Auxiliar #5 - Ejercicios avanzados de análisis amortizado

2 de noviembre de 2020 - Bernardo Subercaseaux

Problema 1. (★★) Una permutación P es un arreglo de tamaño n que contiene exactamente a los elementos $\{1, \ldots, n\}$. Un ciclo en P es una secuencia $i \to P[i] \to P[P[i]] \to \ldots \to i$. Diseñe un algoritmo que cuenta la cantidad de ciclos disjuntos usando espacio adicional $O(1)$.

Solución 1 . Mantendremos un contador de ciclos. Un algoritmo consiste en iterar con $i = 1..n$, en cada paso marcando todos los elementos pertenecientes al ciclo de i. Es decir, se inicializa $j \coloneqq$ `origen` $\coloneqq i$, y luego en cada paso se avanza en el ciclo haciendo $j \coloneqq P[j]$, y marcando que esa posición ya ha sido visitada haciendo por ejemplo $P[j] \coloneqq -P[j]$. Además, se debe revisar si el ciclo se cerró, comparando con la variable `origen`. Al cerrar un ciclo se aumenta el contador, y si incrementa i. Si al aumentar i ocurre que $P[i]$ es una posición ya marcada, se vuelve a aumentar i.

Para analizar la complejidad del algoritmo, podemos concentrarnos en la cantidad de lecturas que se hacen a P. Lo haremos por contabilidad de costos. Los elementos que empiezan un ciclo (es decir, tienen menor índice en P) se leen dos veces, la primera al comenzar una iteración de i, y la segunda al cerrar el ciclo. Los elementos de un ciclo que no son el primero también se leen dos veces, la primera al ser descubiertos cuando se procesa su ciclo, y la segunda cuando el iterador i llega a su posición, para darse cuenta de que ya están marcados. Es decir, cada elemento paga dos lecturas, lo que significa que en total hay $O(n)$ lecturas.

Problema 2. (★★★★) Si bien mantener un arreglo ordenado de n elementos permite buscar elementos en $O(\log n)$ a través de búsqueda binaria, insertar nuevos elementos tarda $O(n)$. Diseñe una estructura que permita mantener n elementos de forma de buscar en tiempo $O(\log^2 n)$ e insertar en tiempo $O(\log n)$ amortizado. ¿Cómo podría soportarse una operación de borrado de elementos?

Solución 2 . Mantendremos $k = \lceil \log(n+1) \rceil$ (el número de bits de n) arreglos ordenados. Tenemos arreglos $A_0, \ldots, A_{k-1}$ donde el A_i tiene tamaño 2^i. Escribiendo n en binario como $b_{k-1}b_{k-2}\cdots b_1 b_0$, mantendremos siempre A_i vacío si $b_i = 0$ o lleno si $b_i = 1$. La búsqueda es fácil, por cada uno de los $O(\log n)$ arreglos hacemos búsqueda binaria, lo que da una complejidad de $O(\log^2 n)$. Para las inserciones, notemos que n está aumentando en 1, lo que cambia algunos bits finales b_i de 1 a 0 y exactamente un bit de 0 a 1. Se debe hacer merge de todos los que corresponden a los bits que se han cambiado de 1 a 0, e insertar además el nuevo elemento. Todos esos elementos quedarán en el arreglo asociado al bit que cambia de 0 a 1. Si el único bit que cambió de 0 a 1 es el i-ésimo, entonces eso cuesta $\left(\sum_{j=0}^{i-1} 2^j\right) + 1 = 2^i$. Pero recordemos del ejercicio del contador binario visto en clases, que el i-ésimo bit cambia 1 de cada 2^i inserciones. Es decir, si consideramos el costo asociado únicamente al i-ésimo bit, es 2^i cada 2^i inserciones, y por tanto es constante al amortizarlo por inserción. Dado que hay $O(\log n)$ bits, el costo amortizado total es $O(\log n)$. Otra forma de analizar este problema es cobrándole $\log n$ a cada elemento insertado y dejando gratis los merge. Notando que cada vez que se hacen los merge, los elementos involucrados avanzan a arreglos A con mayor índice, esto solo ocurre $O(\log n)$ veces, lo que se paga con el cobro inicial.

Se pueden soportar borrados en $O(n)$ peor caso. Primero se busca dónde se encuentra el elemento a eliminar. Supongamos que está en A_o. Sea m el índice del bit menos significativo que vale 1. Lo que haremos será reducir el m-ésimo bit a 0 y repartir sus elementos entre los arreglos asociados a los bits enos significativos. Para esto, si $m \neq o$, intercambiamos el elemento a borrar con uno de A_m que iría en su posición. Luego, simplemente dividimos los elementos de A_m en arreglos $A_{m-1}, \ldots, A_0$. Dado que A_m ya se encontraba ordenado, no es necesario ordenar los arreglos en que se ha dividido. Esto cuesta $O(|A_m|) = O(n)$. Es importante notar que al soportar esta operación se pierde el costo amortizado de inserción. En efecto, una secuencia de $n = 2^\ell$ inserciones deja lleno el ℓ-ésimo arreglo y vacíos los demás. Luego, se hace un borrado, que cuesta $\Omega(n)$ y deja llenos todos los arreglos excepto el más grande. Luego una inserción los vuelve a mezclar, costando nuevamente $\Omega(n)$, y así sucesivamente por n operaciones más. Tenemos una secuencia de $2n$ operaciones, donde n de ellas cuestan $\Omega(n)$ y por tanto el costo total es $\Omega(n^2)$, lo que dice que el costo amortizado ha pasado a ser $\Omega(n)$.

Problema 3. (★★★) Esta pregunta busca analizar una variante de la estructura Union-Find, donde partimos con n elementos, cada uno de los cuáles define una clase separada, y realizaremos $m > n$ operaciones Find.

1. ¿Cuántas operaciones Union puede haber?
2. Modifique el análisis realizado en clases para demostrar que el costo total es $O(n \log n + m)$.
3. Demuestre que si se hacen todas las operaciones Union antes del primer Find, entonces el costo total es $O(m)$.

Solución 3 . Inicialmente hay n clases de equivalencia, y cada Union disminuye el número de clases en 1, por lo que a lo más pueden haber $n - 1$ operaciones de Union. Cada operación Union cuesta $O(1)$ como se ha visto en clases. Para analizar el costo de Find podemos utilizar contabilidad de costos. Le cobraremos una parte a los nodos, y una parte a la operación misma. Le cobraremos a cada nodo cuando un Find que pasa por él, excepto a la raíz y sus hijos directos, cuyo costo lo pagará la operación. Cada vez que le cobramos a un nodo este cambia su raíz, ya que cuando un nodo mantiene su raíz, después de que Find pasa por él este es colgado directamente de su raíz, y a los hijos de la raíz no se les cobra. Cada vez que a un nodo se le cambia la raíz, su árbol se está al menos duplicando, lo que solo puede ocurrir a lo más $O(\log n)$ veces. Es decir, cobramos a lo más $O(\log n)$ por nodo, lo que da $O(n \log n)$ en total. Esto sin contar a la raíz ni sus hijos, a quienes podemos llamar nodos *especiales*. Dado que cada operación Find va subiendo dentro de un árbol, cada operación Find pasa por a lo más 2 nodos especiales, un hijo directo de la raíz y la raíz, y por tanto el costo asociado a estos es constante. Es decir, las m operaciones Find suman $O(m)$ al costo, resultando en $O(n \log n + m)$ como pedido.

Para la segunda parte, notemos que tras todos los Union, las raíces ya no cambian, por lo que los nodos pagan solamente su primer Find, ya que luego quedarán colgados directamente de la raíz. Dado que los nodos pagan solo su primer Find, y hay m operaciones Find, el costo de los nodos es $O(m)$, sumado al de las operaciones (que pagan por la raíz e hijos directos), sigue siendo $O(m)$.