Estructuras de Datos y Algoritmos – IIC2133

I2

10 mayo 2013

1. Considera el algoritmo de ordenación *quickSort* que estudiamos en clase; a continuación repetimos una de sus partes, el algoritmo *partition*.

```
int partition(T[] a, int e, int w) {
   int j = e-1, k = w
   T v = a[w]
   while (true) {
      j = j+1
      while (a[j] < v) j = j+1
      k = k-1
      while (a[k] > v) {
         k = k-1
         if (k == e) break
      }
      if (j >= k) break
      exchange(a[j],a[k])
   }
   exchange(a[j],a[w])
   return j
}
```

a) ¿Cuál es el tiempo de ejecución de *quickSort* cuando todos los elementos del arreglo a son iguales? Justifica tu respuesta.

Respuesta:

Supongamos que al hacer la llamada a *partition*, $e = 0$ y $w = n$. Al ejecutarse la primera iteración del *while* exterior, $j = 1$. Como $a[j]$ no es menor que v (todos los elementos de a son iguales), el primer *while* interior no se ejecuta, y j queda en 1. Similarmente, $k = n\text{-}1$ inicialmente, y, como $a[k]$ no es mayor que v, k queda en n-1. Al ejecutarse la segunda iteración del *while* exterior, $j = 2$ y $k = n\text{-}2$; y así sucesivamente. Es decir, al terminar una iteración del *while* exterior, $k = n\text{-}j$. Como el *while* exterior termina cuando $j = k$, tenemos que $j = n/2$. Por lo tanto, $O(n \log n)$.

b) Escribe una versión no recursiva (y, por lo tanto, iterativa) de *quickSort*. Supón que dispones de un *stack* de números enteros con las operaciones *push*( ), *pop*( ) y *empty*( ) habituales. La idea es que después de llamar a *partition*, en lugar de llamar recursivamente a *quickSort* dos veces, coloques en el stack los índices correspondientes a cada una de las dos partes. Así, tu versión iterativa debe iterar mientras el stack no esté vacío.

Respuesta:

```
void iterQuicksort(int[] a)
   Stack s = new Stack()
   s.push(0); s.push(a.length-1)          Iniciar el stack con los índices extremos del arreglo original.
   int e, m, w
   while (!s.empty())
      w = s.pop(); e = s.pop()            Sacar del stack los índices extrremos de la parte a procesar.
      if (e < w)                          Verificar si la parte tiene al menos dos elementos.
            m = partition(a, e, w)
            s.push(e); s.push(m-1)        Colocar en el satck los índices extremos de la parte izquierda.
            s.push(m+1); s.push(w)        Colocar en el stack los índices extremos de la parte derecha.
```

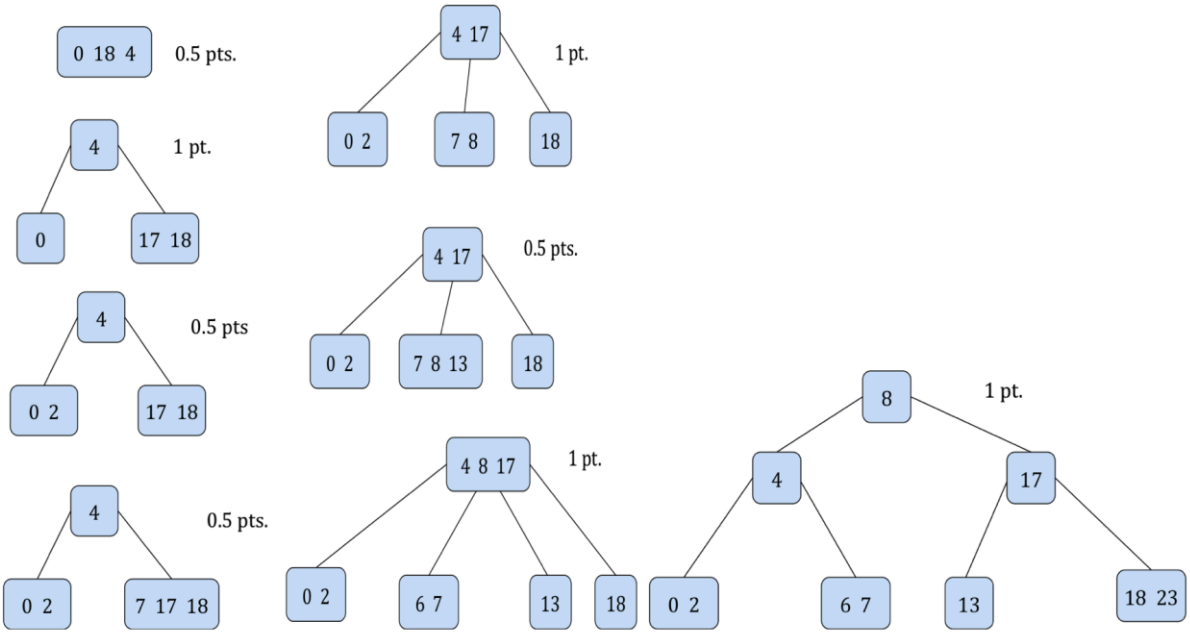
c) Con respecto a la pregunta **b**) anterior, ¿qué se puede hacer, al momento de colocar en el stack los índices correspondientes a cada una de las dos partes, de modo de minimizar la profundidad máxima del stack? Justifica tu respuesta.

Respuesta:

Hay que colocar primero en el stack los índices de la parte más grande. **Falta justificar.**

2. Dibuja los árboles-B resultantes al ir insertando las claves 4, 17, 2, 7, 8, 13, 6 y 23 en un árbol-B de grado mínimo $t = 2$ que inicialmente contiene un solo nodo con las claves 0 y 18.

Respuesta. El puntaje por cada nueva configuración correcta es el siguiente:



3) Explica cómo se podría resolver el problema de selección si los datos están almacenados en un árbol binario de búsqueda (ABB), y, por lo tanto, están ordenados. Así, si no hacemos nada especial con el árbol, cuando nos pidan el k-ésimo dato más pequeño (éste es el *problema de selección*), podríamos hacer un recorrido del árbol en preorden y detenernos cuando hayamos impreso los k datos más pequeños. Sin embargo, este procedimiento tiene complejidad O(n), tanto en promedio, como en el peor caso.

Por lo tanto, la idea es agregar información adicional al árbol, que, aprovechando el hecho de que es un ABB, permita determinar más eficientemente el k-ésimo dato más pequeño. Específicamente:

a) ¿Qué información adicional es necesario almacenar en el árbol?

Respuesta:

Si nos basamos en la estrategia que usa *randomSelect*, podemos almacenar en un campo *size* de cada nodo r del árbol el número total de nodos que hay en el subárbol cuya raíz es r. Este número es igual a la suma del número de nodos que hay en el subárbol izquierdo de r ($r.left.size$) más el número de nodos que hay en el subárbol derecho de r ($r.right.size$) más 1 (el propio r).

b) ¿Cómo se determina el k-ésimo dato más pequeño en este nuevo escenario?

Respuesta:

Si nos basamos en la estrategia que usa *randomSelect*, comparamos k con $r.size$: si son iguales, entonces la respuesta es $r.key$; si $k < r.size$, entonces buscamos el k-ésimo elemento más pequeño recursivamente en el subárbol izquierdo de r; y si $k > r.size$, entonces buscamos el $(k - (r.left.size + 1))$-ésimo elemento más pequeño recursivamente en el subárbol derecho de r.

c) ¿Cuál es la complejidad (de tiempo) de la operación anterior?

Respuesta:

O(h), en que h es la altura del árbol: hace un número constante de comparaciones en un nodo, y de ahí pasa al hijo izquierdo o derecho de ese nodo, si es que pasa, en donde repite el procedimiento. Es decir, en el peor caso hace un número constante de comparaciones por nivel del árbol.

d) ¿Cuánto cuesta mantener actualizada la información adicional del árbol cuando se produce una inserción o una eliminación de un dato?

Respuesta:

Básicamente, sumar (o restar) 1 al campo *size* de cada nodo en la ruta desde la raíz del árbol hasta el nodo insertado (o eliminado).

4) Con respecto a los árboles AVL:

a) Muestra la secuencia de árboles AVL's que se forman al insertar las claves 3, 2, 1, 4, 5, 6, 7, 16, 15 y 14, en este orden, en un árbol AVL inicialmente vacío.

Las inserciones de 3 y 2 son triviales (no tienen puntaje). La inserción de 1 exige una rotación simple a la derecha. La inserción de 4 nuevamente es trivial. Las inserciones de 5, 6 y 7 exigen rotaciones simples a la izquierda. La inserción de 16 es trivial. Finalmente, las inserciones de 15 y 14 exigen rotaciones dobles: en ambos casos, una rotación simple a la derecha seguida de una rotación simple a la izquierda. El árbol AVL resultante tiene como raíz a 4, cuyos hijos son 2 y 7. Los hijos de 2 son 1 y 3; los de 7 son 6 y 15. El único hijo de 6 es 5; los hijos de 15 son 14 y 16.

b) Se ha producido la inserción de un nuevo nodo *x* en un árbol AVL. Escribe el método

Node determinarPivote(*Node x*)

que a partir de *x* sube por el árbol, actualizando los balances de cada nodo encontrado en el camino, hasta llegar al primer nodo que quedó desbalanceado como consecuencia de la inserción de *x*; el método retorna este nodo (para permitir la ejecución de la rotación que corresponda).

Resouesta: Pendiente.

Tiempo: 120 minutos