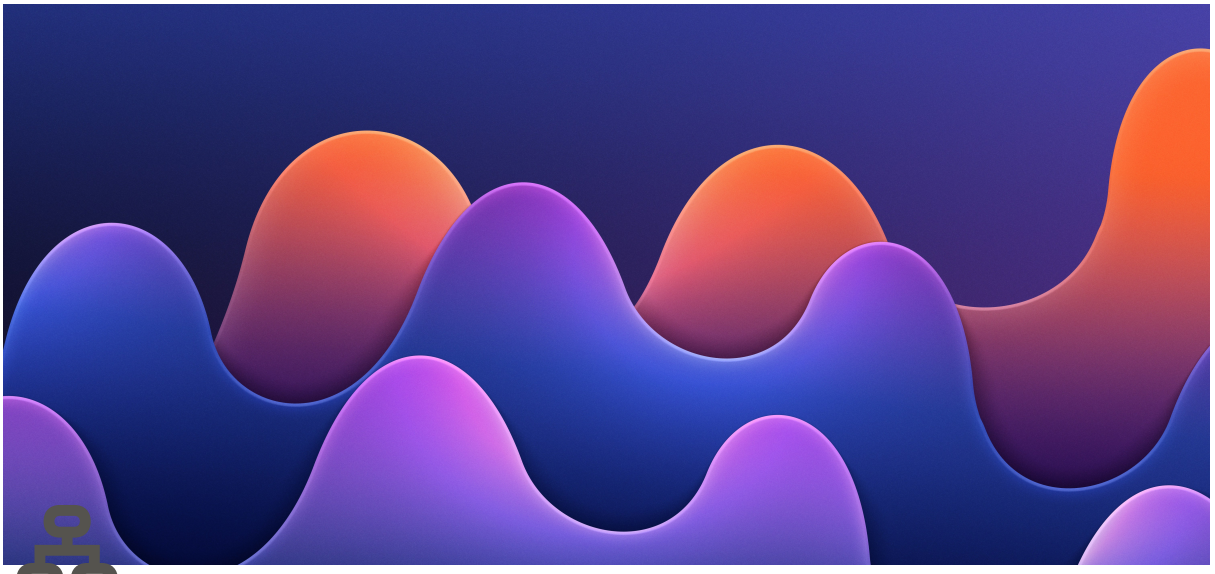


Apuntes I2 - EDD IIC2133

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

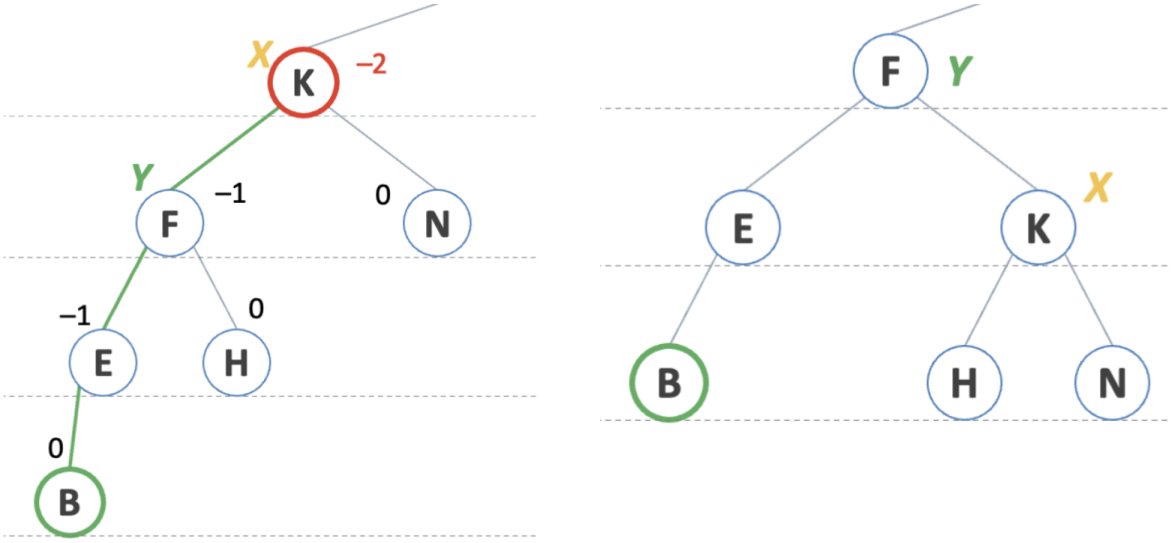
Nos damos cuenta que con inserciones se puede n desbalancear el arbol

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

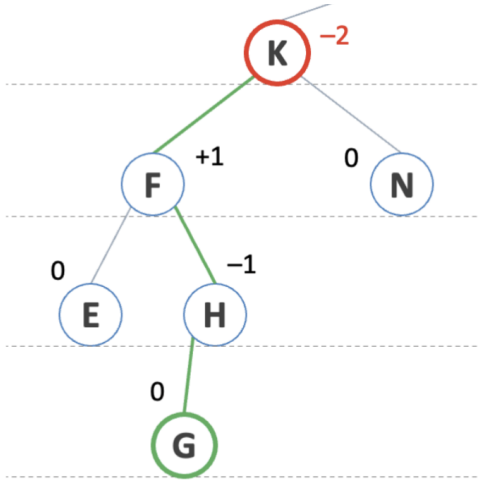
Como antes estaba balanceado, por definicion → Parcheamos el desbalance puntual y se mantiene el balance general

Rotaciones



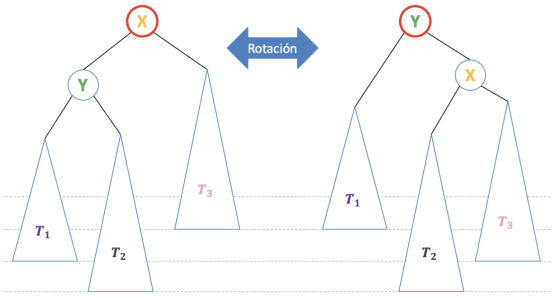
CASO DE CAMBIO DE DIRECCION EN LA INSERCION

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



Esta ruta **no soporta** el mismo balance

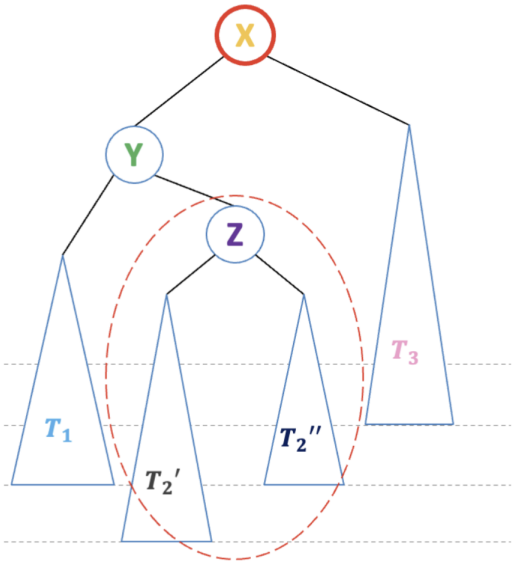
Si aplicamos la misma forma → T2 queda mas profundo que t1 o t3 → Sigue estando desbalanceado:



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**

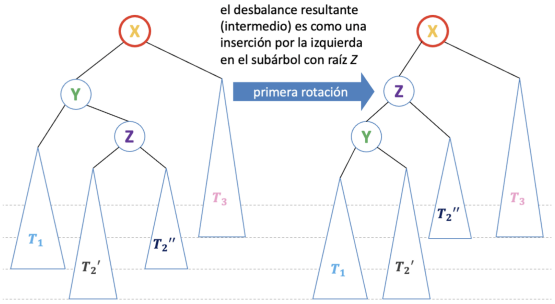


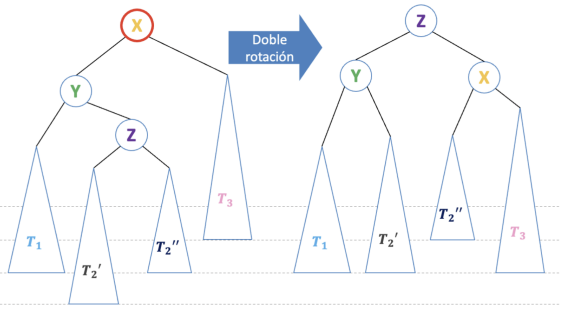
- Recordemos que X es el primero desbalanceado
- Por lo tanto, sabemos que los sub-árboles de T_2 son AVL

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 tendrán llave, y si no son hojas, tendrán 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)

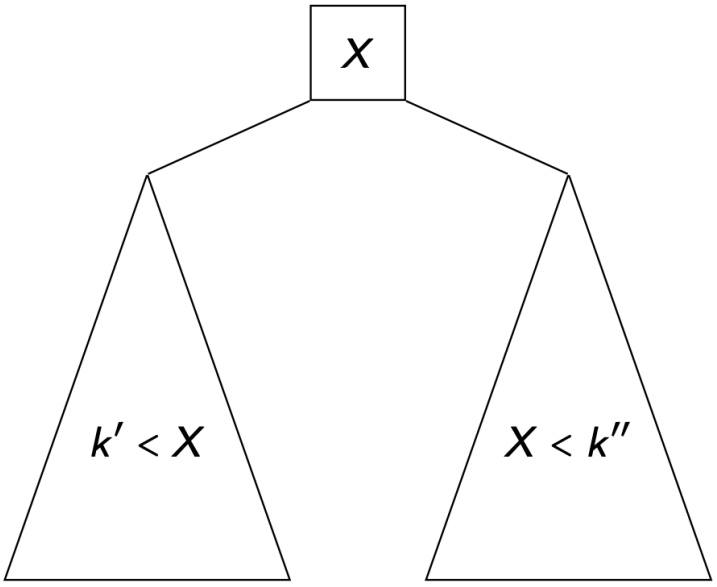
O los tiene todos o ninguno!

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

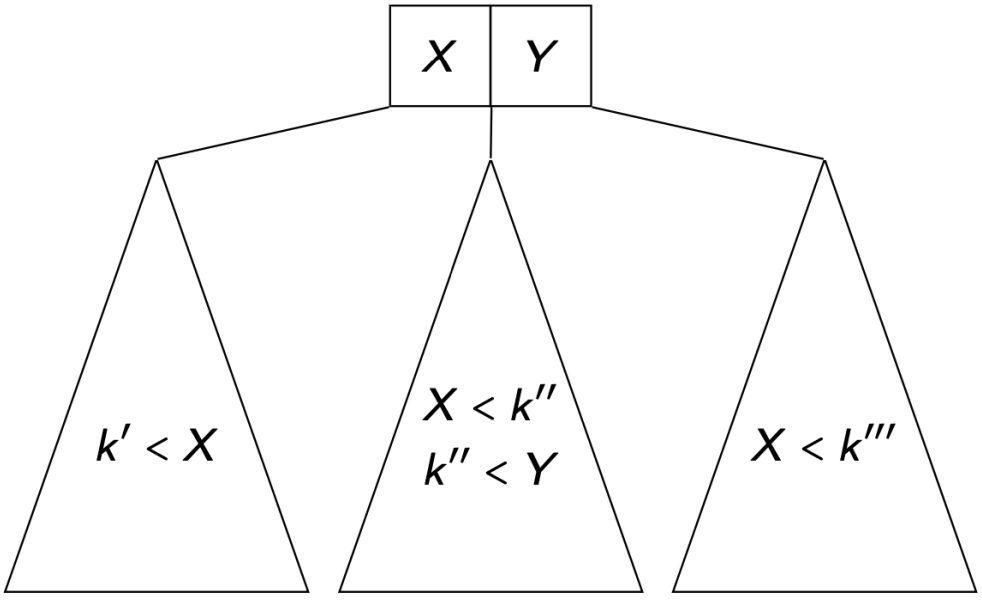
Y de ahi hacia arriba → Siemre hay nodos completos, nunca hay nodos 2 con menos de 2 hijos, y nodos 3 con menos de 3 hijos

- 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



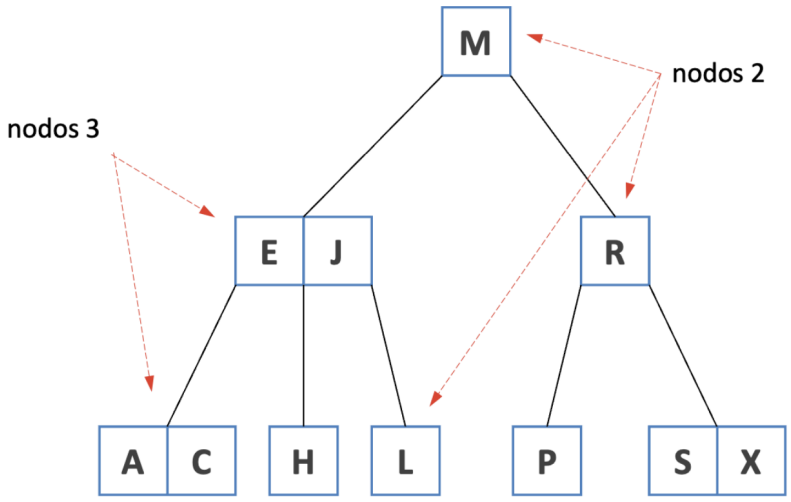
NODO 3



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?

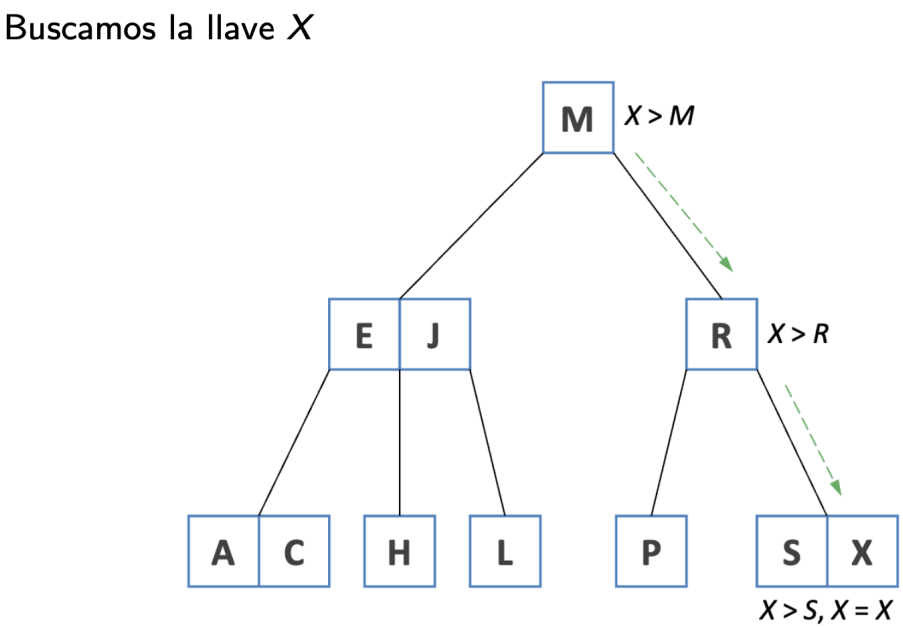
2 if

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

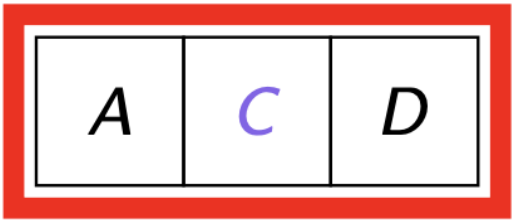
Si esta vacio, inserto la primera hoja (no tiene hijos)



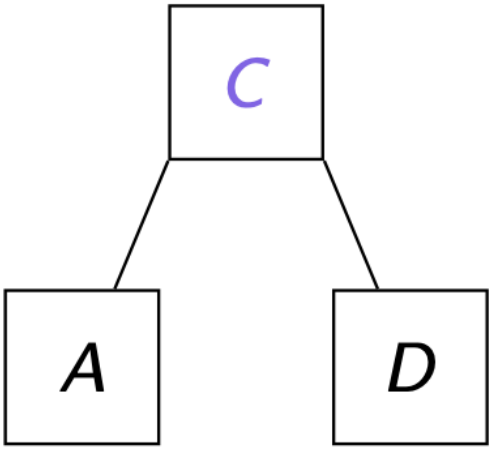
Por definicion siempre insertamos en las hojas :



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



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

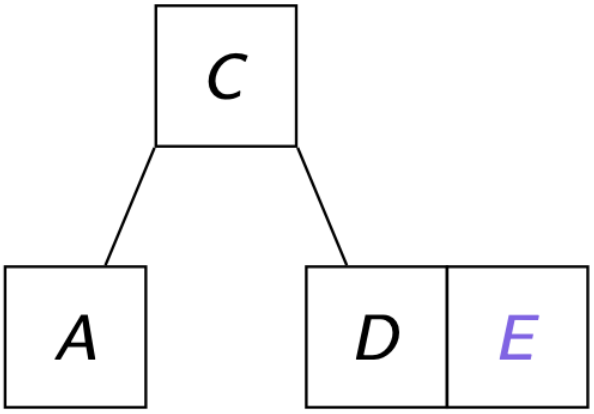


Esto se llama **SPLIT**

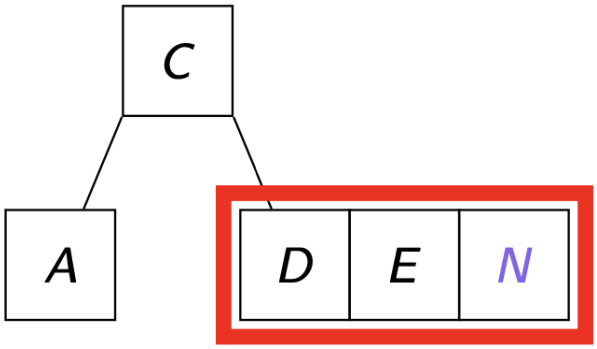
El arbol solo crece → Cuando se llenan los nodos 2-3

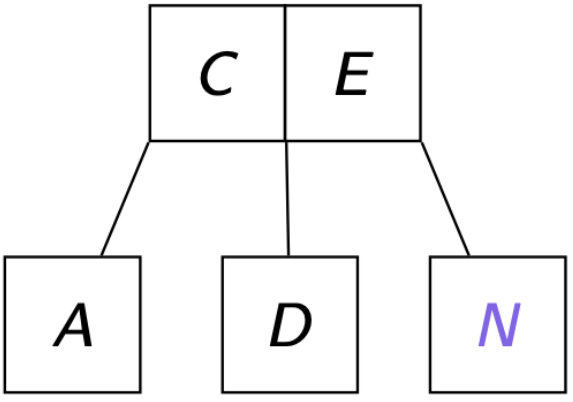
Por eso, crece por debajo de log(n)

Sigamos: Insertar E

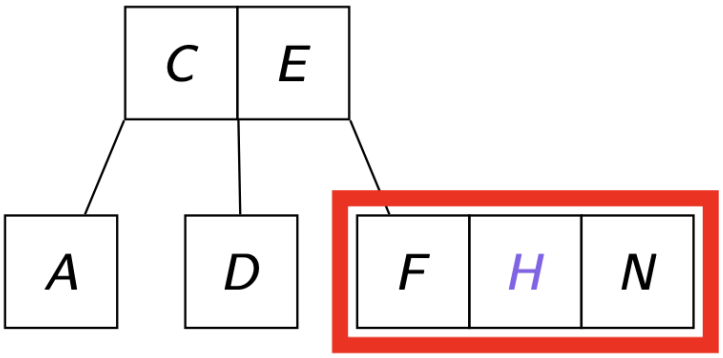


Insertar N

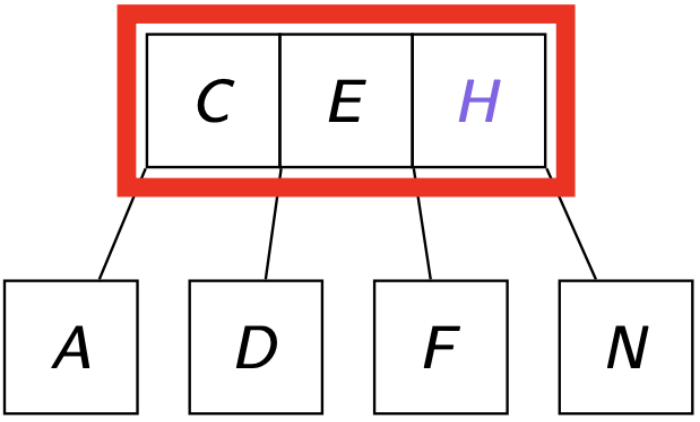


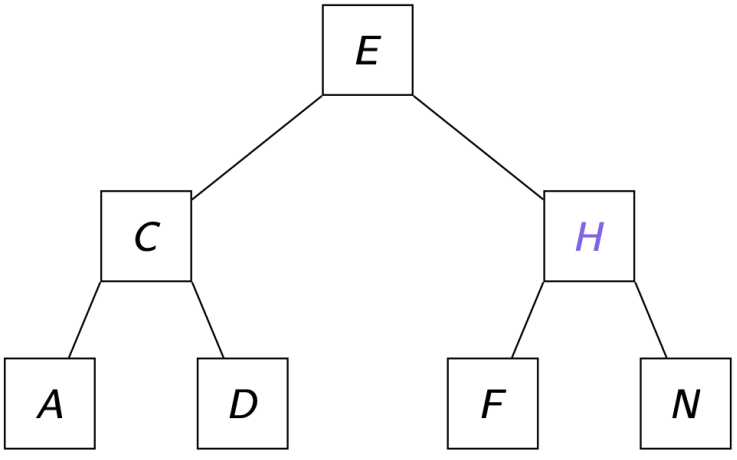


Insertamos F y H



Hora de crecer



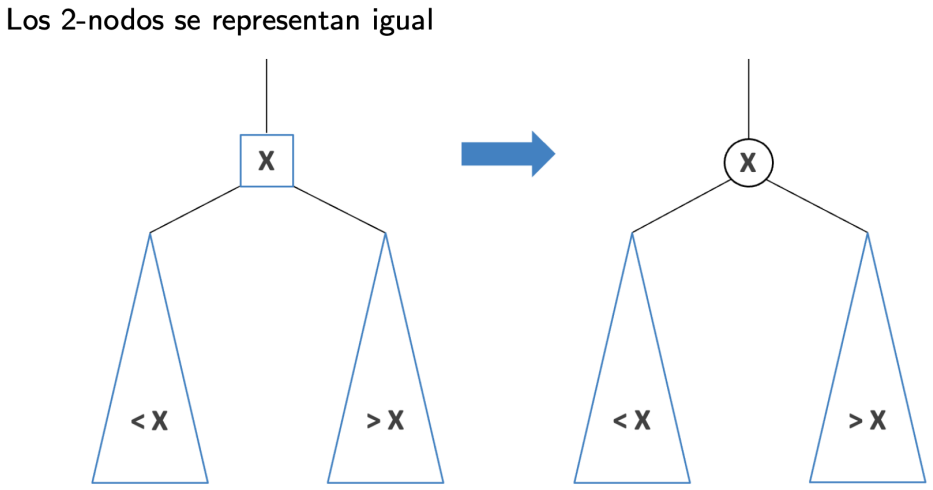


Ahora mezclaremos los AVL y los 2-3 para construir una EDD, mas simple y con la eficiencia de un 2-3

ARBOL ROJO NEGRO

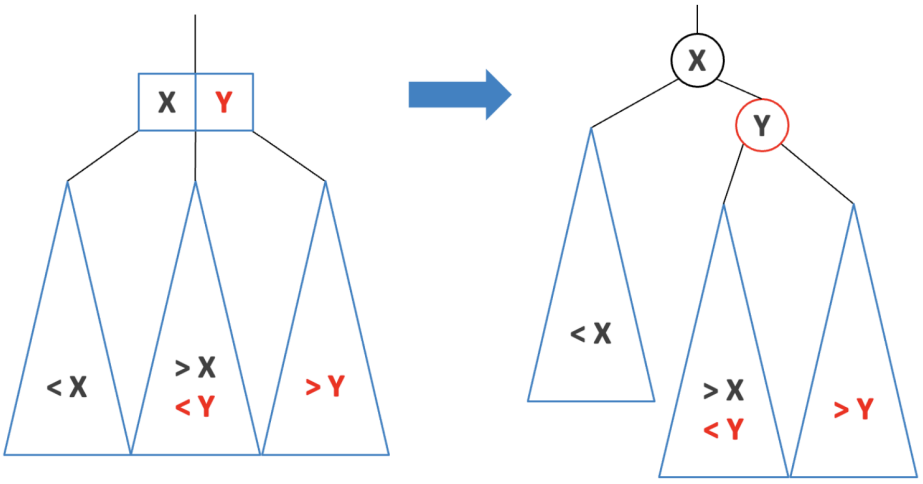
Para llevar esto a un Arbol binario de busqueda, vamos a representar los nodos 2, como un nodo binario, pero los nodos 3, los voy a representar como una jerarquía

Los nodos 2 → Facil



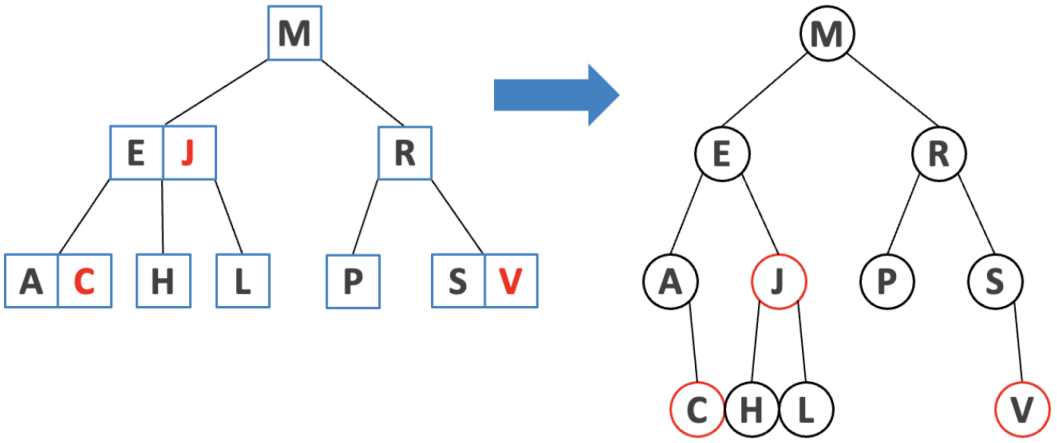
Los nodos 3 → Rotacion AVL

Reasignamos dos de los hijos a un nuevo nodo



NOTAR: La diferencia permitida es de 1

CONVIRTIENDO 2-3 en BINARIO



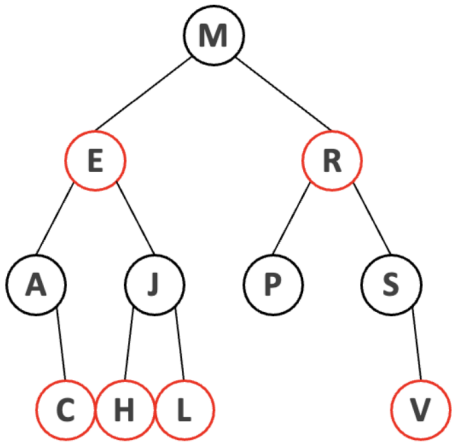
Definicion de ARBOL ROJO O NEGRO

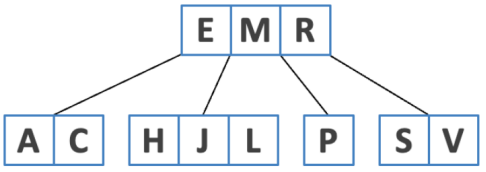
1. Cada nono o es rojo o negro!
2. La raiz del arbol es **negra → If raiz → NEGRO**
3. Si un nodo es rojo, sus hijos deben ser **negros**

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

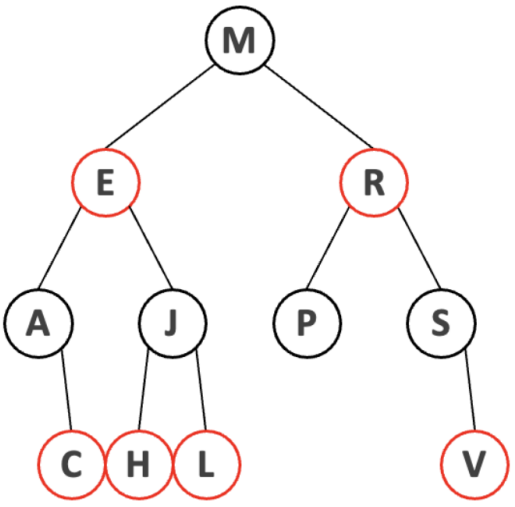
Propiedades → Anotar en los apuntes

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

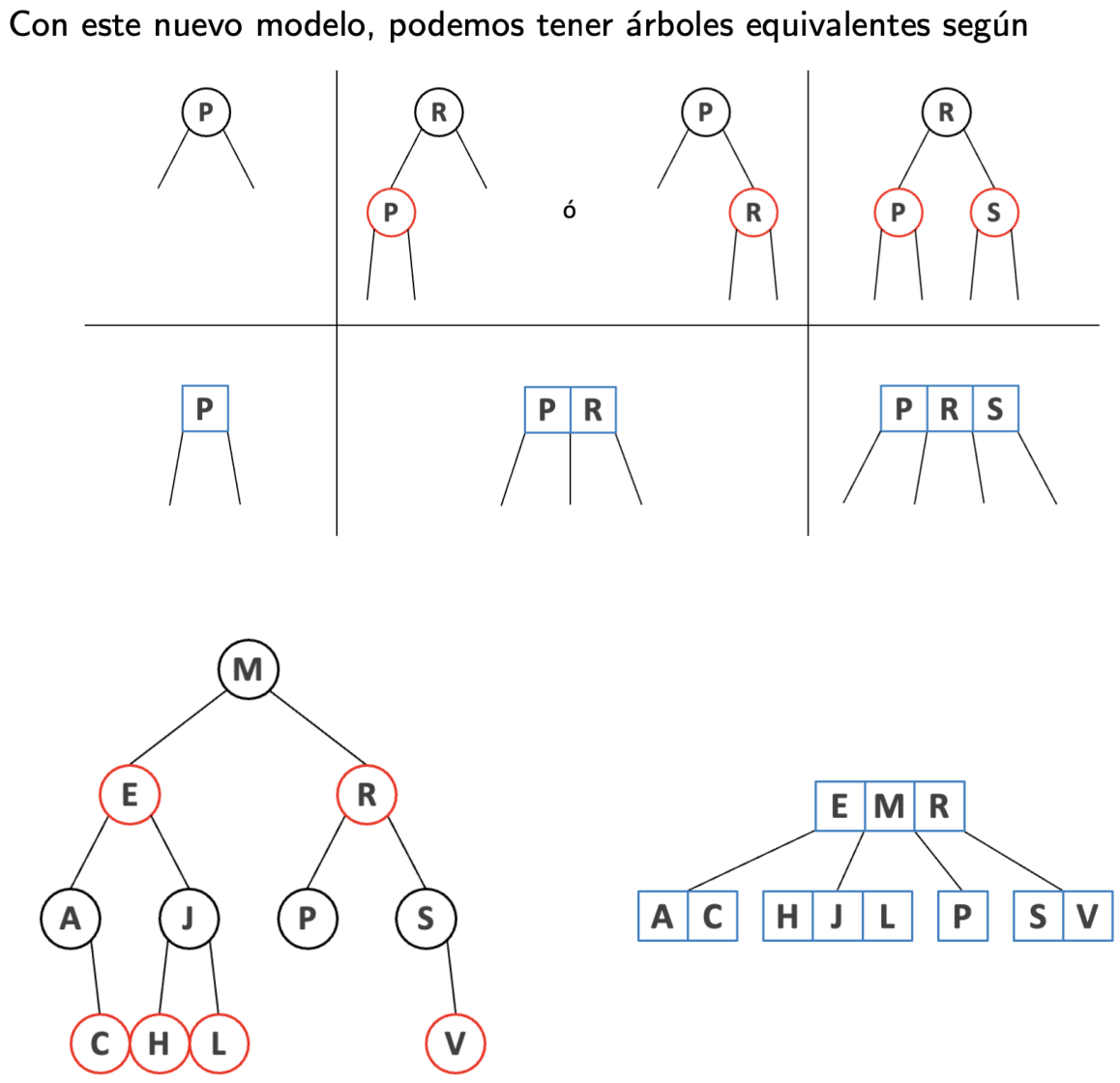
→ Con esto comprobamos si los nodos estan balanceados, porque estan al mismo nivel (del 2-4)

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



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:

CLASE 14

7/10/2024

Estamos viendo acceso al disco por ejemplo, con arboles 2-3, o un m-arbol podemos minimizar el tiempo de busqueda al disco

La siguiente estructura que veremos se suele usar en Base de datos, veremos como se aplicara

ARBOL B+

de orden d, es un arbol de **busqueda** que almacena pares (llave, valor) y cumple con

1. Los nodos internos solo guardan llaves
2. La raíz tiene entre 1 y 2d hijos
3. Los nodos internos tienen entre d y 2d hijos (la mitad o mas) → No puedo tener nodos vacios
4. Las hojas estan a la misma altura y guardan a lo mas 2d pares (llave, valor) **ordenadas por llave**
5. Las hojas estan conectadas formando una lista doblemente ligada → Esto optimiza las busquedas entre rangos

Los nodos de arriba pueden estar vacios, los intermedios medio llenos, y los otros nunca vacios

Los nodos del indice estan a la mitad o mas llenos, nos ayuda a controlar la altura, y no nos obliga a tener lleno todos los nodos intermedios

Hay 2 tipos de nodos, los datos solo las guardare en las hojas, las llaves las llamare desde arriba

```
    llave
      |
      |
llave, valor
```

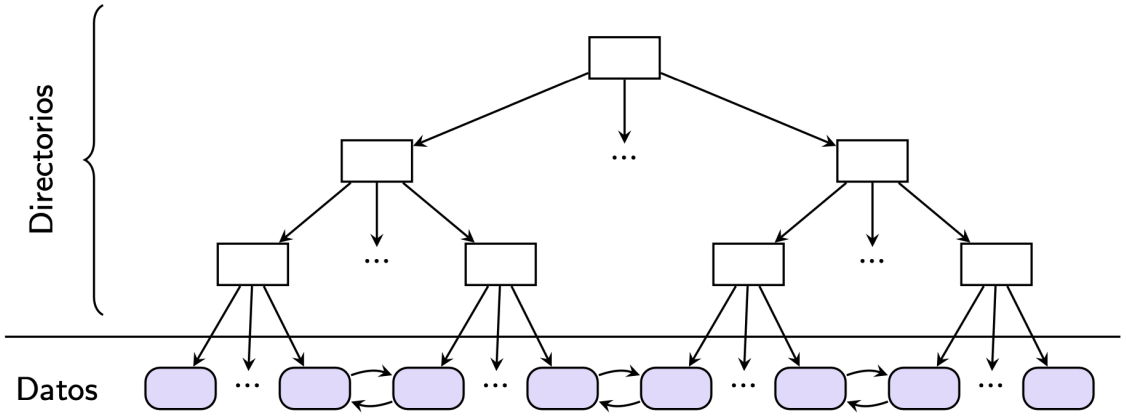
Observaciones

- Siempre esta balanceado.
- Mantiene la eficiencia de bu śqueda: O(log2d (#datos)) .

- Procedimientos eficientes de insertar/eliminar elementos.
- Todos los nodos tienen un uso mınimo del 50% (menos el root).

“B+-trees are by far the most important access path structure in databases and file systems”, Gray y Reuter (1993).

Distinguimos entre nodos que solo permiten navegar (directorios) y aquellos que contienen los pares (datos)

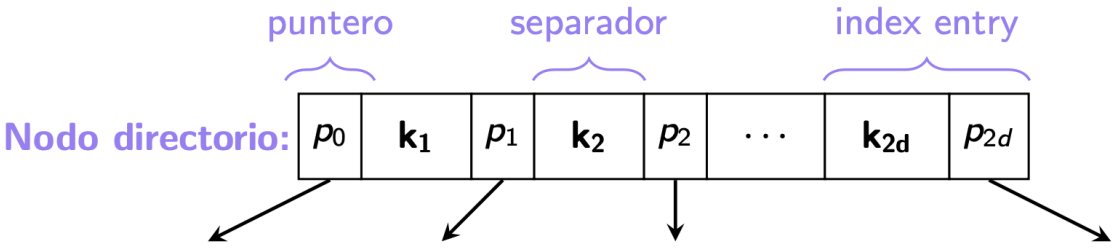


¿QUE ES EL INDICE?

Metodo de acceso que optimiza el acceso a los datos para una **consulta o conjunto de consultas** en particular

ej:

a. Indice de un libro
b. Orden alfabetico
c. Numero de paginas
d. Secciones de un diario

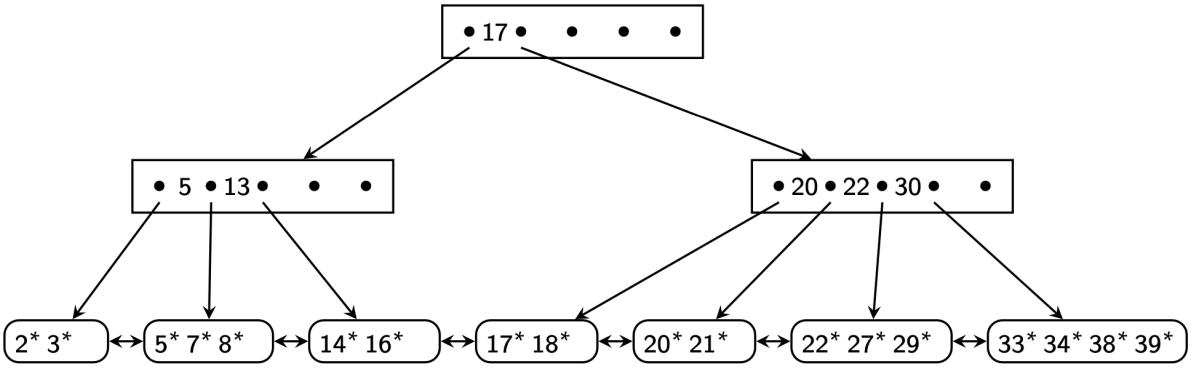


El separador es la llave → Nos dice si tenemos que ir por la izquierda o la derecha

El separador es la llave y el puntero

El nodo directorio se diferencia de las hojas, en que no tienen valores, los que estan almacenados abajo

Ojo: Pueden haber datos que estan arriba y no abajo y viceversa



La lista ligada permite moverse **desde** un punto de interés

Para nuestra grafica, el asterisco * indica que tiene la llave y el puntero al valor de esa llave?

Pregunta de prueba: Darse cuenta si un arbol esta o no cumpliendo (nos entregan uno dibujado)

Darse cuenta que los nodos intermedios de d-2d datos, que solo la raiz puede tener uno solo o llena, y que las hojas de abajo esten doblemente ligadas, y

que esten los punteros adecuadamente ordenados

No confundirse con que en el directorio hayan valores que abajo no esten, y que arriba si, y viceversa

El d se refiere al orden, es capaz de tener hasta 2d (llaves)

INSERCION

Estamos asumiendo que las lalves son unicas, todavia no veremos caso de llaves repetidas (tiene su forma de afrontarlas)

1. P = busquedaEnArbol(k, raiz)
2. Si la cantidad de llaves de P es menor a 2d, insertar k en P (fisicamente lo coloque ordenado)
3. En otro caso: debemos insertar k en P y hacer **SPLIT de [P+k]!**
    a. [p+k] = k1*k2*...k*2dk*2d+1

SPLIT

Para hacer **split** del nodo $[P + k]$ de tamaño $2d + 1$:

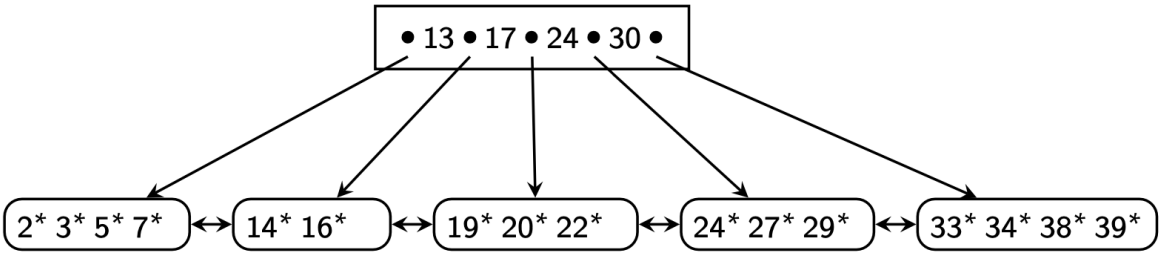
1. Asuma: $[P + k] = k_1^* \, k_2^* \ldots k_{2d}^* \, k_{2d+1}^*$ con $k_i < k_{i+1}$.
2. **Divida** $[P + k]$ en dos nodos:

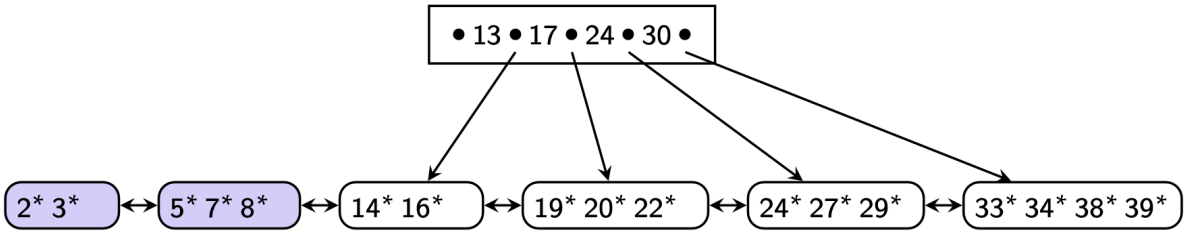
$$P_1 = k_1^* \ldots k_d^* \quad \text{y} \quad P_2 = k_{d+1}^* \ldots k_{2d+1}^*$$

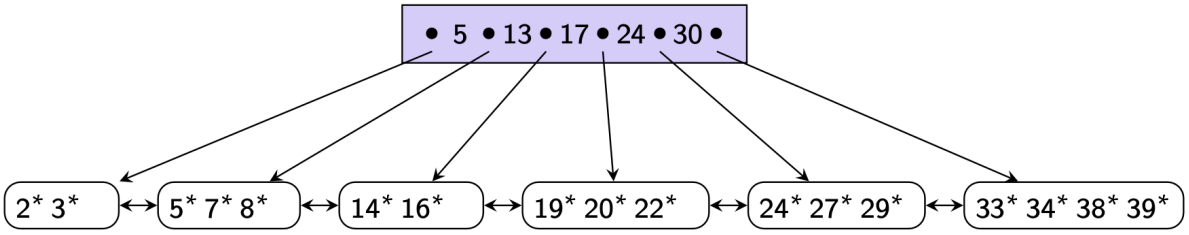
   y reemplace P por P_1, P_2 en la lista doble-ligada de los datos.
3. Seleccione el valor k_{d+1} como **divisor** de P_1 y P_2.
4. Reemplace el puntero p en el nodo P' que apuntaba a el nodo P por $p_1 \, k_{d+1} \, p_2$ donde p_1 apunta a P_1 y p_2 apunta a P_2.
5. Itere sobre el nodo de directorio P' que apuntaba a P (**split** de P').

EJEMPLO

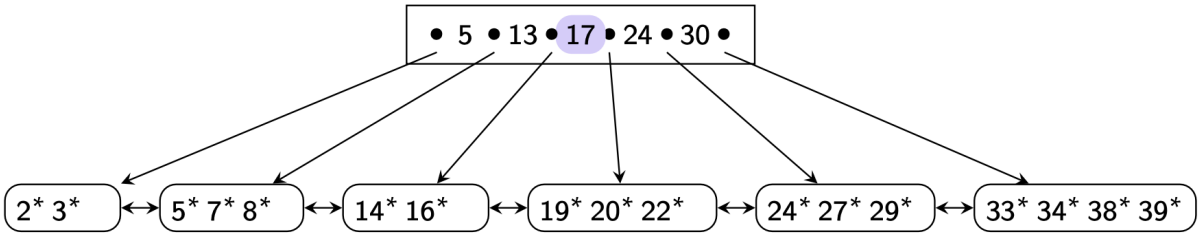
d = 2. 2d= 4 max

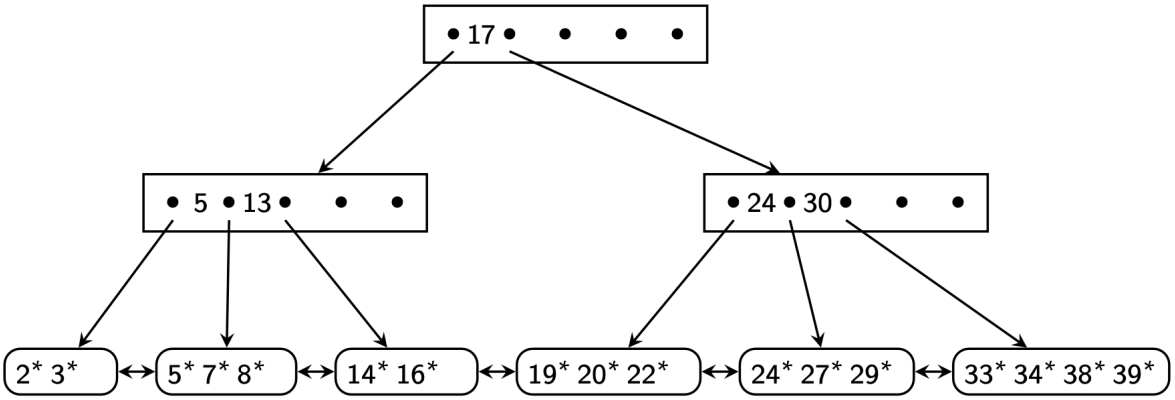






YA no se cumple la propiedad de B+ → SPLIT





Si hay mas orden → La altura del arbol va a ser menor → Busca mas entre las listas doble ligadas

La busqueda en el indice, depende del numero de niveles

La busqueda dentro del nodo O(d) o de O(2d), es mucho menor que o(n)

Conclusiones

- La altura del arbol crece, cuando se llena la raíz
- El nodo raız es el u ńico que se le permite estar lleno al menos del 50%.
-

CLASE 15

9/10/24

Resumen clase pasada

Inserciones

Para inserciones, podemos en vez de hacer split, mover hacia al hermano (si tenemos espacio), con esto el costo computacional se ahorra mucho

Para insertar:

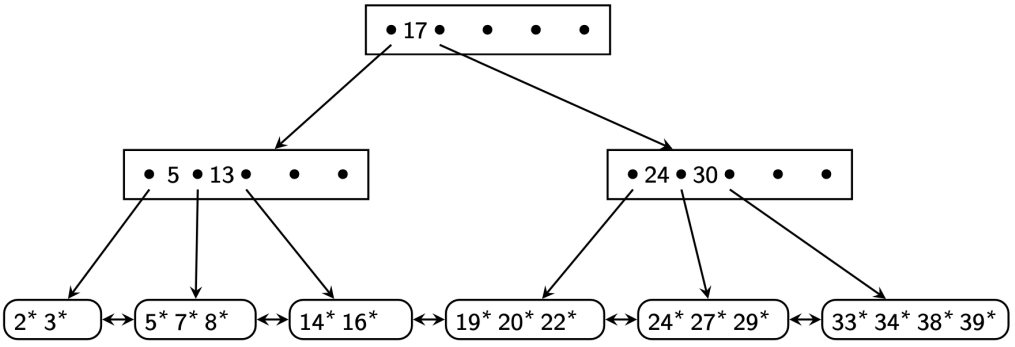
1. Buscar en el arbol → Desde la raiz a la hoja
2. Que tan lleno esta, si la cantidad de llaves es 2d, insertar en en donde se encontro
3. Si no, insertar en donde se encontro, y hacer split de [P+k] P→ Donde estamos k(valor a insertar)

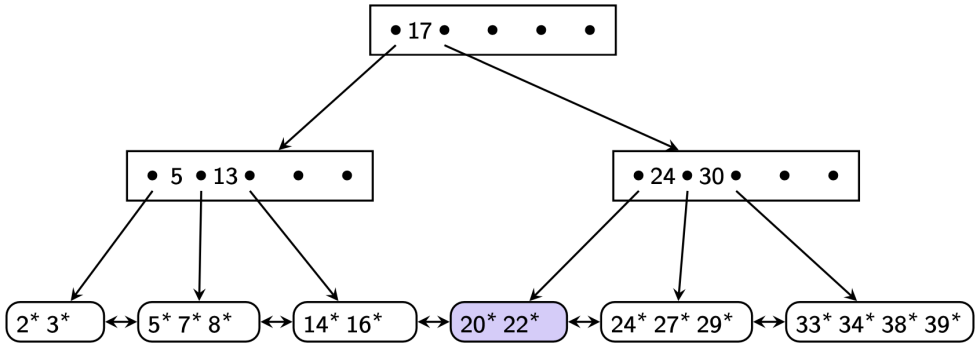
ELIMINACION

1. P = BusquedaenelArbol(k, root)
2. Si el espacio usado en P es mayor o igual a d+1, eliminar k de P
3. En otro caso: debemos eliminar k en P y rebalancear [P − k]

Ejemplo Eliminacion

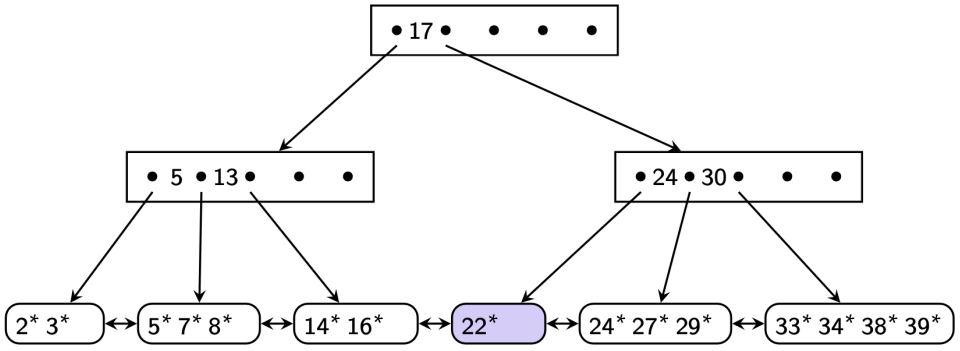
Para eliminar el elemento 19^*:



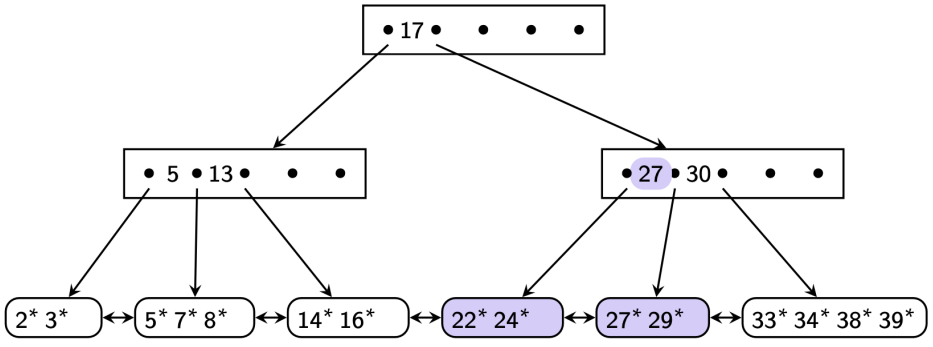


Podemos eliminar 19∗ sin problemas (#-data entries > 2).

Eliminar 20!



Debemos hacer redistribución o "unsplit" de las nodos.

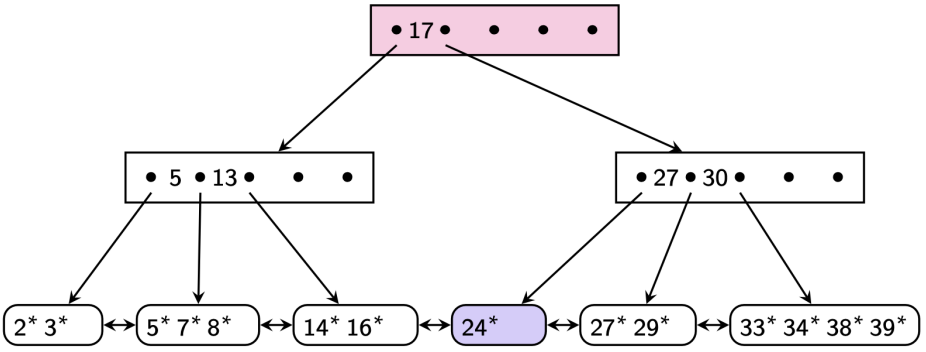


Debemos hacer redistribución o "unsplit" de las nodos.

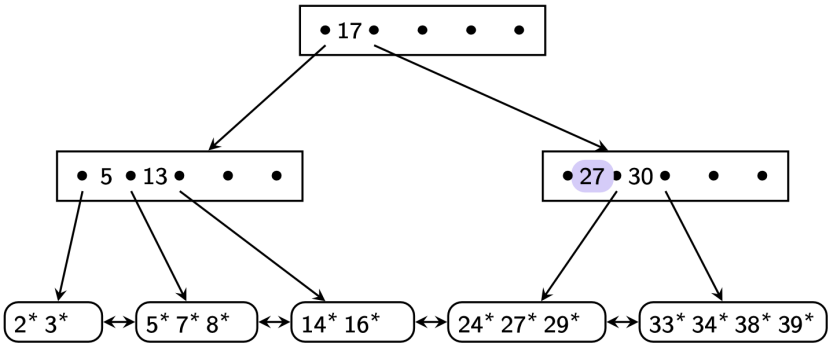
Notar que aqui en vez de subir, se cambio al hermano, y subimos la siguiente llave (27)

- Logica de pseudocódigo:
- Hacer un if, si podemos redistribuir → Hacerlo, si no hacer el siguiente unsplit

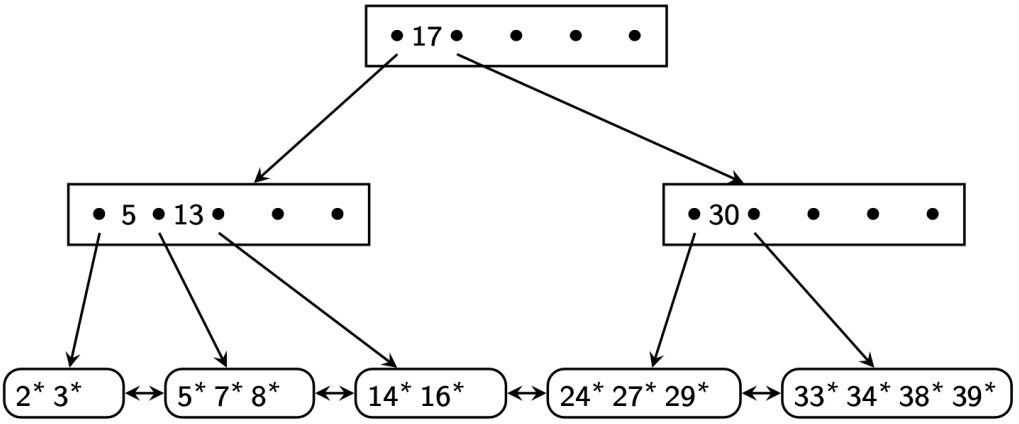
Eliminar 22



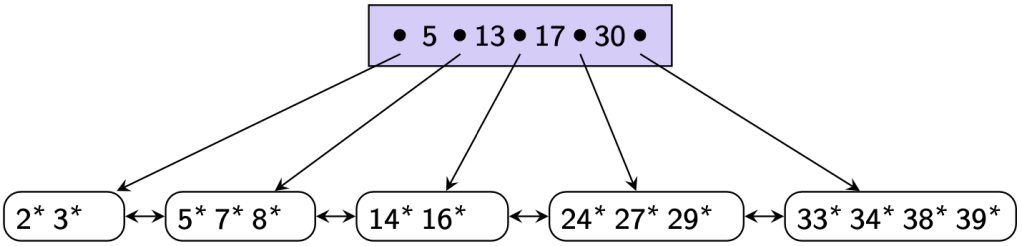
Necesitamos hacer un "unsplit" de los nodos.



Necesitamos hacer un "unsplit" de los nodos.



Y ahora finalmente para cumplor, Unsplit al directorio



LA UNICA MANERA de decrementar altura del arbol: Si el root solo tiene 1 elem y hacer unsplit

EFICIENCIA

Considere:

- n: el número de nodos.
- $2d$: cantidad (máxima) de datos en una hoja.

El costo de cada operación (en I/O):

$$\begin{aligned} \text{Búsqueda:} \quad & \mathcal{O}(\log_{2d}(n)) \\ \text{Insertar:} \quad & \mathcal{O}(\log_{2d}(n)) \\ \text{Eliminar:} \quad & \mathcal{O}(\log_{2d}(n)) \end{aligned}$$

Costo depende logaritmicamente en base d!

Suposicion: No hay duplicados, pero si los consideramos tenemos opciones

1. Usar paginas con **overflow**
2. data entries extendidos con una llave compuesta (k,id)
3. permitir duplicados y flexibilizar los intervalos del directorio:
   - ki ≤ k < ki+1 ⇒ ki ≤ k ≤ ki+1

OPTIMIZACION B+trees

1. **Compresión** de index keys (o directorio).
2. **Bulk** loading.

Bulk Loading: Quiero cargar una tabla completa, toma las hojas, y ordenalos al principio (con que ? : Por lo que queramos: ej: selection sort)

Tomamos la primera llave, hacemos un indice con 1 solo bloque

Hasta que se llene, que no haya un split hacia arriba

Vamos recorriendo las hojas ordenadas de menor a mayor, pero garantizando que solo hacemos split cuando los nodos de directorio se vayan llenando

DICIONARIOS

Recordatorio:

soporta operaciones de tipo:

Asociar un valor a una llave
Actualizar el valor asociado a una llave
Obtener el valor asociado a una llave

En ciertos casos, eliminar de la estructura una asociacio ń llave-valor

Los ABB fueron nuestras primera EDD que implementa Diccionarios

Hay una caracteristicas que nos da el arbol binario que no es un requisito de un diccionario

Y esta es el orden

Cuando implemento un Diccionario con un arbol, puedo pedir que me entregue todo la data ordenadamnete

Ej: Listame los valores ordenadamente

La propiedad de orden es importante → NEcesito un arbol

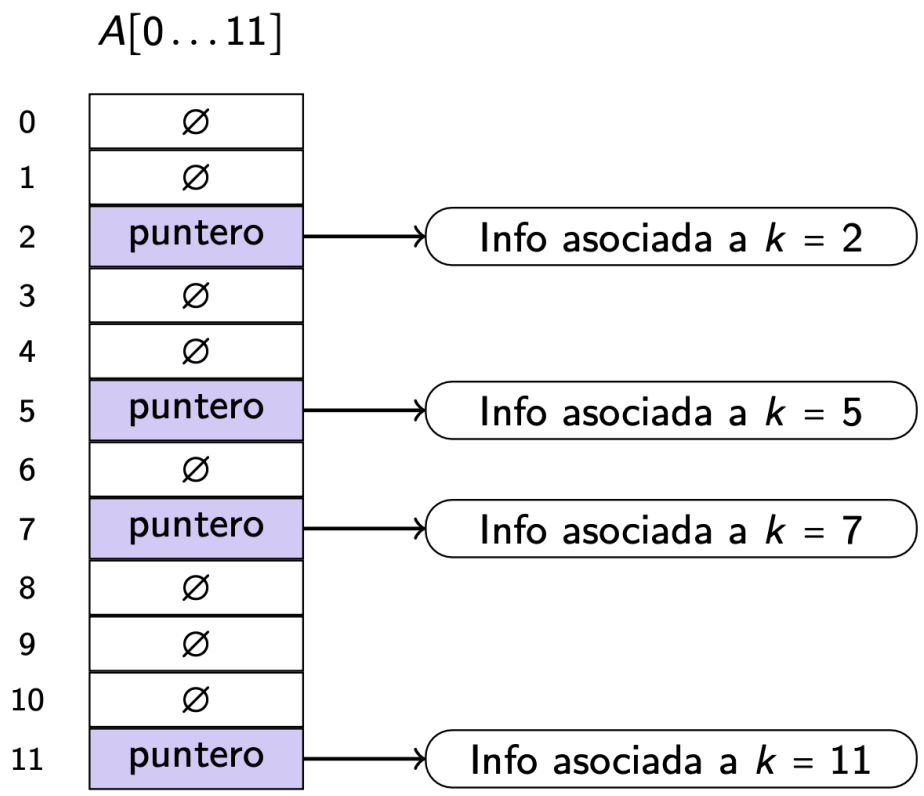
Si sacrifico el arbol → No necesito ordenar

Puedo al sacrificar el orden tener un posible **mejor desempeño**

Lo que podemo0s hacer es poner la llave en la posicion donde corresponde

Ejemplo, busco el valor 7 en la celda 7

Pongo un puntero (en la posicion 7), a donde esta el dato



INSERTAR: O(1) → Voy a la posicion e inserto

BUSCAR: O(1) → Voy a la posicion

Eliminar: O(1) → Lo quito de la posicion

El que tiene la llave tiene que ser un indice

Si mi rut es la llave, tendria un rango entre 0 y 20.000.000

El relacionar una llave, con los indices de un arreglo, es el problema para hacerlo eficiente

Lo que vamos a implementar → va a ser una funcion de HASHING

Con la funcion de hash, por ejemplo divide mi rut en 100.000 (queda con una posicion mas pequeña), pero esto nos genera **COLISIONES**

La funcion de hash, no nos da criterio de orden, pero nos da eficiencia en la insericion, busqueda y eliminacion

El problema define se requiere usar orden o no, si no lo necesitamos, lo mas eficiente podia ser una **TABLA DE HASH**

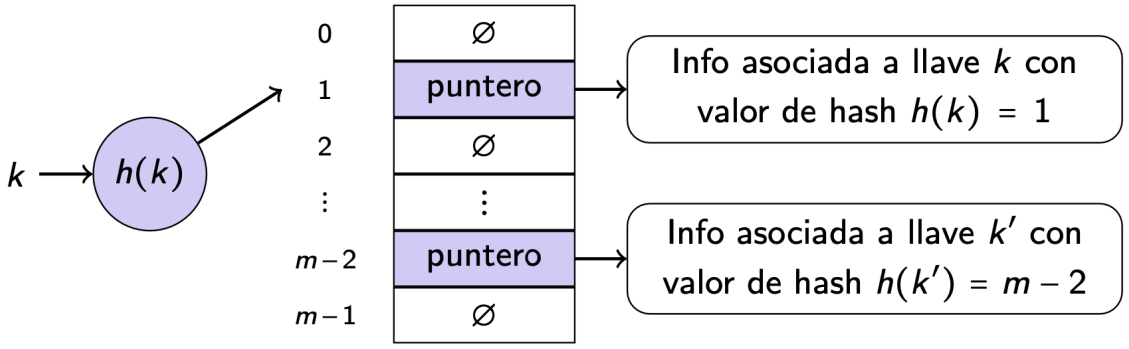
DEFINICION FUNCION HASH

Hash mapea un espacio de llaves a otro mas pequeño

Pero no necesariamente es inyectivo

No podria ser biyectiva

> Definición
>
> Dado $m > 0$ y un conjunto de llaves K, una **tabla de hash** A es una EDD que asocia valores a llaves indexadas usando una función de hash $h : K \to \{0, \ldots, m-1\}$. Diremos que tal A es de tamaño m.



Una funcion que transforme datos, en numeros!

Clase 16

14/10/24

Resumen clase pasada:

1. Hash → Perdemos orden de los arboles, pero ganamos velocidad de acceso O(1)
2. Funcion que nos permite transformar llaves en posiciones de un arreglo
3. Dado que anticipamos el Dominio (el rango de llaves) , es mayor que la cantidad de espacios en el array, (la funcion no iba a ser biyectiva) → COLISIONES
4. Distintas valores de llaves, pueden dar iguales posiciones en el arreglo
5. La forma mas intuitiva → Choque generar

Cuando tenemos valores de llave numeros (usamos divisiones)

Cuando sacamos el mod: se espera que distribuya uniforme

¿Si mis llaves no son numeros? → Transformar en numeros

h(k) = k mod m

m es la cantidad de posiciones en la tabla

si m es un numero primo, casi nunca me da el valor 0 (posicion 0)

Solo me dara cuando tenga una llave igual al tamaño del arreglo (extremo)

A esto se le conoce como **HASHING MODULAR**

Si los numeros de entrada son todos iguales → No va a distribuir uniforme

Todas las llaves con mismo resto → Generaran una **COLISION**

Ejemplo

Tomando $m = 100$ y $K = \{0, \ldots, 999\}$, el hashing modular cumple

- $h(12) = h(112) = \cdots = h(912) = 12$
- $h(18) = \cdots = h(918) = 18$

Ejemplo

Consideremos m = 7, insertamos llave 15 en la siguiente tabla de hash

15 mod 7 = 1

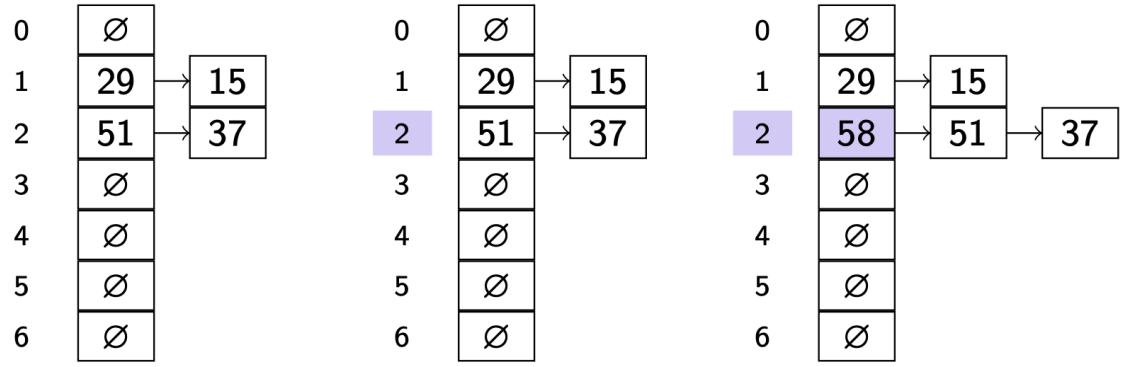
```
[ø, 15, ø, ø, ø, ø, ø, ø]

insertar 16 -> 16 mod 7 = 2

[ø, 15, 16, ø, ø, ø, ø, ø]
```

COLISIONES

Propuesta 1: Encadenamiento → Lista ligada (con puntero al final)



OPERACIONES

```
ChainedHashSearch (A,k):
    Buscar llave k en A[h(k)]

ChainedHashInsert (A,k,v):
    Insertar (k,v) como cabeza de A[h(k)]
```

```
ChainedHashDelete (A,k):
    Eliminar llave k de A[h(k)]
```

La complejidad de estas operaciones → **depende** de que tan largas sean las listas

Conclusion: Una **buena funcion de hash:** repartiria las llaves de manera mas o menos homogenea

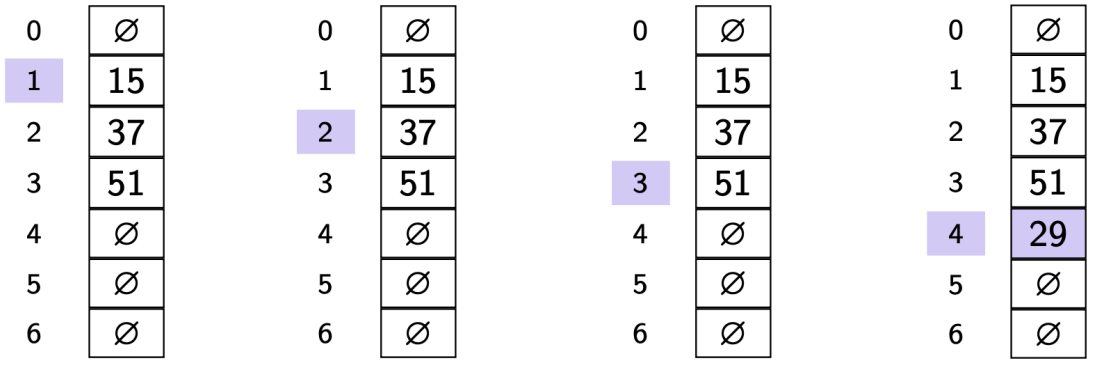
OTRA OPCION (Mala) → **SONDEO LINEAL**

Busca posiciones libre en la misma tabla (lsita ligada sin punteros)

Esta opcion por ejemplo, es para que cuando **NO PODAMOS USAR MEMORIA EXTRA,** insertemos en la misma tabla

Al insertar la llave 29 seguimos la misma idea del sondeo lineal

- Su valor de hash es $h(29) = 29 \bmod 7 = 1$



Su hash natural era 1, asi que busca la posicion desde el uno mas cercana hacia adelante

ESTA ESTRATEGIA NO ES RECOMENDADA CUANDO TENEMOS QUE ELIMINAR DEL DICT!

Que no es recomendable, no dice que no se pueda → IFs IFs

Si la tabla esta muy llena → **DEJA DE SER EFICIENTE**

MODIFICACIONES DE SONDEO

DOBLE HASHING

Usamos dos funciones de hash h1 y h2 y buscamos en

h1(k), h1(k) + h2(k), h1(k) + 2h2(k) , ...

Esto no es **REHASHING!**

Sondeo cuadratico : en caso de colisiones, buscar en

h(k) = H, buscamos en → H, H+1, H+4, H+9,...

FACTOR DE CARGA

Cuan llena esta la tabla

$\lambda = n / m$

n valores almacenados , m rtamaño de tabla

si $\lambda = 1$, en una tabla de hash en encadenamiento → No me dice nada

Pero si se que distribuy uniforme, las colisiones seran pequeñas, dejando mucho mas chico a O(n) buscando en lista

En sondeo lineal, me costara mas con una λ cercana a 1, parecera **O(n)**

RESUMEN

- En encadenamiento: $\lambda = 1$ es ACEPTABLE → (tabla funciona razonalmente O(1))
- En encadenamiento: $\lambda = 10$ → Muchas colisiones, para insertar y para buscar → Van a aumentar

La tabla va a empezar a funcionar mas LENTO

- Direccionamiento abierto (sondeo): $\lambda > 0.5$ resulta en inserciones y busquedas muy lentas

```
Pregunta de prueba: Me interesa el **SINTOMA**

Necesitamos control de acceso implementaado para la bibliotec

El sistema funcion increible, rapido, parejo, impecable

La universidad decide implementar el mismo sistema a todos lo

El sistema empieza a comportarse de manera erratica

Unas veces muy eficiente, y otras muy ineficiente

A un estudiante lo deja entrar bien, a otro no lo deja

Diagnostique y proponga una solucion al problema

Un arbol no podria ser, siempre es proporcional al logaritmo

Lo mas importante para saber que no es un arbol → No discrimi

Desconfio sobre la tabla de hash → Porque un estudiante esta

Intuyo que antes tenia una hash, con una lista con un hash de

Solucion: REHASHING
```

REHASHING

Si λ es grande y ya no es aceptable, las operaciones se vuelven costosas

Una solución es hacer **rehashing**

- Se crea una nueva tabla más grande
- Aproximadamente del doble del tamaño original
- Como el espacio de índices ya no es de tamaño m, se define una nueva función de hash
- Mover los datos a la nueva tabla

Esta es una operación costosa para tablas de hash

- Es $\mathcal{O}(n)$ para n datos insertados
- No obstante, es **infrecuente**

CLASE DE LA MUERTE 17

BACKTRACKING

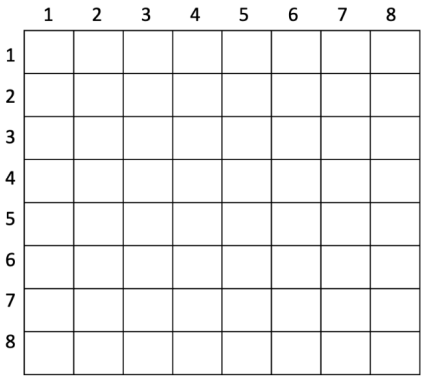
A diferencia de fuerza bruta que prueba todas las combinaciones posibles a un problema Csp

Problema Clasico

8 REINAS

Consideremos el problema de posicionar 8 reinas en un tablero de ajedrez de modo que no se ataquen

- Las reinas se desplazan por filas, columnas y diagonales
- Para lograr el objetivo: deben estar en columnas, filas y diagonales diferentes



Este problema tiene un conjunto de **restricciones**

1. Dos reinas no deben estar en la misma fila
2. Dos reinas no deben estar en la misma columna
3. Dos reinas no deben estar en un camino diagonal

La daigonal → Puedo calcularla con la pendiente! (si dos pendientes son iguales)

Cada una de las posiciones de la reina se define como un vector → Posicion de cada reina

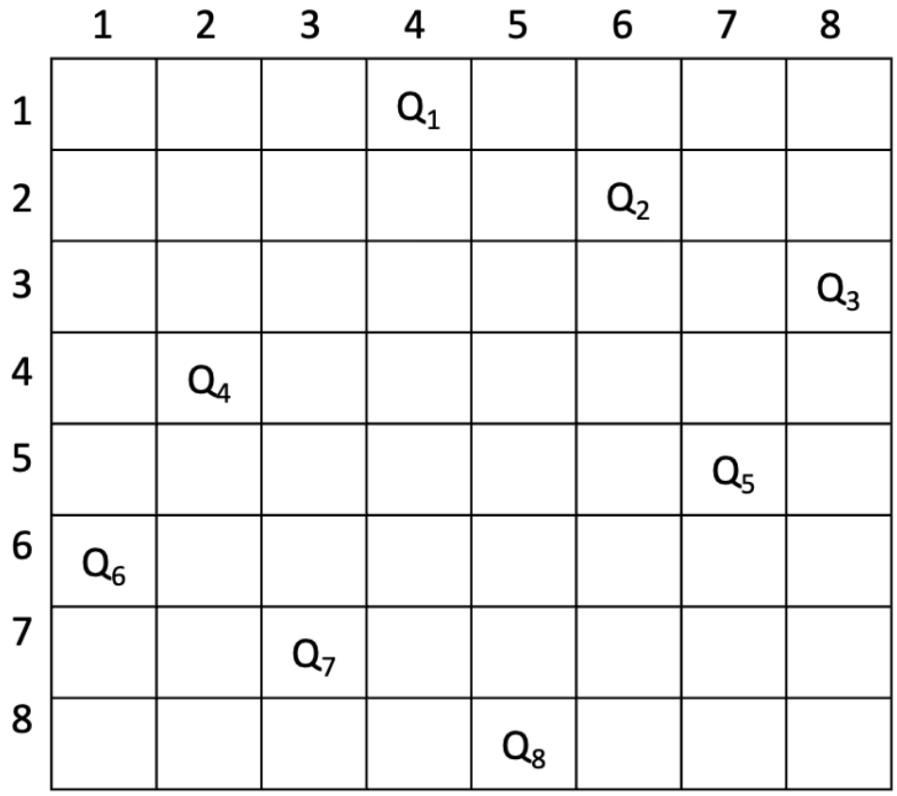
Vectores (x,y) (fila, columna)

- Cada fila y columna en el rango $1 \dots 8$
- Denotamos por x_i a la columna de la reina en la fila i
- Las posiciones de las 8 reinas se describe como un vector

$$(x_1, \dots, x_8)$$

¿Cómo sabemos si $(4, 6, 8, 2, 7, 1, 3, 5)$ es una solución?

Asi se veria (4,6,8,2,7,1,3,5)



Para evaluar que son, primero evaluar si son columans distintas, luego mirar las diagonales

El haber decidido que las reinas estan todas en filas distintas, simplifica el problema (no necesito verificar que estan en la misma fila), me simplifico la

RESTRICCION

CSP

Problema que tienen variables

Cada variable tiene dominio (cada dominio esta asociada a **CADA VARIABLE**)

Y restricciones asociadas a la configuracion del problea

Las variables gneralmente son independiente entre ellas, si fueran dependiente, quizas me esten sobrando variables

Las restricciones pueden evualuar **mas de una variable**

Definición

Un problema de satisfacción de restricciones o *constraint satisfaction problem* **(CSP)** es una tripleta (X, D, C) tal que

- $X = \{x_1, \ldots, x_n\}$ es un conjunto de **variables**
- $D = \{D_1, \ldots, D_n\}$ es un conjunto de **dominios** respectivos
- $C = \{C_1, \ldots, C_m\}$ es un conjunto de **restricciones**

donde cada restricción involucra un subconjunto de variables de X. Una **solución** es una asignación de las variables en sus dominios tal que se satisfacen todas las restricciones.

Observemos que

- No necesariamente las variables son del mismo dominio
- Una restricción C_j puede involucrar 1, 2 o más variables de X

EJEMPLO (8 reinas)

Ejemplo

El **problema de las 8 reinas** efectivamente es un CSP

Variables

- $X = \{x_1, \ldots, x_8\}$
- Cada variable x_i se interpreta como columna de la reina en la fila i

Dominios

- $D = \{B, \ldots, B\}$ con $B = \{1, \ldots, 8\}$
- En este caso el dominio de cada variable en X es el mismo

Restricciones

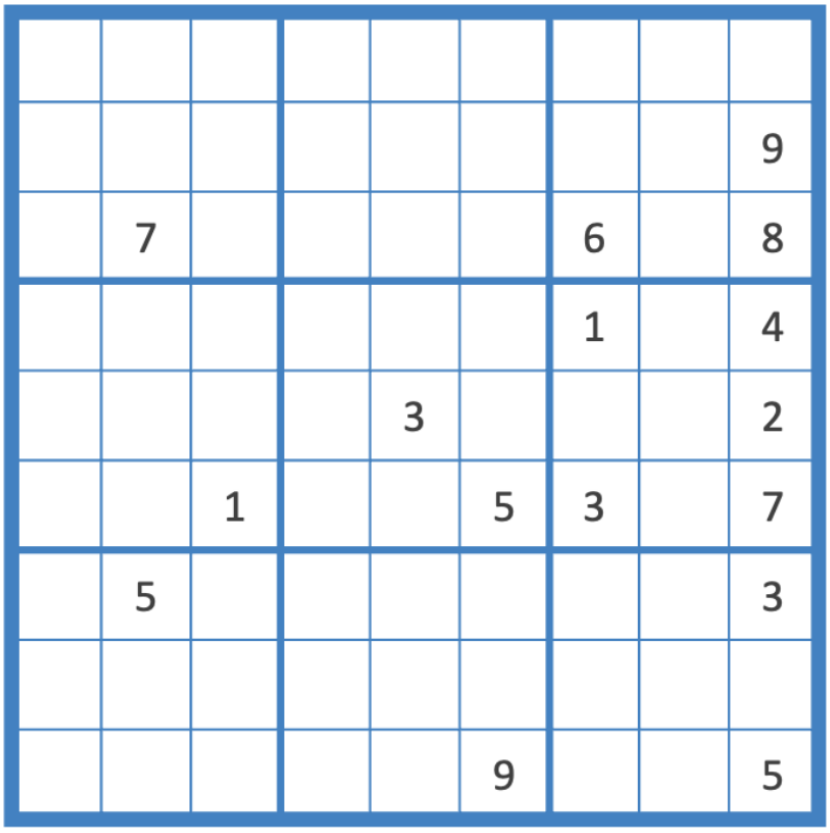
- La restricción sobre las filas está implícita en la elección de las variables
- Para las columnas: $i \neq j \rightarrow x_i \neq x_j$
- Para las diagonales: $i \neq j \rightarrow |(x_j - x_i)/(j - i)| = 1$

D es el dominio que cada una de las variables

La reina 1, se puede colocar desde la 1 a la 8

cada celda es la variable → el valor que puede ir ahi esta entre 1 y 9

SUDOKU



Cada celda es una variable

El dominio de cada variable es de 1 a 9

Restricciones: No puedo repetir un numero en la fila, columna, y cuadrante

- [ ] Estudiar Problemas SAT y NP-Completos

BACKTRACKING

Recordando que fuerza bruta, busca colocar todas las variables y al final preguntar si cumplen todas las restricciones

Bakctracking va colocando variables, y asignamos hasta encontrar una que no podamos asignar, que se sale del dominio → Nos devolvemos a la variable

anterior, si esta variable anteriores se sale tambien, hay que buscar a su anterior, asi recursivamente hasta que en una sea valida,

Vamos reubicando variables en sus dominios hasta que podamos ubicar correctamente

Queremos garantías sobre la existencia de soluciones

- Si el problema tiene solución, queremos saberlo
- Si no tiene, también queremos saberlo

Podemos responder recursivamente la pregunta

Dado un problema, ¿es posible resolverlo?

aprovechando que extender una asignación parcial

$$(x_1, \ldots, x_{k-1}) \to (x_1, \ldots, x_{k-1}, x_k)$$

genera una nueva instancia del problema

Hacemos *Backtracking* para la nueva instancia

Backtracking → Entrega solucion si existe, y si no existe me lo dice!

IDEA DE PSEUDOCODIGO (TIENE SOLUCION?)

input : Conjunto de variables sin asignar X, dominios D, restricciones R

`isSolvable`(X, D, R):

```
1  if X = ∅ : return true
2  x ← alguna variable de X
3  for v ∈ D_x :
4      if x = v no rompe R :
5          x ← v
6          if isSolvable(X − {x}, D, R) :
7              return true
8          x ← ∅
9  return false
```

CB: Todas las variables asignadas

Elegimos dominios

Evaluamos restricciones

Si no se rompen, asignamos valor

Y recursivamente asignamos mas

Abajo, retorna true si tiene solucion entra al if

Si el for, no logro asignar la variable, tiene que devolverse (va a retornar falso)

va a retornar falso → La variable anterior tendra que seguir probando con el dominio

El backtrack en el codigo, es cuando el if(solvable) no retorne true (no entra al if), asigna x en vacio, el for avanza y coloca el sigueinte del dominio!

Clase 18

21/10/24

Tip: Anotar la estrategia de backtracking plantilla y tambien porsiacasop el de las reinas, para saber cuando poner la condicion de llamada

→ IsSolve y Queens

tengo la plantilla de backtracling → Pero no se como validar las restricciones

Descomponer, hacer funciones, las pruebas permiten descomponer los ejercicios, en funciones, no es necesario meter todo un bloque de codigo

COMPLEJIDAD BACKTRACKING

En un conjunto de n variables X = {x1 .... xn}

Con valores posibles de dominio D = {D1,...,Dn}

tenemos $|D1| \times |D2| \times \ldots \times |Dn|$ tuplas posibles

Luego en el caso de $|Di| = K$ para todo i ,

revisar todas las tuplas es **O(K**n)**

La complejidad de las posibles soluciones para CSP cumplen,

- la estrategia de fuerza bruta revisa **todas las tuplas** $\mathcal{O}(K^n)$
- el backtracking puede revisar menos tuplas, pero sigue siendo proporcional $\mathcal{O}(K^n)$

Es decir, asintóticamente estas estrategias tienen la misma complejidad

¿Cuál es más rápido en la práctica?

No olvidar: *Backtracking* es igual o más rápido que la fuerza bruta

Como va evaluando restricciones, va podando todas las tuplas posibles, le gana a fuerza a bruta aunque son proporcionales

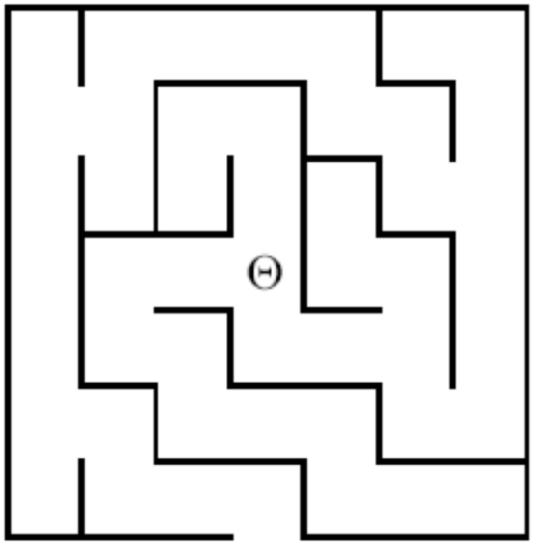
OTROS PROBLEMAS (Interpretaciones)

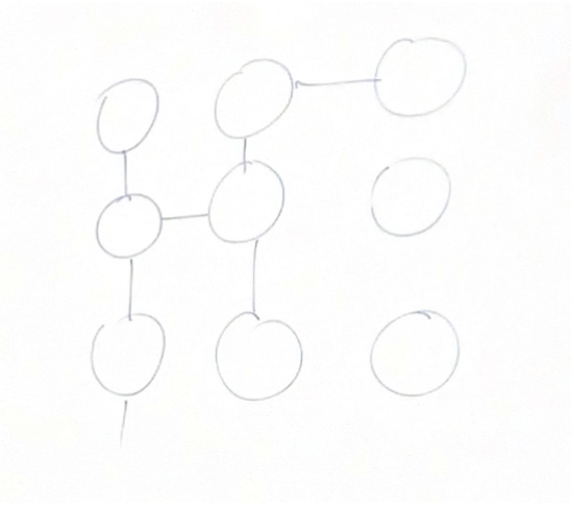
Backtracking se puede interpretar como **Busqueda en grafos implicitos**

Los CSP generan muchas tuplas posibles como asignaciones para las variables de X

- Cada posible asignacion genera un camino
- Las nuevas asignaciones abren nuevos caminos
- A la coleccion de todas estas alternativas le llamamos grafo implıcito

Ej: El caso canonico es resolver un **Laberinto**





Variable: El nodo que estoy recorriendo es valido

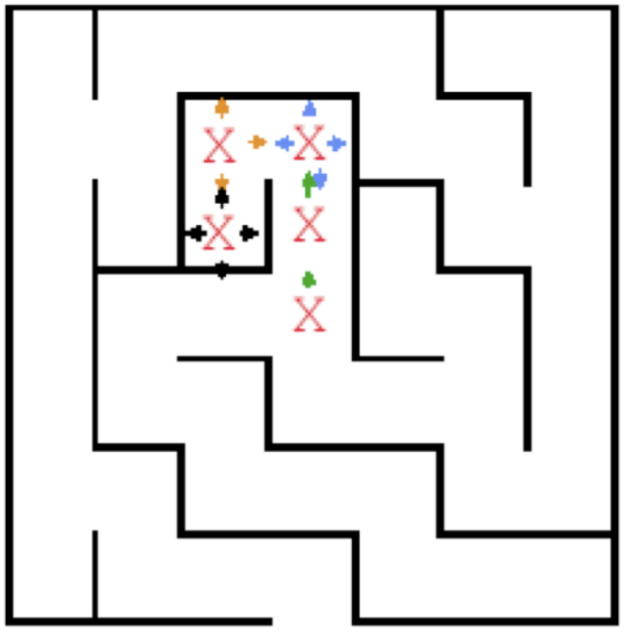
O si es un camino valido

Saber

Dominio: hacia donde avanzar (direcciones)

Restricciones: Poder avanzar! No poder avanzar

No haber recorrido ese camino ej: Nodo al norte ya lo visite, (fui para alla y no logre salir)



En este caso, terminamos el for del dominio, y nos damos cuenta que no llegamos a la salida, con esto, nos damos cuenta que no es parte de la solución

No nos quedan direcciones posibles, nos devolvemos al anterior

Nos devolvemos a la anterior, avanzamos ya hacia el norte, no podemos avanzar izq y derecha,

Nos devolvemos al anterior

... Asi hasta tener otra direccion posible, buscando la solucion haciendo el recorrido hasta llegar a la solucion

Ahora en codigo, en nuestra plantilla de backtracking

Condicion de exito : La posicion en que estoy es la salida

if x= celda salida → Termine

Si no, la celda donde estoy, tengo que recorrer al norte al sur al este y al oeste

Si al recorrer esto, en alguna de ellas no rompo las restricciones

Logro avanzar, sigo avanzando en ese camino

Problemas tipo de backtracking

1. Encontrar la solución a un problema
2. Encontrar la mejor solucion (camino mas corto) → Tengo que comparar caminos para saber cual es el mas corto

DFS no es backtracking, backtracking es CSP que se puede usar para recorrer grafos, es el mas eficiente? no , para eso tenemos estrategias para grafos

SOLuCION

Le agregamos etiquetas a las posiciones, de modo que sabemos cuáles hemos visitado (visited). Todas comienzan como nonvisited y la salida se marca como exit

input : Conjunto de variables sin asignar X, posición x, dominios D, restricciones R

`isSolvable`(X, x, D, R):

```
1   if x = exit : return true
2   if x = visited : return false
3   x ← visited
4   for v ∈ {N, E, S, W} :
5       if x + v ≠ wall :
6           x ← x + v
7           if isSolvable(X, x, D, R) :
8               return true
9   x ← nonvisited
10  return false
```

Con esto, puedo darme cuenta que no hay solucion

Voy asignando las posiciones que visito, como visitada, ya con esto marco si me devuelvo por donde no ir

x + direccion (no es muralla) → Entonces es valida

Le asigno la variable al ir en esa direccion

ejemplo de prueba:

Modifique este algoritmo, para enfcontrar todas las soluciones a este laberinto

Modifique para encontrar al mejor

PROBLEMAS BACKTRACK 2

- Recorrido del caballo de ajedrez
- Problemas de la **mochila (Capacidad versus n° items)**
- Problema de carga
- Coloreo de mapas

- [ ] Copiar materia
- [ ] Ejercitar problemas de backtracking

Pregunta prueba: Variable, descripcion dominio y restricciones

Note que, o podemos observar, o llama la anteción

Para colorear mapas → El especialista prefiere evitar que el rojo quede al lado de el verde (para no molestar a los daltonicos)

Mejoras del problema para que el espacio de búsqueda sea menor : **PODAS**

MEJOR VERSION DE BACKTRACKING

Podemos incorporar estas mejoras según convenga en un problema particular

```
isSolvable(X, D, R):
    if X = ∅ : return true
    x ← la mejor variable de X
    for v ∈ D_x de mejor a peor :
        if x = v no rompe R :
            x ← v, propagar
            if isSolvable(X − {x}, D, R) :
                return true
            x ← ∅, propagar
    return false
```

Esta es la mejor a tener en los apuntes!!!

CLASE 19

23/10/24

Porque queremos algoritmos codiciosos?

Algoritmo Codicioso, ambisioso, grreedy

Estoy tan convencido que mi estrategia es buena, que la voy a usar sin mirar atras

Me quedo con la jugada hasta el final

En el caso de problemas de optimizacion, habitualmente una estrategia codiciciosa es mas rapida que backtracking

Mi estrategia podria ser hasta lineal

“Eligo de la mejor forma “ → O(n)

En otros casos de estrategia codiciosa

Cuando backtracking definitivamente falla → cuando los valores posibles de las variables son infinitos (o reales)

Si en el problema de la mochila, lleno con fracciones y no con enteros → Se me cae

En tiempo eficiente, no podre probar todos los valores posibles, si busco la mas optima menos aun, backtracking deja de ser buena idea

Estrategia codiciosa: Eligir mejor variable → Su valor → Jugarmela con esa

---

Pregunta de prueba: Evaluar si una estrartegia es o no valida para codicioso → Normalmente es una contradiccion

Elegiras siempre el de mayor peso, pero en realidad esto me podría saturar la mochila antes,

Podrian preguntarnos por eso, de manera contradictoria

---

5 Tipos de problemas en Computacion

1. Problemas de desicion: **SI O NO**
2. Problemas de **Busqueda**
3. Problemas de Conteo : Backtracking por ejemplo nos permite encontrar todas las soluciones al tablero de las reinas (y contarlas!)
4. Problemas de Optimizacion: **Mejor Solucion →** Backtracking generalmente no es eficiente aqui, dado que las buscara todas para compararlas entre ellas, una mejor estrategia seria (lo veremos)
5. Problemas de función → Soluciopn explicita

Ejemplo

Considere el **problema de la mochila con objetos fraccionables**.

Tenemos n objetos y una mochila

- Los objetos tienen pesos $\{w_1, \ldots, w_n\}$
- Los objetos tienen ganancias por unidad de peso $\{p_1, \ldots, p_n\}$
- La mochila tiene una capacidad m, en peso
- Incluir una fracción x_k del objeto k proporciona ganancia $p_k x_k$

Backtracking no es una muy buena estrategia para ello

Analisemoslo como un problema de **Optmizacion**

Ejemplo

Interesa llenar la mochila cumpliendo tres condiciones

- Queremos maximizar la ganancia total

$$\sum_{k=1}^{n} p_k x_k$$

  Esta es la **función objetivo.**

- No podemos exceder la capacidad de la mochila

$$\sum_{k=1}^{n} w_k x_k \leq m$$

- Las fracciones deben cumplir

$$0 \leq x_k \leq 1, \quad 1 \leq k \leq n$$

Interesa llenar la mochila cumpliendo tres condiciones

- Maximizar función objetivo $\sum_{k=1}^{n} p_k x_k$
- No podemos exceder la capacidad $\sum_{k=1}^{n} w_k x_k \leq m$
- Las fracciones deben cumplir $0 \leq x_k \leq 1$ para $1 \leq k \leq n$

Una **solución factible** es $\{x_1, \ldots, x_n\}$ que cumple las dos últimas condiciones. Una **solución óptima** es una solución factible que maximiza la función objetivo.

Simplex: Estar en un nodo, y avanzar por el de mayor curva de crecimiento

En palabras simples, sigo el vertice con mejor valor (mayor precio por kilo o por unidad)

¿Que estrategia usamos entonces? → Yo quiero determinar como maximizar el valor, quiero ver de que forma crece mas rapida

Simplex: Toma un punto y busca al que crece mas rapido

Para buscar al optimo global, sigo buscando la curva de los maximos

Estoy encontrando las variables S

busco Conjuntos que me resuelvan el problema

(valores validos)

Voy a buscar el mejor, de la forma en que eligo las variables

Los **algoritmos codiciosos** plantean una estrategia algorítmica basada en el paradigma de subconjuntos

1. Tenemos conjunto $S = \{s_1, \ldots, s_n\}$ con n inputs
2. Queremos un subconjunto $S' \subseteq S$ que satisfaga restricciones
3. Queremos solución factible que maximice o minimice una **función objetivo**

Un subconjunto S' que cumple las restricciones se llama **factible**. Una solución que maximiza/minimiza se llama **óptima**

Esta es una concepción teórica
Veremos su aplicación a problemas concretos

ESTRATEGIA CODICIOSA

Los **algoritmos codiciosos** trabajan en etapas

- Consideran un input a la vez
- Una vez que se decide sobre un input, la decisión **es final**
- Ninguna decisión posterior cambia la actual

En el caso del problema de la mochila

- Se trata de seleccionar un subconjunto de objetos
- Y determinar la fracción x_k

Soluciuon Parcial es cuando S' aun no evaluado todas las variables que tiene adentro

En backtracking hablamos de podas, propagacion y heuristicas

Si con heuristica, podemos elegir la mejor variables podemos llegar a una estrategia lineal O(n)

En greedy es eso, construimos una solucion en funcion de elegir las mejores variables

Para lograr este objetivo, debemos considerar los input en cierto orden

- Se usa un **procedimiento de selección**
- Si la inclusión del próximo input en la solución óptima parcial produce solución infactible, no lo consideramos
- En otro caso, el input se agrega a la solución

¿Qué diferencia hay con backtracking?

Una mala estrategia codiciosa, nos dejara pegado en un optimo local, pero una buena al optimo global

Diferencias con backtracking → 2 Dif principales

1. Toma desiciones y no se arrepiente
2. Me permite evaluar dominios (variables) REALES (continuas) y no solo discretas (backtracking)

EJEMPLO

Ejemplo

Para el problema de la mochila, podemos considerar las siguientes estrategias codiciosas *para la etapa actual del algoritmo*

- Incluir el objeto con mayor ganancia
- Incluir el con menor peso
- Incluir el que tenga mayor cuociente ganancia/peso

Normalmente, la mayoría de las estrategias no producen soluciones óptimas

Ejemplo

Considere la siguiente instancia del problema de la mochila con $m = 20$ y $n = 3$

- pesos $w_1 = 18,\ w_2 = 15,\ w_3 = 10$
- ganancias $p_1 = 25,\ p_2 = 24,\ p_3 = 15$

Algunas soluciones **factibles**

estrategia	x_1	x_2	x_3	$\sum_{k=1}^{n} w_k x_k$	$\sum_{k=1}^{n} p_k x_k$
mayor ganancia	1	2/15	0	20	28.2
menor peso	0	2/3	1	20	31
mayor ganancia/peso	0	1	1/2	20	31.5

En este problema, el óptimo se encuentra privilegiando ganancia/peso

☐ Estudiar este ejemplo

Si en la prueba, nos piden probar que la estrategia no funciona

Nos basta con elegir un **N pequeño, NO COMPLICARSE LA VIDA**

En este caso, la estrategia precio/peso es mejor que las otras

Demostracion de que no es valida → Encontrar una mejor

ALGORITMO GREEDY

input : Arreglo de inputs $A[0 \ldots n-1]$, cantidad de inputs n

```
Greedy(A, n):
1     S ← ∅
2     for i = 1 ... n :
3         x ← Select(A)
4         if Feasible(S, x) :
5             S ← Union(S, x)
6     return S
```

Es el esqueleto, no siempre es igual → Dependerá de cada problema

Select(A), es el primero? → No, es la mejor variable, segun el algoritmo codicioso

Aplico el Control → Cumple con las restricciones (Feasible)

Si chequeo → Agrego al conjunto soluciones,

Cuantas veces → N veces por tener N variables

COMPLEJIDAD: O(n) veces "Teoricamente" Quizas si se implementa una tabla hash, es mas rapido saber el criterio de mejor oheap → logn

PROBLEMA DE LA MOCHILA

A → Elegimos los elementos

x ← Lentejas (lo mas rico)

IS Feasible() → Pusimos todo lo que pudimos de lentejas

Avanzar al siguiente elemento

Problema de prueba SEGURO: Modifique el algoritmo de codiciosa, para probar varias estrategias y ver cual es mas EFICIENTE

DIJO EL PROFE EN CLASES QUE SI CAE EN LA PRUEBA SE EXIME TOTALMENTE DE HABERLO DICHO

CLASE 20

28/10/24

PROGRAMACION DINAMICA

Resumen Clase anterior e introduccion

Backtracking es lindo pero sigue siendo lento

Vimos problemas de optimizacion, menor costo, mayor ganancia, etc etc

Dentro de estas estrategias la clase pasada vimos estrategias codiciosas Greedy

No todas estas soluciones greedy nos llevan a una solucion optima global, puede llevarnos a una optima local

Lo que tiene en comun con backtracking es la Heuristica, si siempre elegimos la mejor variable con su mejor valor, backtracking seria codiciosa

En greedy una vez que tomamos un valor no nos arrepentimos, en backtracking si podemos devolvernos hasta encontrar una mejor solucion

Dependemos que los problemas tengan una estrategia codiciosa (no todos los problemas se pueden resolver con esto)

El problema de dar vuelto con monedas de distinto valor → Tiene estrategia codiciosa para ciertos tipos de valores de monedas (100 50 10 1), peri si tenemos monedas de (75 50 20) no podemos usar estrategia codiciosa → Para el mismo problema → Distinta data

Programacion dinamica es una mezcla entre **DIVIDIR CONQUISTAR y BACKTRACKING**

Dividimos el problema en partes, y evaluar si la mitad de un lado es mejor que la de otra, no se va a quedar con uno o con otro, seguira bajando hasta construir la solucion completa

Si lo pensamos como el backtrack puro, se ve igual de costoso

En esta estrategia es mucho mas **eficiente** que backtracking que hasta

PUEDE SER LINEAL O(n)

Problema de los vehiculos subidos arriba de la barcaza → Que no se caigan, que no se pase el peso

Pero ahora pensemos que cada ticket por auto tiene un valor

Ahora mi estrategia codiciosa para meter la mayor cantidad de autos puede que no sirva , tendriamps que buscar otra estrategia

Progra dinamica: Tomo el primero, evaluo con y sin el segundo, con y sin el tercero, con y sin hasta el ultimo, cuando ya no se puedan subir

Cual es el valor final? → Minimo 0 ningun auto

Hacia arriba → El ticket con este auto es solcuion

Con este tambien → Al final tenremos un arbol de **posibilidades**

Si bien voy bajando por ciertos lados, hay configuraciones que e repiten

Tendre la config de cierto auto repetido, Progra dinamica se va a acordar de el calculo que ya hizo !! → No calcula denuevo lo que ya hizo

Cada vez que necesite calcular en esstricto rigor, ahi recien voy a calcular

Si planteo bien el problema, voy a calcular solo 1 vez el valor de las variables

Con esto podríamos dejarnos O(N)

Problema charlas clase pasada

Si la haciamos sin valores, escogiamos la que terminaba primero

Y eso nos da la mayor cantidad de charlas posibles

Pero con valores, escogeremos las mismas probablemente, pero el valor no va a ser el mismo, podemos ver que si lo hacemos con estrateg codiciosa, puede que no sea la mas optima

Consideremos el problema de asignar charlas en una misma sala

- Tenemos n charlas por asignar
- La charla i tiene hora de inicio s_i y de término f_i
- Es decir, se define el intervalo de tiempo $[s_i, f_i)$

Solo se puede realizar **una charla a la vez**

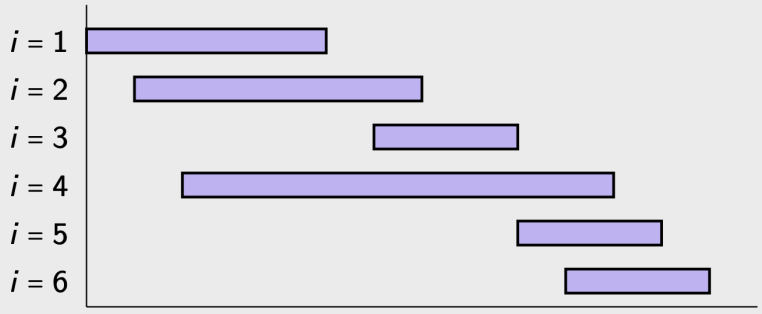
ADEMÁS: si la charla i es realizada, produce una ganancia v_i

¿Qué charlas asignar de manera que maximicemos la ganancia?

Ejemplo

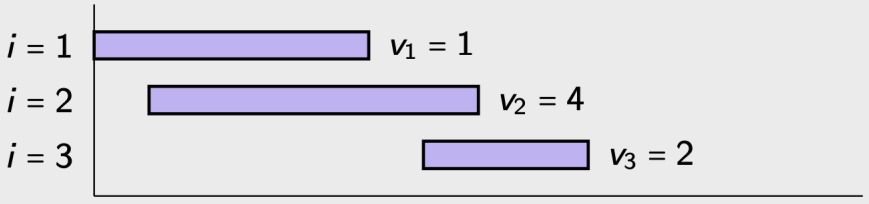
Sean las siguientes charlas con sus intervalos y ganancias

- $i = 1$, $[0,5)$, $v_1 = 2$
- $i = 2$, $[1,7)$, $v_2 = 4$
- $i = 3$, $[6,9)$, $v_3 = 4$
- $i = 4$, $[2,11)$, $v_4 = 7$
- $i = 5$, $[9,12)$, $v_5 = 2$
- $i = 6$, $[10,13)$, $v_6 = 1$



Ejemplo

Podemos pensar en una instancia del problema de forma que la estrategia codiciosa mencionada **no funciona**



En este caso,

estrategia	charlas	ganancia
codiciosa	$\{1,3\}$	3
óptima	$\{2\}$	4

Nuestra estrategia codiciosa no es óptima
en el caso general del problema con ganancias

A destacar que la solucion no es optima para Estrategias Codiciosas!!!

PROGRA DINAMICA

- Se usan en problemas de optimizacion
- Se basa en la existencia de **Subproblemas**
- Adem ás, los subproblemas se solapan, i.e. **comparten sub-subproblemas**

La diferencia con dividir para conquistar es que en esta ultima los **subproblemas son disjuntos**

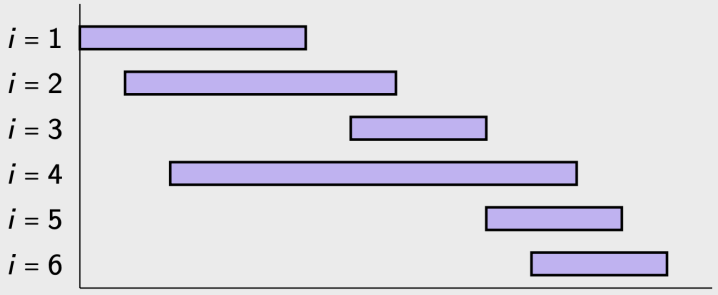
“La clave de programacion dinamica es recordar las soluciones a los subproblemas”

Auxiliar para la restriccion del problema → Organizar el problema

Dadas las charlas $\{1, 2, \ldots, 6\}$ ordenadas por f_i, añadimos

$$b(i) := \begin{cases} j, & j \text{ es la charla que termina más tarde antes de } s_i \\ 0, & \text{no hay tal charla} \end{cases}$$

- $i = 1,\ [0,5),\ v_1 = 2,\ b(1) = 0$
- $i = 2,\ [1,7),\ v_2 = 4,\ b(2) = 0$
- $i = 3,\ [6,9),\ v_3 = 4,\ b(3) = 1$
- $i = 4,\ [2,11),\ v_4 = 7,\ b(4) = 0$
- $i = 5,\ [9,12),\ v_5 = 2,\ b(5) = 3$
- $i = 6,\ [10,13),\ v_6 = 1,\ b(6) = 3$



Consideremos una instancia con charlas $\{1, \ldots, n\}$ ordenadas por f_i.
Supongamos que tenemos una solución óptima Ω para este problema.

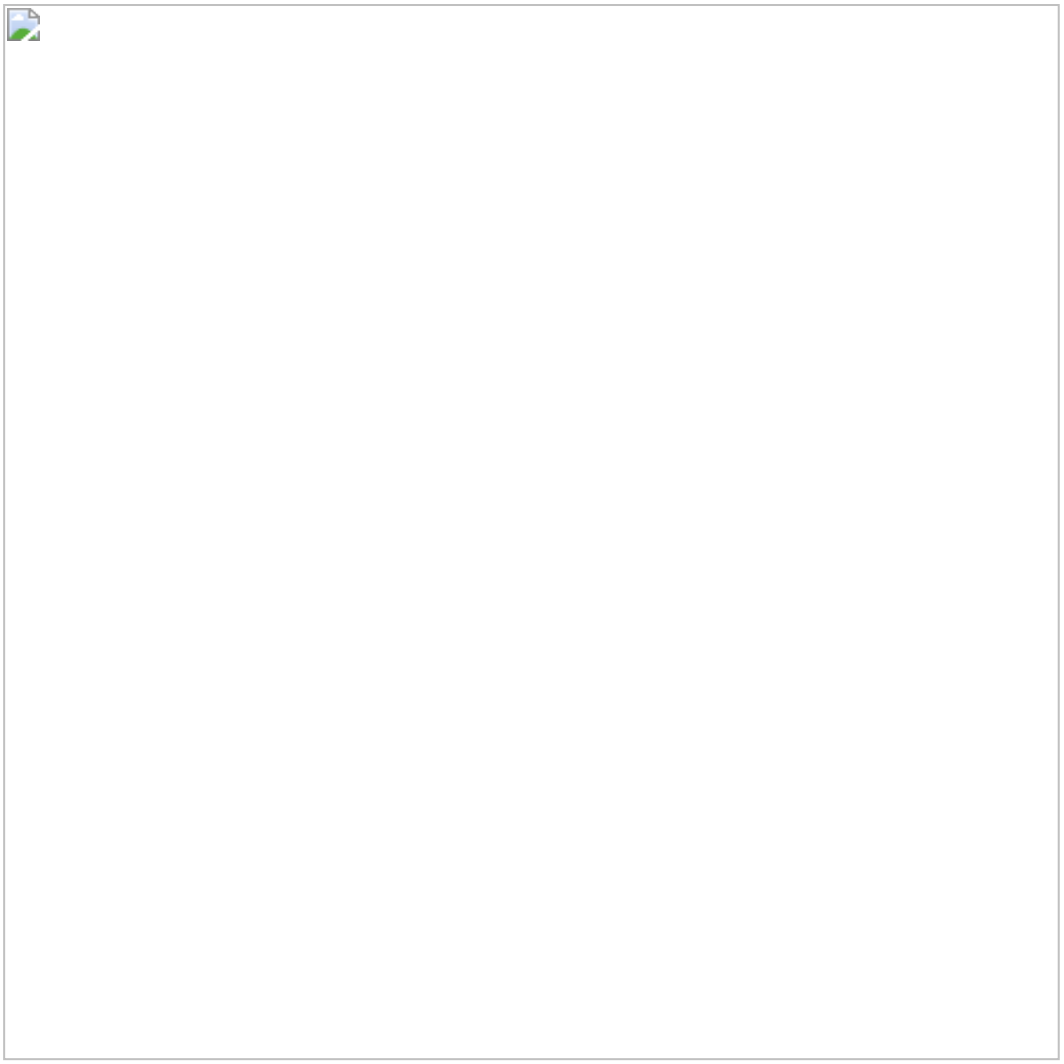
Consideremos la última charla, i.e. n. Tenemos dos opciones

- Si $n \notin \Omega$, entonces Ω es solución del **subproblema** que solo considera las charlas $\{1, \ldots, n-1\}$
- Si $n \in \Omega$, entonces no hay charla r tal que $b(n) < r < n$ que esté en Ω. Además, Ω contiene una solución óptima al **subproblema** con charlas $\{1, \ldots, b(n)\}$

Para encontrar la solución a un problema, necesitamos las soluciones a problemas más pequeños

Partimos de la derecha porque es mas facil de visualizar

- Si n no esta en omega, busco hasta la anterior
- Si esta en omega, saco todas las que chocan con n (un segmento)



Ahora generalizamos este problema, para ir avanzando a traves de J

Formalicemos las ideas anteriores

- Sea Ω_j la solución al problema con charlas $\{1, \ldots, j\}$ y sea $opt(j)$ su ganancia total. **Objetivo final:** obtener Ω_n con valor $opt(n)$
- Para cada $1 \leq j \leq n$, hay dos casos
  - Si $j \in \Omega_j$, entonces $opt(j) = v_j + opt(b(j))$
  - Si $j \notin \Omega_j$, entonces $opt(j) = opt(j-1)$
- Para saber si $j \in \Omega_j$, comparamos las dos opciones

$$\boxed{opt(j) = \max\{v_j + opt(b(j)),\ opt(j-1)\}} \qquad (\bigstar)$$

opt(b(j)) → Es lo que queda para atras

[1 ..... b(j) - - - - n]

→ opt(b(j)) = [1 .... b(j)]

Queremos saber si j va o no va en la solucion

Para eso hacemos el calculo → Una recursion

Opt(j) = max{ vj + opt(b(j)) , opt(j-1) }

ALGORITMO DE OPT

La ecuación (★) permite plantear el siguiente algoritmo recursivo

input : natural $0 \leq j \leq n$
output: ganancia óptima

$\texttt{Opt}(j)$**:**

```
1    if j = 0 :
2        return 0
3    else:
4        return max{v_j + Opt(b(j)), Opt(j − 1)}
```

Notemos que `Opt` requiere

- tener ordenadas las charlas por hora de término y conocer $b(j)$
- suponer que $\texttt{Opt}(0) = 0$

Problema de aproximacion del algoritmo → Guardar los valores (va a calcular todos los valores todas las veces)

Opt(6)
Opt(b(6)) = Opt(3)

● Opt(b(3)) = Opt(1)
● Opt(3−1)=Opt(2)

‣ Opt(2−1)=Opt(1)
Opt(6 − 1) = Opt(5)

● Opt(b(5)) = Opt(3)
‣ Opt(b(3)) = Opt(1)
‣ Opt(3−1)=Opt(2)...

● Opt(5−1)=Opt(4)
‣ Opt(4−1)=Opt(3)...

Notemos que aqui volvemos a contar cada vez que pasamos (no nos acordamos :/ si ya lo calculamos)

- Complejidad O(2**n)

ALGORITMO MODIFICADO → LINEAL

Agregaremos un arreglo global M donde almacenaremos cada $\texttt{Opt}(j)$ la primera vez que lo calculamos

```
RecOpt(j):
    if j = 0 :
        return 0
    else:
        if M[j] ≠ ∅ :
            return M[j]
        else:
            M[j] ← max{v_j + RecOpt(b(j)), RecOpt(j − 1)}
            return M[j]
```

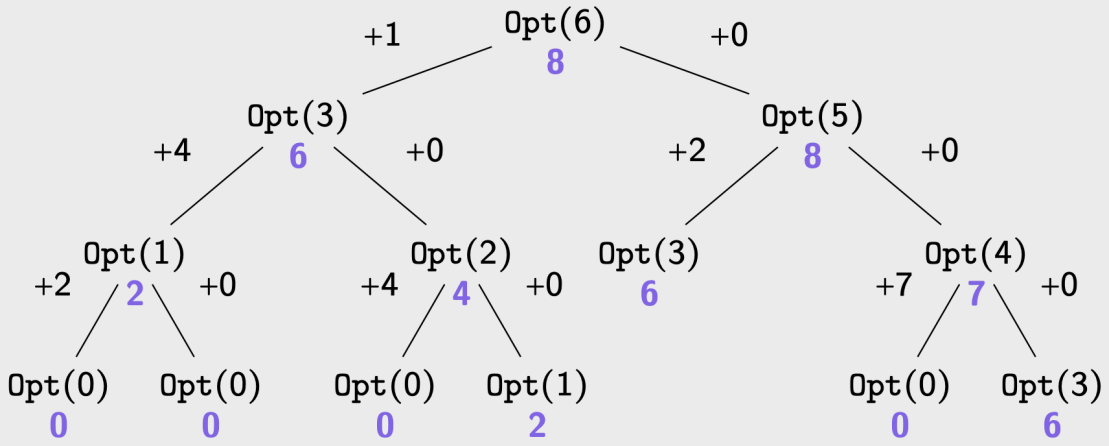
El algoritmo RecOpt toma tiempo O(n)

Problemas complicados → Elementos que se puedan repetir (por ejemplo las monedas, puedo agregar mas de una vez la moneda de 100 pesos) → Es

sutilmente distinto

ESTUDIAR ESTE EJMPLO

- $i = 1,\ [0,5),\ v_1 = 2,\ b(1) = 0$
- $i = 2,\ [1,7),\ v_2 = 4,\ b(2) = 0$
- $i = 3,\ [6,9),\ v_3 = 4,\ b(3) = 1$
- $i = 4,\ [2,11),\ v_4 = 7,\ b(4) = 0$
- $i = 5,\ [9,12),\ v_5 = 2,\ b(5) = 3$
- $i = 6,\ [10,13),\ v_6 = 1,\ b(6) = 3$



- [x] ~~ESTUDIASTE EL EJEMPLO (ver laminas paso a paso)~~

Ejemplo

Utilizando `RecOpt`, obtenemos la matriz de ganancias

$$M = \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 6 & 7 & 8 & 8 \\ \hline \end{array}$$
$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$$

Podemos **deducir la asignación** a partir de M, suponiendo que cada $v_j > 0$.

- Como $M[6] = M[6-1]$, no se incluye la charla 6
- Como $M[5] = v_5 + M[b(5)]$, se incluye la charla 5
- Como $M[3] = v_3 + M[b(3)]$, se incluye la charla 3
- Como $M[1] = v_1 + M[b(1)]$, se incluye 1

Con esto, las charlas asignadas son $\{1, 3, 5\}$

ESTRATEGIA ITERATIVA

También podemos plantear una solución **iterativa** para este problema, computando los subproblemas en orden de tamaño

```
ItOpt:
1    M[0] ← 0
2    for j = 1, . . . , n :
3        M[j] ← max{v_j + M[b(j)], M[j − 1]}
```

Al terminar, `ItOpt` deja en M las ganancias óptimas

El algoritmo `ItOpt` también toma tiempo $\mathcal{O}(n)$

La iteratividad es mas barata que la recursión

CLASE 21

30/10/24

Resumen Clase Anterior :

Problemas Optimizacion puede parecerse a un cpc (resolvible con backtracking), pero que podemos descomponer en problemas mas chicos → No es dividir para conquistar, porque los problemas se solapan → Pueden repetirse.

Progra dinamica → Se trata de encontrara la recurrencia

Tenemos 2 plantillas para progra dinamica → La primera iterativa y la 2da recursiva

Si el problema a resolver requiere entender como se resuelve → El paso a paso , el recursivo se puede ver mas claramente (clean code)

La solución iterativa (aveces for dentro de for), habitualmente es mas dificil de comprender como funciona, pero puede llegar a ser mas eficiente

Problema concurrente

En las pruebas se suelen confundir → Intentan optimizar el valor de las monedas, porque es solo contarlas

El problema del vuelto es el mismo, solo que cambia el enfoque (con el de la mochila)

Es natural (razonal) pensar partir con las monedas mas grandes

- Suponemos que los valores de las monedas, ordenadas de mayor a menor, son {v1, v2, ..., vn}
- Tenemos una cantidad ilimitada de monedas de cada valor

Ejercicio para la prueba: **MODIFICAR ESTE PROBLEMA PARa hacerlo con una cantidad LIMITADA de monedas**

Estrategia codiciosa no sirve siempre

Si {v1,v2,v3} = {6,4,1}

Ejemplo si el vuelto es 8 S= 8, entonces la estrategia produce

8 = 6 + 1 + 1

Y el optimo era 8 = 4 + 4 → Uso menos monedas

Definamos la funcion

---

definimos z(S,n) como el problema de encontrar el menor nu ́mero de monedas para totalizar S

EJERCICIO: Proponga una recurrencia para resolver el problema z(S,n) y plantee un algoritmo a partir de ella.

Hay dos opciones respecto a las monedas de valor vn

- Si se incluye la mondea de valor Vn:
  - $Z(S,n) = Z(S - Vn, n) + 1$

- Si no se usan monedas de valor Vn:
  - $Z(S , n) = Z (S , n - 1)$

Para las monedas de valor Vn:

$Z(S, n) = \min\{Z (S - Vn, n) + 1, Z(S, n-1) \}$

Luego generalizamos para el subconjunto de los primeros j valores de monedas:

$Z(T,k)=\min\{Z(T-vk,n)+1, Z(T,k-1)\}$

Condiciones de exito al final

Condiciones de navegacion en la recurrencia

OJO: donde $Z(T,0) = +\infty$, si $T > 0$, y $Z(0,k)=0$

Ejercicio

Con esto, podemos plantear el siguiente algoritmo iterativo

```
Change(S):
    for T = 1, ..., S :
        Z[T][0] ← +∞
    for k = 0, ..., n :
        Z[0][k] ← 0
    for k = 1, ..., n :
        for T = 1, ..., S :
            Z[T][k] ← Z[T][k − 1]
            if T − v_k ≥ 0 :
                Z[T][k] ← min{Z[T][k], Z[T − v_k, k]}
```

2 typos: min{Z[T][k] → k = -1 y k = +1 en la ultima linea

Problema para pensar: Como dar el mejor vuelto, cuando no tenemos toda la denominacion (como hago para acercarme mas si me faltan ciertas monedas)

Comparacion de Tecnicas

1. Dividir para conquistar → lo vimos por primera vez en mergesort
2. Backtracking
3. Algoritmos codiciosos
4. Porgramacion Dinamica

Cuando preferimos una sobre otra en la practica

Backtracking → Modelamiento e implementacion → Plantear problema , darse cuenta de variables dominios y restricciones

Pregunta ricolina → Aplique esta heuristicas, propagacion de dominio

Dividir para conquistar no deberia salir

CLASE 22

4/10/24

Grafo: Estructura para representar nuestra data

Nos da las relaciones entre los elementos

CLASE 23

6/11/24

DFS

Hint: Tener en el torpedo → DFS

CLASE 24

11/11/24

Terminamos de ver Kosaraju y vimos Kruskal (Arboles MST)

CLASE 25

RESUMEN i2

Problemas tipo

Diccionarios

- Tablas Hash
- Arboles B+

Estrategia Algoritmicas

- Backtracking
- Algoritmos codiciosos (greedy)
- Progra diamica

Algoritmo de Grafos

- DFS

Problema: Greedy, backtracking y pd, generalmente lo preguntan para modelar el probelema

por ejemplo en backtracking para reconocer variables dominio y restriccion

Est

Grafo → Reconozca el problema, modele como un grafo, y aplique DFS, aplique kosaraju

Problemas mas complejos (menos directos)

Esta el recorrido de DFS, modifiquelo para

En vez de en profunidad, cuando encuentre un camino haga tal cosa

MODIFICAR

Como transformamos backtracking en codicioso → No retroceder

Como hacemos un bakctracking codicioso, en la eleccion

Esto es la Heuristica → Elige la mejor variable

Si mi estrategia codiciosa no es optima, backtracking igual encuentra una solucion

Estas cosas

Se pide una estrategia codiciosa, y alguien ya uso backtracking, modifiquelo para bla bla bla

Estudiar Heuristica

Backtracking → Escribir algoritmo mejorado

Si quiero hacer poda, heusristica y propagacion

Propagacion es una forma fuerte para backtracking

DICCIONARIO

Estructura de datos, sin orden como **HASH**

Factor de carga → Como manejo colisiones

- Encapsulamiento → Donde se pueda
- Encadenamiento → Lista ligadas en colisiones

En la prueba podrian preguntar por: Estrategia de sondeo lineal → Elimine este elemento (complicado)

Tipos de problema → Diagnosticar cosas

La tabla de hash no se comporta bien → Factor de carga

Solucion → Mezclar tablas de hash, o con otras cosas (arbol que carga de un hash, hash que carga de un hash)

Hash para pasajeros por vuelo, relacionando la info con el equipaje

La busqueda de equipaje, y de pasajero por vuelo

Tabla principal pasajero , y secundaria con el ticket del pasaje, que apunta al hash de principal (el de los pasajeros)

Saber de operaciones, complejidad, complejidad de colisiones, → Es relevante cuando tiene un factor de carga alto

Si tenemos un factor de carga 1, con encadenamiento

Las listas van a ser corta, con el n total de datos

- [ ] Comprender funciones de hash (aprenderse la de modulo)
- [ ] Propiedades deseables (impacto en las colisiones)
- [ ] Comparar desempeño de tecnicas de resolicion de colisiones
- [ ] Uso de tablas para almcenar y consultar

ARBOLES B+

Idea de busqueda en base de datos o de archivos → Buscar el bloque que lo tiene → Con un identificador de bloque

Busqueda en el disco

D del arbol B+, podemos afirmar cosas sobre la altura

Crece el arbol hacia arriba, aumenta de nivel, solo cuando se llena la raiz

Y se va balanceando razonablemente

Siempre la altura es la minima es para

La pregunta del sem pasado → Orientada a entender la mezcla del B+ con consulta SQL

Mezcla entre 2 indices distintos

ALGORITMOS

BACKTRACING

- se puede arrepentir y cambiar de desicion

Codicioso

- Toman la mejor decicion aparente en el momento y **no se arrepiente**

Dinamica → Cuando el problema se puede descomponer

No esta acotado como codicioso, como problemas de optimizacion

Tambien lo puedo usar, para decidir un camino, conforme a la info que tengo hasta el momento

Satisfaccion de restricciones (cpc):

- Identificacion de varibales → Parte de comprenseion del problema
  - Tips:
  - Si hay un problema de backtracking, generalmente es largo
  - Decidir como modelar el problema → Primero definir las variables
  - Entender el problema → Y luego decidir el modelamiento
  - Puedo elegir alguna variable, le pongo su dominio y restriccion (adecuado al problema), podre resolverlo
  - Generalmente se puede llegar en varias formas

Como aplicar poda

Como aplicar propagacion

Como aplicar heuristica

La forma de la pregunta puede variar un monton

→ 1. Proponga una estrategia para hacer algo con su algoritmo anterior : Puede ser texto, propondria elegir la herutistca y bla blab bla

- No necesariamente hay que llevarla a codigo

→ 2. Laberintos, carga de mochilas, barcazas, problema de **ESTIBAR CARGAS → estudiar**

- (nivelada la carga (balnceado = estibado)), pesos muy similares por ejemplo para un camion para que no se ladee
- Estibar de tal forma de que el lado derecho y el izquierdo sean iguales
- FAlso si no se puede balancear, verdadero si hay una combinacion con que

Generalmente la pregunta de backtracking, requiere concentracion para leer, aplicar (**SOBRE LA PLANTILLA QUE TENEMOS**), mas que el problema del pseudocodigo, si en la modelacion

Va a salir → Reconocer que esta haciendo un codigo de backtracking

Código → Propongación, reconocer que estrategia se esta usando para hacer algo ...

Con codiciosa

Las preguntas generalmente son

DEMUESTRE que cierto algoritmo codicioso no sirve

→ **ENCONTRA CONTRAEJEMPLOS**

Estrategia codiciosa → PAra detectar fallas en estrategias codiciosas

Voy avanzando, y eligo

y el mas grande o mas valioso, estaba al inicio y al final

Que pasa si el primer dato no me resuelve el problema

Si entiendo la forma del set de datos, para encontrarle una fallla a la estrategia → Estoy listo

A veces N = 3 o N = 5 es necesario para demostrar

DINAMICA

No Siempre puede aparecer una pregunta → Si un problema se puede resolver con progra dinamica

Tiene que ver con el planteo de la ec recurrencia

Si el problema se puede descomponer en subproblemas, (disjuntos?)

El problema de orden en una llamada recursiva es menor que otra

Es parte de la recurrencia poder resolver estos problemas

El problema de las monedas, es distinto al de las charlas

El valor, la forma de la recurrencia, esta sumando valores a una funcion de optimizacion

En el caso de las monedas esta sumando a la **Cantidad** de cosas (monedas)

Maximizar numero de vehiculos dentro de la barcaza (no enrredarlo con precios xd)

Tenemis muchas estrategias, tenemos que encontrar la mas adecuada para resolver un problema

DFS

- Busqueda en profundidad

Tema flexible, sabe detectar ciclos, cosas fuertemente conectadas con kosaraju

Normalmente problemas de recorrido

Habitualmente, modificar DFS,

PAra ello, entender que el recorrido DFS genera arboles dentro del grafo

Que genera el orden topologico (orden de escoger datos para el arbol)

Varios arboles dentro de un mismo recorrido

Varios tipos de aristas

Una pregunta podria ser → **IDENTIFIQUE ATAJOS** dentro del recorrido

Encontre un recorrido, tiene un arbol a caminos mas corto

Luego los atajos, no cumplen la cobertura de pasar por todos los nodos

Pero si ayuda a reconocer que hay formas de recorrer en menos pasos