CC4102 - Examen

Profs. Benjamin Bustos y Gonzalo Navarro

17 de Julio de 2024

P1 (1.5 pt)

Dadas n claves numéricas, cada una asociada a un peso, se desea una estructura que resuelva la consulta $peso(x, y)$, la cual entrega la suma de los pesos asociados a todas las claves z tal que $x \leq z \leq y$. Por ejemplo, si las claves numéricas y sus pesos asociados, (clave, peso), son $\{(2, 5), (4, 11), (6, 8), (8, 1)\}$, entonces $peso(3, 7)$ devuelve 19 y $peso(2.5, 3.5)$ devuelve 0.

1. (0.5pt) Se propone usar un árbol balanceado ordenando por claves, donde cada nodo conoce la suma de los pesos y el rango de las claves de su subárbol. Diseñe y analice un algoritmo sobre esta estructura para resolver la consulta $peso(x, y)$ en tiempo $O(\log n)$.
2. (1.0pt) Demuestre que, en el modelo de comparaciones, este tiempo es óptimo.

Solución:

1. La consulta baja por dos caminos, encontrando los subárboles maximales cuyas claves están contenidas en $[x, y]$, donde T es la raíz del árbol y los campos son $T.key$ y $T.peso$ para la clave y el peso, $T.left$ y $T.right$ para los hijos, $T.min$ y $T.max$ para el rango de claves, y $T.sum$ para la suma de pesos.

   $peso(x, y, T)$:

   a) Si $x \leq T.min$ y $T.max \leq y$, retornar $T.sum$.
   b) Si $y < T.key$, retornar $peso(x, y, T.left)$.
   c) Si $x > T.key$, retornar $peso(x, y, T.right)$.
   d) Retornar $T.peso + peso(x, y, T.left) + peso(x, y, T.right)$

   Deben mostrar que la consulta recorre $O(\log n)$ nodos. Por ejemplo, se hacen dos llamadas recursivas no triviales sólo una vez (es un análisis parecido al que se hizo para los van Emde Boas trees).

2. Se puede reducir del problema de buscar una clave, por ejemplo. Se les da peso 1 a todas las claves y x está en el conjunto sii $peso(x, x) = 1$.

P2 (1.5 pt)

Para mejorar los tiempos de inserción en un B-tree, se propone un esquema donde sólo hay entre $k/2$ y k hijos, para $k = \sqrt{B}$ (en vez del típico $k = B$). Los otros $B - \sqrt{B}$ lugares del bloque se usan para almacenar desordenadamente los elementos insertados que aún no se han enviado a los hijos correspondientes. Es decir, al insertar un elemento, simplemente se agrega a la raíz del árbol (lo que es gratis porque mantendremos la raíz en memoria principal). Cuando la raíz rebalsa, todos los elementos se reparten entre los k hijos correspondientes, pagando a lo más 1 I/O por cada hijo. Si alguno de estos hijos rebalsa, sin embargo, se repite el proceso entre sus hijos, y así sucesivamente. Si una hoja rebalsa, se trata como en los B-trees normales. Para buscar y borrar, se hace como siempre, comparando además la clave con todas las que se encuentran almacenadas en cada nodo del camino.

1. (0.5pt) Muestre que el tiempo de búsqueda sigue siendo $O(\log_B N)$ I/Os si se almacenan N elementos. ¿Esperaría que los I/Os empeoren en la práctica? Puede ser útil recordar que $\log_a b = \frac{\log b}{\log a}$.

2. (1.0pt) Muestre que el tiempo de inserción *amortizado* es $O((1/\sqrt{B}) \log_B N)$ I/Os. ¿En qué escenarios recomendaría usar esta estructura en vez de un B-tree clásico?

Solución:

1. Este árbol tiene profundidad $O(\log_{\sqrt{B}} N) = O(\log_B N)$. Esa es la cantidad máxima de nodos que recorremos al buscar, aunque ahora haya que comparar la clave contra varias almacenadas directamente en los nodos del camino. En la práctica la cantidad de I/Os se duplica, sin embargo, pues $\log_{\sqrt{B}} N = 2 \log_B N$.

2. Cada elemento se inserta en la raíz, y luego es traspasado al hijo correspondiente cuando la raíz se llena. Los $B - \sqrt{B}$ elementos a traspasar en este momento se reparten entre $\Theta(\sqrt{B})$ hijos, lo que produce un costo de $O(\sqrt{B})$ para transferir $B - \sqrt{B}$ elementos. Esto da un costo amortizado de $O(1/\sqrt{B})$ I/Os por cada elemento transferido. Repetido a lo largo de los $O(\log_B N)$ niveles a los que se irá transfiriendo el elemento a lo largo de su vida, el costo total de haberlo insertado será de $O((1/\sqrt{B}) \log_B N)$ I/Os. Considerando los costos, y que consultar es en la práctica más lento, este esquema es mejor para casos en que hay muchas inserciones en comparación con las lecturas, por ejemplo para generar logs en disco.

P3 (1.5 pt)

Recuerde que el problema de 3-SAT recibe una fórmula en 3-FNC (es decir, una conjunción $F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, donde cada cláusula es $C_i = (x_i \vee y_i \vee z_i)$, siendo las x_i, y_i y z_i variables o variables negadas) y determina si F es satisfactible. Una versión de optimización busca una asignación de valores a las variables que maximice la cantidad de cláusulas C_i satisfechas.

Demuestre que asignando valores verdadero o falso a las variables en forma aleatoria y uniforme, se obtiene una 8/7-aproximación esperada. Se sugiere seguir estos pasos:

1. Calcule la probabilidad que una cláusula C_i se haga verdadera.
2. Luego, calcule la cantidad esperada de cláusulas satisfechas.
3. Concluya que el resultado es una 8/7-aproximación.

Solución:

En cada cláusula C_i, se tiene que x_i, y_i y z_i se hacen verdaderas con probabilidad $1/2$, con lo que la probabilidad de que C_i se haga verdadera es $1 - 1/8 = 7/8$. Eso es también la esperanza de V_i, la variable aleatoria indicadora que es 1 si C_i se hace verdadera y 0 si no. La cantidad esperada de cláusulas satisfechas es entonces $E[\sum V_i] = \sum E[V_i] = (7/8)n$. Como a lo más se pueden satisfacer las n cláusulas, el factor esperado de aproximación es 8/7.

P4 (1.5 pt)

Considere el problema de, dado un arreglo $A[1..n]$ que se puede preprocesar, calcular $distintos(i, j)$ para cualquier $1 \leq i \leq j \leq n$ dado, es decir, la cantidad de valores distintos en $A[i..j]$. Se propone el siguiente esquema para preprocesar A, llenando arreglos $M_\ell[1..n]$ para $\ell \geq 1$ mediante el procedimiento recursivo $preproc(1, n, 1)$:

$preproc(a, b, \ell)$:

1. Si $a = b$, salir.
2. Sea $m = \lfloor (a + b)/2 \rfloor$.
3. Para $k \in [a..m]$, escribir en $M_\ell[k]$ la cantidad de valores distintos en $A[k..m]$.
4. Para $k \in [m + 1..b]$, escribir en $M_\ell[k]$ la cantidad de valores distintos en $A[m + 1..k]$.
5. Invocar $preproc(a, m, \ell + 1)$ y $preproc(m + 1, b, \ell + 1)$.

Teniendo precalculados los arreglos $M_\ell[1..n]$ para $\ell = 1, \ldots, \lceil \log_2 n \rceil$, describa una 2-aproximación al problema de calcular $distintos(i, j)$ que funcione en tiempo $O(\log n)$ (es decir, debe devolver un valor que esté entre $distintos(i, j)$ y $2 \cdot distintos(i, j)$).

Hint: Piense primero en el caso en que $i \leq m < j$, donde $m = \lfloor (n + 1)/2 \rfloor$.

Solución:

En el caso sugerido, $M_1[i] + M_1[j]$ suman los valores distintos en $A[i..m]$ y en $A[m + 1..j]$, lo que cuenta a lo sumo 2 veces cada valor distinto, por lo que es una 2-aproximación. De no ser así, debemos buscar el nivel donde i y j tengan un punto medio m entre ellos.

El siguiente procedimiento nos entrega la 2-aproximación en tiempo $O(\log n)$, invocando $distintos(i, j, 1, n, 1)$. Este procedimiento asume $i < j$ (si $i = j$ la respuesta es siempre 1).

$distintosAprox(i, j, a, b, \ell)$:

1. Sea $m = \lfloor (a+b)/2 \rfloor$.
2. Si $i \leq m < j$, retornar $M_\ell[i] + M_\ell[j]$.
3. Si $j \leq m$, retornar $distintosAprox(i, j, a, m, \ell + 1)$.
4. Retornar $distintosAprox(i, j, m + 1, b, \ell + 1)$.

Tiempo: 3 horas Con cuatro hojas de apuntes