Estructuras de Datos y Algoritmos – IIC2133

I1

14 abril 2014

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

1. *Colas de prioridades*.

a) ¿Cuál es el número máximo de comparaciones que es necesario realizar para encontrar la clave más grande almacenada en un *min-heap* binario? Justifica

Como se trata de un *min-heap*, la clave más grande tiene que estar en un nodo sin hijos (en un nodo con hijos, la clave del nodo debe ser menor que las claves de sus hijos), es decir, en una hoja; como puede ser cualquier hoja (no hay ninguna propiedad de orden entre las claves de los nodos en un mismo nivel), el número de comparaciones es $n/2$ (estrictamente, $\lceil n/2 \rceil$), si el *min-heap* tiene n nodos.

b) Describe un algoritmo para encontrar todas las claves menores que algún valor, X, almacenadas en un *min-heap* binario? Tu algoritmo debe correr en tiempo $O(K)$, en que K es el número de claves que cumplen la condición.

Como se trata de un *min-heap*, primero hay que mirar la raíz; si la clave de la raíz es menor que X (encontramos una), entonces hay que mirar, recursivamente, cada uno de sus hijos. Cada "mirada" de un nodo es simplemente una comparación.

c) Si el 99% de las operaciones sobre una cola de prioridades son *insert*( ) y sólo el 1% son *xMin*( ), ¿cuál implementación de la cola sería más apropiada: *min-heap* binario, arreglo no ordenado, o arreglo ordenado? Justifica.

El análisis debe tener el siguiente "sabor".
En un *min-heap*, *insert*( ) toma $\log n$ pasos y *xMin*( ) también toma $\log n$ pasos; en promedio, el número de pasos cada 100 operaciones es entonces $100 \log n$.
En un arreglo no ordenado, *insert*( ) toma un paso, pero *xMin*( ) toma n pasos; en promedio, el número de pasos cada 100 operaciones es entonces $99 + n$.
En un arreglo ordenado, *insert*( ) toma en promedio $n/2$ pasos, y *xMin*( ) toma un paso; en promedio, el número de pasos cada 100 operaciones es entonces $44.5n + 1$.
Claramente, el candidato está entre $100 \log n$ y $99 + n$. Para $n \geq 1024$ ($= 2^{10}$), preferimos el *min-heap*; pero para $n \leq 512$ ($= 2^9$), preferimos el arreglo no ordenado. El punto de quiebre está entre $n = 878$ y $n = 879$.

2. *Hashing*.

a) En el caso de *hashing* con direccionamiento abierto, si la tabla empieza a llenarse, entonces la ejecución de las operaciones de búsqueda e inserción empieza a tomar demasiado tiempo. La solución es construir otra tabla que sea el doble de grande, definir una nueva función de *hash*, y revisar la tabla original completa, calculando el nuevo valor de *hash* para cada elemento e insertándolo en la nueva tabla.

Esta operación, llamada *rehashing*, es cara, pero en la práctica, considerando la frecuencia con que debe ejecutarse, no afecta demasiado al desempeño global de la tabla. ¿Por qué?

Supongamos que decidimos hacer *rehashing* cada vez que la tabla está 50% llena (número de claves $n = m/2$, en que m es el tamaño de la tabla); es decir, tienen que haber ocurrido por lo menos $n = m/2$ operaciones de inserción. ¿Cuánto cuesta hacer *rehashing*? Reservar una nueva tabla de tamaño $2m$ y definir una nueva función de *hash* para esta tabla es O(1); suponiendo que calcular el valor de *hash* de una clave es O(1), entonces recorrer la tabla original y calcular el nuevo valor de *hash* para cada clave almacenada es O(m). Es decir, ejecutamos una operación de costo O(m) después de haber hecho $m/2$ = O(m) operaciones de costo O(1) en promedio cada una. Por lo tanto, en términos de O( ), el costo de *rehashing* no suma al costo ya incurrido de las inserciones en la tabla original.

b) Queremos encontrar la primera ocurrencia de un string $P_1P_2\ldots P_k$ en otro string mucho más largo $A_1A_2\ldots A_n$ ($n > k$). ¿Cómo podemos resolver este problema usando *hashing*? Suponiendo que para un string s empleamos una función de *hash* como la siguiente, ¿qué tan eficiente, en términos de k y n, es esta solución?

```
hashValue = 0
for (i = 0; i < s.length(); i++)
   hashValue = 37*hashValue + s.ascii(i)
hashValue = hashValue % tableSize
```

La idea es calcular el valor de *hash* para el string $P_1P_2\ldots P_k$ y luego calcular el valor de *hash* para cada substring de largo k del string $A_1A_2\ldots A_n$. Si los valores de *hash* son distintos, entonces los strings no pueden ser iguales. Si los valores de *hash* son iguales, entonces comparamos los strings carácter por carácter (ya que hay una pequeña posibilidad de que los strings sean distintos).

En general, este método va a tomar un tiempo proporcional al tiempo que toma calcular el valor de *hash* para un string de k caracteres, digamos $f(k)$, multiplicado por el número de tales strings, $n - k$, y más lo que toma comparar carácter por carácter dos strings de k caracteres; es decir, $(n - k)f(k) + k$. En el caso de la función de *hash* dada, $f(k) = ck$; luego, el método toma tiempo proporcional a $ck(n - k) + k$; esencialmente, O(nk).

Sin embargo, observamos que el valor de *hash* del string $A_iA_{i+1}\ldots A_{i+k-1}$, digamos h_i, puede obtenerse directamente a partir del valor de *hash* del string $A_{i-1}A_i\ldots A_{i+k}$, digamos h_{i-1}, sin tener que calcularlo desde cero: $h_i = h_{i-1} - \text{ascii}(A_{i-1})*37^{k-1} + \text{ascii}(A_{i+k})$. Es decir, sólo el cálculo del valor de *hash* del primer substring $A_1A_2\ldots A_k$ toma tiempo proporcional a k; los $n–k–1$ siguientes son todos constantes. Por lo tanto, el método toma tiempo proporcional a $k + (n–k–1) + k$; esencialmente, O($n+k$).

3. *Árboles de búsqueda binarios (ABB).*

a) Supongamos que tenemos una estimación por adelantado de cuán a menudo nuestro programa va a tener acceso a las claves de los datos (es decir, conocemos la frecuencia de acceso a las claves). Si queremos emplear un ABB, inicialmente vacío, para almacenar los datos (según sus claves), ¿en qué orden deberíamos insertar las claves en el árbol? Justifica.

La idea es encontrar lo más rápidamente posible cada clave, cuando la busquemos. Como el número de comparaciones para encontrar una clave depende directamente de su profundidad en el árbol, las claves que están cerca de la raíz se encuentran haciendo pocas comparaciones; y, si el árbol está más o menos balanceado, las claves que están cerca de las hojas se encuentran haciendo varias comparaciones. Luego, nos conviene que las claves que se van a buscar más frecuentemente estén más cerca de la raíz, y que las claves que se van a buscar con menos frecuencia estén más cerca de las hojas. En un árbol que se construye sólo con inserciones (y sin rotaciones), el primer nodo que se inserta se convierte en la raíz del árbol; por lo tanto, en nuestro caso, éste debería ser el de más alta frecuencia esperada de búsqueda.

b) Supongamos que cada vez que nuestro programa tiene acceso a una clave, k, necesita saber cuántas claves en el árbol son menores que k; esto se conoce como el **rango** de k. ¿Qué información adicional sería necesario almacenar en el árbol? ¿Cómo se determinaría el rango de k y cuál sería la complejidad de esta operación? ¿Cuánto costaría mantener actualizada la información adicional del árbol cuando se produce una inserción o una eliminación de una clave?

Como los ABB son ordenados, el rango de k es el número de claves almacenadas "a la izquierda" de la clave k. Así, si k está justamente en la raíz del árbol, entonces su rango es igual al número de nodos en el subárbol izquierdo. En cambio, si k es menor que la clave de la raíz (y, por lo tanto, está en el subárbol izquierdo), entonces su rango (en todo el árbol) es igual a su rango dentro del subárbol izquierdo, ya que las claves en el subárbol derecho son todas mayores que k. Finalmente, si k es mayor que la clave de la raíz (y, por lo tanto, está en el subárbol derecho), entonces su rango es igual al número de claves en el subárbol izquierdo (ya que todas ellas son menores que k) más uno (la clave de la raíz también es menor que k) y más el rango de k dentro del subárbol derecho (el número de claves del subárbol derecho que son menores que k). Por supuesto, el rango de k en el subárbol izquierdo y el rango de k en el subárbol derecho se determina recursivamente de la misma forma.

Por lo tanto, la información adicional que conviene almacenar en el árbol para poder aplicar el algoritmo anterior es, para cada nodo t, almacenar en un campo *size* el número de nodos que hay en el subárbol cuya raíz es t. Esta información se mantiene actualizada fácilmente. Durante una inserción, sumamos uno al campo *size* de cada nodo en la ruta de inserción, y para el nodo insertado hacemos *size* = 1. Luego de hacer una eliminación, restamos uno al campo *size* de cada nodo en la ruta desde el padre del nodo eliminado hasta la raíz. En ambos casos, esto cuesta O(altura del árbol).

4. *Árboles de búsqueda balanceados.*

a) Describe un algoritmo de certificación para árboles AVL, suponiendo que sabemos que el árbol es un ABB. Es decir, tu algoritmo debe verificar que la propiedad de balance del árbol se cumple y que la información de balance mantenida en cada nodo está correcta. Tu algoritmo debe correr en tiempo O(número de nodos en el árbol).

El algoritmo asigna una "altura" a cada nodo: una hoja tiene altura 0; la altura de un nodo que no es una hoja es uno más que la altura de su subárbol más alto. El algoritmo comienza en la raíz del árbol.
Si el nodo es una hoja, entonces su balance debe ser 0. Si el balance no es 0, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: 0.
Si el nodo no es una hoja, entonces el algoritmo se ejecuta recursivamente primero en el subárbol izquierdo y luego el subárbol derecho del nodo. Si estas ejecuciones recursivas vuelven con los valores h_{izq} y h_{der}, respectivamente, entonces el algoritmo verifica la propiedad de árbol AVL (h_{izq} y h_{der} no pueden diferir en más de uno) y el balance del nodo (debe ser 0, –1 o +1, dependiendo de la relación entre h_{izq} y h_{der}). Si cualquiera de estas verificaciones no se cumple, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: uno más que el mayor entre h_{izq} y h_{der} y, si el nodo es la raíz del árbol, el algoritmo responde "verdadero".

b) Dibuja los árboles-B resultantes al ir insertando las claves **4**, **17**, **2**, **7**, **8**, **13**, **6** y **23** en un árbol-B de grado mínimo $t = 2$ que inicialmente contiene un solo nodo (la raíz) con las claves **0** y **18**.

