

UNIVERSIDAD NACIONAL DE INGENIERÍA

Estructuras de Datos

Heaps y colas de prioridad

Un árbol sin punteros, heapsort y compresión de Huffman

Gerson Garrido

Semana 13

Hoy

1. La cola de prioridad: solo importa el siguiente
2. La invariante del heap, más débil que la de un árbol de búsqueda
3. Un árbol completo, sin punteros
4. Insertar y subir; sacar y bajar
5. Construir de abajo hacia arriba: $\Theta(n)$
6. Heapsort
7. Los k mayores, y `std::priority_queue`
8. Huffman: códigos sin prefijos comunes
9. Un compresor de verdad: formato y archivos binarios

> **Lectura**
>
> Apuntes de la semana 13. Código en `code/week-13`.

SECCIÓN 01

La cola de prioridad

01



Solo importa el siguiente

Cola de prioridad *(priority queue)*

`push(x)`	agrega x
`top()`	devuelve el mínimo
`pop()`	saca el mínimo

Nada más: ni buscar, ni recorrer, ni pedir el segundo.

- Sale el de menor valor, no el que llegó antes.
- Lo que es "menor" lo decide un comparador.
- Un sistema operativo: el proceso más urgente.
- Dijkstra, en la semana 15: el lugar pendiente más cercano.

La pregunta de hoy

Si se pide menos que un árbol de búsqueda, ¿se puede gastar menos?

Lo que ya sabemos hacer

Implementación	`push`	`top`	`pop`
Arreglo sin ordenar	O(1) amortizado	Θ(n)	Θ(n)
Arreglo ordenado, mínimo al final	O(n)	O(1)	O(1)
Lista enlazada ordenada	O(n)	O(1)	O(1)
AVL o rojinegro	O(log n)	O(log n)	O(log n)
Heap binario	**O(log n)**, con `push_back`	**O(1)**	**O(log n)**

- Las tres ingenuas: n de cada operación cuestan $\Theta(n^2)$ en el peor caso.
- El árbol balanceado sirve, pero mantiene un orden total que nadie consultará, con un nodo por elemento.
- El heap: un arreglo, sin punteros por elemento.

SECCIÓN 02

La invariante del heap

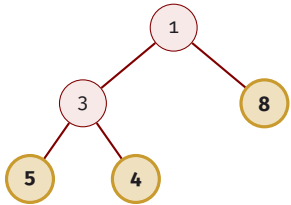
02



Min-heap

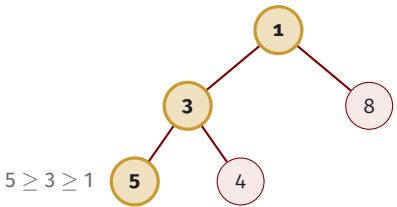
Min-heap

Un árbol binario **completo** en el que **cada padre es menor o igual que sus hijos**.



- Comparen con un árbol de búsqueda: allí está decidido el orden de *todas* las claves.
- Aquí, un nodo no dice nada de hermanos ni primos: el 8 está más arriba que el 5 y el 4.
- Las comparaciones solo valen a lo largo de un camino hacia la raíz.

¿Dónde está el mínimo?



- Tomen cualquier nodo, y suban hasta la raíz.
- En cada paso, el valor es menor o igual.
- Entonces la raíz es menor o igual que todos: **la raíz es un mínimo**.
- Con repetidos, otros pueden valer lo mismo. Nadie vale menos.

Por eso

`top` es O(1).

¿Y el máximo?

terminal

```
$ build/where_is_max
permutaciones: 5040, son heaps: 80
indice  minimo  maximo
     0  80      0
     1  0       0
     2  0       0
     3  0       20
     4  0       20
     5  0       20
     6  0       20
```

- Con claves distintas, un nodo con hijos no es el máximo: su hijo es mayor.
- El máximo está en **alguna** hoja, cualquiera.
- Con **repetidas**, un nodo interno puede valer lo mismo: en `1 2 2 2`, el índice 1.
- Hallarlo es $\Theta(n)$: hay que revisar las hojas.

Idea clave. Ordenado a medias: 80 heaps entre 5040 permutaciones, y solo una ordenada. Esa libertad lo hace barato.

Error común: buscar como en un árbol de búsqueda

terminal

```
$ build/heap_as_bst
busca 1: esta
busca 3: NO ESTA
busca 8: esta
busca 5: NO ESTA
busca 4: NO ESTA
```

- En la raíz, 3 > 1 lo manda a la derecha. El 3 está a la izquierda.
- El heap no dice de qué lado está nada.
- Buscar cuesta $\Theta(n)$ en el peor caso.
- Solo se poda si la clave es menor que el nodo: abajo nada vale menos.

Si necesitan buscar

Necesitan otra estructura.

SECCIÓN 03

Un árbol sin punteros

03

Completo: la altura es exacta

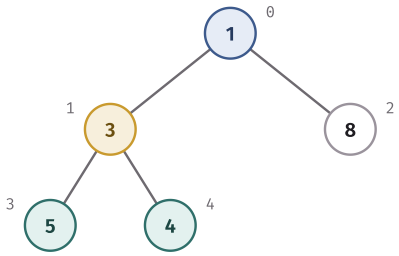
Árbol binario completo ***(complete)***

Todos sus niveles están llenos, salvo quizá el último, que se llena de izquierda a derecha.

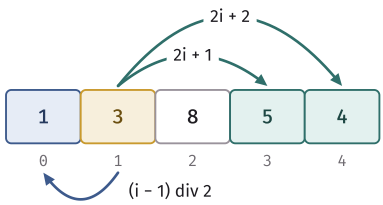
- Los niveles 0 a h $-$ 1 tienen $2^h - 1$ nodos, y el nivel h entre 1 y 2^h.
- Así que $2^h \leq n < 2^{h+1}$: **h =** $\lfloor \log_2 n \rfloor$.
- No es en promedio, ni con claves al azar. Para todo heap no vacío.
- Nada de rotaciones: la forma la decide solo n.

En un arreglo

ÁRBOL



ARREGLO



Con i = 1:

hijos en 2·1 + 1 = 3 y 2·1 + 2 = 4

padre en (1 − 1) div 2 = 0

Por qué esas cuentas

- Antes del hijo izquierdo de i: la raíz, y dos hijos por cada uno de los i nodos anteriores. Son 2i + 1.
- Es hoja si $i \geq \lfloor n/2 \rfloor$: **al menos la mitad son hojas**.

En C++

Con `size_t`, `(i - 1) / 2` con `i == 0` da un número enorme. Se detiene antes: `while (i > 0)`.

SECCIÓN 04

Insertar y sacar

04

Insertar: agregar al final y subir

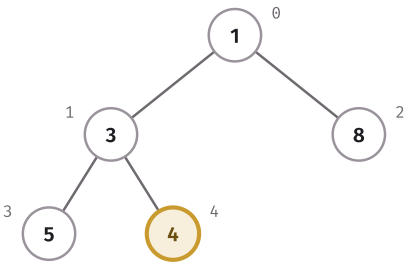
pseudocódigo

```
algoritmo insertar(arreglo, x)
    agregar x al final del arreglo
    i ← último índice
    mientras i > 0 hacer
        padre ← (i - 1) div 2
        si arreglo[padre] ≤ arreglo[i] entonces
            salir del ciclo
        fin si
        cambiar arreglo[padre] con arreglo[i]
        i ← padre
    fin mientras
fin algoritmo
```

- Solo hay un lugar que conserva la forma: el primer hueco del último nivel.
- El orden solo puede fallar entre el nuevo y su padre.
- Al cambiar, el problema sube un nivel. El hermano no tiene problema: ya era mayor o igual que el padre, y el que sube es menor.

Insertar, paso a paso: 5 3 8 1 4

ÁRBOL



ARREGLO

0	1	2	3	4
1	3	8	5	4

i = 4

padre = (4 − 1) div 2 = 1

push 4: entra al final, y 3 ≤ 4. No sube. El heap queda 1 3 8 5 4.

El código, y la secuencia completa

int_heap.h

```
1  void sift_up(std::size_t i) {
2      while (i > 0) {
3          std::size_t parent = (i - 1) / 2;
4          comparisons_++;
5          if (items_[parent] <= items_[i]) {
6              return;
7          }
8          std::swap(items_[parent], items_[i]);
9          i = parent;
10     }
11 }
```

terminal

```
$ build/heap_demo
push 5: [5]
push 3: [3 5]
push 8: [3 5 8]
push 1: [1 3 8 5]
push 4: [1 3 8 5 4]
pop  1: [3 4 8 5]
```

- Como máximo $\lfloor \log_2 n \rfloor$ comparaciones.
- El `push_back` es O(1) amortizado (semana 4).

Sacar: el último va a la raíz, y baja

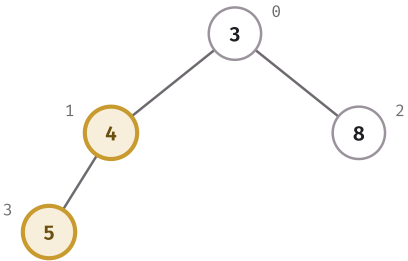
pseudocódigo

```
algoritmo sacar_minimo(arreglo)
    arreglo[0] ← último elemento
    quitar el último elemento
    i ← 0
    mientras verdadero hacer
        menor ← índice del menor entre i y sus hijos
        si menor = i entonces
            salir del ciclo
        fin si
        cambiar arreglo[i] con arreglo[menor]
        i ← menor
    fin mientras
fin algoritmo
```

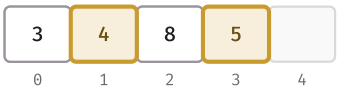
- La forma solo permite quitar la última hoja. Por eso va a la raíz.
- El orden solo puede fallar entre la raíz nueva y sus hijos.
- Se cambia con el **menor** de los hijos: quedará encima de su hermano.
- Dos comparaciones por nivel: como máximo $2\lfloor \log_2 n \rfloor$.

Sacar el mínimo de 1 3 8 5 4

ÁRBOL



ARREGLO



i = 1

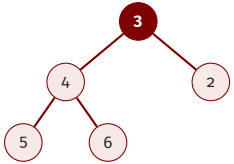
hijo: 2·1 + 1 = 3; el índice 4 ya no existe

Compara el 4 con su único hijo, el 5, y 4 ≤ 5: se queda. El heap queda 3 4 8 5.

Error común: cambiar siempre con el izquierdo

terminal

```
$ build/left_child_only
antes de sacar: es heap: si
arreglo: 3 4 2 5 6
raiz: 3, es heap: no
```



- El mínimo es 2, y `top` devolvería 3.
- Pasa pruebas pequeñas: solo falla si el derecho es el menor.
- Por eso `check_invariants` revisa cada nodo con su padre.
- Y por eso las pruebas al azar lo sacan todo y comparan con `std::priority_queue`.

SECCIÓN 05

Construir de abajo hacia arriba

05

Ya tengo los n elementos

- Con n inserciones: la cota dice O(n log n).
- **Las hojas ya son heaps.**
- El último padre está en $\lfloor n/2 \rfloor - 1$. Sus subárboles son heaps, y solo él puede estar mal.
- Baja. Luego el padre anterior, y así hasta la raíz.

Con `size_t`

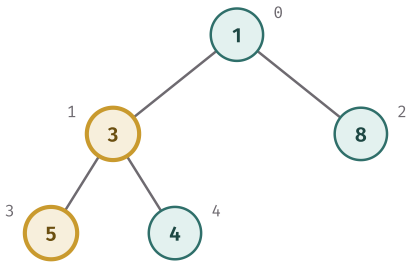
`for (size_t i = n/2 - 1; i >= 0; i--)` no termina nunca. Y con n = 0 o 1 ya arranca mal.

pseudocódigo

```
algoritmo construir(arreglo, n)
    i ← n div 2
    mientras i > 0 hacer
        i ← i - 1
        bajar el elemento de i
    fin mientras
fin algoritmo
```

Construir con 5 3 8 1 4

ÁRBOL



ARREGLO

1	3	8	5	4
0	1	2	3	4

altura	nodos	máximo	bajaron
2	1	2	2
1	1	1	1
0	3	0	0

Las hojas, que son la mayoría, no cuestan nada.

El 5 sigue bajando: tiene hijos 3 y 4, y llega al índice 3. Todo es un heap: 1 3 8 5 4.

Es el mismo arreglo que las 5 inserciones, pero esa coincidencia no es la regla.

Subir cuesta según la profundidad; bajar, según la altura

terminal

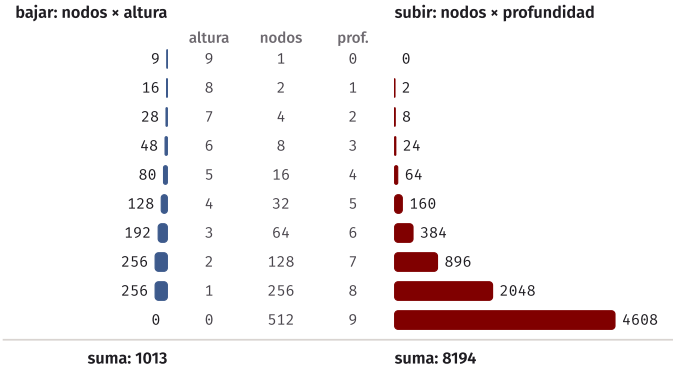
```
$ build/build_counts | tail -n 13
Arbol perfecto con n = 1023, entrada descendente
 altura  nodos  bajan como maximo
      0    512                  0
      1    256                256
      2    128                256
      3     64                192
      4     32                128
      5     16                 80
      6      8                 48
      7      4                 28
      8      2                 16
      9      1                  9
suma: 1013, niveles que bajaron de verdad: 1013
```

- La mitad no baja nada.
- Un cuarto baja 1 como máximo.
- El que puede bajar 9 es uno solo.
- La suma no llega a n.

Idea clave

Casi todos los nodos tienen mucha profundidad y poca altura. Se usa la operación barata para ellos.

Las dos sumas, nivel por nivel



n = 1023, un árbol perfecto, y las dos columnas en la misma escala

El nivel de las hojas tiene 512 nodos: al bajar no cuesta nada, y al subir es el más caro.

La suma por niveles, sin inducción

Con $n = 2^{k+1} - 1$, hay 2^{k-h} nodos de altura h:

$$\text{total} \leq \sum_{h=1}^{k} h \cdot 2^{k-h} = 2^k \sum_{h=1}^{k} \frac{h}{2^h}$$

Cada $h/2^h$ es $1/2^h$ repetido h veces:

$$\begin{array}{rrrrcl} \frac{1}{2}+ & \frac{1}{4}+ & \frac{1}{8}+ & \frac{1}{16}+\ldots & = & 1 \\ & \frac{1}{4}+ & \frac{1}{8}+ & \frac{1}{16}+\ldots & = & \frac{1}{2} \\ & & \frac{1}{8}+ & \frac{1}{16}+\ldots & = & \frac{1}{4} \\ & & & & \vdots & \\ \hline & & & & & 2 \end{array}$$

- Total $\leq 2^k \cdot 2 \leq n + 1$ niveles bajados.
- Dos comparaciones por nivel: unas 2n.
- Si no es perfecto, cabe en uno perfecto de menos del doble de nodos: sigue siendo O(n).
- Cada uno de los $\lfloor n/2 \rfloor$ padres compara por lo menos una vez.

Entonces

Construir cuesta Θ**(n)**.

Contado, con tres entradas

terminal

n	orden	abajo-arriba	/n	n pushes	/n	log2n	
16383	asc	16382	1.00	16382	1.00	14.0	
16383	azar	30692	1.87	37086	2.26	14.0	
16383	desc	32738	2.00	196610	12.00	14.0	
262143	asc	262142	1.00	262142	1.00	18.0	
262143	azar	492980	1.88	597088	2.28	18.0	
262143	desc	524250	2.00	4194306	16.00	18.0	
1048575	asc	1048574	1.00	1048574	1.00	20.0	
1048575	azar	1973024	1.88	2393134	2.28	20.0	
1048575	desc	2097110	2.00	18874370	18.00	20.0	

- **Abajo-arriba:** entre 1 y 2 por elemento, en todas las filas.
- **n pushes, descendente:** cada uno sube hasta la raíz. Crece con log n: es el peor caso.
- **n pushes, al azar:** unas 2.3 por elemento, sin crecer. $\Theta(n \log n)$ es del **peor caso**, no de toda entrada. Esto se mide; no lo demostramos.

SECCIÓN 06

Heapsort

06



Ordenar sin memoria extra

pseudocódigo

```
algoritmo heapsort(arreglo, n)
    construir un heap con el mayor en la raíz
    fin ← n
    mientras fin > 1 hacer
        cambiar arreglo[0] con arreglo[fin - 1]
        fin ← fin - 1
        bajar el elemento de 0, sin pasar de fin
    fin mientras
fin algoritmo
```

- Usa el **mayor** en la raíz. Sacarla deja el hueco al final, justo en su lugar definitivo.

terminal

```
$ build/heapsort_demo | head -n 7
entrada:   [4 1 7 3 9 2 |]
heap:      [9 4 7 3 1 2 |]
saca 9:    [7 4 2 3 1 | 9]
saca 7:    [4 3 2 1 | 7 9]
saca 4:    [3 1 2 | 4 7 9]
saca 3:    [2 1 | 3 4 7 9]
saca 2:    [1 | 2 3 4 7 9]
```

Invariante: a la izquierda de la barra, un heap. A la derecha, lo ya ordenado, mayor o igual que todo el heap.

O(n log n), pero no Θ para toda entrada

terminal

```
$ build/heapsort_counts
        n  entrada  comparaciones    /n  /(n log2 n)
   100000     azar       3018834  30.19         1.82
   100000      asc       3112517  31.13         1.87
   100000     desc       2926640  29.27         1.76
   100000  iguales        299994   3.00         0.18
  1000000     azar      36793121  36.79         1.85
  1000000      asc      37692069  37.69         1.89
  1000000     desc      36001436  36.00         1.81
  1000000  iguales       2999994   3.00         0.15
```

- O(n log n) en el **peor caso**: mergesort necesita memoria, y quicksort puede llegar a n^2.
- O(1) de memoria extra.
- Con todas las claves iguales, nada baja: unas 3n.
- Decir "Θ(n log n) siempre" sería falso.

No es estable

terminal

```
$ build/heapsort_demo | tail -n 2
entrada: 2a 2b 1c
salida:  1c 2b 2a
```

- Los dos 2 salieron en el orden contrario.
- Los cambios largos, de la raíz al final, pasan por encima de claves iguales.
- El índice original como desempate lo arregla, pero ya no es en el lugar.

Salta por toda la memoria

terminal

```
$ build/heapsort_times
       n   heapsort  std::sort  stable_sort  heap/sort
  100000     0.0054     0.0056       0.0060       0.97
  200000     0.0128     0.0115       0.0130       1.11
  400000     0.0295     0.0246       0.0279       1.20
  800000     0.0690     0.0525       0.0586       1.31
 1600000     0.1555     0.1095       0.1239       1.42
 3200000     0.3668     0.2280       0.2610       1.61
```

Medianas de 5 corridas, `-O2`, sin sanitizers, en arm64. Los tiempos dependen de la máquina.

- Bajar va de i a 2i + 1: cada paso duplica el índice.
- Con 100 000 empatan. Con 3.2 millones, entre 1.6 y 1.9 veces `std::sort`, en tres corridas.
- Las comparaciones no lo explican: 1.82 y 1.85 veces n log n.
- ¿La cache? Probablemente, pero aquí no hay `perf`.

Lo que sí se puede contar: un modelo de cache

terminal

```
$ build/heapsort_cache
        n  algoritmo     accesos       fallos  fallos/n
    10000   heapsort      384749         2123      0.21
    10000  mergesort      403636        17529      1.75
   100000   heapsort     4843204       426879      4.27
   100000  mergesort     4966873       276756      2.77
  1000000   heapsort    58341042      8949218      8.95
  1000000  mergesort    58715769      3469588      3.47
  4000000   heapsort   257368630     46618943     11.65
  4000000  mergesort   258862982     14982452      3.75
```

Modelo: 32 KB, líneas de 64 bytes, cada línea en un único lugar. No es la cache de esta máquina.

- Casi los mismos accesos.
- Con pocos datos, heapsort falla **menos**.
- Con muchos, sus fallos por elemento crecen con n, y los de mergesort apenas.
- Concuerda con el reloj. No demuestra que sea la causa.

En libstdc++, `std::sort` cambia a heapsort si la recursión se vuelve muy profunda: es una red de seguridad. Eso es de esta implementación, no del estándar.

SECCIÓN 07

Los k mayores

07



Los k mayores, con un heap de tamaño k

pseudocódigo

```
algoritmo k_mayores(secuencia, k)
    heap ← heap vacío con el menor en la raíz
    para cada x en secuencia hacer
        si tamaño(heap) < k entonces
            insertar x en heap
        sino si k > 0 y x > raíz(heap) entonces
            sacar la raíz de heap
            insertar x en heap
        fin si
    fin para
    sacar todo de heap, y devolverlo al revés
fin algoritmo
```

- La raíz es **el peor de los buenos**: hay que vencerlo para entrar.
- Si x no lo vence, ya hay k que valen por lo menos lo mismo.
- O(n log k) en el peor caso, y O(k) de memoria.
- Sirve con datos que van llegando.
- **Error común:** con el mayor en la raíz, no se sabe a quién descartar.

Pagar solo por lo que se pide

terminal

```
$ build/top_k_bench
n = 2000000
       k  comp. heap  comp. sort  t. heap  t. sort
      10     2000548    50340211   0.0014   0.1373
    1000     2100616    50340211   0.0014   0.1381
  100000     9261431    50340211   0.0301   0.1376
 2000000    43849815    50340211   0.1853   0.1370
```

Claves al azar. Medianas de 3 corridas, `-O2`. Los tiempos dependen de la máquina.

- k = 10: 25 veces menos comparaciones, y unas 100 veces menos tiempo.
- Casi no depende de k si es pequeño: al azar, muy pocos vencen a la raíz. Con una entrada ascendente, todos entran.
- Con k = n, **pierde**, aunque compara menos. Si piden casi todo, ordenen.

std::priority_queue

terminal

```
$ build/pq_demo
std::less:    8 5 4 3 1
std::greater: 1 3 4 5 8
make_heap: 164961 comparaciones, cota 3n = 300000
pop_heap, peor de 1000: 18, cota 2 log2 n = 33
push_heap, peor de 1000: 16, cota log2 n = 16
```

- Solo `push`, `pop`, `top`, `empty`, `size`.
- Las cotas son del estándar. Que `pop_heap` use 18 es de libstdc++.

Convenio

El comparador dice quién va **después**. Con `std::less`, `top` da el **mayor**.

En su `MinHeap`, `less(a, b)` dice que a va **más cerca** de la raíz: al revés.

En Dijkstra, sin `std::greater`, salen primero los más lejanos.

SECCIÓN 08

Códigos de Huffman

08

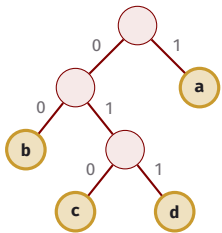
Códigos sin prefijos comunes

- Un texto usa 8 bits por byte, aunque `e` sea mucho más frecuente que `z`.
- Longitud variable: a = 0, b = 01, c = 1. ¿Qué dice `01`? ¿`b`? ¿`ac`?

Sin prefijos comunes *(prefix-free code)*

Ningún código es prefijo de otro. No hacen falta separadores.

¿Por qué? Ser prefijo sería pasar por la hoja de otro, y por una hoja no pasa nada.



Decodificar: baja con cada bit. Al llegar a una hoja, escribe y vuelve a la raíz.

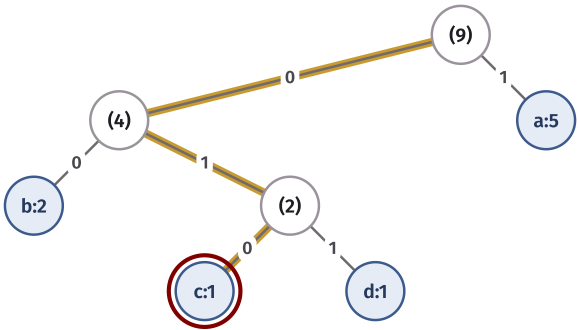
El algoritmo: unir los dos más livianos

pseudocódigo

```
algoritmo huffman(frecuencias)
    heap ← vacío
    para cada símbolo s con frecuencia mayor que 0 hacer
        insertar en heap una hoja con s y su frecuencia
    fin para
    mientras tamaño(heap) > 1 hacer
        a ← raíz(heap)
        sacar la raíz de heap
        b ← raíz(heap)
        sacar la raíz de heap
        insertar un nodo con hijos a y b, y pesos sumados
    fin mientras
    devolver la raíz de heap
fin algoritmo
```

- Costo en bits: la suma de $f(s) \cdot d(s)$.
- Los frecuentes arriba, los raros abajo.
- k símbolos: O(k log k), con $k \leq 256$.
- **Es óptimo**: se demuestra en Algoritmos.
- Empates: otros árboles, los mismos bits. Nuestro formato fija el desempate.

a: 5, b: 2, c: 1, d: 1



CÓDIGOS

s	f	código	f·bits
a	5	1	5
b	2	00	4
c	1	010	3
d	1	011	3

total: 15 bits

BITS LEÍDOS

0 1 0 → **c**

El bit 0 baja a la izquierda y llega a una hoja. Se escribe c, y se vuelve a la raíz.

Cuántos bits gana

- $5 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 =$ **15 bits**, frente a 18 con un código fijo.
- También: $2 + 4 + 9 = 15$. Cada unión agrega un bit a todo su árbol.
- Con frecuencias iguales, no se gana nada.

SECCIÓN 09

Un compresor de verdad

09

El formato

bytes 0-3	`HUF1`
bytes 4-11	tamaño original, 64 bits primero el byte menos significativo
Corriente de bits, si el tamaño no es 0:	
árbol en preorden	interno: bit 0 e hijos hoja: bit 1 y 8 bits
códigos	de cada byte, en orden
relleno	ceros, hasta completar el byte

- Los símbolos son **bytes**: funciona con cualquier archivo. Una `ñ` son dos.
- ¿Árbol o frecuencias? Con frecuencias, hay que desempatar igual al volver, y son hasta 2048 bytes.
- El árbol con 256 símbolos: $255 + 256 \cdot 9 = 2559$ bits, unos 320 bytes.
- Preorden: cada bit dice qué sigue.

El archivo más pequeño: `aab`

terminal

```
0  1 01100010  1 01100001  1 1 0  00
|  |           |           |      relleno
|  hoja b      hoja a      a a b
nodo interno
```

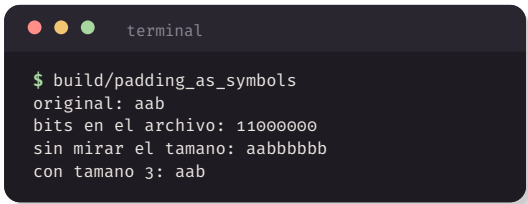
De a 8: `01011000`, `10101100` y `00111000`, es decir, `58 AC 38`.

terminal

```
$ printf aab > build/aab.txt
$ build/solution/compress build/aab.txt build/aab.huf
$ xxd build/aab.huf
00000000: 4855 4631 0300 0000 0000 0000 58ac 38    HUF1........X.8
```

15 bytes para guardar 3: la cabecera solo se recupera en archivos más grandes.

El tamaño original evita tres errores



1. El relleno. Esos ceros también decodifican.

2. Archivo vacío. Tamaño 0: sin árbol, ni símbolos. 12 bytes.

3. Un solo símbolo. Árbol de una hoja, con código de **0 bits**. 14 bytes, para cualquier tamaño.

Error común

Leer un bit *antes* de mirar si ya está en una hoja. Con una sola hoja, baja al hijo `-1`, y lo atrapa ASan. Mire primero, y este caso no necesita código aparte.

Leer un archivo binario sin perder bytes

terminal

```
$ build/stop_at_ff
con char y EOF: 3 bytes de 7
con >>:         5 bytes de 7
con int y EOF:  7 bytes de 7
```

Compilado con `-fsigned-char`. Bytes: `a`, espacio, `b`, `0xFF`, `c`, salto de línea, `d`.

`read_file` viene hecho: con `std::istreambuf_iterator`, que entrega cada byte tal cual.

- `get()` devuelve `int`. En un `char` con signo, `0xFF` vale -1, igual que `EOF`.
- » se salta los espacios, también en modo binario.
- El signo de `char` **depende de la plataforma**: con signo en Linux x86-64, sin signo en Linux arm64, donde ese primer ciclo no termina.
- Por eso los bytes son `std::uint8_t`.
- `std::ios::binary` no cambia nada en Linux. Ninguna prueba puede notar que falte; en Windows sí importa.

Cuánto comprime

terminal

archivo	original	comprimido	%
empty.bin	0	12	-
one_byte.bin	1	14	1400.0
single_symbol.bin	100000	14	0.0
all_bytes.bin	256000	256332	100.1
random.bin	1048576	1048908	100.0
special.bin	100000	29991	30.0
tradiciones.txt	285895	163986	57.4
terminos.tsv	317775	183507	57.7
palabras.txt	12943052	6969665	53.8

Tamaños exactos: no dependen de la máquina.

- Texto en español: 57 %, unos 4.6 bits por byte.
- `random.bin` crece 332 bytes: 12 de cabecera y 320 de árbol.
- Un byte se vuelve 14: la cabecera domina.

Ningún compresor achica todo

Hay 2^n archivos de n bits, y menos de 2^n más cortos. Dos tendrían la misma salida.

Frente a `gzip`

```
terminal

$ gzip -9n -c ../../data/tradiciones.txt | wc -c
118634
$ gzip -9n -c ../../data/palabras.txt | wc -c
2519173
```

	Huffman	gzip
`tradiciones.txt`	57.4 %	41.5 %
`palabras.txt`	53.8 %	19.5 %

- `gzip` también usa Huffman, pero antes reemplaza frases repetidas por una referencia a su copia.
- `palabras.txt`: `amar`, `amara`, `amaran`… Huffman solo ve bytes sueltos.
- Comprimir lo ya comprimido da 100.3 %: los bytes parecen al azar.

SECCIÓN 10

Laboratorio

10

Compresión de Huffman

Sesión 1:

1. `MinHeap<T, Less>`, contra `std::priority_queue`. **Guárdenlo:** va en Dijkstra.
2. Medir: construir frente a insertar.
3. `top_k` y `top_words`, frente a ordenar.

Sesión 2:

4. El árbol, con su heap, a partir de las frecuencias.
5. El árbol en el archivo, `compress` y `decompress`.
6. `cmp`, con binarios, vacíos y de un símbolo. Medir.

Vienen hechos: `count_words`, `read_file`, `write_file`, `BitWriter`, `BitReader`, `count_frequencies`, `make_codes`, y los `main` de los dos programas. Las pruebas cuentan comparaciones y copias, y comparan bytes exactos.

terminal

```
make test-heap
make bench
make test-top
make bench-top
make test-tree
make test-compress
make roundtrip
make bench-compress
make test-lab
```

Para la próxima clase

- Leer los apuntes de la semana 14: grafos, BFS, DFS y union-find.
- Laboratorio: `MinHeap` y el compresor. Dibujen el árbol de `aab` antes de escribir `write_tree`.
- Guarden su `min_heap.h`: la semana 15 lo usa en Dijkstra.

Pregunta

¿Qué le falta a un heap para que el programa encuentre, en un mapa, el lugar pendiente más cercano? ¿Qué harían si la distancia de un lugar que ya está en el heap mejora?