Estructuras de Datos y Algoritmos – IIC2133

I1

6 septiembre 2013

1. *Tablas ralas*. La Escuela de Ingeniería quiere desarrollar una aplicación que le permita manejar en memoria principal información acerca del desempeño de los estudiantes en los cursos que están tomando durante un semestre particular. La Escuela quiere llevar un registro de las notas parciales, pero también quiere saber rápidamente, p.ej., en cuáles cursos está inscrito un estudiante y qué estudiantes están inscritos en un cierto curso. Inicialmente, se pensó en emplear una tabla, en que las filas representan a los cursos y las columnas a los estudiantes. Pero luego quedó claro que desde el punto de vista de uso de memoria la tabla no es muy eficiente: si la Escuela tiene 4,000 estudiantes y ofrece 500 cursos al semestre, la tabla tendría que tener unas $4{,}000 \times 500 = 2{,}000{,}000$ casillas; pero si un estudiante toma en la práctica sólo 5 cursos al semestre, entonces sólo se estaría usando el 1% de las casillas—una tabla *rala*.

a) [**4 pts.**] Describe una estructura de datos basada en listas ligadas para resolver este problema más eficientemente, en que sólo sea necesario ocupar una cantidad de memoria del orden de los miles de casillas ($4{,}000 \times 5 = 20{,}000$), y no de millones; preferentemente, dibuja la estructura de datos.

Por cada estudiante inscrito en un curso, usamos un registro (objeto) de 5 campos: una identificación del estudiante, una identificación del curso, un puntero a la lista de notas parciales de ese estudiante en ese curso, un puntero (imaginémoslo vertical hacia abajo) al objeto que representa otro curso del mismo estudiante, y un puntero (imaginémoslo horizontal hacia la derecha) al objeto que representa otro estudiante en el mismo curso.

b) [**2 pts.**] Suponiendo que ni los estudiantes ni los cursos cambian durante el semestre (es decir, ni se agregan ni se eliminan), ¿cómo debiera funcionar tu estructura de datos para que el usuario de la aplicación pueda referirse a los cursos por sus códigos (p.ej., IIC2133) y a los estudiantes por sus RUT's?

Usamos un arreglo de tamaño 4,000 para los estudiantes y otro de tamaño 500 para los cursos. Cada casilla de estos arreglos tiene dos punteros: uno a la información del estudiante (rut, nombre) o del curso (código, nombre), respectivamente, y otro a uno de los objetos descritos en **a**). Usamos sendas funciones de hash para convertir rut's o códigos de cursos en índices en estos arreglos, y usamos estos índices como los identificadores de los estudiantes y de los cursos en los objetos de **a**).

2. *Salida del laberinto*. Podemos representar un laberinto como un arreglo bidimensional de caracteres: los pasajes se marcan con '0's, los muros con '1's, y la única salida con 'e'; todo el perímetro del laberinto está marcado con '1's, excepto la salida.

Describe un algoritmo para encontrar la salida del laberinto a partir de una cierta posición inicial, si es que existe un camino de '0's entre la posición inicial y la salida. La idea es que, estando en una posición cualquiera, hay que intentar seguir por cada una de las cuatro direcciones posibles: izquierda, derecha, arriba, abajo—siempre en el mismo orden, aunque el camino encontrado no sea óptimo. Si es imposible seguir (y aún no estamos en la salida), entonces hay que retroceder a la posición anterior e intentar una nueva dirección a partir de ahí (p.ej., si desde esta posición ya se había intentado seguir por la izquierda y luego por la derecha, ahora hay que intentar seguir por arriba). Puedes suponer que cuentas con un *stack* en el que puedes almacenar posiciones del laberinto.

Algoritmo:
inicializar el stack
celdaActual = *celda de partida —llamamos "celda" a cada posición del laberinto*
while (celdaActual *no es* celdadeSalida)
marcar celdaActual *como visitada*
poner en el stack las celdas vecinas no visitadas de celdaActual
if (*stack está vacío*)
"no se encontró salida"
else
celdaActual = *sacar una celda del stack*
"encontramos la salida"

3. *Ordenación a base de heaps*. Si bien los datos almacenados en un *max-heap* cumplen una ordenación sólo parcial—**a**) [**1 pt.**] ¿cuál?—en la práctica es posible ordenarlos totalmente y de manera eficiente, tanto en términos de uso de memoria, como en cuanto al número de pasos ejecutados.

La clave almacenada en un nodo es mayor que las claves almacenadas en los hijos del nodo.

La observación que hay que hacer es la siguiente: si sacamos un dato del *max-heap*, ejecutando `xMax()`, sale el dato más grande almacenado en el *max-heap* (y el número de datos almacenados se reduce en uno, es decir, `heapSize = heapSize - 1`). Si a continuación sacamos otro dato, nuevamente sale el dato más grande (y nuevamente `heapSize = heapSize - 1`); este nuevo dato corresponde al segundo más grande de los datos almacenados originalmente.

b) [**3 pts.**] Aprovechando esta observación y suponiendo que el *max-heap* está almacenado en un arreglo `a`, escribe un método `heapSort()` que ordene totalmente, de menor a mayor, los datos del *max-heap*, sin usar un arreglo adicional (es decir, los datos quedan ordenados en el mismo arreglo `a`).

```
heapSort()
    while (heapSize > 0)
        x = xMax() —el método visto en clase
        a[heapSize+1] = x
```

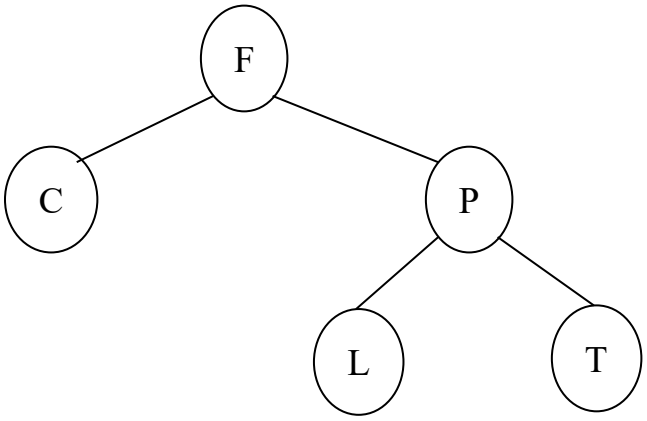
c) [**2 pts.**] Calcula (y justifica) el número de pasos que ejecuta el método, en notación O( ) y en función del número n de datos almacenados originalmente en el *max-heap*.

Cada llamada a `xMax()` ejecuta O(log n) pasos, ya que a su vez hace una llamada a `heapify(1)`. Como después de cada llamada, el tamaño del heap se reduce en 1 (y no vuelve a aumentar), tenemos que el número total de pasos es

$O(\log n) + O(\log n–1) + O(\log n–2) + \ldots + O(\log 1) = O(n \log n)$

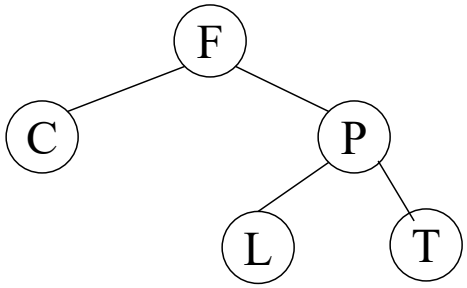
4. *Árboles AVL.*

a) [**4 pts.**] A partir del árbol AVL de la figura, muestra cómo va quedando el árbol después de insertar las claves B, V, X y R, una a continuación de la otra. Específicamente, después de cada inserción explica si se produjo o no un desbalance, y, en caso afirmativo, identifica el pivote, explica cómo se resuelve el desbalance, y muestra el árbol rebalanceado.

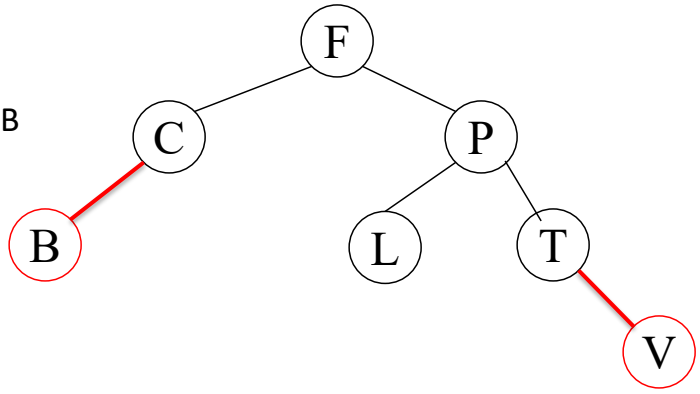


b) [**2 pts.**] Dibuja esquemáticamente (sin las claves) el árbol AVL más pequeño (con el menor número de nodos) de altura 6 (es decir, en que la rama más larga desde la raíz hasta una hoja tiene 7 nodos).

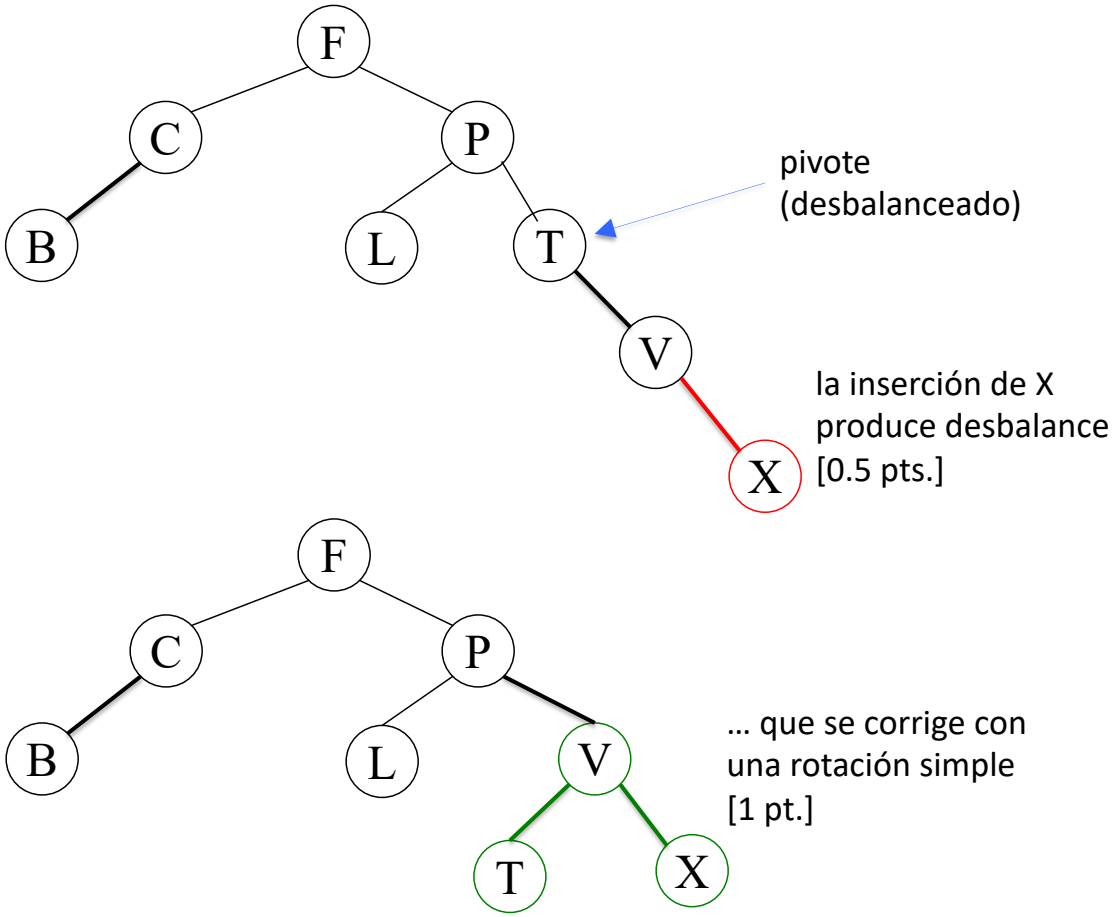
4a)

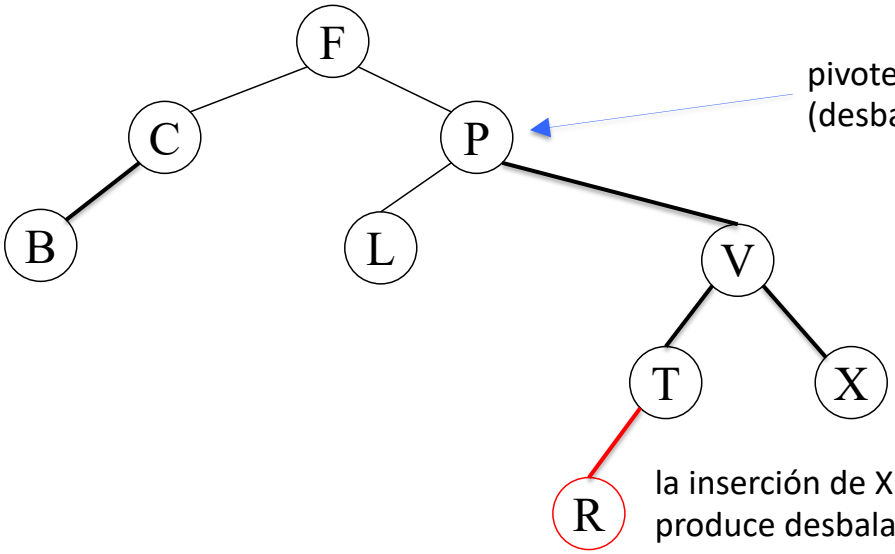


la inserción de B no produce desbalance [0.5 pts.]



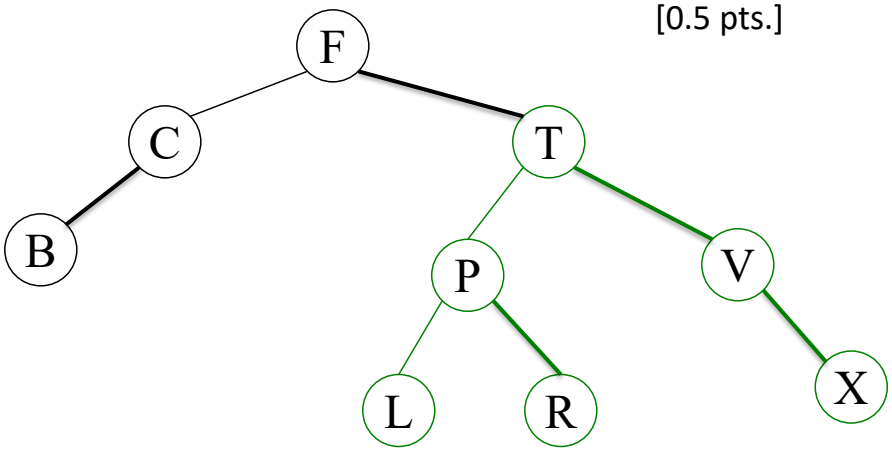
… la de V tampoco [0.5 pts.]


F
C
P
B
L
T
V
X
pivote
(desbalanceado)
la inserción de X
produce desbalance
[0.5 pts.]
F
C
P
B
L
V
T
X
… que se corrige con
una rotación simple
[1 pt.]



pivote
(desbalanceado)

la inserción de X
produce desbalance
[0.5 pts.]



… que se corrige con
una rotación doble
[1 pt.]

4b) El AVL más pequeño de altura 6 (AVL-6) está compuesto de un AVL más pequeño de altura 5 (AVL-5) y un AVL más pequeño de altura 4 (AVL_4); recursivamente, el AVL-5 está compuesto por un AVL-4 y un AVL-3; y el AVL-4, por un AVL-3 y un AVL-2.