Estructuras de Datos y Algoritmos – IIC2133

I1

31 agosto 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. Considera un conjunto de n elementos, de tipo numérico, distintos (es decir, dados dos elementos, a y b, cualesquiera que pertenecen al conjunto, se tiene que $a < b$ o $a > b$). Queremos encontrar la **mediana** del conjunto, es decir, el elemento m tal que la cantidad de elementos menores que m es igual a la cantidad de elementos mayores que m. Por supuesto, podríamos ordenar los n elementos y luego mirar el que queda en la posición $\lceil n/2 \rceil$ (suponemos que n es impar), pero posiblemente vamos a estar haciendo mucho más trabajo que el estrictamente necesario (ya que no nos interesa saber la posición relativa de ninguno de los otros n–1 elementos). Considera además que los elementos los vamos a ir leyendo uno a uno en un dispositivo móvil en que el uso eficiente de la memoria también es importante.

a) Describe una solución eficiente a este problema, a partir de las estructuras de datos estudiadas en clase: *heaps* binarios, tablas de *hash*, y/o árboles de búsqueda binarios.

La idea es mantener en todo momento un conjunto S con los $n/2$ elementos más grandes leídos hasta el momento. Una vez que los primeros $n/2$ elementos han sido leídos, cuando leemos un nuevo elemento, x, lo comparamos con el elemento más pequeño que hay en S, que denotamos por S_{min}. Si x es más grande, entonces reemplaza a S_{min} en S. S ahora tiene un nuevo elemento más pequeño, que puede ser o no x. Al finalizar el input, encontramos el elemento más pequeño en S y lo devolvemos como la respuesta.

Para que esta idea sea eficiente, usamos un min-heap para implementar S. Los primeros $n/2$ elementos los ponemos en el heap en tiempo $O(n/2)$ en total. El tiempo para procesar cada uno de los otros elementos tiene dos componentes: $O(1)$, para probar si el elemento va a parar a S, más $O(\log n/2)$, para eliminar S_{min} e insertar el nuevo elemento, si esto es necesario. Así, el tiempo total es $O(n/2 + n/2(\log n/2)) = O(n/2(\log n/2)) = O(n \log n)$.

b) Determina el número aproximado de operaciones o pasos individuales y la cantidad de celdas de memoria, ambos en función de n, que requiere tu solución en el peor caso.

2. Un supermercado quiere desarrollar una aplicación que le permita manejar en memoria principal información acerca de la venta de los productos a los clientes con tarjeta durante un fin de semana particular. El supermercado quiere llevar un registro de las cantidades vendidas, pero también quiere saber rápidamente, p.ej., en cuáles productos ha comprado un cliente y qué clientes han comprado un cierto producto. Inicialmente, se pensó en emplear una tabla, en que las filas representan a los productos y las columnas a los clientes. Pero luego quedó claro que desde el punto de vista de uso de memoria la tabla no es muy eficiente: si el supermercado tiene 20,000 clientes con tarjeta y ofrece 20,000 productos durante el fin de semana, la tabla tendría que tener unas 20,000 × 20,000 = 400,000,000 casillas; pero si sólo la mitad de los clientes van a comprar el fin de semana y cada uno compra en promedio 40 productos, entonces sólo se estaría usando el uno por mil de las casillas.

a) Describe una estructura de datos basada en listas ligadas para resolver este problema más eficientemente, en que sólo sea necesario ocupar una cantidad de memoria del orden de los cientos de miles de casillas (10,000 × 40 = 400,000), y no de millones; preferentemente, dibuja la estructura de datos.

Por cada cliente que compra un producto, usamos un registro (objeto) de 5 campos: una identificación del cliente, una identificación del producto, la cantidad vendida (de ese producto a ese cliente), un puntero (imaginémoslo vertical hacia abajo) al objeto que representa otro producto comprado por el mismo cliente, y un puntero (imaginémoslo horizontal hacia la derecha) al objeto que representa a otro cliente que compra el mismo producto.

b) Suponiendo que ni los clientes ni los productos cambian durante el fin de semana (es decir, ni se agregan ni se eliminan), ¿cómo debiera funcionar tu estructura de datos para que el usuario de la aplicación pueda referirse a los productos por sus códigos (p.ej., los que se usan al pasarlos por la caja) y a los clientes por sus RUT's?

Usamos dos arreglos de tamaño 20,000, uno para los clientes y otro para los productos. Cada casilla de estos arreglos tiene dos punteros: uno a la información del cliente (rut, nombre) o del producto (código, nombre), respectivamente, y otro a uno de los objetos descritos en **a**). Usamos sendas funciones de hash para convertir rut's o códigos de productos en índices en estos arreglos, y usamos estos índices como los identificadores de los clientes y de los productos en los objetos de **a**).

3. En el caso de hashing con direccionamiento abierto, considera que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en los siguientes casos:

a) Usamos revisión lineal, de modo que la posición intentada es $(h(K) + i)$ modulo T.

b) Usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$, todos divididos módulo T.

	a)	**b**)
0	B_9	B_9
1		B_2
2	A_2	A_2
3	A_3	A_3
4	B_2	
5	A_5	A_5
6	B_5	B_5
7	C_2	
8		C_2
9	A_9	A_9

4. Sobre árboles binarios de búsqueda (ABB's)

a) Describe un algoritmo de certificación para ABB's, suponiendo que sabemos que el árbol es binario. Es decir, tu algoritmo debe verificar que la propiedad de ABB se cumple. Tu algoritmo debe correr en tiempo O(número de nodos en el árbol).

P.ej., un algoritmo recursivo que, a partir de un nodo x, verifica que el hijo izquierdo de x sea abb, que el hijo derecho de x sea abb, y que la clave más grande del hijo izquierdo sea menor que la clave de x y que ésta sea menor que la clave más pequeña del hijo derecho.

b) Supongamos que la búsqueda de una clave k en un ABB termina en una hoja. Consideremos tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las claves a la derecha de la ruta de búsqueda. Tomemos tres claves: $a \in A$, $b \in B$ y $c \in C$. ¿Es cierto o es falso que $a < b < c$? Justifica.

Falso. Para justificar, basta un ejemplo: Supongamos que la búsqueda pasa por un nodo con clave 7, baja al hijo derecho con clave 13 y baja al hijo derecho con clave 17, que es una hoja; así, las últimas tres claves en la ruta de búsqueda son 7, 13 y 17. Entonces, basta que el nodo con clave 13 tenga un hijo izquierdo, cuya clave debe ser mayor que 7 (y menor que 13), p.ej., 11; esta clave queda a la izquierda de la ruta de búsqueda, pero no es menor que cualquier clave en la ruta de búsqueda, en particular, no es menor que 7.

c) ¿Es la operación de eliminación "conmutativa" en el sentido de que elimnar x y luego y de un ABB deja el mismo árbol que eliminar y y luego x? Demuestra que efectivamente lo es o da un contra-ejemplo.

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Consideremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.