Heap (Árbol busqueda binaria)

Conceptos clave

- Colas de prioridad
- Min-Heap
- Max-Heap
- Heapificar (heapify)

Colas de prioridad

- La filosofía de una cola de prioridad se basa en la idea de FIFO (First In First Out)
- Las pilas por otro lado siguen LIFO (Last In First Out)
- Si queremos implementar una cola de prioridad sin conocimientos, se nos ocurriría hacer una lista desordenada -> insertar, complejidad lineal O(n).
- Ordenarla igualmente.
- Podríamos pensar en implementarlo a partir de un AVL (inserción y eliminar en tiempo logaritmico).
- Tenemos una nueva estructura de datos para las colas de prioridad que es particularmente eficiente **HEAP** (montón).
- Tiene la particularidad que es elegante para implementar el TAD cola de prioridad.

Invariante de representación

Recordamos: el invariante de representación es aquello que una estructura debe cumplir para que dicha estructura sea considerada como válida (una instancia válida del tipo).

- En el caso del **HEAP** el invariante de representación nos pide que se cumpla:

-> Sea un arbol binario perfectamente balanceado (con diferencia +- 1 nivel)
-> La prioridad (valor) de cada nodo sea mayor o igual que sus hijos, si es que los tiene.

-> Todo subarbol es un heap (recursivo).
-> Sea "izquierdista", es decir, se vaya completando los niveles de izquierda a derecha.

- Podemos hacer la distinción entre dos tipo de heap: **MAX-HEAP** y **MIN-HEAP**

MAX-HEAP: El elemento de la raíz es el mayor de toda nuestra estructura. La función *this.máximo()* tendrá complejidad O(1) ya que sería devolver S[0].

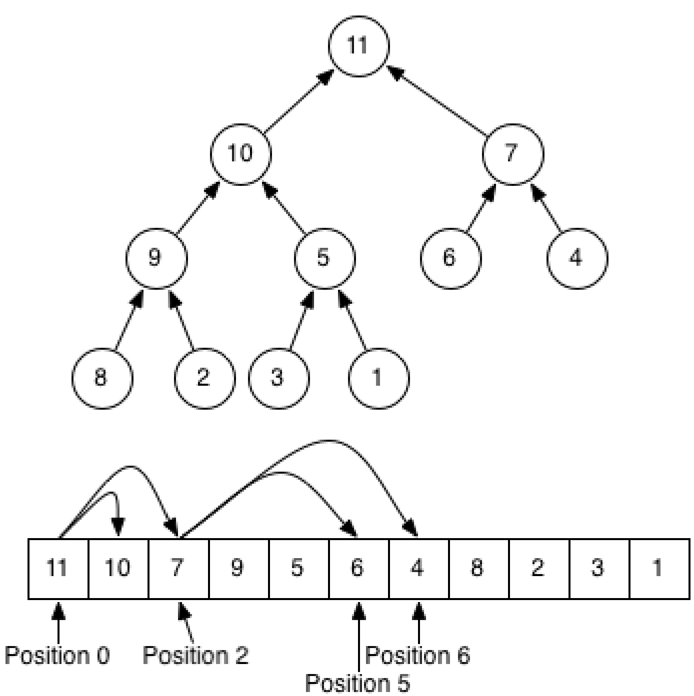
MIN-HEAP: El elemento de la raíz será el menor de toda la estructura. Al igual que máximo, *this.mínimo()* será O(1).

Implementación de mi heap

- Queremos hallar una forma de representar el heap.
- Una buena forma es a partir de arrays

-> Representación con arrays:
Cada nodo V es almacenado en la posición p(V).

- Si V es raíz, entonces p(V) = 0 [se ubica en la posición 0 del array].
- si U es el hijo izquierdo de V, entonces p(U) = 2*P(V) + 1
- si W es el hijo derecho de V, entonces p(W) = 2*P(V) + 2



Beneficios: es particularmente bueno trabajar con una estructura estática a la hora de navegar dentro de la estructura. Muy eficiente en complejidad espacial.

Contras: si tuvieramos que insertar elementos, corremos el riesgo de tener que duplicar el tamaño de mi arreglo (costoso en complejidad temporal y espacial). Tendríamos que duplicarlo cada vez que llenamos.

Algoritmos sobre heaps

- Los siguientes algoritmos van a estar implementados en pseudocódigo.

Siguiente elemento
funcion siguiente(root) {
devuelve el elemento de prioridad máxima -> O(1) -> MAX-HEAP
devuelve el elemento de prioridad mínima -> O(1) -> MIN-HEAP
}

Encolar elemento

- si el nodo a encolar es más grande que su padre, lo reemplazamos por su padre.
  funcion encolar(root) {
  while(!esRaiz(root) && prioridad(elemento) > prioridad(elemento.padre)):
  reemplazo el padre por el elemento y mando el padre a donde estaba el elemento
  }

Desencolar elemento

- Como es FIFO, desencolar es sacar al elemento de amyor prioridad.
- Es importante reconstruir mi arbol de tal forma que al finalizar mi algoritmo se preserve el invariante de representación.

*funcion desencolar(root) {
-> Reemplazar el primer elemento por la última hoja y eliminar la última hoja.
-> bajar(padre)
}

funcion bajar(padre: nodo){
while(!esHoja(padre) && (prioridad(padre) < prioridad(padre.hijoIzquierdo)) || (prioridad(padre) < prioridad(padre.hijoDerecho)))
-> intercambiar p por el hijo de mayor prioridad.

}

Array (cualquiera) a Heap (Array2Heap)

- Partimos por la filosofía de que queremos agarrar un arreglo A = [n0,n1,...,n] tal que el arreglo cumpla con el invariante de representación de un Heap.
- Naturalmente, pensamos en la idea de.

```
While(i < |A|) {
encolar(A[i])
}
```

- Esta idea es buena ya que la complejidad de encolar un Heap es O(log(n)), pero el problema surge cuantos más elementos tengamos en el árbol (es decir, más profundo sea nuestro árbol).

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

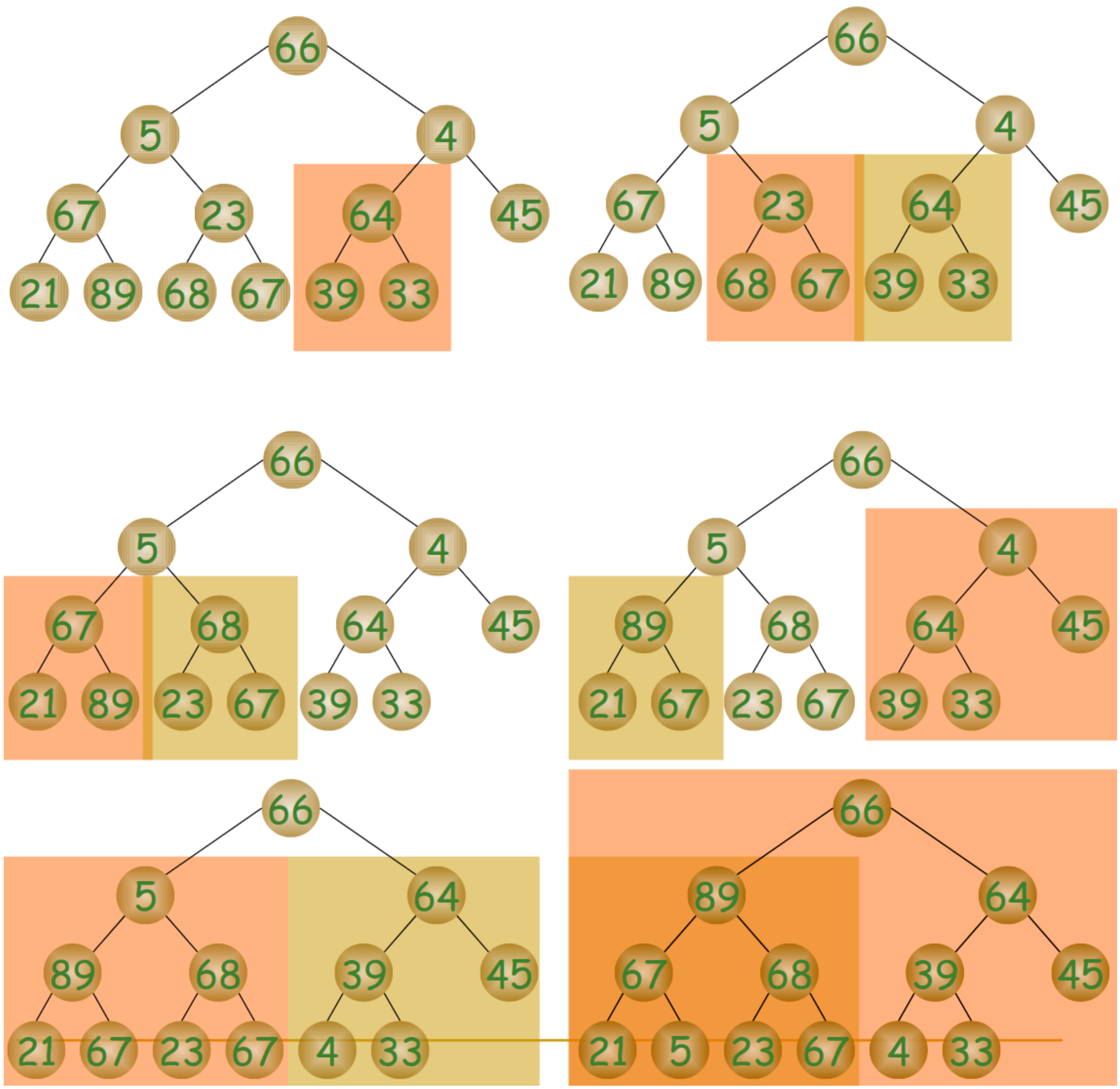
$$\sum_{i=1}^{n} \lg i = \lg n! = \frac{\ln n!}{\ln 2} \approx \frac{1}{\ln 2}(n \ln n - n) = \Theta(n \lg n)$$

¿Podemos mejorar esta implementación para encolar?

Algoritmo de Floyd (Array2Heap V.2)

- Se parte a partir de la estrategia de *bottom-up* (visto en Algoritmos y Estructura de Datos 1), se basa en aplicar la operación *bajar* de árboles binarios tales que los hijos son raices de los Heaps.
- A medida que se ejecuta hacemos la "heapificación" los subárboles con raíz en el penúltimo nivel, antepenúlitimo etc.

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



Algunas aplicaciones del Algoritmo de Floyd (no convencionales)

- Matar un proceso a partir de su PID.
  -> tenemos que "desencolar" el proceso que queremos matar pero luego debemos reestructurar nuevamente la cola de prioridad.
- Eliminación de una clave cualquiera.

- Necesidad del consumidor.