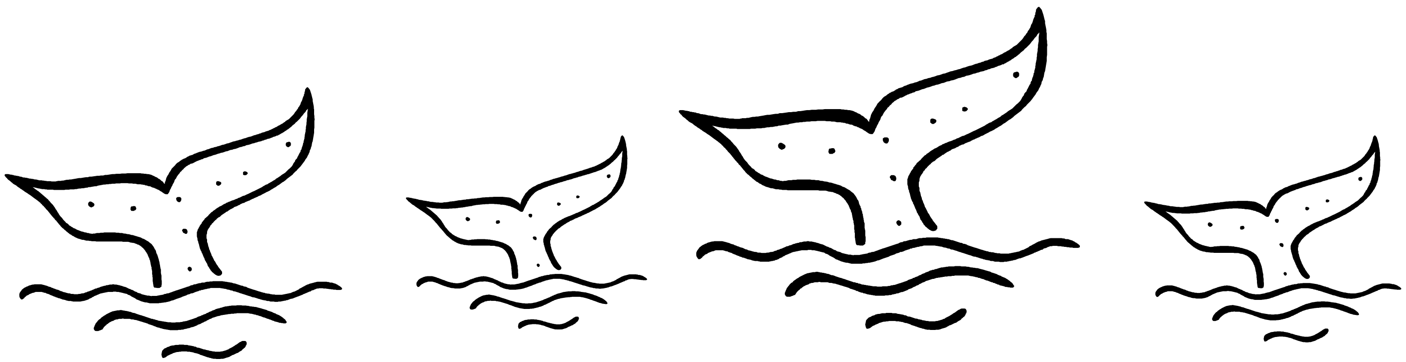
Colas de Prioridad y heaps



Colas de prioridad

- Numerosas aplicaciones
  - Sistemas operativos, algoritmos de scheduling, gestión de colas en cualquier ambiente, etc.
- La prioridad en general la expresamos con un entero, pero puede ser cualquier otro tipo α con un orden $<_\alpha$ asociado.
- Correspondencia entre máxima prioridad y un valor máximo o mínimo del valor del tipo α

TAD Colas de prioridad

```
TAD ColaPrioridad<T> {
   obs s: seq<T>

   proc ColaPrioridadVacía(): ColaPrioridad<T> {
     asegura {res.s = []}
   }
   proc vacía(in c: ColaPrioridad<T>): bool {
     asegura {res = true <==> c.s = []}
   }
   proc apilar(inout c: ColaPrioridad<T>, e: T) {
      requiere {c = C_0}                             â^§
     asegura {c.s = C_0.s + [e]}
  }
  proc desapilarMax(inout c: ColaPrioridad<T>): T {
    requiere {c.s ≠ [] ∧ c = C_0}
    asegura {esMax(C_0.s, res)}
    asegura {(∃i: int)(0 ≤ i < |C_0.s| ∧_L C_0.s[i] = res ∧
          c.s = subsec(C_0.s, 0, i) + subsec(C_0.s, i+1, |C_0.s|) )}
   }
   pred esMax(s: seq<T>, res: T) { res ∈ s ∧ (∀e: T)(e ∈ s →_L e ≤ res)}
}
```

Representación de Colas de Prioridad

- La implementación más eficiente es a través de **heaps**
- Heap significa, literalmente, “montón”

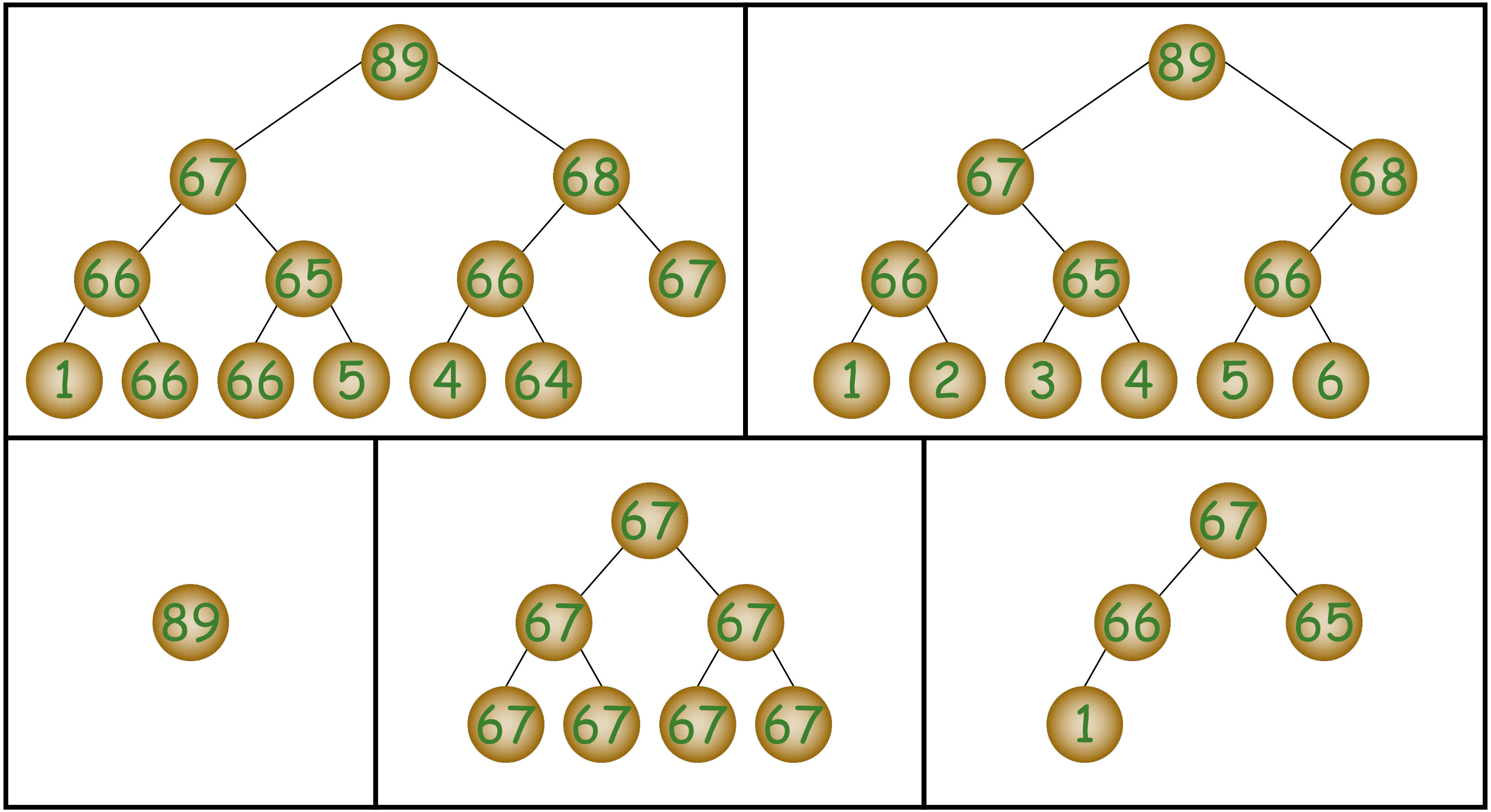
Representación de Colas de Prioridad

- La implementación más eficiente es a través de <u>**heaps**</u>
- Heap significa, también, “parva”, o sea “montón de paja”

Representación de Colas de Prioridad

- Cola de prioridad(α,$<_{\alpha}$) se representa con heap
- Invariante de representación (<u>condición de heap</u>)
  1. Árbol binario perfectamente balanceado
  2. La clave (prioridad) de cada nodo es mayor o igual que la de sus hijos, si los tiene
  3. Todo subárbol es un heap
  4. (no obligatorio): es “izquierdista”, o sea, el último nivel está lleno desde la izquierda.
- (Ojo: ¡no es un ABB, ni una estructura totalmente ordenada!)
- Función de abstracción:
  - Ejercicio (fácil)

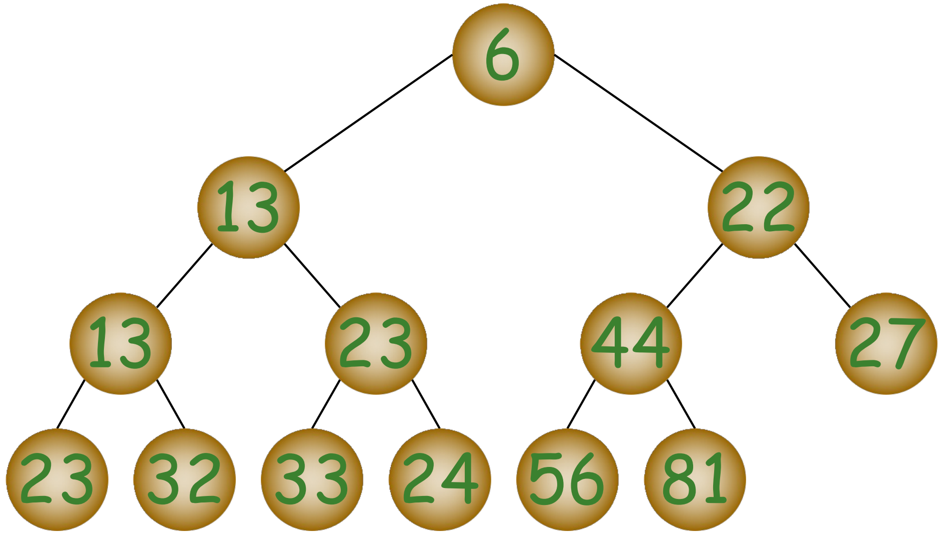
¿Son heaps?



89
67
68
66
65
66
67
1
66
66
5
4
64
89
67
68
66
65
66
1
2
3
4
5
6
89
67
67
67
67
67
67
67
67
66
65
1

max- y min-heap

- La estructura que estamos usando se llama *max-heap*
- Variante: *min-heap*
  - Cambiar “mayor” por “menor”



min-heap

Operaciones sobre un (max-)heap

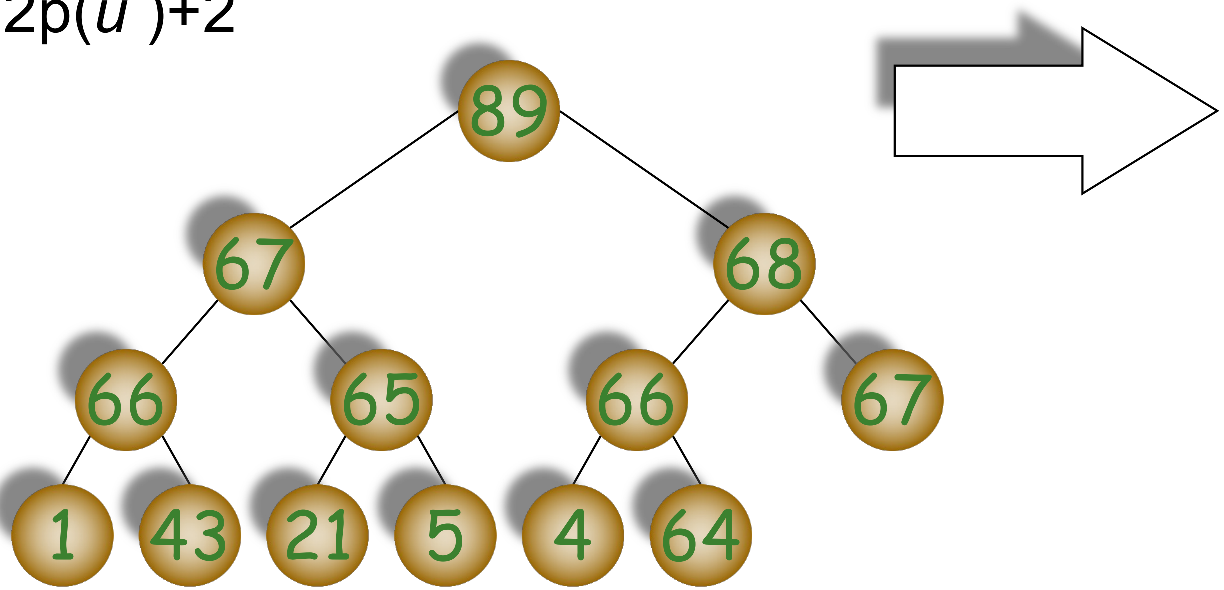
- Básicamente, las mismas que tenemos definidas en el TAD Cola de Prioridad:
  - Vacía: crea un heap vacío
  - Próximo: devuelve el elemento de máxima prioridad, sin modificar el heap.
  - Encolar: agrega un nuevo elemento, hay que restablecer el invariante
  - Desencolar: elimina el elemento de máxima prioridad, hay que restablecer el invariante

Implementación de heaps

- Todas las representaciones usadas para árboles binarios son admisibles
  - representación con punteros, eventualmente con punteros hijo-padre
  - representación con arrays
    - particularmente eficiente

Representación con arrays

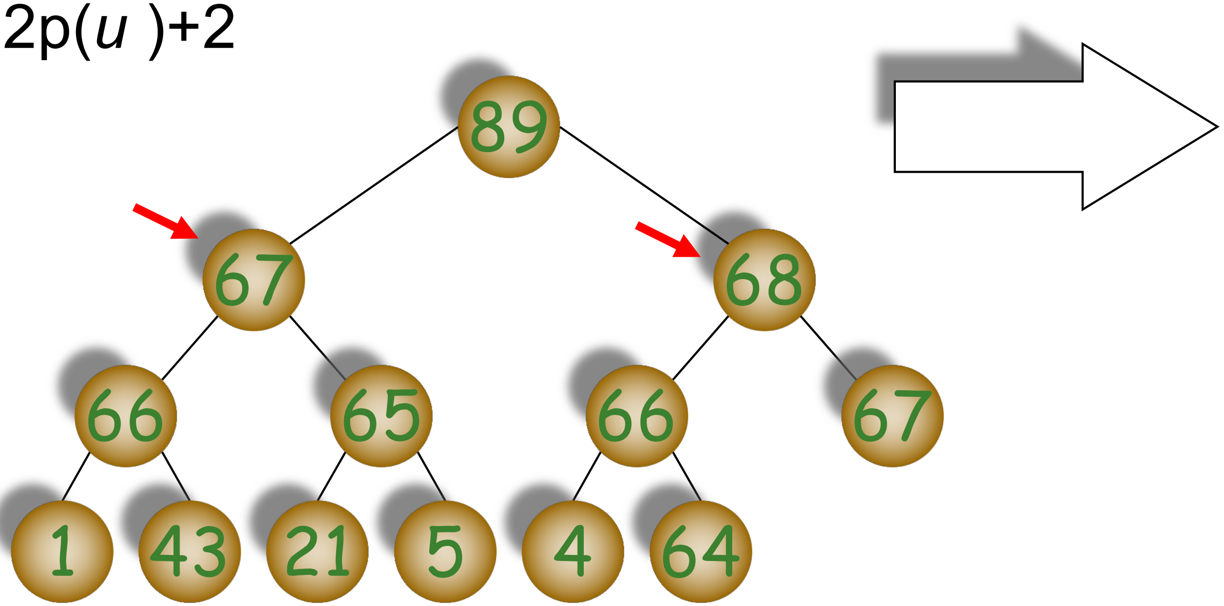
- Cada nodo *v* es almacenado en la posición p(*v*)
  - si *v* es la raíz, entonces p(*v*)=0
  - si v es el hijo izquierdo de *u* entonces p(*v*)=2p(*u*)+1
  - se *v* es el hijo derecho de *u* entonces p(*v*)=2p(*u*)+2

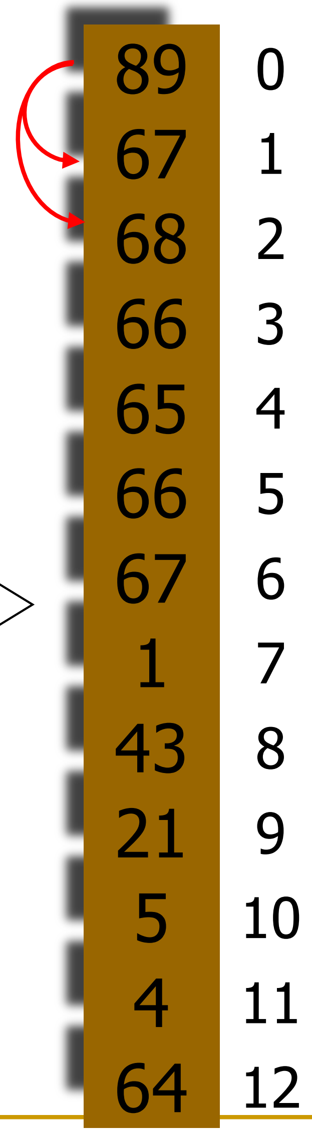


89	0
67	1
68	2
66	3
65	4
66	5
67	6
1	7
43	8
21	9
5	10
4	11
64	12

Representación con arrays

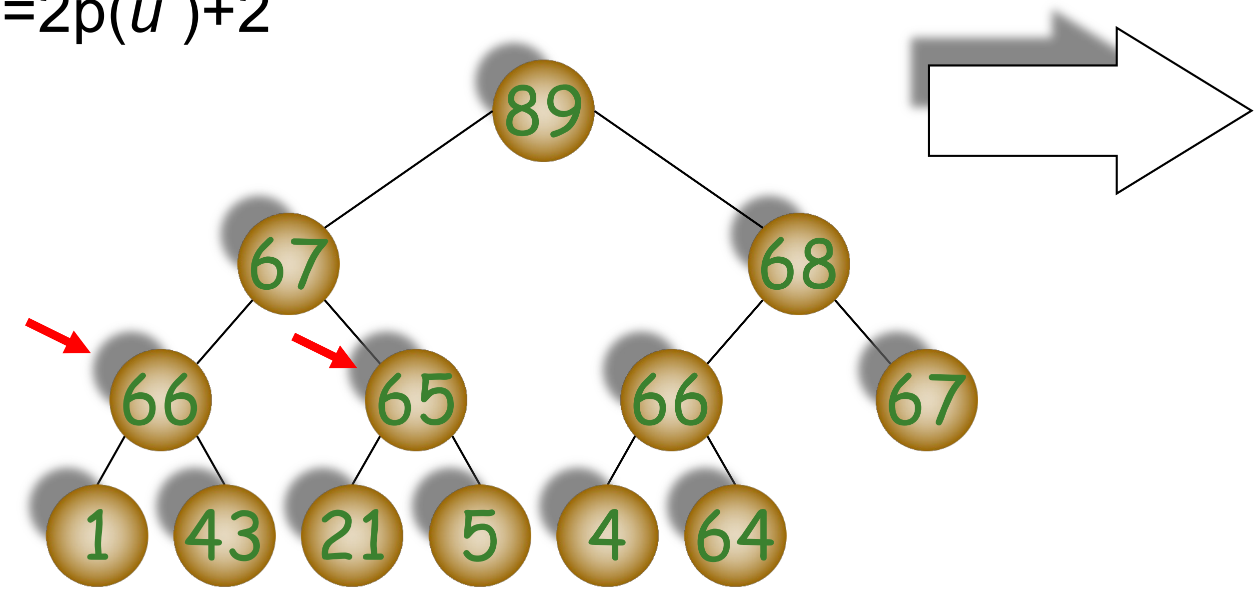
- Cada nodo *v* es almacenado en la posición p(*v*)
  - si *v* es la raíz, entonces p(*v*)=0
  - si v es el hijo izquierdo de *u* entonces p(*v*)=2p(*u*)+1
  - se *v* es el hijo derecho de *u* entonces p(*v*)=2p(*u*)+2

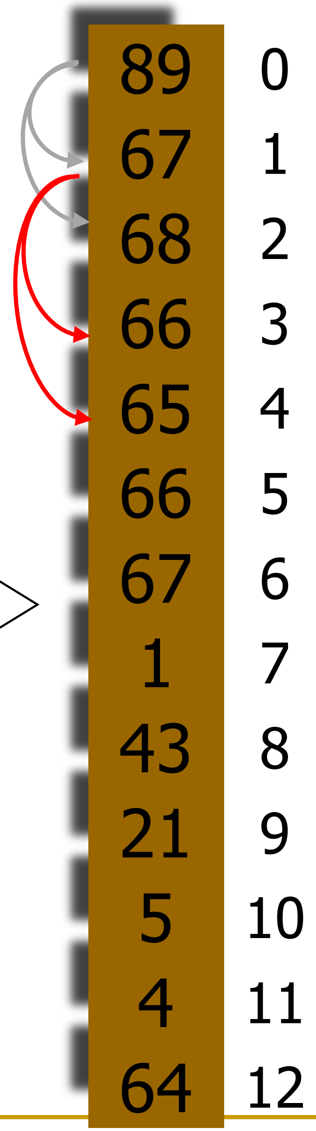




Representación con arrays

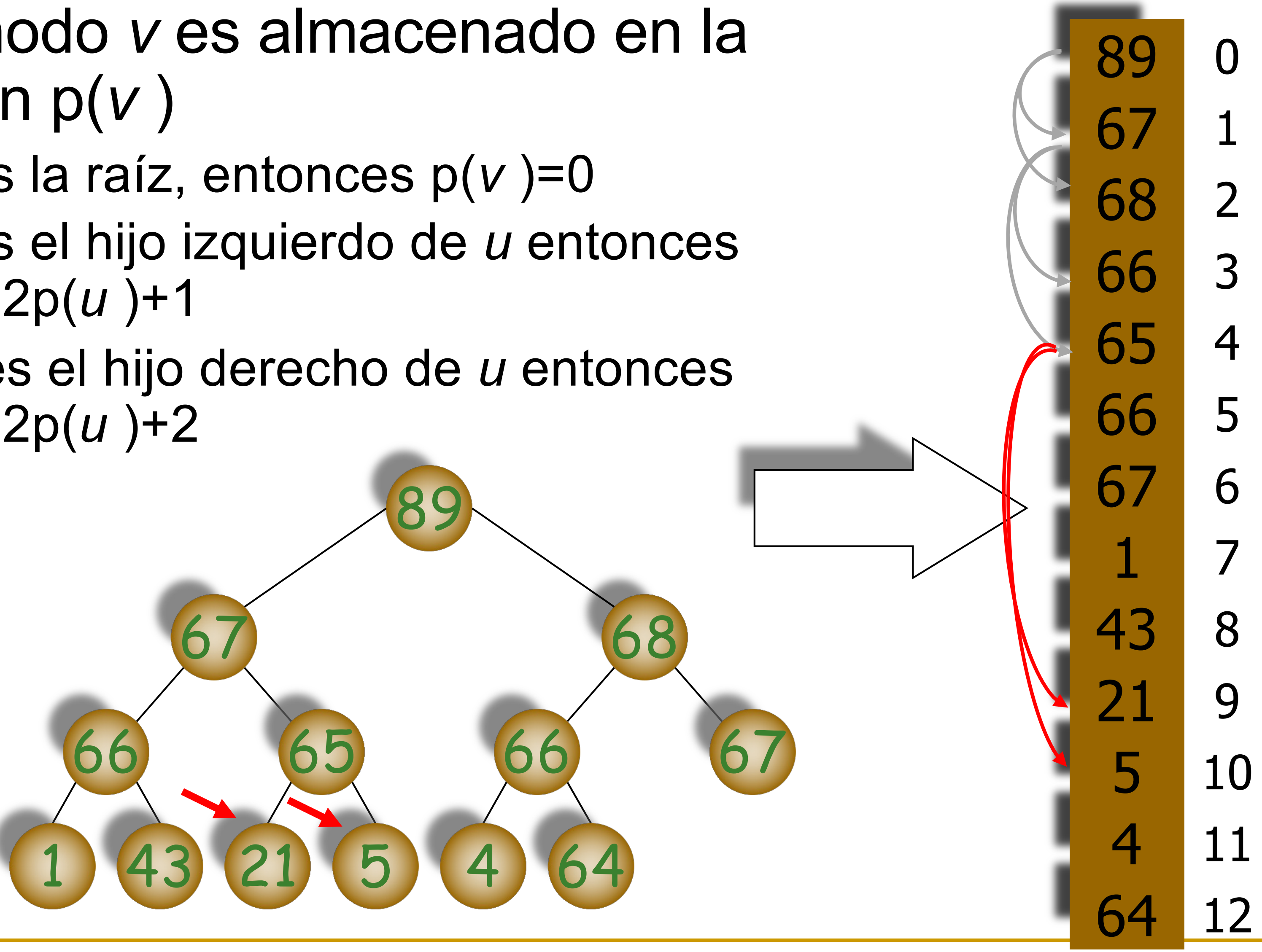
- Cada nodo *v* es almacenado en la posición p(*v*)
  - si *v* es la raíz, entonces p(*v*)=0
  - si v es el hijo izquierdo de *u* entonces p(*v*)=2p(*u*)+1
  - se *v* es el hijo derecho de *u* entonces p(*v*)=2p(*u*)+2





Representación con arrays

- Cada nodo *v* es almacenado en la posición p(*v*)
  - si *v* es la raíz, entonces p(*v*)=0
  - si v es el hijo izquierdo de *u* entonces p(*v*)=2p(*u*)+1
  - se *v* es el hijo derecho de *u* entonces p(*v*)=2p(*u*)+2



Heaps sobre arrays

- Ventajas
  - Muy eficientes en términos de espacio (¡ver desventajas!)
  - Facilidad de navegación
    - padre $i \rightarrow$ hijos j_{izq} y j_{der}
      - $j_{izq} = 2i + 1$, $j_{der} = 2i + 2$
    - hijo $i \rightarrow$ padre j
      - $j = \lfloor (i - 1) / 2 \rfloor$
- Desventaja
  - Implementación estática (puede ser necesario duplicar el arreglo (o achicarlo) a medida que se agregan/eliminan elementos

Algoritmos

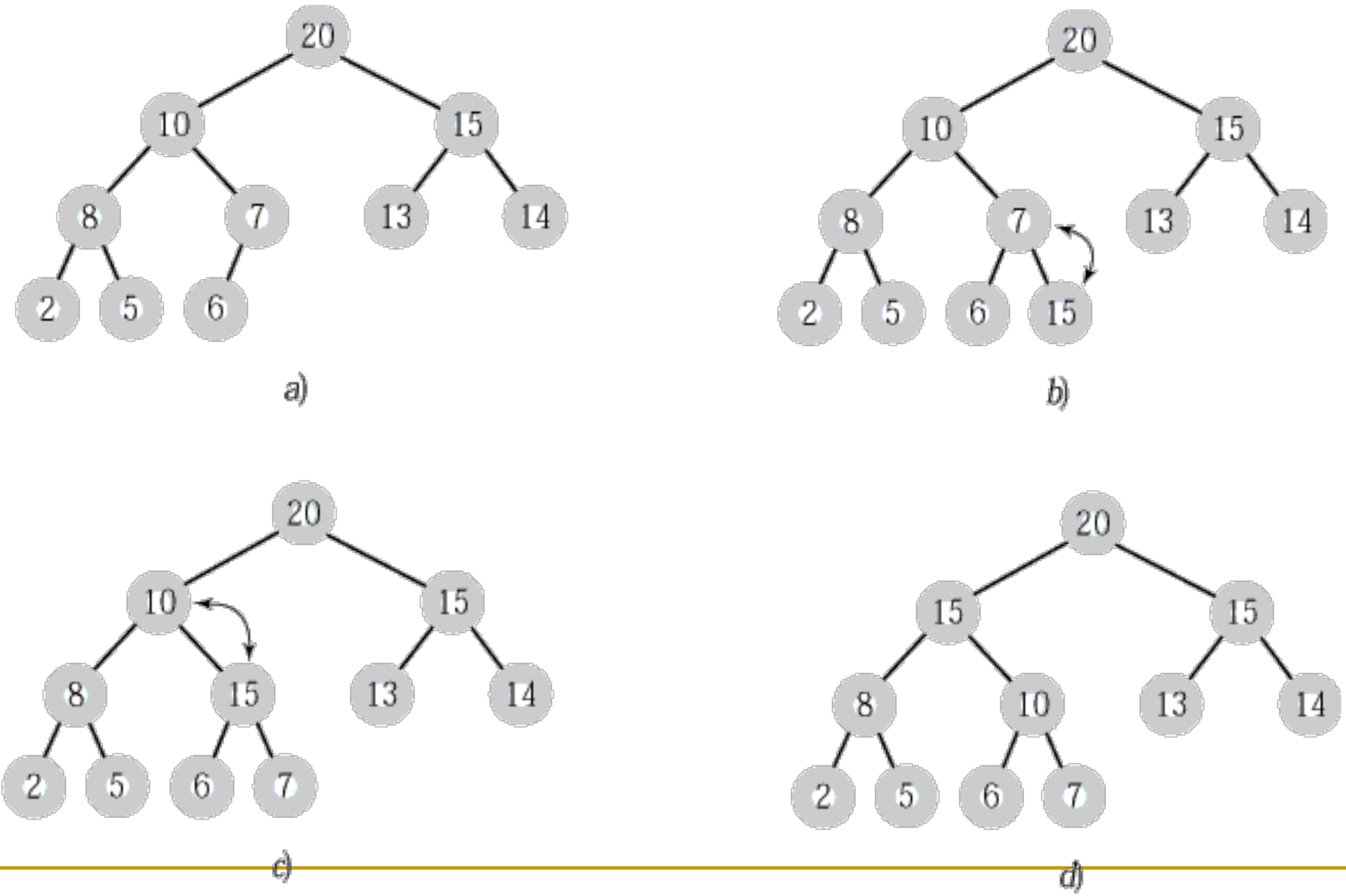
- Próximo:
  - El elemento de prioridad máxima está en la posición 0 del arreglo
  - Operación de costo constante O(1)

Algoritmo Encolar

- Encolar(elemento)
  - Insertar elemento al final del heap
  - Subir (elemento)

- Subir(elemento)
  - while (elemento no es raíz) y_L (prioridad(elemento) > prioridad(padre(elemento)))
    - Intercambiar elemento con padre

Algoritmo Encolar (ejemplo)

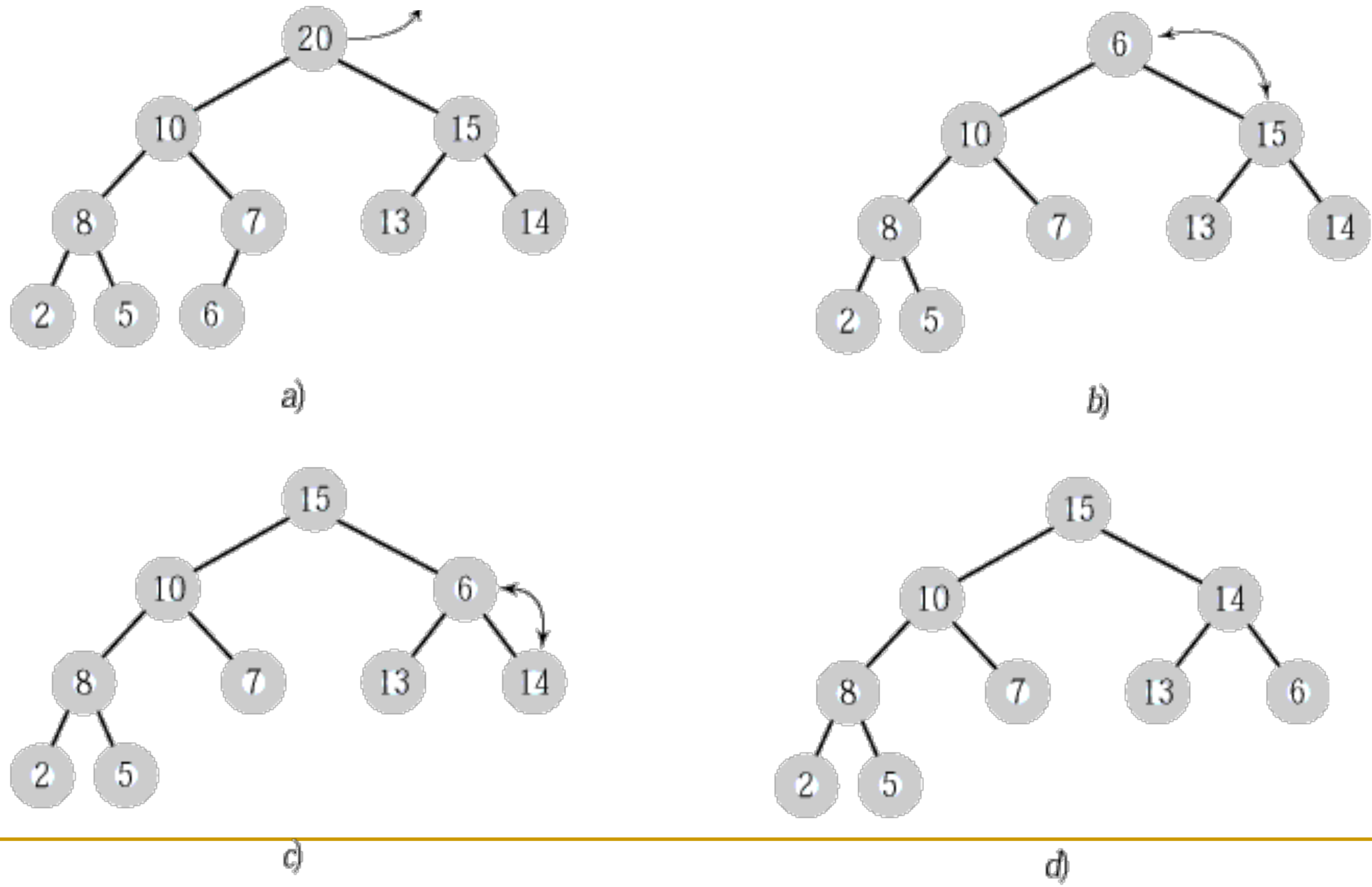


Algoritmo Desencolar

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja
  - Bajar(raíz)

- Bajar(p)
  - while (p no es hoja) y_L (prioridad(p) < prioridad(algún hijo de p))
    - Intercambiar p con el hijo de mayor prioridad

Algoritmo Desencolar (ejemplo)



Costos

- Tanto para encolar como para desencolar, proporcionales a la altura del heap, que es.....

$$O(\lg n)$$

Array2Heap

- Dado un array arr, lo trasforma en un array que representa un heap a través de una permutación de sus elementos
- Algoritmo simple

```
para i desde 1 hasta tam(arr)
    encolar(arr[i]);
```

Array2Heap

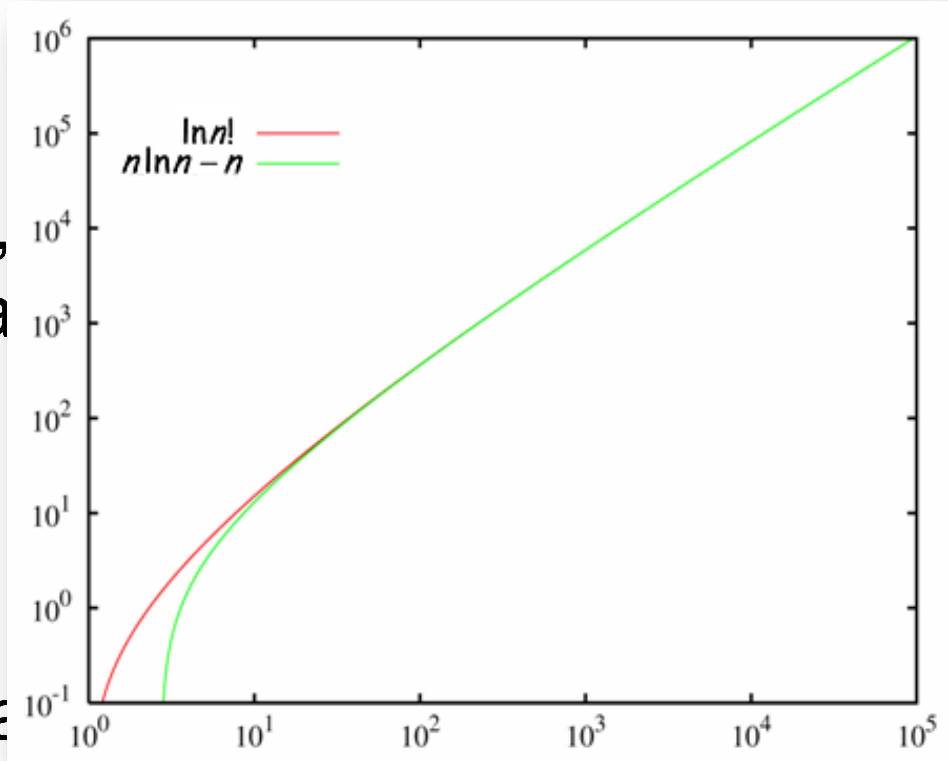
- Dado un array arr, lo trasforma en un array que representa un heap a través de una permutación de sus elementos
- Algoritmo simple

```
        para i desde 1 hasta tam(arr)
            encolar(arr[i]);
```

- Costo (utilizando la aproximación de Stirling del factorial):

$$\sum_{i=1}^{n} \lg i = \lg n! = \frac{\ln n!}{\ln 2} \approx \frac{1}{\ln 2}(n \ln n - n) = \Theta(n \lg n)$$

Array2Heap



- Dado un array arr, e
  representa un hea ón
  de sus elementos
- Algoritmo simple

  para i desde

  encolar(a
- Costo (utilizando la [aproximación de Stirling]() del factorial):

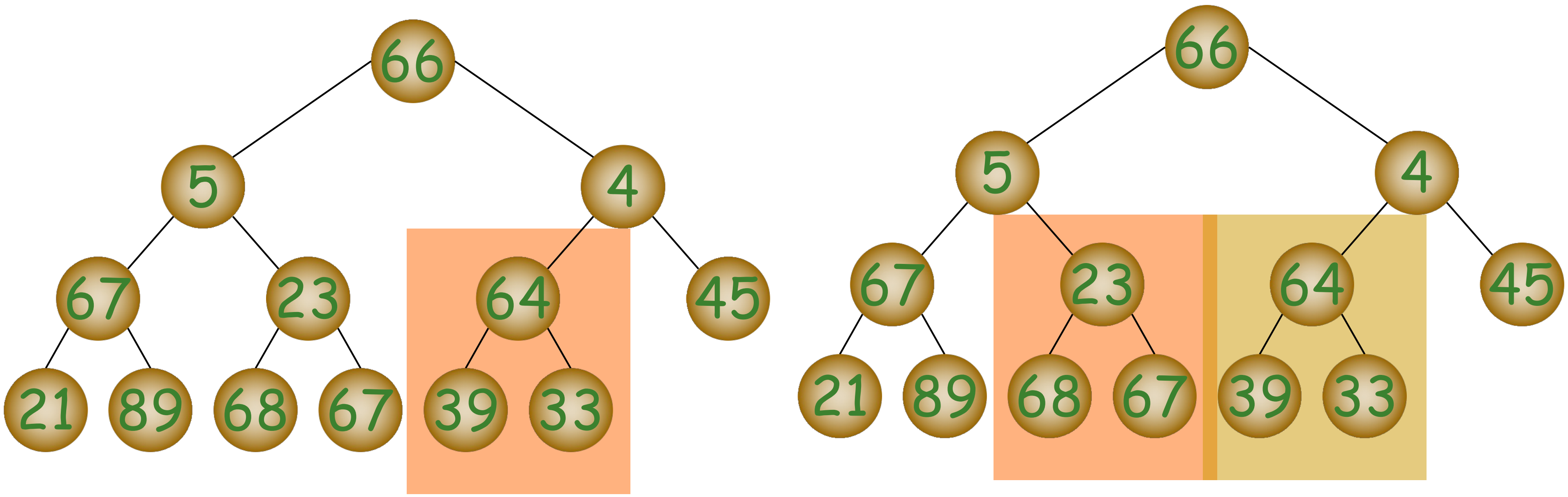
$$\sum_{i=1}^{n} \lg i = \lg n! = \frac{\ln n!}{\ln 2} \approx \frac{1}{\ln 2}(n \ln n - n) = \Theta(n \lg n)$$

Array2Heap/2

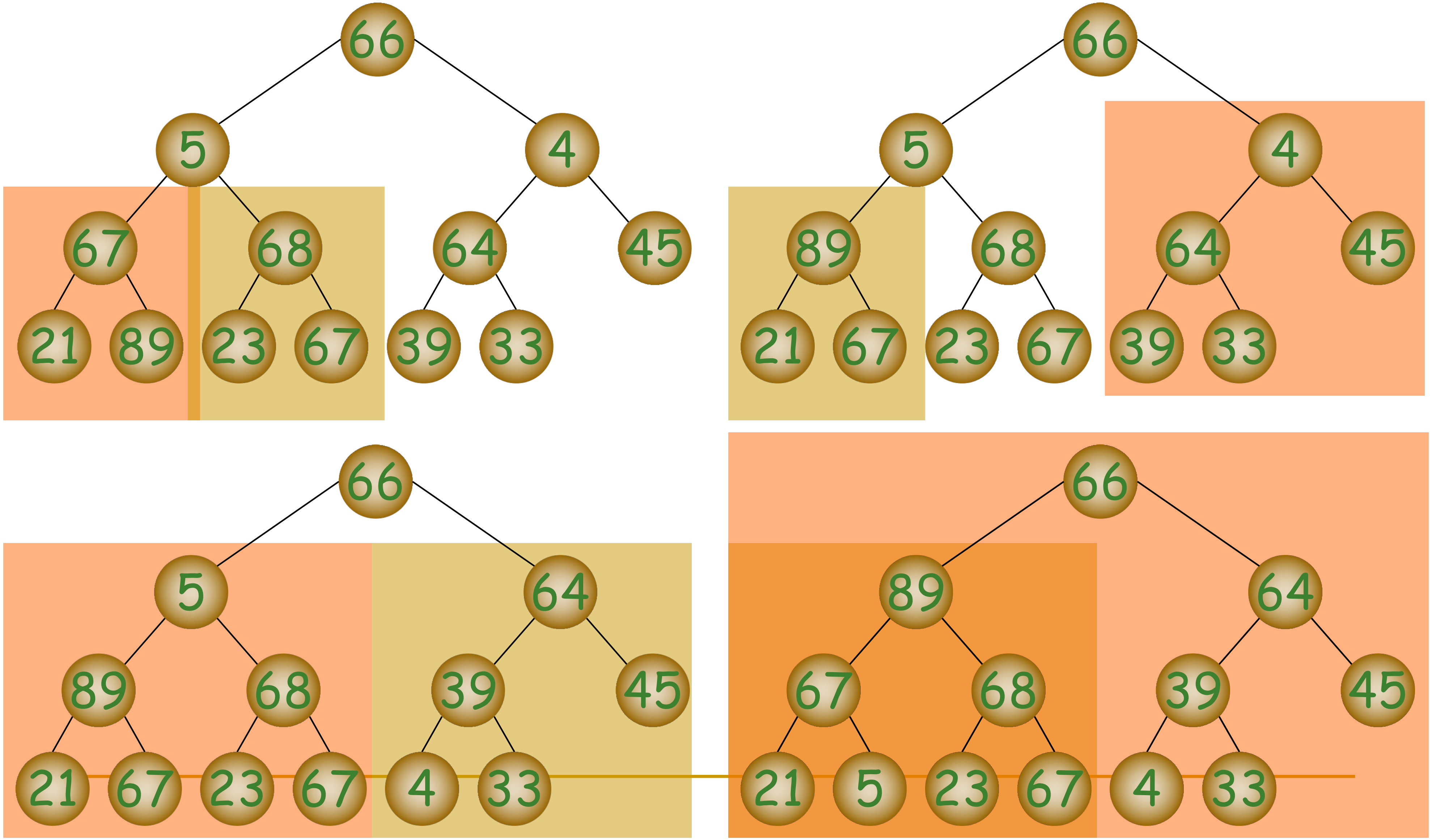
- Podemos hacer algo mejor.
- ¿Dónde están la mayoría de los nodos?
- La idea es aplicar la operación bajar a árboles binarios tales que los hijos de la raíz son raíces de heaps.
- Progresivamente se "heapifican" ("heapify") los subárboles con raíz en el penúltimo nivel, luego los del antepenúltimo, etc.
  - Estrategia bottom-up
- Algoritmo de Floyd

algoritmo de Floyd

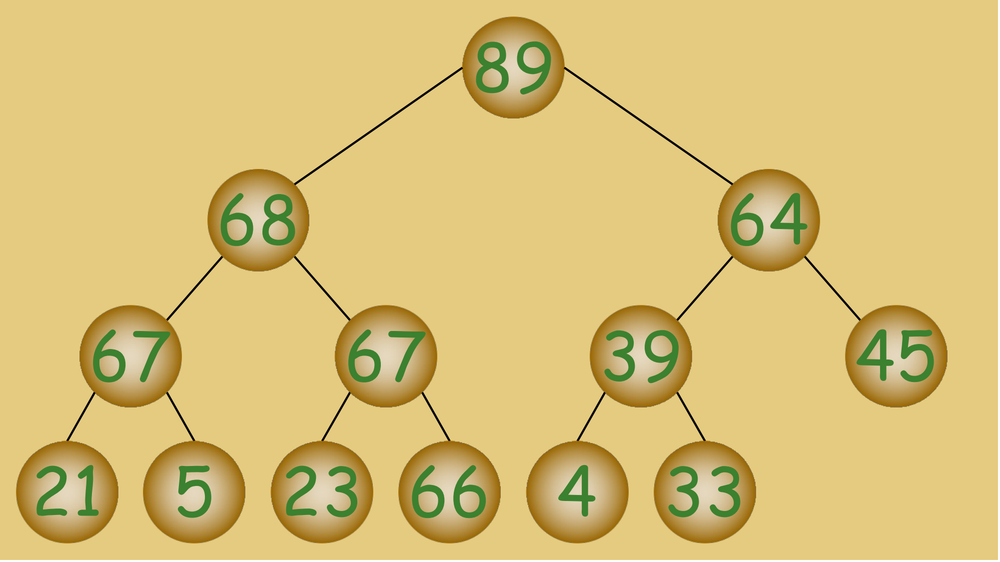
0	1	2	3	4	5	6	7	8	9	10	11	12
66	5	4	67	23	64	45	21	89	68	67	39	33



algoritmo de Floyd/2



algoritmo de Floyd/3



0	1	2	3	4	5	6	7	8	9	10	11	12
89	68	64	67	67	39	45	21	5	23	66	4	33

Análisis del Algoritmo de Floyd

- Caso peor: cada llamada a bajar hace el máximo número de intercambios
- Suponemos un heap con $n = 2^k - 1$ nodos (árbol binario completo de altura k)
  - En el último nivel hay $(n+1)/2$ hojas
  - En el penúltimo nivel hay $(n+1)/4$ nodos
  - En el antepenúltimo nivel hay $(n+1)/8$
  - Y así sucesivamente....

$$\boxed{n_h = n_i + 1}$$

Análisis del algoritmo de Floyd/2

- Una llamada de bajar sobre un nodo de nivel i, provoca como máximo k – i intercambios (op. dominante)
  - 1 intercambio si i es penúltimo nivel, 2 si i es el antepenúltimo, …, k-1 intercambios si i=1
- # max de intercambios =

(# nodos en el penúltimo nivel)·1 +

(# nodos en el antepenúltimo nivel)·2 + … +

(# nodos en el nivel 2)·(k-2) +

(# nodos en el nivel 1)·(k-1), con k = lg(n+1)

Análisis del algoritmo de Floyd/2

- # max de intercambios =
  ((n + 1) / 4) · 1 + ((n + 1) / 8) · 2 + … +
  2 · (lg(n + 1) - 2) + 1·(lg(n + 1) - 1)
- # max de intercambios =

$$\sum_{i=2}^{\log(n+1)} \frac{n+1}{2^i}(i-1) = (n+1)\sum_{i=2}^{\log(n+1)} \frac{i-1}{2^i}$$

$$= (n+1)\left(\sum_{i=2}^{\log(n+1)} \frac{i}{2^i} - \sum_{i=2}^{\log(n+1)} \frac{1}{2^i}\right)$$

Análisis del algoritmo de Floyd/3

considerando que

$$\sum_{i=2}^{\infty} \frac{i}{2^i} = \frac{3}{2} \qquad \sum_{i=2}^{\log(n+1)} \frac{1}{2^i} > 0$$

Deducimos que

$$(n+1)\sum_{i=2}^{\log(n+1)} \frac{i-1}{2^i} < (n+1)\left(\frac{3}{2} - \sum_{i=2}^{\log(n+1)} \frac{1}{2^i}\right) < \frac{3}{2}(n+1)$$

$\Rightarrow$ # max de intercambios = O(n)

Aplicaciones del algoritmo de Floyd

- Implementación de operaciones no standard
  - Eliminación de una clave cualquiera
  - Pueden ser requeridas en algún contexto
  - Ejemplo: kill de un proceso dado su PID
  - O(*n*) para encontrar la clave, O(1) para eliminarla, O(*n*) para reconstruir el invariante de representación con el algoritmo de Floyd
- Ordenamiento de un array
  - Ya lo veremos......

HeapSort

- Valor de las estructura de datos: Selection Sort usa n búsquedas de mínimo. ¿Cómo se hacía eso eficientemente?
- Podemos meter los elementos del arreglo uno a uno en un heap, y luego ir sacándolos.
- Pero: ¿se puede hacer algo todavía mejor?
- ¿Se acuerdan de la operación Array2Heap y del algoritmo de Floyd? Complejidad: O(n)
- Algoritmo de ordenamiento de un array basado en un heap
- Algoritmo
  - heap ← array2heap (A)
  - para i desde n-1 hasta 0 hacer
    - max ← próximo (heap)
    - desencolar
    - A[i] ←max
- Costo: O(n) + O(n log n)
- Notar que no requiere memoria adicional

Resumen

- Vimos colas de prioridades
- Implementación eficiente de las mismas - Heaps
- Implementación en árboles y arrays
- Maneras de crear eficientemente un heap - Floyd
- HeapSort