… previamente en IIC2133

Caso promedio de `Quicksort`

Con lo anterior, las comparaciones ecuando el pivote queda en $A[q]$ son

$$n - 1 + C(q) + C(n - q - 1)$$

Para ver el caso promedio, ponderamos para todos los q posibles obteniendo

$$C(n) \; = \; n - 1 + \frac{1}{n} \sum_{q=0}^{n-1} \left(C(q) + C(n - q - 1) \right)$$

Además, notando que

$$\sum_{q=0}^{n-1} C(n - q - 1) \;\; = \;\; C(n-1) + C(n-2) + \cdots + C(0) \;\; = \;\; \sum_{q=0}^{n-1} C(q)$$

obtenemos la regla simplificada

$$C(n) \; = \; n - 1 + \frac{2}{n} \sum_{q=0}^{n-1} C(q)$$

Caso promedio de `Quicksort`

Para la ecuación de recurrencia

$$C(n) = n - 1 + \frac{2}{n}\sum_{q=0}^{n-1} C(q)$$

tenemos dos casos base

- $C(0) = 0$, pues corresponde al caso base de `Quicksort`
- $C(1) = 0$, pues `Partition` no itera si hay un solo elemento

Caso promedio de `Quicksort`

Amplificamos por n la recurrencia y la reescribimos para $n-1$

$$\begin{aligned} nC(n) &= n(n-1) + 2\sum_{q=0}^{n-1} C(q) \\ (n-1)C(n-1) &= (n-1)(n-2) + 2\sum_{q=0}^{n-2} C(q) \end{aligned}$$

Restando ambas ecuaciones obtenemos

$$nC(n) = (n+1)C(n-1) + 2n - 2$$

Dividimos por $n(n+1)$ y comenzamos una serie de inecuaciones

$$\begin{aligned} \frac{C(n)}{n+1} &= \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\ &\leq \frac{C(n-1)}{n} + \frac{2}{n+1} \end{aligned}$$

Caso promedio de Quicksort

$$
\begin{aligned}
\frac{C(n)}{n+1} &\leq \frac{C(n-1)}{n} + \frac{2}{n+1} \\
&\leq \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\leq \frac{C(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\dots \\
&\leq \frac{C(1)}{2} + \sum_{k=2}^{n} \frac{2}{k+1} \\
&\leq \sum_{k=1}^{n} \frac{2}{k} \leq \int_1^n \frac{2}{x} dx = 2\log(n)
\end{aligned}
$$

Concluimos que una cota superior para el número de comparaciones es

$$C(n) \leq 2(n+1)\log(n)$$

`Quicksort` es $\mathcal{O}(n\log(n))$ en el caso promedio

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	?	?	?	?

Mejoras en `Quicksort`

- Para sub-secuencias pequeñas (e.g. $n \leq 20$) podemos usar `InsertionSort`
  - A pesar de no tener una complejidad mejor
  - Eso es solo cuando hablamos asintóticamente
  - En la práctica, en instancias pequeñas tiene mejor desempeño
  - No olvidar que `Quicksort` es **recursivo**... eso cuesta recursos
- Usar la mediana de tres elementos como pivote
  - *Informar* la elección de pivote
  - Dado A, escogemos 3 elementos $A[k_1], A[k_2], A[k_3]$
  - En $\mathcal{O}(1)$ encontramos la mediana entre ellos
- Particionar la secuencia en 3 sub-secuencias: menores, iguales y mayores
  - Mejora para caso con datos repetidos
  - Evita que `Partition` particione innecesariamente sub-secuencias en que todos los valores son iguales

Heaps y heapsort

Clase 6

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

Estructuras basadas en arreglos

Estructuras basadas en arreglos

Hasta ahora, hemos usado arreglos y listas ligadas

Estructuras basadas en arreglos

Hasta ahora, hemos usado arreglos y listas ligadas

- Son una representación directa de la memoria

Estructuras basadas en arreglos

Hasta ahora, hemos usado arreglos y listas ligadas

- Son una representación directa de la memoria
- Permiten acceso por índice en tiempo $\mathcal{O}(1)$

Estructuras basadas en arreglos

Hasta ahora, hemos usado arreglos y listas ligadas

- Son una representación directa de la memoria
- Permiten acceso por índice en tiempo $\mathcal{O}(1)$
- Los usamos en algoritmos de ordenación

Estructuras basadas en arreglos

Hasta ahora, hemos usado arreglos y listas ligadas

- Son una representación directa de la memoria
- Permiten acceso por índice en tiempo $\mathcal{O}(1)$
- Los usamos en algoritmos de ordenación

Hoy veremos un uso particular que aprovecha el acceso por índice para definir una **nueva EDD**

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**
- Consultar cuál es el dato **más prioritario**

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**
- Consultar cuál es el dato **más prioritario**
- Recorrer los datos en **orden de prioridad**

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**
- Consultar cuál es el dato **más prioritario**
- Recorrer los datos en **orden de prioridad**

Como primer acercamiento, ya conocemos un tipo de **prioridad**

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**
- Consultar cuál es el dato **más prioritario**
- Recorrer los datos en **orden de prioridad**

Como primer acercamiento, ya conocemos un tipo de **prioridad**

- Si interesa el que llegó último

Estructuras basadas en arreglos

Definición

Una **cola de prioridades** es una EDD que permite

- Almacenar datos según cierta **prioridad**
- Consultar cuál es el dato **más prioritario**
- Recorrer los datos en **orden de prioridad**

Como primer acercamiento, ya conocemos un tipo de **prioridad**

- Si interesa el que llegó último
- O si interesa el que llegó primero

Colas FIFO

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.
  - Arreglo: $\mathcal{O}(1)$ en general (salvo que se llene)

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.
  - Arreglo: $\mathcal{O}(1)$ en general (salvo que se llene)
  - Lista: $\mathcal{O}(1)$ con puntero al último elemento

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.
  - Arreglo: $\mathcal{O}(1)$ en general (salvo que se llene)
  - Lista: $\mathcal{O}(1)$ con puntero al último elemento
- **Extracción:** se elimina la cabeza de la cola.

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.
  - Arreglo: $\mathcal{O}(1)$ en general (salvo que se llene)
  - Lista: $\mathcal{O}(1)$ con puntero al último elemento
- **Extracción:** se elimina la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ si no reubicamos

Colas FIFO

Una **cola FIFO** (*first in first out*) es una cola donde la prioridad es el orden de llegada

- Primer elemento es el **más** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **menos** prioritario: *lleva menos tiempo en la cola*

Las operaciones en las colas son

- **Inserción:** se inserta al final de la cola.
  - Arreglo: $\mathcal{O}(1)$ en general (salvo que se llene)
  - Lista: $\mathcal{O}(1)$ con puntero al último elemento
- **Extracción:** se elimina la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ si no reubicamos
  - Lista: $\mathcal{O}(1)$

Colas LIFO

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ (recorriendo al revés)

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ (recorriendo al revés)
  - Lista: $\mathcal{O}(1)$

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ (recorriendo al revés)
  - Lista: $\mathcal{O}(1)$
- **Extracción:** se elimina la cabeza de la cola.

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ (recorriendo al revés)
  - Lista: $\mathcal{O}(1)$
- **Extracción:** se elimina la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ si no reubicamos

Colas LIFO

Una **cola LIFO o stack** (*last in first out*) es una cola donde la prioridad es el orden de llegada invertido

- Primer elemento es el **menos** prioritario: *lleva más tiempo en la cola*
- Último elemento es el **más** prioritario: *lleva menos tiempo en la cola*

Las operaciones en los stacks son

- **Inserción:** se inserta en la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ (recorriendo al revés)
  - Lista: $\mathcal{O}(1)$
- **Extracción:** se elimina la cabeza de la cola.
  - Arreglo: $\mathcal{O}(1)$ si no reubicamos
  - Lista: $\mathcal{O}(1)$

Colas de prioridades (redefinición)

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

- **Insertar** un dato con prioridad dada

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

- **Insertar** un dato con prioridad dada
- **Extraer** el dato con mayor prioridad

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

- **Insertar** un dato con prioridad dada
- **Extraer** el dato con mayor prioridad
- Idealmente, **cambiar** la prioridad de un dato

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

- **Insertar** un dato con prioridad dada
- **Extraer** el dato con mayor prioridad
- Idealmente, **cambiar** la prioridad de un dato

A diferencia de las colas FIFO y LIFO, el orden de llegada no es equivalente a la posición en la cola

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Para el arreglo ordenado por valor

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Para el arreglo ordenado por valor

- Inserción en la *posición correcta* $\mathcal{O}(n)$

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Para el arreglo ordenado por valor

- Inserción en la *posición correcta* $\mathcal{O}(n)$
- Extracción del último elemento $\mathcal{O}(1)$

Colas de prioridades

Ejemplo

Si la prioridad de una cola de prioridades A es el valor de los datos, y todos son naturales, tenemos dos opciones **extremas**

1. Usar un arreglo sin orden entre sus elementos
2. Usar un arreglo que siempre esté ordenado

La complejidad en estos dos escenarios es diferente.

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Para el arreglo ordenado por valor

- Inserción en la *posición correcta* $\mathcal{O}(n)$
- Extracción del último elemento $\mathcal{O}(1)$

¿Se puede hacer mejor?

Sumario

Orden de los elementos

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

En el contexto de datos en una EDD A, podemos distinguir

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

En el contexto de datos en una EDD A, podemos distinguir

- Orden total: todos los elementos de A están ordenados

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

En el contexto de datos en una EDD A, podemos distinguir

- Orden total: todos los elementos de A están ordenados
- Orden parcial: hay sub-sectores de A que están ordenados y conocemos bien la división de los sub-sectores

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

En el contexto de datos en una EDD A, podemos distinguir

- Orden total: todos los elementos de A están ordenados
- Orden parcial: hay sub-sectores de A que están ordenados y conocemos bien la división de los sub-sectores

¿Necesitamos un orden **total** de los datos para lograr colas eficientes?

Orden de los elementos

Para hacer eficientes las colas, necesitamos cierto orden

En el contexto de datos en una EDD A, podemos distinguir

- Orden total: todos los elementos de A están ordenados
- Orden parcial: hay sub-sectores de A que están ordenados y conocemos bien la división de los sub-sectores

¿Necesitamos un orden **total** de los datos para lograr colas eficientes?
No! Basta con "*cierto orden*" entre algunos elementos

Hacia una implementación de colas eficientes

Hacia una implementación de colas eficientes

Utilizaremos un enfoque de sub-estructuras ordenadas

Hacia una implementación de colas eficientes

Utilizaremos un enfoque de sub-estructuras ordenadas

- Seguiremos un enfoque recursivo

Hacia una implementación de colas eficientes

Utilizaremos un enfoque de sub-estructuras ordenadas

- Seguiremos un enfoque recursivo
- Estructura recursiva + algoritmos recursivos

Hacia una implementación de colas eficientes

Utilizaremos un enfoque de sub-estructuras ordenadas

- Seguiremos un enfoque recursivo
- Estructura recursiva + algoritmos recursivos
- Cada sub-estructura debe tener cierta información disponible

Hacia una implementación de colas eficientes

Utilizaremos un enfoque de sub-estructuras ordenadas

- Seguiremos un enfoque recursivo
- Estructura recursiva + algoritmos recursivos
- Cada sub-estructura debe tener cierta información disponible

Definiremos nuestra primera EDD recursiva

Árboles binarios

Árboles binarios

Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

Árboles binarios

Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave

Árboles binarios

Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave
2. El nodo puede tener hasta dos AB's asociados mediante punteros

Árboles binarios

Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave
2. El nodo puede tener hasta dos AB's asociados mediante punteros
    - Hijo izquierdo

Árboles binarios

Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave
2. El nodo puede tener hasta dos AB's asociados mediante punteros
   - Hijo izquierdo
   - Hijo derecho

Árboles binarios

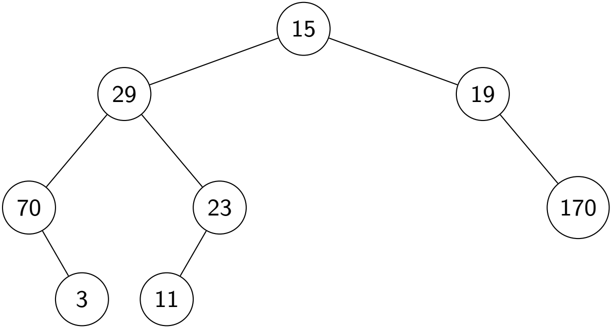
Definición

Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave
2. El nodo puede tener hasta dos AB's asociados mediante punteros
   - Hijo izquierdo
   - Hijo derecho

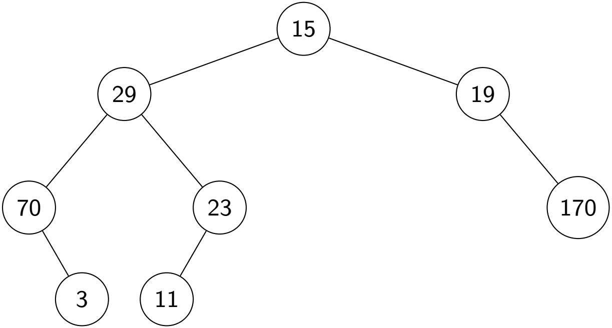
El árbol binario A tiene hijos $A.left$ y $A.right$, y llave $A.key$

Árboles binarios




divide and conquer...

Árboles binarios



Observemos que no hay un orden a priori entre sus elementos

Heaps binarios

Heaps binarios

Definición

Un **Max heap binario** H es un árbol binario tal que

Heaps binarios

Definición

Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios

Heaps binarios

Definición

Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$

Heaps binarios

Definición

Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$
- $H.key > H.right.key$

Heaps binarios

Definición

Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$
- $H.key > H.right.key$

A estas condiciones les llamamos **propiedad de heap**

Heaps binarios

Definición

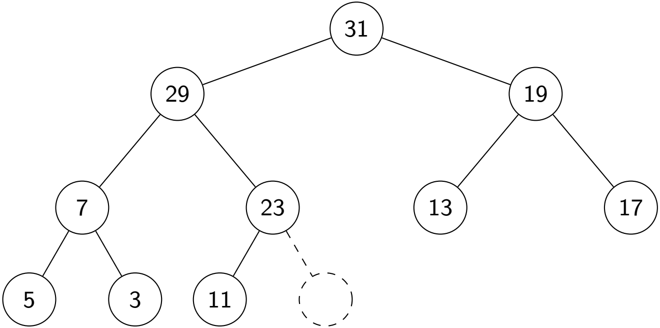
Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$
- $H.key > H.right.key$

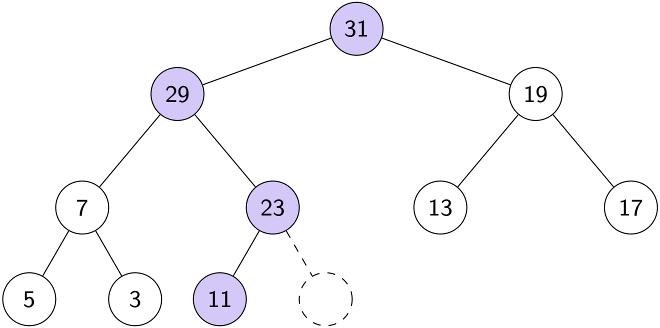
A estas condiciones les llamamos **propiedad de heap**

Todo hijo tiene llaves menores que el padre...
pero entre hermanos no hay ninguna restricción

Heaps binarios

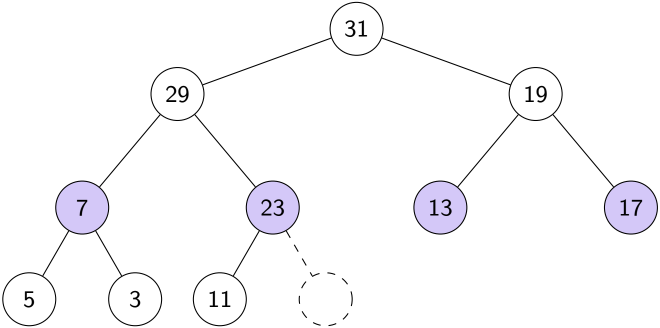


Heaps binarios



Todo camino hasta hoja descendiente
visita valores estrictamente decrecientes

Heaps binarios

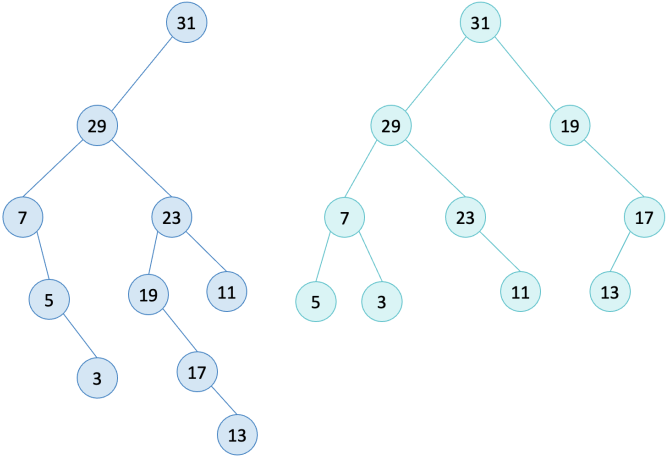


Los nodos de un mismo nivel no satisfacen un orden específico

Balance en heaps

Balance en heaps

En principio un heap no tiene garantías de altura



Balance en heaps

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**

Balance en heaps

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**

- Ojo: esto no significa que si hay h niveles, haya $n = 2^h - 1$ nodos

Balance en heaps

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**

- Ojo: esto no significa que si hay h niveles, haya $n = 2^h - 1$ nodos
- Lo que interesa es que antes de agregar un nivel, el último disponible se complete

Balance en heaps

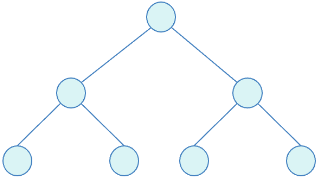
Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**

- Ojo: esto no significa que si hay h niveles, haya $n = 2^h - 1$ nodos
- Lo que interesa es que antes de agregar un nivel, el último disponible se complete

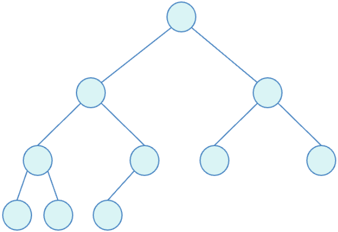
Podremos hacer esto gracias a la propiedad de heap

Balance en heaps

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**



árbol binario lleno, cuando
el número n de nodos cumple
$n = 2^d - 1$



árbol binario lleno, cuando
el número n de nodos cumple
$2^d \le n < 2^{d+1}$

Balance en heaps

Mantener los heaps balanceados permite

Balance en heaps

Mantener los heaps balanceados permite

- Minimizar la altura del árbol representado

Balance en heaps

Mantener los heaps balanceados permite

- Minimizar la altura del árbol representado
- Implementar el heap de forma **compacta** en un arreglo

Balance en heaps

Mantener los heaps balanceados permite

- Minimizar la altura del árbol representado
- Implementar el heap de forma **compacta** en un arreglo

¡No necesitaremos punteros!


WAIT...
WHAAAT?

Balance en heaps

La representación permite recorrer descendientes sin punteros

Balance en heaps

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$

Balance en heaps

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Balance en heaps

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Además, permite ubicar los elementos del nivel h sin punteros

Balance en heaps

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Además, permite ubicar los elementos del nivel h sin punteros

- El primer elemento del nivel h es $A[2^h - 1]$

Balance en heaps

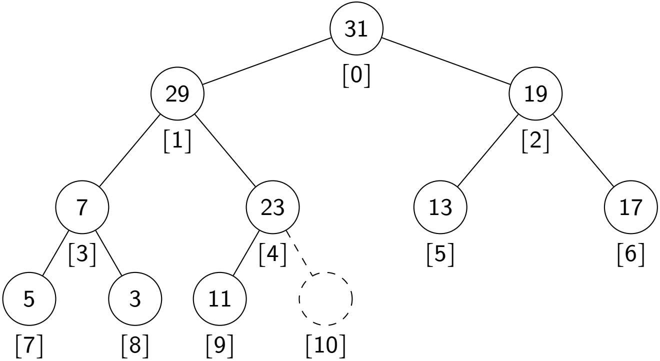
La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Además, permite ubicar los elementos del nivel h sin punteros

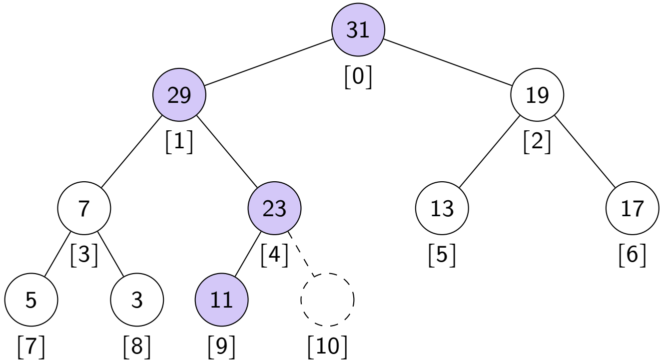
- El primer elemento del nivel h es $A[2^h - 1]$
- Los 2^h elementos consecutivos corresponden al nivel h

Heaps binarios: representación compacta



31	29	19	7	23	13	17	5	3	11	
0	1	2	3	4	5	6	7	8	9	10

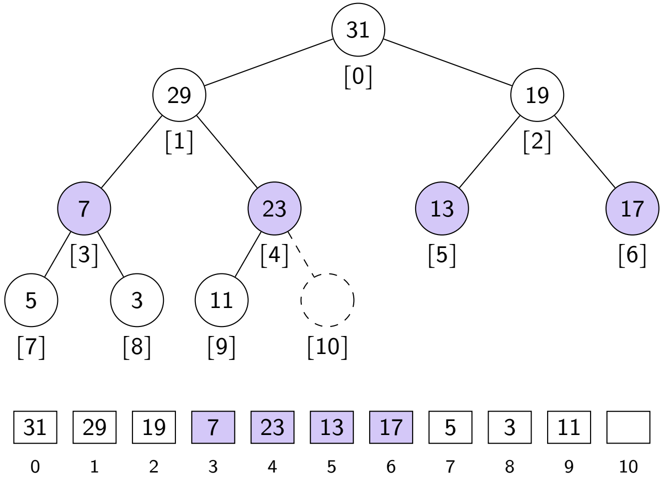
Heaps binarios: representación compacta



31	29	19	7	23	13	17	5	3	11	
0	1	2	3	4	5	6	7	8	9	10

Línea de descendientes sin uso de punteros

Heaps binarios: representación compacta



Nivel 2 sin uso de punteros

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

Al **insertar** y **extraer**

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

Al **insertar** y **extraer**

1. Efectuamos la operación manteniendo un árbol binario casi-lleno

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

Al **insertar** y **extraer**

1. Efectuamos la operación manteniendo un árbol binario casi-lleno
2. Reestablecemos la propiedad de heap

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

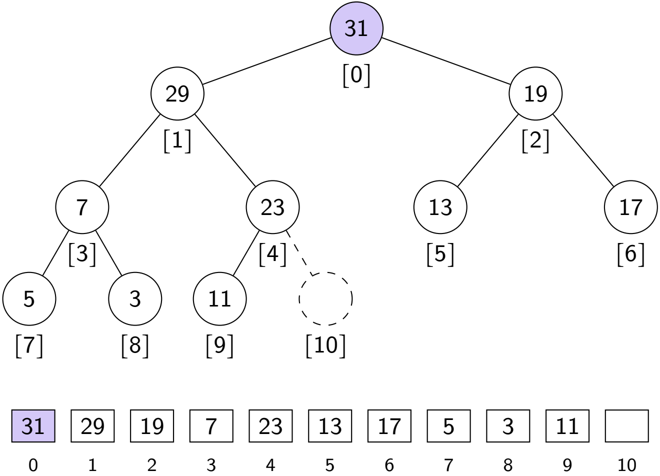
Al **insertar** y **extraer**

1. Efectuamos la operación manteniendo un árbol binario casi-lleno
2. Reestablecemos la propiedad de heap

Cada operación involucra dos fases

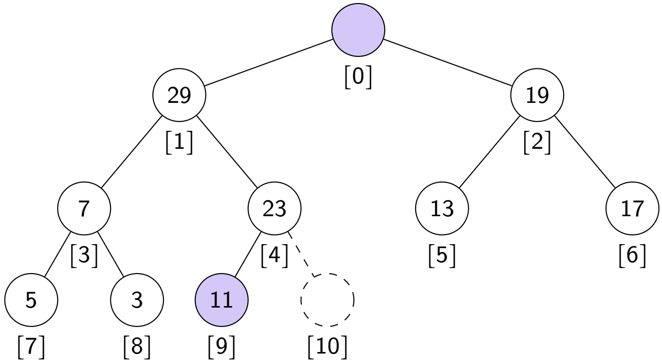
Balance en heaps: extracción

Al extraer, sacamos el elemento más prioritario



Balance en heaps: extracción

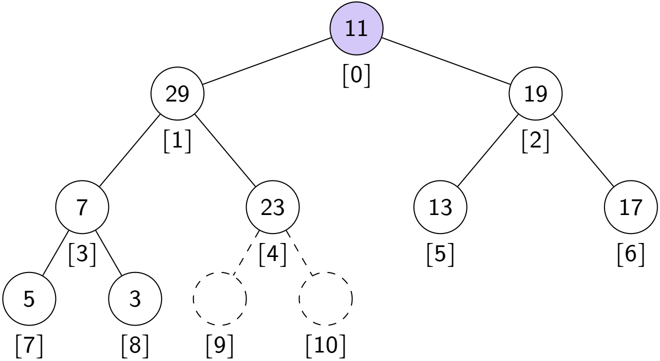
Al sacarlo, el árbol **ya no está casi-lleno**. Movemos el último elemento del arreglo



0	1	2	3	4	5	6	7	8	9	10
	29	19	7	23	13	17	5	3	11	

Balance en heaps: extracción

Ahora el árbol **está casi-lleno**, pero no se cumple la propiedad de heap



11	29	19	7	23	13	17	5	3		
0	1	2	3	4	5	6	7	8	9	10

Balance en heaps: extracción

input : heap representado como arreglo $H[0 \dots n-1]$
output: elemento más prioritario

$\texttt{Extract}(H)$**:**

$i \leftarrow$ última celda no vacía de H
$best \leftarrow H[0]$
$H[0] \leftarrow H[i]$
$H[i] \leftarrow \varnothing$
$\texttt{SiftDown}(H, 0)$
return *best*

Balance en heaps: extracción

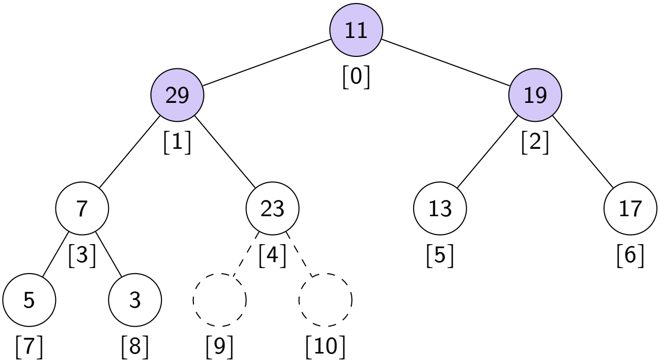
input : heap representado como arreglo $H[0 \ldots n-1]$
output: elemento más prioritario

```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

Intercambiamos antes de reestablecer la propiedad de heap con `SiftDown`

Balance en heaps: extracción

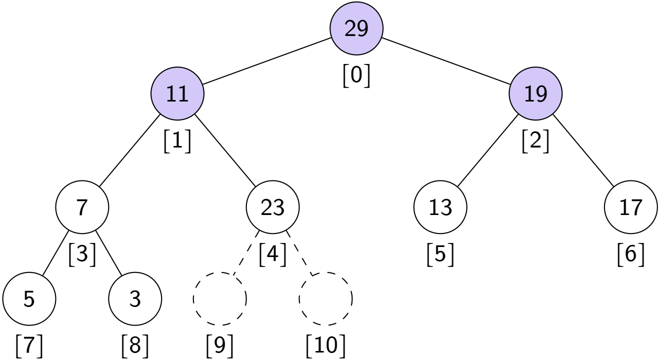
Vemos si hay hijos y comparamos sus prioridades: solo intercambiamos si alguno es mayor



11	29	19	7	23	13	17	5	3		
0	1	2	3	4	5	6	7	8	9	10

Balance en heaps: extracción

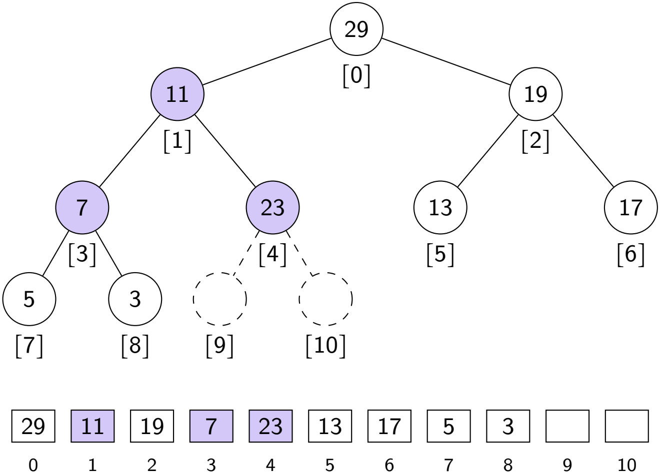
Intercambiamos con su hijo más prioritario



29	11	19	7	23	13	17	5	3		
0	1	2	3	4	5	6	7	8	9	10

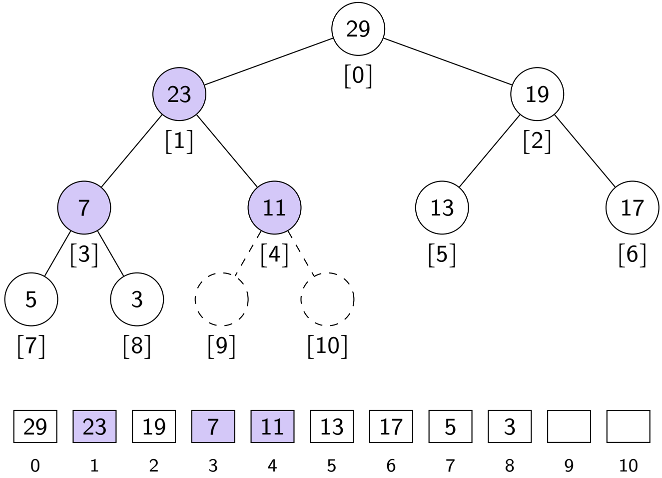
Balance en heaps: extracción

Repetimos el proceso recursivamente. Comparamos con los hijos y vemos si alguno es mayor



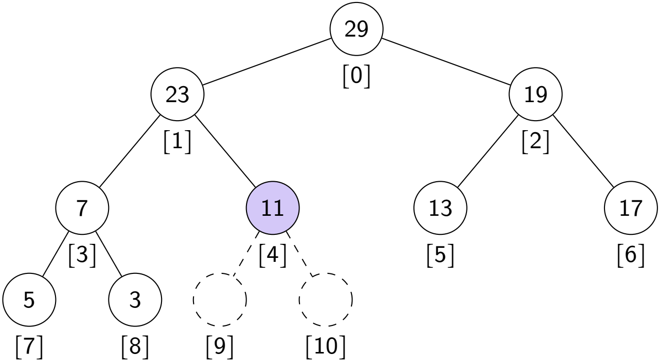
Balance en heaps: extracción

Corresponde intercambiar con el hijo derecho



Balance en heaps: extracción

Chequeamos nuevamente y en este caso no hay hijos mayores: terminamos



29	23	19	7	11	13	17	5	3		
0	1	2	3	4	5	6	7	8	9	10

Balance en heaps: extracción

input : heap representado como arreglo $H[0 \dots n-1]$, índice $0 \leq i \leq n-1$

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

Balance en heaps: extracción

input : heap representado como arreglo $H[0 \dots n-1]$, índice $0 \leq i \leq n-1$

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

Para un arreglo de largo n, este método es $\mathcal{O}(\log(n))$ gracias a que es un árbol casi-lleno

Balance en heaps: inserción

La inserción sigue la misma idea de la extracción

input : heap como arreglo $H[0 \ldots n-1]$, elemento e

$\texttt{Insert}(H)$**:**

$i \leftarrow$ primera celda vacía de H

$H[i] \leftarrow e$

$\texttt{SiftUp}(H, i)$

Balance en heaps: inserción

La inserción sigue la misma idea de la extracción

input : heap como arreglo $H[0 \dots n-1]$, elemento e

$\texttt{Insert}(H)$**:**

$i \leftarrow$ primera celda vacía de H

$H[i] \leftarrow e$

$\texttt{SiftUp}(H, i)$

La inserción se hace al final del arreglo y luego se reubica con `SiftUp`

Balance en heaps: inserción

input : heap representado como arreglo $H[0 \dots n-1]$,
índice $0 \leq i \leq n-1$

$\texttt{SiftUp}(H, i)$**:**
  if *i tiene padre* **:**
    $j \leftarrow \lfloor i/2 \rfloor$
    if $H[j] < H[i]$ **:**
      $H[j] \leftrightarrows H[i]$
      $\texttt{SiftUp}(H, j)$

Balance en heaps: inserción

input : heap representado como arreglo $H[0 \dots n-1]$,
índice $0 \leq i \leq n-1$

```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```

Para un arreglo de largo n, este método también es $\mathcal{O}(\log(n))$

… previamente en IIC2133

Colas de prioridades (redefinición)

Definición

Una **cola de prioridades** o **cola highest priority first out** es una EDD que permite

- **Insertar** un dato con prioridad dada
- **Extraer** el dato con mayor prioridad
- Idealmente, **cambiar** la prioridad de un dato

A diferencia de las colas FIFO y LIFO, el orden de llegada no es equivalente a la posición en la cola

Heaps binarios

Definición

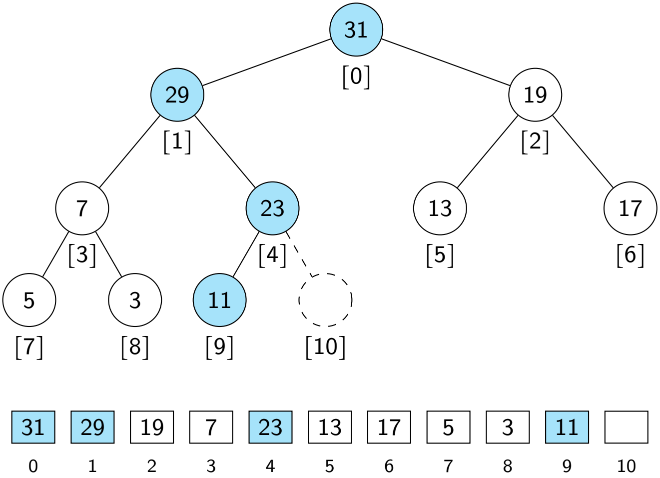
Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$
- $H.key > H.right.key$

A estas condiciones les llamamos **propiedad de heap**

Todo hijo tiene llaves menores que el padre...
pero entre hermanos no hay ninguna restricción

Heaps binarios: representación compacta



Línea de descendientes sin uso de punteros

Balance en heaps

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Además, permite ubicar los elementos del nivel h sin punteros

- El primer elemento del nivel h es $A[2^h - 1]$
- Los 2^h elementos consecutivos corresponden al nivel h

Balance en heaps

Ahora que contamos con una representación compacta, nos interesa asegurar el **balance** del heap

Al **insertar** y **extraer**

1. Efectuamos la operación manteniendo un árbol binario casi-lleno
2. Reestablecemos la propiedad de heap

Cada operación involucra dos fases

Balance en heaps: extracción

input : heap representado como arreglo $H[0 \dots n-1]$
output: elemento más prioritario

$\texttt{Extract}(H)$:

  $i \leftarrow$ última celda no vacía de H
  $best \leftarrow H[0]$
  $H[0] \leftarrow H[i]$
  $H[i] \leftarrow \varnothing$
  $\texttt{SiftDown}(H, 0)$
  return *best*

Intercambiamos antes de reestablecer la propiedad de heap con `SiftDown`

Balance en heaps: extracción

input : heap representado como arreglo $H[0 \dots n-1]$, índice $0 \leq i \leq n-1$

$\texttt{SiftDown}(H, i)$:

- **if** *i tiene hijos* :
  - $j \leftarrow$ hijo de i con mayor prioridad
  - **if** $H[j] > H[i]$:
    - $H[j] \leftrightarrows H[i]$
    - $\texttt{SiftDown}(H, j)$

Para un arreglo de largo n, este método es $\mathcal{O}(\log(n))$ gracias a que es un árbol casi-lleno

Balance en heaps: inserción

La inserción sigue la misma idea de la extracción

input : heap como arreglo $H[0 \ldots n-1]$, elemento e

`Insert`(H):

$i \leftarrow$ primera celda vacía de H

$H[i] \leftarrow e$

`SiftUp`(H, i)

La inserción se hace al final del arreglo y luego se reubica con `SiftUp`

Balance en heaps: inserción

input : heap representado como arreglo $H[0 \ldots n-1]$,
índice $0 \leq i \leq n-1$

`SiftUp`(H, i):
  if i *tiene padre* :
    $j \leftarrow \lfloor i/2 \rfloor$
    if $H[j] < H[i]$:
      $H[j] \leftrightarrows H[i]$
      `SiftUp`(H, j)

Para un arreglo de largo n, este método también es $\mathcal{O}(\log(n))$

Sumario

Construcción de un heap

Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap H preexistente

Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap H preexistente

Si tenemos un arreglo A y queremos obtener un heap podemos usar una de dos estrategias

Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap H preexistente

Si tenemos un arreglo A y queremos obtener un heap podemos usar una de dos estrategias

1. Iterar para cada elemento de A, insertando sobre un heap originalmente vacío

Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap H preexistente

Si tenemos un arreglo A y queremos obtener un heap podemos usar una de dos estrategias

1. Iterar para cada elemento de A, insertando sobre un heap originalmente vacío
2. Utilizar `SiftDown` para ciertos elementos de A

Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap H preexistente

Si tenemos un arreglo A y queremos obtener un heap podemos usar una de dos estrategias

1. Iterar para cada elemento de A, insertando sobre un heap originalmente vacío
2. Utilizar `SiftDown` para ciertos elementos de A

Esta última forma es *in place* y sencilla

Construcción de un heap

input : arreglo $A[0 \ldots n-1]$

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ loop decreciente
    `SiftDown`(A, i)


WAIT...
WHAAAT?

Construcción de un heap

input : arreglo $A[0 \ldots n-1]$

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ loop decreciente
    `SiftDown`(A, i)

Observación: los elementos de A en los cuales no se llama directamente `SiftDown` son hojas del último nivel del árbol

Construcción de un heap

input : arreglo $A[0 \dots n-1]$

`BuildHeap`(A)**:**

  for $i = \lfloor n/2 \rfloor - 1 \dots 0$ **:** $\triangleright$ loop decreciente

    `SiftDown`(A, i)

Observación: los elementos de A en los cuales no se llama directamente `SiftDown` son hojas del último nivel del árbol

Respecto a su complejidad

Construcción de un heap

input : arreglo $A[0 \dots n-1]$

$\texttt{BuildHeap}(A)$**:**
    for $i = \lfloor n/2 \rfloor - 1 \dots 0$ **:** $\triangleright$ loop decreciente
        $\texttt{SiftDown}(A, i)$

Observación: los elementos de A en los cuales no se llama directamente `SiftDown` son hojas del último nivel del árbol

Respecto a su complejidad

- La complejidad asintótica *directa* es $\mathcal{O}(n \log(n))$

Construcción de un heap

input : arreglo $A[0 \ldots n-1]$

`BuildHeap`(A)**:**
    for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ loop decreciente
        `SiftDown`(A, i)

Observación: los elementos de A en los cuales no se llama directamente `SiftDown` son hojas del último nivel del árbol

Respecto a su complejidad

- La complejidad asintótica *directa* es $\mathcal{O}(n \log(n))$
- Se puede demostrar que una mejor cota es $\mathcal{O}(n)$

Construcción de un heap

input : arreglo $A[0 \dots n-1]$

`BuildHeap`(A)**:**

for $i = \lfloor n/2 \rfloor - 1 \dots 0$ **:** $\quad \triangleright$ loop decreciente

`SiftDown`(A, i)

Observación: los elementos de A en los cuales no se llama directamente `SiftDown` son hojas del último nivel del árbol

Respecto a su complejidad

- La complejidad asintótica *directa* es $\mathcal{O}(n \log(n))$
- Se puede demostrar que una mejor cota es $\mathcal{O}(n)$

> `BuildHeap` deja A como un heap en tiempo $\mathcal{O}(n)$

Heaps para ordenar

Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

Y sabemos la propiedad de heap: cada nodo es estrictamente mayor que sus descendientes

Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

Y sabemos la propiedad de heap: cada nodo es estrictamente mayor que sus descendientes

¿Podemos aprovechar estos hechos para ordenar un arreglo A?

Ordenando con heaps

Dado un heap H

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

Ordenando con heaps

Dado un heap *H*

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más
- Es decir, reducimos el **tamaño del heap**

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más
- Es decir, reducimos el **tamaño del heap**
- A este parámetro le llamamos A.`heap_size`

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más
- Es decir, reducimos el **tamaño del heap**
- A este parámetro le llamamos A.`heap_size`

Cambiamos el tamaño del heap para que `SiftDown`
sepa hasta dónde llegar moviendo elementos

Ordenando con heaps

input : arreglo $A[0 \dots n-1]$

$\texttt{HeapSort}(A)$**:**
    $\texttt{BuildHeap}(A)$
    for $i = n-1 \dots 1$ **:** $\triangleright$ loop decreciente
        $A[0] \leftrightharpoons A[i]$
        $A.\texttt{heap\_size} = A.\texttt{heap\_size} - 1$
        $\texttt{ShiftDown}(A, 0)$

Respecto a su complejidad

Ordenando con heps

input : arreglo $A[0 \ldots n-1]$

$\texttt{HeapSort}(A)$**:**

$\quad \texttt{BuildHeap}(A)$

$\quad$ **for** $i = n-1 \ldots 1$ **:** $\qquad \triangleright$ loop decreciente

$\qquad A[0] \leftrightharpoons A[i]$

$\qquad A.\texttt{heap\_size} = A.\texttt{heap\_size} - 1$

$\qquad \texttt{ShiftDown}(A, 0)$

Respecto a su complejidad

- $\texttt{BuildHeap}$ $\qquad \mathcal{O}(n)$

Ordenando con heags

input : arreglo $A[0 \dots n-1]$

```
HeapSort(A):
    BuildHeap(A)
    for i = n-1 ... 1 :          ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size - 1
        ShiftDown(A, 0)
```

Respecto a su complejidad

- `BuildHeap` $\mathcal{O}(n)$
- `SiftDown` se repite $\mathcal{O}(n)$ veces $\mathcal{O}(n \log(n))$

Ordenando con heaps

input : arreglo $A[0 \dots n-1]$

$\texttt{HeapSort}(A)$**:**

  $\texttt{BuildHeap}(A)$

  for $i = n-1 \dots 1$ **:** $\triangleright$ loop decreciente

    $A[0] \leftrightarrows A[i]$

    $A.\texttt{heap\_size} = A.\texttt{heap\_size} - 1$

    $\texttt{ShiftDown}(A, 0)$

Respecto a su complejidad

- `BuildHeap` $\mathcal{O}(n)$
- `SiftDown` se repite $\mathcal{O}(n)$ veces $\mathcal{O}(n \log(n))$
- Total $\mathcal{O}(n + n\log(n)) = \mathcal{O}(n\log(n))$

Ordenando con heps

input : arreglo $A[0 \ldots n-1]$

```
HeapSort(A):
    BuildHeap(A)
    for i = n-1 ... 1 :        ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size - 1
        ShiftDown(A, 0)
```

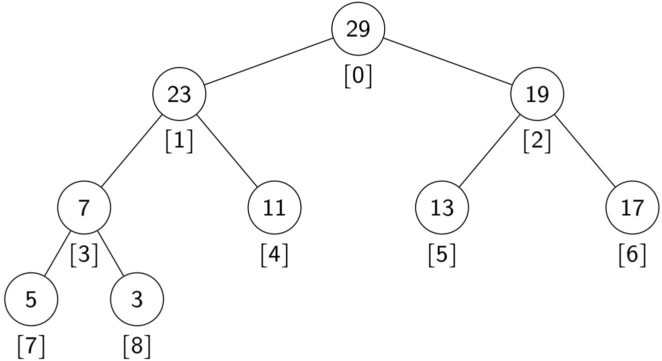
Respecto a su complejidad

- `BuildHeap` $\mathcal{O}(n)$
- `SiftDown` se repite $\mathcal{O}(n)$ veces $\mathcal{O}(n\log(n))$
- Total $\mathcal{O}(n + n\log(n)) = \mathcal{O}(n\log(n))$

`HeapSort` ordena en tiempo $\mathcal{O}(n\log(n))$

Heapsort en acción

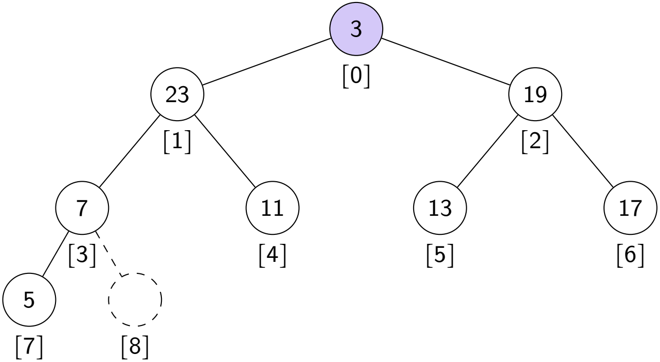
Supongamos que ya contamos con el heap resultante de `BuildHeap`



29	23	19	7	11	13	17	5	3
0	1	2	3	4	5	6	7	8

Heapsort en acción

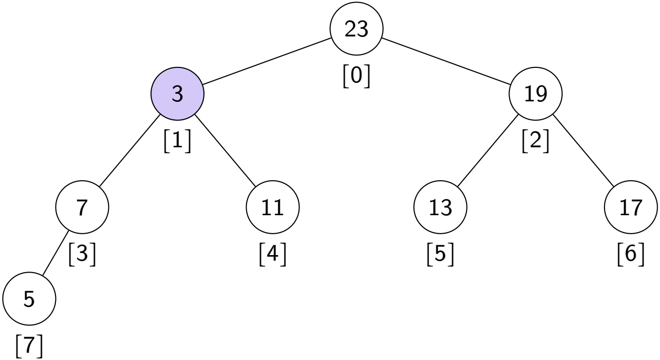
Movemos el primer elemento y reducimos el tamaño del heap en 1



3	23	19	7	11	13	17	5	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

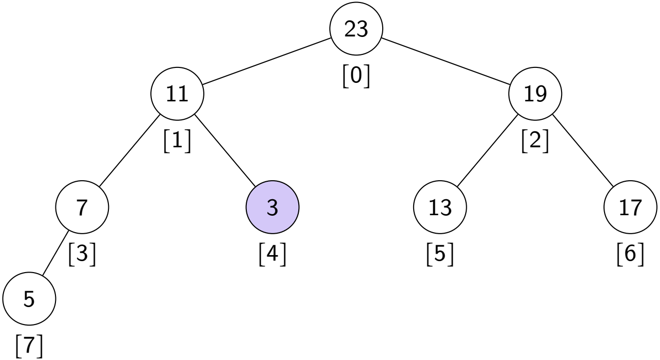
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \dots 7]$)



23	3	19	7	11	13	17	5	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

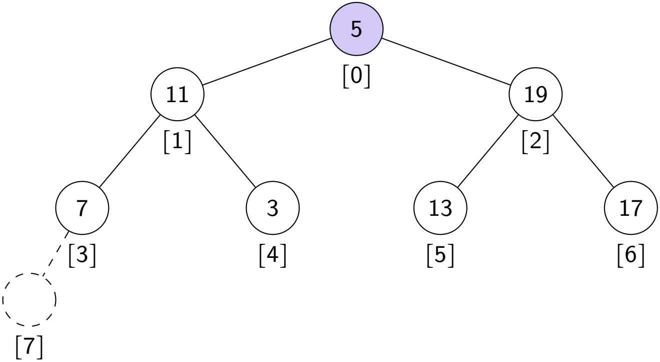
Aplicamos `SiftDown`$(A, 1)$ (el heap es $A[0 \ldots 7]$)



23	11	19	7	3	13	17	5	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

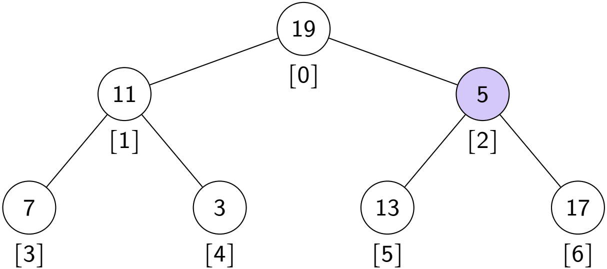
Repetimos el proceso con la nueva raíz



5	11	19	7	3	13	17	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

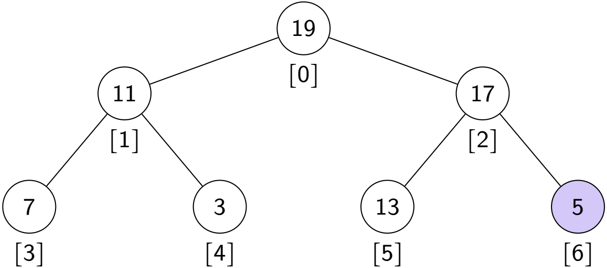
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 6]$)



19	11	5	7	3	13	17	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

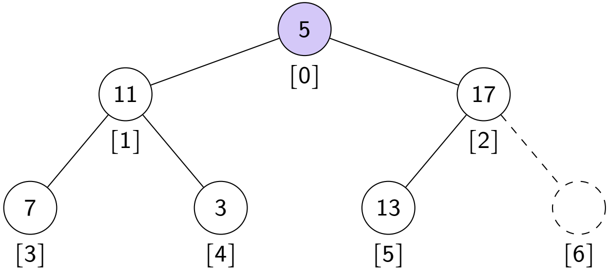
Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0 \ldots 6]$)



19	11	17	7	3	13	5	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

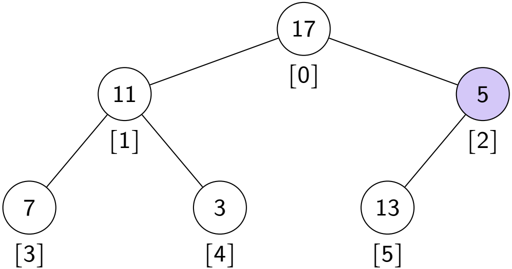
Repetimos el proceso con la nueva raíz



5	11	17	7	3	13	19	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

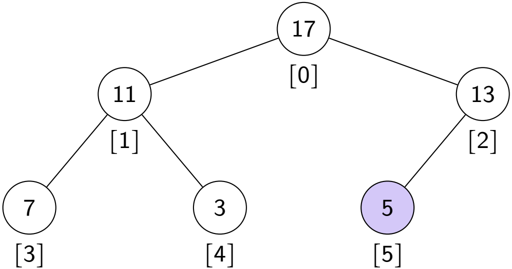
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 5]$)



17	11	5	7	3	13	19	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

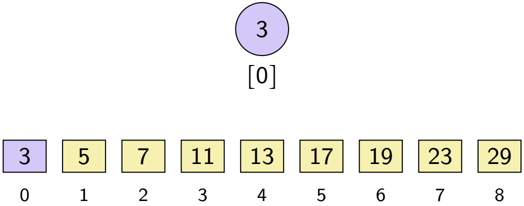
Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0 \ldots 5]$)



17	11	13	7	3	5	19	23	29
0	1	2	3	4	5	6	7	8

Heapsort en acción

El proceso termina cuando queda solo un nodo en el heap: es el mínimo



Complejidad de algoritmos de ordenación

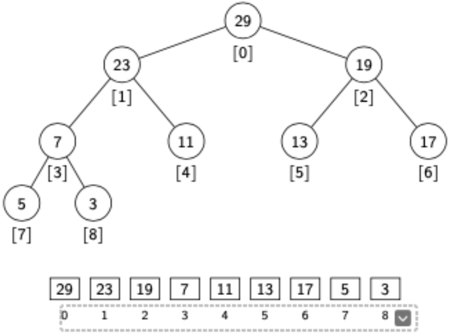
Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$

Heap & HeapSort

- Atributos generales
  - O(n log(n))
  - O(1) en memoria
  - Inventado por J. W. J. Williams in 1964
  - Selection Sort con la EDD adecuada
  - ¿Estable?
  - ¿Envenenable?

¿Cuándo usar Heap Sort?



Sumario

Objetivos de la clase

Objetivos de la clase

- ☐ Comprender el concepto de cola de prioridad

Objetivos de la clase

- ☐ Comprender el concepto de cola de prioridad
- ☐ Comprender la estructura de heaps binarios y su propiedad de heap

Objetivos de la clase

- ☐ Comprender el concepto de cola de prioridad
- ☐ Comprender la estructura de heaps binarios y su propiedad de heap
- ☐ Comprender operaciones básicas en heaps

Objetivos de la clase

- ☐ Comprender el concepto de cola de prioridad
- ☐ Comprender la estructura de heaps binarios y su propiedad de heap
- ☐ Comprender operaciones básicas en heaps
- ☐ Aplicar heaps para ordenar