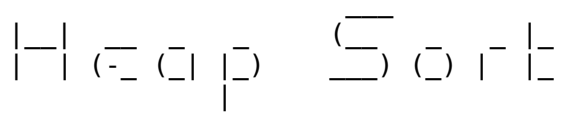
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

Se presenta ahora un método de ordenamiento del grupo de “tranformar y conquistar”, en inglés “transform and conquer”. Es un algoritmo muy interesante y rápido, especialmente diseñado para sacar ventaja de una estructura llamada “heap” que puede ordenar rápidamente una lista.

▬▬▬▬ 2. Representación de un Árbol Binario en un Arreglo.

En esta parte queremos estudiar como un árbol binario completo, se puede utilizar para representar un vector.

Para representar un árbol binario mediante un arreglo tenemos que tomar en consideración varias cosas:

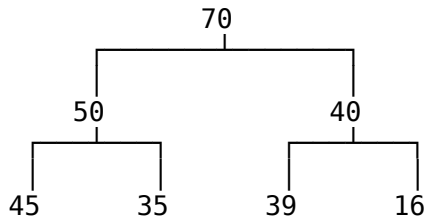
• Almacenar todos los elementos.

• Mantener las relaciones de padre e hijos.

• Y hacerlo de la manera más eficiente posible.

>>> Un Primer Árbol

Supongamos que tenemos el siguiente árbol:



Este árbol se puede representar mediante el siguiente arreglo:

Árbol: [70,50,40,45,35,39,16]
1 2 3 4 5 6 7

Observemos que se encuentran presentes todos los elementos, los cuales fueron incluidos al recorrer el árbol por niveles de izquierda a derecha. Y la relación de los elementos se mantiene por:

Si un nodo está en la posición “i” entonces:

El hijo izquierdo está en 2*i

El hijo derecho está en 2*1+1

El padre se encuentra en floor(i/2)

Veamos un ejemplo:

El nodo con el número 50 se encuentra en i=2.

El hijo izquierdo está en 2*i = 2*2 = 4

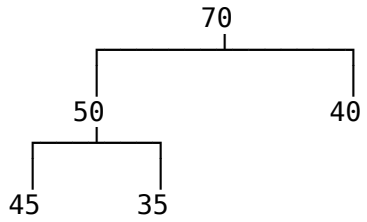
El hijo derecho está en 2*i+1 = 2*2+1 = 5

El padre está en la posición floor(i/2) = floor(2/2) = 1

>>> Un Segundo Árbol

Veamos este otro árbol binario y su representación:

Supongamos que tenemos el siguiente árbol:



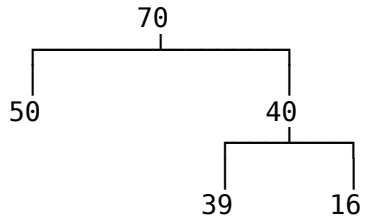
Este árbol se puede representar mediante el siguiente arreglo:

Árbol: [70,50,40,45,35]
          1 2 3 4 5

Lo que produce un vector continuo, sin lugares vacíos.

>>> Un Tercer Árbol.

Si tenemos este árbol:



Este árbol se puede representar mediante el siguiente arreglo:

Árbol: [70,50,40,(),(),39,16]
          1 2 3 4 5 6 7

En este caso, para poder utilizar correctamente las fórmulas de los subíndices, debemos dejar los hijos nulos dentro del arreglo.

▬▬▬ 3. Árbol Binario Completo.

En los ejemplos anteriores se puede observar el concepto de árbol binario completo, en inglés “complete binary tree”.

Un árbol binario está completo cuando al recorrer sus elementos por niveles de izquierda a derecha, se obtiene un vector continuo sin elementos nulos o vacíos entre dos elementos.

Dicho de otra manera, un árbol binario está completo, si al recorrerlo por niveles, todos los niveles están llenos. Y si el último nivel no está lleno, entonces todos los elementos se encuentran hacia el lado izquierdo del árbol.

Como pueden ver en los ejemplos anteriores:

El primer ejemplo es un árbol completo.

Es además un árbol lleno, porque todos sus niveles están llenos.

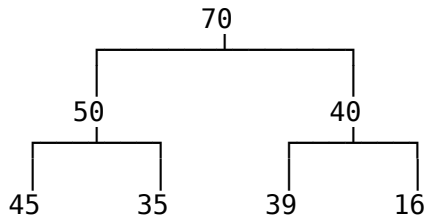
El segundo ejemplo es un árbol completo.

El tercer ejemplo *no* es un árbol completo.

Al mantener esta característica, se puede establecer que el alto de un árbol binario completo está dada por:

Si tiene “n” elementos, entonces
la altura será de $\log_2(n)$

Por ejemplo observe este árbol completo:



Como tiene 7 elementos, su altura será de:

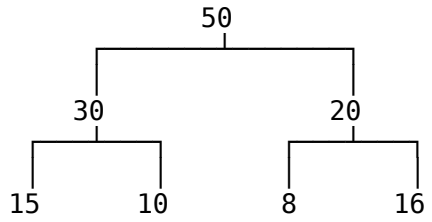
$\log_2(n) = \log_2(7) = 2.8073 \sim 2$

▬▬▬ 4. Estructura de Heap.

Un árbol se considera un “max heap” si tiene dos condiciones.

- Es un árbol binario completo.
- Cada nodo es mayor o igual que sus descendientes.

Por ejemplo este árbol es un max heap.

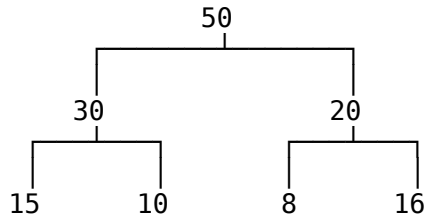


Árbol: [50,30,20,15,10, 8,16]
1 2 3 4 5 6 7

Existe una estructura similar llamada “min heap”, y sus cualidades se pueden deducir fácilmente. En este capítulo estaremos tratando únicamente con el “max heap”.

▬▬▬ 5. Insertar en un Max Heap.

Se tiene el siguiente árbol, que cumple con las condiciones de un “max heap”.



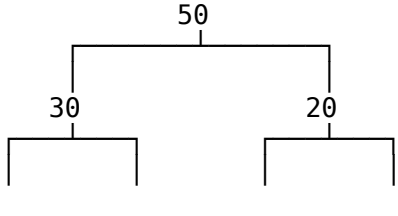
Árbol: [50,30,20,15,10, 8,16]
1 2 3 4 5 6 7

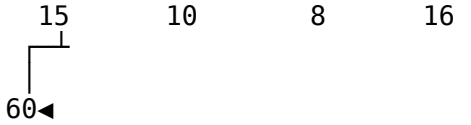
Supongamos que queremos insertar un elemento con valor de 60.

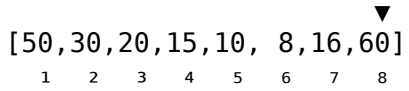
Para ingresar este valor y mantener las cualidades de un árbol binario completo y con las condiciones del “max heap”, se debe insertar en la última línea lo más a la izquierda posible y luego se acomoda en su lugar. En caso que el último nivel esté lleno, se debe crear un nuevo nivel.

El nuevo elemento se debe insertar en el último nivel disponible, lo más a la izquierda posible, lo que hace que se localice al final del vector.

>>> Inserción inicial.

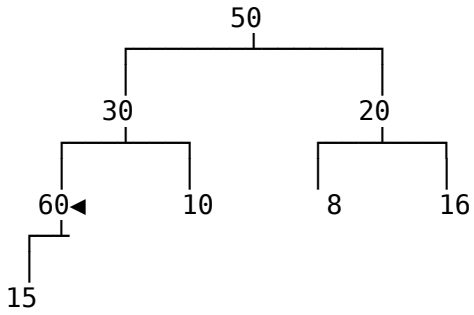


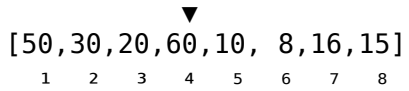


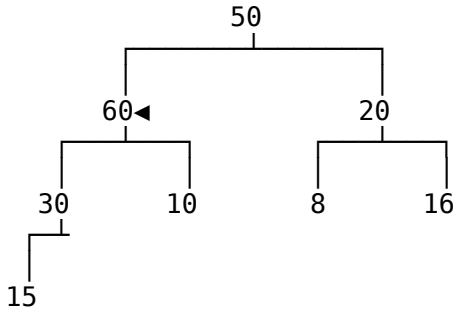


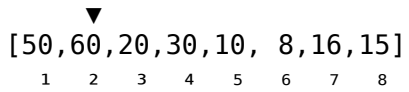
>>> Ajuste del Arbol Binario

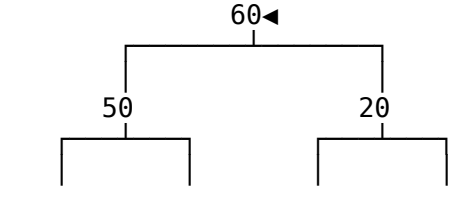
Se compara el valor de 60 con sus padres haciendo un “swap” cons esos números. Inicialmente tenemos:

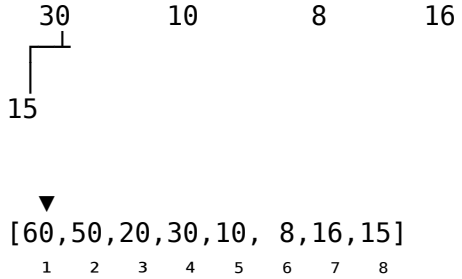












>>> Tiempo de Ejecución para Insertar.

Cuánto tiempo tomó el poner el valor de 60 en su posición correcta?

Tomó la altura del árbol, en un árbol con 8 elementos la altura es:

$\log_2(8) = 3$

En un árbol con “n”elementos la altura es:

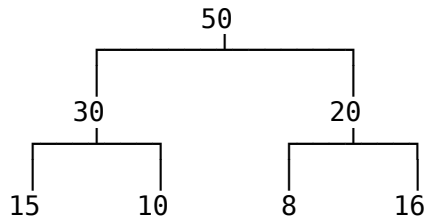
$\log_2(n)$

Por lo tanto el tiempo que se toma para una inserción, en el peor caso, sería de:
$O(\log_2(n))$

Y el mejor tiempo, sería que se inserte un elemento que está en la posición correcta, por lo que el mejo tiempo estará dado por:
$O(1)$

▬▬▬ 6. Borrar en un Max Heap.

Supongamos que tenemos el siguiente árbol, que cumple con las condiciones de un “max heap”.



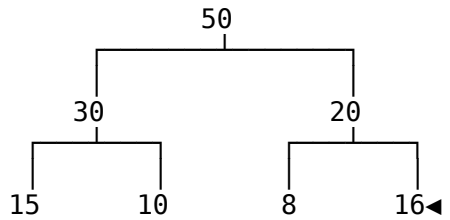
Árbol: [50,30,20,15,10, 8,16]
1 2 3 4 5 6 7

En el “max heap”, solo se permite borrar el elemento de la raíz.

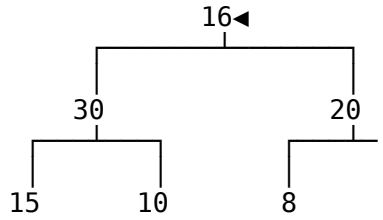
Veamos cómo se realiza este proceso para preservar la estructura del “max heap”.

>>> Se remueve el elemento de la raíz.

Al remover el máximo elemento, se pone el último elemento del vector.
Es decir, el elemento del último nivel que esté más a la derecha.

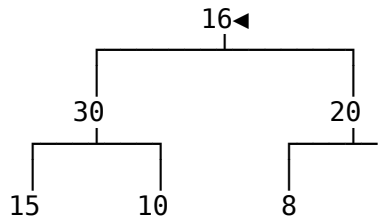



```
                                  ▼
Árbol: [50,30,20,15,10, 8,16]
          1  2  3  4  5  6  7
```

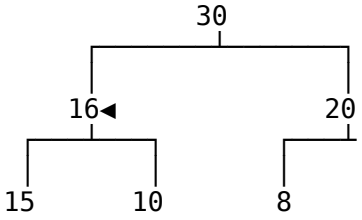



```
       ▼
Árbol: [16,30,20,15,10, 8]
          1  2  3  4  5  6
```

>>> Se mueve de la raíz hacia abajo.




```
       ▼
Árbol: [16,30,20,15,10, 8]
          1  2  3  4  5  6
```



Árbol: [30,16,20,15,10, 8]
1 2 3 4 5 6

Como 16 > max(15,10) tenemos el árbol final.

>>> Tiempo de Ejecución para Borrar.

El tiempo de ejecución para borrar un elemento depende de la altura del árbol.

De nuevo estamos en el tiempo:

$O(\log_2(n))$

▬▬▬ 7. “Heap Sort” con Insercción y Borrado.

El “heap sort” tiene dos partes.

Inicialmente debemos crear un “max heap” con el conjunto de elementos que debemos de ordenar.

Luego, vamos sacando elementos del “max heap”, hasta que quede vacío y así obtendremos una lista ordenada.

▬▬▬ 8. Proceso de Insercción.

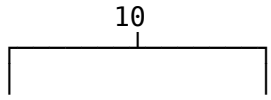
Supongamos que tenemos esta lista de elementos. Evidentemente no están ordenados y no son un max heap. Deseamos crear con ellos una estructura de max heap.

>>> Se crea el max heap con un elemento

Se tiene el siguiente arreglo, con el que se construye un “heap” con un solo elemento.

vec: [10,20,15,30,40]

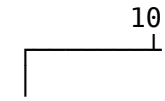
```
          1   2   3   4   5
```



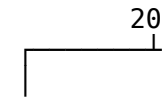

```
heap: [10,(),(),(),()]
          1   2   3   4   5
```

>>> Se agrega el siguiente elemento

```
              ▼
vec:  [10,20,15,30,40]
          1   2   3   4   5
```



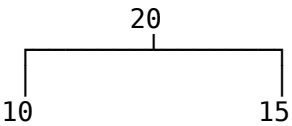
Y se ajusta:



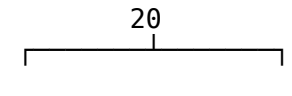

```
heap: [20,10,(),(),()]
          1   2   3   4   5
```

>>> Se agrega el siguiente elemento

```
                  ▼
vec:  [10,20,15,30,40]
          1   2   3   4   5
```

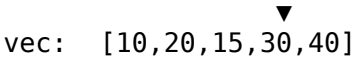


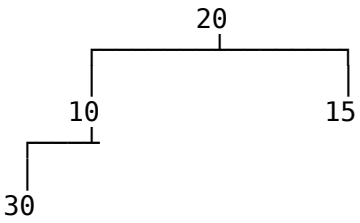
Y se ajusta:



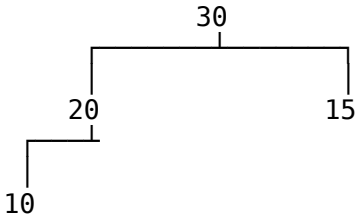
heap: [20,10,15,(),()]
1 2 3 4 5

>>> Se agrega el siguiente elemento



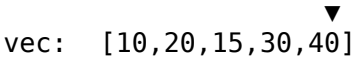


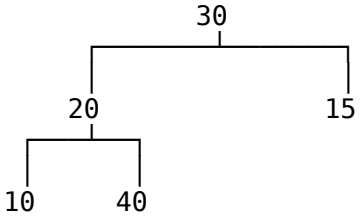
Y se ajusta:



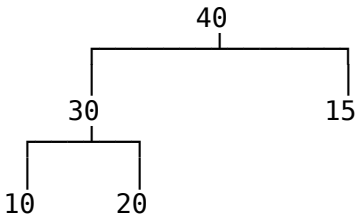
heap: [30,20,15,10,()]
1 2 3 4 5

>>> Se agrega el siguiente elemento





Y se ajusta:



heap: [40,30,15,10,20]
1 2 3 4 5

>>> Tiempo de Ejecución.

Se han insertado “n” elementos en el árbol.

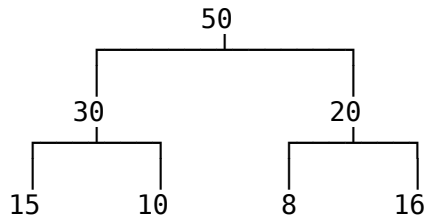
Y sabíamos que el tiempo de inserción de cada elemento es de $\log_2(n)$.

Por lo tanto toma un tiempo de $O(n * \log_2(n))$

▬▬▬ 9. Proceso de Borrado.

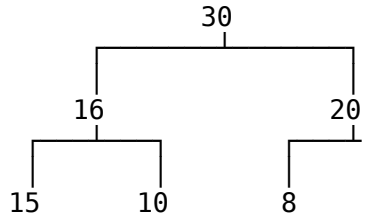
Cada vez que elimino un elemento del “max heap”, obtengo el elemento de mayor valor. Al borrar el elemento 50, el tamaño del vector sigue igual, y puedo poner el mayor valor al final del arreglo de la siguiente manera:

>>> Tenemos este “max heap”:



Árbol: [50,30,20,15,10, 8,16]
1 2 3 4 5 6 7

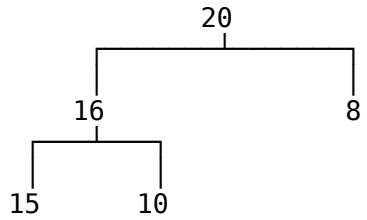
>>> Al eliminar el 50 tenemos:




```
                                    ▼
Árbol   : [30,16,20,15,10, 8,()]
            1   2   3   4   5   6   7

Solución: [50]
            1
```

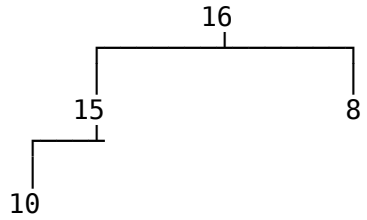
>>> Al eliminar el 30 tenemos:




```
                        ▼
Árbol   : [20,16, 8,15,10,(),()]
            1   2   3   4   5   6   7

Solución: [30,50]
            1   2
```

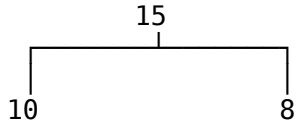
>>> Al eliminar el 20 tenemos:




```
                    ▼
Árbol   : [16,15, 8,10,(),(),()]
            1   2   3   4   5   6   7

Solución: [20, 30,50]
            1   2   3
```

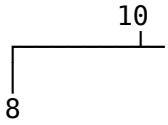
>>> Al eliminar el 16 tenemos:




```
                    ▼
Árbol   : [15,10, 8,(),(),(),()]
            1   2   3   4   5   6   7

Solución: [16,20,30,50]
            1   2   3   4
```

>>> Al eliminar el 15 tenemos:




```
                ▼
Árbol   : [10, 8,(),(),(),(),()]
            1   2   3   4   5   6   7

Solución: [15,16,20,30,50]
            1   2   3   4   5
```

>>> Al eliminar el 10 tenemos:


```
            ▼
Árbol   : [ 8,(),(),(),(),(),()]
            1   2   3   4   5   6   7

Solución: [10,15,16,20,30,50]
```

1 2 3 4 5 6

>>> Se finaliza el proceso:

Árbol : [(),(),(),(),(),(),()]
1 2 3 4 5 6 7

Solución: [8,10,15,16,20,30,50]
1 2 3 4 5 6 7

Al continuar este proceso hasta dejar el “max heap” vacío, se puede observar que el resultado final será un vector con los elementos ordenados de menor a mayor.

Árbol: [8,10,15,16,20,30,50]
1 2 3 4 5 6 7

>>> Tiempo de Ejecución.

Se han borrado “n” elementos en el árbol.

Y sabíamos que el tiempo de borrado de cada elemento es de $\log_2(n)$.

Por lo tanto toma un tiempo de $O(n * \log_2(n))$

▬▬▬ 10. Tiempo de Ejecución.

>>> Heap Sort con Inserciones Sucesivas.

El proceso de insertar en un “max heap” toma un tiempo de $O(n * \log_2(n))$.

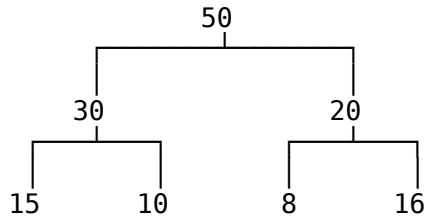
El proceso de borrar los elementos toma un tiempo de $O(n * \log_2(n))$.

En total el tiempo es de $O(2*(n * \log_2(n))) = O(n * \log_2(n))$.

▬▬▬ 11. “Heap Sort” con un Segundo Método.

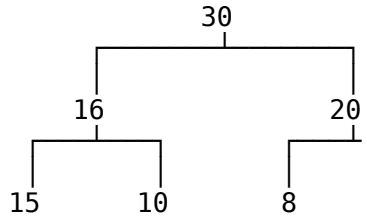
Cada vez que elimino un elemento del max heap, obtengo el elemento de mayor valor. Al borrar el elemento 50, el tamaño del vector sigue igual, y puedo poner el mayor valor al final del arreglo de la siguiente manera:

>>> Tenemos este heap:



Árbol: [50,30,20,15,10, 8,16]
1 2 3 4 5 6 7

>>> Al eliminar el 50 tenemos:

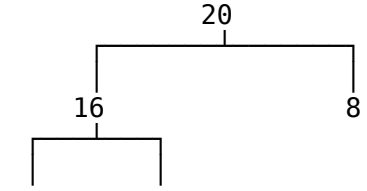


Árbol: [30,16,20,15,10, 8,()]
1 2 3 4 5 6 7

Árbol: [30,16,20,15,10, 8,50]
1 2 3 4 5 6 7

Si continuamos este proceso, sacando el mayor y poniéndolo en la última posición disponible, podemos llegar a tener todos los elementos en orden.

>>> Al eliminar el 30 tenemos:

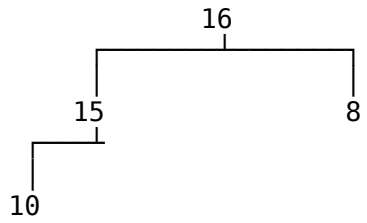


15 10

Árbol: [20,16, 8,15,10,(),()]
1 2 3 4 5 6 7

Árbol: [20,16, 8,15,10,30,50]
1 2 3 4 5 6 7

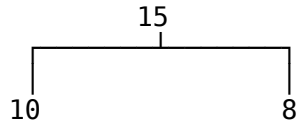
>>> Al eliminar el 20 tenemos:



Árbol: [16,15, 8,10,(),(),()]
1 2 3 4 5 6 7

Árbol: [16,15, 8,10,20,30,50]
1 2 3 4 5 6 7

>>> Al eliminar el 16 tenemos:



Árbol: [15,10, 8,(),(),(),()]
1 2 3 4 5 6 7

Árbol: [15,10, 8,16,20,30,50]
1 2 3 4 5 6 7

>>> Se continua el proceso.

Al continuar este proceso hasta dejar el “max heap” vacío, se puede observar que el resultado final será un vector con los elementos ordenados de menor a mayor.

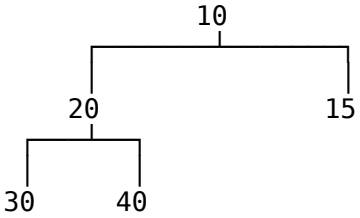
Árbol: [8,10,15,16,20,30,50]
1 2 3 4 5 6 7

▬▬▬ 12. Heapify.

Dada una lista de números, se busca crear un árbol de “heap”, utilizando el mismo arreglo. Es decir, no se necesita un arreglo adicional Este proceso se conoce como “heapify”.

>>> Entradas iniciales:

Se tiene el siguiente arreglo, el cual no es un “max heap”.



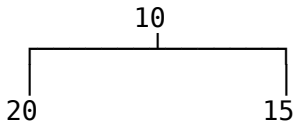
vec: [10,20,15,30,40]
1 2 3 4 5

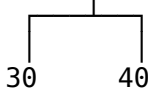
En lugar de crear una nueva estructura con cada uno de los elementos, se recorrerá el vector de derecha a izquierda.

Si el elemento no tiene hijos se sigue hacia la izquierda.

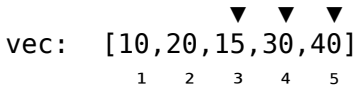
Si el elemento tiene hijos, se ajusta hacia abajo para mantener la estructura del heap.

>>> Vamos hacia el último elemento.
40 no tiene hijos es un heap
30 no tiene hijos es un heap
15 no tiene hijos es un heap



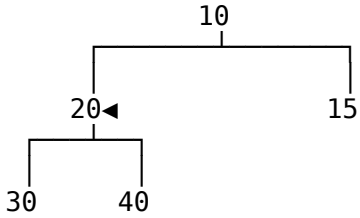


```
30     40
```

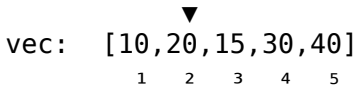


```
vec:  [10,20,15,30,40]
        1  2  3  4  5
```

>>> Se hace el primer ajuste.
    Lo enviamos hacia abajo

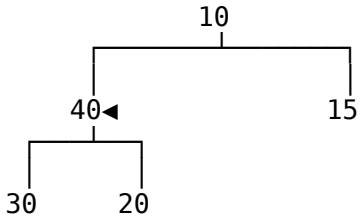


```
          10
    20◄             15
30     40
```

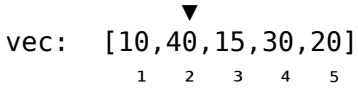


```
vec:  [10,20,15,30,40]
        1  2  3  4  5
```

Ajustamos:

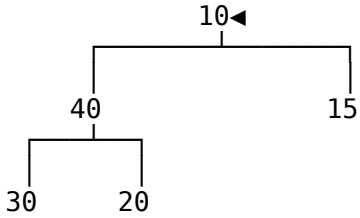


```
          10
    40◄             15
30     20
```



```
vec:  [10,40,15,30,20]
        1  2  3  4  5
```

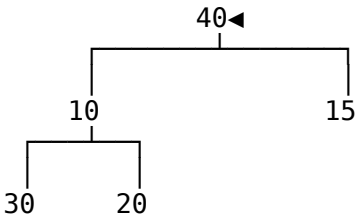
>>> Ajustamos el siguiente elemento.
    Lo enviamos hacia abajo



```
          10◄
    40              15
30     20
```

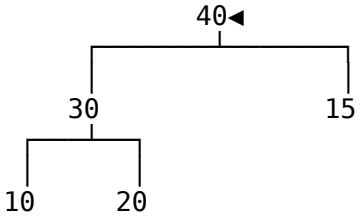
```
          ▼
vec:  [10,40,15,30,20]
        1  2  3  4  5
```

Ajustamos:




```
          ▼
vec:  [40,10,15,30,20]
        1  2  3  4  5
```

Ajustamos:

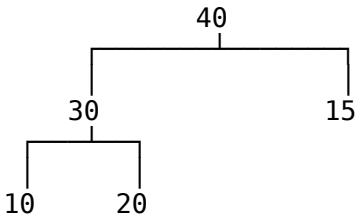



```
          ▼
vec:  [40,30,15,10,20]
        1  2  3  4  5
```

▬▬▬ 13. “Heap Sort” con Heapify.

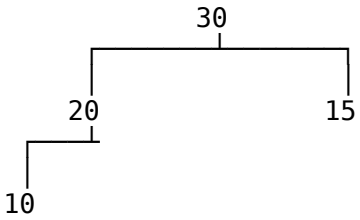
Probablemente ya conocen la siguiente parte. Una vez construido el heap, se borran los elementos uno a uno y se crea la lista ordenada.

>>> Se tiene un max heap de la forma:



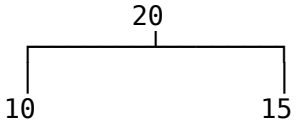
heap: [40,30,15,10,20]
1 2 3 4 5

>>> Se elimina un elemento:



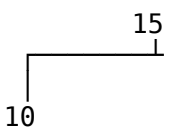
heap: [30,20,15,10,40]
1 2 3 4 5

>>> Se elimina un elemento:



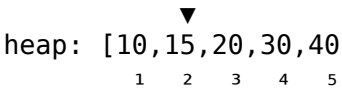
heap: [20,10,15,30,40]
1 2 3 4 5

>>> Se elimina un elemento:

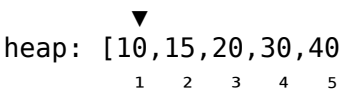


heap: [15,10,20,30,40]
1 2 3 4 5

>>> Se elimina un elemento:



>>> Se elimina un elemento:



>>> El Heap Sort.

El proceso completo del heap sort es el siguiente:

• Se recibe una lista de elementos y se construye un “max heap”. El max heap se puede construir por inserciones sucesivas o se puede construir con heapify.

• Una vez que se tiene el “max heap”, se eliminan todos los elementos del heap, guardándolos en la parte final del vector.

▬▬▬ 14. Tiempo de Ejecución.

>>> Heap Sort con Heapify.

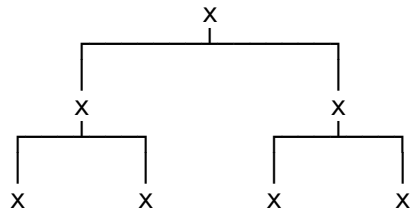
El tiempo de “heapify” es lineal O(n).

El proceso de borrar los elementos toma un tiempo de $n * \log_2 n$.

En total el tiempo es de $O(n + (n * \log_2 n)) = O(n * \log_2 n)$.

>>> Por qué Heapify es O(n)?

Hagamos unas cuantas observaciones con el siguiente árbol. Los números se han omitido por facilidad.



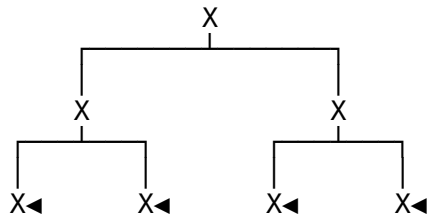
El árbol tiene 7 nodos.

La altura del árbol está dada por:

log(7) = 2.80 ~ 3 niveles.

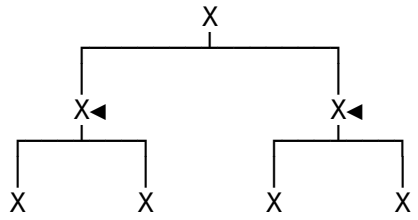
Cuando se realiza el proceso de Heapify, aproximadamente la mitad de los nodos tienen que hacer una operación para ver si tienen que moverse.

7/2 = 3.50 ~ 4



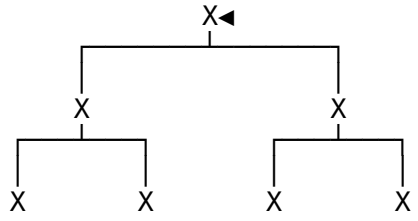
Aproximadamente, 7/4 de los nodos tienen que hacer 2 operaciones.

7/4 = 1.75 ~ 2



Y 7/8 de los nodos tienen que que hacer 3 operaciones.

7/8 = 0.875 ~ 1



Entonces el número total de movimientos hacia abajo está dado por:

$$1*\left(\frac{7}{2}\right) + 2*\left(\frac{7}{4}\right) + 3*\left(\frac{7}{8}\right)$$

Se puede expresar como:

$$1*\left(\frac{7}{2^1}\right) + 2*\left(\frac{7}{2^2}\right) + 3*\left(\frac{7}{2^3}\right)$$

Equivalentemente:

$$7*\left(\frac{1}{2^1}\right) + 7*\left(\frac{2}{2^2}\right) + 7*\left(\frac{3}{2^3}\right)$$

Supongamos que en lugar de 7 elementos, tenemos “n” elementos.
La altura del árbol estará dada por k = log(n)
Entonces, la fórmula se escribirá como:

$$n*\left(\frac{1}{2^1}\right) + n*\left(\frac{2}{2^2}\right) + n*\left(\frac{3}{2^3}\right) + \ldots + n*\left(\frac{k}{2^k}\right)$$

Que es equivalente a:

$$n * \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots + \frac{k}{2^k}\right)$$

Se puede constatar que conforme k → +inf se tiene que:

```
(%i) float(sum (k/2^k, k, 1, 100));
2
```

Por lo tanto se tiene que es equivalente a:

$$n * \left(2\right)$$

Así que el tiempo está dado por:

2*n = O(n)

▬▬▬ 15. Resumen.

• El heap es una estructura especialmente diseñada para ser útil para el ordenamiento de números.

• Existe el “min heap” y el “max heap”. En esta sección se ha descrito únicamente el proceso del “max heap”. El del “min heap” se puede deducir con facilidad de este proceso.

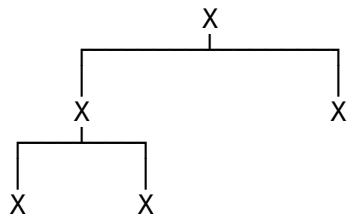
• Con una lista de datos se puede crear un “max heap” de dos formas. Ya sea por inserciones sucesivas en otro vector, y luego borrando datos del “max heap”. O utilizando el proceso de “heapify”.

▬▬▬ 16. Ejercicios.

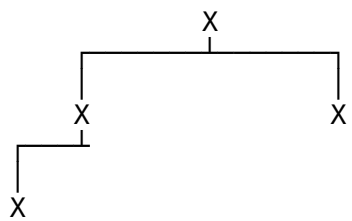
●● Ejercicio 1.

Indique cuáles de las siguiente estructuras son árboles binarios completos.

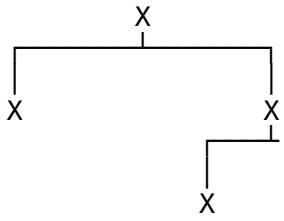
>>> Arbol A.



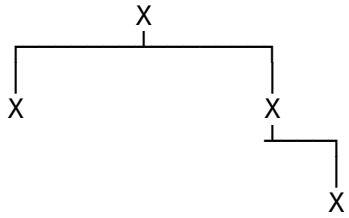
>>> Arbol B.



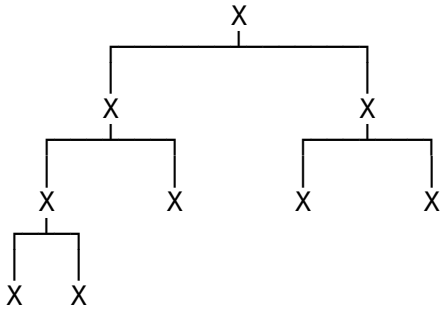
>>> Arbol C.



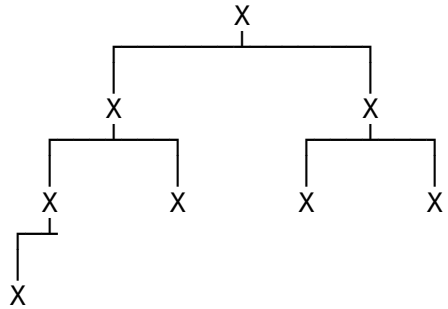
>>> Arbol D.



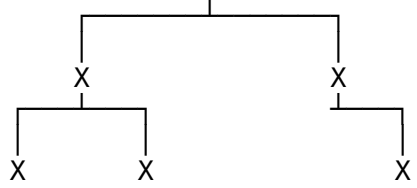
>>> Arbol E.



>>> Arbol F



>>> Arbol G.



●● Ejercicio 2.

Con el siguiente vector utilice el algoritmo de inserciones sucesivas para crear un max heap.

[10,20,25,12,40,15,18]

●● Ejercicio 3.

Con el siguiente vector utilice el algoritmo de heapify para crear un max heap.

[10,20,25,12,40,15,18]

●● Ejercicio 4.

Utilice paso a paso el algoritmo de heap sort con la siguiente lista:

[8, 4, 7, 1, 3, 5]

●● Ejercicio 5.

Utilice paso a paso el algoritmo de heap sort con la siguiente lista:

[70,50,40,45,35,39,16,10, 9]