Universidad Nacional de Ingeniería
Ciencias de la Computación
Heapsort

Yuri Nuñez Medrano *
ynunezm@gmail.com

Resumen

Se analizará el Algoritmo Heapsort.

1. Analisis del Heapsort

El Heapsort es parecido al merge sort, pero a diferencia del insertion sort el heapsort tiene un tiempo de ejecución de $O(nlgn)$. Parecida a Insertion sort, pero a diferencia de merge sort, el heapsort en su lugar sólo tiene un número constante de elementos de un array.

Algorithm 1: MAX_HEAPIFY(A,i)

```
1  l=LEFT(i)
2  r=RIGHT(i)
3  if l ≤ A.heapsize_ size and A[l]>A[i] then
4  |  largest=l
5  else largest=i;
6  if r ≤ A.heapsize_ size and A[r]>A[largest] then
7  |  largest=r
8  if largest ≠ i then
9  |  exchange A[i] with A[largest]
10 |  MAX_HEAPIFY(A,largest)
```

Algorithm 2: BUILD_MAX _HEAP(A)

```
1  A.heap_size=A.lenght
2  for i = ⌊A.length/2⌋ to 1 do
3  |  MAX_HEAPIFY(A,i)
```

El la estructura de datos heap es un objeto array que puede ser visto cercanamente como un arbol binario. Cada nodo del arbol corresponde a un elemento del array. El arbol esta completamente llenado con excepciones en el ultimo nivel, como en la figura 1.

*Escuela de Ciencias de la Computación, 27-08-15

Algorithm 3: HEAPSORT(A)

```
1  BUILD_MAX_HEAP(A)
2  for i = ⌊A.length⌋ to 2 do
3  |  exchange A[1] with A[i]
4  |  A.heap_size=A.heap_size-1
5  |  MAX_HEAPIFY(A,1)
```

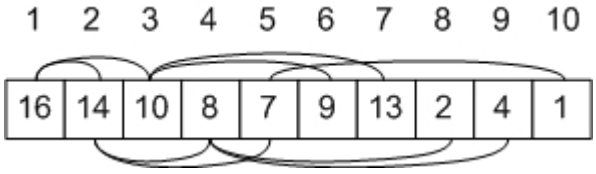

Figura 1: Array

Los indices del arreglo contienen los valores de los nodos, como en la figura 2.
Tenemos el array A, en el el que A.length: es el numero de elementos del array.
A.heap_size: es el tamaño de elementos evaluados.
El heapsize ≤ length
Raiz: primer elemento A[1]
padre(i)=i/2
izquierdo(i)=2.i
derecho(i)=2.i+1
el indice de un nodo hijo es menor que el del padre.
La altura h es lgn como en la figura 3.

1.1. Analisis de Heapify

Segun el algoritmo 1 podemos observar que esta compuesto por resultados de variables y condicionales y una funcion recursiva dentro de una condicional, por lo tanto de la linea 1 a la 7 será constantes.
El algoritmo 1 depende de dos datos "A" que es el array completo, e "i", y dentro tambien depende de la variable

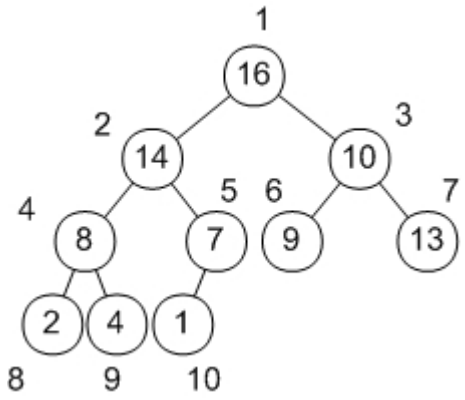

Figura 2: Arbol binario

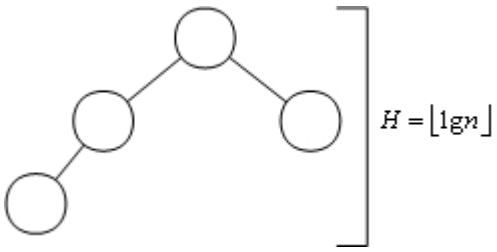

Figura 3: Altura

A.heapsize. La variable "i" es importante pues te indica las veces que tiene que evaluar tres nodos (papa, hijo izquierdo e hijo derecho), mientras menor sea esta realizara mas procesos, dependiendo de la altura del arbol. Si se da el caso que empieze por un $i = 1$, entonces este nodo evaluara el hijo izquierdo $i{,}2 = 2$, y el hijo derecho $i{,}2 + 1 = 3$ y continuara por el hijo que tenga mas valor en el indice, se repita el proceso hasta una altura h.
Entonce aproximadamente se procesara menos que 2 nodos de 3.
$T(n) \leq T(\frac{2n}{3}) + \Theta(1)$.
donde n es el número de nodos del árbol.
por el método maestro tenemos $O(lgn)$.

1.2. Analisis de BUILD_MAX_HEAP(A)

En el la linea 2 del algoritmo 2 podemos apreciar un bucle que se puede entender aproximadamente n veces la el tiempo de ejecución de Heapify lgn, lo que resultaria un costo total de $O(nlgn)$ sin embargo la cota es muy amplia, podemos aproximarla mas si evaluamos.Tomando en cuenta que los nro de nodos es un arbol al reves

nro nodos - altura.
por nivel

1	H
2	H-1
2^2	H-2
...	...
2^{H-1}	H-(H-1)=1
$\leq 2^H$	0

A medida que avanza la linea 2 el bucle empieza en n/2 disminuyendo hasta 1. lo que cuando menor sea el i el padre evaluara a mas nodos hijos. En un heap de en elementos, tiene una altura de$\lfloor lgn \rfloor$, y le numero de nodos en el monticulo de altura h es $\lceil n/2^{h+1} \rceil$. Como se hace un llamado a Heapify (A, i), para casi todo i su complejidad se puede medir como.

$\sum_{h=0}^{\lfloor lgn \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h})$
Evaluamos la última sumatoria por la formula $x = 1/2$ de:.
$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$.
para $|x| < 1$.
$\sum_{h=0}^{\infty} \lceil \frac{h}{2^h} \rceil = \frac{1/2}{(1-1/2)^2}$
$= 2$
Así podemos limitar la ejecución de BUILD_MAX_HE AP(A) como.
$O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h})$
$= O(n)$

1.3. Priority queues

El Heapsort es un buen algoritmo pero, el algoritmo quicksort es mejor en la practica, pero el tipo de estructura heap, tiene varios usos. Como un heaps, las colas de prioridad son dos: las colas de prioridad maxima y las colas de prioridad minima. Una cola de prioridad es una estructura que mantiene un conjunto de S de elementos, cado uno asociado con un valor llamado key. Una cola de maxima prioridad soporta las siguiente operaciones:
INSERT(S,x) unserta el elemento x en un conjunto S, es equivalene a $S = S \cup \{x\}$.
MAXIMUN(S) retorna el elemento S con el mayor Key.
EXTRACT _MAX(S) remueve y retorna el elemento de S con mayor Key.
INCREASE_KEY(S,x,k) incrementa el valor de la key del elemento x a un nuevo valor k,cual se supone que es al menos tan grande como el valor del key actual de x.

En los siguientes algoritmos tendremos implementado la cola de prioridad maxima, en el HEAP_MAXIMUN con un tiempo de ejección de $O(1)$ en el algoritmo

Algorithm 4: HEAP_MAXIMUN(A)

```
1 return A[1]
```

La extracción del máximo HEAP_EXTRACT_MAX(A) con un tiempo de ejecución de $O(lgn)$. Es similar a algoritmo 3 en las lineas del 3 al 5. del HEAPSORT

Algorithm 5: HEAP_EXTRACT_MAX(A)

```
1 if A.heap_size<1 then
2     error "heap underflow"
3 max = A[1]
4 A[1]=A[A.heap_size]
5 A.heap_size=A.heap_size-1
6 MAX_HEAPIFY(A,1)
7 return max
```

El algoritmo 6 HEAP_INCREASE_KEY(A,i,key) con un tiempo de ejecución de $O(lgn)$.

Algorithm 6: HEAP_INCREASE_KEY(A,i,key)

```
1 if key<A[i] then
2     error "new key is smaller than current key"
3 A[i]=key
4 while i>1 and A[PARENT(i)]<A[i] do
5     exchange A[i] with A[PARENT(i)]
6     i=PARENT(i)
```

El algoritmo 7 MAX_HEAP_INSERT(A,key) con un tiempo de ejecución de $O(lgn)$.

Algorithm 7: MAX_HEAP_INSERT(A,key)

```
1 A.heap_size =A.heap_size+1
2 A[A.heap_size]=−∞
3 HEAP_INCREASE_ KEY(A,A.heap_size,key)
```

Referencias

[H.Cormen et al., 2009] H.Cormen, T., Leiserson, C., and Riverson, R. L. (2009). *Algorithms*. The MIT Press.