

Códigos de Huffman



Códigos de Huffman

- Los códigos de Huffman comprimen los datos de manera muy eficaz: comúnmente ahorran del 20% al 90%, dependiendo de las características de los datos que se comprimen.
- Se considera que los datos deben ser una secuencia de caracteres.
- El algoritmo voraz de Huffman utiliza una tabla que da con qué frecuencia ocurre cada carácter (es decir, su frecuencia) para construir una forma óptima de representar cada carácter como una cadena binaria.

Código Binario

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	0	1	0	0

Bit más significativo

Bit menos significativo

1 carácter = 1 byte = 8 bits

- Supongamos que tenemos un archivo de datos de 100.000 caracteres que deseamos almacenar de forma compacta.
- Observamos que los caracteres del archivo ocurren con las frecuencias dadas por la **Figura 1**.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Es decir, solo aparecen 6 caracteres diferentes, y aparece el carácter **a** 45.000 veces.
- Tenemos muchas opciones sobre cómo representar dicho archivo de información.

- Aquí, consideramos el problema de diseñar un código binario de caracteres (o código para abreviar) en el que cada carácter está representado por una cadena binaria única, que llamamos **codeword** (palabra clave).
- Si usamos un código de longitud fija, necesitamos 3 bits para representar 6 caracteres:
 $a = 000, b = 001, c = 010, d = 011, e = 100, f = 101.$
- Este método requiere 300.000 bits para codificar el archivo completo. ¿Podemos hacerlo mejor?

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Un archivo de datos de 100.000 caracteres contiene solo los caracteres **a** – **f**, con las frecuencias indicadas.
- Si asignamos a cada carácter una palabra de código de 3 bits, podemos codificar el archivo en 300.000 bits.
- Usando el código de longitud variable que se muestra, podemos codificar el archivo solo en 224.000 bits.

- Un código de longitud variable puede funcionar considerablemente mejor que un código de longitud fija, porque:
 - Da a los caracteres frecuentes palabras de código cortas.
 - Y a los caracteres poco frecuentes asigna palabras de código largas.
- La **figura 1** muestra dicho código; aquí la cadena de 1 bit 0 representa el carácter **a**, y la cadena de 4 bits 1100 representa al carácter **f**.

- Este código requiere:

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$

- para representar el archivo, un ahorro de aproximadamente un 25%.
- De hecho, este es un código de caracteres óptimo para este archivo.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100
	0	10	110	1110	11110	11111



Códigos de prefijo




— Códigos de prefijo

- Consideramos aquí solo los códigos en los que ningún **codeword** es también un prefijo de algún otro **codeword**.
- Estos códigos se denominan códigos de prefijo.
- Un código de prefijo siempre puede lograr la compresión de datos óptima entre cualquier código de caracteres, por lo que no sufrimos pérdida de generalidad al restringir nuestra atención al códigos de prefijos.

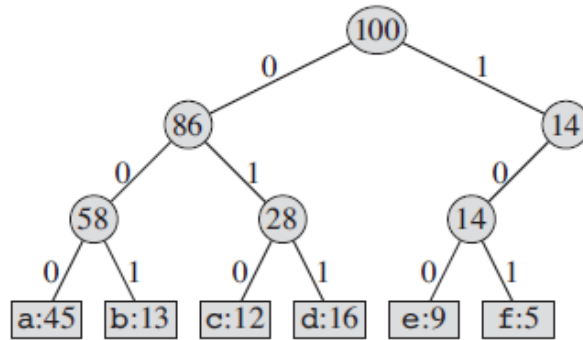
- La codificación es siempre sencilla para cualquier código de caracteres binario; simplemente concatenamos el **codeword** que representa cada carácter del archivo.
- Por ejemplo, con la longitud variable del código de prefijo de la Figura 1, codificamos el archivo de 3 caracteres **abc** como $0 \cdot 101 \cdot 100 = 0101100$, donde “.” denota concatenación.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

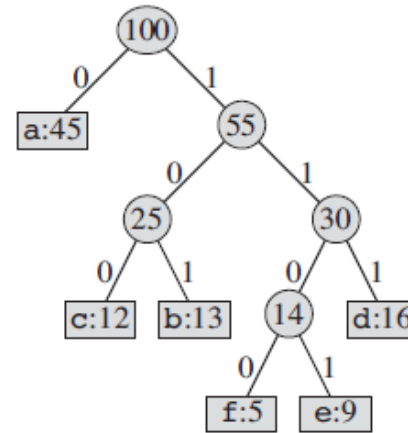
- Los códigos de prefijo son deseables porque simplifican la decodificación.
- Dado que ningún **codeword** es un prefijo de cualquier otro, el **codeword** que comienza un archivo codificado no es ambiguo.
- Simplemente podemos identificar el **codeword** inicial, traducirlo al carácter original y repetir el proceso de decodificación en el resto del archivo codificado.
- En nuestro ejemplo, la cadena 001011101 se analiza de forma única como $0 \cdot 0 \cdot 101 \cdot 1101$, que se decodifica como: **aabe**.

- 
- El proceso de decodificación necesita una representación conveniente para el código de prefijo, por lo que podemos seleccionar fácilmente el codeword inicial.
 - Un árbol binario cuyas hojas son los caracteres dados proporcionan una de esas representaciones.
 - Interpretamos el codeword binario para un carácter como la ruta simple desde la raíz hasta ese carácter, donde 0 significa "ir al hijo de la izquierda" y 1 significa "ir al hijo de la derecha".

- Tenga en cuenta que estos no son árboles binarios de búsqueda, ya que las hojas no necesitan aparecer en orden ordenado y los nodos internos no contienen claves de caracteres.



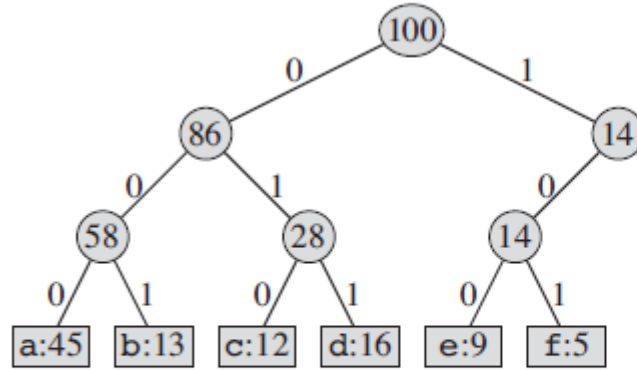
(a)



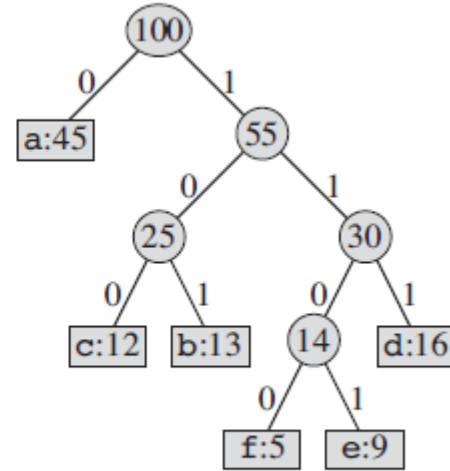
(b)

La siguiente figura muestra los árboles para los dos códigos de nuestro ejemplo.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a)




(b)

Figura 2. Árboles correspondientes a los esquemas de codificación de la Figura 1.

- Cada hoja está etiquetada con un carácter y su frecuencia de aparición.
- Cada nodo interno está etiquetado con la suma de las frecuencias de las hojas en su subárbol.

(a) El árbol correspondiente al código de longitud fija $a = 000, \dots, f = 101$.

(b) El árbol correspondiente al código de prefijo óptimo $a = 0, b = 101, \dots, f = 1100$.

- 
- Un código óptimo para un archivo siempre está representado por un árbol binario completo, en el que cada nodo no hoja tiene dos hijos.
 - El código de longitud fija en nuestro ejemplo no es óptimo ya que su árbol, que se muestra en la Figura 2 (a), no es un árbol binario completo: ya que contiene palabras de código que comienzan con 10... pero ninguno a partir del 11....

- Desde ahora podemos restringir nuestra atención a árboles binarios completos, podemos decir que si C es el alfabeto del que se extraen los caracteres y todas las frecuencias de caracteres son positivas, entonces el árbol para un código de prefijo óptimo tiene exactamente $|C|$ hojas, una para cada letra del alfabeto, y exactamente $|C| - 1$ nodos internos.

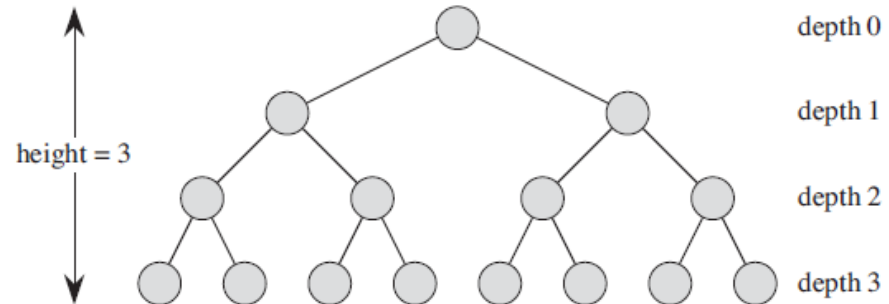
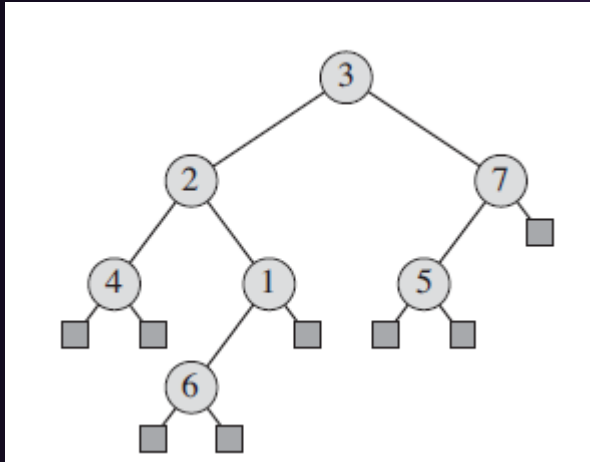


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

- Dado un árbol T correspondiente a un código de prefijo, podemos calcular fácilmente el número de bits necesarios para codificar un archivo.
- Para cada carácter c del alfabeto C , deje que el atributo $c.freq$ denote la frecuencia de c en el archivo y sea $d_T(c)$ quien denote la profundidad de la hoja de c en el árbol.
- Tenga en cuenta que $d_T(c)$ es también la longitud del **codeword** para el carácter c .
- Por tanto, el número de bits necesarios para codificar un archivo es:

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

- Que definimos como el costo del árbol T .




Construyendo el código de Huffman



Construyendo el código de Huffman

- Huffman inventó un algoritmo voraz que construye un código de prefijo óptimo llamado *código de Huffman*.
- En el pseudocódigo que sigue, asumimos que C es un conjunto de n caracteres y que cada carácter $c \in C$ es un objeto con un atributo $c.freq$ dando su frecuencia.
- El algoritmo construye el árbol T correspondiente al código óptimo de abajo hacia arriba.

- 
- Comienza con un conjunto de hojas $|C|$ y realiza una secuencia de $|C| - 1$ operaciones de "fusión" para crear el árbol final.
 - El algoritmo usa una cola de prioridad mínima Q , afinada en el atributo *freq*, para identificar los dos objetos menos frecuentes y fusionarlos.
 - Cuando fusionamos dos objetos, el resultado es un nuevo objeto cuya frecuencia es la suma de las frecuencias de los dos objetos que se fusionaron.

Código de Huffman

HUFFMAN(C)

1 $n = |C|$

2 $Q = C$

3 for $i = 1$ to $n - 1$

4 allocate a new node z


5 $z.left = x = \text{EXTRACT-MIN}(Q)$

6 $z.right = y = \text{EXTRACT-MIN}(Q)$

7 $z.freq = x.freq + y.freq$

8 INSERT(Q, z)

9 return EXTRACT-MIN(Q) // return the root of the tree

- 
- Como el alfabeto contiene 6 letras, el tamaño de la cola inicial es $n = 6$ y 5 pasos de combinación para construir el árbol.
 - El árbol final representa el código de prefijo óptimo.
 - El **codeword** para una letra es la secuencia de etiquetas en la ruta desde la raíz hasta la letra.
 - La línea 2 inicializa la cola de prioridad mínima Q con los caracteres en C.

- El bucle *for* en las líneas 3-8 extrae repetidamente los dos nodos *x* y *y* de menor frecuencia de la cola, reemplazándolos en la cola con un nuevo nodo *z* que representa su fusión.
- La frecuencia de *z* se calcula como la suma de las frecuencias de *x* y *y* en la línea 7.
- El nodo *z* tiene a *x* como hijo izquierdo y a *y* como hijo derecho. (Este orden es arbitrario; cambiar el hijo izquierdo y derecho de cualquier nodo produce un código diferente de el mismo costo.)
- Después de *n - 1* fusiones, la línea 9 devuelve el nodo que queda en la cola, que es la raíz del árbol de código.

- Cabe mencionar que el algoritmo produciría el mismo resultado si elimináramos las variables x y y en las líneas 5 y 6.
- Asignando los resultados directamente a $z.left$ y $z.right$ y cambiando la línea 7 a $z.freq = z.left.freq + z.right.freq$
- EXTRACT-MIN(S) – Remueve y retorna el elemento de S con la clave más pequeña.

```
HUFFMAN(C)
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8      INSERT( $Q, z$ )
9  return EXTRACT-MIN( $Q$ )    // return the root of the tree
```

(a) f:5 e:9 c:12 b:13 d:16 a:45

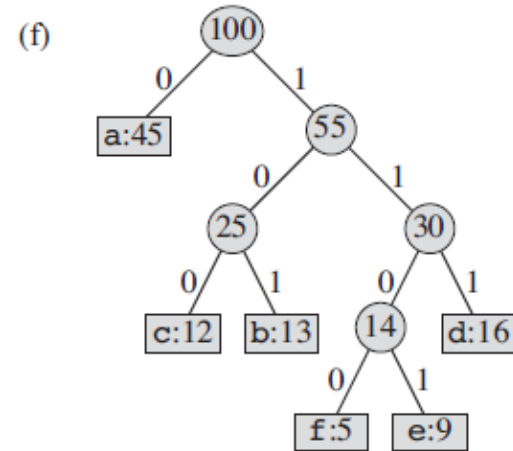
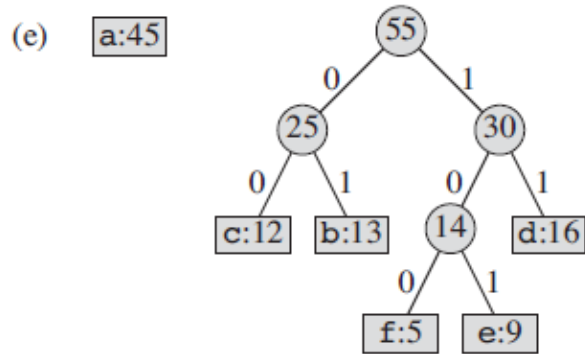
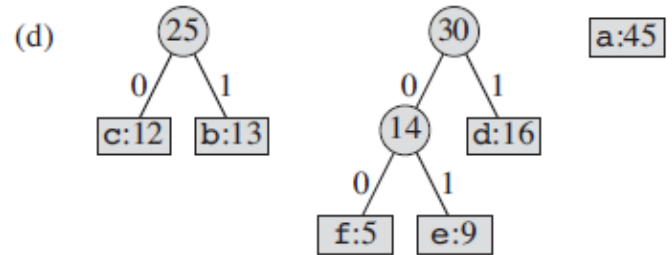
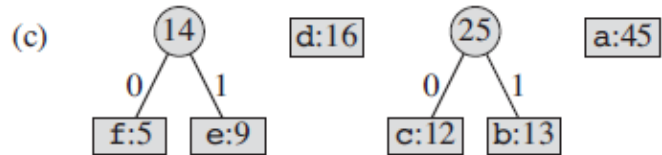
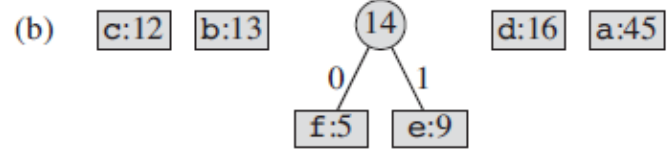


Figura 3. Los pasos del algoritmo de Huffman.

Figura 3. Los pasos del algoritmo de Huffman para las frecuencias de ejemplo.

- ❖ Cada parte muestra el contenido de la cola ordenada en orden creciente por frecuencia.
- ❖ A cada paso, los dos los nodos con frecuencias más bajas se fusionan.
- ❖ Las hojas se muestran como rectángulos que contienen un carácter y su frecuencia.
- ❖ Los nodos internos se muestran como círculos que contienen la suma de las frecuencias de sus hijos.
- ❖ Una arista que conecta un nodo interno con sus hijos se etiqueta como 0 si es un hijo a la izquierda y 1 si es un hijo derecho.
- ❖ El **codeword** de una letra es la secuencia de etiquetas en las aristas que conectan la raíz con la hoja de esa letra.

(a) El conjunto inicial de $n = 6$ nodos, uno para cada letra.

(b) - (e) Etapas intermedias.

(f) El árbol final.

Complejidad del código de Huffman

- Para analizar el tiempo de ejecución del algoritmo de Huffman, asumimos que Q está implementado como un min-heap binario.
- Para un conjunto C de n caracteres, podemos inicializar Q en la línea 2 en $O(n)$ usando el procedimiento BUILD-MIN-HEAP.
- El ciclo for en las líneas 3 a 8 se ejecuta exactamente $n - 1$ veces, y
- dado que cada operación del montículo requiere un tiempo $O(\lg n)$, el ciclo contribuye en $O(n \lg n)$ al tiempo de ejecución.

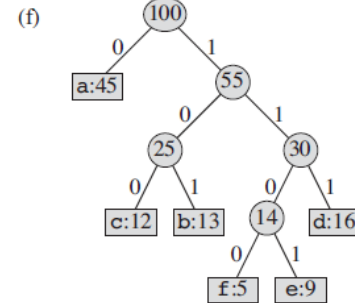
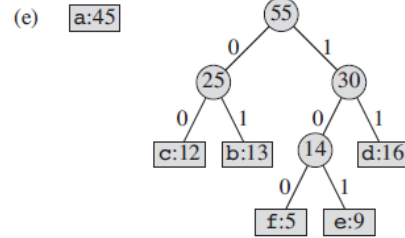
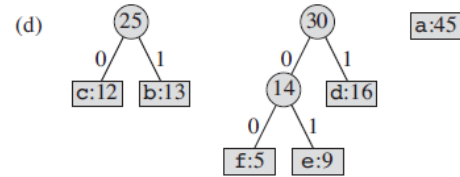
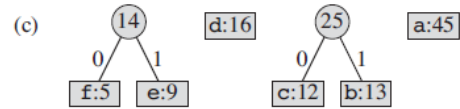
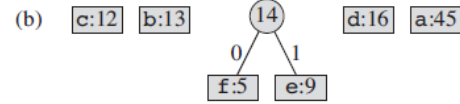
- Por tanto, el tiempo de ejecución total de HUFFMAN de un conjunto de n caracteres es $O(n \lg n)$.
- Podemos reducir el tiempo de ejecución a $O(n \lg \lg n)$ reemplazando el min-heap binario con un árbol de van Emde Boas.

— Ejercicio – Códigos de Huffman

- ¿Cuál es un código de Huffman óptimo para el siguiente conjunto de frecuencias basado en los primeros 8 números de Fibonacci.

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

(a) f:5 e:9 c:12 b:13 d:16 a:45



$$B(T) = \sum_{c \in C} c \cdot \text{freq} \cdot d_T(c)$$

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

