Estructuras de Datos y Algoritmos – IIC2133

I1

31 agosto 2016

Nota: *Cuando se pide que des o describas 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. Dado un mapa, queremos colorear las regiones de manera tal que nunca dos regiones adyacentes (limítrofes) tengan el mismo color, usando a lo más cuatro colores. Para esto, conviene representar el mapa como un grafo, en que los nodos corresponden a regiones y hay una arista entre dos nodos si y sólo si las dos regiones correspondientes son adyacentes. Generalizando, el problema de colorear un grafo consiste en determinar todas las formas diferentes en que un grafo dado puede ser coloreado usando a lo más m colores. Escribe un algoritmo de ***backtracking*** para resolver este problema.

Suponiendo que el grafo tiene n nodos, representamos los nodos simplemente por los números 1, 2, …, n; y representamos las aristas por una matriz G tal que $G[i][j] = 1$ si y sólo si hay una arista entre los nodos i y j, y $G[i][j] = 0$ en otro caso. Además, representamos los m colores por los números 1, 2, …, m, y las soluciones por las tuplas $(x_1, \ldots, x_n)$, en que x_i es el color del nodo i.

Se sugiere escribir un ciclo principal, en que asignan todos los colores válidos para el nodo k; cada vez que se asigna un nuevo color al nodo k, este ciclo principal se llama recursivamente para asignar todos los colores válidos para el nodo k+1. Y se sugiere escribir otro ciclo para hacer la asignación de un color a un nodo; este ciclo asigna el "color" 0 si no puede asignar un color válido.

Respuesta:

Como se deduce del enunciado, sigue el patrón del problema de las 8 reinas.

Inicializamos el vector x en ceros.

Primero, asignamos el color 1 al vértice 1: $x[1] = 1$; y tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 2. En este proceso, cada vez que asignamos un color válido al vértice 2, tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 3; etc. Si no podemos asignar un color válido al vértice k, entonces tenemos que cambiar la última asignación de color que hicimos al vértice k–1. Cuando asignamos un color válido al vértice n, imprimimos el vector x.

En código:

```
colorear(k):
   repeat:
      proxColor(k)
      if x[k] == 0: return
      if k == n: print(x)
      else: colorear(k+1)
   until False

proxColor(k):
   repeat:
      x[k] = x[k]+1 % m+1
      if x[k] == 0: return
      for j = 1 … n:
         if G[k,j] ≠ 0 and x[k] == x[j]: break
      if j == n+1: return
   until False
```

2. Describe un algoritmo para eliminar una clave de un árbol 2-3; el algoritmo recibe como parámetros un puntero a la raíz del árbol y la clave que se va a eliminar. Por supuesto, tu algoritmo debe dejar como resultado un árbol 2-3. (En este problema, el algoritmo se puede describir en prosa y con dibujos.)

La idea general es primero encontrar la clave en el árbol [**a**) describe esta parte del algoritmo en términos de pasos más básicos: **1**].

Si la clave está en una hoja, entonces simplemente la eliminamos.

Si la clave está en la raíz o en un nodo interior, entonces la reemplazamos por la clave predecesora o la clave sucesora [**b**) describe esta parte del algoritmo en términos de pasos más básicos: **2**],

… y eliminamos (recursivamente) esta clave del árbol (hasta llegar a una hoja).

Si la hoja es un nodo 3, entonces estamos listos;

… si es un nodo 2, entonces hay que ser más creativos: considera que se produjo un "hoyo" en árbol y hay que hacerlo subir por el árbol hasta que pueda ser eliminado [**c**) describe esta parte del algoritmo en términos de pasos más básicos: **3**].

Respuestas:

a) Buscamos la clave en el nodo que estamos mirando; el primer nodo que miramos es obviamente la raíz del árbol. Si encontramos la clave en este nodo, entonces pasamos a **b**); de lo contrario, tenemos que descender a uno de los hijos del nodo y, recursivamente, buscar ahí. En este caso, usamos el hecho de que las claves del nodo están ordenadas para decidir a cuál hijo tenemos que descender: si la clave buscada es menor que la menor de las claves del nodo, entonces bajamos al hijo izquierdo del nodo; si la clave buscada es mayor que la mayor de las claves del nodo, entonces bajamos al hijo derecho del nodo; si la clave buscada está entre la menor y la mayor claves del nodo (en un nodo 3), entonces bajamos al hijo del medio.

b) Como es un nodo interior, entonces la clave predecesora (o la sucesora, se puede elegir cualquiera) está siempre en una hoja, más abajo en el árbol: la clave predecesora/sucesora es la más grande/pequeña de las claves almacenadas en el subárbol izquierdo/derecho (respecto de la clave que estamos eliminando). Para encontrarla (supongamos que buscamos la predecesora), bajamos a la raíz del subárbol correspondiente; si es una hoja, entonces es la clave más a la derecha en la hoja; de lo contrario, bajamos por el puntero más a la derecha del nodo y buscamos recursivamente ahí. Esta clave predecesora la ponemos en el lugar de la clave que queremos eliminar, sacándola de la hoja en la que estaba.

c) Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 2, con clave Y, entonces restructuramos este subárbol de la siguiente manera: ponemos el "hoyo" en el lugar en que está X y hacemos que X y Y formen un nodo 3 hijo del "hoyo"; con esto, el "hoyo" sube un nivel y hay que lidiar con él en este nuevo nivel.

Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 3, con claves Y y Z, entonces restructuramos así: Y queda en el lugar de X, X queda en el lugar del "hoyo", y uno de los hijos del nodo 3 original pasa a ser hijo de X. En este caso, el "hoyo" es efectivamente eliminado (no sube).

Si el "hoyo" es hijo de un nodo 3 y su hermano inmediato —a su derecha o su izqueirda— es un nodo 2 (el otro hermano puede ser un nodo 2 o un nodo 3), entonces cambiamos el padre por un nodo 2 (dejamos en el padre sólo una de las dos claves originales) y bajamos la otra clave, que pasa a formar un nodo 3 con el hermano inmediato del "hoyo". El "hoyo" es eliminado.

Finalmente, si el hermano inmediato del "hoyo" es un nodo 3, entonces separamos este hermano en dos nodos 2, de modo que uno de ellos reemplaza al "hoyo"; en este proceso, hay que redistribuir las claves de los dos nodos 3. El "hoyo" es eliminado.

3. Considera un **árbol de búsqueda** inicialmente vacío; y considera las siguientes 9 letras como claves a ser insertadas en el árbol: *A*, *C*, *E*, *H*, *L*, *M*, *P*, *R* y *S*. Ejecuta la inserción, letra por letra y en el orden dado, para cada uno de los siguientes tipos de árbol:

a) [**1**] Un árbol de búsqueda binario sin propiedades de balance.

b) [**2.5**] Un árbol AVL.

c) [**2.5**] Un árbol 2-3.

Respuesta: al final del pdf.

4. **a**) Considera las expresiones aritméticas de la forma (1 + ((2+3) * (4*5))), es decir, "totalmente parentizadas". (Estas expresiones se pueden definir formalmente de la manera recursiva: una *expresión aritmética* es ya sea un número, o un paréntesis abierto seguido por una expresión aritmética seguido por un operador seguido por otra expresión aritmética seguido por un paréntesis cerrado.)

E.W. Dijkstra desarrolló un algoritmo para evaluar este tipo de expresiones, empleando dos *stacks*, uno para los operandos y otro para los operadores, y revisando la expresión de izquierda a derecha:

- Coloca (*push*) los operandos en el stack de los operandos.
- Coloca (*push*) los operadores en el stack de los operadores.
- Ignora los paréntesis abiertos.
- Al encontrar un paréntesis cerrado, saca (*pop*) un operador, saca (*pop*) dos operandos, aplica el operador a los operandos, y coloca (*push*) el resultado en el stack de operandos.
- Cuando se procesa el último paréntesis cerrado, queda un valor en el stack de operandos; ese es el valor de la expresión.

i) [**1**] Ejecuta el algoritmo de Dijkstra para evaluar la expresión (1 + ((2+3) * (4*5))). Muestra el contenido de cada uno de los stacks a medida que vas procesando cada elemento de la expresión.

ii) [**2**] Analiza el algoritmo. ¿Cuántas operaciones básicas, o pasos, debe ejecutar en general, como función del largo de la expresión? ¿Cuál es la profundidad máxima que puede llegar a tener cada uno de los stacks, es decir, de qué propiedades de la expresión depende la profundidad y cómo depende?

Respuesta:

i)

stack de operandos	stack de operadores
Ø	Ø
1	Ø
1	+
1	+
1	+
1 2	+
1 2	+ +
1 2 3	+ +
1 5	+
1 5	+ *
1 5	+ *
1 5 4	+ *
1 5 4	+ * *
1 5 4 5	+ * *
1 5 20	+ *
1 100	+
101	

ii) Sea n el largo de la expresión y |h| que dice cuántas veces aparece h en la expresión.
Podemos notar por la estructura de la expresión que |(| = |)| = |operadores| = x.
Además que |operandos| = x+1.

Luego n = 4x+1, x = (n-1)/4

Por lo tanto:

|operadores| = (n-1)/4
|operandos| = (n+3)/4

Tamaño stacks:
El máximo del stack de operadores va a ser cuando estén todos los operadores en el stack, lo cual es (n-1)/4, lo mismo ocurre con el de operandos, el cual es (n+3)/4.
Cantidad operaciones:
- Por cada operando/operador se ejecuta un push (1 operación)
- Por cada) se ejecuta 3 pop, se calcula el valor y luego 1 push (5 operaciones)
Por lo tanto la cantidad de operaciones es:
x + (x+1) + 5x = 6x+1 = (3n-1)/2.
(1+(2*3))
n = 9, realizará (3*9-1)/2 = 13 operaciones.

b) *Min-heaps* binarios:

i) ¿Cuál es el número máximo de comparaciones que es necesario realizar para encontrar la clave más grande almacenada en un *min-heap* binario? Justifica

ii) Describe un algoritmo para encontrar todas las claves menores que algún valor, X, almacenadas en un *min-heap* binario? Tu algoritmo debe correr en tiempo $O(K)$, en que K es el número de claves que cumplen la condición.

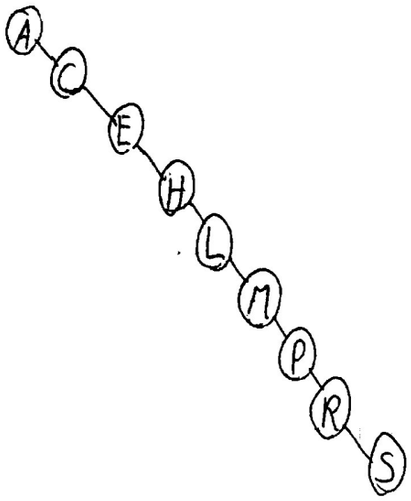
Respuestas:

i) La clave más grande sólo puede estar en una hoja del *min-heap* (mirado como árbol binario), ya que para cualquier nodo que tenga hijos, las claves de los hijos son mayores que la clave del nodo. Las claves en la hojas no tienen ningún orden particular, por lo que hay que mirarlas todas; en un *min-heap* con n claves, hay $\lceil n/2 \rceil$ hojas.

ii) Si la raíz es menor que X, entonces primero imprime la raíz, luego baja recursivamente al hijo izquierdo, y luego baja recursivamente al hijo derecho. Si el nodo que está mirando es mayor que X, entonces no hace nada con ese nodo.

PAUTA PREGUNTA 3.

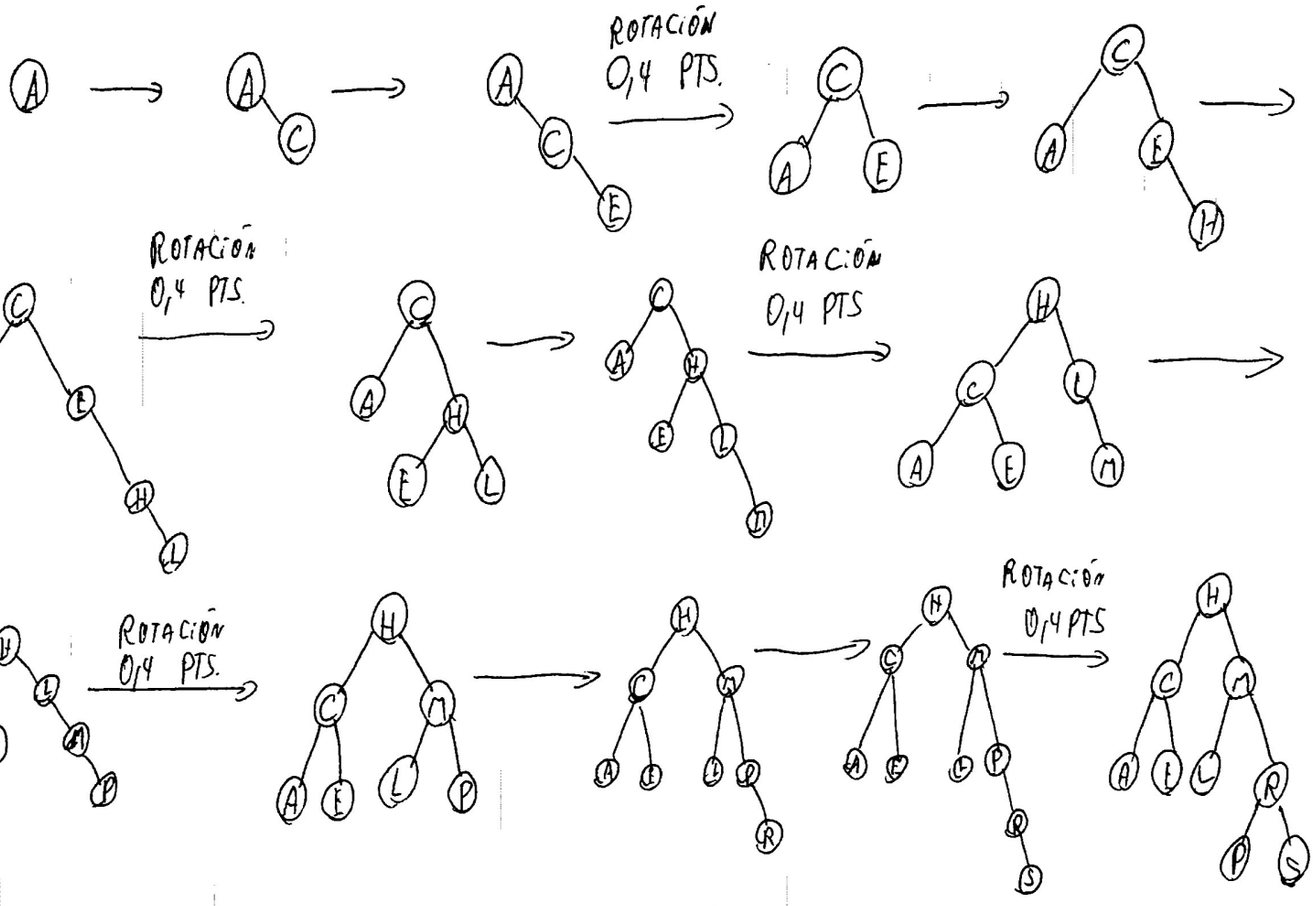
a)



1 PUNTO POR DEJAR EL ÁRBOL ASÍ.

(NO IMPORTA SI SE SALTAN LOS PASOS DE INSERCIÓN)

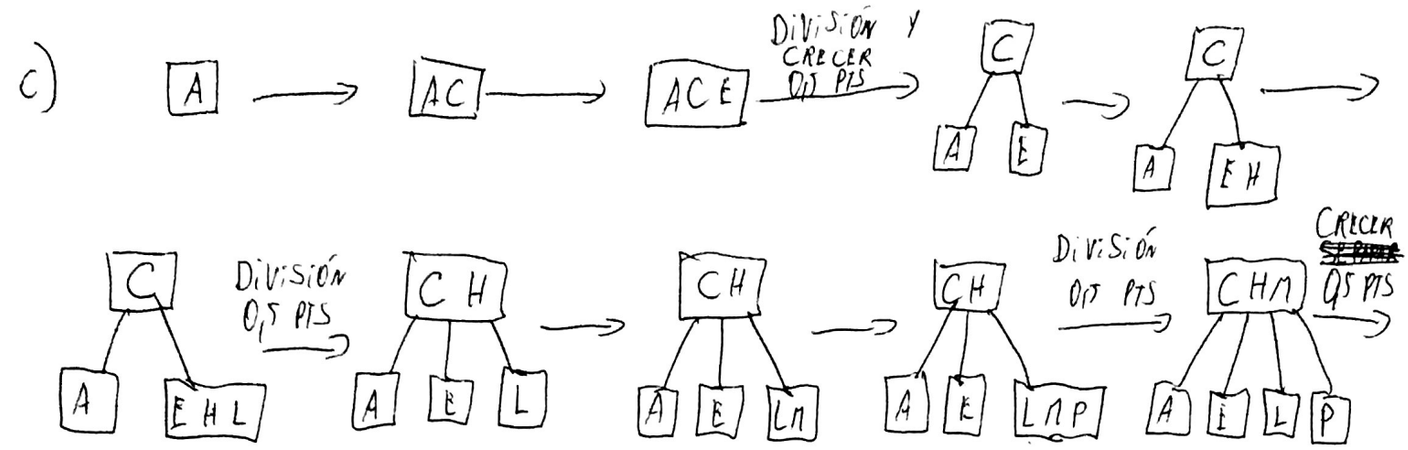
b)

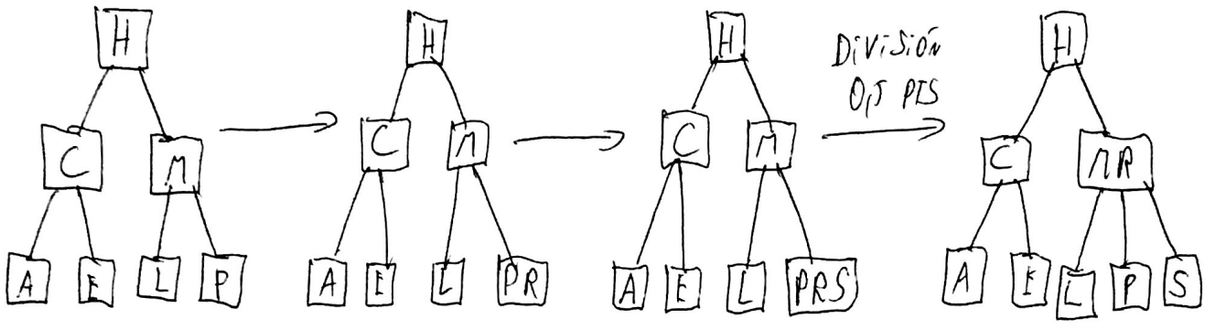


$\underbrace{0,4 \cdot 5}_{\text{ROTACIONES}} + \underbrace{0,5}_{\text{HACER TODAS LAS INSERCIONES EN LAS HOJAS CORRESPONDIENTES}} = 2,5 \text{ PTS.}$

SE PUEDEN SALTAR PASOS MIENTRAS QUEDE CLARO COMO LE HACEN LAS ROTACIONES INSERCIONES

c)





SI SE SALTAN PASOS PERO SI ENTIENDE, ESTÁ BIEN.