Estructuras de Datos y Algoritmos – IIC2133

I1

14 abril 2016

Nota 1: *Tienes que sumar **18 puntos** para obtener un 7.*

Nota 2: *Cuando se pide un **algoritmo**, se espera algo parecido a lo siguiente*:

```
selectionSort(a):
   for k = 0 … n-1:
      min = k
      for j = k+1 … n:
         if a[j].key < a[min].key:
            min = j
      exchange(a[k], a[min])
```

1. [**5**] Tenemos una lista de $N(N–1)/2$ números enteros que representan las distancias entre todos los pares posibles formados a partir de N puntos ubicados sobre una línea recta. Queremos determinar la posición (relativa), x_i, de cada uno de los N puntos, suponiendo que el primer punto está en la posición 0, es decir, $x_1 = 0$. Escribe un algoritmo de ***backtracking*** para resolver el problema.

P.ej., si la lista es L = [1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10], claramente $x_6 = 10$, ya que 10 es la distancia más grande en la lista (dado que el largo de la lista es 15, sabemos que $N = 6$). Como la siguiente distancia más grande es 8, entonces $x_2 = 2$ o $x_5 = 8$. Como estos dos casos son simétricos, nos da lo mismo elegir cualquiera de ellos como válido; elegimos $x_5 = 8$. La siguiente distancia más grande en L es 7, de modo que $x_4 = 7$ o $x_2 = 3$. Si $x_4 = 7$, entonces las distancias $x_6 – 7 = 3$ y $x_5 – 7 = 1$ deberían aparecer en L; y, efectivamente, aparecen. Y si $x_2 = 3$, entonces $3 – x_1 = 3$ y $x_5 – 3 = 5$ deberían aparecer; y también aparecen. De modo que por ahora no podemos saber si $x_4 = 7$ o $x_2 = 3$. Tenemos que probar una de estas dos posibilidades y ver si nos lleva a una solución; si no nos lleva a una solución, entonces tenemos que probar la otra.

Suponemos L ordenada (si L no está ordenada, primero la ordenamos) y N correctamente dado (si N no está dado, lo calculamos a partir de $N(N–1)/2 = \text{length}(L)$); declaramos el arreglo x de N componentes: x_1 a x_N.

Tal como en el ejemplo, asignamos $x_1 = 0$ y $x_N = L[\text{length}(L)]$, y procesamos L en orden de $L[N–1]$ a $L[1]$, es decir, de mayor a menor. Para cada $L[k]$, $k = N–1, N–2, \ldots, 2, 1$, hacemos lo siguiente: **inferimos los valores de algunos x_q's de modo que sean consistentes** con los valores ya asignados a otros x_p's. Si con esto todos los valores de x quedan asignados, entonces quiere decir que encontramos una solución, así que imprimimos x y terminamos. Pero si aún quedan valores por asignar, entonces pasamos al próximo $L[k]$. Si al momento de inferir los valores, hay más de una forma de hacerlo, entonces **probamos cada una ⇒ *backtracking***.

Conviene manejar L como un conjunto dinámico, de manera que sea fácil eliminar y reinsertar (para el *backtracking*) el máximo y también otros valores arbitrarios; p.ej., un árbol de búsqueda, con valores repetidos (varias distancias pueden ser las mismas, como en el ejemplo). Así, en lugar de "para cada $L[k]$, $k = N–1, N–2, \ldots, 2, 1$", ejecutamos el equivalente a "para el $L[k]$ más grande que va quedando".

En el siguiente pseudocódigo, `asignar` es el procedimiento recursivo que efectivamente implementa el backtracking; recibe como parámetros `x`, `L`, `n` y los valores enteros `izq` y `der`, que delimitan el rango de índices de las posiciones `x[i]` que aún faltan por asignar. Antes de llamar a `asignar` por primera vez, asignamos `x[1] = 0`, `x[n]` = el máximo valor en `L` (que además lo eliminamos de `L`) y asignamos tentativamente `x[n-1]` = el (nuevo) máximo valor en `L` (que también lo eliminamos de `L`). Si `x[n]-x[n-1]` es un valor en `L`, entonces quiere decir que esta última asignación es plausible y llamamos a `asignar` para ver si efectivamente lleva a una solución; al hacer la llamada, `izq` = 2 y `der` = `n-2`, porque `x[2]`, …, `x[n-2]` aún no están asignados.

```
x[1] = 0
x[n] = delMax(L)
x[n-1] = delMax(L)
if x[n]-x[n-1] ∈ L:
    eliminar de L la distancia x[n]-x[n-1]
    return asignar(x, L, n, 2, n-2)
else:
    return False

def asignar(x, L, n, izq, der):
    if vacia(L):
        return True
    solucion = False
    dmax = max(L)
    if abs(x[j]-dmax) ∈ L, j = 1,…,izq-1,der+1,…,n:
        x[der] = dmax
        for j = 1,…,izq-1,der+1,…,n:
            eliminar de L la distancia abs(x[j]-dmax)
        solucion = asignar(x, L, n, izq, der-1)
        if not solucion:
            reinsertar en L las distancias eliminadas recientemente
    if not solucion and abs(x[n]-dmax-x[j]) ∈ L, j = 1,…,izq-1,der+1,…,n:
        x[izq] = x[n]-dmax
        for j = 1,…,izq-1,der+1,…,n:
            eliminar de L la distancia abs(x[n]-dmax-x[j])
        solucion = asignar(x, L, n, izq+1, der)
        if not solucion:
            reinsertar en L las distancias eliminadas recientemente
    return solucion
```

2. **a**) [**2**] Supongamos que la búsqueda de una clave k en un **árbol de búsqueda binario** termina en una hoja. Consideremos tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las claves a la derecha de la ruta de búsqueda. Tomemos tres claves: $a \in A$, $b \in B$ y $c \in C$. ¿Es cierto o es falso que $a < b < c$? Justifica.

Es falso y basta mostrar un contraejemplo.

b) [**2**] Dibuja un **árbol AVL**, preferiblemente pequeño, tal que las tres próximas inserciones, sin mediar eliminaciones, produzcan desbalances que deban corregirse con rotaciones dobles. Muestra el árbol inicial y, para cada inserción, muestra el desbalance, identifica el pivote y corrige el desbalance.

Ver archivo "I1-1-2016-pauta-p2b". La idea es que el árbol inicialmente tiene que tener nodos con balances –1 o +1, balance que después de la inserción queda –2 o +2. Para que la rotación necesaria sea doble, el nodo insertado tiene que ser un hijo izquierdo, si se insertó en el subárbol derecho del que va a ser finalmente el nodo pivote, o vice versa.

c) [**2**] Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un **árbol 2-3** inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

Un árbol 2-3 con una raíz y sus hijos tiene a lo más 4 nodos y puede almacenar a lo más 8 claves (dos claves por nodo). Como son exactamente 8 claves las que queremos almacenar, éstas tiene que quedar almacenadas de la siguiente manera: la raíz tiene las claves 5 y 18; el hijo izquierdo, 1 y 3; el hijo del medio, 8 y 13; y el hijo derecho, 19 y 24. Para lograr esta configuración final hay varias posibilidades; aquí vamos a ver una.

Podemos insertar primero las claves 1, 5 y 13, en cualquier orden, con lo cual queda 5 en la raíz, y 1 y 13 como hijos izquierdo y derecho. (En vez de 1 puede ser 3 y en vez de 13 puede ser 8. Por otra parte, en vez de empezar con 1, 5 y 13, es decir, empezar "por la izquierda", podríamos empezar por la derecha con 8-13, 18 y 19-24.) A partir de ahora, podemos insertar 3 en cualquier momento.

Ahora tenemos que conseguir que 18 quede en la raíz, junto con 5. Insertamos 18, que va a acompañar a 13, y a continuación insertamos 24, que va al mismo nodo de 13 y 18; como este nodo tiene ahora tres claves —13, 18 y 24 (lo que no puede ser)— lo separamos en dos nodos con las claves 13 y 24, respectivamente, y subimos la clave 18. Ahora podemos insertar 8 y 19, en cualquier orden.

3. **a**) [**3**] En clase estudiamos los procedimientos **`up`** y **`down`** que operan sobre ***maxHeaps***; su propósito es restaurar la propiedad de *maxHeap* cuando el valor de una clave (o prioridad) almacenada en el heap aumenta o disminuye, respectivamente. Una forma de construir un *maxHeap* a partir de las claves almacenadas en un arreglo **`q`** —con n claves, en las posiciones **`q`**[1] hasta **`q`**[n]— es ir insertando las claves una a una en un heap inicialmente vacío, usando el procedimiento **`insert`** estudiado en clase, que a su vez hace uso del procedimiento **`up`**; llamémosla la forma U.

Una segunda forma —llamémosla D— es aplicar el siguiente algoritmo al propio arreglo **`q`**:

```
k = n/2
while k ≥ 1:
    down(k)
    k = k-1
```

Compara las formas U y D de construir el *maxHeap*. En particular, **i**) calcula detalladamente el número de operaciones —básicamente, comparaciones e intercambios— que cada una realiza en el peor caso; y **ii**) demuestra que ambas construyen el mismo *maxHeap*, o bien da un ejemplo que muestre que en general construyen *maxHeaps* distintos (para un mismo arreglo **`q`**).

i) La forma U requiere O($n$log$n$) operaciones; la forma D, sólo O(n).

En U, insertamos la primera clave en un heap vacío, la segunda, en uno con una clave, la tercera, en uno con dos claves, …, la k-ésima, en uno con k–1 claves, …, y la n-ésima, en uno con n–1 claves. En cada inserción hay que comparar la clave que estamos insertando con la clave del padre y, si corresponde, intercambiar ambas claves; y así sucesivamente hasta llegar a la raíz, haciendo un número constante de operaciones —una comparación y a lo más un intercambio— por nivel. Es decir, el número de operaciones es

$T_U(n) = c_U(\lceil \log 1 \rceil + \lceil \log 2 \rceil + \ldots + \lceil \log n \rceil) = \mathrm{O}(n \log n)$

En D, consideramos que las n/2 claves en la segunda mitad de `q` forman un heap c/u, de tamaño 1, por lo que inicialmente las "saltamos" y procesamos las n/2 claves en la primera mitad de "atrás para adelante" (por eso, `k = n/2` antes de entrar al `while`, y `k = k-1` antes de salir). El procesamiento consiste en comparar la clave en la posición k con las claves hijas, en las posiciones $2k$ y $2k$+1, y, si corresponde, intercambiarla con la mayor de las hijas, en cuyo caso repetimos estas acciones con respecto a las nuevas hijas de la clave (si las hay). Así, procesar una clave en un nivel toma un número constante de operaciones: a lo más 3 comparaciones y un intercambio. El número total de operaciones de D sale de notar que n/4 claves hay que procesarlas en sólo un nivel, n/8 claves en a lo más dos niveles, n/16 claves en a lo más 3 niveles, etc.:

$T_D(n) = c_D(\, n/4 + 2(n/8) + 3(n/16) + 4(n/32) + \ldots + (k-1)(n/2^k) + \ldots + (\log n - 1)\,) = \mathrm{O}(n)$

ii) Ejemplo de que U y D construyen *maxHeaps* **distintos**: Si `q` = [1, 2, 3], entonces U construye un heap con 3 en la raíz, 1 como hijo izquierdo y 2 como hijo derecho; en cambio, D construye uno con 3 en la raíz (la raíz tiene que tener la misma clave en ambos casos), 2 como hijo izquierdo y 1 como hijo derecho.

b) [**2**] El ítem más grande en un ***maxHeap*** está en la posición 1 del arreglo, y el segundo más grande está en la posición 2 o 3. Para un *maxHeap* de tamaño 31, da la lista de posiciones en donde **i) podría** estar el k-ésimo ítem más grande, y **ii) no podría** estar el k-ésimo ítem más grande; para k = 2, 3 y 4, suponiendo que los valores son distintos. Justifica.

i) La regla del *maxHeap* es que el ítem en la posición k es más grande que los ítemes en las posiciones $2k$ y $2k$+1; o, mirado desde otro punto de vista, dado un ítem, sólo sabemos que es más grande que sus hijos y **más pequeño que su padre**.

Esta última observación significa que los ítemes en una misma rama crecen en valor si recorremos la rama desde la hoja hacia la raíz. Por lo tanto, el segundo ítem más grande en el heap es uno que a lo más está a distancia 1 de la raíz (sólo puede haber un ítem más grande que el segundo más grande y ése tiene que estar en la raíz); es decir, las posiciones 2 o 3.

El tercer ítem más grande está a lo más a distancia 2 de la raíz; es decir, las posiciones 2, 3, 4, 5, 6 y 7. En particular, si el segundo ítem más grande está en la posición 2, entonces el tercero más grande puede estar en las posiciones 3, 4 o 5; y si el segundo está en la posición 3, entonces el tercero puede estar en las posiciones 2, 6 o 7.

Y el cuarto ítem más grande va a estar a lo más a una distancia 3 de la raíz; es decir, las posiciones 2, 3, …, 13, 14 y 15. Para acotar un poco más este conjunto, es necesario saber dónde están el segundo y el tercer ítemes más grandes, pero todas esas posiciones del arreglo son posibles para el cuarto ítem más grande.

ii) El complemento: el segundo no puede estar en la posición 4 (porque esta posición está a distancia 2 de la raíz) ni siguientes; el tercero no puede estar en la posición 8 (porque esta posición está a distancia 3 de la raíz) ni siguientes; y el cuarto no puede estar en la posición 15 ni siguientes.

4. a) [**2**] Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. ¿Qué tan eficientemente puede resolverse este problema? Justifica.

Se puede hacer en tiempo $O(n^2 \log n)$: Primero, ordenamos la lista de menor a mayor, en tiempo $O(n \log n)$. Luego, para cada par de números, sumamos los dos números y buscamos en la lista ya ordenada, empleando búsqueda binaria, un número que sea el negativo de la suma; si lo encontramos, entonces incrementamos el contador de los tríos que suman 0. Hay $O(n^2)$ pares (los podemos generar sistemáticamente con dos *loops*, uno anidado en el otro) y cada búsqueda binaria se puede hacer en tiempo $O(\log n)$.

b) [**2**] Considera la siguiente versión de **`mergeSort`**:

```
mergeSort(a,b,e,w)  —a es el arreglo a ordenar entre a[e] y a[w]; b es un arreglo auxiliar
    if e >= w:
        return
    m = (e+w)/2
    mergeSort(a,b,e,m)
    mergeSort(a,b,m+1,w)
    if a[m] > a[m+1]:
        merge(a,b,e,m+1,w)  —el procedimiento merge estudiado en clase
```

Demuestra que el número de comparaciones usadas por este algoritmo para ordenar un arreglo **`a`** que ya está ordenado es lineal con respecto al tamaño n del arreglo.

La versión es diferente a la estudiada en clase debido a la línea **`if a[m] ≥ a[m+1]:`**. Esta línea implica que el procedimiento **`merge`**, que es el que ocupa un tiempo proporcional a **`w`** – **`e`**, se va a ejecutar sólo cuando el dato más grande de la primera mitad del arreglo **`a`** sea mayor que el dato más pequeño de la segunda mitad, lo cual ***no ocurre nunca*** ya que el arreglo está ordenado.

Luego, la ecuación de recurrencia para este caso es $T(n) = 2T(n/2) + c$, con c = constante

(c representa la ejecución de `if e >= w:`, con resultado falso, seguida de `m = (e+w)/2`),

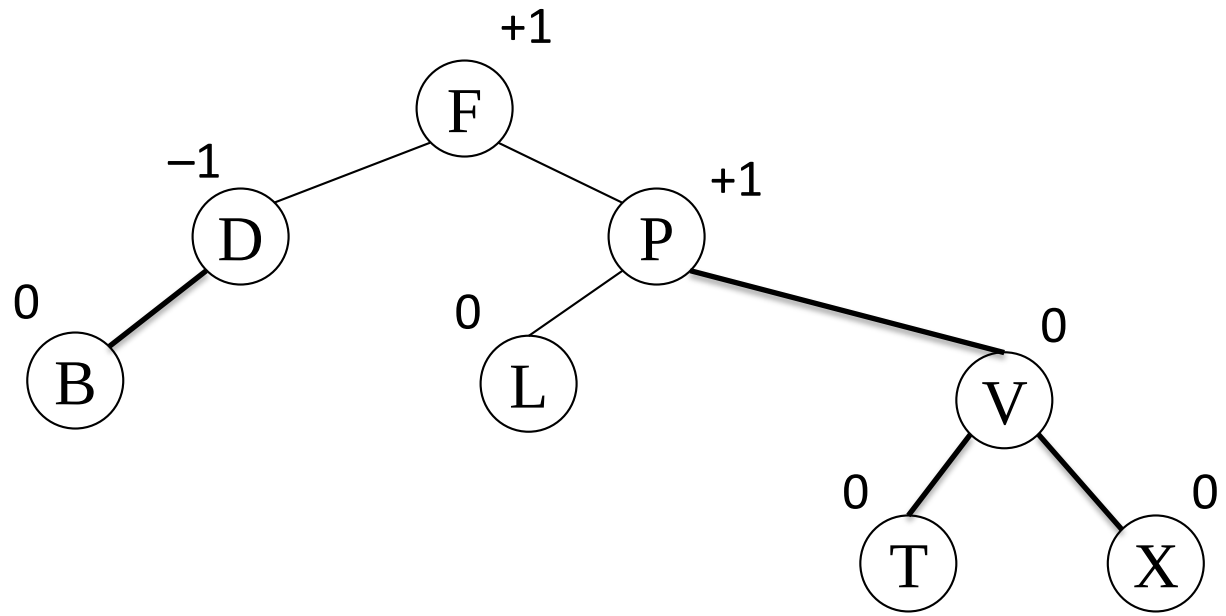
cuya solución es $T(n) = nT(1) + (n–1)c$; como $T(1) = 1$, por la línea **`if e >= w:`**, entonces $T(n) = n + (n–1)c = O(n)$.

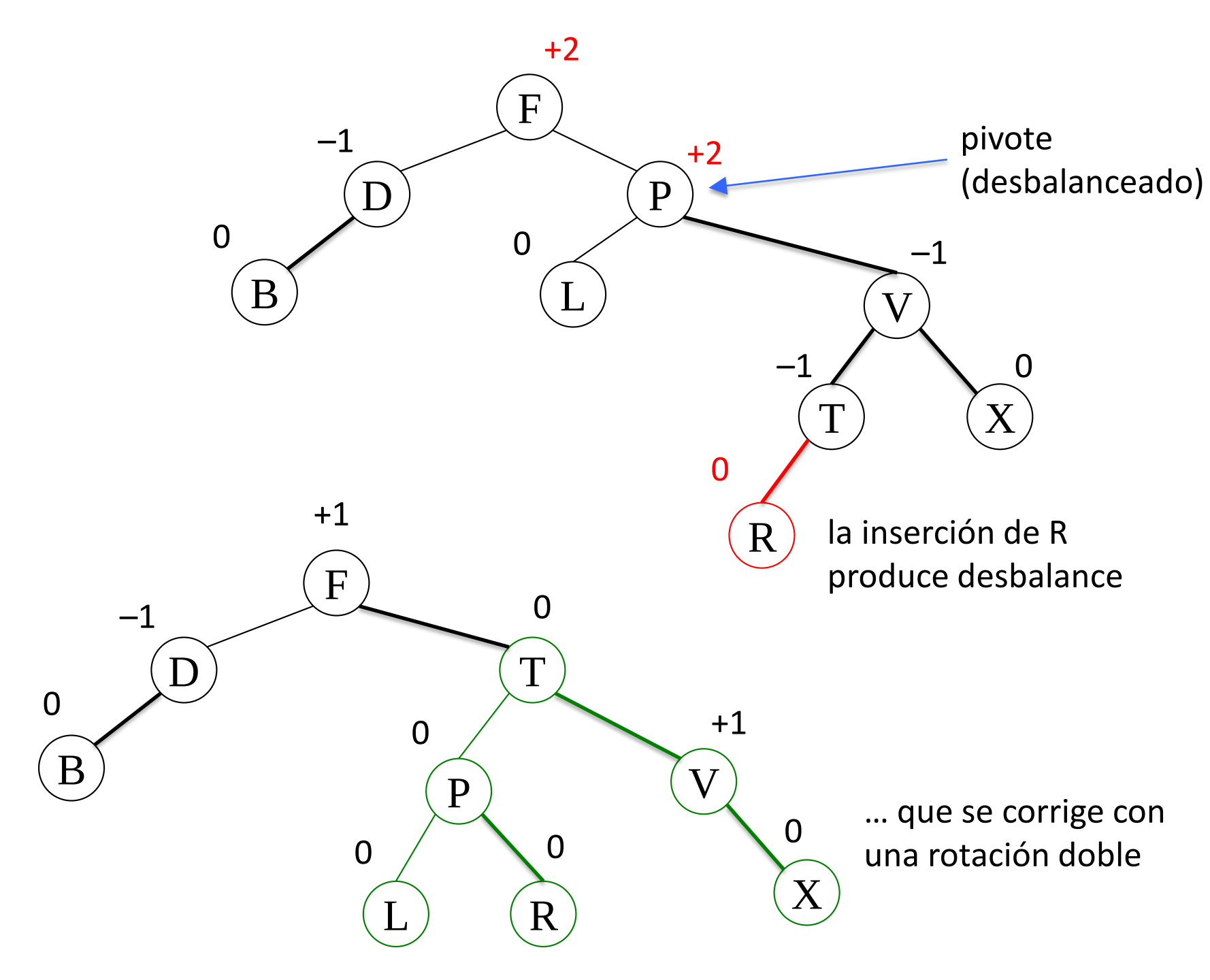
c) [**2**] Supongamos que elegimos el dato en la posición del medio del arreglo como pivote, en vez de elegir el dato en el extremo derecho (como en el procedimiento `partition` estudiado en clase). ¿Es ahora menos probable que **`quickSort`** requiera tiempo cuadrático? Justifica.

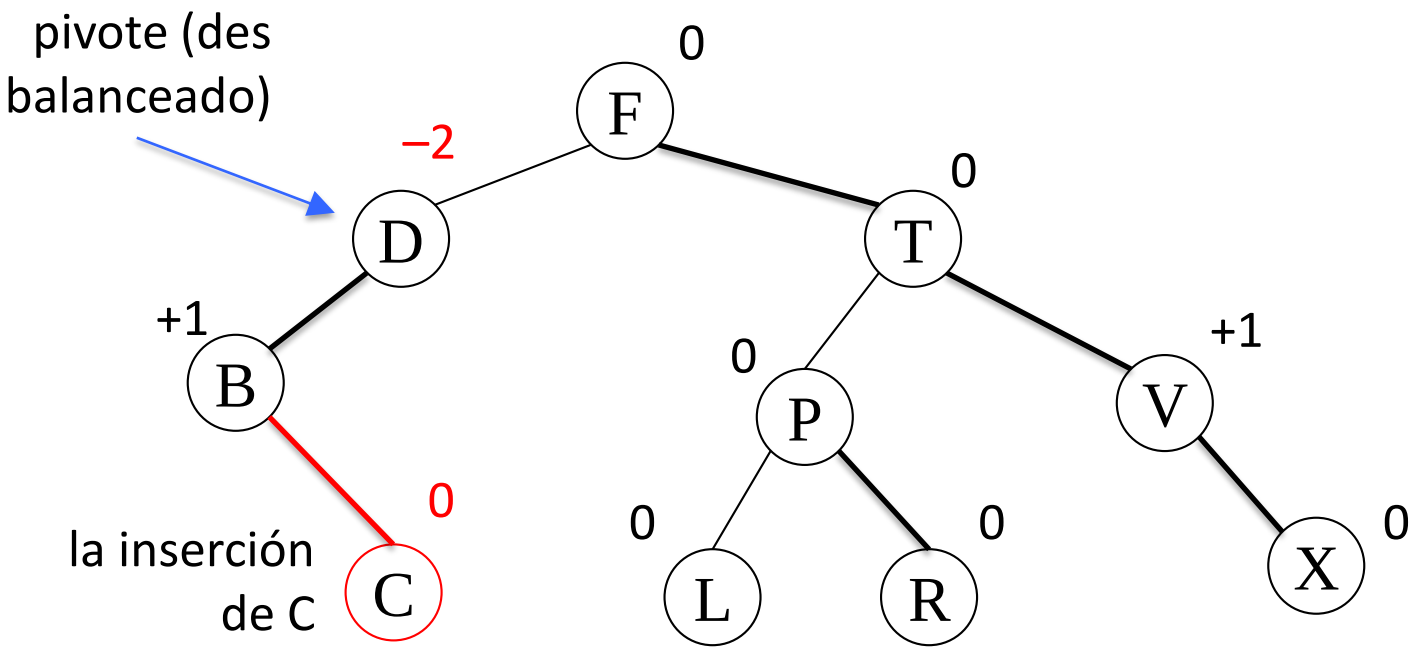
No, es igual de probable: siempre podría ocurrir que ese pivote es el dato más grande (o más pequeño) del (sub)arreglo.

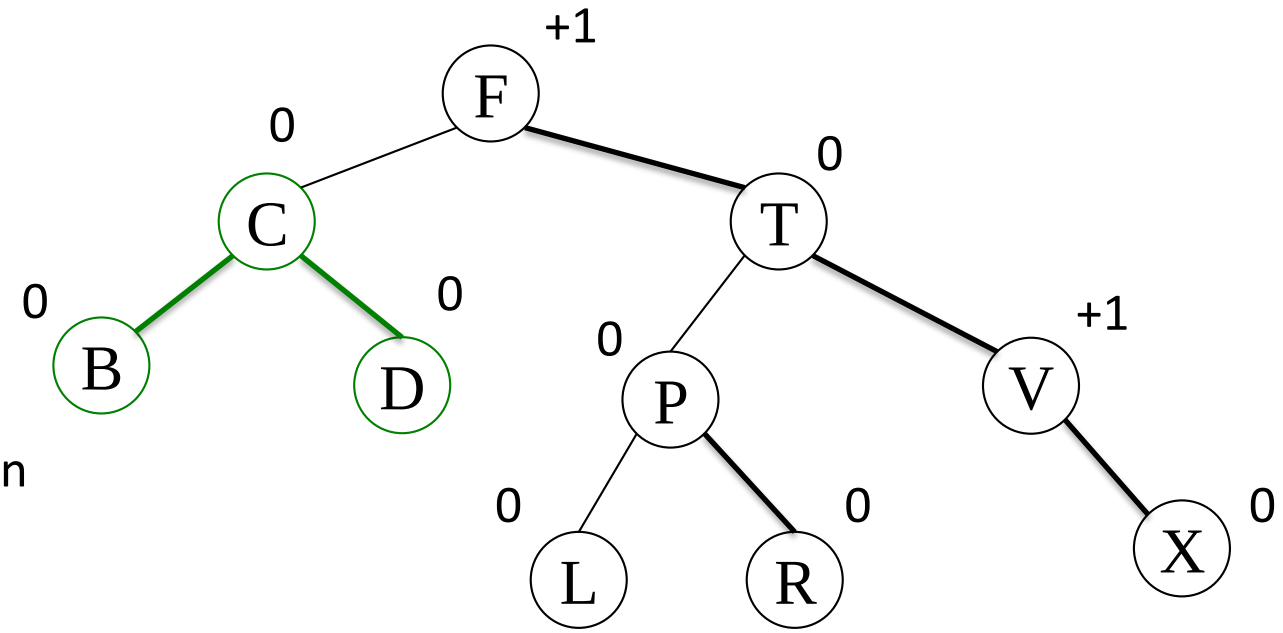
I1 (1-2016)
2b)

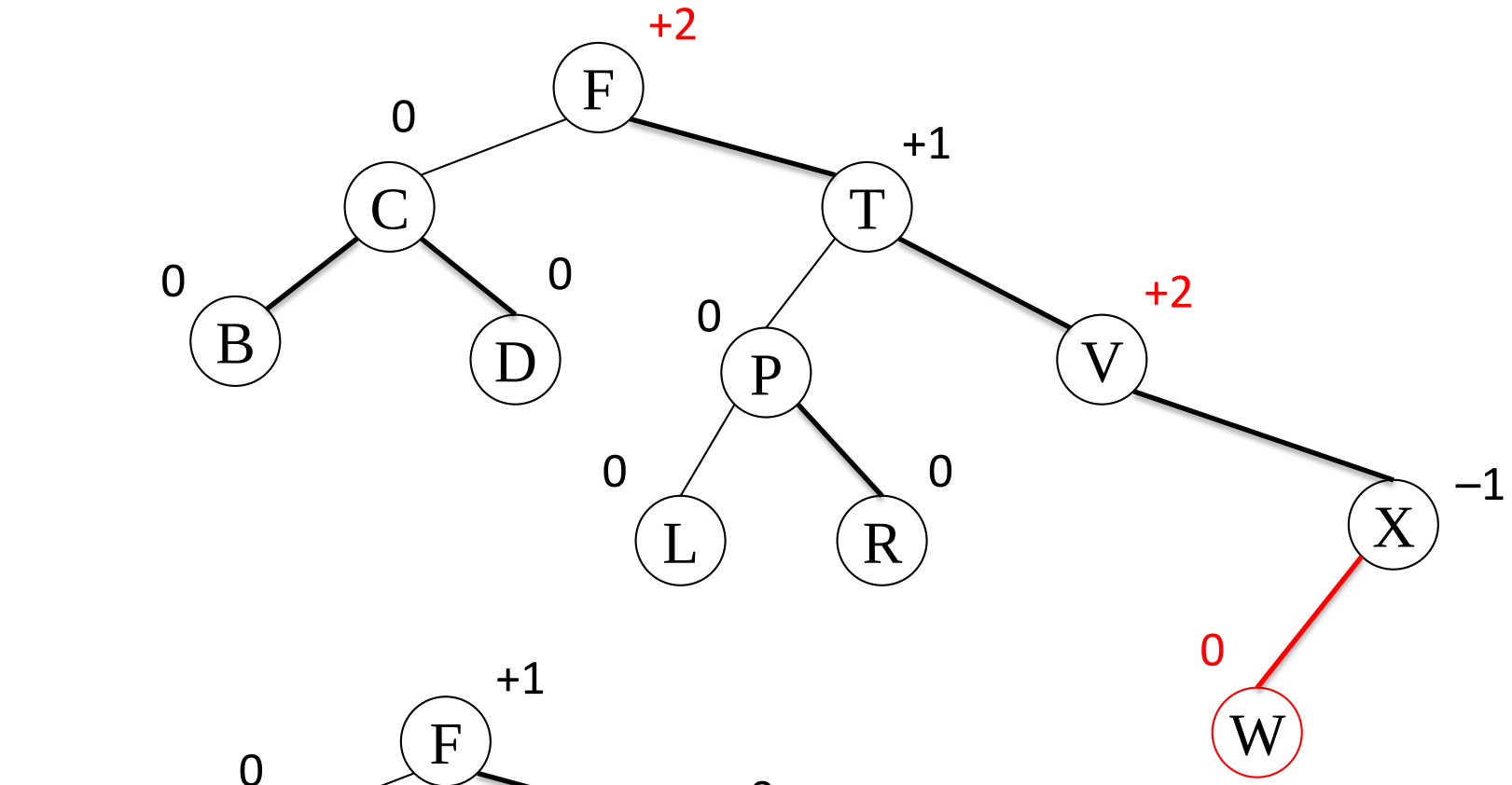
Árbol AVL inicial,
con balances

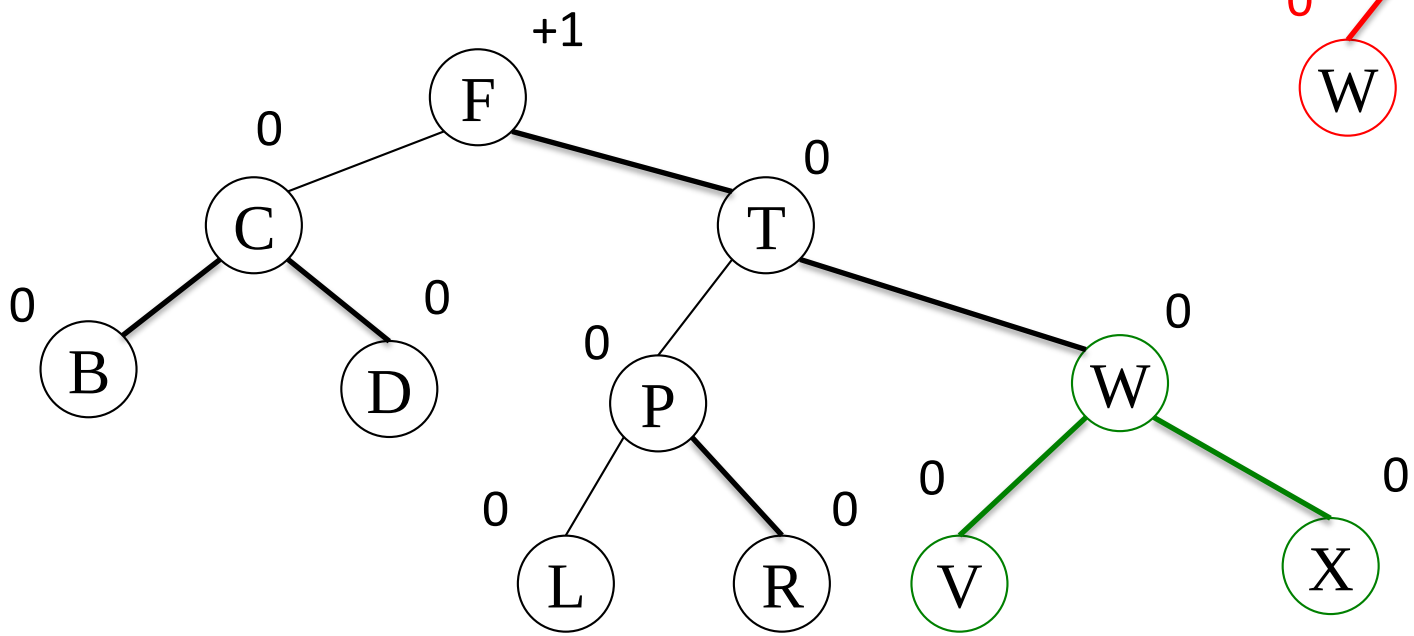



+2
F
–1
D
+2
P
pivote
(desbalanceado)
0
B
0
L
–1
V
–1
T
0
X
0
R
la inserción de R
produce desbalance
+1
F
–1
D
0
T
0
B
0
P
+1
V
0
L
0
R
0
X
… que se corrige con
una rotación doble


pivote (des balanceado)
Árbol AVL inicial, con balances
0
F
–2
0
D
T
+1
0
+1
B
P
V
0
0
0
0
la inserción de C produce desbalance
C
L
R
X


+1
F
0
0
C
T
0
0
+1
B
D
0
P
V
0
0
0
0
… que se corrige con una rotación doble
L
R
X


+2
F
0
+1
C
T
0
0
0
+2
B
D
P
V
0
0
0
–1
L
R
X
0
W


+1
F
0
0
C
T
0
0
0
0
B
D
P
W
0
0
0
0
L
R
V
X