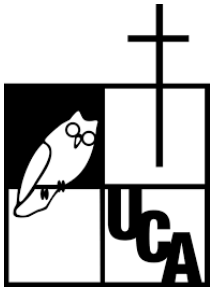
UNIVERSIDAD CENTROAMERICANA JOSÉ SIMEÓN CAÑAS



Reporte Etapa III

Ing. Jorge Alfredo López Sorto

Integrantes:

William Josué Pineda Martínez **00225919**

Grupo:

#10
Binary Fetch

Fecha de entrega: 23 de noviembre del 2023.

Algoritmo de huffman
Historia

Fue desarrollado por David A. Huffman mientras era estudiante de doctorado en el MIT, y publicado en "A Method for the Construction of Minimum-Redundancy Codes".

En 1951, a David Huffman y a sus compañeros de clase de la asignatura “Teoría de la Información” se les permitió optar entre la realización de un examen final o la presentación de un trabajo. El profesor Robert. M. Fano asignó las condiciones del trabajo bajo la premisa de encontrar el código binario más eficiente. Huffman, ante la imposibilidad de demostrar qué código era más eficiente, se rindió y empezó a estudiar para el examen final. Mientras estaba en este proceso vino a su mente la idea de usar árboles binarios de frecuencia ordenada y rápidamente probó que éste era el método más eficiente.

Con este estudio, Huffman superó a su profesor, quien había trabajado con el inventor de la teoría de la información Claude Shannon con el fin de desarrollar un código similar. Huffman solucionó la mayor parte de los errores en el algoritmo de codificación Shannon-Fano. La solución se basaba en el proceso de construir el árbol de abajo a arriba en vez de al contrario.

El algoritmo de Huffman es un algoritmo de compresión de datos desarrollado por David A. Huffman en 1952. Se utiliza para comprimir datos, como texto o archivos, de manera eficiente al asignar códigos de longitud variable a diferentes símbolos en función de su frecuencia de aparición. Los símbolos más frecuentes reciben códigos más cortos, lo que resulta en una comprensión más efectiva.

Aquí está cómo funciona el algoritmo de Huffman:

1. Análisis de frecuencia: El primer paso consiste en analizar el texto o los datos que se van a comprimir para determinar la frecuencia de aparición de cada símbolo (por ejemplo, caracteres en un archivo de texto). Se crea una tabla de frecuencias que registra la cantidad de veces que cada símbolo aparece en los datos.
2. Creación del árbol de Huffman: A continuación, se crea un árbol binario de Huffman utilizando la tabla de frecuencias. El árbol se construye de abajo hacia arriba, comenzando con nodos hoja para cada símbolo y fusionándose gradualmente en nodos padres. Los nodos se organizan de manera que los símbolos más frecuentes están más cerca de la raíz del árbol, y los símbolos menos frecuentes están más lejos.
3. Asignación de códigos: A medida que se construye el árbol de Huffman, se asignan códigos binarios a cada símbolo en función de su posición en el árbol. Los símbolos se etiquetan con códigos de longitud variable, con los símbolos más frecuentes teniendo códigos más cortos. Los códigos se asignan de manera que no haya ninguna ambigüedad en la decodificación, lo que significa que ningún código sea un prefijo de otro.
4. Compresión: Una vez que se ha construido el árbol de Huffman y se han asignado los códigos, se utiliza este árbol para comprimir los datos originales. Cada símbolo se reemplaza por su código correspondiente antes de almacenar o transmitir los datos comprimidos.

5. Decodificación: Para descomprimir los datos, se utiliza el mismo árbol de Huffman y se recorre el árbol para convertir los códigos binarios en símbolos originales.

La codificación de Huffman se utiliza a menudo en algún otro método de compresión. Como la deflación y códec multimedia como JPEG y MP3 que tienen una cuantificación digital basada en la codificación de Huffman.

Ejemplo

La tabla describe el alfabeto a codificar, junto con las frecuencias de sus símbolos. En el gráfico se muestra el árbol construido a partir de este alfabeto siguiendo el algoritmo descrito.

A = 0.15	B =0.30	C = 0.2	D = 0.05	E = 0.15	F = 0.05	G = 0.10

se escogen los valores de min valor

A = 0.15	B = 0.30	C = 0.2	0.10	E = 0.15	G = 0.10

A = 0.15	B = 0.30	C = 0.2	0.20	E = 0.15

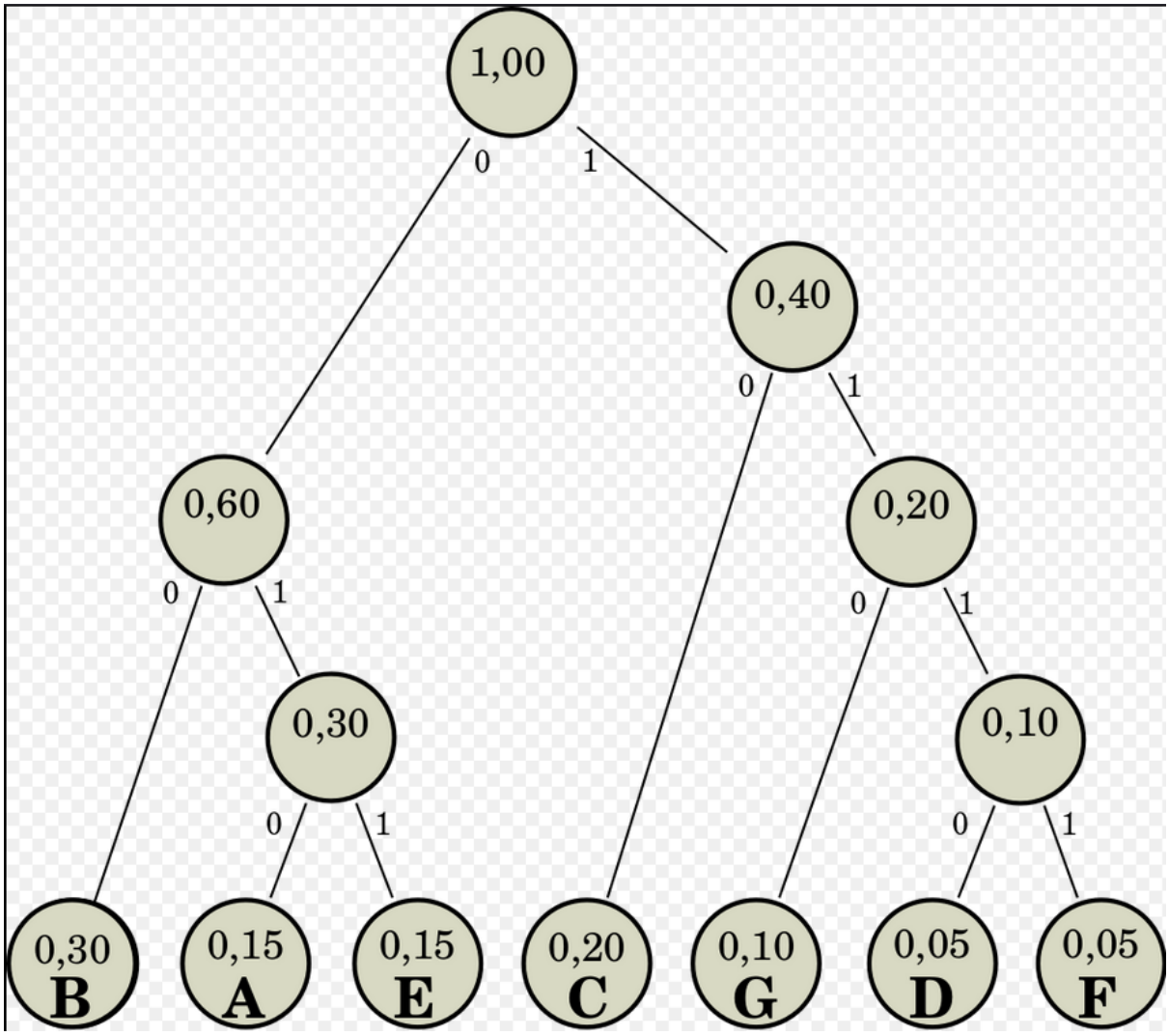
A = 0.15	B = 0.30	C = 0.2	0.10	E = 0.15	G = 0.10

A = 0.15	B = 0.30	C = 0.2	0.20	E = 0.15

0.6	0.4

1

< == raíz del árbol huffman



Codificación de Huffman

La técnica funciona creando un árbol binario de nodos Un nodo puede ser un nodo hoja o un nodo interno. Inicialmente, todos los nodos son nodos hoja, que contienen el carácter en sí, el peso (frecuencia de aparición) del carácter. Los nodos internos contienen peso de carácter y enlaces a dos nodos secundarios. Como convención común, un poco `0` representa seguir al hijo izquierdo, y un poco `1` representa seguir al hijo correcto. Un árbol terminado tiene `n` nudos de hojas y `n-1` nodos internos. Se recomienda que Huffman Tree descarte los caracteres no utilizados en el texto para producir las longitudes de código más óptimas.

Se puede ver con facilidad cuál es el código del símbolo **E**: subiendo por el árbol se recorren ramas etiquetadas con **1**, **1** y **0**; por lo tanto, el código es **011**.

Para obtener el código de **D** se recorren las ramas **0**, **1**, **1** y **1**, por lo que el código es **1110**.

La operación inversa también es fácil de realizar: dado el código **10** se recorren desde la raíz las ramas **1** y **0**, obteniéndose el símbolo **C**.

Para descodificar **010** se recorren las ramas **0**, **1** y **0**, obteniendo el símbolo **A**.

Pseudocódigo algoritmo de Huffman

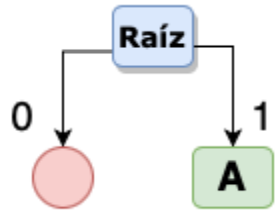
Ejemplo 2

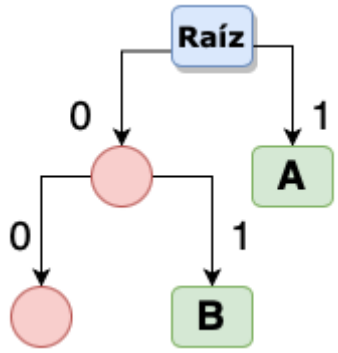
Cadena de texto original: AADCBABCDABCCDABABDCA

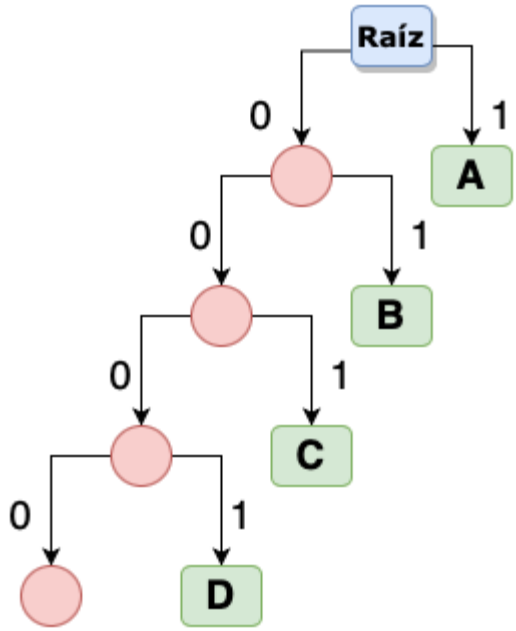
Construyendo un árbol binario

Letra	Frecuencia
A	7
B	5
C	5
D	4

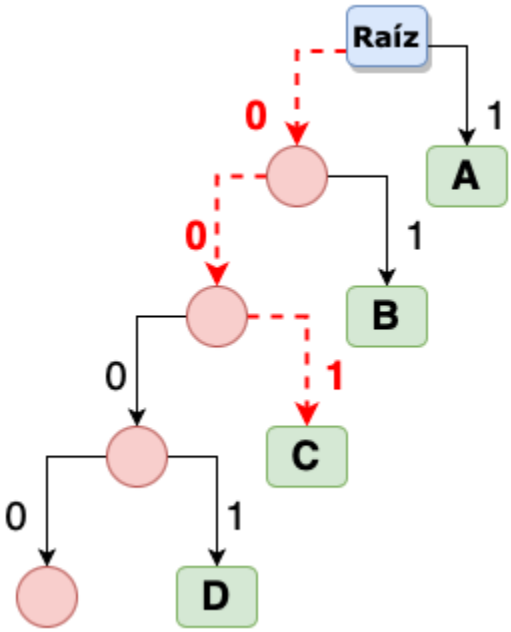
Ahora vamos construir un árbol binario que para crear una ruta para acceder a cada una de las palabras de nuestra cadena, para ello nuestro árbol tendrá una raíz y a partir de ahí derivamos 2 ramas, las ramas de la izquierda tendrán siempre el valor de cero y las ramas de la derecha el valor de 1. Siempre vamos a colocar en las ramas de la derecha la palabra con mayor frecuencia de repetición, en este caso, la letra ‘A’ es la que se repite más veces, por lo tanto la primer aproximación de nuestro árbol es la siguiente:







Recorriendo el árbol



La letra C estará representada solo por 3 bits y no por 8 bits. Los bits que representan a ‘C’ son 001, resultado de recorrer nuestro árbol para llegar a ella. Para la letra ‘A’ solo hace falta hacer un simple recorrido y obtendremos 1 como su representación binaria, es decir, redujimos de 8 bits a 1 bit para representar a la letra ‘A’.

La representación binaria del resto de letras es la siguiente:

Letra	Frecuencia	Representación
A	7	1
B	5	01
C	5	001
D	4	0001

La cadena de caracteres original es:
AADCBABCDABCCDABABDCA

En código binario es:
01000001010000010100010001000011010000100100000101000010010000110100010001000010000010100001001000001010000100100010001000011010000

Sin embargo, ahora con la nueva representación, resultado de nuestro árbol binario, nuestra cadena de bits queda reducida así:

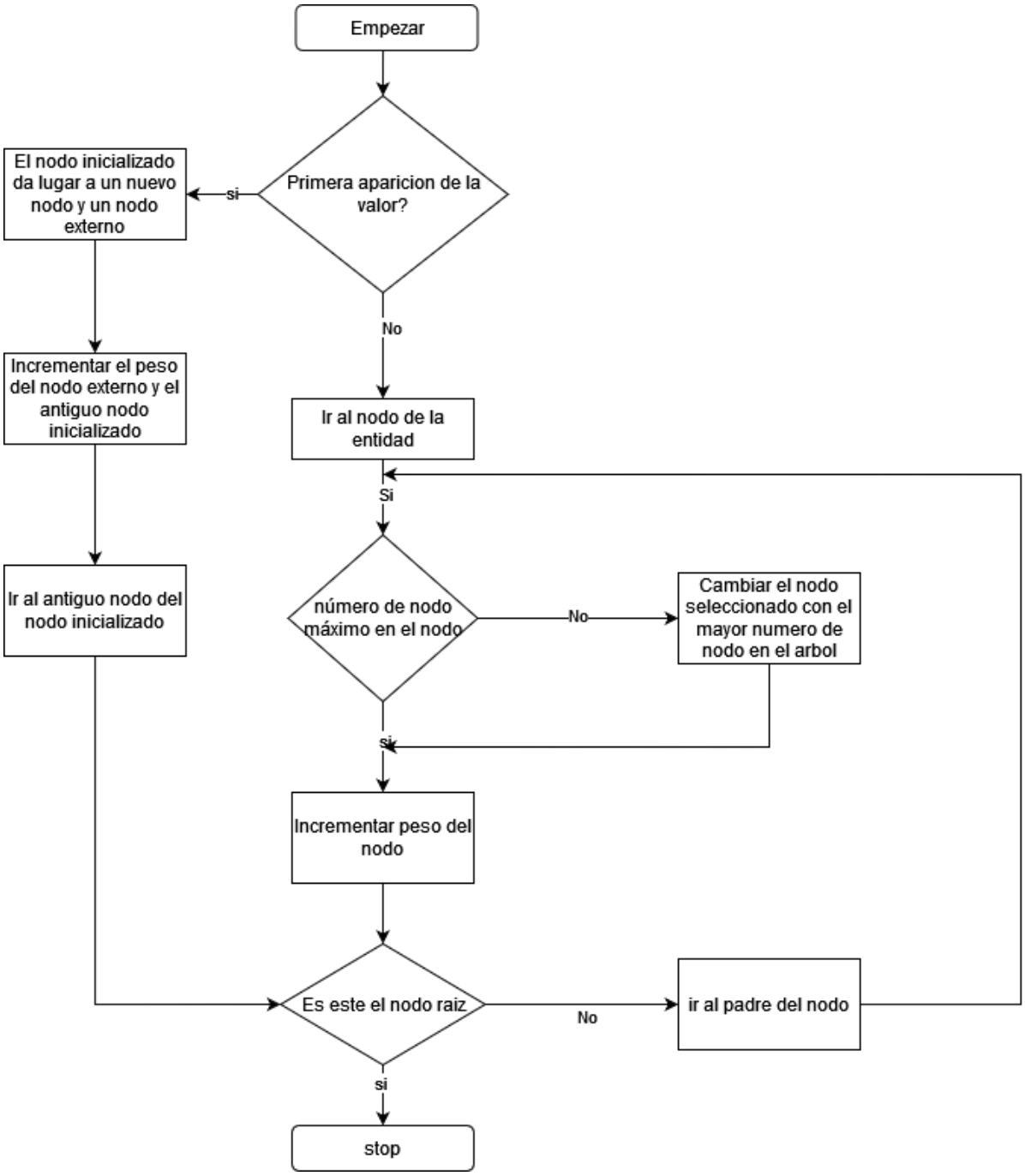
110001001011010010001101001001000110110100010011

Hemos pasado de tener 168 bits a solo 48 bits

Pseudocódigo

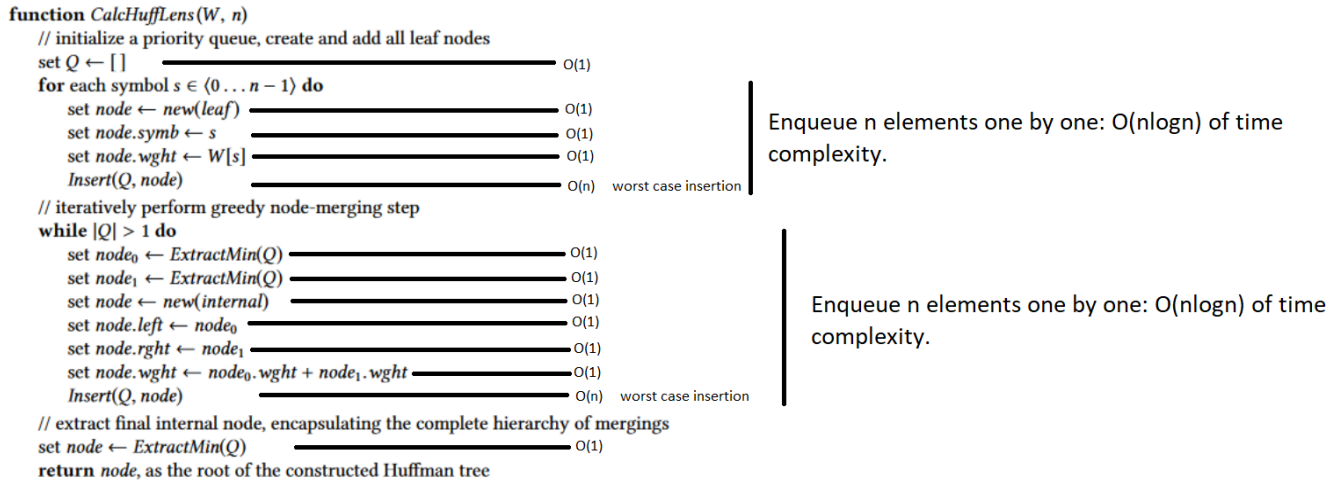
```
function CalcHuffLens(W, n)
    // initialize a priority queue, create and add all leaf nodes
    set Q ← [ ]
    for each symbol s ∈ ⟨0 . . . n − 1⟩ do
        set node ← new(leaf)
        set node.symb ← s
        set node.wght ← W[s]
        Insert(Q, node)
    // iteratively perform greedy node-merging step
    while |Q| > 1 do
        set node_0 ← ExtractMin(Q)
        set node_1 ← ExtractMin(Q)
        set node ← new(internal)
        set node.left ← node_0
        set node.rght ← node_1
        set node.wght ← node_0.wght + node_1.wght
        Insert(Q, node)
    // extract final internal node, encapsulating the complete hierarchy of mergings
    set node ← ExtractMin(Q)
    return node, as the root of the constructed Huffman tree
```

Flujograma


Empezar
Primera aparicion de la valor?
si
El nodo inicializado da lugar a un nuevo nodo y un nodo externo
No
Incrementar el peso del nodo externo y el antiguo nodo inicializado
Ir al nodo de la entidad
Si
Ir al antiguo nodo del nodo inicializado
número de nodo máximo en el nodo
No
Cambiar el nodo seleccionado con el mayor numero de nodo en el arbol
si
Incrementar peso del nodo
Es este el nodo raiz
No
ir al padre del nodo
si
stop

Análisis del Caso Base, Mejor de los Casos y Peor de los casos para el algoritmo.

Análisis de Time complexity
Caso base



La complejidad de tiempo es: T(n) = O(n log n) + O (n log n)

T(n) = O(n log n)

Best case

```
function CalcHuffLens(W, n)
    // initialize a priority queue, create and add all leaf nodes
    set Q ← [ ]                                                    O(n)
    for each symbol s ∈ ⟨0 . . . n − 1⟩ do                         O(1)
        set node ← new(leaf)                                       O(1)
        set node.symb ← s                                          O(1)
        set node.wght ← W[s]                                       O(1)
        Insert(Q, node)                                            O(1)
    // iteratively perform greedy node-merging step
    while |Q| > 1 do                                               O(1)   n+1
        set node0 ← ExtractMin(Q)                                  O(1)
        set node1 ← ExtractMin(Q)                                  O(1)
        set node ← new(internal)                                   O(1)
        set node.left ← node0                                      O(1)
        set node.rght ← node1                                      O(1)
        set node.wght ← node0.wght + node1.wght                    O(1)
        Insert(Q, node)                                            O(1)
    // extract final internal node, encapsulating the complete hierarchy of mergings
    set node ← ExtractMin(Q)                                       O(1)
    return node, as the root of the constructed Huffman tree
```

T(n) = O(n)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)

T(n) = O(n)

Worst case

```
function CalcHuffLens(W, n)
    // initialize a priority queue, create and add all leaf nodes
    set Q ← [ ]                                                    O(n)
    for each symbol s ∈ ⟨0 . . . n − 1⟩ do
        set node ← new(leaf)                                       O(1)
        set node.symb                                              O(n)   O(1)
        set node.wght ←                                            O(n)   W[s]   O(1)
        Insert(Q, node)                                            O(log n)
    // iteratively perform greedy node-merging step
    while |Q| > 1 do                                               O(1)
        set node0 ← ExtractMin(Q)                                  O(n)
        set node1 ← ExtractMin(Q)                                  O(n)
        set node ← new(internal)                                   O(1)
        set node.left ← node0                                      O(1)
        set node.rght ← node1                                      O(1)
        set node.wght ← node0.wght + node1.wght                    O(1)
        Insert(Q, node)                                            O(log n)
    // extract final internal node, encapsulating the complete hierarchy of mergings
    set node ← ExtractMin(Q)                                       O(n)
    return node, as the root of the constructed Huffman tree
```

Enqueue n elements one by one: O(nlogn) of time complexity.

Enqueue n elements one by one: O(nlogn) of time complexity.

T(n) = O(n) + O(1) + O(1) +O(1) + O(log n) + O(1) + O(1) +O(n) + O(n) +O(1) +O(1) +O(1) + O(log n) + O(n) + O(n log n) + O(n log n)

T(n) = O(n log n)

Conclusión

La complejidad de tiempo de este código es O (n log n), donde n es el número de caracteres en la cadena de entrada. Esto se debe a que el código primero crea una cola de prioridad de nodos, lo que lleva O(n log n) tiempo. Luego, construye el árbol de Huffman extrayendo repetidamente los dos nodos con la frecuencia más baja de la cola de prioridad y creando un nuevo nodo con su frecuencia combinada. Este proceso se repite n-1 veces, lo que da como resultado una complejidad de tiempo total de O (n log n).