Estructuras de Datos y Algoritmos – IIC2133

Examen

25 noviembre 2010

1) En el caso de hashing con direccionamiento abierto, supón 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 cada uno de los dos casos siguientes:

a) Si usamos revisión lineal, de modo que la posición intentada es $(h(K) + i) \bmod T$.

b) Si usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)-1$, $h(K)+4$, $h(K)-4$, $h(K)+9$, ..., $h(K)+(T-1)^2/4$, $h(K)-(T-1)^2/4$.

a)

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

b)

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

2) Con respecto a los siguientes algoritmos de ordenación, (**a**) ¿cuáles son estables y cómo se sabe que lo son? Para los que no lo son, (**b**) explica cómo se los puede hacer estables y a qué costo. Recuerda que un algoritmo de ordenación es ***estable*** si los datos con igual valor aparecen en el resultado en el mismo orden que tenían al comienzo.

i) Insertionsort. ii) Mergesort. iii) Heapsort. iv) Quicksort.

3) Se tiene que programar un conjunto de n actividades en una máquina. Cada actividad a_i tiene una hora de inicio s_i, una hora de término f_i, y un valor v_i. El objetivo es maximizar *el valor total de las actividades programadas* (y no necesariamente el número de actividades programadas).

Sea C_{ij} el conjunto de actividades que empiezan después que la actividad a_i termina y que terminan antes que empiece la actividad a_j. Sea A_{ij} una solución óptima a C_{ij}, es decir, A_{ij} es un subconjunto de actividades mutuamente compatibles de C_{ij} que tiene valor máximo.

a) Supongamos que A_{ij} incluye la actividad a_k. Demuestra que este problema tiene la propiedad de ***subestructura óptima***: una solución óptima al problema contiene soluciones óptimas a subproblemas.

b) Sea $val[i, j]$ el valor de una solución óptima para el conjunto C_{ij}. Demuestra que este problema tiene la propiedad de ***subproblemas traslapados***: si usamos un algoritmo recursivo para resolver el problema, este tiene que resolver los mismos subproblemas repetidamente.

4) Con respecto a los árboles rojo-negro:

a) Supón que "absorbemos" todo nodo rojo dentro de su padre negro, de modo que los hijos del nodo rojo son ahora hijos del padre negro. [**2** pts.] ¿Cuáles son los posibles grados de un nodo negro después de que todos sus hijos rojos son absorbidos? [**1** pt.] ¿Qué pasa con la profundidad de las hojas del árbol resultante?

b) Supón que insertamos un nodo *x* en un árbol rojo-negro y luego lo eliminamos inmediatamente. ¿Es el árbol resultante el mismo que el inicial? Justifica tu respuesta.

a) [**2** pts.] El grado grado de un nodo negro después de absorber sus hijos rojos es

2, si sus hijos eran negros
3, si un hijo era negro y el otro rojo
4, si ambos hijos eran rojos

[**1** pt.] Todas las hojas del árbol resultante tienen la misma profundidad.

b) No (en general). Basta con mostrar un contraejemplo en que cambie la forma o los colores.
P.ej., un árbol de raíz 3 (negra), y con hijo izquierdo 2 (rojo): si insertamos el nodo 1 (rojo), como hijo izquierdo de 2, y hacemos una rotación simple a la derecha (para evitar un nodo rojo con padre también rojo), queda el árbol con raíz 2 (negra), e hijos derecho 1 e izquierdo 3 (ambos rojos). Si ahora eliminamos el nodo 1, que acabamos de insertar, queda el árbol con raíz 2 (negra), y con hijo derecho 3 (rojo), distinto del árbol inicial.

5) Con respecto a las rutas más cortas en grafos direccionales:

a) Un compañero de curso te dice que implementó el algoritmo de Dijkstra. El programa produce las distancias d y los padres π para cada vértice del grafo. Da un algoritmo de tiempo $O(V+E)$ para verificar el resultado del programa de tu compañero. Supón que todas las aristas tienen costos no negativos. En particular, ¿qué debes verificar con respecto al vértice de parida s?; ¿qué debes verificar con respecto a los otros vértices?; y ¿cómo te puedes asegurar que el programa de tu compañero efectivamente encontró las rutas más cortas desde s?

b) Da un algoritmo de tiempo $O(V+E)$ para encontrar las rutas más cortas desde un vértice de partida s, si el grafo es acíclico (y, por lo tanto, puede ser ordenado topológicamente).

a)

Para s, hay que verificar $s.d = 0$ y $s.\pi = NIL$.
Para los otros vértices v, hay que verificar $v.d = v.\pi.d + w(v.\pi, v)$; o bien $v.d = \infty$ si y solo si $v.\pi = NIL$.
Si cualquiera de estas verificaciones falla, entonces el resultado del programa del compañero es incorrecto.
De lo contrario, todavía hay que asegurarse que haya encontrado las rutas más cortas desde s. Para esto, basta con reducir cada arista una vez: si algún $v.d$ cambia, entonces el resultado es incorrecto; de lo contrario, es correcto.

b)

```
Ordenar el grafo topológicamente
para cada vértice v del grafo {
    v.d = ∞
    v.π = NIL
}
s.d = 0
para cada vértice u del grafo, en el orden producido por la ordenación topológica
    para cada vértice v adyacente a u
        if ( v.d > u.d + w(u, v) ) {
            v.d = u.d + w(u, v)
            v.π = u
        }
```

6) Considera las operaciones de inserción y eliminación *defensivas* en árboles B. Por "defensivas" queremos decir que, durante una inserción, dividimos los nodos que están llenos *antes* de saber si ese nodo va a recibir una clave más como consecuencia de la inserción; y, durante una eliminación, le entregamos una clave extra a un nodo que tiene pocas claves (pero tiene un hermano que tiene suficientes claves), o bien mezclamos nodos hermanos cuando ambos tienen pocas claves.

   El propósito de las estrategias anteriores es poder "bajar" por el árbol, desde la raíz, buscando el nodo apropiado, sin tener que volver a subir durante una misma operación. Sin embargo, hay una situación en que es necesario volver a subir. Explica clara y precisamente cuándo se produce esta situación y cómo se maneja.

Al eliminar una clave k de un nodo interno x, cuando el hijo y justo a la izquierda (o el hijo z justo a la derecha) de k tiene t claves o más [**2** pts.]. En este caso, se baja al hijo y (o z) buscando la clave k' predecesora (o sucesora) de k, que está en una hoja; luego, se elimina k' de esa hoja, y se reemplaza k por k' en x [**2** pts.]. Para reemplazar k por k', es necesario subir nuevamente hasta x [**2** pts.].