBINA( $y_1, y_2, \dots, y_k$ )
    end if
end procedure
```

- Ocasionalmente, si $k = 1$, se habla del esquema de *reducción*.

El esquema de divide y vencerás se caracteriza adicionalmente por las siguientes propiedades:

- No se resuelve más de una vez un mismo subproblema
- El tamaño de los subejemplares es, en promedio, una fracción del tamaño del ejemplar original, es decir, si x es de tamaño n , el tamaño esperado de un subejemplar cualquiera x_i es n/c_i donde $c_i > 1$. Con frecuencia, esta condición se cumplirá siempre, no sólo en promedio.

Coste de los algoritmos divide y vencerás

El análisis de un algoritmo divide y vencerás requiere, al igual que ocurre con otros algoritmos recursivos, la resolución de recurrencias. En su caso más simple las recurrencias son de la forma

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{si } n > n_0, \\ b(n) & \text{si } n \leq n_0. \end{cases}$$

Este tipo de recurrencia corresponde al coste $T(n)$ de un algoritmo donde conocemos la solución directa siempre que n , el tamaño de x , sea menor o igual a n_0 , y en caso contrario, se descompone el ejemplar en subejemplares cuyo tamaño es aproximadamente n/b , haciendo a llamadas recursivas, siendo $f(n)$ el coste conjunto de las funciones de dividir y de combinar. Si bien es habitual que $a = b$, existen muchas situaciones en las que esto no sucede.

Teorema

Si $g(n) = \Theta(n^k)$ entonces la solución de la recurrencia

$$T(n) = \begin{cases} g(n) + a \cdot T(n/b) & \text{si } n > n_0, \\ f(n) & \text{si } n \leq n_0, \end{cases}$$

donde $a \geq 1$ y $b > 1$ son constantes, satisface

$$T(n) = \begin{cases} \Theta(g(n)) & \text{si } \alpha < k, \\ \Theta(g(n) \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k, \end{cases}$$

siendo $\alpha = \log_b a$.

Demostraci \ddot{o} n: Supondremos que $n = n_0 \cdot b^j$. Aplicando la recurrencia repetidamente,

$$\begin{aligned} T(n) &= gf(n) + a \cdot T(n/b) \\ &= g(n) + a \cdot f(n/b) + a^2 \cdot T(n/b^2) \\ &= \dots \\ &= g(n) + a \cdot g(n/b) + \dots + a^{j-1} \cdot g(n/b^{j-1}) \\ &\quad + a^j \cdot T(n/b^j) \\ &= \sum_{0 \leq i < j} a^i \cdot g(n/b^i) + a^j \cdot T(n_0) \\ &= \Theta \left(\sum_{0 \leq i < j} a^i \left(\frac{n}{b^i} \right)^k \right) + a^j \cdot f(n_0) \\ &= \Theta \left(n^k \cdot \sum_{0 \leq i < j} \left(\frac{a}{b^k} \right)^i \right) + a^j \cdot f(n_0). \end{aligned}$$

Puesto que $f(n_0)$ es una constante y tenemos que el segundo término es $\Theta(n^\alpha)$ ya que

$$\begin{aligned} a^j &= a^{\log_b(n/n_0)} = a^{\log_b n} \cdot a^{-\log_b n_0} \\ &= \Theta(b^{\log_b a \cdot \log_b n}) \\ &= \Theta(n^{\log_b a}) = \Theta(n^\alpha). \end{aligned}$$

Ahora tenemos tres casos diferentes a considerar: según que a/b^k sea menor, igual o mayor que 1. Alternativamente, ya que $\alpha = \log_b a$, según que α sea mayor, igual o menor que k . Si $k > \alpha$ (equivalentemente, $a/b^k < 1$) entonces la suma que aparece en el primer término es una serie geométrica acotada por una constante, es decir, el primer término es $\Theta(n^k)$. Como asumimos que $k > \alpha$, es el primer término el que domina y $T(n) = \Theta(n^k) = \Theta(g(n))$.

Si $\alpha = k$, tendremos que $a/b^k = 1$ y la suma vale j ; ya que $j = \log_b(n/n_0) = \Theta(\log n)$, concluimos que $T(n) = \Theta(n^k \cdot \log n) = \Theta(g(n) \cdot \log n)$. Finalmente, si $k < \alpha$ entonces $a/b^k > 1$ y el valor de la suma es

$$\begin{aligned} \sum_{0 \leq i < j} \left(\frac{a}{b^k} \right)^i &= \frac{(a/b^k)^j - 1}{a/b^k - 1} = \Theta \left(\left(\frac{a}{b^k} \right)^j \right) \\ &= \Theta \left(\frac{n^\alpha}{n^k} \right) = \Theta(n^{\alpha-k}). \end{aligned}$$

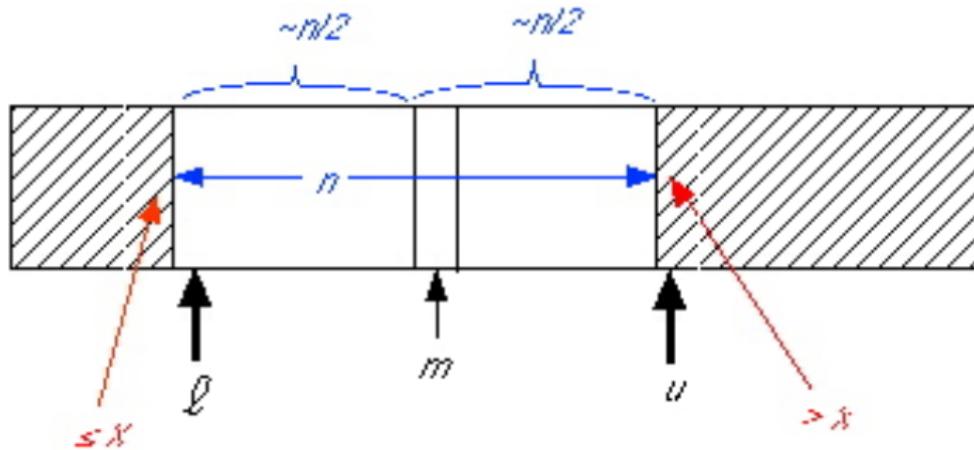
Por tanto, es el segundo término el que en este caso domina y $T(n) = \Theta(n^\alpha)$.

Búsqueda dicotómica

Problema: Dado un vector $A[1..n]$ de n elementos, en orden creciente, y un elemento x , determinar el valor i , $0 \leq i \leq n$, tal que $A[i] \leq x < A[i + 1]$ (convendremos que $A[0] = -\infty$ y $A[n + 1] = +\infty$).

Si el vector A fuese vacío la respuesta es sencilla, pues x no está en el vector. Pero para poder obtener una solución recursiva efectiva, debemos realizar la siguiente generalización: dado un segmento o subvector $A[\ell + 1..u - 1]$, $0 \leq \ell \leq u \leq n + 1$, ordenado crecientemente, y tal que $A[\ell] \leq x < A[u]$, determinar el valor i , $\ell \leq i \leq u - 1$, tal que $A[i] \leq x < A[i + 1]$. Este es un ejemplo clásico de *inmersión de parámetros*.

Ahora sí, si $\ell + 1 = u$, el segmento en cuestión no contiene elementos y $i = u - 1 = \ell$.



▷ Llamada inicial: **BINSEARCH**($A, x, 0, n + 1$)

procedure **BINSEARCH**(A, x, ℓ, u)

Require: $0 \leq \ell < u \leq n + 1, A[\ell] \leq x < A[u]$

Ensure: Retorna i tal que $A[i] \leq x < A[i + 1]$ y $\ell \leq i < u$

if $\ell + 1 = u$ **then**

return ℓ

else

$m := (\ell + u) \text{ div } 2$

if $x < A[m]$ **then**

return **BINSEARCH**(A, x, ℓ, m)

else

return **BINSEARCH**(A, x, m, u)

end if

end if

end procedure

La llamada inicial es $\text{BINSEARCH}((A, x, 0, A.\text{SIZE}() + 1))$, donde $A.\text{SIZE}()$ nos da n , el número de elementos del vector A .

Prescindiendo de la pequeña diferencia entre el tamaño real de los subejemplares y el valor $n/2$, el coste de la búsqueda binaria o dicotómica viene descrito por la recurrencia

$$B(n) = \Theta(1) + B(n/2), \quad n > 0,$$

y $B(0) = b_0$, ya que en cada llamada recursiva sólo se hace otra, sea en el subvector a la izquierda, sea el subvector a la derecha del punto central. Empleando el Teorema 1, $\alpha = \log_2 1 = 0$ y por lo tanto $B(n) = \Theta(\log n)$.

Algoritmo de Karatsuba y Ofmann

El algoritmo tradicional de multiplicación tiene coste $\Theta(n^2)$ para multiplicar dos enteros de n bits, ya la suma de dos enteros de $\Theta(n)$ bits tiene coste $\Theta(n)$.

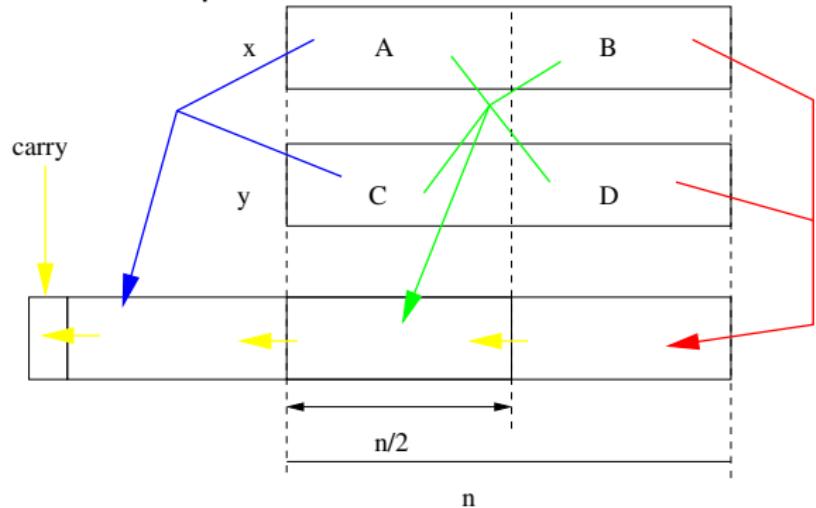
$$\begin{aligned} 231 \times 659 &= 9 \times 231 + 5 \times 231 \times 10 \\ &\quad + 6 \times 231 \times 100 \\ &= 2079 + 11550 + 138600 \\ &= 152229 \end{aligned}$$

También es de coste cuadrático el algoritmo de multiplicación *a la rusa*:

```
 $z := 0$ 
▷  $x = X \wedge y = Y$ 
while  $x \neq 0$  do
    ▷ Inv:  $z + x \cdot y = X \cdot Y$ 
    if  $x$  es par then
         $x := x \text{ div } 2; y := 2 \cdot y$ 
    else
         $x := x - 1; z := z + y$ 
    end if
end while
▷  $z = X \cdot Y$ 
```

Supongamos que x e y son dos números positivos de $n = 2^k$ bits (si n no es una potencia de 2 ó x e y no tienen ambos la misma longitud, podemos añadir 0's por la izquierda para que así sea).

Una idea que no funciona:



$$\begin{aligned}
 x \cdot y &= (A \cdot 2^{n/2} + B) \cdot (C \cdot 2^{n/2} + D) \\
 &= A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D.
 \end{aligned}$$

Efectuada la descomposición $x = \langle A, B \rangle$ y $y = \langle C, D \rangle$, se calculan 4 productos ($A \cdot B$, $A \cdot D$, $B \cdot C$ y $B \cdot D$) y se combinan las soluciones mediante sumas y desplazamientos (*shifts*). El coste de estas operaciones es $\Theta(n)$. Por lo tanto, el coste de la multiplicación mediante este algoritmo divide y vencerás viene dado por la recurrencia

$$M(n) = \Theta(n) + 4 \cdot M(n/2), \quad n > 1$$

cuya solución es $M(n) = \Theta(n^2)$. En este caso $\alpha = 1$, $a = 4$ y $b = 2$, y $\alpha = 1 < c = \log_2 4 = 2$.

El algoritmo de Karatsuba y Ofmann (1962) realiza la multiplicación dividiendo los dos números como antes pero se realizan las siguientes 3 multiplicaciones (recursivamente)

$$U = A \cdot C$$

$$V = B \cdot D$$

$$W = (A + B) \cdot (C + D)$$

Y el resultado se calcula a partir de las tres soluciones obtenidas

$$x \cdot y = U \cdot 2^n + (W - (U + V)) \cdot 2^{n/2} + V.$$

El algoritmo requiere realizar 6 adiciones (una de ellas es de hecho una resta) frente a las 3 adiciones que se empleaban en el algoritmo anterior. El coste mejora pues el coste de las funciones de división y de combinación del algoritmo de Karatsuba y Ofmann sigue siendo lineal y disminuye el número de llamadas recursivas.

$$M(n) = \Theta(n) + 3 \cdot M(n/2)$$

$$M(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1,5849625\dots})$$

La constante oculta en la notación asintótica es grande y la ventaja de este algoritmo frente a los algoritmos básicos no se manifiesta hasta que n es relativamente grande (del orden de 200 a 250 bits por multiplicando).

procedure **MULT**(x, y, i, j, i', j')

Require: $1 \leq i \leq j \leq n, 1 \leq i' \leq j' \leq n, j' - i' = j - i$

Ensure: Devuelve el producto de $x[i..j]$ por $y[i'..j']$

$n := j - i + 1$

if $n < M$ **then**

 Usar un método simple para calcular el resultado

else

$m := (i + j) \text{ div } 2; m' := (i' + j') \text{ div } 2$

$U := \text{MULT}(x, y, m + 1, j, m' + 1, j')$

$V := \text{MULT}(x, y, i, m, i', m')$

$U.\text{DECALA_IZQUIERDA}(n)$

$W_1 := x[i..m] + y[i'..m']$

$W_2 := x[m + 1..j] + y[m' + 1..j']$

 Añadir un bit a W_1 ó W_2 si es necesario

$W := \text{MULT}(W_1, W_2, \dots)$

$W := W - (U + V)$

$W.\text{DECALA_IZQUIERDA}(n \text{ div } 2)$

return $U + W + V$

end if

end procedure

Ordenación por fusión

El algoritmo de ordenación por fusión (**mergesort**) fue uno de los primeros algoritmos eficientes de ordenación propuestos. Diversas variantes de este algoritmo son particularmente útiles para la ordenación de datos residentes en memoria externa. El propio *mergesort* es un método muy eficaz para la ordenación de listas enlazadas.

La idea básica es simple: se divide la secuencia de datos a ordenar en dos subsecuencias de igual o similar tamaño, se ordena recursivamente cada una de ellas, y finalmente se obtiene una secuencia ordenada fusionando las dos subsecuencias ordenadas. Si la secuencia es suficientemente pequeña puede emplearse un método más simple (y eficaz para entradas de tamaño reducido).

Supondremos que nuestra entrada es una lista enlazada L conteniendo una secuencia de elementos x_1, \dots, x_n . Cada elemento se almacena en un nodo con dos campos: `info` contiene el elemento y `next` es un apuntador al siguiente nodo de la lista (indicaremos que un nodo no tiene sucesor con el valor especial `next = null`). La propia lista L es, de hecho, un apuntador a su primer nodo. La notación *apuntador* → *campo* indica el campo referido del objeto apuntado como en C y C++. Es equivalente a la expresión $p^.campo$ de Pascal y Modula-2.

procedure SPLIT(L, L', n)

Require: $L = [\ell_1, \dots, \ell_m], m \geq n$

Ensure: $L = [\ell_1, \dots, \ell_n], L' = [\ell_{n+1}, \dots, \ell_m]$

$p := L$

while $n > 1$ **do**

$p := p \rightarrow next$

$n := n - 1$

end while

$L' := p \rightarrow next; p \rightarrow next := \text{null}$

end procedure

procedure MERGESORT(L, n)

if $n > 1$ **then**

$m := n \text{ div } 2$

SPLIT(L, L', m)

MERGESORT(L, m)

MERGESORT($L', n - m$)

▷ fusiona las listas L y L'

$L := \text{MERGE}(L, L')$

end if

end procedure

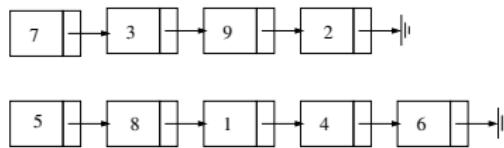
Para fusionar razonamos de la siguiente forma: Si L ó L' es vacía la lista resultante es la otra lista, es decir, la que eventualmente es no vacía. Si ambas son no vacías comparamos sus respectivos primeros elementos: el menor de los dos será el primer elemento de la lista fusionada resultante y a continuación vendrá la lista resultado de fusionar la sublista que sucede a ese primer elemento con la otra lista.

```
procedure MERGE( $L, L'$ )
    if  $L = \text{null}$  then return  $L'$ 
    end if
    if  $L' = \text{null}$  then return  $L$ 
    end if
    if  $L \rightarrow \text{info} \leq L' \rightarrow \text{info}$  then
         $L \rightarrow \text{next} := \text{MERGE}(L \rightarrow \text{next}, L')$ 
        return  $L$ 
    else
         $L' \rightarrow \text{next} := \text{MERGE}(L, L' \rightarrow \text{next})$ 
        return  $L'$ 
    end if
end procedure
```

Mergesort

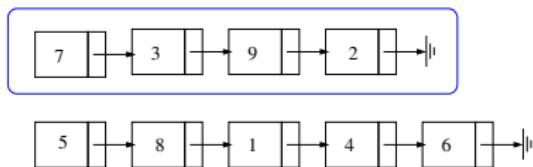


Mergesort

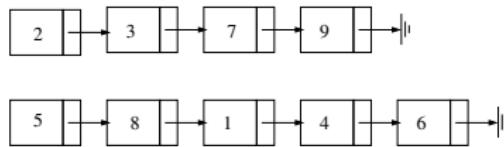


Mergesort

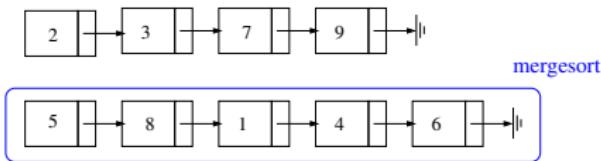
mergesort



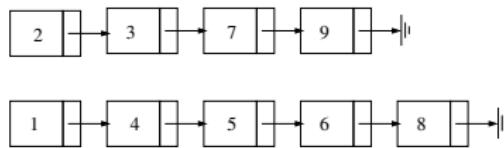
Mergesort



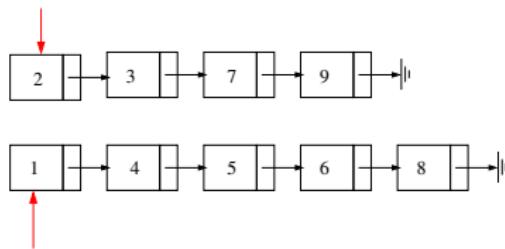
Mergesort



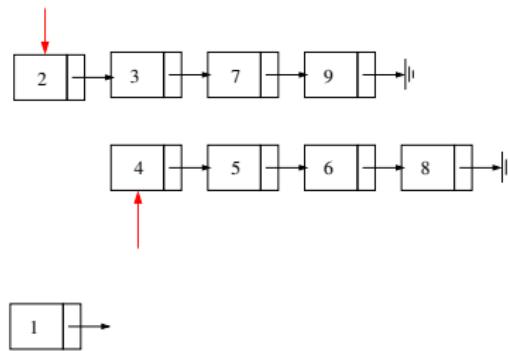
Mergesort



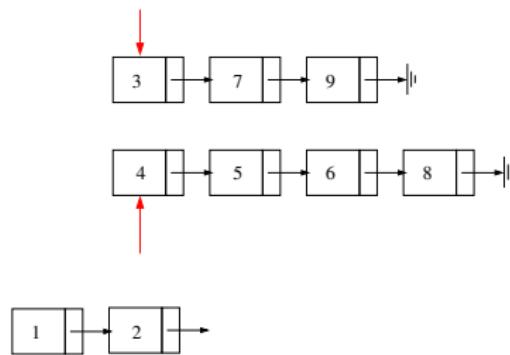
Mergesort



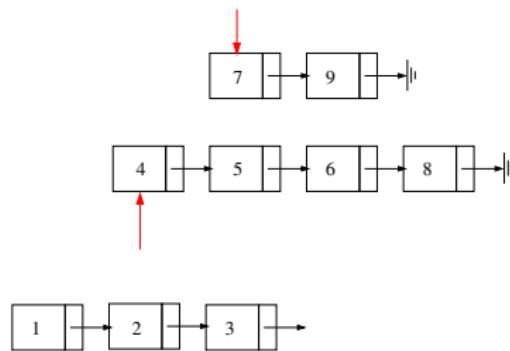
Mergesort



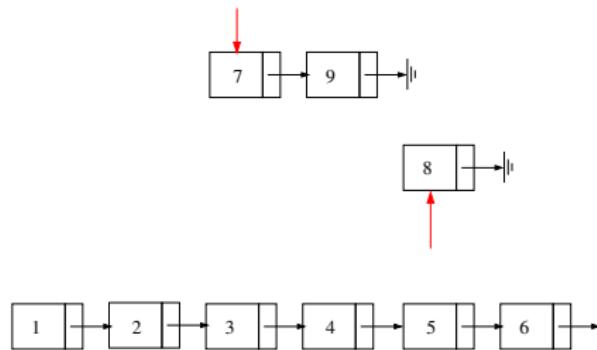
Mergesort



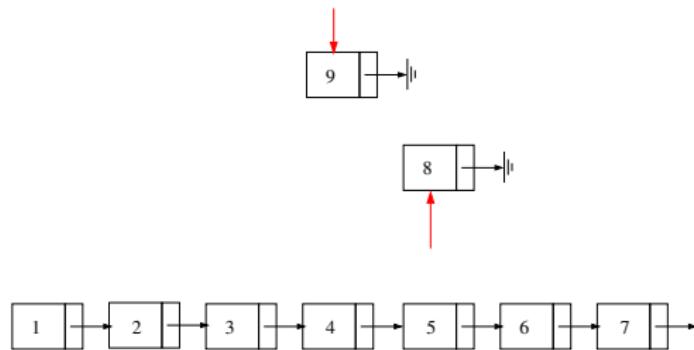
Mergesort



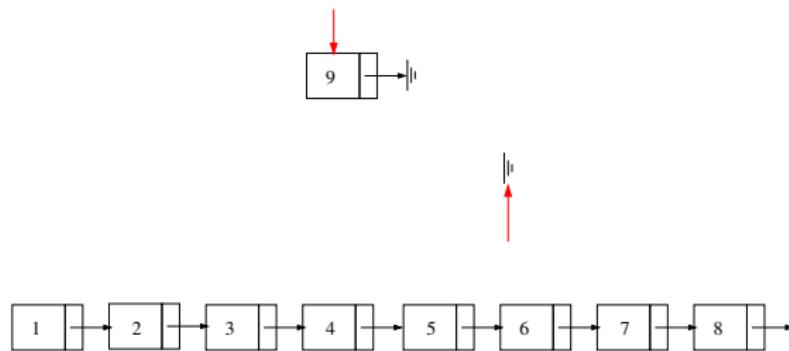
Mergesort



Mergesort



Mergesort



Mergesort



Cada elemento de L y L' es “visitado” exactamente una vez, por lo que el coste de la operación MERGE es proporcional a la suma de los tamaños de las listas L y L' ; es decir, su coste es $\Theta(n)$. La función definida es recursiva final y podemos obtener con poco esfuerzo una versión iterativa algo más eficiente.

Observese que si el primer elemento de la lista L es igual al primer elemento de L' se pone en primer lugar al que proviene de L , por lo que MERGESORT es un algoritmo *estable*.

El coste de MERGESORT viene descrito por la recurrencia

$$\begin{aligned} M(n) &= \Theta(n) + M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + M\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &= \Theta(n) + 2 \cdot M\left(\frac{n}{2}\right) \end{aligned}$$

cuya solución es $M(n) = \Theta(n \log n)$, aplicando el segundo caso del Teorema 1.

Algoritmo de Strassen

El algoritmo convencional de multiplicación de matrices tiene coste $\Theta(n^3)$ para multiplicar dos matrices $n \times n$.

```
for i := 1 to n do
    for j := 1 to n do
        C[i, j] := 0
        for k := 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j]
        end for
    end for
end for
```

Para abordar una solución con el esquema divide y vencerás se descomponen las matrices en bloques:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right)$$

donde $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$, etc.

Cada bloque de C requiere dos multiplicaciones de bloques de tamaño $n/2 \times n/2$ y dos sumas cuyo coste es $\Theta(n^2)$. El coste del algoritmo divide y vencerás así planteado tendría coste

$$M(n) = \Theta(n^2) + 8 \cdot M(n/2),$$

es decir, $M(n) = \Theta(n^3)$.

Para conseguir una solución más eficiente debemos reducir el número de llamadas recursivas.

Strassen (1969) propuso una forma de hacer justamente esto. En el libro de Brassard & Bratley se detalla una posible manera de hallar la serie de fórmulas que producen el resultado deseado. Un factor que complica las cosas es que la multiplicación de matrices no es conmutativa, a diferencia de lo que sucede con la multiplicación de enteros.

Se obtienen las siguientes 7 matrices $n/2 \times n/2$, mediante 7 productos y 14 adiciones/sustracciones

$$M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12})$$

$$M_2 = A_{11} \cdot B_{11}$$

$$M_3 = A_{12} \cdot B_{21}$$

$$M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$$

$$M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$$

Mediante 10 adiciones/sustracciones más, podemos obtener los bloques de la matriz resultante C :

$$C_{11} = M_2 + M_3$$

$$C_{12} = M_1 + M_2 + M_5 + M_6$$

$$C_{21} = M_1 + M_2 + M_4 - M_7$$

$$C_{22} = M_1 + M_2 + M_4 + M_5$$

Puesto que las operaciones aditivas tienen coste $\Theta(n^2)$, el coste del algoritmo de Strassen viene dado por la recurrencia

$$M(n) = \Theta(n^2) + 7 \cdot M(n/2),$$

cuya solución es $\Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$.

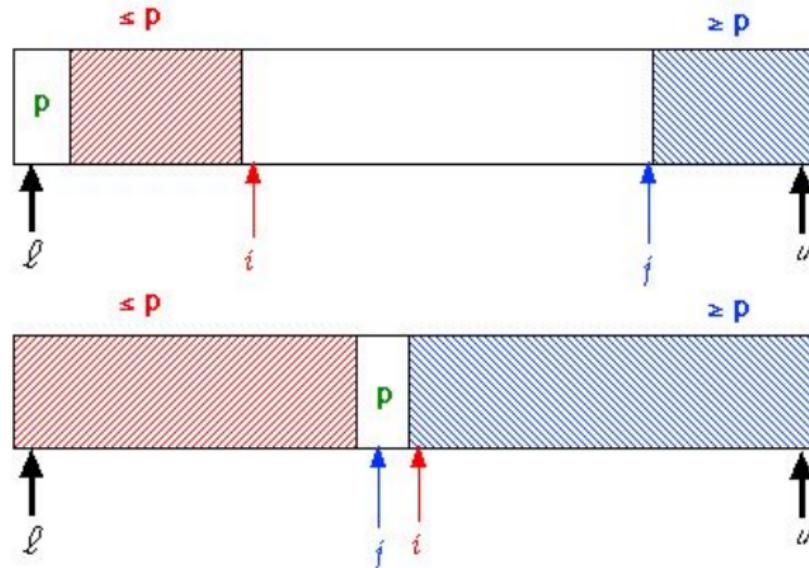
El algoritmo de Strassen tuvo un enorme impacto teórico ya que fue el primer algoritmo de multiplicación de matrices cuya complejidad era $o(n^3)$; lo cual tenía también implicaciones en el desarrollo de algoritmos más eficientes para el cálculo de matrices inversas, de determinantes, etc. Además era uno de los primeros casos en que las técnicas algorítmicas superaban lo que hasta el momento parecía una barrera infranqueable. En años posteriores se han obtenido algoritmos todavía más eficientes. El más eficiente conocido es el de Coppersmith y Winograd (1986) cuyo coste es $\Theta(n^{2,376\dots})$.

El algoritmo de Strassen no es competitivo en la práctica, excepto si n es muy grande ($n \gg 500$), ya que las constantes y términos de orden inferior del coste son muy grandes.

QuickSort

QUICKSORT (Hoare, 1962) es un algoritmo de ordenación que usa el principio de divide y vencerás, pero a diferencia de los ejemplos anteriores, no garantiza que cada subejemplar tendrá un tamaño que es fracción del tamaño original.

La base de *quicksort* es el procedimiento de PARTICIÓN: dado un elemento p denominado **pivote**, debe reorganizarse el segmento como ilustra la figura.



El procedimiento de partición sitúa a un elemento, el pivote, en su lugar apropiado. Luego no queda más que ordenar los segmentos que quedan a su izquierda y a su derecha. Mientras que en MERGESORT la división es simple y el trabajo se realiza durante la fase de combinación, en Quicksort sucede lo contrario.

Para ordenar el segmento $A[\ell..u]$ el algoritmo queda así

```
procedure QUICKSORT( $A, \ell, u$ )
Ensure: Ordena el subvector  $A[\ell..u]$ 
if  $u - \ell + 1 \leq M$  then
    usar un algoritmo de ordenación simple
else
    PARTICI $\ddot{\iota}$   $\frac{1}{2}N(A, \ell, u, k)$ 
     $\triangleright A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
    QUICKSORT( $(A, \ell, k - 1)$ )
    QUICKSORT( $(A, k + 1, u)$ )
end if
end procedure
```

En vez de usar un algoritmo de ordenación simple (p.e. ordenación por inserción) con cada segmento de M o menos elementos, puede ordenarse mediante el algoritmo de inserción al final:

`QUICKSORT(A , 1, $A.\text{SIZE}()$)`

`INSERTSORT(A , 1, $A.\text{SIZE}()$)`

Puesto que el vector A está quasi-ordenado tras aplicar `QUICKSORT`, el último paso se hace en tiempo $\Theta(n)$, donde $n = A.\text{SIZE}()$.

Se estima que la elección óptima para el umbral o corte de recursión M oscila entre 20 y 25.

Existen muchas formas posibles de realizar la partición. En Bentley & McIlroy (1993) se discute un procedimiento de partición muy eficiente, incluso si hay elementos repetidos. Aquí examinamos un algoritmo básico, pero razonablemente eficaz.

Se mantienen dos índices i y j de tal modo que $A[\ell + 1..i - 1]$ contiene elementos menores o iguales que el pivote p , y $A[j + 1..u]$ contiene elementos mayores o iguales. Los índices barren el segmento (de izquierda a derecha, y de derecha a izquierda, respectivamente), hasta que $A[i] > p$ y $A[j] < p$ o se cruzan ($i = j + 1$).

procedure PARTICI $\ddot{\text{o}}$ N(A, ℓ, u, k)

Require: $\ell \leq u$

Ensure: $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$

$i := \ell + 1; j := u; p := A[\ell]$

while $i < j + 1$ **do**

while $i < j + 1 \wedge A[i] \leq p$ **do**

$i := i + 1$

end while

while $i < j + 1 \wedge A[j] \geq p$ **do**

$j := j - 1$

end while

if $i < j + 1$ **then**

$A[i] := A[j]$

end if

end while

$A[\ell] := A[j]; k := j$

end procedure

El coste de QUICKSORT en caso peor es $\Theta(n^2)$ y por lo tanto poco atractivo en términos prácticos. Esto ocurre si en todos o la gran mayoría de los casos uno de los subsegmentos contiene muy pocos elementos y el otro casi todos, p.e. así sucede si el vector está ordenado creciente o decrecientemente. El coste de la partición es $\Theta(n)$ y entonces tenemos

$$\begin{aligned}Q(n) &= \Theta(n) + Q(n - 1) + Q(0) \\&= \Theta(n) + Q(n - 1) = \Theta(n) + \Theta(n - 1) + Q(n - 2) \\&= \dots = \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{0 \leq i \leq n} i\right) \\&= \Theta(n^2).\end{aligned}$$

Sin embargo, en promedio, el pivote quedará más o menos centrado hacia la mitad del segmento como sería deseable —justificando que *quicksort* sea considerado un algoritmo de divide y vencerás.

Para analizar el comportamiento de QUICKSORT sólo importa el orden relativo de los elementos. También podemos investigar exclusivamente el número de comparaciones entre elementos, ya que el coste total es proporcional a dicho número.

Supongamos que cualquiera de los $n!$ ordenes relativos posibles tiene idéntica probabilidad, y sea q_n el número medio de comparaciones.

$$\begin{aligned} q_n &= \sum_{1 \leq j \leq n} \mathbb{E}[\# \text{ compar.} \mid \text{pivot es } j\text{-ésimo}] \times \Pr\{\text{pivot es } j\text{-ésimo}\} \\ &= \sum_{1 \leq j \leq n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\ &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq j \leq n} (q_{j-1} + q_{n-j}) \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq j < n} q_j \end{aligned}$$

Para resolver esta recurrencia emplearemos el denominado *continuous master theorem* (CMT).

El CMT considera recurrencias de divide y vencerás con el siguiente formato

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j, \quad n \geq n_0$$

para un entero positivo n_0 , una función t_n , denominada *función de peaje*, y unos pesos $\omega_{n,j} \geq 0$. Los pesos deben satisfacer dos condiciones adicionales:

- ① $W_n = \sum_{0 \leq j < n} \omega_{n,j} \geq 1$
- ② $Z_n = \sum_{0 \leq j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1.$

El paso fundamental es hallar una *función de forma* $\omega(z)$ que aproxima los pesos $\omega_{n,j}$.

Definición

Dado un conjunto de pesos $\omega_{n,j}$, $\omega(z)$ es una función de forma para el conjunto de pesos si

① $\int_0^1 \omega(z) dz \geq 1$

- ② existe una constante $\rho > 0$ tal que

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) dz \right| = \mathcal{O}(n^{-\rho})$$

Un método simple y que funciona usualmente para calcular funciones de forma consiste en sustituir j por $z \cdot n$ en $\omega_{n,j}$, multiplicar por n y tomar el límite para $n \rightarrow \infty$.

$$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n}$$

Las extensiones a los números reales de muchas funciones discretas son inmediatas, p.e. $j^2 \rightarrow z^2$.

Para los números binomiales se puede emplear la aproximación

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

La extensión de los factoriales a los reales viene dada por la función gamma de Euler $\Gamma(z)$ y la de los números armónicos es la función $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.

Por ejemplo, en quicksort los pesos son todos iguales:

$\omega_{n,j} = \frac{2}{n}$. La función de forma correspondiente es
 $\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n} = 2$.

Teorema (Roura, 1997)

Sea F_n descrita por la recurrencia

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j,$$

sea $\omega(z)$ una función de forma correspondiente a los pesos $\omega_{n,j}$, y $t_n = \Theta(n^a(\log n)^b)$, para $a \geq 0$ y $b > -1$ constantes.

Sea $\mathcal{H} = 1 - \int_0^1 \omega(z)z^a dz$ y $\mathcal{H}' = -(b+1) \int_0^1 \omega(z)z^a \ln z dz$.

Entonces

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{si } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{si } \mathcal{H} = 0 \text{ y } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{si } \mathcal{H} < 0, \end{cases}$$

donde $x = \alpha$ es la única solución real no negativa de la ecuación

$$1 - \int_0^1 \omega(z)z^x dz = 0.$$

Consideremos q_n . Ya hemos visto que los pesos son $\omega_{n,j} = 2/n$ y $t_n = n - 1$. Por tanto $\omega(z) = 2$, $a = 1$ y $b = 0$. Puede comprobarse fácilmente que se dan todas las condiciones necesarias para la aplicación del CMT. Calculamos

$$\mathcal{H} = 1 - \int_0^1 2z \, dz = 1 - z^2 \Big|_{z=0}^{z=1} = 0,$$

por lo que tendremos que aplicar el caso 2 del CMT y calcular \mathcal{H}'

$$\mathcal{H}' = - \int_0^1 2z \ln z \, dz = \frac{z^2}{2} - z^2 \ln z \Big|_{z=0}^{z=1} = \frac{1}{2}.$$

Por lo tanto,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1,386 \dots n \log_2 n + o(n \log n). \end{aligned}$$

QuickSelect

El problema de la selección consiste en hallar el j -ésimo de entre n elementos dados. En concreto, dado un vector A con n elementos y un rango j , $1 \leq j \leq n$, un algoritmo de selección debe hallar el j -ésimo elemento en orden ascendente. Si $j = 1$ entonces hay que encontrar el mínimo, si $j = n$ entonces hay que hallar el máximo, si $j = \lfloor n/2 \rfloor$ entonces debemos hallar la mediana, etc.

Es fácil resolver el problema con coste $\Theta(n \log n)$ ordenando previamente el vector y con coste $\Theta(j \cdot n)$, recorriendo el vector y manteniendo los j elementos menores de entre los ya examinados. Con las estructuras de datos apropiadas puede rebajarse el coste a $\Theta(n \log j)$, lo cual no supone una mejora sobre la primera alternativa si $j = \Theta(n)$.

QUICKSELECT (Hoare, 1962), también llamado FIND y *one-sided* QUICKSORT, es una variante del algoritmo QUICKSORT para la selección del j -ésimo de entre n elementos.

Supongamos que efectuamos una partición de un subvector $A[\ell..u]$, conteniendo los elementos ℓ -ésimo a u -ésimo de A , y tal que $\ell \leq j \leq u$, respecto a un pivote p . Una vez finalizada la partición, supongamos que el pivote acaba en la posición k . Por tanto, en $A[\ell..k - 1]$ están los elementos ℓ -ésimo a $(k - 1)$ -ésimo de A y en $A[k + 1..u]$ están los elementos $(k + 1)$ -ésimo a u -ésimo. Si $j = k$ hemos acabado ya que hemos encontrado el elemento solicitado. Si $j < k$ entonces procedemos recursivamente en el subvector de la izquierda $A[\ell..k - 1]$ y si $j > k$ entonces encontraremos el elemento buscado en el subvector $A[k + 1..u]$.

procedure QUICKSELECT(A, ℓ, j, u)

Ensure: Retorna el $(j + 1 - \ell)$ - $\zeta^{\frac{1}{2}}$ simo menor elemento de $A[\ell..u]$,

$\ell \leq j \leq u$

if $\ell = u$ **then**

return $A[\ell]$

end if

PARTICIÓN $\zeta^{\frac{1}{2}}N(A, \ell, u, k)$

if $j = k$ **then**

return $A[k]$

end if

if $j < k$ **then**

return QUICKSELECT($A, \ell, j, k - 1$)

else

return QUICKSELECT($A, k + 1, j, u$)

end if

end procedure

Puesto que QUICKSELECT es recursiva final es muy simple obtener una versión iterativa eficiente que no necesita espacio auxiliar.

En caso peor, el coste de QUICKSELECT es $\Theta(n^2)$. Sin embargo, su coste promedio es $\Theta(n)$ donde la constante de proporcionalidad depende del cociente j/n . Knuth (1971) ha demostrado que $C_n^{(j)}$, el número medio de comparaciones necesarias para seleccionar el j -ésimo de entre n es:

$$\begin{aligned} C_n^{(j)} &= 2((n+1)H_n - (n+3-j)H_{n+1-j} \\ &\quad - (j+2)H_j + n+3) \end{aligned}$$

El valor máximo se alcanza para $j = \lfloor n/2 \rfloor$; entonces $C_n^{(j)} = 2(\ln 2 + 1)n + o(n)$.

Consideremos ahora el análisis del coste promedio C_n suponiendo que j adopta cualquier valor entre 1 y n con igual probabilidad.

$$C_n = n + \mathcal{O}(1)$$

$$+ \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{núm. de comp.} \mid \text{el pivote es el } k\text{-ésimo}] ,$$

puesto que el pivote es el k -ésimo con igual probabilidad para toda k .

La probabilidad de que $j = k$ es $1/n$ y entonces ya habremos acabado. La probabilidad de continuar a la izquierda es $(k - 1)/n$ y entonces se harán de hacer C_{k-1} comparaciones. Análogamente, con probabilidad $(n - k)/n$ se continuará en el subvector a la derecha y se harán C_{n-k} comparaciones.

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq k < n} \frac{k}{n} C_k. \end{aligned}$$

Aplicando el CMT con la función de forma

$$\lim_{n \rightarrow \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

obtenemos $\mathcal{H} = 1 - \int_0^1 2z^2 dz = 1/3$ y $C_n = 3n + o(n)$.

Podemos obtener un algoritmo cuyo coste en caso peor sea lineal si garantizamos que cada paso el pivote escogido divide el vector en dos subvectores cuya talla sea una fracción de la talla del vector original, con coste $\mathcal{O}(n)$ (incluyendo la selección del pivote). Entonces, en caso peor,

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

donde $p < 1$. Puesto que $\log_{1/p} 1 = 0 < 1$ concluimos que $C(n) = \mathcal{O}(n)$. Por otra parte, es bastante obvio que $C(n) = \Omega(n)$; luego, $C(n) = \Theta(n)$.

Este es el principio del algoritmo de selección de Rivest y Floyd (1970).

La única diferencia entre el algoritmo de selección de Hoare y el de Rivest y Floyd reside en la manera en que elegimos los pivotes. El algoritmo de Rivest y Floyd obtiene un pivote de “calidad” empleando el propio algoritmo, recursivamente, para seleccionar los pivotes!

De hecho, el algoritmo calcula una pseudomediana y elige ésta como pivote. Se subdivide el subvector $A[\ell..u]$ en bloques de q elementos (excepto posiblemente el último bloque), con q constante e impar, y para cada uno de ellos se obtiene su mediana. El coste de esta fase es $\Theta(n)$ y como resultado se obtiene un vector con $\lceil n/q \rceil$ elementos. Sobre dicho vector se aplica el algoritmo de selección para hallar la mediana. Como el pivote seleccionado es la pseudomediana de los n elementos originales, ello garantiza que al menos $n/2q \cdot q/2 = n/4$ elementos son mayores que el pivote.

Por lo tanto, el coste $C(n)$ en caso peor satisface

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4).$$

Utilizando la versión discreta del CMT, $C(n) = \mathcal{O}(n)$ si

$$\frac{3}{4} + \frac{1}{q} < 1.$$

Rivest y Floyd usaron el valor $q = 5$ en su formulación original de este algoritmo.

Parte I

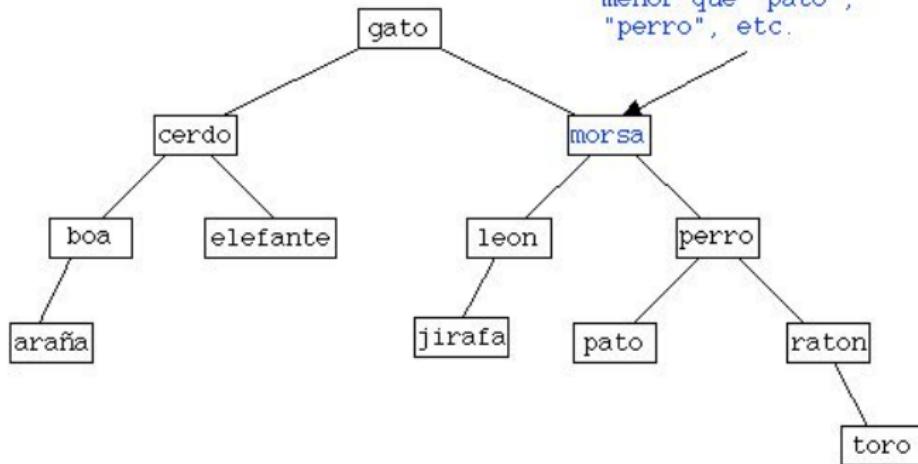
Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- Tablas de Hash
- Colas de prioridad
- Grafos y Recorridos

Definición

Un **árbol binario de búsqueda** T es un árbol binario tal que o es vacío o bien contiene un elemento x y satisface

- ① Los subárboles izquierdo y derecho, L y R , respectivamente, son árboles binarios de búsqueda.
- ② Para todo elemento y de L , $\text{CLAVE}(y) < \text{CLAVE}(x)$, y para todo elemento z de R , $\text{CLAVE}(z) > \text{CLAVE}(x)$.



"morsa" es mayor que
"leon" y "jirafa"; y
menor que "pato",
"perro", etc.

Lema

Un recorrido en inorder de un árbol binario de búsqueda T visita los elementos de T por orden creciente de clave.

Vamos a considerar ahora el diseño del algoritmo de búsqueda en árboles binarios de búsqueda (**BSTs**, en lo sucesivo). Dada la naturaleza recursiva de la definición de BST es razonable abordar el diseño de este algoritmo de manera recursiva.

Sea T el BST que representa al diccionario y k la clave buscada. Si $T = \square$ entonces k no se encuentra el diccionario y ello se habrá de indicar de algún modo conveniente. Si T no es vacío entonces tendremos que considerar la relación que existe entre la clave del elemento x que ocupa la raíz de T y la clave dada k .

Si $k = \text{CLAVE}(x)$ la búsqueda ha tenido éxito y finalizamos, retornando el elemento x (o la información asociada a x que nos interesa). Si $k < \text{CLAVE}(x)$, se sigue de la definición de los BSTs, que si hay un elemento en T cuya clave es k entonces dicho elemento se habrá de encontrar en el subárbol izquierdo de T , por lo que habrá que efectuar una llamada recursiva sobre el hijo izquierdo de T . Análogamente, si $k > \text{CLAVE}(x)$ entonces la búsqueda habrá de continuar recursivamente en el subárbol derecho de T .

```
template <typename Clave, typename Valor>
class Diccionario {
public:
    ...
    void busca(const Clave& k,
               bool& esta, Valor& v) const throw(error);
    void inserta(const Clave& k,
                 const Valor& v) throw(error);
    void elimina(const Clave& k) throw();
    ...
private:
    struct nodo_bst {
        Clave _k;
        Valor _v;
        nodo_bst* _izq;
        nodo_bst* _der;
        // constructora de la clase nodo_bst
        nodo_bst(const Clave& k, const Valor& v,
                  nodo_bst* izq = NULL, nodo_bst* der = NULL);
    };
    nodo_bst* raiz;

    static nodo_bst* busca_en_bst(nodo_bst* p,
                                   const Clave& k) throw();
    static nodo_bst* inserta_en_bst(nodo_bst* p,
                                    const Clave& k, const Valor& v) throw(error);
    static nodo_bst* elimina_en_bst(nodo_bst* p,
                                    const Clave& k) throw();
    static nodo_bst* juntar(nodo_bst* t1,
                           nodo_bst* t2) throw();
    static nodo_bst* reubicar_max(nodo_bst* p) throw();
    ...
}
```

El método BUSCA emplea el método privado BUSCA_EN_BST. Éste último recibe un apuntador *p* a la raíz del BST en el que se ha de hacer la búsqueda y una clave *k*. Devuelve un apuntador, o bien nulo, si *k* no está presente en el BST, o bien que apunta al nodo del BST que contiene la clave *k*.

```
template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::busca(const Clave& k,
    bool& esta, Valor& v) const throw(error) {

    nodo_bst* p = busca_en_bst(raiz, k);
    if (p == NULL)
        esta = false;
    else {
        esta = true;
        v = p -> _v;
    }
}
```

La implementación recursiva del método privado BUSCA_EN_BST es casi inmediata a partir de la definición de BST. Si el árbol es vacío ó su raíz contiene la clave k , devolvemos el apuntador p ya que apunta al lugar correcto (a nulo o al nodo con la clave). En caso contrario, se compara k con la clave almacenada en el nodo raíz al que apunta p y se prosigue recursivamente la búsqueda en el subárbol izquierdo o derecho, según toque.

```
// implementación recursiva
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::busca_en_bst(nodo_bst* p,
    const Clave& k) throw() {
    if (p == NULL or k == p->_k)
        return p;

    // p != NULL and k != p->_k
    if (k < p->_k)
        return busca_en_bst(p->_izq, k);
    else // p->_k < k
        return busca_en_bst(p->_der, k);
}
```

Puesto que el algoritmo de búsqueda recursivo es recursivo final es inmediato obtener una versión iterativa.

```
// implementación iterativa
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::busca_en_bst(nodo_bst* p,
    const Clave& k) throw() {

    while (p != NULL and k != p->_k) {
        if (k < p->_k)
            p = p->_izq;
        else // p->_k < k
            p = p->_der;
    }
    return p;
}
```

El algoritmo de inserción es también extremadamente simple y se obtiene a partir de un razonamiento similar al utilizado para desarrollar el algoritmo de búsqueda: si la nueva clave es menor que la clave en la raíz entonces el nuevo elemento se ha de insertar (recursivamente) en el subárbol izquierdo; si es mayor, la inserción se realiza en el subárbol derecho.

El método público `INSERTA` se apoya en otro método privado de clase llamado `INSERTA_EN_BST`. Éste recibe un apuntador a la raíz del BST donde se debe insertar el par $\langle k, v \rangle$ y nos devuelve un apuntador a la raíz del BST resultante de la inserción. Si k no aparece en el BST, se añade un nodo con el par $\langle k, v \rangle$ en el lugar apropiado. Si k ya existía, entonces el método modifica el valor asociado a k , reemplazando el valor anterior por v .

Antes de pasar a la implementación de las inserciones, veamos la implementación (trivial) de la constructora de la clase `nodo_bst`:

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst(const Clave& k,
    Valor& v, nodo_bst* izq, nodo_bst* der) throw(error) :
    _k(k), _v(v), _izq(izq), _der(der) {
}
```

```
template <typename Clave, typename Valor>
void Diccionario::inserta(const Clave& k,
    const Valor& v) throw(error) {

    raiz = inserta_en_bst(raiz, k, v);

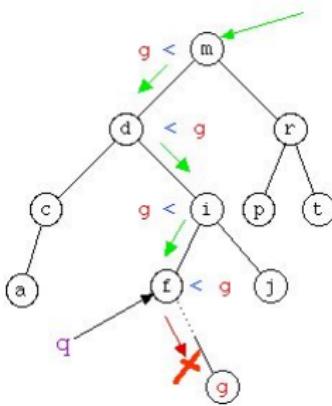
}

// implementación recursiva
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo*
Diccionario<Clave,Valor>::inserta_en_bst(nodo_bst* p,
    const Clave& k, const Valor& v) throw(error) {

    if (p == NULL)
        return new nodo_bst(k, v);

    // p != NULL, continúa la inserción en el
    // subárbol apropiado o reemplaza el valor
    // asociado si p -> _k == k
    if (k < p -> _k)
        p -> _izq = inserta_en_bst(p -> _izq, k, v);
    else if (p -> _k < k)
        p -> _der = inserta_en_bst(p -> _der, k, v);
    else // p -> _k == k
        p -> _v = v;
    return p;
}
```

La versión iterativa es más compleja, ya que además de localizar la hoja en la que se ha de realizar la inserción, deberá mantenerse un apuntador *padre* al que será padre del nuevo nodo.



```
// implementación iterativa
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::inserta_en_bst(nodo_bst* p,
    const Clave& k, const Valor& v) throw(error) {

    // el BST está vacío
    if (p == NULL)
        return new nodo_bst(k, v);

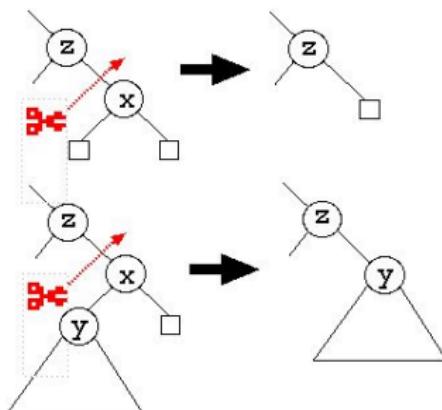
    // el BST no está vacío
    nodo_bst* padre = NULL;
    nodo_bst* p_orig = p;

    // buscamos el punto de inserción
    while (p != NULL and k != p->_k) {
        padre = p;
        if (k < p->_k)
            p = p->_izq;
        else // p->_k < k
            p = p->_der;
    }

    // insertamos el nuevo nodo como hoja
    // o modificamos el valor asociado, si
    // ya había un nodo con la clave k dada
    if (p == NULL) {
        if (k < padre->_k)
            padre->izq = new nodo_bst(k, v);
        else // k > padre->_k
            padre->der = new nodo_bst(k, v);
    }
    else // k == p->_k
        p->_v = v;

    return p_orig;
}
```

Sólo nos queda por considerar la eliminación de elementos en BSTs. Si el elemento a eliminar se encuentra en un nodo cuyos dos subárboles son vacíos basta eliminar el nodo en cuestión. Otro tanto sucede si el nodo x a eliminar sólo tiene un subárbol no vacío: basta hacer que la raíz del subárbol no vacío quede como hijo del padre de x .



El problema surge si hay que eliminar un nodo que contiene dos subárboles no vacíos. Podemos reformular el problema de la siguiente forma: dados dos BSTs T_1 y T_2 tales que todas las claves de T_1 son menores que las claves de T_2 obtener un nuevo BST que contenga todas las claves: $T = \text{JUNTAR}(T_1, T_2)$. Obviamente:

$$\text{JUNTAR}(T, \square) = T$$

$$\text{JUNTAR}(\square, T) = T$$

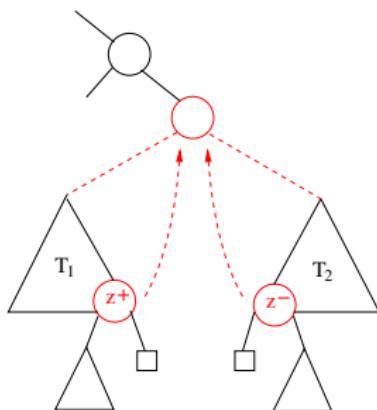
En particular, $\text{JUNTAR}(\square, \square) = \square$.

```
template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::elimina(
    const Clave& k) throw() {
    raiz = elimina_en_bst(raiz, k);
}

template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::elimina_en_bst(nodo_bst* p,
    const Clave& k) throw() {
    // la clave no está
    if (p == NULL) return p;

    if (k < p->_k)
        p->_izq = elimina_en_bst(p->_izq, k);
    else if (p->_k < k)
        p->_der = elimina_en_bst(p->_der, k);
    else { // k == p->_k
        nodo_bst* to_kill = p;
        p = juntar(p->_izq, p->_der);
        delete to_kill;
    }
    return p;
}
```

Sea z^+ la mayor clave de T_1 . Puesto que es mayor que todas las demás en T_1 pero al mismo tiempo menor que cualquier clave de T_2 podemos construir T colocando un nodo raíz que contenga al elemento de clave z^+ , a T_2 como subárbol derecho y al resultado de eliminar z^+ de T_1 —llámémosle T'_1 — como subárbol izquierdo. Además puesto que z^+ es la mayor clave de T_1 el correspondiente nodo **no** tiene subárbol derecho, y es el nodo “más a la derecha” en T_1 , lo que nos permite desarrollar un procedimiento ad-hoc para eliminarlo.



```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::juntar(nodo_bst* t1,
    nodo_bst* t2) throw() {

    // si uno de los dos subárboles es
    // vacío, la implementación es trivial
    if (t1 == NULL) return t2;
    if (t2 == NULL) return t1;

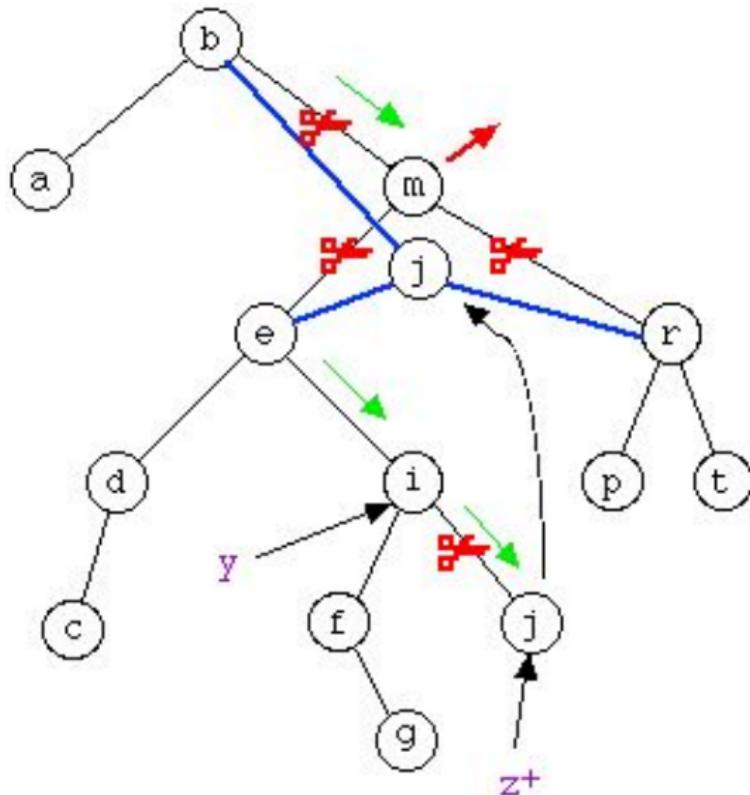
    // t1 != NULL y también t2 != NULL
    nodo_bst* z = reubica_max(t1);
    z -> _der = t2;
    return z;

    // alternativa: z = reubica_min(t2);
    //               z -> _izq = t1;
    //               return z;
}
```

El método privado REUBICA_MAX recibe un apuntador p a la raíz de un BST T y nos devuelve un apuntador a la raíz de un nuevo BST T' . La raíz de T' es el elemento máximo de T . El subárbol derecho de T' es nulo y el subárbol izquierdo de T' es el BST que se obtiene al eliminar el elemento máximo de T . La única “dificultad” estriba en tratar adecuadamente la situación en la que el elemento máximo del BST se encuentra ya como raíz del mismo—dicha situación debe detectarse y en ese caso el método no debe hacer nada.

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_bst*
Diccionario<Clave,Valor>::reubica_max(
    nodo_bst* p) throw() {

    nodo_bst* padre = NULL;
    nodo_bst* p_orig = p;
    while (p -> _der != NULL) {
        padre = p;
        p = p -> _der;
    }
    if (padre != NULL) {
        padre -> _der = p -> _izq;
        p -> _izq = p_orig;
    }
    return p;
}
```



Un razonamiento análogo nos lleva a una versión de JUNTAR en la que se emplea la clave mínima z^- del árbol T_2 para que ocupe la raíz del resultado, en el caso en que T_1 y T_2 no son vacíos.

Se ha evidenciado experimentalmente que conviene alternar entre el predecesor z^+ y el sucesor z^- del nodo a eliminar en los borrados (por ejemplo, mediante una decisión aleatoria o con bit que va pasando alternativamente de 0 a 1 y de 1 a 0) y no utilizar sistemáticamente una de las versiones.

Un BST de n elementos puede ser equivalente a una lista, pues puede contener un nodo sin hijos y $n - 1$ con sólo un hijo (p.e. si insertamos una secuencia de n elementos con claves crecientes en un BST inicialmente vacío). Un BST con estas características tiene altura n . En caso peor, el coste de una búsqueda, inserción o borrado en dicho árbol es $\Theta(n)$. En general, una búsqueda, inserción o borrado en un BST de altura h tendrá coste $\Theta(h)$ en caso peor. Como función de n , la altura de un árbol puede llegar a ser n y de ahí que el coste de las diversas operaciones es, en caso peor, $\Theta(n)$.

Pero normalmente el coste de estas operaciones será menor. Supongamos que cualquier orden de inserción de los elementos es equiprobable. Para búsquedas con éxito supondremos que buscamos cualquiera de las n claves con probabilidad $1/n$. Para búsquedas sin éxito o inserciones supondremos que finalizamos en cualquiera de las $n + 1$ hojas con igual probabilidad.

El coste de una de estas operaciones va a ser proporcional al número de comparaciones que habrá que efectuar entre la clave dada y las claves de los nodos examinados. Sea $C(n)$ el número medio de comparaciones y $C(n; k)$ el número medio de comparaciones si la raíz está ocupada por la k -ésima clave.

$$C(n) = \sum_{1 \leq k \leq n} C(n; k) \times \mathbb{P}[\text{raíz es } k\text{-ésima}] .$$

$$\begin{aligned}
C(n) &= \frac{1}{n} \sum_{1 \leq k \leq n} C(n; k) \\
&= 1 + \frac{1}{n} \sum_{1 \leq k \leq n} \left(\frac{1}{n} \cdot 0 + \frac{k-1}{n} \cdot C(k-1) + \frac{n-k}{n} \cdot C(n-k) \right) \\
&= 1 + \frac{1}{n^2} \sum_{0 \leq k < n} (k \cdot C(k) + (n-1-k) \cdot C(n-1-k)) \\
&= 1 + \frac{2}{n^2} \sum_{0 \leq k < n} k \cdot C(k).
\end{aligned}$$

Otra forma de plantear esto es calcular $I(n)$, el valor medio de la longitud de caminos internos (IPL). Dado un BST su longitud de caminos internos es la suma de las distancias desde la raíz a cada uno de los nodos.

$$C(n) = 1 + \frac{I(n)}{n}$$

La longitud media de caminos internos satisface la recurrencia

$$I(n) = n - 1 + \frac{2}{n} \sum_{0 \leq k < n} I(k), \quad I(0) = 0. \quad (1)$$

Esto es así porque cada nodo que no sea la raíz contribuye al IPL total 1 más su contribución al IPL del subárbol en que se encuentre. Otra razón por la que resulta interesante estudiar el IPL medio es porque el coste de construir un BST de tamaño n mediante n inserciones es proporcional al IPL.

Podemos asociar a cada ejecución de QUICKSORT un BST como sigue. En la raíz se coloca el pivote de la fase inicial; los subárboles izquierdo y derecho corresponden a las ejecuciones recursivas de QUICKSORT sobre los subvectores a la izquierda y a la derecha del pivote.

Consideremos un subárbol cualquiera de este BST. Todos los elementos del subárbol, salvo la raíz, son comparados con el nodo raíz durante la fase a la que corresponde ese subárbol. Y recíprocamente el pivote de una determinada fase ha sido comparado con los elementos (pivotes) que son sus antecesores en el árbol y ya no se comparará con ningún otro pivote. Por lo tanto, el número de comparaciones en las que interviene un cierto elemento no siendo el pivote es igual a su distancia a la raíz del BST. Por esta razón $I(n) = Q(n)$.

Para resolver la recurrencia de $I(n)$ (longitud de caminos internos en BSTs) calculamos $(n + 1)I(n + 1) - nI(n)$:

$$\begin{aligned}(n + 1)I(n + 1) - nI(n) &= (n + 1)n - n(n - 1) + 2I(n) \\&= 2n + 2I(n);\end{aligned}$$

$$(n + 1)I(n + 1) = 2n + (n + 2)I(n)$$

$$\begin{aligned}
I(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1} I(n) = \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{n+2}{n} I(n-1) \\
&= \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{2(n-2)(n+2)}{n(n-1)} + \frac{n+2}{n-1} I(n-2) \\
&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=k} \frac{n-i}{(n-i+1)(n-i+2)} + \frac{n+2}{n-k+1} I(n-k) \\
&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=n} \frac{i}{(i+1)(i+2)} \\
&= \mathcal{O}(1) + 2(n+2) \sum_{1 \leq i \leq n} \left(\frac{2}{i+2} - \frac{1}{i+1} \right) \\
&= \mathcal{O}(1) + 2(n+2) \left(\frac{2}{n+2} + \frac{1}{n+1} + H_n - 2 \right) \\
&= 2nH_n - 4n + 4H_n + \mathcal{O}(1),
\end{aligned}$$

donde $H_n = \sum_{1 \leq i \leq n} 1/i$.

Puesto que $H_n = \ln n + \mathcal{O}(1)$ se sigue que

$$\begin{aligned} I(n) &= Q(n) = 2n \ln n + \mathcal{O}(n) \\ &= 1,386 \dots n \log_2 n + \mathcal{O}(n) \end{aligned}$$

y que $C(n) = 2 \ln n + \mathcal{O}(1)$.

Los BSTs permiten otras varias operaciones siendo los algoritmos correspondientes simples y con costes (promedio) razonables. Algunas operaciones como las de búsqueda o borrado por rango (e.g. busca el 17º elemento) o de cálculo del rango de un elemento dado requieren una ligera modificación de la implementación estándar, de tal forma que cada nodo contenga información relativa a tamaño del subárbol en él enraizado. Concluimos esta parte con un ejemplo concreto: dadas dos claves k_1 y k_2 , $k_1 < k_2$ se precisa una operación que devuelve una lista ordenada de todos los elementos cuya clave k está comprendida entre las dos dadas, esto es, $k_1 \leq k \leq k_2$.

Si el BST es vacío, la lista a devolver es también vacía. Supongamos que el BST no es vacío y que la clave en la raíz es k . Si $k < k_1$ entonces todas las claves buscadas deben encontrarse, si las hay, en el subárbol derecho. Análogamente, si $k_2 < k$ se proseguirá la búsqueda recursivamente en el subárbol izquierdo. Finalmente, si $k_1 \leq k \leq k_2$ entonces puede haber claves que caen dentro del intervalo tanto en el subárbol izquierdo como en el derecho. Para respetar el orden creciente en la lista, deberá buscarse recursivamente a la izquierda, luego listar la raíz y finalmente buscarse recursivamente a la derecha.

```
// El método en_rango se ha de declarar como
// método público de la clase Diccionario.
// Dicho método usa a otro auxiliar,
// en_rango_en_bst, que se habra declarado
// como método privado de clase (static).
template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::en_rango(
    const Clave& k1, const Clave& k2,
    list<pair<Clave,Valor> >& result) const throw(error) {
    en_rango_en_bst(raiz, k1, k2, result);
}

template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::en_rango_en_bst(nodo_bst* p,
    const Clave& k1, const Clave& k2,
    list<pair<Clave,Valor> >& result) const throw(error) {
    if (p == NULL) return;

    if (k1 <= p->_k)
        en_rango_en_bst(p->_izq, k1, k2, result);

    if (k1 <= p->_k and p->_k <= k2)
        result.push_back(make_pair(p->_k, p->_v));

    if (k <= p->_k)
        en_rango_en_bst(p->_der, k1, k2, result);
}
```

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- **Árboles balanceados (AVLs)**
- Tablas de Hash
- Colas de prioridad
- Grafos y Recorridos

Árboles equilibrados

El problema de los BSTs estándar es que, como resultado de ciertas secuencias de inserciones y/o borrados, pueden quedar muy desequilibrados.

En caso peor la altura de un BST de tamaño n es $\Theta(n)$ y en consecuencia el coste en caso peor de las operaciones de búsqueda, inserción y borrado es $\Theta(n)$.

A fin de evitar este problema, se han propuesto diversas soluciones; la más antigua y una de las más elegantes y sencillas, es el **equilibrado en altura**, propuesto en 1962 por Adelson-Velskii y Landis.

Los árboles de búsqueda resultantes se denominan **AVLs** en honor a sus inventores.

Definición

Un AVL T es un árbol binario de búsqueda tal que o es vacío o bien cumple que

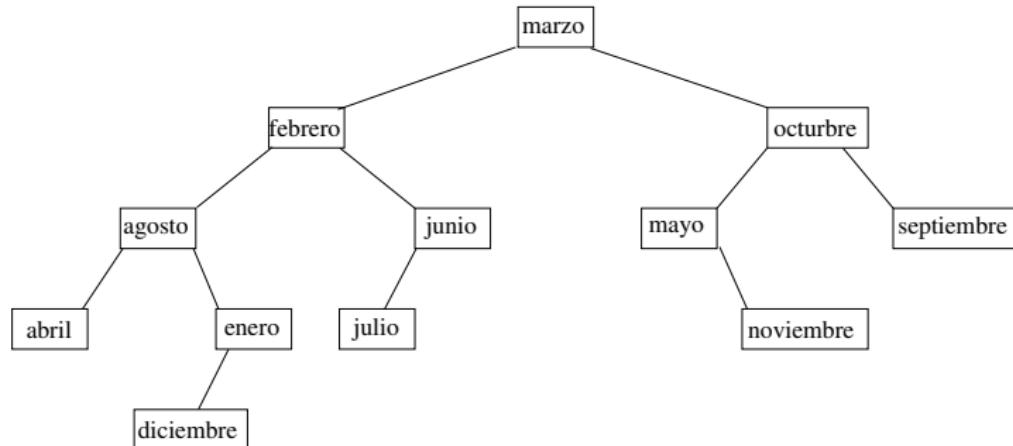
- ① Los subárboles izquierdo y derecho, L y R , respectivamente, son AVLs.
- ② Las alturas de L y de R difieren a lo sumo en una unidad:

$$|\text{altura}(L) - \text{altura}(R)| \leq 1$$

La naturaleza recursiva de la definición anterior garantiza que la condición de equilibrio se cumple en cada nodo x de un AVL. Si denotamos $\text{bal}(x)$ la diferencia entre las alturas de los subárboles izquierdo y derecho de un nodo cualquiera x tendremos que $\text{bal}(x) \in \{-1, 0, +1\}$.

Por otro lado todo AVL es un árbol binario de búsqueda, por lo que el algoritmo de búsqueda en AVLs es idéntico al de BSTs, y un recorrido en inorder de un AVL visitará todos sus elementos en orden creciente de claves.

{enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre
noviembre, diciembre}



Lema

La altura $h(T)$ de un AVL de tamaño n es $\Theta(\log n)$.

Demostración. Puesto que el AVL es un árbol binario se cumple que $h(T) \geq \lceil \log_2(n + 1) \rceil$, es decir, $h(T) \in \Omega(\log n)$. Ahora vamos a demostrar que $h(T) \in \mathcal{O}(\log n)$.

Sea N_h el mínimo número de nodos necesario para construir un AVL de altura h . Claramente $N_0 = 0$ y $N_1 = 1$. El AVL más desequilibrado posible de altura $h > 1$ se consigue poniendo un subárbol, digamos el izquierdo, de altura $h - 1$ empleando el mínimo posible de nodos N_{h-1} y otro subárbol, el derecho, que tendrá altura $h - 2$ (pero no puede tener menos!) en el que pondremos también el mínimo posible de nodos, N_{h-2} . Por tanto,

$$N_h = 1 + N_{h-1} + N_{h-2}$$

Veamos algunos valores de la secuencia $\{N_h\}_{h \geq 0}$:

$$0, 1, 2, 4, 7, 12, 20, 33, 54, 88, \dots$$

La similitud con la secuencia de los números de Fibonacci llama rápidamente nuestra atención y no es casual: puesto que $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ se cumple que $N_h = F_{h+1} - 1$ para toda $h \geq 0$. En efecto,

$$N_0 = F_1 - 1 = 1 - 1 = 0 \text{ y}$$

$$N_h = 1 + N_{h-1} + N_{h-2} = 1 + (F_h - 1) + (F_{h-1} - 1) = F_{h+1} - 1$$

El número n -ésimo de Fibonacci cumple:

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,$$

donde $\phi = (1 + \sqrt{5})/2 \approx 1,61803\dots$ es la *razón áurea*.

Consideremos ahora un AVL de tamaño n y altura h . Entonces $n \geq N_h$ por definición y

$$n \geq F_{h+1} - 1 \geq \frac{\phi^{h+1}}{\sqrt{5}} - \frac{3}{2}$$

Luego,

$$(n + \frac{3}{2}) \frac{\sqrt{5}}{\phi} \geq \phi^h$$

Tomando logaritmos en base ϕ , y simplificando

$$h \leq \log_\phi n + \mathcal{O}(1) = 1,44 \log_2 n + \mathcal{O}(1)$$

El lema queda demostrado.

Del lema anterior deducimos que el coste de cualquier búsqueda en un AVL de tamaño n será, incluso en caso peor, $\mathcal{O}(\log n)$.

El problema reside en cómo conseguir que se cumpla la condición de equilibrio en cada uno de los nodos del AVL tras una inserción o un borrado.

La idea es que ambos algoritmos actúan igual que los correspondientes en BSTs, pero cada uno de los nodos en el camino que va de la raíz al punto de inserción (o borrado) debe ser comprobado para verificar que continúa cumpliendo la condición de balance y si no es así hacer algo al efecto.

En primer lugar nos damos cuenta de que será necesario que cada nodo almacene información sobre su altura (o su balance), para evitar cómputos demasiado costosos.

```
template <typename Clave, typename Valor>
class Diccionario {
public:
    ...
    void buscar(const Clave& k,
                bool& esta, Valor& v) const throw(error);
    void inserta(const Clave& k,
                 const Valor& v) throw(error);
    void elimina(const Clave& k) throw();
    ...

private:
    struct nodo_avl {
        Clave _k;
        Valor _v;
        int _alt;
        nodo_avl* _izq;
        nodo_avl* _der;
        // constructora de la clase nodo_avl
        nodo_avl(const Clave& k, const Valor& v,
                  int alt = 1,
                  nodo_avl* izq = NULL,
                  nodo_avl* der = NULL);
    };
    nodo_avl* raiz;
    ...
    static int altura(nodo_avl* p) throw();
    static void actualizar_altura(nodo_avl* p) throw();
};
```

```
int max(int x, int y) {
    return x > y ? x : y;
}

template <typename Clave, typename Valor>
static int Diccionario<Clave,Valor>::altura(
    nodo_avl* p) throw() {

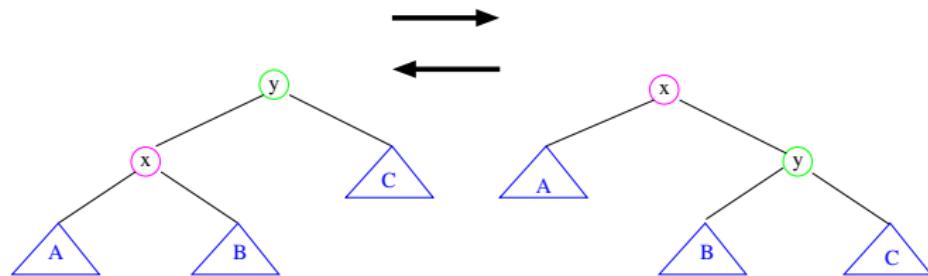
    if (p == NULL)
        return 0;
    else
        return p -> _alt;
}

template <typename Clave, typename Valor>
static void Diccionario<Clave,Valor>::actualizar_altura(
    nodo_avl* p) throw() {

    p -> _alt = 1 + max(altura(p -> _izq), altura(p -> _der));
}
```

Para re establecer el balance en un nodo que ya no cumpla la condición de balance se utilizan *rotaciones*.

La figura muestra las rotaciones más simples. Obsérvese que si el árbol de la izquierda es un BST ($A < x < B < y < C$) entonces el de la derecha también lo es, y viceversa.



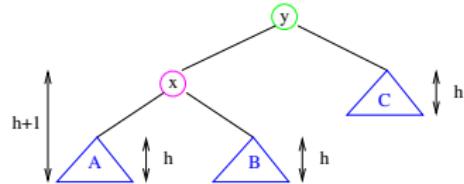
Supongamos un subárbol de un AVL con raíz y en el que estamos haciendo la inserción de una clave k tal que $k < x < y$ y que finalizado el proceso de inserción recursiva se produce un desequilibrio en y .

Si C tiene altura h entonces el subárbol enraizado en x ha de tener altura h ó altura $h + 1$. Si la altura de x fuera h entonces la inserción de la nueva clave k no podría provocar el desequilibrio en y , de manera que vamos a suponer que la altura de x es $h + 1$.

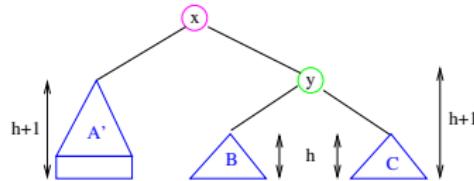
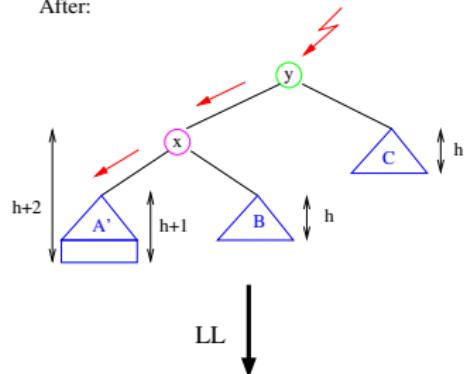
Razonando de manera parecida llegamos a la conclusión de que la altura de A tiene que ser h . Así pues la altura de y , previa a la inserción era $h + 2$. Como consecuencia de la inserción vamos a suponer que la altura del subárbol A se incrementa, de manera que el nuevo subárbol A' tiene altura $h + 1$. Si suponemos que x mantiene la condición de AVL tras la inserción (pero y no) eso nos lleva a concluir que B también tenía altura h .

Tras la inserción el subárbol enraizado en x pasa a tener altura $h + 2$ de manera que $\text{bal}(y) = +2$!

Before:



After:

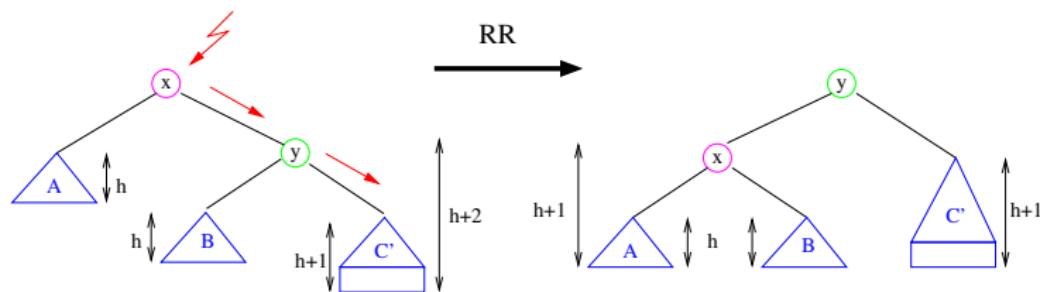


Al aplicar la rotación sobre el nodo y , se preserva la propiedad de BST y además se reestablece la propiedad de AVL. Por hipótesis, A' , B y C son AVLs. Tras la rotación la altura de y es $h + 1$ y su balance 0, y la altura de x pasa a ser $h + 2$ y su balance 0.

No sólo hemos solventado el problema de desequilibrio en y : el subárbol donde hemos aplicado la inserción es un AVL y tiene la misma altura que el subárbol antes de hacer la inserción, de manera que ninguno de los antecesores de y en el AVL va a estar desbalanceado.

A la rotación que hemos empleado se le suele denominar LL (left-left).

Un análisis análogo revela que si la clave insertada k cumple $x < y < k$ y se produce un desbalanceo en el nodo x , entonces podemos reestablecer el equilibrio aplicando una rotación RR (right-right) sobre el nodo x . La rotación RR es la simétrica de la rotación LL.



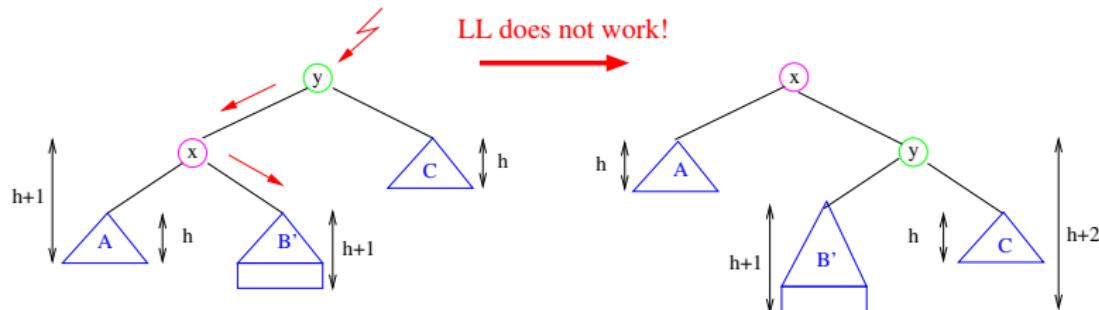
Analicemos ahora el caso de un subárbol de un AVL con raíz y en el que estamos haciendo la inserción de una clave k , pero esta vez $x < k < y$ y que finalizado el proceso de inserción recursiva se produce un desequilibrio en y .

Si C tiene altura h entonces el subárbol enraizado en x ha de tener altura $h + 1$, la altura de B ha de ser h (aquí razonamos igual que cuando hicimos el análisis de la situación LL).

Como consecuencia de la inserción vamos a suponer que la altura del subárbol B se incrementa, de manera que el nuevo subárbol B' tiene altura $h + 1$. Si suponemos que x mantiene la condición de AVL tras la inserción (pero y no) eso nos lleva a concluir que A también tenía altura h .

Tras la inserción el subárbol enraizado en x pasa a tener altura $h + 2$ de manera que $\text{bal}(y) = +2$!

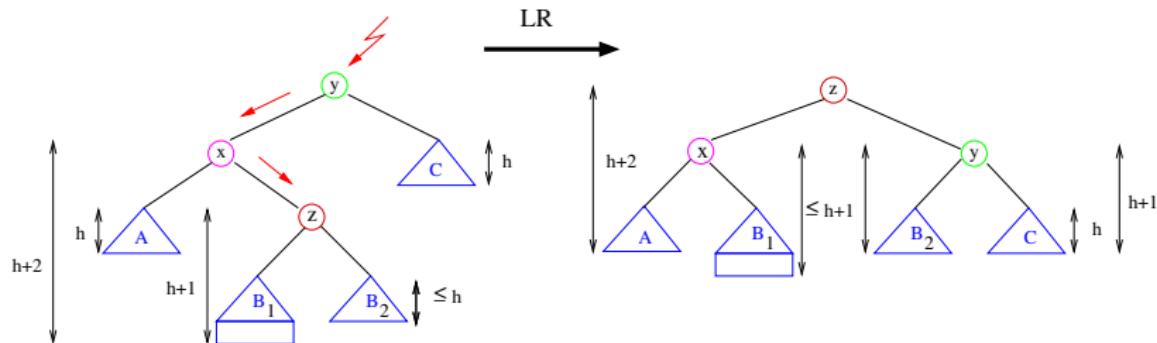
Si aplicamos la rotación simple LL sobre el nodo y , se preserva la propiedad de BST pero no se reestablece la propiedad de AVL. Por hipótesis, A , B' y C son AVLs. Pero tras la rotación LL la altura de y es $h + 2$ y su balance +1, y la altura de x pasa a ser $h + 3$ y su balance -2!



Vamos a tener que pensar alguna otra cosa y hacer un análisis más fino.

Vamos a suponer que la raíz del subárbol B' es z y que tiene subárboles B_1 y B_2 , ambos AVLs. Puesto que B' tiene altura $h + 1$ y es un AVL, al menos uno de los dos subárboles B_i tiene altura h y el otro B_j tiene altura h ó $h - 1$.

Si aplicamos una rotación que lleva a z a la raíz entonces x es la raíz de su subárbol izquierdo y sus hijos son A y B_1 , y y es la raíz de su subárbol derecho y sus hijos son B_2 y C .



Nótese que antes de la rotación el inorden era

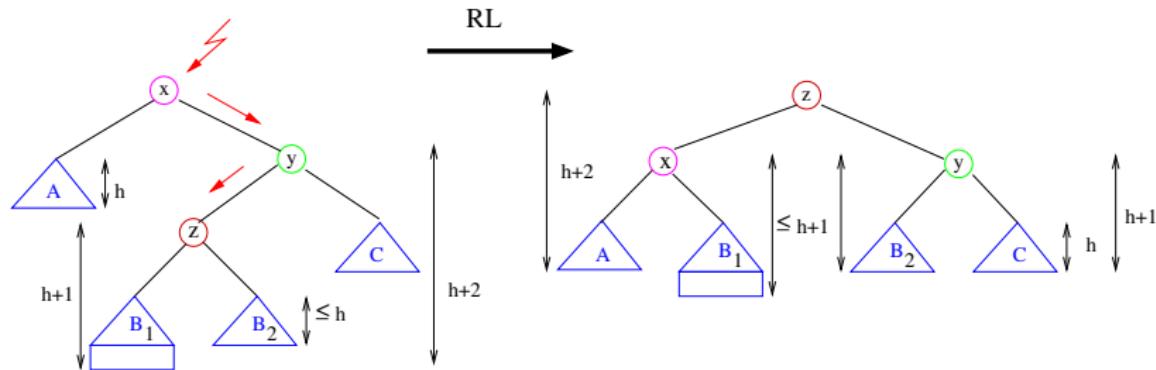
$A < x < B_1 < z < B_2 < y < C$, exactamente igual que tras la rotación, esto, este nuevo tipo de rotación también preserva la propiedad de BST.

Al aplicar la rotación sobre el nodo y , se preserva la propiedad de BST y además se reestablece la propiedad de AVL. Por hipótesis, A , B y C son AVLs. Tras la rotación la altura de x es $h + 1$ y su balance 0 ó +1, la altura de y es $h + 1$ y su balance 0 ó -1, y la altura de z pasa a ser $h + 2$ y su balance 0.

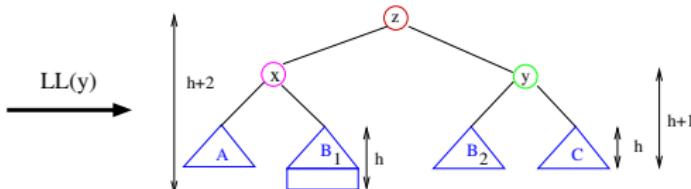
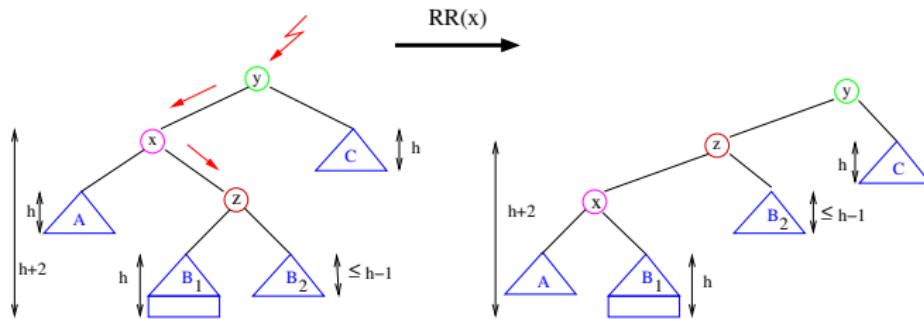
Igual que con las rotaciones simples LL y RR, no sólo arreglamos el problema de desequilibrio en y : el subárbol donde hemos aplicado la inserción es un AVL y tiene la misma altura que el subárbol antes de hacer la inserción, de manera que ninguno de los antecesores de y en el AVL va a estar desbalanceado.

A esta nueva rotación se le denomina rotación doble LR (left-right).

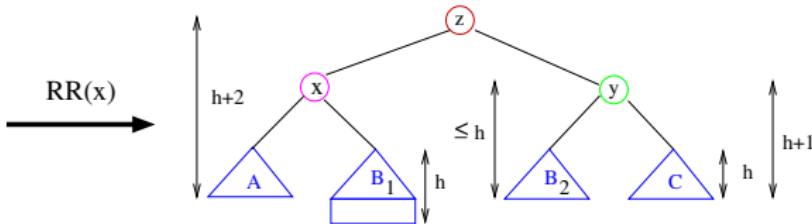
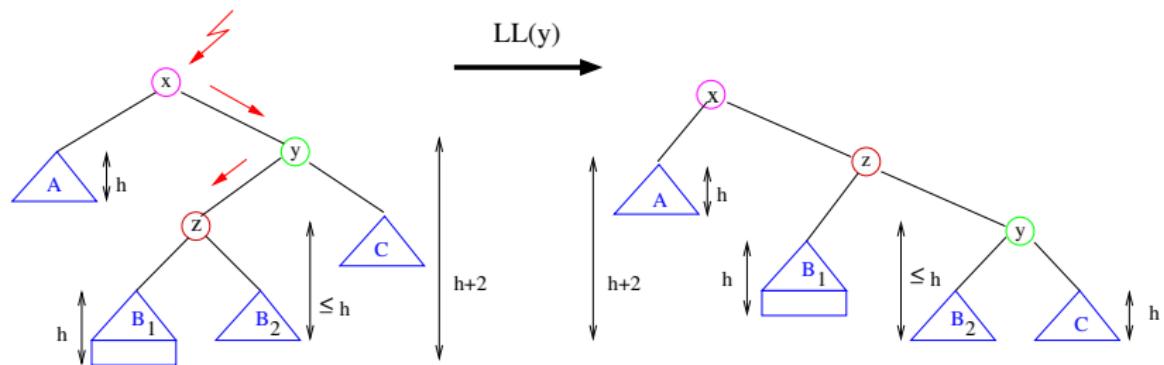
Para la situación en que una clave se inserta a la derecha y luego hacia la izquierda, el desequilibrio que se pudiera producir se corrige mediante una rotación RL, que es completamente análoga a la LR, cambiando derecha por izquierda y viceversa.



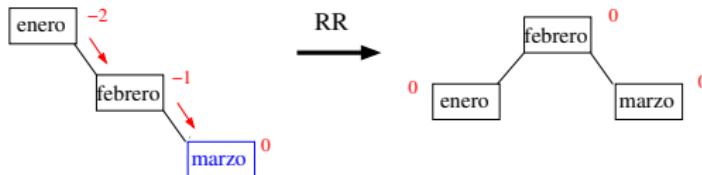
Las rotaciones dobles (LR y RL) se denominan así porque podemos descomponerlas como una secuencia de dos rotaciones simples. Por ejemplo la rotación doble LR sobre el nodo y consiste en primero aplicar una rotación RR sobre x y a continuación una rotación LL sobre y (su hijo izquierdo ha pasado a ser z y no x).



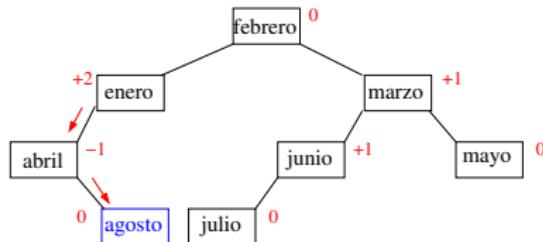
De manera similar, una rotación RL se compone de una rotación LL seguida de una rotación RR.



Ejemplo de los meses del año en un AVL:

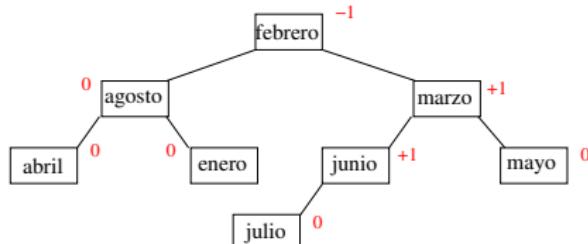


+ {enero, febrero, marzo}

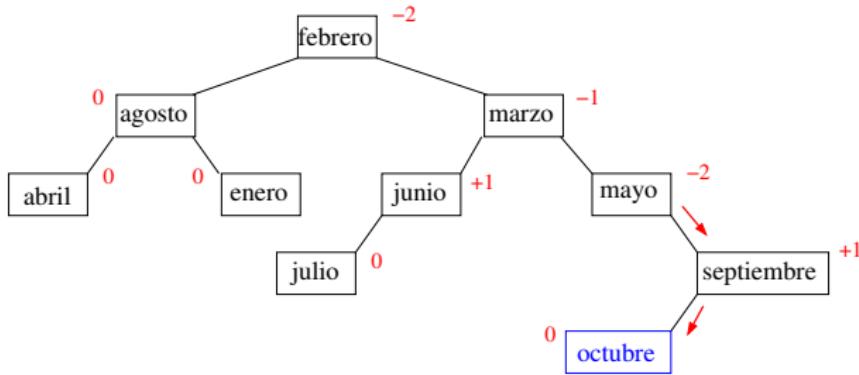


+ {abril, mayo, junio, julio, agosto}

↓ LR

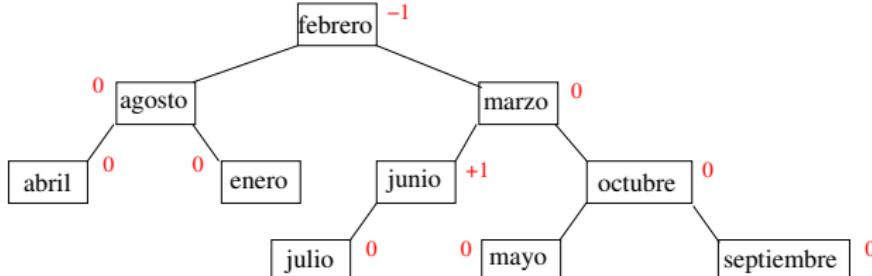


Ejemplo de los meses del año en un AVL:

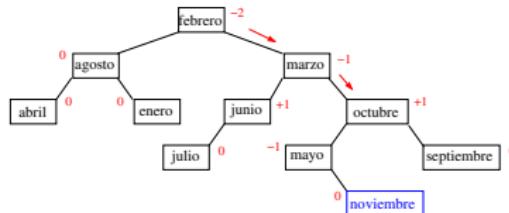


+ {septiembre, octubre}

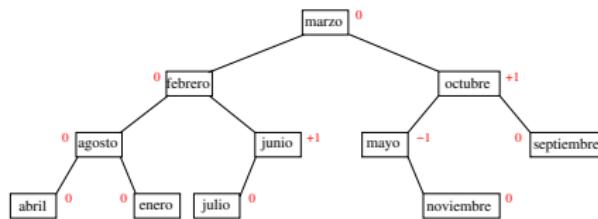
↓
RL



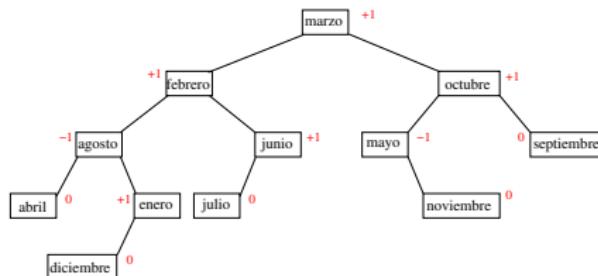
Ejemplo de los meses del año en un AVL:



+ {noviembre}



+ {noviembre}



+ {diciembre}

De nuestro análisis previo podemos establecer los siguientes hechos y sus consecuencias:

- ① Una rotación simple o doble sólo involucra la modificación de unos pocos apuntadores y actualizaciones de alturas, y su coste es por tanto $\Theta(1)$, independiente de la talla del árbol.
- ② En una inserción en un AVL sólo habrá que efectuarse a lo sumo una rotación (simple o doble) para reestablecer el equilibrio del AVL. Dicha rotación se aplica, en su caso, en uno de los nodos en el camino entre la raíz y el punto de inserción.
- ③ El coste en caso peor de una inserción en un AVL es $\Theta(\log n)$.

```
template <typename Clave, typename Valor>
class Diccionario {
public:
    ...
private:
    struct nodo_avl {
        ...
    };
    nodo_avl* raiz;
    ...
    static nodo_avl* inserta_en_avl(nodo_avl* p,
        const Clave& k, const Valor& v) throw(error);
    static nodo_avl* rotacionLL(nodo_avl* p) throw();
    static nodo_avl* rotacionLR(nodo_avl* p) throw();
    static nodo_avl* rotacionRL(nodo_avl* p) throw();
    static nodo_avl* rotacionRR(nodo_avl* p) throw();
    ...
};
```

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::rotacionLL(nodo_avl* p)
throw() {
    nodo_avl* q = p -> _izq;
    p -> _izq = q -> _der;
    q -> _der = p;
    actualiza_altura(p);
    actualiza_altura(q);
    return q;
}

template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::rotacionRR(nodo_avl* p)
throw() {
    nodo_avl* q = p -> _der;
    p -> _der = q -> _izq;
    q -> _izq = p;
    actualiza_altura(p);
    actualiza_altura(q);
    return q;
}
```

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::rotacionLR(nodo_avl* p)
throw() {
    p -> _izq = rotacionRR(p -> _izq);
    return rotacionLL(p);
}

template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::rotacionRL(nodo_avl* p)
throw() {
    p -> _der = rotacionLL(p -> _der);
    return rotacionRR(p);
}
```

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::inserta_en_avl(nodo_avl* p,
    const Clave& k, const Valor& v) throw(error) {

    if (p == NULL)
        return new nodo_avl(k, v);

    if (k < p -> _k) {
        p -> _izq = inserta_en_avl(p -> _izq, k, v);
        // comprobamos si hay desequilibrio en p
        // y rotamos si es necesario
        if (altura(p -> _izq) - altura(p -> _der) == 2) {
            // p -> _izq no puede ser vacío
            if (k < p -> _izq -> _k) // caso LL
                p = rotacionLL(p);
            else // caso LR
                p = rotacionLR(p);
        }
    }
    else if (p -> _k < k) { // análogo al caso anterior ...
        p -> _der = inserta_en_avl(p -> _der, k, v);
        if (altura(p -> _der) - altura(p -> _izq) == 2) {
            if (p -> _der -> _k < k) // caso RR
                p = rotacionRR(p);
            else // caso RL
                p = rotacionRL(p);
        }
    }
    else // p -> _k == k
        p -> _v = v;

    actualiza_altura(p);

    return p;
}
```

Si bien resulta posible escribir una versión iterativa del algoritmo de inserción, resulta complicada ya que una vez efectuada la inserción del nuevo nodo en la hoja que corresponde deberemos deshacer el camino hasta la raíz para detectar si en alguno de los nodos del camino se produce un desequilibrio y entonces aplicar la rotación correspondiente. Este “deshacer el camino” ocurre de manera natural con la recursividad, a la vuelta de cada llamada recursiva comprobamos si en el nodo desde el que se hace la llamada recursiva hay o no desequilibrio al terminar la inserción. Para una versión iterativa necesitaríamos que cada nodo contuviese un apuntador explícito a su padre o bien tendremos que almacenar la secuencia de nodos visitados durante la “bajada” en una pila, y después ir desapilando uno a uno para deshacer el camino.

El algoritmo de borrado en AVLs se fundamenta en las mismas ideas, si bien el análisis de qué rotación aplicar en cada caso es un poco más complejo.

Baste decir que el coste en caso peor de un borrado en un AVL es $\Theta(\log n)$. Los desequilibrios se solucionan aplicando rotaciones simples o dobles, pero a diferencia de lo que sucede con las inserciones, podemos tener que aplicar varias. No obstante, todas las rotaciones (de coste $\Theta(1)$) se aplican a lo sumo una vez sobre cada uno de los nodos del camino desde la raíz hasta el punto de borrado, y como sólo hay $\mathcal{O}(\log n)$ de éstos, el coste es logarítmico en caso peor.

El algoritmo de borrado en AVLs es relativamente más complejo que el de inserción, y la versión iterativa aún más. En las siguientes transparencias se muestra el código del algoritmo de borrado, en su versión recursiva, sin más comentarios.

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::elimina_en_avl(nodo_avl* p,
    const Clave& k) throw() {
if (p == NULL) return p;
if (k < p->_k) {
    p->_izq = elimina_en_avl(p->_izq, k);
    // comprobamos si hay desequilibrio en p
    // y rotamos si es necesario
    if (altura(p->_der) - altura(p->_izq) == 2) {
        // p->_der no puede ser vacío
        if (altura(p->_der->_izq)
            - altura(p->_der->_der) == 1)
            p = rotacionRL(p);
        else
            p = rotacionRR(p);
    }
}
else if (p->_k < k) { // analogo al caso anterior ...
    p->_der = elimina_en_avl(p->_der, k);
    if (altura(p->_izq) - altura(p->_der) == 2) {
        if (altura(p->_izq->_der)
            - altura(p->_der->_izq) == 1)
            p = rotacionLR(p);
        else
            p = rotacionLL(p);
    }
}
else { // p->_k == k
    nodo_avl* to_kill = p;
    p = juntar(p->_izq, p->_der);
    delete to_kill;
}
actualiza_altura(p);
return p;
}
```

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::juntar(nodo_avl* t1,
    nodo_avl* t2) throw() {

    // si uno de los dos subárboles es
    // vacío, la implementación es trivial
    if (t1 == NULL) return t2;
    if (t2 == NULL) return t1;

    // t1 != NULL y también t2 != NULL
    nodo_avl* z;
    elimina_min(t2, z);
    z -> _izq = t1;
    z -> _der = t2;
    actualiza_altura(z);
    return z;
}
```

```
template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_avl*
Diccionario<Clave,Valor>::elimina_min(
    nodo_avl*& p, nodo_avl*& z) throw() {

    if (p -> _izq != NULL) {
        elimina_min(p -> _izq, z);
        // comprobamos si hay desequilibrio en p
        // y rotamos si es necesario
        if (altura(p -> _der) - altura(p -> _izq) == 2) {
            // p -> _der no puede ser vacío
            if (altura(p -> _der -> _izq)
                - altura(p -> _der -> _der) == 1)
                p = rotacionRL(p);
            else
                p = rotacionRR(p);
        }
    } else {
        z = p;
        p = p -> _der;
    }
    actualiza_altura(p);
}
```

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- **Tablas de Hash**
- Colas de prioridad
- Grafos y Recorridos

Tablas de dispersión (hash)

Una *tabla de dispersión* (ing: *hash table*) permite almacenar un conjunto de elementos (o pares clave-valor) mediante una *función de dispersión* h que va del conjunto de claves al conjunto de índices o posiciones de la tabla (p.e. $0..M - 1$). Idealmente la función de dispersión h haría corresponder a cada uno de los n elementos (de hecho a sus claves) que queremos almacenar una posición distinta. Obviamente, esto no será posible en general y a claves distintas les corresponderá la misma posición de la tabla.

Pero si la función de dispersión “dispersa” eficazmente las claves, el esquema seguirá siendo útil, ya que la probabilidad de que el diccionario a representar mediante la tabla de hash contenga muchas claves con igual valor de h será pequeña. Dadas dos claves x e y distintas, se dice que x e y son *sinónimos*, o que colisionan, si $h(x) = h(y)$. El problema básico de la implementación de un diccionario mediante una tabla de hash consistirá en la definición de una estrategia de *resolución de colisiones*.

```
template <typename T> class Hash {
public:
    int operator()(const T& x) const throw();
};

template <typename Clave, typename Valor,
          template <typename> class HashFunct = Hash>
class Diccionario {
public:
    ...
    void busca(const Clave& k,
               bool& esta, Valor& v) const throw(error);
    void inserta(const Clave& k, const Valor& v) throw(error);
    void elimina(const Clave& k) throw();
    ...
private:
    struct nodo {
        Clave _k;
        Valor _v;
        ...
    };
    nat _M; // tamaño de la tabla
    nat _n; // núm. de elementos en la tabla
    double _alfa_max; // factor de carga

    // direccionamiento abierto
    nodo* _Thash; // tabla con los pares <clave,valor>

    // separate chaining
    // nodo** _Thash; // tabla de punteros a nodo

    static int hash(const Clave& k) throw() {
        HashFunct<Clave> h;
        return h(k) % _M;
    }
};
```

Funciones de *hash*

Una buena función de hash debe tener las siguientes propiedades:

- 1 Debe ser fácil de calcular
- 2 Debe dispersar de manera razonablemente uniforme el espacio de las claves, es decir, si dividimos el conjunto de todas las claves en grupos de sinónimos, todos los grupos deben contener aproximadamente el mismo número de claves.
- 3 Debe enviar a posiciones distantes a claves que son similares.

La construcción de buenas funciones de hash no es fácil y requiere conocimientos muy sólidos de varias ramas de las matemáticas. Está fuertemente relacionada con la construcción de generadores de números pseudoaleatorios. Como regla general da buenos resultados calcular un número entero positivo a partir de la representación binaria de la clave (p.e. sumando los bytes que la componen) y tomar módulo M , el tamaño de la tabla. Se recomienda que M sea un número primo.

La clase `Hash<T>` define el operador `()` de manera que si `h` es un objeto de la clase `Hash<T>` y `x` es un objeto de la clase `T`, podemos “aplicar” `h(x)`. Esto nos devolverá un número entero. La operación privada `hash` de la clase `Diccionario` calculará el módulo mediante `h(x) % _M` de manera que obtengamos un índice válido de la tabla, entre 0 y `_M - 1`.

```
// especialización del template para T = string
template <> class Hash<string> {
public:
    int operator()(const string& x) const throw() {

        int s = 0;
        for (int i = 0; i < x.length(); ++i)
            s = s * 37 + x[i];
        return s;
    };

// especialización del template para T = int
template <> class Hash<int> {
static long const MULT = 31415926;
public:
    int operator()(const int& x) const throw() {

        long y = ((x * x * MULT) << 20) >> 4;
        return y;
    };
};
```

Otras funciones de hash más sofisticadas emplean sumas ponderadas o transformaciones no lineales (p.e. elevan al cuadrado el número representado por los k bytes centrales de la clave).

Resolución de colisiones

Existen dos grandes familias de estrategias de resolución de colisiones. Por razones históricas, que no lógicas, se utilizan los términos **dispersión abierta** (ing: *open hashing*) y **direcciónamiento abierto** (ing: *open addressing*). Estudiaremos dos ejemplos representativos: tablas abiertas con **sinónimos encadenados** (ing: *separate chaining*) y **sondeo lineal** (ing: *linear probing*).

Sinónimos encadenados

En las tablas con sinónimos encadenados cada entrada de la tabla da acceso a una lista enlazada de sinónimos.

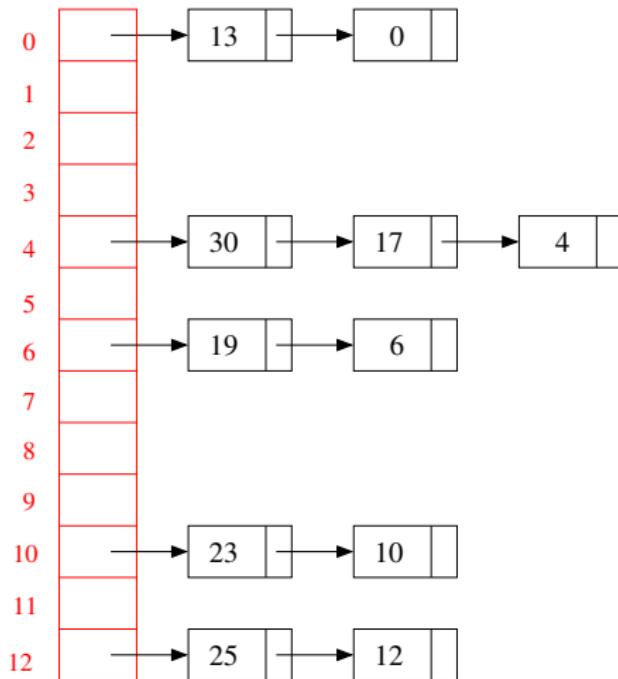
```
template <typename Clave, typename Valor,
          template <typename> class HashFunct = Hash>
class Diccionario {
    ...
private:
    struct nodo {
        Clave _k;
        Valor _v;
        nodo* _sig;
        // constructora de la clase nodo
        nodo(const Clave& k, const Valor& v,
              nodo* sig = NULL);
    };
    nodo** _hash; // tabla de punteros a nodo
    int _M;       // tamaño de la tabla
    int _n;        // numero de elementos
    double _alfa_max; // factor de carga

    nodo* busca_sep_chain(const Clave& k) const throw();
    void inserta_sep_chain(const Clave& k,
                           const Valor& v) throw(error);
    void elimina_sep_chain(const Clave& k) throw();
};
```

$$M = 13$$

$$X = \{ 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30 \}$$

$$h(x) = x \bmod M$$



Para la inserción, se accede a la lista correspondiente mediante la función de hash, y se recorre para determinar si ya existía o no un elemento con la clave dada. En el primer caso, se modifica la información asociada; en el segundo se añade un nuevo nodo a la lista con el nuevo elemento.

Puesto que estas listas de sinónimos contienen por lo general pocos elementos lo más efectivo es efectuar las inserciones por el inicio (no hace falta usar fantasmas, cierre circular, etc.). Existen variantes en que las listas de sinónimos están ordenadas o se sustituyen por árboles de búsqueda, pero no comportan una ventaja real en términos prácticos (el coste asintótico teórico es mejor, pero el criterio no es efectivo ya que el número de elementos involucrado es pequeño).

La búsqueda es también simple: se accede a la lista apropiada mediante la función de hash y se realiza un recorrido secuencial de la lista hasta que se encuentra un nodo con la clave dada o la lista se ha examinado sin éxito.

```
template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void Diccionario<Clave,Valor,HashFunct>::inserta(const Clave& k,
                                                 const Valor& v) throw(error) {

    if (_n / _M > _alfa_max)
        // la tabla está demasiado llena ... enviar un error
        // o redimensionar

    inserta_sep_chain(k, v);
}

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void Diccionario<Clave,Valor,HashFunct>::inserta_sep_chain(
    const Clave& k, const Valor& v) throw(error) {

    int i = hash(k);
    nodo* p = _Thash[i];

    // buscamos en la lista de sinónimos
    while (p != NULL and p -> _k != k)
        p = p -> _sig;

    // lo insertamos al principio
    // si no estaba
    if (p == NULL) {
        _Thash[i] = new nodo(k, v, _Thash[i]);
        ++_n;
    }
    else
        p -> _v = v;
}
```

```
template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void Diccionario<Clave,Valor,HashFunct>::busca(const Clave& k,
                                                bool& esta, Valor& v) const throw(error) {

    nodo* p = buscar_sep_chain(k);
    if (p == NULL)
        esta = false;
    else {
        esta = true;
        v = p -> _v;
    }
}

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
Diccionario<Clave,Valor,HashFunct>::nodo*
Diccionario<Clave,Valor,HashFunct>::busca_sep_chain(const
                                                     Clave& k) const throw() {

    int i = hash(k);
    nodo* p = _Thash[i];

    // buscamos en su lista de sinónimos
    while (p != NULL and p -> _k != k)
        p = p -> _sig;

    return p;
}
```

```
template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void Diccionario<Clave,Valor, HashFunct>::elimina(const
          Clave& k) throw() {

    eliminar_sep_chain(k);
}

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void Diccionario<Clave,Valor,HashFunct>::eliminar_sep_chain
(const Clave& k) throw() {

    int i = hash(k);
    nodo* p = _Thash[i];
    nodo* ant = NULL; // apuntara al anterior de p

    // buscamos en su lista de sinónimos
    while (p != NULL and p -> _k != k) {
        ant = p;
        p = p -> _sig;
    }

    // si p != NULL, lo quitamos de la
    // lista teniendo en cuenta que puede
    // ser el primero
    if (p != NULL) {
        --_n;
        if (ant != NULL)
            ant -> _sig = p -> _sig;
        else
            _Thash[i] = p -> _sig;
        delete p;
    }
}
```

Sea n el número de elementos almacenados en la tabla con encadenamiento de sinónimos. En promedio, cada lista tendrá $\alpha = n/M$ elementos y el coste de búsquedas (con y sin éxito), de inserciones y borrados será proporcional a α . Si α es un valor razonablemente pequeño entonces podemos considerar que el coste promedio de todas las operaciones sobre la tabla de hash es $\Theta(1)$, ya que α es constante.

Al valor α se le denomina **factor de carga** (ing: *load factor*). Otra variantes de hash abierto almacenan los sinónimos en una zona particular de la propia tabla, la llamada **zona de excedentes** (ing: *cellar*). Las posiciones de esta zona no son direccionables, es decir, la función de hash nunca toma sus valores.

Direccionamiento abierto

En las estrategias de direccionamiento abierto los sinónimos se almacenan en la tabla de hash, en la zona de direccionamiento. Básicamente, tanto en búsquedas como en inserciones se realiza una secuencia de sondeos que comienza en la posición $i_0 = h(k)$ y a partir de ahí continúa en i_1, i_2, \dots hasta que encontramos una posición ocupada por un elemento cuya clave es k (búsqueda con éxito), una posición libre (búsqueda sin éxito, inserción) o hemos explorado la tabla en su totalidad. Las distintas estrategias varían según la secuencia de sondeos que realizan. La más sencilla de todas es el sondeo lineal: $i_1 = i_0 + 1, i_2 = i_1 + 1, \dots$, realizándose los incrementos módulo M .

Sondeo lineal

```
template <typename Clave, typename Valor,
          template <typename> class HashFunct = Hash>

class Diccionario {
    ...
private:
    struct nodo {
        Clave _k;
        Valor _v;
        bool _libre;
        // constructora de la clase nodo
        nodo(const Clave& k, const Valor& v, bool libre = true);
    };
    nodo* _hash; // tabla de nodos
    int _M; // tamaño de la tabla
    int _n; // numero de elementos
    double _alfa_max; // factor de carga

    int busca_linear_probing(const Clave& k) const throw();
    void inserta_linear_probing(const Clave& k,
                                const Valor& v) throw(error);
    void elimina_linear_probing(const Clave& k) throw();
};
```

$$M = 13$$

$$X = \{ 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30 \}$$

$$h(x) = x \bmod M \quad (\text{incremento } 1)$$

| | |
|----|----|
| 0 | 0 |
| 1 | |
| 2 | |
| 3 | |
| 4 | 4 |
| 5 | |
| 6 | 6 |
| 7 | |
| 8 | |
| 9 | |
| 10 | 10 |
| 11 | |
| 12 | 12 |



| | | |
|----|----|----------|
| 0 | 0 | occupied |
| 1 | 13 | occupied |
| 2 | | free |
| 3 | | free |
| 4 | 4 | occupied |
| 5 | 17 | occupied |
| 6 | 6 | occupied |
| 7 | 19 | occupied |
| 8 | | free |
| 9 | | free |
| 10 | 10 | occupied |
| 11 | 23 | occupied |
| 12 | 12 | occupied |

$$+ \{ 13, 17, 19, 23 \}$$

| | | |
|----|----|----------|
| 0 | 0 | occupied |
| 1 | 13 | occupied |
| 2 | 25 | occupied |
| 3 | | free |
| 4 | 4 | occupied |
| 5 | 17 | occupied |
| 6 | 6 | occupied |
| 7 | 19 | occupied |
| 8 | 30 | occupied |
| 9 | | free |
| 10 | 10 | occupied |
| 11 | 23 | occupied |
| 12 | 12 | occupied |

$$+ \{ 25, 30 \}$$

```
// Solo se invoca si hay al menos un sitio
// no ocupado en la tabla: _n < _M
template <typename Clave, typename Valor,
          template <typename> class HashFunct>
void
Diccionario<Clave,Valor,HashFunct>::inserta_linear_probing(
    const Clave& k, const Valor& v) throw(error) {
    int i = hash(k);
    while (not _Thash[i]._libre and _Thash[i]._k != k)
        i = (i + 1) % _M;
    _Thash[i]._k = k; _Thash[i]._v = v;
    if (_Thash[i]._libre) ++_n;
    _Thash[i]._libre = false;
}
```

```

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
int Diccionario<Clave,Valor,HashFunct>::busca(
    const Clave& k,
    bool& esta, Valor& v) const throw(error) {

    int i = busca_linear_probing(k);

    if (not _Thash[i]._libre and _Thash[i]._k == k) {
        esta = true;
        v = _Thash[i]._v;
    }
    else
        esta = false;
}

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
int Diccionario<Clave,Valor,HashFunct>::busca_linear_probing(
    const Clave& k) const throw() {

    int i = hash(k);
    int vistos = 0; // para asegurarnos una pasada,
                    // no más; es innecesario si
                    // n < _M ya que entonces habrá
                    // una posición i tal que _libre == true

    while (not _Thash[i]._libre and _Thash[i]._k != k
           and vistos < _M) {
        ++vistos;
        i = (i + 1) % _M;
    }
    return i;
}

```

El borrado en tablas de direccionamiento abierto es algo más complicado. No basta con marcar la posición correspondiente como “libre” ya que una búsqueda posterior podría no hallar el elemento buscado aunque éste se encontrase en la tabla. Se ha de recorrer los elementos que siguen al que se borra en el mismo *clúster* desplazando al lugar desocupado a todo elemento cuya posición inicial de hash preceda o sea igual a la posición desocupada. Para ello necesitamos la función $\text{displ}(j, i)$ que nos da la distancia entre las posiciones j e i en el orden cíclico: si $j > i$ hay que “dar la vuelta” pasando por la posición $M - 1$ y regresando a la posición 0.

```
int displ(j, i, M) {
    if (i >= j)
        return i - j;
    else
        return M + (i - j);
}
```

```
// asumimos para simplificar que _n < _M

template <typename Clave, typename Valor,
          template <typename> class HashFunct>
int Diccionario<Clave,Valor,HashFunct>::elimina_linear_probing(
    const Clave& k) const throw() {

    int i = busca_linear_probing(k);

    if (not _Thash[i]._libre) {
        // _Thash[i] es el elemento que se quiere eliminar
        int free = i; i = (i + 1) % _M; int d = 1;
        while (not _Thash[i]._libre) {
            int i_home = hash(_Thash[i]._k);
            if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
            }
            i = (i + 1) % _M; ++d;
        }
        _Thash[free]._libre = true; --_n;
    }
}
```

El sondeo lineal ofrece algunas ventajas ya que los sinónimos tienden a encontrarse en posiciones consecutivas, lo que resulta ventajoso en implementaciones sobre memoria externa. Generalmente se usaría junto a alguna técnica de *bucketing*: cada posición de la tabla es capaz de albergar $b > 1$ elementos. Por otra parte tiene algunas serias desventajas y su rendimiento es especialmente sensible al factor de carga $\alpha = n/M$.

Si α es próximo a 1 (tabla llena o casi llena) el rendimiento será muy pobre. Si $\alpha < 1$ el coste de las búsquedas con éxito (y modificaciones) será proporcional a

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

y el de búsquedas sin éxito e inserciones será proporcional a

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right).$$

Un fenómeno indeseable que se acentúa cuando α es próximo a 1 es el *apiñamiento* (ing: *clustering*). Muchos elementos no pueden ocupar su posición “preferida” al estar ocupada por otro elemento que no tiene porque ser un sinónimo (*invasor*). Grupos de sinónimos más o menos dispersos acaban fundiéndose en grandes “clusters” y cuando buscamos una cierta clave tenemos que examinar no sólo sus sinónimos sino un buen número de claves que no tienen relación alguna con la clave buscada.

Redimensionamiento

Muchos lenguajes de programación permiten crear tablas fijando su tamaño en tiempo de ejecución. Entonces es posible utilizar la llamada técnica de *redimensionamiento* (ing: *resizing*). Si el factor de carga es muy alto, superando un cierto valor umbral, se reclama a la memoria dinámica una tabla cuyo tamaño es aproximadamente el doble del tamaño de la tabla en curso y se reinserta toda la información de la tabla en curso sobre la nueva tabla.

Para ello se recorre secuencialmente la tabla en curso y cada uno de los elementos presentes se inserta por el procedimiento habitual en la nueva tabla, usando una función de hash distinta, obviamente. La operación de redimensionamiento requiere tiempo proporcional al tamaño de la tabla en curso (y por tanto $\Theta(n)$); pero hay que hacerla sólo muy de vez en cuando. El redimensionamiento permite que el diccionario crezca sin límites prefijados, garantizando un buen rendimiento de todas las operaciones y sin desperdiciar demasiada memoria. La misma técnica puede aplicarse a la inversa, para evitar que el factor de carga sea excesivamente bajo (desperdicio de memoria).

Puede demostrarse que, aunque una operación individual podría llegar tener coste $\Theta(n)$, una secuencia de n operaciones de actualización tendrá coste total $\Theta(n)$.

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- Tablas de Hash
- **Colas de prioridad**
- Grafos y Recorridos

Una **cola de prioridad** (cat: *cua de prioritat*; ing: *priority queue*) es una colección de elementos donde cada elemento tiene asociado un valor susceptible de ordenación denominado **prioridad**.

Una cola de prioridad se caracteriza por admitir inserciones de nuevos elementos y la consulta y eliminación del elemento de mínima (o máxima) prioridad.

```
template <typename Elemt, typename Prio>
class ColaPrioridad {
public:
    ...
    // Añade el elemento x con prioridad p a la cola de
    // prioridad.
    void inserta(cons Elemt& x, const Prio& p) throw(error)

    // Devuelve un elemento de mínima prioridad en la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    Elemt min() const throw(error);

    // Devuelve la mínima prioridad presente en la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    Prio prio_min() const throw(error);

    // Elimina un elemento de mínima prioridad de la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    void elim_min() throw(error);

    // Devuelve cierto si y sólo si la cola está vacía.
    bool vacia() const throw();
};
```

```
// Tenemos dos arrays Peso y Simb con los pesos atómicos
// y símbolos de n elementos químicos,
// p.e., Simb[i] = "C" y Peso[i] = 12.2.
// Utilizamos una cola de prioridad para ordenar la
// información de menor a mayor símbolo en orden alfabético
ColaPrioridad<double, string> P;
for (int i = 0; i < n; ++i)
    P.inserta(Peso[i], Simb[i]);
int i = 0;
while(not P.vacia()) {
    Peso[i] = P.min();
    Simb[i] = P.prio_min();
    ++i;
    P.elim_min();
}
```

Se puede usar una cola de prioridad para hallar el k -ésimo elemento de un vector no ordenado. Se colocan los k primeros elementos del vector en una max-cola de prioridad y a continuación se recorre el resto del vector, actualizando la cola de prioridad cada vez que el elemento es menor que el mayor de los elementos de la cola, eliminando al máximo e insertando el elemento en curso.

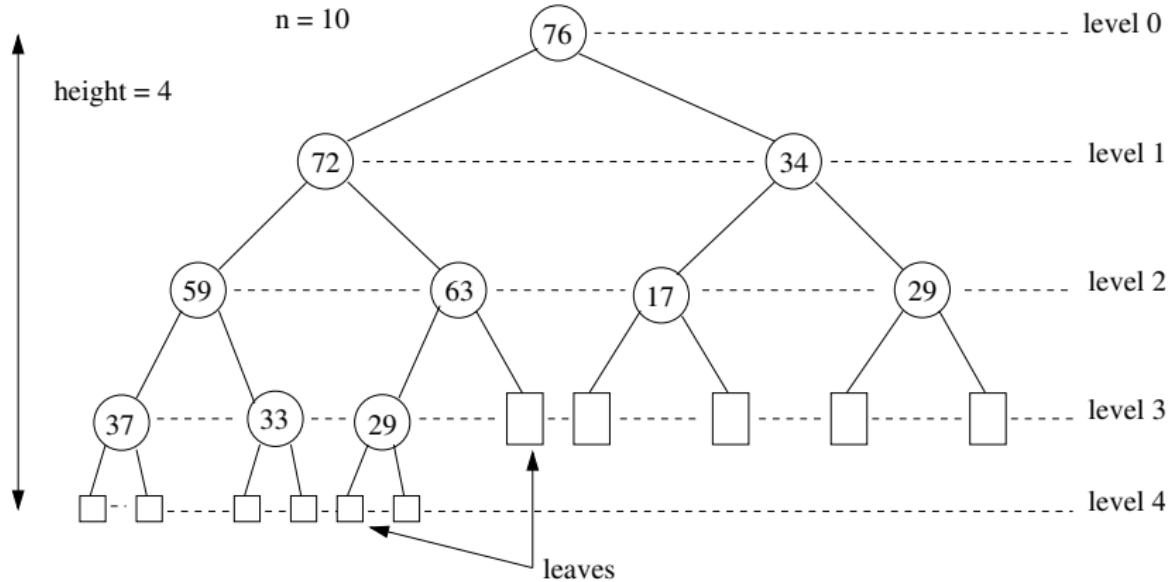
- Muchas de las técnicas empleadas para la implementación de diccionarios puede usarse para implementar colas de prioridad (no las tablas de hash ni los tries)
- P.e., con árboles binarios de búsqueda equilibrados se puede conseguir coste $\mathcal{O}(\log n)$ para inserciones y eliminaciones

Definición

Un *montículo* (ing: **heap**) es un árbol binario tal que

- ① todos las hojas (subárboles son vacíos) se sitúan en los dos últimos niveles del árbol.
- ② en el antepenúltimo nivel existe a lo sumo un nodo interno con un sólo hijo, que será su hijo izquierdo, y todos los nodos a su derecha en el mismo nivel son nodos internos sin hijos.
- ③ el elemento (su prioridad) almacenado en un nodo cualquiera es mayor (menor) o igual que los elementos almacenados en sus hijos izquierdo y derecho.

Se dice que un montículo es un árbol binario quasi-completo debido a las propiedades 1-2. La propiedad 3 se denomina **orden de montículo**, y se habla de **max-heaps** o **min-heaps** según que los elementos sean \geq ó \leq que sus hijos.



Proposición

- ① *El elemento máximo de un max-heap se encuentra en la raíz.*
- ② *Un heap de n elementos tiene altura*

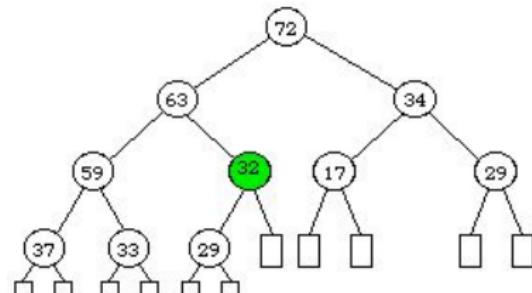
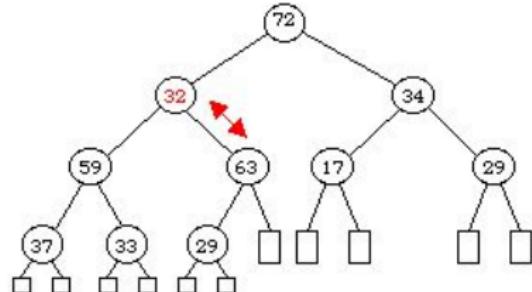
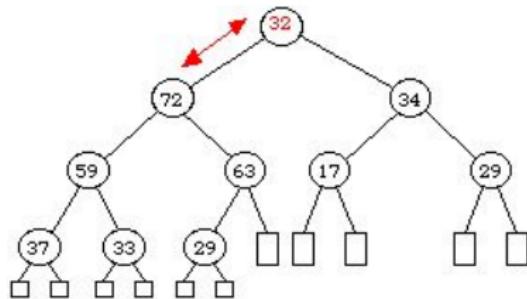
$$h = \lceil \log_2(n + 1) \rceil.$$

La consulta del máximo es sencilla y eficiente pues basta examinar la raíz.

Cómo eliminar el máximo?

- 1 Ubicar al último elemento del montículo (el del último nivel más a la derecha) en la raíz, sustituyendo al máximo
- 2 Reestablecer el invariante (orden de heap) **hundiendo** la raíz.

El método `hundir` intercambia un nodo dado con el mayor de sus dos hijos si el nodo es menor que alguno de ellos, y repite este paso hasta que el invariante de heap se ha reestablecido



Cómo añadir un nuevo elemento?

- ① Colocar el nuevo elemento como último elemento del montículo, justo a la derecha del último o como primero de un nuevo nivel
- ② Reestablecer el orden de montículo **flotando** el elemento recién añadido

En el método `flotar` el nodo dado se compara con su nodo padre y se realiza el intercambio si éste es mayor que el padre, iterando este paso mientras sea necesario

Puesto que la altura del *heap* es $\Theta(\log n)$ el coste de inserciones y eliminaciones será $\mathcal{O}(\log n)$.

Se puede implementar un *heap* mediante memoria dinámica, con apuntadores al hijo izquierdo y derecho y **también** al padre en cada nodo

Pero es mucho más fácil y eficiente implementar los *heaps* mediante un vector. No se desperdicia demasiado espacio ya que el *heap* es quasi-completo; en caso necesario puede usarse el redimensionado

Reglas para representar un *heap* en vector:

- ① $A[1]$ contiene la raíz.
- ② Si $2i \leq n$ entonces $A[2i]$ contiene al hijo izquierdo del elemento en $A[i]$ y si $2i + 1 \leq n$ entonces $A[2i + 1]$ contiene al hijo derecho de $A[i]$
- ③ Si $i \geq 2$ entonces $A[i/2]$ contiene al padre de $A[i]$

Las reglas anteriores implican que los elementos del heap se ubican en posiciones consecutivas del vector, colocando la raíz en la primera posición y recorriendo el árbol por niveles, de izquierda a derecha.

```
template <typename Elemt, typename Prio>
class ColaPrioridad {
public:
    ...
private:
    // la componente 0 no se usa; el constructor de la clase
    // inserta un elemento ficticio
    vector<pair<Elemt, Prio> > h;

    int nelems;

    void flotar(int j) throw();
    void hundir(int j) throw();
};
```

```
template <typename Elemtypename Prio>
bool ColaPrioridad<Elemtypename Prio>::vacia() const throw() {
    return nelems == 0;
}

template <typename Elemtypename Prio>
Elemtypename ColaPrioridad<Elemtypename Prio>::min() const throw(error) {
    if (nelems == 0) throw error(ColaVacia);
    return h[1].first;
}

template <typename Elemtypename Prio>
Priotypename ColaPrioridad<Elemtypename Prio>::prio_min() const throw(error) {
    if (nelems == 0) throw error(ColaVacia);
    return h[1].second;
}
```

```
template <typename Elemtypename Prio>
void ColaPrioridad<Elemtypename Prio>::inserta(cons Elemtypename x,
                                                cons Prio& p) throw(error) {
    ++nelems;
    h.push_back(make_pair(x, p));
    flotar(nelems);
}

template <typename Elemtypename Prio>
void ColaPrioridad<Elemtypename Prio>::elim_min() const throw(error) {

    if (nelems == 0) throw error(ColaVacia);
    swap(h[1], h[nelems]);
    --nelems;
    h.pop_back();
    hundir(1);
}
```

```
// Versión recursiva.  
// Hunde el elemento en la posición j del heap  
// hasta reestablecer el orden del heap; por  
// hipótesis los subárboles del nodo j son heaps.  
  
// Coste: O(log(n/j))  
template <typename Elemt, typename Prio>  
void ColaPrioridad<Elemt,Prio>::hundir(int j) throw() {  
  
    // si j no tiene hijo izquierdo, hemos terminado  
    if (2 * j > nelems) return;  
  
    int minhijo = 2 * j;  
    if (minhijo < nelems and  
        h[minhijo].second > h[minhijo + 1].second)  
        ++minhijo;  
  
    // minhijo apunta al hijo de mínima prioridad de j  
    // si la prioridad de j es mayor que la de su menor hijo  
    // intercambiar y seguir hundiendo  
    if (h[j].second > h[minhijo].second) {  
        swap(h[j], h[minhijo]);  
        hundir(minhijo);  
    }  
}
```

```
// Versión iterativa.  
// Hunde el elemento en la posición j del heap  
// hasta reestablecer el orden del heap; por  
// hipótesis los subárboles del nodo j son heaps.  
  
// Coste: O(log(n/j))  
template <typename Elemt, typename Prio>  
void ColaPrioridad<Elemt,Prio>::hundir(int j) throw() {  
  
    bool fin = false;  
    while (2 * j <= nelems and not fin) {  
        int minhijo = 2 * j;  
        if (minhijo < nelems and  
            h[minhijo].second > h[minhijo + 1].second)  
            ++minhijo;  
        if (h[j].second > h[minhijo].second) {  
            swap(h[j], h[minhijo]);  
            j = minhijo;  
        } else {  
            fin = true;  
        }  
    }  
}
```

```
// Flota al nodo j hasta reestablecer el orden del heap;
// todos los nodos excepto el j satisfacen la propiedad
// de heap

// Coste: O(log j)
template <typename Elemt, typename Prio>
void ColaPrioridad<Elemt,Prio>::flotar(int j) throw() {

    // si j es la raíz, hemos terminado
    if (j == 1) return;

    int padre = j / 2;
    // si el padre tiene mayor prioridad
    // que j, intercambiar y seguir flotando
    if (h[j].second < h[padre].second) {
        swap(h[j], h[padre]);
        flotar(padre);
    }
}
```

Heapsort

Heapsort (Williams, 1964) ordena un vector de n elementos construyendo un *heap* con los n elementos y extrayéndolos, uno a uno del *heap* a continuación. El propio vector que almacena a los n elementos se emplea para construir el *heap*, de modo que **heapsort** actúa *in-situ* y sólo requiere un espacio auxiliar de memoria constante. El coste de este algoritmo es $\Theta(n \log n)$ (incluso en caso mejor) si todos los elementos son diferentes.

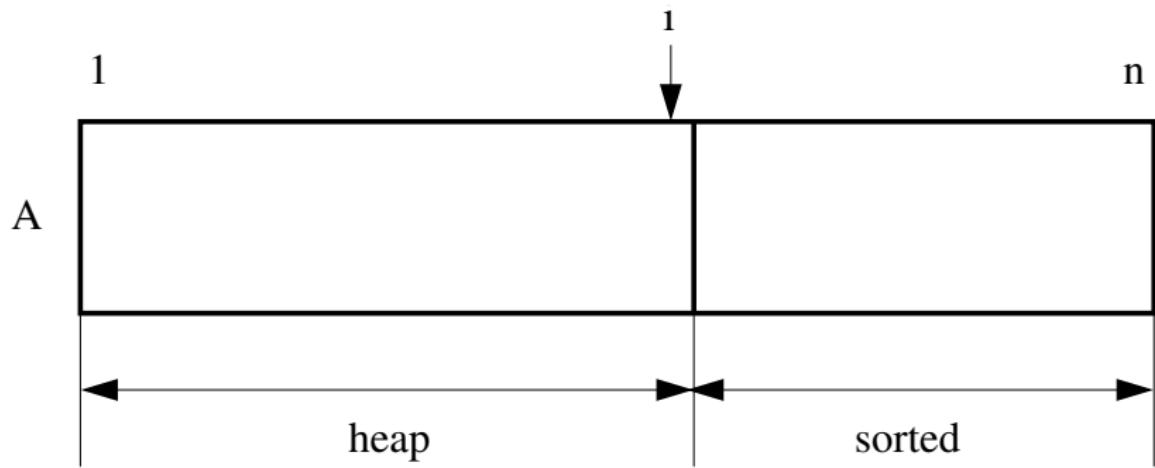
En la práctica su coste es superior al de quicksort, ya que el factor constante multiplicativo del término $n \log n$ es mayor.

```
// Ordena el vector v[1..n]
// (v[0] no se usa)
// de Elem's de menor a mayor

template <typename Elem>
void heapsort(Elem v[], int n) {

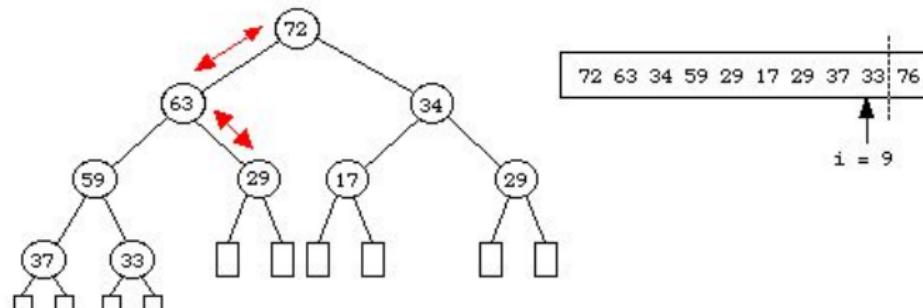
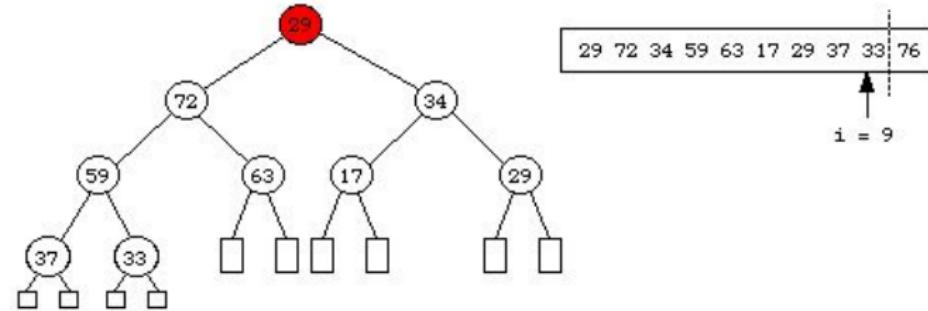
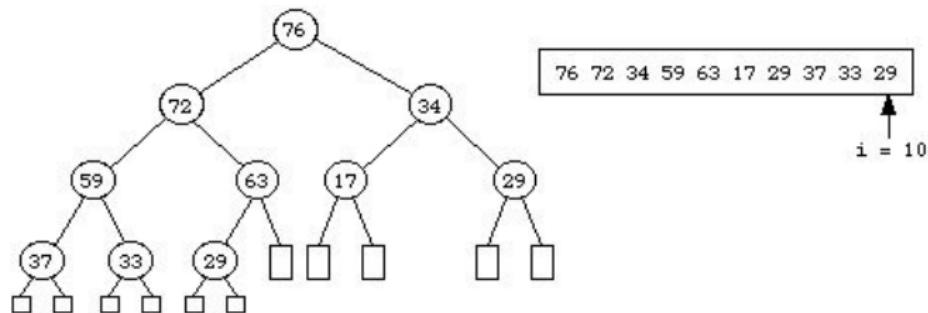
    crea_max_heap(v, n);
    for (int i = n; i > 0; --i) {
        // saca el mayor elemento del heap
        swap(v[1], v[i]);

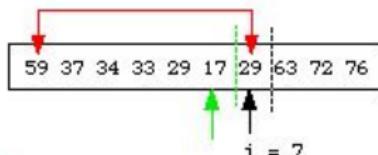
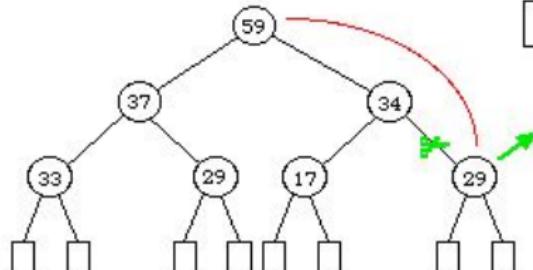
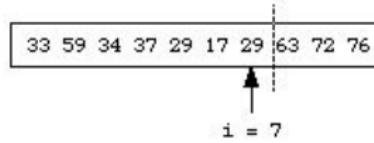
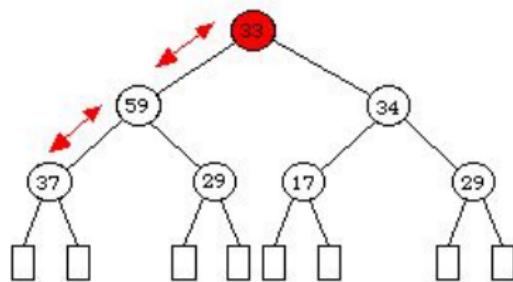
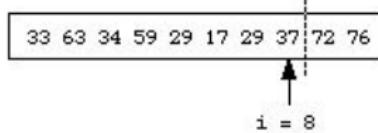
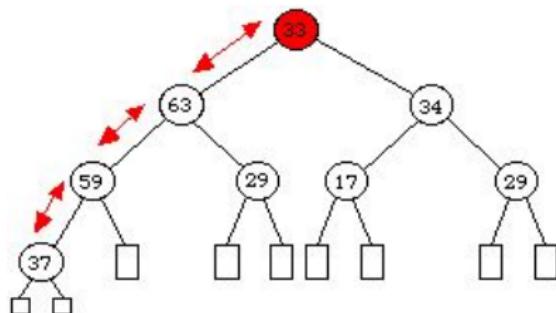
        // hunde el elemento de indice 1
        // para re establecer un max-heap en
        // el subvector v[1..i-1]
        hundir(v, i-1, 1);
    }
}
```



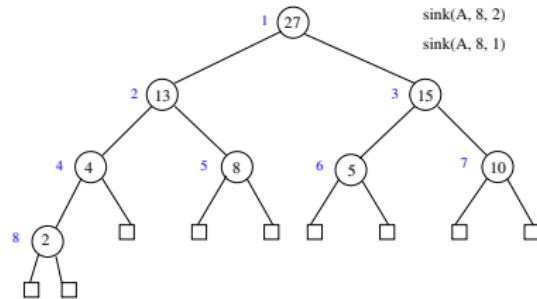
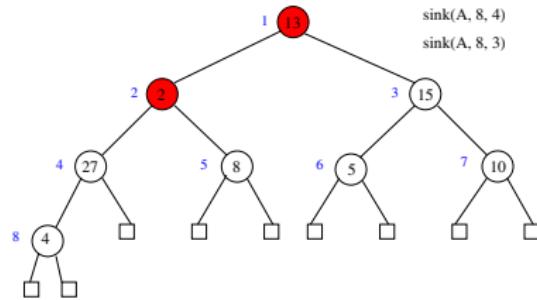
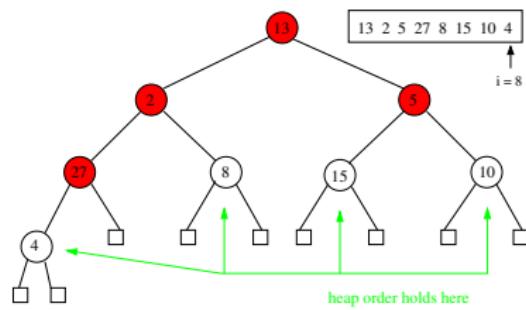
$$A[1] \leq A[i+1] \leq A[i+2] \leq \dots \leq A[n]$$

$$A[1] = \max_{1 \leq k \leq n} A[k]$$





```
// Da estructura de max-heap al
// vector v[1..n] de Elem's; aquí
// cada elemento se identifica con su
// prioridad
template <typename Elem>
void crea_max_heap(Elem v[], int n) {
    for (int i = n/2; i > 0; --i)
        hundir(v, n, i);
}
```



Sea $H(n)$ el coste en caso peor de heapsort y $B(n)$ el coste de crear el *heap* inicial. El coste en caso peor de `hundir(v, i - 1, 1)` es $\mathcal{O}(\log i)$ y por lo tanto

$$\begin{aligned} H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\ &= B(n) + \mathcal{O}\left(\sum_{1 \leq i \leq n} \log_2 i\right) \\ &= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n) \end{aligned}$$

Un análisis simple de $B(n)$ indica que $B(n) = \mathcal{O}(n \log n)$ ya que hay $\Theta(n)$ llamadas a `hundir`, cada una de las cuales tiene coste $\mathcal{O}(\log n)$

Por tanto $H(n) = \mathcal{O}(n \log n)$, de hecho $H(n) = \Theta(n \log n)$ en caso peor

Podemos refinar el análisis de $B(n)$:

$$\begin{aligned}B(n) &= \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\&= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\&= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n)\end{aligned}$$

Puesto que $B(n) = \Omega(n)$, podemos afirmar que $B(n) = \Theta(n)$.

Demostración alternativa: Sea $h = \lceil \log_2(n + 1) \rceil$ la altura del *heap*. En el nivel $h - 1 - k$ hay como mucho

$$2^{h-1-k} < \frac{n+1}{2^k}$$

nodos y cada uno de ellos habrá de hundirse en caso peor hasta el nivel $h - 1$; eso tiene coste $\mathcal{O}(k)$

Por lo tanto,

$$\begin{aligned} B(n) &= \sum_{0 \leq k \leq h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\ &= \mathcal{O}\left(n \sum_{0 \leq k \leq h-1} \frac{k}{2^k}\right) \\ &= \mathcal{O}\left(n \sum_{k \geq 0} \frac{k}{2^k}\right) = \mathcal{O}(n), \end{aligned}$$

ya que

$$\sum_{k \geq 0} \frac{k}{2^k} = 2.$$

En general, si $|r| < 1$,

$$\sum_{k \geq 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$

Aunque $H(n) = \Theta(n \log n)$, el análisis detallado de $B(n)$ es importante: utilizando un *min-heap* podemos hallar los k menores elementos en orden creciente de un vector (y en particular el k -ésimo) con coste:

$$\begin{aligned} S(n, k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

y si $k = \mathcal{O}(n / \log n)$ entonces $S(n, k) = \mathcal{O}(n)$.

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Árboles binarios de búsqueda
- Árboles balanceados (AVLs)
- Tablas de Hash
- Colas de prioridad
- Grafos y Recorridos

Definición

Un **grafo (no dirigido)** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices** (también llamados **nodos**) y E es un conjunto de **aristas**; cada arista $e \in E$ es un par no ordenado $\{u, v\}$ donde u y v ($u \neq v$) son elementos de V .

Definición

Un **grafo dirigido** o **digrafo** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices o nodos** y E es un conjunto de **arcos**; cada arco $e \in E$ es un par (u, v) donde u y v ($u \neq v$) son elementos de V .

Si en vez de un conjunto de arcos o aristas tenemos multiconjuntos de arcos o aristas, y se permiten **bucles** (esto es, arcos de la forma (u, u) o aristas $\{u, u\}$) entonces tenemos **multigrafos** (dirigidos y no dirigidos, respectivamente).

Dado un arco $e = (u, v)$ se denomina **origen** a u y **destino** a v . Se dice que u es un **predecesor** de v y que v es un **sucesor** de u . Para una arista $e = \{u, v\}$ se dice que u y v son sus **extremos**, que la arista es **incidente** a u y v , que u y v son **adyacentes**.

Definición

Un **camino** $P = v_0, v_1, v_2, \dots, v_n$ de longitud n en un grafo $G = \langle V, E \rangle$ es una secuencia de vértices de G tal que para toda i , $0 \leq i < n$

$$\{v_i, v_{i+1}\} \in E$$

El vértice v_0 es el origen del camino P y v_n es su destino.

Un camino en un digrafo se define de manera análoga: para cada i , (v_i, v_{i+1}) es un arco del digrafo.

Se dice que un camino es **simple** si no se repite ningún vértice.
Un camino en el que $v_0 = v_n$ es un **ciclo**.

Definición

Un grafo $G = \langle V, E \rangle$ es **conexo** si y sólo si, para todo par de vértices u y v , existe un camino que va de u a v en G .

Para digrafos la definición es idéntica, sólo que entonces se dice que el digrafo es **fuertemente conexo**.

Definición

Dado un grafo $G = \langle V, E \rangle$, el grafo $H = \langle V', E' \rangle$ se dice que es un **subgrafo** de G si y sólo si $V' \subseteq V$, $E' \subseteq E$, y para toda arista $e' = (u', v') \in E'$ se cumple que u' y v' pertenecen a V' .

Si E' contiene todas las aristas de E que son incidentes a dos vértices de V' , se dice que H es el subgrafo **inducido** por V' .

La definición de subgrafo de un grafo dirigido es completamente análoga.

Definición

Una **componente conexa** C de un grafo G es un subgrafo inducido maximal conexo de G . Maximal quiere decir que si se añade cualquier vértice v a $V(C)$ entonces el subgrafo inducido correspondiente no es conexo—en particular, no existe camino entre v y los restantes vértices de C .

En el caso de digrafos, se habla de **componentes fuertemente conexas**: una componente fuertemente conexa del digrafo G es un subgrafo dirigido inducido maximal fuertemente conexo.

Definición

Un grafo conexo y sin ciclos se denomina **árbol (libre)**. Si G es un grafo conexo, un subgrafo $T = \langle V, E' \rangle$ (es decir que contiene los mismos vértices que G) es un **árbol de expansión** si T es un árbol.

Los árboles libres no tienen raíz, y no existe orden entre los vértices adyacentes a un vértice dado del árbol.

Lema

Si $G = \langle V, E \rangle$ es un árbol entonces $|E| = |V| - 1$.

A menudo trabajaremos con grafos o digrafos con **etiquetas** en las aristas o arcos: el etiquetado de un grafo o digrafo es una función $\phi : E \rightarrow \mathcal{L}$ entre el conjunto de aristas o arcos de G y el conjunto (eventualmente infinito) de etiquetas \mathcal{L} .

Cuando las etiquetas son números (enteros, racionales, reales), se dice que el grafo o digrafo es **ponderado** y a la etiqueta $\phi(e)$ de una arista o arco e se le suele denominar **peso**.

Implementación

Los grafos se implementan habitualmente de una de las dos siguientes formas:

- ① Mediante **matrices de adyacencia**: la componente $A[i, j]$ de la matriz nos dice si el arco entre los vértices i y j existe o no; eventualmente, $A[i, j]$ puede contener también el peso del arco entre los vértices i y j
- ② Mediante **listas de adyacencia**: tenemos una tabla T de listas enlazadas; la lista $T[i]$ es la lista de sucesores del vértice i

La implementación mediante matrices de adyacencia es muy costosa en espacio: si $|V| = n$, se requiere espacio $\Theta(n^2)$ para representar el grafo, independientemente del número de aristas/arcos que haya en el grafo. Sólo es interesante si necesitamos poder decidir la existencia (o no) de una arista o arco con máxima eficiencia.

Por regla general, la implementación que se usará es la de listas de adyacencia. El espacio que se requiere es $\Theta(n + m)$, donde $m = |E|$ y $n = |V|$; el espacio utilizado es por lo tanto lineal respecto al tamaño del grafo.

```
typedef int vertex;
typedef pair<vertex, vertex> edge;
typedef list<edge>::iterator edge_iter;

class Graph {
// grafos no dirigidos con V = {0, ..., n-1}
public:
// crea un grafo vacío (sin vértices y sin aristas)
Graph();

// crea un grafo con n vértices y sin aristas
Graph(int n);

// añade un vértice al grafo; el nuevo vértice tendrá
// identificador n, donde n era el número de vértices
// del grafo antes de añadir el vértice
void add_vertex();

// añade la arista (u,v) o e = (u,v) al grafo
void add_edge(vertex u, vertex v);
void add_edge(edge e);

// consultoras del numero de vertices y de aristas
int nr_vertices() const;
int nr_edges() const;

// devuelve lista de adyacentes a un vertice
list<edge> adjacent(vertex u) const;
...

private:
    int n, m;
    vector<list<edge> > T;
    ...
}
```

Recorridos

Definición

Dado un grafo conexo $G = \langle V, E \rangle$, un **recorrido** del grafo es una secuencia que contiene todos y cada uno de los vértices en $V(G)$ exactamente una vez y tal que para cualquier vértice v en la secuencia se cumple que, o bien v es el primer vértice de la secuencia, o bien existe una arista $e \in E(G)$ que une al vértice v con otro vértice que le precede en la secuencia.

El recorrido de un grafo es una secuencia de los recorridos de las componentes conexas de G . No hay ningún vértice que aparezca en el recorrido sin que se haya completado el recorrido de las componentes conexas correspondientes a los vértices que le preceden en el recorrido.

En el caso de los digrafos, un recorrido que comienza en un cierto vértice v visitará todos y cada uno de los vértices accesibles desde v exactamente una vez; la visita de un vértice $w \neq v$ implica que existe algún otro vértice visitado previamente del cual w es sucesor.

Los recorridos nos permiten visitar todos los vértices de un grafo, con arreglo a la topología del grafo, pues se utilizan las aristas o arcos del grafo para ir haciendo las sucesivas visitas.

Mediante un recorrido (o una combinación de recorridos) podemos resolver eficientemente numerosos problemas sobre grafos. Por ejemplo

- Determinar si un grafo es conexo o no
- Hallar las componentes conexas de un grafo no dirigido
- Hallar las componentes fuertemente conexas de un grafo dirigido
- Determinar si un grafo contiene ciclos o no
- Decidir si un grafo es 2-coloreable, equivalentemente, si es bipartido
- Decidir si un grafo es biconexo o no (la eliminación de un vértice no desconecta al grafo)
- Hallar el camino más corto (con menor número de aristas/arcos) entre dos vértices dados
- etc.

Recorrido en profundidad (DFS)

En el **recorrido en profundidad** (ing: *Depth-First Search*, DFS) de un grafo G , se visita un vértice v y desde éste, recursivamente, se realiza el recorrido de cada uno de los vértices adyacentes/sucesores de v no visitados. Cuando se visita por vez primera un vértice v se dice que se ha **abierto** el vértice; el vértice se **cierra** cuando se completa, recursivamente, el recorrido en profundidad de los vértices adyacentes/sucesores.

La numeración **directa** o **numeración DFS** de un vértice es el número de orden en el que se abre el vértice; así, el vértice en el que se inicia el recorrido tiene numeración directa 1.

La numeración **inversa** de un vértice es el número de orden en el que se cierra el vértice. El primer vértice del recorrido para el cual todos sus vecinos/sucesores ya han sido visitados tiene numeración inversa 1, y así sucesivamente.

▷ *visitado*, *ndfs*, *ninv*, *num_dfs*, *num_inv* son variables globales

procedure DFS(*G*)

for *v* $\in V(G)$ **do**

visitado[*v*] := **false**

ndfs[*v*] := 0; *ninv*[*v*] := 0

end for

num_dfs := 0; *num_inv* := 0

for *v* $\in V(G)$ **do**

if \neg *visitado*[*v*] **then**

 DFS-REC(*G*, *v*, *v*)

end if

end for

end procedure

```

procedure DFS-REC( $G, v, \text{padre}$ )
    PRE-VISIT( $v$ )
     $\text{visitado}[v] := \text{true}$ 
     $\text{num\_dfs} := \text{num\_dfs} + 1; \text{ndfs}[v] := \text{num\_dfs}$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg \text{visitado}[w]$  then
            PRE-VISIT-EDGE( $v, w$ )
            DFS-REC( $G, w, v$ )
            POST-VISIT-EDGE( $v, w$ )
        else
             $\triangleright$  si  $w \neq \text{padre}$  entonces hemos encontrado un ciclo
        end if
    end for
    POST-VISIT( $v$ )
     $\text{num\_inv} := \text{num\_inv} + 1; \text{ ninv}[v] := \text{num\_inv}$ 
end procedure

```

```
// Macros para escribir recorridos en un grafo
#define forall(v,G) for(vertex (v) = 0; (v) < (G).nr_vertices(); ++(v) )

#define forall_adj(e, u, G) \
    for(edge_iter (e) = (G).adjacent((u)).begin(); \
        (e) != (G).adjacent((u))].end() ; ++(e) )

#define target(eit) ((eit) -> second)
#define source(eit) ((eit) -> first)
```

```
void DFS(const Graph& G) {
    vector<bool> visitado(G.nr_vertices(), false);
    vector<int> ndfs(G.nr_vertices(), 0);
    vector<int> ninv(G.nr_vertices(), 0);
    int num_dfs = 0;
    int num_inv = 0;

    forall(v, G)
        if (not visitado[v])
            DFS(G, v, v, visitado, num_dfs, num_inv, ndfs, ninv);
}
```

```
void DFS(const Graph& G, vertex v, vertex padre,
         vector<bool>& visitado,
         int& num_dfs, int& num_inv,
         vector<int>& ndfs,
         vector<int>& ninv) {

    PRE-VISIT(v);
    visitado[v] = true;
    ++num_dfs; ndfs[v] = num_dfs;
    forall_adj(e, v, G) {
        vertex w = target(e);
        if (not visitado[w]) {
            PRE-VISIT-EDGE(v,w);
            DFS(G, w, v, visitado, num_dfs, num_inv, ndfs, ninv);
            POST-VISIT-EDGE(v,w);
        } else {
            // si w != padre entonces hemos encontrado un ciclo
        }
    }
    POST-VISIT(v);
    ++num_inv; ninv[v] = num_inv;
}
```

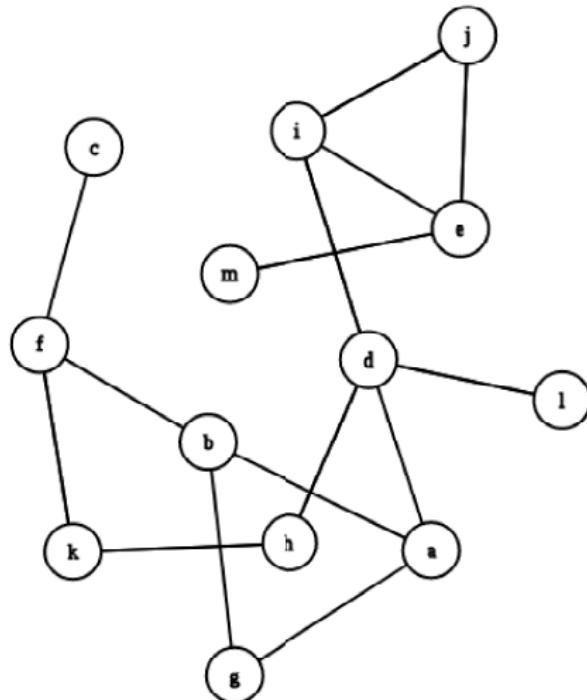
El recorrido en profundidad de una componente conexa induce un árbol de expansión, al que se denomina árbol del recorrido T_{DFS} .

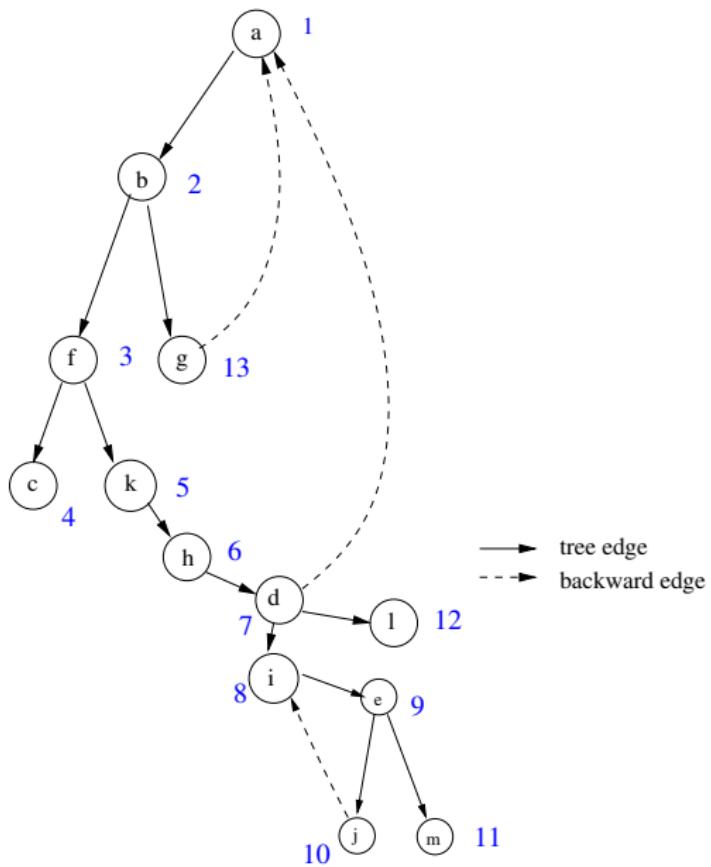
Las aristas de la componente conexa se clasifican entonces en **aristas de árbol** (*tree edges*) y **aristas de retroceso** (*backward edges*). Éstas últimas unen el vértice que se acaba de abrir con un vértice visitado (abierto o cerrado) y por tanto cierran ciclos. El recorrido de todo el grafo da lugar a un bosque de expansión.

Para cada vértice v , $CC[v]$ será el número de la componente conexa en la que está v . $TreeEdges[i]$ es el conjunto de aristas de árbol en la componente i , y $BackEdges[i]$ el conjunto de aristas de retroceso. El número de componentes conexas viene dado por ncc .

```
procedure DFS( $G$ )
  for  $v \in V(G)$  do
     $visitado[v] := \text{false}$ 
     $CC[v] := 0$ 
  end for
   $ncc := 0$ 
  for  $v \in V(G)$  do
    if  $\neg visitado[v]$  then
       $ncc := ncc + 1$ 
       $TreeEdges[ncc] := \emptyset$ 
       $BackEdges[ncc] := \emptyset$ 
      DFS-REC( $G, v, v$ )
    end if
  end for
end procedure
```

```
procedure DFS-REC( $G, v, \text{padre}$ )
     $\text{visitado}[v] := \text{true}$ 
     $CC[v] := ncc$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg \text{visitado}[w]$  then
             $TreeEdges[ncc] := TreeEdges[ncc] \cup \{(v, w)\}$ 
            DFS-REC( $G, w, v$ )
        else if  $w \neq \text{padre}$  then
             $BackEdges[ncc] := BackEdges[ncc] \cup \{(v, w)\}$ 
        end if
    end for
end procedure
```





Aplicaciones del DFS

Detectar ciclos o calcular las componentes conexas son ejemplos triviales de aplicación del recorrido en profundidad. Otro algoritmo simple que se puede diseñar aplicando el DFS es determinar si un grafo es **bipartido**. Recordemos que un grafo G es bipartido si existe una partición $\langle A, B \rangle$ del conjunto de vértices V (es decir, $A \cup B = V; A \cap B = \emptyset$) tal que toda arista de G tiene un extremo en A y el otro en B :
Equivalentemente, G es bipartido si y sólo si es 2-coloreable, y si y sólo si no contiene ningún ciclo de longitud impar (porqué?)

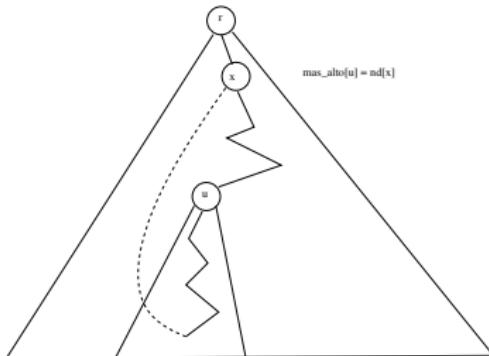
```
procedure ESBIPARTIDO( $G$ )
  for  $v \in V(G)$  do
     $color[v] := -1$ 
  end for
   $color := 0$ 
   $es\_bip := \text{true}$ 
  for  $v \in V(G)$  while  $es\_bip$  do
    if  $color[v] = -1$  then
       $es\_bip := \text{ESBIPARTIDO-REC}(G, v, color)$ 
    end if
  end for
  return  $es\_bip$ 
end procedure
```

```
procedure ESBIPARTIDO-REC( $G, v, c$ )
     $es\_bip := \text{true}$ 
     $color[v] := c$ 
    for  $w \in G.\text{ADJACENT}(v)$  while  $es\_bip$  do
        if  $color[w] = -1$  then
             $es\_bip := \text{ESBIPARTIDO-REC}(G, w, v, 1 - c)$ 
        else
             $es\_bip := (c \neq color[w])$ 
        end if
    end for
    return  $es\_bip$ 
end procedure
```

Aplicaciones del DFS: Grafos Biconexos

Un grafo conexo G se dice que es **biconexo** si al eliminar uno cualquiera de sus vértices sigue siendo conexo.

Supongamos que efectuamos un DFS empezando en el vértice r y que para cada vértice u determinamos el número DFS más bajo que es alcanzable siguiendo un camino desde u hasta uno de sus descendientes en el T_{DFS} y a continuación “subimos” con una arista de retroceso. Denominaremos $\text{mas_alto}[u]$ a dicho número.



Un vértice u es un **punto de articulación** de G si su eliminación desconecta el grafo. Si G no tiene puntos de articulación entonces G es biconexo. Para ver si un vértice es punto de articulación o no deberemos considerar varios casos:

- ① u es una hoja del T_{DFS} (no tiene descendientes): entonces no es un punto de articulación ya que nunca se desconectaría el grafo
- ② $u = r$ es la raíz del T_{DFS} : es punto de articulación si y sólo si tiene más de un descendiente en el T_{DFS} , su eliminación desconectaría los subárboles, pero si sólo hay uno su eliminación no desconecta a los demás vértices.

- ③ u no es hoja y no es la raíz: entonces es punto de articulación si y sólo si alguno de los vértices adyacentes descendientes w de u cumple $\text{mas_alto}[w] \geq \text{ndfs}[u]$, es decir, si algún descendiente no puede “escapar” del subárbol enraizado en u sin pasar por u .

Por otro lado para cada vértice u , $\text{mas_alto}[u]$ es el mínimo entre: 1) $\text{ndfs}[u]$, 2) $\text{mas_alto}[v]$ para todo v descendiente directo de u en el T_{DFS} y 3) $\text{ndfs}[v]$ para todo v adyacente a u mediante una arista de retroceso.

```
procedure BICONEXO( $G$ )
Require:  $G$  es conexo
    for  $v \in V(G)$  do
         $visitado[v] := \text{false}$ 
         $ndfs[v] := 0$ 
         $punto\_art[v] := \text{false}$ 
         $num\_desc[v] := 0$ 
    end for
     $num\_dfs := 0$ 
     $es\_biconexo := \text{true}$ 
    ▷ Basta lanzar un DFS porque  $G$  es conexo
     $v :=$  un vértice cualquiera de  $G$ 
    BICONEXO-REC( $G, v, v, es\_biconexo, ndfs, \dots$ )
    return  $es\_biconexo$ 
end procedure
```

```

procedure BICONEXO-REC( $G, v, \text{padre}, \dots$ )
     $\text{num\_dfs} := \text{num\_dfs} + 1$ ;  $\text{ndfs}[v] := \text{num\_dfs}$ 
     $\text{visitado}[v] := \text{true}$ 
     $\text{mas\_alto}[v] := \text{ndfs}[v]$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg \text{visitado}[w]$  then
             $\text{num\_desc}[v] := \text{num\_desc}[v] + 1$ 
            BICONEXO-REC( $G, w, v, \dots$ )
             $\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{mas\_alto}[w])$ 
             $\text{punto\_art}[v] := \text{punto\_art}[v] \vee (\text{mas\_alto}[w] \geq \text{ndfs}[v])$ 
        else if  $\text{padre} \neq w$  then
             $\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{ndfs}[w])$ 
        end if
    end for
    if  $v = \text{padre}$  then  $\triangleright v$  es la raíz del  $T_{\text{DFS}}$ 
         $\text{punto\_art}[v] := \text{num\_desc}[v] > 1$ 
    end if
     $\text{es\_biconexo} := \text{es\_biconexo} \wedge \neg \text{punto\_art}[v]$ 
end procedure

```

Análisis del DFS

Supongamos que el trabajo que se realiza en cada visita de un vértice (PRE- y POST-VISIT) y cada visita de una arista, sea del árbol o de retroceso, es $\Theta(1)$. Entonces el coste del DFS es

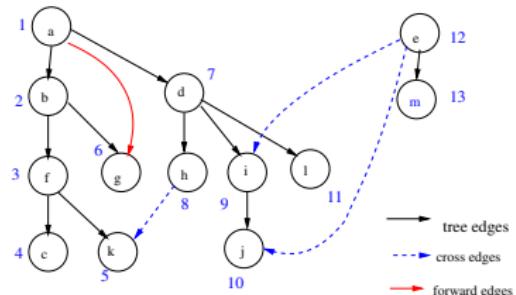
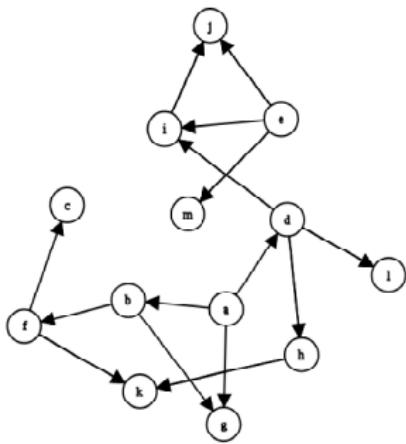
$$\begin{aligned} \sum_{v \in V} (\Theta(1) + \Theta(\text{grado}(v))) &= \Theta\left(\sum_{v \in V} (1 + \text{grado}(v))\right) = \\ &\Theta\left(\sum_{v \in V} 1 + \sum_{v \in V} \text{grado}(v)\right) = \Theta(n + m) \end{aligned}$$

DFS en Digrafos

El recorrido en profundidad de un digrafo tiene propiedades algo distintas de las del DFS de un grafo no dirigido. Al lanzar un DFS desde un vértice v de un digrafo G se visitarán los vértices (no visitados) de la componente fuertemente conexa de v pero además todos los otros vértices **accesibles** desde v . El DFS de un digrafo induce sucesivos árboles de recorrido, enraizados en los vértices desde los cuales se lanzan los recorridos recursivos. Los conceptos de numeración DFS y de numeración inversa son igual que en el caso de grafos no dirigidos.

El DFS de un digrafo también induce una clasificación de los arcos, pero es algo diferente:

- ① Arcos de árbol: van del vértice v en curso a un vértice no visitado w
- ② Arcos de retroceso: van del vértice v en curso a un vértice antecesor w en el árbol T_{DFS} ; se cumple que $\text{ndfs}[w] < \text{ndfs}[v]$ y que $\text{ndfs}[w]$ está abierto
- ③ **Arcos de avance** (*forward edges*): van del vértice v en curso a un vértice descendiente pero previamente visitado w en el T_{DFS} ; se cumple que $\text{ndfs}[v] < \text{ndfs}[w]$
- ④ **Arcos de cruce** (*cross edges*): van del vértice v en curso a un vértice previamente visitado w en el mismo T_{DFS} o en un árbol diferente; se cumple que $\text{ndfs}[w] < \text{ndfs}[v]$, pero el vértice w ya está cerrado ($ninv[w] \neq 0$)



Resto de la sección sobre Grafos y Recorridos:



Parte II

Algoritmos Voraces

Parte II

Algoritmos Voraces

- Algoritmos Voraces Simples y Técnicas Generales
 - Caminos Mínimos: Algoritmo de Dijkstra
 - Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
 - Particiones
 - Códigos de Huffman

Introducción

Los **algoritmos voraces** (eng: *greedy algorithms*) son algoritmos iterativos que construyen una solución óptima¹ a un problema de optimización, tomando decisiones localmente óptimas entre las opciones disponibles en cada momento. Por lo general, son fáciles de diseñar, implementar y analizar (su coste es típicamente $\Theta(n)$, $\Theta(n \log n)$ o $\Theta(n^2)$); el problema estriba en demostrar que siempre obtienen una solución óptima (o no).

¹No siempre.

```
procedure GREEDY( $X$ )
     $sol := \emptyset$ 
    Inicializar el conjunto de candidatos  $C \equiv C(X)$ 
     $i := 1$ 
    while  $\neg$ ES SOLUCIÓN( $sol, X$ ) do
        Seleccionar el mejor candidato  $c \in C$ 
        Añadir la decisión  $c$  a  $sol$ 
        Actualizar el conjunto  $C$ 
         $i := i + 1$ 
    end while
    return  $sol$ 
end procedure
```

Parte II

Algoritmos Voraces

- Algoritmos Voraces Simples y Técnicas Generales
- Caminos Mínimos: Algoritmo de Dijkstra
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones
- Códigos de Huffman

Algoritmo de Dijkstra

Dado un grafo dirigido $G = \langle V, E \rangle$ etiquetado, el algoritmo de Dijkstra (1959) nos permite hallar los caminos mínimos desde un vértice $s \in V$ dado a todos los restantes vértices del grafo. Si el grafo G contuviera ciclos de peso negativo la noción de camino mínimo no estaría bien definida para todo par de vértices, pero el algoritmo de Dijkstra puede fallar en presencia de arcos con peso negativo, incluso si no hay ciclos de peso negativo. Supondremos por lo tanto que todos los pesos $\omega : E \rightarrow \mathbb{R}^+$ son positivos.

Sea $\mathcal{P}(u, v)$ el conjunto de caminos entre dos vértices u y v de G . Dado un camino $\pi = [u, \dots, v] \in \mathcal{P}(u, v)$ su peso es la suma de los pesos de los n arcos que lo forman:

$$\omega(\pi) = \omega(u, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{n-1}, v).$$

Sea $\Delta(u, v) = \min\{\omega(\pi) \mid \pi \in \mathcal{P}(u, v)\}$ y $\pi^*(u, v)$ un camino de $\mathcal{P}(u, v)$ cuyo peso es mínimo. Si $\mathcal{P}(u, v) = \emptyset$ entonces tomamos $\Delta(u, v) = +\infty$, por convenio. Consideraremos en primer lugar la versión del algoritmo de Dijkstra que calcula los pesos de los caminos mínimos desde un vértice a todos los restantes, y más tarde la versión que calcula adicionalmente los caminos propiamente dichos.

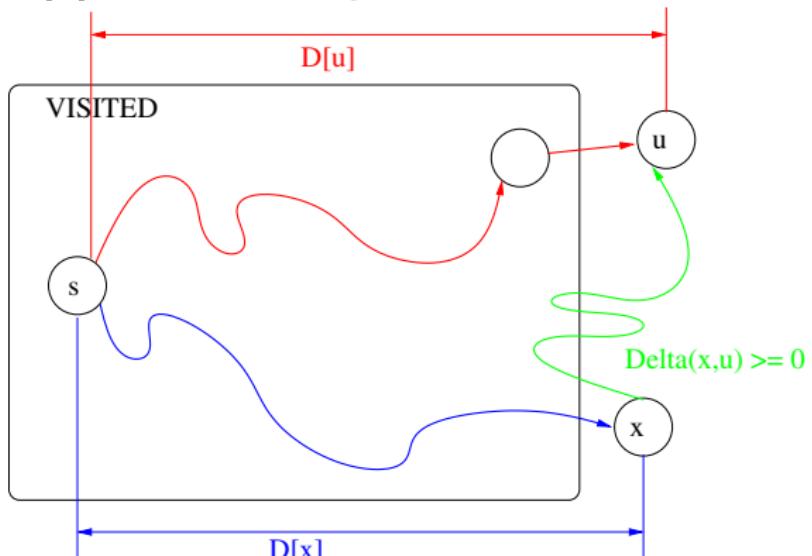
- ▷ $G = \langle V, E \rangle$ es un grafo dirigido con pesos positivos
- ▷ $s \in V$
- DIJKSTRA(G, s, D)
- ▷ Para todo $u \in V$, $D[u] = \Delta(s, u)$

- ▷ $G = \langle V, E \rangle$ es un grafo dirigido con pesos positivos
- ▷ $s \in V$
- DIJKSTRA(G, s, D, cam)
- ▷ Para todo $u \in V$, $D[u] = \Delta(s, u)$
- ▷ Para todo $u \in V$, $D[u] < +\infty \implies cam[u] = \pi^*(s, u)$

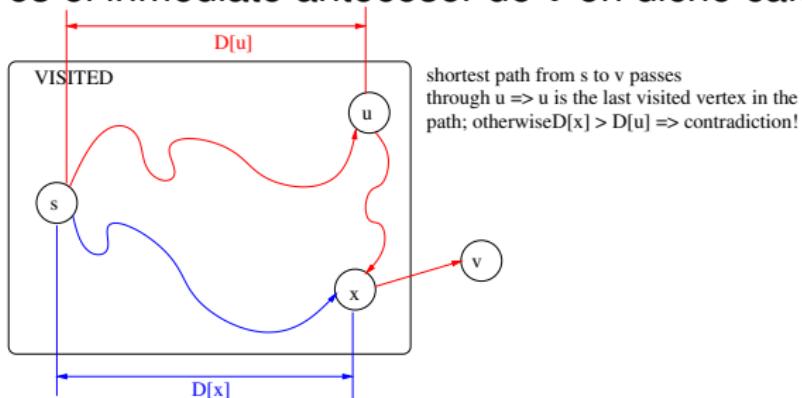
El algoritmo de Dijkstra actúa en una serie de etapas. En todo momento el conjunto de vértices V se divide en dos partes: los vértices *vistos* y los vértices *no vistos* o *candidatos*. Al inicio de cada etapa, si u es un vértice visto entonces $D[u] = \Delta(s, u)$; si u es un candidato entonces $D[u]$ es el peso del camino mínimo entre s y u que pasa **exclusivamente** por vértices intermedios vistos. Este es el invariante del algoritmo de Dijkstra.

En cada etapa un vértice pasa de ser candidato a ser visto. Y cuando todos los vértices son vistos entonces tenemos completamente resuelto el problema.

¿Qué vértice candidato debe seleccionarse en cada etapa para pasar a ser visto? Intuitivamente, aquél cuya D sea mínima de entre todos los candidatos. Sea u dicho vértice. Entonces $D[u]$ no sólo es el peso del camino mínimo entre s y u que sólo pasa por vistos (según el invariante), sino el peso del camino mínimo. En efecto, si el camino mínimo pasase por algún otro vértice no visto x , tendríamos que el peso de dicho camino es $D[x] + \Delta(x, u) < D[u]$, pero como $\Delta(x, u) \geq 0$ y $D[u]$ es mínimo llegamos a una contradicción.



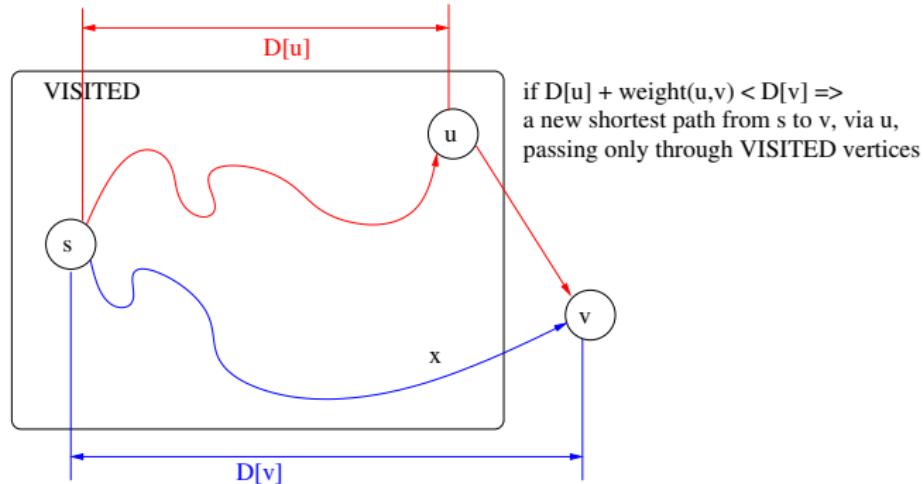
Ahora debemos pensar cómo mantener el resto del invariante. Para los vértices vistos D ya tiene el valor adecuado (incluído u , el vértice que pasa de candidatos a vistos, como acabamos de ver). Pero si v es un candidato, $D[v]$ tal vez haya de cambiar ya que tenemos un vértice adicional visto, el vértice u . Un sencillo razonamiento demuestra que si el camino mínimo entre s y v que pasa por vértices vistos incluye a u entonces u es el inmediato antecesor de v en dicho camino.



De ahí se sigue que el valor de D sólo puede cambiar para los vértices v sucesores de u . Esto ocurrirá si y sólo si

$$D[v] > D[u] + \omega(u, v).$$

Si v fuera sucesor de u pero ya estuviera visto la condición anterior no puede ser cierta.



procedure DIJKSTRA(G, s, D)

▷ D es el resultado; $D[u] = \Delta[s, u]$ para todo $u \in V(G)$

▷ $cand$ es un subconjunto de vértices de $V(G)$

for $v \in V(G)$ **do**

$D[v] := +\infty$

end for

$D[s] := 0$

$cand := V(G)$

while $cand \neq \emptyset$ **do**

$u :=$ el vértice de $cand$ con D mínima

$cand := cand - \{u\}$

for $v \in \text{SUCESORES}(G, u)$ **do**

$d := D[u] + \text{PESO}(G, u, v)$

if $d < D[v]$ **then**

$D[v] := d$

end if

end for

end while

end procedure

| D | | | | | | CANDIDATOS |
|-----|----------|----------|----------|----------|----------|------------------------|
| 1 | 2 | 3 | 4 | 5 | 6 | |
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | $\{1, 2, 3, 4, 5, 6\}$ |
| 0 | 3 | 6 | ∞ | ∞ | ∞ | $\{2, 3, 4, 5, 6\}$ |
| 0 | 3 | 6 | 7 | 4 | ∞ | $\{3, 4, 5, 6\}$ |
| 0 | 3 | 6 | 7 | 4 | 9 | $\{3, 4, 6\}$ |
| 0 | 3 | 6 | 7 | 4 | 8 | $\{4, 6\}$ |
| 0 | 3 | 6 | 7 | 4 | 8 | $\{6\}$ |
| | | | | | | \emptyset |

Para conseguir que el algoritmo compute los caminos mínimos propiamente dichos, empezamos observando que si el camino mínimo entre s y u pasa por x entonces la parte de ese camino que nos lleva de s a x ha de ser necesariamente el camino mínimo entre s y x . Por lo tanto, bastará con computar un *árbol de caminos mínimos* que implícitamente se representa mediante una tabla cam tal que:

$$cam[v] = \begin{cases} s & \text{si } v = s, \\ u & \text{si } (u, v) \text{ es el último arco de } \pi^*(s, v), \\ \perp & \text{si } \Delta(s, v) = +\infty, \end{cases}$$

donde $cam[v] = \perp$ indica que $cam[v]$ no está definido.

Claramente, los únicos cambios que serán necesarios son:

- ① inicializar $cam[s] := s$
- ② incluir la actualización de cam en el bucle interno

```
...
for  $v \in \text{SUCESORES}(G, u)$  do
     $d := D[u] + \text{PESO}(G, u, v)$ 
    if  $d < D[v]$  then
         $D[v] := d$ 
         $cam[v] := u$ 
    end if
end for
...
```

Si queremos el camino completo de s a v podemos deshacer el recorrido a la inversa:

$$cam[u], cam[cam[u]], \dots, cam[\dots [cam[u]] \dots]$$

hasta que lleguemos a s .

Sea n el número de vértices de G y m el número de arcos. Si el grafo G se implementa mediante matriz de adyacencias, el coste del algoritmo de Dijkstra es $\Omega(n^2)$ ya que se hacen n iteraciones del bucle principal y dentro de cada una de ellas se incurriría en un coste como mínimo $\Omega(n)$ para recorrer los sucesores del vértice seleccionado. Descartaremos esta posibilidad, y supondremos que el grafo se implementa mediante listas de adyacencia que nos darán mejores resultados tanto en tiempo como en espacio.

Supongamos, para simplificar, que $V(G) = \{1, \dots, n\}$ y que implementamos el conjunto *cand* y el diccionario D mediante sencillas tablas indexadas de 1 a n ($cand[i] = \text{cierto}$ si y sólo si el vértice i es un candidato). Entonces el coste del algoritmo de Dijkstra es $\Theta(n^2 + m) = \Theta(n^2)$ puesto que se hacen n iteraciones del bucle principal, buscar el mínimo de la tabla D en cada iteración tiene coste $\Theta(n)$ y actualizar la tabla D tiene coste proporcional al número de sucesores del vértice seleccionado (combinando la obtención de los vértices sucesores y la etiqueta de los arcos correspondientes).

Si $V(G)$ es un conjunto arbitrario, podemos conseguir el mismo rendimiento utilizando tablas de hash para implementar $cand$ y D . Y el problema sigue siendo la ineficiencia en la selección del vértice con D mínima.

Para mejorar la eficiencia podemos convertir $cand$ en una cola de prioridad cuyos elementos son vértices y donde la prioridad de cada vértice es su correspondiente valor de D

- ▷ $cand$ es una cola de prioridad de mínimos, cuyos
- ▷ elementos son vértices de G y la prioridad de un
- ▷ elemento u es $D[u]$

...

- ▷ $D[v] = +\infty$ para todo $v \neq s$; $D[s] = 0$

for $v \in V(G)$ **do**

$cand.\text{INSERTA}(v, D[v])$

end for

while $\neg cand.\text{ES_VACIA}()$ **do**

$u := cand.\text{MIN}(); cand.\text{ELIM_MIN}()$

for $v \in \text{SUCESORES}(G, u)$ **do**

...

end for

El problema es que dentro del bucle que recorre los sucesores de u se puede modificar el valor de D (la prioridad) de vértices candidatos. Por ello deberemos dotar a la cola de prioridad de una operación adicional que, dado un elemento, nos permita decrementar su prioridad (los valores de D se modifican siempre a la baja).

```
...
for  $v \in \text{SUCESORES}(G, u)$  do
     $d := D[u] + \text{PESO}(G, u, v)$ 
    if  $d < D[v]$  then
         $D[v] := d$ 
        cand..DECR_PRIO( $v, d$ )
    end if
end for
...
```

Si los costes de las operaciones sobre la cola de prioridad son $\mathcal{O}(\log n)$ entonces el coste del bucle principal es

$$\begin{aligned} D(n) &= \sum_{v \in V(g)} \mathcal{O}(\log n) \cdot (1 + \# \text{ sucesores de } v) \\ &= \Theta(n \log n) + \mathcal{O}(\log n) \sum_{v \in V(g)} \# \text{ sucesores de } v \\ &= \Theta(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}((n + m) \log n) \end{aligned}$$

Por otra parte, el coste de las inicializaciones previas es $\mathcal{O}(n \log n)$.

Por lo tanto el coste que nos queda es $\mathcal{O}((n + m) \log n)$. Puede demostrarse que, de hecho, el coste en caso peor del algoritmo de Dijkstra es $\Theta((m + n) \log n)$. Pero en muchos casos el coste es menor, porque no hay que usar `decr_prio` para todos los sucesores del vértice seleccionado y/o porque el coste de las operaciones sobre la cola de prioridad es frecuentemente menor que $\Theta(\log n)$.

Al crear *can_d* sabemos cuántos vértices tiene el grafo y podemos crear dinámicamente una estructura de datos para ese tamaño. Adicionalmente podemos evitar la redundancia de la tabla *D*, ya que para cada elemento de *can_d* tenemos su prioridad, que es su valor de *D*. Necesitamos por lo tanto una clase con la funcionalidad combinada típica de las colas de prioridad y de los diccionarios:

```
template <class Elem, class Prio>
class ColaPrioDijkstra<ELEM, Prio> {
public:
    ColaPrioDijkstra(int n = 0);
    void inserta(const Elem& e, const Prio& prio);
    Elem min() const;
    void elim_min();
    Prio prio(const Elem& e) const;
    bool contiene(const Elem& e) const;
    void decr_prio(const Elem& e, const Prio& nuevaprio);
    bool es_vacia() const;
    ...
};
```

```
typedef vector<list<pair<int,double>> > grafo;
typedef list<pair<int,double>>::const_iterator arco;
...
void Dijkstra(const grafo& G, int s, ... ) {
    ColaPrioDijkstra<int,double> cand(G.size());
    for (int v = 0; v < G.size(); ++v)
        cand.inserta(v, INFINITY);
    cand.decr_prio(s, 0);
    while(not cand.es_vacia()) {
        int u = cand.min(); double du = cand.prio(u);
        cand.elim_min();
        for (arco e = G[u].begin(); e != G[u].end(); ++e) {
            // e apunta al par <v, peso(u,v)>
            // e -> first() == v
            // e -> second() == peso(u,v)
            int v = e -> first();
            if (cand.contiene(v)) {
                double d = du + e -> second();
                if (d < cand.prio(v))
                    cand.decr_prio(v, d);
            }
        }
    }
}
```

Pueden conseguirse costes logarítmicos o inferiores en todas las operaciones sobre la cola de prioridad utilizando un *heap* implementado en vector.

Además necesitaremos una tabla de hash que nos permita “traducir” vértices a índices. Y una tabla de índices y otra de posiciones en el *heap*.

```
template <class Elem, class Prio>
class ColaPrioDijkstra<ELEM,PRIOR> {
    ...
private:
    vector<PRIOR> prio;
    vector<ELEM> elems;
    int nelems; // == elems.size()
    vector<int> index;
    vector<int> pos;
    hash_map<ELEM,int> identif;
```

Los vectores *prio* y *elems*, junto al contador *nelems* representan al *heap*. Si $\text{elems}[i] = e_j$ entonces $\text{index}[i] = j$ y $\text{pos}[j] = i$, es decir, *pos* nos da la posición en el *heap* del elemento e_j e *index* el identificador del elemento en la posición i del *heap*, siendo $\text{identif}[e_j] == j$.

La constructora de la clase crea las tablas *prio*, *e*, *index* y *pos* con n componentes y el map.

Cada vez que se inserta un nuevo elemento se le asigna el índice *nelems* + 1, se inserta en *identif*, se coloca en el heap en la posición *nelems* + 1, y se le hace flotar. La operación flotar se encarga de mantener actualizadas las tablas *pos* e *index*. Para *elim_min*, se intercambia el último nodo del *heap* con la raíz

```
// intercambiamos el ultimo elemento del heap con la raíz
swap(prio[1], prio[nelems]);
swap(elems[1], elems[nelems]);
// actualizamos pos e index
swap(pos[index[1]], pos[index[nelems]]);
swap(index[1], index[nelems]);
```

y a continuación se hunde la raíz, cuidando de mantener también actualizadas las tablas *pos* e *index*.

La operación `decr_prio` requiere averiguar el identificador j del elemento e_j cuya prioridad se va a decrementar usando la tabla de hash, usar la tabla $pos[j]$ para obtener la posición i del nodo que le corresponde en el heap, y una vez modificada la prioridad, flotar el nodo.

Es fácil comprobar que las operaciones de la clase, exceptuando la constructora, tienen coste $\Theta(1)$ (`es_vacia`, `min`, `prio`, `contiene`) o $\mathcal{O}(\log n)$ (`inserta`, `elim_min`, `decr_prio`).

Parte II

Algoritmos Voraces

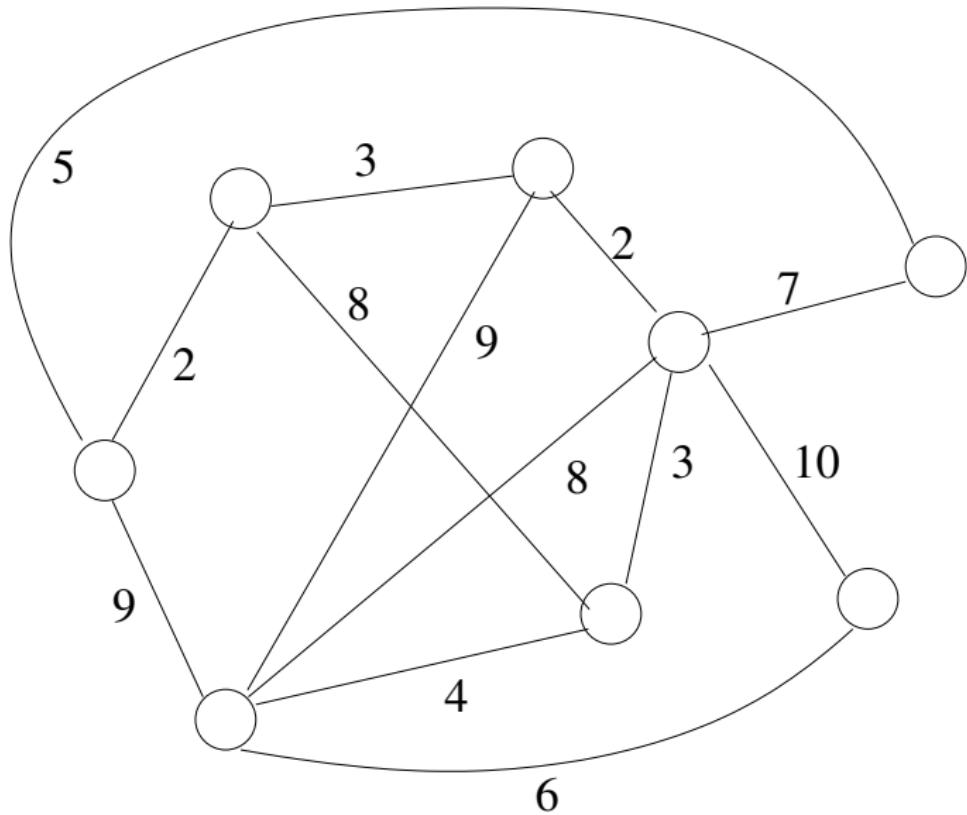
- Algoritmos Voraces Simples y Técnicas Generales
- Caminos Mínimos: Algoritmo de Dijkstra
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones
- Códigos de Huffman

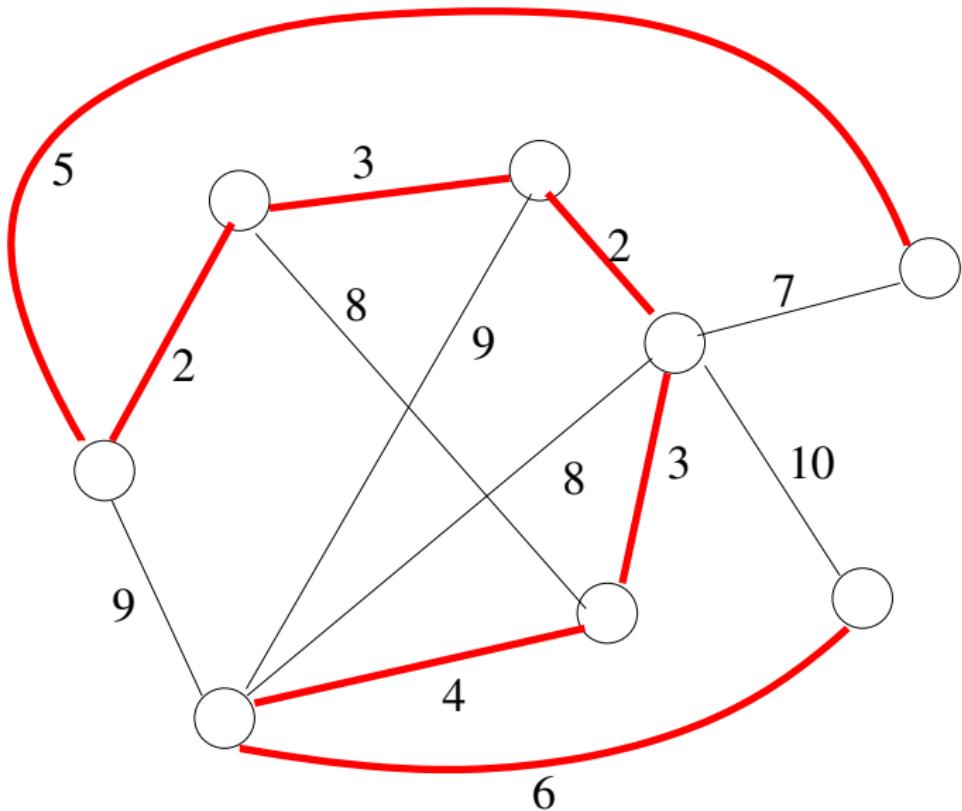
Árboles de Expansión Mínimos

Dado un grafo no dirigido y conexo $G = \langle V, E \rangle$ con pesos o costes en las aristas $\omega : E \rightarrow \mathbb{R}$, un **árbol de expansión mínimo** (*MST: minimum spanning tree*) $T = \langle V, A \rangle$ es un subgrafo de G tal que tiene el mismo conjunto de vértices ($V(T) = V(G)$), es un árbol (es decir, es conexo y acíclico) y su coste

$$\omega(T) = \sum_{e \in A} \omega(e)$$

es mínimo entre todos los posibles árboles de expansión de G .





Existen multitud de algoritmos para calcular un MST de un grafo. No obstante todos ellos siguen un esquema voraz:

```
A :=  $\emptyset$ ; Candidatas :=  $E$ 
while  $|A| \neq |V(G)| - 1$  do
    Seleccionar una arista  $e \in$  Candidatas que
        no crea un ciclo en  $T$ 
     $A := A \cup \{e\}$ 
    Candidatas := Candidatas –  $\{e\}$ 
end while
```

Diremos que un conjunto de aristas $A \subset E(G)$ es **prometedor** si y sólo si:

- 1 A no contiene ciclos
- 2 A es un subconjunto de las aristas de un MST del grafo G

Un **corte** del grafo G es una partición de su conjunto de vértices en dos subconjuntos C y C' no vacíos y disjuntos:

$$V(G) = C \cup C'; \quad C \cap C' = \emptyset$$

Una arista e **respeta** un corte $\langle C, C' \rangle$ si sus dos extremos están ambos en C o ambos en C' , en caso contrario se dice que e **cruza** el corte.

Teorema

Sea A un conjunto prometedor de aristas que respetan un cierto corte $\langle C, C' \rangle$ del grafo G . Sea $e \in E(G)$ la arista de mínimo peso que cruza el corte $\langle C, C' \rangle$. Entonces

$$A \cup \{e\}$$

es prometedor

El teorema anterior nos da la “receta” para diseñar algoritmos de cálculo de un MST: una vez que hayamos definido cuál es el corte que corresponde a cada iteración, la selección de la arista consistirá en localizar la arista e de mínimo peso que cruza el corte. Por definición, como A respeta el corte y e lo cruza, e no puede crear un ciclo en A .

Más aún: la corrección de los algoritmos que se fundamentan en esta idea queda automáticamente establecida.

Demostración:

Sea A' el conjunto de aristas de un MST T' tal que $A \subset A'$.

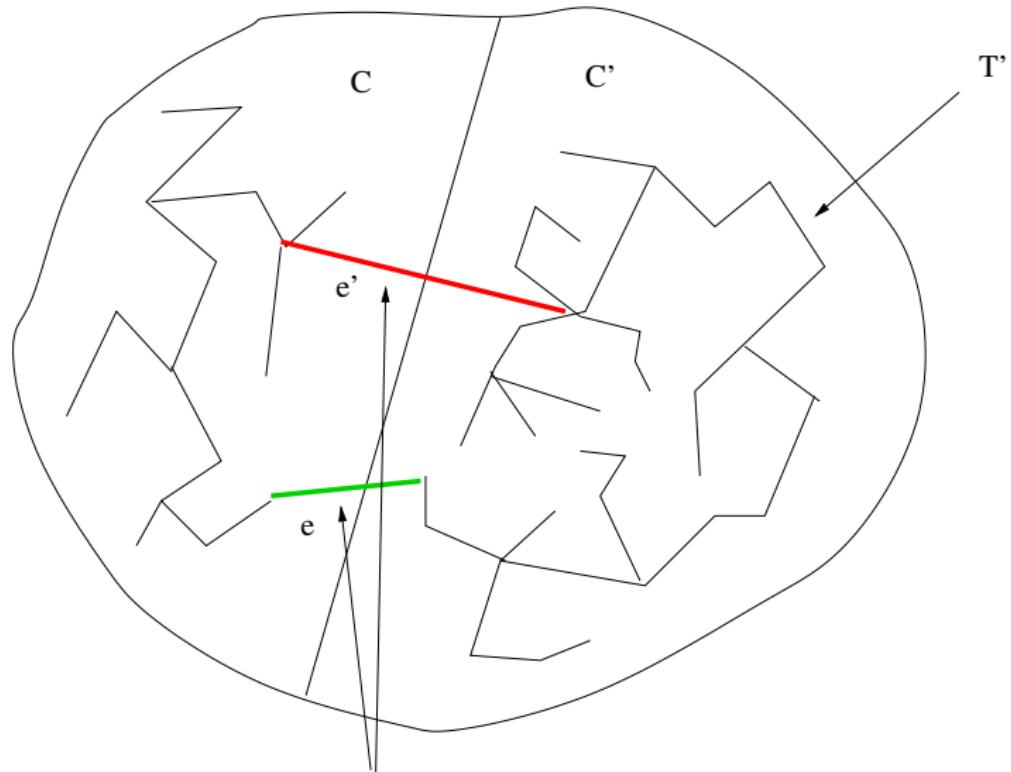
Puesto que A respeta el corte, al menos una arista de las que cruzan el corte tendría que pertenecer a A' , en caso contrario A' no sería conexo. Supongamos que una de dichas aristas es e' . Si e' es la arista de mínimo peso que cruza el corte, como $A \cup \{e'\}$ es prometedor, hemos demostrado el teorema. ¿Pero qué ocurre si e' no es la arista de mínimo peso que cruza el corte?

Demostración (cont.):

El coste del MST T' incluirá el coste de las aristas de A , el coste $\omega(e')$ y el coste de otras aristas. Si agregásemos a T' la arista e de mínimo peso que cruza el corte, crearíamos un ciclo, porque T' es un árbol. Sea e' la arista de T' que cruza el corte forma parte de dicho ciclo. De manera que

$$T = T' \cup \{e\} - \{e'\}$$

también es un árbol de expansión. Y su coste es menor o igual que el de T' puesto que sólo cambiamos el coste de e' por el de e , que es el mínimo. Como T' es MST, llegamos a una contradicción a menos que e y e' tengan el mismo coste, y por tanto T y T' tendrían el mismo coste. El teorema queda demostrado, ya que $A \cup \{e\}$ es un subconjunto de las aristas de T , que es un MST.



we can replace e' in T' by e to obtain a new spanning tree with smaller cost!!

Algoritmo de Prim

En el algoritmo de Prim, se mantiene un conjunto de vértices $Vistos \subset V(G)$, y en cada etapa del algoritmo se selecciona la arista de mínimo peso que une a un vértice u de $Vistos$ con un vértice v no visto (i.e., en $V(G) - Vistos$).

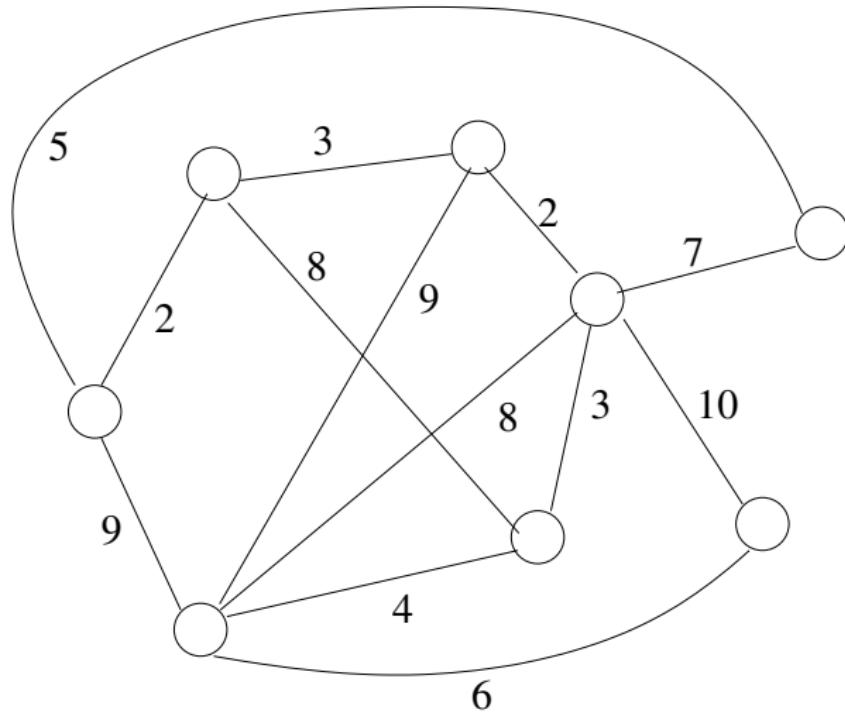
Dicha arista no puede crear un ciclo y se añade al conjunto A . Asimismo el vértice v pasa a ser de $Vistos$.

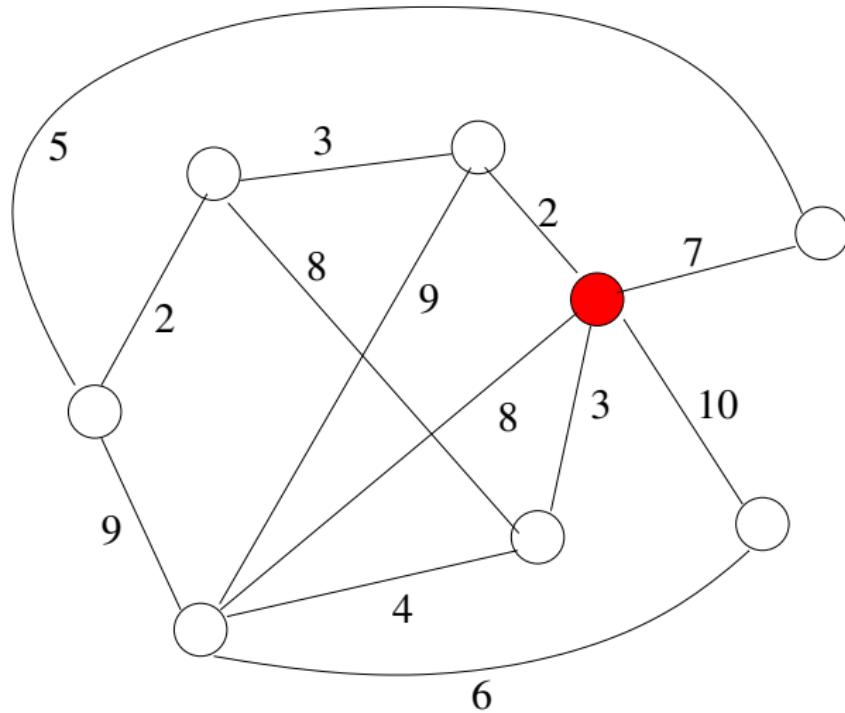
La corrección del algoritmo es inmediata; el conjunto A respeta el corte $(Vistos, V(G) - Vistos)$ y seleccionamos la arista de mínimo peso que cruza el corte para agregarla a A .

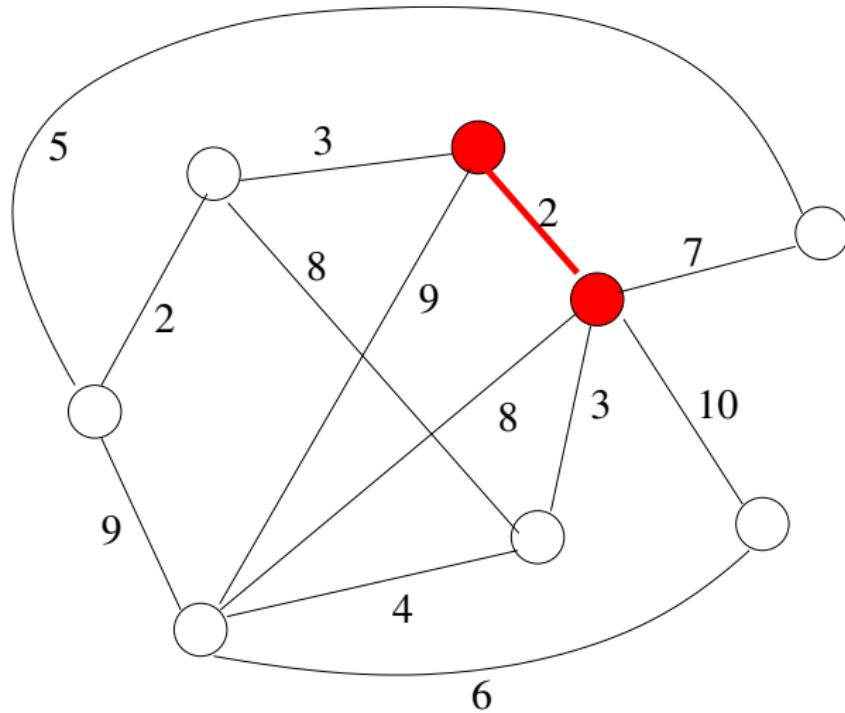
```

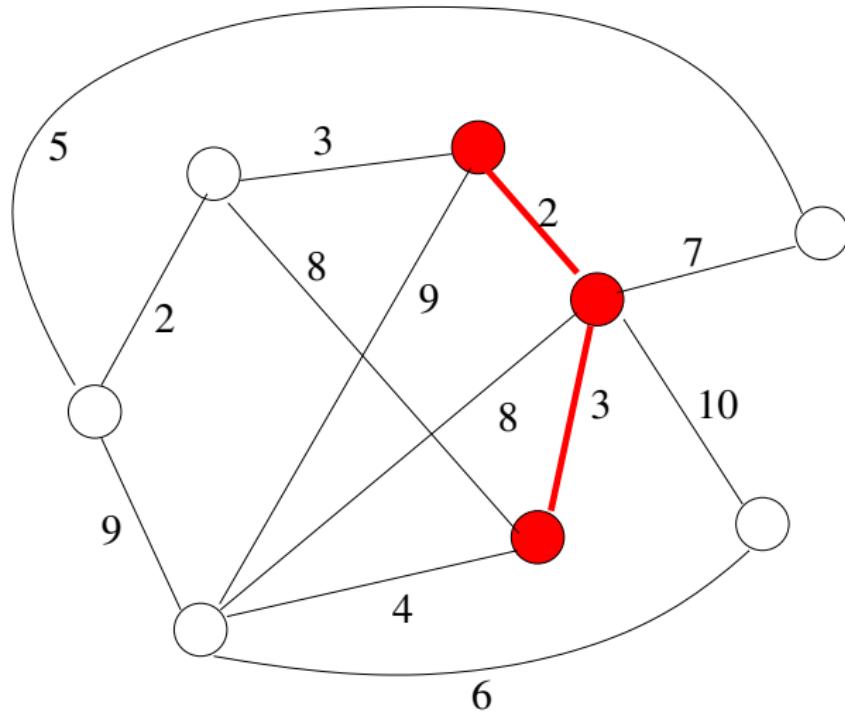
procedure PRIM( $G$ )
     $A := \emptyset$ ;  $Vistos := \{s\}$ 
     $Candidatas := \emptyset$ 
    for  $v \in G.\text{ADYACENTES}(s)$  do
         $Candidatas := Candidatas \cup \{(s, v)\}$ 
    end for
    while  $|A| \neq |V(G)| - 1$  do
        Seleccionar la arista  $e = (u, v)$ 
        de  $Candidatas$  de mínimo peso
         $A := A \cup \{e\}$ 
        Sea  $u$  el vértice en  $Vistos$ :
         $Vistos := Vistos \cup \{v\}$ 
        for  $w \in G.\text{ADYACENTES}(v)$  do
            if  $w \notin Vistos$  then
                 $Candidatas := Candidatas \cup \{(v, w)\}$ 
            end if
        end for
    end while
    return  $A$ 
end procedure

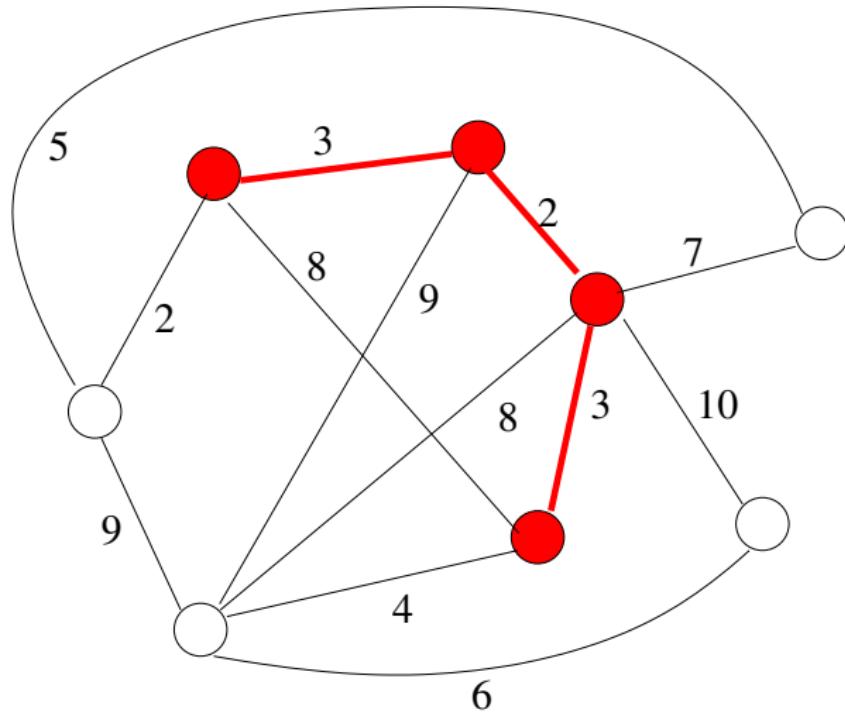
```

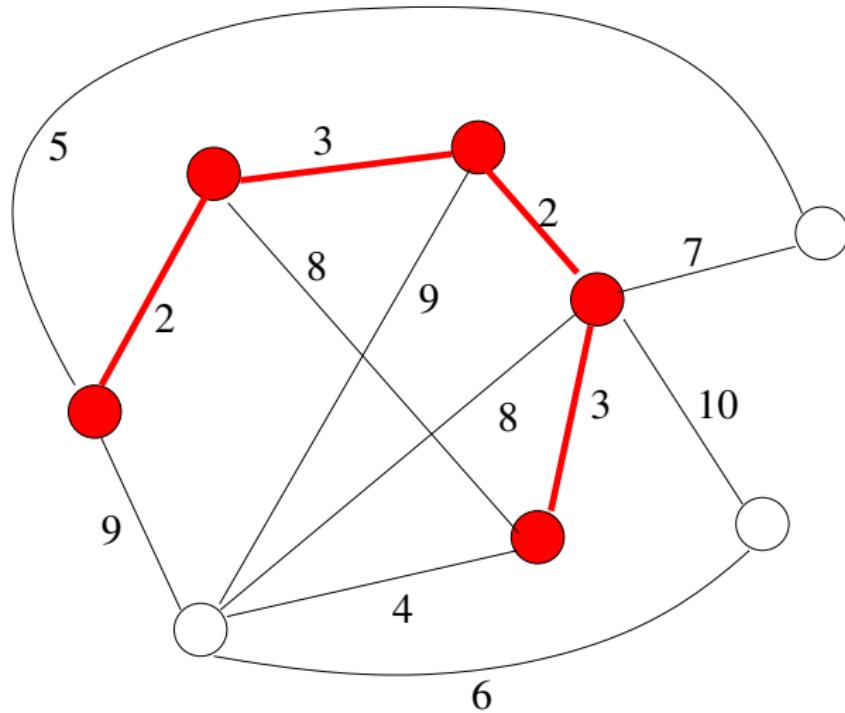


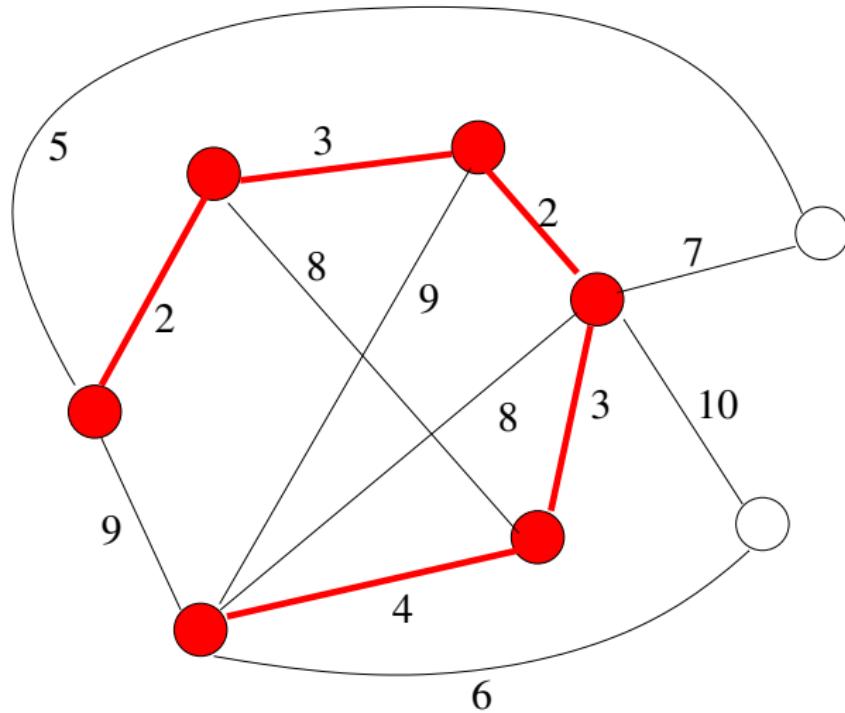












El coste del algoritmo de Prim dependerá de cómo implementemos la selección de la arista candidata. El bucle principal hace $n - 1$ iteraciones, pues en cada iteración añadimos una arista al árbol A , o equivalentemente, en cada iteración *vemos* un vértice nuevo y terminaremos cuando todos estén *vistos*.

En el interior del bucle principal se hacen dos tareas “costosas”: seleccionar la arista de mínimo peso en *Candidatas* y añadir ciertas aristas adyacentes al vértice recién visto al conjunto de *Candidatas*.

El coste de la segunda tarea es proporcional al grado del vértice v y en total aportará al coste

$$\sum_{v \in V(G)} \Theta(\text{grado}(v)) = \Theta\left(\sum_{v \in V(G)} \text{grado}(v)\right) = \Theta(m)$$

Pero si usamos una implementación ingenua para el conjunto de *Candidatas*, el coste de seleccionar una arista es $\mathcal{O}(m)$ y el coste total del algoritmo de Prim es $\mathcal{O}(n \cdot m)$.

Para conseguir mejorar sensiblemente el coste del algoritmo el conjunto de *Candidatas* debe implementarse como un min-heap, siendo la prioridad de cada arista su coste.

Entonces la selección (y eliminación) de la arista de coste mínimo en cada iteración pasa a ser $\mathcal{O}(\log m)$.

Por otro lado, el coste de ir añadiendo aristas a *Candidatas* lo tenemos que recalcular:

$$\begin{aligned}\sum_{v \in V(G)} \Theta(\text{grado}(v)(1 + \log m)) &= \Theta\left(\sum_{v \in V(G)} \text{grado}(v)(1 + \log m)\right) \\ &= \Theta(m \log m)\end{aligned}$$

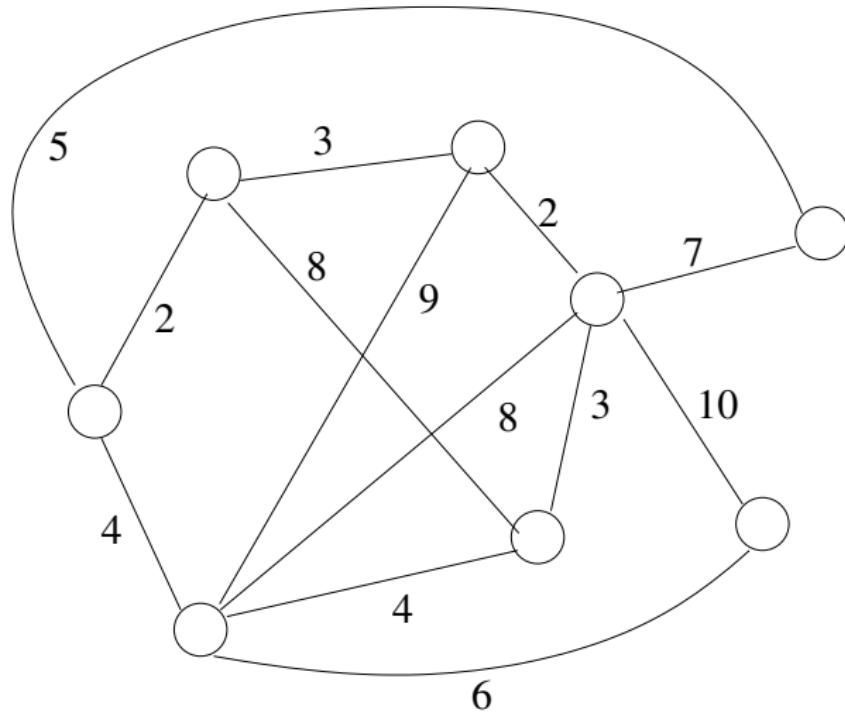
En total: $\Theta((n + m) \log m)$

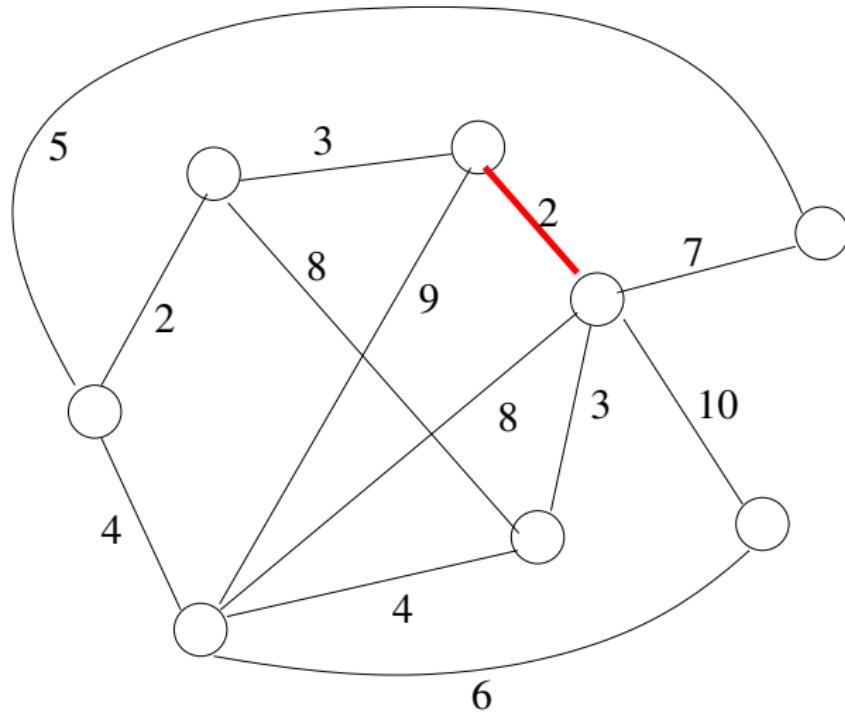
Algoritmo de Kruskal

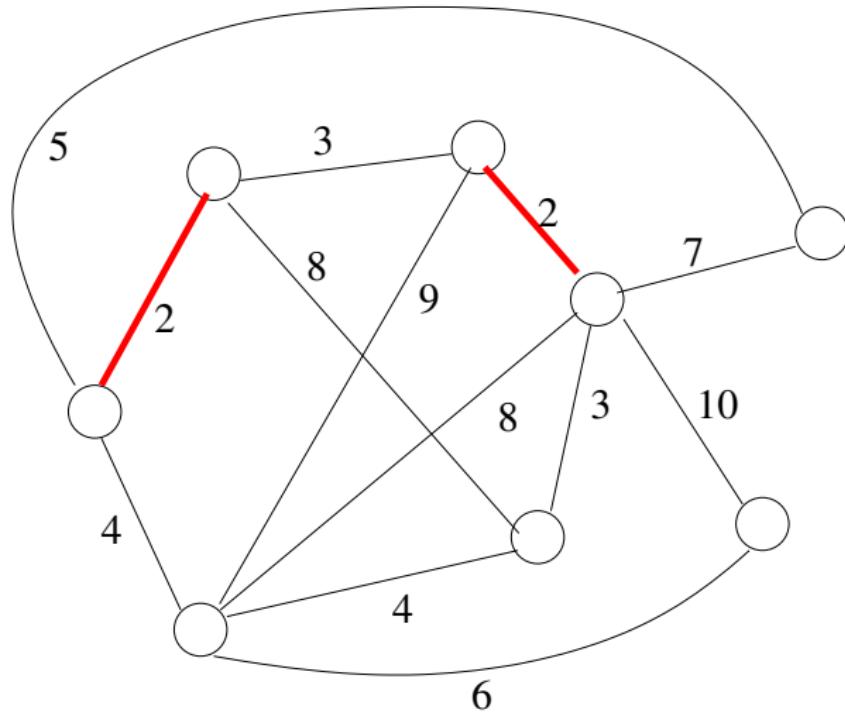
En el algoritmo de Kruskal se mantiene en todo momento un conjunto A de aristas que es un bosque (un conjunto de árboles). En cada paso se toma una arista e de mínimo peso entre todas las posibles; si dicha arista cierra un ciclo se descarta, si no cierra un ciclo, e se agrega a A .

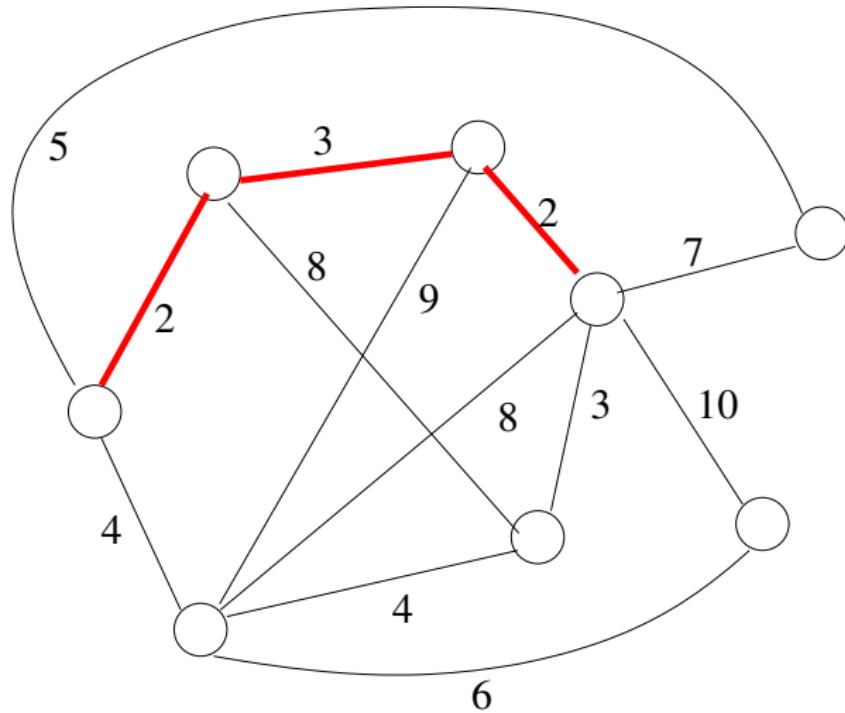
Cuando la arista $e = (u, v)$ escogida no cierra un ciclo, se define como corte el que forman los vértices en la misma componente conexa de u (mediante aristas de A) por un lado, y los restantes vértices del grafo por otro. Claramente A respeta el corte y e es la arista de mínimo peso que respeta ese corte.

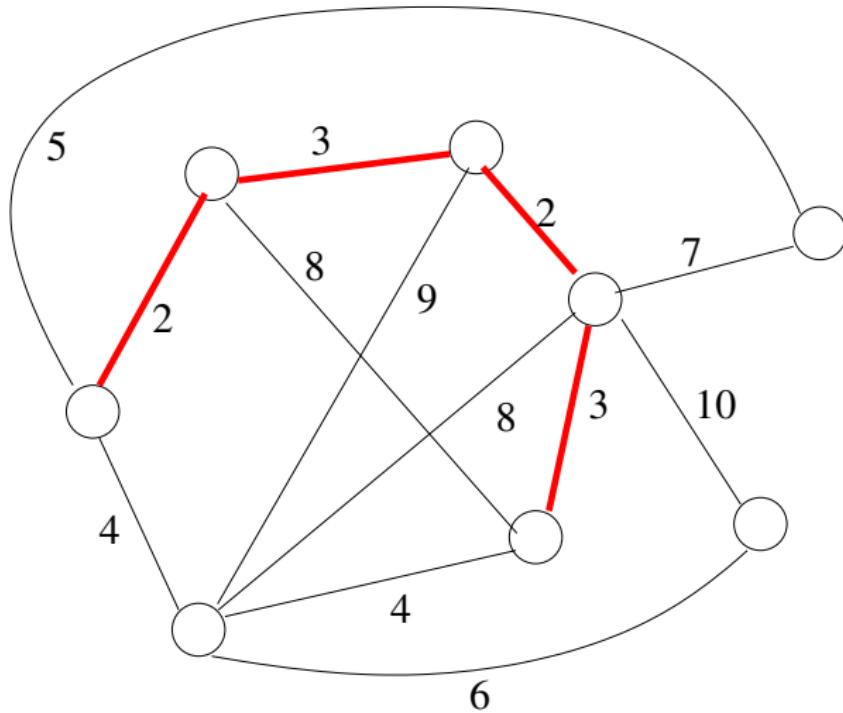
```
procedure KRUSKAL( $G = \langle V, E \rangle$ )
    Ordenar  $E$  por peso creciente
     $A := \emptyset$ 
    while  $|A| \neq |V(G)| - 1$  do
         $e = (u, v) := \text{SIGUIENTE}(E)$ 
        if  $e$  no cierra un ciclo en  $A$  then
             $A := A \cup \{e\}$ 
        end if
    end while
    return  $A$ 
end procedure
```

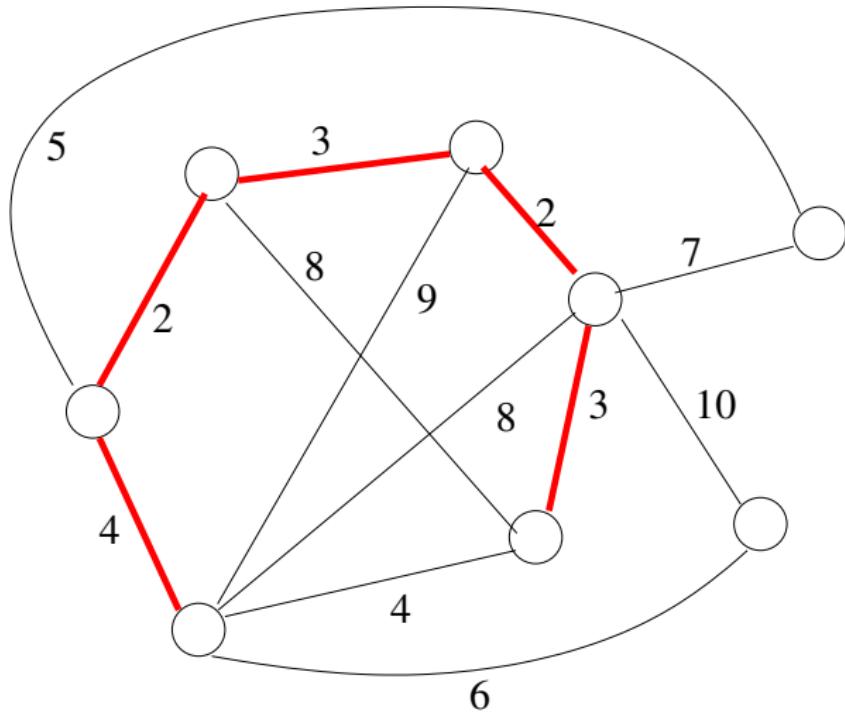


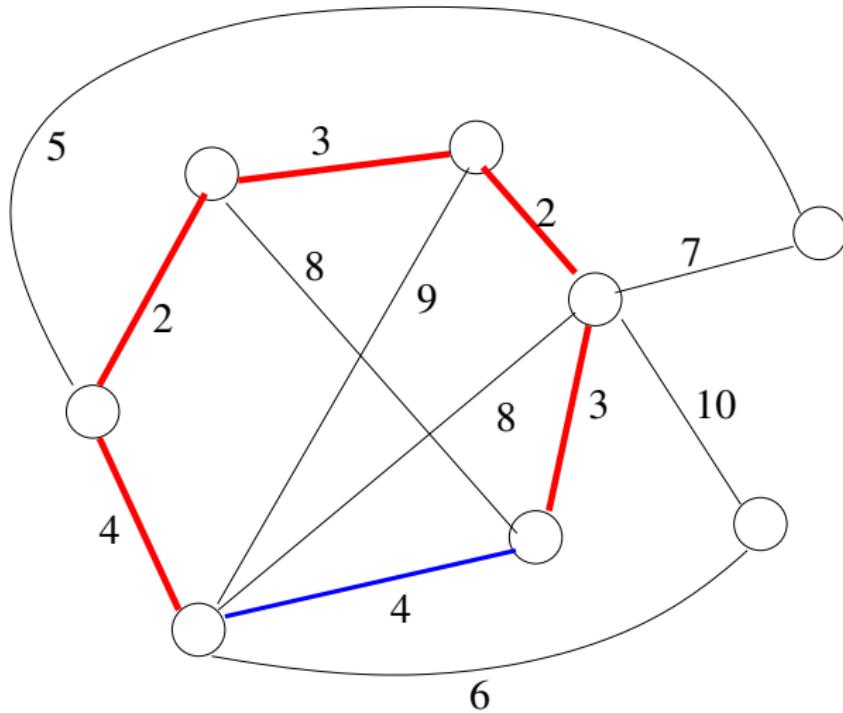


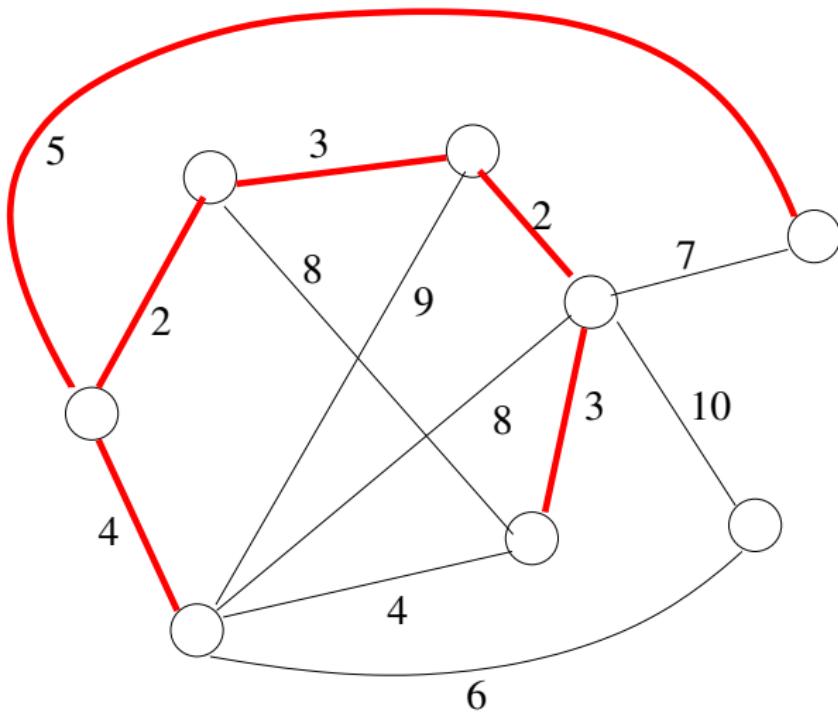


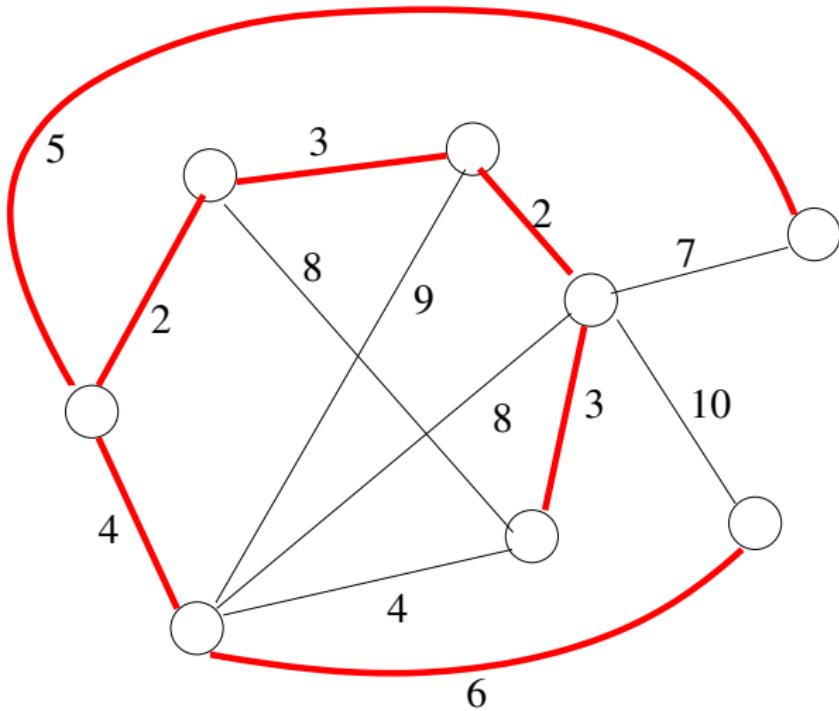












Para determinar si una cierta arista e cierra un ciclo en A de manera eficiente usaremos una estructura de datos “astuta” denominada **PARTICION** (también se les conoce como MFSETS y UNION-FIND).

Inicialmente creamos una **PARTICION** en la que cada vértice del grafo constituye un **bloque** por sí solo. Cada vez que agreguemos una arista $e = (u, v)$ al conjunto A , los bloques a los que pertenecen u y v se fusionan en un solo bloque (*merge, union*).

Cada bloque de vértices es pues una componente conexa de A ; si existe un camino entre los vértices w y w' usando las aristas de A , w y w' pertenecerán a un mismo bloque de la **PARTICIÓN**.

El test que determina si una nueva arista cierra un ciclo entonces consiste simplemente en ver si los vértices u y v pertenecen a un mismo bloque de la `PARTICIÓN` o no. Si ya están conectados, la nueva arista cerraría un ciclo; en caso contrario, la nueva arista no cierra un ciclo y los vértices pasan a estar conectados.

Habitualmente la clase `PARTICIÓN` proporciona una operación `FIND` que dado un elemento u nos devuelve el **representante** del bloque en el que está u . Dos elementos están en el mismo bloque si y sólo si los representantes de sus bloques son idénticos.

procedure KRUSKAL($G = \langle V, E \rangle$)

 Ordenar E por peso creciente

 ▷ Se crea una partición inicial, cada vértice en un bloque distinto

$P.\text{MAKE_UNIONFIND}(V)$

$A := \emptyset$

while $|A| \neq |V(G)| - 1$ **do**

$e = (u, v) := \text{SIGUIENTE}(E)$

if $P.\text{FIND}(u) \neq P.\text{FIND}(v)$ **then**

$A := A \cup \{e\}$

$P.\text{UNION}(u, v)$

end if

end while

return A

end procedure

Para calcular el coste del algoritmo de Kruskal tenemos por un lado el coste de la ordenación de las aristas

$\Theta(m \log m) = \Theta(m \log n)$, siendo $m = |E(G)|$. La creación de la partición inicial tiene coste $\Theta(n)$, siendo $n = |V(G)|$. Por otro lado, el bucle siguiente hace al menos $n - 1$ iteraciones (tantas como aristas tiene el MST hallado) pero puede llegar a hacer m iteraciones en caso peor. El coste del cuerpo del bucle vendrá dominado por el coste de las llamadas a las operaciones FIND y UNION. Si el bucle principal realiza N iteraciones, $n - 1 \leq N \leq m$, se harán $2N$ llamadas a FIND. Por otro lado, se hacen siempre exactamente $n - 1$ llamadas a UNION.

Es muy fácil obtener implementaciones de la PARTICIÓN que garanticen que el coste de FIND y UNION es $\mathcal{O}(\log n)$, lo que nos llevaría a un coste

$$\Theta(m \log n) + \mathcal{O}(m \log n) = \Theta(m \log n)$$

para el algoritmo.

Se puede mejorar el rendimiento notablemente (aunque no en caso peor) de la siguiente forma:

- ① En vez de ordenar las aristas por peso, se crea un min-heap con coste $\Theta(m)$
- ② Se usa una versión de UNION-FIND que garantiza que $\mathcal{O}(m)$ FINDs + $\mathcal{O}(n)$ UNIONS tienen coste $\mathcal{O}((m+n)\alpha(m,n))$. La función $\alpha(m,n)$, llamada función inversa de Ackermann crece de manera extremadamente lenta, a los efectos prácticos, por muy grandes que sean m y n , $\alpha(m,n) \leq 4$.
- ③ El bucle principal seguirá teniendo coste $\mathcal{O}(m \log n)$ pero en la mayoría de casos será más cercano a $\mathcal{O}(n \log n)$ porque no se suelen hacer muchas más del mínimo de $n - 1$ iteraciones.

El coste del algoritmo vendrá dominado por el coste de extraer aristas durante las iteraciones del bucle principal y en caso peor es $\Theta(m \log n)$; en promedio será cercano a $\Theta(n \log n)$, lo cual es muy ventajoso, sobre todo si $m \gg n$.

Parte II

Algoritmos Voraces

- Algoritmos Voraces Simples y Técnicas Generales
- Caminos Mínimos: Algoritmo de Dijkstra
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- **Particiones**
- Códigos de Huffman

Particiones

Una *partición* Π de un conjunto no vacío \mathcal{A} es una colección de subconjuntos no vacíos $\Pi = \{\mathcal{A}_1, \dots, \mathcal{A}_k\}$ tal que

- ① Si $i \neq j$ entonces $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$.
- ② $\mathcal{A} = \bigcup_{1 \leq i \leq k} \mathcal{A}_i$.

Se suele denominar *bloque* de la partición Π a cada uno de los \mathcal{A}_i 's. Las particiones y las relaciones de equivalencia están estrechamente ligadas. Recordemos que \equiv es una relación de equivalencia en \mathcal{A} si y sólo si

- ① \equiv es reflexiva: para todo $a \in \mathcal{A}$, $a \equiv a$.
- ② \equiv es transitiva: si $a \equiv b$ y $b \equiv c$, entonces $a \equiv c$, para cualesquiera a, b y c en \mathcal{A} .
- ③ \equiv es simétrica: $a \equiv b$ si y sólo si $b \equiv a$, para cualesquiera a y b en \mathcal{A} .

Dada una partición Π de \mathcal{A} , ésta induce una relación de equivalencia \equiv_{Π} definida por

$$x \equiv_{\Pi} y \iff x \text{ e } y \text{ pertenecen a un mismo bloque } A_i \in \Pi$$

Y a la inversa, dada una relación de equivalencia \equiv en \mathcal{A} , ésta induce una partición $\Pi = \{A_x\}_{x \in \mathcal{A}}$, donde

$$A_x = \{y \in \mathcal{A} \mid y \equiv x\}.$$

Al subconjunto de elementos equivalentes a x se le denomina *clase de equivalencia* de x . Cada uno de los bloques de la partición inducida por una relación \equiv es por lo tanto una clase de equivalencia. Nótese que si $x \equiv y$ entonces $A_x = A_y$. Un elemento cualquiera de la clase A_x se denomina *representante* de la clase.

En muchos algoritmos, especialmente en algoritmos sobre grafos, es importante poder representar particiones de un conjunto finito de manera eficiente.

Vamos a suponer que el conjunto *soporte* sobre el que se define la partición es $\{1, \dots, n\}$; sin excesiva dificultad, empleando alguna de las técnicas vistas en temas precedentes podemos representar de manera eficiente una biyección entre un conjunto finito A de cardinalidad n y el conjunto $\{1, \dots, n\}$ y con ello una partición sobre el conjunto soporte A en caso necesario.

Adicionalmente, supondremos que el conjunto soporte es estático, es decir, ni se añaden ni se eliminan elementos. No obstante, con poca dificultad extra podemos obtener representaciones eficientes para particiones de conjuntos dinámicos.

Generalmente el tipo de operaciones que una clase Particion debe soportar son las siguientes: 1) dados dos elementos i y j , determinar si pertenecen al mismo bloque o no; 2) dados dos elementos i y j fusionar los bloques a los que pertenecen (si procede) en un sólo bloque, devolviendo la partición resultante.

Frecuentemente el primer tipo de operación se realiza mediante una operación FIND que dado un elemento i , devuelve un representante de la clase a la que pertenece i . Si dos elementos i y j tienen el mismo representante, entonces han de estar en el mismo bloque.

El segundo tipo de operación se llama MERGE o UNION. De ahí que las particiones se les llame a menudo estructuras **union-find** o **mfsets** (abreviación de *merge-find sets*).

La operación MAKE crea una partición de $\{1, \dots, n\}$ consistente en n bloques cada uno de los cuales contiene un elemento.

En muchas aplicaciones antes de hacer cualquier UNION se habrá determinado previamente si los elementos i y j cuyos bloques habrían de unirse, están o no en el mismo bloque; para ello se habrá preguntado si $P.\text{FIND}(i) = P.\text{FIND}(j)$ o no. Por esta razón es habitual que en una clase Particion la operación UNION sólo actúe sobre elementos que son representantes de sus respectivas clases.

```
procedure  $F(\dots)$ 
    ...
     $ri := P.\text{FIND}(i)$ 
     $rz := P.\text{FIND}(j)$ 
    if  $ri \neq rz$  then
         $P.\text{UNION}(ri, rz)$ 
        ...
    end if
end procedure
```

Puesto que la operación MAKE recibe como parámetro el número de elementos n del conjunto soporte, podremos utilizar implementaciones en vector, reclamando un vector con el número apropiado de componentes a la memoria dinámica si nuestro lenguaje de programación lo soporta. También será posible utilizar este tipo de implementación si el valor de n está acotado y dicha cota no es irrazonablemente grande. En cualquier caso podremos modificar sin demasiado esfuerzo las implementaciones que se verán a continuación para que sean completamente dinámicas y funcionen correctamente en aquellos casos en que no se puedan crear vectores cuyo tamaño se fija en tiempo de ejecución o si el conjunto soporte ha de soportar inserciones y borrados.

Quick-find consiste representar la partición mediante un vector P y almacenar en la componente i de P el representante de la clase a la que pertenece i . De este modo la operación FIND tiene coste constante, ya que basta examinar $P[i]$. Sin embargo, la operación UNION tendrá coste $\Theta(n)$ ya que cada uno de los elementos de la clase en la que está j (los k 's tales que $P[k] = P[j]$) han de cambiar de representante ($P[k] := P[i]$). O bien los elementos del mismo bloque que i han de pasar al bloque de j .

Con algunas modificaciones puede evitarse el recorrido completo del vector P y restringirlo a los elementos del bloque de j (o del bloque de i); pero aún así el coste de una UNION sigue siendo lineal en n en el caso peor, ya que cualquiera de los dos bloques puede contener una fracción considerable del total de los elementos.

Aunque resulta un tanto forzado, conviene contemplar la representación *quick-find* como un bosque de árboles; cada árbol representa a un bloque de la partición en un momento dado. Los árboles están representados mediante apuntadores al padre, siendo la raíz de cada árbol el representante del bloque correspondiente. Puesto que la raíz de un árbol no tiene padre, las raíces se apuntan a sí mismas. Del invariante de la representación de *quick-find* se sigue que todos los árboles tienen altura 1 (si sólo contienen un elemento) o altura 2 (todos los elementos de un bloque excepto el representante están en el segundo nivel, apuntando a la raíz).

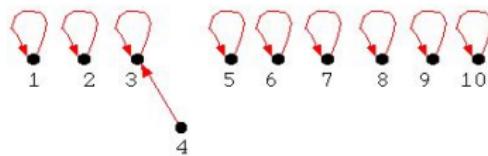
make(10)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |



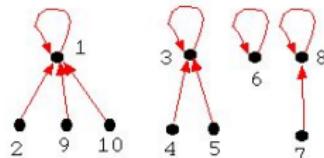
union(3, 4)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 | 10 |



union(1,2); union(4,5); union(1,9);
union(2, 10); union(8,7)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 3 | 3 | 6 | 8 | 8 | 1 | 1 |
| 1 | 1 | 3 | 3 | 3 | 6 | 8 | 8 | 1 | 1 |

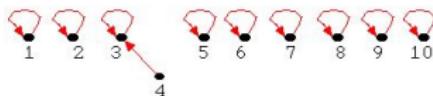
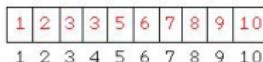


Otra estrategia, *quick-union*, explota la correspondencia entre bloques y árboles del siguiente modo: para unir los bloques de i y j se localizan al representante de i , digamos u , y se coloca a u como hijo de j . Alternativamente, podemos localizar ambos representantes y hacer que uno de ellos sea hijo del otro.

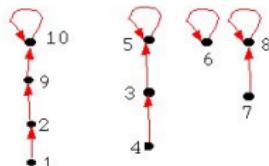
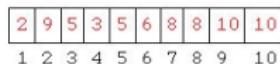
```
procedure MAKE( $n$ )
  for  $i := 1$  to  $n$  do
     $P[i] := i$ 
  end for
end procedure
procedure UNION( $i, j$ )
   $u := \text{FIND}(i)$ 
   $v := \text{FIND}(j)$ 
   $P[u] := v$ 
end procedure
▷  $1 \leq i \leq n$ 
procedure FIND( $i$ )
  while  $P[i] \neq i$  do
     $i := P[i]$ 
  end while
  return  $i$ 
end procedure
```

Aunque en general los árboles resultantes de una secuencia de uniones serán relativamente equilibrados y el coste de las operaciones será bajo, en caso peor podemos crear árboles poco equilibrados de modo que tanto una UNION como un FIND tengan coste proporcional al número de elementos involucrados. Por ejemplo, si realizamos una secuencia de UNIONES de tal modo que la clase en la que está j sólo contenga a j , obtendremos árboles equivalentes a listas.

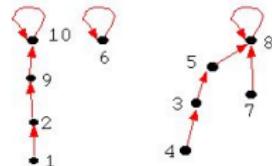
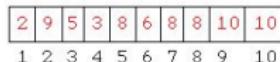
```
make(10); union(4, 3)
```



```
union(1,2); union(4,5); union(1,9);  
union(2, 10); union(7,8)
```



```
union(4,7);
```



Los comentarios previos sugieren posibles soluciones al problema. En la *unión por peso* el árbol con menos elementos es que el que se añade como hijo del que tiene más elementos. En la *unión por rango* el árbol de menor altura es el que se pone como hijo del de mayor altura. Tanto una como otra estrategia son fáciles de implementar, pero requieren, en principio, que se almacene información auxiliar sobre el tamaño o la altura de los árboles.

Se puede evitar el uso de espacio auxiliar observando que sólo se necesita esta información de tamaño o altura para las raíces y que el espacio correspondiente a sus apuntadores es esencialmente inútil, ya que sólo se precisaría un bit que indique que i es una raíz o no. Por ejemplo, podemos adoptar el convenio de que si $P[i] < 0$ entonces i es una raíz y $-P[i]$ es el tamaño del árbol.

```
procedure UNION( $i, j$ )
     $u := \text{FIND}(i)$ 
     $v := \text{FIND}(j)$ 
    if  $-P[u] > -P[v]$  then
        ▷  $u$  es el árbol “grande”
         $u := v$ 
    end if
    ▷  $u$  es el árbol “pequeño”
     $P[v] := P[v] + P[u]$ 
     $P[u] := v$ 
end procedure
▷  $1 \leq i \leq n$ 
procedure FIND( $i$ )
    while  $P[i] > 0$  do
         $i := P[i]$ 
    end while
    return  $i$ 
end procedure
```

El rendimiento $\mathcal{O}(\log n)$ de estas operaciones es consecuencia directa del siguiente lema.

Lema

Dado un bloque de tamaño k en una Particion con unión por peso, la altura del árbol correspondiente es $\leq \log_2 k$.

Demostración:

Si $k = 0$, el lema es obviamente cierto. Supongamos que es cierto para todos los tamaños hasta k y demostraremos entonces que es cierto para $k + 1$. Sea t el árbol correspondiente a un bloque de tamaño $k + 1$. Dicho bloque es el resultado de la unión de dos bloques de tamaños r y s , $r \leq s \leq k$. El árbol t tiene altura $h(t) \leq \max\{\log_2 r + 1, \log_2 s\}$, aplicando la hipótesis de inducción, y por la definición de unión por peso. Supongamos que $\log_2 r + 1 \leq \log_2 s$. Entonces

$$\begin{aligned} h(t) &\leq \max\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 s < \log_2(k + 1). \end{aligned}$$

Demostración (cont.):

Por otro lado, si $\log_2 r + 1 > \log_2 s$, y teniendo en cuenta que $k + 1 = r + s \geq 2r$,

$$\begin{aligned} h(t) &\leq \max\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 r + 1 = \log_2(2r) \leq \log_2(k + 1). \end{aligned}$$

□

Todavía puede conseguirse un mejor rendimiento empleando una heurística de *compresión de caminos*. La idea es reducir la distancia a la raíz de los elementos en el camino de i hasta la raíz durante una operación $\text{FIND}(i)$. Mientras se asciende desde i hasta la raíz se disminuye la altura del árbol.

Subsiguentes FINDS que afecten a i o alguno de los elementos que eran antecesores suyos serán más rápidos. La compresión de caminos acorta caminos de manera que poco a poco los árboles adoptan la forma que tendrían con *quick-find*, pero no teniendo que encargarse de ello la operación UNION ni compactándose todo un árbol de una sola vez.

- 1: Compresión por mitades: se modifica el apuntador de cada elemento en el camino desde i hasta $\text{FIND}(i)$, excepto a $\text{FIND}(i)$ y el hijo de éste, para que apunte a su “abuelo”.

```
procedure FIND( $i$ )
```

```
    ...
     $j := i; k := P[j]$ 
    while  $P[k] > 0$  do
         $P[j] := P[k]$ 
         $j := k$ 
         $k := P[j]$ 
    end while
    return  $k$ 
end procedure
```

- 2: Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta $\text{FIND}(i)$, para que apunte a la raíz.

```
procedure FIND( $i$ )
    ...
     $j := i$ 
    while  $P[j] > 0$  do
         $j := P[j]$ 
    end while
     $k := i$ 
    while  $P[k] > 0$  do
         $tmp := P[k]$ 
         $P[k] := j$ 
         $k := tmp$ 
    end while
    return  $k$ 
end procedure
```

Se ha demostrado que una secuencia de m UNIONes y n FINDs usando una de estas dos técnicas tiene coste $\mathcal{O}((m + n) \cdot \alpha(m, n))$, donde $\alpha(m, n)$ es la denominada *función inversa de Ackermann*. Puesto que $\alpha(m, n) \leq 4$ para cualesquiera valores de m y n concebibles en la práctica, el coste puede considerarse $\mathcal{O}(m + n)$. Aunque el coste de una UNION o un FIND no es constante, el *coste amortizado* sí lo es ya que el coste de las $m + n$ operaciones es $\mathcal{O}(m + n)$ en caso peor.

Parte II

Algoritmos Voraces

- Algoritmos Voraces Simples y Técnicas Generales
- Caminos Mínimos: Algoritmo de Dijkstra
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones
- Códigos de Huffman



Parte III

Programación Dinámica

Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- Algoritmos de Floyd y de Warshall
- Problema de la Mochila
- Problema del Viajante de Comercio
- Construcción de BSTs Óptimos

Introducción

La **programación dinámica** (PD) es un esquema de resolución de problemas de optimización que se fundamenta en dos ingredientes fundamentales:

- ① El **principio de optimalidad** (Bellman & Dreyfus, 1962)
“Un problema satisface el principio de optimalidad si cualquier subsolución (solución parcial) de la solución óptima de una instancia es solución óptima de la correspondiente subinstancia.”
- ② La **memoización**: las soluciones de los subproblemas se almacenan en una tabla para evitar hacer más de una vez la misma llamada recursiva; de hecho, con la memoización se elimina la recursividad y se concluye con un algoritmo iterativo

Son muchos los problemas que admiten una solución mediante PD. El esquema es conceptualmente recursivo, pero por lo general se trabaja con versiones iterativas, organizando cuidadosamente el orden de resolución de los subproblemas y utilizando la *memoización*, es decir, almacenando resultados intermedios en memoria con el fin de optimizar su rendimiento.

Algunos problemas que se resuelven mediante este esquema incluyen:

- Cálculo de la distancia de edición entre cadenas.
- Cálculo de los caminos mínimos entre todos los pares de vértices de un grafo (algoritmo de Floyd).
- Cálculo del BST ponderado estático óptimo.
- El problema de la mochila entera.
- El problema del viajante de comercio.
- Cálculo de la derivación incontextual más verosímil.
- Cálculo de los caminos mínimos desde un vértice a los restantes, con pesos eventualmente negativos (algoritmo de Bellman-Ford).
- Cálculo de la secuencia de mezclas óptima.
- Cálculo de la clausura transitiva de un grafo (algoritmo de Warshall).
- Cálculo de la cadena de multiplicación de matrices óptima.

Pasos para desarrollar una solución de programación dinámica:

- ① Establecer una recurrencia para el valor (coste, beneficio) de una solución óptima de la instancia dada en términos de los valores a soluciones óptimas de subinstancia, aplicando el **principio de optimalidad**
- ② Diseñar la estructura de datos apropiada para almacenar soluciones intermedias (**memoización**) y determinar el orden en que dichas soluciones intermedias deben obtenerse (cómo “rellenar” la tabla).

- ③ Implementar el algoritmo iterativo, calculando su coste en tiempo y espacio
- ④ Detectar posibles optimizaciones en tiempo y/o espacio
- ⑤ Diseñar el algoritmo que reconstruye una solución óptima, utilizando la tabla de memoización o alguna estructura de datos auxiliar que permite “recordar” cuáles son las subinstancias que proporcionan las soluciones parciales con las que se construye la solución óptima.

Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- Algoritmos de Floyd y de Warshall
- Problema de la Mochila
- Problema del Viajante de Comercio
- Construcción de BSTs Óptimos

Distancia de edición

Dados dos strings $x = x_1 \cdots x_m$ e $y = y_1 \cdots y_n$, la **distancia de edición** entre ambos, $\text{dist}(x, y)$, es el mínimo número de operaciones de edición (inserción de un carácter, borrado de un carácter, sustitución de un carácter por otro) necesarias para convertir x en y .

Por ejemplo, $\text{dist}(\text{BARCO}, \text{ARCOS}) = 2$, pues hay que borrar la B inicial y agregar una S al final para pasar de un string al otro.

Otro ejemplo: $\text{dist}(\text{PALAS}, \text{ATRAS}) = 3$, ya que hay que borrar la P inicial, sustituir la L por T (o por R) e insertar una R (o una T).

- 1 Empezaremos estableciendo la recurrencia para la distancia de edición. Sea $\delta_{i,j}$ la distancia de edición entre el prefijo $x_1 \dots x_i$ de longitud i de x y el prefijo $y_1 \dots y_j$ de longitud j de y , $0 \leq i \leq m$, $0 \leq j \leq n$. Entonces la distancia buscada es $\text{dist}(x, y) = \delta_{m,n}$. La base de recursión es simple:

$$\delta_{0,j} = j, \quad 0 \leq j \leq n,$$

$$\delta_{i,0} = i, \quad 0 \leq i \leq m,$$

puesto que hay que insertar j caracteres para convertir la cadena vacía en $y_1 \dots y_j$, y análogamente, hay que borrar i caracteres para convertir $x_1 \dots x_i$ en la cadena vacía.

- 1 Para el caso general, la distancia de edición será uno de las tres siguientes posibilidades:
- usar el mínimo de operaciones para convertir $x_1 \dots x_{i-1}$ en $y_1 \dots y_j$ y luego borrar x_i , ó
 - usar el mínimo de operaciones para convertir $x_1 \dots x_i$ en $y_1 \dots y_{j-1}$ y luego insertar y_j , ó
 - usar el mínimo de operaciones para convertir $x_1 \dots x_{i-1}$ en $y_1 \dots y_{j-1}$ y sustituir, si es necesario, x_i por y_j .

- 1 Poniendo todo junto, hay que tomar la opción que minimiza la distancia:

$$\delta_{i,j} = \min(\delta_{i-1,j} + 1, \delta_{i,j-1} + 1, \delta_{i-1,j-1} + s(x_i, y_j)),$$

donde $s(x_i, y_j) = 0$ si $x_i = y_j$ y $s(x_i, y_j) = 1$ si $x_i \neq y_j$.

- ② Usaremos una matriz o tabla bidimensional D con $m + 1$ filas por $n + 1$ columnas de manera que $D[i, j] = \delta_{i,j}$. La base de la recursión nos permite llenar la fila 0 y la columna 0 de la tabla D . Por otro lado en la recurrencia obtenida en el paso anterior observamos que el valor $\delta_{i,j}$ depende del valor en la columna inmediatamente anterior $(i, j - 1)$ y de dos de los valores de la fila previa $(i - 1, j)$ e $(i - 1, j - 1)$. Esto significa que la matriz D debe rellenarse por filas de arriba ($i = 1$) a abajo ($i = m$) y, para cada fila, por columnas, de izquierda ($j = 1$) a derecha ($j = n$).

3

procedure EDITDIST(x, y)▷ $m = |x|, n = |y|, D$: matriz $[0..m, 0..n]$ de enteros**for** $j := 0$ **to** n **do** $D[0, j] := j$ **end for****for** $i := 1$ **to** m **do** $D[i, 0] := i$ **end for****for** $i := 1$ **to** m **do****for** $j := 1$ **to** n **do** ▷ $s(a, b)$ retorna 0 si $a = b$, y 1 si $a \neq b$

$$D[i, j] := \min(D[i - 1, j] + 1, D[i, j - 1] + 1,$$

$$D[i - 1, j - 1] + s(x[i], y[j]))$$

end for**end for****return** $D[m, n]$ **end procedure**

- ③ El coste del algoritmo viene naturalmente dominado por el coste del bucle principal, en el que se hacen $m \cdot n$ iteraciones, cada una de las cuales tiene coste $\Theta(1)$. El **coste en espacio y en tiempo** del algoritmo es $\Theta(m \cdot n)$.

Para fijar ideas, si $m = c \cdot n$ entonces el tamaño de la entrada (= suma de las longitudes de los strings) es $m + n = \Theta(n)$, y el coste del algoritmo en tiempo y espacio es $\Theta(n^2)$, es decir, cuadrático respecto al tamaño de la entrada.

- 4 Resulta evidente que no necesitamos tener una matriz D completa: para hallar la entrada (i, j) nos basta tener las entradas previas de la fila i y la fila $i - 1$. Podemos entonces usar un par de vectores $D[0..n]$ y $Dprev[0..n]$ de manera que mantengamos el siguiente invariante: en la iteración que ha de calcular $\delta_{i,j}$ se cumple que $Dprev[k] = \delta_{i-1,k}$ para toda k y $D[k] = \delta_{i,k}$ para toda $k < j$.

El coste del algoritmo en tiempo sigue siendo $\Theta(m \cdot n)$, pero el coste en espacio lo hemos reducido a $\Theta(n)$. Si $n > m$ convendrá intercambiar x e y al inicio: la distancia de edición es la misma.

4

procedure EDITDIST(x, y)▷ $m = |x|, n = |y|, D, Dprev$: vectores $[0..n]$ de enteros**for** $j := 0$ **to** n **do** $Dprev[j] := j$ **end for****for** $i := 1$ **to** m **do** $D[0] := i$ **for** $j := 1$ **to** n **do**

$$D[j] := \min(Dprev[j] + 1, D[j - 1] + 1,$$

$$Dprev[j - 1] + s(x[i], y[j]))$$

end for▷ $D \equiv \delta_{i..}, Dprev \equiv \delta_{i-1..}$ $Dprev := D$ ▷ Copia D en $Dprev$ **end for****return** $D[n]$ **end procedure**

Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- **Algoritmos de Floyd y de Warshall**
- Problema de la Mochila
- Problema del Viajante de Comercio
- Construcción de BSTs Óptimos

Algoritmo de Floyd



Robert W. Floyd (1936–2001) Stephen Warshall (1935–2006)

El algoritmo de Floyd (1962) para los caminos mínimos entre todos los pares de vértices de un grafo dirigido G se basa en el esquema de PD. En realidad estamos resolviendo n^2 problemas de optimización, uno por cada par de vértices, pero están relacionados unos con otros y los resolveremos simultáneamente construyendo un espacio de estados adecuado. Concretamente consideramos el (sub)problema “camino mínimo de i a j que pase exclusivamente por los primeros k vértices de G (excluídos los extremos)”. Sea $P_{i,j}^{(k)}$ la distancia de dicho camino.

Para solución $k = 0$ la solución es trivial: si $i = j$ entonces el coste del caamino mínimo es $P_{i,i}^{(0)} = 0$; si $i \neq j$ y $(i, j) \in E$ entonces el coste es el del arco que une i con j ; si $i \neq j$ y $(i, j) \notin E$ entonces el coste es $+\infty$ ya que no existe un camino entre i y j usando cero vértices intermedios.

La respuesta buscada son los valores $P_{i,j}^{(n)}$, es decir, las distancia miínimas entre todos los pares de vértices (i, j) , usando cualesquiera vértices.

La recurrencia para $P_{i,j}^{(k)}$ si se conoce $P_{u,v}^{(k-1)}$ para todo par de vértices (u, v) . En efecto, el mejor camino de i a j será uno que no usa el vértice k ($= P_{i,j}^{(k-1)}$) o bien un camino que va de i a k y de k a j ($= P_{i,k}^{(k-1)} + P_{k,j}^{(k-1)}$). Fijoos en el uso del principio de optimalidad, de manera bastante intuitiva, en este problema: si π es un camino óptimo entre u y v que pasa por un vértice w entonces los subcaminos que van de u a w y de w a v son necesariamente mínimos. Si no lo fueran, entonces π no sería el mínimo!

Formalmente, si $k > 0$,

$$P_{i,j}^{(k)} = \min\{P_{i,j}^{(k-1)}, P_{i,k}^{(k-1)} + P_{k,j}^{(k-1)}\}$$

A priori parece necesitarse una tabla de $\Theta(n^3)$ entradas para resolver el problema, pero puesto que los $P_{i,j}^{(k)}$'s dependen exclusivamente de los valores $P_{r,s}^{(k-1)}$ podríamos mantener exclusivamente dos tablas con n^2 entradas que mantengan los costes de las “fases” $k - 1$ y k . El orden en que se resolviesen los n^2 subproblemas de la fase k sería irrelevante.

Pero podemos solucionar el problema con una sola tabla $n \times n$ ya que durante la fase k los únicos valores que necesitamos para evaluar el coste $P_{i,j}^{(k)}$ son el valor previo $P_{i,j}^{(k-1)}$ y los de la fila k ($P_{k,j}^{(k-1)}$) y la columna k ($P_{i,k}^{(k-1)}$).

Pero durante la fase k no puede cambiar ningún valor de la fila ni la columna k ya que $P_{k,k}^{(k-1)} = 0$.

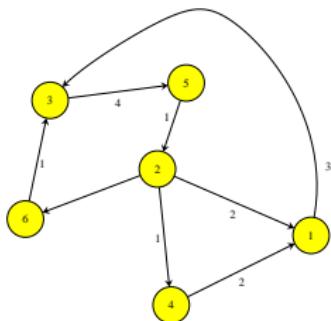
Utilizaremos además otra tabla U de tamaño $n \times n$ para registrar cuáles son los caminos mínimos. La entrada $U[i, j] = 0$ si el camino mínimo entre i y j no existe o consiste en un único arco que une i y j . Por otra parte, si $U[i, j] = k > 0$, ($i \neq k, j \neq k$) entonces el camino se compone de dos caminos óptimos: el que va de i a k y el que va de k a j . El procedimiento que permite recuperar el camino mínimo entre dos vértices a partir de la información en U es sencillo.

```
typedef vector < vector<double> > AdjMatrix;
typedef AdjMatrix MinCostMatrix;
typedef vector< vector<int> > MinPathMatrix;
// Pre:
// G[i][j] == peso del arco (i,j) si existe,
//           == +infinity si el arco (i,j) si no existe
// Post: ver la descripcion del texto
void floyd(const AdjMatrix& G, MinCostMatrix& P,
           MinPathMatrix& U) {
    int n = G.size();
    P = G;
    for (int i = 0; i < n; ++i) P[i][i] = 0;
    U = MinPathMatrix(n, vector<int>(n, -1));
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (P[i][k]+P[k][j] < P[i][j]) {
                    P[i][j] = P[i][k]+P[k][j];
                    U[i][j] = k;
                }
}
```

El coste en espacio del algoritmo de Floyd es $\Theta(n^2)$. Por otra parte, es obvio que el coste en tiempo es $\Theta(n^3)$. Una alternativa al uso de este algoritmo consiste en emplear el algoritmo de Dijkstra, con un vértice distinto como fuente en cada ocasión. El coste de esta alternativa, usando listas de adyacencias y un *heap* para seleccionar el candidato de cada fase es, en caso peor, $\Theta(mn \log n)$, donde m es el número de arcos.

Si el grafo es denso ($m = \Omega(n^2)$) entonces el algoritmo de Floyd será preferible por su menor coste asintótico y su gran sencillez. Si el grafo es disperso ($m \ll n^2$) será más conveniente emplear n veces el algoritmo de Dijkstra.

Si el algoritmo de Dijkstra no utilizase el *heap* y el grafo viniese representado mediante una matriz de adyacencia entonces el coste de n -Dijkstra sería $\Theta(n^3)$ y el algoritmo de Floyd resultaría más atractivo por su manifiesta simplicidad.



$$P^{(0)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & \infty & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

$$P^{(1)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & 5 & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & \infty & \infty \\ \infty & 1 & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

$$P^{(2)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & 5 & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & \infty & \infty \\ 3 & 1 & 6 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

$$P^{(4)} = P^{(3)} = \begin{pmatrix} 0 & \infty & 3 & \infty & 7 & \infty \\ 2 & 0 & 5 & 1 & 9 & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & 9 & \infty \\ 3 & 1 & 6 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 5 & 0 \end{pmatrix}$$

$$P^{(5)} = \begin{pmatrix} 0 & 8 & 3 & 9 & 7 & 10 \\ 2 & 0 & 5 & 1 & 9 & 2 \\ 7 & 5 & 0 & 6 & 4 & 7 \\ 2 & 10 & 5 & 0 & 9 & 12 \\ 3 & 1 & 6 & 2 & 0 & 3 \\ 8 & 6 & 1 & 7 & 5 & 0 \end{pmatrix}$$

$$P^{(6)} = \begin{pmatrix} 0 & 8 & 3 & 9 & 7 & 10 \\ 2 & 0 & 3 & 1 & 7 & 2 \\ 7 & 5 & 0 & 6 & 4 & 7 \\ 2 & 10 & 5 & 0 & 9 & 12 \\ 3 & 1 & 4 & 2 & 0 & 3 \\ 8 & 6 & 1 & 7 & 5 & 0 \end{pmatrix}$$

Si en vez de inicializar la matriz P con los costes trabajamos con la matriz de adyacencia de G y cambiamos el cálculo del \min por \vee y $+$ por \wedge , el algoritmo resultante es el denominado algoritmo de Warshall para el cálculo de la clausura transitiva de G . Es decir, al finalizar $P[i, j] = \text{true}$ si y sólo si existe un camino entre i y j en G . No es un algoritmo de PD propiamente dicho ya que no estamos optimizando ningún coste.

```
typedef vector< vector<bool> > AdjMatrix;
void Warshall(AdjMatrix& G) {
    int n = G.size();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                G[i][j] = G[i][j] or (G[i][k] and G[k][j]);
}
```

Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- Algoritmos de Floyd y de Warshall
- Problema de la Mochila**
- Problema del Viajante de Comercio
- Construcción de BSTs Óptimos



Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- Algoritmos de Floyd y de Warshall
- Problema de la Mochila
- Problema del Viajante de Comercio**
- Construcción de BSTs Óptimos



Parte III

Programación Dinámica

- Introducción a la Programación Dinámica
- Distancia de edición
- Algoritmos de Floyd y de Warshall
- Problema de la Mochila
- Problema del Viajante de Comercio
- Construcción de BSTs Óptimos



Parte IV

Flujos sobre Redes y Programación Lineal

Parte IV

Flujos sobre Redes y Programación Lineal

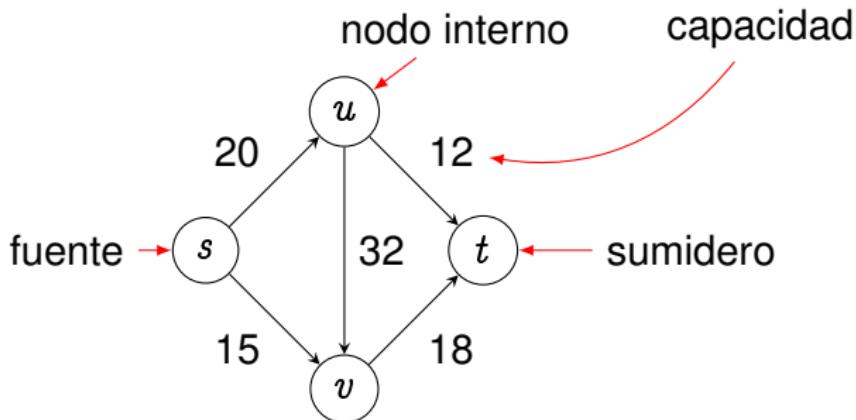
- Flujos sobre Redes
- Programación Lineal

Flujos sobre Redes

Definición

Una **red s - t** es un grafo dirigido $G = \langle V, E \rangle$ donde

- ① Cada arco e tiene una capacidad $c_e > 0$.
- ② El vértice $s \in V$, denominado **fuente** (ing: *source*) es el único vértice de G con grado de entrada 0.
- ③ El vértice $t \in V$, denominado **sumidero** (ing: *sink*) es el único vértice de G con grado de salida 0.



Definición

Un **flujo** f sobre una red G es una función $f : E(G) \rightarrow \mathbb{R}$ tal que:

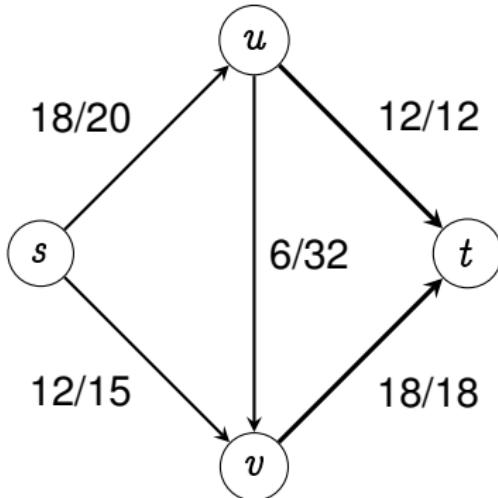
- 1 Para todo arco $e \in E(G)$,

$$0 \leq f(e) \leq c_e \quad (\text{condiciones de capacidad})$$

- 2 Para todo vértice $v \in V(G) \setminus \{s, t\}$,

$$\sum_{e=(v,w)} f(e) - \sum_{e=(w,v)} f(e) = 0 \quad (\text{conservación del flujo})$$

Un ejemplo de flujo



Las etiquetas x/y sobre los arcos denotan el flujo acarreado por el arco (x) y su capacidad (y); empleamos un trazo más grueso para indicar que un arco lleva flujo, y muy grueso para indicar que está saturado.

Si definimos

$$f^{\text{out}}(v) = \sum_{e=(v,w)} f(e) \quad \text{flujo saliente}$$

$$f^{\text{in}}(v) = \sum_{e=(w,v)} f(e) \quad \text{flujo entrante}$$

las condiciones de conservación del flujo las podemos reescribir

$$f^{\text{out}}(v) - f^{\text{in}}(v) = 0$$

para todo $v \in V(G) \setminus \{s, t\}$.

Salvo que el flujo f sea nulo ($f(e) = 0$ para todo e), la fuente s emite flujo ($f^{\text{out}}(s) > 0$ y $f^{\text{in}}(s) = 0$) y el sumidero t lo absorbe ($f^{\text{in}}(t) > 0$ y $f^{\text{out}}(t) = 0$).

Si para un arco e se cumple que $f(e) = c_e > 0$ se dice que el arco está saturado.

Dado un conjunto de vértices A ,

$$f^{\text{out}}(A) = \sum_{v \in A} f^{\text{out}}(v)$$

$$f^{\text{in}}(A) = \sum_{v \in A} f^{\text{in}}(v)$$

El **valor** $v(f)$ de un flujo f es el flujo saliente de s , que coincide con el flujo entrante en t

$$v(f) = f^{\text{out}}(s) = f^{\text{in}}(t)$$

El problema a resolver se puede entonces formular entonces en los siguientes términos:

“Dada G , una red s - t , hallar un flujo f máximo, es decir, un flujo sobre G cuyo valor $v(f)$ es máximo.”

Obs: una red G puede admitir varios flujos máximos, todos de idéntico valor.

Un cota trivial al máximo flujo alcanzable:

$$\begin{aligned}v(f) &= f^{\text{out}}(s) = \sum_{e=(s,v)} f(e) \\&\leq \sum_{e=(s,v)} c_e\end{aligned}$$

Definición

Un **corte $s-t$** de una red G es una partición $\langle A, B \rangle$ del conjunto de vértices $V(G)$ tal que:

- $A \cup B = V(G)$
- $A \cap B = \emptyset$
- $s \in A$ y $t \in B$

Definición

Dado $\langle A, B \rangle$, un corte $s-t$ de G , su **capacidad** es

$$c(A, B) = \sum_{\substack{e=(u,v) \\ u \in A, v \in B}} c_e$$

Lema

Sea $\langle A, B \rangle$ un corte $s-t$ de una red G , y f un flujo cualquiera sobre la red. Entonces

$$f^{\text{out}}(A) - f^{\text{in}}(A) = v(f)$$

Demostración:

Puesto que $f^{\text{out}}(s) = v(f)$ y $f^{\text{in}}(s) = 0$

$$\begin{aligned} f^{\text{out}}(A) &= f^{\text{out}}(s) + \sum_{v \in A \setminus \{s\}} f^{\text{out}}(v) \\ &= v(f) + \sum_{v \in A \setminus \{s\}} f^{\text{out}}(v) \end{aligned}$$

$$\begin{aligned} f^{\text{in}}(A) &= f^{\text{in}}(s) + \sum_{v \in A \setminus \{s\}} f^{\text{in}}(v) \\ &= \sum_{v \in A \setminus \{s\}} f^{\text{in}}(v) \end{aligned}$$

Demostración (cont.):

Luego

$$\begin{aligned}f^{\text{out}}(A) - f^{\text{in}}(A) &= v(f) + \sum_{v \in A \setminus \{s\}} f^{\text{out}}(v) - \sum_{v \in A \setminus \{s\}} f^{\text{in}}(v) \\&= v(f) + \sum_{v \in A \setminus \{s\}} (f^{\text{out}}(v) - f^{\text{in}}(v)) \\&= v(f)\end{aligned}$$

puesto que $f^{\text{out}}(v) - f^{\text{in}}(v) = 0$ para todo $v \notin \{s, t\}$. \square

Se puede demostrar de manera parecida que para cualquier corte $\langle A, B \rangle$ y cualquier flujo f

$$f^{\text{in}}(B) - f^{\text{out}}(B) = f^{\text{out}}(A) - f^{\text{in}}(A) = v(f)$$

Teorema

Dados un corte $s-t$ cualquiera $\langle A, B \rangle$ y un flujo f cualquiera

$$v(f) \leq c(A, B)$$

“El valor de un flujo no puede exceder la capacidad de ningún corte de la red”

La cota trivial $v(f) \leq \sum_{e=(s,v)} c_e$ es un caso particular del teorema, tomando $A = \{s\}$ y $B = V(G) \setminus \{s\}$.

Demostración:

Comenzaremos demostrando un resultado intermedio importante:

$$f^{\text{out}}(A) - f^{\text{in}}(A) = \sum_{e \text{ sale de } A} f(e) - \sum_{e \text{ entra en } A} f(e)$$

Por el lema demostrado anteriormente

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

Tenemos

$$f^{\text{out}}(A) = \sum_{v \in A} f^{\text{out}}(v) = \sum_{v \in A} \sum_{e=(v,w)} f(e)$$

$$f^{\text{in}}(A) = \sum_{v \in A} f^{\text{in}}(v) = \sum_{v \in A} \sum_{e=(w,v)} f(e)$$

$$f^{\text{out}}(A) - f^{\text{in}}(A) = \sum_{v \in A} \left(\sum_{e=(v,w)} f(e) - \sum_{e=(w,v)} f(e) \right)$$

Demostración (cont.):

Consideraremos tres casos:

- 1 Arcos que salen de v hacia un vértice fuera de A , es decir, de la forma (v, w) con $w \in B$: contribuyen el término

$$\sum_{\substack{e=(v,w) \\ w \in B}} f(e)$$

- 2 Arcos que llegan a v desde un vértice fuera de A (entran en A), es decir, de la forma (w, v) siendo $w \in B$: contribuyen el término

$$- \sum_{\substack{e=(w,v) \\ w \in B}} f(e)$$

Demostración (cont.):

- ③ Arcos “internos” a A , de la forma (x, y) con $x, y \in A$: cuando sumamos sobre todos los vértices v de A contribuyen 0

$$\begin{aligned} & \sum_{v \in A} \left(\sum_{\substack{e=(v,w) \\ w \in A}} f(e) - \sum_{\substack{e=(w,v) \\ w \in A}} f(e) \right) \\ &= \sum_{\substack{e=(v,w) \\ v \in A, w \in A}} f(e) - \sum_{\substack{e=(w,v) \\ v \in A, w \in A}} f(e) = 0 \end{aligned}$$

Demostración (cont.): Recapitulando

$$\begin{aligned}v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) = \sum_{v \in A} \sum_{\substack{e=(v,w) \\ w \in B}} f(e) - \sum_{v \in A} \sum_{\substack{e=(w,v) \\ w \in B}} f(e) \\&= \sum_{e \text{ sale de } A} f(e) - \sum_{e \text{ entra en } A} f(e)\end{aligned}$$

Como todos los flujos son positivos o cero,

$$\begin{aligned}v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) = \sum_{e \text{ sale de } A} f(e) - \sum_{e \text{ entra en } A} f(e) \\&\leq \sum_{e \text{ sale de } A} f(e) \leq \sum_{e \text{ sale de } A} c_e = c(A, B)\end{aligned}$$

□

El algoritmo de Ford-Fulkerson y el teorema *maxflow-mincut*

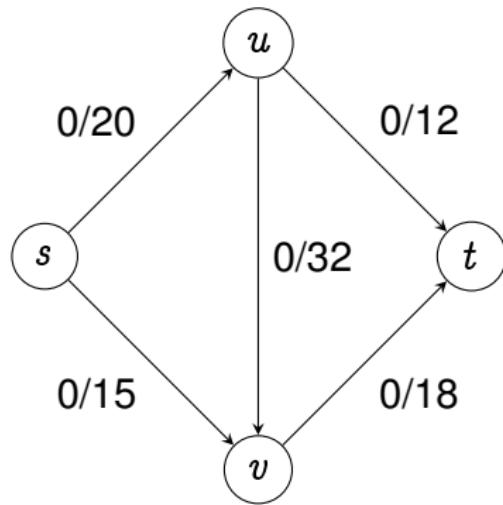
El algoritmo de Ford-Fulkerson (1954) calcula un flujo máximo a través de una serie de etapas sucesivas. Se parte de un flujo nulo y cada etapa envía flujo adicional de s a t , incrementando el valor del flujo en curso.

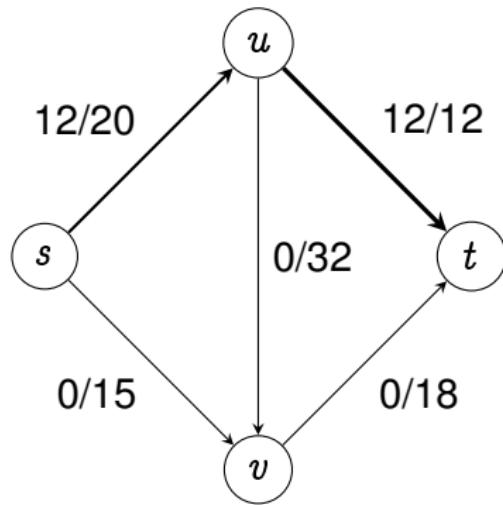
Para ello se busca, en cada etapa, un camino de s a t en el que se pueda incrementar el flujo que atraviesa los arcos del camino. El valor del flujo se va acercando progresivamente al valor máximo. El algoritmo finaliza cuando no podemos encontrar un camino que nos lleve de s a t en el que podamos incrementar el flujo.

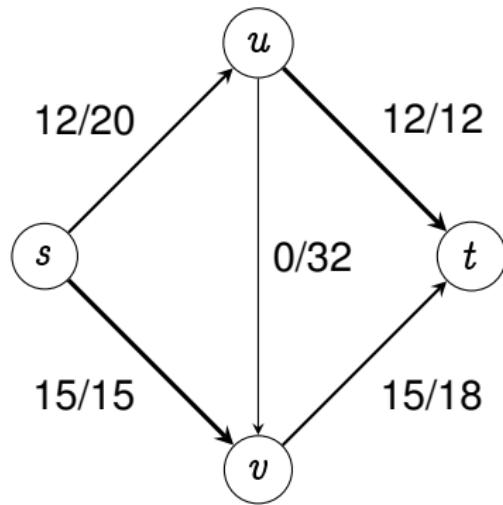
De ahora en adelante asumiremos que todas las capacidades c_e de la red son enteros positivos, y nuestro análisis de la corrección del algoritmo y de su tiempo de ejecución se basarán en esta hipótesis.

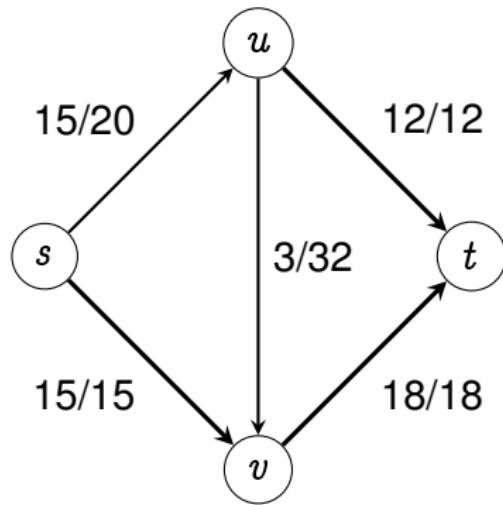
El algoritmo de Ford-Fulkerson va calculando flujos sucesivos f_0, f_1, f_2, \dots de manera que $v(f_i) < v(f_{i+1})$, con f_0 el flujo nulo; entre dos flujos sucesivos f_i y f_{i+1} sólo son diferentes los flujos enviados a través de los arcos de un cierto camino, todos los restantes arcos llevan el mismo flujo en f_i y en f_{i+1} . A medida que evoluciona el algoritmo podemos pensar en sucesivas redes $G_0 = G, G_1, \dots$ donde el flujo asignado a los arcos e de G_i les deja una capacidad remanente o **residual** $c'_e = c_e - f_i(e)$.

Este procedimiento nos puede llevar a un callejón sin salida si elegimos “mal” el camino que nos lleva de s a t , pues podemos saturar arcos vitales para llegar de s a t bloqueando otros posibles flujos de mayor valor. Por ello debemos considerar la posibilidad de retornar el flujo o parte de él que hayamos enviado a través de un arco.







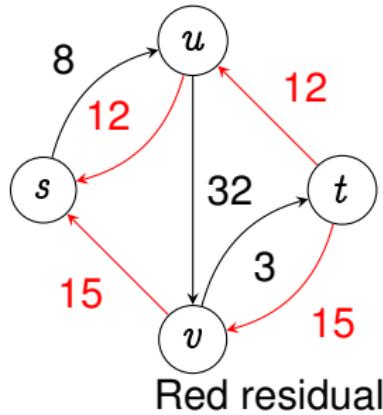
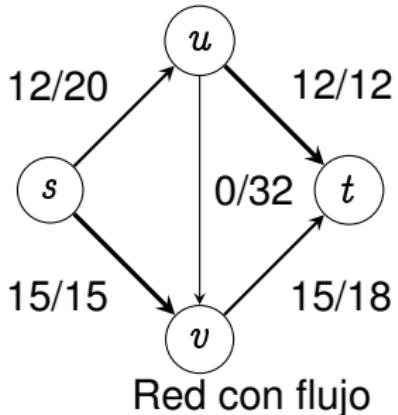


Definición

Dada una red $G = \langle V, E \rangle$ y un flujo f sobre G , la **red residual** (o *grafo residual*) $G_f = \langle V_f, E_f \rangle$ se define como sigue:

- 1 $V_f = V$
- 2 Si $e = (u, v) \in E$ y $f(e) < c_e$ entonces $e = (u, v) \in E_f$ y $\bar{c}_e = c_e - f(e)$ (**arcos de avance**; ing: *forward edges*)
- 3 Si $e = (u, v) \in E$ y $f(e) > 0$ entonces $e' = (v, u) \in E_f$ y $\bar{c}_{e'} = f(e)$ (**arcos de retroceso**; ing: *backward edges*)

Observación: en el grafo residual todas las capacidades residuales \bar{c} son números enteros positivos si los flujos lo son



Los arcos de avance de la red residual se muestran con línea sólida y etiquetados por la capacidad residual ($\bar{c}_e = c_e - f(e)$); los arcos de retroceso se muestran con línea discontinua y etiquetados por su capacidad residual ($\bar{c}_{(v,u)} = f((u,v))$)

Caminos de aumentación

Dada una red G y un flujo f , un **camino de aumentación** (ing: *augmenting path*) es un camino simple entre s y t en el grafo residual G_f .

```
procedure BOTTLENECK( $P$ )
    return la capacidad residual mínima de un arco de  $P$ 
end procedure
procedure PUSHFLOW( $P, f$ )
     $b := \text{BOTTLENECK}(P)$ 
    for  $e = (u, v) \in P$  do
        if  $e$  es un arco de avance en  $G_f$  then
             $f(e) := f(e) + b$ 
        else  $\triangleright e$  es un arco de retroceso
             $e' := (v, u)$ 
             $f(e') := f(e') - b$ 
        end if
    end for
    return  $f$ 
end procedure
```

Observación: Si para todos los arcos e el flujo $f(e)$ es un número entero positivo, lo mismo ocurre con $f' = \text{PUSHFLOW}(P, f)$, es decir, $f'(e)$ es un número entero positivo también.

Proposición

Sea $f' = \text{PUSHFLOW}(P, f)$. Entonces f' es un flujo válido sobre la red G y $v(f') = v(f) + \text{BOTTLENECK}(P) > v(f)$

Demostración:

Sea $b = \text{BOTTLENECK}(P)$.

- 1) Condiciones de capacidad: Sólo se modifica el flujo en los arcos de P .

Supongamos que e es un arco de avance. Entonces $f'(e) = f(e) + b$. Pero $\bar{c}_e = c_e - f(e) \geq b$ pues b es la capacidad residual mínima, luego $c_e \geq b + f(e) = f'(e)$. Como $f(e) \geq 0$ y $b > 0$ tenemos que $f'(e) > 0$.

Supongamos ahora que $e = (u, v)$ es un arco de retroceso. Entonces $f'((v, u)) = f((v, u)) - b$. La capacidad residual de e es $\bar{c}_e = f((v, u))$ y por tanto $c_{(u,v)} \geq f((v, u)) > f'((v, u)) = f((v, u)) - b \geq 0$, ya que $b \leq \bar{c}_e$.

Demostración (cont.):

- ② Conservación del flujo: Consideremos cualquier vértice que no está en el camino P . Como el flujo f' es igual al flujo f en los arcos que entran y que salen de v , la conservación del flujo se cumple para dichos vértices. Para los vértices $v \notin \{s, t\}$ que forman parte del camino P tenemos que considerar cuatro posibles casos (FF,FR,RF,RR), según que el camino P llegue al vértice por un arco de avance o uno de retroceso, y que el camino salga de v por un arco de avance o uno de retroceso.

Demostración (cont.):

Por ejemplo, supongamos el caso FR. Los arcos $e_1 = (u, v)$ de avance y $e_2 = (v, w)$ de retroceso forman parte del camino P . Ningún arco de salida de v en el grafo original cambia, luego $f^{\text{out}}(v) = f'^{\text{out}}(v)$. Pero los arcos de entrada (u, v) y (w, v) sí cambian su flujo. No obstante como $f'((u, v)) = f((u, v)) + b$ y $f'((w, v)) = f((w, v)) - b$, $f^{\text{in}}(v) = f'^{\text{in}}(v)$, de lo cual deducimos que el flujo se conserva en v . Los otros tres casos (FF,RF,RR) se demuestran razonando de manera análoga.

Demostrado que f' es válido y puesto que P necesariamente debe comenzar con un arco de avance que sale de s (G_f no puede tener arcos de retroceso que salgan de s !),

$$v(f') = f'^{\text{out}}(s) = f^{\text{out}}(s) + b = v(f) + b$$

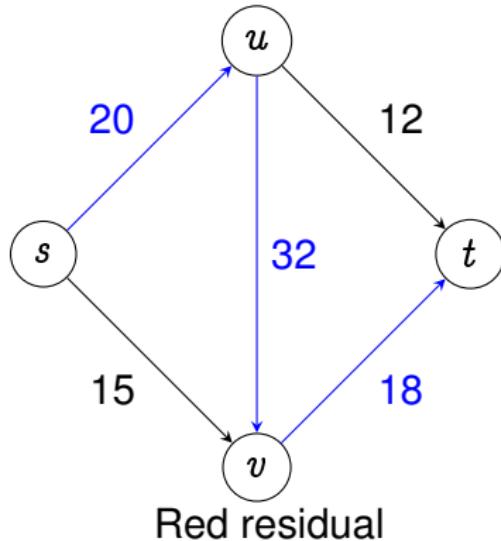
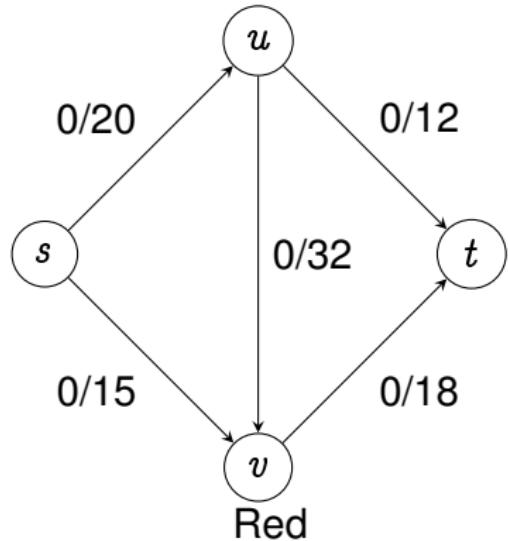
□

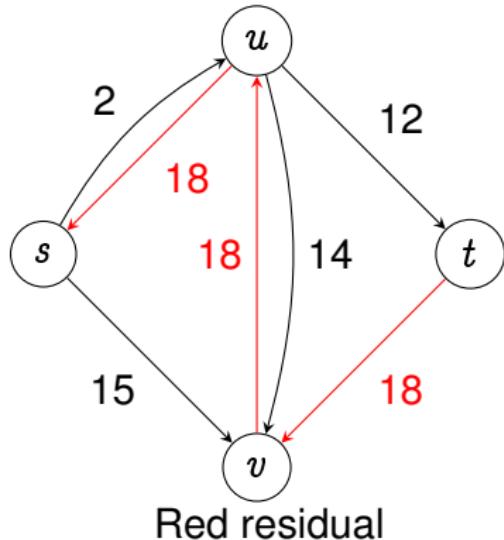
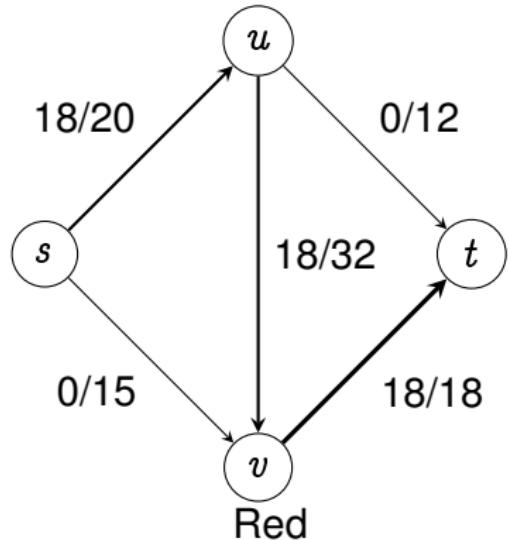
El algoritmo de Ford-Fulkerson (FF)

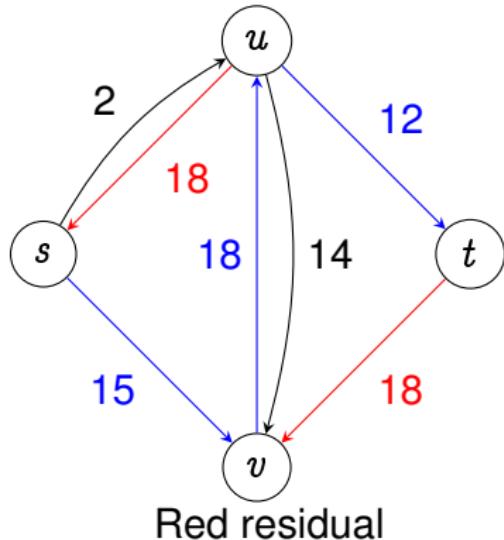
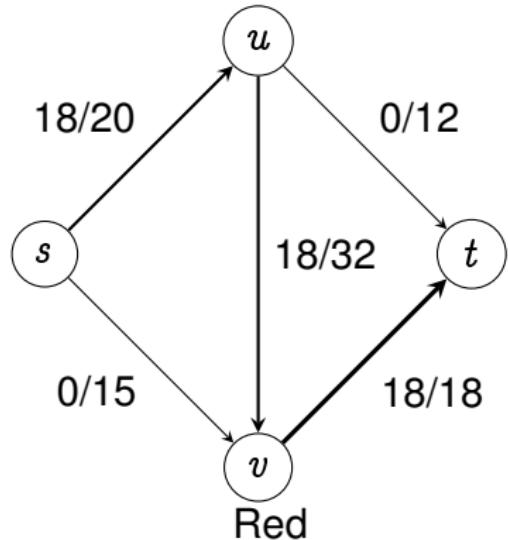
Require: G una red $s-t$

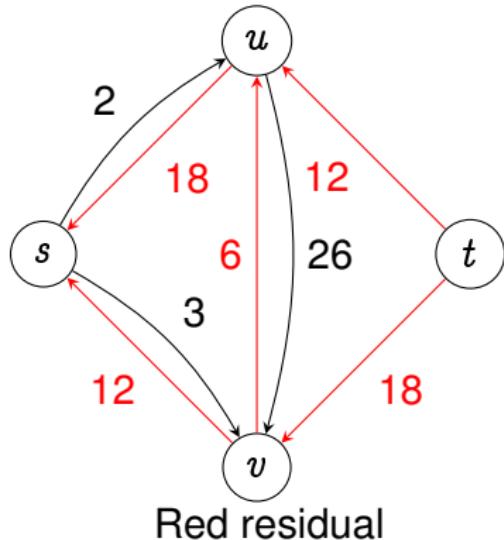
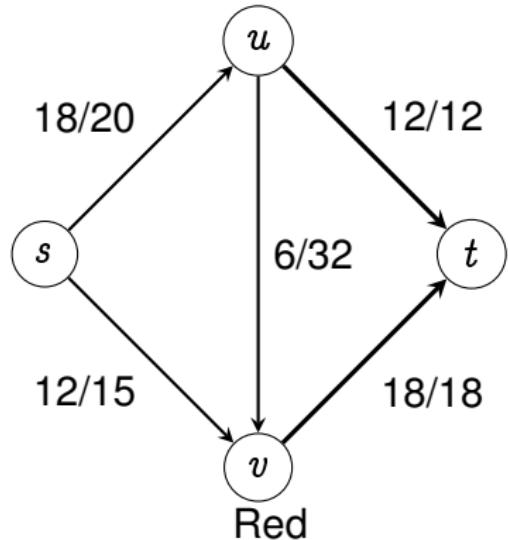
Ensure: $f = \text{FORD-FULKERSON}(G)$ es un flujo máximo sobre G

```
procedure FORD-FULKERSON( $G$ )
    for  $e \in E(G)$  do
         $f(e) := 0$ 
    end for
     $G_{res} := G$ 
    while existen caminos de aumentación entre  $s$  y  $t$  en  $G_{res}$  do
         $P :=$  un camino de aumentación
         $f := \text{PUSHFLOW}(P, f)$ 
        ACTUALIZARRESIDUAL( $G_{res}, P, f$ )
    end while
    return  $f$ 
end procedure
```









Terminación y tiempo de ejecución de FF

En cada iteración del bucle principal del algoritmo el valor del flujo f aumenta al menos en una unidad. Por tanto el número de iteraciones del algoritmo está acotado superiormente por

$$C = \sum_{e=(s,v)} c_e,$$

y la terminación del algoritmo está garantizada.

Cada iteración puede implementarse con coste $\mathcal{O}(m + n)$ ya que el grafo residual nunca tiene más de $2m$ arcos ($m = |E(G)|$) y un camino de aumentación contiene a lo sumo $n = |V(G)|$ vértices.

El coste total del algoritmo es $\mathcal{O}((m + n) \cdot C)$. En caso peor, este coste puede llegar a ser muy elevado; es posible construir redes en las que C sea muy grande, y en cada iteración sólo aumente una unidad el valor del flujo f .

Corrección

El algoritmo de Ford-Fulkerson se termina cuando el grafo residual G_f no contiene un camino de s a t . ¿Por qué eso nos garantiza que el flujo f es entonces máximo?

Sea f_{FF} el flujo hallado por el algoritmo. En el correspondiente grafo residual $G_{f_{FF}}$ no hay ningún camino de s a t . Definamos dos conjuntos de vértices:

$$A_{FF} = \{v \in G_{f_{FF}} \mid v \text{ es accesible desde } s\}$$

$$B_{FF} = \{v \in G_{f_{FF}} \mid v \text{ no es accesible desde } s\}$$

El par $\langle A_{FF}, B_{FF} \rangle$ es un corte $s-t$ del grafo G : todo vértice de G pertenece a A_{FF} o a B_{FF} , s pertenece a A_{FF} (porque siempre podemos acceder a s desde s !) y t pertenece a B_{FF} , puesto que $G_{f_{FF}}$ no podemos acceder a t desde s .

Ya hemos visto antes que

$$\begin{aligned} v(f_{FF}) &= f^{\text{out}}(A_{FF}) - f^{\text{in}}(A_{FF}) \\ &= \sum_{e \text{ sale de } A_{FF}} f(e) - \sum_{e \text{ entra } A_{FF}} f(e) \end{aligned}$$

Sea e un arco que sale de A_{FF} (sale de un cierto $v \in A_{FF}$ y llega a un vértice $w \in B_{FF}$). No puede existir un arco $e = (v, w)$ en $G_{f_{FF}}$, porque sino tendríamos la posibilidad de acceder a w desde s , lo cual es una contradicción. La única forma de que (v, w) no sea un arco en $G_{f_{FF}}$ es que el flujo sature e , dejando una capacidad residual nula (y por tanto e no aparece en el residual). Así que para todo arco e que sale de A_{FF} , $f(e) = c_e$.

Sea $e = (w, v)$ un arco que entra en A_{FF} (sale de un cierto $w \in B_{FF}$ y llega a un cierto $v \in A_{FF}$). Razonando como antes, no puede existir un arco de retroceso $e' = (v, w)$ en el grafo residual $G_{f_{FF}}$, porque sino w sería accessible desde s y por definición w no es accessible. Para que no exista el arco de retroceso $e' = (v, w)$ en $G_{f_{FF}}$ tiene que ocurrir que el arco $e = (w, v)$ del grafo original no lleve flujo, porque si $f(e) > 0$ entonces $e' = (v, w)$ estaría en el residual y su capacidad $\bar{c}_{e'} = f(e)$. Para todo arco e que entra en A_{FF} se tiene que cumplir que $f(e) = 0$.

Por lo tanto

$$\begin{aligned}v(f_{FF}) &= \sum_{e \text{ sale de } A_{FF}} f(e) - \sum_{e \text{ entra } A_{FF}} f(e) \\&= \sum_{e \text{ sale de } A_{FF}} c_e - 0 \\&= \sum_{e \text{ sale de } A_{FF}} c_e = c(A_{FF}, B_{FF})\end{aligned}$$

Por un teorema que hemos visto anteriormente ningún flujo puede tener un valor que exceda la capacidad de ningún corte $s-t$. Como $v(f_{FF})$ coincide con $c(A_{FF}, B_{FF})$ la única conclusión posible es que $v(f_{FF})$ es máximo y que $c(A_{FF}, B_{FF})$ es mínima.

Acabamos de demostrar un importante teorema.

Teorema (Maxflow-mincut)

Para toda red s - t con capacidades enteras, existe un flujo máximo f^* cuyo valor coincide con la capacidad de un corte $\langle A^*, B^* \rangle$ de capacidad mínima.

$$v(f^*) = c(A^*, B^*)$$

El algoritmo de Ford-Fulkerson nos retorna un flujo f_{FF} de valor máximo; el corte $\langle A_{FF}, B_{FF} \rangle$ tiene capacidad mínima.

Parte IV

Flujos sobre Redes y Programación Lineal

- Flujos sobre Redes
- Programación Lineal



Parte V

Estructuras de Datos Avanzadas

Parte V

Estructuras de Datos Avanzadas

- Tries
- Skip Lists

Tries

Las claves que identifican a los elementos de una colección están formadas por una secuencia de símbolos (p.e. caracteres, dígitos, bits), siendo esta descomposición más o menos “natural”, y dicha circunstancia puede aprovecharse ventajosamente para implementar las operaciones típicas de un diccionario de manera notablemente eficiente.

Adicionalmente, es frecuente que necesitemos ofertar operaciones del TAD basadas en esta descomposición de las claves en símbolos: por ejemplo, podemos querer una operación que dada una colección de palabras C y una palabra p nos devuelva la lista de todas las palabras de C que contienen a la palabra p como subcadena.

Consideremos un alfabeto finito $\Sigma = \{\sigma_1, \dots, \sigma_m\}$ de cardinalidad $m \geq 2$. Mediante Σ^* denotaremos, como es habitual en muchos contextos, el conjunto de las secuencias o cadenas formadas por símbolos de Σ . Dadas dos secuencias u y v denotaremos $u \cdot v$ la secuencia resultante de concatenar u y v . Para la secuencia de longitud 0, es decir, la secuencia vacía usaremos la notación λ .

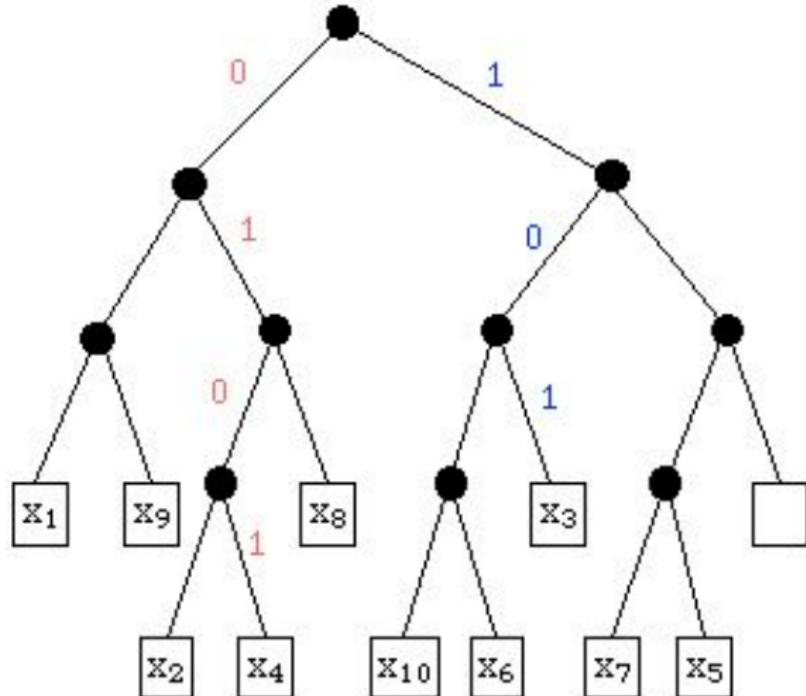
Definición

Dado un conjunto finito de secuencias $X \subset \Sigma^*$ de idéntica longitud, el *trie* T correspondiente a X es un árbol m -ario definido recursivamente de la siguiente manera:

- ① Si X contiene un sólo elemento o ninguno entonces T es un árbol consistente en un único nodo que contiene al único elemento de X o está vacío.
- ② Si $|X| \geq 2$, sea T_i el trie correspondiente a $X_i = \{y \mid x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$. Entonces T es un árbol m -ario constituido por una raíz \circ y los m subárboles T_1, T_2, \dots, T_m .

$m=2$

$x_1 = 000101$
 $x_2 = 010001$
 $x_3 = 101000$
 $x_4 = 010101$
 $x_5 = 110101$
 $x_6 = 100111$
 $x_7 = 110001$
 $x_8 = 011111$
 $x_9 = 001110$
 $x_{10} = 100001$



Lema

Si las aristas del trie T correspondiente a un conjunto X se etiquetan mediante los símbolos de Σ de tal modo que la arista que une la raíz con su primer subárbol se etiqueta σ_1 , la que une la raíz con su subárbol se etiqueta σ_2 , etc. entonces las etiquetas del camino que nos llevan desde la raíz hasta una hoja no vacía que contiene a x constituyen el prefijo más corto que distingue únicamente a x ; es decir, ningún otro elemento de X empieza con el mismo prefijo.

Lema

Sea p la etiqueta correspondiente a un camino que va de la raíz de un trie T hasta un cierto nodo (interno u hoja) de T . Entonces el subárbol enraizado en dicho nodo contiene todos los elementos de X que tienen en común al prefijo p (y no más elementos).

Lema

Dado un conjunto $X \subset \Sigma^$ de secuencias de igual longitud, su trie correspondiente es único. En particular T no depende del orden en que se “presenten” los elementos de X .*

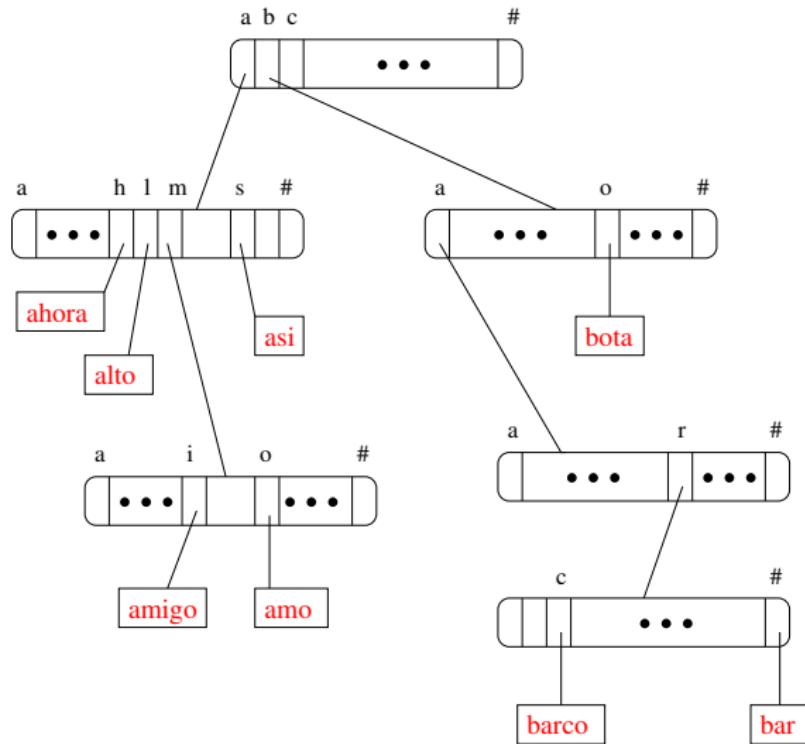
Lema

La altura de un trie T es igual a la longitud mínima de prefijo necesaria para distinguir cualesquiera dos elementos del conjunto al que corresponde el trie. En particular, si ℓ es la longitud de las secuencias en X , la altura de T será $\leq \ell$.

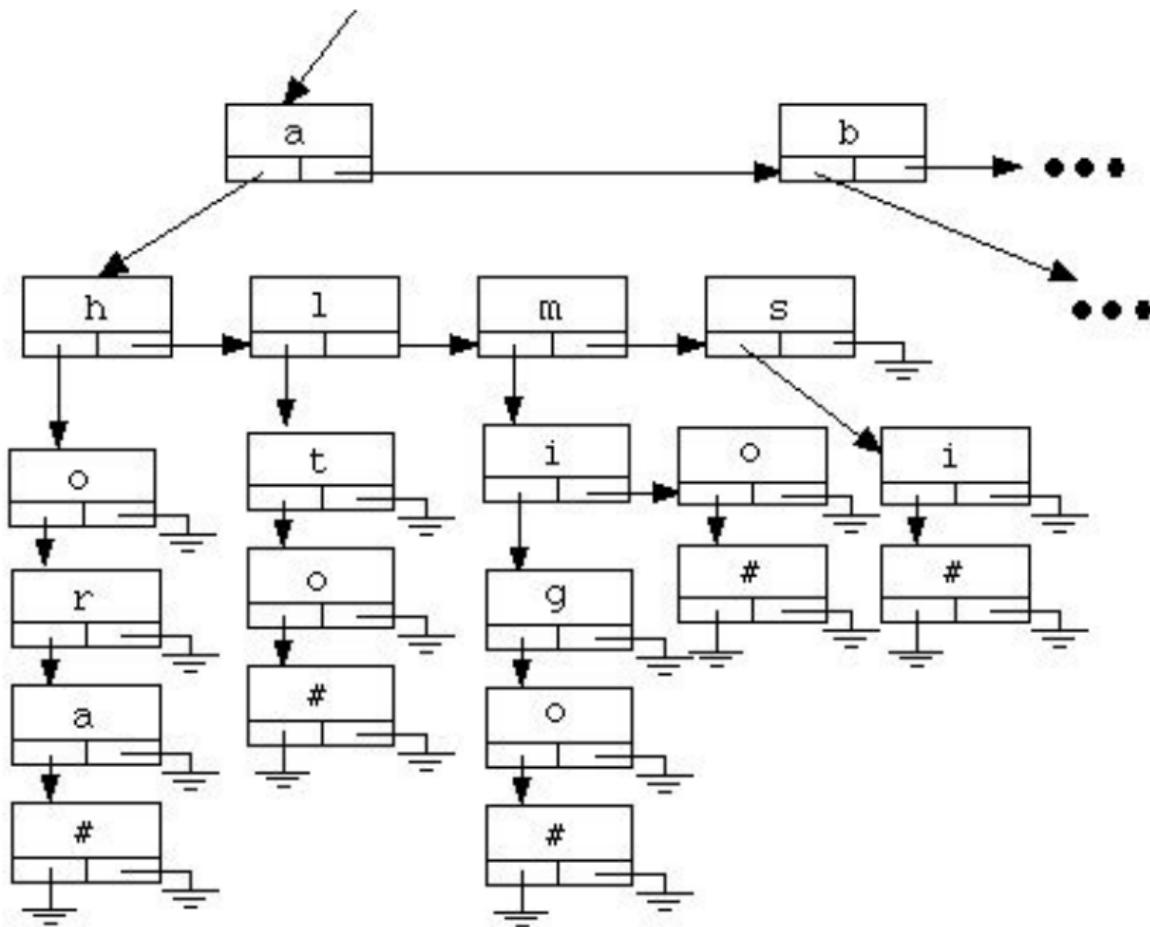
La definición de tries impone que todas las secuencias sean de igual longitud, lo cual es muy restrictivo. Pero si no exigimos esta condición entonces tenemos un problema que habremos de afrontar: si un elemento x es prefijo propio de otro elemento y , cómo podremos distinguirlo? Cómo diferenciamos las situaciones en la que x e y pertenecen ambos a X de las situaciones en la que sólo y está en X ?

Una solución habitual consiste en ampliar Σ con un símbolo especial (p.e. $\ddot{\#}$) de fin de secuencia, y marcar cada una de las secuencias en X con dicho símbolo. Ello garantiza que ninguna de las secuencias (marcadas) es prefijo propio de otra. El “precio” a pagar es que hay que trabajar con un alfabeto de $m + 1$ símbolos.

$X = \{ \text{ahora, alto, amigo, amo, asi, bar, barco, bota, ...} \}$



Las técnicas de implementación de los tries son las convencionales para árboles. Si se utiliza un vector de apuntadores por nodo, los símbolos de Σ suelen poderse utilizar directamente como índices (eventualmente, se habrá de utilizar una función $ind : \Sigma \rightarrow \{1, \dots, m\}$). Las hojas que contienen los elementos de X pueden almacenar exclusivamente los sufijos restantes, ya que el prefijo está ímplicitamente codificado en el camino desde la raíz a la hoja. En el caso en que se utilice la representación primogénito-siguiente hermano, cada nodo almacena un símbolo y sendos apuntadores al primogénito y al siguiente hermano. Puesto que suele haber definido un orden sobre Σ la lista de hijos de cada nodo suele ordenarse de acuerdo a áquel.



Aunque es más costoso en espacio emplear nodos del trie para representar las palabras completas, resulta ventajoso evitar la necesidad de nodos de distinto tipo, apuntadores a nodos de diferentes tipos, o representaciones poco eficientes para los nodos (p.e. *unions*).

```
// La clase Clave debe soportar las siguientes
// operaciones:
// x.length() devuelve la longitud >= 0 de una clave x
template <typename Clave>
int length() throw();

// x[i] devuelve el i-ésimo simbolo de x,
// lanza un error si i < 0 o i >= x.length()
template <typename Simbolo, typename Clave>
Simbolo operator[](const Clave& x, int i) throw(error);

template <typename Simbolo, typename Clave, typename Valor>
class DiccDigital {
public:
    ...
private:
    struct nodo_trie {
        Simbolo _c;
        nodo_trie* _primg;
        nodo_trie* _sigher;
        Valor _v;
    };
    nodo_trie* raiz;
    ...
};
```

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::busca(
    const Clave& k, bool& esta, Valor& v) const
    throw(error) {

    nodo_trie* p = busca_en_trie(raiz, k, 0);
    if (p == nullptr)
        esta = false;
    else {
        esta = true;
        v = p -> _v;
    }
}

// Pre: p es la raíz del subárbol conteniendo
// las claves cuyos i-1 primeros símbolos
// coinciden con los i-1 símbolos iniciales de k
// Coste: Θ(longitud(k))
template <typename Simbolo, typename Clave,
          typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_trie*
DiccDigital<Simbolo,Clave,Valor>::busca_en_trie(
    nodo_trie* p, const Clave& k,
    int i) const throw() {

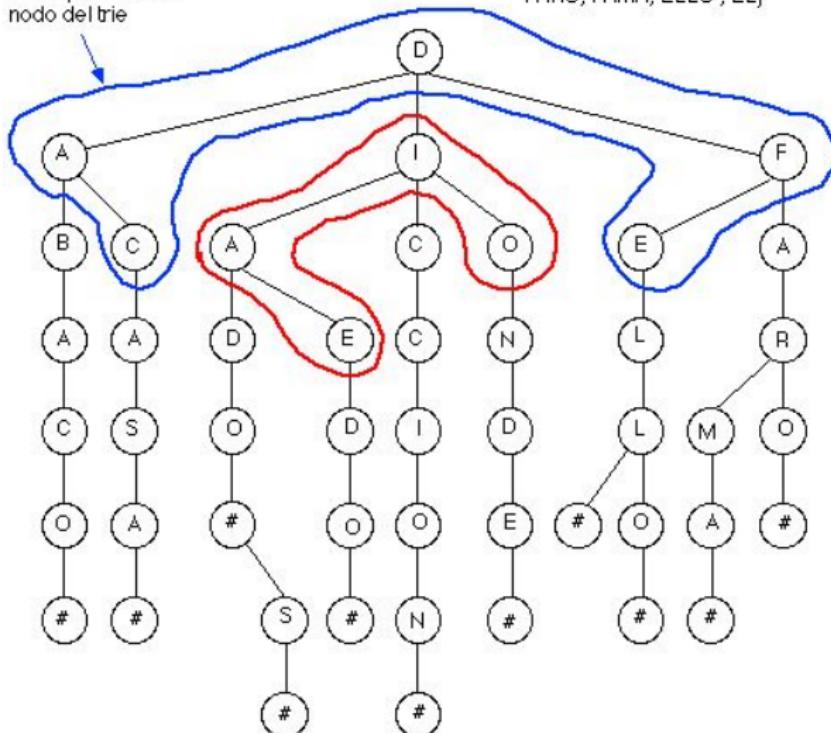
    if (p == nullptr)      return nullptr;
    if (i == k.length())  return p;
    if (p -> _c > k[i])  return nullptr;
    if (p -> _c < k[i])
        return busca_en_trie(p -> _sigher, k, i);
    // p -> _c == k[i]
    return busca_en_trie(p -> _primg, k, i+1);
}
```

Una solución que intenta combinar eficiencia en el acceso a los subárboles y en espacio consiste en implementar cada “nodo” del trie como un BST. La estructura resultante se denomina *árbol ternario de búsqueda* ya que cada uno de sus nodos contiene tres apuntadores: dos apuntadores al hijo izquierdo y derecho, respectivamente, en el BST, y un apuntador a la raíz del subárbol al que da acceso el nodo.

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
class DiccDigital {
public:
    ...
    void inserta(const Clave& k, const Valor& v)
        throw(error);
    ...
private:
    struct nodo_tst {
        Simbolo _c;
        nodo_tst* _izq;
        nodo_tst* _cen;
        nodo_tst* _der;
        Valor _v;
    };
    nodo_tst* raiz;
    ...
static nodo_tst* inserta_en_tst(nodo_tst* t,
                                int i, const Clave& k, const Valor& v)
    throw(error);
    ...
};
```

corresponde a un
nodo del trie

$X = \{\text{DICCION, DADO, DADOS, DEDO; DONDE, ABACO, CASA, FARO, FAMA, ELLO, EL}\}$



```
template <typename Simbolo,
          typename Clave,
          typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::inserta(
    const Clave& k, const Valor& v) throw(error) {
    // Simbolo() es un simbolo nulo,
    // p.e. si Simbolo==char entonces
    // Simbolo() == '\0'
    k[k.length()] = Simbolo(); // añadir centinela
                                // al final de la clave
    raiz = inserta_en_tst(raiz, 0, k, v);
}
```

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_tst*
DiccDigital<Simbolo,Clave,Valor>::inserta_en_tst(
    nodo_tst* t, int i,
    const Clave& k, const Valor& v)
throw(error) {

    if (t == nullptr) {
        t = new nodo_tst;
        t -> _izq = t -> _der = t -> cen = nullptr;
        t -> _c = k[i];
        if (i < k.length() - 1) {
            t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
        } else { // i == k.length() - 1; k[i] == Simbolo()
            t -> _v = v;
        }
    } else {
        if (t -> _c == k[i])
            t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
        if (k[i] < t -> _c)
            t -> _izq = inserta_en_tst(t -> _izq, i, k, v);
        if (t -> _c < k[i])
            t -> _der = inserta_en_tst(t -> _der, i, k, v);
    }
    return t;
}
```

La descomposición digital de las claves puede emplearse además de para la búsqueda para la ordenación. Los algoritmos de ordenación basados en estos principios se denominan de **ordenación digital** (ing: *radix sort*).

Consideremos un vector de n elementos cada uno de los cuales es una secuencia de ℓ bits. Dado un elemento x , $\text{bit}(x, i)$ denotará su i -ésimo bit.

Si ordenamos el vector con relación al bit de mayor peso, cada bloque resultante de acuerdo al bit de siguiente peso, y así sucesivamente, habremos ordenado el vector en su totalidad.

```
// Llamada inicial:  
// radix_sort(A, 0, A.size()-1, i{\color{red} \$ell-1\$});  
template <typename Elem, typename Symb>  
void radix_sort(vector<Elem>& A, int i, int j, int r) {  
  
    if (i < j and r >= 0) {  
        int k;  
        radix_split(A, i, j, r, k);  
        radix_sort(A, i, k, r - 1);  
        radix_sort(A, k + 1, j, r - 1);  
    }  
}
```

```
// bit(x, r) devuelve el bit r-ésimo de x
// r == 0 => bit de menor peso
// r == ;{\color{red}ell-1$}; => bit de mayor peso
template <typename Elem, typename Symb>
void radix_split(vector<Elem>& A, int i, int j,
                 int r, int& k) {
    int u = i; int v = j;
    while (u < v + 1) {
        while (u < v + 1 and bit(A[u],r) == 0) ++u;
        while (u < v + 1 and bit(A[v],r) == 1) --v;
        if (u < v + 1) swap(A[u], A[v]);
    }
    k = v;
}
```

Cada etapa de *radix sort* tiene un coste no recursivo lineal; puesto que el número de etapas es ℓ el coste del algoritmo es $\Theta(n \cdot \ell)$. Otra forma de deducir el coste consiste en considerar el coste asociado a cada elemento de A : un elemento cualquiera de A es examinado (y eventualmente intercambiado con otro) a lo sumo ℓ veces, de ahí que el coste total sea $\Theta(n \cdot \ell)$.

Parte V

Estructuras de Datos Avanzadas

- Tries
- Skip Lists

Skip lists

Una **skip list** es una estructura de datos muy sencilla que permite la implementación de un TAD diccionario con poco esfuerzo de manera eficiente. Las skip lists consiguen tener rendimiento $\Theta(\log n)$ en promedio en búsquedas y actualizaciones gracias al uso de aleatorización. Los costes promedio no dependen de que la entrada sea aleatoria o no: sólo de las decisiones aleatorias tomadas por los algoritmos. Las skip lists fueron inventadas por W. Pugh en 1989.

Una skip list S que representa a un conjunto de elementos X consiste en un cierto número de listas enlazadas no vacías, ordenadas por las claves de los elementos que contienen y numeradas de 1 en adelante, de modo que se satisface

- ① Todos los elementos de X pertenecen a la lista 1.
- ② Si x pertenece a la lista i entonces, con probabilidad q , x pertenece también a la lista $i + 1$.

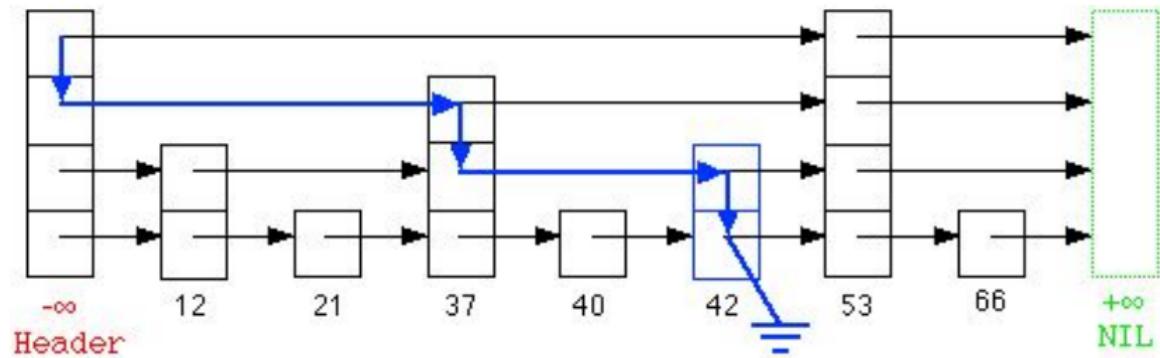
Dado un elemento x , su **nivel** es el número de listas en las que está incluído. De la definición anterior se desprende que el nivel de cada elemento es una variable aleatoria independiente y

$$\mathbb{P}[\text{nivel}(x) = i] = p \cdot q^{i-1}, \quad p = 1 - q.$$

Para implementar una skip list, cada elemento se almacenará en un nodo, que asimismo contendrá tantos apuntadores como correspondan al nivel del elemento. Cada uno de dichos apuntadores apunta al sucesor de x en la correspondiente lista. Adicionalmente, usaremos un nodo ficticio de cabecera con apuntadores a los primeros elementos de cada lista. El nivel de la skip list es el máximo nivel de entre sus elementos y éste será el número de apuntadores en el *header*.

```
template <typename Clave, typename Valor>
class Diccionario {
public:
    ...
private:
    struct nodo_skip_list {
        Clave _k;
        Valor _v;
        int _alt;
        nodo_skip_list** _sig;

        nodo_skip_list(Clave k, Valor v, int alt) :
            _k(k), _v(v), _alt(alt),
            _sig(new nodo_skip_list*[alt])) {
        }
    };
    nodo_skip_list* _header;
    int _nivel;
    double _p; // p.e., _p = 0.5
    ...
};
```



```
template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::busca(const Clave& k,
    bool& esta, Valor& v) const throw(error) {

    nodo_skip_list* p =
        buscar_en_skip_list(_header, _nivel-1, k);
    if (p == nullptr)
        esta = false;
    else {
        esta = true;
        v = p -> _v;
    }
}

template <typename Clave, typename Valor>
Diccionario<Clave,Valor>::nodo_skip_list*
Diccionario<Clave,Valor>::buscar_en_skip_list(
    nodo_skip_list* p,
    int l, const Clave& k) const throw() {

    while (l >= 0)
        if (p -> _sig[l] == nullptr ;$\\mid\\mid$; k <= p ->_sig[l] -> _k)
            --l;
        else
            p = p -> _sig[l];

    if (p -> _sig[0] == nullptr ;$\\mid\\mid$; p -> _sig[0] -> _k != k)
        // k no está
        return nullptr;
    else // k está, modificamos el valor asociado
        return p -> _sig[0];
}
```

Para realizar la inserción de un nuevo elemento se procede en cuatro fases:

- 1: Se busca en la skip list la clave k dada. Se emplea un bucle de búsqueda ligeramente distinto, para que anote en un vector de apuntadores los últimos nodos examinados en cada nivel. Dichos nodos son los potenciales predecesores del nuevo nodo. Basta anotar cuál es el último nodo visitado en el nivel ℓ antes de bajar de nivel.
- 2: Si la clave k ya existe se modifica el valor asociado y se termina.

```
template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::inserta_en_skip_list(...) {

    nodo_skip_list** pred = new nodo_skip_list*[l + 1];
    while (l >= 0)
        if (p -> _sig[l] == nullptr ;$\\mid\\mid$; k <= p ->_sig[l] -> _k) {
            pred[l] = p; // <===== anotar el predecesor
            --l;
        } else {
            p = p -> _sig[l];
        }
    if (p -> _sig[0] == nullptr ;$\\mid\\mid$; p -> _sig[0] -> _k != k)
        // k no está; añadimos un nuevo nodo ...
    else // k está, modificamos el valor asociado
        p -> _sig[0] -> _v = v;
}
```

- 3: En caso contrario, se crea un nuevo nodo con la clave y valor dados, utilizando procedimiento aleatorio para decidir el nivel r del elemento
- 4: Se enlaza el nuevo nodo en las r primeras listas

```
template <typename Clave, typename Valor>
class Diccionario {
public:
    ...
private:
    ...
    util::Random _rng; // generador de números aleatorios
                       // asociado a la skip list
};

template <typename Clave, typename Valor>
void Diccionario<Clave,Valor>::inserta_en_skip_list(...) {

    ...
    // añadir nuevo nodo
    // generar aleatoriamente su altura
    int alt = 1; while (_rng() > _p) ++alt;
    nodo_skip_list* nn = new nodo_skip_list(k, v, alt);
    if (alt > _nivel) {
        // añadir nuevos niveles al header
    }

    // enlazar el nuevo nodo en las listas
    // enlazadas pertinentes
    for (int i = alt - 1; i >= 0; --i) {
        nn -> _sig[i] = pred[i] -> _sig[i];
        pred[i] -> _sig[i] = nn;
    }
}
```

```
...
if (alt > _nivel) {
    // añadir nuevos niveles al _header y a pred
    // (desde i = _nivel hasta i = alt - 1)

    // nuevo header y nueva tabla pred
    nodo_skip_list** _new_header = new nodo_skip_list*[alt];
    nodo_skip_list** new_pred = new nodo_skip_list*[alt];

    // copiamos
    for (int i = _nivel - 1; i >= 0; --i) {
        _new_header -> _sig[i] = _header -> _sig[i];
        new_pred -> _sig[i] = pred -> _sig[i];
    }

    // los niveles desde _nivel a alt - 1 están vacíos
    for (int i = alt - 1; i >= _nivel; --i) {
        _new_header -> _sig[i] = nullptr;
        new_pred -> _sig[i] = nullptr;
    }

    // eliminamos el header y la tabla pred antiguas
    delete[] _header;
    delete[] pred;

    // actualizamos
    _header = _new_header;
    pred = new_pred;
    _nivel = alt;
}
...
```