Estructuras de Datos y Algoritmos – IIC2133

I2

4 octubre 2013

1. *Árboles B⁺ —una variación de los árboles B.* Un **árbol B⁺** de orden m tiene las siguientes propiedades:

i) Los datos están almacenados en las hojas.

ii) Los nodos que no son hojas almacenan hasta m–1 claves (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) Describe un algoritmo para insertar un nuevo dato en (una hoja de) un árbol B⁺.

b) Describe un algoritmo para eliminar un dato de (una hoja de) un árbol B⁺.

Tus descripciones pueden ser en prosa, pero deben ser claras y precisas; numera pasos, identifica casos.

Respuestas

Básicamente, se pueden emplear los mismos algoritmos "defensivos" del árbol B vistos en clase, aunque simplificando la eliminación, porque ahora sólo eliminamos datos que están en las hojas.

También son aceptables los algoritmos más simples, que primero insertan o eliminan un dato en una hoja, y luego restauran las propiedades del árbol. En la inserción, si es que la hoja queda con más de t datos, hay que dividirla en dos, por lo que hay que agregar una nueva clave al nodo padre; si este nodo ya tiene m–1 claves, entonces también hay que dividirlo en dos, pasando una nueva clave a su padre; y así sucesivamente hasta, posiblemente, llegar a la raíz; si es necesario dividir la raíz, entonces hay que crear una nueva raíz, cuyos hijos serán los dos nodos resultantes de la división de la raíz original.

En la eliminación, si la hoja queda con menos de t/2 datos, entonces se le puede traspasar un dato de un hermano "inmediato"; excepto si ambos hermanos tienen sólo t/2 datos cada uno, en cuyo caso hay que fusionar la hoja con alguno de sus hermanos, haciendo que su padre quede con una clave menos. Si este nodo tiene sólo m/2 hijos, entonces se aplica la misma estrategia; y así sucesivamente hasta, posiblemente, llegar a la raíz; si la raíz llega a quedar con un solo hijo, entonces simplemente la eliminamos y convertimos ese hijo en la nueva raíz.

2. *Otro algoritmo de inserción en árboles rojo-negros*. En lugar de primero insertar el nuevo nodo (pintándolo de rojo) y después hacer rotaciones y/o cambios de color (*hacia arriba*) para restaurar las propiedades del árbol, podemos ir haciendo los ajustes a medida que vamos *bajando* por el árbol hacia el punto de inserción, de modo que cuando insertamos el nuevo nodo simplemente lo pintamos de rojo y sabemos que su padre es negro. El procedimiento es el siguiente.

Mientras bajamos por el árbol, cuando vemos un nodo X que tiene dos hijos rojos, intercambiamos colores: pintamos X de rojo y sus hijos de negro. Esto producirá un problema sólo si el padre P de X es rojo. Pero en este caso, simplemente aplicamos las rotaciones apropiadas, que se ilustran en las diapositivas 35 y 36 adjuntas.

a) Muestra cómo opera este nuevo algoritmo de inserción cuando insertamos un nodo con la clave K en el árbol de la diapositiva 37 adjunta; específicamente, muestra lo que ocurre a medida que llegas a cada nivel.

Respuesta

Cuando se baja de la raíz "H" a su hijo derecho "T" no pasa nada, ya que "T" (el X del algoritmo) tiene sólo un hijo rojo, "P". Cuando se baja de "T" a "P" tampoco pasa nada, ya que "P" (el X del algoritmo) es rojo. Cuando se baja de "P" a "L" estamos en el caso descrito en el algoritmo: "L" (el X del algoritmo) es negro y sus dos hijos, "J" y "N", son rojos. [*Hasta aquí no hemos cambiado nada, sólo bajamos por el árbol*: **0.5 pts.**]

Entonces, cambiamos colores: pintamos a "L" de rojo y a sus hijos "J" y "N" de negro. Pero ahora "L" y su padre "P" son rojos; como el hermano "W" de "P" es negro, estamos en la situación descrita en la diap. 35: "L" es X, "P" es P, "T" es G, "W" es S. [**1 pt.**]

Aplicando la rotación y cambios de colores sugeridos en la diap. 35, queda "P" de negro como nuevo hijo derecho de "H" y con dos hijos rojos, "L" y "T", que en total tienen cuatro hijos negros, "J", "N", "R" y "W" (este último mantiene sus dos hijos rojos, "V" y "Z", aunque ahora los tres están un nivel más abajo). [**1 pt.**]

Nosotros seguimos en "L", que ahora es rojo y está un nivel más arriba, así como sus hijos, "J" y "N", ahora negros. Bajamos a "J", que es una hoja negra, y por lo tanto insertamos "K" como hijo derecho de "J" y lo pintamos rojo. [**0.5 pts.**]

b) Los casos ilustrados en las diapositivas 35 y 36 se producen cuando el hermano S del padre P de X es negro. Explica claramente qué pasa con el caso en que S es rojo.

Respuesta

Este caso en realidad ***no se produce***, debido a los ajustes que vamos haciendo a medida que bajamos por el árbol. Si es que encontramos un nodo Y con dos hijos rojos, sabemos que sus nietos tienen que ser todos negros; y como también pintamos de negro los hijos de Y, entonces incluso después de las posibles rotaciones no vamos a encontrar otro nodo rojo en los próximos dos niveles.

3. *Número de operaciones necesarias.*

a) Tienes n tuercas y n pernos. Las tuercas son todas de distintos tamaños y los pernos son todos de distintos tamaños, pero sabes que a cada tuerca le corresponde exactamente un perno. Tu problema es encontrar los emparejamientos correctos: para cada perno, la tuerca que es del mismo tamaño. Si la única operación que puedes hacer es tratar de colocar un perno en una tuerca—lo que te va a decir si el perno o la tuerca es más grande o si son del mismo tamaño—prueba que el número de comparaciones necesarias para resolver el problema está acotado inferiormente por $n \log n$. [Recuerda: No puedes comparar dos pernos entre ellos ni dos tuercas entre ellas.]

Respuesta

El algoritmo —cualquiera que sea— tiene que realizar comparaciones hasta juntar la información necesaria para hacer los emparejamientos correctos. Así como en el caso de ordenación por comparación, la ejecución del algoritmo puede verse como un árbol; en este caso, cada nodo representa un par (perno, tuerca) y tiene tres hijos (el árbol es "trinario"): el perno es más grande que la tuerca, la tuerca es más grande, son iguales. Las hojas corresponden a emparejamientos correctos de los n pernos con las n tuercas.

¿Cuál es la altura del árbol? Es el número de comparaciones que el algoritmo tiene que hacer en el peor caso para encontrar los n emparejamientos correctos. ¿De cuántas maneras podrían darse estos emparejamientos? Si numeramos los pernos arbitrariamente de 1 a n, y numeramos las tuercas de 1 a n, el emparejamiento correcto del perno i podría ser con cualquiera de las n tuercas, y el emparejamiento correcto de los n pernos va a ser con una permutación de (los números del 1 al n que representan a) las tuercas. Como esta permutación puede ser cualquiera de las $n!$ permutaciones de las tuercas, el árbol tiene $n!$ hojas.

Como un árbol trinario de altura h tiene a lo más 3^h hojas, deducimos que $3^h \geq n!$, lo que finalmente implica $h \geq \Omega(n \log n)$.

b) Explica cómo se puede ordenar n enteros, que están en el rango 0 a $n^3–1$, en tiempo $O(n)$.

Respuesta

Los números en ese rango pueden ser considerados como números de tres dígitos, en que cada dígito va de 0 a $n–1$. (También se pueden ver como números el triple de largos, en número de dígitos, que los números en el rango 0 a $n–1$.) Aplicando `radixSort`, hay tres llamadas a `countingSort` (ordenación estable), en que cada una toma tiempo $O(n+n) = O(n)$, de modo que en total el algoritmo toma tiempo $O(n)$.

4. *quickSort*.

a) ¿Qué valor de q (o j) devuelve `partition` cuando todos los elementos del arreglo `A[p…r]` son iguales? Justifica.

Respuesta

Devuelve $q = (p+r-1)/2$. Los `while`'s internos no se ejecutan, ya que los elementos del arreglo son iguales al pivote, no mayores ni menores. Por lo tanto, el efecto del `while` externo es que en cada iteración `j` se incrementa en 1 y `k` se decrementa en 1 (y el `exchange` dentro del `while` intercambia de posición dos elementos iguales). Así, `j` se "encuentra" con `k` en la mitad del rango `p`, …, `r`–1 (ya que `k` parte en `r`–1), y en ese punto se detiene el `while`. El `exchange` fuera del while no cambia el valor de `j`.

b) ¿Cuál es el tiempo de ejecución de `quickSort` cuando todos los elementos del arreglo `A` son iguales? Justifica.

Respuesta

$O(n\log n)$. Como vimos en clase, cuando `partition` produce dos particiones de similar tamaño, `quickSort` tiene su mejor desempeño. Y de **a**), este es el caso cuando todos los elementos del arreglo son iguales.

c) Considera la siguiente versión de `quickSort`, que sólo hace la primera llamada recursiva:

```
quickSort'(A, p, r)
while (p < r)
   q = partition(A, p, r)
   quickSort'(A, p, q-1)
   p = q+1
```

Explica por qué `quickSort'` ordena correctamente el arreglo `A`.

Respuesta

`quickSort'` hace la misma partición que `quickSort`, y luego también se llama recursivamente con parámetros `A`, `p` y `q-1`. La diferencia es que en este punto `quickSort` se llama recursivamente por segunda vez con parámetros `A`, `q+1` y `r`; en cambio, `quickSort'` asigna `p = q+1` y hace otra iteración del `while`. Sin embargo, esto ejecuta las mismas operaciones que la segunda llamada recursiva, con `A`, `q+1` y `r`, ya que en ambos casos `A` y `r` tienen los mismos valores que antes y `p` tiene el valor antiguo de `q+1`.