PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 - Programación Avanzada
1^{er} semestre 2016

Actividad 04

Árboles y Listas Ligadas

Instrucciones

En computación entedemos por compresión de datos a un caso particular de codificación, cuyo objetivo es la reducción del volumen de datos en bits con que representamos una información dada. En general la compresión se basa en buscar repeticiones de series de símbolos y almacenar estos símbolos solo con el número de veces que ellos se repiten. La **codificación Huffman** es un algoritmo usado para la compresión de datos que consiste en crear una tabla de valores (de 0's y 1's) basada en la probabilidad estimada de aparición de cada caracter para codificar cada símbolo. Usted deberá escribir un código que le permita, dado un texto, obtener su compresión usando la codificación Huffman.

Para que quede más claro el proceso de compresión veamos un ejemplo. Dado un texto, primero debemos generar una *tabla de frecuencias* que contiene cada caracter y la cantidad de apariciones de este en el texto.

Texto: `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaddddd`
`ddddddddddddbbbbbbbbbbbbbbccccccccccccceeeeeeeeeeffffff`

Letra	Apariciones
E	9
A	45
B	13
D	16
F	5
C	12

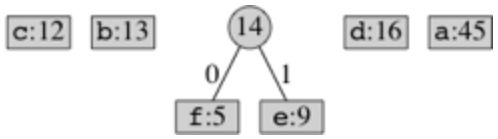
Luego, procedemos a crear el **Árbol de Huffman**. Para esto, debemos crear nodos que contengan tanto al caracter como el número de apariciones y colocar los nodos en una **lista ligada** de forma que vayan quedando ordenados (aquel con más prioridad primero).

Primero ordenamos los nodos en orden ascendente de apariciones:

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

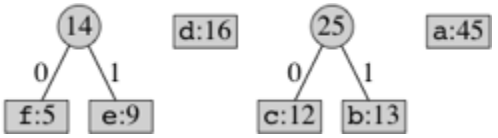
Luego tomamos los dos nodos más prioritarios (en este caso `f:5` y `e:9`), creamos un nuevo nodo uniendolos (`fe:14`) y agregamos el nodo en la posición que le corresponda:

(b)



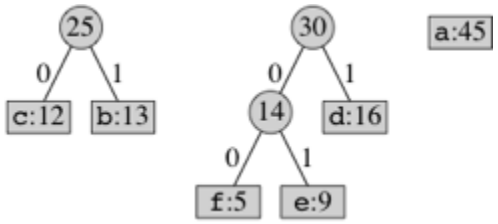
Volvemos a repetir el proceso, en este caso tomamos `c:12` y `b:13` para formar `cb:25`:

(c)



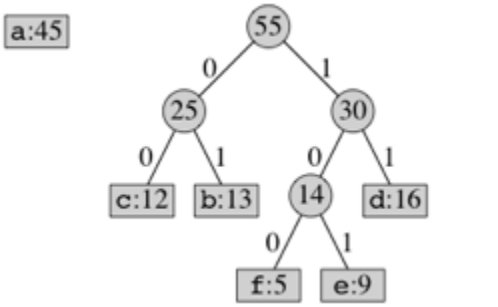
Ahora, los dos nodos más prioritarios son `fe:14` y `d:16`, que forman el nuevo nodo `fed:30` (notar como vamos *acumulando* los caracteres en cada nuevo nodo):

(d)



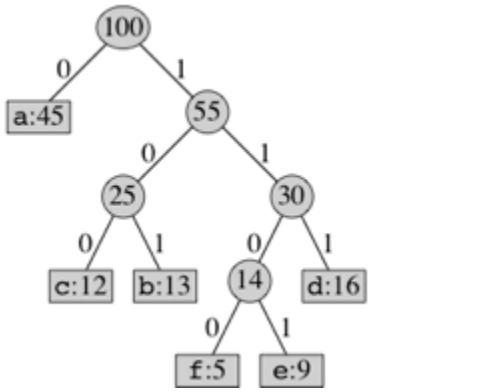
Repetimos para `cb:25` y `fed:30`, formando `cbfed:55`:

(e)



Repetimos para `a:45` y `cbfed:55`, formando el último nodo (`acbfed:100`). Con esto, nuestro árbol está listo:

(f)



Una vez que tenemos formado el árbol, etiquetamos la rama izquierda de cada nodo con un 0 y la derecha con un 1. La ruta para llegar a cada nodo hoja es la codificación que se asignará a dicho caracter. Volviendo al primer ejemplo:

Letra	Código
A	0
B	101
C	100
D	111
E	1101
F	1100

Una vez que hemos obtenido la tabla, solo debemos reemplazar cada caracter por su versión codificada:

```
00000000000000000000000000000000000000000000011111111111111111
11111111111111111111111111111111011011011011011011011011011101
10110110010010010010010010010010010010010010011011011011011011
                101110111011101110011001100110011001100
```

Notar que el string **sin comprimir** requiere **800 bits** para ser almacenado, mientras que la versión comprimida obtenida solo ocupa **224 bits**.

Requerimientos

- Se evaluará el uso de las estructuras de datos de Python que se presentan en el material de clases. Debe añadir los comportamientos necesarios a estas estructuras para lograr su objetivo.

Notas

- Para ordenar los nodos primero debe colocar aquellos con menos apariciones, en caso de empate deberá colocar primero aquellos que tengan menos nodos hoja (use la *acumulación de caracteres* del ejemplo) y en caso de que siga existiendo empate ordene estos nodos en orden alfabético inverso según su símbolo. Por ejemplo:

```
T:1 < R:1 < H:1 < B:1 < F:2
  H:1 < B:1 < F:2 < TR:2
    F:2 < TR:2 < HB:2
      HB:2 < FTR:4
        HBFTR:6
```

- Usted **no puede heredar de list** o de otra estructura de datos de Python. Todo comportamiento en las clases pedidas deben ser implementadas en su código.

To - Do

- (3.00 pts) Clase `LinkedList()`
  - (2.00 pts) `agregar_nodo(nodo)`: Este método recibe un nodo y lo coloca en la posición correcta de acuerdo a la prioridad (revisar el algoritmo explicado)
  - (1.00 pts) `obtener_nodo()`: Este método debe quitar y retornar el nodo de mayor prioridad
- (3.00 pts) Clase `HuffmanTree(texto)`: Esta clase recibirá el texto a compirmir y luego hará el proceso de compresión paso a paso, llamando a cada uno de los siguientes métodos.

- (0.50 pts) `generar_frecuencias()`: Dado el texto en el constructor, deberá guardar la cantidad de repeticiones de cada caracter en un atributo de la clase.
- (0.50 pts) `generar_nodos()`: Crea una lista ligada (`LinkedList`) y agrega los nodos, los que contienen cada caracter y su número de apariciones.
- (2.00 pts) `generar_arbol()`: Usando el algoritmos explicado debe ordenar los nodos y formar el árbol de Huffman.

Bonus

- (0.90 pts)Adicionalmente, luego implementar el algoritmo para formar el árbol, puede ser bonificado si desarrolla la implementación del siguiente código **en la clase** `HuffmanTree`, mostrando que funciona el algoritmo:
  - (0.70 pts) `generar_tabla()`: Con el árbol anterior debe recorrer el árbol y guardar la codificación de cada caracter. Es decir, generar las tablas con las etiquetas.
  - (0.20 pts) `comprimir()`: Aquí debe usar la tabla obtenida con el método anterior y retornar el string de 0's y 1's.
- (0.10 pts) Además, conociendo la técnica para comprimir, cree una **funcion** `decomprimir` que dada la tabla de codificación y el string de 0's y 1's decomprima el string mostrando el mensaje inicial.

Tips

- En el caso de los métodos de la clase **HuffmanTree** siga el orden recomendado para crear las funciones, así le será mas fácil el desarrollo de la actividad

Entrega

- **Lugar:** GIT - Carpeta: Actividades/AC04
- **Hora:** 16:50