CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Heapsort

Objetivos

– Heap (montículo) de tipo max y min

– Heapsort

– Estructura de datos: cola de prioridades

Heapsort

Heapsort es un algoritmo de ordenamiento **basado en comparaciones** que corre en tiempo $\Theta(n \log n)$ para arreglos de n elementos

A diferencia de mergesort, heapsort ordena los elementos "in place"

Heapsort debe su nombre a la utilización de una estructura de datos llamada **heap binario** la cual implementa una **cola de prioridades**

Heap binario

Un heap binario es un arreglo de elementos el cual se interpreta como un árbol binario **casi lleno**: el árbol tiene todos sus niveles llenos excepto, tal vez, el último nivel (llenado de izquierda a derecha)

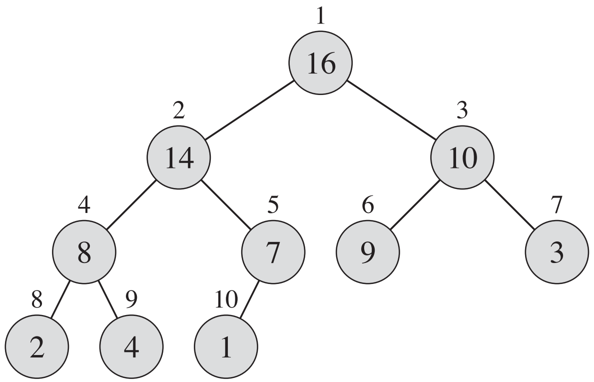


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Adicionalmente el heap debe cumplir la **propiedad de (max-)heap:**

Heap binario

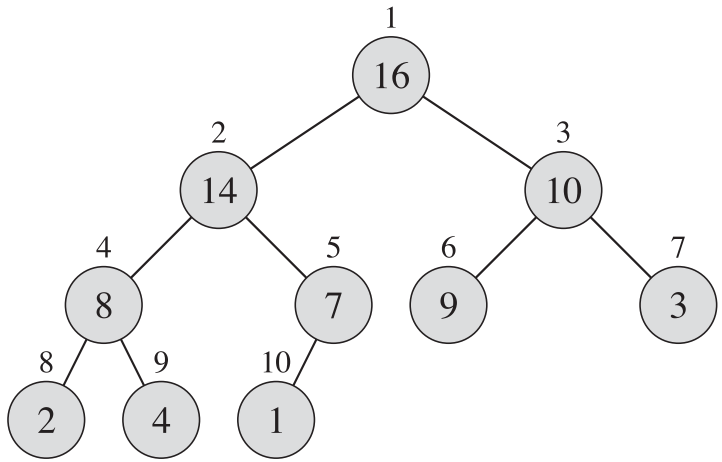


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Representación de heaps binarios

Un heap binario se representa con un arreglo A:

– A.*length* denota el tamaño o dimensión del arreglo

– A.*heap-size* es el número de elementos en A que pertenecen al heap; i.e. el heap está formado por $A[1 \ldots A.\textit{heap-size}]$

Invariante: $0 \leq A.\textit{heap-size} \leq A.\textit{length}$

– Funciones sobre indices:

$$\begin{aligned} \text{Parent}(i) &= \lfloor i/2 \rfloor \\ \text{Left}(i) &= 2i \\ \text{Right}(i) &= 2i+1 \end{aligned}$$

– La **altura** de un nodo n es la longitud del camino (# aristas) más largo de n a una hoja. La **altura del heap** es la altura de la raíz. La altura de un heap de $n \geq 1$ elementos es $\lfloor \log_2 n \rfloor$ (ejercicio)

Ejemplo de heap binario

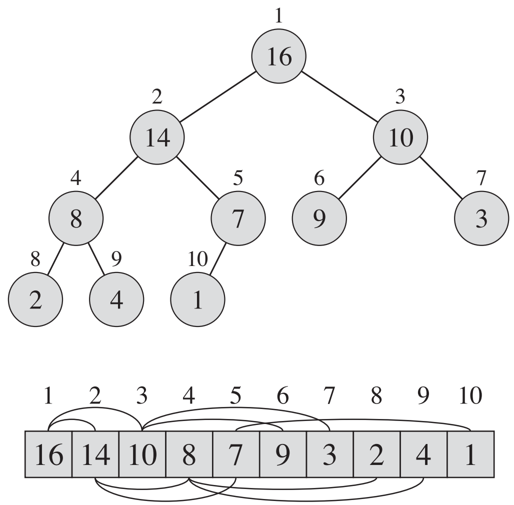


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Operaciones sobre heaps

Max-Heapify: función para restaurar la propiedad de max-heap (corre en tiempo $O(\log n)$)

Build-Max-Heap: construye un heap binario a partir de un arreglo en tiempo $O(n)$

Heapsort: ordena un arreglo de n elementos “in-place” en tiempo $O(n \log n)$

Max-Heapify: Restaurar la propiedad de heap

Max-Heapify se llama sobre un nodo i del heap

Max-Heapify asume:

– los subárboles Left(i) y Right(i) satisfacen la propiedad de heap

– el subárbol i puede no satisfacer la propiedad de heap

Max-Heapify corre en tiempo $O(\log n)$ y al finalizar, el subárbol con raíz i es un heap y contiene los mismos elementos que existían antes de la llamada

Max-Heapify: Restaurar la propiedad de heap

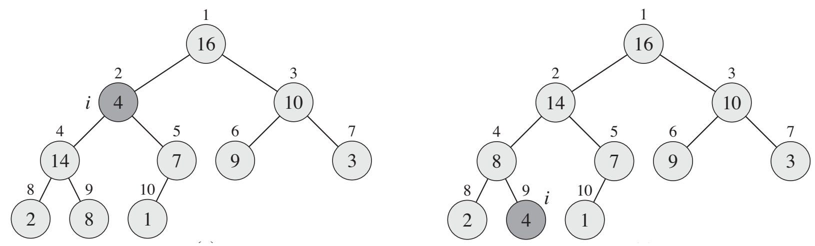


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Resultado de aplicar Max-Heapify en el nodo i en el árbol del lado izquierdo

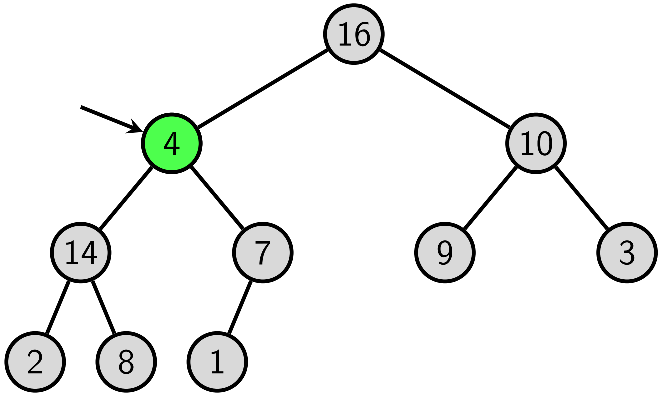
Max-Heapify: Restaurar la propiedad de heap

Input: arreglo $A[1 \ldots n]$ con n elementos e índice i. Asume que los subárboles Left(i) y Right(i) son heaps

Output: arreglo A con los elementos en subárbol i reordenados formando un heap

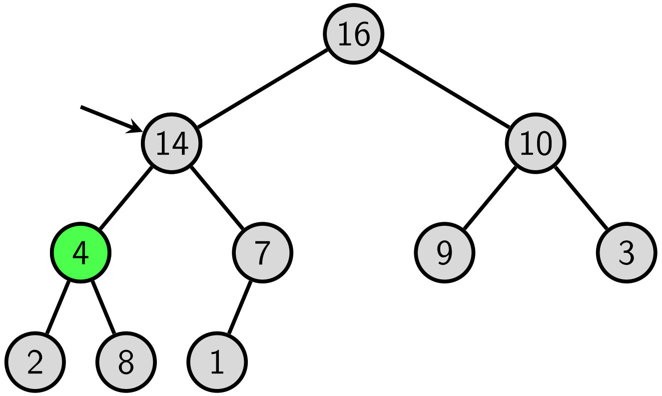
```
Max-Heapify(array A, int i)
    l = Left(i)
    r = Right(i)

    if l <= A.heap-size && A[l] > A[i] then
        largest = l
    else
        largest = i

    if r <= A.heap-size && A[r] > A[largest] then
        largest = r

    if i != largest then
        Intercambiar A[i] con A[largest]
        Max-Heapify(A, largest)
```

Max-Heapify: Restaurar la propiedad de heap



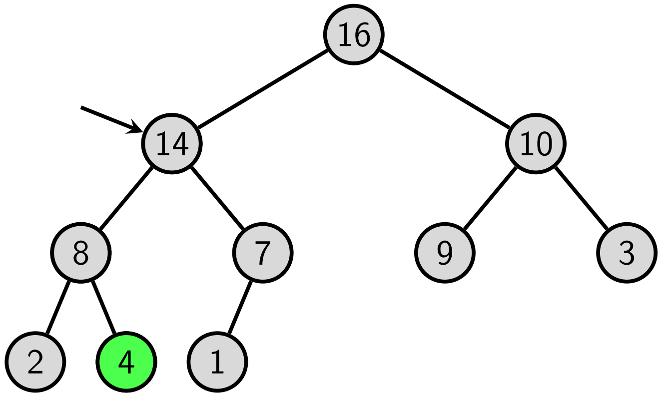
16	4	10	14	7	9	3	2	8	1

Max-Heapify: Restaurar la propiedad de heap



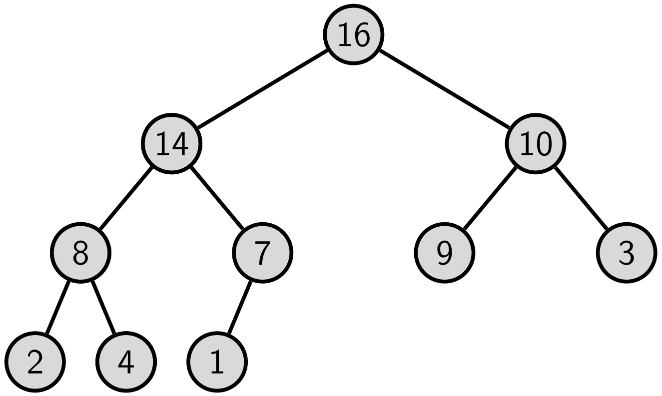
16	14	10	4	7	9	3	2	8	1

Max-Heapify: Restaurar la propiedad de heap



16	14	10	8	7	9	3	2	4	1

Max-Heapify: Restaurar la propiedad de heap



16	14	10	8	7	9	3	2	4	1

Análisis de Max-Heapify

Dos análisis:

1. `Max-Heapify`(A, i) toma tiempo $O(h)$ donde h es la altura del nodo i. Como $h = O(\log n)$, entonces `Max-Heapify` toma tiempo $O(\log n)$ donde $n = A.$*heap-size*

2. Denotemos por $T(n)$ el tiempo de `Max-Heapify`(A, i) donde n es el número de nodos en el subárbol i

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

(en el peor caso la recursión se hace sobre un subárbol con $\leq \lfloor 2n/3 \rfloor$ nodos)

Por el 2do caso del Teorema Maestro, $T(n) = O(\log n)$

Build-Max-Heap: Construir un heap

Podemos utilizar `Max-Heapify` para construir un heap

La idea es construir el heap de forma iterativa procediendo desde las hojas hasta la raíz (procedimiento **bottom-up**):

– construimos heaps para cada una de las hojas (nodos en el último nivel del árbol)

– construimos heaps para cada uno de los nodos en el penúltimo nivel

– continuamos nivel por nivel hasta llegar a la raíz del árbol

Build-Max-Heap: Construir un heap

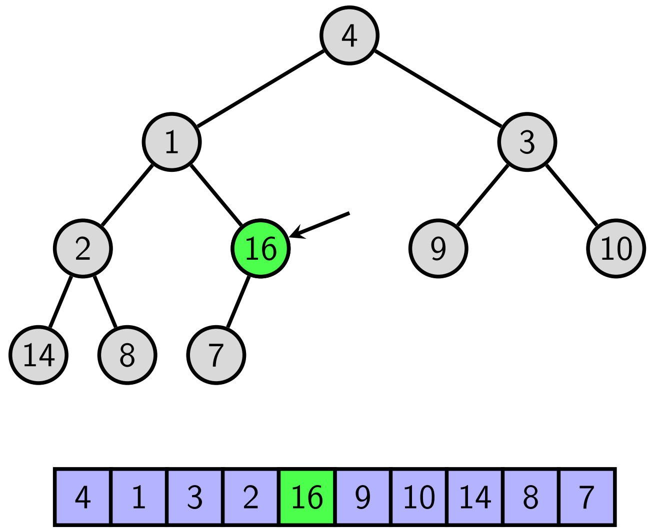
Por definición, si n es una hoja del árbol, n no tiene hijos y n por si sólo es un heap de tamaño 1. Por lo tanto, el procedimiento no necesita considerar las hojas

Input: arreglo $A[1 \ldots n]$ con n elementos

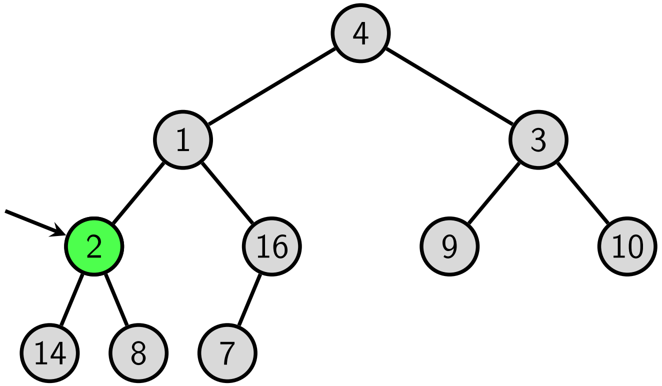
Output: arreglo A con elementos repordenados formando un heap

```
Build-Max-Heap(array A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ to 1        % sólo considera nodos internos
        Max-Heapify(A, i)
```


Build-Max-Heap: Construir un heap

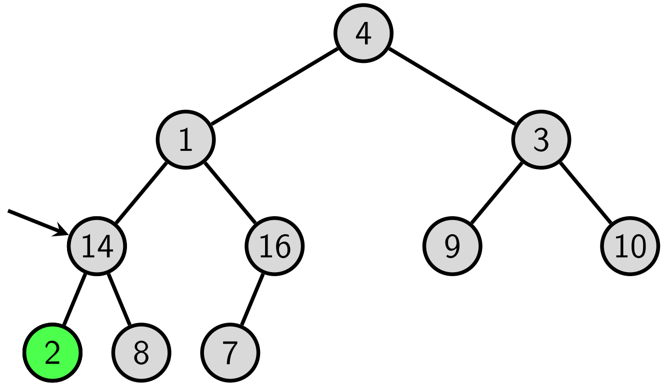


Build-Max-Heap: Construir un heap



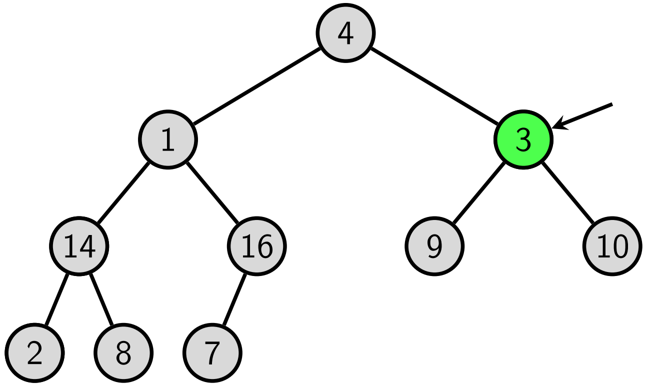
4	1	3	2	16	9	10	14	8	7

Build-Max-Heap: Construir un heap



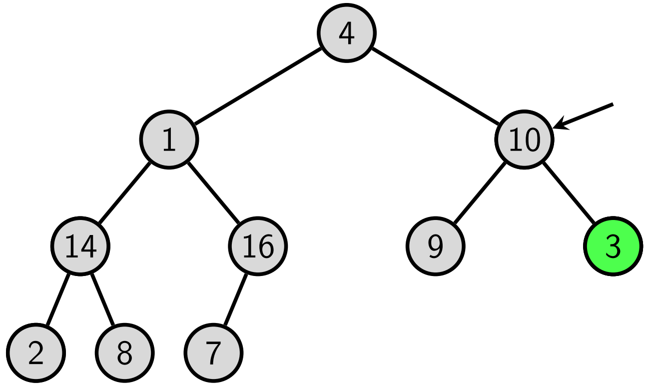
4	1	3	14	16	9	10	2	8	7

Build-Max-Heap: Construir un heap



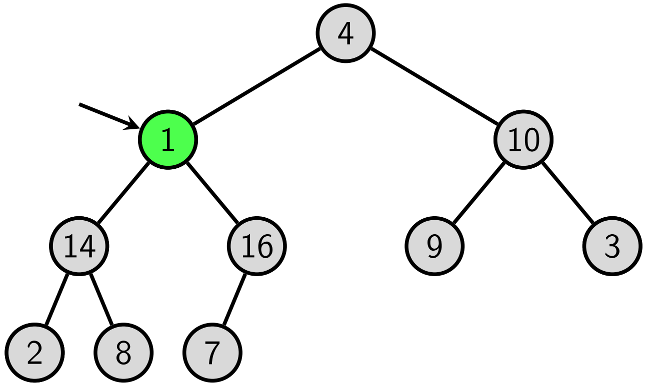
4	1	3	14	16	9	10	2	8	7

Build-Max-Heap: Construir un heap



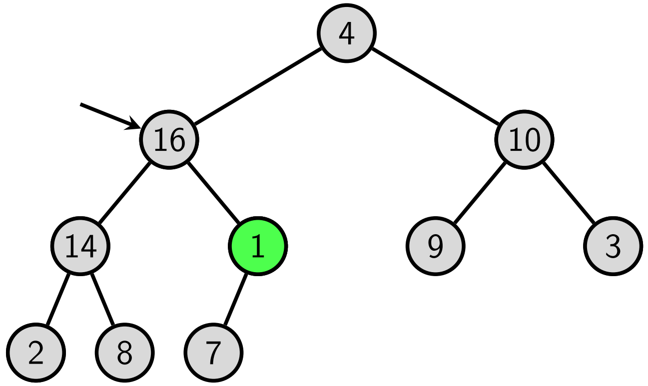
4	1	10	14	16	9	3	2	8	7

Build-Max-Heap: Construir un heap



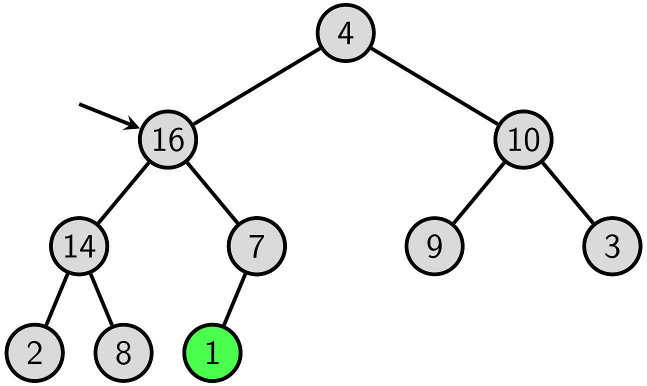
4	1	10	14	16	9	3	2	8	7

Build-Max-Heap: Construir un heap



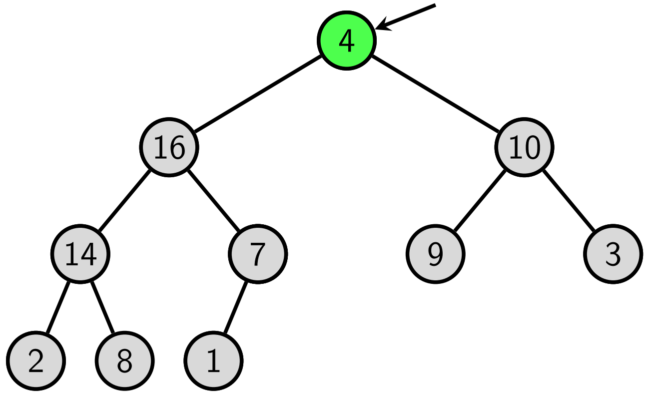
4	16	10	14	1	9	3	2	8	7

Build-Max-Heap: Construir un heap



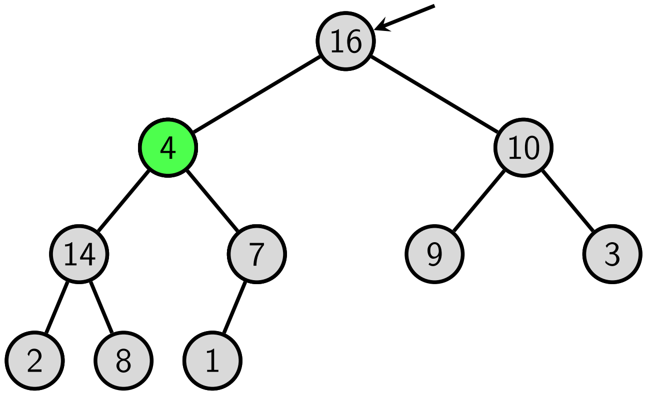
4	16	10	14	7	9	3	2	8	1

Build-Max-Heap: Construir un heap



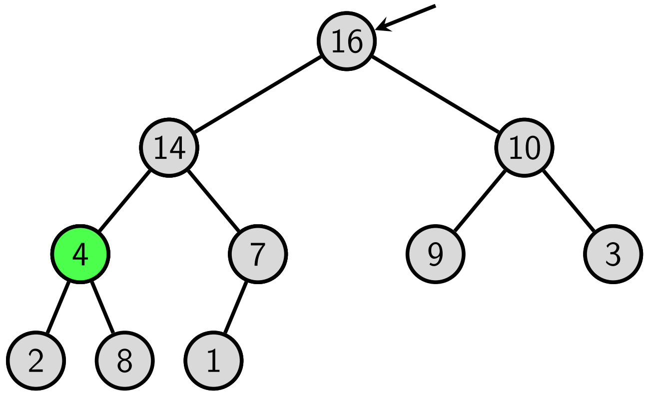
4	16	10	14	7	9	3	2	8	1

Build-Max-Heap: Construir un heap



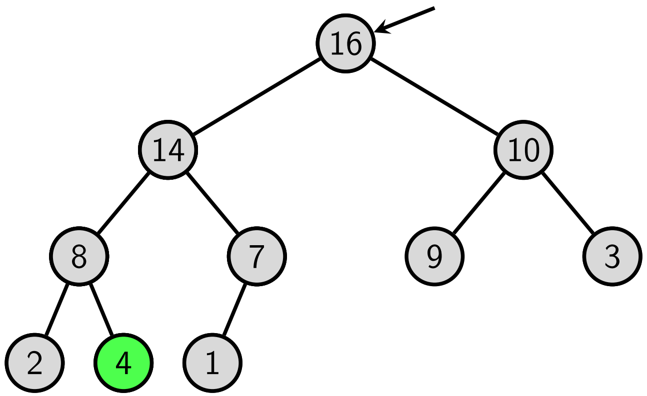
16	4	10	14	7	9	3	2	8	1

Build-Max-Heap: Construir un heap



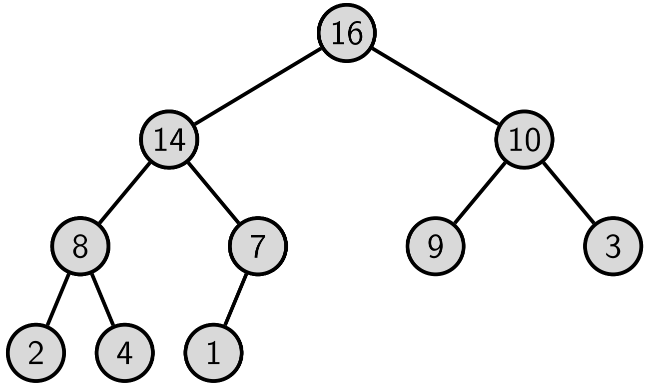
16	14	10	4	7	9	3	2	8	1

Build-Max-Heap: Construir un heap



16	14	10	8	7	9	3	2	4	1

Build-Max-Heap: Construir un heap



16	14	10	8	7	9	3	2	4	1

Correctitud de Build-Max-Heap

Para mostrar la correctitud de `Build-Max-Heap`, utilizamos el siguiente **invariante de lazo**:

Al comienzo de cada iteración, los nodos $i+1, i+2, \ldots, n$ son raíces de max-heaps

Input: arreglo $A[1 \ldots n]$ con n elementos

Output: arreglo A con elementos repordenados formando un heap

```
Build-Max-Heap(array A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ to 1              % sólo considera nodos internos
        Max-Heapify(A, i)
```


Correctitud de Build-Max-Heap

Para mostrar la correctitud de `Build-Max-Heap`, utilizamos el siguiente **invariante de lazo**:

Al comienzo de cada iteración, los nodos $i+1, i+2, \ldots, n$ son raíces de max-heaps

Para establecer la correctitud del invariante, debemos mostrar:

– el invariante es cierto para la primera iteración (**inicialización**)

– el invariante se mantiene después de cada iteración (**mantenimiento**)

– al terminar el lazo, el invariante es útil para mostrar la correctitud del algoritmo (**terminación**)

Correctitud de Build-Max-Heap

Para mostrar la correctitud de `Build-Max-Heap`, utilizamos el siguiente **invariante de lazo**:

Al comienzo de cada iteración, los nodos $i+1, i+2, \ldots, n$ son raíces de max-heaps

Inicialización:

Previo a la primera iteración del lazo, $i = \lfloor n/2 \rfloor$

Cada nodo $i+1, i+2, \ldots, n$ es una hoja del árbol (ejercicio) y por lo tanto un heap de tamaño 1

Correctitud de Build-Max-Heap

Para mostrar la correctitud de `Build-Max-Heap`, utilizamos el siguiente **invariante de lazo**:

Al comienzo de cada iteración, los nodos $i+1, i+2, \ldots, n$ son raíces de max-heaps

Mantenimiento:

Considere una iteración i. Los hijos del nodo i tienen índices mayor a i

Por HI, los hijos de i son heaps. Está es la condición requerida por `Max-Heapify` para garantizar que al terminar el nodo i es raíz de un heap

Los índices $j > i$ que no pertenecen al subárbol i no son afectados y seguirán siendo raíces de heaps al terminar `Max-Heapify(A,i)`

Por lo tanto, al terminar la iteración i, los nodos $i, i+1, \ldots, n$ son heaps

Correctitud de Build-Max-Heap

Para mostrar la correctitud de `Build-Max-Heap`, utilizamos el siguiente **invariante de lazo**:

Al comienzo de cada iteración, los nodos $i+1, i+2, \ldots, n$ son raíces de max-heaps

Terminación:

Al termina el lazo, $i = 0$ y el invariante establece que el subárbol cuya raíz es el nodo 1 es un heap. Por lo tanto, el algoritmo `Build-Max-Heap` es correcto

Análisis de Build-Max-Heap

- `Build-Max-Heap(A)` llama a `Max-Heapify` $\lfloor n/2 \rfloor = \Theta(n)$ veces donde $n = A.$*length*
- `Max-Heapify(A,i)` toma tiempo $O(h)$ donde h es altura de i
- Para todo nodo en el árbol, su altura h está acotada por $O(\log n)$
- Entonces, `Build-Max-Heap(A)` se ejecuta en tiempo $O(n \log n)$

Análisis es **correcto pero no ajustado**. Con un **análisis mejor** mostraremos que `Build-Max-Heap(A)` toma **tiempo lineal**

Análisis de Build-Max-Heap

- `Max-Heapify(A,i)` se ejecuta en tiempo $O(h)$ donde h es la altura del nodo i
- Las altura máxima del árbol es $\lfloor \log n \rfloor$ y la mínima es 0 (hojas)
- Existen a lo sumo $\lceil n/2^{h+1} \rceil$ nodos de altura h en el árbol (ejercicio)

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

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

Heapsort

Veamos como usar `Build-Max-Heap` y `Max-Heapify` para construir un algoritmo de ordenamiento que corre en tiempo $O(n \log n)$

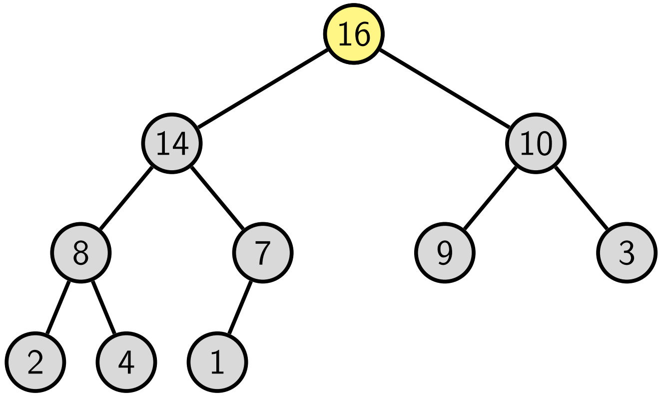
Dado un arreglo A, lo primero es construir un heap con los elementos de A en tiempo $O(n)$ usando `Build-Max-Heap(A)`

Al ser un heap, el elemento $A[1]$ es un mayor elemento en A y lo podemos colocar en la última posición en el orden final

Heapsort intercambia $A[1]$ con el último elemento del heap y restablece la propiedad de heap que puede perderse en el intercambio

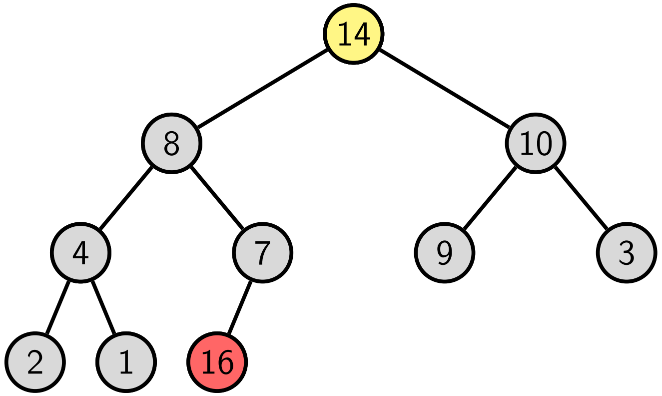
El algoritmo se repite hasta vaciar el heap

Heapsort



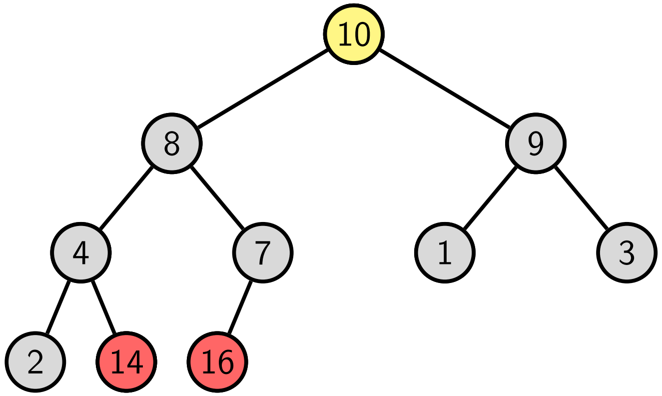
16	14	10	8	7	9	3	2	4	1

Heapsort



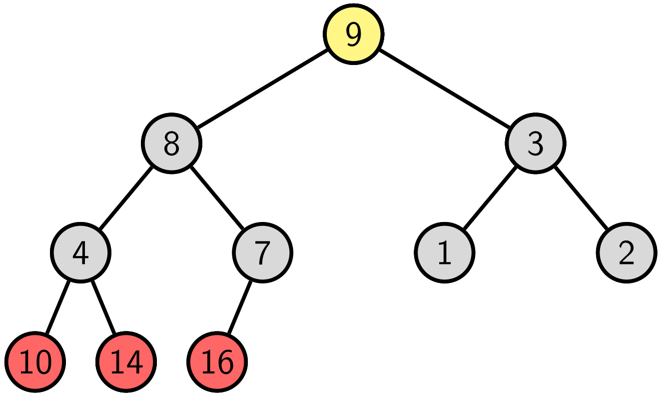
14	8	10	4	7	9	3	2	1	16

Heapsort



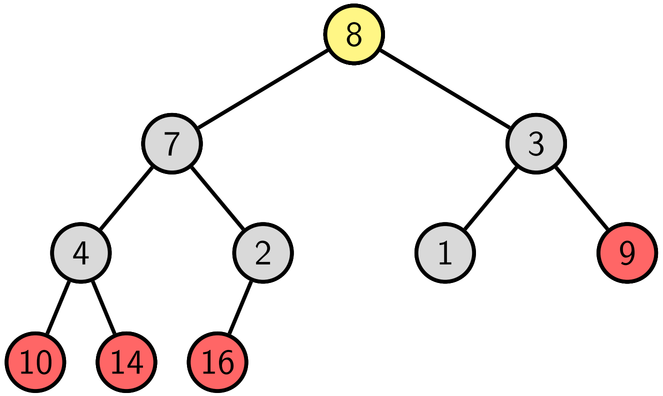
10	8	9	4	7	1	3	2	14	16

Heapsort



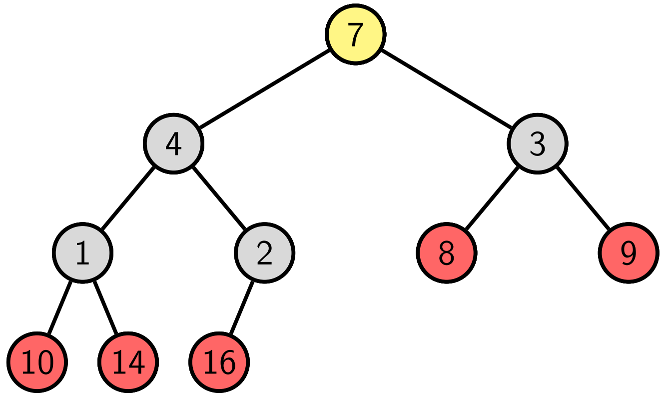
9	8	3	4	7	1	2	10	14	16

Heapsort



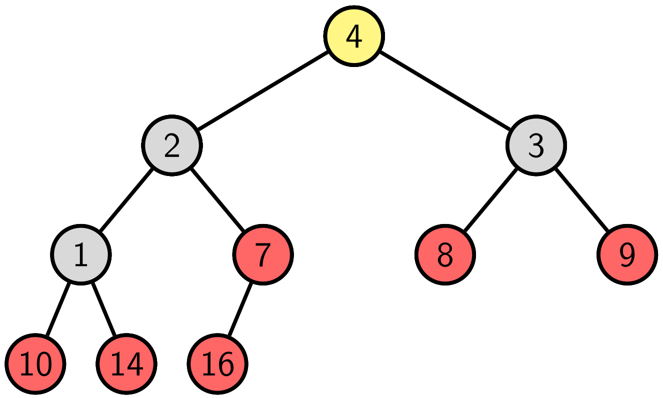
8	7	3	4	2	1	9	10	14	16

Heapsort



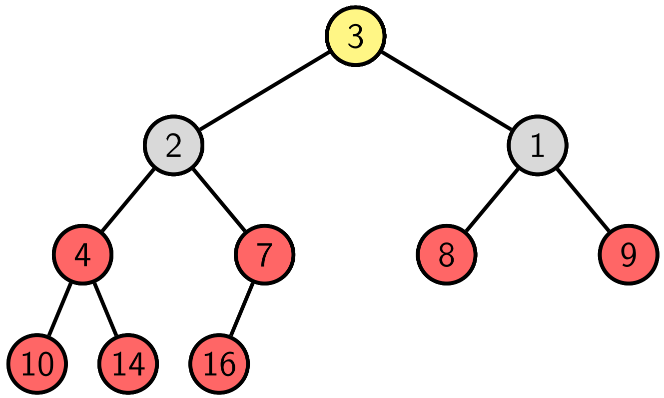
7	4	3	1	2	8	9	10	14	16

Heapsort



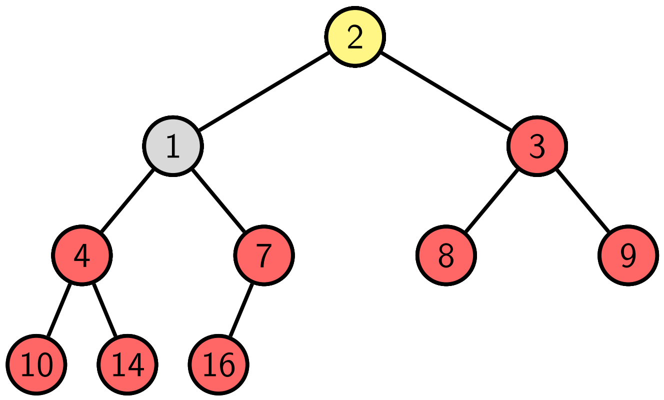
4	2	3	1	7	8	9	10	14	16

Heapsort



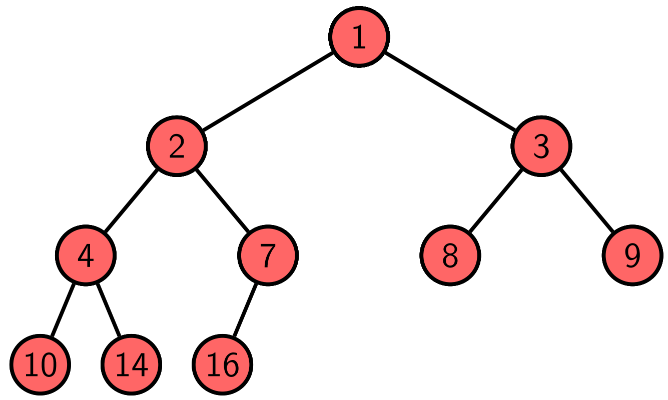
3	2	1	4	7	8	9	10	14	16

Heapsort



2	1	3	4	7	8	9	10	14	16

Heapsort



1	2	3	4	7	8	9	10	14	16

Heapsort

Input: arreglo $A[1 \ldots n]$ con n elementos

Output: arreglo A con elementos reordenados de menor a mayor

```
Heapsort(array A)
    Build-Max-Heap(A)
    for i = A.length to 2
        Intercambiar A[1] con A[i]
        A.heap-size = A.heap-size - 1       % reducir tamaño del heap
        Max-Heapify(A, 1)
```

Heapsort toma tiempo $O(n \log n)$ en el peor caso: $O(n)$ para `Build-Max-Heap(A)` + $O(n \log n)$ para el lazo

Cola de prioridades

Una cola de prioridades es una **estructura de datos** que mantiene un conjunto S de elementos

Cada elemento $x \in S$ tiene asociado una **prioridad o clave**

Una cola de prioridades **tipo max** soporta las operaciones:

- `Insert(x,S)`: inserta el elemento x en el conjunto S
- `Maximum(S)`: retorna un elemento con máxima prioridad en S
- `Extract-Max(S)`: remueve y retorna un elemento con máxima prioridad en S
- `Increase-Key(S,x,k)`: incrementa la prioridad del elemento x hasta el valor k (se asume que la prioridad de x es menor a k)

Implementacion de cola de prioridades con un heap

```
Heap-Maximum(array A)
    return A[1]

Heap-Extract-Max(array A)
    if A.heap-size < 1 then
        error "no existen elementos en la cola"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    Max-Heapify(A, 1)
    return max
```

`Heap-Maximum(A)` se ejecuta en tiempo constante y `Heap-Extract-Max(A)` en tiempo $O(\log n)$ donde $n = A.\textit{heap-size}$

Implementacion de cola de prioridades con un heap

```
Heap-Increase-Key(array A, int i, int key)
    if key < A[i] then
        error "la nueva clave es menor a la existente"
    A[i] = key
    while i > 1 && A[Parent(i)] < A[i]
        Intercambiar A[i] con A[Parent(i)]
        i = Parent(i)

Heap-Insert(array A, int key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = -∞
    Heap-Increase-Key(A, A.heap-size, key)
```

`Heap-Increase-Key(A,i,key)` y `Heap-Insert(A,key)` se ejecutan en tiempo $O(\log n)$ donde $n = A.\textit{heap-size}$

© 2017 Blai Bonet

Resumen

- Los heaps son estructuras de datos fundamentales en computación
- Operaciones básicas: `Max-Heapify` y `Build-Max-Heap`
- Aún cuando `Build-Max-Heap` toma tiempo $O(n)$, si tomará tiempo $O(n \log n)$ también lo podríamos usar para un algoritmo de heapsort con garantía $O(n \log n)$
- Estructura de datos: cola de prioridad

© 2017 Blai Bonet

Ejercicios (1 de 3)

1. (6.1-1) ¿Cuál es el número mínimo y máximo de elementos que puede tener un heap de altura h?
2. (6.1-2) Muestre que un heap con n elementos tiene altura $\lfloor \log n \rfloor$
3. (6.2-5) Escriba una versión iterativa de `Max-Heapify`
4. Justifique la recurrencia $T(n) \leq T(2n/3) + \Theta(1)$ para el tiempo de corrida de `Max-Heapify`(A,i) donde n es el número de nodos en el subárbol i
5. Muestre que en un heap con n elementos, todo elemento en posición $i > \lfloor n/2 \rfloor$ es una hoja del heap

© 2017 Blai Bonet

Ejercicios (2 de 3)

6. (6.3-2) ¿Por qué el lazo en `Build-Max-Heapify` decrementa la variable de inducción i desde $\lfloor A.\textit{heap-size} \rfloor$ hasta 1 en lugar de incrementarla desde 1 hasta $\lfloor A.\textit{heap-size} \rfloor$?

7. (6.3-3) Muestre que el número de nodos de altura h es a lo sumo $\lceil n/2^{h+1} \rceil$

8. (6.4-3) ¿Cuál es el tiempo de corrida de `Heapsort` cuando el arreglo A de longitud n ya se encuentra ordenado y cuando se encuentra ordenado de forma decreciente?

9. Modifique `Heapsort` para que ordene los $r - p + 1$ elementos en el arreglo A en las posiciones $A[p \ldots r]$. Su algoritmo debe tener la firma `Heapsort(array A, int p, int r)`

10. (6.4-4) De un ejemplo de un arreglo A de tamaño n para el cual `Heapsort` toma tiempo $\Omega(n \log n)$

Ejercicios (3 de 3)

11. (6.5-8) La operación `Heap-Delete(A,i)` elimina el elemento i del heap A. De una implementación de `Heap-Delete` que corra en tiempo $O(\log n)$ donde n es el tamaño del heap

12. (6.5-9) Considere k colas $Q_1, Q_2, \ldots, Q_k$ de elementos ordenados. Para cada cola $Q = Q_i$, tenemos la operación `Dequeue(Q)` que elimina el primero de la cola Q y la operación `Empty(Q)` que retorna un booleano indicando si la cola Q está vacía

    Diseñe un algoritmo que dadas las k colas genera un arreglo ordenado de tamaño n con todos los elementos en las colas, donde n es el número total de elementos en todas las k colas. Su algoritmo debe correr en tiempo $O(n \log k)$

13. Implemente una operacion `Max-Decrease-Key(A,i,k)` que decrementa la clave del elemento i al valor k. Asume que la clave de i es mayor a k. Su algoritmo debe correr en tiempo $O(\log n)$.