Estructuras de Datos y Algoritmos – IIC2133

I1

16 abril 2015

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea, en que cada paso es descrito con la precisión suficiente, p.ej., en un pseudo lenguaje de programación similar a C, para que un computador lo pueda ejecutar.)

1. *Estructuras de datos básicas.*

a) Describe una **estructura de datos** que funcione en memoria principal para majenar una pequeña biblioteca en nuestro departamento. La biblioteca tiene libros y tiene lectores. Los lectores son los estudiantes, profesores y funcionarios del departamento, todos con nombre y RUT. Los libros tienen título y tienen autor (el nombre de una persona). Tanto los lectores como los libros permanecen fijos durante el semestre (marzo a julio, agosto a diciembre). Al bibliotecario le interesa poder prestar libros a los lectores, recibir de vuelta libros que estaban prestados, y conocer la información habitual eficientemente: ¿Qué lectores hay registrados en la biblioteca? ¿Qué libros tiene la biblioteca? ¿Cuáles de ellos están disponibles y cuáles están prestados? ¿Cuál lector tiene un determinado libro prestado? ¿Cuáles libros están prestados a un determinado lector? Tu estructura puede incluir **arreglos**, **listas ligadas** en varias direcciones, y **tablas de hash**. **Justifica**.

Podemos manejar tanto los libros como los lectores en arreglos, en que cada casillero tiene varios campos; puede haber un tercer arreglo para los autores de los libros. Todos estos arreglos son en realidad tablas de hash; las claves son los RUT's de los lectores, los títulos de los libros, y los nombres de los autores. Los casilleros del arreglo de los autores son punteros a una lista ligada de punteros a cada uno de los libros (casilleros del arreglo de libros) del autor correspondiente. Para saber qué lectores y qué libros hay, simplemente recorremos los arreglos corresppndientes. Además, hay que representar los préstamos.

(Podríamos usar una tabla de doble entrada, es decir, una matriz, en que, p.ej., las columnas son los libros y las filas los lectores. Cada casilla de la matriz correspondería a un boolean, para indicar si el libro está prestado —1— o no —0— a un determinado lector. Si hay 500 lectores y 1,000 libros, la martiz tendría 500,000 casillas; pero a lo más 1,000 de estas casillas podrían tener un 1, cuando todos los libros están prestados.)

Siguiendo la sugerencia del enunciado, representamos los préstamos por listas ligadas a partir de los lectores. Cada elemento de una lista ligada es un puntero a un libro: el libro que está prestado al lector correspondiente. Como un lector puede tener varios libros prestados, ponemos todos esos elementos en una lista doblemente ligada. Además, para saber a qué lector está prestado un determinado libro, cada elemento de las listas ligadas es apuntado "de vuelta" por el libro correspondiente.

b) La evaluación computacional de expresiones aritméticas *infix* simples, en que cada operador aparece entre sus dos argumentos, tal como

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

puede hacerse a partir de representar la expresión en notación *postfix*: cada operador aparece *después* que sus dos argumentos, en lugar de entre ellos, y los paréntesis no son necesarios:

$$5\ 9\ 8\ +\ 4\ 6\ *\ *\ 7\ +\ *$$

Describe un **algoritmo**, para evaluar expresiones postfix simples, como la anterior. Tu algoritmo debe leer la expresión elemento a elemento de izquierda a derecha: si es un número, debe guardarlo para procesarlo más adelante; si es un operador, debe realizar la operación correspondiente con los números guardados, y debe guardar el resultado para procesarlo más adelante (si fuera necesario). ¿Cuál es la **complejidad** de tu algoritmo en función del número de elementos de la expresión? **Justifica**.

[**50%**] El algoritmo está prácticamente descrito en el enunciado. Supongamos que la expresión viene en un archivo (o en un arreglo) de elementos de algún tipo (o clase) **T**, y que tenemos un stack **s** inicialmente vacío. La clase **T** cuenta con métodos para saber si un objeto de tipo **T** es un número (en ese caso se puede calcular su valor) o un operador ('+' o '*').

```
while (!file.eof())
   item = file.read()
   if (item.isNumber())
      s.push(item.value())
   if (item.isPlus())
      s.push(s.pop()+s.pop())
   if (item.isTimes())
      s.push(s.pop()*s.pop())
print(s.pop())
```

[**50%**] El algoritmo es O(*n*), en que *n* es el número de elementos en la expresión. Leemos cada elemento de la expresión una vez, y, por cada elemento leido, realizamos un número constante de pasos: primero, averiguamos el tipo del elemento (tres o cuatro opciones); y luego, hacemos uno o dos push's o pop's.

2. *Colas priorizadas*.

a) Compara las implementaciones de una cola priorizada usando un **heap binario** y usando un **árbol de búsqueda binario**. Considera el caso en que la cola está inicialmente vacía y a continuación ocurren n operaciones de inserción de claves y n operaciones de extracción del elemento con la mayor clave, intercaladas arbitrariamente (por supuesto, la primera operación es una inserción y la última una extracción). ¿Cuántos pasos, en notación O( ), se ejecutan en cada una de las dos implementaciones? Considera tanto el mejor caso como el peor caso para ambas implementaciones. **Justifica**.

La intercalación consistente en alternar una inserción y una extracción —es decir, una inserción, una extracción, una inserción, una extracción, una inserción, etc.— es el mejor caso para ambas estructuras: la cola nunca tiene más de un elemento y el total de $2n$ operaciones toma $O(n)$ pasos.

El pero caso se da cuando primero se ejecutan las n inserciones y después las n extracciones. El heap tiene la gracia de estar siempre balanceado, por lo que su altura está acotada por $O(\log n)$, en cambio el árbol puede convertirse en una lista ligada, por lo que su altura está acotada por $O(n)$. De modo que en el peor caso el heap ejecuta $O(n \log n)$ pasos, mientras que el árbol puede llegar a ejecutar $O(n^2)$ pasos.

b) Tenemos k fuentes de datos. Los datos de cada fuente vienen ordenados de mayor a menor. Queremos enviar la totalidad de los datos recibidos por un único canal de salida, también ordenados de mayor a menor. El dispositivo electrónico que hace esta mezcla tiene una capacidad limitada de memoria. Describe un **algoritmo** para este dispositivo, que le permita hacer su tarea empleando un arreglo a de k casilleros, en que cada uno tiene dos campos: $a[j].data$ puede almacenar un dato, y $a[j].num$ puede almacenar un número entero entre 1 y k. Para recibir un dato desde la fuente i, se ejecuta $receive[i]( )$, y para enviar un dato x por el canal de salida, se ejecuta $send(x)$. Si la cantidad total de datos en las k fuentes es n, ¿cuál es la cantidad total de pasos que ejecuta tu algoritmo, en notación O( )?

[**67%**] La idea es armar un (max-)heap binario de tamaño k, inicialmente con el primer dato (el mayor) de cada fuente, de modo que en la raíz quede el mayor de todos los datos:

```
for (i = 0; i < k; i = i+1)
    x.data = receive[i]()
    x.num = i
    insertObject(x)
```

A continuación, hay que ir sacando el dato que está en la raíz, enviándolo por el canal de salida, y reemplazándolo en el heap por el próximo dato recibido de la misma fuente de donde provenía el que salió:

```
for (j = 0; j < n-k; j = j+1)
    x = xMax()
    i = x.num
    send(x.data)
    x.data = receive[i]()
    if (x.data ≠ null) insertObject(x)
```

[**33%**] $O(n \log k)$, ya que el `for` ejecuta k operaciones `insertObject(x)`, c/u de las cuales ejecuta un número de pasos proporcional a $\log k$, lo que da un subtotal de $k \log k$); y el `while` ejecuta $n–k$ operaciones *insertObject(x)*, c/u de las cuales nuevamente ejecuta un número de pasos proporcional a $\log k$, lo que da otro subtotal de $(n–k)\log k$.

3. *Árboles de búsqueda.*

a) La especificación de un árbol de búsqueda (no necesariamente balanceado) indica que cada nuevo elemento que se inserta debe quedar como la raíz del árbol. ¿En qué situaciones puede ser esto útil? ¿Qué tan eficientemente, en notación O( ), puede implementarse una operación `Insertar-Nodo()` que logre este objetivo? **Justifica**.

[**50%**] La raíz del árbol es el elemento que se encuentra con una comparación. Por lo tanto, me conviene que un nuevo elemento que acabo de insertar quede como raíz si a continuación lo voy a buscar varias veces; y si en el mediano plazo la probabiidad de buscar ese elemento se mantiene alta, ya que cuando insertemos otro elemento, que va a quedar como nueva raíz, la raíz anterior va a quedar como hija de la nueva raíz (o sea, se va a mantener en la parte "de arriba" del árbol por un tiempo).

[**50%**] Para lograr esto, `Insertar-Nodo()` debe hacer lo siguiente. Primero, una inserción "normal", en que el nodo insertado queda como hoja del árbol. Luego, "hacer subir" este nodo mediante rotaciones simples hasta dejarlo en la raíz del árbol: si el nodo es un hijo izquierdo, entonces se hace una rotación a la derecha, y viceversa. Recordemos que las rotaciones preservan la propiedad de árbol de búsqueda. Con cada rotación, el nodo "sube" un nivel en el árbol; por lo tanto, se necesita un número de rotaciones igual a la altura del árbol para que el nodo finalmente quede como raíz, es decir, O(*altura de árbol*).

b) Considera un **árbol AVL** inicialmente vacío, en el que queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. ¿Es posible insertar estas claves en el árbol en algún orden tal que **nunca** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave *k* tal que el número de claves menores que *k* —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que *k* —que van a ir a parar al subárbol derecho— sean muy parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

c) Considera un **árbol rojo-negro** que corresponda a un **árbol 2-3**, tal como lo vimos en clase. Escribe el **algoritmo** que debe seguir el procedimiento de inserción en el árbol rojo-negro de manera que corresponda al procedimiento que seguiría en el árbol 2-3.

Correspondencia básica. En un árbol rojo-negro, un nodo rojo contiene la clave izquierda (menor) y los hijos izquierdo y del medio de un nodo 3 del árbol 2-3. La clave derecha y el hijo derecho del nodo 3 se convierten en la clave e hijo derecho de un nodo negro en el árbol rojo-negro; el hijo izquierdo de este nodo negro es el nodo rojo anterior. Todos los otros nodos 2 del árbol 2-3 corresponden a nodos negros en el árbol rojo-negro.

Dividimos la respuesta en tres partes.

a) Al insertar en el árbol 2-3, el caso más simple es cuando insertamos en un nodo 2 (que es una hoja), que así se convierte en un nodo 3 (y sigue siendo una hoja).

En el correspondiente árbol rojo-negro, insertamos un hijo a un nodo negro. La clave del hijo puede ser menor o mayor que la clave de su padre, es decir, puede quedar como hijo izquierdo o hijo derecho, respectivamente; además, el hijo debe ser pintado de rojo. Sin embargo, estos dos nodos deben quedar finalmente en una configuración que corresponda al nodo 3 (según la descripción de arriba). Esto será así si el hijo insertado es un hijo izquierdo. Pero si es un hijo derecho, es necesario hacer una rotación a la izquierda.

b) Un caso menos simple es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre en un nodo 2; supongamos que el otro hijo de este nodo 2 también es un nodo 2. Es decir, en el árbol rojo-negro, insertamos en una configuración de un nodo negro con un hijo izquierdo rojo y sin hijo derecho (corresponde al nodo 3); el padre del nodo negro también es negro (corresponde al nodo 2) y tiene otro hijo negro (el otro nodo 2). Dependiendo del valor de la clave insertada, el nuevo nodo puede ir a parar como hijo izquierdo o derecho del nodo rojo, o como el hijo derecho, hasta ahora inexistente, del nodo negro: configuración inicial, justo después de la inserción.

En el árbol 2-3 el resultado es el siguiente: el nodo 2 se convierte en un nodo 3 (con dos claves y tres hijos) y el nodo 3 se convierte en tres nodos 2 (con una clave cada uno y sin hijos).

Por lo tanto, en el árbol rojo-negro la configuración final debe ser un nodo negro con dos hijos: el izquierdo rojo y el derecho negro; el hijo izquierdo rojo a su vez tiene dos hijos negros. A través de una o dos rotaciones, a uno u otro lado, y algunos cambios de colores, convertimos la configuración inicial en esta configuración final.

c) Finalmente, el caso más complicado es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre también es un nodo 3. En el árbol rojo-negro correspondiente, la inserción es en una configuración de un nodo negro con dos hijos, izquierdo rojo y derecho negro, en que el hijo izquierdo tiene a su vez dos hijos negros, uno de los cuales, el izquierdo, tiene finalmente un hijo rojo. El nuevo nodo va a ir a parar inicialmente como hijo izquierdo o derecho de este nodo rojo, o como hijo derecho de su padre negro, y en cualquier caso pintado de rojo. En la configuración final, las hojas y sus padres son negros; pero la parte complicada se da porque alguna clave sube de nivel, potencialmente repitiendo el problema original dos niveles más arriba. Por supuesto, este problema se puede resolver aplicando recursivamente el algoritmo descrito (pero esto es más fácil decirlo que especificarlo precisamente).