

Árboles

EEDD - GRADO EN ING. INFORMÁTICA - UCO

Montículos

ChangeLog

1/4/2025

- Adaptado el diseño de heap para usar un array externo como zona de almacenamiento.

2/4/2025

- Modificada especificación para usar un objeto comparador que induce el orden del heap.

EEDD - GRADO EN ING. INFORMÁTICA - UCO

Motivación

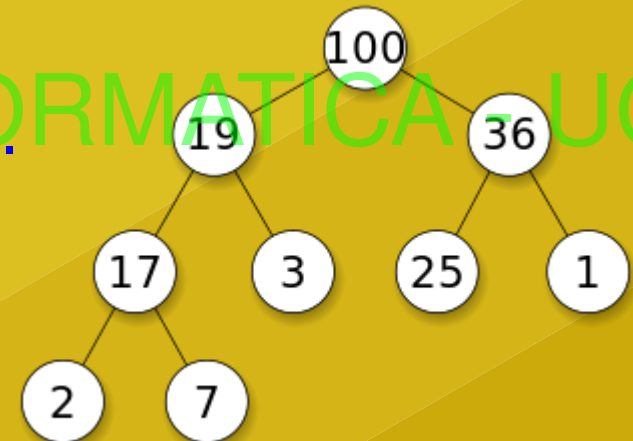
- Queremos diseñar un enrutador de paquetes con prioridad.
- ¿**Qué**? Hay tres operaciones básicas a especificar:
 - **Añadir** un nuevo paquete a la cola.
 - **Consultar** el paquete de mayor prioridad.
 - **Eliminar** el paquete de mayor prioridad.
- ¿**Cómo**? ¿Qué complejidades tendríamos?

Contenidos

- Definición de un montículo.
- Representación acotado de un árbol binario.
- Árboles cuasi-completos.
- Operaciones de flotado y hundimiento.
- Aplicaciones de los montículos: ordenación y colas de prioridad.

Heap

- Montículo: es un árbol binario que mantiene dos invariantes:
 - En cada subárbol, la raíz es $<$ ($>$) que todos sus descendientes.
 - Es un árbol binario cuasi-completo.
- Usado para:
 - Ordenar (algoritmo HeapSort).
 - Implementar una cola de prioridad.



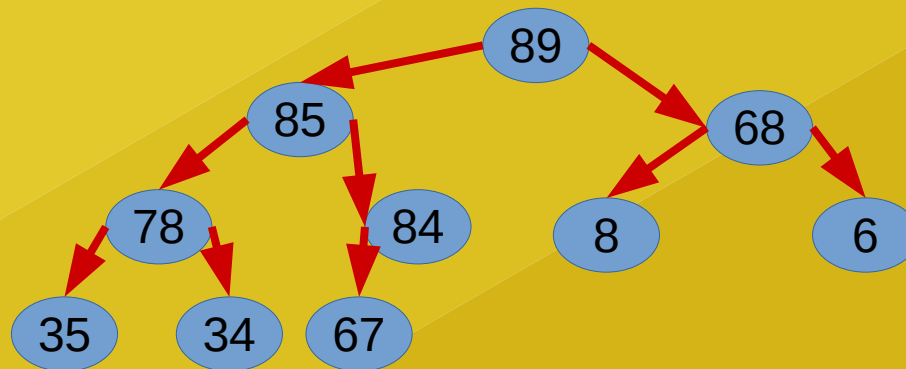
Heap

- Dibuja un árbol binario que cumpla las invariantes de montículo máximo para la secuencia de valores:
 - {68, 34, 78, 84, 67, 6, 8, 35, 89, 85}

¿?

Heap

- Dibuja un árbol binario que cumpla las invariantes de montículo máximo para la secuencia de valores:
 - {68, 34, 78, 84, 67, 6, 8, 35, 89, 85}



Heap

- TAD Heap[G].

- **Makers:**

- make(data:DArray[T], comp:Comp)
 - Post-c: size()=data.size()

- **Observers:**

- isEmpty():Bool
 - size():Integer
 - post-c: isEmpty() || size()>0
 - item():T //return the heap's root item.

- **Mutators:**

- insert(it:T) //insert a new item.
 - Post-c: not isEmpty()
 - Post-c: size()=(old.size()+1)
 - remove() //remove the root node.
 - Pre-c: not isEmpty()
 - Post-c: size()=(old.size()-1)

- **Invariants:**

- It is a complete binary tree.
 - isEmpty() || For each not empty left|right subtree, comp(root item, subtree item)

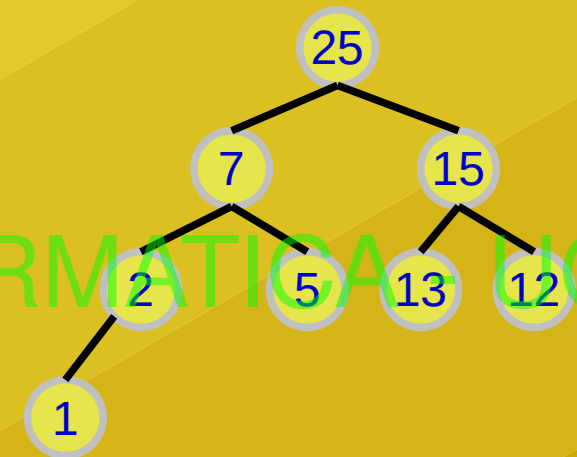
Comp[T]::operator(a:T,b:T):Bool
//Return true if a is before in order to b

Heap

- Operación de inserción:

insert(14)

- insertar en la primera hoja disponible (árbol binario quasi-completo).
- Comprobar invariante en el sub-árbol.
- Si no se cumple, intercambiar el nodo hijo con el padre (“Flotar”) y repetir comprobación en nivel superior.

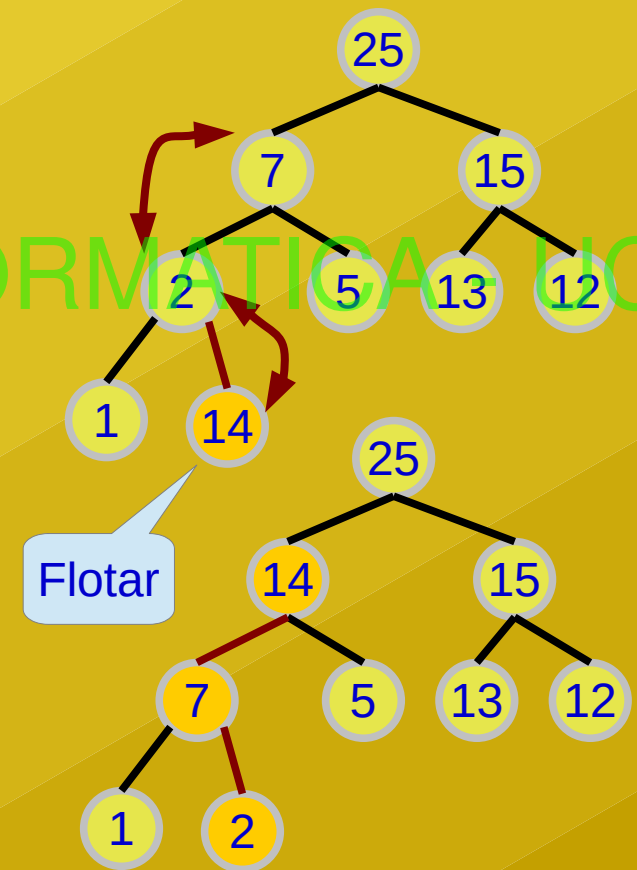


¿?

Heap

- Operación de inserción:
 - insertar en la primera hoja disponible (árbol binario quasi-completo).
 - Comprobar invariante “heap” en el sub-árbol.
 - Si no se cumple, intercambiar el nodo hijo con el padre (“Flotar”) y repetir comprobación en nivel superior.

insert(14)



Heap

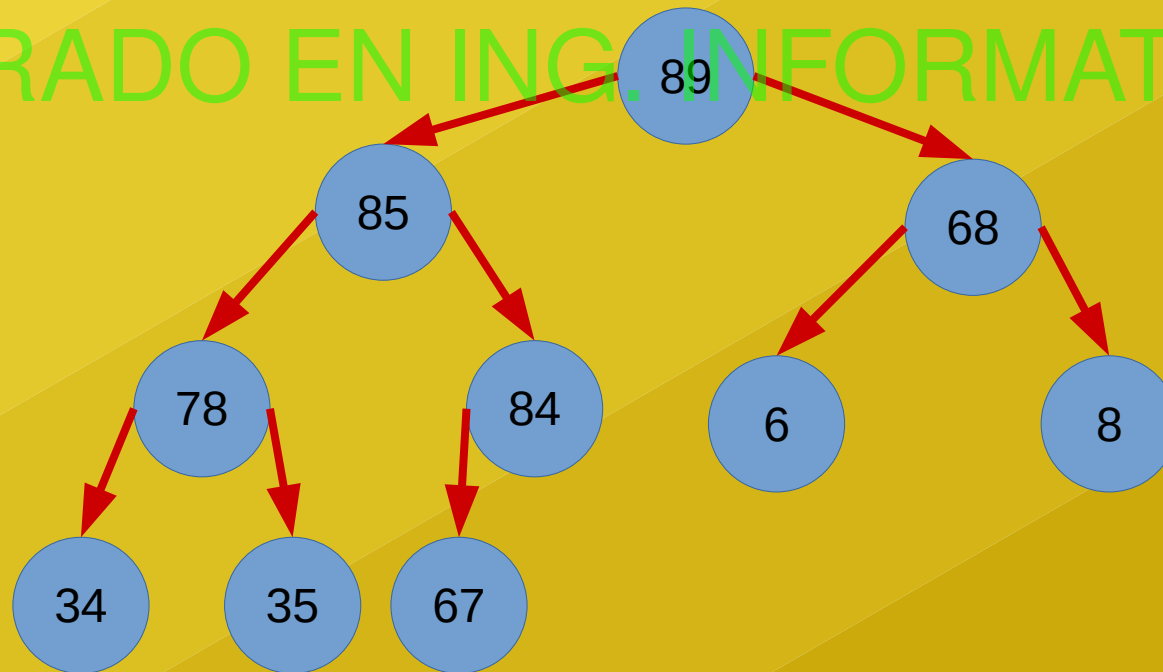
- Crear el montículo máximo para la secuencia:
 - {68,34,78,84,67,6,8,35,89,85}

EEDD - GRADO EN ING. INFORMÁTICA - UCO

¿?

Heap

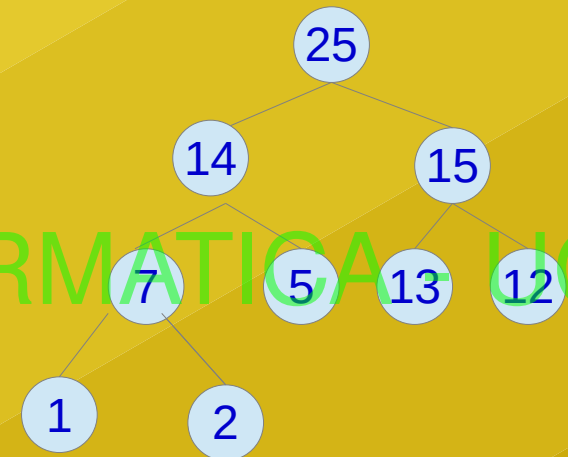
- Crear el montículo máximo para la secuencia:
 - {68,34,78,84,67,6,8,35,89,85}



Heap

- Operación de borrado:
 - Borrar siempre la raíz.
 - Intercambiar la raíz con el item de la última hoja, y borrar esa hoja.
 - Si no se cumple invariante intercambiar el nodo padre con hijo mayor/menor (“Hundir”) y repetir comprobación en nivel inferior.

remove()



¿?

Heap

- Operación de borrado:
 - Borrar siempre la raíz.
 - Intercambiar la raíz con el item de la última hoja, y borrar esa hoja.
 - Si no se cumple invariante “heap”, intercambiar el nodo padre con hijo mayor/menor (“Hundir”) y repetir comprobación en nivel inferior.

remove()

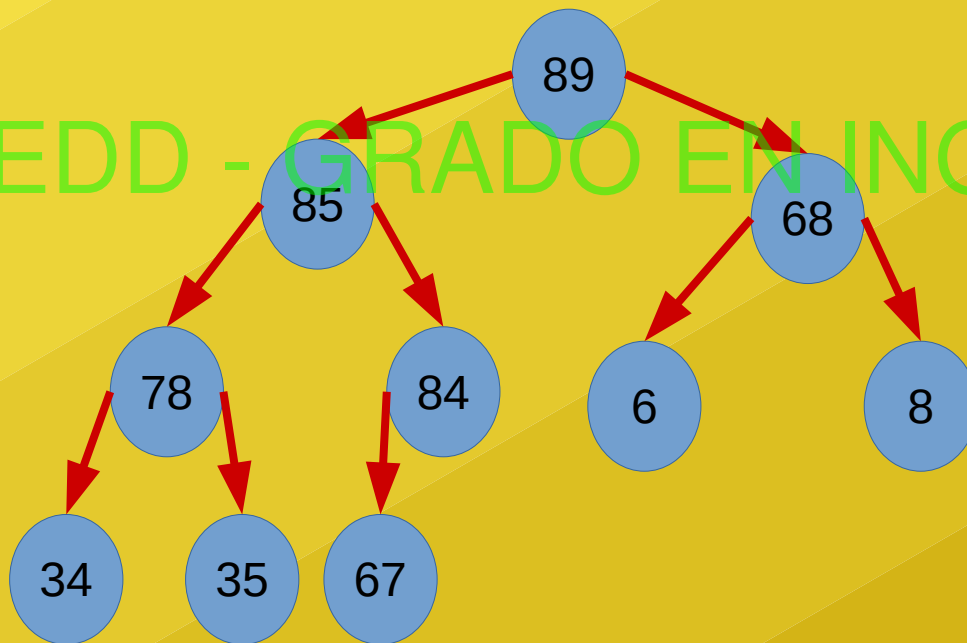
Hundir



Heap

- Dado el montículo siguiente, eliminar la cima.

remove()

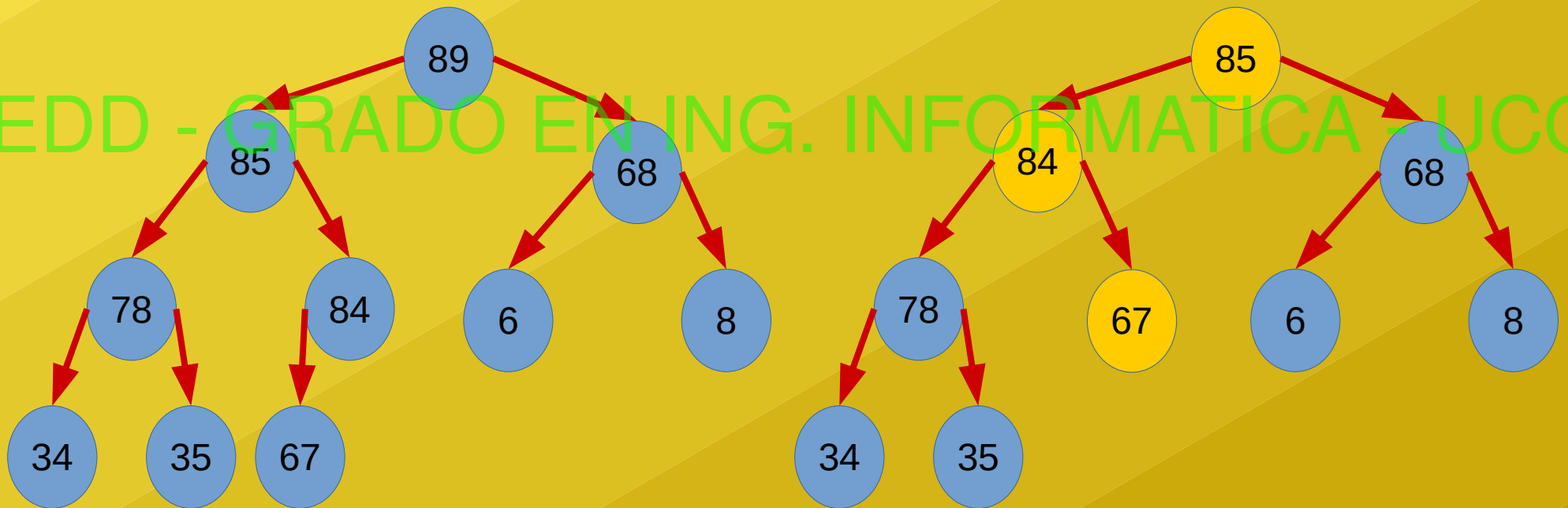


¿?

Heap

- Dado el montículo siguiente, eliminar la cima.

remove()



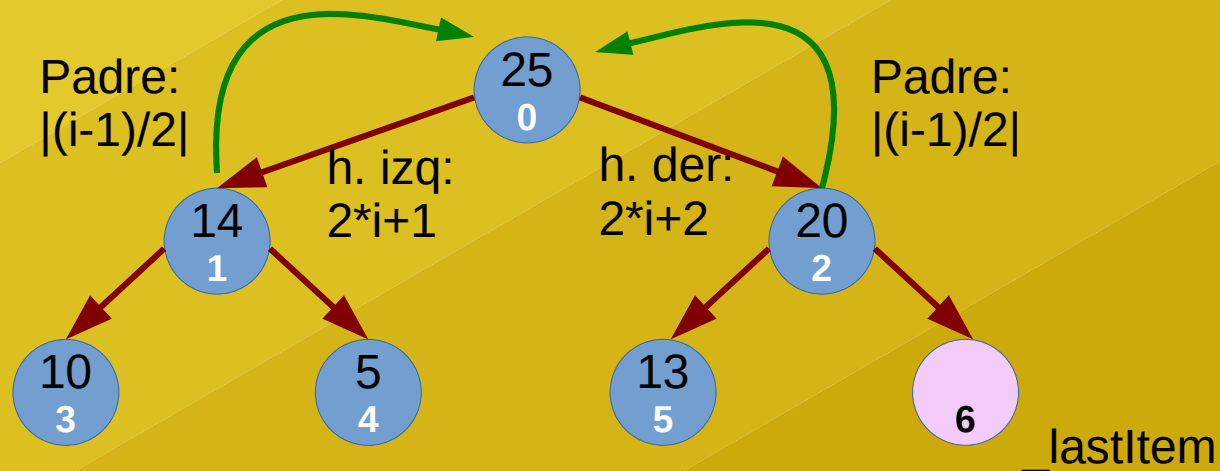
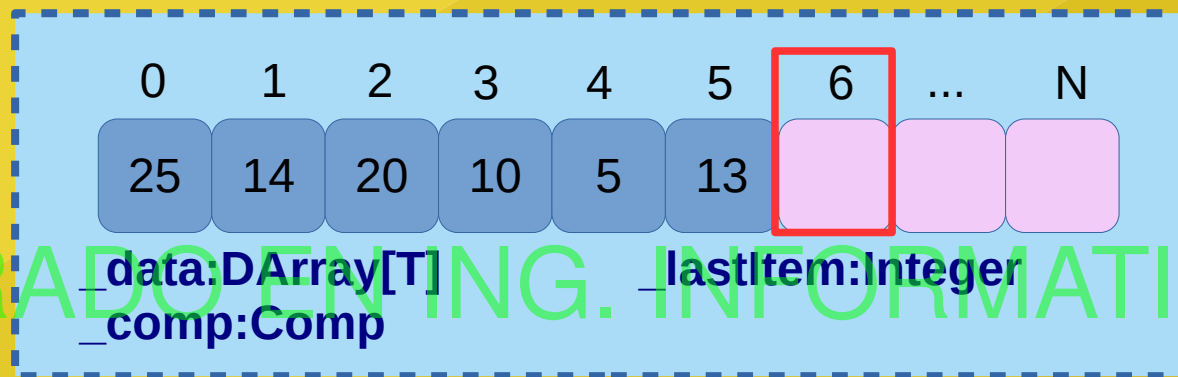
Heap

- Diseño.
 - ¿cómo flota una hoja?
 - ¿cómo localizo la hoja a crear/eliminar manteniendo la invariante árbol cuasi-completo?

EEDD - GRADO EN ING. INFORMÁTICA - UCO

Heap

- Diseño de un árbol binario usando un DArray.

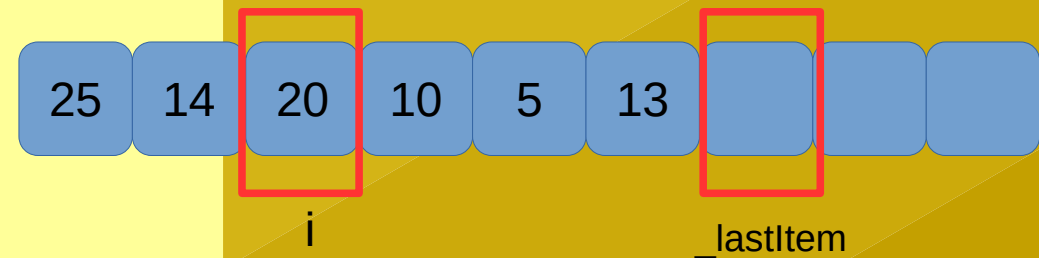
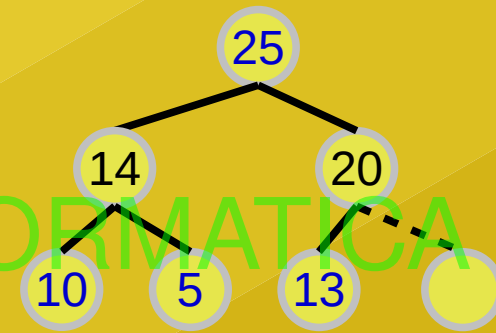


Heap

- Heap[T]: diseño.

```
Heap[T]::create(Var data:DArray[T], comp:Comp[T])
  _data <- data
  _lastItem <- data.size()
  _comp <- comp
  heapify()
Heap[T]::isEmpty:Bool
  return _lastItem == 0
Heap[T]::size():Integer
  return _lastItem
Heap[T]::item():T //0( )
  pre-c: not isEmpty()
  return _data[0]
```

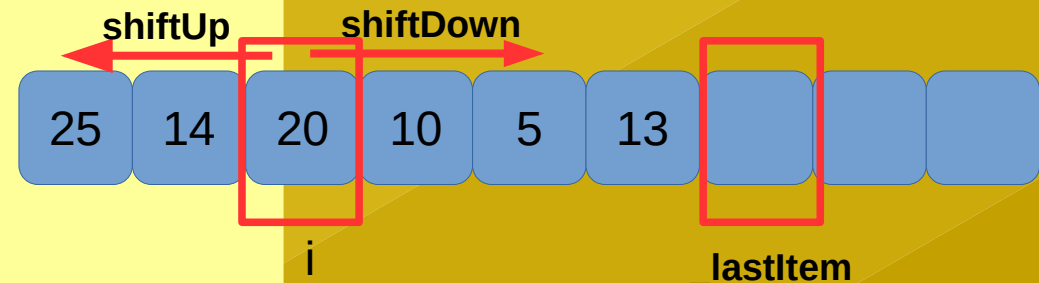
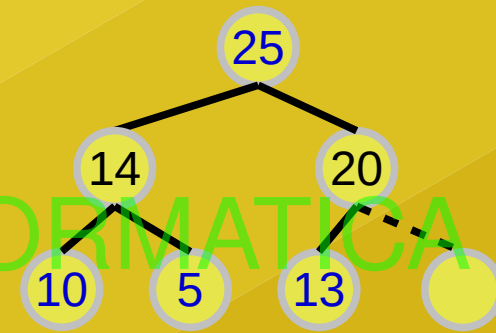
```
leftChild(i:Integer):Integer
  return i*2+1
rightChild(i:Integer):Integer
  return i*2+2
parent(i:Integer):Integer
  return (i-1) div 2
```



Heap

- Heap[T]: diseño.

```
Heap[T]::shiftUp(i:Integer): //0( )  
  If i>0 and comp(_data[i], _data[parent(i)]) Then  
    swap(_data[i], _data[parent(i)])  
    shiftUp(parent(i))  
  EndIf  
Heap[T]::shiftDown(i:Integer): //0( )  
  Var: n, lc, rc:Integer  
  n ← i  
  lc ← leftChild(i)  
  rc ← rightChild(i)  
  If lc<_lastItem And comp(_data[lc],_data[n]) Then  
    n ← lc  
  EndIf  
  If rc<_lastItem And comp(_data[rc],_data[n]) Then  
    n ← rc  
  EndIf  
  If i <> n Then  
    swap(_data[i], _data[n])  
    shiftDown(n)  
  EndIf
```



Heap

- Heap[T]: diseño.

```
Head[T]::insert(T item) //O( )
  If _lastItem = _data.size() Then
    _data.pushBack(item)
  Else
    _data[_lastItem] ← item
    shiftUp(_lastItem)
    _lastItem ← _lastItem + 1

Head[T]::remove() // O( )
  Pre-c: not isEmpty()
  _lastItem ← _lastItem - 1
  If _lastItem > 0 Then
    swap(_data[0], _data[_lastItem])
    ShiftDown(0)
  End-If
```

Heap

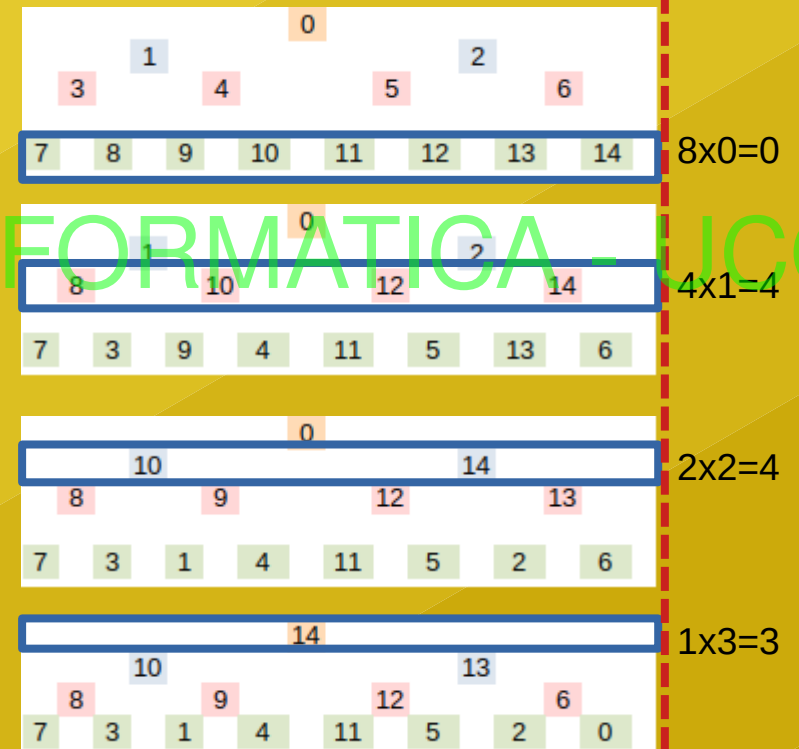
- Algoritmo HeapSort: Paso 1 (heapify).

//Make the array to be a heap.

Head[T]::heapify() //O()*

If size()>0 Then

For i ← (size() div 2)-1 To 0 Inc -1 do
shiftDown(i)



$$(0 * n/2) + (1 * n/4) + (2 * n/8) + \dots + (h * 1) < N \rightarrow O(N)$$

Total=11 < 15

*Ver: stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity

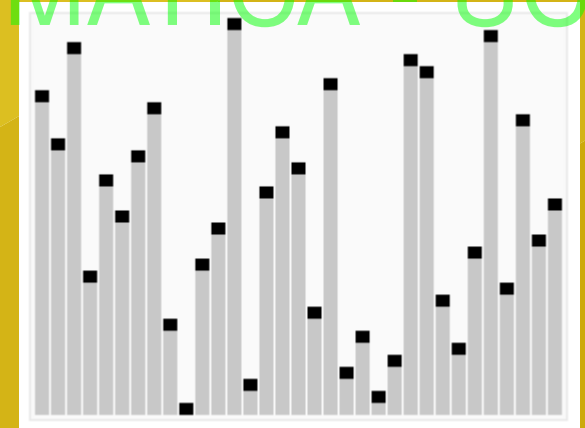
fjmadrid@uco.es

Heap

- Algoritmo HeapSort: Paso 2.
 - Los montículos fueron inventados para ordenar vectores.

```
Algorithm Heapsort(Var data:Array[T],  
                    comp:Comp[T])  
  
Var  
    heap:Heap[T]  
Begin  
    heap := Head[T]::create(data,comp) //heapify  
    While Not heap.isEmpty() Do  
        heap.remove()  
End.
```

¿O ()?



fuentes: Wikipedia

Heap

- Resumiendo.
 - Un Heap es una ED usada para tener disponible con $O(1)$ el máximo/mínimo elemento de un conjunto y mejor desempeño en inserción que una lista ordenada.
 - Es un tipo de árbol binario que mantiene dos invariantes:
 - Es un árbol binario completo.
 - En todos los sub árboles, la raíz es mayor/menor que todos sus descendientes.
 - Se inventaron para ordenar vectores: algoritmo Heapsort.
 - También se usan para implementar una cola de prioridad.
 - También nos permite seleccionar los K mayores/menores elementos de un conjunto, por ejemplo para buscar los K vecinos más cercanos.

Referencias

- Lecturas recomendadas:
 - Caps. 10, 11 y 12 de “Estructuras de Datos”, A. Carmona y otros. U. de Córdoba. 1999.
 - Caps 9 y 13.5 de “*Data structures and software development in an object oriented domain*”, Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
 - Wikipedia:
 - [en.wikipedia.org/wiki/Heap_\(data_structure\)](http://en.wikipedia.org/wiki/Heap_(data_structure))