AED: Heaps

gabriel.spranger@utec.edu.pe
macarena.oyague@utec.edu.pe

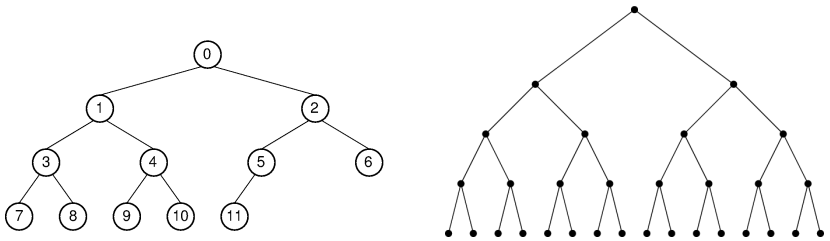
Mayo 2022

Contents

1 Terminología

1.1 Árbol Completo

Árbol binario donde cada *nivel* está *lleno*, sin contar el último, el cual **podría estar o no *lleno*** y además los nodos de este último nivel están **pegados hacia la izquierda**.



(a) Un árbol binario completo. (b) Un árbol binario completo y lleno.

Figure 1: Dos ejemplos de árboles binarios completos.

1.2 Heap

Es un árbol binario completo que mantiene la propiedad de **min-heap** o **max-heap**. La propiedad **min-heap** implica que el valor de un nodo x es **menor o igual** al valor de todos los descendientes de x. La propiedad **max-heap** es lo mismo pero **mayor o igual**.

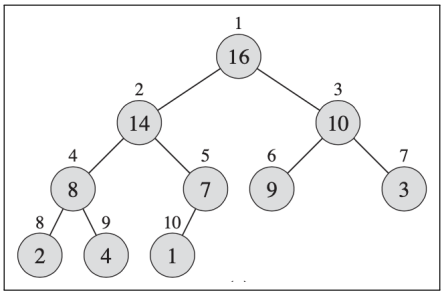


Figure 2: Un heap que cumple la propiedad **max-heap**. Imagen tomada del Cormen.

2 ¿Cómo representar un heap?

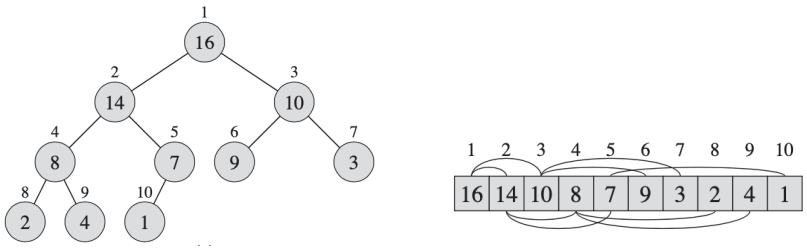
Hay varias maneras, dos de ellas son las siguientes:

1. Con un nodo que tiene un hijo izquierdo y derecho.

La ventaja de esta implementación es que es simple y familiar a un *binary search tree*. Ya sabemos cómo implementar varios algoritmos sobre un árbol que maneja esta estructura.

2. Con un array donde cada posición i representa un nodo cuyo padre está en la posición $\lfloor \frac{i}{2} \rfloor$ y cuyo hijo izquierdo y derecho están en $2i$ y $2i+1$, respectivamente.

   La ventaja de esta implementación es que usa menos memoria ya que en la implementación anterior por cada nodo se tiene que guardar el valor que almacena y dos punteros, mientras que en esta implementación cada nodo solo guarda su valor. También las consultas PARENT, LEFT y RIGHT son muy simples y requieren de pocas instrucciones de hardware para ejecutar. Por ejemplo: PARENT es un *shift right logical* de una posición; LEFT es un *shift left logical* de una posición; RIGHT es un *shift left logical* de una posición y un *add*. Además, como veremos en la sección de INSERT, es mucho más fácil implementarlo con un array que con el método del nodo con puntero izquierdo y derecho. Otro beneficio es que tener un heap en un array nos hace el *level-order traversal* o BFS gratis:



(a) Un max-heap.

(b) El mismo max-heap en su representación con un array.

Figure 3: Notar que para recorrer el heap a modo de BFS, basta con recorrer del array de inicio a fin. Imágenes tomadas del Cormen.

3 Operaciones

Como se mencionó anteriormente, un heap mantiene la propiedad de **max-heap** o **min-heap** y además es un **árbol completo** (y como la altura de un árbol completo siempre es logarítmica, las operaciones que dependan de la altura del heap correrán en tiempo logarítmico). Estas dos propiedades hacen que ciertas operaciones que veremos en esta sección corran en tiempo logarítmico y otras en tiempo constante. Hay dos tipos de operaciones: consultas y mutaciones. La idea es que las mutaciones modifiquen la estructura de datos de tal manera que las propiedades descritas anteriormente se mantengan, de lo contrario se pierden los beneficios que estas propiedades traen (operaciones en tiempo logarítmico y

constante).

¿Por qué un heap con n nodos tiene altura logarítmica?

Primero definamos cotas inferiores y superiores sobre el número total de nodos de un heap con altura h. Recordemos que el heap es un árbol completo, es decir, todos los niveles estan llenos excepto por el último que puede o no estar lleno. La cota superior sería:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Ya que cada nivel de un árbol binario tiene como máximo 2^i nodos y como queremos una cota superior, asumimos que el heap tiene todos sus niveles llenos (árbol completo perfecto).

La cota inferior ocurre cuando todos los niveles están llenos y el último nivel tiene un solo nodo:

$\sum_{i=0}^{h-1} 2^i$, todos los niveles llenos excepto por el último, por eso $h-1$

La sumatoria anterior equivale a:

$$2^h - 1$$

A lo que le sumamos 1 por el nodo solitario del último nivel del heap:

$$2^h - 1 + 1 = 2^h$$

Entonces:

$$2^h \leq n \leq 2^{h+1} - 1$$

$$2^h \leq n < 2^{h+1}$$

Sacándole logaritmo en base 2 a todo:

$$h \leq \lg n < h + 1$$

$$h = \lfloor \lg n \rfloor$$

Por lo tanto la altura de un heap es logrítmica.

3.1 Consultas

3.1.1 Min y Max

Dada la propiedad de **min-heap**, tenemos que dado un nodo x, todos los descendientes de x tienen un valor menor o igual a x. Por lo tanto, si tenemos un heap que cumple con la propiedad de **min-heap**, entonces el nodo raíz tiene el menor valor de los n nodos del heap. Lo mismo ocurre para un **max-heap**. Por lo tanto, para hallar el mínimo o el máximo (de un min o max heap, respectivamente), basta con consultar el valor del nodo raíz, el cual se encuentra en $A[1]$ (porque manejamos nuestro heap en un array A). Como esta consulta es un simple acceso de memoria, corre en $\mathcal{O}(1)$.

3.2 Mutaciones

3.2.1 Heapify Down

Asumiendo que lidiamos con un max-heap, el algoritmo *heapify down* empieza desde un nodo x que podría estar violando la propiedad de max-heap, pero se asume que ambos *subheaps* de x son max-heaps.

Ya que x *posiblemente* está violando la propiedad de máx heap, significa que su hijo izquierdo o derecho podrían ser mayores que x. Si resulta que x es mayor a ambos, el algoritmo acaba, ya que no hay nada que arreglar, dado que x satisface la propiedad de max-heap (ya que es mayor que sus hijos) y sus hijos también satisfacen la propiedad ya que el algoritmo asume que ambos *subheaps* ya son max-heaps. Pero si el hijo izquierdo o derecho es mayor (llamémosle z), entonces intercambiamos el valor de x por el valor z y ahora el heap que tiene como raíz a z cumple la propiedad de max-heap, pero no estamos seguros si el heap que ahora tiene a x como raíz (antes tenía a z como raíz) cumple con la propiedad de max-heap, entonces llamamos recursivamente el algoritmo *heapify down* en este nodo.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Figure 4: Pseudocódigo de *heapify down*. Imagen tomada del Cormen.

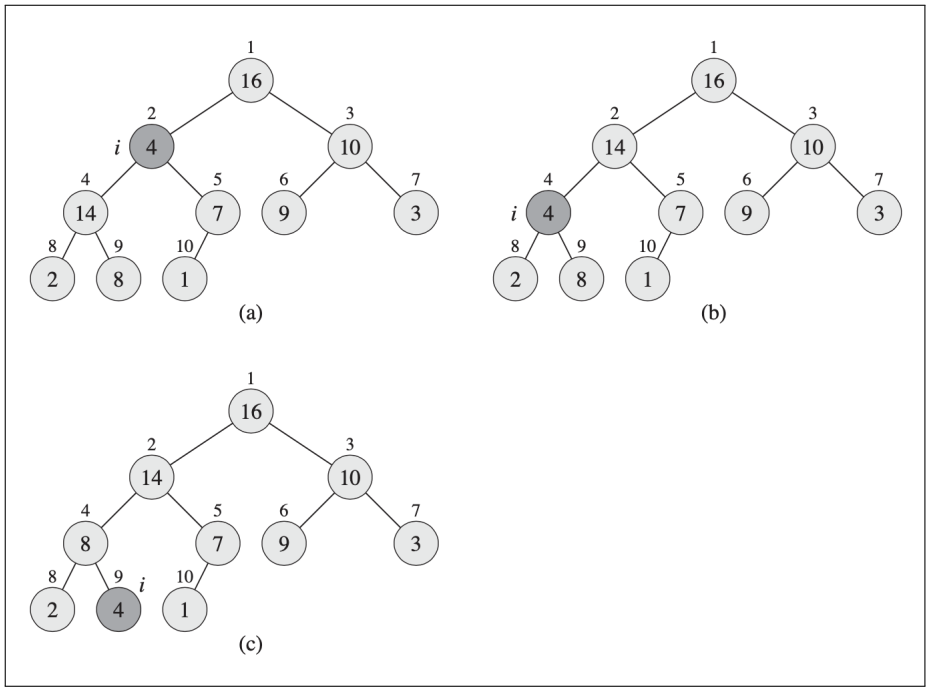


Figure 5: Ilustración del algoritmo *heapify down*. El algoritmo empieza desde el nodo con valor 4. Luego, se llama de nuevo sobre la nueva ubicación del 4, hasta que al final el heap que tiene a 4 como raíz cumple la propiedad de max-heap. Imagen tomada del Cormen.

La complejidad de *heapify down* es $\mathcal{O}(1)$ para hallar el nodo con el máximo valor e intercambiar si es necesario sumado con el tiempo tomado por la llamada

recursiva. Para hallar una cota superior sobre el tiempo tomado por la llamada recursiva y así poder decir que *heapify down* corre en $\mathcal{O}$(algo), tenemos que hallar el peor caso de *heapify down*. Esto se ve mejor definiendo una recurrencia:

$$T(n) \leq T(\text{número máximo de nodos en un subárbol}) + \mathcal{O}(1)$$

Donde $T(n)$ representa en tiempo de ejecución de *heapify down*, el $+ \mathcal{O}(1)$ representa el trabajo constante de hallar el nodo con el máximo valor e intercambiarlo si es necesario y T(número máximo de nodos en un subarbol) representa el tiempo de ejecución de la llamada recursiva de *heapify down*. Como queremos hallar una cota superior (sino no podemos usar la notación $\mathcal{O}$, por ello notar el $\leq$), tenemos que tomar en cuenta el *número máximo de nodos en un subárbol*. Resulta que este número máximo es $\frac{2n}{3}$. Por lo tanto, la recurrencia quedaría así:

$$T(n) \leq T(\frac{2n}{3}) + \mathcal{O}(1)$$

Resolviendo esta recurrencia mediante el Teorema Maestro (hay otros métodos para resolverla), que también se encuentran descritos en el capítulo 4 del Cormen, nos da que:

$$T(n) = \mathcal{O}(\lg\ n)$$

Para los de CS: verán más análisis de este tipo en Análisis y Diseño de Algoritmos (ADA).

3.2.2 Construir

Asumiendo que lidiamos con un max-heap, dada la propiedad de "corrección" del algoritmo *heapify down*, podemos aprovecharla para convetir un array cualquiera en un max-heap. Recuerden que *heapify down* requiere que dado un nodo en la posición i que posiblemente está rompiendo la propiedad del max-heap, ambos hijos suyos cumplan la propiedad. Uno podría pensar que se podría llamar *heapify down* desde el índice 1 hasta el n para asegurarnos así que el array termine siendo un max-heap, pero eso no funcionaría porque los hijos del nodo en el índice 1 del array que queremos convertir a un heap, puede ser que tampoco sean max-heaps.

Tomando en cuenta el requerimiento de *heapify down* para que funcione bien, podemos aprovechar el hecho de que las hojas son max-heaps triviales (como no tienen hijos, cumplen trivialmente la propiedad de max-heap). Entonces, podemos empezar llamando a *heapify down* desde el "último padre" (ya que sus hijos son hojas y por lo tanto max-heaps, así que *heapify down* funcionará ahí), hasta llegar a la raíz (el "primer padre") y cuando se llegue acá, ambos hijos serán max-heaps y también *heapify down* funcionará.

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

Figure 6: Pseudocódigo de BUILD-MAX-HEAP. El bucle *for* también podría leerse como *desde el último padre hasta el primer padre*. Imagen tomada del Cormen.

Recordar que PARENT$(i) = \lfloor \frac{i}{2} \rfloor$, entonces $\left\lfloor \frac{A.length}{2} \right\rfloor$ es el padre del último nodo (el nodo en el índice $A.length$), por lo tanto, el "el útimo padre", cuyos hijos son hojas y por lo tanto max-heaps.

Si desean ver una demostración breve de por qué BUILD-MAX-HEAP funciona, vayan a la sección 6.3 del Cormen.

Un primer análisis sobre el tiempo de ejecución de BUILD-MAX-HEAP nos podría decir que como una llamada a MAX-HEAPIFY toma $\mathcal{O}(\lg\ n)$ y se hacen $\frac{n}{2}$ llamadas a esta función, el tiempo de ejecución de BUILD-MAX-HEAP está acotado superiormente por $\mathcal{O}(n\ \lg\ n)$. Sea $T(n)$ el tiempo de ejecución de BUILD-MAX-HEAP dado que el array A tiene longitud n:

$$T(n) = \frac{n}{2}\mathcal{O}(\lg\ n) + \mathcal{O}(1)$$

$$T(n) \leq n\mathcal{O}(\lg\ n) + \mathcal{O}(1)$$

$$T(n) = \mathcal{O}(n\ \lg\ n)$$

Pero este es un análisis muy flojo. Para ver una explicación completa de por qué este algoritmo corre en $O(n)$ les recomendamos revisar la sección 6.3 del Cormen.

3.2.3 Heapify Up

Asumiendo que lidiamos con un max-heap, el algoritmo *heapify up* hace algo muy similar a *heapify down* solo que es más directo. Dado un nodo x, *heapify up* compara el valor de x con su padre y si el valor de x es mayor, entonces intercambia ambos valores y se repite este proceso hasta que el padre sea mayor o ya no haya padre.

```
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

Figure 7: Pseudocódigo de *heapify-up* para un **max-heap**. Notar que i es el índice del nodo donde se desea empezar el *heapify-up*. "**while** $i > 1$" se lee como "mientras siga teniendo un padre" (porque el padre está en la posición 1). Imagen tomada del Cormen.

Como en el peor caso el nodo i empieza desde el último nivel del max-heap hasta que se convierte en la raíz, es decir, recorre toda la altura del max-heap y como sabemos un max-heap es un árbol completo y como un árbol completo siempre tiene altura $\lfloor \lg n \rfloor$, *heapify up* corre en $\mathcal{O}(\lg n)$.

3.2.4 Insert

Asumiendo que lidiamos con un max-heap, para insertar, ponemos el nuevo valor al final del array y como puede ser que el nuevo nodo tenga un valor mayor al de su padre, usamos el algoritmo *heapify up* para asegurarnos que luego de la inserción se siga cumpliendo la propiedad de max-heap. Como insertar al final del array es $\mathcal{O}(1)$* y *heapify up* toma $\mathcal{O}(\lg n)$, *insert* corre en $\mathcal{O}(\lg n)$.

*Depende de la implementación del array. Si se usa un vector, entonces insertar al final toma $\mathcal{O}(1)$ *amortizado* (análisis amortizado: también lo verán en ADA). Si quieren saber más sobre esto, escríbannos por Discord.

3.2.5 Extract Min y Extract Max

Asumiendo que lidiamos con un max-heap, el objetivo de *extract max* es obtener el valor máximo y eliminarlo del max-heap. Para esto intercambiamos $A[1]$ (el máximo) con $A[A.length]$ (la "última hoja"), guardamos una copia de $A[A.length]$ (el cual ahora es el máximo) y lo eliminamos (quizás con un POP-BACK si implementamos el array con un vector). Finalmente, como la "última hoja" está ahora como raíz, de todas maneras se está incumpliendo la propiedad de max-heap, por lo tanto es necesario usar el algoritmo *heapify down* para reubicar ese nodo en su posición correcta. Los intercambios y la eliminación del último elemento toman $\mathcal{O}(1)$, mientras que *heapify down* toma $\mathcal{O}(\lg n)$, por lo tanto *extract max* corre en $\mathcal{O}(\lg n)$.

3.2.6 Increase Key

Asumiendo que lidiamos con un max-heap, *increase key* apunta a aumentar el valor de un nodo que ya está en el max-heap. Como puede ser que el nuevo valor del *key* o valor, sea mayor al del padre, hacemos uso de *heapify up* para asegurarnos que se mantenga la propiedad. Notar que si no se provee el índice del nodo, se tiene que buscar linealmente ($\mathcal{O}(n)$) por la ubicación del nodo y

luego ejecutar el algoritmo de *increase key*. Si se provee el índice, *increase key* corre en $\mathcal{O}(\lg\ n)$, caso contrario, corre en $\mathcal{O}(n)$.

3.2.7 Decrease Key

Asumiendo que lidiamos con un max-heap, *decrease key* apunta a disminuir el valor de un nodo que ya está en el max-heap. Como puede ser que el nuevo valor del *key* o valor, sea menor al de uno de sus hijos, hacemos uso de *heapify down* para asegurarnos que se mantenga la propiedad. Notar que si no se provee el índice del nodo, se tiene que buscar linealmente ($\mathcal{O}(n)$) por la ubicación del nodo y luego ejecutar el algoritmo de *decrease key*. Si se provee el índice, *decrease key* corre en $\mathcal{O}(\lg\ n)$, caso contrario, corre en $\mathcal{O}(n)$.

4 Heapsort

El objetivo de *heapsort* es recibir un arreglo y ordenarlo. Para esto, se construye un max-heap en $\mathcal{O}(n)$, luego se hacen n *extract max*. Con un max-heap, se obtiene un orden descendiente, mientras que con un min-heap se obtiene un orden ascendiente. El tiempo de ejecución de *heapsort sería*:

$$T(n) = n\mathcal{O}(\lg\ n) + \mathcal{O}(n)$$

$$T(n) = \mathcal{O}(n\ \lg\ n)$$

5 Palabras Finales

- Los heaps tienen un montón de aplicaciones: hallar *minimum spanning trees*, ya que se puede obtener la arista con menor peso y eliminarla (*extract min*) en tiempo logarítmico; *dijkstra* para también hallar la arista con menor peso (*extract min*) en tiempo logarítmico; manejar la lista de procesos del sistema operativo junto con su prioriddad; etc.
- Hay una estructura de datos llamada *Fibonacci Heap* que tiene tiempos de ejecución mucho mejores que un heap, pero mediante un análisis **amortizado**.

Procedure	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$
INSERT	$\Theta(1)$
MINIMUM	$\Theta(1)$
EXTRACT-MIN	$O(\lg n)$
UNION	$\Theta(1)$
DECREASE-KEY	$\Theta(1)$
DELETE	$O(\lg n)$

Figure 8: Tiempos de ejecución **amortizado** para algunas operaciones de un *Fibonacci Heap*. Imagen tomada del Cormen.

6 Referencias

- Introduction to Algorithms (Cormen) Capítulo 6: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf