Analisis y diseño de algoritmos.
Heap.

Juan Gutiérrez

May 20, 2021

1 Heaps

Una estructura de datos *heap* es un arreglo (indexado desde 1) que puede ser visto como un arbol binario casi lleno (ver ejemplo).

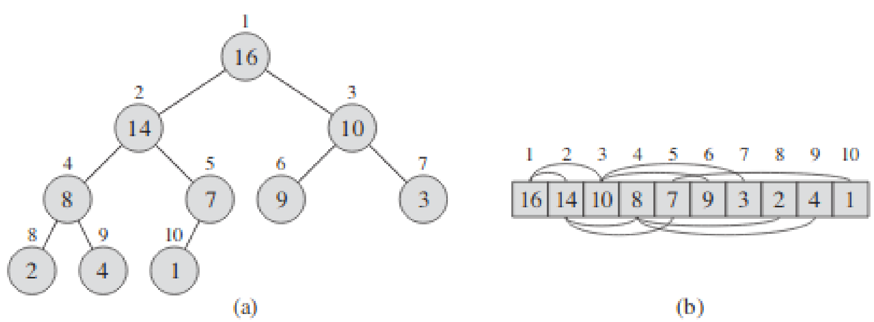

Figure 1: Tomada del libro Cormen, Introduction to Algorithms

Cada nodo del árbol corresponde a un elemento del arreglo. Todos los niveles del árbol están completos, excepto posiblemente el último.

Un arreglo A que representa un heap tiene dos atributos: *A.length*, *A.heap-size*, donde *heap-size* $\leq$ *length* (más adelante entenderemos esta diferencia).

La raíz del árbol es el elemento $A[1]$. Dado un índice i de un nodo, note que

- El padre de i es $\lfloor i/2 \rfloor$.
- El hijo izquierdo de i es $2i$.
- El hijo derecho de i es $2i + 1$.

Demostración. Demostraremos, por inducción en i, que los hijos izquierdo y derecho del índice i son $2i$ y $2i + 1$ respectivamente.

Si $i = 1$ entonces los hijos izquierdo y derecho son $2 = 2i$ y $3 = 2i + 1$ respectivamente.

Suponga ahora que $i > 1$. Por hipótesis de inducción, los hijos izquierdo y derecho de $i - 1$ son $2(i-1) = 2i - 2$ y $2(i-1)+1 = 2i - 1$.

Como los hijos de i son los nodos que vienen inmediatamente después de los hijos de $i-1$, tenemos que los hijos izquierdo y derecho de i son $2i$ y $2i+1$.

Ahora mostraremos que el padre de i es $\lfloor i/2 \rfloor$.

Si i es par entonces $i = 2k$ para algún entero k y portanto i es el hijo izquierdo de $k = i/2 = \lfloor i/2 \rfloor$. Si i es impar entonces $i = 2k+1$ para algún entero k y portanto i es el hijo derecho de $k = (i-1)/2 = \lfloor i/2 \rfloor$. $\square$

Entonces, podemos acceder en tiempo constante al padre e hijos de un índice i:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

Figure 2: Tomada del libro Cormen, Introduction to Algorithms

La *altura de un nodo* en un heap es el número de aristas en el camino máximo de dicho nodo hacia una hoja (este camino solo utiliza descendientes). La *altura de un heap* es la altura de su raíz

Ejercicio 1.1. *Cual es el mínimo y máximo número de elementos en un heap con altura h?*

La respuesta es 2^h y $2^{h+1} - 1$.

Demostracion 1: Note que el número de nodos a distancia d de la raíz, cuando $d < h$, es 2^d (probar por inducción).

También, el número de nodos con distancia h es mayor o igual que 1 pero menor o igual que 2^h (probar). Sea x esta cantidad de nodos. Luego, el total de nodos es igual a $\sum_{d=0}^{h-1} 2^d + x = 2^h - 1 + x \in [2^h, 2^{h+1} - 1]$. $\square$

Demostración 2: Comenzaremos probando, por inducción en d, que si un nodo i está a distancia d de la raíz, entonces $2^d \leq i \leq 2^{d+1} - 1$.

Cuando $d = 0$, tenemos que $i = 1$, y portanto $2^d = 2^0 = 1 \leq i \leq 2^1 - 1 = 1$.

Cuando $d > 0$, tenemos que $\textsc{Parent}(i) = \lfloor i/2 \rfloor$ está a distancia $d - 1$ de la raíz. Entonces, por hipótesis de inducción:

$$2^{d-1} \leq \lfloor i/2 \rfloor \leq 2^d - 1.$$

Luego $i/2 < \lfloor i/2 \rfloor + 1 \leq 2^d$, lo que implica que $i < 2^{d+1}$ y portanto $i \leq 2^{d+1} - 1$. También $2^{d-1} \leq \lfloor i/2 \rfloor \leq i/2$, lo que implica que $2^d \leq i$. Concluimos que $2^d \leq i \leq 2^{d+1} - 1$.

Ahora usaremos la propiedad anterior para demostrar lo pedido. Como el heap tiene altura h, el nodo n está a distancia h de la raíz. Portanto $2^h \leq n \leq 2^{h+1} - 1$. Luego el número mínimo de elementos es 2^h y el número máximo de elementos es $2^{h+1} - 1$. □

Ejercicio 1.2. *Pruebe que un heap con n nodos tiene altura $\lfloor \lg n \rfloor$*

Solución. Dado un heap con n nodos, del ejercicio anterior sabemos que $2^h \leq n < 2^{h+1}$, donde h es la altura del heap. Luego $h \leq \lg n < h + 1$, lo que implica que $h = \lfloor \lg n \rfloor$.

Propiedad 1.1. *Un heap con n nodos tiene altura $\Theta(\lg n)$*

Proof. Directamente del Ejercicio 1.2. □

2 Heap Property

Existen dos tipo de heaps: Max-heaps y Min-heaps. Dependiendo del tipo se debe cumplir la correspondiente propiedad.

- En un *Max-heap* se debe cumplir, para cada nodo i, $A[\text{PARENT}(i)] \geq A[i]$
- En un *Min-heap* se debe cumplir, para cada nodo i, $A[\text{PARENT}(i)] \leq A[i]$

Para este capítulo usaremos principalmente Max-heap.

Muchas veces nuestro heap no está cumpliendo la propriedad de max-heap, el siguiente algoritmo se encarga de modificar el heap a manera de que se cumpla.

El algoritmo recibe como entrada un heap A y un índice i tal que los heaps con raíces LEFT(i) y RIGHT(i) **son max-heaps** (ya cumplen la propiedad). Al finalizar el algoritmo, el heap con raíz i será Max-heap.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Figure 3: Tomada del libro Cormen, Introduction to Algorithms

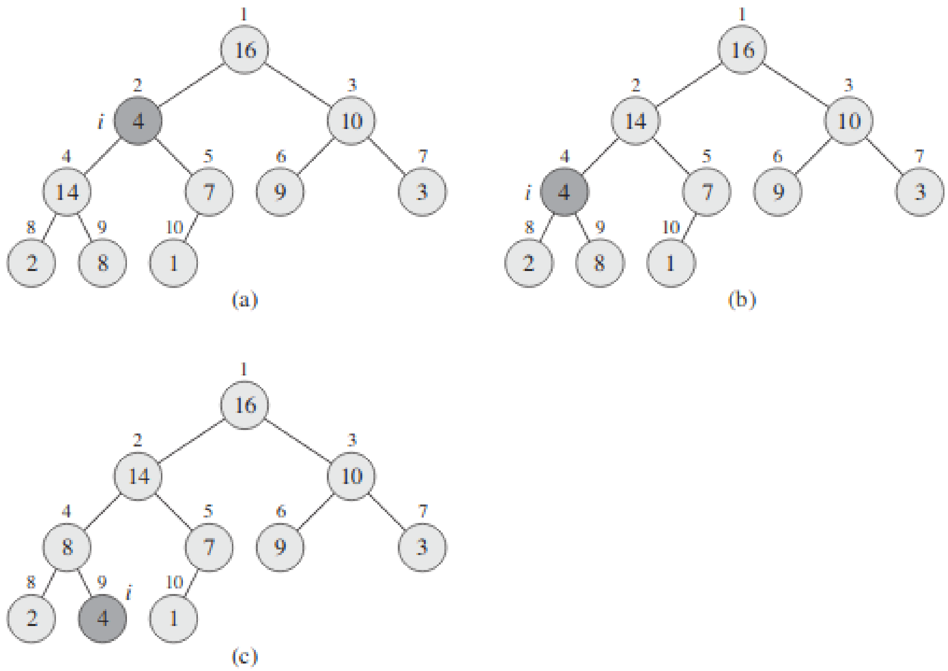

Figure 4: Tomada del libro Cormen, Introduction to Algorithms

Tiempo de ejecución de MAX-HEAPIFY. Note que las líneas 1 al 9 tienen tiempo constante. Luego, en el peor caso, tenemos que

$$T(n) \leq T(2n/3) + k.$$

¿Como se obtiene el $2n/3$? Sean n_i y n_d la cantidad de nodos de cada subárbol izquierdo y derecho respectivamente. Es claro que $n_i + n_d = n - 1$. Observe también que $n_i \leq 2n_d + 1$ (ejercicio), lo que implica que $n_d \geq \frac{n_i - 1}{2}$. Luego $n - 1 = n_i + n_d \geq n_i + \frac{n_i - 1}{2} = \frac{3n_i - 1}{2}$. Lo que implica que $n_i \leq \frac{2n-1}{3}$. Portanto, en el peor caso, el árbol izquierdo tendrá tamaño $\frac{2n-1}{3} \leq \frac{2n}{3}$.

Al resolver la recurrencia por teorema maestro, obtenemos que $T(n) = \Theta(\lg n)$ en el peor caso (portanto el algoritmo es $O(\lg n)$).

Ejercicio 2.1. *Un arreglo ordenado de manera creciente es un Min-heap, es un Max-heap?*

Ejercicio 2.2. *Considere el siguiente arreglo:* $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$. *Es un Min-heap, es un Max-heap?*

Ejercicio 2.3. *Corra la rutina* MAX-HEAPIFY(A,3) *en el arreglo* $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$

3 Construyendo un max-heap

En esta sección, mostraremos como construir un Max-heap a partir de un arreglo $A[1..n]$ cualesquiera.

Para ello haremos uso de MAX-HEAPIFY. Considere el siguiente algoritmo, llamado BUILD-MAX-HEAP. El algoritmo recibe un arreglo $A[1..n]$ e intercambia sus elementos de manera que el arreglo resultante es un Max-Heap.

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Figure 5: Tomada del libro Cormen, Introduction to Algorithms

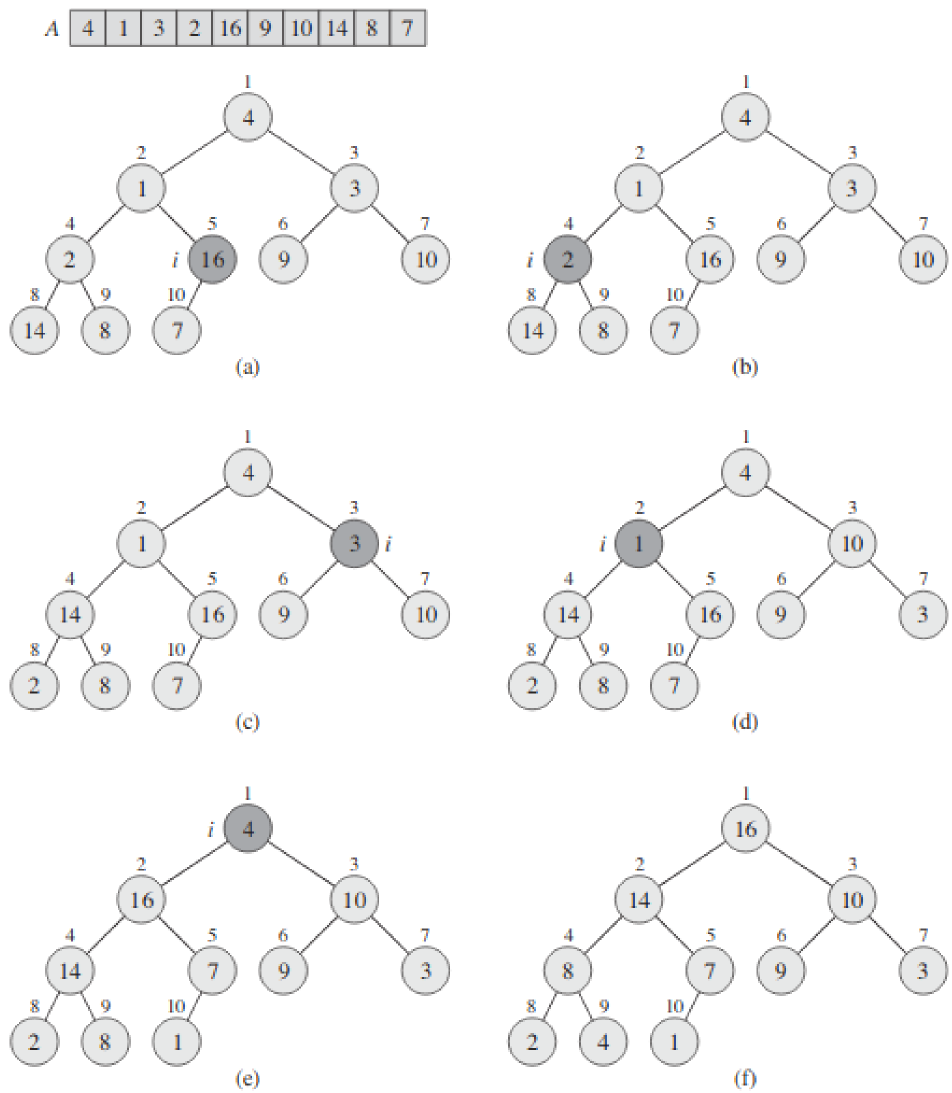

Figure 6: Tomada del libro Cormen, Introduction to Algorithms

Considere la invariante: “Al inicio de cada iteración del bucle for, cada nodo $i+1, i+2, \ldots, n$ es la raíz de un Max-Heap”.

- Inicialización: Al inicio de la primera iteración, tenemos $i = \lfloor n/2 \rfloor$. Como cada nodo $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ es una hoja (ejercicio), dicha hoja es un Max-Heap trivial y la propiedad se cumple.

- Manuntención: Por la invariante, dado un nodo i, sus hijos $2i$ y $2i+1$ son

Max-Heaps. Luego, al usar la subrutina MAX-HEAPIFY, el subarbol con raíz i también será un Max-Heap.

- Finalización: Al terminar tenemos que $i = 0$ y por lo tanto cada nodo $1, 2, \ldots, n$ es la raíz de un Max-Heap. Por tanto A ya es un Max-Heap.

Analizaremos el tiempo de ejecución de BUILD-MAX-HEAP.

Un primer análisis nos indica que hacemos aproximadamente $n/2$ llamadas a la subrutina MAX-HEAPIFY, la cual consume tiempo $O(\lg n)$. Portanto tenemos un tiempo de ejecución $O(n \lg n)$.

Sin embargo, el n de cada llamada recursiva es siempre menor que el n original. Nos conviene expresar el tiempo de ejecución de cada llamada en función a la altura del nodo en cuestión. Ya que, si la altura de i es h, una llamada a MAX-HEAPIFY(A, i) consumirá tiempo $R(h) = O(h)$. Supongamos que dicho tiempo es menor o igual hk.

Tenemos la siguiente propriedad

Propiedad 3.1. *En un heap con n nodos, existen como máximo $\lceil n/2^{h+1} \rceil$ nodos de altura h.*

Luego, como sabemos que la altura de un heap con n nodos es $\lfloor \lg n \rfloor$ (ver un ejercicio anterior). Obtenemos que el tiempo de ejecución $T(n)$ de BUILD-MAX-HEAP.

$$
\begin{aligned}
T(n) &= \sum_{h=1}^{\lfloor \lg n \rfloor} |\{i : \text{la altura de } i \text{ es } h\}| \cdot R(h) \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil kh \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} (\frac{n}{2^{h+1}} + 1)kh \\
&= \frac{kn}{2} \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \\
&\leq \frac{kn}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \\
&= kn + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \qquad (1) \\
&= O(n)
\end{aligned}
$$

Donde la ecuación (1) vale pues $\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$ (ver hoja de ejercicios de introducción).

Ejercicio 3.1. *Ilustre la operación* Build-Max-Heap *en el arreglo* $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$.

Ejercicio 3.2. *Que sucede si en lugar de derecementar i, hacemos el for incrementando i desde 1 hasta* $\lfloor A.length/2 \rfloor$

Ejercicio 3.3. *Muestre que existen como máximo* $\lceil n/2^{h+1} \rceil$ *nodos con altura h en un heap con n nodos.*

4 El algoritmo Heapsort

Queremos ordenar un arreglo de manera *creciente*.

La idea del algoritmo es aprovechar que en un Max-Heap, el elemento de mayor valor se encuentra en la raíz, esto es, el elemento $A[1]$. Por lo tanto, podemos intercambiar este elemento con el elemento de la posición $A[n]$ y reorganizar el heap $A[1..n-1]$ a manera de convertirlo en Max-Heap. Luego de esto, el segundo menor elemento estará en la posición $A[1]$.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
     exchange A[1] with A[i]
     A.heap-size = A.heap-size - 1
     MAX-HEAPIFY(A, 1)
```

Figure 7: Tomada del libro Cormen, Introduction to Algorithms

Como Build-Max-Heap consume tiempo $O(n)$ y existen $n-1$ llamadas a Max-Heapify, cada una de las cuales consume tiempo $O(\lg n)$, el tiempo de ejecución del algoritmoHeapsort es $O(n \lg n)$.

Ejercicio 4.1. *Ilustre la operación de* HeapSort *en el arreglo* $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.

Ejercicio 4.2. *Cual es el tiempo de ejecución de* HeapSort *cuando el arreglo está ordenado?*

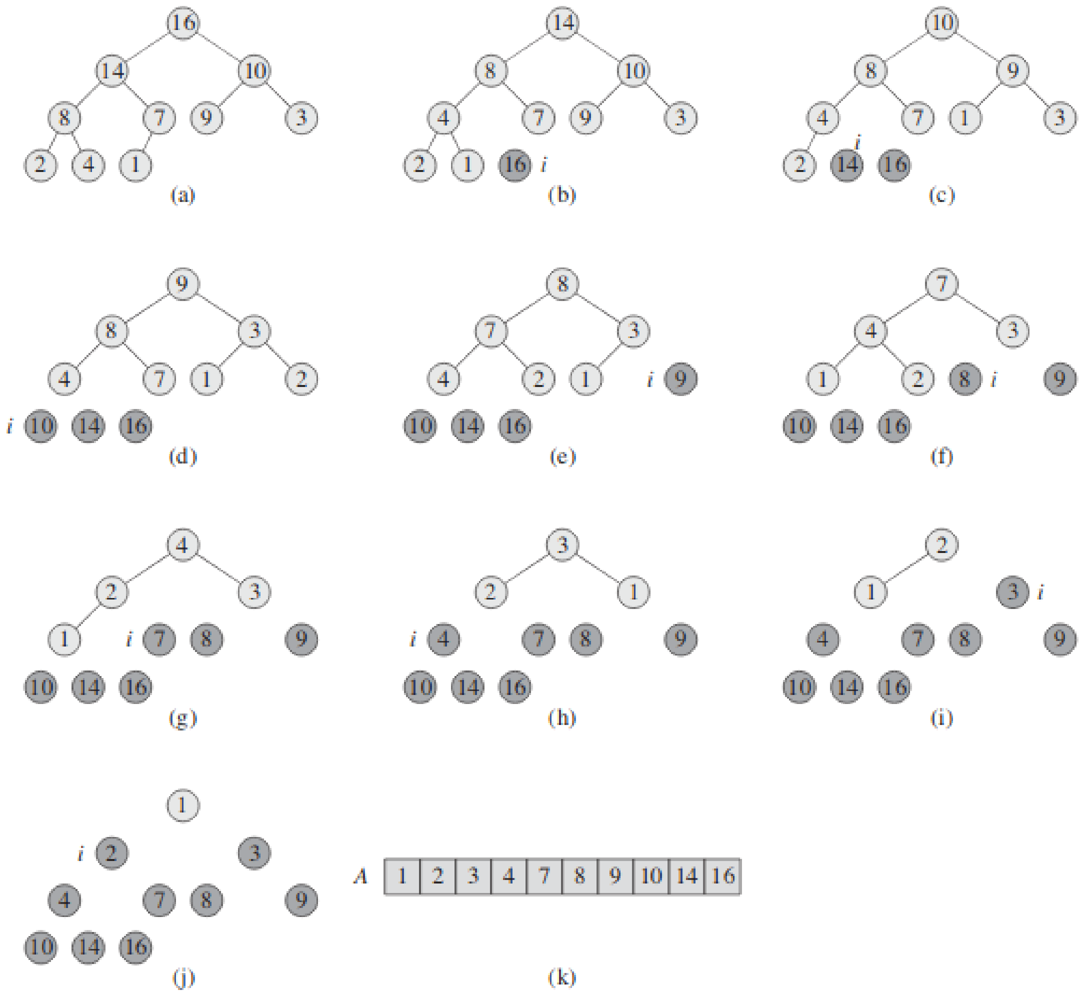

Figure 8: Tomada del libro Cormen, Introduction to Algorithms

5 Fila de Prioridades

La estructura de datos Heap tiene más usos aparte del Heapsort, en esta sección presentaremos una de las más populares: filas de prioridades. Nos centraremos en filas Max-Priority.

Una *fila de prioridad* es un arreglo $A[1..n]$, cuyos valores son llamados *keys*. Tenemos las siguientes operaciones:

- HEAP-MAXIMUM(A): retorna el elemento con mayor valor en A. Lo haremos en $\Theta(1)$.

- Heap-Extract-Max(A): remueve y retorna el elemento de A con el mayor valor. Lo haremos en $O(\lg n)$.
- Heap-Increase-Key(A, i, key): incrementa el valor de la llave del elemento $A[i]$ al nuevo valor key. Lo haremos en $O(\lg n)$.
- Max-Heap-Insert(A, key): inserta el elemento con valor key en A. Lo haremos en $O(\lg n)$.

Recibe: un max-heap A

Devuelve: el elemento máximo en A

```
HEAP-MAXIMUM(A)
return A[1]
```

Figure 9: Tomada del libro Cormen, Introduction to Algorithms

Recibe: un max-heap A

Devuelve: el elemento máximo en A. Además retira el elemento del heap.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Figure 10: Tomada del libro Cormen, Introduction to Algorithms

Recibe: un max-heap A, un índice i y un valor key

Incrementa el valor de $A[i]$ a key, además modifica A de manera que siga siendo max-heap.

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Figure 11: Tomada del libro Cormen, Introduction to Algorithms

Recibe: un max-heap A y un valor key

Inserta el valor key en A.

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Figure 12: Tomada del libro Cormen, Introduction to Algorithms

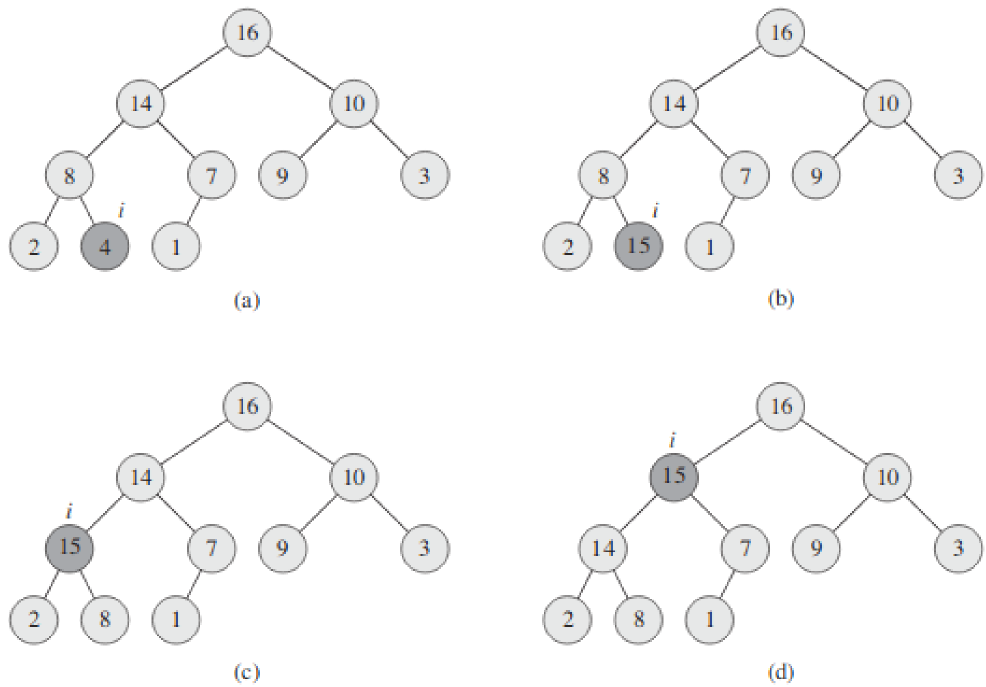

Figure 13: Simulación del Heap-Increase-Key. Tomada del libro Cormen, Introduction to Algorithms

Ejercicio 5.1. *Simule la operación* Heap-Extract-Max *en* $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$

Ejercicio 5.2. *Simule la operación* Max-Heap-Insert$(A, 10)$ *en* $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$

Ejercicio 5.3. *Construya una operación* Heap-Delete(A, i).