Algoritmos y Estructuras de Datos III
Práctica: Algoritmo de Kruskal y estructuras de datos union-find

Guido Tagliavini Ponce

01/10/2014

1. Algoritmo de Kruskal

Recordemos que dado un grafo $G = (V, E)$ conexo y con pesos en las aristas, el algoritmo de Kruskal encuentra un árbol generador mínimo de G. El algoritmo visto en la Teórica es el siguiente:

```
1 KRUSKAL(G = (V, E))
2 begin
3     T = ∅
4     while |T| < |V| − 1 do
5         Tomar e ∈ E de peso mínimo entre los ejes que no forman un circuito simple con los ejes de T
6         T = T ∪ {e}
7     end
8     return T
9 end
```

Esta forma de enunciar el algoritmo está buena para entenderlo y probar que es correcto, pero no está tan buena para analizar su implementación y complejidad. Particularmente, la línea 5 enuncia un paso que no parece ser nada sencillo, y que podría ser descompuesto en otros pasos más elementales. ¿Cómo podemos desenrollar esa línea? Una forma sería tomar los ejes E en orden creciente por peso y, para cada uno, chequear que no formen un circuito simple con los ejes de T, quedándonos con el primero que cumpla esto. Además, si un eje es chequeado, no nos interesa considerarlo en futuras iteraciones del ciclo principal. Esto se debe a que a lo sumo agregamos una vez cada eje a T, y si al chequear un eje resulta que forma un circuito simple, entonces en futuras iteraciones seguirá formándolo.

Por otro lado, notemos que dado un eje (u, v), chequear si forma un circuito simple con los ejes de T equivale a chequear si u y v están en la misma componente conexa del grafo con ejes T.

Lema. *Sea $H = (V, T)$ un grafo. Sea (u, v) un eje que no está en T. Entonces al agregar (u, v) a H se forma un circuito simple si y sólo si u y v forman parte de la misma componente conexa de H.*

Demostración. ($\Rightarrow$) Si (u, v) forma un circuito simple en H, entonces tomando ese circuito y quitándole el eje (u, v) obtenemos un camino simple de u a v. Luego, u y v están en la misma componente conexa.

($\Leftarrow$) Si u y v están en la misma componente conexa de H, entonces existe un camino simple $C = \langle u, \ldots, v \rangle$ entre ellos. Como (u, v) no está en T, tampoco está en C, con lo cual $C \circ \langle u \rangle$ es un circuito simple en $H + (u, v)$. □

Basándonos en todo lo anterior, podemos reescribir el algoritmo de Kruskal como sigue.

```
1  KRUSKAL(G = (V, E))
2  begin
3      A = E
4      T = ∅
5      while |T| < |V| − 1 do
6          Tomar un eje (u, v) de A de peso mínimo
7          A = A − {(u, v)}
8          if u y v no están en la misma componente conexa de (V, T) then
9              T = T ∪ {(u, v)}
10         end
11     end
12     return T
13 end
```

Observemos que el algoritmo es greedy, en el sentido de que siempre agrega un eje de peso mínimo que no viole cierta condición.

1.1. Complejidad

El ciclo principal realiza, a lo sumo, $|E|$ iteraciones, puesto que en el peor de los casos chequea cada uno de los ejes de G. Calculemos el costo de cada iteración.

Las líneas 6 y 7 las podemos llevar a cabo ordenando previamente los ejes de G por peso creciente, por lo que, con un costo $O(|E| \lg |E|)$ pagado una única vez inicialmente, podemos llevar a cabo estas dos líneas con costo $O(1)$. Para ordenar los ejes de G respetando estos tiempos, debemos ser capaces de extraer los ejes del grafo con sus pesos, en forma eficiente. Una representación con listas de adyacencia permite reconocer todos los ejes en tiempo $O(|V| + |E|)$. Asumiendo que G es conexo, esta complejidad es $O(|E|)$, que es absorbida por el $O(|E| \lg |E|)$ del ordenamiento. Como $|E| \leq |V|^2$ entonces $O(\lg |E|) = O(\lg |V|^2) = O(\lg |V|)$ y, por ende, la complejidad del ordenamiento es $O(|E| \lg |V|)$.

No resulta tan fácil calcular la complejidad de la línea 8, ya que no está claro cómo realizar dicha operación. Una forma de determinar si u y v están en la misma componente conexa es vía DFS, que tiene costo lineal en el tamaño de la componente de alguno de u o v, por lo que resulta $O(|T|) = O(|V|)$. Con este approach la línea 9 es $O(1)$. Por lo tanto cada iteración cuesta $O(|V|)$ y, en total, Kruskal tiene un costo $O(|E| \lg |V| + |E||V|) = O(|E||V|)$.

Sin embargo, es posible hacer algo mejor que esto. Notemos que las dos operaciones principales de cada iteración son:

1. Determinar si u y v forman parte de la misma componente conexa de (V, T).
2. Agregar (u, v) a T y, por lo tanto, unir dos componentes conexas de (V, T).

En la implementación con DFS, nunca unimos explícitamente dos componentes conexas de (V, T), pues no lleva un registro de las evolución de (V, T). Esto tiene la desventaja de que cada vez que ejecuta la primera operación, el algoritmo debe recorrer el grafo para analizar su estructura y reconocer los cambios generados por el agregado de ejes. Por esta razón, esta solución padece un desbalance del costo de las dos operaciones.

Podemos intentar balancear los costos. Si pensamos a las componentes conexas como conjuntos disjuntos de vértices, entonces la primera operación se reduce a determinar si dos vértices forman parte del mismo conjunto, y la segunda operación equivale a unir dos conjuntos. Entonces buscamos algún tipo de estructura de datos eficiente, que mantenga conjuntos *disjuntos* de elementos, y que admita las operaciones:

1. Determinar si u y v forman parte del mismo conjunto.
2. Unir el conjunto que contiene a u con el que contiene a v.

Una estructura de este tipo se denomina *estructura de datos union-find*, puesto que debe ser capaz de unir conjuntos y determinar el conjunto al que pertenece un elemento. Notar que una estructura de este tipo sólo es adecuada en un contexto en el que los conjuntos involucrados son disjuntos ya que, en caso contrario, el conjunto al que pertenece un elemento podría no estar unívocamente determinado. Por esto es que también se las suele denominar *estructuras de datos para conjuntos disjuntos*.

Una tal estructura podría permitir obtener un mejor tiempo en el algoritmo de Kruskal. A continuación presentamos estructuras union-find, que estudiaremos en forma completamente independiente de este algoritmo.

2. Estructuras de datos union-find

Para comparar conjuntos resulta útil distinguir, en cada uno, un elemento que llamamos *representante*, de modo tal que dos conjuntos sean iguales si y sólo si su representante es el mismo.

En base a lo discutido, definimos las tres funciones de la interfaz de una estructura union-find:

- MAKE-SET(x). Crea el conjunto $\{x\}$.
- FIND-SET(x). Devuelve el representante del conjunto al que pertenece x.
- UNION(x, y). Une el conjunto que contiene a x con el que contiene a y.

2.1. Representación con listas

La primera estructura que vamos a considerar, representa cada conjunto con una lista. Cada elemento del conjunto es un nodo de la lista. Definimos, por simplicidad, al primer elemento de la lista como el elemento representante del conjunto. Cada nodo x posee un puntero $next[x]$ al siguiente elemento de la lista, y un puntero $set[x]$ al objeto lista (espacio en memoria que mantiene información de la misma) que contiene a x. Un objeto lista S posee un puntero $head[S]$ al primer nodo de la lista, y un puntero $tail[S]$ al último nodo.

La función MAKE-SET(x) simplemente crea una lista y agrega el elemento x. La función FIND-SET(x) devuelve el primer elemento de la lista de la que forma parte x. La función UNION(x, y) une los elementos de la lista de y al final de la lista de x.

```
1  MAKE-SET(x)
2  begin
3      Crear una lista S
4      head[S] = x
5      tail[S] = x
6      next[x] = NIL
7      set[x] = S
8  end
```

```
1  FIND-SET(x)
2  begin
3      return head[set[x]]
4  end
```

```
1  UNION(x, y)
2  begin
3      next[tail[set[x]]] = head[set[y]]
4      tail[set[x]] = tail[set[y]]
5      node = head[set[y]]
6      while node ≠ NIL do
7          set[node] = set[x]
8          node = next[node]
9      end
10 end
```

El costo de MAKE-SET y FIND-SET es $\Theta(1)$. El costo de UNION(x, y) es $\Theta(size(y))$, donde $size(y)$ es la cantidad de elementos del conjunto que contiene a y.

Observemos que en el caso que el conjunto de y sea mucho más grande que el de x, sería conveniente actualizar el atributo set de los elementos del conjunto de x, en lugar de los del conjunto de y. En general, resulta conveniente actualizar el set de los elementos del conjunto más pequeño. Esta simple mejora se denomina *heurística de unión balanceada*.

2.1.1. Heurística de unión balanceada

Para saber cuál conjunto conviene actualizar, debemos conocer el tamaño de cada uno de ellos. Computarlo tomaría tiempo lineal en el tamaño de la lista, y la heurística perdería todo el sentido. Por lo tanto, necesitamos mantener un atributo $size[S]$ con la cantidad de elementos de S. Las implementación de las funciones con estos cambios queda como ejercicio. La complejidad de UNION(x, y) pasa a ser $\Theta(\text{mín}\{size(x), size(y)\})$.

Lo interesante de esta heurística es la complejidad que logra, no en una operación individual, sino en una secuencia de operaciones de MAKE-SET, FIND-SET y UNION.

Teorema. *Usando una representación con listas, junto con la heurística de unión balanceada, una secuencia de* m *operaciones* MAKE-SET, FIND-SET *y* UNION, *de las cuales* n *son* MAKE-SET, *ejecutan en tiempo* $O(m + n \lg n)$.

Demostración. Como se ejecutan exactamente n MAKE-SET, hay n elementos involucrados.

Primero veamos que el costo de todas las operaciones UNION es $O(n \lg n)$. Para esto, podemos contar la cantidad de veces que un elemento actualiza su puntero set. Sea x un elemento cualquiera. Sabemos que cada vez que el puntero $set[x]$ es actualizado, x forma parte del conjunto más pequeño de los dos que son unidos. Entonces, la primera vez que x es actualizado, su conjunto tiene 1 elemento, y luego de la unión pasa a tener al menos 2 elementos. La segunda vez, su conjunto tiene al menos 2 elementos y luego de la unión pasa a tener al menos 4. En general, la k-ésima vez que $set[x]$ es actualizado, su conjunto pasa a tener, al menos, 2^k elementos. Como un conjunto tiene, a lo sumo, n elementos, entonces $2^k \leq n$, i. e., $k \leq \lg n$. Luego, $set[x]$ es actualizado $O(\lg n)$ veces. Como son exactamente n elementos, entonces el costo total de UNION es $O(n \lg n)$.

Como hay $O(m)$ operaciones MAKE-SET y FIND-SET, cada una con costo $O(1)$, entre los tres tipos de operaciones suman $O(m + n \lg n)$. □

En la demostración se puede ver que el costo que aporta la ejecución de todas las operaciones UNION es $O(n \lg n)$. Como comenzamos con n conjuntos y cada unión reduce la cantidad de conjuntos en uno, después de $n - 1$ operaciones UNION queda un único conjunto. Por ende, hay a lo sumo $n - 1$ uniones. Concluimos que el costo promedio de UNION, utilizando una unión balanceada, es $O(\lg n)$.

Comparemos este costo promedio, contra la versión sin balanceo. Consideremos la siguiente secuencia de $m = 2n - 1$ operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET:

MAKE-SET(x_1)
MAKE-SET(x_2)
⋮
MAKE-SET(x_n)
UNION(x_2, x_1)
UNION(x_3, x_2)
⋮
UNION(x_n, x_{n-1})

La i-ésima operación UNION tiene costo $\Theta(i)$, ya que une $\{x_{i+1}\}$ con $\{x_1, \ldots, x_i\}$, y todos los elementos de este último conjunto deben actualizar su atributo set. Entonces, el costo de todas las operaciones de unión es $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$. En promedio, resulta $\Theta(n)$, que es peor que el costo promedio $O(\lg n)$ de la versión balanceada.

2.2. Representación con bosques

En esta segunda estructura representamos cada conjunto como un árbol con raíz. El representante es, naturalmente, la raíz. Tradicionalmente, en una estructura de árbol, cada nodo tiene punteros a sus hijos. En esta representación de conjuntos disjuntos con bosques, cada nodo de cada árbol apunta únicamente a su padre, excepto la raíz que apunta a sí misma. Al padre de un elemento x lo mantenemos en el atributo $parent[x]$.

La función MAKE-SET(x) simplemente setea el atributo $parent[x]$ a sí mismo. La función FIND-SET(x) busca la raíz del árbol en el que se encuentra el elemento x. La función UNION(x, y) pone a la raíz del árbol que contiene a y como hijo de la raíz del árbol que contiene a x.

```
1 MAKE-SET(x)
2 begin
3     parent[x] = x
4 end
```

```
1 FIND-SET(x)
2 begin
3     if parent[x] = x then
4         return x
5     end
6     return FIND-SET(parent[x])
7 end
```

```
1 UNION(x, y)
2 begin
3     rx =FIND-SET(x)
4     ry =FIND-SET(y)
5     parent[ry] = rx
6 end
```

El costo de MAKE-SET es $\Theta(1)$. El costo de FIND-SET(x) es $\Theta(depth(x))$, donde $depth(x)$ es la profundidad del nodo x en su árbol. El costo de UNION(x, y) es $\Theta(depth(y))$.

El problema que tiene la estructura es que los árboles pueden desbalancearse completamente, haciendo que FIND-SET y UNION, que dependen de cuán alto sea el árbol, sean costosas. La solución es, como antes, incluir algún tipo de heurística que haga que el costo de las operaciones se mantenga estable.

2.2.1. Heurística de unión por altura

Para cada elemento x, mantenemos un atributo $height[x]$ con la altura del subárbol del cual x es raíz. Al unir dos conjuntos con representantes x e y, si $height[x] > height[y]$, ponemos a y como hijo de x, y la altura $height[x]$ no aumenta. Si $height[x] < height[y]$, x pasa a ser hijo de y, y la altura $height[y]$ no aumenta. Finalmente, si $height[x] = height[y]$ elegimos cualquiera de ellos como padre, ponemos al otro nodo como hijo, e incrementamos la altura del primero en una unidad.

Como afirma el siguiente resultado, esta estrategia logra balancear los árboles.

Teorema. *Usando una representación con bosques, junto con la heurística de unión por altura, a lo largo una secuencia de operaciones* MAKE-SET, FIND-SET *y* UNION, *vale que* $2^{height[x]} \leq size(x)$ *para toda raíz* x, *donde* $size(x)$ *es la cantidad de elementos del conjunto que contiene a* x.

Demostración. Veamos que al término de toda ejecución de cualquiera de las operaciones, vale la desigualdad. Como FIND-SET no modifica la estructura de los árboles, basta ver que vale para toda operación MAKE-SET y UNION. Es evidente que las operaciones MAKE-SET cumplen.

Usamos un argumento inductivo en la secuencia de operaciones UNION. La primera operación UNION es entre dos nodos aislados, y obviamente cumple la desigualdad. Supongamos que todas las operaciones UNION hasta cierto punto cumplen la desigualdad, y tomemos aquella inmediatamente posterior. Sean x e y los representantes de los conjuntos a unir. Por hipótesis inductiva, $2^{height[x]} \leq size(x)$ y $2^{height[y]} \leq size(y)$. Llamemos z al nodo raíz del árbol que se obtiene de la unión. Tenemos tres casos:

- Si $height[x] > height[y]$ entonces $z = x$, y

$$2^{height[z]} = 2^{height[x]} \leq size(x) < size(x) + size(y) = size(z)$$

- Si $\texttt{height}[x] < \texttt{height}[y]$ es análogo al caso anterior.
- Si $\texttt{height}[x] = \texttt{height}[y]$ entonces alguno de los dos nodos es elegido, arbitrariamente, como padre. Entonces

$$2^{\texttt{height}[z]} = 2^{\texttt{height}[x]+1} = 2 \cdot 2^{\texttt{height}[x]} \leq 2 \cdot \min\{\text{size}(x), \text{size}(y)\} \leq \text{size}(x) + \text{size}(y) = \text{size}(z)$$

□

Corolario. *Usando una representación con bosques junto con la heurística unión por altura, una secuencia de* m *operaciones* MAKE-SET, FIND-SET *y* UNION, *de las cuales* n *son* MAKE-SET, *ejecutan en tiempo* $O(m \lg n)$.

Demostración. Como hay exactamente n operaciones MAKE-SET, cada conjunto tiene a lo sumo n elementos. Cada operación FIND-SET y UNION tiene costo lineal en la altura del árbol. Por el teorema previo, esta altura es $O(\lg n)$. Luego, las $O(m)$ operaciones FIND-SET y UNION tienen costo $O(m \lg n)$.

Las n operaciones MAKE-SET tienen costo $O(1)$ cada una, por lo que suman un costo $O(n)$ al total. Como $n \leq m$, dicho costo se ve absorbido por el tiempo $O(m \lg n)$ de las otras dos operaciones. □

2.2.2. Heurística *path compression*

Cada vez que preguntamos por el representante del conjunto de un elemento x, recorremos el único camino desde x hasta la raíz del árbol del que forma parte. Esta estrategia consiste en, aprovechando el recorrido hacia la raíz, vincular directamente al representante con todos los nodos del camino, actualizándoles su atributo $\texttt{parent}$ como la raíz. En otras palabras, comprimimos los caminos desde todos esos nodos hacia la raíz. De este modo, las próximas llamadas a FIND-SET o UNION que involucren a un nodo de dicho camino, correrán más rápido.

Dado que esta heurística sólo se aplica durante una llamada a FIND-SET, podríamos intentar combinarla con la heurística de unión por altura, que sólo interviene en la operación UNION. El problema es que la heurística unión por altura cambia la estructura del árbol, haciendo que el atributo $\texttt{height}$ de ciertos nodos quede invalidado.

```
1 FIND-SET(x)
2 begin
3     if parent[x] ≠ x then
4         parent[x] = FIND-SET(parent[x])
5     end
6     return parent[x]
7 end
```

2.2.3. Heurística *union by rank*

En lugar de mantener un atributo $\texttt{height}$, que resulta demasiado rígido como para poder combinarlo con *path compression*, utilizamos una versión relajada del mismo. Para cada x, definimos $\texttt{rank}[x]$ como alguna cota superior para la altura del subárbol con raíz en x. Es decir, lo único que sabemos de este atributo es que $\texttt{height}[x] \leq \texttt{rank}[x]$. La forma de utilizar y actualizar $\texttt{rank}$ es la misma que $\texttt{height}$. Lo inicializamos en 0, y, en cada unión, la raíz con $\texttt{rank}$ más pequeño se cuelga de la otra. En caso de empate elegimos alguna como raíz de la unión, e incrementamos *en uno* su $\texttt{rank}$. Si bien, por definición, no habría inconveniente en incrementar el atributo en cualquier valor mayor que 1, el hacerlo sólo en una unidad permite preservar el balance.

```
1 MAKE-SET(x)
2 begin
3     parent[x] = x
4     rank[x] = 0
5 end
```

```
1  UNION(x, y)
2  begin
3      rx =FIND-SET(x)
4      ry =FIND-SET(y)
5      if rank[rx] < rank[ry] then
6          parent[rx] = ry
7      else
8          parent[ry] = rx
9          if rank[rx] = rank[ry] then
10             rank[rx] = rank[rx] + 1
11         end
12     end
13 end
```

Notemos que la heurística *union by rank* utilizada en solitario es igual a la heurística de unión por altura. La definición de rank permite combinar esta heurística con *path compression*, ya que que la compresión de caminos sólo achica alturas.

2.2.4. Combinando *union by rank* con *path compression*

¿Qué complejidad se obtiene combinando estas dos heurísticas? Calcularla no es nada fácil, y no lo haremos acá. Simplemente vamos a enunciar el resultado que indica que es mejor que las de las anteriores representaciones y heurísticas. Si les interesa, pueden encontrar la demostración en [1].

Teorema. *Usando una representación con bosques, junto con las heurísticas path compression y union by rank, una secuencia de* m *operaciones* MAKE-SET, FIND-SET *y* UNION, *de las cuales* n *son* MAKE-SET, *ejecutan en tiempo* $O(m\alpha(n))$.

¿Qué es $\alpha(n)$? Primero, presentamos a la famosa función de Ackermann, que se define como

$$A(m, n) = (2 \uparrow^{m-2} (n + 3)) - 3$$

La función α es la inversa de la anterior. Más precisamente,

$$\alpha(n) = \min\{k \in \mathbb{N}_0 : A(k, 1) \geq n\}$$

La flechita $\uparrow^m$ es una función que generaliza la suma, el producto, la exponenciación, etc. El producto xy no es más que y copias de x sumadas,

$$x \uparrow^0 y = \underbrace{x + \cdots + x}_{y \text{ veces}} = xy$$

La exponenciación x^y no es más que y copias de x multiplicadas,

$$x \uparrow^1 y = \underbrace{x \ldots x}_{y \text{ veces}} = x^y$$

Así siguiendo,

$$x \uparrow^2 y = \underbrace{x^{x^{\cdots^x}}}_{y \text{ veces}}$$

Notemos que $x \uparrow^1 y = x^y = x \cdot x^{y-1} = x \cdot (x \uparrow^1 (y - 1)) = x \uparrow^0 (x \uparrow^1 (y - 1))$. Esto explica la idea atrás de la definición formal:

$$x \uparrow^m y = \begin{cases} y + 1 & \text{si } m = -2 \\ x & \text{si } m = -1 \text{ e } y = 0 \\ 1 & \text{si } m \geq 0 \text{ e } y = 0 \\ x \uparrow^{m-1} (x \uparrow^m (y - 1)) & \text{si } m \geq -1 \text{ e } y > 0 \end{cases}$$

Todo esto debería hacernos una buena idea de que $A(m, n)$ crece tremendamente rápido. Algunos valores concretos son:

$A(0, 1) = 2$
$A(1, 1) = 3$
$A(2, 1) = 7$
$A(3, 1) = 2047$
$A(4, 1) \gg 10^{80}$

y 10^{80} es aproximadamente el número de átomos en el universo observable. Como A crece terriblemente rápido, α crece terriblemente lento:

$$\alpha(n) = \begin{cases} 0 & \text{si } 0 \leq n \leq 2 \\ 1 & \text{si } n = 3 \\ 2 & \text{si } 4 \leq n \leq 7 \\ 3 & \text{si } 8 \leq n \leq 2047 \\ 4 & \text{si } 2048 \leq n \leq A(4, 1) \end{cases}$$

Así que es poco probable que alguna vez en nuestras vidas consideremos algún valor de n para el que $\alpha(n) > 4$. Por esta razón decimos que $\alpha(n)$ es constante para cualquier valor práctico de n.

2.3. Conclusión

¿Qué estructuras de datos vimos? ¿Qué complejidades tienen? Hagamos un cuadro con las estructuras y los costos para m operaciones, de las cuales n son MAKE-SET.

Estructura \Heurística	Ninguna	Unión balanceada	Unión por altura	*Path compression + union by rank*
Listas	$O(mn)$	$O(m + n \lg n)$	-	-
Bosques	$O(mn)$	-	$O(m \lg n)$	$O(m\alpha(n))$

3. Tarea

Recomiendo hacer los siguientes ejercicios para entender todo lo tratado en este apunte.

1. Programar todas las versiones de union-find vistas.
2. Calcular la complejidad del algoritmo de Kruskal, para cada una de las versiones de union-find vistas.
3. Programar el algoritmo de Kruskal, con complejidad $O(|E| \lg |V|)$. Para verificar sus programas, pueden utilizar el siguiente juez online:

   `http://www.spoj.com/problems/MST/`
4. *Heurística de unión por tamaño*. Este ejercicio propone analizar otra heurística de unión para la representación con bosques. La misma consiste en mantener, para cada elemento x, un atributo $size[x]$ que indica la cantidad de nodos del subárbol con raíz en x. En cada unión entre los conjuntos de dos representantes x e y, si $size[x] > size[y]$, ponemos a y como hijo de x. Si $size[x] < size[y]$, ponemos a x como hijo de y. Finalmente, si $size[x] = size[y]$ elegimos cualquiera de ellos como padre y ponemos al otro nodo como hijo. En todos los casos incrementamos lo que corresponda al atributo $size$ del padre.

   Probar que esta estrategia mantiene los árboles balanceados. Es decir, probar que a lo largo de cualquier secuencia de operaciones, vale que $2^{height(x)} \leq size[x]$ para toda raíz x. Concluir que cada operación FIND-SET y UNION ejecuta en tiempo logarítmico en la cantidad total de elementos.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.