Estructuras de Datos y Algoritmos – IIC2133

I1

13 septiembre 2010

1. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una lista ligada de casilleros disponibles. Pare esto, supón que cada casillero puede almacenar un boolean y ya sea un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista deberían correr en tiempo esperado $O(1)$. Específicamente, explica lo siguiente:

a) [**1** pt] El papel del boolean.

El boolean es para saber si el casillero tiene un elemento (y un puntero a otro elemento o null), o si tiene dos punteros (a los casilleros delante y detrás en la lista ***doblemente ligada*** *de casilleros disponibles).*

b) [**3** pts.] ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos ***lista de colisiones*** *a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento).*

Inserción*: Se aplica la función de hash al elemento; supongamos que da* k. *Si el casillero* k *está disponible (su boolean vale* true*), lo sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y almacenamos ahí el elemento. Para esto, cambiamos el boolean a* false *y ponemos el puntero en* nil.

Si el casillero k *está ocupado (su boolean vale* false*), hay dos posibilidades: es el primer elemento de la lista de colisiones que comienza en el casillero* k*; o es algún otro elemento en alguna lista de colisiones que comienza en otro casillero.*

Eliminación:

Búsqueda:

c) [**2** pts.] ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo esperado $O(1)$?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) esperado.

Las operaciones sobre la lista de casilleros disponibles corren en tiempo O(1) gracias a que es doblemente ligada (si no, el único problema ocurre cuando se saca un casillero específico de la lista de casilleros disponibles).

2. Demuestra que cualquier ABB arbitrario de n nodos puede ser transformado en cualquier otro ABB arbitrario de n nodos usando $O(n)$ rotaciones.

a) Primero, prueba que con a lo más n–1 rotaciones a la derecha puedes convertir cualquier ABB en uno que es simplemente una lista de n nodos ligada por los punteros a los hijos derechos.

*Llamemos **lista derechista** a la lista de nodos ligada por los punteros a los hijos derechos; queremos convertir el árbol en una lista derechista de n nodos. Inicialmente, la lista derechista tiene al menos un nodo: la raíz del árbol.*

Cada rotación a la derecha respecto de un nodo que está en la lista derechista, y tiene un hijo izquierdo, agrega ese hijo izquierdo a la lista derechista. La lista comienza con al menos un nodo, que es la raíz. Por lo tanto, con a lo más n–1 rotaciones a la derecha, todos los nodos han pasado a la lista.

Ahora observa que si conocieras la secuencia de rotaciones a la derecha que transforman un ABB arbitrario T en una lista ligada T', entonces podrías ejecutar esta secuencia en orden inverso, cambiando cada rotación a la derecha por su rotación inversa a la izquierda, para transformar T' de vuelta en T.

b) Explica cómo transformar un ABB T_1 en otro T_2, pasando por la lista ligada única T' de nodos de T_1 (que es la misma que la de los nodos de T_2). ¿Cuántas rotaciones a lo más es necesario hacer para esto?

*T' es la lista derechista. Como vimos en **a**), podemos convertir T1 en T' con a lo más n–1 rotaciones a la derecha. Como los nodos de T2 son los mismos que los de T1, entonces también podríamos convertir T2 en T' con a los más n–1 rotaciones a la derecha. Si a partir de T' aplicamos estas últimas n–1 rotaciones en orden inverso, y cada vez rotamos a la izquierda en vez de la derecha, obtenemos T2.*

3. ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica tus respuestas.

Recuerda que al insertar un nodo, x, lo insertamos como una hoja y lo pintamos de rojo. Si el padre, p, de x es negro, terminamos. Si p es rojo, tenemos dos casos: el hermano, s, de p es negro; s es rojo. Si s es ne-gro, realizamos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g, de p y s es negro; entonces, cambiamos los colores de g, p y s, y revisamos el color del padre de g.

Hacemos O(log n) cambios de color y O(1) rotaciones. Como dice arriba, esto ocurre solo cuando p *es rojo.*

Si s *es negro, hacemos exactamente una o dos rotaciones, dependiendo de si* x *es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.*

Si s *es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos:* g, *que queda rojo, y sus hijos* p *y* s, *que quedan negros. Como* g *queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.*

4. Un arreglo a tiene inicialmente el siguiente contenido:

$$a = [\ 0\ \ 18\ \ 14\ \ 17\ \ 19\ \ 8\ \ 13\ \ 6\ \ 4\ \ 23\ \ 0\ \ 12\ \ 15\ ].$$

Transfórmalo en un *max-heap* binario usando el siguiente algoritmo:

```
for (k = 5; k >= 0; k = k-1) heapify(k)
```

Muestra el resultado, como árbol binario, después de cada iteración.