Trabajo Práctico 1

El objetivo de este TP es dar una implementación del algoritmo de codificación de Huffman, usado para la compresión de datos. Para dar una implementación eficiente del mismo, trabajaremos con árboles binarios y heaps.
Se adjunta a este enunciado dos archivos Haskell: `Huffman.hs`, donde debe escribir todo el código que se pide en las consignas, y `Heap.hs` que provee una implementación de heaps que puede usar en su resolución.
La entrega debe consistir únicamente del archivo `Huffman.hs` con todas sus respuestas, incluyendo un comentario con los nombres de los integrantes del grupo y otro con la respuesta del ejercicio 7.

Un poco de contexto sobre codificación

Como bien sabemos, las computadoras almacenan información codificándola en dígitos binarios, unos y zeros, que llamamos *bits*. Para determinar cómo representamos caracteres con bits existen estándares de codificación que, básicamente, mapean caracteres a secuencias de bits.
Por la década del 60 surgió el estándar ASCII, que utiliza 7 bits para representar caracteres[1]. Por ejemplo, en ASCII la letra A se representa como la secuencia 1000001. Dado que cada caracter se representa con exactamente 7 bits, podemos representar, a lo sumo, $2^7 = 128$ caracteres distintos. Si bien es más que suficiente para representar todos los caracteres del alfabeto romano, no lo es si queremos incluir otros alfabetos.
En los años ochenta surgió el estándar Unicode, diseñado para soportar todos los sistemas de escritura digitalizables, incluyendo no sólo lenguas no romanas, sino también símbolos técnicos y científicos (entre ellos nuestra querida λ). Para ello, Unicode usa más bits por caracter: 8, 16 o 32, según la variante del estándar.
Si bien el problema de la expresibidad puede solucionarse fácilmente agregando más bits a la representación, estos sistemas de codificación que usan una cantidad fija de bits por caracter tienen otro problema, que se intensifica conforme se incrementa ese número de bits. Estos sistemas conllevan cierto despercidio de espacio en memoria dado que no todos los caracteres se utilizan con la misma frecuencia. Por ejemplo, si en lugar de usar 8 bits para representar la E y 8 para la Z, usásemos 6 bits para la E y 10 para Z, en la mayoría de los textos del castellaño usaríamos menos espacio en total, puesto que la letra E es aproximadamente 26 veces más frecuente que la Z[2].
La codificación de Huffman es un método que permite codificar caracteres con longitud variable de bits, de manera que la longitud total de la codificación binaria sea mínima. En realidad, no es específico a caracteres sino que también se puede usar para comprimir imágenes, música y videos, entre otros. Sin embargo, en este TP trabajaremos únicamente con caracteres.

Codificaciones de longitud fija y variable

Supongamos que queremos codificar la frase **la luna llena**. Si usamos la codificación ASCII, este string de 13 caracteres necesita de 13x8 = 104 bits. A continuación se muestra el subconjunto de la codificación ASCII relevante a este ejemplo:

'l'	**01101100**
'a'	**01100001**
' '	**00100000**
'u'	**01110101**
'n'	**01101110**
'e'	**01100101**

De esta forma, **la luna llena** queda codificado en ASCII de la siguiente forma:

011011000110000100100000011011000111010101101110011000010010000001101100011011000110010101101110011000010

[1] En realidad, se usan 8 bits por caracter para simplificar la implementación usando un byte por caracter.
[2] Frecuencia de aparición de letras, Wikipedia.

Para decodificar, simplemente leemos en grupos de 8 bits y buscamos en la tabla a qué caracter corresponde esa secuencia de bits. Por ejemplo, los primeros 8 bits son **01101100**, que corresponden a la letra **l**. Como ASCII es un estándar universal, para decodificar no necesitamos guardar ninguna información auxiliar.

Supongamos ahora que queremos utilizar un sistema personalizado para codificar nuestro mensaje con el objetivo de usar menos espacio en memoria. La primera observación es que no necesitamos representar todos los caracteres del alfabeto ni símbolos especiales, sino que necesitamos representar solamente 6 caracteres distintos. Por lo tanto, con sólo 3 bits podemos representar todos los caracteres que necesitamos. Por ejemplo, podemos usar la siguiente representación:

'l'	**000**
'a'	**001**
' '	**010**
'u'	**011**
'n'	**100**
'e'	**101**

De esta manera, nuestro mensaje usaría solamente 13x3=39 bits:

000001010000011100001010000000101100001

Similarmente, para decodificar el mensaje, tenemos que agrupar los bits de a tres y buscar a qué caracter corresponde en la tabla. Por supuesto, esta vez debemos proveer la tabla de codificación que usamos porque no es un estándar conocido.

Con esta codificación usamos solamente un 37.5 % del espacio que habíamos usado originariamente con la codificación ASCII. Sin embargo, si usamos una cantidad variable de bits para codificar cada caracter podemos reducir aún más el tamaño total de la codificación.
Intuitivamente, la idea es utilizar códigos más cortos para caracteres más frecuentes y códigos más largos para caracteres menos frecuentes. Veamos la cantidad de apariciones que tiene cada caracter en la frase:

'l'	4
'a'	3
' '	2
'n'	2
'u'	1
'e'	1

Consideremos la siguiente asignación de caracteres a códigos:

'l'	**10**
'a'	**01**
' '	**00**
'n'	**111**
'u'	**1101**
'e'	**1100**

Con esta codificación, la frase **la luna llena** se codifica como:

10010010110111101001010110011101

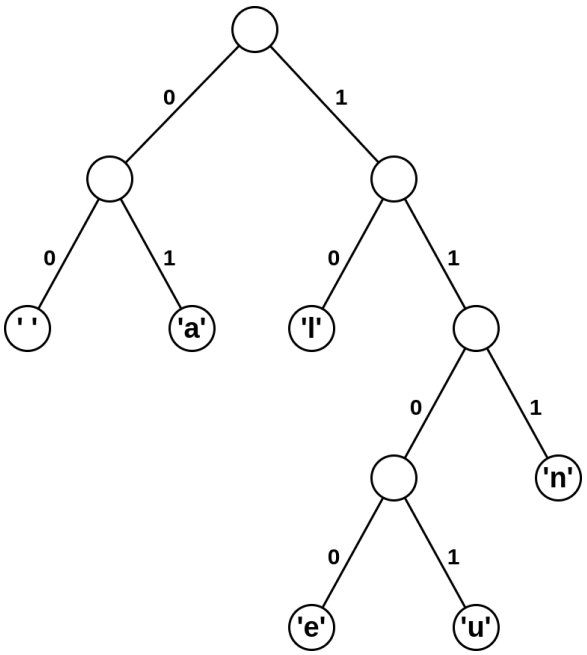
que usa sólo 32 bits, es decir sólo un 82 % del tamaño de la codificación anterior y un 30.8 % del tamaño de la codificación ASCII.
Pero, ¿cómo podemos decodificarla ahora que la longitud de la codificación de un caracter es variable? Simplemente leemos bits de izquierda a derecha hasta que coincida con el código de un caracter. Por ejemplo, para la codificación de arriba leemos **1** que no coincide con ningún caracter, entonces agregamos el siguiente bit y obtenemos **10**, que coincide con 'l'. Y repetimos hasta consumir toda la secuencia. Sin embargo, ¿es correcto este procedimiento?. ¿Qué pasa si leemos **10** y concluímos que corresponde a 'l' pero luego sigue **01** y **1001** corresponde a otro caracter? Observamos que en la asignación elegida esto no es posible porque ningún código es prefijo de otro. De esta manera, el procedimiento de decodificación descripto es correcto.

La codificación de Huffman

Dado un string a codificar, la codificación de Huffman genera mapeos de los caracteres del string a códigos binarios de longitud variable, garantizando que la codificación del string tenga longitud mínima y de manera que ningún código sea prefijo de otro.
Formalmente, podemos definir a este problema de codificación en los siguientes términos:

- Entrada:
  - Un conjunto $A = \{a_1, a_2, \ldots, a_n\}$ de n símbolos que llamamos alfabeto.
  - Un conjunto $W = \{w_1, w_2, \ldots, w_n\}$ de valores positivos correspondientes al peso de los símbolos de A, i.e. w_i es el peso de a_i.
- Salida:
  Un mapeo C de elementos de A en secuencias de bits, que llamamos códigos, de manera que:
  - $\sum_{i=1}^{n} w_i \cdot length(C(a_i))$ sea mínima;
  - $\forall\, 1 \leqslant i, j \leqslant n \cdot i \neq j \Rightarrow C(a_i)$ no es prefijo de $C(a_j)$.

Para obtener una codificación C de los símbolos de A que verifique ambas propiedades, el algoritmo de codificación de Huffman propone construir un árbol de codificación. Dicho árbol es un árbol binario donde las hojas representan los símbolos del alfabeto, y el código correpondiente a los mismos está dado por el camino de la raíz a su correspondiente hoja, donde las aristas izquierdas representan ceros y las derechas unos. Como veremos más adelante, la forma en la que está construído el árbol garantiza el cumplimiento de las dos propiedades enunciadas.
A continuación se muestra un árbol de codificación para el ejemplo anterior:



Para decodificar un código, simplemente leemos la secuencia de bits a decodificar y recorremos el árbol de codificación. Comenzamos por la raíz y visitamos el hijo izquierdo si el bit es 0 o el derecho si el bit es 1, hasta llegar a una hoja; y repetimos este proceso hasta leer toda la entrada. En el ejemplo anterior, para decodificar **1001001011011110100101011001 1101** leemos **10** y llegamos al símbolo 'l', luego leemos **01** y llegamos a 'a', y así siguiendo hasta leer **01** que corresponde nuevamente a 'a', obteniendo **la luna llena** como resultado.
Mientras que para codificar un string, simplemente reemplazamos cada símbolo por su correspondiende código dado por el árbol. Por ejemplo, para codificar **la luna llena** vemos que al caracter 'l' le corresponde el código **10**, a 'a' le corresponde **01**, a ' ' le corresponde **00**, etc, obteniendo **1001001011011110100101011001 1101** como resultado.

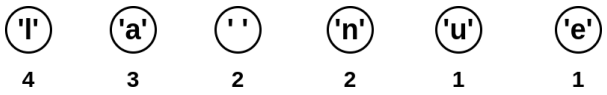
Mediante la resolución de los ejercicios que siguen obtendrá una implementación sencilla de la codificación de Huffman en Haskell, incluyendo la construcción del árbol de codificación, la codificación de strings y la decodificación de códigos.

Árboles de codificación

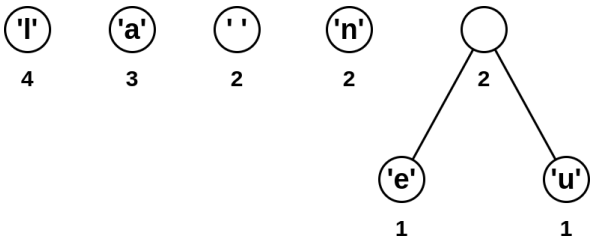
A continuación veremos cómo construir un árbol de codificación óptimo, es decir tal que la longitud de la codificación del string sea mínima. El algoritmo de Huffman puede enunciarse informalmente de la siguiente manera:

1. Por cada símbolo a_i de A, construir un árbol t_i que consiste en un único nodo etiquetado con a_i y con un peso asociado dado por w_i. Llamamos B al bosque formado por los árboles t_i.
2. Si B tiene un único árbol t, entonces t es un árbol de codificación óptimo. Sino, pasar al paso 3.
3. Extraer de B dos árboles t_a y t_b con menor peso, y formar un nuevo árbol t que tenga por hijos izquierdo y derecho a t_a y t_b (en cualquier orden) y cuyo peso sea la suma de los pesos de t_a y t_b y agregarlo a B. Volver a 2.

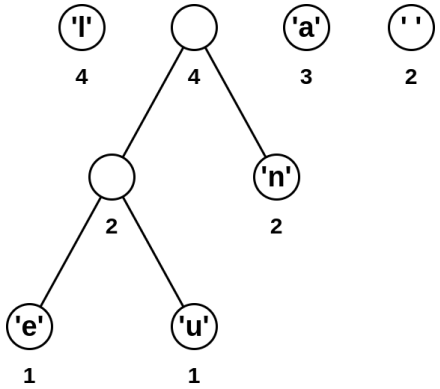
Veamos, paso por paso, la aplicación de este algoritmo para obtener un árbol de codificación para el ejemplo con el que estamos trabajando. Para empezar, construímos los 6 árboles correspondientes a los símbolos del alfabeto como indica el paso 1, donde el peso de los nodos está dado por la cantidad de apariciones del caracter que representa:



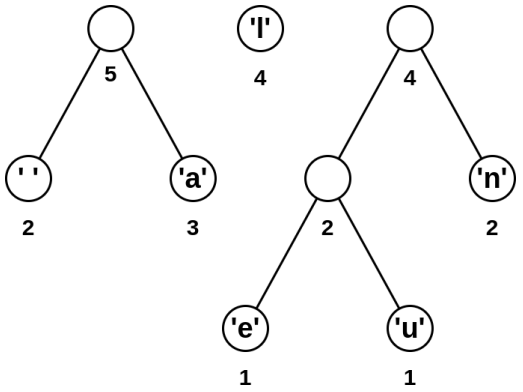
A continuación, como no hay un único árbol en el bosque, elegimos dos árboles con peso mínimo (que en este caso sólo pueden ser los etiquetados con ‘e’ y ‘u’) y los combinamos como se especifica en el paso 3:



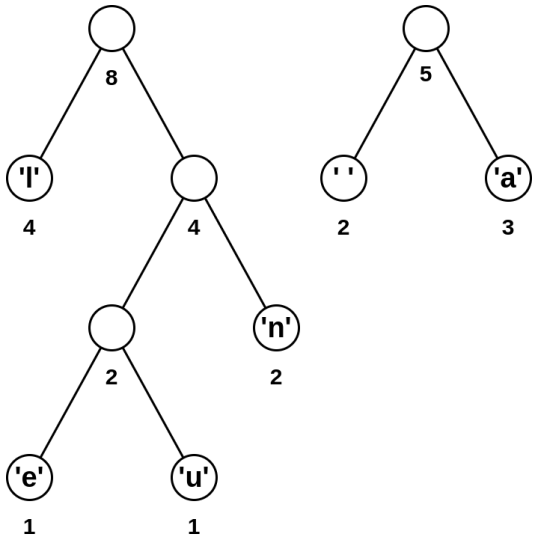
Notar que, alternativamente, podríamos haber puesto la ‘e’ a la derecha y la ‘u’ a la izquierda. Continuando, repetimos los pasos 2 y 3. En esta oportunidad hay 3 árboles con peso mínimo (todos los de peso 2). Podemos elegir cualesquiera dos de ellos para combinarlos, por ejemplo:



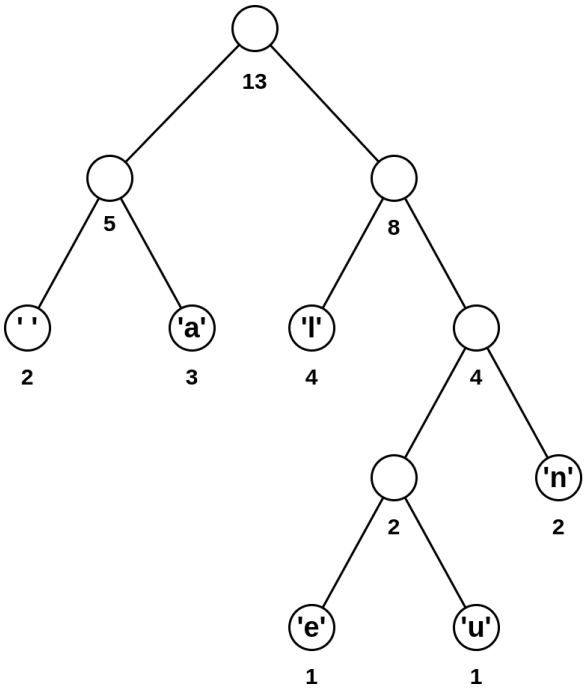
Repetimos nuevamente los pasos 2 y 3, esta vez combinando los árboles de peso 2 y 3:



En la siguiente iteración combinamos los árboles de peso 4:



Y finalmente combinamos los únicos dos árboles que quedaron, obteniendo el siguiente árbol:



Nótese que este árbol de codificación óptimo no es único, pues en la segunda iteración podríamos haber elegido otro par de árboles para combinar o en cualquiera de las iteraciones podríamos haber intercambiado el subárbol derecho y el izquierdo. Sin embargo, el algoritmo garantiza que un árbol construído de esta manera es óptimo.
Intuitivamente, la optimalidad del árbol se cumple porque al construir el árbol ponemos primero a los caracteres menos frecuentes, que son los que quedan con mayor profundidad en el árbol; lo que se traduce en códigos más largos para los caracteres menos frecuentes y más cortos para los más frecuentes. Por otro lado, que ningún código sea prefijo de otro se cumple por construcción: si un código fuese prefijo de otro, implicaría que una hoja es descendiente de otra hoja, lo cual es imposible.

Para representar árboles de codificación en Haskell usaremos el siguiente tipo de datos algebraico:

```
data HTree = Leaf Char Int | Node HTree HTree Int
```

donde las hojas están etiquetadas con caracteres que representan los símbolos a codificar y todo árbol tiene un peso asociado que representamos con un valor entero. Observamos que para implementar el algoritmo de Huffman, necesitaremos definir una relación de comparación entre árboles de codificación para decidir cuáles son los de menor peso.

1. Dar una instancia de la clase Ord para el tipo HTree. Para ello, primero debe dar una instancia de la clase Eq.

Por otro lado, para determinar el peso de los árboles tenemos que contar las apariciones de cada símbolo en un string, para lo cual definimos el tipo FreqMap:

```
type FreqMap = Map Char Int
```

donde el tipo Map está definido en la librería Data.Map e implementa diccionarios[3].

2. Definir una función buildFreqMap :: String → FreqMap que dado un string, compute la cantidad de apariciones de cada uno de sus caracteres.
Sugerencia: la función insertWith de Data.Map puede ser de utilidad.

3. Definir una función buildHTree :: FreqMap → HTree que compute un árbol de codificación óptimo para los símbolos y su cantidad de apariciones dados, siguiendo el algoritmo descripto anteriormente. Para implementarlo eficientemente, use un heap para tener acceso eficiente a los árboles de menor peso. En el módulo Heap se provee una implementación de min-heaps basada en leftist heaps, como se vio en clase.

Codificando strings

Para este TP representamos a los bits mediante el siguiente tipo de datos:

```
data Bit = Zero | One
```

y a los códigos como listas de bits:

```
type Code = [Bit]
```

Como se explicó anteriormente, para codificar un string simplemente debemos reemplazar cada caracter por su codificación binaria dada por el árbol de codificación que usemos. Para eso, debemos recorrer el camino de la raíz a la hoja que está etiquetada con el caracter en cuestión. En principio, tendríamos que recorrer todos los caminos de la raíz a una hoja hasta encontrar la hoja que necesitamos, por lo cual no resulta práctico trabajar directamente con árboles de codificación. En su lugar, podemos recorrer el árbol de codificación una única vez para encontrar todos los códigos, almacenarlos en un diccionario, y luego usar ese diccionario para recuperar el código correspondiente a cada caracter. Para ello definimos el tipo CodeMap de la siguiente manera:

```
type CodeMap = Map Char Code
```

[3]Puede consultar la documentación aquí.

4. Definir una función buildCodeMap :: HTree → CodeMap que dado un árbol de codificación construya un diccionario de códigos de caracteres.

5. Definir una función encode :: CodeMap → String → Code que codifique un string, dado un diccionario de códigos.

Decodificando códigos

Para decodificar un código necesitamos recorrer el árbol de codificación con el que se codificó, tal como se explicó al comienzo de esta sección.

6. Definir una función decode :: HTree → Code → String que decodifique un código, dado el árbol con el que se codificó.

Analizando resultados

Hasta aquí, hemos utilizado el algoritmo de codificación de Huffman para codificar un string particular: para un string dado calculamos la cantidad de apariciones de cada caracter, construímos un árbol de codificación óptimo para ese string y lo usamos para codificar y decodificar ese string particular. En este ejercicio se propone usar el algoritmo de Huffman para obtener una tabla de codificación para la compresión de strings en general, con el objetivo de compararlo con sistemas que usan códigos de longitud fija.

7. En el código está definida engFM :: FreqMap que representa la frecuencia con la que aparecen algunos caracteres en textos del idioma inglés[4]. Por simplicidad, sólo se incluyen los 26[5] caracteres alfabéticos en minúsculas, el espacio, la coma y el punto.

Resuelva las siguientes consignas:

1. Obtenga un árbol de codificación para engFM.
2. Defina no menos de 10 strings correspondientes a textos del idioma inglés. Pueden ser, por ejemplo, oraciones o parráfos de algún libro de literatura, artículos o páginas web. En lo posible, extraiga texto de diversas fuentes. Codifique dichos strings usando el árbol del apartado anterior. Recuerde usar sólo minúsculas y obviar apóstrofes, guiones y otros caracteres especiales. Compare la longitud de las codificaciones que obtuvo con la longitud de las codificaciones que usan una longitud fija de 5 bits por caracter[6].
3. Escriba una breve conclusión sobre los resultados obtenidos.

[4] Letter frequency y English punctuation, Wikipedia.
[5] En Inglés la ñ no forma parte del alfabeto.
[6] Como necesita representar sólo 29 caracteres, 5 bits son suficientes.