Laboratorio 3

Juan Diego Sique Martínez

Agosto 2018

1. Primer inciso

Al desordenar el heap lo que se procura es que siempre el elemento más grande quede en la posición raíz de nuestra estructura, de manera que al llegar al final del arreglo ya se encontrará ordenado. (Asumiendo que es un «max-heap»)

Mi algoritmo pretende principalmente utilizar las propiedades del heap como su raíz y su longitud para así ir ordenando sin la necesidad de intercambiar elementos fuera de la función «Max-Heapify».

Lo primero es que en vez de utilizar el atributo *heap.size* que lo que hace es indicar hasta donde se debe de hacer heapify, usaré el atributo de mi invención llamado *heap.head* que pretende indicar a partir de dónde se hace «heap», actualizando los índices, es decir que si el *heap.head* posee un valor de 1 el índice del array donde se comienza a aplicar «heapify» es 1, pero si posee el valor 5, el elemento con índice número 5 en el arreglo será considerado como el primero de la lista. En pocas palabras *heap.head* nos muestra cual será considerado el primer elemento.

Data: Un arreglo **A**
Result: Un arreglo ordenado utilizando las propiedades de un «heap»
initialization;
A.heap.head = 0;
BUILD-MAX-HEAP(A);
for *i = 2 to A.lenght* **do**
    A.heap.head = A.heap.head + 1 ;
    MAX-HEAPIFY(A, i);
end

Algorithm 1: Heap-Sort

El proceso «Max-Heapify», no sufrió modificación alguna, utilizaré el modelo iterativo del laboratorio anterior. A continuación el algoritmo:

Data: Un arreglo **A**, un entero **i** que representa el índice del arreglo
Result: Una corrección de las ramas del arreglo donde se arregla para que cumpla con la propiedad de un «heap»
initialization;
while *i > -1* **do**
  l ← LEFT(i);
  r ← RIGHT(i);
  if *l ≤ heap.size(A) and A[l] > A[i]* **then**
    largest ← l;
  else
    largest ← i;
  end
  if *r ≤ heap.size(A) and A[r] > A[largest]* **then**
    largest ← r;
  end
  if *largest ≠ i* **then**
    exchange A[largest] ↔ A[i];
    i ← largest;
  else
    i ← -1
  end
end

Algorithm 2: Max-Heapify

Los algoritmos «Left» y «Right» también sufren modificaciones, precisamente por la manera de enfocar el *heap.head*:

Data: Un índice **i**
Result: La posición del hijo izquierdo del elemento
initialization;
return *2 × i - A.heap.head*

Algorithm 3: Left

Data: Un índice **i**
Result: La posición del hijo derecho del elemento
initialization;
return *2 × i + 1 - A.heap.head*

Algorithm 4: Right

Finalmente, luego de efectuar las correcciones para no arruinar el arreglo intercambiando el primer elemento por el último como lo sugiere el libro. TRas ver que igual se sigue recorriendo todos los elementos sin intercambiar el «running-time» no se ve afectado, por lo tanto conserva su $O(n \times log(n))$.

2. Segundo inciso

2.1. ¿Cuál es el «running-time» de «Quicksort» cuando todos los valores son iguales?

Al analizar el algoritmo «Quicksort» que nos proporciona el libro es notable ver que ingresa al método «Partition», donde retorna un valor idéntico a la longitud del arreglo. Luego de aplicar la recursividad es posible verificar el el algoritmo «Quicksort» se ejecutó **3** veces, «Partition» se llevo a cabo sólo una vez y el número de intercambios realizados fue n. Por lo tanto se concluye que el «running-time» del algoritmo en el caso de que todos los elementos sean iguales es $O(n)$.

2.2. Ilustrando el funcionamiento de «Partition»

Para demostrar cómo funciona el algoritmo «Partition» se usará el siguiente arreglo: [13,19,9,5,12,8,7,4,21,2,6,11].

Aquí se ve el arreglo en la primera iteración del ciclo «for».

i	p, j										
13	19	9	5	12	8	7	4	21	2	6	11

Tras la segunda interación.

i	p	j									
13	19	9	5	12	8	7	4	21	2	6	11

Tras la tercera iteración.

p, i		j									
13	19	9	5	12	8	7	4	21	2	6	11

Luego del intercambio.

p, i		j									
9	19	13	5	12	8	7	4	21	2	6	11

Tras la cuarta iteración.

p, i			j								
9	19	13	5	12	8	7	4	21	2	6	11

Luego de aplicar el intercambio.

p	i		j								
9	5	13	19	12	8	7	4	21	2	6	11

Tras aplicar la quinta iteración.

p	i			j							
9	5	13	19	12	8	7	4	21	2	6	11

Luego de la sexta iteración.

p	i				j						
9	5	13	19	12	8	7	4	21	2	6	11

Luego de aplicar el intercambio en la sexta iteración.

p		i			j						
9	5	8	19	12	13	7	4	21	2	6	11

Tras la séptima iteración.

p			i			j					
9	5	8	19	12	13	7	4	21	2	6	11

Luego de aplicar el intercambio.

p			i			j					
9	5	8	7	12	13	19	4	21	2	6	11

Tras el octavo proceso.

p				i			j				
9	5	8	7	12	13	19	4	21	2	6	11

Luego de su intercambio.

p				i			j				
9	5	8	7	4	13	19	12	21	2	6	11

Tras la novena iteración.

p				i				j			
9	5	8	7	4	13	19	12	21	2	6	11

Se procede a la décima iteración.

p					i				j		
9	5	8	7	4	13	19	12	21	2	6	11

Luego de realizar el intercambio respectivo.

p					i				j		
9	5	8	7	4	2	19	12	21	13	6	11

Ahora toca la décimoprimera iteración.

p						i				j	
9	5	8	7	4	2	19	12	21	13	6	11

Tras el intercambio propio.

p						i				j	
9	5	8	7	4	2	6	12	21	13	19	11

Ha concluido el ciclo «for». Se procede a realizar realizar el útltimo intercambio y el valor de retorno es **8**.

p						i				j	
9	5	8	7	4	2	6	11	21	13	19	12

2.3. ¿Por qué es más utilizado «Quicksort» y no «Heapsort» sabiendo que el «worst-case running time» de «Heapsort» es mejor?

Al analizar el algoritmo «Quicksort» que nos proporciona el libro es notable ver que ingresa al método «Partition», donde retorna un valor idéntico a la longitud del arreglo. Luego de aplicar la recursividad es posible verificar el el algoritmo «Quicksort» se ejecutó **3** veces, «Partition» se llevo a cabo sólo una vez y el número de intercambios realizados fue n. Por lo tanto se concluye que el «running-time» del algoritmo en el caso de que todos los elementos sean iguales

es $O(n)$. Cosa que no es así en «Heapsort» puesto que no identifica con rapidez si el algoritmo ya se encuentra ordenado o todos los elementos son iguales, es decir mantiene su complejidad $O(n \times log(n))$. En conclusión, se le prefiere por su crecimiento asintótico menor en el mejor de los casos.

3. Tercer inciso

Se encuentra en un «Jupyter Notebook» adjunto.