Nuevas tecnologías de la programación

Práctica 3: práctica final de la asignatura

E.T.S. de Ingenierías Informática y de Telecomunicación
Departamento de Ciencias de la Computación
e Inteligencia Artificial
Universidad de Granada

Curso 2023-2024

Práctica 3: práctica final de la asignatura

Índice

1 Objetivos

En esta práctica se trabajará, de forma opcional, con diferentes estructuras de almacenamiento de información, con el objetivo de practicar los conceptos básicos de diseño orientado a objetos en Scala. En concreto, podéis elegir entre:

- la representación funcional de conjuntos basada en la noción matemática de funciones características (hasta 7 puntos).
- la representación de listas, similares a las proporcionadas por Scala, pero implementada por nosotros. En este caso se proporciona la estructura de clases a usar (hasta 7 puntos).

- la representación de árboles binarios. Se trata de la sección más abierta y que debéis definir de forma completa si optáis por esta alternativa (hasta 10 puntos). Se trata de implementar las estructuras necesarias (tipo árbol binario) para permitir la codificación de caracteres mediante códigos Huffman.
- la entrega conjunta de las dos primeras alternativas también permite optar a una puntuación de 9.

2 Conjuntos mediante funciones características

En la práctica se trabaja, sin pérdida de generalidad, con conjuntos definidos por propiedades aplicadas sobre enteros. Como ejemplo motivador, pensemos en la forma de representar el conjunto de todos los enteros negativos: $x < 0$ sería la función característica.

```
1 (x : Int) => x < 0
```

Siguiendo esta idea, la representación del conjunto se hará definiendo la clase **Conjunto** que define un dato miembro que permite almacenar la función característica (función que recibe como argumento un valor entero y devuelve un valor booleano indicando pertenencia o no). Se deben implementar los siguientes métodos (queda a vuestra elección el lugar y forma más adecuada para implementarlos):

- **apply**, que recibe como argumento un valor entero e indica si este valor pertenece o no al conjunto.
- **toString**: ofrece una visión del contenido del conjunto. Para visualizar el conjunto se asume que se itera sobre un rango de valores dado por una constante llamada **LIMITE** (desde -LIMITE hasta +LIMITE) y se muestran aquellos que pertenecen al conjunto.
- **conjuntoUnElemento**: creación de un conjunto dado por un único elemento.
- **union**: dados dos objetos de la clase **Conjunto** produce su unión.
- **intersección**: intersección de dos objetos.
- **diferencia**: diferencia de dos objetos (el conjunto resultante está formado por aquellos valores que pertenecen al primer conjunto, pero no al segundo).
- **filtrar**: dado un conjunto y una función tipo $Int => Boolean$, devuelve como resultado un conjunto con los elementos que cumplen la condición indicada.
- **paraTodo**: comprueba si un determinado predicado se cumple para todos los elementos del conjunto. Esta función debe implementarse de forma recursiva, definiendo una función auxiliar, ya que hay que iterar sobre el rango de valores dado por LIMITE.
- **existe**: determina si un conjunto contiene al menos un elemento para el que se cumple un cierto predicado. Debe basarse en el método anterior.
- **map**: transforma un conjunto en otro aplicando una cierta función.

3 Clase Lista

La declaración de esta estructura debe basarse en la distinción básica entre una lista vacía y una lista con elementos. Se plantea el siguiente diseño:

```scala
/**
  * Interfaz generica para la lista
  * @param A
  */
sealed trait Lista[+A]

/**
  * Objeto para definir lista vacia
  */
case object Nil extends Lista[Nothing]

/**
  * Clase para definir la lista como compuesta por elemento inicial
  * (cabeza) y resto (cola)
  * @param cabeza
  * @param cola
  * @param A
  */
case class Cons[+A](cabeza : A, cola : Lista[A]) extends Lista[A]
```

En primer lugar, se trata de analizar el diseño y explicar las razones por la que se utilizan los diferentes calificadores y elementos, así como proponer los cambios que podáis considerar de interés. Investigad además qué implica el uso del signo más en la indicación de tipo A en **Lista** y en **Cons**. A partir de estos elementos (o los que consideréis oportunos) y en el cuerpo de un objeto denominado **Lista** se trata de implementar los siguientes métodos:

```scala
/**
 * Metodo para permitir crear listas sin usar new
 * @param elementos secuencia de elementos a incluir en la lista
 * @param A
 * @return
 */
def apply[A](elementos : A*) : Lista[A] = ???

/**
 * Obtiene la longitud de una lista
 * @param lista
 * @param A
 * @return
 */
def longitud[A](lista : Lista[A]) : Int = ???

/**
 * Metodo para sumar los valores de una lista de enteros
 * @param enteros
 * @return
 */
def sumaEnteros(enteros : Lista[Int]) : Double = ???
```

```scala

/**
 * Metodo para multiplicar los valores de una lista de enteros
 * @param enteros
 * @return
 */
def productoEnteros(enteros : Lista[Int]) : Double = ???

/**
 * Metodo para agregar el contenido de dos listas
 * @param lista1
 * @param lista2
 * @param A
 * @return
 */
def concatenar[A](lista1: Lista[A], lista2: Lista[A]): Lista[A] = ???

/**
 * Funcion de utilidad para aplicar una funcion de forma sucesiva a los
 * elementos de la lista con asociatividad por la derecha
 * @param lista
 * @param neutro
 * @param funcion
 * @param A
 * @param B
 * @return
*/
def foldRight[A, B](lista : Lista[A], neutro : B)(funcion : (A, B) => B): B
↪  = ???

/**
 * Suma mediante foldRight
 * @param listaEnteros
 * @return
 */
def sumaFoldRight(listaEnteros : Lista[Int]) : Double = ???

/**
 * Producto mediante foldRight
 * @param listaEnteros
 * @return
 */
def productoFoldRight(listaEnteros : Lista[Int]) : Double = ???

/**
 * Reemplaza la cabeza por nuevo valor. Se asume que si la lista esta vacia
 * se devuelve una lista con el nuevo elemento
 *
 * @param lista
 * @param cabezaNueva
 * @param A
 * @return
*/
def asignarCabeza[A](lista : Lista[A], cabezaNueva : A) : Lista[A] = ???

   /**
     * Devuelve el primer elemento de la lista
```

```
79        * (si no esta vacia). Por eso se devuelve Option
80        * @param lista
81        * @tparam A
82        * @return
83        */
84      def head[A](lista : Lista[A]) : Option[A] = ???
85
86  /**
87   * Elimina el elemento cabeza de la lista
88   * @param lista
89   * @param A
90   * @return
91   */
92  def tail[A](lista : Lista[A]): Lista[A] = ???
93
94  /**
95   * Elimina los n primeros elementos de una lista
96   * @param lista lista con la que trabajar
97   * @param n numero de elementos a eliminar
98   * @param A tipo de datos
99   * @return
100  */
101 def eliminar[A](lista : Lista[A], n: Int) : Lista[A] = ???
102
103 /**
104  * Elimina elementos mientra se cumple la condicion pasada como
105  * argumento
106  * @param lista lista con la que trabajar
107  * @param criterio predicado a considerar para continuar con el borrado
108  * @param A tipo de datos a usar
109  * @return
110  */
111 def eliminarMientras[A](lista : Lista[A], criterio: A => Boolean) : Lista[A]
    ↪  = ???
112
113
114 /**
115  * Elimina el ultimo elemento de la lista. Aqui no se pueden compartir
116  * datos en los objetos y hay que generar una nueva lista copiando
117  * datos
118  * @param lista lista con la que trabajar
119  * @param A tipo de datos de la lista
120  * @return
121  */
122 def eliminarUltimo[A](lista : Lista[A]) : Lista[A] = ???
123
124 /**
125  * foldLeft con recursividad tipo tail
126  * @param lista lista con la que trabajar
127  * @param neutro elemento neutro
128  * @param funcion funcion a aplicar
129  * @param A parametros de tipo de elementos de la lista
130  * @param B parametro de tipo del elemento neutro
131  * @return
132  */
133 @annotation.tailrec
```

```
134 def foldLeft[A, B](lista : Lista[A], neutro: B)(funcion : (B, A) => B): B =
    ↪  ???
```

4 Códigos Huffman

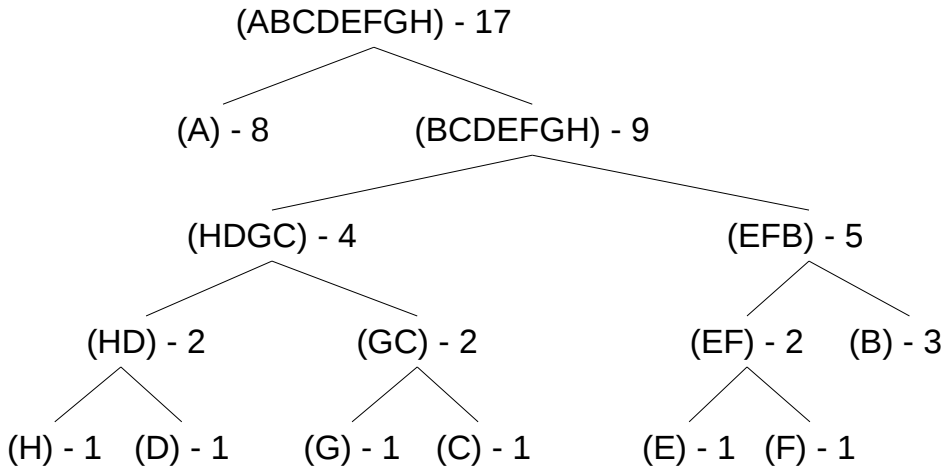
4.1 Introducción

La codificación de Huffman es un algoritmo de compresión que puede usarse para comprimir cadenas de caracteres. 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 cada 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 menos frecuencia de aparición.

4.1.1 Representación

Un código de Huffman se representa mediante un árbol binario donde los caracteres que forman el alfabeto aparecen en los nodos terminales. Imaginemos que el mensaje a codificar es $AAAAAAAABBBCDEFGH$. Si se construye un código de Huffman a partir de esta cadena (lo normal sería obtenerlo a partir del análisis de uso de los caracteres de un determinado lenguaje) se observa que las frecuencias de aparición de los diferentes caracteres son: $A(8)$, $B(3)$, $C(1)$, $D(1)$, $E(1)$, $F(1)$, $G(1)$, $H(1)$.

El árbol que contiene el código Huffman creado a partir del texto es el mostrado a continuación:



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 los pesos 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.

4.1.2 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 1001 (para llegar a él se selecciona el hijo a la derecha del nodo raíz (1); en el nodo para $BCDEFGH$ se debe seleccionar el hijo de la izquierda (0); en el nodo para $HDGC$ se selecciona el hijo a la izquierda (0); en el hijo para HD se selecciona el hijo a la derecha (1)). Por su parte, el carácter A, el más frecuente, tiene 0 como código. El carácter B (el segundo en frecuencia de aparición) se codifica como 111.

4.1.3 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 HG (H se codifica como 1000 y G como 1010).

4.2 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 (**hojas**) y no terminales (**internos**)
- para los nodos internos 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
- se puede dotar al diseño de las características que consideréis más oportunas, añadiendo, en la medida de lo posible todas las características de programación funcional que sea posible
- se indican aquí algunas funciones que pueden resultar de utilidad para los nodos del árbol:
  - calcular peso: devuelve el peso asociado al nodo (en los nodos hoja es un dato miembro; en los nodos intermedios se obtendrá calculando los pesos de los nodos inferiores)
  - obtener caracteres: devuelve la lista de caracteres que representa el nodo, considerando todos los nodos inferiores
  - generar árbol: recibe como argumento una lista de nodos y genera un nuevo árbol a partir de ellos

Parte del diseño consiste en decidir dónde se ubicará cada una de las funcionalidades indicadas en todo el guión.

4.2.1 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 puede usarse un método encargado de construir el árbol, que recibirá como argumento una lista de caracteres (el texto a partir del que generar el código). Como operaciones auxiliares para implementar esta tarea podrían seguirse los pasos siguientes:

- calcular las frecuencias de aparición de cada carácter. El método encargado de esta tarea debe generar una estructura que permita almacenar pares del tipo (carácter - contador de ocurrencias)
- esas parejas pueden ordenarse en función a la frecuencia de aparición, de forma que los caracteres con menor frecuencia aparezcan al principio. Una vez ordenadas puede construirse una lista de nodos hoja (uno por carácter). Como las parejas de carácter y frecuencia están ordenadas, también lo estará la lista de nodos (por su peso, que equivale a frecuencia)
- función **singleton**, que compruebe si una lista de nodos contiene a un único elemento
- función **combinar**, que combine todos los nodos contenidos en una lista de nodos (inicialmente todos serían nodos hoja). Su funcionamiento puede basarse en:
  - se eliminan de la lista de nodos aquellos dos con menor peso
  - se combinan mediante la creación de un nodo intermedio
  - este nodo se inserta en la lista de nodos por combinar. La inserción de realizarse de forma que se preserve el orden (según el peso)
  - función **repetir**, 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 (usando la característica llamada **currying**):

```
1 repetir(singleton,combinar)(listaNodos)
```

    donde el argumento listaNodos tendría tipo $List[Nodo]$
- usando este conjunto de funciones se implementará la funcionalidad de creación del árbol de codificación

4.3 Funcionalidad a probar

4.3.1 Decodificación

El software debe ofrecer la posibilidad de decodificar una lista de 0's y 1's (almacenada en un String) que ha sido codificada mediante un árbol específico. Es decir, la función de decodificación recibirá como argumento un código (una cadena formada exclusivamente 0 y 1) y devuelve a su vez otra cadena con los caracteres que se corresponden con el código. Se muestra a continuación la estructura del árbol representado en el gráfico anterior:

```
val codigo : Nodo = Interno( AHDGCEFB
    Hoja(A, 8)
    Interno( HDGCEFB
        Interno( HDGC
            Interno( HD
                Hoja(H, 1)
                Hoja(D, 1),
            2)
        Interno( GC
            Hoja(G, 1)
            Hoja(C, 1), 2),
        4)
    Interno( EFB
        Interno( EF
            Hoja(E, 1)
            Hoja(F, 1),
        2)
    Hoja(B, 3), 5),
    9),
17)
```

Hay que tener en cuenta que el tipo asociado a *codigo* es el correspondiente al nodo abstracto (supertipo de nodo). Este nodo sería el nodo raíz del árbol de codificación. Usando este árbol, el código siguiente

```
val mensajeSecreto: String = "011110111001"
```

se debe descodificar produciendo la cadena *"ABCD"*.

4.3.2 Codificación

También se debe implementar la funcionalidad para codificar, que recibirá como argumento una cadena con el contenido del mensaje y genera la cadena (ceros y unos) que representa la codificación del mensaje. En el ejemplo anterior, la codificación de la cadena *"ABCD"* debe generar la secuencia incluida arriba en el fragmento de código, almacenada en el valor llamado **mensajeSecreto**.

Una posible forma de codificar un mensaje consiste en ir separando el primer carácter del mensaje y obtener su codificación sobre el árbol (si se ha llegado a un nodo hoja no hay que agregar nada al código; en caso contrario se selecciona el nodo hijo que contiene al carácter y se agrega al código un 0 en caso de haberse seleccionado el izquierdo y 1 en caso contrario). Posteriormente la codificación debe seguir trabajando con el resto de caracteres.

El esquema de funcionamiento 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:

```
type TablaCodigo=Map[Char, String]
```

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:

```
codificarConTabla(tabla : TablaCodigo)(caracter : Char) : String
```

La creación de la tabla puede hacerse visitando el árbol de codificación y deber dotarse al sistema de la posibilidad de convertir un árbol en una tabla de este estilo.

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

Esta estructura, y las operaciones vistas, deben usarse para implementar un método llamado de decodificación rápida, que recibirá como argumento el árbol de codificación (que se usa para la generar la tabla con la función anterior) y el texto a codificar.

4.4 Pruebas a realizar

4.4.1 Prueba inicial

La prueba inicial consistirá en comprobar la construcción del árbol de codificación simple incluida en este guión en la sección sobre representación (4.1.1.). Para ello el texto a usar para construir el árbol es $AAAAAAAABBBCDEFGH$. Si analizáis el árbol construido debe coincidir con el gráfico mostrado.

4.5 Prueba sobre español

La segunda prueba (más general) consiste en construir un árbol a partir de la cadena de caracteres leída del archivo **regenta.txt** (proporcionado también). Para determinar la codificación se filtran todos los caracteres del texto leído del archivo para quedarnos únicamente con los caracteres sin acentuar (en mayúscula y minúscula). Usando el árbol obtenido a partir de dicho texto la codificación de la frase **"La regenta de Benito Perez Galdos"** generará el mensaje incluido en el archivo de prueba y almacenado en **mensajeSecreto**. Obviamente, la decodificación del mensaje debe producir la cadena original.

La obtención de la cadena almacenada en el archivo puede hacerse con la siguiente función de utilidad (se requiere importar **scala.io.Source**):

```
/**
 * metodo de lectura de archivo que devuelve una cadena
 * con todos los caracteres alfabeticos a considerar
 * @param nombreArchivo
 * @param filtrar indica si se hace filtrado o se lee
 *                el archivo como tal
 * @return
 */
def leerArchivo(nombreArchivo : String, filtrar : Boolean) : String = {
  val contenido = Source.fromFile(nombreArchivo).getLines().mkString
  if (filtrar == true) {
     contenido.filter(caracter => (caracter >= 'a' && caracter <= 'z') ||
        (caracter >= 'A' && caracter <= 'Z')).mkString
  }
```

```
15      else contenido
16    }
17  }
```

Como se aprecia, se usa un argumento auxiliar para indicar si deben filtrarse caracteres o no. Para generar el árbol sí que debe realizarse filtrado. Si se usa esta función para leer el archivo de un código formado por 0's y 1's entonces no debe utilizarse el filtrado.

5 Observaciones

Al igual que en prácticas anteriores el código implementado debe superar un determinado conjunto de test de prueba que garanticen su correcto funcionamiento.

6 Defensa de la práctica y entrega del material

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. Además, se enviará correo a **mgomez@decsai.ugr.es** para concretar día y hora para la defensa. La fecha límite para la defensa de la práctica será la del día del examen de la asignatura.