Práctica final: códigos Huffman

Nuevas tecnologías de la programación

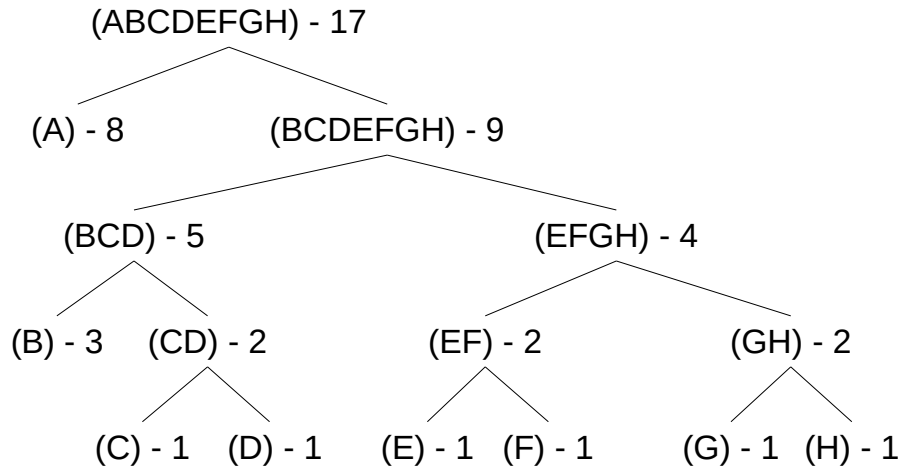
Contenido:

1 Introducción

La codificación de Huffman es un algoritmo de compresión que puede usarse para codificar archivos de texto. En un texto normal, no comprimido, todos los caracteres se representan mediante el mismo número de bits (8 usualmente, el tamaño de un byte). Sin embargo, con esta codificación ada carácter puede tener una representación con diferente número de bits. Los caracteres que aparecen más frecuentemente se codifican con menos bits que aquellos que tienen menor frecuencia de aparición.

1.1 Representación

Un código de Huffman se representa mediante un árbol binario donde los caracteres que forman el alfabeto aparecen en los nodos terminales. El ejemplo que aparece seguidamente muestra un código Huffman para un texto en el que el alfabeto (el conjunto de caracteres posibles) es $\{A, B, C, D, E, F, G, H\}$.



Se observa que el nodo raíz representa el conjunto completo de caracteres que aparece en el texto a codificar. El contador asociado a cada nodo indica las ocurrencias de los caracteres que representa. Todos los nodos intermedios, no terminales, representan conjuntos de caracteres.

Los nodos terminales representan a un único carácter y el contador indica el número de veces que aparece en el texto analizado. De esta forma, puede considerarse que cada nodo del árbol de codificación representa el conjunto de caracteres de todos los nodos ubicados bajo él. Y su peso será igual a la suma de sus nodos hijo.

También cabe observar la naturaleza recursiva del árbol de codificación: cada subárbol es, a su vez, un código Huffman válido para un alfabeto menor.

1.2 Codificación

Dado un código Huffman puede obtenerse la representación codificada de un carácter recorriendo el árbol desde la raíz hasta la hoja que lo contiene. A medida que se recorre este camino se anota un 0 cuando se elige la rama de la izquierda y un 1 al seleccionar ramas de la derecha. De esta forma, el carácter D se codifica como 1011. El carácter A, el más frecuente, tiene 0 como código.

1.3 Decodificación

La decodificación también comienza desde la raíz del árbol. Dada una secuencia de bits a decodificar se tratan sucesivamente los bits y, para cada 0 se selecciona la rama de la izquierda y para cada 1 la de la derecha. Al alcanzar un nodo hoja se guarda el carácter del nodo alcanzado y se prosigue el proceso de decodificación desde la raíz. Usando el árbol de codificación de ejemplo, la secuencia 10001010 se corresponde con BAC.

2 Implementación

Se recomienda basar la implementación en la siguiente estructura de clases:

- una clase abstracta para representar los nodos del árbol de codificación (clase **Nodo**)
- clases concretas para nodos terminales (**NodoHoja**) y no terminales (**NodoIntermedio**)

- para los nodos intermedios habrá que almacenar hijos a derecha e izquierda (tipo **Nodo**), la lista de caracteres representados por el nodo y el peso o contador correspondiente
- para los nodos terminales basta con almacenar el carácter que representa y su peso
- todas estas clases auxiliares pueden declararse dentro de un objeto tipo singleton que representa esta forma de codificar (**Huffman**)
- este objeto debería contar con la siguiente funcionalidad:
  - calcular peso: recibe como argumento un nodo y devuelve el peso asociado calculando los pesos de los nodos inferiores, desde las hojas hasta sus hijos
  - obtener caracteres: recibe como argumento un árbol de codificación (un nodo, su raíz) y devuelve la lista de caracteres que representa, considerando todos los nodos inferiores
  - generar árbol: recibe como argumento los subárboles a izquierda y derecha y genera un nuevo árbol a partir de ellos

Conviene que las clases no abstractas de nodos se declaren como clases tipo **case**, para facilitar su posterior procesamiento mediante coincidencia de patrones. Por ejemplo, haciendo uso del método de generación de árboles de codificación podríamos obtener el árbol del ejemplo de la siguiente forma:

```
1  val hojaG=NodoHoja('G',1)
2  val hojaH=NodoHoja('H',1)
3  val nodoGH=generarArbol(hojaG, hojaH)
4  val hojaE=NodoHoja('E',1)
5  val hijaF=NodoHoja('F',1)
6  val nodoEF=generarArbol(hojaE, hojaF)
7  val nodoEFGH=generarArbol(nodoEF, nodoGH)
8  val hojaC=NodoHoja('C',1)
9  val hojaD=NodoHoja('D',1)
10 val nodoCD=generarArbol(hojaC. hojaD)
11 val hojaB=NodoHoja('B',3)
12 val nodoBCD=generarArbol(hojaB, nodoCD)
13 val nodoBCDEFGH=generarArbol(nodoBCD, nodoEFGH)
14 val hojaA=NodoHoja('A',8)
15 val total=NodoIntermedio(hojaA, nodoBCDEFGH)
```

Se observa que la creación de objetos de las clase **NodoHoja** y **NodoIntermedio** no precisa usar el operador **new**, al tratarse de clases tipo **case**. De cualquier forma, todas estas indicaciones de diseño pueden considerarse como meras recomendaciones.

2.1 Construcción de árboles de codificación

Dado un texto, es posible calcular y construir un árbol de codificación analizando sus caracteres y contadores de ocurrencia. Para generar este árbol se precisa un método (**generarArbolCodificacion**) que reciba como argumento una lista de caracteres y devuelve el nodo raíz del árbol. Para implementar esta tarea podrían seguirse los pasos siguientes:

- comenzar escribiendo una función (**obtenerTuplasOcurrencias**) que calcule la frecuencia de aparición de cada carácter en el texto a analizar
- implementar una función (**generarListHojasOrdenadas**) que genere una lista con todos los nodos hoja del árbol de codificación. Esta lista de nodos terminales debe estar ordenada por pesos, de forma ascendente
- función **singleton**, que compruebe si una lista de nodos (árboles) contiene a un único elemento
- función **combinar**, que combine todos los nodos terminales. Su funcionamiento se basa en:
  - elimina de la lista de árboles (nodos) los dos con menos peso
  - los combina para formar un nodo intermedio con ellos
  - inserta este nodo (árbol) en la lista de nodos a combinar. La inserción de realizarse de forma que se preserve el orden
  - función **hasta**, que haga llamadas a las funciones definidas en pasos anteriores hasta que la lista de nodos contenga un único elemento. Esta función podría llamarse de la siguiente forma:

    ```
    1 hasta(singleton,combinar)(arboles)
    ```

    donde el argumento arboles tendría tipo $List[Nodo]$
- usando este conjunto de funciones se implementará un método llamado **generarArbolCodificacion** que recibirá como argumento la lista de caracteres a analizar y devuelve el árbol generado

El paso entre una cadena de texto normal y la lista de caracteres puede realizarse mediante la función:

```
1 def stringAListaCaracteres(cadena:string):List[Char] = str.toList
```

3 Funcionalidad a probar

3.1 Decodificación

El software debe ofrecer la posibilidad de decodificar una lista de 0's y 1's que ha sido codificada mediante un árbol específico que se facilitará. La función de decodificación tiene la siguiente declaración:

```
1 def decodificar(arbol : Nodo, bits : List[Int]) : List[Char]
```

La secuencia de 0's y a's se denomina $mensajeSecreto$ y el árbol de codificación se denomina $codigoHuffmanFrances$.

3.2 Codificación

También se debe definir una función para codificar, con la siguiente declaración:

```
1 def codificar(arbol : Nodo, texto : List[Char]) : List[Int]
```

La función anterior es simple, pero ineficiente al basarse en recursividad. Incluso en textos con moderado tamaño puede generar problemas de ejecución. Por eso, otra forma más eficiente de codificación consiste en disponer de una tabla de códigos, con el siguiente tipo:

```
1 type TablaCodigo=List[(Char, List[Int])]
```

en la que se dispone del código asociado a cada carácter de forma directa (sin necesidad de tener que recorrer el árbol). Esta tabla puede accederse mediante una función como:

```
1 codificarConTabla(tabla : TablaCodigo)(caracter : Char) : List[Int]
```

La creación de la tabla puede hacerse visitando el árbol de codificación, Se dispondrá de una función a tal efecto:

```
1 def convertirArbolTabla(arbolCodificacion : Nodo) : TablaCodigo
```

Esta estructura, y las operaciones vistas, deben usarse para implementar un método llamado *codificacionRapida* que recibirá como argumento el árbol de codificación y el texto a codificar. Obviamente, el árbol de codificación debe usarse para crear la tabla de códigos mencionada con anterioridad.

4 Defensa de práctica

Al final de la realización de la práctica se entregará un archivo comprimido con el contenido completo de la práctica, tal y como se integra en el proyecto con el entorno de desarrollo que hayáis usado. Se incluirá también un pequeño documento indicando el entorno de desarrollo y una breve valoración de la práctica (si los conceptos vistos son novedosos, si os ha parecido de interés, problemas encontrados, etc) en tres o cuatro líneas.

Pueden realizarse los casos de prueba que se consideren necesarios para ir comprobando que el código funciona de forma correcta.

La fecha límite de entrega se fija para la semana posterior a la del último examen de la convocatoria de Junio (28 de Junio de 2016). Antes de esa fecha límite, en cualquier momento podéis avisar por correo electrónico y fijamos un momento para la defensa de la misma, que se hará de forma presencial en el despacho 31 de la planta 4 del edificio de despachos.