

# La sala de urgencias

- Al llegar a una sala de urgencia, la persona es evaluada
- Se le asigna un número del 1 al 5 según la urgencia
- Cuando se desocupa un box, se hace pasar a la siguiente persona
- La persona que pasa es la con mayor urgencia que aún está en espera
- En caso de empate, pasa la persona que llegó primero

¿Cuánto cuesta decidir cuál es la próxima persona que hay que hacer pasar?

# La cola de prioridades

Una estructura de datos con las siguientes operaciones:

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

E idealmente:

- Cambiar la prioridad de un dato ya insertado

# Propiedad de orden de la cola



Claramente, hay que mantener cierto orden de los datos

¿Cuál es el costo —en términos del número de operaciones o pasos básicos— de mantener los datos **ordenados**?

¿Y al llegar  $n$  datos nuevos?

# ¿Es necesario un orden total?

Solo necesitamos saber cuál es el dato más prioritario

Quizás podamos darnos el lujo de no tener un orden total de los datos

¡Necesitamos algún tipo de estructura interna!

# Cómo aprovechar la propiedad de orden parcial



¿Qué información contenida en la estructura debe estar fácilmente disponible en todo momento?

¿Será posible hacer una estructura recursiva?

¿Por qué querríamos tener estructuras recursivas?

- los algoritmos para recorrerlas o buscar información en ellas son también recursivos y, por lo tanto, más simples
- la implementación de la estructura se simplifica

# El *heap* binario: Una estructura recursiva

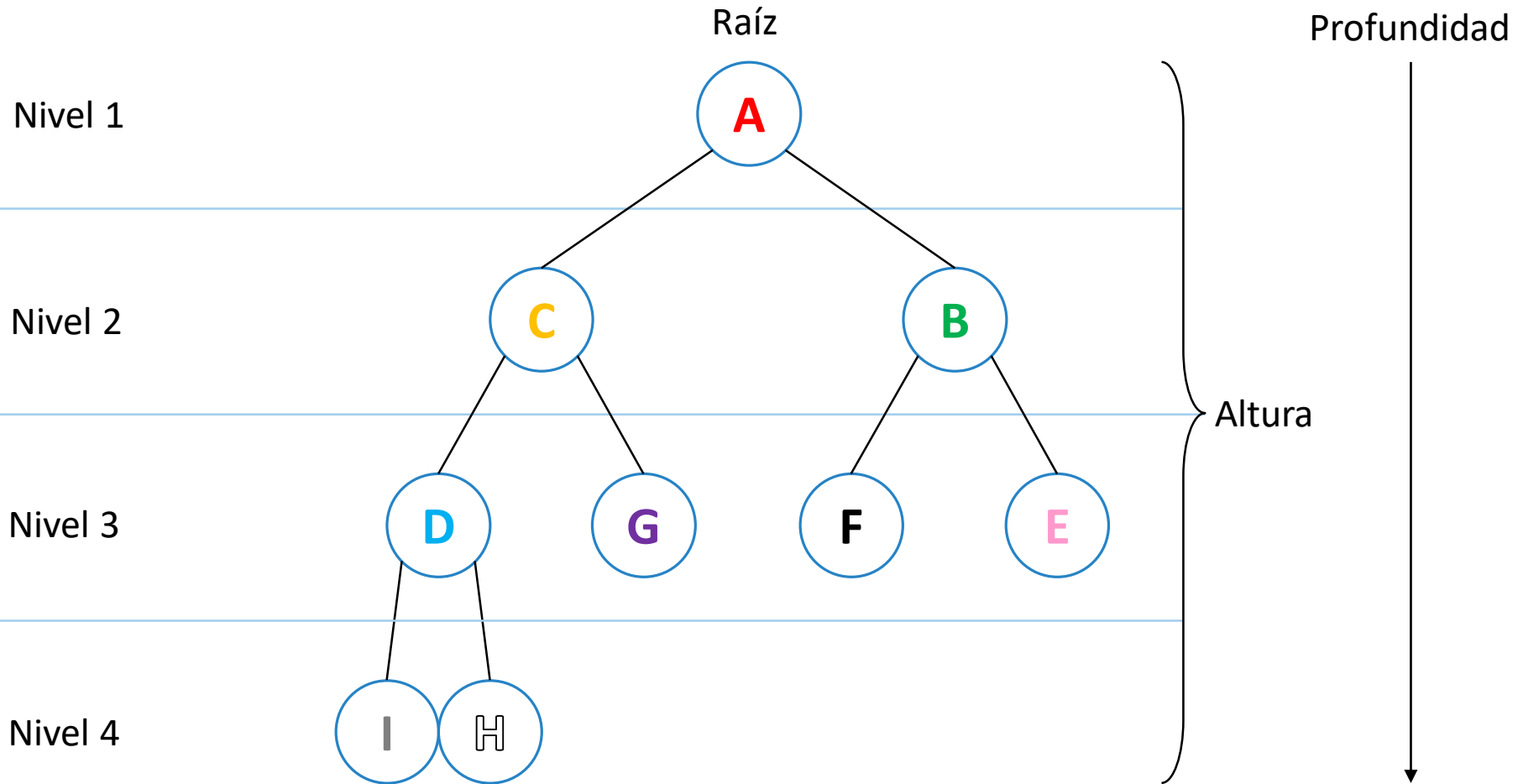
Es un **árbol binario**, con el elemento más prioritario como raíz:

- ( ver próx. diapositiva )

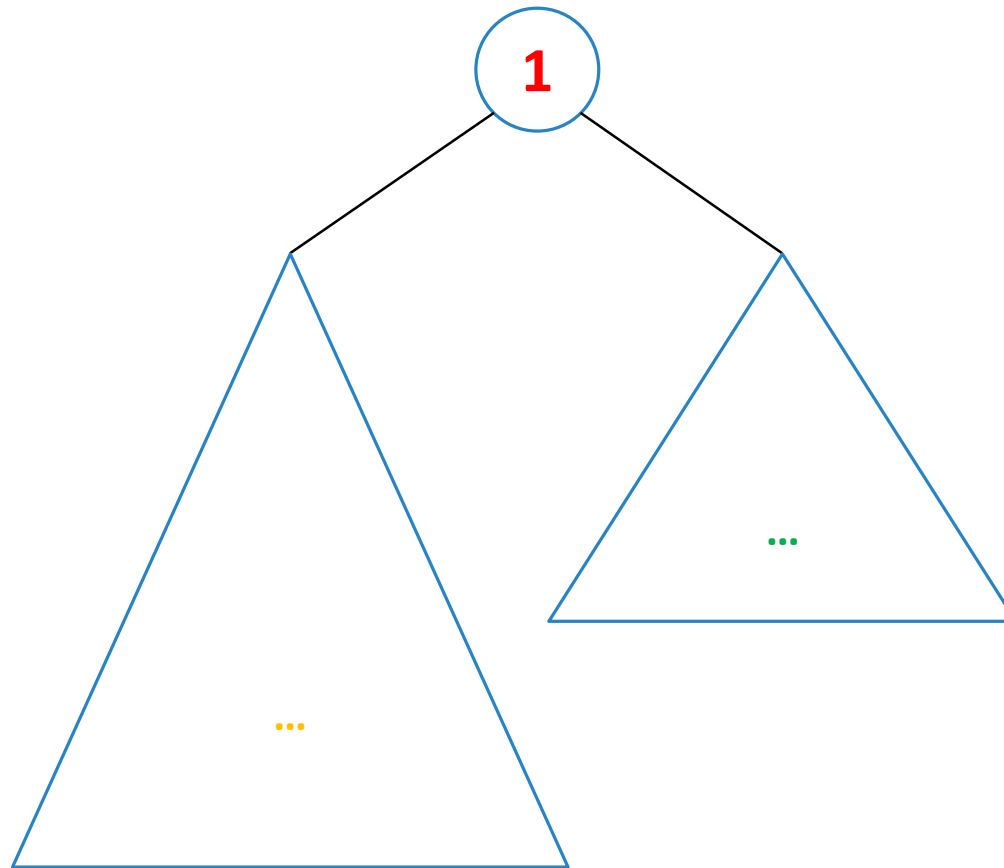
Los demás datos están divididos en dos grupos:

- ( ver diapositivas subsiguientes )
- cada grupo está organizado a su vez —recursivamente— como un heap binario
- estos dos heaps binarios cuelgan de la raíz como sus hijos

# Anatomía de un heap binario



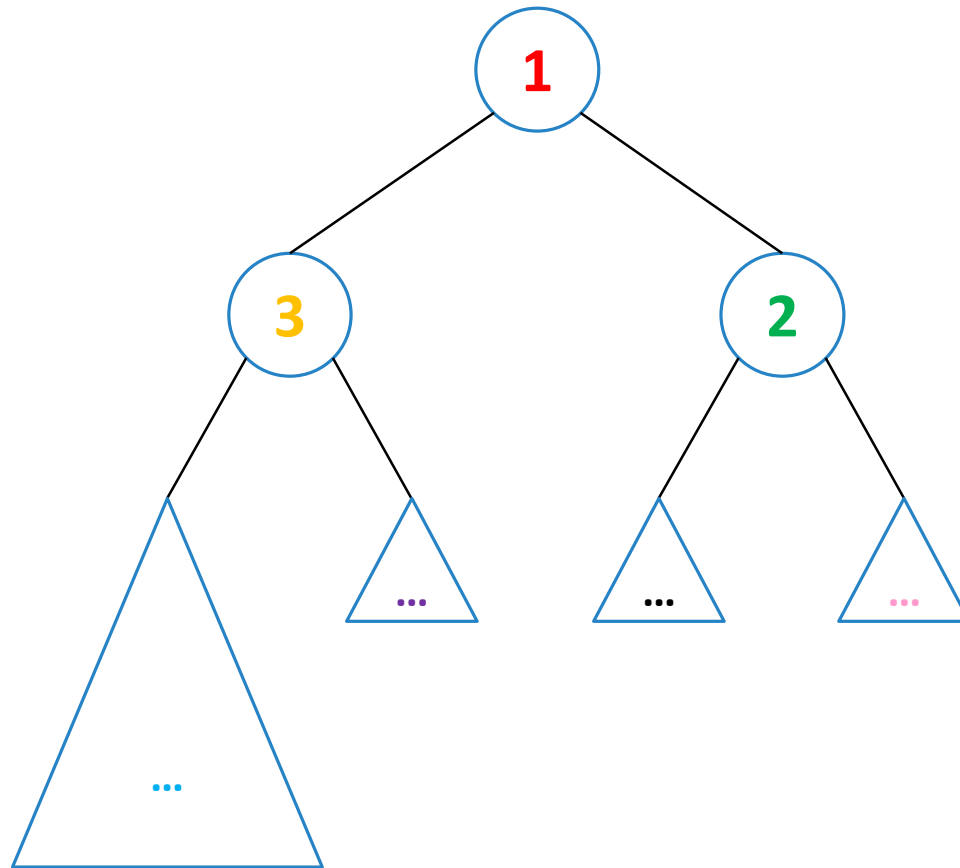
# Un heap binario: La raíz y sus dos hijos



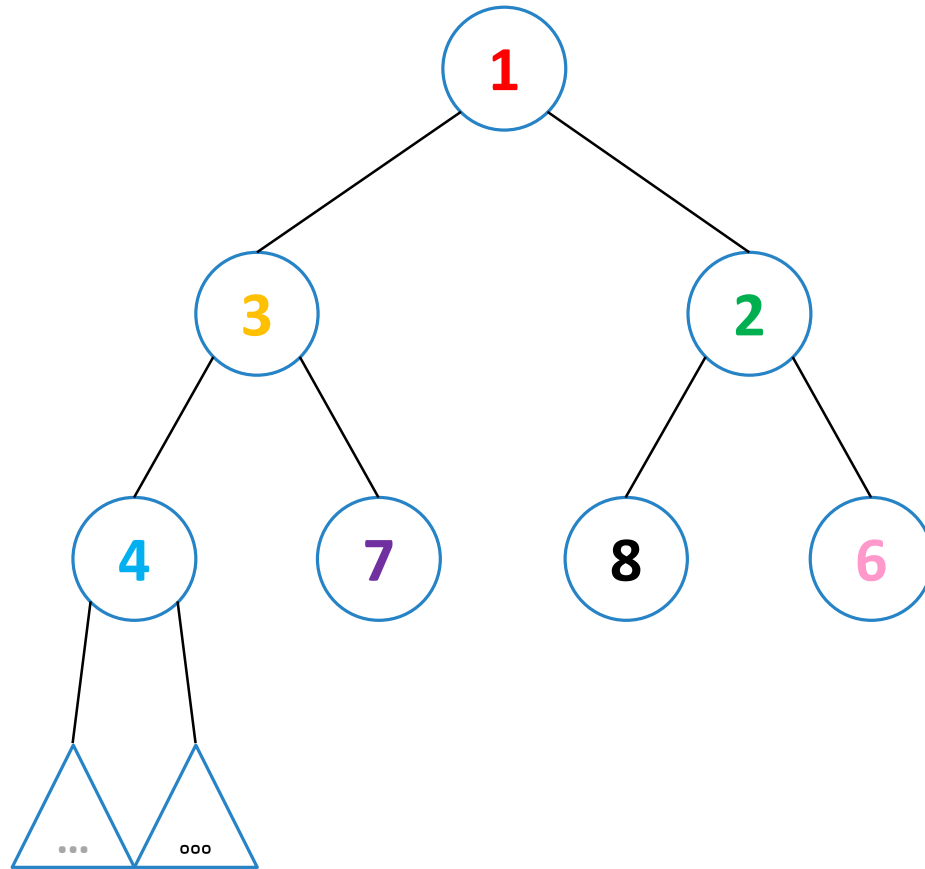


# Un heap binario:

Cada hijo de la raíz es un heap binario



# Un heap binario: Recursivo a todo nivel



# Operaciones del heap



Al insertar y extraer elementos, el heap debe reestructurarse para conservar sus propiedades

¿Cómo se definen estas operaciones (de manera recursiva)?

Idealmente queremos que el heap se vaya llenando por nivel

*extract*( $H$ ):

$best \leftarrow H.root, \quad next \leftarrow \emptyset$

*if*  $H$  tiene un hijo,  $H'$ :

$next \leftarrow \textcolor{teal}{extract}(H')$

*else if*  $H$  tiene dos hijos:

$H' \leftarrow$  el hijo de  $H$  de mayor prioridad

$next \leftarrow \textcolor{teal}{extract}(H')$

$H.root \leftarrow next$

*return*  $best$

*insert*( $H, e$ ):

*if*  $H$  tiene 0 o 1 hijos:

$H' \leftarrow$  un nuevo hijo para  $H$

$H'.root \leftarrow e$

*else*:

$H' \leftarrow$  el hijo de  $H$  de menor altura

*insert*( $H', e$ )

*if*  $H'.root > H.root$ :

$H'.root \rightleftharpoons H.root$

# Altura de un heap binario



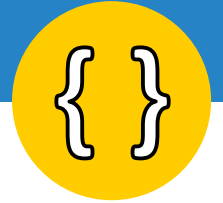
Dado que el número de pasos de los algoritmos depende de la altura del heap,

... ¿cuál es la altura de un heap con  $n$  datos?

Considerando eso,

¿Cuál es la complejidad de sus operaciones?

# Una implementación simple de un heap binario

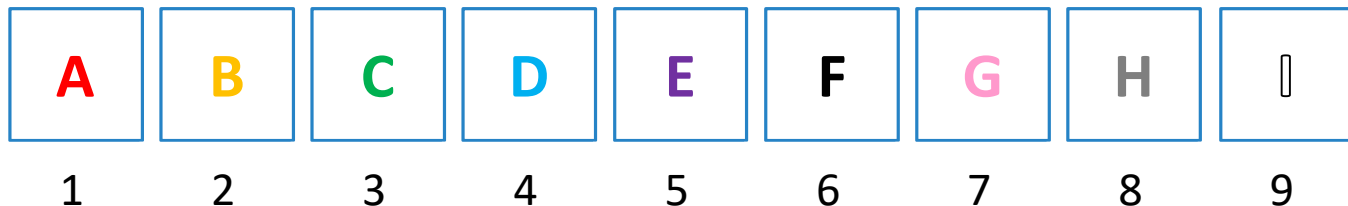
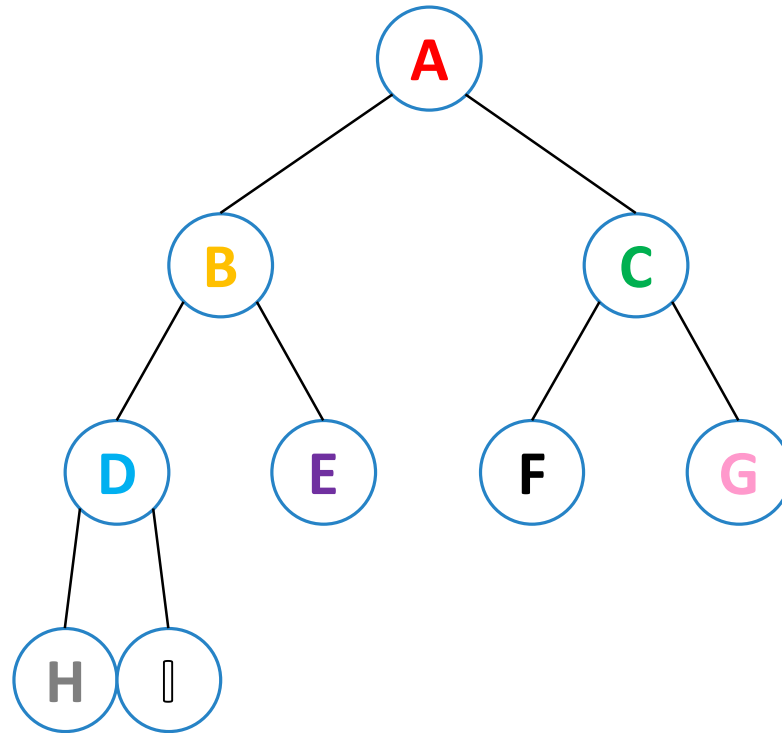


Si podemos suponer una cantidad máxima de datos que pueden estar en el heap simultáneamente

... y como la inserción puede hacerse en cualquier hoja

Entonces, es posible implementar el heap de forma compacta en un **arreglo**

# Un heap binario como un arreglo





# Cambio de prioridad



Supongamos que tenemos acceso directo a un elemento del heap

¿Cómo podemos cambiar la prioridad de ese elemento?

*sift up*( $H, i$ ):

*if*  $i$  tiene padre:

$i' \leftarrow$  el padre de  $i$

*if*  $H[i'] > H[i]$ :

$H[i'] \rightleftharpoons H[i]$

*sift up*( $H, i'$ )

*sift down*( $H, i$ ):

*if*  $i$  tiene hijos:

$i' \leftarrow$  el hijo de  $i$  de mayor prioridad

*if*  $H[i'] > H[i]$ :

$H[i'] \rightleftharpoons H[i]$

*sift down*( $H, i'$ )

# *Selection sort usando heaps ...*



La clase pasada vimos selection sort

¿Será posible usar un heap para mejorar su rendimiento?

# ... se llama *heapsort*

Para la secuencia inicial de datos, A.

1. Convertir A en un min-heap con los datos
2. Definir una secuencia ordenada, B, inicialmente vacía
3. Extraer el menor dato  $x$  de A e insertarlo al final de B
4. Si quedan elementos en A, volver a 2.

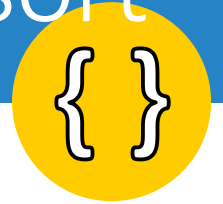
# Propiedades de *heapsort*



¿Cómo se demuestra que heapsort es correcto?

¿Cuál es su complejidad?

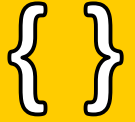
# Necesidad de memoria para heapsort



En la práctica, se usa un mismo arreglo para el arreglo inicial A y el arreglo resultado B

Eso significa que **heapsort** no requiere memoria adicional

# Implementando el *heap* como arreglo



Se puede definir la inserción y la extracción desde *sift up* y *down*.

Esto facilita **mucho** la implementación



*insert*( $H, e$ ):

$i \leftarrow$  *la primera celda en blanco de  $H$*

$H[i] \leftarrow e$

*sift up*( $H, i$ )

*extract*( $H$ ):

$i \leftarrow$  *la ultima celda no vacia de  $H$*

$best \leftarrow H[0]$

$H[i] \rightleftharpoons H[0]$

$H[i] \leftarrow \emptyset$

*sift down*( $H, 0$ )

*return*  $best$