Ejercicio de evaluación contínua: tiempo amortizado

José Antonio Álvarez Ocete

4 de octubre de 2017

1. Enunciado

La pregunta planteada viene de la mano de la eficiencia del método *quitar()* en la implementación de la pila vista en clase:

```
void Pila::quitar()
{
  assert (nelem > 0);
  nelem--;
  if (nelem<reservados/4) resize(reservados/2);
}
```

Sabemos que el método *resize(n)* tiene orden de eficiencia *O(n)*. Sin embargo, *quitar()* es de orden *O(1)*. ¿Por qué? Además, sustituyendo el 4 por un 2 pasamos a tener *O(n)*. ¿Por qué?

2. Solución

Respondamos primero a la primera pregunta. Intuitivamente es sencillo: la llamada al método ocurre muy pocas veces. Veámoslo con un caso análogo pero más sencillo de entender, la inserción en el vector dinámico la cual es exactamente la misma que la del método *poner()* en la Pila. La implementación es la siguiente:

```
// Asumimos que el vector ha sido creado con capacidad = 1
void vector_dinamico::insert(int value)
{
  if (nelem==capacidad) resize(capacidad*2); //Duplicamos la capacidad si está
      lleno
  vector[nelem] = value;
  nelem++;
}
```

¿Cuál sería la eficiencia al insertar n elementos? Asumamos que n es potencia de 2 para hacer los cálculos más sencillos. Llamaríamos al *resize()* cada vez que el número de elementos alcanza una potencia de 2, y duplicaría la capacidad. Es decir copiaremos en total los siguientes elemenos:

$$1 + 2 + 4 + 8 + ... + n/2 + n$$

Esta suma tiende a *2n*. Por lo tanto nos tomaría aproximadamente *2n* unidades de tiempo insertar n elementos en el vector, obteniendo una eficiencia de *O(1)* al insertar un elemento.

Lo mismo ocurre en el caso de la pila. Si queremos retirar todos elementos de la pila (n), llamaríamos a *resize()* para los elementos:

$$n/4 + n/16 + n/64 + ... + 1 = \sum_{i=1}^{p} \frac{n}{4^i}, j \geq 1 cumpliendo : 4^j = n$$

Estudiando esta progresión geométrica obtenemos (la razón es menor que 1):

$$\sum_{i=1}^{p} \frac{n}{4^i} = n * \sum_{i=1}^{p} (\frac{1}{4})^i = n * \frac{1/4}{1-1/4} = \frac{n}{3}$$

Haciendo un cálculo análogo al del ejemplo anterior, quitamos n elementos de la pila en un tiempo *O(n/3)* obteniendo una eficiencia de *O(1)* al quitar un solo elemento.

Nota: Hemos asumido que el *resize()* implementado en la clase *Pila* copia unicamente los elementos del vector que estamos usando. Es decir, hasta *nelems*.
¿Qué ocurre si modificamos el método de la siguiente forma?

```
void Pila::quitar()
{
  assert (nelem > 0);
  nelem--;
  if (nelem<reservados/2) resize(reservados/2);
}
```

Si aplicamos el mismo razonamiento pensaremos que la eficiencia sigue siendo de orden *O(1)*. Sin embargo, este no es el peor caso posible. El problema de esta implementación es que si justo después de reducir la pila a la mitad queremos introducir un nuevo elemento tenemos que volver a hacer *resize()* para ampliar la capacidad del vector. ¿Y si volvemos a quitar un elemento? ¿Y si, justo después, volvemos a introducir otro elemento? Este es, de hecho, el peor caso posible.

Repitiendo este proceso de forma indefinida (introducir y quitar elementos con la pila al límite de su capacidad) haríamos *resize()* en cada llamada a *quitar()* y cada vez que introducimos un elemento. Esto provocaría que la eficiencia fuese *O(n)* tanto para *quitar()* como para *poner()*.

Esto no ocurre en el caso anterior. Esto se debe a que dejamos un trozo de pila vacía cada vez que la reducimos (un cuarto para ser exactos). De esta forma podemos hacer tantos *poner()* entre dos *quitar()* como queramos: no provocaremos que la pila se reduzca con un solo *quitar()*.