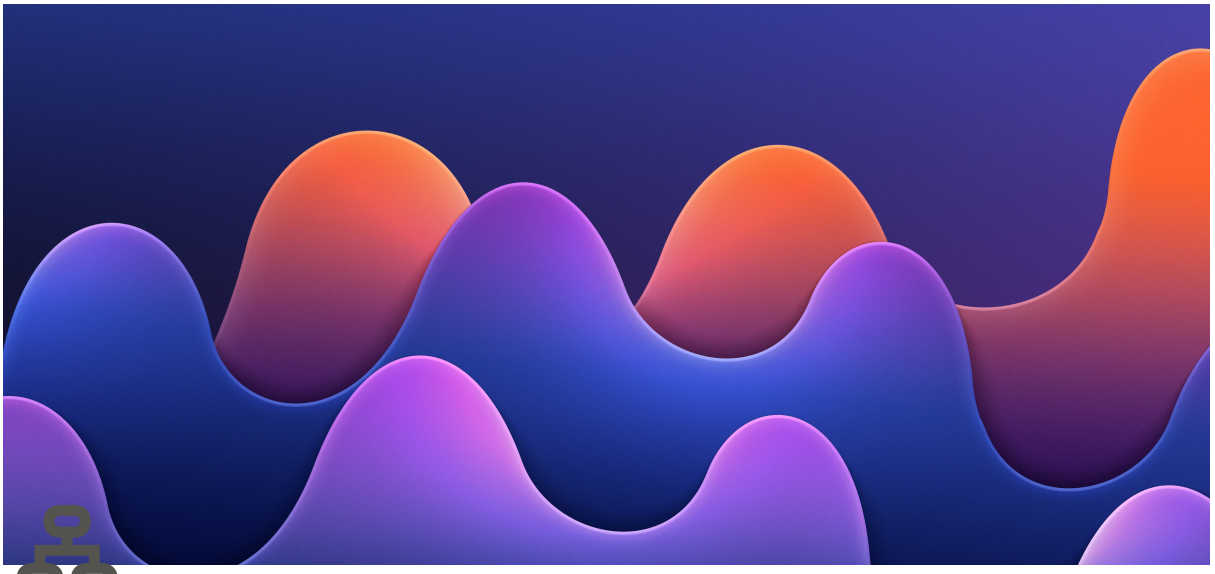


Apuntes - EDD IIC2133

Clase 1

SELECTION SORT

Tips: Implementar tanto en Python como en C

- El ordenar tiene ventajas asociadas a **ENCONTRAR**
- Es más fácil encontrar algo cuando buscando algo **ORDENADO**

PROBLEMA CONCEPTUAL

Al no tener ordenado, el peor caso que tenemos al buscar el dato particular, es que no este, y tenemos que recorrer uno por uno hasta el final

Una de las cosas que podemos hacer si tenemos datos ordenados es usar **BINARY SEARCH,** partir en la mitad el arreglo, y buscar con complejidad **Lóg2(n)**

Comparado al caso desordenado tenemos que hacer complejidad O(**N)**

Si los datos son mucho, el log es mucho mas pequeño que el lineal

Ojo: Si puedo poner una etiqueta, puedo ordenar !!!

SECUENCIA

Una secuencia de números a1,a2,a3,...,an se dice **ORDENADA**, (no decrecientemente) si se cumple que

a1 ≤ a2 ≤ ..... ≤ an

¿Qué es ordenar una secuencia ordinaria? —:

EJEMPLO DE ALGORTIMO

Busca el menor , Lo saca de la lista, y lo coloca al final de otra

Imagen:

Complejidad: O(N**2) : Ya que por cada n elementos recorrió máximo los n elementos para encontrar el menor

Complejidad Memoria: EL DOBLE , sacamos una lista y escribimos otra

Pseudocidigo!

```
// Pseudocodigo
1 Encontrar(L)
    2   Iniciamos L' Vacia
    3   Encontramos el menor valor en L
    4   Borrar el valor encontrado
    5   Escribirlo al final de L'
    6   Si quedan valores en L, volver a la linea 2
        Return L
```

DEMOSTRACIÓN CON INDUCCION

Vamos a demostrar con inducción que Cumple el algoritmo lo pedido:

Necesitamos caso base:

Tomar un elemento sacarlo y verificar si esta ordenado: Se cumple porque solo tiene un elemento

CB: Se cumple de n = 1

HI: Suponer que se cumple para n (pasos)

Por definición en el paso N, nuestro algoritmo cumple

TI: Por probar que se cumple para el n + 1

Recorremos toda la lista, encontramos el menor, lo borramos , lo ponemos en la lista de salida,

Está ordenada? Si, en los pasos anteriores saque todos los anteriores que son menores que el que estoy tomando en el paso n + 1

En la lista original solo quedan valores mas grande

Por definición se que cualquiera de la lista de salida es menor que el que añadimos

CORRECTITUD DE ALGORITMOS

Es correcto cuando **Termina y Cumple su Proposito**

PRINCIPIO DE INDUCCIÓN SIMPLE

CB, HI, TI (Discretas)

Esto es simple, el caso base es trivial, el inductivo es donde esta la magia

A veces el caso base son varios : Normamente en ese caso elegir caso base mas intuitivo, si se cumple, es OK

A veces se cae, porque el algoritmo requiere de mas casos, por ejemplo, con 2 listas, el caso base no podría ser uno, y tendría que ser 2

VOLVIENDO AL EJEMPLO ANTERIOR

Para hacer la inducción, partimos con un solo dato y la lista vacía, sacamos el valor, lo colocamos en la lista de salida, y la lista original está vacía, por lo tanto el algoritmo acaba

Esta la lista de salida ordenada? : SI

Por lo tanto, se cumple el caso base

> Demostración (finitud)
>
> Primero demostraremos que termina en una cantidad finita de pasos.
>
> Sea n la cantidad (finita) de valores en la lista original L.
>
> En cada iteración del algoritmo se borra un valor de L y se escribe en la lista nueva L'.
>
> Luego de n iteraciones, todos los valores en L fueron borrados. Debido a la línea 5, que verifica si existen valores en L, el algoritmo no sigue iterando.
>
> Por lo tanto, el algoritmo termina en una cantidad finita de pasos. □

Este sería la explicación mas formal, Sirve como respuesta para una prueba

P(n) := Los primeros n valores borrados de L, son los menores de L, y están ordenados en L′

El algoritmo que acabos de ver se llama **SELECTION SORT**

SELECTION SORT

Elegir ordenadamente

1. Ir a la Lista
2. Buscar el mas pequeño
3. Sacarlo y ponerlo en la Lista L′

Revisar cis.usfca.edu para analizar los algoritmos de ordenamiento?

☐ El link esta en las laminas de clase, revisar el link al momento de tener el ppt

- La **complejidad** de **SELECTION SORT: O(N**2)**

De forma explícita

- Buscar el menor elemento de A implica revisar n elementos en la primera iteración
- En la siguiente iteración, como A tiene un elemento menos implica revisar $n-1$ elementos
- Este proceso se repite hasta agotar A cuando tiene 1 elemento
- El tiempo del algoritmo es la suma de los tiempos de cada iteración

$$T(n) = n + (n-1) + \cdots + 2 + 1 = \sum_{i=1}^{n} i = \frac{n(n-1)}{2}$$

- Concluimos que $T(n) \in \mathcal{O}(n^2)$

Buenas Prácticas:

- Tener en Arrays, un **puntero guardado al final,** Así no tenemos que recorrer la lista completa, Para no tener que recorrer toda la lista para indexar

TIP

☐ Intentar programar como pseudo código de SELECTION sort, para que haga el mismo algoritmo pero en una misma lista y no tener que duplicar código

Clase 2

Ordenamos para buscar

Complejidad **O(n**2)**

La forma en que vienen los datos, no afecta la complejidad

Selection Sort es muy fácil de entender, se usa para datos no tan grandes, es simple, y rapido.

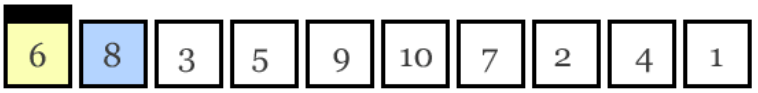
- No me voy a equivocar en usarlo → Es + Facil
- Se usa para un set **PEQUEÑO** De datos
- Es **Rapido** para pocos datos

SELECTION SORT (IN PLACE)

```
// input : Secuencia A[0...n-1], largo n >= 2
// output: vacio primeramente

// PSEUDOCODIGO
SelectionSort(A, n): // I = POSICION SALIDA // J = POSICION A
    for i = 0...n-2: // Posicion que estoy ordenando -> posic
        min = i
        for j = i + 1....n.1:
            if A[j] < A[min]:
                min = j
        A[i] -><- A[min] // Intercambio

// C
POR HACER!!!!!
List* SelectionSort(){
    ...
}
```



Yellow is smallest number found
Blue is current item
Green is sorted list

TIP

☐ Jugar con el algoritmo, tratar de usar slectionsort para mas de una pinta, por ejemplo dos atributos para ordenar

ATRIBUTOS GENERALES

1. O(n**2)
2. O(n) swaps: N intercamibios
3. O(1) en memoria → In place
4. No adaptativo - Sin mejor caso
5. No estable! → Si los datos de entrada venían 2 datos iguales distinguibles (al mirar la salida) no garantiza igual orden en 2 criterios

EN PRUEBAS

Para la prueba: APUNTES hecho a mano, (o en ipad impreso) → ~~TORPEDO~~ PARA NO MEMORIZAR ALGORITMOS

PREGUNTAS:

- Nos den el algoritmo, y arreglarlo : El
- Aquí esta el algoritmo → Decir lo que hace → funciona? termina? complejidad?

- Demostrar correctitud?

De la clase anterior

Si ya tenemos los datos ordenados, y queremos ordenar algunos de manera mas rapida que selection sort!!!

Esto sería selection sort

“Imaginemonos que tenemos que ordenar naipes, y no pasaron bloques de a 4, puedo tomar el 5 y ver donde va, tomar el 6o y ver donde va” → Ordenamiento en la practica

Da la idea intuitiva, de que si ya estaban ordenados los datos, es mas eficiente que aplicar logica de selection sort, ya se que la salida esta ordenado, no tendre que moverlo todo!

```
// PSEUDOCODIGO
// Input: Secuencia de datos A
// output Nueva secuencia de datos B, ordenada

 InsertionSort(A): // No estamos ordenadamente, estamos inser
    1 Dada secuencia B, inicialmente VACIA
    2 Tomar primer dato de x de A
    3 Sacar x de A
    4 Insertar x en B de manera que B quede ordenada
    5 Si quedan datos en A, volver a la linea 2
    return B

// La lista B se construye vacia, vamos a dejar ordenados en
```

DIFERENCIAS ENTRE SELECTION Y INSERTION

En cada paso de insertion sort, la lista B esta **Ordenada**

Insertion sort:

1. Toma el primer elemento de A, lo saca y lo mete en B
2. Toma el siguiente elemento de B, lo inserta en B, pero lo inserta Ordenadamente

Intuitivamente sabemos que termina

Para procesar los n elementos de la lista de entrada serian N pasos

La linea 4 de Insertar ordenadamente, tiene una complejidad de O(n)

Porque tenemos que mover los elementos de un array, y moverlos cuesta O(n)

Por lo tanto, la complejidad final de **InsertionSort** sería **O(n)**

Proposicion:

En el paso N, la lista esta Ordenado

CB: Tomar el primer elemento, lo pongo en la de salida, y esta ordenado → Cumple caso base

Para hipotesis y tesis, en el paso N, tenemos la lista ordenada, tomamos un siguiente elemento, y lo insertamos ordenadamente en la lista de salida, y con eso seguirá ordenada

Demostracion formal (tipo discretas)

Definir la lista de salida, y la posicion del dato nuevo i

Como el dato nuevo satisface que esta entre a_1 y a_i

Cuando lo inserte en la nueva lista, seguira entre ellos

- [ ] Revisar PPT!!!!!!

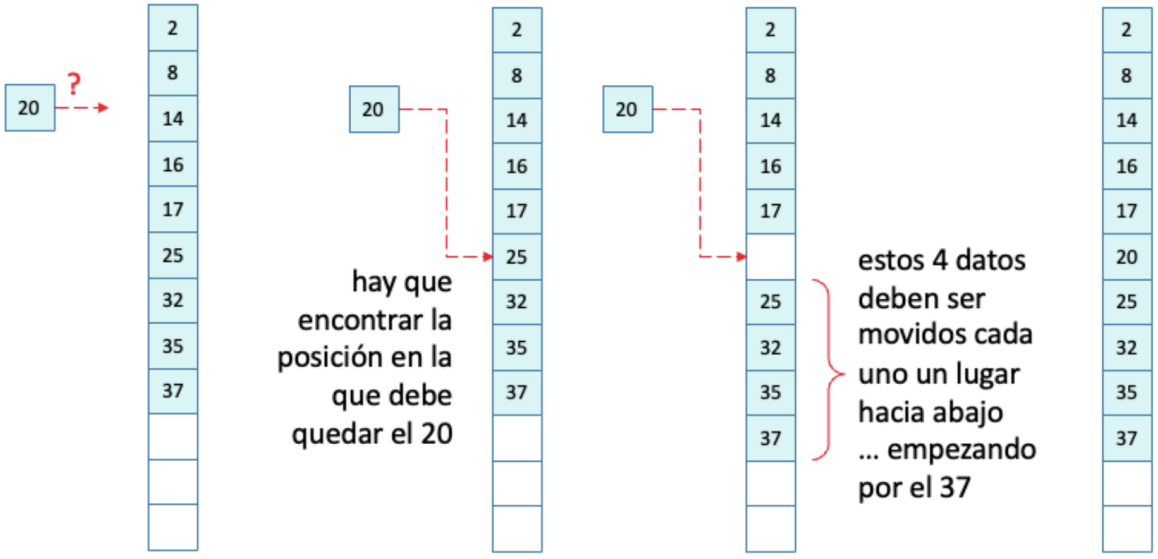
Complejidad de la insercion?

CASOS

1. Usando Arreglos: Saber posicion final del arreglo
2. En lista, basta tener un puntero al final de la lista

El tema de insercion

Puede ser lineal? → Uno a uno contar? → Puede ser, pero en grandes cantidades de datos ineficiente



Existen muchas formas de implementar esto, alguna mas eficientes que otras

En el caso promedio, voy a mover la mitad de los datos

INSERTAR EN ARREGLOS

O(N) en Arreglos

Se puede usar en una prubea → Usarlo y no demostrarlo

INSERTAR EN LISTAS

Ubicar posición en el arreglo → Tenemos que recorrer la lista

O(N) en Listas

CONCLUSION

Como la primera iteracion es O(N), y la Insercion es O(N):

Complejidad de **INSERTIONSORT: O(N**2)**

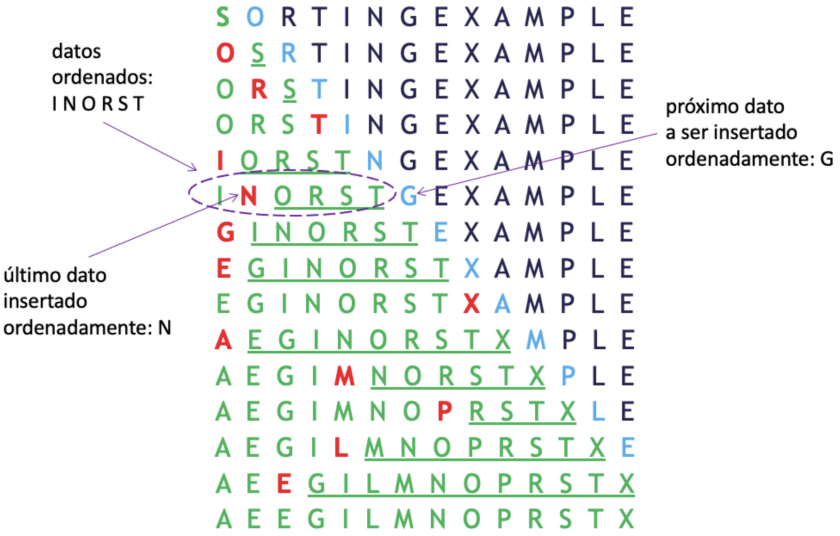
SELECTION SORT (IN PLACE) → la misma lista

```
InsertionSort(A, n):
    for i = 1...n-1:
        j = i
        while (j>0) y (A[j] < A[j-1]): // J > 0 -> No salirno
            Intercambiar A[j] con A[j-1]
            j = j - 1 // Para revisar a la izquierda


// Notar que si los datos vienen ordenados -> Nunca entra en
// Por eso O(1) por decirlo asi

// LA diferencia:
// Selection Sort, no depende de como vienen los datos
// Selection sort, si depende de como vengan
```





- Verde es el actual
- Procesar las celestes
- Negras no las vistas aun
- Rojas las movidas

Esta forma, la veremos despues en **QUICKSORT,** no es tan grato iterativamente → Es mejor verla recursivamente

INVERSIONES

Si entre o y 1, existe un i y j

- Si **i < j**
- y tenemos que **A[i] > A[j]**,

Entonces diremos que el par ordenado **(i,j) es una INVERSION**

Solo entramos al While cuando haya una inversion → La cantidad de whiles sera la cantidad de inversiones

For de i → O(n)

Dentro de el, cantidad inversiones, que depende del numero de iversiones

O(N + k) →

- N : Cantidad de Elem de lista
- K : seria la cantidad de inversiones !!!

MEJOR CASO: 0 Inversiones → Mejor caso de insertionsort : **O(N)**

PEOR CASO: Cada indice esta invertido → **O(N**2)**

En el peor caso, el arreglo tiene el número máximo de inversiones

- El arreglo está ordenado decrecientemente
- Al comparar cada índice i con $j > i$, se cuenta una inversión

$$k = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- De forma equivalente, el número de inversiones es el número de pares de índices

$$k = \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

- La complejidad en el peor caso es $\mathcal{O}(n^2)$

¿Qué hay del caso promedio?

☐ Estudiar esto → Hay dudas de como funciona

CASO PROMEDIO

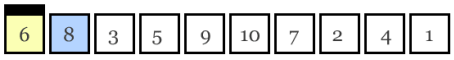
Queremos calcular el numero de inversiones promedio

Numero total de inversioones posibles / El numero total de permutaciones posibles

INSERT vs SELECTION (In Place)

```
SelectionSort (A, n):
1    for i = 0 ... n − 2 :
2        min ← i
3        for j = i + 1 ... n − 1 :
4            if A[j] < A[min] :
5                min ← j
6        A[i] ⇋ A[min]
```

```
InsertionSort (A, n):
1    for i = 1 ... n − 1 :
2        j ← i
3        while (j > 0) ∧ (A[j] < A[j − 1]) :
4            A[j] ⇋ A[j − 1]
5            j ← j − 1
```



Yellow is smallest number found
Blue is current item
Green is sorted list



Clase 3 - MergeSort

De la clase pasada

El costo de una **Recursion** va al S.O

Es notorio que un algoritmo eficiente como Quicksort nlogn, pero cuando vamos al terreno de muchos datos, porque aunque sea bonita, se paga

Insertion sort en el mejor caso O(n)

Es estable

La gracia que no tiene selection sort, puede ordenar mientras van llegando, puede ordenarlos en el orden de llegada

Usarlo con cantidad pequeña, menos de 100 datos → SelectionSort buena Opcion

MERGESORT

O(n) → Merges

O(log) → Divisiones

Usamos Dividir para conquistar

Ahora lo usaremos como algoritmo

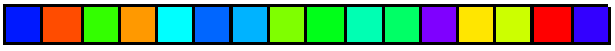
Como estrategia sirve para descomponer problemas de arboles, bases de datos, etc

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o m ás) sub-problemas del mismo tipo
2. Resolver recursivamente cada sub-problema
3. Encontrar soluci ón al problema original combinando las soluciones a los sub-problemas

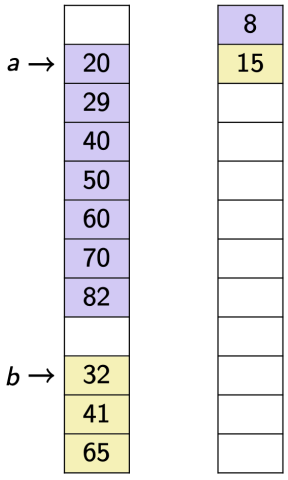
Binary Search Usa Dividir para conquistar

MERGE SORT ANIMACION

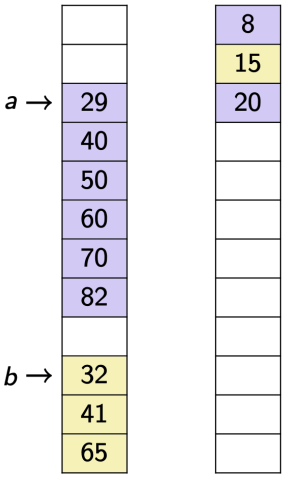


1. MERGE

```
// NO IN PLACE! -> Iniciar lista vacia C
input : Secuencias **ordenadas** A y B
output: Nueva secuencia ordenada C
Merge(A, B ):
    1 Iniciamos C vacia
    2 Sean a y b los primeros elementos de A y B
    3 Extraer de su secuencia respectiva el menor entre a y b
    4 Insertar el elemento extraido al **final** de C
    5 Si quedan elementos en A y B, volver a l ínea 2
    6 Concatenar C con la secuencia que au ń tenga elementos
    7 return C
```



Estado luego de la segunda iteración



Estado luego de la tercera iteración

DEMOSTRACION CORRECTITUD MERGE

1. **TERMINA (FINITITUD):**
    a. En cada iteracion, Siempre se extrae un elemento de A o B y se inserta en C

b. Y cuando una secuencaia (A o B) se vacía → Se llena C con la otra

c. En total se realizan

   i. **n = |A| + |B| INSERCIONES** y

   ii. un numero menor a **n COMPARACIONES**

d. Por lo tanto el algoritmo termina en una **cantidad finita de PASOS**

2. **CORRECTITUD → El algoritmo cumple lo pedido**

   a. Para A, B consideramos la propiedad

      i. **P(n) :=** Luego de insertar el n-esimo elemento en **C → A, B , C están ORDENADAS**

   b. **CASO BASE: P(1) :** Este es el estado luego de insertar 1 elemento en C

   - Como extrajimos el **menor elemento** de una secuencia, estas siguen ordenadas,
   - trivialmente se cumple si la lista queda vacia
   - Como **C** Solo tiene **1 ELEMENTO** → este esta ordenado.

   □ .

   c. **HIPOTESIS INDUCTIVA:**

   - **SUPONEMOS** que al agregar el n-esimo elemento a C, todas las sec (A,B,C) siguen ordenadas

   d. **TESIS INDUCTIVA:**

   - **Por DEMOSTRAR:** al agregar el (n+1)esimo elemento a C, todas las secuencias siguen ordenadas

   2 CASOS

   1. Si quedan elementos en A y en B → sea C_n+1 el menor entre las cabezas de A y B

2. Sin perdida de generalidad, si solo quedan elementos en A → sea C_n+1 la cabeza de A.

Se elimina C_n+1 de su secuencia y se agrega en C

> Demostración (propósito)
>
> Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.
>
> Por **H.I.** tenemos que los primeros n elementos de C cumplen
>
> $$c_1 \leq \cdots \leq c_n$$
>
> Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones
>
> - A o B no están ordenadas (ya probamos que lo están)
> - c_{n+1} es extraído en una iteración anterior por el criterio de selección
>
> Luego, concluimos el resultado buscado
>
> $$c_1 \leq \cdots \leq c_n \leq c_{n+1}$$
>
> □

COMPLEJIDAD MERGE

1. EN MEMORIA

- La ejecucion del algoritmo que vimos se hacia pidiendo memoria extra n = |A| + |B| → necesitamos memoria adicional **O(n)**

- Tambien se puede realizar in place
  → Usar el mismo espacio reservado a A y B → memoira adicional **O(1)**
  → Impacta la complejidad del tiempo

1. COMPLEJIDAD CON ESPACIO EXTRA

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
   - Se saca un elemento de la restante $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$

Usando $\mathcal{O}(n)$ memoria adicional, `Merge` es $\mathcal{O}(n)$

😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄😄

2. COMPLEJIDAD IN PLACE

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en al comienzo $\mathcal{O}(n)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$
2. Reubicación de la secuencia no vacía restante: no necesario
   - El resto de los elementos está en su posición correcta

Usando $\mathcal{O}(1)$ memoria adicional, `Merge` es $\mathcal{O}(n^2)$

☹️

Merge **in place** → Es mucho mas ineficiente , pero usa menos memoria

Merge **Not in place** → Mucho mas eficiente, pero usa mas memoria

MergeSort

Nace de usar Dividir para conquistar para ordenar, usando Merge → ¿Donde?:

IDEA GENERAL DE MERGESORT

La idea general para ordenar usando Merge define un nuevo algoritmo que llamaremos MergeSort

1. **Dividir** la secuencia original en **dos sub-secuencias**
2. Llamamos **recursivamente** a **MergeSort** sobre las dos sub-secuencias
3. **Combinamos las secuencias** ordenadas resultantes mediante **Merge**

ALGORITMO MERGE SORT

```
input : Secuencia A
output: Secuencia ordenada B

MergeSort (A):
    1 if |A|=1 : return A
    2 Dividir A en mitades A1 y A2
    3 B1 ← MergeSort(A1)
    4 B2 ← MergeSort(A2)
    5 B ← Merge(B1,B2) // ORDENAR AMBAS SECUENCIAS
    6 return B
```

CLASE 4

21/08/2024

Costo de memoria adicional → Costo de mezclar 2 listas

Estrategia de dividir para conquistar

Llegamos hasta 2 listas ordenadas tiriviales → 1 solo elemento

Mezclamos → tenemos 2 listas ordenadas → mezclamos → 4 listas ordenadas

.....

O(n) mezclas

log(n)

Complejidad Mergesort: **O(nlog(n))**

No se ve afectado por los datos de entrada

Siempre va a ser nlogn

HASTA EL MOMENTO

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

- Notemos la mejora en tiempo con MergeSort a cambio de memoria adicional

DIVIDIR PARA CONQUISTAR

1. Divide el problema original en dos o mas subproblemas
2. Resolver recursivamente cada sub-problema

3. Encontrar solucio ́n al problema original combinando las soluciones a los sub-problemas

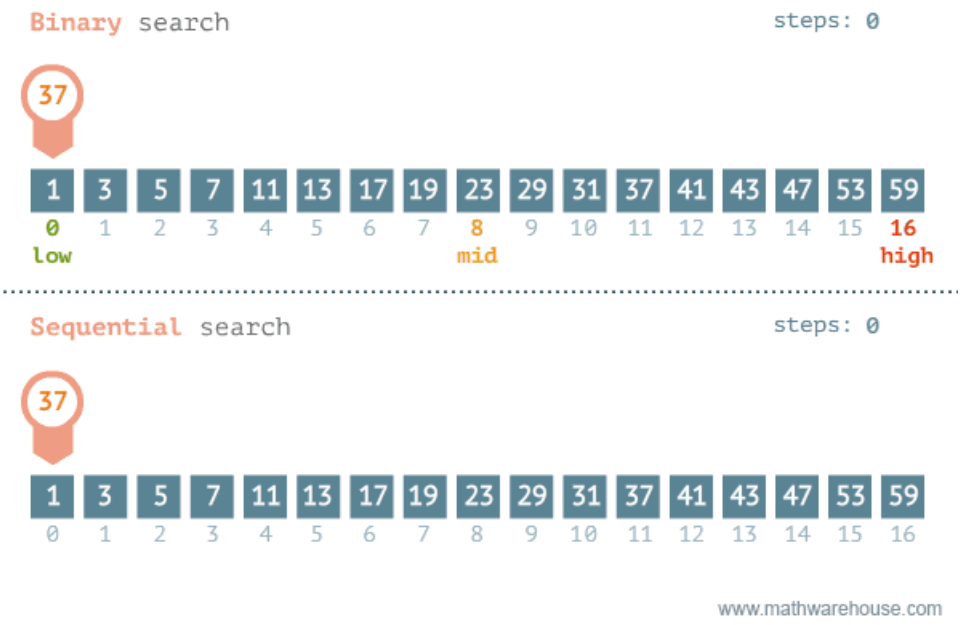
BUSQUEDA BINARIA

- [ ] Escribir en el torpedo para la prueba!

```
input : Secuencia A[0,...,n−1], elemento x,indices i, f
output: Índice m∈{0,...,n−1} o −1
// Asumimos que esta ORDENADO

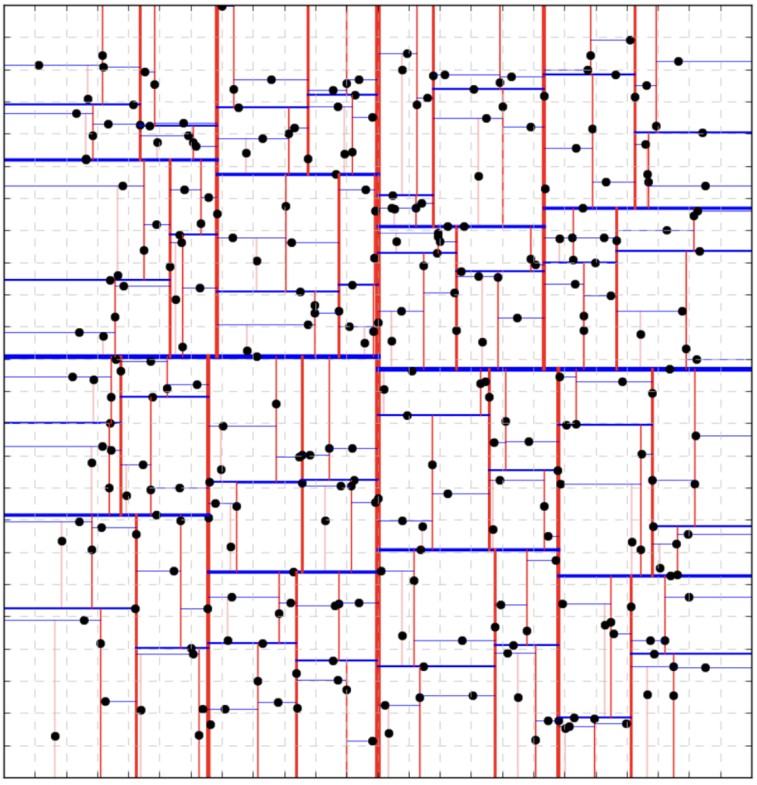
BinarySearch (A,x,i,f):
    1 if f<i : return −1
    2 m = ⌊i+f⌋/2
    3 if A[m] = x : return m
    4 if A[m] > x :
        5 return BinarySearch (A,x,i,m−1) // Lado izquierdo
    6 return BinarySearch (A, x , m + 1, f ) // Lado derecho
```

- La division en subproblemas se hace escogiendo un pivote central (linea 2)
- La naturaleza de la busqueda hace que sea necesario resolver solo uno de los subproblemas
- Solucion al problema es exactamente la solucion al llamado recursivo



EJEMPLO

- Consideremos el problema de procesar un gran conjunto de coordenadas en 2D
- Para repartir la carga, se reparten los datos en zonas rectangulares
- La idea es que los rect ángulos particionen el espacio 2D
- Ademas queremos que cada zona tenga la misma cantidad de datos



Si tengo datos ordenados con la coordenada x, no necesariamente estan ordenados con la coordenada y

Asumo que en mi base de datos, los datos estan al **azar**

El problema quiere llevarnos a encontrar la mediana, donde podamos dividir el problema en mitades! → Horizontal y verticalmente

Objetivos

1. Aprender a **elegir pivote** para **PARTICIONAR SECUENCIA**
2. Identificar cuando tal estrategia encuentra la **MEDIANA**
3. Comprender algoritmo de **PARTITION**
4. Comprender el uso de Partition en Median para encontrar el elemento central
5. Comprender el uso de Partition para ordenar secuencias

MEDIANA

Para encontrarla, buscaremos en el arreglo

Si esta ordenada → esta al medio

Si no esta ordenada → buscar

Mayor de la mitad de los menores

Menor de la mitad de los mayores

ALGORITMO

Definición

Dada una secuencia ordenada de n valores $a_0, a_1, \ldots, a_{n-1}$, llamaremos **mediana** al valor M_e dado por

- Si n es impar,

$$M_e = a_{\lfloor n/2 \rfloor}$$

- Si n es par,

$$M_e = \frac{a_{n/2} + a_{n/2-1}}{2}$$

Tiene la **MISMA CTD de ELEM MAYORES Y MENORES**

Estrategia: Tomaremos un **PIVOTE**

- Es ese pivote la mediana?
- Si no, ¿la mediana es **mayor o menor**?
- Visualmente

Dada la siguiente secuencia de datos (de largo impar)



y el **pivote** aleatorio 6

- Elementos menores: 3 5 2 1 4 3
- Elementos mayores: 7 9 7 8

Con esta información, ¿es 6 la mediana? ¿dónde se ubica esta?

No los estoy ordenando, los estoy **ubicando**

Se que el 6 **NO ES LA MEDIANA**

LA mediana va estar donde hayan mas elementos

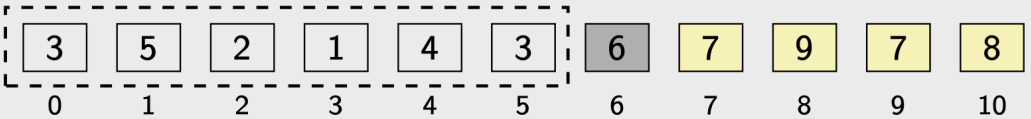
Aplicar recursivamente la misma idea

Reoorganizada ≠ Reordenada!

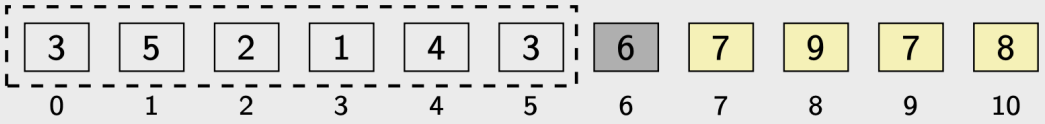


Como la mediana deberia estar en = [11/2] = 5

Como la posición de la mediana debiera ser $\lfloor 11/2 \rfloor = 5$ y el pivote quedó en la posición 6, revisamos el tramo a la izquierda del pivote



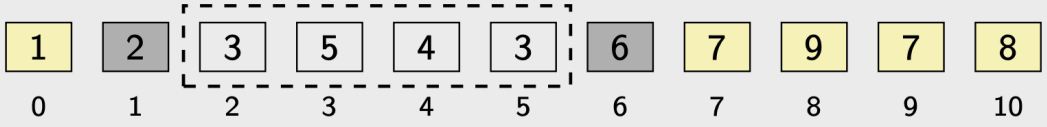
Tomando aleatoriamente el pivote 2



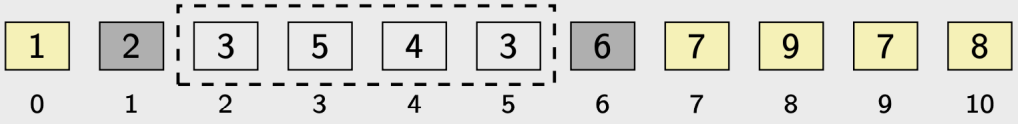
- Elementos menores: 1
- Elementos mayores: 3 5 4 3

Reordenamos el tramo analizado para ver si el pivote es la mediana

Nuevamente, como la posición del pivote es 1, nos concentramos recursivamente en el sub-tramo derecho del último pivote

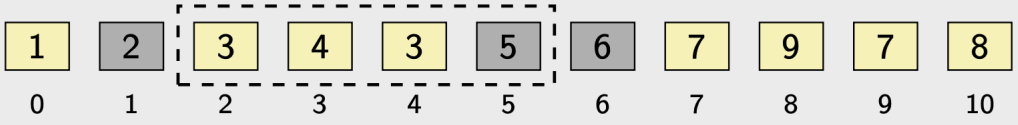


Con el pivote 5

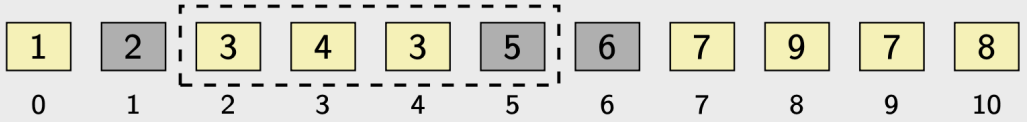


- Elementos menores: 3 4 3
- Elementos mayores: ∅

Reubicamos los elementos para ver la posición del pivote actual



Reubicamos los elementos para ver la posición del pivote actual



Como el pivote está en la posición $\lfloor n/2 \rfloor = \lfloor 11/2 \rfloor = 5$, **es la mediana**

- Todos los pivotes estan quedando de manera ordenada

Esta estrategia de pivotes y particiones → Nos sirve para ordenar estos pivotes

Todos los menores que el estan a la derecha y los mayores a la derecha

Por lo tanto → Recursivamente aplicando eso → **Llegamos al final**

Si cada vez que eligo el pivote, nunca elegi el 5, hasta el ultimo momento → En el peor caso → **ORDENAMOS EL ARREGLO**

UBICACION DEL PIVOTE

Si suponemos que no hay elementos iguales al pivote, elegir un pivote p particiona la secuencia en

- Elementos mayores al pivote, $M = \{a_i \mid a_i > p\}$
- Elementos menores al pivote, $m = \{a_i \mid a_i < p\}$

Luego, si reubicamos los elementos de la secuencia de forma que

- Primero se colocan los elementos de m
- Luego se coloca p
- Finalmente se colocan los elementos de M

es claro que p está en su posición ordenada

ALGORITMO PARTITION

```
// suponiendo que no hay elementos repetidos en la secuencia
input : Secuencia A[0,...,n-1],indices i, f
output:  Índice del pivote aleatorio en la secuencia ordenada

Partition (A,i,f):
    1 p ← pivote aleatorio en A[i , f ]
    2 m, M ← secuencias vacias
    3 Insertar p en M // Obligatorip p sera el menor de M
    4 for x ∈ A[i,f]: // Recorrer el arreglo -> Y elegir dond
        5 if x < p : Insertar x en m
        6 elif x > p : Insertar x en M
    7 A[i,f] ← Concat(m,M)
    8 return i + |m| // Retorna DONDE ESTA PARADO EL PIVOTE
```

Solo estamos **buscando mediana** por ahora → TODAVIA NO ESTAMOS ORDENANDO

- Se ubican los menores, el pivote y luego los mayores
- i: # elems. a la izq. de $A[i]$
- $|m|$: # menores a p en $A[i, f]$
- Por lo tanto, se retorna la posición correcta de p

Su retorno es la cantidad de elementos a la izquierda de p en la secuencia ordenada

Las inserciones al final de la EDD

- $\mathcal{O}(1)$ en arreglos y listas
- Se hace una por elemento x
- Total: $\mathcal{O}(n)$

La concatenación

- $\mathcal{O}(1)$ en listas
- $\mathcal{O}(n)$ en arreglos

Esta version lo hace con memoria adicional

Además, usa memoria adicional $\mathcal{O}(n)$ para mantener las secuencias m, M

MEDIANA

```
input : Secuencia A[0,...,n-1]
output: Elemento en posición central de A

Median (A): i←0
    f←n-1 x←Partition(A,i,f)
    while x no es el centro :
        if x < centro : i ← x + 1
        else: f ← x - 1
        x ← Partition(A, i , f )
    return A[x ]
```

- x: # datos menores al pivote
- `Median` entrega el elemento en la posición central ordenada
- Para n impar, $\texttt{Median}(A)$ corresponde a la mediana
- Para n par, $\texttt{Median}(A)$ es uno de los dos elementos centrales

- Podemos parametrizar Median para encontrar el k-esimo estad ístico de orden

Como Partition funciona en O(n) en encontar al pivote (peor caso) y median llama O(n) a partition → O(n**2) en el peor caso

COMPLEJIDAD MEDIAN

El **mejor caso**

- Escoger la mediana como primer pivote
- Esto significa separar los demás elementos en m y M
- Eso toma $\mathcal{O}(n)$ por pivote $\mathcal{O}(n)$
- Como solo se debe tomar un pivote: Total: $\mathcal{O}(n)$

El **peor caso**

- Que todos los datos menos la mediana sean escogidos como pivote
- Para cada pivote hay que separar menores y mayores $\mathcal{O}(n)$
- Como solo se deben tomar $\mathcal{O}(n)$ pivotes: Total: $\mathcal{O}(n^2)$

SI todos los elementos son pivotes → Esta ordenado

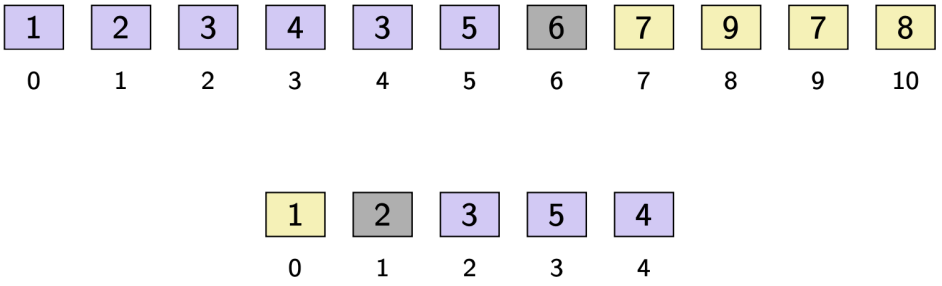
Por lo tanto podemos ordenar con partition

ORDENAR CON PARTITION

se llama **QUICKSORT → ¿porque no PARTITIONSORT? : Nadie sabe**

Quicksort busca que todos los elementos sean **PIVOTES**

Luego de cada ejecución de `Partition`, el pivote queda en su posición ordenada



Además, las dos sub-secuencias están "*del lado correcto*" del pivote

PREGUNTA ¿Cual es la diferencia entre elegir pivote al azar y no? → Es mas factible que se pueda caer en el peor caso → ENVENENAMIENTO

Nuevamente podemos usar la estrategia dividir para conquistar

1. División del problema en dos sub-problemas con `Partition`
2. Aplicar recursivamente `Partition` para cada sub-secuencia
3. No necesitamos combinar nada: la secuencia queda ordenada

Llamaremos **Quicksort** a este algoritmo

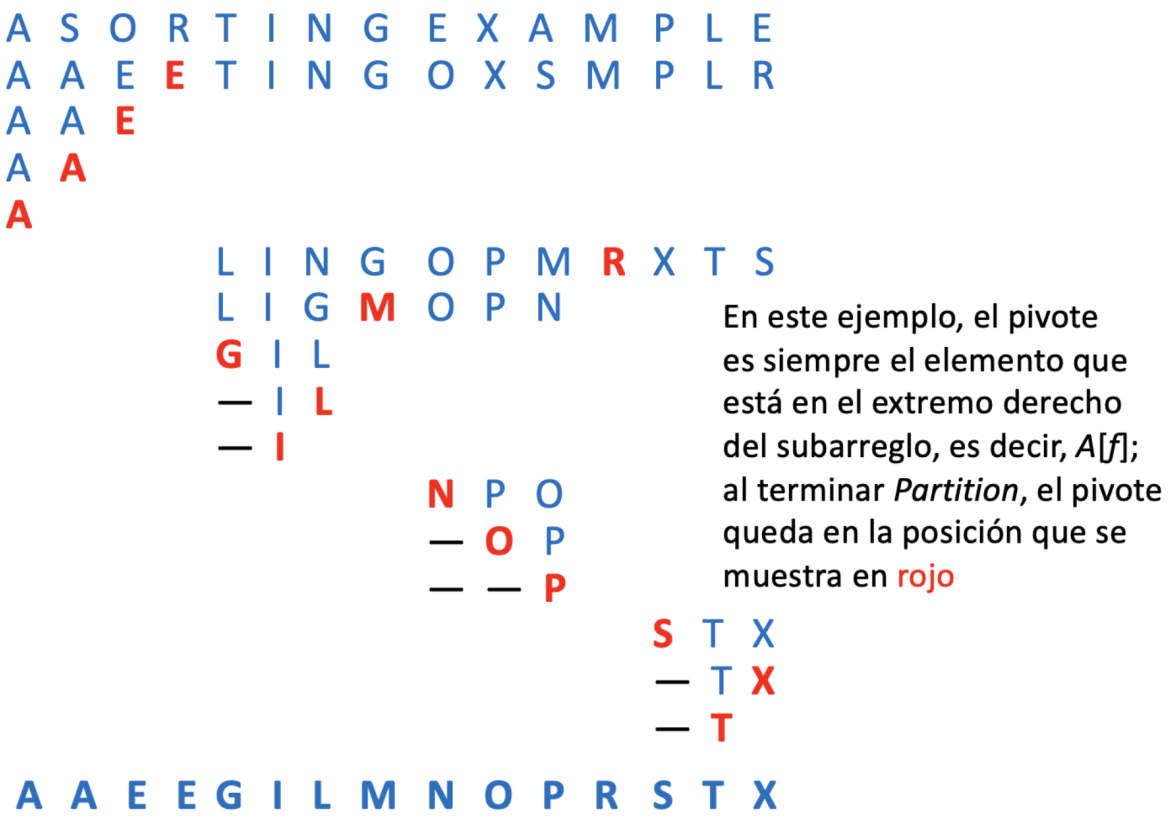
QUICKSORT

```
input : Secuencia A[0,...,n−1], indices i,f
output: ø

QuickSort (A,i,f):
    if i ≤ f :
        p ← Partition(A, i , f )
        Quicksort(A, i , p − 1)
        Quicksort(A, p + 1, f )

// ELEGANTE
// El llamado inicial es Quicksort(A, 0, n − 1)
```

EJEMPLO VISUAL



En este caso no elegimos pivote al azar → Elegiremos el ultimo como pivote

Por un tema de legibilidad y entender el equema, si fuera random seria dificil de seguir

COMPLEJIDAD MEMORIA

→ Depende de **PARTITION**

Versión *in place* de `Partition`

input : Arreglo $A[0, \ldots, n-1]$, índices i, f
output: Índice del pivote aleatorio en la secuencia ordenada

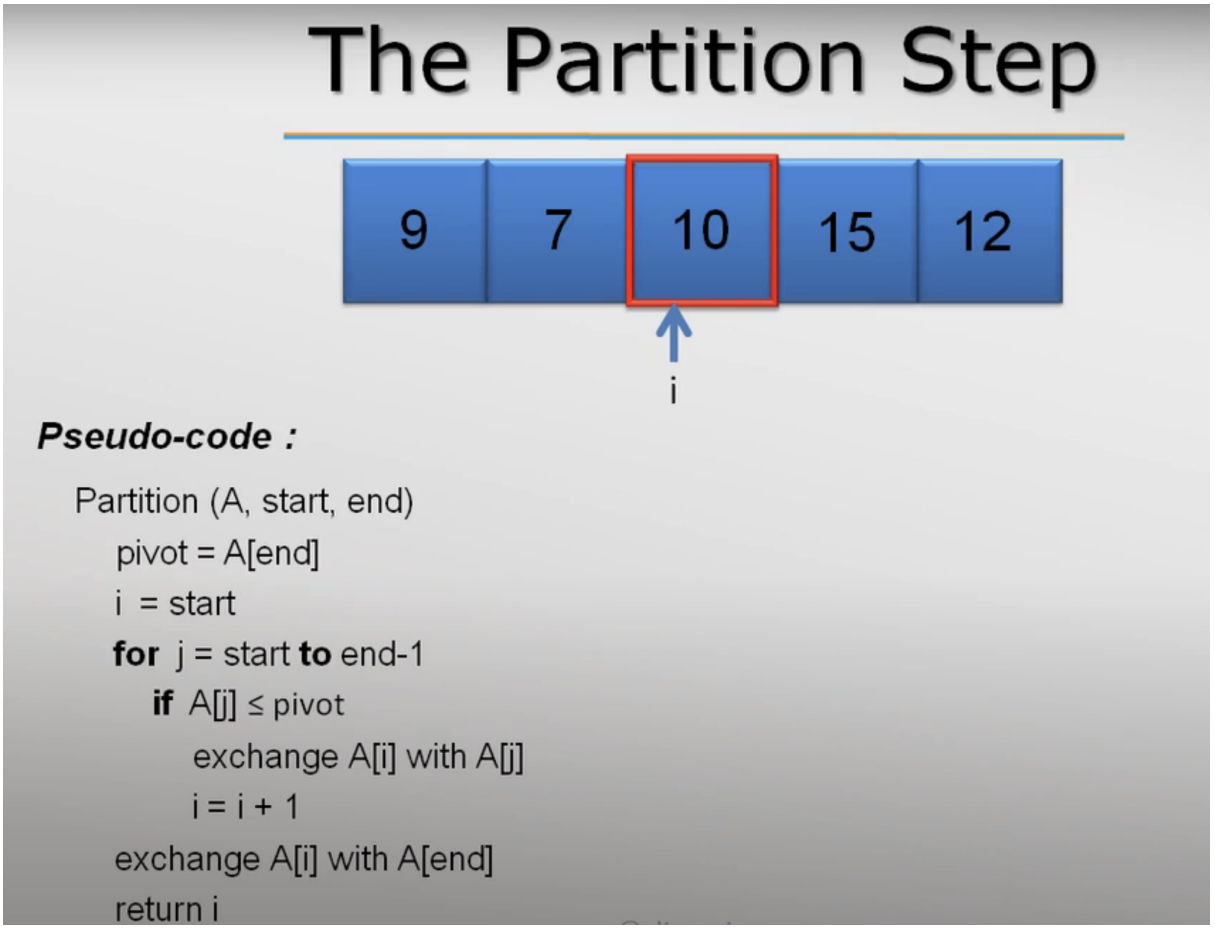
`Partition` (A, i, f)**:**

```
1   x ← índice aleatorio en {i, ..., f}
2   p ← A[x]
3   A[x] ⇄ A[f]
4   j ← i
5   for k = i ... f − 1 :
6       if A[k] < p :
7           A[j] ⇄ A[k]
8           j ← j + 1
9   A[j] ⇄ A[f]
10  return j
```

J: Posicion del pivote

Si el valor de J es mayor que el pivote → No tengo que cambiar

☐ ESTUDIAR ESTO, ESTA ENRREDADO



https://www.youtube.com/watch?v=UqVxcO4oNXA

PROBLEMA

Por eso elegimos pivotes al azar

Si la secuencia viene ordenada → ES EL PEOR CASO

En caso de que el pivote sea el primero siempre → Tendra que devolverse hacia atras siempre

Terrible O(n**2)

CLASE 5

26/08/2024

Tanto isertion como selection son estrategias que enfrentan directamente

No buscan descomponer la complejidad de la tarea, si no que van de frente

Merge Sort y Quicksort, son estrategias que aplican la forma DIVIDIR PARA CONQUISTAR para pensar que es mas facil separar un problema y atacarlo de manera mas simples

Estudiamos la complejidad

Selection O(n**2) → En todos sus casos Promedio peor y mejor

Insertion Promedio y peor O(n**2) y un mejor caso O(n) → Cuando el set viene ordenado

Si yo se que los datos ya fueron ordenados antes, y la probabilidad con la que se desordenan es baja, insertion sort puede ser la mejor opcion

Ej: App movil que ordena lista de valores para presentar al usuario

Se ordena habitualmente, a menos que se inserten datos nuevos, la lista no cambia

Insertion sort podria ser la mejor estrategia para ordenar, sin requerir estrategias recursivas

En el papel, se comportarian mejor las otras

Pero despende del problema, segun el problema nosotros elegimos el algrotimo de como abordarla

MergeSort : NlLogN. → Va dividiendo y ordenando → Usa costo adicional

Quicksort lo veremos hoy

- Vimos el peor caso: ORDENADO y PIVOTE EL PRIMERO
  - Esto hace que siempre queden toda la particion por arriba
  - Tendra que hacer N divisiones N veces
  - Por lo tanto: **PEOR CASO O(N**2)**
  - Igual de malo que selection sort
  - El caer en el peor caso → Depende de la eleccion del pivote (nuestra implementacion)
  - Por eso se hace al azar → Caemos menos problablemente en el peor caso

RESUMEN

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

QUICKSORT (A grandes rasgos)

En lugar de dividir para conquistar → Quicksort particiona para ordenar

Particiona, y haz quicksort de ambos lados

Particionar → Encuentra menores y mayores del pivote, y sabemos que el pivote esta ordenado

A sus lados tendremos que elegir pivotes tambien, y ordenamos cada particion

Y asi recursivamente

Partition cuando llegue a un solo dato, devuelve el pivote,

```
input : Secuencia A[0,...,n−1], indices i,f
output: ø

QuickSort (A,i,f):
    if i ≤ f :
        p ← Partition(A, i , f )
        Quicksort(A, i , p − 1)
        Quicksort(A, p + 1, f )
```

PARTITION (In place)

```
input : Arreglo A[0,...,n−1], índices i, f
output:  Índice del pivote aleatorio en la secuencia ordenada

Partition (A, i , f ):
1   x ←ındice aleatorio en {i,...,f}
2 p ← A[x]
3 A[x] ⇄ A[f]
4 j←i
5 for k = i...f−1:
6    if A[k] < p :  // Si llego a uno menor que el pivote
7        A[j] ⇄ A[k] // INTERCAMBIO
8        j ← j + 1
9  A[j] ⇄ A[f] // Intercambia la posicion de J, con la de Piv
10 return j

// J: Es la posicion del menor de los mayores -> PIVOTE
// Luego J representa la POSICION del PIVOTE

// Los mayores estan a la derecha
// Los menores estan a la izquierda
```

- [ ] Jugar con este algoritmo, probar y cambiar el nombre

https://prod-files-secure.s3.us-west-2.amazonaws.com/a2fa890e-9d59-46fb-b314-c0d33e3d7a3c/f63ec366-bceb-43ec-bff3-9f4b4c00019a/Grabacion_de_pantalla_2024-08-26_a_la(s)_12.42.47_p._m..mov

¿Cuando usar quicksort? → Excelente algoritmo de proposito general

Salvo con casos especificos: - Como estar ordenados

Tiene buen desempeño, no requiere memoria adicional

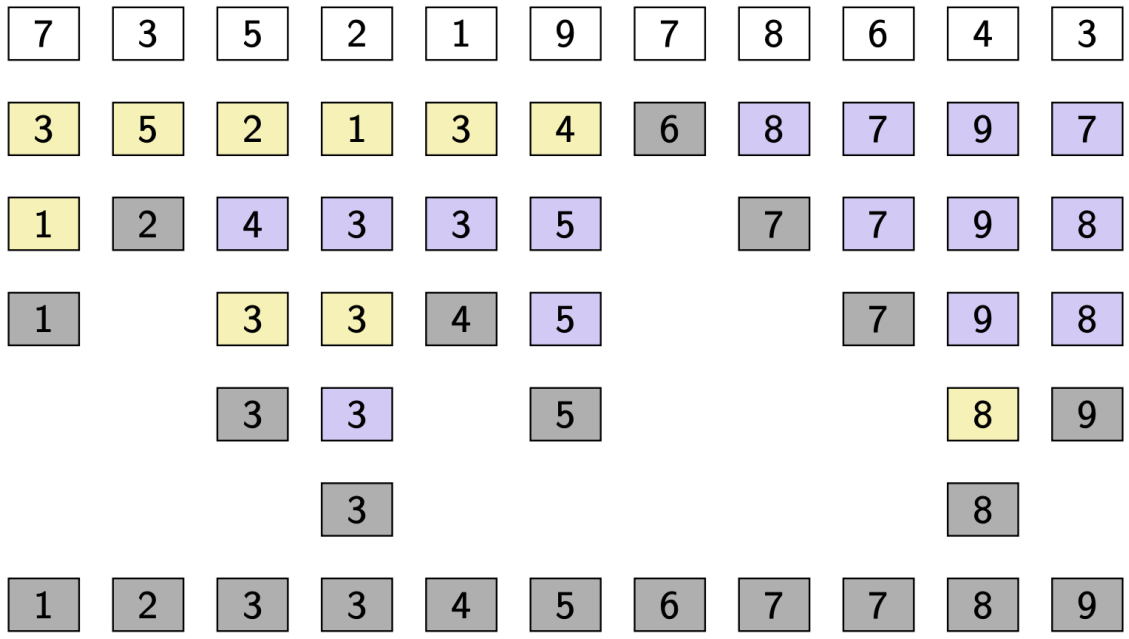
Pero tiene el costo de la recursion

Si se requiere para muchos datos, eleva el costo de recursion

Si no tenemos info sobre los datos que vamos a ordenar, es un excelente indicio para usar Quicksort

---

QUICKSORT EJEMPLO



Hoy veremos complejidad de Partition y de Quicksort + Correctitud

EN CUANTO A SSOO

No conviene paralelizar Insertion y Selection porque solo van haciendo

Para Merge y Quick si podria, pero Merge tiene cuello de botella, solo un procesador puede hacer un merge de 2 listas

Merge paraleliza mientras baja, pero consolida hacia arriba

En cambio quick, puedo pasarle partition a cada procesador

CORRECTITUD PARTITION

1. Termina? → SI tiene un for, que termina, las posibles loops será para una cantidad finita de datos → Trivialmente Termina
2. Hace lo correcto? → Demostraremos que el Pivote queda en su posición
    - Eso significa que a un lado estan los mayores y al otro los menores

CB: 1 Elemento → Escogemos el primer dato, y lo ponemos al final

K y j al inicio y comparo la posicion de k, con el valor del pivote

A[k] con el valor del pivote

Tenemos 2 posibles situaciones:

Que sea menor que el pivote

Que sea mayor o igual que el pivote

1. Si es menor, entra en el loop, y hacemos cambiazo
2. Si es mayor o igual que el pivote → No hago nada, dejo j en la posicion y aumento k en una posicion,
    - La posicion de J, es el mayor de los menores → No

En ambos casos la posicion de J, es la posicion Ordenada del Pivote

HI: En N posiciones tenemos una posicion de J y K que estan avanzando en el arreglo

```
// ejemplo

5 j 8 6 5 4
j.  k

Si es menor que el pivote, lo voy a intercambiar
Si es mayor, voy a aumentar k en 1
```

TI: Vamos a aumentar el k, y tendremos los mismos dos casos

Si es menor,

Si es mayor, no hace nada y aumenta hacia adelante

Demostración (posición del pivote)

Probaremos que el pivote se encuentra en su posición ordenada en la secuencia. Para esto, dado que en la línea final se pone el pivote en la posición j, demostraremos la siguiente propiedad para un pivote p dado

$$\begin{aligned}\mathbf{P(n)} \quad := \quad & \text{Luego de la } n\text{-ésima iteración, } A[i \ldots j-1] \\ & \text{contiene solo elementos menores a } p \text{ y } A[j \ldots k-1] \\ & \text{solo elementos mayores o iguales a } p\end{aligned}$$

I el inicio y j-1 (una posicion antes de j) → Menores al pivote

J .... k-1 → Mayores al privote

J es el **menor de los mayores**

Caso Base

1. **Caso base.** $\mathbf{P(1)}$ es el estado de $A[i \dots k-1]$ luego de la primera iteración. Tenemos dos casos
   - Si $A[i] < p$, como $i = j$, se mantiene $A[i]$ en su posición y j, k aumentan en 1. Luego, $A[i \dots j-1]$ tiene solo el elemento $A[i]$, el cual es menor a p y $A[j \dots k-1]$ no tiene elementos pues $j = k$.
   - Si $A[i] \geq p$, no se aumenta j y sí k, por lo tanto $A[i \dots j-1]$ considera 0 elementos y $A[j \dots k-1]$ tiene un elemento mayor o igual a p.

Hipotesis Inductiva

2. **Hipótesis inductiva (H.I.)** Suponemos se cumple luego de la n-ésima iteración, $A[i \dots j-1]$ contiene solo elementos menores a p y $A[j \dots k-1]$ solo mayores o iguales.

Tesis Inductiva

Al término de la $(n+1)$-ésima iteración, nuevamente pueden haber ocurrido dos casos

- Si $A[k] < p$, este elemento se pone en la posición j antes de aumentarla en 1. Por **H.I.** todos los anteriores a él también son menores a p. Además, como los datos de $A[j \dots k-1]$ eran mayores o iguales a p antes de aumentar los índices, con el intercambio se mantiene la condición.
- Si $A[k] \geq p$, no se aumenta j y no se hacen intercambios. Por **H.I.** los elementos en $A[i \dots j-1]$ son menores a p y como $A[k]$ al igual que los elementos de $A[j \dots k-1]$ son mayores o iguales, luego de aumentar k el rango $A[j \dots k-1]$ sigue cumpliendo la propiedad. □

Otra estrategia de Quicksort → Dividir en 3 partes → Menores iguales y Mayores

Correctitud de Quicksort

- La llamada cuando la cantidad de elementos a llamar es 0 se detiene, so las llamadas recursivas van reduciendo el tamaño hasta llegar a 0, se que termina
- Porque cada vez voy quitando un elemento, llaman tanto a la izquierda y a la derecha, pero sin el pivote
- En el peor caso, tendremos n-1 llamadas,
- Y ademas Partition es Correcto

Termina

Demostración (finitud)

Para demostrar que `Quicksort` termina, primero usamos el hecho de que `Partition` termina y por ello la línea 2 toma una cantidad finita de pasos.

Ahora, notamos que cada llamado recursivo de `Quicksort` se hace a una instancia de tamaño estrictamente menor al original.

En efecto, la llamada `Quicksort`(A, i, f) establece el *tamaño* de la instancia analizada como

$$d = f - i$$

Tenemos dos casos según el valor de d

- Si $d < 0$, `Quicksort` termina sin hacer más llamados recursivos

Demostración (finitud)

El otro caso es

- Si $d \geq 0$, se hacen dos llamados recursivos de tamaños respectivos

$$d_1 = p - 1 - i < d \qquad d_2 = f - p - 1 < d \quad \text{para } i \leq p \leq f$$

Es decir, con cada llamado resursivo d disminuye al menos en 1.

Con ambos casos, tenemos que en cada llamado se reduce al menos en 1 el tamaño de la instancia y el algoritmo termina al llegar a tamaño 0.

Como el llamado inicial es `Quicksort`$(A, 0, n-1)$, la profundidad máxima de la recursión es $n-1$ y por lo tanto, el algoritmo termina. □

Hace lo que se pide (ORDENACION)

Demostración (ordenación)

Debido a que probamos la correctitud de `Partition` y que `Quicksort` divide en sub-problemas con `Partition`, esta demostración será más sencilla.

Sabemos que `Partition` ubica correctamente el pivote al terminar. Basta demostrar que todo elemento de $A[0 \ldots, n-1]$ es elegido como pivote en algún llamado de `Quicksort`.

El llamado `Quicksort`(A, i, f) verifica si hay elementos en el rango $i \ldots f$

- Si $i > f$, no hay elementos que puedan ser escogidos como pivote
- Si $i \leq f$ el rango es de $f - i > 0$ elementos, se escoge un pivote y se realizan dos llamados cuyos rangos suman $f - i - 1$ elementos

Dado que en cada llamado no vacío se escoge un pivote, los llamados siempre reducen su tamaño y el algoritmo solo deja de hacer recursión cuando el llamado es vacío, todo elemento es escogido pivote en algún llamado. Como `Partition` es correcto, todo elemento queda ordenado. □

Demostrar que llama a todos los pivotes

Si todos los datos en algun momento son pivote → Esta ordenado

Si todo elemento llamado desde la entrada es pivote → Entonces Quicksort ordena

COMPLEJIDAD

Complejidad de Partition → O(n)

Quicksort

Mejor caso de llamadas recursivas:

- Si pienso en mergesort, aplica divide para conquistar
- Si se divide exactamente en la mitad
- Tenemos log(n) divisiones

En quicksort, si el pivote en cada division queda exactamente al medio → Mejor Caso

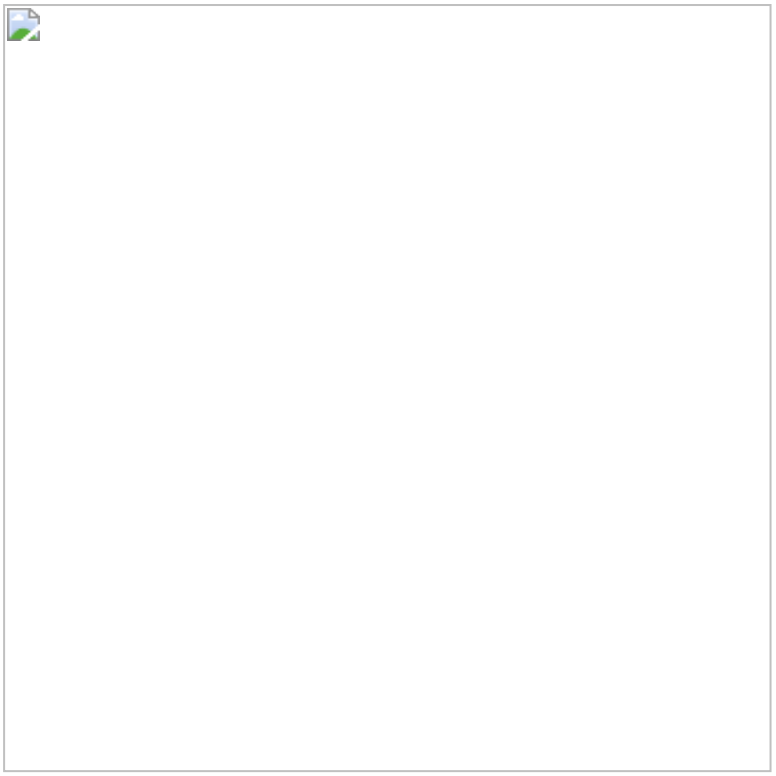
Misma cantidad de llamadas recursivas por la izquierda y por la derecha

Luego NLog(n)

En el peor caso N divisiones hasta llegar al final

MEjor caso: Divisiones justo al centro (triangulo isoceles hacia abajo)

Peor caso: Dividir siempre en los extremos (trianguno rectangulo hacia abajo)



Mejor caso

- `Partition` genera sub-secuencias del *mismo* tamaño
- Ecuación de recurrencia $T(n) = n + 2T(n/2)$ (como en `MergeSort`)
- Complejidad $\mathcal{O}(n \log(n))$

Peor caso

- `Partition` genera sub-secuencias de tamaño 0 y $n - 1$
- Ecuación de recurrencia $T(n) = n + T(n-1)$
- Complejidad resultante: $\mathcal{O}(n^2)$

El problema se reduce al azar (elección del pivote)

CASO PROMEDIO QUICKSORT

Justificacion mediante La integral > Sumatoria → Para sacarse un Logaritmo ;)

```
Partition (A, i, f):
1    x ← índice aleatorio en
       {i, ..., f};    p ← A[x]
2    A[x] ⇄ A[f]
3    j ← i
4    for k = i ... f − 1 :
5        if A[k] < p :
6            A[j] ⇄ A[k]
7            j ← j + 1
8    A[j] ⇄ A[f]
9    return j
```

- En la primera llamada se hacen $n-1$ comparaciones, pues $i=0$ y $f=n-1$
- Supongamos que `Partition` termina retornando q, i.e. deja al pivote en $A[q]$
- Las llamadas recursivas de `Quicksort` se harán sobre los
  - q elementos a la izq
  - $n-q-1$ elementos a la der
- Los llamados recursivos aportan
$$C(q) + C(n-q-1)$$

Con lo anterior, las comparaciones ecuando el pivote queda en $A[q]$ son

$$n-1+C(q)+C(n-q-1)$$

Para ver el caso promedio, ponderamos para todos los q posibles obteniendo

$$C(n) \;=\; n-1+\frac{1}{n}\sum_{q=0}^{n-1}(C(q)+C(n-q-1))$$

Además, notando que

$$\sum_{q=0}^{n-1} C(n-q-1) \;=\; C(n-1)+C(n-2)+\cdots+C(0) \;=\; \sum_{q=0}^{n-1} C(q)$$

obtenemos la regla simplificada

$$C(n) \;=\; n-1+\frac{2}{n}\sum_{q=0}^{n-1} C(q)$$

$$\begin{aligned} &\leq \quad \frac{C(1)}{2}+\sum_{k=2}^{n}\frac{2}{k+1} \\ &\leq \quad \sum_{k=1}^{n}\frac{2}{k} \;\leq\; \int_1^n \frac{2}{x}dx \;=\; 2\log(n) \end{aligned}$$

Concluimos que una cota superior para el número de comparaciones es

$$C(n) \leq 2(n+1)\log(n)$$

`Quicksort` es $\mathcal{O}(n\log(n))$ en el caso promedio

QUICKSORT en Promedio → **O(N LOG (N))**

MEJORAS PARA QUICKOSRT

1. No Usarlo → Para secuencias pequeñas, mejor usar insertion sort
    a. Quicksort es recursivo, usa recursos
2. Usar mediana para **3 elementos(escoger):**
    a. A[k1], A[k2], A[k3]
    b. En O(1) encontramos mediana entre ellos
3. Particionar subsecuencia en 3, MENORES IGUALES Y MAYORES
    a. Mejora para caso de repetidos

RESUMEN

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	?	?	?	?

Recordar que en el peor caso, es cuando eligo el pivote el incial, y el arreglo viene ordenado, por lo que hara n-1 partitions, y recorrera n veces → N**2

CLASE 6

28/08/24

Colas de prioridades

Fila de elementos con cierta logica para ser analizados

Operaciones basicas : Insertar y sacar

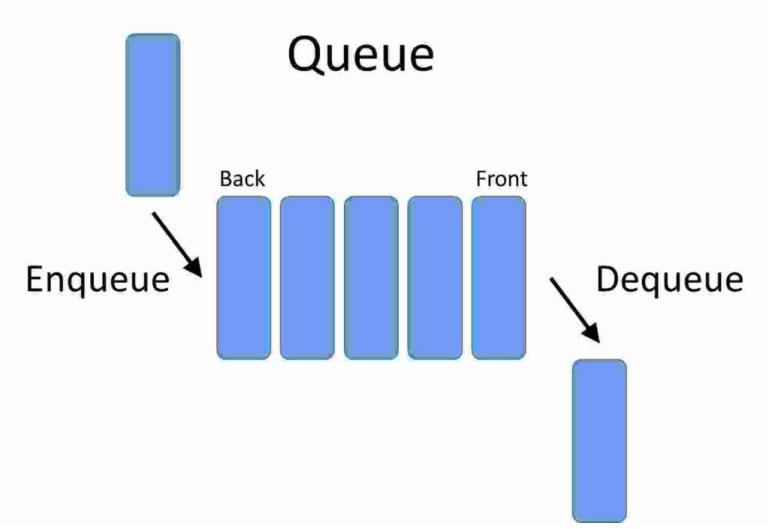
Saco al primero al atender,

Inserto al final de la cola

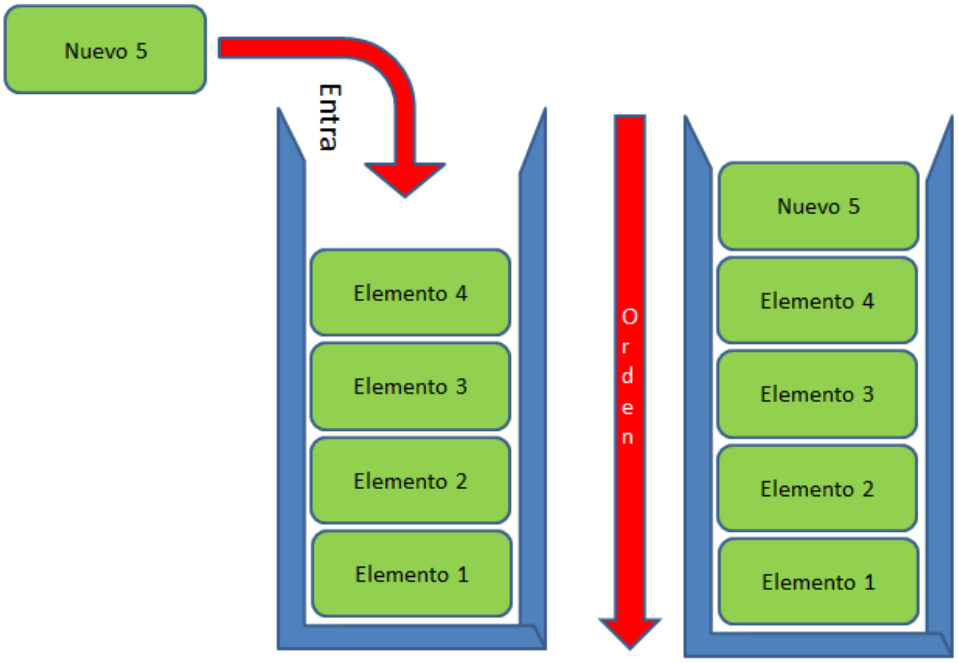
Ej: La pila o Stack

Aqui nos interesa el ultimo que llego

COLA



STACK



FIFO

First i first out → Cola de prioridad segun orden de llegada

- Primer elemento es el m ás prioritario: lleva m ás tiempo en la cola
- Ultimo elemento es el menos prioritario: lleva menos tiempo en la cola

Las operaciones en las colas son

- Insercion: se inserta al final de la cola.
  - Arreglo: O(1) en general (salvo que se llene)
  - Lista: O(1) con puntero al u ltimo elemento

Extraccion: se elimina la cabeza de la cola.

- Arreglo: O(1) si no reubicamos
- Lista: O(1)

Ejemplo prueba: Si quiero aterrizar primero al que tiene menos combustible que en vez de el primero que va llegando → Heaps nos ayuda a modelar distintas prioridades

LIFO

Last in first out → Stack de prioridad con orden inverso

Operaciones son las mismas

Se elimina la cabeza de la fila

COLA DE PRIORIDAD

De maxima o minima prioridad

Es un EDD que permite

- insertar un dato con prioridad alta,
- extraer y
- eventualmente cambiar prioridad de un dato

El orden de llegada no es la prioridad de esta cola

No se en que orden de prioridades vienen mis elementos para meterlos a la cola

Puede llegar cualqueir elemento

Como resuelvo ?

→ Podria recorrer (O(n)) para buscar el menor

→ Tenerla ordenada (O(n)) con insertion sort, y buscae en O(1) para buscar el menor

Ambas estrategias son O(n)

La insercion y extraccion cambia depende de lo que implementemos

Para el arreglo sin orden

- Inserción al final $\mathcal{O}(1)$
- Extracción buscando el máximo valor $\mathcal{O}(n)$

Para el arreglo ordenado por valor

- Inserción en la *posición correcta* $\mathcal{O}(n)$
- Extracción del último elemento $\mathcal{O}(1)$

Si descompongo el problema podria ser mejor (dividir y conquistar)

Quizas ordenarlo por completo no, pero parcialmente podria ser util

HEAP

Estructura gererquica que tiene un orden parcial

Los elementos que estan mas arriba del heap tienen mas prioridad que los que estan mas abajo

Cuando necesite encontrar a uno, estara en la posicion mas arriba

Y para insertar, el costo sera logaritmico

Como esta medio ordenado → Es mas rapido que Lineal O(n) (ORDEN EFICIENTE)

Estructura Recursiva: **ARBOL**

- Descompondremos la forma en que almacenamos los datos

ARBOL BINARIO

1era Estructura Recursiva que veremos

No es para buscar (No es arbol binaro de busqueda)

El heap sirve para saber cual es el mayor o menor, pero no para buscar entremedio

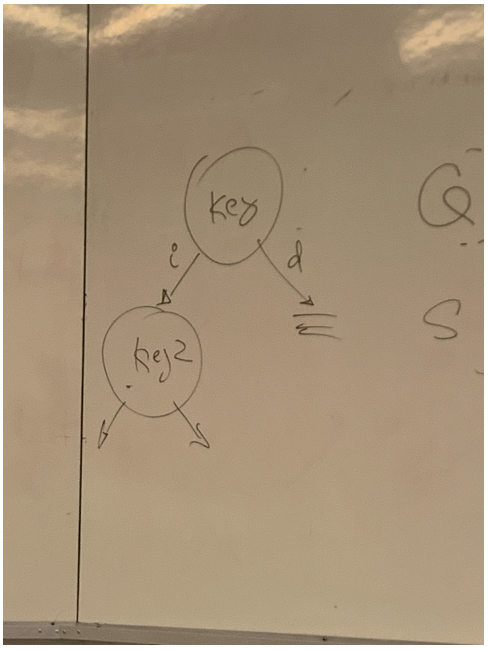
Definición

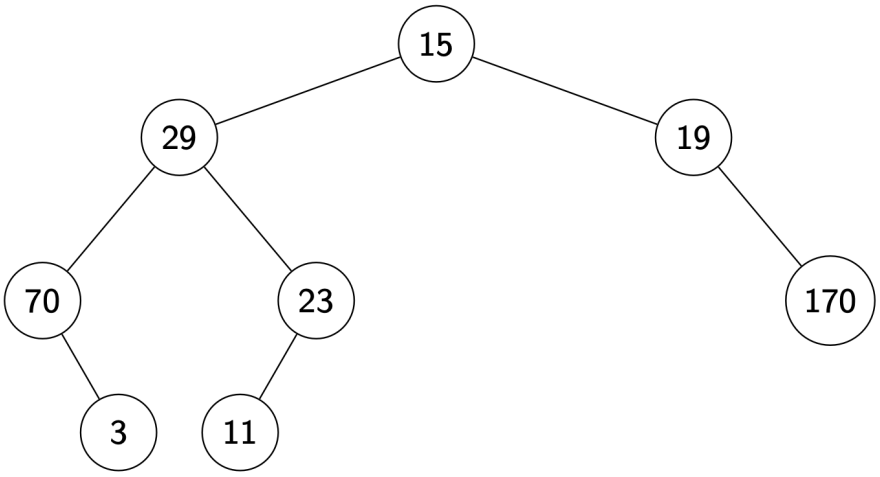
Un **árbol binario (AB)** es una estructura de datos que almacena **llaves** asociándolas mediante punteros según una estrategia recursiva

1. Un AB tiene un **nodo** que contiene una llave
2. El nodo puede tener hasta dos AB's asociados mediante punteros
   - Hijo izquierdo
   - Hijo derecho

El árbol binario A tiene hijos $A.left$ y $A.right$, y llave $A.key$

Contiene un hijo izquierdo y un hijo derecho (**NODOS**)





Observemos que no hay un orden a priori entre sus elementos

Voy a tener Log(n) niveles

Me gustaria que el elemento mas prioritario este arriba

Cuando inserto? → La relacion me indique la prioridad alta esta arriba

La relacion entre nodos y hijos es **JERARQUICA**

MAX HEAP BINARIO

Definición

Un **Max heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son Max heaps binarios
- $H.key > H.left.key$
- $H.key > H.right.key$

A estas condiciones les llamamos **propiedad de heap**

Todo hijo tiene llaves menores que el padre...
pero entre hermanos no hay ninguna restricción

Si el padre tiene 12, los hijos tienen que ser menores que 12 en un **MAX HEAP**

Si fuera un **Min HEAP,** tendrían que ser mayores que 12

Esta ordenado por capas , y el mayor esta arriba, y que el mayor esta arriba, y que el mayor esta arriba

Por lo tanto el mayor de toda la estructura esta arriba

Si quiero que recorrer el arbol para buscar un valor → Tendre que recorrerlo entero

Pero si sabre en O(1) → Es que el mayor valor esta **ARRIBA**

- Ahora lo vemos como un grafo, pero lo veremos como arreglos , y el acceso al mayor es O(1) → **MUY EFICIENTE**

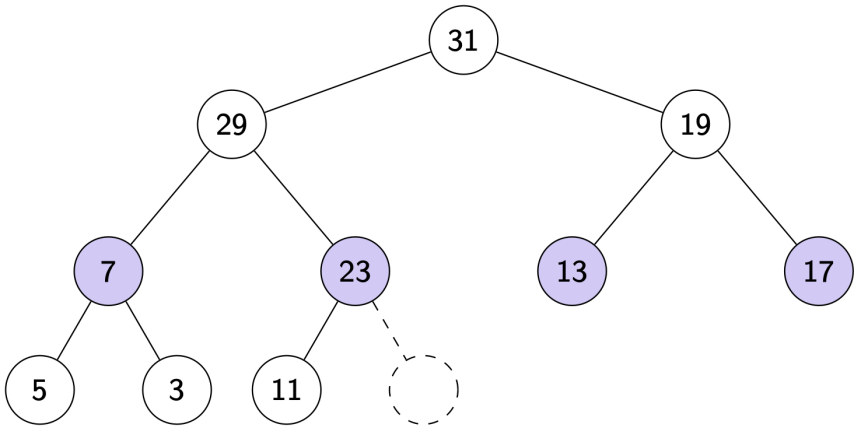
Vamos a ir llenando esta estructura por niveles

Ademas de la jerarquia, garantizo que esta blanceado

Para que no quede tan desnivelado

Nivelado: Cuando llegue un siguiente dato, lo voy colocando completando niveles

Cuando inserto un dato, me preocupo que sea menor que el de arriba, si no, lo intercambio, y asi **recursivamente hasta la RAIZ**



Los nodos de un mismo nivel no satisfacen un orden específico

GRACIA DE TENER ORDENADO

Para llegar a una hoja del elemento, tengo que hacer **LOG (n)**

Para llegar al nodo mas lejano, va a ser o(n)

Cuantas comparaciones se hacen → Log(n)

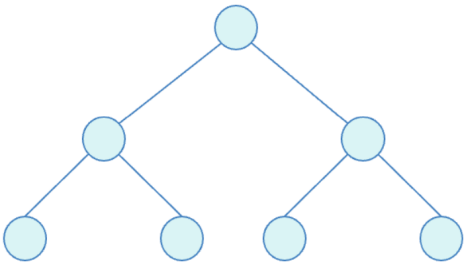
Ya no hago como insertion sort N comparaciones → si no que LogN

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**

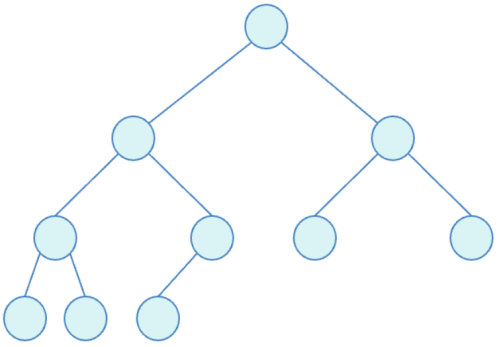
- Ojo: esto no significa que si hay h niveles, haya $n = 2^h - 1$ nodos
- Lo que interesa es que antes de agregar un nivel, el último disponible se complete

Podremos hacer esto gracias a la propiedad de heap

Almacenaremos los heaps completándolos **por nivel**, i.e. como **árboles binarios casi-llenos**



árbol binario lleno, cuando el número n de nodos cumple $n = 2^d - 1$



árbol binario lleno, cuando el número n de nodos cumple $2^d \leq n < 2^{d+1}$

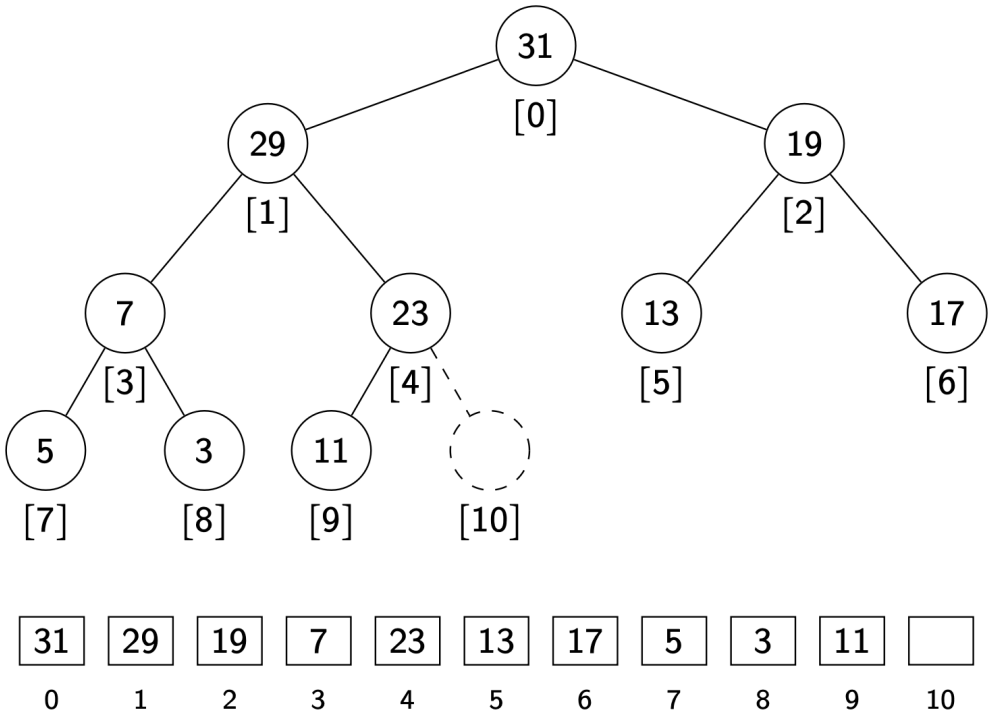
LOGICA PARA INDEXAR PUNTEROS

La representación permite recorrer descendientes sin punteros

- El elemento $H[k]$ es padre de $H[2k+1]$ y $H[2k+2]$
- El padre del elemento $H[k]$ es $H[\lfloor (k-1)/2 \rfloor]$

Además, permite ubicar los elementos del nivel h sin punteros

- El primer elemento del nivel h es $A[2^h - 1]$
- Los 2^h elementos consecutivos corresponden al nivel h



2∗1 + 1 = 3 → Izquierdo

2∗1 + 2 = 4 → Derecho

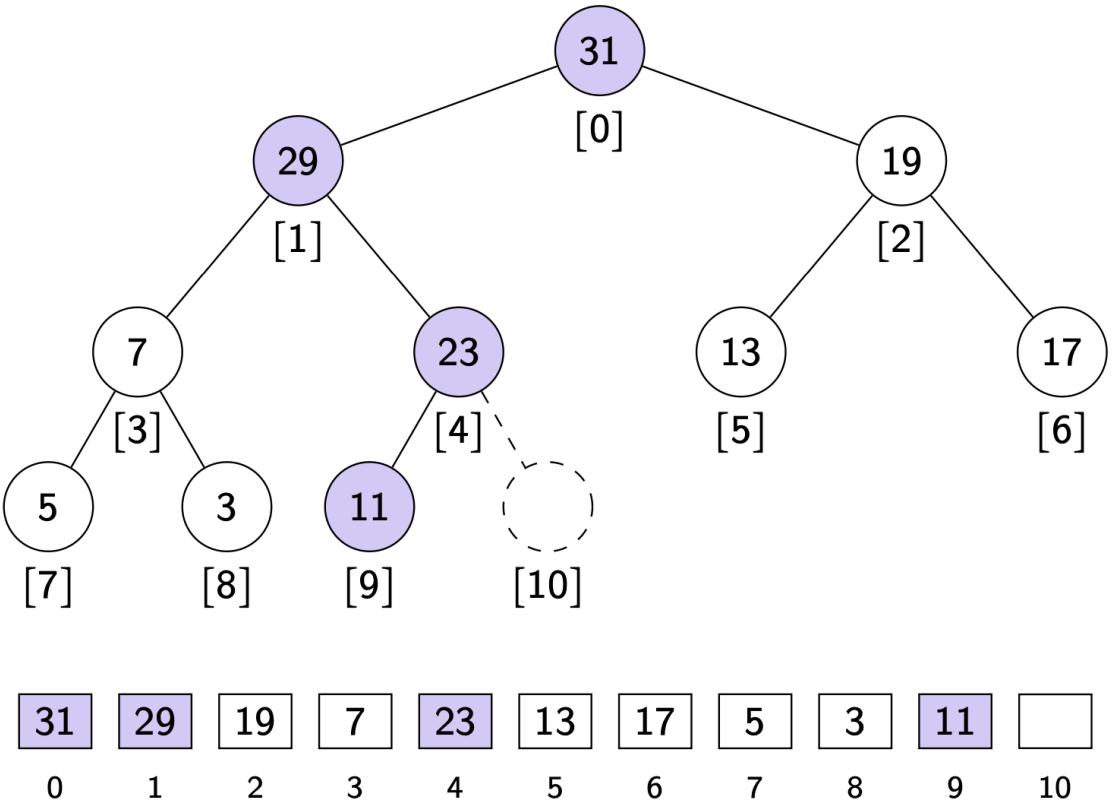
2∗2 +1 = 5 → Izquierdo

2∗2 + 2 = 6 → Derecho

2∗3 +1 = 7 → Izquierdo

2∗3 + 2 = 8 → Derecho

etc



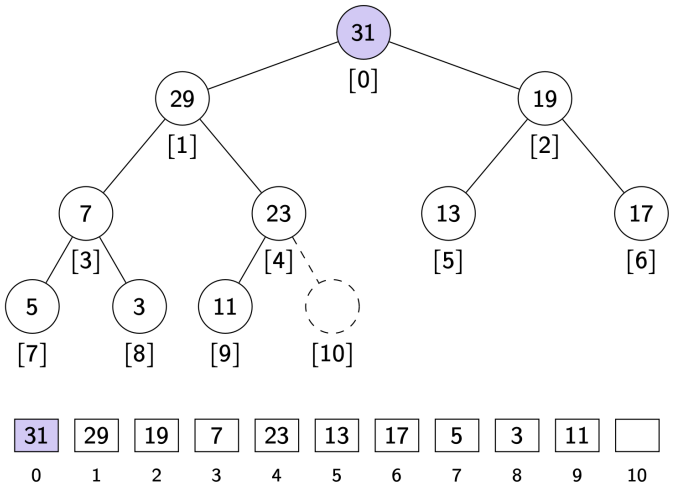
Con aritmetrica → Puedo encontrar las ramas del arbol

Para insertar y extraer, tengo que usar estas operaciones manteniendo el orden

EJEMPLO DE INSERCION

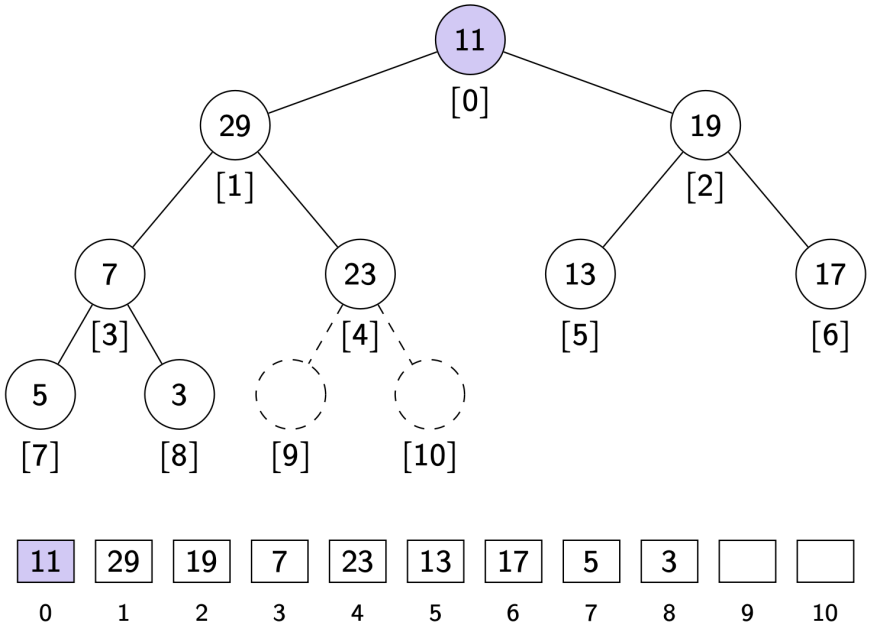
1. Sacar al mas prioritario

Al extraer, sacamos el elemento más prioritario

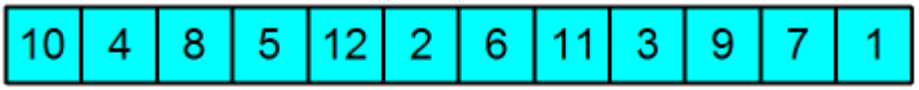


Al sacarlo, el arbol ya no esta casi lleno, moveremos ahora el ultimo elemento → 11

Ahora el árbol **está casi-lleno**, pero no se cumple la propiedad de heap



VISUALIZACION



https://www.youtube.com/watch?v=MkHuwIrWZmI&t=1s

BALANCE HEAPS → Extraccion

El algoritmo para balancear se ve asi mas o menos (ir bajando)

```
input : heap representado como arreglo H [0 . . . n − 1]
output: elemento mas prioritario

Extract(H ):
    i ← ultima celda no vacıa de H best ← H[0]
    H[0] ← H[i]
    H[i] ← ø
    SiftDown(H , 0) // Si el nodo tiene hijos, compara con el
    return best
```

Y se llama recursivamente a **SHIFTFOWN → Solo comparo (mirar) → No intercambiar**

Solo en ese caso intercambio

```
input : heap representado como arreglo H [0 . . . n − 1], índ

SiftDown(H , i ):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇆ H[i]
            SiftDown(H , j)
```

BALANCE HEAPS → Inserción

Sigue la misma idea de la extracción

```
input : heap como arreglo H[0 . . . n − 1], elemento e

Insert(H ):
    i ← primera celda vacıa de H
    H[i] ← e
    SiftUp(H , i )
```

```
input : heap representado como arreglo H [0 . . . n − 1], ind

SiftUp(H , i ):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇆ H[i]
            SiftUp(H, j)
```

Para un arreglo de N elementos → O(Log(n))

La logica de HeapSort es muy sencilla

El primer elemento esta ordenado

Si lo saco del Heap, y lo pongo en otro lado,

Tengo que reconsruir el Heap

Log (n)

Lo pongo en otro lado

N reducciones de tamaño de Heap

Reconstrucciones de Log (n)

Finalmente (N Log N)

Problema: Como hacemos para tener el HEAP, si ya lo tengo es facil ordenarlo

La pregunta es como tener eficientemente un Heap en primer lugar

CLASE 7

2/09/2024

RECORDANDO HEAPS

Sirven para por ejemplo:

1. Implementacion de Sistemas operativos
2. Ordenes que no dependen de la llegada, si no que algun atributo de los datos es lo que da la prioridad
3. Por ejemplo el siguiente proceso, que entra al Sistema Operativo

4. El SO maneja como HEAP, el proceso siguiente, porque es muy eficiente, cual es el siguiente es el unico que importa

En la prueba → Pueden salir problemas de heap

HEAPSORT

1era problema → Construir el Heap, para tener el primer elemento mas prioritario en la primera posicion

Herramientas: Inserción → LogN y tenemos que insertar n elementos → NLogN

CONSTRUCCION DE UN HEAP

Estrategia 1: Iterar para cada elemento de A, insertando sobre un heap originalmente vacio

Estrategia 2: SiftDown para ciertos elementos de A → De la mitad para arriba

ALGORTIMO

input : arreglo $A[0 \ldots n-1]$

```
BuildHeap(A):
    for i = ⌊n/2⌋ − 1 ... 0 :        ▷ loop decreciente
        SiftDown(A, i)
```

(Desde la mitad para abajo)

VIDEO → BuildHeap

https://prod-files-secure.s3.us-west-2.amazonaws.com/a2fa890e-9d59-46fb-b314-c0d33e3d7a3c/7f6d3eb5-eb87-46de-b673-9b968332b1f6/Grabacion_de_pantalla_2024-09-02_a_la(s)_12.40.07_p._m..mov

Observacion: los elementos de A en los cuales no se llama directamente SiftDown son hojas del ultimo nivel del arbol

COMPLEJIDAD

Asistotica directa es O(n Log n)

Se puede demostrar que una mejor cota es O(n)

Pregunta Tonta: Como construir un Min Heap desde un Max Heap

Resupuesta: Simplemente BuildHEap y luego MinHeap → Porque minheap ordena en esa forma cualquier arreglo de entrada que entre

Tips: Desconfiar de una pregunta que es parte de una pregunta mas grande

Las respuesta pueden ser mas directas

El arreglo A viene desordenado → Implicitamente vamos a ordenar de menor a mayor

HEAPSORT

input : arreglo $A[0 \dots n-1]$

```
HeapSort(A):
    BuildHeap(A)
    for i = n − 1 ... 1 :          ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size − 1
        ShiftDown(A, 0)
```

Heap size funciona exclusivamente para que sepa donde pararse

Esta implicito que ordena de menor a mayor → MaxHeap

COMPLEJIDAD

Respecto a su complejidad

- `BuildHeap` $\mathcal{O}(n)$
- `SiftDown` se repite $\mathcal{O}(n)$ veces $\mathcal{O}(n\log(n))$
- Total $\mathcal{O}(n + n\log(n)) = \mathcal{O}(n\log(n))$

`HeapSort` ordena en tiempo $\mathcal{O}(n\log(n))$

Es muy parecido a Selection Sort

Tiene un lado del arreglo que esta ordenado, y un lado que esta desordenado, en el lado que esta desordenado, tengo que encontrarlo

En este caso, como tengo un Heap, la busqueda es log N, por lo tanto :

Heap Sort: Es un SelectionSort → Pero con la estructura de datos correcta!!!

RESUMEN

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$

HEAPSORT:

1. **No es estable**

   Como el orden es parcial → No podemos garantizar

2. No es Envenenable

Cuando podemos usar Heapsort

Es mas lento que Quicksort → Salvo en el peor caso

Quicksort hace menos intercambios que HeapSort (general)

HeapSort se usa como mecanismo de prevención

Ahora si

CLASE 7

ORDENACION LINEAL → Categorizar!

- Cuando no tenemos info sobre la data, lo mejor que podemos hacer, es ordenar en NLogN
- Ordenar en O(n) → Pero para ordenar en O(n), tenemos que **TENER INFORMACION** Sobre los **DATOS** que vamos a **ORDENAR**
- En los supuestos, tenemos que cumplir con ciertos atributos

Lo primero: Sabemos que lo que vamos a ordenar, tiene una cantidad de datos distintos

Por ej: Voy a ordenar numeros.

Tengo entre 0 y k valores distintos

Ej: Voy a ordenar una bolsa de m&m por color

De esta forma, me puedo aprovechar de construir → Una estrategia de orden que ordena sin comparar

No estoy corrigiendo Inversiones → Estoy **CATEGORIZANDO MI DATA**

Voy a ordenar por categorias, y voy a contar por grupos

ALGORITMO

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

`CountingSort` (A, k)**:**

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

Arreglo de entrada A → Viene desordenada, con K tipos

C → Arreglo de salida → Donde van las cuentas

B → Arreglo de salida por ordenar

Si estoy ordenando 1 millon de datos, y tengo 1millon de datos distintos, es O(n)

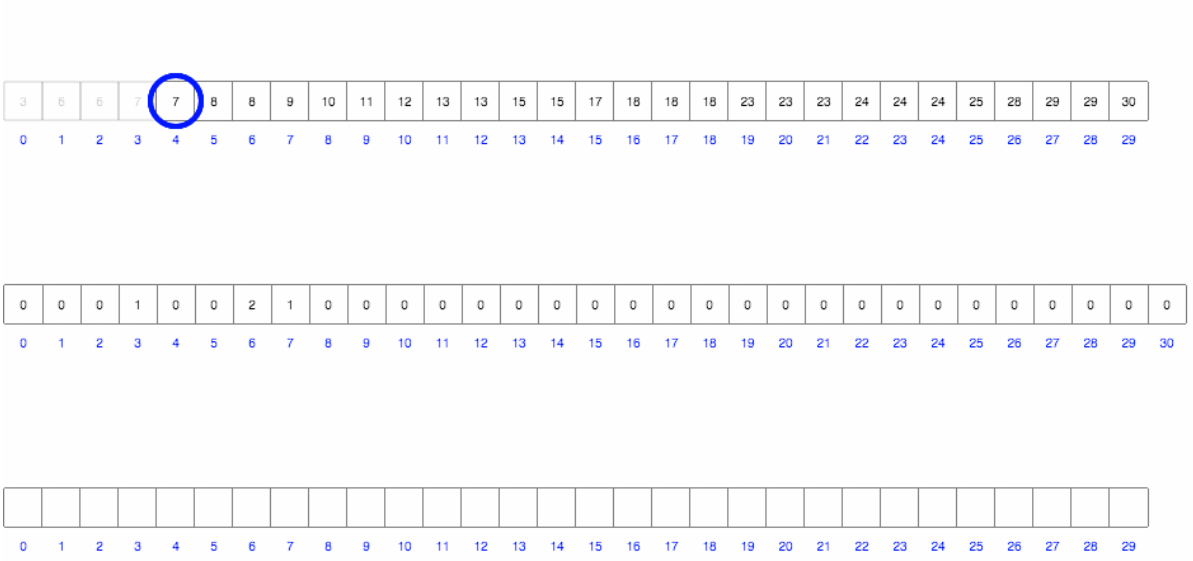
Si tengo un 50% de datos variables → Es perfectamente razonable

COMPLEJIDAD

- La complejidad del algoritmo es $\Theta(n+k)$
- Si $k \in \mathcal{O}(n)$, entonces `CountingSort()` es $\Theta(n)$

K pertenece a O(n) → Especialmente que K sea menor que N preferentemente

VISUALIZACION



Counting Sort → Es Estable

Es muy util, cuando veamos la implementación de Radixsort

CLASE 8:

Lo que se verá en esta clase

Como ordenar por digitos!! → RadixSort

Concpeto de datos mas y menos significativos

La gracia es que radixsort nos permite usar counting sort para ordenar cosas mas genericas que solamente un dato, un color

Porque podemos utilizar caracteres posibles, para ordenar datos mas ambiguos → no sabemos cuantas patentes de autos existen por ejemplo

LA implementacion comun de RadixSort → Usando Countig Sort dentro de Radix

Radix Sort → Ordena por **Digito**

Ordenar por unidad decena centena etc → Ordenar segun Digitos significativos

Palabras pueden tener largos distintos, si no que pensamos por ordenar alfabeticamente

- Usaremos Arboles de Busqueda, para construir **DICCIONARIOS**

Radix sort se utilizaba en un proceso Mecanico → Infinitamente Lento

Esta estrategia de orden lineal es buenisimo

RADIX SORT (Idea)

---

numero natural de d digito : n0 n1 ... n _d-1

Vamos ordenar partiendo por el valor mas bajo () cuando todos tengan el mismo largo

Cuando no son todos del mismo largo → Todos los que tienen 1 letra, 2 letras

Conclusion : Estamos **ORDENANDO POR COLUMNAS**

Problema → Genera llamados recursivos

PAra que el orden lineal funcione → La estrategia tiene que ser **ESTABLE**

Cuando ordeno → Voy a utilizar una estrategia de ordenamiento estable → La columna que ordene seguira **ordenada,**

ORDENACION ESTABLE

Dada la sec a[0 ... n-1] y de salida B[o ... n-1] secuencia resultante de ordenar A, usando algoritmo S

si a y a’ elementos en A tales que para el algoritmo A

☐ COPIAR

RADIX SORT

- Ordena por digito **menos significativo n(d-1)**
- Luego, usando el mismo arreglo, ordena por digito n(d-2), con un **ALGORITMO ESTABLE**
- Luego de ordenar k digitos, los datos estan ordenados si solo miramos fragmento nd-k... nd-1
- Solo requiere de **D pasadas** para ordenar la sec completa

```
RadixSort(A, d ):
    for j = 0 . . . d − 1 :
        StableSort(A,j) // Ejemplo : CountigSORT -> O(n + k)

// COmplejidad O(d*n) -> D constante ->> O(n)
```

	Arreglo inicial	Ordenado por unidad	Ordenado por decena	Ordenado por centena
0	0 6 4	0 0 **0**	0 **0** 0	**0** 0 0
1	0 0 8	0 0 **1**	0 **0** 1	**0** 0 1
2	2 1 6	5 1 **2**	0 **0** 8	**0** 0 8
3	5 1 2	3 4 **3**	5 **1** 2	**0** 2 7
4	0 2 7	0 6 **4**	2 **1** 6	**0** 6 4
5	7 2 9	1 2 **5**	1 **2** 5	**1** 2 5
6	0 0 0	2 1 **6**	0 **2** 7	**2** 1 6
7	0 0 1	0 2 **7**	7 **2** 9	**3** 4 3
8	3 4 3	0 0 **8**	3 **4** 3	**5** 1 2
9	1 2 5	7 2 **9**	0 **6** 4	**7** 2 9

- Se mueven por bloques
- LA gracia de usar Counting Sort para ordenar Decenas
- El orden que venia de las unidades, seguira ordenadas
- Cuando ordene por decenas, va a mantener el orden de las unidades

RadixSort → 1. No necesita llamadas recursivas

2. Se puede hacer in Place → Pierde Eficiencia → Por ejemplo con InsertionSort (Pero este es de orden O(n**2))
3. Counting sort es in place? → NO
4. Entonces Radix Sort si se implementa con CountingSort →→→ No es in place (Pero es de orden O(n))

LA gracia de hacerlo con el **digito menos significativo** es porque va manteniendo el orden

Cuando sean datos de largo distitno → Pasamos a fijarnos en el **mas significatvio**

Supongamos que A tiene n datos naturales con d dígitos y se usa algoritmo estable lineal

- Si cada dígito puede tomar k valores distintos...
- Entonces `RadixSort` toma tiempo $\Theta(d \cdot (n + k))$
- Si d es constante y $k \in \mathcal{O}(n)$, entonces `RadixSort` es $\Theta(n)$

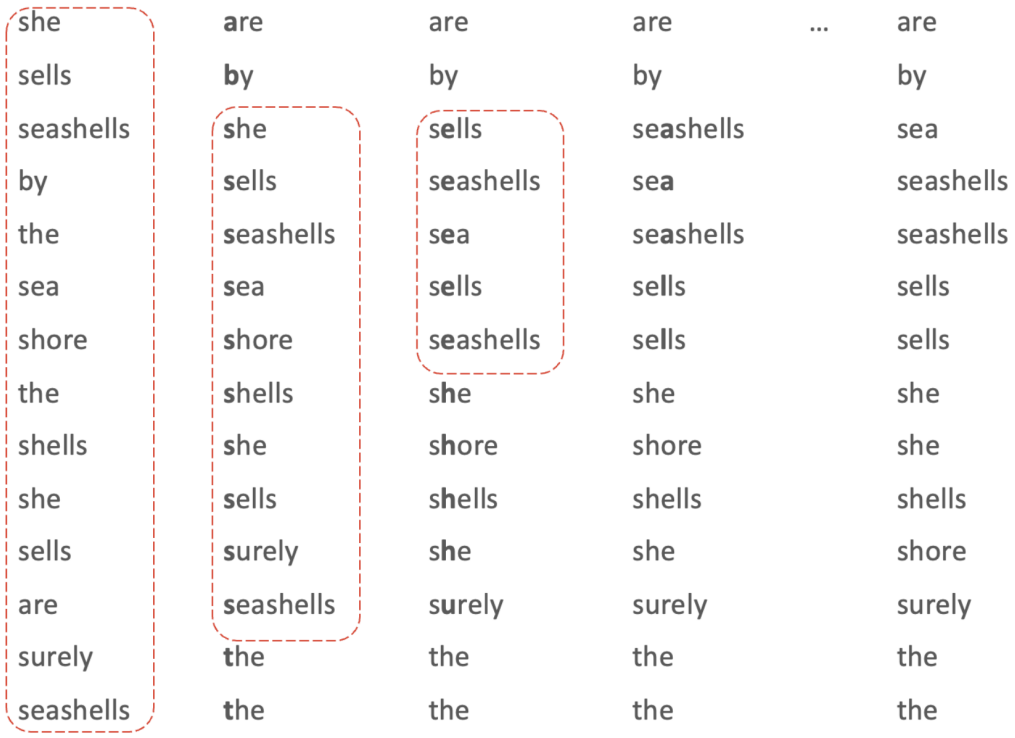
Por lo tanto → **RADIX SORT es LINEAL**

2 IMPLEMENTACIONES de RADIX SORT

1. **LSD (Least Significant Digit):**
   a. Si todos los strings son de **mismo largo**: (Patentes, IPS, Telefonos)
   b. Funciona bien si el largo es **PEQUEÑO**
   c. No recursivo
2. **MSD (Most Significant Digit):**
   a. Si los strings tienen largo diferente
   b. Ordenamos con CountingSort() por primer caracter
   c. **Recursivamente** ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es comu ń en cada subarreglo)
   d. Como Quicksort, puede ordenar de forma independiente

Pregunta prueba: IP, quiero detectar si hay trafico anormal desde una IP de origen externo hasta uno interno → RESPUESTA: Podria usar un Radix Sort por LSD, para ordenar todas las direcciones IP, que vienen ordenadas por rangos de numero, y hacerlo en tiempo LINEAL

MSD EJEMPLO



CUIDADOS CON MSD

- Si un string s1 es **prefijo**de otro s2, s1 < s2
  - she ≤ shells
- Pueden usarse diferentes alfabetos
  - Binarios
  - Minusculas
  - Mayusculas + minusulas + digitos
  - ASCII
  - Unicode
- Para Subarreglos pequeños (e.g $|A| \leq 10$)
  - cambiar a **InsertionSort** que sepa que los primeros k caracteres son **iguales**

Conclusion: Ventaja de saber info de la data → Estrategia mas adecuadas como RADIXSORT o CountingSort

CLASE 8: Arboles Binarios de Busqueda

Ya vimos arboles binarios, pero de manera recursiva para implementar el Heap, podiendo mantener la estructura a un costo de log(n), podemos encontrar lo que buscamos mas eficientemente, nos gastamos cada vez que insertamos u quitamos un log de n, pero al buscar va a ser eficiencia

La palabra busqueda, nos marca que buscar un dato sea muy eficiente, tenemos que gastar mas tiempo en la inserción

En Arbol Binario simple → HEAP → O(n) buscar un dato

Arbol Binario de Busqueda → **O(log n)** buscar un dato → MAS **EFICIENTE** EN LA BUSQUEDA

Truco: **Mantener el Orden**

ARBOL BINARIO BUSQUEDA (ABB)

- Dada una **Llave** asociamos un **valor**
  - ej: Mi rut: “Unico” → Llave
  - En cada rut le asocio valores → Nombre, edad, numalumno
- Si la llave **no esta**, Lo **sabemos eficientemente**!
- Si la llave **esta**, Lo **sabemos eficientemente**!
- Podemos modificar y eliminar pares de llave-valor
- Spoiler: Eliminacion en disccionarios → La eliminacion se puede volver compleja segun hijos → DESICIONEs

ASUMIREMOS → Que las llaves son **UNICAS**

Buscamos → Facilitar la busqueda → Eficiencia

Limitaciones Arreglos y listas

Notemos que en listas y arreglos → No es muy eficiente

- Lista hay que recorrer toda para buscar, y o(1) para insertar
- para insertar o(n) en arreglos

MODIFICACION DE LISTAS

ABB

LA estrutcura de datos, tiene montada la busqueda binaria

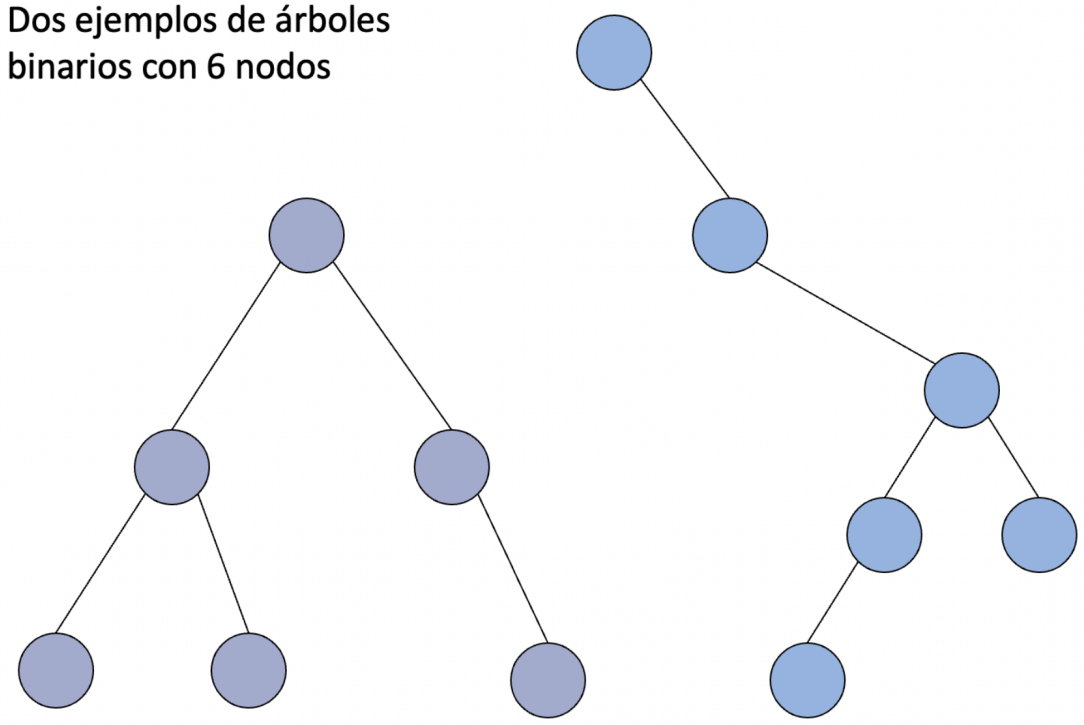
Esta Dividir para Conquistar en la Estructura

Para la estrategia de implementacion, es bueno que tenga un **PUNTERO AL PADRE**

Ese es el puntero **RAIZ**

Se usa por buena practica de **PROGRAMACION**

No todos los nodos tienen la misma cantidad de hijos



Depende del orden de llegada de los datos

Si queda un dato a la raiz, y todos los que llegan son mayores o menores

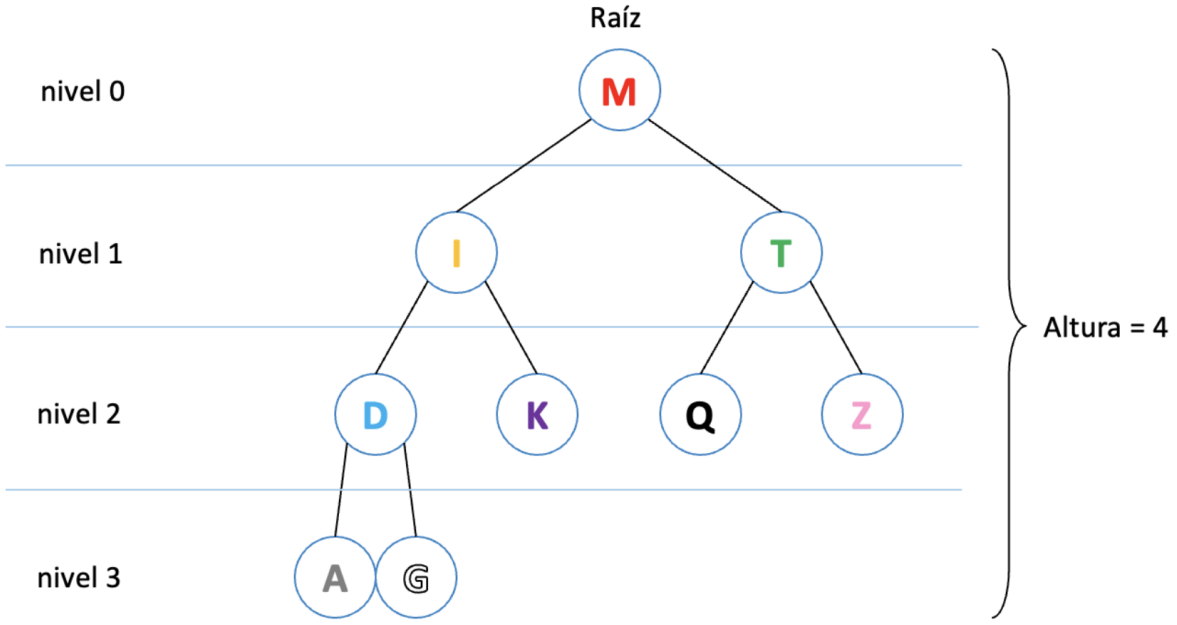
Los datos venian ordenados

Lo que tenemos entonces → Es una Lista

Es el peor caso → En vez de tener comportamiento logaritmico → tendriamos comportamineto Lineal

Si vienen ordenados , todos quedan a un lado → LISTA

ANATOMIA DEL ARBOL BINARIO



La busqueda, va a ser proporcional a la Altura

El mejor caso Log (n) → Lo mismo que el Heap , esta bien **balanceado** el árbol

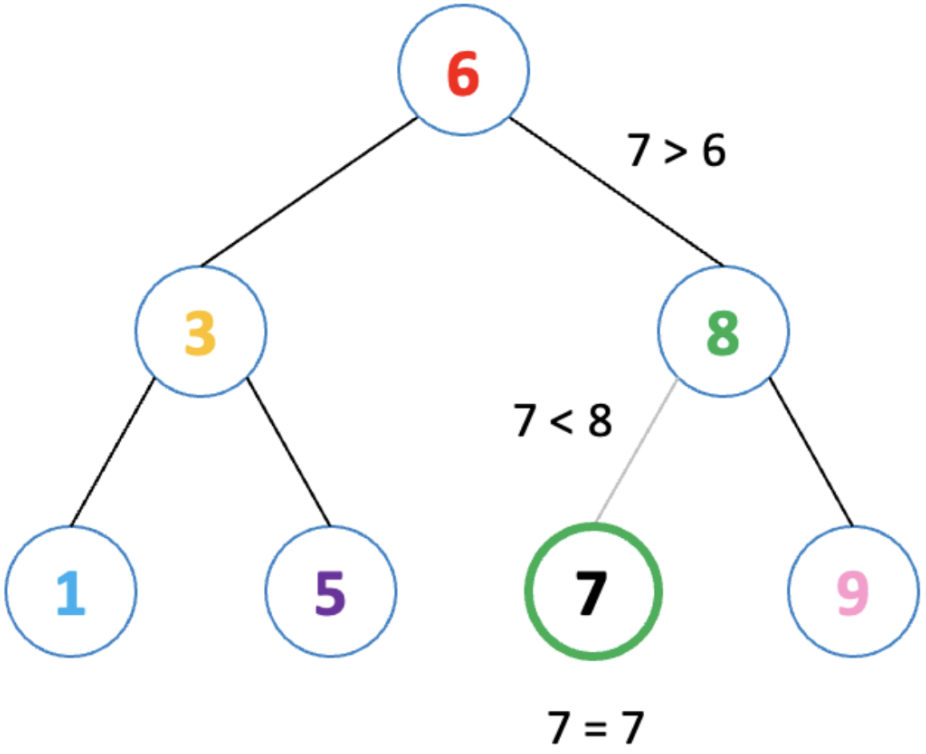
OPERACIONES ABB

Completamos las definiciones de atributos de un **nodo** x en un ABB

- $x.key$ es la llave del nodo
- $x.value$ es su valor
- $x.left$ el puntero a su hijo izquierdo
- $x.right$ el puntero a su hijo derecho
- $x.p$ el puntero al padre

En general no incluiremos $x.value$ en los algoritmos.
Solo será un espacio de almacenamiento

Busqueda



En cuanto busca → Porporcional a la Altura → Si es que el Arbol esta bien balanceado = O(altura)

ALGORITMO

input : Árbol binario de búsqueda A, llave buscada k
output: Árbol binario de búsqueda, o $\varnothing$ si no se encuentra

```
Search (A, k):
1     if A = ∅ ∨ A.key = k :
2         return A
3     if k < A.key :
4         return Search(A.left, k)
5     return Search(A.right, k)
```

Para insertar → Buscar

Si al buscar encuentro → Actualizo

Si al busca no encuentro → Inserto

Clase 9

MODIFICAR EL ARBOL

1. **INSERCION**
2. **ELIMINACION**

Estos dos algoritmos **parten con una busqueda**

Buscamos la llave en el arbol que queremos modificar

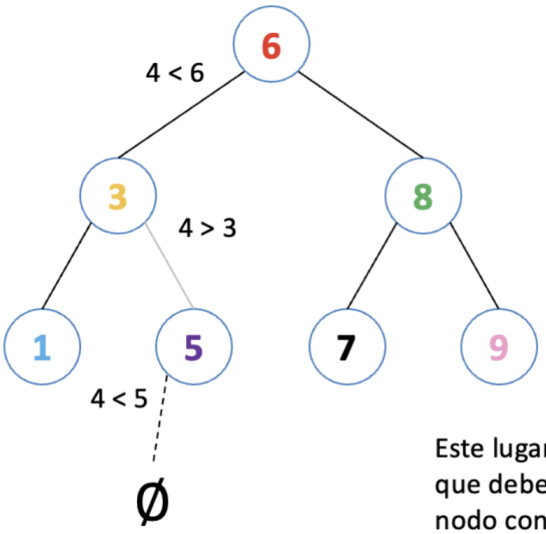
Estas pueden afectarnos las propiedades de arbol de busqueda

Ejemplo

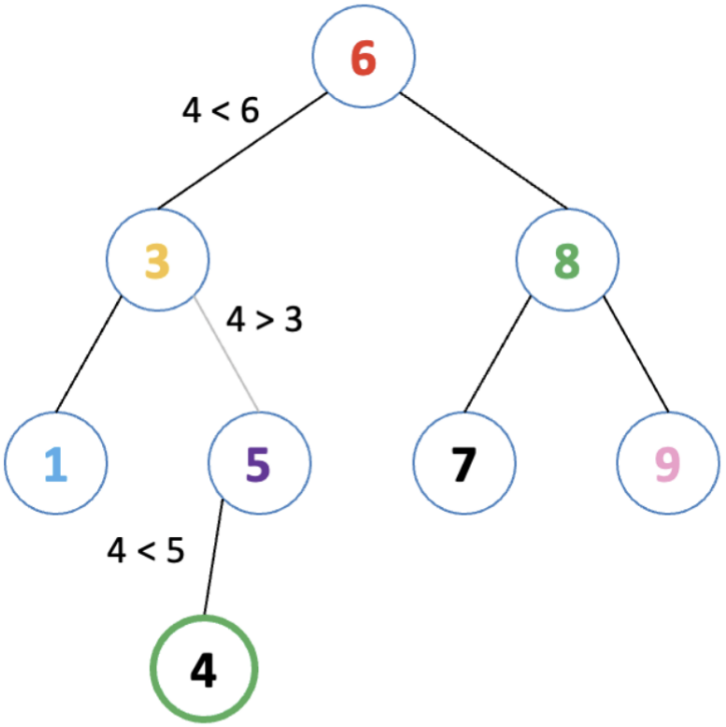
Queremos insertar el 4:

Si encontramos nulo la posicion que encontramos, insertamos

Si ya estaba, depende del problema → Podemos modificar un atributo, por ejemplo actualizar algun valor de la llave



Este lugar vacío es el lugar en que debería haber estado el nodo con clave 4 si hubiera estado en el árbol



PESUDOCODIGO DE SEARCH

Modificaremos la busqueda, para retornar el Padre tambien, con esto nos aseguramos de no perder el padre al insertar! → Nos facilita la pega

```
input  : ABB A, ABB padre p, llave buscada k
output: Tupla con ABB encontrado y su padre
Search(A, p, k):
1    if A = ∅ ∨ A.key = k :
2        return (A, p)
3    if k < A.key :
4        return Search(A.left, A, k)
5    return Search(A.right, A, k)
```

PSEUDOCODIGO DE INSERT

```
input  : Árbol binario de búsqueda A, llave k, valor v
Insert (A, k, v):
1    (B, p) ← Search(A, ∅, k)   ▷ versión que indica el padre
2    if B = ∅ :
3        B ← nodo vacío
4        B.key ← k
5        Conectar B al padre p en la posición adecuada
6    B.value ← v
```

Paso 1: Busqueda → retornara vacío si no estaba, see ejecuta el if del vacio, Si estaba es actualizar el valor

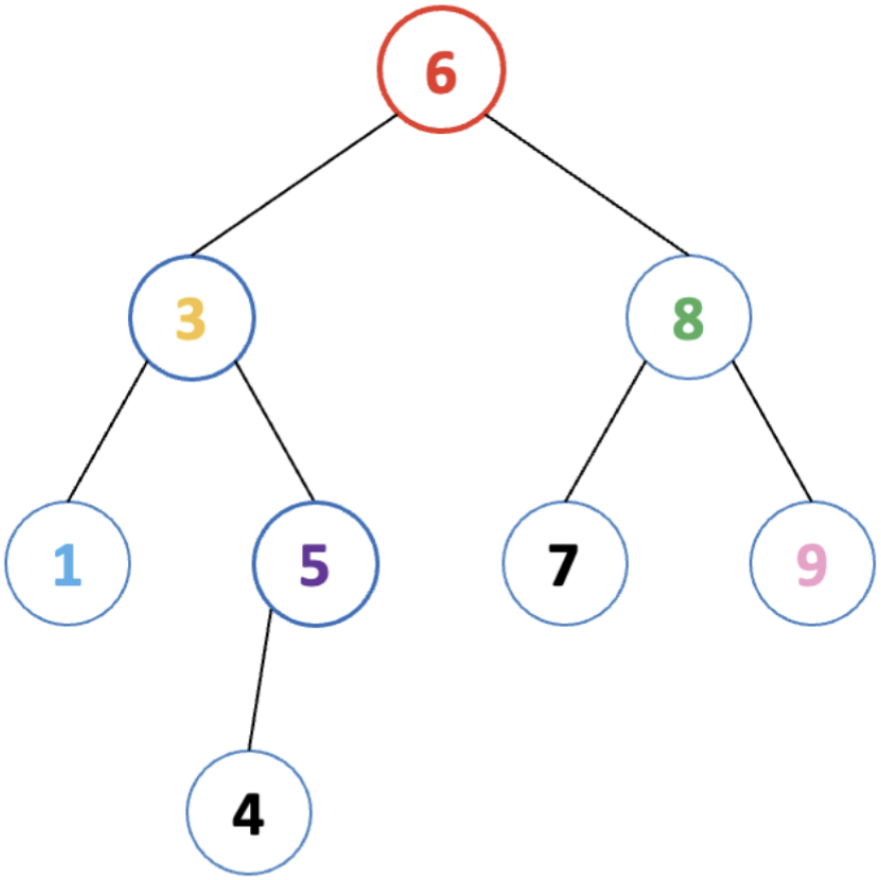
Este algoritmo mantiene la propiedad de Arbol, porque lo inserta ordenadamente, donde debía ir

La eliminacion → Es un poco mas compleja

Si eliminamos una hoja, es facil, simplemente lo sacamos al nodo

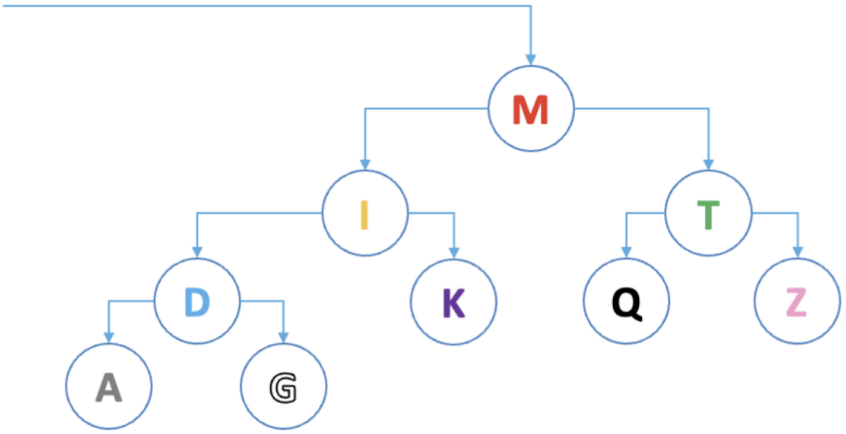
Si queremos eliminar a un padre (tiene hijos) hay que arrastrar los hijos hacia arriba

Que pasa si queremos eliminar la **raiz → PROBLEMA GRANDOTE!!!**



PSEUDOCODIGO DE ELIMINACION

Nos interesa encontrar el sucesor/antecesor del nodo extraído




```
Min (A):
1    if A.left = ∅ :
2        return A
3    return Min(A.left)
```

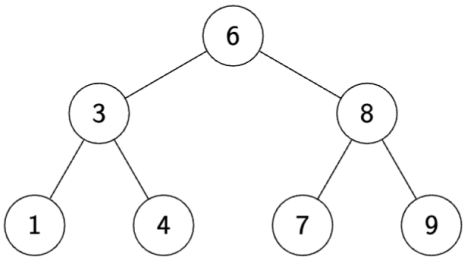
```
Max (A):
1    if A.right = ∅ :
2        return A
3    return Max(A.right)
```

```
Delete (A, k):
1    (D, p) ← Search(A, ∅, k)   ▷ Permite saber el padre de D
2    if D ≠ ∅ :
3        if D es hoja :  D ← ∅ y se elimina la referencia en p
4        elif D tiene un solo hijo H :  D ← H y se actualiza p
5        else:
6            R ← Min(D.right)
7            t ← R.right
8            D.key ← R.key
9            D.value ← R.value
10           R ← t
```

ARBOLES AVL → CLASE 9

Repasando:

ABB garantiza → Orden

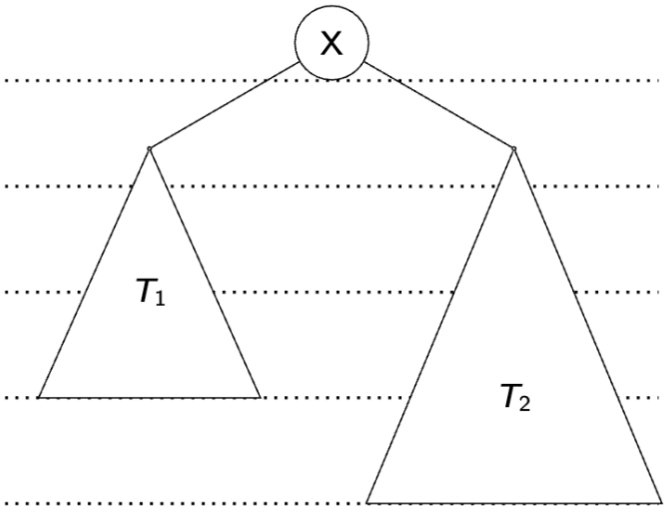


y la gracia es que si colapsamos los nodos...



- Las operaciones en promedio son O(**log n**)
- El peor caso es O(**n**)

Definiremos **BALANCE DE ARBOL → Como maximo 1 nivel de altura entre 2 nodos**



La *propiedad de balance* debe cumplir dos condiciones

1. Asegurar que la altura de un árbol con n nodos sea $\mathcal{O}(log(n))$
2. Ser fácil de mantener

Este último punto es limitante

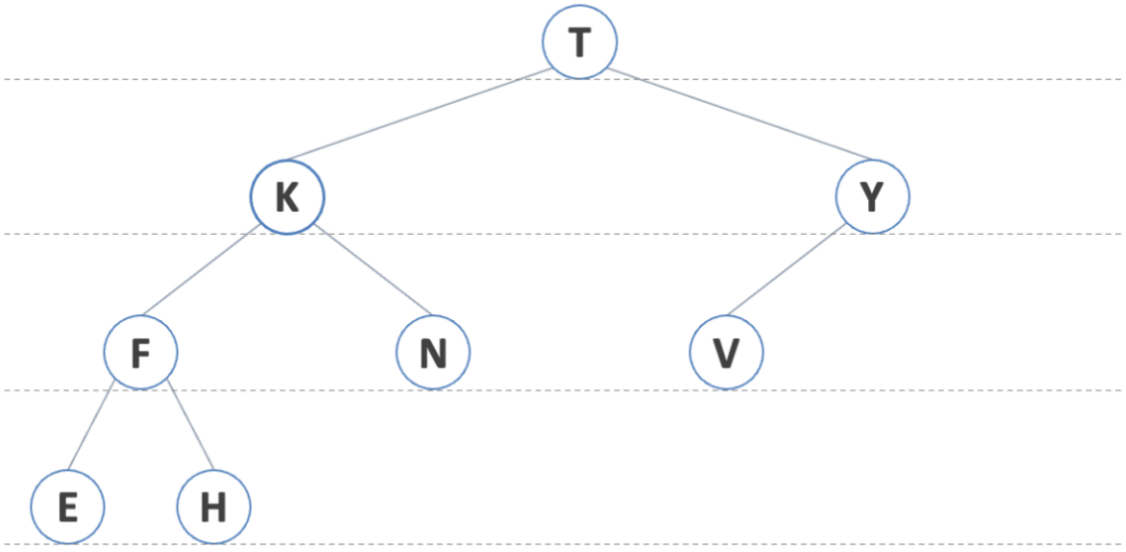
- Llamaremos al (re)balanceo después de las operaciones
- Queremos complejidad **no mayor** que $\mathcal{O}(\log(n))$
- De esa forma las operaciones seguirán siendo $\mathcal{O}(\log(n))$

AVL

Un arbol binario de **busqueda esta AVL-Balanceada si:**

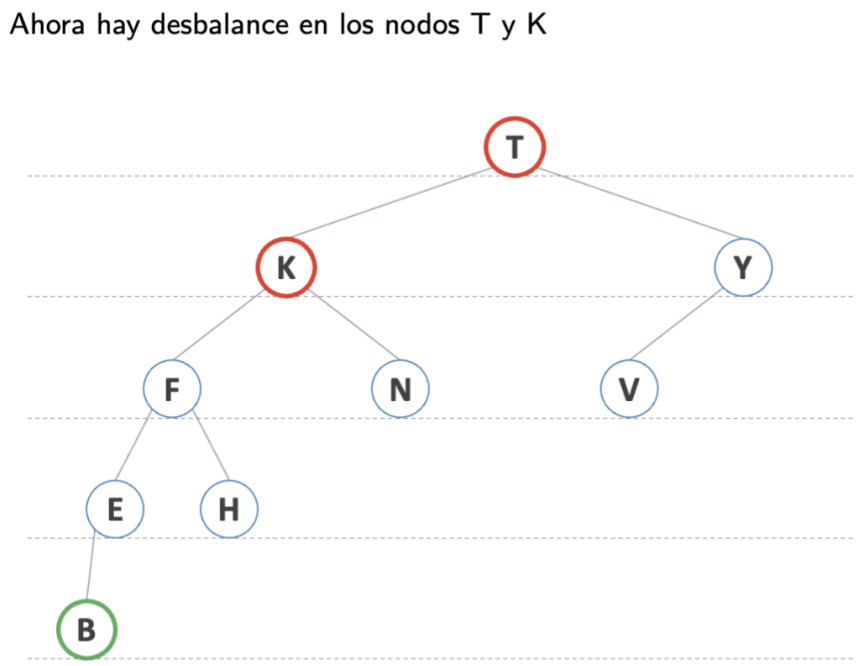
1. Las alturas de los hijos difieren a lo mas 1 entre sí
2. Cada hijo esta **AVL-Balanceado (Recursivamente)**

El siguiente Arbol esta AVL-Balanceado

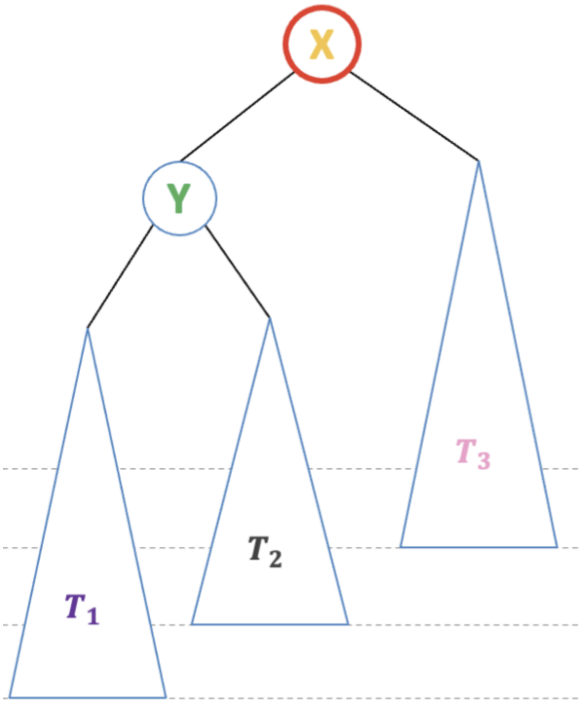


Con esto cada vez que ocurre un desvalance, solo tengo que preocuparme de balancear esa zona del arbol, si corrigo lo el desvalance de un lado, ya esta balanceado totalmente

AL insertar se generan **DESBALANCES - AVL**



Basicamente se ve así:

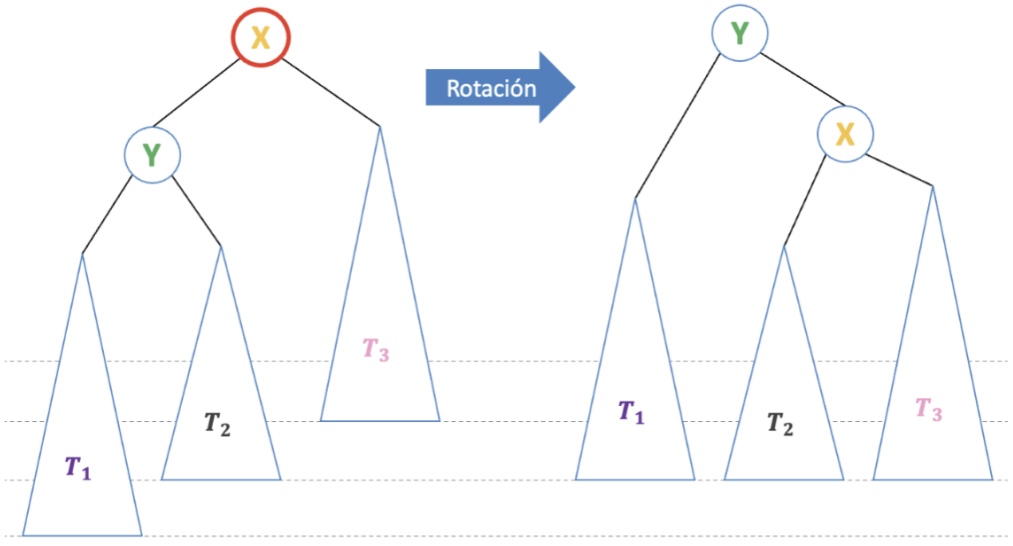


Como corrigo esto → Basta con balancear el nodo x, para corregir el problema

Que pasa si llevo y a la posicion de x, & x lo hago hijo de y:

ROTACION:

Realizamos un intercambio que llamaremos **rotación**



Los 3 subArboles se balancean

Cuanto cuesta → O(1) → Solo tengo que cambiar punteros

Como ya de antes estaban balanceados los anteriores, cuando inserte se avl-desbalanceo

Los Arboles AVL → Son arboles binarios de busqueda **BALANCEADOS**

CLASE 10

11/09/2024

RESUMEN CLASE PASADA

Estuvimos viendo Arboles AVL

Su gracia es que tiene acciones asociadas a loa modificacion (insercion y eliminacion)

que permiten mantener el balance de un arbol

Definimos el balance: Hijo derecho y izquierdo recursivamente tienen hijos con diferencia de 1 de nivel, entonces el arbol esta balanceado

Por lo tanto, al hacer operaciones, hay que mantener balanceado

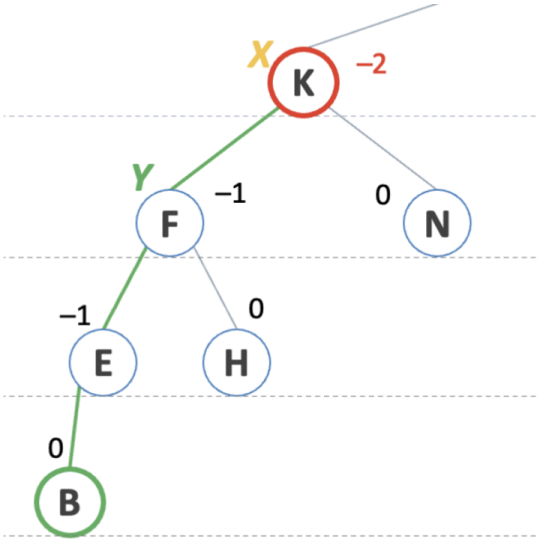
Nos damos cuenta que con inserciones se puede n desbalancear el arbol

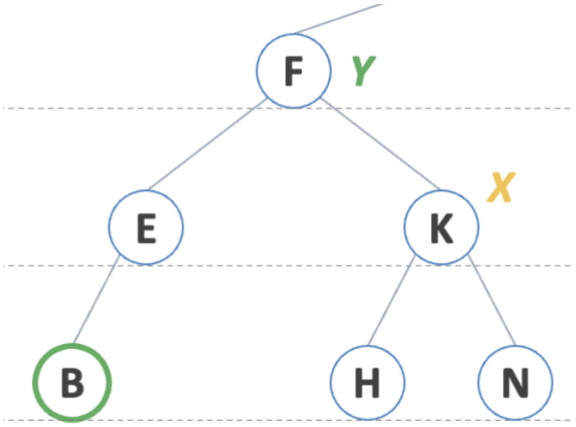
Caso 1: Avanzamos por la izq izq izq , y quedo el sub arbol de la izquierda, mas grande que el derecho → **DESBALANCE HACIA EL EXTERIOR**

Si identificamos el primer nodo que se desbalancea, izq izq izq, y nos fijamos en el nodo, y su hijo por el lado de la insercion y los **ROTAMOs** Podemos recomponer el balance del arbol

Como antes estaba balanceado, por definicion → Parcheamos el desbalance puntual y se mantiene el balance general

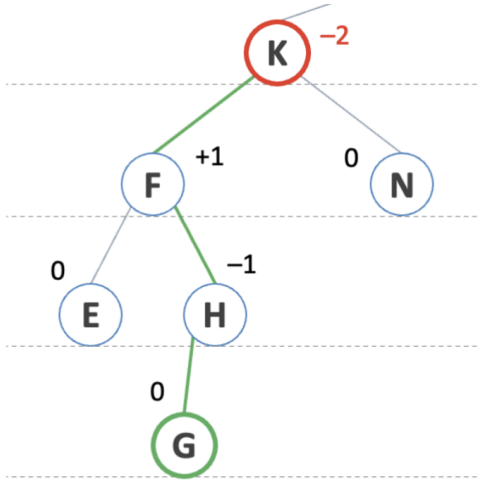
Rotaciones





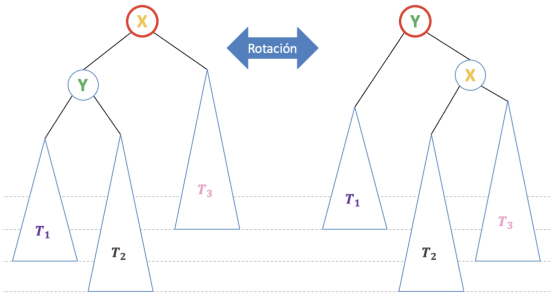
CASO DE CAMBIO DE DIRECCION EN LA INSERCION

Por ejemplo insertamos en → Izq izq izq y al final **derecha**



Esta ruta **no soporta** el mismo balance

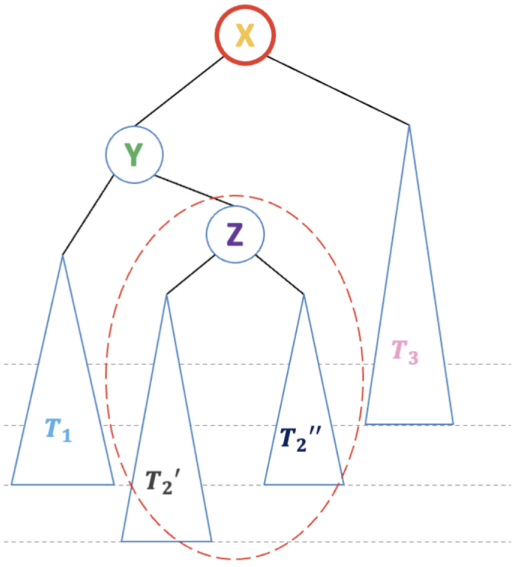
Si aplicamos la misma forma → T2 queda mas profundo que t1 o t3 → Sigue estando desbalanceado:



El truco para arreglar la insercion con cambio de direccion (cambio de ruta)

La forma de arreglar esto, es la **ROTACION DOBLE**

Lo que logramos, es hacer devolvernos al **caso anterior**

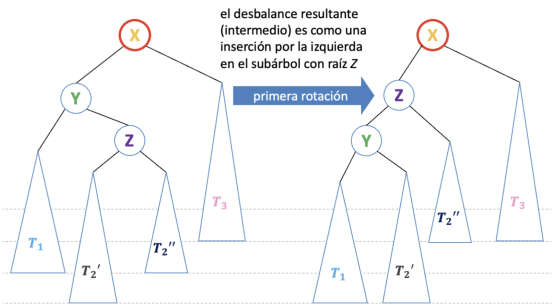


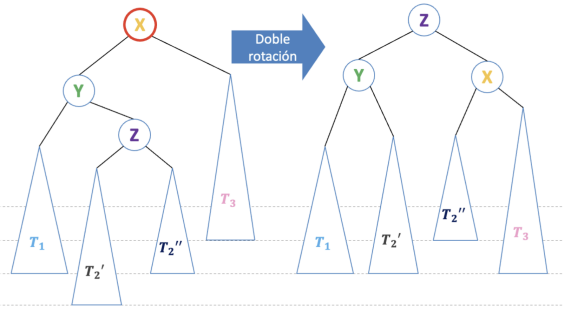
- Recordemos que X es el primero desbalanceado
- Por lo tanto, sabemos que los sub-árboles de T_2 son AVL

Con esto logramos volver a la forma anterior, T1 (t2′) esta mas abajo que t2 (t2′′)

Y resolvemos, de la misma forma

Rotamos x con Z





1. ROTAMOS A LA IZQUIERDA: Y y Z.

2. ROTAMOS a la DERECHA:

Lo que hacemos para implementar esto en el codigo

Identifico

Hice insercion

Me devuelvo buscando ruta

Etiqueto los nodos con los punteros

Se que por y el lado izquierda y.left = t1

y.rigut = z

z. left = t2′

etc

Lo que hago en el codigo es solo : **AJUSTAR PUNTEROS**

```
La estrategia en estos casos define los nodos X, Y, Z seg«un

1. X es el primer nodo desbalanceado que encontramos cuando s

2. Y es el hijo de X en la ruta de la inserci«on

3. Z es el hijo de Y en la ruta de la inserci«on
```

Como se puede hacer en un Codigo:

Stack auxiliar de movimientos, baje por izq, baje por izq, baje por derecha y izquierda

Llevo el registro de como me moví hasta llegar a la insercion!

→ Asi sabemos como devolverse

EJEMPLO EN CLASES

Lista de nodos a insertar en un AVL: {M,D,H,B,A,C,S,K,I}

1era insercion : M

Esta balanceado si

2a insercion

M , D

Balance :

```
     M (-1)
   /
H (0)
```

3a Insercion

```
   M (-2)
  /
D (+1)
 \
    H (0)
```

Aqui se perdio el desbalance

La estrategia dice

```
   M (-2) -> X
  /
 D (+1) -> Y
  \
     H (0) -> Z ---> Este es Z por el cambio de direccion
```

Rotacion 1: Intercambiar Z con Y

```
     M (-2) -> X
    /
   H (+1) -> Z
  /
D   (0) -> Y
```

Rotacion 2: X con Z

```
          H
         /   \
        D     M
```

Siguiente insertamos B

```
          H -1
         /   \
        D -1    M 0
       /
     B 0
```

```
          H -2
         /   \
 Y ->  D -2    M 0
       /
```

```
   B -1  -> X
  /
A 0
```

Hora de rotacion:

CASO 1: Rotar x con y

```
        H -2
       /   \
             B    M 0
    /  \
  /
A 0     D -1
```

Balanceado

Insertamos C

```
      X H -2 -> Primero en desbalancear : X
       /   \
          Y   B +1  M 0
    /  \
  A 0   D -1 Z -> Cambio de direccion
            /
           C 0
```

Rotacion 1: Y con Z

```
      X H -2 -> Primero en desbalancear : X
       /   \
          Y   B +1  M 0
    /  \
  A 0   D -1 Z -> Cambio de direccion
```

```
                  /
                C 0
```

ESTUDIAR ESTO → Puede ser pregunta de prueba

AVL: Arbol binario de busqueda → Con BALANCE

CONCLUSION

Las rotaciones se hacen en O(1)

La insercion es O(h) → Segun la h (altura) del arbol

En caso de rotacion O(1)

Total O(h + h + 1) = **O(h)**

La relacion entre h y n?

La minima altura para un n de nodos dados, va a estar dado por el arbol completo

Si puediera hacer eso, la mayor cantidad de nodos la minima altura

Eso es Log N

El truco es calcular el minimo nivel de maximo nivel de desbalances (1)

El hijo izq, tenga un nivel menos que el hijo derecho

Si lo pienso recursivamente, hay un caso base: 1 nodo balanceado

2 nodos a la izq o derecha → Balanceado

De ahi en adelante

1 nodo al izq 2 nodos a la izq 3 nodos a la izq y otro a la derecha

Con esa estrategia logramos calcular que el nivel minimo tambien es log n

Proporcionalmente entonces es **log n**

CLASE 11

23/09/24

Antes:

- EDD y Algoritmos (estrategias)
- Sorting {selection, insertion, ... , radix}
- selection y insertion compara (1 dato a la vez)
- selection el menor el menor el menor
- insertion
- Problema → Solo corrige una inversion a la vez
- Teorema: Esto nos genera una complejidad de n**2
- Solo en el caso de los datos ordenados es O(n)

Merge

- Despues nos fuimos a descomponer el problema
- Se resuelven varias inversiones
- Merge: Mezclar 2 listas ordenadas
- Dividir para conquistar, para que llegar a lista de tamaño uno, que se mezclan recursivamente
- Requiere memoria adicional, pero nos permite resolver mas de una inversion en un paso
- En este caso, con costo de memoria adicional, logramos una complejidad de nlog(n)

Quicksort

- LLegamos a quicksort, que se aprovecha de definir un pivote, y asegurar de tener todos los mayores a la derecha, y los menores a la izquierda

- REcursivamente los pivortes van quedando en su posicion ordenada
- Nuevamente llegamos a ordenar n pivores y logn divisiones → nlogn
- Pero: TIENE PEOR CASO → N**2
- Escoger como pivote siempre en el extremo de un segmento de datos ordenados
- Por eso Quicksort se define escogiendo **aleatoriamente** un pivote

Luego vimos → Heap sort, que ordena en funcion de tener una EDD

Heap sort como una cola de prioridades, nos permite tener un elemento mas o menos grande, y tener todos ordenados relativmanete hacia abajo

Luego podemos sacar siempre a la raiz, y ordenar

Siempre la complejidad es nlogn → Sort Entero!

HeapSort, no es estable → Porque no se sabe la posicion exacta de donde esta el mayor o menor, puede estar a la derecha o izquierda

En general se cumplen estas complejidades

Orden Lineal → Tenemos informacion adicional sobre los datos

Counting sort, sabemos el rango de datos distintos que puede haber, ese numero acotado generalmente es **MENOR A LA CTD DE DATOS A ORDENAR**

Esta estrategia → No compara → Cuenta cuantos datos distintos hay de cada uno

Ej: Contar los m&m, contar es O(n) contar antes O(n), poner en la posicon O(n) → O(n+n+n) = O(n)

Radix Sort

Esta estrategia funciona ordenando cosas mas generales, por ejemplo patentes, que quiero ordenar por digitos, para eso necesitamos un algoritmo de ordenamiento estable

Por ejemplo counting sort

Diccionarios

Definimos el dicc como EDD que permite almacenar la data de manera **EFICIENTE,** que sea eficiente **BUSCAR**

Heap, es un caso especial de un arbol binario

Pero no es un Arbol binario de busquda

Arbol binario de busqueda

Vamos insertando a un lado de cada nodo insertado, → Dividir para conquistar **en la data**

Aplico una estrategia algoritmica para construir una EDD

Si el arbol esta razonalmente equilibrado, la alutra del arbol se relacion logartimicamente con el numero de nodos

La eficiencia de busqueda, es practicmanete busqueda binario → Logaritmico

Para insertar, buscamos dentro del arbol → Search, log (n), si esta , se actualiza si no, se inserta

El problema : **DESBALANCEAR**

Insertar nodos en orden, nos generara una **LISTA → Peor caso**

Para lidiar con eso vimos los arboles avl

ARBOLES AVL

Estrategia de balance, no podemos forzar a que un arbol siempre tenga cierta altura, la estrategia de balance es que la rama izquierda de un nodo cualquiera con la derecha, no pueden tener una diferencia de mas de 1 (+2 o -2)

Con esto defininimos formas para encontrar al primer no desbalanceado a la izquierda o a la derecha, y generamos estrategias de rotaciones (**simples y dobles**) para balancear

El costo de rotar es O(1)

Luego las operaciones de un AVL son logartimicas : Con eso tenemos una EDD tenemos operacioes en O(logn)

Dado que buscar log n y rotar o(1)

Cuando el arbol esta lleno, la altura es proporcional a logn

Usando magia, vimos que la altura esta con el minimo numero de nodos, es similar a un fibonacci

A continuacion veremos los arboles 2-3,

En lugar de hacer avl, insertar para ver si se desbalanceo, y luego arreglarlo

En el 2-3, inserto garantizando que se mantiene altura, y reparar

ARBOL 2-3

Lo que hacemos es modificar el arbol binario de busqueda

Teniendo 2-nodos : Que tendran llave, y si no son hojas, tendran 2 hijos

3-nodos: que tendran 2 llaves distintas y ordenadas, y si no, son hojas tedrán 3 hijos

Garantiza que la altura es proporcional al numero de nodos

Que es un arbol 2-3?

Es un arbol de llave valor

Tiene 2 tipos de nodos

Nodos 2: Tienen una llave

Nodos 3: Tienen 2 llaves → Y la menor esta izq, y la mayor a la der

Los nodos pueden no tener hijos, o tener exactamente 2 hijos (nodo 2) o 3 (nodo 3)

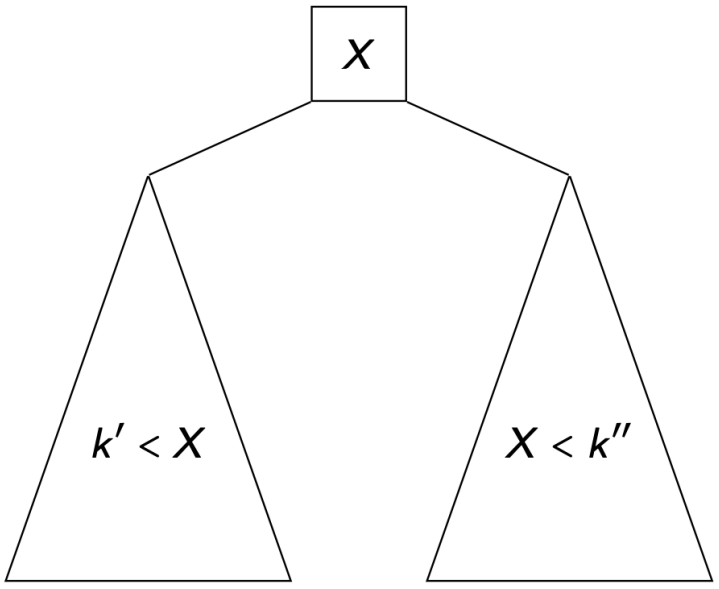
O los tiene todos o ninguno!

Conclusion: Nos permite garantizar que las hojas estan siempre todas al mismo nivel

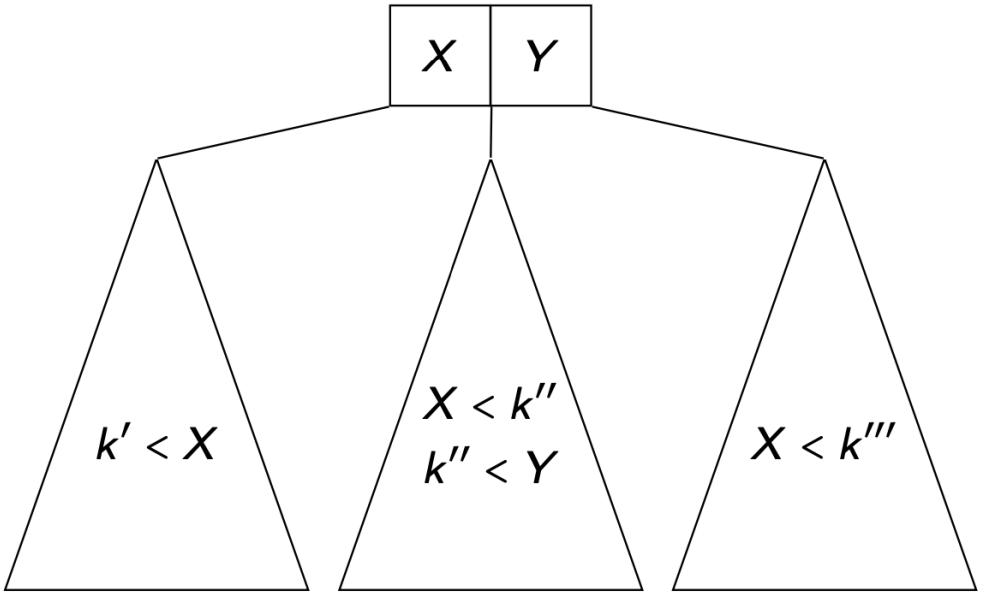
Y de ahi hacia arriba → Siemre hay nodos completos, nunca hay nodos 2 con menos de 2 hijos, y nodos 3 con menos de 3 hijos

- Si es 2-nodo con llave k
  - las llaves k' del hijo izquierdo son $k' < k$
  - las llaves k'' del hijo derecho son $k < k''$
- Si es 3-nodo con llaves $k_1 < k_2$
  - las llaves k' del hijo izquierdo son $k' < k_1$
  - las llaves k'' del hijo central son $k_1 < k'' < k_2$
  - las llaves k''' del hijo derecho son $k_2 < k'''$

NODO 2



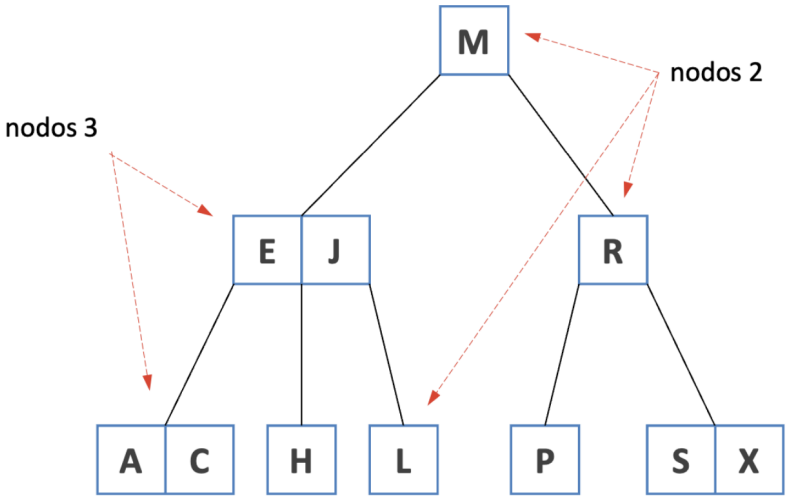
NODO 3



en la derecha era Y < k´´´

Al igual que un arbol binario de busqueda, la forma en que se construye, depende del orden de **llegada**

Un ejemplo de árbol 2-3 con llaves alfabéticas



Lo que se puede observar, es sigue siendo un arbol binario de busqueda

Si entro por la raiz, se que por la izq estan los menores y lo de der son mayores

Es un arbol 2-3 (de busqueda)

PREGUNTA PRUEBA → ESTUDIAR

Pregunta hipotetica: Algoritmo para listar ordenadamente los nodos de un arbol 2-3

En que cambia de un arbol binario de busqueda?

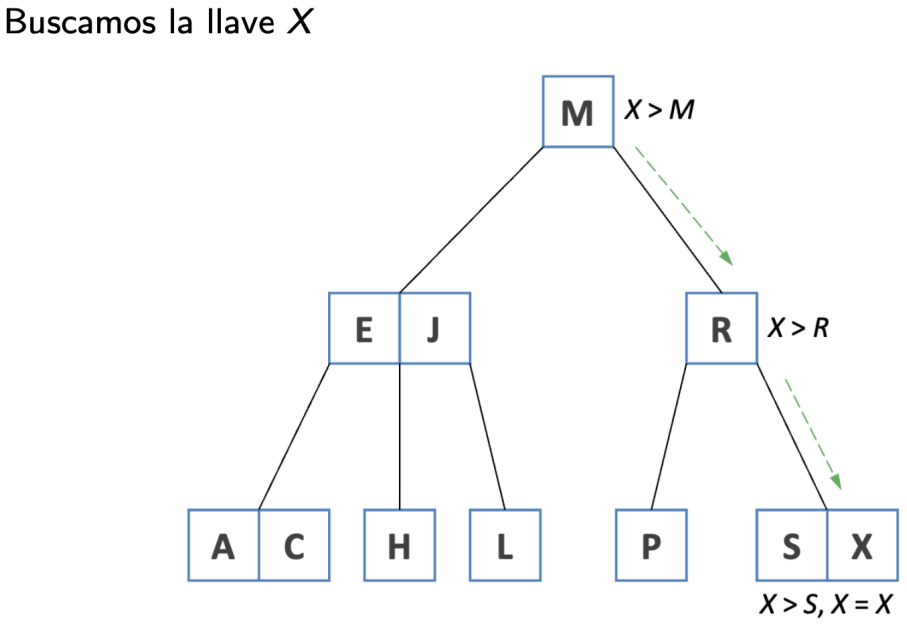
2 if

if nodo 2 : Recorrido hacia 2 nodos

if nodo 3: Recorriendo hacia 3 nodos

El resto, es todo igual que un arbol binario de busqueda

busqueda



Que pasa si quisiera insertar un nodo z? → Tendria que transformar en un nodo 4 al s x , para no romper la propiedad del arbol

Por lo que el de al medio, tendria que subir

Insercion

Al insertar llaves seguiremos la siguiente estrategia

- Se inserta como llave *múltiple* en una hoja existente
- Si la hoja era 2-nodo, queda como 3-nodo y terminamos
- Si la hoja era 3-nodo, queda como 4-nodo (con 3 llaves) por ahora
- La llave central del 4-nodo sube como llave múltiple al padre (**split**)
- Se repite la modificación de forma recursiva hacia la raíz

El árbol solo crece en altura cuando la raíz se llena
al recibir una llave desde un hijo

Conclusion: Estos desde la parte teorica son intersantes, pero en la vida real, practicamente, se usan para introducir los rojo negro y b+ para los indices en bases de datos

ORDEN DE INSERCION

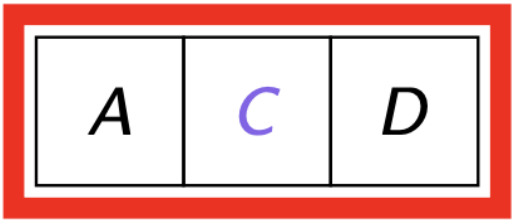
Si esta vacio, inserto la primera hoja (no tiene hijos)



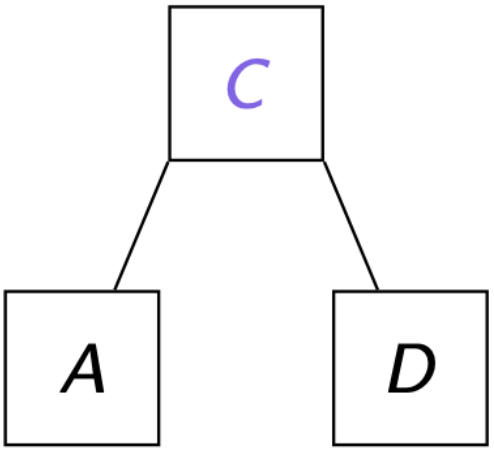
Por definicion siempre insertamos en las hojas :



Vuelvo a insetar en el nodo, pero se vuelve incorrecto (nodo 4)



Por algoritmo, el de al medio sbe → **AUMENTO EN ALTURA**

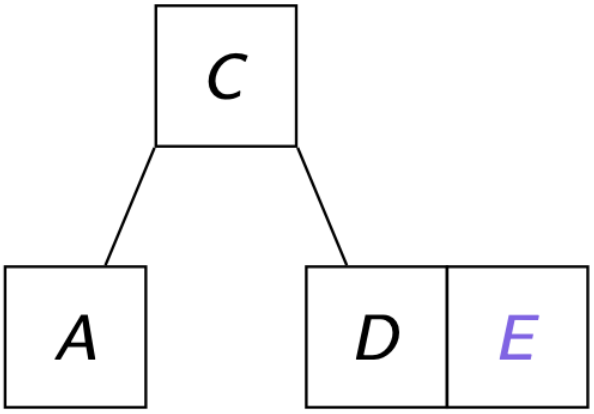


Esto se llama **SPLIT**

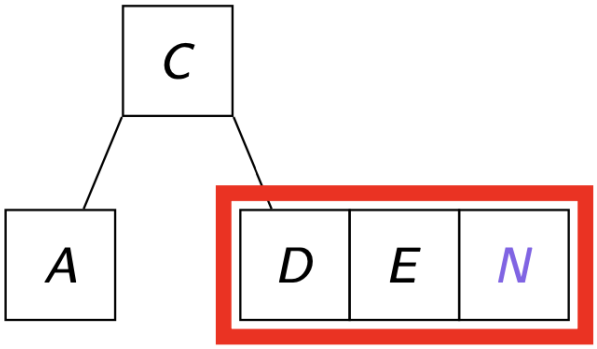
El arbol solo crece → Cuando se llenan los nodos 2-3

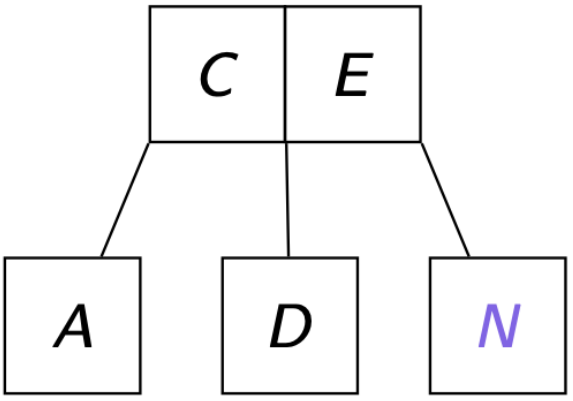
Por eso, crece por debajo de log(n)

Sigamos: Insertar E

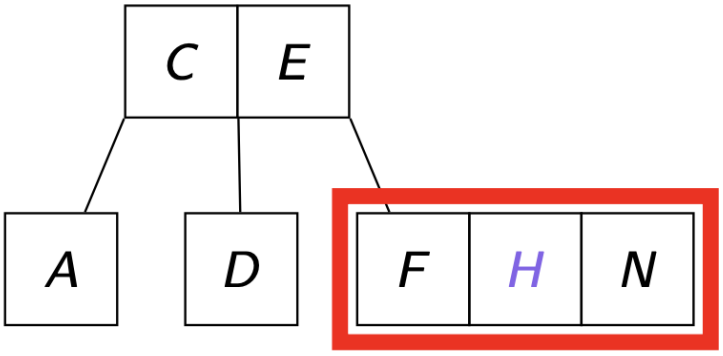


Insertar N

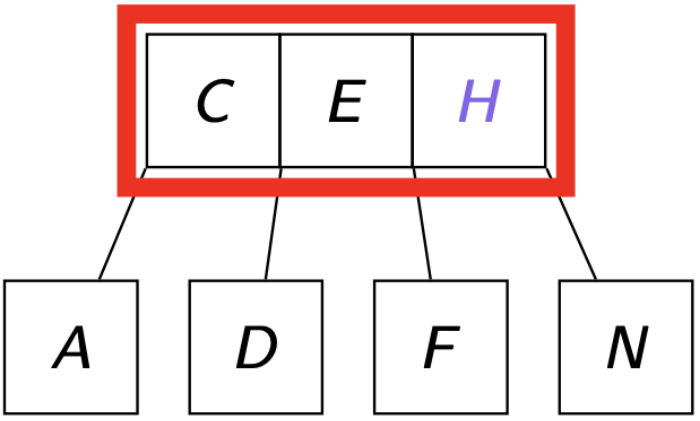


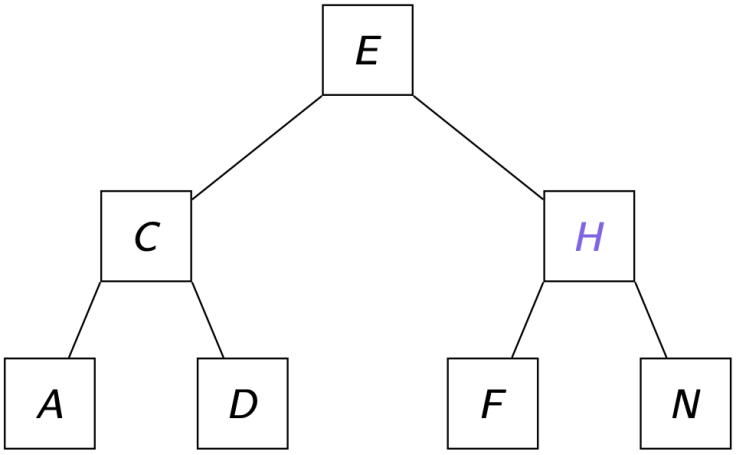


Insertamos F y H



Hora de crecer





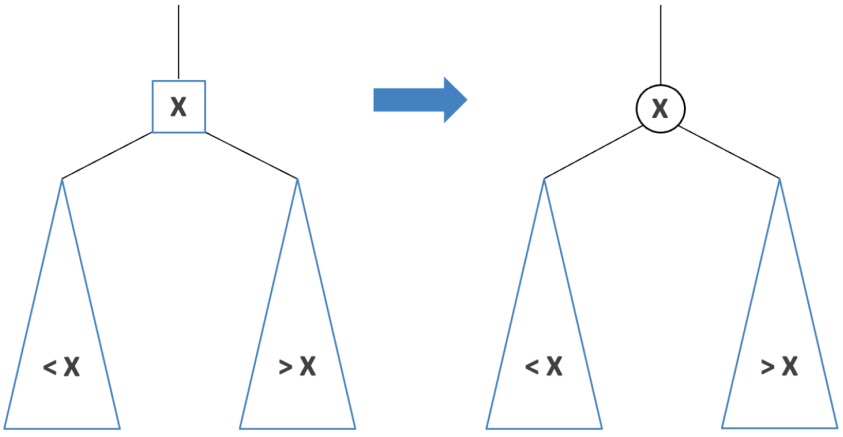
Ahora mezclaremos los AVL y los 2-3 para construir una EDD, mas simple y con la eficiencia de un 2-3

ARBOL ROJO NEGRO

Para llevar esto a un Arbol binario de busqueda, vamos a representar los nodos 2, como un nodo binario, pero los nodos 3, los voy a representar como una jerarquía

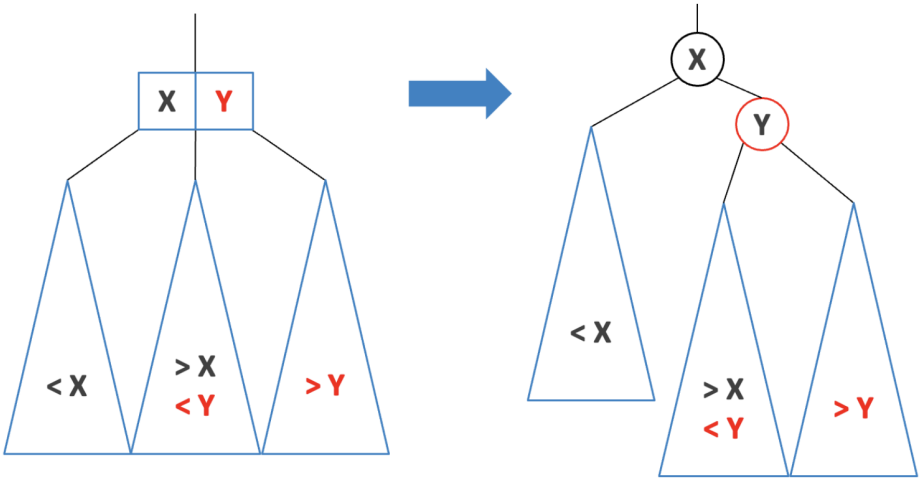
Los nodos 2 → Facil

Los 2-nodos se representan igual



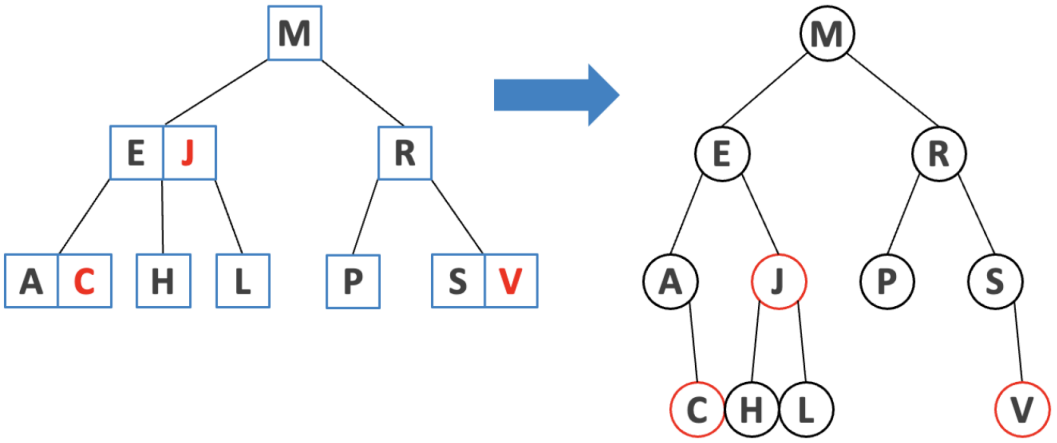
Los nodos 3 → Rotacion AVL

Reasignamos dos de los hijos a un nuevo nodo



NOTAR: La diferencia permitida es de 1

CONVIRTIENDO 2-3 en BINARIO

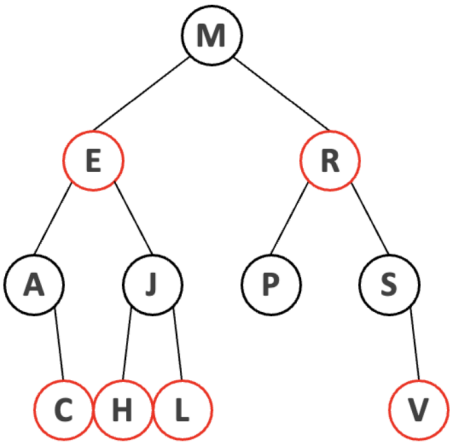


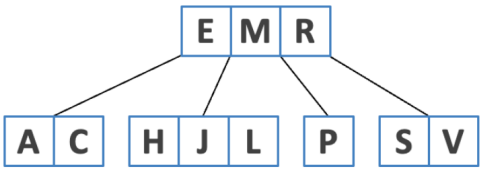
Definicion de ARBOL ROJO O NEGRO

1. Cada nono o es rojo o negro!
2. La raiz del arbol es **negra → If raiz → NEGRO**
3. Si un nodo es rojo, sus hijos deben ser **negros**
4. La cantidad de nodos negros camino a cada hoja desde un nodo cualquiera → **ES LA MISMA**

Ademas → Las hojas vacías se consideran nodos **negros**

Posible pregunta de prueba → Rojo negro → Devolverlo a 2-3 o bueno → 2-4 (nodos 4: 3 llaves y 4 hijos)





Recomendacion para la prox clase, ver las inserciones en ROJO-NEGRO

El nodo que se inserta → SIEMPRE ES ROJO

CLASE 12

25/09/24

Introduccion

Que tener en el informe

1. Tener los pseudocodigos, para tener a mano para modificarlos,
2. Tener la forma de demostrar correctitud, termina y hace lo que hay que hacer
3. Revisar estrategias para mirar correctitud → (en unos casos miramos la salida, en otros como se descompone, etc)
4. Podria haber una pregunta que no es sorting, y es Heap,
5. La edd de un heap, se usa mucho en la practica → Para busqueda rapidas

No entra 2-3 ni rojo-negro (usualmente 2-3 se usa para rojo-negro)

Pregunta facil: inserte un nodo en un 2-3

Pregunta dificil: haga el pseudocodigo → Dificil (si lo pensamos en la prueba y no antes)

ARBOL ROJO-NEGRO

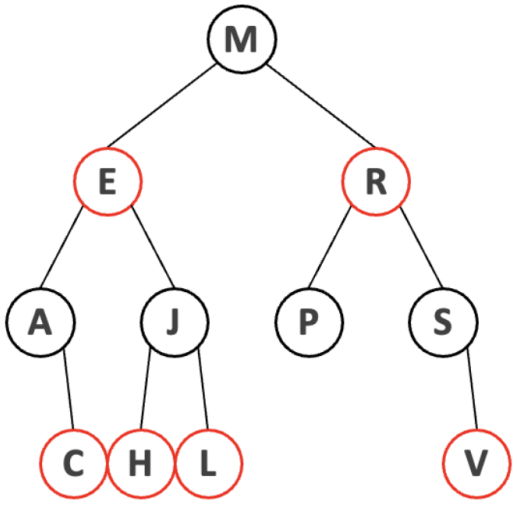
Propiedades → Anotar en los apuntes

1. Cada nodo es rojo o negro

2. La raiz del arbol es negra

3. Si un nodo es rojo, sus hijos deben ser negros

   → Si un hijo rojo tiene un hijo rojo
   → El arbol esta desbalanceado



4. La cantidad de nodos negros camino a cada hoja desde un nodo cualquiera debe ser la misma

→ Con esto comprobamos si los nodos estan balanceados, porque estan al mismo nivel (del 2-4)

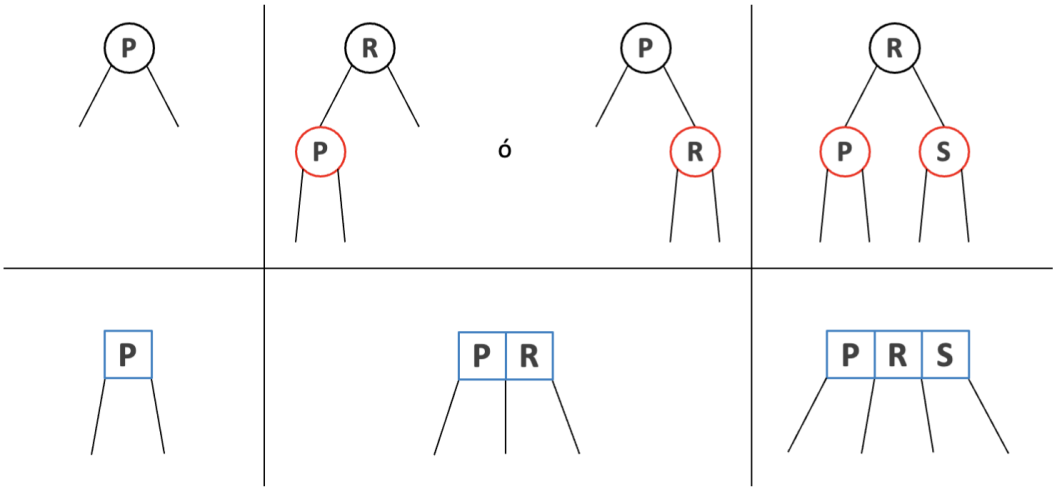
El split en un rojo negro → Es solo un cambio de color

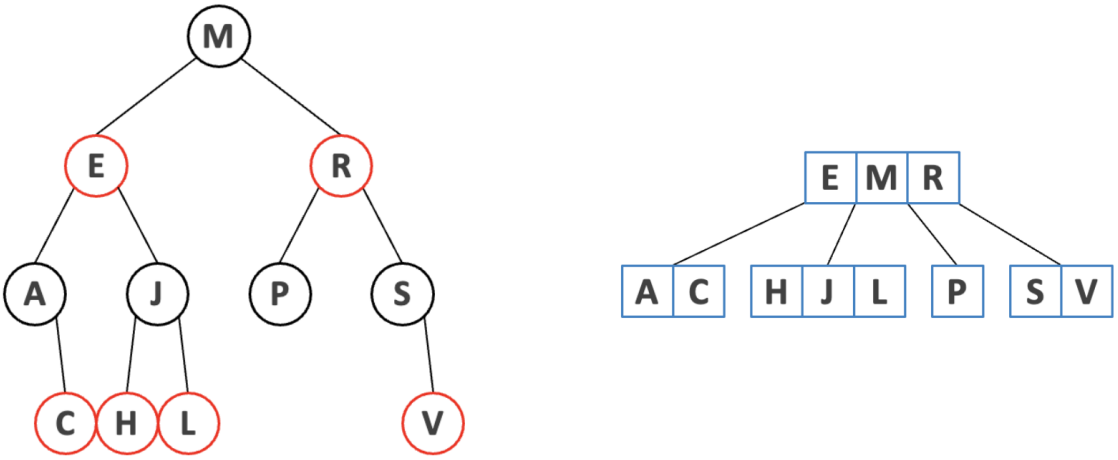
Todo arbol 2-4 tiene un rojo-negro equivalente, y viceversa!!!

No todo arbol rojo-negro tiene un arbol 2-3 equivalente

Pregunta de prueba: Darnos un arbol, revisar si no es rojo-negro, y si no, ajustar

Con este nuevo modelo, podemos tener árboles equivalentes según





Ambos son arboles de busqueda, como el arbol garantiza que las operaciones son O(logn) → El otro tambien me facilita operaciones de O(logn)

INSERCIONES

Lograr la habilidad de insertar → Requiere ayuda para entender que estamos haciendo

Tenemos el arbol rojo-negro, y necesitamos un 2-4 para entender que esta sucediendo

El algortimo me hace entender la relacion entre el rojo negro y el 2 cuatro

Observaciones

El arbol rojo-negro se evita tener que recorrer todo el arbol para encontrar desbalances, tan solo se fija en los colores de los nodos tíos

Si el tio es rojo, significa que se lleno, insertar sería tranforma en un nodo-5

Si inserto debajo de un nodo negro, un nodo rojo, no tengo que hacer nada → Esta valido

CLASE 13

RESUMEN I1

Preguntas: Modificar algo existente, mejorar algo, o pasar un algoritmo nuevo, evalauar correctitud, complejidad, ver que hace

Tenemos que entender pseudocodigo → Entender que hacen

Ej de pregunta: Puedo modificar un Heap, para evaluar 2 prioridades en vez de una?

Puedo volver estable un algoritmo → Por ej para implementarlo en radix sort

Objetivos a evaluar en la I1

- [ ] Comprender y comparar algoritmos de ordenacion clasicos
- [ ] Modificar algoritmos conocidos para resolver problemas
- [ ] Diseñar algoritmos usando tecnicas e ideas estudiadas
- [ ] Demostrar correctitud y complejidad de algoritmos

- [ ] Comprender el funcionamiento de EDDs de árboles
- [ ] Comparar estructuras basadas en arboles

Selection sort → La mejor forma de no olvida: nemotecnia : Seleccionar ordenado

Busca al menor, y lo inserta en la salida (in place: insertandolo al inicio)

Problemas de prueba: NO DISPONE MEM ADICIONAL

Con esto por ejemplo → No podemos usar mergesort → Porque se vuelve ineficiente sin memoria adicional

No distingue arreglos ordenados desde ya, siempre iterará por el arreglo, buscando al menor, que puede ser el primero (no se da cuenta).

O(n**2)

Insertion sort: → Inserta ordenadamente

Toma un elemento, y busca hacia atras su posicion (not in place: busca su posicion en el arreglo nuevo → y)

distingue arrays ordenados : Tiene el mejor caso O(n)

Depende de los intercambios que haya que hacer en el array de salida basicamente → Eso nos dirá la complejidad

MergeSort: Mezcla mitades ordenadas recursivamente

Recursivamente vamos sacando mitades, y al llegar a una lista de tamaño 1, en ella, se van ordenando con el algoritmo **merge**

MergeSort ordena mejor que n**2 , porque al dividir el problema, resuelve mas de una inversion al mismo tiempo, por el teorema, si no hace mas que uno, no es menor a n**2, aqui merge sort, corrige inversiones por niveles (como si fuera un arbol)

Dividiendo el problema, podemos corregir mas de una inversion a la vez

Probable: Nos daran un algoritmo perversamente parecido a merge sort → Estudiarse la ec de recurrencia, porque

QUICKSORT: ordena pivotes de forma recursiva

Salio de la idea de buscar la mediana, que podriamos encontrarla sin tener que ordenarla toda, solo usando pivotes

Nos dimos cuenta que partition, deja un pivote en su posicion ordenada, lo que logramos con esto, es que quicksort, implementa partition, para que todos los elementos hayan sido pivotes → Con esto nos aseguramos que cada uno esta ordenado → Con eso el arreglo esta ordenado

Quicksort siempre se puede mejorar → Implementacion de mejoras:

Primera mejora : Usar insertion sort XD : Porque la recursion tiene un precio (no hay cerveza gratis) → e.g para n ≤ 20 (sub-secuencias pequeñas), cuando los tamaños son pequeños, insertion sort aun en n**2, se comporta mejor que quicksort en nlogn, para tamaños pequeños

Ademas se aprovecha del orden de estar ordenado → Mejora : Usar insertion para secuencias pequeñas

Segunda mejora : Mediana de 3 elementos como pivote

Si yo escogo aleatoriamente 3 elementos, con esto aumento la probabilidad de que mi pivote, no sea un elemento borde (caso borde), y tomo la mediana entre los 3, la mediana entre ellas, no podria ser un extremo, y con esto envenenar el algoritmo

Tercera mejora: PArticionar en 3, con esto mejoramos los casos en que hayan muchos empates

Esto modifica partition → Vamos a necesitar 2 pivotes (j1 y j2)

Hepas, lo vimos como apoyo para ordenar con una estrategia de dividir paraconquistar, para ordenar parcialmente una data en un arbol, un heap es una cola con prioridad

Conocemos tres: FIFO, LIFO (stack), o HEAP con orden de prioridad, el elemento que quiero extraer es el que tiene mayor prioridad,

Habitualmente las preguntas de heaps → Implementacion de cola priorizada

Ej: Tengo una cola de pacientes, necesito saber cual se encuentra en peor estado, quiero saber cual es el avion con menos combustible para hacerlo aterrizar → HEAPS

No hay que duplicar data, mejor crear heap de indices, que apunta a la misma data original

Por ejemplo si tengo los datos ordenados, y tengo un heap adicional, que solamente la llave con el código, llego a la celda del heap, y con un puntero me voy al heap origianal

BuildHeap, no olvidarla : Lo unico que hace, es apartir de un arreglo cualquiera

Complejidad: O(n) → Con la reduccion que hace, y la cantidad de operaciones → Es posible demostrar que

PREGUNTA A ESTUDIAR: Cambiar un Maxheap a un MinHeap en un tiempo mejor a nlogn, la respuesta era BuildHEAP O(n)

Estudiar porque puede entrar una variacion de esta pregunta

Que pasa si tengo un maxheap, y sobre ese tengo implementado un heapsort, y que ahora en vez de ordenar de creciente, ordenar decreciente, pero no podemos empeorar la

Un algoritmo que lo puede hacer, seria recorrer al reves,

Si me construyo un minheap y uso heapsort, no es asi

- [ ] La funcion a hacer es, → ESTUDIAR

Como me pueden pedir prioridad orden de llegada,

Que problema tiene un heap, respecto a una cola fifo, como las operaciones son logartimicas, la cola fifo es o(1) de salida y de entrada, pero Heaps no es para ordenar por llegada, el logaritmico

Orden Lineal:

Ojo tener restricciones: Diccionario de datos → Conjunto actoado de datos

Si fueran los numeros naturales (infinitos) → No podriamos realizar

Si el orden de datos (rango) es menor o igual que la cantidad de datos → Me va a dar un orden lineal, pero si es mayor el rango de datos (por ej: el doble), se me va a n**2 el algoritmo

Ojo con mas significativo y menos significativos

Por ej: Un string se construye de derecha a izquierda, si me pidieran ordenar en otro orden → Por ejemplo ordenar en otro orden

Arboles: