Estructuras de Datos y Algoritmos – IIC2133

I1

5 de septiembre, 2018

1. Árboles AVL

a) Queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 en un árbol AVL inicialmente vacío. ¿Es posible insertar estas claves en el árbol en algún orden tal que ***nunca*** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave k tal que el número de claves menores que k —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que k —que van a ir a parar al subárbol derecho— sean parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

b) Considera la inserción de una clave x en un árbol AVL T. Definimos la *ruta de inserción* de x como la secuencia de nodos, empezando por la raíz de T, cuyas claves son comparadas con x durante la inserción. El ***procedimiento de rebalanceo*** —una vez hecha la inserción— primero sube (de vuelta, desde el nodo recién insertado) por la ruta de inserción examinando cada nodo r que está en la ruta: si r es raíz de un (sub)árbol AVL-balanceado, entonces se sigue subiendo; de lo contrario, se ejecuta la rotación que vimos en clase en torno a la arista r—*hijo izquierdo* o r—*hijo derecho*, según corresponda. Muestra los árboles AVL que se van formando al insertar las claves 3, 2, 1, 4, 5, 6, 7 y 16, en este orden, en un árbol AVL inicialmente vacío.

Inicialmente se genera el árbol con raíz 3, y se agrega como hijo izquierdo el nodo con key 2. Luego se inserta a la izquierda de 2 la clave 1 y se hace una rotación simple en la que queda 2 como raíz y sus hijos 1 y 3 a la izquierda y derecha respectivamente.
Al insertar el 4 este queda a la derecha del 3, y luego al insertar el 5 este queda a la derecha del 4 y se hace una rotación simple entre 3, 4 y 5 y queda 4 como hijo derecho de 2 y 3 y 5 como hijos de 4 a la izquierda y derecha respectivamente.
Luego se inserta el 6 a la derecha del 5 y esto crea un desbalance en el nodo raíz, por lo que se hace una rotación simple entre 2, 4 y 5 quedando como raíz el nodo 4. Del nodo cuatro cuelga a la izquierda el nodo 2, el cual tiene a su izquierda el nodo 1 y a su derecha el nodo 3. A la derecha de 4 cuelga el nodo 5, el cual tiene el nodo 6 como hijo derecho.
Al insertar el nodo 7 este queda a la derecha del nodo 6 y se hace una rotación simple entre los nodos 5, 6 y 7, quedando el nodo 6 como padre de 5 y 7.
Al insertar el nodo 16 queda como hijo derecho del nodo 7 y no hay rotación.

2. Árboles de búsqueda binarios (no necesariamente balanceados)

En clase vimos cómo se elimina una clave de un árbol de búsqueda binario (ABB). Eliminar la clave cuando el nodo que ocupa no tiene hijos o tiene sólo un hijo, *Ti* o *Td*, es fácil.

a) 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***. [**1 pt.**]

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 la clave sucesora, que a lo más tiene un hijo.

b) ¿Es la eliminación por copia "conmutativa" en el sentido de que eliminar *x* y luego *y* de un ABB deja el mismo árbol que eliminar *y* y luego *x*? Demuestra que lo es o da un contraejemplo. [**1 pt.**]

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Consideremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.

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

A partir de la regla para insertar claves, que, a su vez, cumple 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 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) Muestra con ejemplos que la eliminación por mezcla puede tanto aumentar como reducir la altura del árbol original. [**2 pts.**]

Para simplificar (y generalizar un poco), eliminamos la raíz del árbol. Entonces, 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. Ordenación

Una observación que hicimos en clase sobre los algoritmos de ***ordenación por comparación de elementos adyacentes***, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operaciones que ejecutan) radica justamente en que sólo comparan e intercambian elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia de *insertionSort*( ), pero que comparara elementos que están a una distancia > 1 entre ellos, entonces podríamos esperar un mejor desempeño.

a) Calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenar el siguiente arreglo **a** de menor a mayor; muestra que entiendes cómo funciona *insertionSort*( ): **a** = **[**11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1**]**.

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc. En el caso del arreglo **a**, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces. Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = **55**.

b) Calcula ahora cuántas comparaciones entre elementos hace el siguiente algoritmo *shellSort*( ) para ordenar el mismo arreglo **a**. Muestra que entiendes cómo funciona *shellSort*( ); en particular, ¿qué relación tiene con *insertionSort*( )?

```
shellSort(a):
   gaps[] = {5,3,1}
   t = 0
   while t < 3:
      gap = gaps[t]
      j = gap
      while j < a.length]:
         tmp = a[j]
         k = j
         while k >= gap and tmp < a[k-gap]:
            a[k] = a[k-gap]
            k = k-gap
         a[k] = tmp
         j = j+1
   t = t+1
```

*Notemos que las comparaciones entre elementos de **a** se dan sólo en la comparación **tmp < a[k-gap]**; el algoritmo realiza **11** de estas comparaciones con resultado **true** y otras **17** con resultado **false**; en total, **28**.*

*Primero, realiza insertionSort entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en **a**, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.*

Luego, realiza insertionSort entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza insertionSort entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

c) **[**Independiente de a) y b)**]** Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. Da una forma de resolver este problema con complejidad mejor que $O(N^3)$.

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 $O(\log n)$.

4. *Merge* y *heaps*

En clase estudiamos el procedimiento *merge*, que recibe como input dos secuencias ordenadas de datos, S_1 y S_2, y produce como output una nueva secuencia ordenada con todos los datos de S_1 y S_2. Vimos que si la cantidad total de datos es n, entonces la complejidad de *merge* es O(n).

a) ¿Cómo funcionaría un procedimiento *merge-3*, que en lugar de recibir dos secuencias ordenadas, recibe tres secuencias ordenadas, S_1, S_2 y S_3, e igualmente produce como output una nueva secuencia ordenada con todos los datos de S_1, S_2 y S_3 ? Calcula cuál es la complejidad de *merge-3*, si la cantidad total de datos es n.

Al igual que al hacer merge con 2 secuencias, simplemente tomo el elemento menor de las secuencias y lo agrego a la secuencia final. Elegir el elemento menor toma a lo más 2 comparaciones, por lo que en total el algoritmo toma O(n) pasos.

b) Explica cuidadodamente cómo funcionaría un procedimiento *merge-k*, que recibe k secuencias ordenadas y produce una nueva secuencia ordenada con todos los datos de las k secuencias de input. Calcula cuál es la complejidad de *merge-k*, si la cantidad total de datos es n, en términos de n y k.

Esta vez tomar el menor elemento de las secuencias no es tan fácil. Para hacerlo utilizo un min heap de tamaño k en el cual agrego las cabezas de las secuencias. Hasta que no me queden elementos voy sacando la cabeza del heap y los remplazo por el siguiente elemento de la secuencia correspondiente. Esto toma tiempo O(nlogk).

Esto también se puede resolver en el mismo tiempo haciendo merge-2 entre pares de secuencias. Luego a las secuencias restantes se les aplica la misma operación hasta que solo queda una secuencia.