

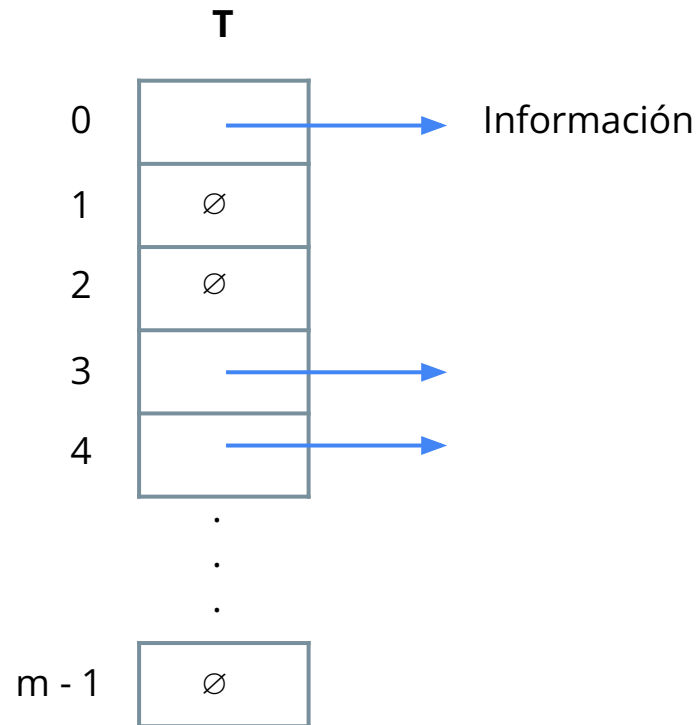
# Repaso I2

Hashing - Backtracking

# Tabla de Hash

$T[k] = \emptyset$     **no está**

$T[k] \neq \emptyset$     **está**



# Función de Hash

- **Calcular** índice a partir de la clave
- **Obtener** el valor asociado a una clave

**Definimos** una función **h**:

$$h : \mathbf{D} \rightarrow \{0, 1, \dots, m-1\}$$

Dominio **D** de claves  
o keys

k

$$h(k) = 3$$

**T**

0

1

2

3

4

⋮

⋮

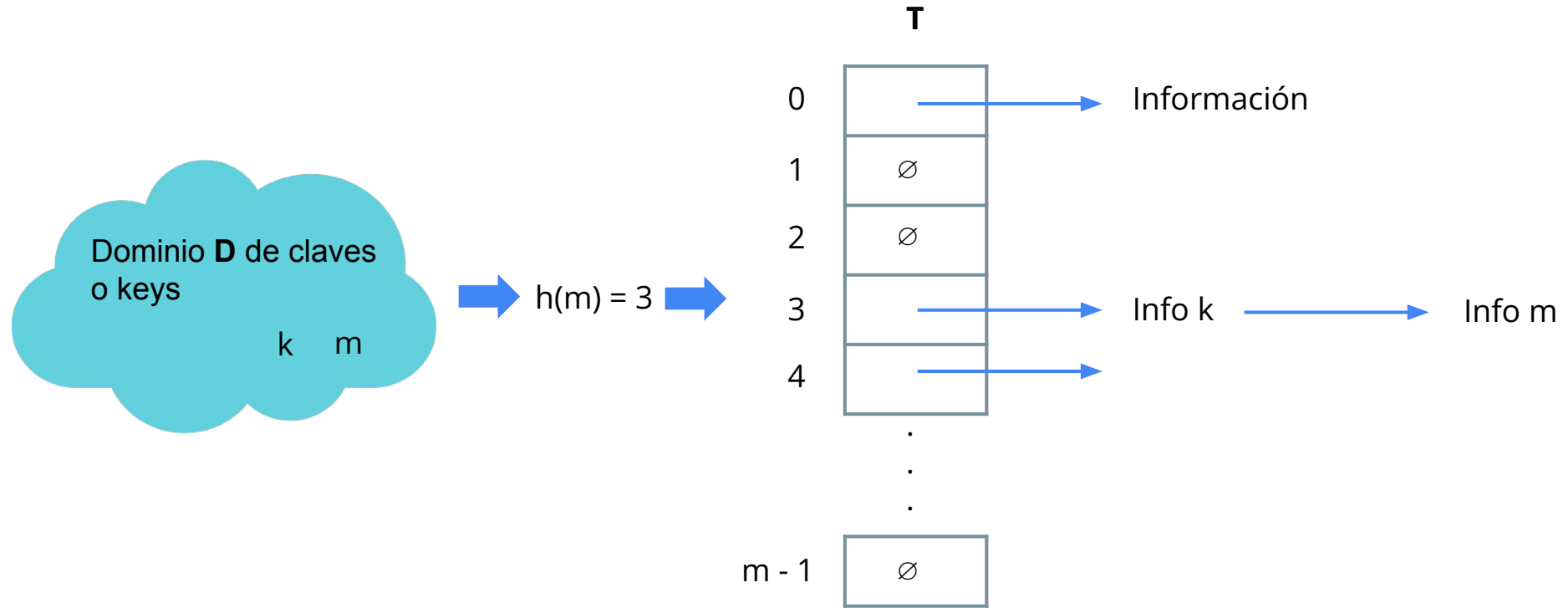
⋮

m - 1

Información

Información

# Encadenamiento



# Ejercicio 1. Pizzeria M97

La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de los millones de pedidos que recibe por segundo.

Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

# Ejercicio 1. Pizzeria Mg7

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.
- ¿Qué esquema de resolución de colisiones debería usarse y por qué?
- ¿Qué es lo que se guarda en la tabla?

# Ejercicio 1. Pizzeria Mg7

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.

# Ejercicio 1. Pizzeria Mg7

- Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente.
- Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo key. En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió.



# Ejercicio 1. Pizzeria Mg7

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?

# Ejercicio 1. Pizzeria Mg7

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?
- Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea  $O(1)$  y se atienda en orden de llegada. (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son  $O(\log(n))$ ).

# Ejercicio 1. Pizzeria Mg7

- ¿Qué esquema de resolución de colisiones debería usarse y por qué?
- Los *values* de un mismo key se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea  $O(1)$  y se atienda en orden de llegada. (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son  $O(\log(n))$ ).
- La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura.

# Ejercicio 1. Pizzeria Mg7

- ¿Qué es lo que se guarda en la tabla?

# Ejercicio 1. Pizzeria Mg7

- ¿Qué es lo que se guarda en la tabla?
- En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió.

# Ejercicio 1. Pizzeria Mg7

- ¿Qué es lo que se guarda en la tabla?
- En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió. Si varias personas pidieron la misma pizza, se guarda como value una lista ligada de los clientes que pidieron ese sabor de pizza.
- Pero como el dominio de las key es infinito, debemos ir despejando las celdas de la tabla cuando una key se queda sin values, ya que no tenemos memoria infinita. Para esto es necesario usar encadenamiento ya que permite eliminar keys de la tabla sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean *hasheados* a la misma celda de la tabla (colisión de sabores), las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga

## Ejercicio 2. Función de Hash

- Función de hash: Sumar los dígitos del número repetidamente hasta que quede un número.
- El caso del dígito 9 se tiene que  $h(9) = 0$
- Con esto se espera con esto que el resultado sea probablemente correcto.

Consideración importante: Notar que el hash modulariza por mod 9

## Ejercicio 2. Función de Hash

- Queremos multiplicar 2 número X e Y y verificar que se cumpla  $X*Y= Z$ . Para verificar esto, utilizamos una función de hash  $h()$ , tal que se cumpla que:

$$h( h(X)* h(Y)) = h(Z)$$



## Ejercicio 2. Función de Hash

- Queremos multiplicar 2 número X e Y y verificar que se cumpla  $X*Y= Z$ . Para verificar esto, utilizamos una función de hash  $h()$ , tal que se cumpla que:

$$h( h(X)* h(Y)) = h(Z)$$

## Ejercicio 3. Heaps + Hash

Una tienda quiere premiar a sus  $k$  clientes más rentables. Para ello cuenta con la lista de las  $n$  compras de los últimos años, en que cada compra es una tupla de la forma  $(ID\ Cliente, Monto)$ . La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras. Explica cómo usar tablas de hash y heaps para resolver este problema en tiempo esperado —o promedio—  $O(n + n\log(k))$ .

El problema se separa en dos partes.

- Calcular la rentabilidad de cada cliente
- Buscar los  $k$  clientes más rentables

# Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas  $(ID\ Cliente, Rentabilidad)$ , donde la rentabilidad para el ID Cliente  $i$  es la suma de todos los montos de las tuplas de la forma  $(i, Monto)$ .

Para esto creamos una tabla de hash  $T$  donde se almacenan tuplas  $(ID\ Cliente, Rentabilidad)$ . Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

# Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

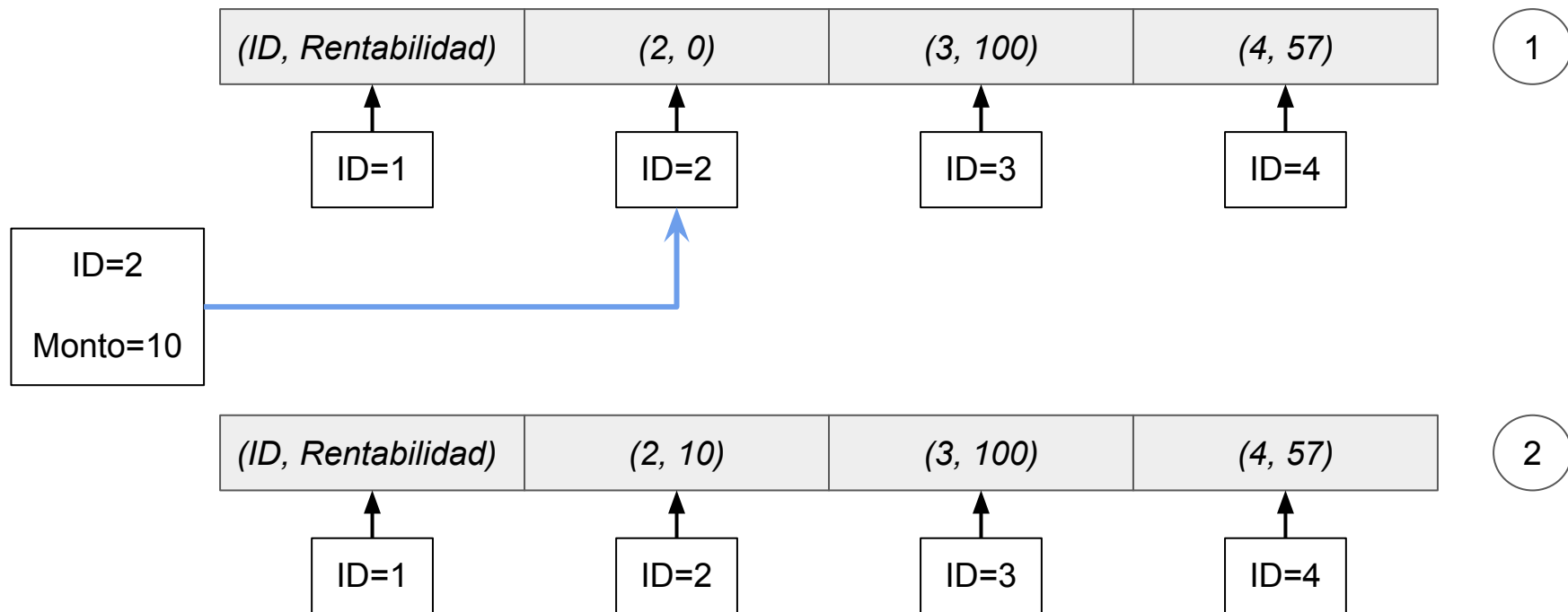
Queremos obtener el set de tuplas  $(ID\ Cliente, Rentabilidad)$ , donde la rentabilidad para el ID Cliente  $i$  es la suma de todos los montos de las tuplas de la forma  $(i, Monto)$ .

Para esto creamos una tabla de hash  $T$  donde se almacenan tuplas  $(ID\ Cliente, Rentabilidad)$ . Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

$(ID, Rentabilidad)$	$(2, 0)$	$(3, 100)$	$(4, 57)$
ID=1	ID=2	ID=3	ID=4

# Ejercicio 3. Heaps + Hash



# Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas  $(ID\ Cliente, Rentabilidad)$ , donde la rentabilidad para el ID Cliente  $i$  es la suma de todos los montos de las tuplas de la forma  $(i, Monto)$ .

Para esto creamos una tabla de hash  $T$  donde se almacenan tuplas  $(ID\ Cliente, Rentabilidad)$ . Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las  $n$  tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

# Ejercicio 3. Heaps + Hash

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas  $(ID\ Cliente, Rentabilidad)$ , donde la rentabilidad para el ID Cliente  $i$  es la suma de todos los montos de las tuplas de la forma  $(i, Monto)$ .

Para esto creamos una tabla de hash  $T$  donde se almacenan tuplas  $(ID\ Cliente, Rentabilidad)$ . Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las  $n$  tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

La inserción en esta tabla tiene tiempo esperado  $O(1)$  como se ha visto en clases. Como se realizan  $n$  inserciones, esta parte tiene tiempo esperado  $O(n)$ .

## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables

Sea  $m$  el total de clientes distintos. Creamos un min-heap de tamaño  $k$  que contendrá los  $k$  clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los  **$k$  más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en  $T$ , insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre  $T$ .



## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables

Sea  $m$  el total de clientes distintos. Creamos un min-heap de tamaño  $k$  que contendrá los  $k$  clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los  **$k$  más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en  $T$ , insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre  $T$ .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables

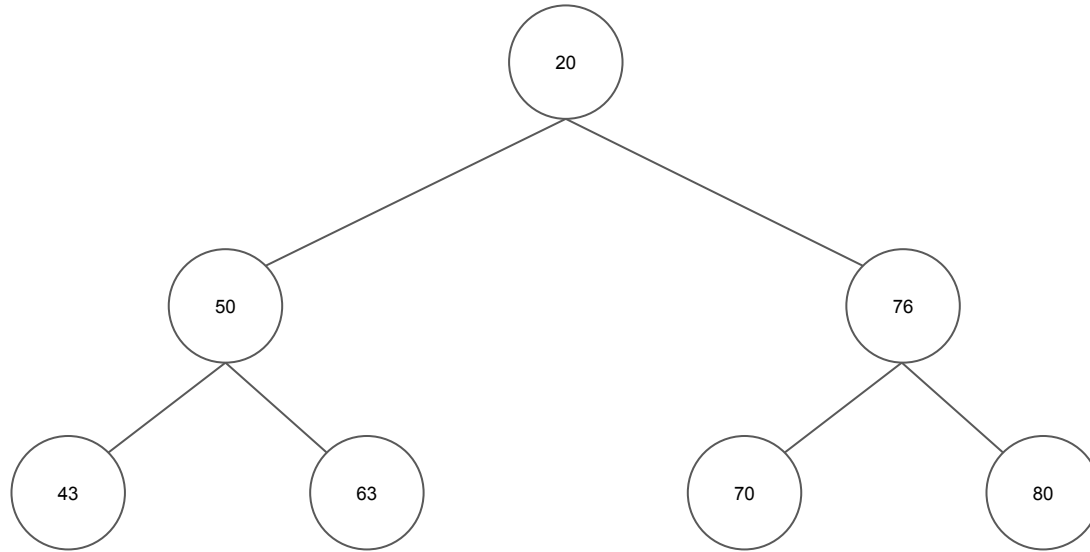
Sea  $m$  el total de clientes distintos. Creamos un min-heap de tamaño  $k$  que contendrá los  $k$  clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los  **$k$  más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en  $T$ , insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre  $T$ .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

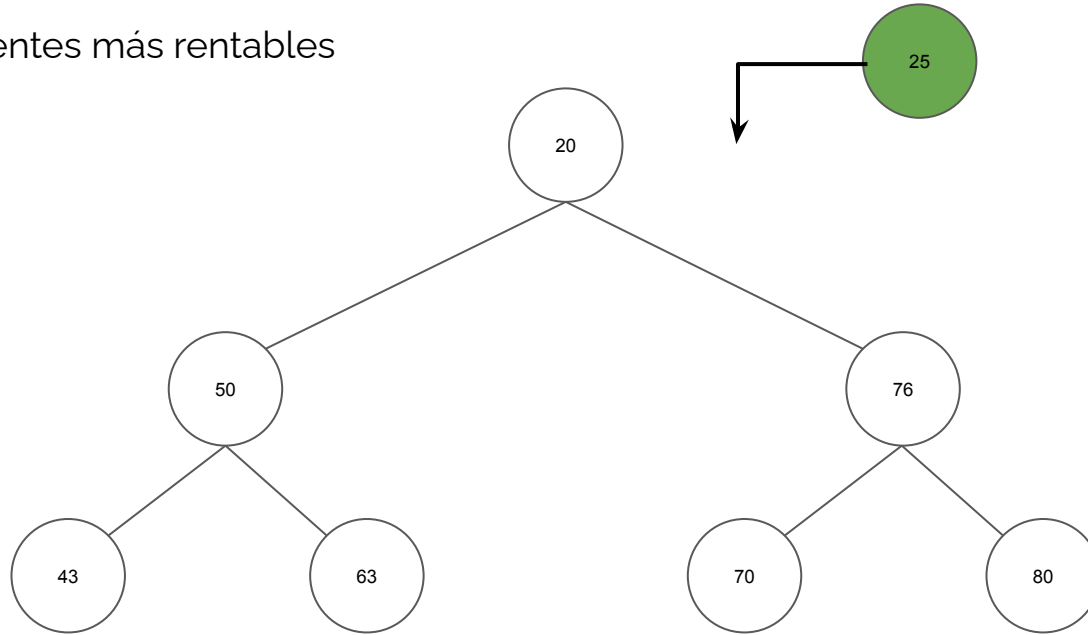
## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables



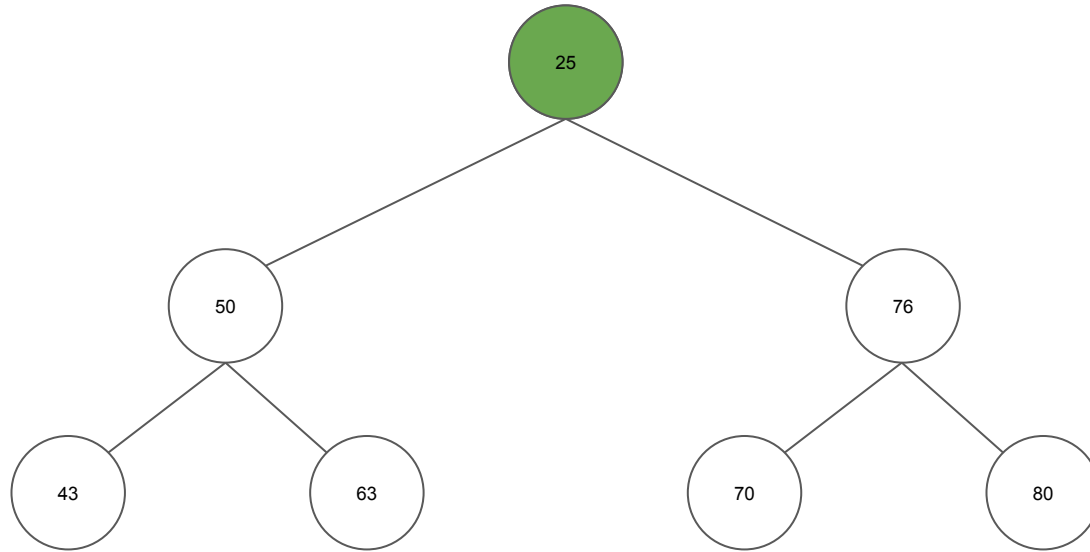
### Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables



## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables

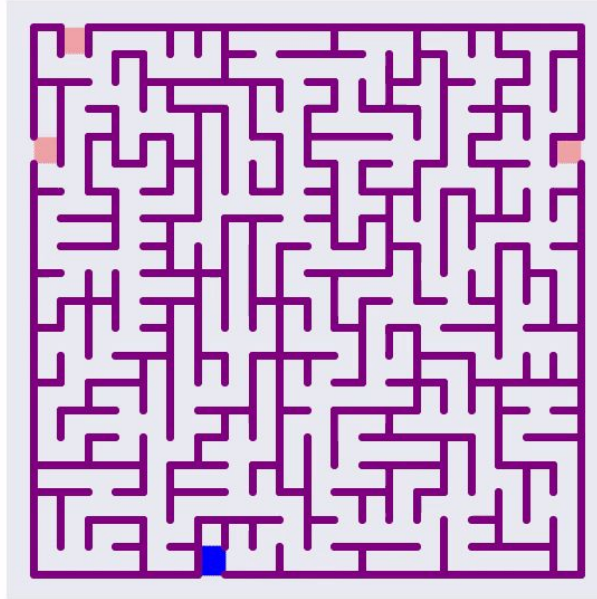


## Ejercicio 3. Heaps + Hash

Buscar los  $k$  clientes más rentables

Cada inserción en el heap toma  $O(\log k)$ . En el peor caso insertamos los  $m$  elementos en el heap, por lo que esta parte es  $O(m \log(k))$ . Pero como en el peor caso  $m=n$ , esta parte es  $O(n \log k)$ .

# Backtracking



# Backtracking

6	2	3	7	1	8	9	4	5
4	5	9	2	3	6	1	8	7
8	7	1		9		3	2	6
9		4		7		2	5	1
7	1	8	9	5	2	6	3	4
2	6					7	9	8
5				8	7	4	1	2
1				6		8	7	3
3						5	6	9



# ¿Por qué Backtracking?

- Problemas de decisión: Búsqueda de una solución factible.
- Problemas de optimización: Búsqueda de la mejor solución.
- Problema de enumeración: Búsqueda de todas las soluciones posibles.

# Backtracking

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$

*return false*

# Backtracking

¿Se puede mejorar este algoritmo?

# Backtracking

Hay tres mejoras posibles:

- Podas
- Propagación
- Heurísticas

# Poda

Se deducen restricciones a partir de las restricciones o asignaciones anteriores que pueden ser agregadas al problema.

En otras palabras, estamos podando parte del conjunto de caminos a soluciones posibles.

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  no es válida, *continue*

$x \leftarrow v$

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$

*return false*

# Propagación

Cuando a variable se le asigna un valor, se puede propagar esta información para luego poder reducir el dominio de valores de otras variables.

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  alguna variable de  $X$

*for*  $v \in D_x$ :

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$ , propagar

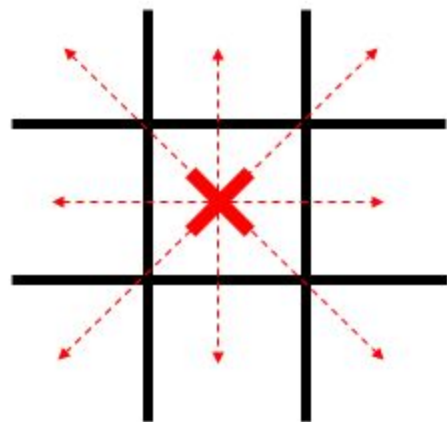
*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

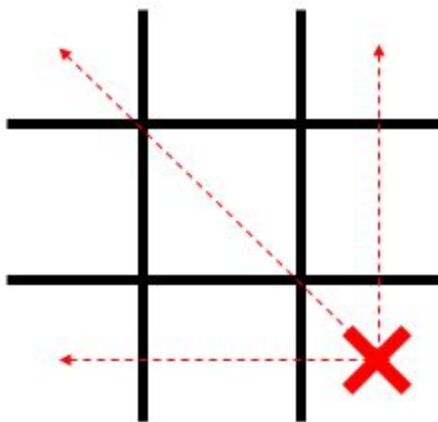
$x \leftarrow \emptyset$ , propagar

*return false*

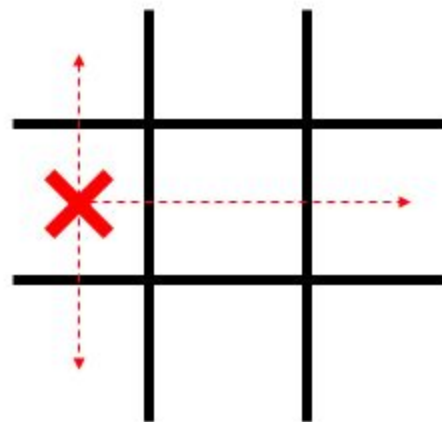
# Heurística



4 ways to win the game



3 ways to win the game



2 ways to win the game

# Heurística

Una heurística es una aproximación al mejor criterio para abordar un problema.

*is solvable*( $X, D, R$ ):

*if*  $X = \emptyset$ , *return true*

$x \leftarrow$  la mejor variable de  $X$

*for*  $v \in D_x$ , de mejor a peor:

*if*  $x = v$  viola  $R$ , *continue*

$x \leftarrow v$

*if is solvable*( $X - \{x\}, D, R$ ):

*return true*

$x \leftarrow \emptyset$

*return false*



# Ejercicio 1 - Kakuro

	23	19	11
22			
21			
10			

# Ejercicio 1 - Kakuro

- Variables?
- Dominio?
- Restricciones?

	23	19	11
22			
21			
10			

# Ejercicio 1 - Kakuro

- Variables: Valores de celdas
- Dominio:  $\{1, \dots, \text{MAX}\}$
- Restricciones: Las filas y columnas suman los valores esperados. Cada número es usado una sola vez por suma.

	23	19	11
22			
21			
10			

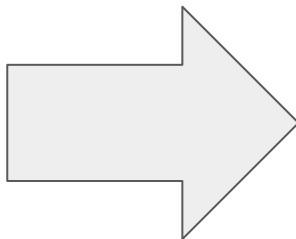
# Ejercicio 1 - Kakuro

- Variables: Valores de celdas
- Dominio:  $\{1, \dots, \text{MAX}\}$
- Restricciones: Las filas y columnas suman los valores esperados. Cada número es usado una sola vez por suma.

	23	19	11
22	9	7	6
21	8	9	4
10	6	3	1

## Ejercicio 2 - Corredores: 2021-2 I2

- N corredores.
- Se registra el tiempo del primer corredor en llegar.
- El tiempo del resto de los corredores se registra en función del primer corredor.



- Lista T: De largo N, que registra los tiempos de los corredores, partiendo del tiempo 0.
- Lista D: de largo  $N(N-1)/2$ , que registra la diferencia de tiempos entre cada par de corredor. Ordenadas de menor a mayor.

# Ejercicio 2 - Corredores: 2021-2 I2

Problema: Se perdió la lista T, y tenemos que volver a generarla a partir de la lista D.

Importante de notar:

- El dominio de la lista son los valores D
- Reconocer la condición de término (el último elemento de D).
- Identificar el largo de la lista T.
- Consideración: valores de D no repetidos
- Tiempos no repetidos.

# Ejercicio 2 - Corredores: 2021-2 I2

Pseudocódigo:

```
solve(X, D):  
    if X[-1] == D[-1]:  
        print(X)  
        return True  
    u = 0  
    for value in D:  
        if valid_value(value):  
            D.pop(u)  
            X.append(value)  
            if solve(X, D):  
                return True  
            X.pop()  
            D = funcion_insert(D, value, u)  
        u += 1  
    return False
```