

Algoritmos de ordenación

¿Por qué es importante?

- . Es la base sobre la que se construyen muchos otros algoritmos.
 - . Históricamente, se ha utilizado más tiempo de computación en ordenación que en cualquier otra cosa.
 - . Es el problema más profundamente estudiado en ciencias de la computación.
-

Algoritmos de inserción

- . Agregan los elementos uno a uno en su posición correcta dentro de una parte ya ordenada del arreglo.

Insertion Sort (inserción directa)

- . Es un algoritmo corto.
- . Es una buena solución si son pocos los elementos que necesitan ser ordenados.
- . Si son muchos los elementos a ordenar, no es una buena opción, pues para ordenarlos requerimos mucho tiempo.
- . **Big O (n)**, en el mejor caso, qué ocurre **si el arreglo ya está ordenado**.
- . Big O (n^2), en el peor caso, si el arreglo está en orden inverso.
- . En el caso promedio, Big O es cuadrático también.

Funciona de la siguiente manera:

- . Tomamos aisladamente al primer elemento del array, y consideramos que está ordenado.
- . Entonces pasamos al segundo elemento, es decir `Array[1]`.
- . Lo almacenamos temporalmente en una variable “temp”.
- . Lo comparamos con `Array[0]`; si es menor (lo que quiere decir que antes hay un elemento que es mayor), entonces los intercambiamos.

. Esta comparación se hace con todos los elementos a la izquierda de nuestro elemento inicial.

Comienzo

```
(1) Desde  $i = 2$  hasta  $N$  hacer
(2)    $Aux \leftarrow V[i]$ 
(3)    $j = i - 1$ 
(4)   mientras  $j > 0$  y  $Aux.clave < V[j].clave$  hacer
(5)      $V[j+1] \leftarrow V[j]$ 
(6)      $j \leftarrow j - 1$ 
(7)   fin mientras
(8)    $V[j+1] \leftarrow Aux$ 
(9) fin desde
Fin
```

Algoritmo de ordenación Shell

. Surge en 1959 por Donald Shell.

. Algoritmo sub cuadrático, ligeramente más largo que el de inserción, lo que lo convierte en el más simple de los algoritmos rápidos.

. Se le puede llamar “clasificación por disminución de incrementos”.

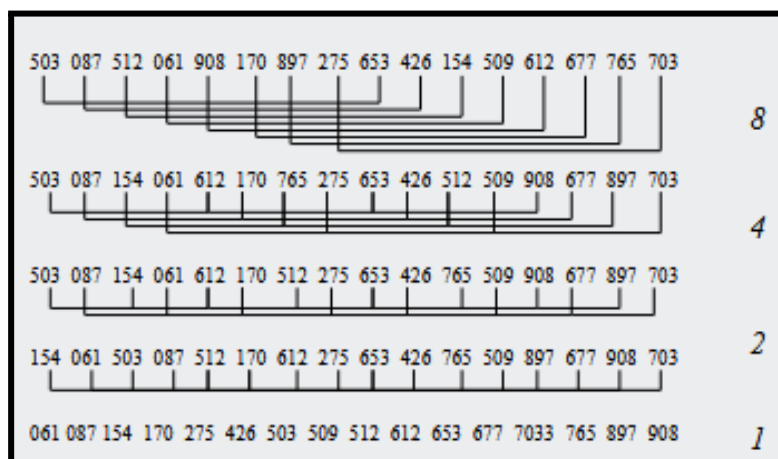
. Busca mejorar la lentitud del Insertion Sort cuando los elementos están muy desordenados.

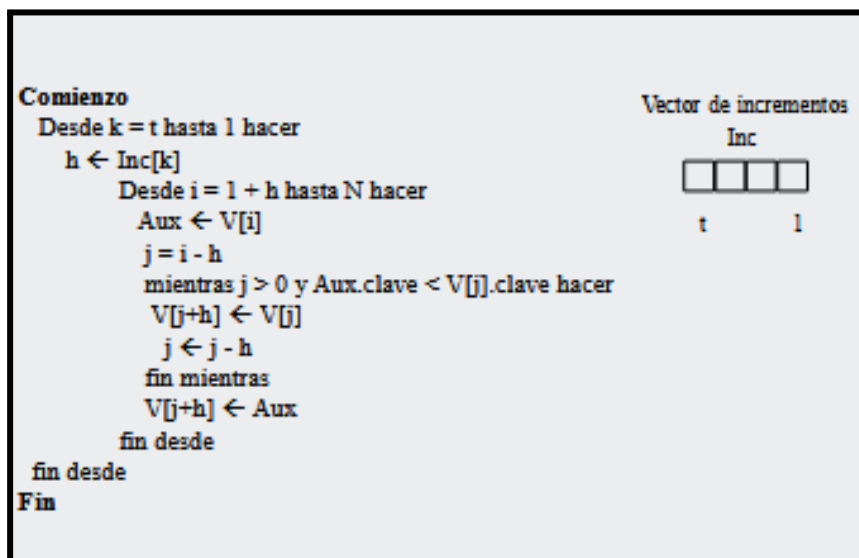
Consiste en dar grandes saltos en vez de dar pasos pequeños.

Ejemplo: Supongamos que tenemos 16 registros. Primero los dividimos en dos grupos, agrupados de la siguiente manera: (R1, R9), (R2, R10), ..., (R8, R16). y clasificamos cada grupo por separado.

A posteriori, dividimos los elementos en 4 grupos de cuatro, y clasificamos cada grupo por separado.

Continuamos así hasta tener un solo grupo con los 16 elementos.





Incrementos o “gaps”

- . El tema de los incrementos es ciertamente complicado, pues no se conoce la mejor secuencia para grandes valores de N .
- . Lo que sí se puede decir, es que, si es posible, los incrementos no deben ser múltiplos de sí mismos, de forma que las cadenas se mezclen entre sí lo más a menudo posible.
- . ¿Cuáles son buenas elecciones de incrementos?:
 - Sedgwick: 1, 5, 19, 41, 109, ... (presentados en orden inverso).
 - Hibbard: 1, 3, 7, 15, ...
 - Shell original: $n/2$, $n/4$, ..., 1 (el prestado como ejemplo antes), aunque no es la mejor decisión.
- . Cualquiera sea la decisión de secuencia de incremento, todos ellos terminan con un último incremento de 1, en el que simplemente se aplica un Insertion sort, pero como ahora está ordenado, es mucho más rápido a que si no lo estuviera.

Complejidad temporal

- . Si hablamos de complejidad, no es fácil de calcular con precisión, pero en el peor caso estaríamos entre $O(n^2)$ y $O(n \log^2 n)$.
- . Con buenas secuencias, como la de Sedgwick, lo bajamos a $O(n^{1.26})$ en práctica.
- . El mejor caso es cerca de $O(n \log n)$.
- . Sin dudas si aplicamos la secuencia original, el peor caso es cuadrático.

. Si utilizamos alguna mejora, Allan Weiss menciona que el tiempo puede bajar a $O(n^{3/2})$, en el peor caso.

. En el caso de Sedwick, como mencionamos, en un caso promedio puede llegar a ser de $O(n^{1.26})$, y en el peor de los casos, $O(n^{4/3})$.

En resumen, el orden de tiempo de ejecución del Shell Insertion depende plenamente del tipo de secuencia de incremento que se utilice, y su función es esencialmente resolver los problemas de lentitud del algoritmo de inserción estándar, dividiendo los elementos en grandes intervalos, los cuales se van a acortando hasta que llegamos a un conjunto, lo suficientemente ordenado para aplicar un insertion Sort que nos lleve Big $O(n)$, y no cuadrático, como lo sería naturalmente.

Ejemplo con incremento {5, 3 y 1}

1	2	3	4	5	6	7	8	9	10	11	12	13
81	94	11	96	12	35	17	95	28	58	41	75	15

Inicio, este es nuestro array inicial.

1	2	3	4	5	6	7	8	9	10	11	12	13
81	94	11	96	12	35	17	95	28	58	41	75	15

Incremento de 5

1	2	3	4	5	6	7	8	9	10	11	12	13
81	94	11	96	12	35	17	95	28	58	41	75	15
35	17	11	28	12	41	75	15	96	58	81	94	95

Una vez definido el gap, intercambiamos si el elemento de la derecha es más grande que el de la izquierda.

1	2	3	4	5	6	7	8	9	10	11	12	13
35	17	11	28	12	41	75	15	96	58	81	94	95

Incremento de 3

1	2	3	4	5	6	7	8	9	10	11	12	13
35	17	11	28	12	41	75	15	96	58	81	94	95
28	12	11	35	15	41	58	17	94	75	81	96	95

Idem que con la imagen 3.

1	2	3	4	5	6	7	8	9	10	11	12	13
28	12	11	35	15	41	58	17	94	75	81	96	95

Llegamos al final, ahora podemos aplicar un insertion sort. Se dice que el incremento o el gap es de 1.

1	2	3	4	5	6	7	8	9	10	11	12	13
28	12	11	35	15	41	58	17	94	75	81	96	95
11	12	15	17	28	35	41	58	75	81	94	95	96

Nos queda todo ordenado.

Algoritmos de Intercambio

. Reordenan elementos intercambiándolos hasta que estén en la posición correcta.

QuickSort

- . Es del tipo “divide y vencerás” o “diviser pour mieux régner” o "Teile und herrsche" o "Dividi et impera" o "Divide and conquer" o “分割して征服する” o "Divideix i venceràs".
- . Tiene un tiempo promedio de Big O($n \log n$), siendo el peor caso, cuadrático. De todos modos, Weiss cita que con un poco de esfuerzo el peor caso se hace muy improbable.
- . Es recursivo.

Funciona de la siguiente manera:

- . La idea es clasificar un conjunto de elementos $V[1] \dots V[N]$. Para esto vamos a tomar a un elemento del conjunto el cual llamaremos “pivot”, este elemento será $V[P]$.
- . Es recomendable que este elemento sea la mediana del conjunto, de forma que esté precedido y sucedido por más o menos la mitad de los elementos del conjunto.

Elección del pivot

- . No existe solo una opción, y de hecho, la elección del pivot quedará a nuestra discreción, aunque es cierto que existen mejores opciones que otras.
- . Según Weiss, elegir el primer elemento como el pivot es la elección más popular, pero a su vez, la más ingenua. Es aceptable si la entrada de datos es aleatoria, pero si la entrada ya ha sido previamente ordenada o está en orden inverso, entonces el pivot proporciona una partición poco adecuada, pues se trata de uno de los elementos extremos.
- . **NO es recomendable utilizar el primer elemento como pivot.**
- . **Y, NO es recomendable elegir el último elemento como pivot.**

. En ambos casos podríamos terminar en el peor caso Big $O(n^2)$.

. Una **elección segura**, es elegir el **elemento del medio**, el que está ubicado en la celda $((low + high) / 2)$. Siendo high el último elemento y low el primero.

. Si es esta nuestra elección, está prácticamente garantizado que nunca tendremos el peor caso cuadrático.

Método de la mediana de tres

. Vamos a elegir la mediana entre estos 3 elementos:

- $V[low]$ - el primer elemento.
- $V[high]$ - el último elemento.
- $V[(low + high) / 2]$ - el elemento del medio.

. La mediana, significa el valor intermedio entre esos 3.

Paso a paso:

Calculamos el índice del medio:

- $int\ mid = (low + high) / 2;$
- $if\ (V[low] > V[mid])\ swap(V, low, mid);$
- $if\ (V[low] > V[high])\ swap(V, low, high);$
- $if\ (V[mid] > V[high])\ swap(V, mid, high);$

. Luego de realizar esas 3 comparaciones:

- $V[low]$ tiene el menor de los 3.
- $V[high]$ tiene el mayor.
- $V[mid]$ tiene la mediana - **es el que usamos como pivote.**

. Este método, si bien no garantiza que el pivote quede exactamente en el centro del conjunto, si es un hecho de que está lo más cerca del centro posible.

Código Mediana de 3 (Weiss)

```
int mid = (low + high) / 2;
```

```
// Ordenar los tres valores
```

```
if (array[low] > array[mid])
```

```
    swap(array, low, mid);
```

```
if (array[low] > array[high])
```

```
    swap(array, low, high);
```

```
if (array[mid] > array[high])
```

```
    swap(array, mid, high);
```

```
// Usar la mediana como pivote
```

```
swap(array, mid, high - 1);
```

```
int pivot = array[high - 1];
```

Partition

. Volvemos al tiro con lo que nos quedamos antes. Una vez elegimos nuestro pivote, vamos a recorrer el arreglo con dos índices:

- i : empieza en la izquierda, busca valores mayores o iguales al pivote.
- j : empieza en la derecha, busca valores menores al pivote.

. Cada vez que encontramos dos valores mal posicionados respecto al pivote, los intercambiamos.

. Cuando i y j se cruzan, colocamos el pivote en su posición final.

```
function particion( i, j: integer; pivote: TipoClave): integer;
{divide V[i], ..., V[j] para que las claves menores que pivote estén a la izquierda y las mayores
o iguales a la derecha. Devuelve el lugar donde se inicia el grupo de la derecha.}
var
  L,R : de tipo enteros; {cursores de acuerdo a la descripción anterior}
COM
  L := i;
  R := j;
  Repetir
    intercambia(V[L],V[R]);
    {ahora se inicia la fase de rastreo}
    mientras V[L].clave < pivote hacer L := L + 1; fin mientras
    mientras V[R].clave >= pivote hacer R := R - 1; fin mientras
  Hasta que L > R
  Devolver L;
end; {particion}
```

. Una vez aplicada la partición, nos queda un arreglo, de modo que, todos los elementos menores al pivote están a la izquierda de este, y todos los mayores o iguales a este, a la derecha.

Recursión

. En cada llamada:

- Se elige un nuevo pivote (mismo criterio).
- Se hace una nueva partición.
- Se divide en dos.
- Se llama recursivamente otra vez.

. Esto sigue hasta que los subarreglos tengan 0 o 1 elementos, lo que significa que ya están ordenados.

```
quicksort( i,j: tipo entero);  
//clasifica los elementos V[i],...,V[j] del arreglo externo V  
  
pivote : TipoClave; {el valor del pivote}  
IndicePivote : tipo entero; {el índice de un elemento de V donde clave es el pivote}  
k : tipo entero; {índice al inicio del grupo de elementos  $\geq$  pivote}  
COM  
IndicePivote  $\leftarrow$  EncuentraPivote(i,j);  
SI IndicePivote  $\diamond 0$  entonces {no hacer nada si todas las claves son iguales}  
    pivote  $\leftarrow$  V[IndicePivote].clave;  
    k  $\leftarrow$  particion(i,j,pivote);  
    quicksort(i,k-1);  
    quicksort(k,j);  
FIN SI;  
FIN; //quicksort
```

Complejidad espacial

. $O(\log(n))$.

Bubble Sort

- . Uno de los algoritmos más simples de ordenamiento.
- . Se caracteriza por hacer muchos intercambios entre elementos consecutivos.

Funcionamiento:

- . Es tremendamente bobo. Consiste en repetir múltiples pasadas por el arreglo, comparando pares de elementos adyacentes y haciendo intercambios si están en el orden incorrecto. Esto hace que los valores más grandes "suban" hacia el final del arreglo, como si "flotaran", de ahí su nombre.

Complejidad temporal

- . Siempre Big $O(n^2)$, pues siempre hace n^2 comparaciones, para n elementos. Internamente son dos bucles "for".

Complejidad espacial

- . $O(1)$.

```
(1)  Desde i = 1 hasta N-1 hacer
(2)    Desde j = N hasta i +1 hacer
(3)      Si V[j].clave < V[j-1].clave entonces
(4)        intercambia (V[j], V[j-1])
      Fin si
    Fin desde
  Fin desde
```

Algoritmos de selección

. Buscan el elemento más pequeño (o más grande) y lo colocan en su lugar final. Repetido hasta que todo esté ordenado.

Selection sort (selección directa)

- . Realiza exactamente $N-1$ intercambios.
- . Arroja un orden de tiempo de ejecución de $O(n^2)$ en todos los casos.


Funciona de la siguiente manera:

- . En cada iteración, se busca en el arreglo, el elemento más chico. Cuando se encuentra, se intercambia con el primer elemento del arreglo restante.
- . Lógicamente, primero se intercambia por arreglo[0], luego por arreglo[1],

Ejemplo: [40, 25, 89, 12, 77]

- . Busco el más chico: 12 => lo intercambio por 40 => [12, 25, 89, 40, 77].
- . Busco el más chico nuevamente: 25 => como ya está en el segundo lugar, no toco nada.
- . Busco el más chico nuevamente: 40 => lo cambio por 89 => [12, 25, 40, 89, 77].
- . Busco el más chico nuevamente: 77 => lo cambio por 89 => [12, 25, 40, 77, 89].

Selección directa: análisis del orden del tiempo de ejecución



```
(1) Desde i = 1 hasta N - 1 hacer
(2)   ÍndiceDelMenor ← i
(3)   ClaveMenor ← V[i].clave
(4)   Desde j = i + 1 hasta N hacer
(5)     Si V[j].clave < ClaveMenor entonces
(6)       ÍndiceDelMenor ← j
(7)       ClaveMenor ← V[j].clave
(8)   Fin si
(9)   Fin desde
(10)  intercambia (V[i], V[ÍndiceDelMenor])
      Fin desde
```

- Las sentencias de las líneas 2,3,5,6,7 y 10 son todas de $O(1)$.
- El bloque de sentencias que abarca la sentencia 4, se ejecuta exactamente $N-i$ veces.
- Ese bloque, más la sentencia de intercambio, se ejecutan exactamente $N-1$ veces.
- Por lo tanto, este método es $O(N^2)$, con la particularidad que los intercambios son siempre $N-1$.

Algoritmos y estructuras de Datos

14

Ventajas

- . Fácil y bobo de implementar.
- . Requiere pocos movimientos (a diferencia de bubble sort o insertion sort).

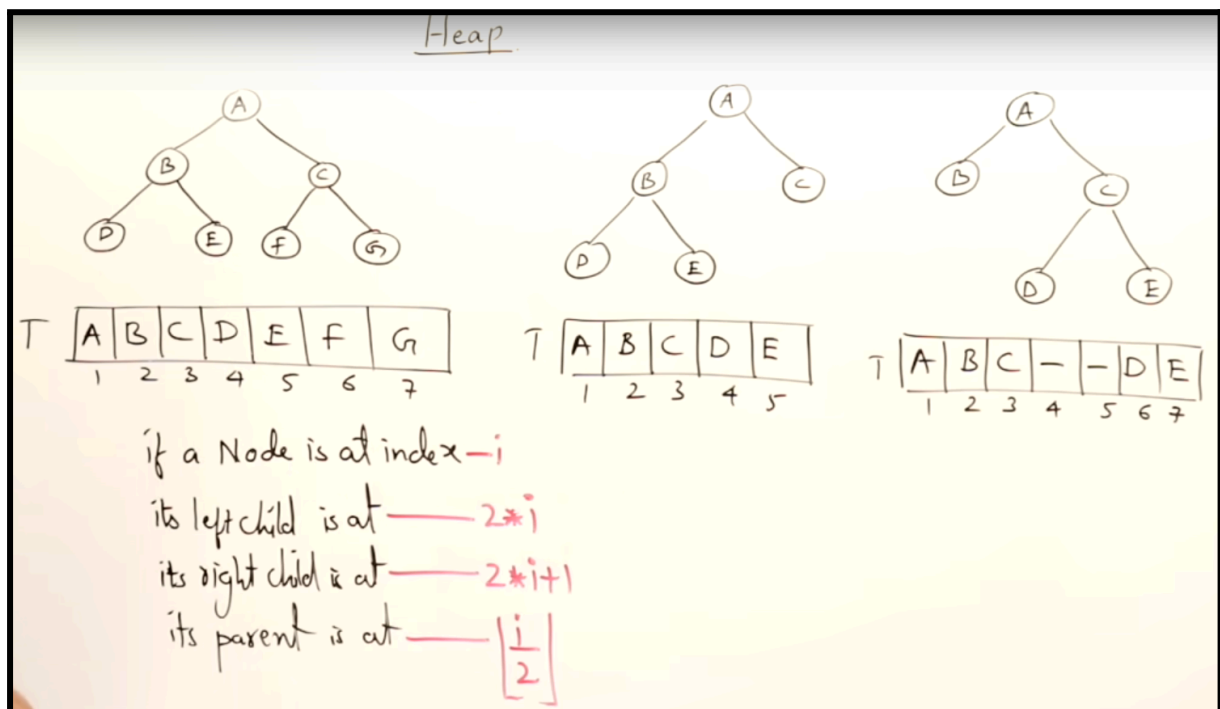
Desventajas

- . Poco eficiente en arreglos grandes.
 - . Tiempo de ejecución siempre cuadrático.
 - . No aprovecha si el arreglo ya está parcialmente ordenado (siempre hace $N-1$ intercambios).
-

HeapSort

Para entender esto, antes debemos tener algunos conceptos claros:

Representación de BinaryTree en Arrey:



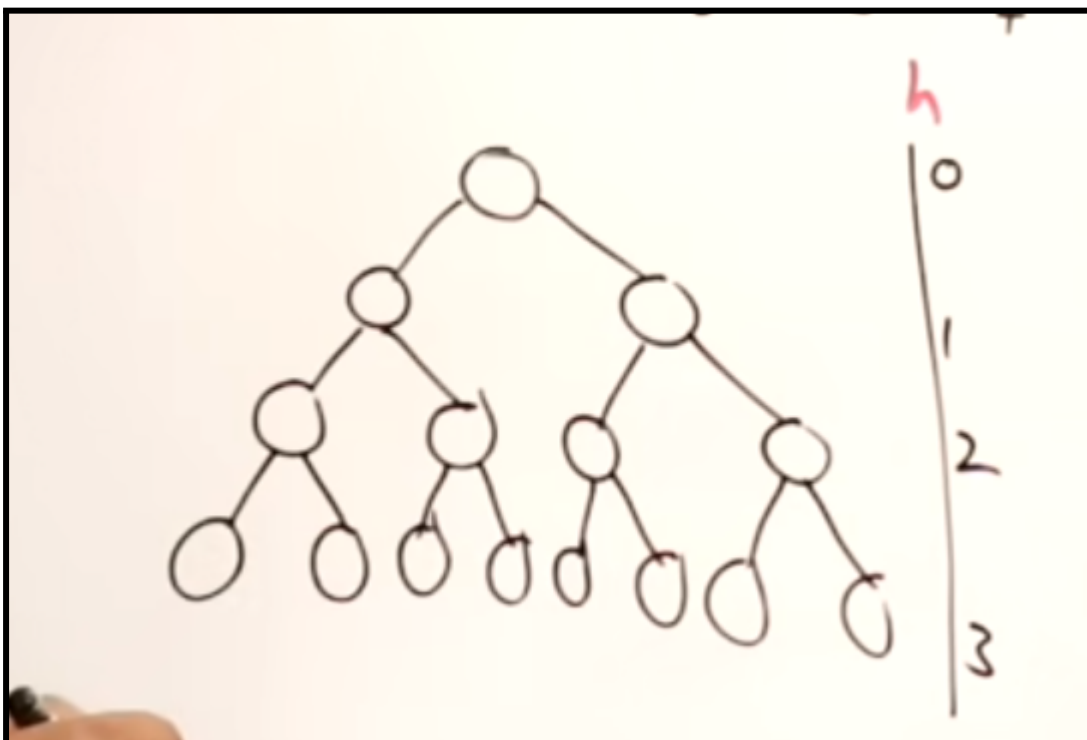
YouTube: Abdul Bari / Imagen 1

- . Aplicamos recorrido por niveles, para determinar las posiciones de cada nodo del árbol en nuestro arreglo.
- . Según Abdul, se puede saber, mirando únicamente el arreglo, que rol cumple cada nodo, es decir, quien es hijo izquierdo, derecho o padre.

- . Se ve clarito que. si tomamos al nodo “A” como raíz, que es el ubicado en el índice “1”, entonces, su hijo izquierdo debe estar necesariamente en el índice: $2*i$.
 - . Vamos a corroborarlo: $A = 1 \Rightarrow 2*1 = 2 = B$. Se cumple.
 - . Vamos a ver quien es el padre del nodo F, ubicado en el índice 6. Hacemos función piso de $i/2$, es decir $6/2 = 3 = C$. Si vemos en el árbol, se ve clarito que F es el hijo izquierdo de C.
 - . Si el árbol no está completo y le faltan hijos en el subárbol derecho de la raíz, el arreglo nos quedaría igual, solo que recortado.
 - . Si el árbol no está completo y le faltan hijos en el subárbol izquierdo de la raíz, entonces hay que reemplazar los hueco faltantes con “blanks”, como dice Abdul. Básicamente dejarlos vacíos.
-

Full Binary Tree (Árbol binario lleno)

- . Si el árbol binario en cuestión está lleno, entonces no existe la posibilidad de añadir un nodo más.



Árbol binario lleno / Imagen 2

. Nótese, que teniendo un árbol de altura 3, entonces en cada uno de esos niveles no debe de haber un hueco en donde meter un nodo más. Es decir, todas las plazas están ocupadas en cada uno de los niveles del árbol.

Complete Binary Tree (Árbol binario completo)

. A diferencia del anterior, un árbol binario está completo si no existen huecos en el mismo. Esto se ve mucho más fácil con la representación mediante arreglos.

. Desvíen su atención a la imagen 1. Se ve clarito que en el tercer árbol, a su representación por arreglo le faltan dos nodos, pues hay huecos, ya que para ese árbol, no tenemos ocupadas dos de las plazas totales del árbol. Ergo, ese árbol binario, **NO ES COMPLETO**.

. En el segundo árbol, ¿faltan elementos? Si. Es más que claro que no es un árbol lleno, pero, ¿es completo? **SI, LO ES**. Ya que, mirando su arreglo, no hay huecos restantes, ya que los elementos que faltan no están entre medio de otros elementos del árbol (de otros nodos).

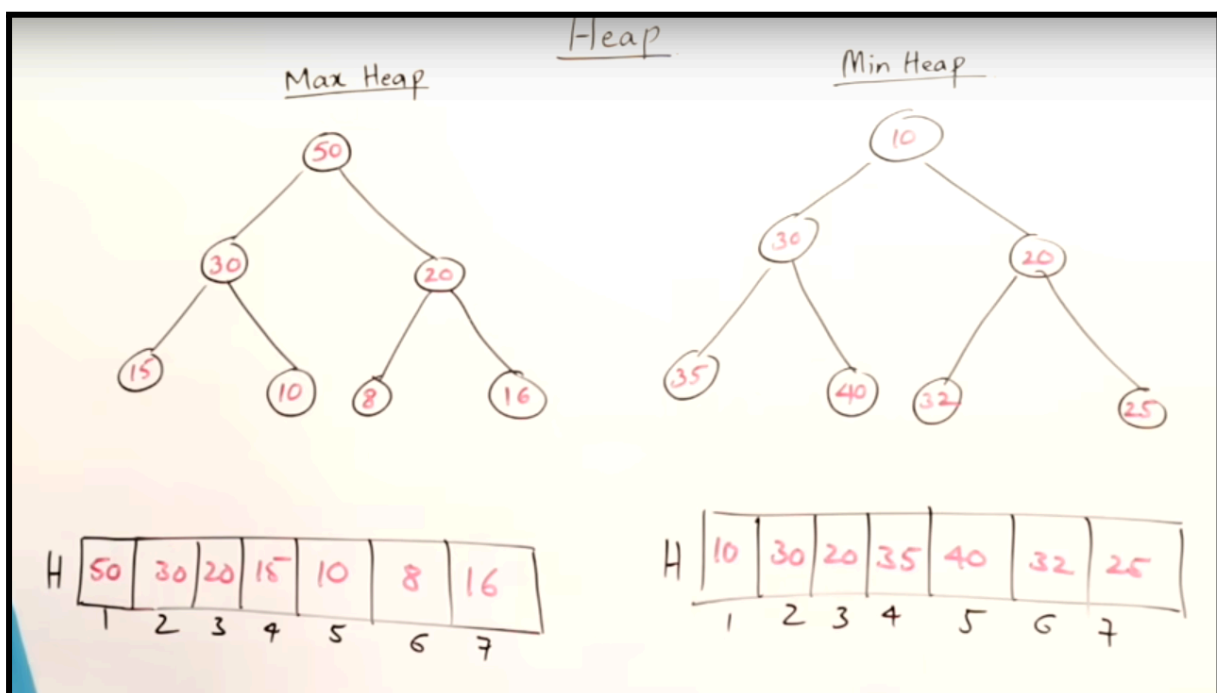
. El primer árbol, **ES COMPLETO Y ES LLENO**.

. El segundo árbol, **ES COMPLETO Y NO ES LLENO**.

. El tercer árbol, **NO ES COMPLETO Y NO ES LLENO**.

Heap

. Un **montículo (heap)** es un **árbol binario completo**. Eso quiere decir, que si el árbol binario está lleno, también es un heap (pues si el árbol está lleno entonces es completo).



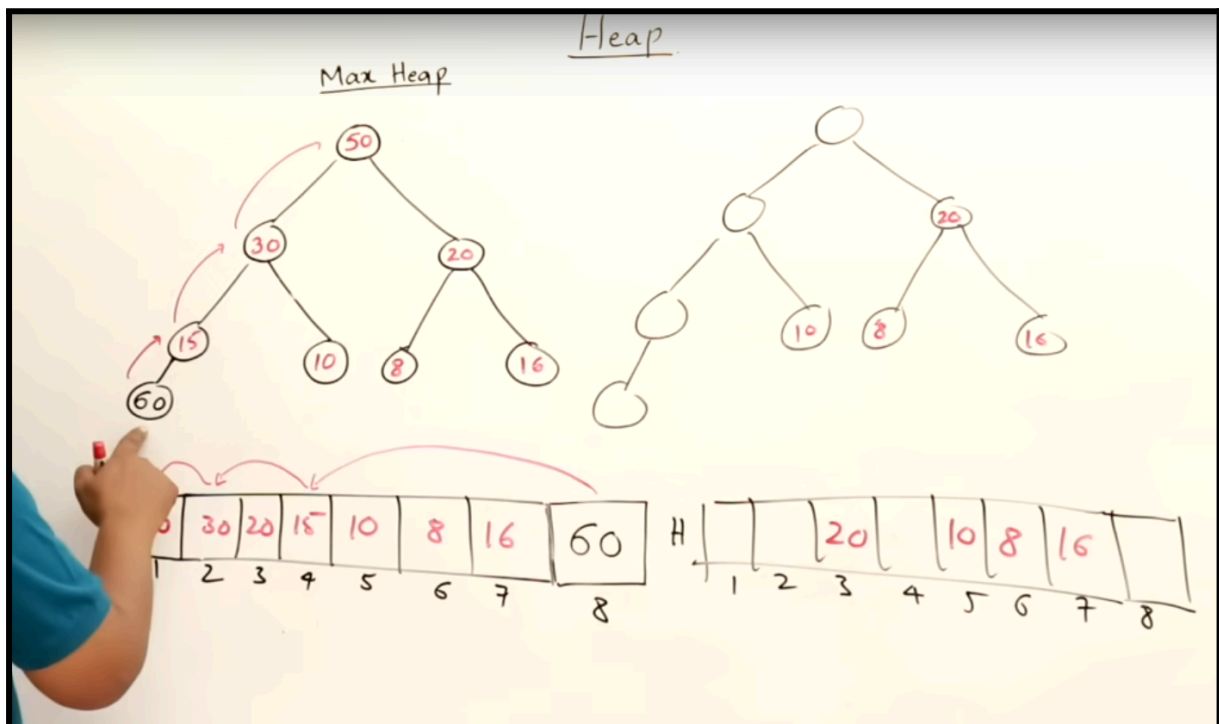
- . Si en el heap, el primer elemento, y consecuentemente todos los padres son mayores que sus hijos; entonces a eso se le llama MAX-HEAP.
 - . Si los descendientes son mayores que sus padres, lo que quiere decir que el heap crece de arriba a abajo, entonces se le llama MIN-HEAP.
-

Inserción en un MAX-HEAP

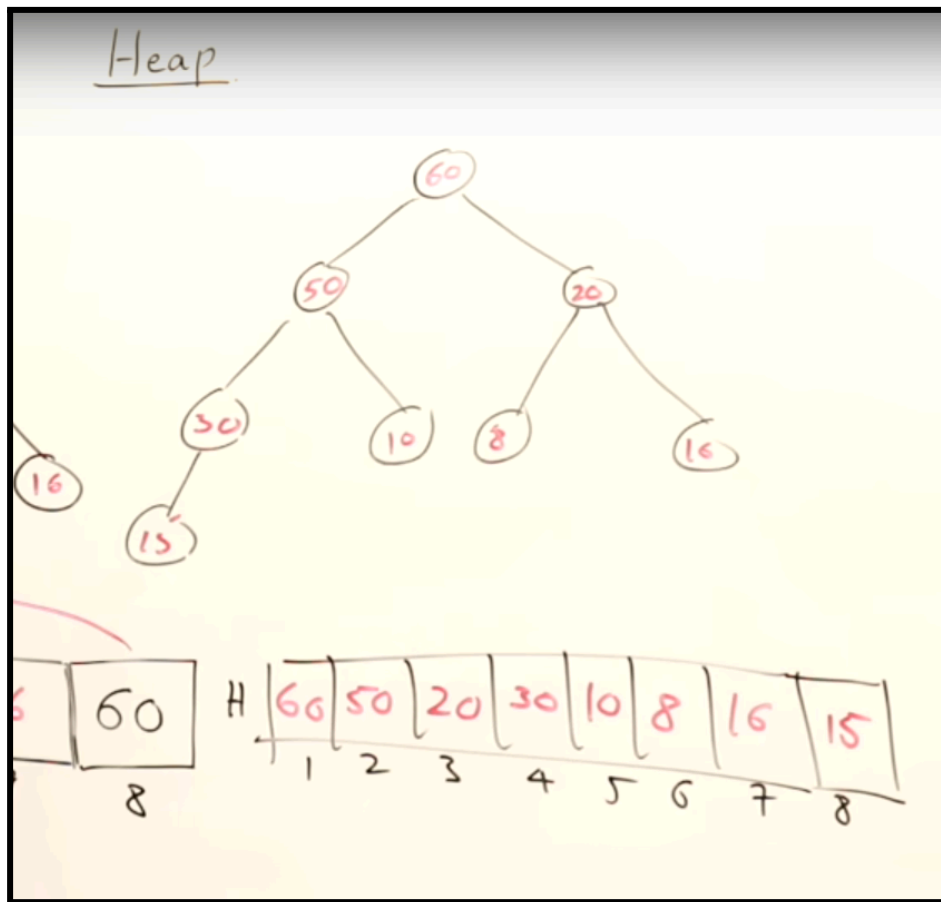
- . Vamos a tomar el max heap de antes. La idea es insertar un nuevo nodo 60.
- . Es erróneo pensar, que como 60 es más grande que la propia raíz, debería insertarlo directamente ahí y desplazar a la derecha el 50 y todos sus descendientes. De ese modo estaría generando huecos, y por ende perdería la propiedad del árbol completo.

Funcionamiento:

- . Inserto el 60 en la última posición del array, quedando, visualmente a la izquierda del todo de mi max heap.
- . Luego lo que voy a hacer es comparar ese 60 con sus padres, hasta llegar a la raíz.
- . Su índice es 8, entonces hago $i/2 = 8/2 = 4$ (llego al nodo 15, su padre), luego hago $i/2 = 4/2 = 2$ (nodo 30), luego hago $i/2 = 2/2 = 1$ (llego a la raíz 50).



. El nodo 60 va a ir “escalando” entre sus descendientes hasta llegar a la raíz, una vez llega, va a correr a todos ellos una posición, logrando ordenar el max heap.



. De este modo, tenemos un árbol binario completo.

Complejidad de insertar en max heap?

. Nótese que la cantidad de intercambios que hicimos es igual a la altura del árbol. Por lo cual insertar un elemento del árbol lleva el mismo tiempo que encontrar la altura del mismo, que es exactamente Big O ($\log n$).

. En este caso tenemos el peor caso, pues 60 era más grande que cualquier elemento del heap. Si el elemento fuera 6, siendo el más pequeño entre todos, entonces insertarlo solo nos llevaría Big O (1), siendo el mejor caso.

Eliminar en max heap

. Chad Abdul, muy inteligentemente, nos proporciona la analogía de que, un heap es como un montículo de manzanas en el super, supongamos que el TATA.

Cuando vamos a agarrar una manzana de la montaña, ¿cual agarramos? una del costado? sos nabo? no, claramente elegis la primera, la más linda seguramente.

En la eliminación en max heap se sigue la misma lógica, solo vas a sacar el primer elemento, osea, la raíz.

. Al eliminar la raíz, lógicamente generamos un desequilibrio, que, naturalmente debemos arreglar. ¿Qué hacemos?

. Selecciono el último elemento del arreglo del max heap. Ese elemento va a ser la nueva raíz.

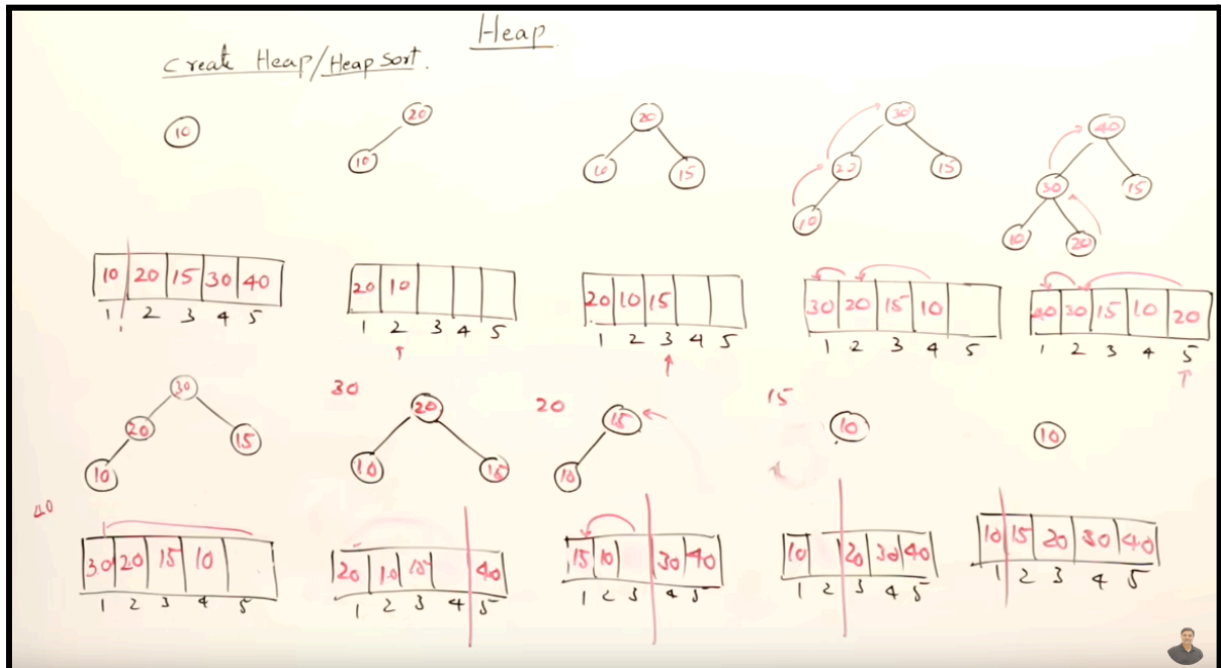
. Después, comparo los dos hijos; y selecciono el más grande para ocupar su lugar. Eso lo voy a hacer recursivamente hasta que los elementos menores me queden abajo y los mayores arriba.

. Si quiero aplicar el mismo procedimiento para un min heap, funciona exactamente igual, solo que ahora el heap crece de arriba a abajo, por lo tanto debo procurar mantener el mínimo siempre arriba.

Complejidad temporal

. Al igual que para insertar, nos va a llevar la altura del árbol, osea Big O ($\log n$).

Estrategia Heap Sort



- . Tengo un conjunto de elementos desordenado:
- . Primero los inserto en un heap (max heap en la foto).
- . Segundo, los elimino del heap, haciendo paso a paso la estrategia de eliminación mencionada anteriormente.
- . A medida que vaya eliminando, se me van a ir quedando los elementos ordenados de derecha a izquierda, en el mismo arreglo original. Esto ocurre, ya que en cada eliminación, estoy sacando el mayor elemento, el cual va quedando a la derecha del arreglo.

Complejidad temporal

- . Consideremos que tenemos un conjunto de n elementos a insertar. Insertar en un max heap, como vimos toma $\text{Big O}(\log n)$, si es para n elementos, entonces lleva $\text{Big O}(n \log n)$.
- . Para eliminar, ocurre igual, es n veces $\log n$. Ergo, $\text{Big O}(n \log n)$.
- . Total: $2 \times (n \log n) = \text{Big O}(n \log n)$.
- . Y eso se cumple para TODOS los casos, tanto el mejor, como el promedio, como el peor.

Algoritmo de ordenación “Divide y vencerás”

- . Divide el arreglo, ordena cada mitad y luego los mezcla.
-

QuickSort

- . Ya visto.
-

MergeSort

- . Recursivo.
- . Tiene dos funciones principales, la primera divide el arreglo en subarreglos; y la segunda los mergea.

Funcionamiento:

- . Parto de un arreglo de n elementos.
- . Divido esos elementos en subArreglos.
- . Recursivamente, divido esos subArreglos hasta llegar a pequeños arreglos de dos elementos.
- . Comparo esos dos elementos, y, los coloco en el subArreglo anterior, ordenados.
- . Aplico recursión hasta que todos los elementos queden ordenados.

Complejidad temporal

- . Big O ($n \log n$) en todos los casos.

Complejidad especial

- . $O(n)$, pues necesita crear subarreglos constantemente.
- . Esto es una desventaja ante otros algoritmos de ordenación (Bubble sort, insertion sort, selection sort), que usan $O(1)$.

Algoritmos de ordenación por coneteo

Counting Sort

. Algoritmo lineal muy eficiente cuando los elementos a ordenar son números enteros, no negativos en un rango acotado.

Funcionamiento:

. Se cuenta cuántas veces aparece cada valor del arreglo de entrada (usando un arreglo auxiliar `count[]`).

. El tamaño de ese arreglo, va a ser igual al valor del elemento más grande del arreglo original.

Complejidad temporal

. Big O ($n + k$) => donde n es el número de elementos del arreglo, y k es el valor máximo que puede tomar cualquier elemento.

Complejidad espacial

. $\Theta(n + k)$.

Algoritmo de clasificación por distribución

BucketSort

- . Se crean buckets para poner elementos dentro.
 - . Se aplica algún tipo de algoritmo de ordenamiento (ejemplo: insertion sort) para los elementos en cada bucket.
 - . Finalmente los sacamos y unimos para conseguir el arreglo ordenado.
-
- . Es funcional si nuestros elementos son uniformes. Si hay mucha repetición de elementos, entonces se puede volver lento.

Complejidad temporal

- . En el peor caso, Big $O(n^2)$, si todos los elementos caen en la misma cubeta, habría que hacer insertion sort para esos n elementos, pudiendo desenlazar en un n^2 .
- . Caso promedio y mejor caso es Big $O(n + m)$.

Complejidad espacial

- . Hay que añadir esas cubetas o urnas, por lo que hace $O(n + m)$.
 - . Es decir, que en la práctica, binsort corre en Big $O(n)$ y ocupa $O(n)$ espacio extra para su funcionamiento.
-

Radix Sort

Radix sort es un algoritmo de ordenación no comparativo que clasifica registros con claves compuestas por varios campos (o dígitos) ordenando iterativamente por cada campo, del menos significativo al más significativo.

Supuestos

- Las claves de los registros están formadas por k componentes: f_1, f_2, \dots, f_k , donde cada f_i es de un tipo t_i .
- Se desea clasificar los registros en orden lexicográfico respecto a esas claves.

- Ejemplos de claves:

- Una fecha: {día: 1..31, mes: 1..12, año: 1900..1999}

- Una cadena de texto: array[1..10] of char

Idea principal

Radix sort funciona aplicando Bin Sort o algún método estable sobre cada componente de la clave,

comenzando por el menos significativo (f_k) hasta el más significativo (f_1).

Cada paso asegura que el orden previo se conserva gracias a que se utiliza una ordenación estable.

Tabla comparativa de algoritmos sorting

Algoritmo	Tipo	Peor caso	Caso promedio	Mejor caso	Espacio extra
Insertion Sort	Inserción	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Shell Sort	Inserción mejorada	$O(n^2)$	$O(n^{\{1.26\}})$	$O(n \log n)$	$O(1)$
Merge Sort	Mezcla	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
QuickSort	Intercambio / Divide y vencerás	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Bubble Sort	Intercambio	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Selection Sort	Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
HeapSort	Selección (Heap)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Bucket Sort	Distribución (Buckets)	$O(n^2)$	$O(n + m)$	$O(n + m)$	$O(n + m)$
Radix Sort	Distribución (Radix)	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(n + k)$
Counting Sort	Distribución (Conteo)	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$