Ordenamiento y estadísticas de orden

Agustín J. González

ELO 320: Estructura de Datos y Algoritmos

Ordenamiento y Estadísticas de Orden

Problema de ordenamiento:

Entrada: Una secuencia de n números $(a_1, a_2, ..., a_n)$

Salida: Una permutación $(a_1´, a_2´, a_3´,...,a_n´)$ de la entrada tal que $a_1´ \leq a_2´ \leq a_3´ \leq ... \leq a_n´$

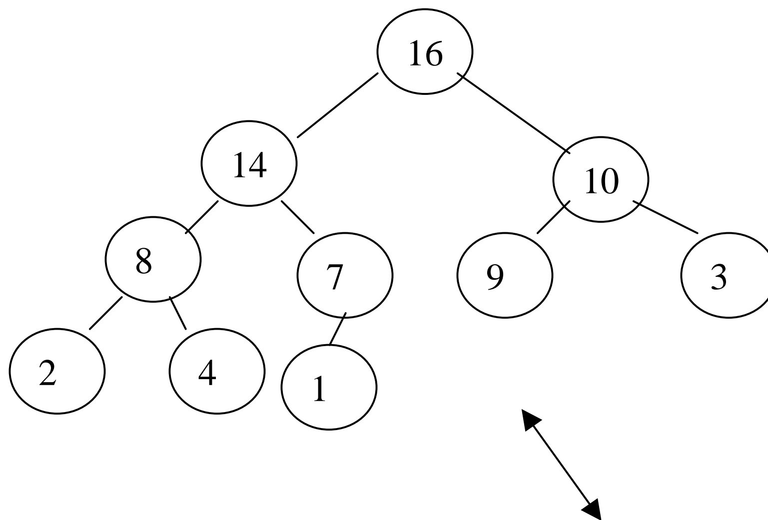
Soluciones: Insertion-Sort, merge-sort, heapsort, quicksort. ---> $\Omega(n \lg n)$

La estadística de orden i-ésimo de un conjunto de n números, es el i-ésimo número más pequeño.

---> $O(n)$

Heapsort

- La estructura de datos “heap” es un arreglo de objetos que pueden ser vistos como un árbol binario completo.
- Ej.



16	14	10	8	7	9	3	2	4	1

Propiedades del heap

- El arreglo puede contener más entragas (=length(A)) que el heap (= heap_size(A))
- Obviamente heap_size(A) ≤ length(A)
- La raíz del árbol es A[1] (Con índices partiendo de 1)
- Dado un índice i de un nodo, su padre, hijo izquierdo e hijo derecho son determinados como:

```
Parent(i)
    return ⌊ i/2 ⌋
Left(i)
    return 2i
Right(i)
    return 2i+1
```

Estos procedimientos se pueden implementar como macros o código "in-line".

Propiedad heap : A [Parent(i)]≥ A [i] para todo nodo diferente de la raiz.

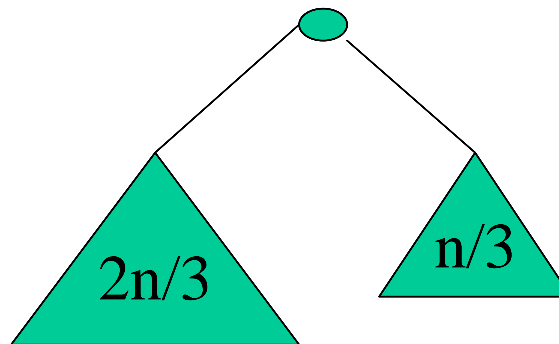
Procedimientos básicos usados en algoritmos de ordenamiento y en colas de prioridad

- Heapify(A,i): Entrada: Left(i) y Right(i) son heaps, pero A[i] puede ser menor que sus hijos.
- Salida: Heapify mueve A[i] hacia abajo para que al sub-árbol con raíz i sea un heap.

```
Heapify(A,i)
    le= Feft(i)
    ri= Right(i)
    if( le <= heap_size(A) && A[le] > A[i] )
        largest = le
    else
        largest = i
    if ( ri <= heap_size(A) && A[ri] > A[largest] )
       largest = ri
    if (largest != i)
       exchange A[i] <-> A[largest]
       Heapify(A, largest)
```

Análisis de tiempo de Ejecución

- $T(n) = \Theta(1)$ + Tiempo de Heapify sobre uno de los sub-árboles.
- El peor caso para el tamaño del sub-árbol es $2n/3$. Éste ocurre cuando la última fila está la mitad llena.



$$T(n) \leq T(2n/3) + \Theta(1)$$

$$\Longrightarrow T(n) = O(\lg n)$$

Construyendo el heap: Build_Heap

- La construcción del heap se logra aplicando la función heapify de manera de cubrir el arreglo desde abajo hacia arriba.
- Notar que los nodos hojas, ya son heap. Éstos están en A[⌊ (n/2+1) ⌋.. n].
- El procedimiento Build_Heap va a través de los nodos restantes y corre heapify en cada uno.

```
Build_Heap(A) {
        heap_size [A] = length [A]
        for i = ⌊length(A) /2 ⌋ downto 1
                do Heapify(A,i)
}
```

Ejemplo:

4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7

Análisis del tiempo de ejecución

- Cada llamado a Heapify tiene un costo O(lgn) y como a lo más hay n de estos llamados, una cota superior para el costo de Build_heap es O(nlgn).
- Un mejor análisis conduce a O(n)
- ¿Cuántos nodos hay de altura h? $n/2^{h+1}$
- Para nodos de altura h el costo de Heapify es O(h).
- Luego el costo es

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$\therefore$ Build_Heap puede ser acotado por :

$$\mathrm{O}\left(\mathrm{n} \sum_{\mathrm{h}=0}^{\mathrm{lgn}} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Algoritmo Heapsort

- 1.- Construir un heap invocando a Build_Heap
- 2.- Intercambiar el primer elemento, la raíz y mayor elemento, del heap con el último.
- 3.- Restituir la propiedad heap en el heap de n-1 elementos.

```
Heapsort(A)
        Build_Heap(A)
        for (i= lenght(A) downto 2 ) do
                exchange A[1] <-> A[i]
                heap_size [A] = heap_size [A]
                Heapify(A,1)
```

- El costo de Heapsort es O(n lg n) porque el llamado a Build_Heap toma O(n) y luego tenemos n-1 llamados a Heapify cuyo tiempo es O(lgn).

Colas de Prioridad

- Heapsort es muy bueno, pero es superado por quicksort (lo veremos luego).
- La aplicación más popular de heapsort es para implementar colas de prioridad.
- Una cola de prioridad es una estructura de datos que mantiene un conjunto de S de elementos cada uno asosiado con una clave key.
- La cola de prioridad soporta las siguientes operaciones:

  Insert(S,x): inserta x en el conjunto S

  Maximum(S): retorna el elemento con mayor clave.

  Extract-Max(S): remueve y retorna el elemento con mayor clave.
- Aplicaciones:
  Itineración de tareas en sistemas de tiempo compartido
  colas de prioridad en simuladores conducidos por evento (Event-driven simulator)

Operaciones en Colas de prioridad

- Heap_Maximum(S): retorna el nodo raiz en tiempo $\theta(1)$.
- Heap_Extract_Max(A)

```
        if heap_size[A] <1 then error “Heap undeflow”
        max = A[1]
        A[1] = A[heap_size [A]]
        heap_size [A] = heap_size [A]]-1
        Heapify(A,1)
        return max
```

Tiempo de Heap_Extract_Max : O(lg n) básicamente el tiempo de Heapify

- Heap_Insert(A,key)

```
        heap_size [A] = heap_size [A]+1
        i = heap_size [A]
        while (i > 1 and  A[Parent(i)]  < key) do
                A[i]  = A[Parent(i)]
                i = Parent(i)
         A[i] = key
```

Tiempo de Heap_Insert : O(lg n) porque es el recorrido desde la nueva hoja a la raíz.