

Tarea 2

Análisis de Algoritmos

Profesora: Andrea Rodríguez.

Alumnos: Diego Varas.

Jeremías Torres. Lunes 28, mayo 2018

Introducción

En este informe se pretende entregar solución a la tarea 2, la cual consiste en proponer una estrategia de manera que el costo amortizado de n inserciones en una cola de prioridad diseñada con un min o max heap sea O(n), ya que este costo de inserción es O(logn) y su costo amortizado es O(nlogn) sobre n operaciones.

Algoritmo de Solución

Para esta tarea se ha diseñado la estructura de datos PQHeap, la cual está construida mediante un min heap.

Definición de Variables y funciones:

* **int \*a;** vector de tipo entero donde se almacenará el min heap.
* **int count;** variable que sirve para contar los elementos que contiene el vector.
* **int index;** variable que indica de que posición se empieza a almacenar datos en el vector a.
* **int num;** variable que indica el tamaño inicial del vector.
* **bool empty();** función que indica si la cola está vacía o no.
* **int size();** función que indica el tamaño de la cola.
* **int top();** función que muestra el valor que se encuentra en la raiz del arbol.
* **void push(int);** función que permite insertar elementos a la cola.
* **void solución();** funcion que imprime todo el contenido de la cola.

La función en la cual enfocaremos este análisis es la función push.

1 void PQHeap::push(int n){

2 if(index >= num-1){

3 int tmp = num;

4 num = tmp\*2;

5 a = (int \*)realloc(a,num\*sizeof(int));

6 }

7 a[index]=n;

8

9 int tmp = index;

10 while(tmp > 1 && a[tmp] < a[tmp/2]){

11 swap(a[tmp], a[tmp/2]);

12 tmp =tmp/2;

13 }

14 index++;

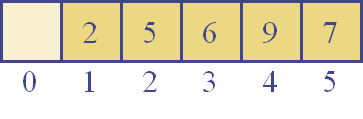
15 count++;

16 }

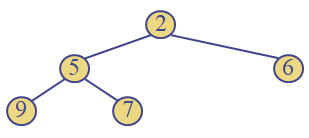
De la linea 2 a la linea 6 del codigo, sirve para en caso de que se inserten más de 10 elementos en la cola el vector a expanda su tamaño(num) el doble, para asi tener mas espacio disponible para insertar elementos.

De la línea 10 a la línea 13 del código, sirve para poder insertar de manera correcta los elementos en la cola, esta parte nos permite que siempre que se inserte un elemento menor al último que está en la cola en la cola, se haga swap, de manera que el elemento suba por el árbol hasta encontrar su posición.

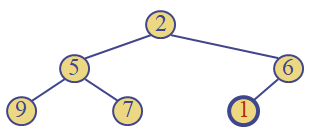
**Por ejemplo:**



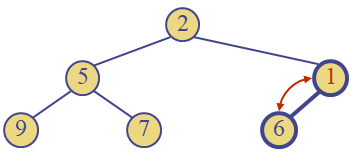
Se tiene el siguiente árbol el cual gráficamente es



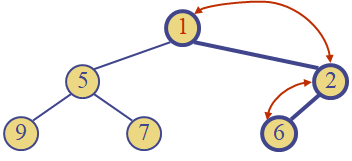
A este árbol en cuestión se le inserta un siguiente nodo que contiene el valor 1.



Como se puede apreciar el valor insertado no puede estar en esa posición, ya que nuestra estructura almacena el menor valor del árbol en su raíz, dado que es un min heap. Aquí es donde se debe hacer swap del elemento insertado con su padre.



Como se puede apreciar el valor insertado subió de nivel en el árbol, pero aun no llega a su posición correcta, ya que este valor es el mínimo valor entre todos los que están insertados en el árbol, por ende debe llegar a la raíz.



Nuevamente se realiza el swap con su padre, llegando así a la raíz del árbol, quedando ordenado en su totalidad como un min heap.

Análisis de Correctitud

Para demostrar que el algoritmo es correcto usaremos una invariante en el loop while. Esta dice que el valor del nodo insertado es menor que el valor de su nodo padre, es decir:

tmp > 1 && a[tmp] < a[tmp/2]

* Cuando a[1] (teniendo en cuenta que a[0] no es utilizado en nuestro algoritmo) significa que existe un solo elemento, por lo que se cumple la invariante cuando se inserta a[2].
* Luego cuando se inserte en a[3] seguirá cumpliendo la invariante, ya que el padre de ese nodo será a[1] (como tmp = 3, entonces tmp/2 = 1, el cual equivale al nodo padre en el árbol).
* Si tenemos en cuenta que el largo del arreglo es n, entonces según lo demostrado anteriormente para n+1 seguirá cumpliendo la invariante.

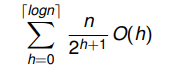
Análisis de Complejidad

Queremos crear nuestro min heap con n inserciones. Se considera el arreglo como un árbol binario, en el cual 1 es la raíz, 2 es el hijo izquierdo y 3 el hijo derecho y así sucesivamente.

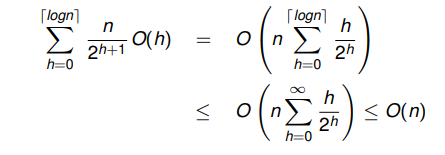
Considerando la altura de nuestro árbol, el cual será h, tendremos subárboles con altura h+1, los cuales serán convertidos en min heap. El costo de esto es O(h).

Con esto nuestro árbol de altura h tiene a lo más n/2^h+1 nodos.

Luego el costo total queda dado por:



Luego desarrollando llegamos a:

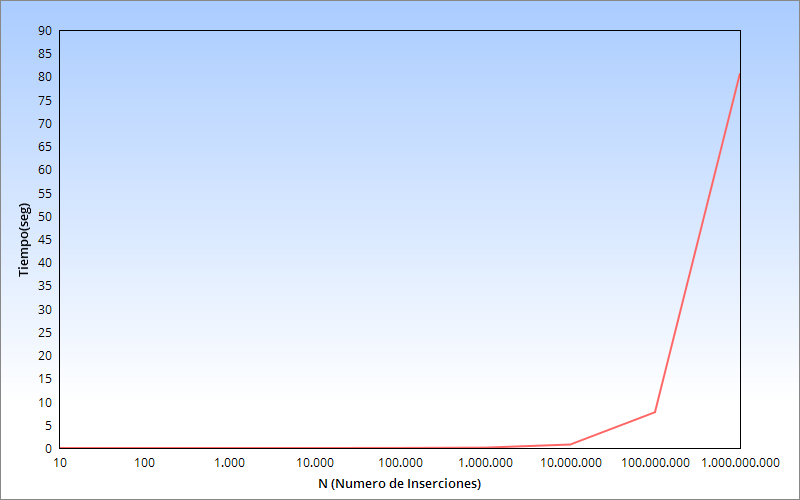


Por lo tanto, el costo total de crear nuestro min heap con n inserciones es O(n) y su costo amortizado queda dado por O(n)/n, luego el costo amortizado de la inserción es O(1). Si tenemos en cuenta que se realizan n inserciones entonces O(n\*1) = O(n).

Evaluación Experimental

Se hicieron pruebas al algoritmo con diferente valores de N (número de inserciones), los valores que tomó N van desde 10 hasta 1.000.000.000, los valores que se insertaron fueron generados de manera aleatoria. Las pruebas se realizaron en una laptop con procesador Intel Core i3-5005U CPU @ 2.00GHz y 8gb de memoria Ram.

|  |  |
| --- | --- |
| **N** | **Tiempo (segundos)** |
| 10 | 0.000000 |
| 100 | 0.000000 |
| 1.000 | 0.000000 |
| 10.000 | 0.000000 |
| 100.000 | 0.015625 |
| 1.000.000 | 0.078125 |
| 10.000.000 | 0.750000 |
| 100.000.000 | 7.734380 |
| 1.000.000.000 | 80.79690 |



Dado que el costo de hacer una inserción en la mayoría de los casos es O(1), con el análisis experimental se ve reflejado que nuestro algoritmo se comporta de manera lineal para un número de inserciones grande (10.000.000 inserciones) lo cual es aceptable, también muestra que superando ese número aumenta de forma brusca (entre 10.000.000 y 1.000.000.000 inserciones).

Conclusión

De este trabajo se puede concluir que crear una estructura de datos en nuestro caso una cola de prioridad porque se utilizó como estructura la utilización de un min heap, lo cual nos permitia al hacer inserciones almacenar los valores de manera ordenada en la cola, lo cual resulta más beneficioso que solo llenar una cola sin orden alguno y a su vez tener que utilizar algún tipo de ordenamiento, lo que resultaría más costoso.