Estructuras de datos y Algoritmos II

UNIDAD 1: Algoritmos de ordenamiento

Profesor: Ing. Jonathan Roberto Torres Castillo

Ingenieria en Computación

Universidad Nacional Autónoma de México

Cuarta sesión

1. Algunas definiciones
2. Algoritmo HeapSort.
3. Descripción del Algoritmo HeapSort.
4. Ejemplos y aplicaciones.

Definiciones

Heap: Montículo

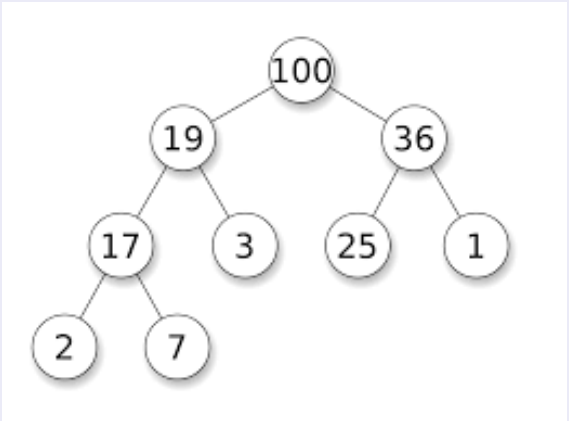
Árbol binario donde **todos los padres son mayores que los hijos** (nodos superiores mayores que los nodos inferiores).

El montículo se construye teniendo en cuenta que al finalizar la organización del arreglo en el árbol, deben quedar ubicados a un solo lado los hijos, esto es al lado izquierdo .

Algoritmos de ordenamiento: HeapSort

Algoritmo de ordenación 'HeapSort': Descripción y características

Fue publicado originalmente por **J.W.J. Williams** llamándolo "Algorithm 232" en la revista "Communications of the ACM" en **1964**. Este algoritmo consiste en ordenar en un árbol y luego extraer del nodo que queda como raíz en sucesivas iteraciones obteniendo el conjunto ordenado.



HeapSort: Características

Este algoritmo consiste en almacenar todos los elementos en un montículo y luego extraer el nodo que queda como raíz en iteraciones sucesivas obteniendo el conjunto ordenado. Para esto el método realiza los siguientes pasos:

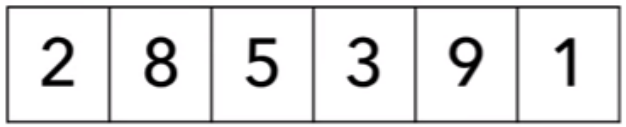
HeapSort: Características

- Construir un montículo inicial con todos los elementos del vector $A[1], A[2], \ldots., A[n]$.
  - Se llenan de izquierda a derecha los espacios.
  - Comparación: el/los hijos (espacio inferior) se comparan con el padre (nodo superior) y se debe cumplir la relación de que *padre* $>$ *hijo*.
  - Si la condición *padre* $>$ *hijo* es falsa, se inserta el nodo inferior en el espacio del nodo superior. se continua con el procedimiento hasta llegar a la cabeza del árbol, donde este el último elemento va a ser el valor mayor del montículo.
- Intercambiar los valores de $A[1]$ y $A[n]$ (siempre se queda el máximo en el extremo).
- Reconstruir el montículo con los elementos $A[1], A[2], \ldots\ldots, A[n-1]$
- Repetir desde el paso 2 hasta que el montículo quede con 1 solo elemento.

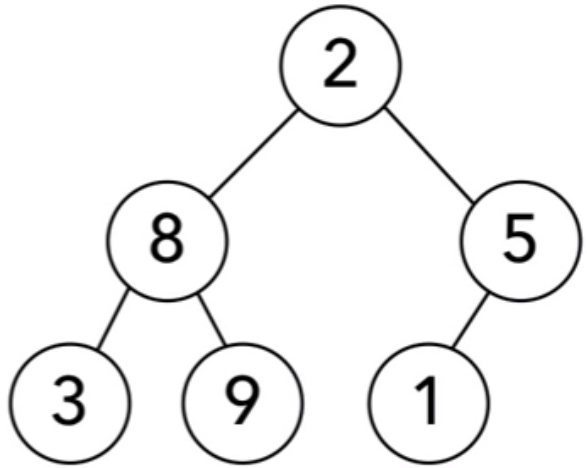
HeapSort: Características

Este es un proceso iterativo que partiendo de un montículo inicial, repite intercambiar los extremos, decrementar en 1 la posición del extremo superior y reconstruir el montículo del nuevo vector.

HeapSort: Construcción del Árbol

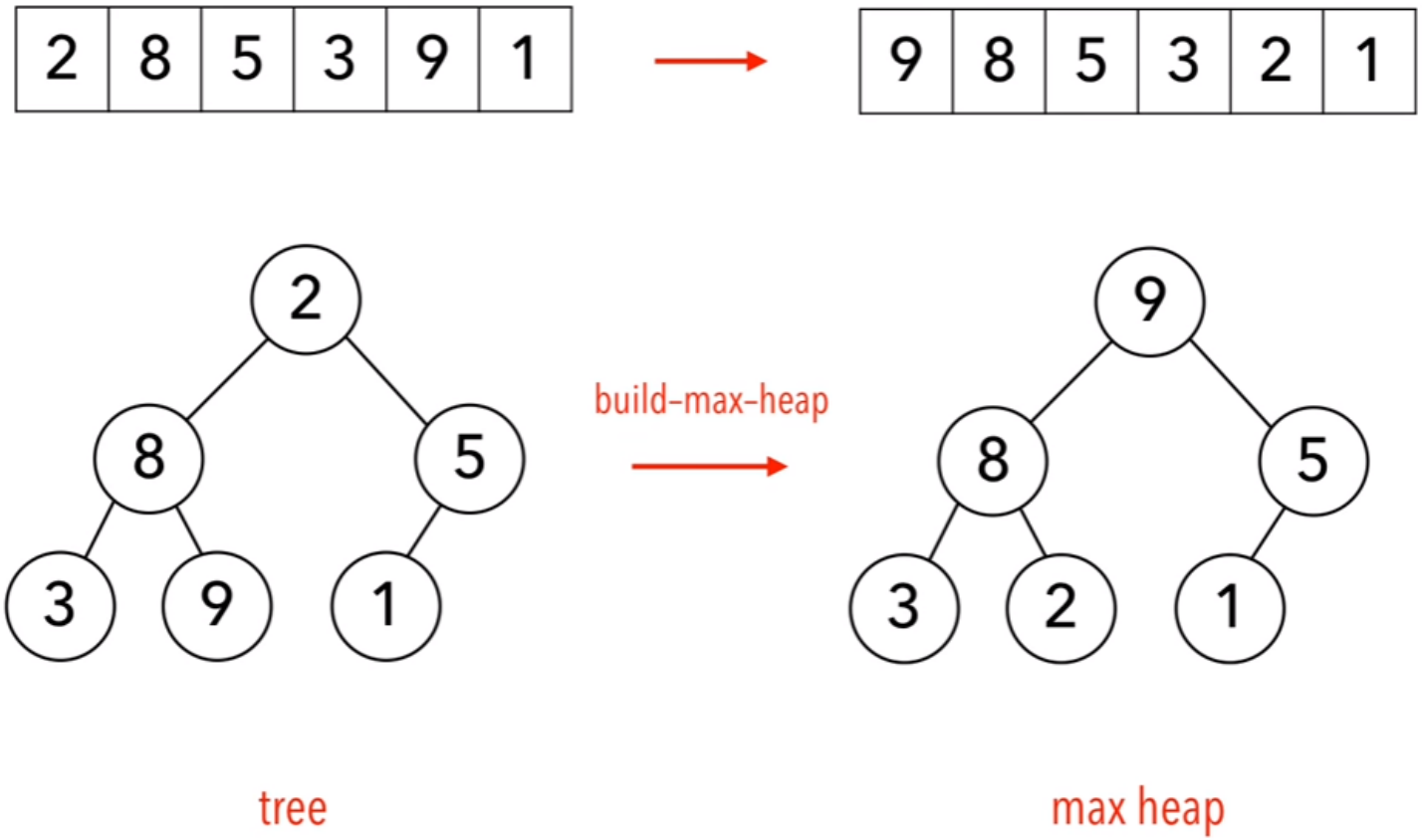


array

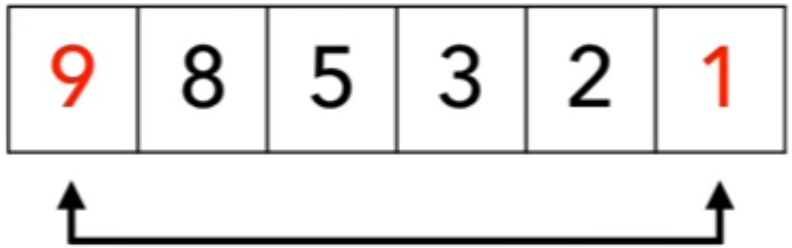


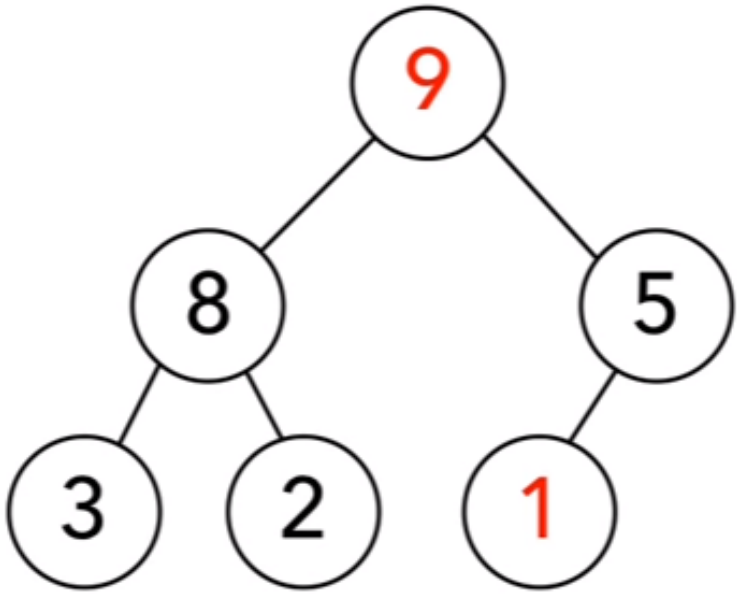
tree

HeapSort:MaxHeapIfy

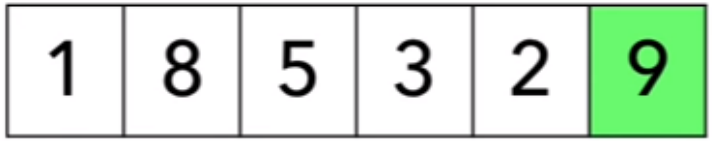


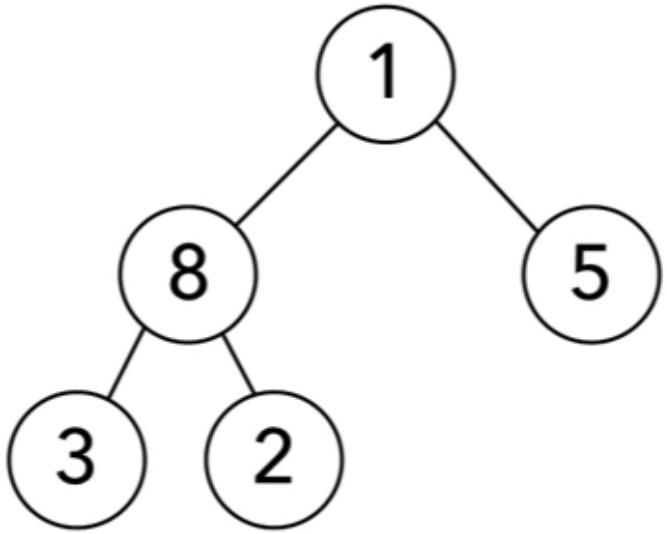
HeapSort: Intercambio de Raíz





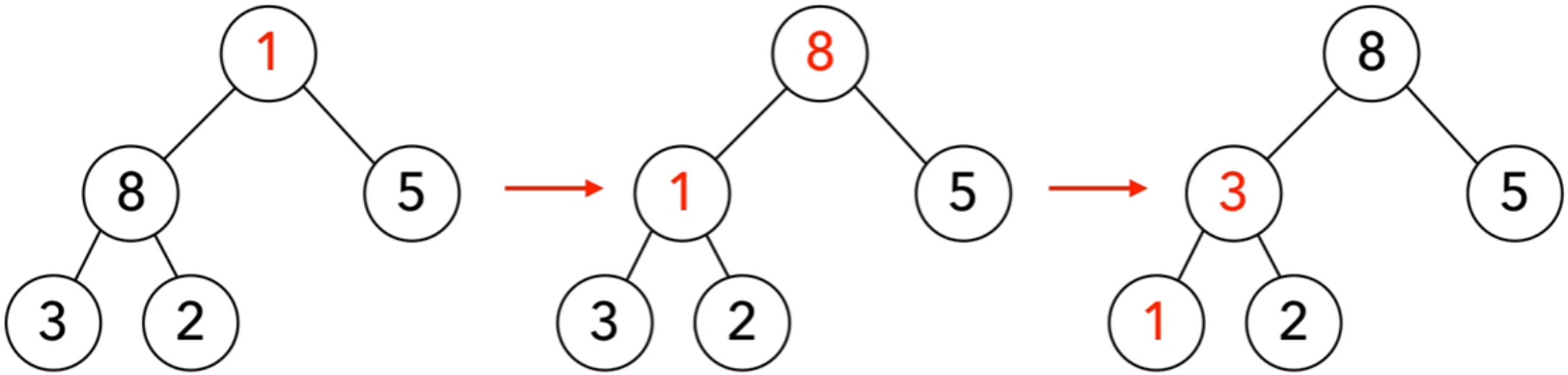
HeapSort: Eliminación del ultimo elemento



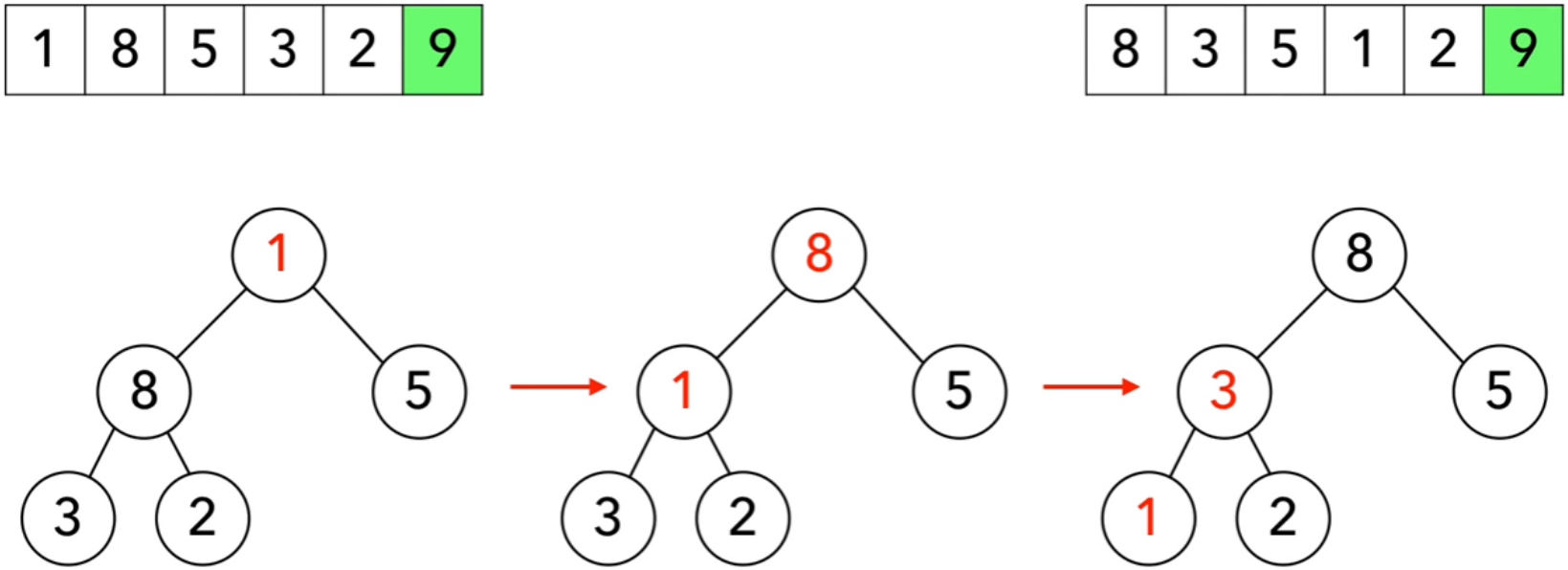


HeapSort:MaxHeapIfy

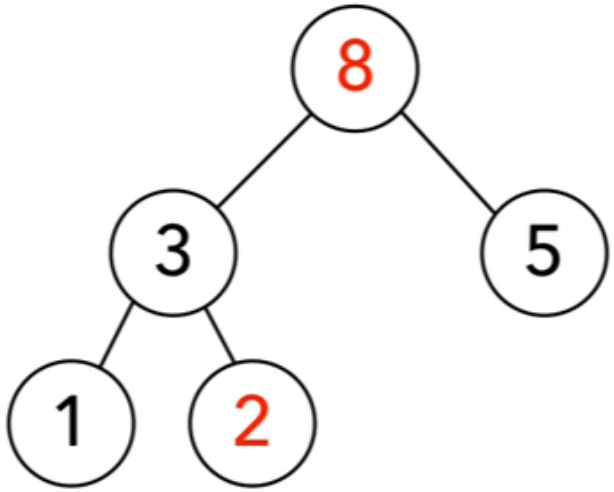
1	8	5	3	2	9



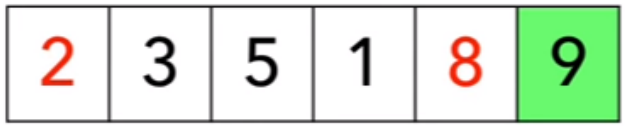
HeapSort: MaxHeapIfy

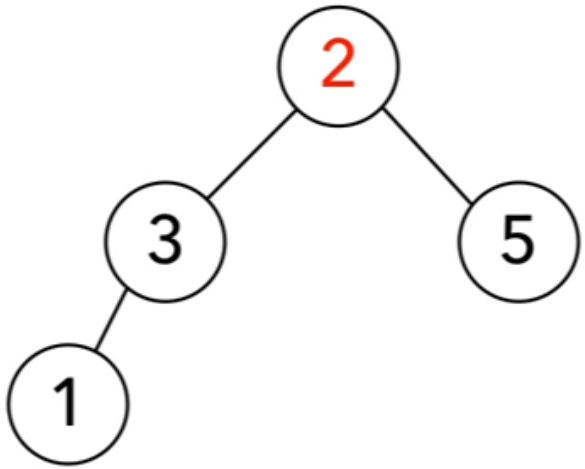


HeapSort: Intercambio de Raíz

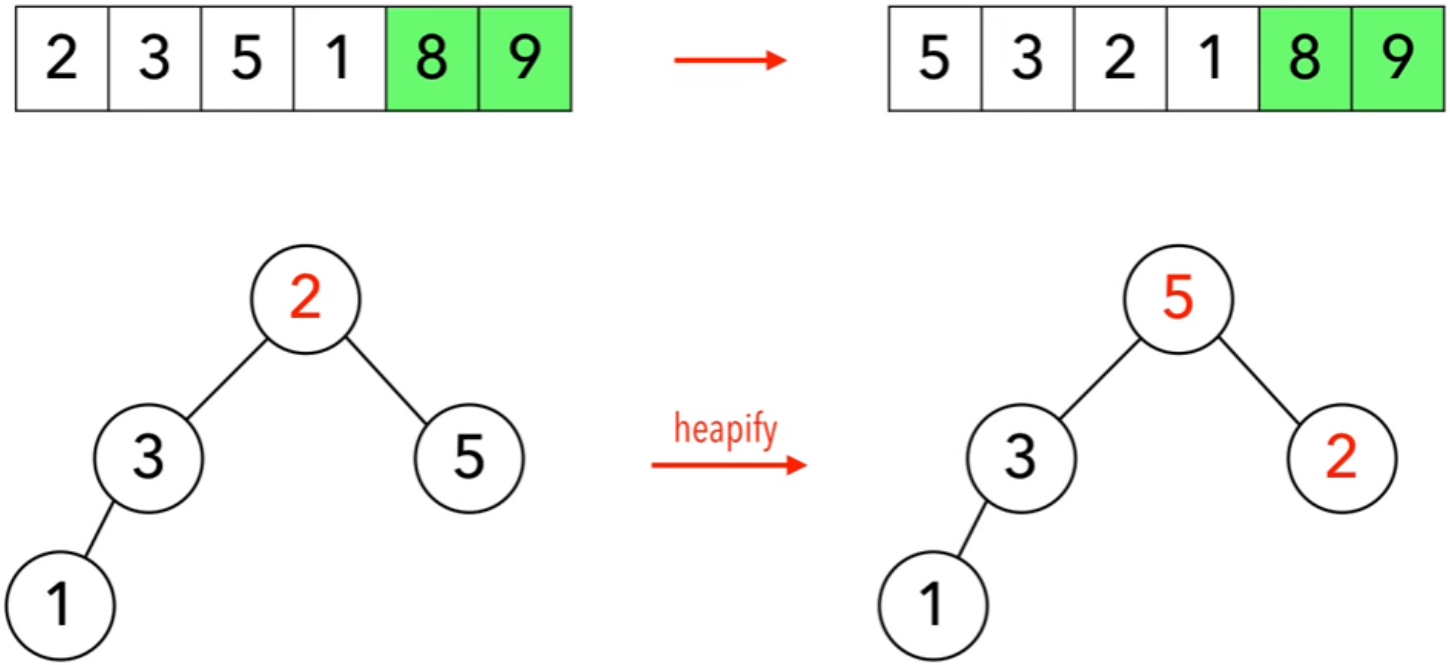


HeapSort: Nuevo Heap (Eliminación del ultimo elemento)

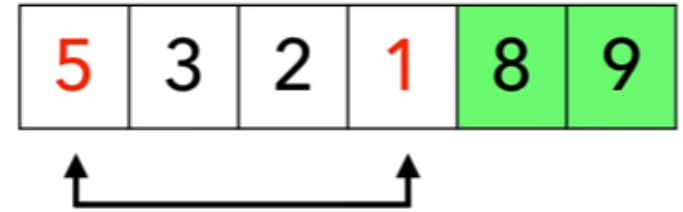


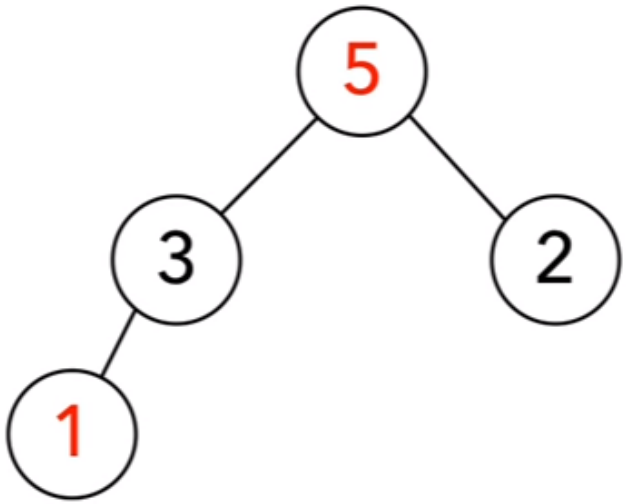


HeapSort: MaxHeapIfy

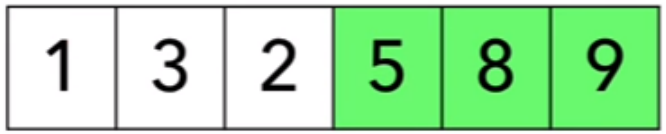


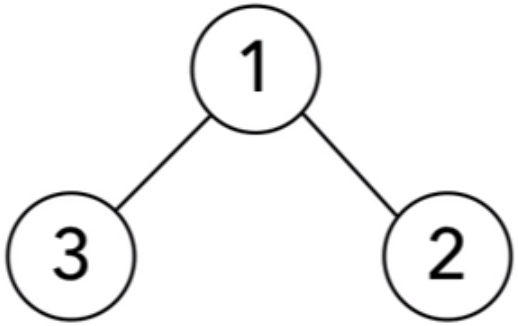
HeapSort: Intercambio de Raíz





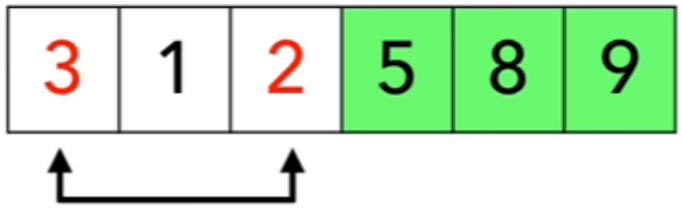
HeapSort: Nuevo Heap (Eliminación del ultimo elemento)

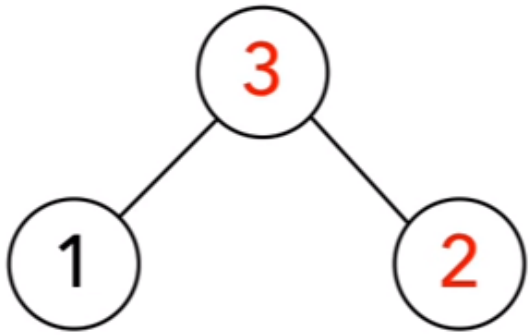




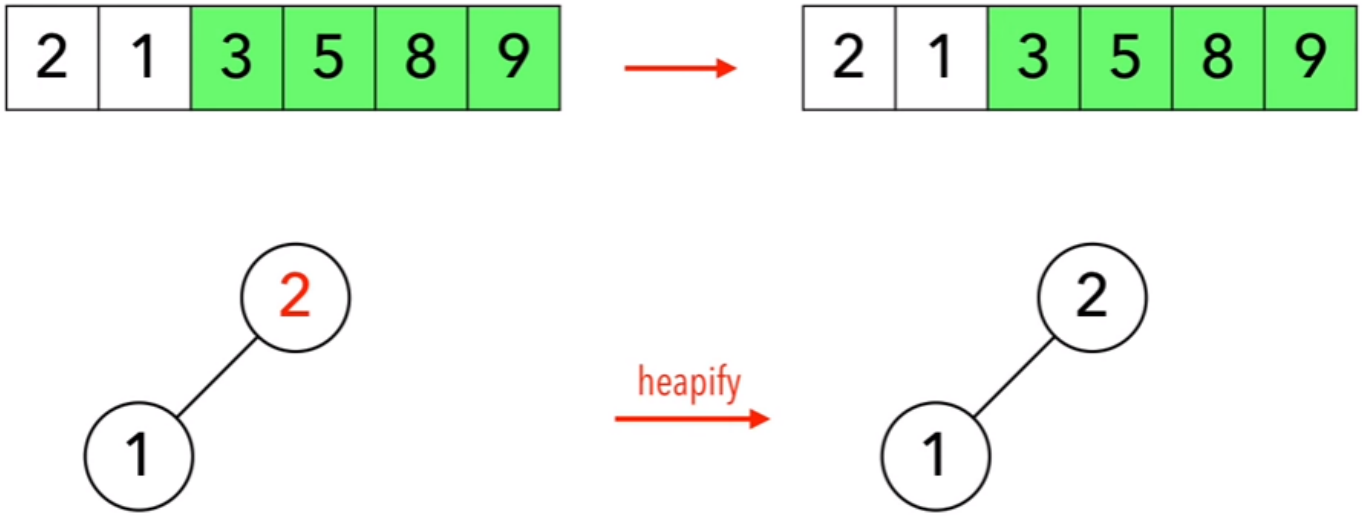
HeapSort: MaxHeapIfy

HeapSort: Intercambio de Raíz

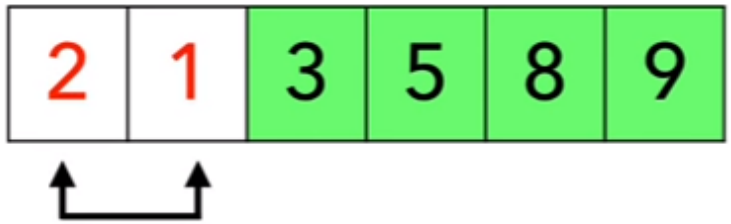


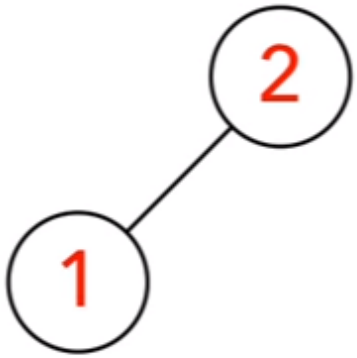


HeapSort: MaxHeapIfy

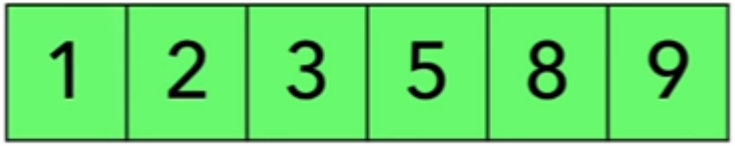


HeapSort: Intercambio de Raíz





HeapSort: Resultado



HeapSort

Un algoritmo general se puede representar como:

```
OrdenacionHeapSort(A)
Inicio
construirHeapMaxIni(A)
Para i=longiudDeA hasta 2 hacer
    Intercambia(A[1], A[i])
    TamanoHeapA=TamanoHeapA-1;
    MaxHeapify (A,1,TamanoHeap)
Fin
```

La función `construirHeapMaxIni( )` construye el heap inicial tal que sea un HeapMaximo, `Intercambia()` función que intercambia de lugar los elementos $A[1]$ y $A[i]$; y `MaxHeapify ()` permite que el heap modificado mantenga la propiedad de orden de un HeapMaximo [1], [2], [3].

HeapSort:MaxHeapify

```
MaxHeapify (A,i)
Inicio
L= hIzq( i)
R=hDer(i)
Si L <TamanoHeapA y A[L]>A[i]
    posMax=L
En otro caso
    posMax = i
Fin Si
Si R<TamaoHeapA y A[R]> A[posMax] entonces
    posMax =R
Fin Si
Si posMax =~ i entonces
    Intercambia(A[i], A[posMax])
    MaxHeapify(A,posMax)
Fin Si
```

HeapSort: construirHeapMaxIni

```
construirHeapMaxIni( A )
Inicio
TamanoHeapA=longiudDeA
Para  i=[ longiudDeA/2], hasta 1
    MaxHeapify(A,i)
Fin Para
Fin
```

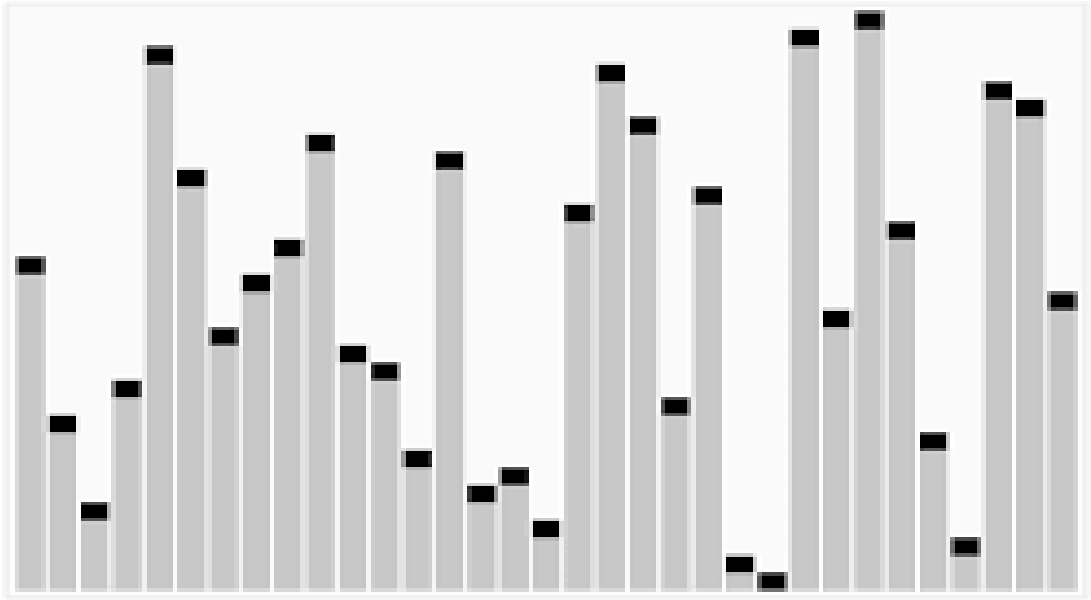
HeapSort: OrdenacionHeapSort

```
OrdenacionHeapSort( A)
Inicio
construirHeapMaxIni( A)
Para i=longiudDeA hasta 2 hacer
    Intercambia(A[1], A[i])
    TamanoHeapA= TamanoHeapA-1;
    MaxHeapify (A,1,TamanoHeap)
Fin
```

Complejidad del Algoritmo HeapSort

Eficiencia

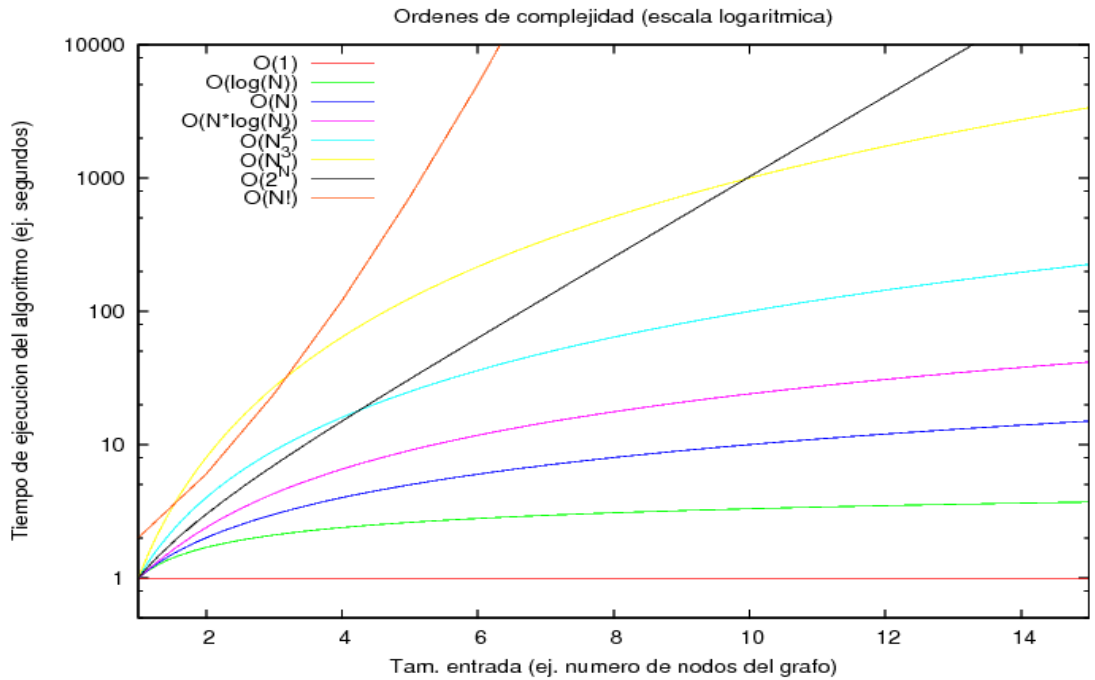
El algoritmo de ordenación por montículos o Heap Sort recorre el conjunto de elementos desde la posición de la mitad hasta la primera organizando el montículo correspondiente a dicho elemento. Una vez terminado este proceso, se inicia el proceso de ordenación intercambiando el primer elemento por el último del arreglo y reorganizando el montículo a partir de la primera posición.



Complejidad del Algoritmo HeapSort

Eficiencia

La complejidad del algoritmo de ordenación por montículos es $O(nlogn)$ teniendo en cuenta que el proceso de organizar el montículo en el peor caso solamente tiene que hacer intercambios sobre una sola línea de elementos desde la raíz del árbol hasta alguna de las hojas para un máximo de $nlog(n)$ intercambios.



Referencias I

Thomas H Cormen.

Introduction to algorithms.

MIT press, 2009.

Donald Knuth.

The art of programming.

ITNow, 53(4), 2011.

Elba Karen Sáenz García.

Guía práctica de estudio 1, algoritmos de ordenamiento parte 1.

Facultad de ingeniería, UNAM.