Estructuras de Datos y Algoritmos – IIC2133

I2

9 mayo 2016

Nota: *Cuando se pide un* ***algoritmo****, se espera algo parecido a lo siguiente*:

```
selectionSort(a):
   for k = 0 … n-1:
      min = k
      for j = k+1 … n:
         if a[j].key < a[min].key:
            min = j
      exchange(a[k], a[min])
```

1. Un **árbol B⁺** (una variante de los *árboles B*) de orden m tiene las siguientes propiedades:

i) Los datos están almacenados únicamente en las hojas, ordenados por clave.

ii) Los nodos que no son hojas almacenan hasta $m-1$ claves (ordenadas, para guiar la búsqueda); la clave i-ésima representa la clave más pequeña en el subárbol $(i+1)$-ésimo.

iii) La raíz es ya sea una hoja o tiene entre dos y m hijos.

iv) Todos los nodos que no son hojas (excepto la raíz) tienen entre $\lceil m/2 \rceil$ y m hijos.

v) Todas las hojas están a la misma profundidad y tienen entre $\lceil t/2 \rceil$ y t datos, para algún t.

a) [**3**] Describe un algoritmo eficiente para insertar un nuevo dato en (una hoja de) un árbol B⁺.

Se puede emplear el mismo algoritmo "defensivo" del árbol B visto en clase: Si el nodo por el que vamos pasando, en nuestro camino de la raíz a la hoja donde se va a hacer la inserción, está lleno (es decir, tiene $m-1$ claves), entonces lo dividimos en dos antes de descender al hijo correspondiente. Al llegar a la hoja, si ésta tiene espacio (es decir, tiene menos que t datos), entonces simplemente insertamos el nuevo dato allí. En caso contrario, primero dividimos la hoja en dos, dejando en cada hoja nueva la mitad de los datos de la hoja original, y luego insertamos el dato en la hoja nueva correspondiente según la clave del dato. En todo este proceso, cada vez que dividimos un nodo, ya sea una hoja o no, hay que agregar un hijo más al padre del nodo; esto no es problema, porque cuando pasamos por el padre (cuando veníamos bajando desde la raíz) nos aseguramos de dejarlo con espacio: si estaba lleno, lo dividimos en dos.

También es válido el algoritmo más directo, que primero simplemente busca (a partir de la raíz) la hoja donde tiene que hacer la inserción. Una vez allí, si la hoja tiene menos que t datos, entonces simplemente inserta el dato y termina; pero si la hoja ya tiene t datos, entonces (al igual que en el algoritmo anterior) primero divide la hoja en dos, dejando en cada hoja nueva la mitad de los datos de la hoja original, y luego inserta el dato en la hoja nueva que corresponda. En este último caso, tiene que agregar un hijo adicional al padre de la hoja original. Si este nodo tiene menos que $m-1$ claves, entonces agregar el hijo adicional no es problema; pero si ya tiene $m-1$ claves, entonces hay que dividirlo en dos y agregarle recursivamente un hijo adicional a su padre. Este proceso puede llegar hasta la raíz.

En cualquiera de los dos algoritmos, si hay que dividir la raíz, entonces se la divide y los dos nodos resultantes pasan a ser hijos de una nueva raíz. Por esto está la propiedad iii) y la excepción mencionada en la propiedad iv).

Ambos algoritmos son $O(\log n)$, ya que hacen un número constante de pasos por cada nivel del árbol; la complejidad de dividir un nodo es independiente de n. La diferencia es que el primer algoritmo hace una sola "pasada", potencialmente más lenta, por los niveles del árbol (hacia abajo, de la raíz a una hoja), mientras que el segundo hace inicialmente una pasada rápida hacia abajo, pero podría tener que hacer una segunda pasada hacia arriba.

b) [**2**] Si tuvieras que imprimir todos los datos almacenados en la base de datos, ordenados de menor a mayor clave, ¿cuál de los dos tipos de árboles —árbol-B o árbol-B⁺— es más apropiado? Justifica.

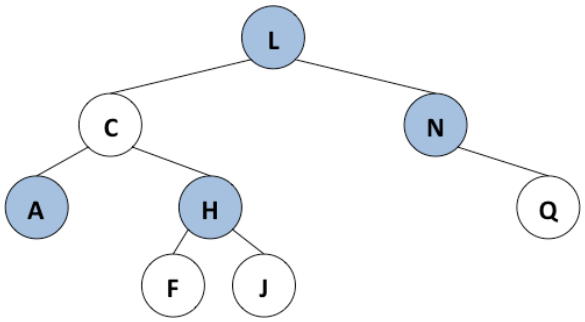
El árbol B⁺, ya que como todos los datos están en las hojas, basta recorrer éstas secuencialmente en orden. Por supuesto, esto requiere conectar las hojas entre ellas, para que cuando se termina de recorrer una, se pueda pasar directamente a la que la sigue (en orden), sin tener que ir a buscarla a los nodos padres: es decir, cada hoja tiene un puntero a la hoja que está inmediatamente a su derecha (en la representación gráfica del árbol).

2. Consideremos **árboles rojo-negros** definidos según las 4 propiedades enunciadas en clase (es decir, como una representación alternativa de árboles 2-3-4, y no simplemente de árboles 2-3).

a) [**1**] Describe un árbol rojo-negro con n claves que presenta la mayor razón posible de nodos rojos a nodos negros; justifica. ¿Cuál es esta razón?

Ya que los nodos rojos no pueden tener hijos rojos, el árbol pedido es uno en que cada nodo negro tiene dos hijos rojos, en cuyo caso la razón es 2 a 1.

b) [**3**] Considera el árbol rojo-negro que se muestra a la derecha, en que los nodos oscuros (*A*, *H*, *L*, *N*) son negros. Inserta la clave *D* y explica, muestra y ejecuta los pasos necesarios hasta volver a tener un árbol rojo-negro válido.



Al insertar *D*, queda como hijo izquierdo rojo de *F* (aquí tiene que haber un dibujo).

Como *F* también es rojo, tenemos un problema. Como *J*, el hermano de *F*, es rojo [caso descrito en la diap. 74], resolvemos el problema (en este nivel) intercambiando colores: *H* queda rojo, *F* y *J*, negros (dibujo).

Como *C*, el padre de *H*, es rojo, nuevamente tenemos un problema [caso mencionado en la diap. 76]. Como *A*, el hermano de *H*, es negro, no podemos simplemente intercambiar colores como antes; en cambio, hacemos una rotación simple a la izquierda en torno a *C*–*H*, sin cambiar colores [caso descrito en la diap. 70]: *H*, rojo, queda en lugar de *C*, también rojo, que queda como hijo izquierdo de *H*; *F*, negro, queda como hijo derecho de *C*, y *J*, negro, sigue como hijo derecho de *H* (dibujo).

Como *H* y *C*, padre e hijo, son rojos, y *N*, el hermano de *H*, es negro [caso descrito en la diap. 70], hacemos una rotación simple a la derecha en torno a *L*–*H*: *H* queda en la raíz y lo pintamos negro; *L* queda como hijo derecho de *H* y lo pintamos rojo; y *J* pasa a ser el hijo izquierdo de *L* (dibujo).

c) [**2**] En clase vimos —a propósito de los árboles AVL— que las rotaciones simples preservan la propiedad de árbol binario de búsqueda. ¿Qué pasa en el caso de los árboles rojo-negro con respecto a las propiedades adicionales, sobre el número de nodos negros y sobre nodos rojos consecutivos, en una ruta simple desde la raíz hasta una hoja? Demuestra que estas propiedades son preservadas por las rotaciones simples, o bien da ejemplos que muestren que no es así.

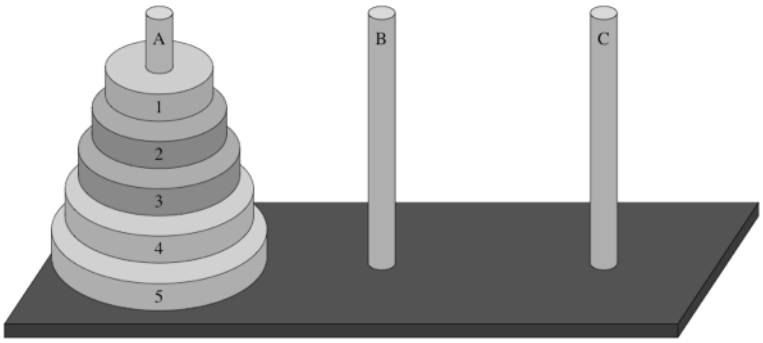
Las rotaciones simples no preservan ninguna de estas dos propiedades. Al poner al hijo en el lugar del padre, si el hijo es rojo (y su padre, negro), puede quedar como hijo de otro nodo rojo; p.ej., en **b**), al hacer una rotación a la derecha en torno a *H*–*F*, queda *F* como hijo derecho de *C*. En este mismo caso, el nodo negro, que estaba como raíz de un subárbol, aportando al número de nodos negros en todas las rutas hasta las hojas, ahora bajó un nivel y fue reemplazado por un nodo rojo, por lo que el número de nodos negros en algunas de esas rutas disminuyó en uno; p.ej., en **b**), al hacer una rotación a la derecha en torno a *L*–*C*, la ruta de *C* a *A* tiene un solo nodo negro, pero las otras (*C* a *F*, *C* a *J*, *C* a *Q*) tienen dos.

3. [**5**] Considera una **tabla de hash** en que las colisiones se resuelven mediante encadenamiento: Indica tres estructuras de datos fundamentalmente diferentes para implementar este "encadenamiento" (una de ellas puede ser listas doblemente ligadas). Explica de manera breve y precisa las ventajas y desventajas principales de cada una.

En lugar de una lista ligada, en que la inserción se hace siempre al comienzo de la lista, pero hay que buscar secuencialmente, se podría usar un árbol binario de búsqueda, e incluso una nueva tabla de hash. El árbol permite insertar, buscar y eliminar en tiempo proporcional al logaritmo del número de ítemes que hay en el árbol; esto es en promedio, o bien en el peor caso si el árbol se mantiene balanceado. La tabla permite buscar, insertar y eliminar en tiempo constante, en promedio.

4. En el problema conocido las *torres de Hanōi*, hay 3 agujas verticales, llamadas *A*, *B* y *C* (ver figura). En la aguja *A* hay una torre de *n* discos, todos de distintos tamaños, ordenados hacia arriba de más grande a más pequeño (en la figura, $n = 5$). El problema consiste en mover la torre entera de discos de la aguja *A* a la aguja *B*, usando la aguja *C* como auxiliar, siguiendo las siguientes reglas:

- Sólo se puede mover un disco a la vez.
- Sólo se puede mover el disco de más arriba de una de las torres y depositarlo encima de otra torre.
- Ningún disco, en ningún movimiento, puede ser depositado sobre otro más pequeño.



a) Escribe la ecuación de recurrencia que resuelve el problema de mover la torre de *n* discos de la aguja *A* a la aguja *B* [0.5]; calcula la complejidad de resolver esta ecuación recursivamente [0.5].

b) Explica cómo usar programación dinámica para mejorar la solución en **a**) [2.5]; calcula la complejidad usando este método [0.5].

c) Si pudiéramos llevar a cabo una secuencia arbitraria de movimientos de manera instantánea, ¿cómo se vería afectada la complejidad de la solución en **a**)? [0.5] ¿Y la de la solución en **b**)? [0.5].

En tus respuestas, considera tanto las complejidades de M(*n*), el costo de mover los discos, y H(*n*), el costo de calcular la solución. La complejidad del problema es la de H + M.