Auxiliar 6 - Análisis Amortizado

CC4102 - Diseño y Análisis de Algoritmos
Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

26 de Octubre del 2015

1. Suponga que se le entrega una implementación de un *stack*, en la cual las operaciones PUSH y POP toman tiempo constante (¿cómo implementaría esto?). Utilice estructuras de este tipo para implementar una cola (*queue*) para la cual las operaciones ENQUEUE y DEQUEUE tienen un costo amortizado constante.

2. Un *quack* es una mezcla entre queues y stacks. Puede ser visto como una lista de elementos, escrita de derecha a izquierda, que soporta las siguientes operaciones:

   - QUACKPUSH(x) agrega x en el extremo izquierdo.
   - QUACKPOP(x) remueve y retorna el elemento más a la izquierda.
   - QUACKPULL(x) remueve y retorna el elemento más a la derecha.

   Implemente un quack usando tres stacks (idénticos a los del problema anterior) de modo que cada operación tenga un costo amortizado constante. Puede utilizar $O(1)$ memoria. Almacene los elementos sólo una vez en cualquiera de los 3 stacks.

3. Considere una tabla de tamaño s que almacena n elementos, que se va llenando con inserciones y debemos realocarla cuando no queda espacio para insertar ($s = n$). Esta vez, también ocurren borrados y no queremos que s sea mucho mayor que n, para no desperdiciar espacio. Al realocar la tabla para agrandarla o reducirla debemos pagar un costo de $O(n)$.

   (a) Muestre que duplicar la tabla cuando se llena, y reducirla cuando $n = s/2$ no consigue un costo amortizado constante.

   (b) Considere la estrategia de duplicar cuando la tabla se llena, y reducirla a $s/2$ cuando $n = s/4$. Demuestre que esta estrategia obtiene un costo amortizado constante utilizando la siguiente función potencial:

   $$\Phi = \begin{cases} 2 \cdot n - s & \text{si } n \geq s/2 \\ s/2 - n & \text{si } n < s/2 \end{cases}$$

   (c) Analice el caso general en que queremos obtener un factor de carga α, con $0 < \alpha < 1/2$.

4. Suponga que tenemos un contador de modo que el costo de incrementar el contador es igual al número de dígitos (en el caso binario, bits) que deben ser modificados. Vimos en clases que si el contador comienza en 0, el costo amortizado de un incremento es $O(1)$. En este problema buscamos implementar un contador que permite *incrementos* y *decrementos*. Sólo consideraremos secuencias de operaciones que mantienen el contador en un valor no negativo.

   (a) Muestre que incluso en secuencias que mantienen el contador con un valor no negativo, es posible que una secuencia de n operaciones que comience con el contador en 0 y que permita incrementos y decrementos tenga un costo amortizado de $\Omega(\log n)$ por operación (equivalentemente, un costo total de $\Omega(n \log n)$).

(b) Para arreglar este problema, considere el siguiente *sistema de números ternario redundante*. Un número se representa como una secuencia de *trits*, que, como su nombre lo indica, pueden tomar tres posibles valores: 0, +1 o −1. El valor de un número $t_{k-1}, ..., t_0$ (donde cada t_i es un trit) se define como

$$\sum_{i=0}^{k-1} t_i 2^i$$

El proceso de incrementar un número de este tipo es análogo al de la operación en números binarios. Se añade 1 al trit menos significativo; si el resultado es 2, se cambia el trit a 0 y se propaga una *reserva* hacia el siguiente trit. Se repite el proceso hasta que no exista carry.

El proceso de decrementar es similar: se resta 1 al trit menos significativo; si el resultado es −2, se reemplaza por 0 y se "pide" al siguiente trit.

Mediremos el costo de un incremento o decremento como el número de trits que se ve alterado. Comenzando de 0, se realiza una secuencia de n incrementos y decrementos (sin un orden particular).

Demuestre que, utilizando esta representación, el costo amortizado por operación es $O(1)$ (equivalentemente, que el costo total para las n operaciones es $O(n)$. Hint: Puede utilizar el método de contabilidad o de función de potencial para llegar al resultado.

Antes de Empezar...

El análisis amortizado nos sirve cuando ciertas operaciones son mucho más costosas que otras. Esto significa que analizar el peor caso (como hacíamos anteriormente) es bastante injusto con el algoritmo. Queremos una especie de costo promedio de una operación; sin embargo, no es el costo promedio usual: analizaremos *secuencias* de operaciones.

Recordemos que tenemos (al menos) tres formas de abordar el análisis amortizado:

- Hacer un **análisis global**, sumando todos los costos de alguna forma.
- Hacer **contabilidad** de costos, moviendo el costo de ciertas operaciones a otras. En otras palabras, reorganizamos los costos que queremos contar, de modo de que el conteo sea más sencillo. Una forma típica es asignar un costo adicional a operaciones pequeñas, que luego es cobrado al momento de realizar una operación costosa.
- Utilizar una **función potencial** ϕ. Luego estimamos el costo como el costo verdadero más las variaciones de potencial. Si la función de potencial está bien diseñada, típicamente cancelará el costo de las operaciones costosas. Es necesario, sí, que la función potencial que se utilice nunca tome valores menores al inicial (pues queremos siempre sobreestimar el costo del algoritmo a analizar).

Con esto en mente, resolvamos los problemas.

Soluciones

1. Usaremos dos stacks para simular la cola. El primer stack, In, será la *entrada* de la cola, y el otro, Out, será la *salida*. De esta forma, los algoritmos funcionan así:

   - Enqueue(x): Hacemos In.Push(x).
   - Dequeue(x): Si Out está vacío, repetimos Out.Push(In.Pop()) hasta que In esté vacío. Retornamos Out.Pop().

   Notemos, primero, que tenemos que mostrar que la estructura se comporta como una cola. Como In y Out son stacks, si x entra a la estructura antes que y, saldrá *después* que y de In, y por lo tanto saldrá *antes* de Out. Por lo tanto, la estructura se comporta como una cola.

   Ahora es necesario ver que el costo amortizado de las operaciones es constante. Notemos que Dequeue es la operación "problemática", pues el caso en que Out está vacío puede resultar en muchas operaciones. La gracia está en notar que si bien pueden moverse muchos elementos de In a Out, cada uno de esos elementos *entró de a uno* a la estructura. Utilizamos entonces una estrategia de contabilidad.

   Cada vez que un elemento entra a la estructura (a través de un Enqueue) le asignamos una ficha. De esta forma, Enqueue tendrá un costo amortizado $O(1)$ (es necesario sumar el costo de la ficha, pero $O(1) + 1 = O(1)$).

   Los elementos utilizan esta ficha para pagar el costo de ser movidos a Out, por lo que la operación de Dequeue sólo cuesta hacer el Pop de Out, que es $O(1)$. Es importante que los elementos sólo son movidos a Out una vez en su "vida": si no fuera así, no nos alcanzarían las fichas para pagar.

   También se puede hacer el análisis utilizando $\phi = 2 \times |\text{In}|$ como función potencial y considerando que se parte de una estructura vacía. Se los dejo propuesto, debería ser fácil :)

2. La forma de construir esta estructura es muy parecida a la parte anterior. En adición a los stacks In y Out, tendremos un stack Temp. Mantendremos la cuenta de cuántos elementos tiene cada stack, lo que usa $O(1)$ de memoria adicional. Por simplicidad, consideraremos que las operaciones sobre los stacks cuestan exactamente 1.

   Las operaciones son similares al caso anterior: QuackPull es básicamente un Dequeue, mientras que QuackPop es hacer un Pop de In. Es necesario, sí, definir lo que se hace si alguno de los stacks está vacío al momento de realizar un Pop.

   Si se hace un QuackPull de un Out vacío, se mueve la mitad de In a Temp haciendo operaciones de Push y Pop. Luego se mueve el resto de In a Out. Finalmente, se mueve el contenido de Temp a In. El caso de hacer un QuackPop cuando In está vacío se trata de forma simétrica. La idea de esto es mantener elementos tanto en In como en Out, ya que nos pueden pedir un QuackPull o un QuackPop en el futuro.

   Para el análisis utilizaremos una función potencial. A veces es útil pensar en la función potencial como "¿cuán indefensa está mi estructura ante las operaciones que puedan venir?". Nuestro quack se vuelve vulnerable (en el sentido de que pueden aparecer operaciones costosas) cuando In o Out están vacíos. Al contrario, si ambos stacks tienen un tamaño similar, estamos súper preparados para cualquier operación que nos lancen. Usamos entonces la siguiente función potencial, que va con esta idea metida:

$$\phi = c||\text{In}| - |\text{Out}||$$

   con c una constante que determinaremos en el camino.

   Analicemos las operaciones con esta función potencial. Para esto, hay que agregar el $\Delta\phi$ al costo de cada operación.

   - Un QuackPush cuesta $1 + \Delta\phi = 1 \pm c = O(1)$ (pues $|\text{In}|$ crece en 1 y $|\text{Out}|$ queda igual).
   - Un QuackPop que se da cuando In no está vacío cuesta $1 + \Delta\phi = 1 \pm c = O(1)$, pues $|\text{In}|$ disminuye en 1 y $|\text{Out}|$ permanece igual. El mismo análisis se da en el caso de un QuackPull con un Out no vacío.
   - Si se hace un QuackPull cuando Out está vacío, el costo original es $1 + 3|\text{In}|$ debido a los traspasos de elementos entre In, Out y Temp (no es difícil de verificar).

     Notemos que partimos con Out vacío y terminamos con la misma cantidad de elementos en In y Out. En otras palabras, $\Delta\phi = -c|\text{In}|$. El costo amortizado, entonces, es $1 + 3|\text{In}| - c|\text{In}|$. En este punto somos astutos y elegimos $c = 3$, con lo que el costo amortizado resulta ser constante. El análisis de un QuackPop con un In vacío es análogo, y el mismo valor de c sirve.

   Así, cumplimos con todo lo que se nos pedía :)

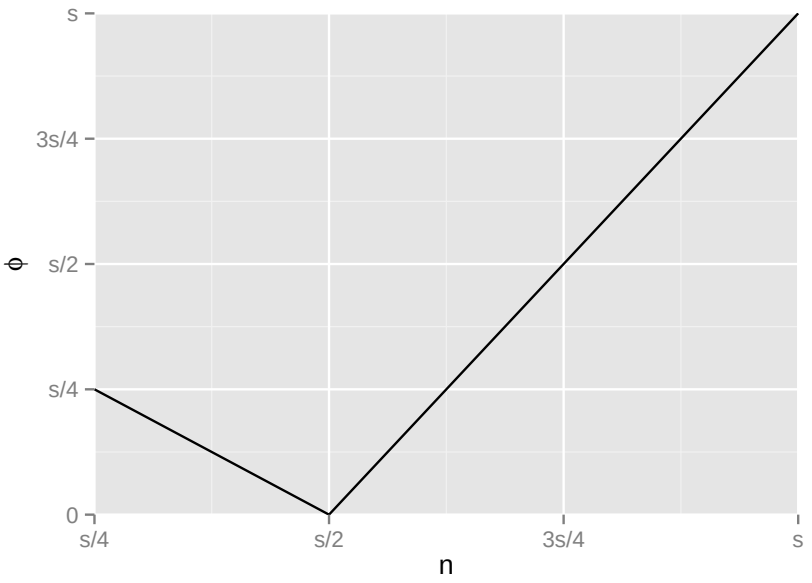
3. (a) Duplicar la tabla cuando se llena y reducirla cuando llega a la mitad no es una buena estrategia. Consideremos una tabla que se encuentra llena con n elementos. La inserción de un nuevo elemento fuerza a duplicar la tabla, con un costo de n. Si a continuación se remueve un elemento, se fuerza a reducir la tabla, nuevamente con un costo de n. Podemos repetir estas dos operaciones de forma alternada, con el mismo costo cada vez. En otras palabras, encontramos una secuencia de inserciones y borrados que tiene un costo de $O(n)$ por operación, lo que es bastante malo.

   (b) Nos entregan la función potencial a utilizar. Sólo tenemos que calcular el costo de cada operación usándola, agregando los cambios en la función potencial. Consideremos por simplicidad que el costo de duplicar y el de reducir la tabla es n.

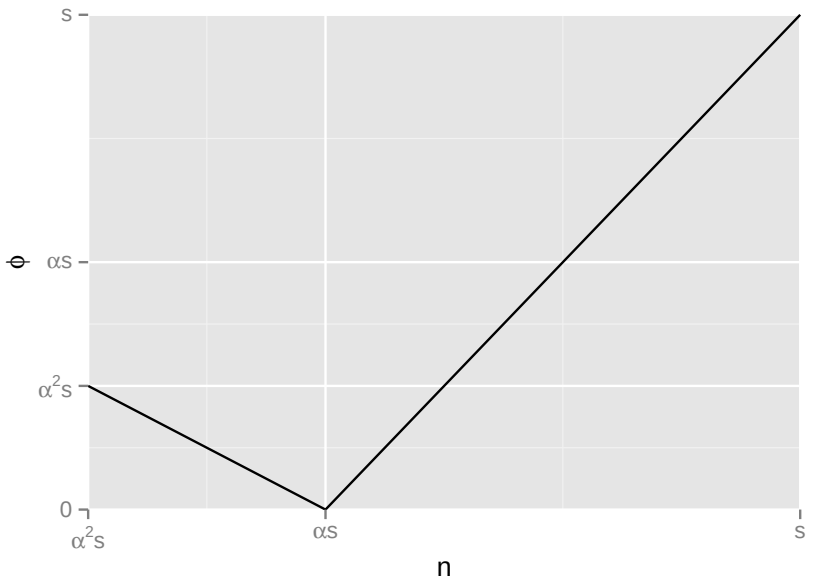
- Una inserción que no hace crecer la tabla tiene costo 1. El potencial puede cambiar en $+2$ o en -1, dependiendo de en qué tramo de la función potencial estemos, por lo que el costo amortizado puede ser 0 o 3, siendo constante en ambos casos.
- Un borrado que no hace crecer la tabla tiene costo 1. El potencial puede cambiar en -2 o en $+1$. Nuevamente, el costo amortizado resulta constante en ambos casos.
- Una inserción que duplica la tabla tiene costo $n + 1$. El potencial pasa de n a 0, por lo que $\Delta\phi = -n$. Así, el costo amortizado es sólo 1: el de insertar el nuevo elemento.
- Un borrado que reduce la tabla nuevamente tiene costo $n + 1$. El potencial pasa de $s/2 - n = 2n - n = n$ a 0, por lo que $\Delta\phi = -n$. Así, el costo amortizado es sólo 1: el de insertar el nuevo elemento.

Luego hemos demostrado que las operaciones toman un costo amortizado constante.

(c) En el caso anterior las operaciones de duplicar y reducir la tabla siempre nos dejaban la tabla con $s/2$ elementos. Ahora queremos obtener un factor de carga α. Nos gustaría que la función potencial pagara por todo, como en el caso anterior. Dibujemos cómo se veía la función potencial antes, para ver cómo la modificaremos.



Lo que queremos ahora es que el llenado de la tabla esté en torno a α en vez de $1/2$. En otras palabras, cuando la tabla se llena, la agrandamos en un factor $1/\alpha$ (de modo de que quede llena en una fracción α), lo que nos cuesta $n = s$. Cuando la cantidad de elementos en la tabla disminuya a un nivel $\alpha^2 s$, la achicamos en un factor α, lo que nos cuesta $n = \alpha^2 s$. Entonces, nos gustaría una función potencial que tome la siguiente forma:



Esta función es la siguiente (es cosa de hacer un poco de ecuación de la recta):

$$\Phi = \begin{cases} n/\alpha - s & \text{si } n \geq \alpha s \\ \frac{-1}{\alpha(1-\alpha)}(n - \alpha s) & \text{si } n < \alpha s \end{cases}$$

Con esto, queda hacer el análisis: en particular, es necesario mostrar los valores que $\Delta\phi$ toma al agrandar y achicar la tabla.

Al agrandar la tabla, $\Delta\phi = 0 - s = -n$, pues $n = s$ en este caso (la tabla está llena). Luego una inserción que agranda la tabla tiene un costo amortizado de 1 (el cambio de potencial cancela el costo de mover los elementos a la nueva tabla).

Al achicar la tabla, $\Delta\phi = 0 - \alpha^2 s = -n$ pues $n = \alpha^2 s$ en este caso. Luego un borrado que achica la tabla tiene un costo amortizado de 1 (el cambio de potencial cancela el costo de mover los elementos a la nueva tabla).

Adicionalmente, las operaciones que no causan modificaciones en el tamaño de la tabla a lo más sufren un costo adicional constante (de $1/\alpha$ o de $1/\alpha(1-\alpha)$). En conclusión, todas las operaciones tienen un costo amortizado constante.

4. (a) Sea $2^{b+1} \leq n \leq 2^{b+2}$. Usamos 2^b operaciones para llevar el contador a un 1 seguido de $b-1$ ceros. Esto cuesta al menos 2^b. Luego alternamos decrementos e incrementos del contador. Cada una de estas operaciones cuesta $b-1$, pues hay que flippear los últimos $b-1$ bits del contador. Luego el costo total de estas operaciones es al menos $b \cdot 2^b$; el costo amortizado por operación será al menos $b \cdot 2^b/n \geq b/4 = \Omega(\log n)$.

   (b) Usaremos el método de contabilidad. Notemos que cuando el contador se incrementa o decrementa, los 1s cambian a 0 o siguen en 1, y los -1 cambian a 0 o siguen igual. Luego podemos pagar 2 cada vez que un trit se convierte desde un 0 a otro valor. Con esto, cada trit tendrá suficiente para pagar cuando se convierta de nuevo en un 0.

   Para acotar el costo amortizado, entonces, sólo nos faltaría demostrar que cada incremento no cambia demasiados trits que tengan 0's a otros valores (de lo contrario, tendríamos que pagar demasiadas monedas adicionales).

   De hecho, lo que sucede es que cada incremento o decremento convierte **a lo más un trit que estaba 0 en otro valor**. ¿Por qué se cumple esto? Notemos que sólo traspasamos un carry cuando un 1 se convierte en un 2 o un -1 se convierte en un -2. Esto significa que cuando cambiamos un 0 a un 1 o a un -1, la operación de incremento

o decremento *debe* terminar. Luego al incrementar o decrementar pagamos 2 por el cambio del trit 0 a otro valor, que ocurre (a lo más) sólo una vez; los demás cambios en los trits son pagados por las monedas antes ahorradas.

Una forma de hacer el análisis usando una función de potencial es tomando $\phi =$ (número de trits en -1 o 1). Se los dejo propuesto.