

CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Heapsort

© 2016 Blai Bonet

Objetivos

- Heap (montículo) de tipo max y min
- Heapsort
- Estructura de datos: cola de prioridades

© 2016 Blai Bonet

Heapsort

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

© 2016 Blai Bonet

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)

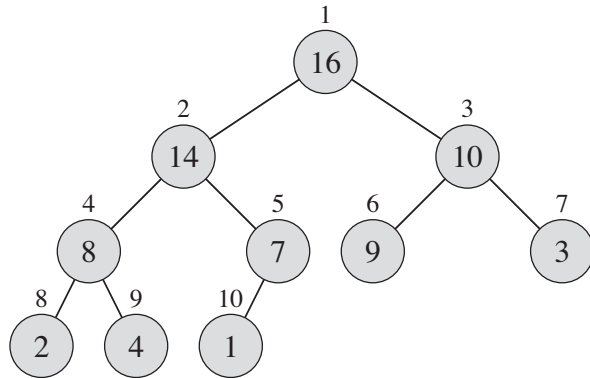


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

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

Heap binario

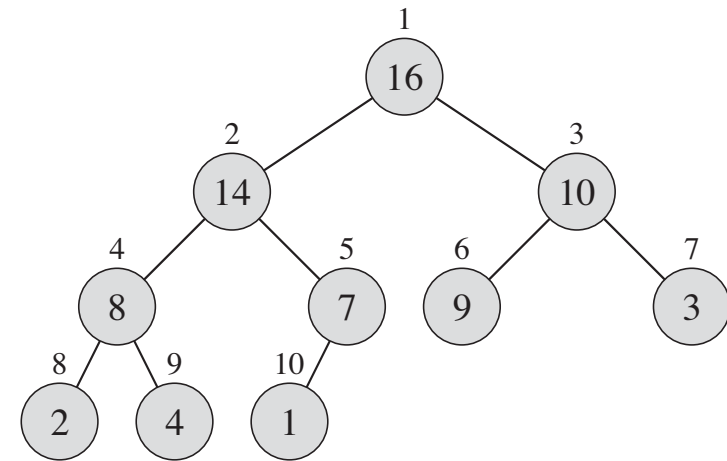


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

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 $A[1 \dots A.heap-size]$

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

- Funciones sobre índices:

$$\text{Parent}(i) = \lfloor i/2 \rfloor$$

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2i + 1$$

- 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)

© 2016 Blai Bonet

Ejemplo de heap binario

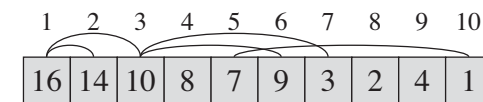
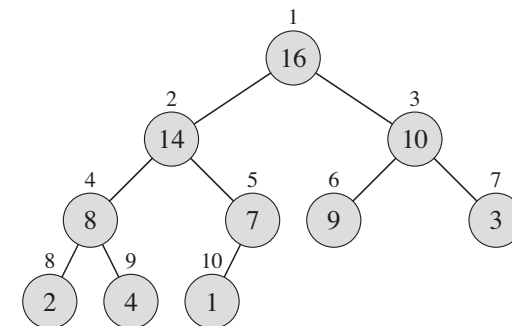


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Operaciones sobre heaps

Max-Heapify: función clave para mantener 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

Max-Heapify se llama sobre un nodo i del heap

Max-Heapify asume:

- los subárboles $\text{Left}(i)$ y $\text{Right}(i)$ 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

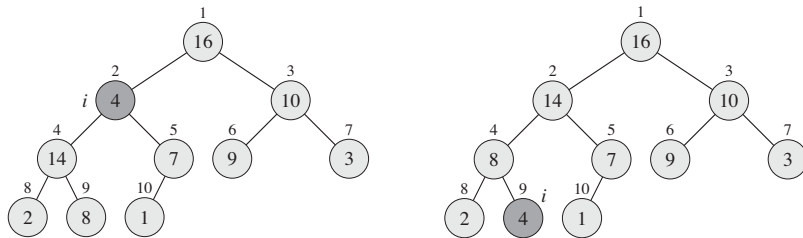


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

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

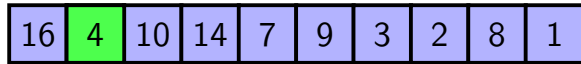
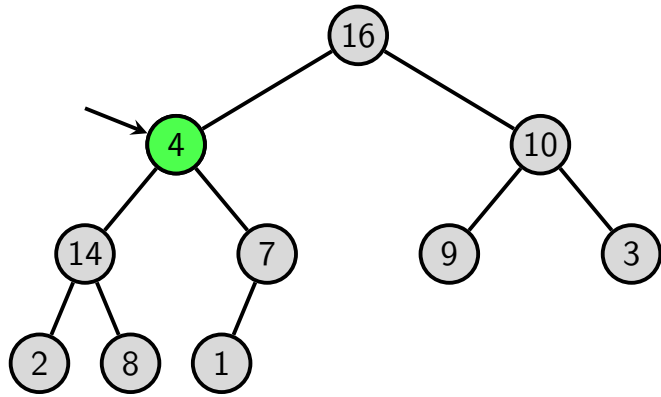
Max-Heapify

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

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

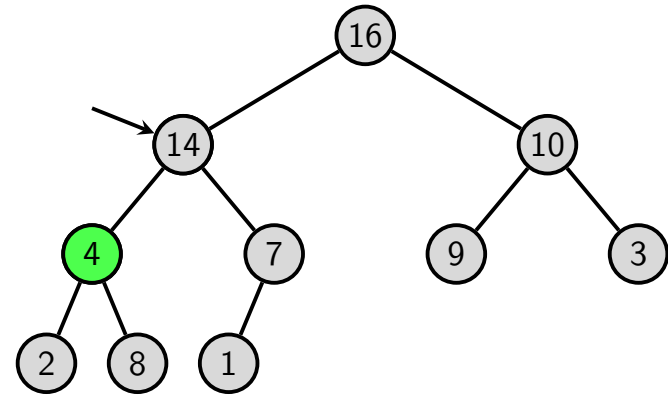
```
1 Max-Heapify(array A, int i)
2   l = Left(i)
3   r = Right(i)
4
5   if l <= A.heap-size && A[l] > A[i] then
6     largest = l
7   else
8     largest = i
9
10  if r <= A.heap-size && A[r] > A[largest] then
11    largest = r
12
13  if i != largest then
14    Intercambiar A[i] con A[largest]
15    Max-Heapify(A, largest)
```

Max-Heapify



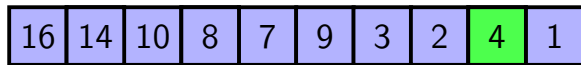
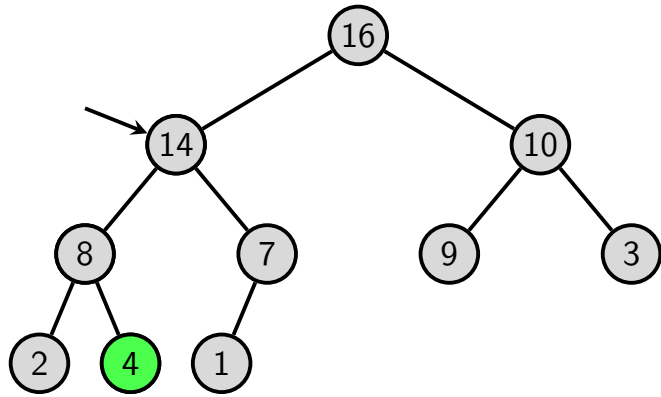
© 2016 Blai Bonet

Max-Heapify



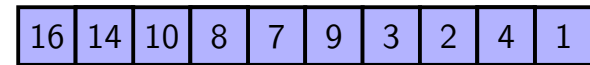
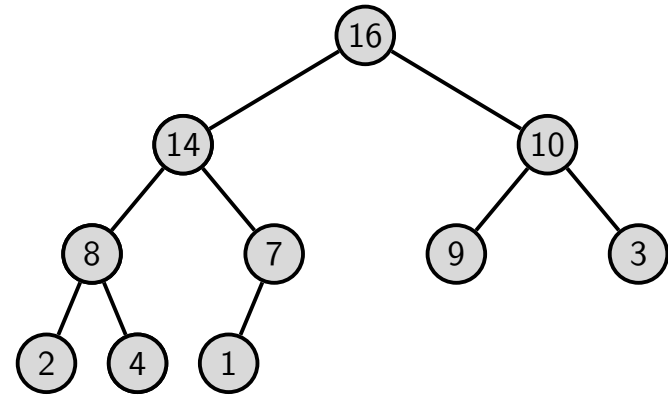
© 2016 Blai Bonet

Max-Heapify



© 2016 Blai Bonet

Max-Heapify



© 2016 Blai Bonet

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 $n = A.\text{heap-size}$

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

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

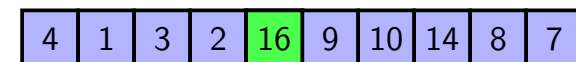
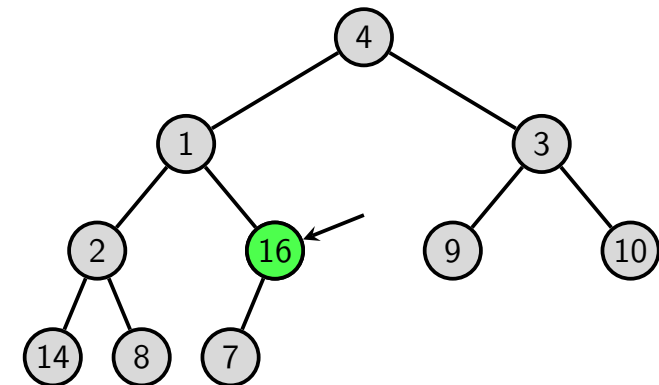
Por definición, si n es una hoja del árbol, n no tiene hijos y n por sí sólo es un heap de tamaño 1. Por lo tanto, el procedimiento bottom-up no necesita considerar las hojas

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

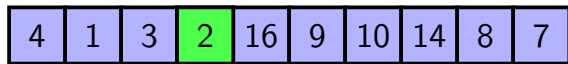
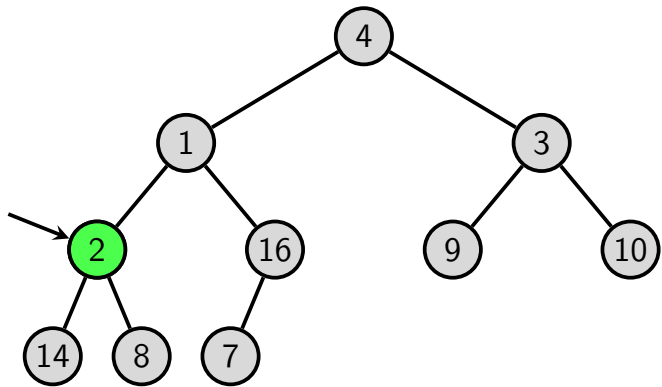
Output: arreglo A con elementos reordenados formando un heap

```
1 Build-Max-Heap(array A)
2   A.heap-size = A.length
3   for i =  $\lfloor A.length/2 \rfloor$  to 1           % sólo considera nodos internos
4     Max-Heapify(A, i)
```

Build-Max-Heap

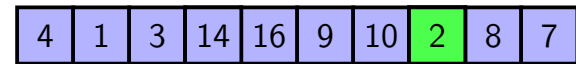
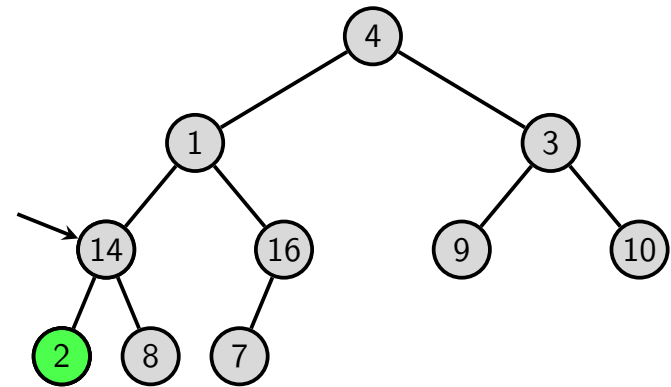


Build-Max-Heap



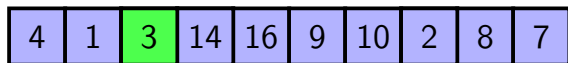
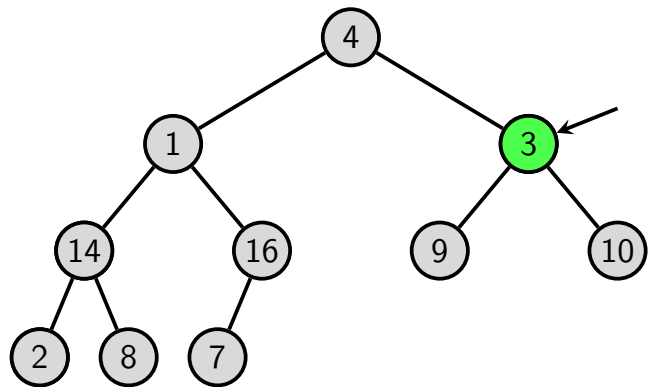
© 2016 Blai Bonet

Build-Max-Heap



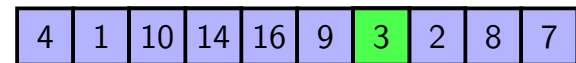
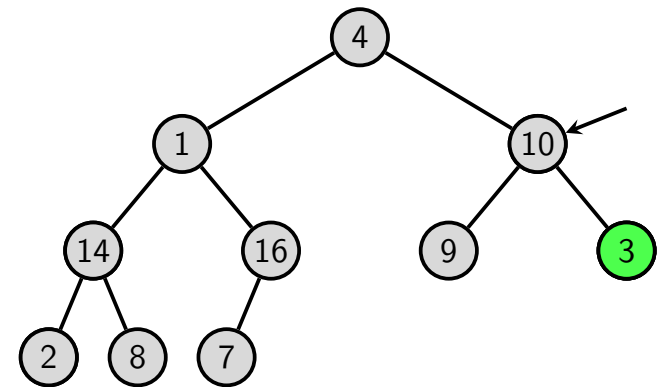
© 2016 Blai Bonet

Build-Max-Heap



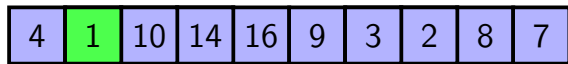
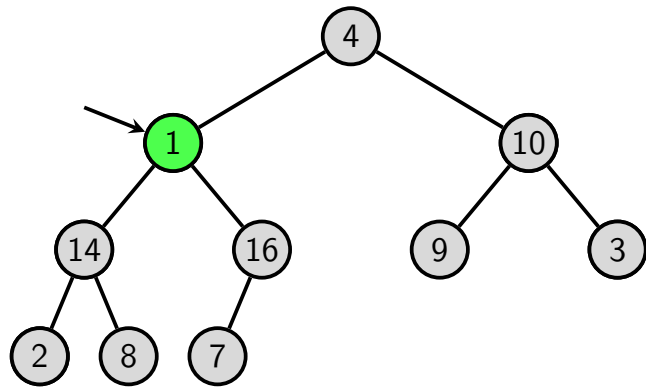
© 2016 Blai Bonet

Build-Max-Heap



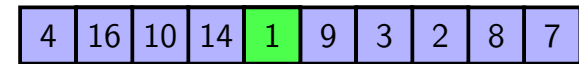
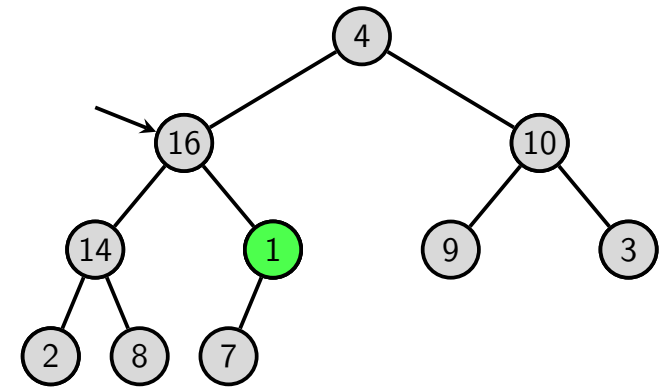
© 2016 Blai Bonet

Build-Max-Heap



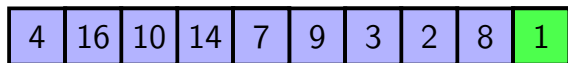
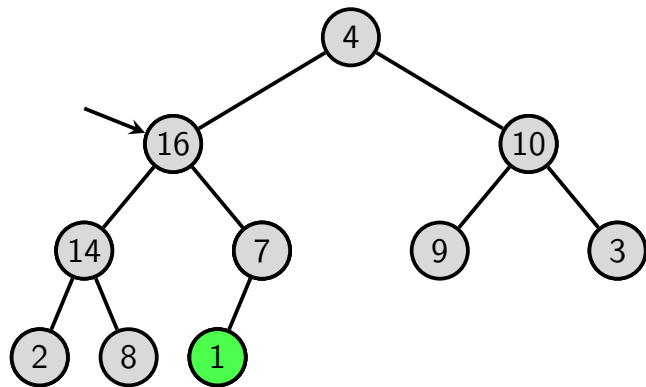
© 2016 Blai Bonet

Build-Max-Heap



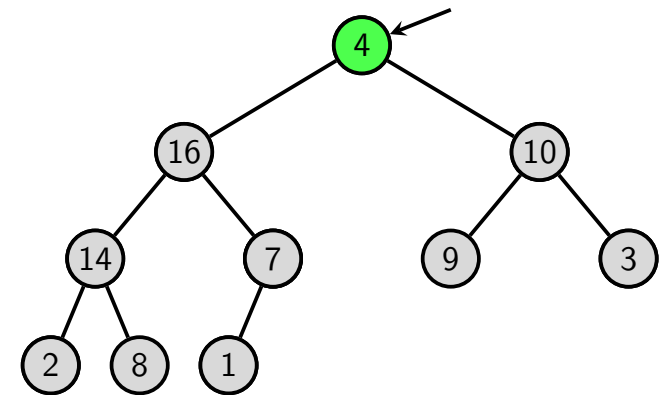
© 2016 Blai Bonet

Build-Max-Heap



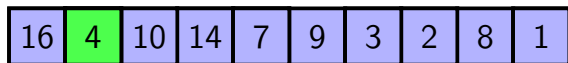
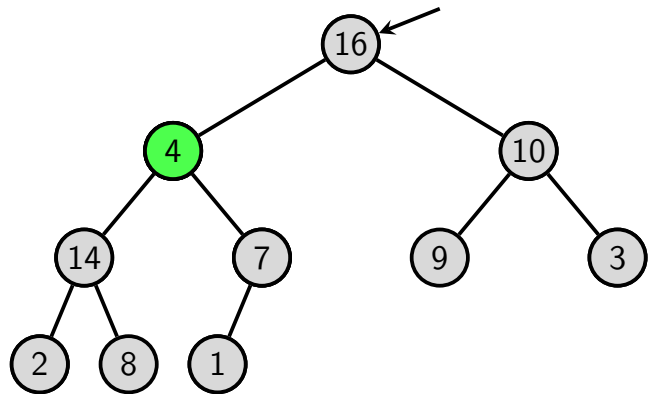
© 2016 Blai Bonet

Build-Max-Heap



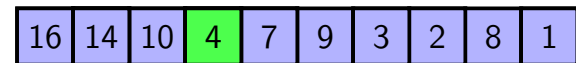
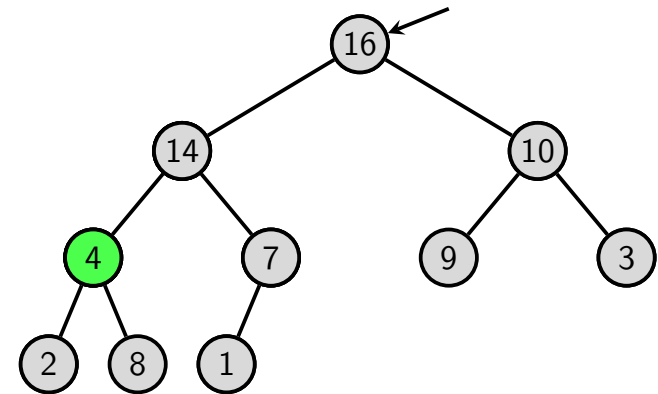
© 2016 Blai Bonet

Build-Max-Heap



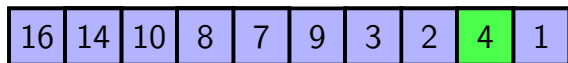
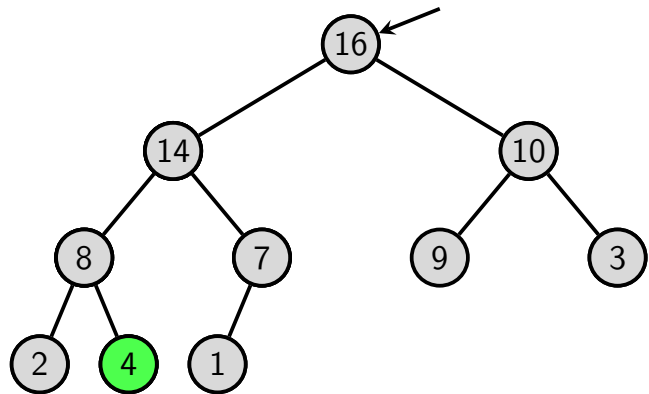
© 2016 Blai Bonet

Build-Max-Heap



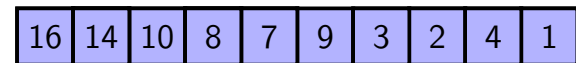
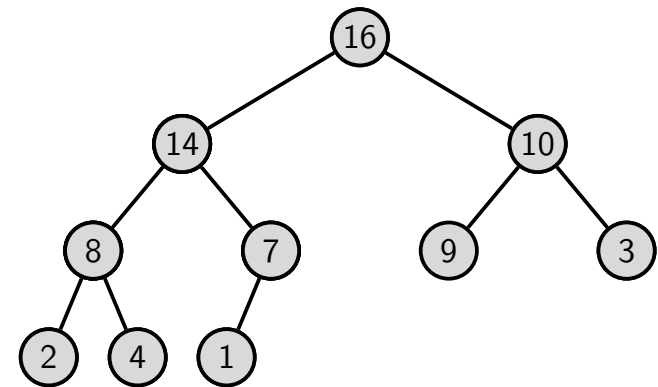
© 2016 Blai Bonet

Build-Max-Heap



© 2016 Blai Bonet

Build-Max-Heap



© 2016 Blai Bonet

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 $i + 1, i + 2, \dots, n$ son raíces max-heaps

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

Output: arreglo A con elementos reordenados formando un heap

```
Build-Max-Heap(array A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  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 $i + 1, i + 2, \dots, n$ son raíces 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 $i + 1, i + 2, \dots, n$ son raíces max-heaps

Inicialización:

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

Cada nodo $i + 1, i + 2, \dots, n$ 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 $i + 1, i + 2, \dots, n$ son raíces 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 por y seguirán siendo raíces de heaps al terminar **Max-Heapify(A, i)**

Por lo tanto, al terminar la iteración i , los nodos $i, i + 1, \dots, n$ 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 $i + 1, i + 2, \dots, n$ son raíces 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** $\lceil n/2 \rceil = \Theta(n)$ 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
- Existen a lo sumo $\lceil n/2^{h+1} \rceil$ nodos de altura h en el árbol (ejercicio)

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

Heapsort

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

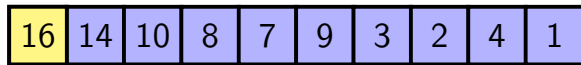
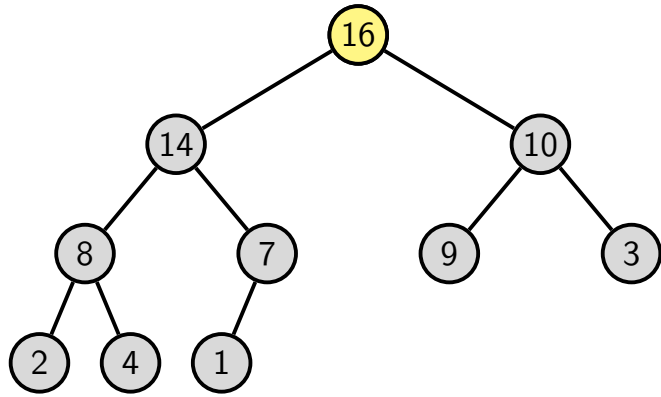
Dado un arreglo A , lo primero es construir un heap con los elementos de A en tiempo $O(n)$ usando **Build-Max-Heap(A)**

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 coloca $A[1]$ en la posición final intercambiándolo con $A[n]$ y restablece la propiedad de heap que puede perderse al colocar $A[n]$ en $A[1]$ usando **Max-Heapify(A, 1)**

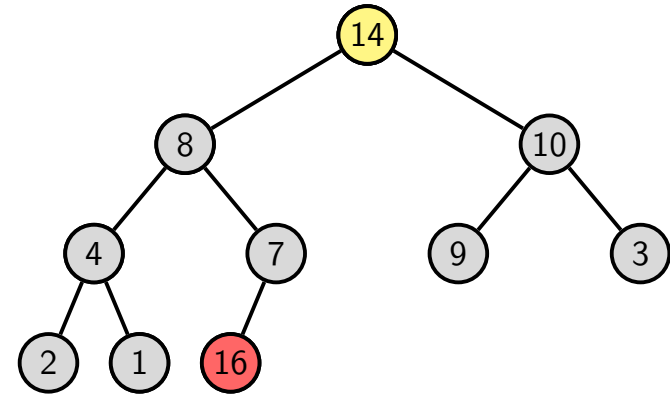
El algoritmo se repite hasta vaciar el heap

Heapsort



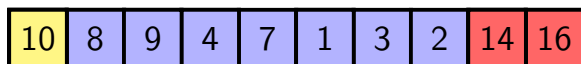
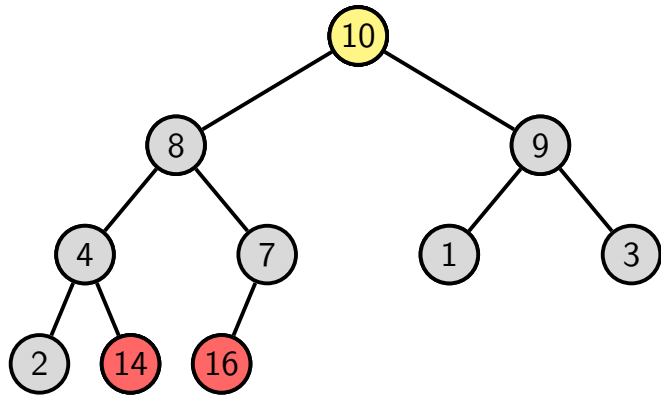
© 2016 Blai Bonet

Heapsort



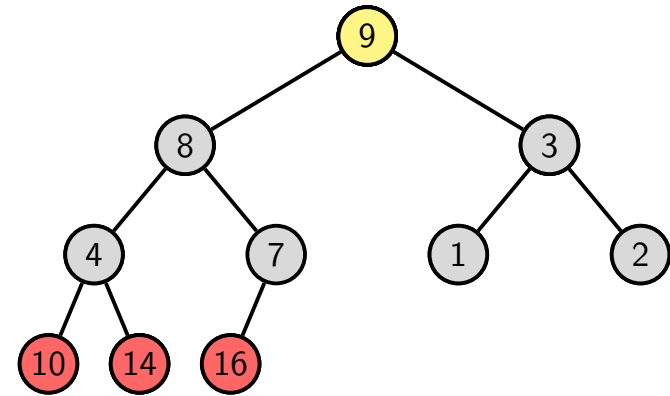
© 2016 Blai Bonet

Heapsort



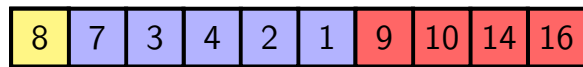
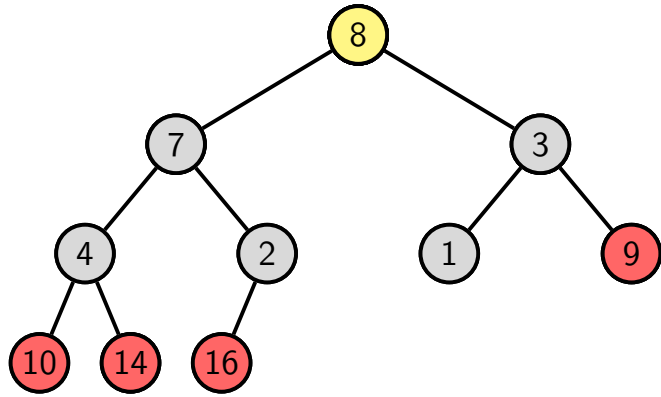
© 2016 Blai Bonet

Heapsort



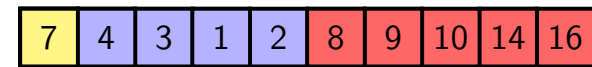
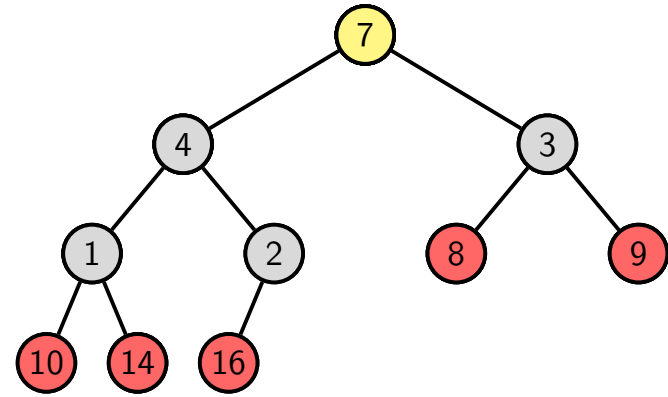
© 2016 Blai Bonet

Heapsort



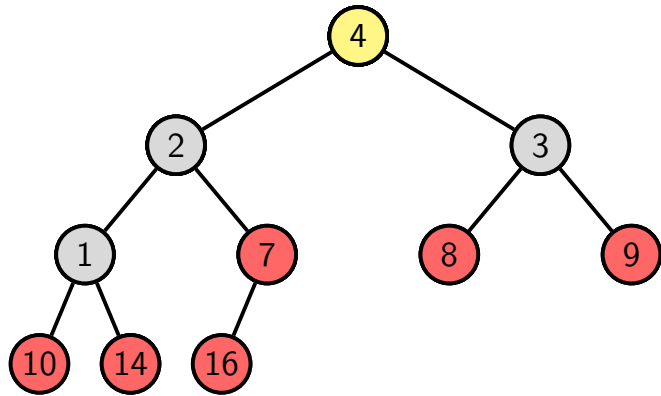
© 2016 Blai Bonet

Heapsort



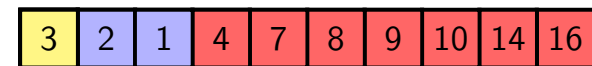
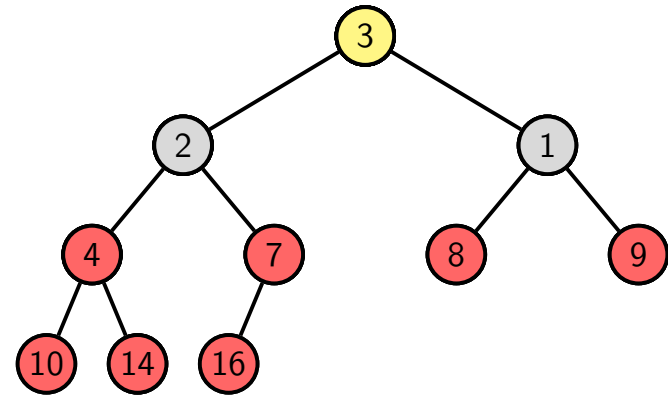
© 2016 Blai Bonet

Heapsort



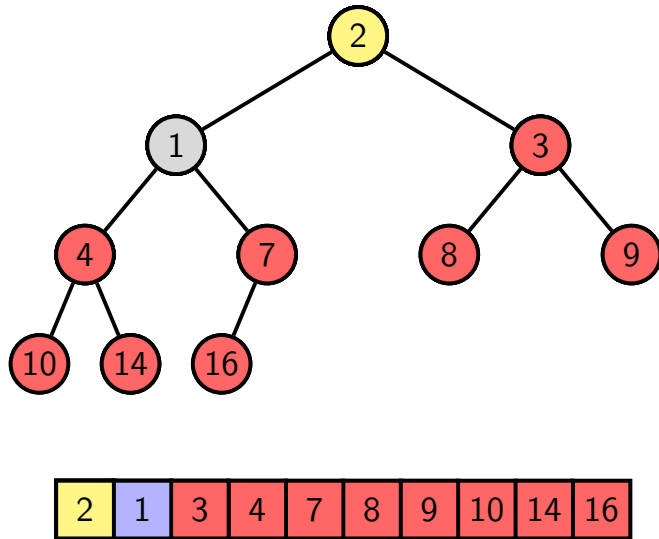
© 2016 Blai Bonet

Heapsort



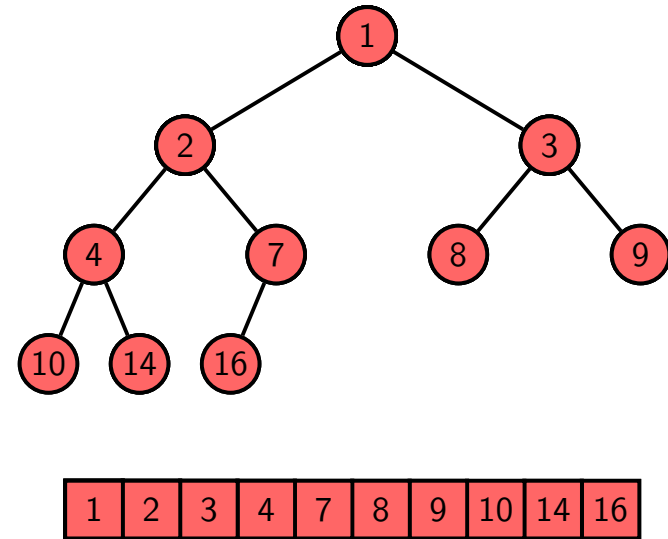
© 2016 Blai Bonet

Heapsort



© 2016 Blai Bonet

Heapsort



© 2016 Blai Bonet

Heapsort

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

Output: arreglo A con elementos reordenados de menor a mayor

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

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

© 2016 Blai Bonet

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)

© 2016 Blai Bonet

Implementacion de cola de prioridades con un heap

```
1 Heap-Maximum(array A)
2   return A[1]
3
4 Heap-Extract-Max(array A)
5   if A.heap-size < 1 then
6     error "no existen elementos en la cola"
7   max = A[1]
8   A[1] = A[A.heap-size]
9   A.heap-size = A.heap-size - 1
10  Max-Heapify(A, 1)
11  return max
```

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

Implementacion de cola de prioridades con un heap

```
1 Heap-Increase-Key(array A, int i, int key)
2   if key < A[i] then
3     error "la nueva clave es menor a la existente"
4   A[i] = key
5   while i > 1 && A[Parent(i)] < A[i]
6     Intercambiar A[i] con A[Parent(i)]
7     i = Parent(i)
8
9 Heap-Insert(array A, int key)
10  A.heap-size = A.heap-size + 1
11  A[A.heap-size] = -∞
12  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 $n = A.heap-size$

Referencias entre objetos e índices en el heap

En ciertas aplicaciones, como agendamiento de tareas (scheduling) y simulación basada en eventos (event-driven simulation), los objetos son exactamente las claves que se guardan en la cola

Sin embargo, en general, tenemos objetos opacos con prioridades asociadas y la estructura de datos debe:

- mantener una referencia (apuntador) dentro de cada nodo, adicional a la prioridad, de forma de asociar objetos con prioridades

Por otro lado, la aplicación que usa la cola de prioridades debe:

- mantener referencias entre los objetos guardados en el heap y los índices de los nodos para dichos objetos

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
- Mantenimiento de **referencias cruzadas** entre índices del heap y objetos

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

Ejercicios (2 de 3)

6. (6.3-2) ¿Por qué el lazo en **Build-Max-Heapify** decrementa la variable de inducción i desde $\lfloor A.heap-size \rfloor$ hasta 1 en lugar de incrementarla desde 1 hasta $\lfloor A.heap-size \rfloor$?
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 $A[p \dots r]$. 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, \dots, 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 operación **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 $O(\log n)$.