Estructuras de Datos y Algoritmos – IIC2133

I1

3 de abril, 2018

1. Árboles AVL

a) Considera un árbol AVL que almacena las claves 1, …, 6 de la siguiente manera: 4 está en la raíz, 2 y 5 son sus hijos; 1 y 3 son hijos de 2; y 6 es hijo de 5. En este árbol, inserta las siguientes claves, en el orden dado: 7, 16, 15, 14 y 13.

En cada caso, muestra el árbol resultante justo después de la inserción, pero antes de rebalancear el árbol (si fuera necesario), e indica si es necesario o no rebalancear el árbol. Si es necesario rebalancear el árbol, entonces explica por qué es necesario, identifica la acción que hay que realizar, y muestra el árbol resultante después de realizarla.

Respuesta

Denotaremos las aristas como "nodo padre"–"nodo hijo". La inserción de 7 produce un desbalance en 5 y requiere una rotación a la izquierda en torno a la arista 5-6 [**1 pt.**]. La inserción de 16 no produce desbalances [**0.5 pts.**]; pero la de 15 produce un desbalance en 7 y exige una doble rotación: a la derecha en torno a 16-15, y a la izquierda en torno a 7-15 [**1.5 pts.**]. La inserción de 14 produce un desbalance en 6 y también exige una rotación doble: a la derecha en torno a 15-7, y a la izquierda en torno a 6-7 [**2 pts.**]. Finalmente, la inserción de 13 produce un desbalance en 4 (la raíz) y exige una rotación en torno a 4-7.

El árbol resultante tiene a 7 en la raíz, con hijos 4 y 15. Los hijos de 4 son 2 y 6, los hijos de 2 son 1 y 3, y el único hijo de 6 es 5. Los hijos de 15 son 14 y 16, y el único hijo de 14 es 13.

b) Considera la inserción de una clave x en un árbol T. Definamos 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. Como vimos en clase, el algoritmo de rebalanceo primero sube (de vuelta) por la ruta de inserción examinando cada nodo que está ahí. Con respecto a esta "subida" (es decir, todavía antes de rebalancear propiamente), explica claramente:

- en qué consiste realmente "examinar" el nodo;
- qué casos pueden darse y qué hay que hacer en cada uno de ellos (para mantener el nodo actualizado); y
- cuándo se detiene.

Por último, al momento de detenerse la subida, y ya actualizado el nodo correspondiente, una posibilidad es que este nodo tenga el rol de "pivote" en el proceso de rotación que habría que hacer a continuación. Como también vimos en clase, la rotación necesaria puede ser simple o doble:

- ¿cómo sabe el algoritmo cuál rotación hay que aplicar?

Respuestas

Examinar el nodo consiste en revisar el valor del balance del nodo.

El valor del balance puede ser –1, 0 o +1. Si es 0, entonces hay que cambiarlo a –1 o +1, según corresponda, y seguir subiendo; solo en este caso se sigue subiendo.

Si, en cambio, el valor del balance del nodo examinado x es –1 o +1, entonces hay dos posibilidades, dependiendo del hijo de x desde el cual, al subir, se llegó a x: se pudo haber llegado a x por el lado que corrige el –1 o +1, y lo cambia a 0; o se pudo haber llegado por el lado que aumenta el desbalance (dejándolo temporalmente en –2 o +2):

- El primer caso es cuando llegamos a x desde su hijo derecho y el balance de x es –1, o cuando llegamos a x desde su hijo izquierdo y el balance de x es +1. En este caso, solo hay que cambiar el balance de x a 0 (y no se sigue subiendo).
- El segundo caso es cuando llegamos a x desde su hijo derecho y el balance de x es +1, o cuando llegamos a x desde su hijo izquierdo y el balance de x es –1. Lo que ocurre aquí es que x queda desbalanceado según el criterio de balance AVL; x es el pivote que vimos en clase. En este caso, hay que efectuar un rebalanceo por la vía de una rotación, ya sea simple o doble (y tampoco se sigue subiendo).

Así, la "subida" se detiene cuando encuentra un nodo con balance –1 o +1. Entonces, se procede a actualizar el balance, tal como está explicado.

La "subida" también se detiene si se llega a la raíz.

Finalmente, en el caso en que hay que realizar una rotación (el "segundo caso"), para saber cuál es la rotación que hay que realizar, hay que recordar los dos últimos nodos examinados antes de llegar a x, es decir, el hijo y y el nieto z de x: si y y z son ambos hijos izquierdos o son ambos hijos derechos, entonces la rotación necesaria es simple; en cambio, si uno es hijo izquierdo y el otro es hijo derecho, entonces la rotación necesaria es doble.

2. Heaps

Tienes dos heaps: A, de tamaño m, y B, de tamaño n. Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los $m+n$ elementos son todos distintos, si $m = 2^k-1$ para algún entero k, y si $m/2 < n \leq m$, explica cómo construir un heap C con los elementos de $A \cup B$.

a) ¿Cuál es la profundidad del árbol A y cuál es la profundidad del árbol B ?

Respuesta

Si consideramos la profundidad de la raíz como 0, entonces la profundidad tanto de A como de B es $k-1$.

b) ¿Cuál va a ser la profundidad del árbol C ? Justifica.

Respuesta

Dado que tanto A como B tienen profundidad $k-1$, al unirlos en un nuevo árbol, C, mediante una raíz común, este nuevo árbol necesariamente tiene profundidad k.

También se puede argumentar que, en el extremo, es decir, si $n = m$, entonces el ábol resultante de la unión tiene $2^k-1 + 2^k-1 = 2^{k+1}-2$ elementos. Como $2^{k+1}-2 = (2^{k+1}-1)-1$, se trata de un árbol binario completo en todos sus niveles menos en el último (le falta un elemento), por lo que su profundidad es k.

c) Describe un algoritmo *eficiente* para la construcción de C. Explica cuál es la complejidad de tu algoritmo.

Respuesta

Saquemos el último elemento de B, es decir, el de más a la derecha en el nivel de más abajo; llamémoslo x. Creamos un nuevo árbol binario con x como raíz y A y B como subárboles. Todo esto es O(1). Finalmente, aplicamos *heapify* sobre x. Esta operación, como vimos en clase, es $O(k) = O(\log m)$.

3. Ordenación por comparación de elementos adyacentes

Sea a un arreglo de números distintos. Si $j < k$ y $a[j] > a[k]$, entonces el par (j, k) se llama una *inversión* de a.

a) ¿Cuál es exactamente el número promedio de inversiones que puede tener un arreglo a de n números distintos? (Hint: Considera el arreglo a y el arreglo a totalmente invertido, que llamamos a'; entonces el par (j, k) es una inversión de a o es una inversión de a'.) Justifica.

Respuesta

Dado cualquier arreglo a y su inverso a' el pal de elementos (i, j) es una inversión en alguno de los dos arreglos. En total hay n*(n-1)/2 pares de elementos, por lo que en promedio hay n*(n-1)/4 inverisones.

b) A partir de (a), justifica que cualquier algoritmo de ordenación que ordena intercambiando elementos adyacentes —por ejemplo, *insertionSort*— requiere tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

Respuesta

Al intercambiar dos elementos adyacentes, sólo resolvemos una inversión a la vez. Como en promedio hay $n(n\text{-}1)/4$ inversiones, el algoritmo necesita realizar $n(n\text{-}1)/4 = \Omega(n^2)$ intercambios en promedio.

c) ¿Cuál es la relación entre el tiempo de ejecución de *insertionSort* y el número de inversiones del arreglo de entrada? Justifica.

Respuesta

Dado un arreglo a de entrada de n elementos, siempre debe iterar sobre el arreglo completo, por lo que como mínimo toma tiempo $\Omega(n)$. Luego, por cada inversión insertion sort hace un intercambio. Por lo tanto, el tiempo de insertion sort es O(n + inversiones), lo que en el mejor caso es O(n) y en peor caso es $O(n^2)$.

4. mergeSort y quickSort

a) Siguiendo con las inversiones, definidas en la pregunta 3, describe un algoritmo, basado en *mergeSort*, que determine el número de inversiones de un arreglo de n números distintos en tiempo proporcional a $n\log n$. Explica la complejidad de tu algoritmo.

Respuesta

El número de inversiones de un arreglo es el número de inversiones que hay en su mitad izquierda, más el número de inversiones que hay en su mitad derecha, y más el número de inversiones que hay entre los elementos de la mitad izquierda con respecto a los elementos de la mitad derecha. Sea **a** el arreglo, y **e** y **w** los índices extremos de la parte de **a** en que vamos a calcular el número de inversiones:

```
invCount(int[] a, int e, int w):
     if (e < w):
          int m = (e+w)/2
          return invCount(a, e, m) + invCount(a, m+1, w) + countInvs(a, e, m, w)
     else return 0

int countInvs(int[] a, int e, int m, int w):
     int[] b = new int[a.length]
     int j = e, k = m+1, p = 0, count = 0
     while (j <= m && k <= w):
          if (a[j] > a[k]):
               b[p] = a[k]; k++; count = count + (m-j+1)
          else:
               b[p] = a[j]; j++
          p++
     while (j <= m): b[p] = a[j]; j++
     while (k <= w): b[p] = a[k]; k++
     p = 0;
     for (j = e; j <= w; j++): a[j] = b[p]; p++
     return count
```

b) Como vimos en clase, el desempeño de *quickSort* depende fuertemente de cómo resultan las particiones, lo que a su vez depende de cuál elemento del (sub)arreglo se elige como pivote.

- Explica cuál es la relación entre el resultado de las particiones y el desempeño de *quickSort*.
- Explica los pro y los contra de elegir como pivote el elemento de más a la derecha del (sub)arreglo, como es el caso de la versión del algoritmo que vimos en clase.
- Otro método para elegir el pivote es el llamado "la mediana de 3": se elige como pivote la mediana entre el elemento de más a la izquierda del (sub)arreglo, el elemento al medio del (sub)arreglo y el elemento de más a la derecha del (sub)arreglo. ¿Qué ventajas presenta este método frente al anterior?

Respuesta

Particiones vs desempeño: Es importante la forma en que se obtiene la partición en cada llamado a _quicksort_ pues si esta es muy desbalanceada, habrá un impacto sobre la complejidad del algoritmo completo. En efecto, lo ideal es que las partes obtenidas luego de particionar un subarreglo sean de tamaños parecidos para que se realicen O(nlog(n)) llamados en total. Para lograr este objetivo, no

basta fijar una posición del pivote para particionar, sino que además se debe considerar si el arreglo ya está ordenado/semi-ordenado antes de comenzar.

Pivote extremo-derecho: Si se toma siempre el elemento extremo-derecho de cada sub-arreglo el algoritmo toma O(n^2) cuando el arreglo original está casi ordenado (respectivamente, ordenado de mayor a menor). Esto ocurre debido a que al armar las particiones, muy pocos elementos serán mayores (resp. menores) que el pivote y por lo tanto, el llamado recursivo siguiente se hará para un sub-arreglo de tamaño casi igual al actual. El peor caso es cuando el arreglo ya está ordenado, en cuyo caso cada llamado se hará para un sub-arreglo con un elemento menos que el paso anterior.

Mediana de 3: La ventaja inmediata es que este método no cae en los peores casos de usar como pivote el extremo-derecho o el extremo-izquierdo. Si el arreglo está ordenado/ordenado de mayor a menor, como el elemento central es la mediana de los tres elementos escogidos, no se hará un llamado con un sub-arreglo de largo similar al actual. En cualquier otro caso, el método no presenta mayor ventaja que usar un pivote aleatorio.