DecreaseKey (x, y, H)

decrementa elem en pos x de heap H en una cant y

IncreaseKey (x, y, H)

incrementa la clave.

DeleteKey (x)

decreaseKey (x, ∞, H)

DeleteMin (H)

insertar en un árbol es de orden log n (log en el peor de los casos recorre desde hoja hasta raíz)

Entonces, para crear un árbol puedo insertar de a uno (n log n) operaciones total

BuildHeap Filtra hacia abajo cada elem.

se elije < de los hijos, se compara el menor con el padre

se empieza filtrando desde lo q' está en tam/2

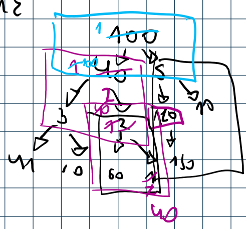
se filtran nodos con hijos, el resto son hojas

100, 40, 5, 32, 120, 10, 41, 50, 60, 1, 150

Heap → datos[ ]
→ cant = 0

→ datos [100, 40, 5, 32, 120, 10, 41, 50, 60, 1, 150]
→ cant = 12

12/2 = 6



un quilombo pero es fácil.

en filtrado de cada nodo recorremos su altura.

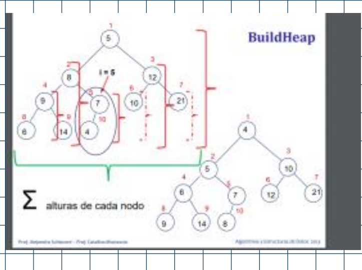
Para calcular tiempo de ejecución, suma alturas de todos los nodos

en cant/2 tengo últimas hojas

me fijo si tengo q' filtrar, elijo al < de los hijos

voy al elem i-1 (4 en el caso), repito

el filtrado hacia abajo para cuando llegué a una hoja.



en árbol binario lleno de altura H con $2^{h+1}-1$ nodos, suma es $\underbrace{2^{h+1}-1}_{\text{cant de nodos}} - (h+1)$

O(n)

Un árbol binario completo tiene entre 2^h y $2^{h+1} - 1$ nodos, el teorema implica que esta suma es de O(n) donde n es el número de nodos.

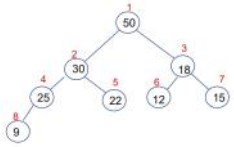
a) *Algoritmo que usa una heap y requiere una cantidad aproximada de (n log n) operaciones.*

- Construir una MinHeap, realizar **n** DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.
  - Desventaja: **requiere el doble de espacio**

b) *Algoritmo **HeapSort** que requiere una cantidad aproximada de (n log n) operaciones, pero **menos espacio**.*

- Construir una MaxHeap con los elementos que se desean ordenar, intercambiar el último elemento con el primero, decrementar el tamaño de la heap y filtrar hacia abajo. Usa sólo el espacio de almacenamiento de la heap.

Ejemplo:



HeapSort (cont.)