Reto 4

Pablo Baeyens Fernández
José Manuel Muñoz Fuentes
Darío Sierra Martínez

Estructura de Datos

1. Introducción

En este reto se propone la obtención de un procedimiento para guardar un árbol binario de estructura y tipo de dato arbitrarios en disco y otro procedimiento para reconstruir correctamente ese árbol a partir de sus datos en disco, de forma que el tamaño en disco del árbol sea el mínimo posible.

1.1. Problema

Propondremos una estructura de fichero que tendrá el menor tamaño en disco posible cumpliendo las siguientes premisas:

- **La reconstrucción del árbol es correcta y unívoca.** El procedimiento no debe guardar dos archivos distintos al aplicarlo a un mismo árbol, y dos archivos iguales deben generar al mismo árbol.
- **Los datos se guardan en binario, y no como texto.** Las etiquetas del árbol se almacenarán en el archivo de forma similar a como lo hacen en memoria. Si son un tipo de dato simple, se guardarán como tal. Si son un puntero a un vector de datos que concluye en un elemento terminador, se guardarán estos elementos (incluyendo el terminador). Si son una estructura o un vector de datos de tamaño fijo, se guardarán como la yuxtaposición de los elementos de esa estructura.
- **Las etiquetas no se modifican.** No se aprovechará el contenido de las etiquetas para, modificando bits que no sean relevantes, introducir información en ellas. Por ejemplo, si el tipo de dato almacenado son enteros sin signo de 32 bits y se sabe que ningún elemento es mayor que $2^{30} - 1$, no se aprovecharán los dos primeros bits para incluir información en ellos.
- **No se aplica ningún algoritmo de compresión a los datos.** Existen algoritmos de compresión de datos que podrían ayudar a reducir el tamaño de los archivos, tanto en las etiquetas como en los datos adicionales que se incluyan en el fichero. Supondremos que su uso queda fuera del interés de este reto.
- **No se incluyen bits que identifiquen el tipo de archivo o garanticen su integridad.** No habrá cabecera en los archivos, ni se usará un número mágico, ni se añadirán bytes para forzar un tipo de suma de verificación con un valor concreto. Esto también queda fuera de las intenciones del reto. Por ello, será responsabilidad del operario saber si el archivo que está intentando abrir como árbol es realmente un árbol guardado con este formato.

Para ello daremos una estructura que prescindirá de centinelas y en su lugar almacenará al principio del archivo una **clave** que identificará unívocamente la estructura del árbol. Así, el reto se reduce a describir dos algoritmos que computen las aplicaciones $c : A \to B$, $d : c(A) \to A$, donde:

- A es el conjunto de los árboles binarios de un número finito de nodos.

- B es el conjunto de las cadenas de longitud finita formadas por 0 y 1.
- c y d son aplicaciones inyectivas.
- $d \circ c = 1_A$, es decir, d es una inversa por la izquierda de c.
- Para todo árbol $a \in A$, $|c(a)|$ (la longitud de la cadena) es mínima.

2. Estructura de fichero

El archivo se compondrá de la yuxtaposición de una **clave** y un bloque de **datos**.

- La **clave** será una lista de bits de longitud indefinida. Para cada nodo del árbol se añadirá un 1 a la lista si el nodo tiene un hijo a la izquierda o un 0 en caso contrario, y a continuación se añadirá un 1 si el nodo tiene un hijo a la derecha o un 0 en caso contrario. Este procedimiento se repite con los nodos hijos si existen (primero el de la izquierda y después el de la derecha), de forma que se almacena el par de bits de cada nodo en preorden.

  Para que la siguiente sección no comience en una posición intermedia de un byte, si al escribir la lista de bits en disco el número de elementos no es un múltiplo de 8, se añadirán ceros al final hasta que lo sea. (Esto en ningún caso incrementará el tamaño del archivo, puesto que, como el próximo bloque siempre tiene por tamaño un múltiplo de un byte, el sistema operativo añadiría bits al final del archivo hasta que el tamaño fuese un múltiplo de un byte).

- El bloque de **datos** lo formará el grupo de datos, almacenados en preorden sin espacios entre ellos. La posición donde empieza un nuevo dato se determinará a partir del tamaño de los tipos de datos (si son de tamaño fijo) o de la posición de un elemento terminador (si son de tamaño variable).

En principio este formato no es compatible con un posible árbol vacío, ya que presupone que el nodo raíz existe. Esto se solventa con facilidad haciendo que se guarden los árboles vacíos como archivos vacíos y considerando los archivos vacíos como árboles vacíos.

2.1. Tamaño

Con este formato, el tamaño es igual a la suma de los tamaños del contenido de las etiquetas más el tamaño de la clave. Puesto que hay dos bits para cada etiqueta y el tamaño de la clave siempre es un múltiplo de un byte y no de fracciones de un byte, este tamaño resulta ser de $\lceil \frac{n}{4} \rceil$ bytes, siendo n el número de etiquetas. Esto supone, por ejemplo, que si las etiquetas son números enteros, el tamaño del archivo se incremente en un $6,25\,\%$ respecto al tamaño que ocuparían exclusivamente las etiquetas.

3. Procedimientos

Tanto al leer como al escribir la clave del árbol, será necesario avanzar en bloques de dos bits en lugar de bloques de un byte. En una implementación de estos métodos se requerirá la simulación de entrada y salida en bloques más pequeños de un byte leyendo o extrayendo un byte y haciendo cuatro operaciones, una para cada grupo de dos bits, con el byte.

3.1. Lectura

A la hora de leer el archivo que almacena el árbol, en principio no se sabe dónde termina la clave y dónde empieza el bloque de etiquetas. Aunque es posible determinar esa posición leyendo el archivo para después generar simultáneamente la estructura del árbol y su contenido, optaremos en lugar de eso por obtener en primer lugar la estructura y después rellenar las etiquetas con los valores correctos, lo que permitirá leer el archivo de forma lineal, sin saltos.

Para obtener la estructura del árbol se partirá de un árbol con un único nodo (a no ser, claro está, que el archivo esté vacío) y se leerán los bits de dos en dos. En cada par de bits, si el primero de ellos es un 1, se le añadirá al nodo actual un hijo a la izquierda; y si el segundo es un 1, se le asignará un hijo a la derecha. Tras añadir (o no) los hijos, se pasa al siguiente nodo en preorden. Este procedimiento sigue hasta que se llegue a un nodo vacío.

Una vez generada la estructura, se lee el bloque de datos, asignando a cada nodo la etiqueta encontrada en el archivo, recorriendo los nodos del árbol en preorden. A la hora de leer la primera etiqueta, si el último par de bits leído no constituía los dos últimos bits de un byte, se pasará al siguiente byte, puesto que el bloque de datos empezará en el comienzo de un byte y no en una posición intermedia.

A continuación exponemos el algoritmo que genera la estructura. Inicialmente le pasamos el nodo raíz (que se generará de antemano a no ser que el archivo sea vacío, en cuyo caso este algoritmo no se ejecutará) y la posición 0. Tanto el nodo como la posición serán tratados por referencia por el algoritmo. La posición debe tratarse por referencia porque, hasta que no se haya generado completamente la estructura que surge del hijo a la izquierda, no se conocerá la posición en la que comienza la información a partir del hijo a la derecha.

`AvanzaArmazon`(n, s, p):
Entrada: n, nodo actual; s, cadena a leer, p; posición en la cadena

`izq` := $s[p] ==$ '1';
`der` := $s[p+1] ==$ '1';

si *izq* **entonces**
 | **Inserta** hijo izquierdo a n
fin
si *der* **entonces**
 | **Inserta** hijo derecho a n
fin

p += 2;

si *izq* **entonces**
 | `AvanzaArmazon`$(n.\text{izquierdo}, s, p)$
fin
si *der* **entonces**
 | `AvanzaArmazon`$(n.\text{derecho}, s, p)$
fin

El bloque de datos se leería a continuación, a partir de la posición inmediatamente posterior al último byte del archivo que correspondía a la clave.

3.2. Escritura

Para guardar el árbol simplemente escribiremos en un archivo la clave del árbol seguida del bloque de datos.

Para cada nodo, se añadirá un 1 a la clave si tiene un hijo a la izquierda o un 0 en caso contrario, y se añadirá un 1 si tiene un hijo a la derecha o un 0 en caso contrario. Si el hijo a la izquierda existía, se repetirá este proceso en el hijo a la izquierda; y si el hijo a la derecha existía, se procederá igual con este hijo a la derecha (después de terminar con el hijo a la izquierda si este existía). De esta forma se recorren los nodos en preorden y se rellena la cadena.

Después se rellena el bloque de datos (saltando al siguiente byte si el último byte en el que se escribió no quedó completo) leyendo las etiquetas en preorden.

Así queda el algoritmo que escribe la clave. Inicialmente lo llamaremos con el nodo raíz y una cadena vacía (que será modificada por referencia) como parámetros. Para simplificar su exposición, definiremos la operación `+=` entre cadenas como la concatenación o anexión a la primera cadena, en su final, de la segunda cadena.

`GuardaBits`(n, s):
Entrada: n, nodo actual; s, cadena

si *n tiene hijo* ***izquierdo*** **entonces**
| s += '1'
en otro caso
| s += '0'
fin
si *n tiene hijo* ***derecho*** **entonces**
| s += '1'
en otro caso
| s += '0'
fin

si *n tiene hijo* ***izquierdo*** **entonces**
| `GuardaBits`(n.izquierdo, s)
fin
si *n tiene hijo* ***derecho*** **entonces**
| `GuardaBits`(n.derecho, s)
fin

A continuación se escribiría en el archivo el contenido de las etiquetas, en preorden. Si el árbol resultaba ser vacío, se escribirá un archivo vacío y no se ejecutará este algoritmo ni la escritura de etiquetas posterior.

3.3. Lectura de información del árbol

Es posible obtener cierta información de la estructura del árbol simplemente leyendo la clave, sin generar el árbol completo:

- **Offset:** Es posible determinar la posición donde comienza el bloque de datos en el archivo. Para ello, fijamos una variable entera con signo a 1, y por cada byte de archivo le sumamos el peso de Hamming de ese byte (es decir, el número de bits que toman el valor 1) y le restamos 4, hasta que la variable tome un valor no positivo. Cuando esto ocurra, el byte que debería leerse a continuación es el mismo byte donde comienza el bloque de datos.

  La variable representa el número de nodos que están pendientes de ser añadidos al árbol a cada momento. Cuando esta llega a 0, no quedan más nodos que añadir, por lo que lo siguiente será el bloque de datos.

- **Tamaño:** Puede obtenerse el número de nodos del árbol incluso si el tamaño del tipo de dato es variable (si fuese fijo, sería aún más fácil de obtener a partir del tamaño del archivo). Esto se puede consguir sumando el offset (según se obtuvo en el procedimiento anterior y contando el inicio de archivo como la posición 0) multiplicado por 4 al valor que tomó la variable al ejecutar el procedimiento anterior.

  Al hacer esto se está anulando la resta de 4 por cada byte al número de nodos pendiente de ser añadidos a cada paso. Por ello, se obtendrá el número total de nodos.

4. Implementación

Para probar la solución propuesta se adjunta una implementación de dos clases que aplican los procedimientos descritos y un código en `Prueba.cpp` que, importando las clases (que están en `ArbolClave.cpp`), obtiene un árbol de gran tamaño con etiquetas y estructura al azar, lo guarda según este procedimiento, lo lee

de nuevo y verifica que el árbol leído de disco y el que se había guardado coinciden. El árbol generado no se borra tras la ejecución del programa, por lo que puede explorarse con un editor hexadecimal tras la ejecución.

El programa puede modificarse para que el tipo de dato usado para las etiquetas o su máximo valor posible sea distinto. En el ejemplo se ha escogido el tipo de dato **int** y un máximo de **100** para que, al leer el archivo con el editor hexadecimal, sea evidente dónde termina la clave inicial y dónde comienzan las etiquetas (los tres primeros bits de cada etiqueta tienen el valor `00`). También puede modificarse la función que decide si se añadirá o no un nuevo elemento, para que el número de componentes sea menor o mayor. Según está en el ejemplo, el árbol rondará las **175 000 componentes**, pero reduciendo o aumentando la constante 10,0 de la función `NuevoElemento` se puede reducir o aumentar, respectivamente, el número de componentes.

La implementación del tipo de dato de árbol binario, en `ArbolBinario.h`, es la misma que se ha puesto como ejemplo en la plataforma de la asignatura, con cambios menores.