Examen

Estructuras de Datos y Algoritmos – IIC2133

2 diciembre 2013

Tiempo: 2 horas y 30 minutos

1) En el caso de *hashing* con encadenamiento, describe una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una *lista ligada de casilleros disponibles*. Para esto, considera 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 debieran correr en tiempo O(1) en promedio. Específicamente, explica:

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: *insert*, *delete* y *find*?

Llamemos ***lista de colisiones*** *a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento). Sea x el dato que nos interesa, y supongamos que al aplicarle la función de hash a x, da k.*

insert*: Si el casillero k está disponible (su boolean vale true), lo sacamos de la lista (en tiempo O(1) porque es doblemente ligada) y almacenamos ahí x. 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 una lista de colisiones que comienza en el casillero k: sacamos un casillero disponible, almacenamos x en este casillero, y agregamos el casillero a la lista que comienza en el casillero k*
>
> *O es algún otro elemento en alguna lista de colisiones que comienza en otro casillero: hay que sacar el casillero k de esta otra lista, reemplazándolo por un casillero disponible, y luego almacenar x en el casillero k y poner el puntero de este casillero en nil (es el primer elemento de la nueva lista que contiene a los datos cuyo valor de hash es k)*

delete*: Sabemos que x está en la tabla. El casillero k es el primer casillero de una lista simplemente ligada tal que uno de sus casilleros contiene a x: revisamos la lista hasta llegar a este casillero; entonces, lo sacamos de esta lista (que hay que actualizar apropiadamente) y lo ponemos en la lista de casilleros disponibles.*

find*: Si el casillero k no tiene a x, entonces simplemente seguimos la cadena de punteros que empieza aquí.*

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

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) En clase vimos cómo se elimina una clave de un árbol binario de búsqueda (ABB, no necesariamente balanceado).

a) [**1 pt.**] Eliminar la clave cuando el nodo que ocupa no tiene hijos o tiene sólo un hijo, *Ti* o *Td*, es fácil: describe las acciones correspondientes.

Si el nodo no tiene hijos, entonces simplemente se elimina; si el nodo tiene un hijo, entonces ese hijo se coloca en el lugar del nodo.

b) [**1 pt.**] Es más difícil eliminar una clave cuando el nodo que ocupa tiene ambos hijos, *Ti* y *Td*: describe las acciones correspondientes. Esta forma de eliminación se llama *eliminación por copia*.

Se busca la clave sucesora (o predecesora) de la clave eliminada, y se la coloca, junto con su descendencia, en lugar de ésta (de la eliminada); luego, se elimina "recursivamente" la clave sucesora, para lo cual se aplica a), ya que a lo más tiene un hijo.

Otra forma de eliminar una clave cuyo nodo tiene ambos hijos es *eliminación por mezcla*: el nodo es ocupado por su (hijo y) subárbol izquierdo, *Ti*, mientras que su subárbol derecho, *Td*, se convierte en el subárbol derecho del nodo más a la derecha de *Ti*.

c) [**2 pts.**] Justifica que esta eliminación respeta las propiedades de ABB.

A partir de la regla para insertar claves, que, a su vez, obedece a la propiedad fundamental de ABB, sabemos que en el proceso de inserción de cualquiera de las claves que están en Ti o Td —llamemos k a una clave cualquiera en Ti o Td— pasamos por la clave —llamémosla j— que estamos eliminando, y que, por lo tanto, k podría haber ido a parar al lugar de j, posición en la que habría cumplido la propiedad de ABB con respecto al resto del árbol. Por lo tanto, poner el subárbol Ti en la posición que ocupaba el nodo con la clave j es perfectamente válido.

La pregunta entonces es, ¿qué hacemos con Td? De nuevo, por la propiedad de ABB, las claves de Td son todas mayores que las claves de Ti. La única posición que corresponde a claves mayores que todas las claves de Ti, pero al mismo tiempo menores que las otras claves del árbol mayores que j es como hijo derecho de la clave más a la derecha de Ti —llamémosla m; obviamente, esta posición está "desocupada": m sólo puede ser la clave más a la derecha de Ti si no tiene hijo derecho.

d) [**2 pts.**] Muestra con ejemplos que esta eliminación puede tanto aumentar como reducir la altura del árbol original.

Para simplificar (y generalizar un poco), eliminamos la raíz del árbol; Ti "sube" a esta posición y agregamos Td como hijo derecho del nodo más a la derecha de Ti. La altura del árbol original, T, era H(T) = max{ H(Ti), H(Td) } + 1. La altura del nuevo árbol, T', puede ser desde H(T') = H(Ti) hasta H(T') = H(Ti) + H(Td), dependiendo de la profundidad del nodo más a la derecha de Ti. En el primer caso, H(T') es claramente menor que H(T); en el segundo, H(T') claramente puede ser mayor que H(T).

3) Considera las siguientes tres formas de resolver el problema de selección, consistente en encontrar el k-ésimo dato más pequeño de un conjunto de n datos:

i) Ordena los datos de menor a mayor, y elige el k-ésimo.

ii) Coloca los datos en un *min-heap* y ejecuta k operaciones *xMin*.

iii) Usa el algoritmo `randomSelect()` —similar a `quicksort()`, pero que sólo hace recursión sobre una de las dos partes definidas por `randomPartition()`.

Explica las ventajas y desventajas de cada forma y bajo cuáles condiciones usarías unas u otras. Considera aspectos tales como tiempo de ejecución, cantidad n de datos, fracción k/n, frecuencia de ejecución de la selección, y conjunto fijo de datos frente a datos que cambian (se agregan unos y se eliminan otros) a lo largo del tiempo.

i) toma O(n log n) para ordenar los n datos; luego, ***cualquier*** *selección es O(1). Si los datos son fijos y hay que hacer muchas selecciones, éste es el método preferido.*

ii) toma O(n) para colocar los n datos en un heap; luego, cada xMin toma O(log n). Encontrar el k-ésimo dato más pequeño toma O(n) + O(k log n), por lo que funciona mejor para casos con $k/n \approx 0$. Por otra parte, sacar los k objetos del heap significa reconstruir el heap para otra consulta o tener una segunda copia guardada, lo que agrega complejidad en ejecución o en uso de memoria.

iii) toma O(n) en promedio para entregar el k-ésimo dato más pequeño. Si hay que hacer una sola selección, éste es el método preferido. Incluso si hay que hacer un núnero pequeño de selecciones (< logn), éste es el método preferido. Por otra parte, si los datos cambian entre una selección y otra, éste es el mejor método.

4) Considera un grafo no direccional conectado *G*. Un *puente* es una arista que, si se saca, separaría *G* en dos subgrafos disjuntos.

a) [**3 pts.**] Describe un algoritmo eficiente para encontrar todos los puentes de un grafo no direccional.

Una arista (u, v) es un puente ⇔ (u, v) no pertenece a un ciclo (simple del grafo); DFS encuentra los ciclos en un grafo.

Luego, ejecutemos DFS sobre el grafo, y miremos el bosque DFS producido: una arista de árbol (u, v) en este bosque es un puente ⇔ no hay aristas hacia atrás que conecten un descendiente de v a un ancestro de u.

¿Cómo determinamos esto? Recordemos el número —la marca de tiempo— v.d que DFS asigna a cada vértice v : una arista hacia atrás (x, y) conectará un descendiente x de v a un ancestro y de u si y.d < v.d.

Luego, si v.d es menor que todos los números d que pueden ser alcanzados mediante una arista hacia atrás desde cualquier descendiente de v, entonces (u, v) es un puente.

Es decir, basta modificar un poco dfsVisit visto en clase: asignar a cada vértice v un tercer número v.low que sea el menor de todos y.d alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar dfsVisit(v) v.d = v.low, entonces (u, v) es un puente.

b) [**2 pts.**] Justifica que tu algoritmo efectivamente encuentra todos los puentes.

Hay que demostrar la premisa de a): Una arista (u, v) es un puente ⇔ (u, v) no pertenece a un ciclo (simple del grafo). La demostración "sale" de la definición de "puente":

=> Si (u,v) es un puente, entonces por definición no puede formar parte de un ciclo

<= Si (u,v) no pertenece a un ciclo, entonces no hay otra manera de ir de u a v que no sea a través de (u,v); por lo tanto, si sacamos (u,v), u y v quedan desconectados entre ellos; y, por lo tanto, todos los vértices alcanzables desde u quedan desconectados de todos los vértices alzanzables desde v; es decir, el grafo queda desconectado

c) [**1 pt.**] Determina la complejidad de tu algoritmo.

Lo que hicimos fue modificar un poco dfsVisit: asignamos a cada vértice v un tercer número v.low que es el menor de todos y.d alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar dfsVisit(v) v.d = v.low, entonces (u, v) es un puente. Esta modificación no cambia la complejidad de dfsVisit.

5) Tenemos una máquina que puede procesar tareas, de a una a la vez, y tenemos un conjunto de n tareas que procesar. Además, tenemos un tiempo total T para usar la máquina y sabemos que la tarea i ocupa un tiempo t_i para ser procesada, $i = 1, \ldots, n$. Queremos encontrar un subconjunto de tareas tal que todas puedan ser procesadas por la máquina en el tiempo T y al mismo tiempo maximicen el tiempo efectivo de uso de la máquina.

a) [**1 pt.**] Muestra (basta con un ejemplo) que este problema no puede ser resuelto en general usando un algoritmo codicioso basado en procesar las tareas en orden decreciente de tiempo t_i (tampoco se podría en orden creciente).

Si T es par y tienes tres tareas que toman tiempos T/2, T/2 y T/2 + 1. Al ordenarlas de mayor a menor tiempo, quedan T/2 + 1, T/2, T/2; y al elegirlas en ese orden sólo alcanzas a procesar la primera, ocupando la máquina durante un tiempo T/2 + 1. Pero la solución óptima es procesar las otras dos tareas, cada una de las cuales toma tiempo T/2 y por lo tanto ocupan la máquina durante un tiempo total T/2 + T/2 = T.

b) [**2 pts.**] Justifica que este problema cumple la propiedad de subestructura óptima.

La propiedad de subestructura óptima significa que la solución óptima al problema incluye soluciones óptimas a problemas del mismo tipo pero más pequeños; "más pequeños" en este caso significa menos tareas y/o menos tiempo total T. Esto lo podemos ver así:

La solución óptima puede incluir o no la tarea 1.

Si la incluye, entonces observamos que si sacamos esta tarea de la solución, el resto de las tareas que pertenecen a la solución son una solución óptima al problema (más pequeño) de elegir las tareas que maximizan el tiempo efectivo de uso de la máquina de entre las tareas 2 a la n, cuando el tiempo total posible es $T - t1$.

Por el contrario, si no la incluye, entonces la solución óptima es igual a la que habríamos obtenido si las tareas a progra-mar en el tiempo total T fueran sólo las tareas 2 a la n, es decir, sin incluir la tarea 1.

c) [**2 pts.**] Plantea una formulación recursiva del valor de la solución óptima, incluyendo condiciones de borde.

Definamos como opt(p, q, t) el problema de encontrar el subconjunto óptimos de tareas de entre las tareas p, p+1, …, q, cuando el tiempo total disponible en la máquina es t (así, el problema original es opt(1, n, T)). Entonces, generalizando el argumento de b),

$$opt(k, n, t) = max\{ opt(k+1, n, t), opt(k+1, n, t - t_{k+1}) + t_{k+1} \},$$

sabiendo que $opt(n+1, n, t) = 0$, *si* $t \geq 0$, *y* $opt(n+1, n, t) = -\infty$, *si* $t < 0$.

d) [**1 pt.**] Escribe un algoritmo de programación dinámica que aproveche la formulación recursiva anterior para resolver el problema.

El algoritmo debe ir llenando una tabla M con los valores de opt(k, n, t):

```
M[0..n][0..T]
for each t = 0,1,…,T: M[0][t] = 0
for k = 1, 2, …, n
    for t = 0,…, T
        —usar la recurrencia de c) para calcular M[k][t], que corresponde a opt(k,n,t)
return M[n][T]
```

6) Considera una tabla de hash que queremos compartir entre múltiples procesos. La tabla permite dos operaciones: *insert*, para agregar un nuevo objeto a la tabla, y *find*, para leer la información de un objeto almacenado en la tabla. Como la operación *insert* modifica el contenido de la tabla, exigimos que se ejecute bajo exclusión mutua, tanto con respecto a otras operaciones *insert*, como con respecto a operaciones *find*; es decir, cuando un proceso está ejecutando un *insert*, ningún otro proceso puede estar ejecutando alguna operación. Sin embargo, permitimos múltiples operaciones *find* concurrentes; es decir, si algún proceso está ejecutando un *find*, entonces otros procesos también pueden ejecutar *find* al mismo tiempo.

Escribe los protocolos de sincronización de **a**) *insert* y **b**) *find*, que empleen, por ejemplo, semáforos binarios.

Se necesita un semáforo —r— para asegurar la exclusión mutua total de los insert; se necesita un contador —n— para saber cuántos procesos están haciendo find; y finalmente se necesita otro semáforo —s— para tener acceso a este contador.

```
int n = 0
binarySemaphore r = 1, s = 1

find:
    P(s)
        n = n+1; if (n == 1) P(r)
    V(s)
    —aquí se hace el find propiamente tal
    P(s)
        n = n–1; if (n == 0) V(r)
    V(s)

insert:
    P(r)
    —aquí se hace el insert propiamente tal
    V(r)
```

Examen (preguntas adicionales)

Estructuras de Datos y Algoritmos – IIC2133

2 diciembre 2013

Reemplazo de I2

A) Con respecto a los árboles rojo-negro:

a) ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción? Justifica.

Al insertar un nodo, x, lo insertamos como una hoja y lo pintamos de rojo. Si su padre, p, es negro, terminamos. Si p es rojo, tenemos dos casos: el hermano, s, de p es negro; s es rojo. Si s es negro, 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.

b) Justifica que los nodos de cualquier árbol AVL *T* pueden ser pintados "rojo" y "negro" de manera que *T* se convierte en un árbol rojo-negro.

Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja.

Esto es así: el árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa). En tal caso, el número de nodos en la ruta más larga crece según 2h, y el de la ruta más corta, según h+1, en que h es la altura del árbol.

Así, para cualquier ruta (simple) desde la raíz a una hoja, sea d la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros d nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar d nodos rojos.

B) Con respecto a los siguientes algoritmos de ordenación,

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

a) ¿Cuáles son estables y cómo se sabe que lo son? 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

[1] Insertionsort y Mergesort son estables.

[1] Insertionsort (diapositiva #19 de los apuntes) es estable porque al comparar las claves de b y a[j-1], "subimos" (o avanzamos hacia a[0]) solo si la clave de b es estrictamente menor que la de a[j-1].

[1] Mergesort (diapositiva # 26 de los apuntes) es estable porque al comparar el a[p], de la primera "mitad", con el a[q], de la segunda mitad, colocamos a[p] en el resultado temporal (tmp) solo si su clave es estrictamente menor que la de a[q].

b) Para los que no son estables, explica cómo se los puede hacer estables y a qué costo.

Heapsort y Quicksort no son estables.

[2] Una forma de hacerlos estables es guardar el índice de cada elemento (la ubicación del elemento al comienzo) junto con el elemento. Así, cuando comparamos dos elementos, los comparamos según sus valores (key) y, si son iguales, usamos los índi-ces para decidir.

[1] Por cada elemento, se necesita almacenar adicionalmente su índice original. Si los índices van entre 0 y n, cada uno se puede almacenar en log n bits. Así, en total, se necesita n log n espacio adicional.

Reemplazo de I3

C) Con respecto a los árboles de cobertura de costo mínimo (MST),

a) Demuestra o refuta la siguiente afirmación: Un árbol de extensión de costo mínimo incluye, para cada vértice v, la arista de costo mínimo entre las aristas incidentes en v.

La afirmación es verdadera; la demostramos por contradicción. Sea t el árbol; sea (v, u) la arista de costo mínimo incidente en v; y supongamos que (v, u) no está en t. Si incluimos (v, u) en t, formamos un ciclo; éste debe incluir una arista (v, x), $x \neq u$, cuyo costo es mayor que el costo de (v, u): $w(v, u) < w(v, x)$. Si sacamos (v, x) del ciclo volvemos a tener un MST cuya única diferencia con t es que la arista (v, u) reemplaza a la arista (v, x). Como $w(v, u) < w(v, x)$, este árbol tiene costo menor que t, contradiciendo la suposición de que t es un árbol de extensión de costo mínimo.

b) Si todos los costos de las aristas son números enteros en el rango 1 a W, en que W es una constante, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

*Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E α(V)) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal). Por lo tanto, Kruskal es O(E log E). Ahora, bajo la supsición dada y usando countingSort, podemos ordenar las aristas en O(W + E) = O(E) —**ya que W es constante**. De modo que ahora el algoritmo toma O(E α(V)).*

c) Si todos los costos de las aristas son distintos, muestra que el MST es único.

Como todos los costos de las aristas son distintos, para cada corte hay una única arista liviana. A partir de las diapositivas #47 a 49, deducimos que el MST es único.

D) Considera el ***algoritmo de Dijkstra***, que determina las rutas más cortas desde s a cada uno de los vértices v de un grafo direccional $G = (V, E)$ en que los costos de las aristas son ≥ 0:

```
void Dijkstra(Vertex s)
    Init(s); S = ∅;
    Queue q = new Queue(V)
    while ( !q.empty() )
        Vertex u = q.xMin()
        S = S ∪ {u}
        for ( each v in α[u] )
            reduce(u,v)
```

Aplícalo al grafo representado por la siguiente matriz de adyacencias, para determinar las longitudes de las rutas más cortas desde el vértice d (las casillas vacías representan ∞):

	a	b	c	d	e	f	g	h	i	j
a	0				1			10		
b		0	2							
c			0							
d	4			0				1		
e					0	3				
f		1	3			0	7		1	
g							0			
h					5			0	9	
i									0	2
j							1			0

Muestra los valores del vector **d** después de cada iteración.

d = [∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ ∞] = inicial
d = [4 ∞ ∞ 0 ∞ ∞ ∞ 1 ∞ ∞] = después que sale $u = d$
d = [4 ∞ ∞ 0 6 ∞ ∞ 1 10 ∞] = después que sale $u = h$
d = [4 ∞ ∞ 0 5 ∞ ∞ 1 10 ∞] = después que sale $u = a$
d = [4 ∞ ∞ 0 5 8 ∞ 1 10 ∞] = después que sale $u = e$
d = [4 9 11 0 5 8 15 1 9 ∞] = después que sale $u = f$
d = [4 9 11 0 5 8 15 1 9 ∞] = después que sale $u = b$
d = [4 9 11 0 5 8 15 1 9 11] = después que sale $u = i$
d = [4 9 11 0 5 8 15 1 9 11] = después que sale $u = c$
d = [4 9 11 0 5 8 12 1 9 11] = después que sale $u = j$
d = [4 9 11 0 5 8 12 1 9 11] = después que sale $u = g$