Bases de Datos

Clase 11: Fundamentos de EDD y Almacenamiento e Índices

Hasta ahora

Sabemos como funciona un sistema, pero

¿Cómo se evalúa una consulta?

Demora en Búsqueda

“La diferencia de tiempo entre tener un dato en RAM versus traerlo de disco es comparable a la de tomar el sacapuntas del escritorio donde estoy sentado versus tomarme un avión a Punta Arenas para ir a buscarlo y regresar.”

En general, lo más lento del DBMS es ir a buscar los datos a disco, por lo que queremos minimizar el número de I/O

Modelo de costos

Memoria Principal vs. Memoria Secundaria



Memoria Secundaria

- Datos guardados en memoria secundaria
- La lectura se hace por **páginas**
- Una página tiene un tamaño de *B* tuplas



Memoria Principal

- Los datos son llevados a memoria principal
- La memoria tiene una capacidad de *M* páginas

Modelo de costos

Memoria Principal vs. Memoria Secundaria



Se cuentan los accesos (lectura/escritura) a memoria secundaria

Memoria Secundaria

- Datos guardados en memoria secundaria
- La lectura se hace por **páginas**
- Una página tiene un tamaño de *B* tuplas



Los accesos a memoria principal son despreciables

Memoria Principal

- Los datos son llevados a memoria principal
- La memoria tiene una capacidad de *M* páginas

Modelo de costos



¿Cuánto cuesta leer n tuplas de memoria secundaria?

$$\left\lceil \frac{n}{B} \right\rceil$$

¿Cuántas páginas usa una relación R?

$$\left\lceil \frac{|R|}{B} \right\rceil$$



¿Y qué tiene que ver esto con Bases de Datos?

Cómo se guarda una tabla

Supongamos una tabla T(a int, b text) con 9 tuplas

Disco Duro

Tupla 1	Tupla 2	Tupla 3	Tupla 4	Tupla 5	Tupla 6	Tupla 7	Tupla 8	Tupla 9
Página 1			**Página 2**			**Página 3**		

Costo I/O (Input/Output)

¡El costo más grande de las bases de datos es traer las páginas del disco duro a memoria RAM!

Queremos responder las consultas haciendo la menor cantidad de lecturas al disco duro

Llamamos **costo de I/O** al número de páginas llevadas a memoria para responder una consulta

Costo de una consulta

Disco Duro

Tupla 1	Tupla 2	Tupla 3	Tupla 4	Tupla 5	Tupla 6	Tupla 7	Tupla 8	Tupla 9
Página 1			**Página 2**			**Página 3**		

¿Cuál es el costo en I/O de hacer la siguiente consulta?

SELECT * FROM T

$$\left\lceil \frac{|R|}{B} \right\rceil = \left\lceil \frac{9}{3} \right\rceil$$

El costo es 3, porque debo leer las 3 páginas

Costo de una consulta

Disco Duro

Tupla 1	Tupla 2	Tupla 3	Tupla 4	Tupla 5	Tupla 6	Tupla 7	Tupla 8	Tupla 9
Página 1			**Página 2**			**Página 3**		

¿Cuál es el costo en I/O de hacer la siguiente consulta?

SELECT T.a FROM T

$$\left\lceil \frac{|R|}{B} \right\rceil = \left\lceil \frac{9}{3} \right\rceil$$

El costo nuevamente es 3, porque debo leer las 3 páginas

Costo de una consulta

Disco Duro

Tupla 1	Tupla 2	Tupla 3	Tupla 4	Tupla 5	Tupla 6	Tupla 7	Tupla 8	Tupla 9
Página 1			**Página 2**			**Página 3**		

¿Cuál es el costo en I/O de hacer la siguiente consulta?

SELECT * FROM T WHERE T.a = 4

El costo nuevamente es 3, porque debo leer las 3 páginas

$$\left\lceil \frac{|R|}{B} \right\rceil = \left\lceil \frac{9}{3} \right\rceil$$

Costo de una consulta

Disco Duro

Tupla 1	Tupla 2	Tupla 3	Tupla 4	Tupla 5	Tupla 6	Tupla 7	Tupla 8	Tupla 9
Página 1			**Página 2**			**Página 3**		

¿Cuál es el costo en I/O de hacer la siguientes consultas?

SELECT * FROM T WHERE T.a = 4
SELECT * FROM T WHERE T.a >= 4

$$\left\lceil \frac{|R|}{B} \right\rceil$$

¿Podemos hacer esto mejor?

Con **Índices**

Pero primero veamos Fundamentos de Estructuras de Datos

Repaso: Memoria RAM

Memoria RAM

Un **bit** es la unidad indivisible de información computacional que puede valer 0 o 1. Además, un **bit** es una celda física indivisible de almacenamiento en un computador.

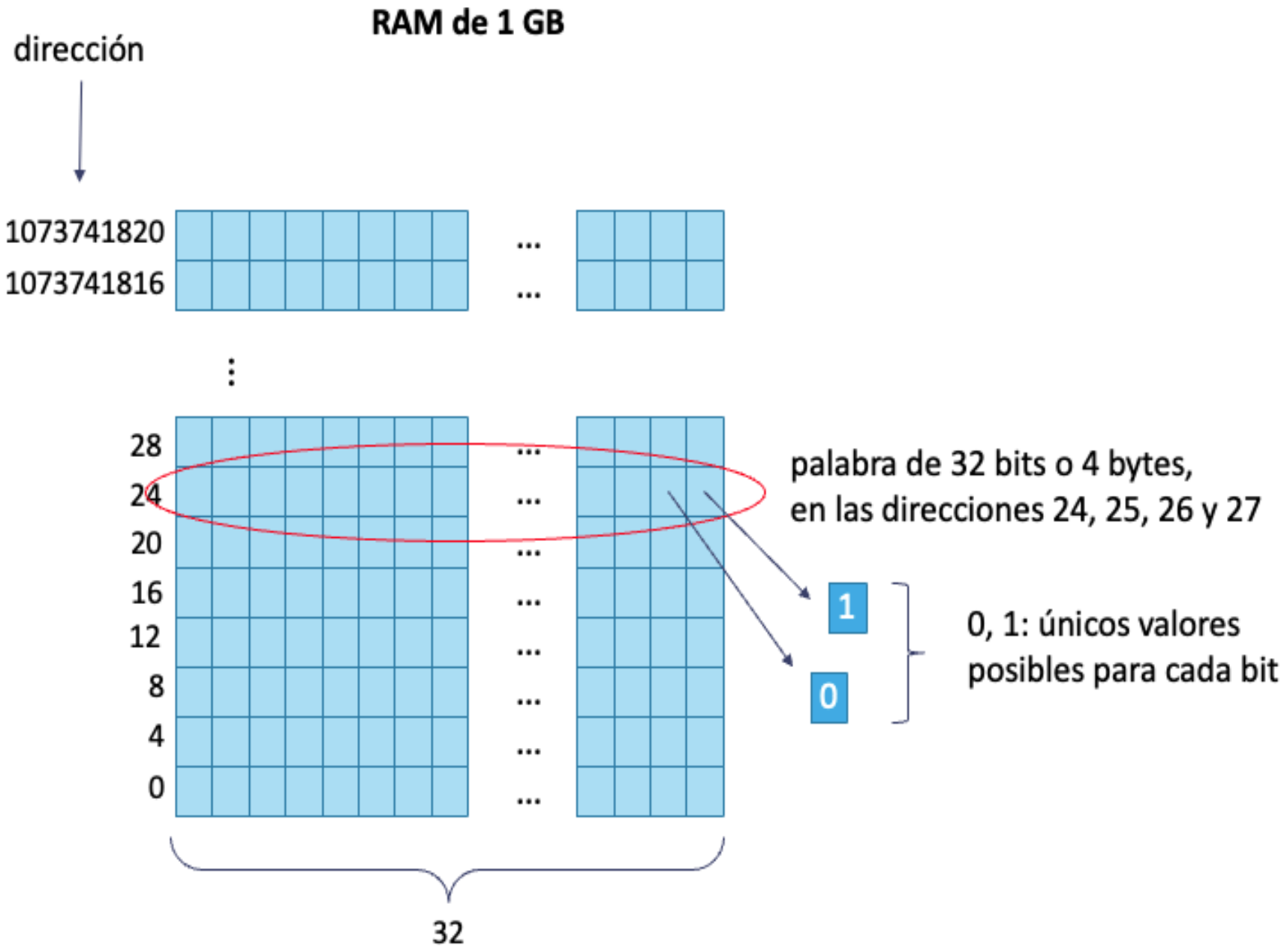
La memoria principal de un computador (RAM) puede ser imaginada como una gran tabla, arreglo o matriz de bits:

- Cierto número de columnas
- Algunos miles de millones de filas

Cada fila de esta tabla tiene una dirección única:

- Corresponde a su posición relativa dentro de la tabla
- Es un natural que parte en 0 (dirección de la primera fila) y aumenta de 4 en 4

Memoria RAM

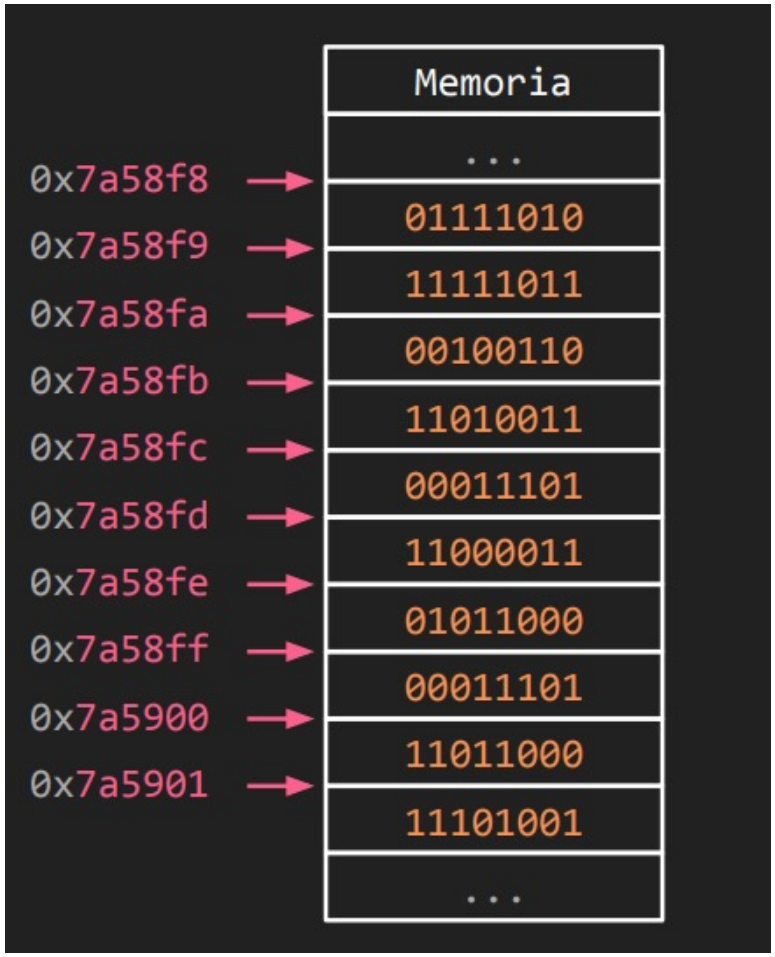

RAM de 1 GB
dirección
1073741820
1073741816
28
24
20
16
12
8
4
0
palabra de 32 bits o 4 bytes,
en las direcciones 24, 25, 26 y 27
1
0
0, 1: únicos valores
posibles para cada bit
32

Memoria RAM y Punteros

Cada variable de un programa tiene:

- Posición (**dirección de memoria**)
- Tamaño, en número de bytes
- **Valor**

Puede pensarse en la memoria como una tabla gigante que asocia direcciones a bytes

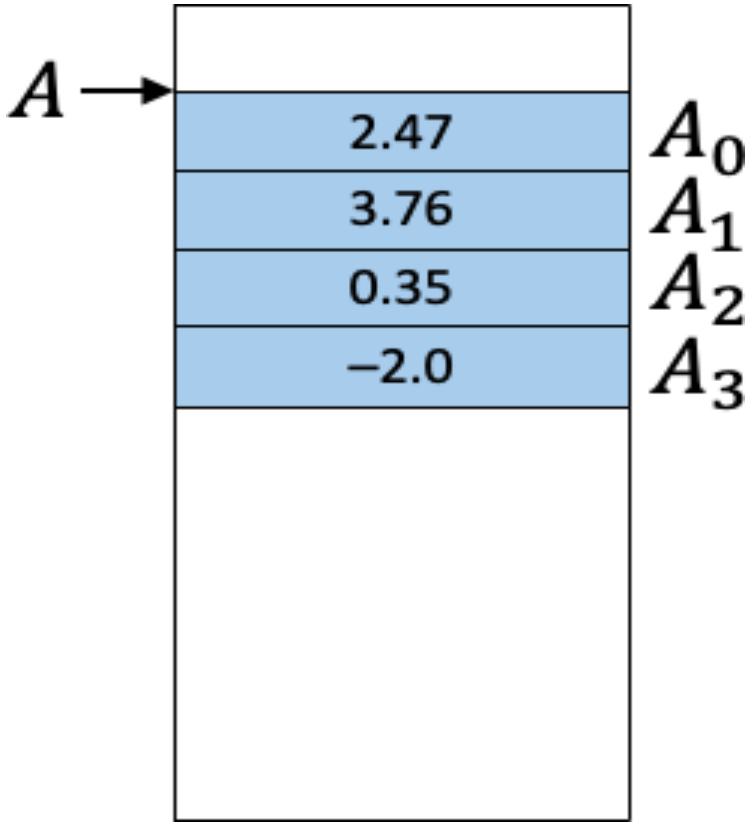


Un **puntero** es una variable cuyo **valor** es la **dirección** de memoria de otra variable.

Arreglos

Un **arreglo** es una secuencia de celdas que tiene largo fijo:

- Cada celda es del mismo tamaño
- Almacenan valores del mismo tipo



Se almacena en memoria de manera contigua

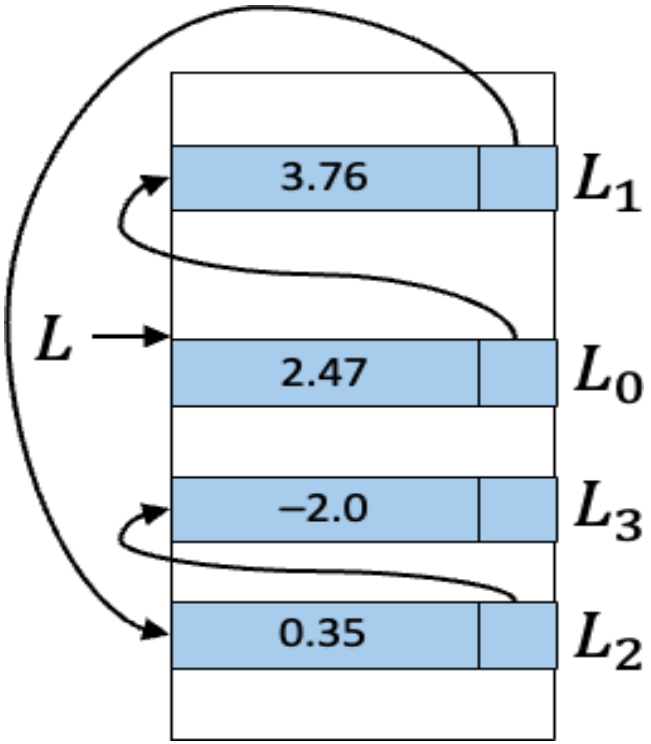
	2.47	**3.76**	**0.35**	**−2.0**	

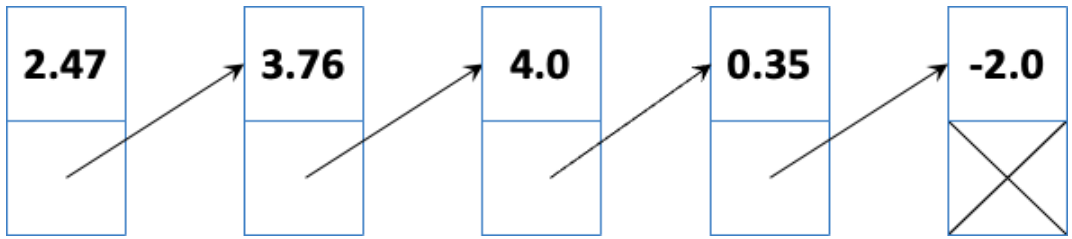
Listas Ligadas

Una lista ligada es una secuencia de celdas que tiene largo variable

- Cada celda es del mismo tamaño

Se almacena en memoria de manera aleatoria, utilizando **punteros**





Diccionarios

Un **diccionario** es una estructura de datos con las siguientes operaciones:

- 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 asociación llave-valor

Diccionarios

El principal objetivo de los diccionarios es la búsqueda eficiente de llaves

El objetivo secundario es inserción/modificación eficiente de pares llave-valor

Por esto, el orden de las llaves deja de ser relevante.

¿Podemos buscar e insertar mas rápido si nos olvidamos de mantener el orden?

Diccionarios

El principal objetivo de los diccionarios es la búsqueda eficiente de llaves

El objetivo secundario es inserción/modificación eficiente de pares llave-valor

Por esto, el orden de las llaves deja de ser relevante.

¿Podemos buscar e insertar mas rápido si nos olvidamos de mantener el orden?

Diccionarios

Para motivar nuestra siguiente estructura, consideremos un escenario ideal:

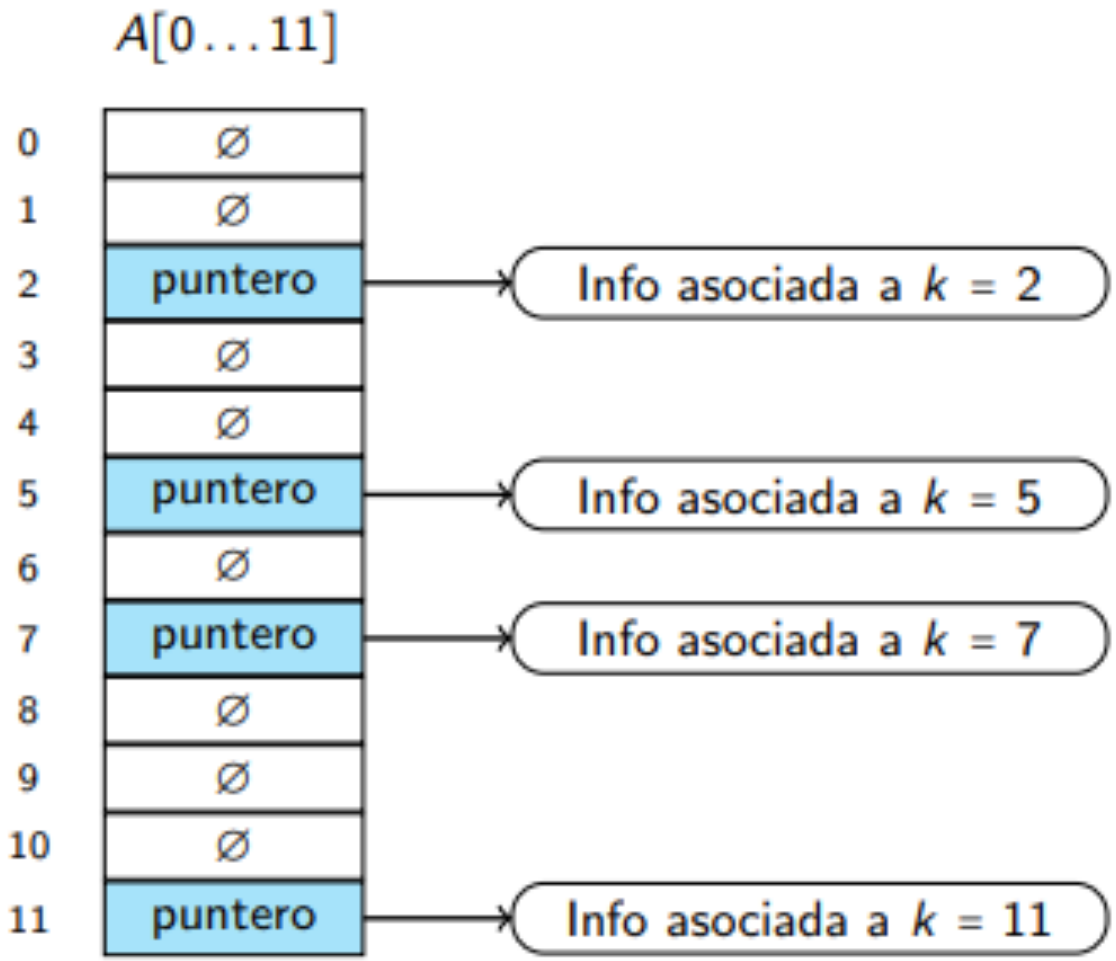
Conjunto de llaves posibles K = {0, . . . , 11} fijo y conocido. Dada una llave k ∈ K, interesa saber si esta se encuentra asociada a un valor en la EDD. Este escenario se puede manejar con la siguiente EDD básica:

Arreglo A[0 . . . 11] iniciado con ø en cada celda

No almacenamos las llaves en las celdas, sino un puntero al valor asociado a la llave k en A[k].

Es decir, A[k] = ø ⇔ no hay valor asociado a k en A

Diccionarios



Diccionarios

El escenario de llaves K puede ocurrir en aplicaciones prácticas:

- Rango de valores razonable
- Llaves siempre naturales (para ser usadas como índices de arreglos)

Ejemplo:

En la universidad hay aproximadamente 25.000 estudiantes este año. Asignamos un $k \in \{0, \ldots, 24.999\}$ a cada estudiante y usamos cada natural como índice del arreglo A.

Diccionarios

Cada estudiante ya posee un rut único:

- Rango de rut's abarca hasta el 25.000.000
- Cantidad de estudiantes mucho menor (25.000)
- Problema: solo 1/1000 celdas del arreglo A indexado por ruts estarán ocupadas

No solo los ruts son llaves posibles:

- Números de teléfono
- Patentes de vehículos
- ...

¿Como nos acercarnos a un conjunto de llaves K razonable?

Con **Tablas de Hash**

Tablas de Hash

Funciones de Hash

Dado un espacio de llaves K y un natural m > 0, una función de hash se define como

$$h : K \rightarrow \{0, \ldots, m - 1\}$$

Dado $k \in K$, llamaremos valor de hash de k a la evaluación h(k). Notemos que:

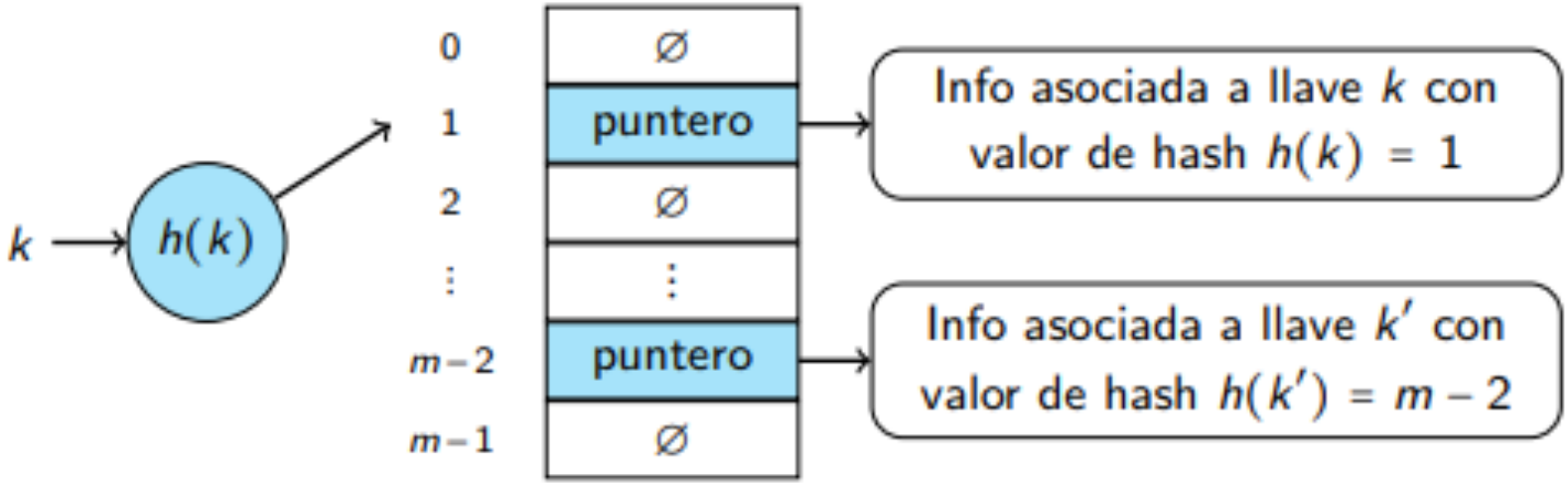
- Una función de hash nos permite mapear un espacio de llaves a otro mas pequeño (con m razonable)
- Una función de hash no necesariamente es inyectiva,
- Si m < | K | , no puede ser inyectiva.
- En la práctica, m << | K |

Tablas de Hash

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

$$h : K \rightarrow \{0, \ldots, m - 1\}$$

Diremos que tal A es de tamaño m.



Tablas de Hash

El ejemplo ideal que estudiamos es una tabla de hash.
La función de hash es:

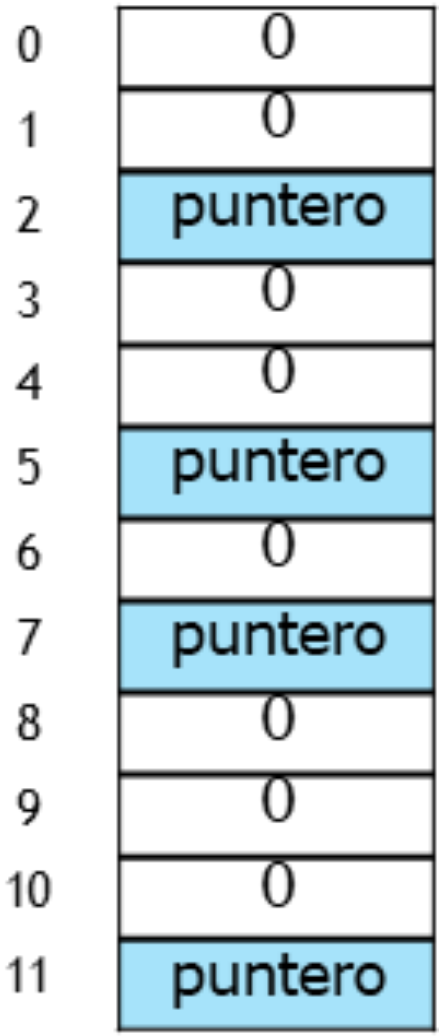
$h : K \rightarrow K$ dada por $h(k) = k$

Las operaciones de diccionario son sencillas:

```
IdentityHashSearch (A, k):
        return A[k]

IdentityHashInsert (A, k, v ):
  A[k] = v

IdentityHashDelete (A, k):
  A[k] = 0
```



Tablas de Hash

Usar la misma estrategia para hashing general seria:

```
HashSearch  (A, k):
  return A[h(k)]

HashInsert  (A, k, v ):
  A[h(k)] = v

HashDelete  (A, k):
  A[h(k)] = ø
```

Pero sabemos que h no necesariamente es inyectiva

Es decir, puede ocurrir una colisión $h(k1) = h(k2)$ para $k1 \neq k2$

Tablas de Hash

Para ejemplificar el problema de las colisiones, consideremos la siguiente función de hash:

$$h(k) = k \bmod m$$

Se le conoce como hashing modular y corresponde al resto al dividir k entre m. Notemos que $h(k) \in \{0, \ldots, m-1\}$ para todo k. Todas las llaves con el mismo resto al dividir entre m generan una colisión, i.e.

$h(k1) = h(k2) \Leftrightarrow k1 \equiv m\ k2$

Ejemplo. Tomando m = 100 y K = {0, . . . , 999}, el hashing modular cumple:

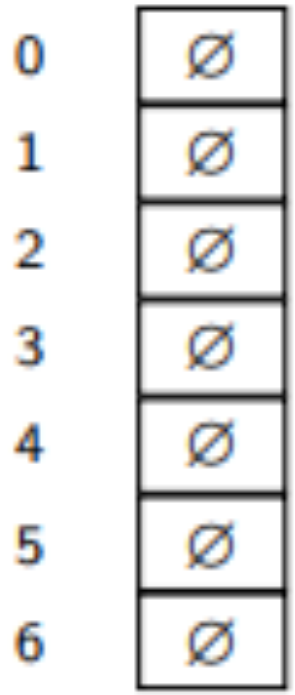
$h(12) = h(112) = \ldots = h(912) = 12$

$h(18) = \ldots = h(918) = 18$

Tablas de Hash - Inserciones

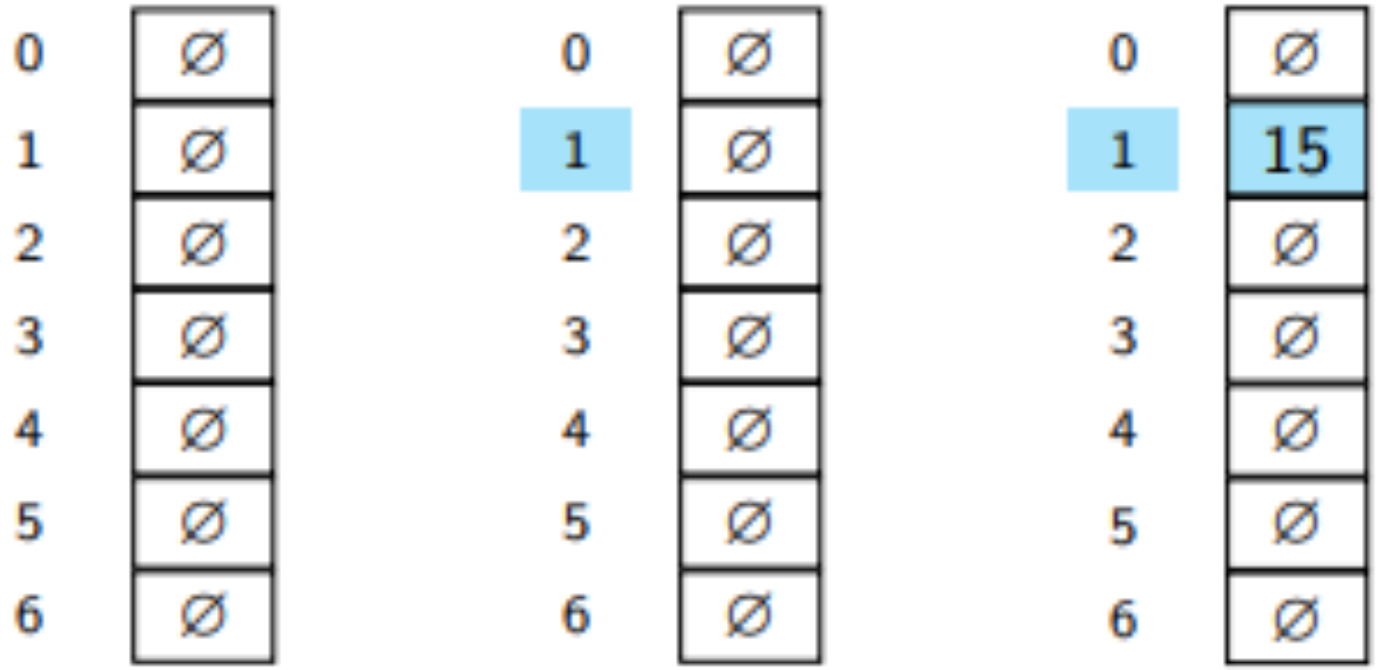
Usaremos la función de hashing modular para experimentar con inserciones:

Consideremos m = 7. Insertemos la llave 15 en la siguiente tabla de hash. Su valor de hash es h(15) = 15 mod 7 = 1



Tablas de Hash - Inserciones

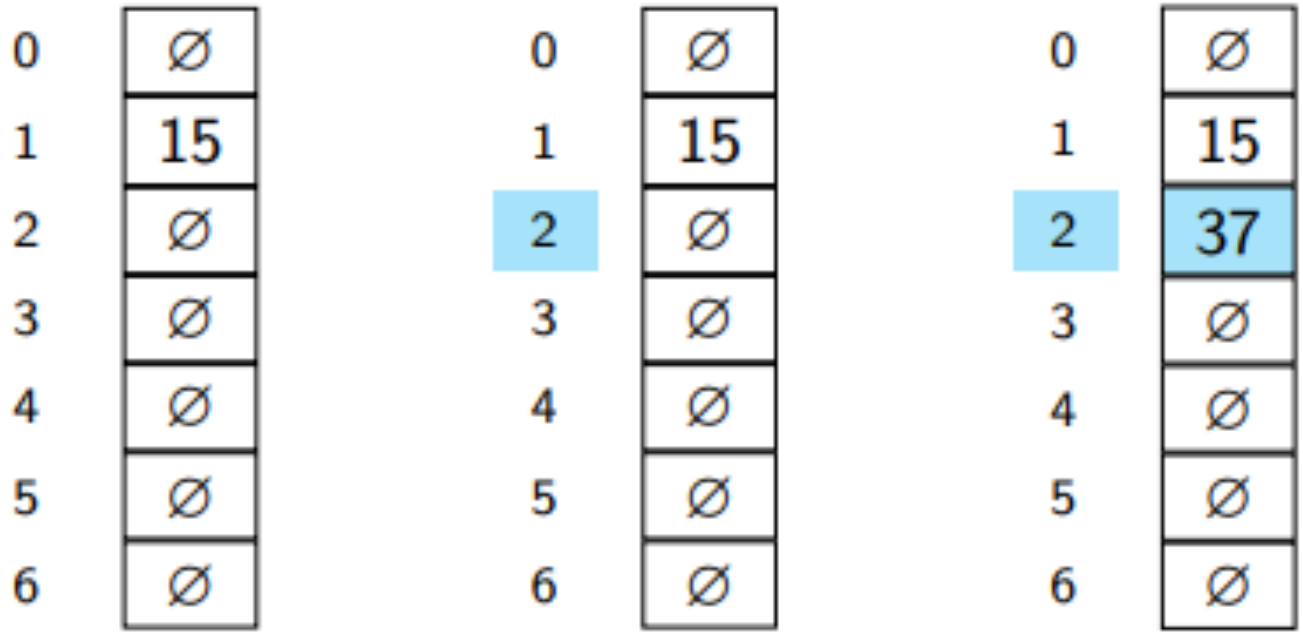
La posición $h(15) = 1$ esta libre y guardamos la llave



Tablas de Hash - Inserciones

Ahora insertamos la llave 37:

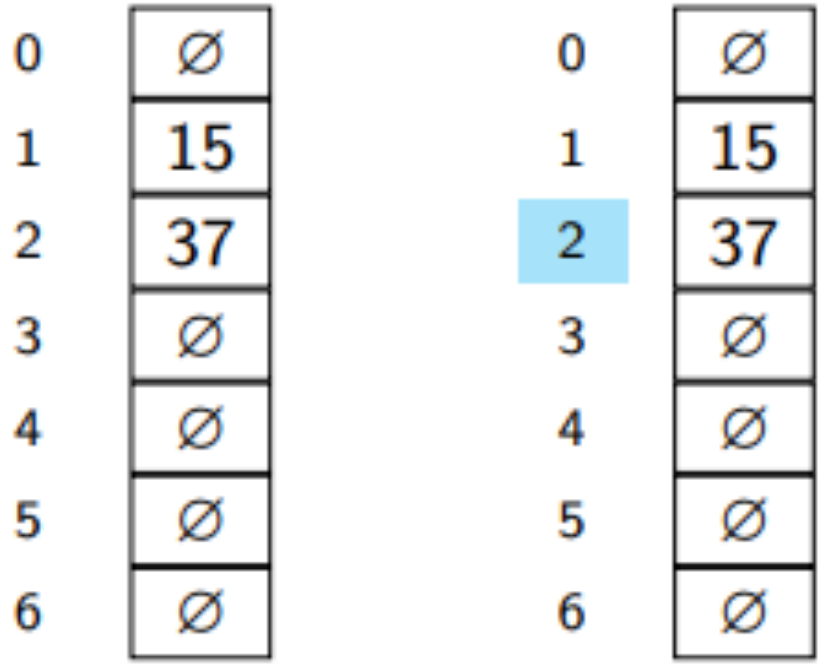
Su valor de hash es h(37) = 37 mod 7 = 2



Tablas de Hash - Inserciones

Ahora insertamos la llave 51

Su valor de hash es h(51) = 51 mod 7 = 2

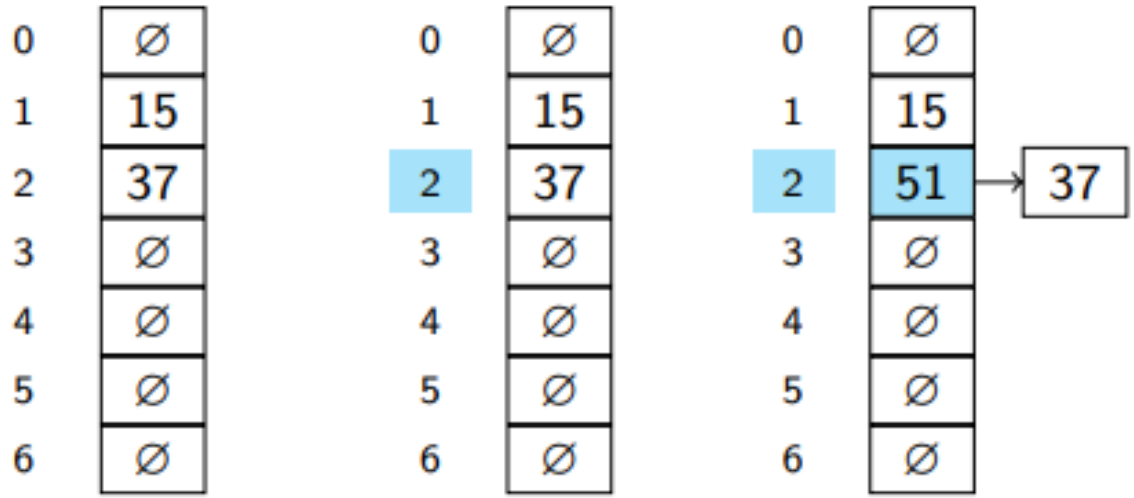


¿Que hacemos con la colisión?

Tablas de Hash - Inserciones

Primera propuesta: encadenamiento.

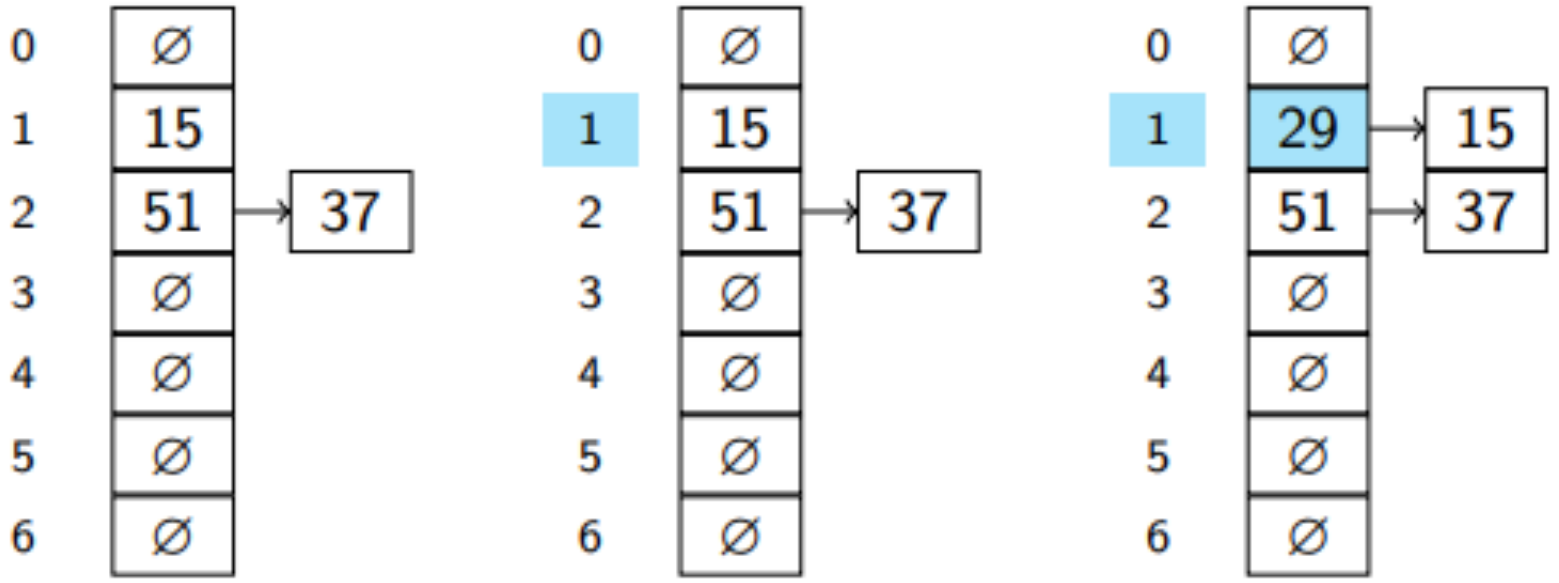
- Cada valor guardado es un nodo de una lista ligada
- Cada colisión agrega un nodo al principio/final de la lista



Tablas de Hash - Inserciones

Al insertar la llave 29 seguimos la misma idea

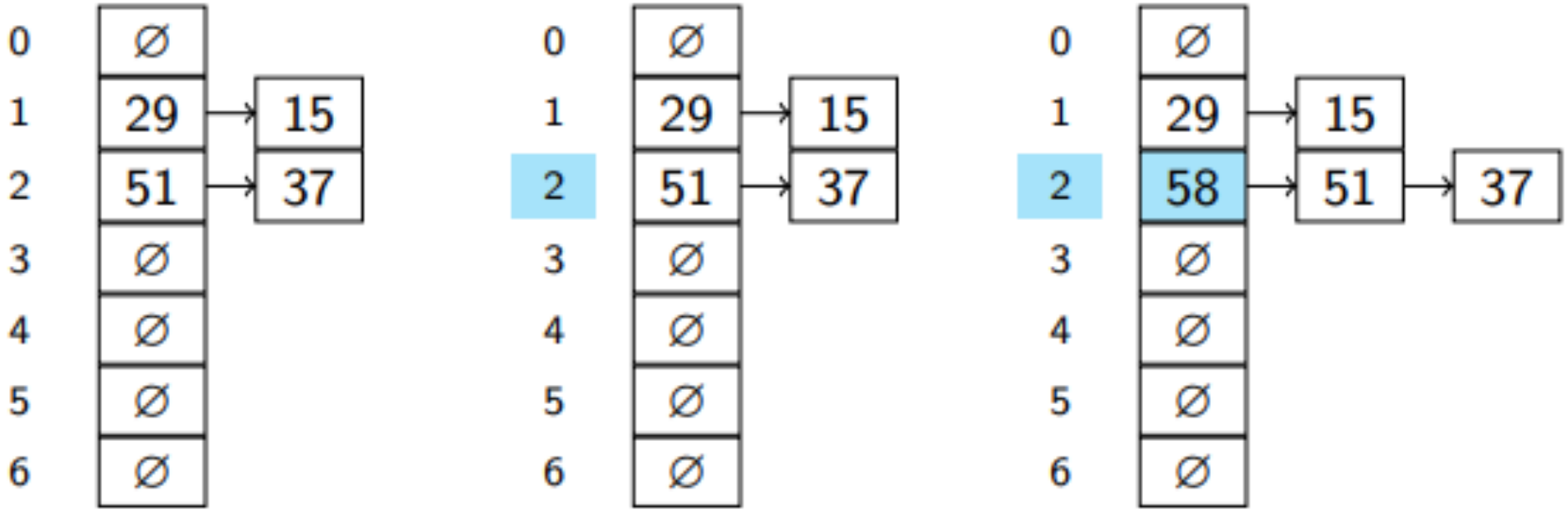
Su valor de hash es h(29) = 29 mod 7 = 1



Tablas de Hash - Inserciones

Al insertar la llave 58 seguimos la misma idea

Su valor de hash es h(58) = 58 mod 7 = 2



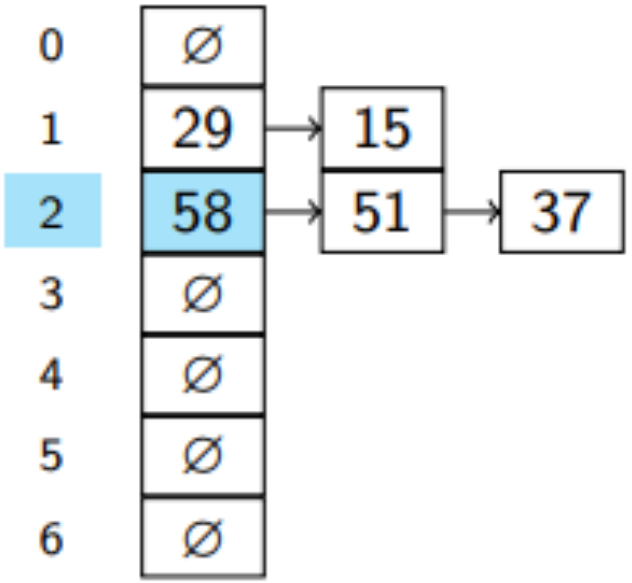
Tablas de Hash - Inserciones

Las operaciones de diccionario involucran la lista ligada A[h(k)]

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)]

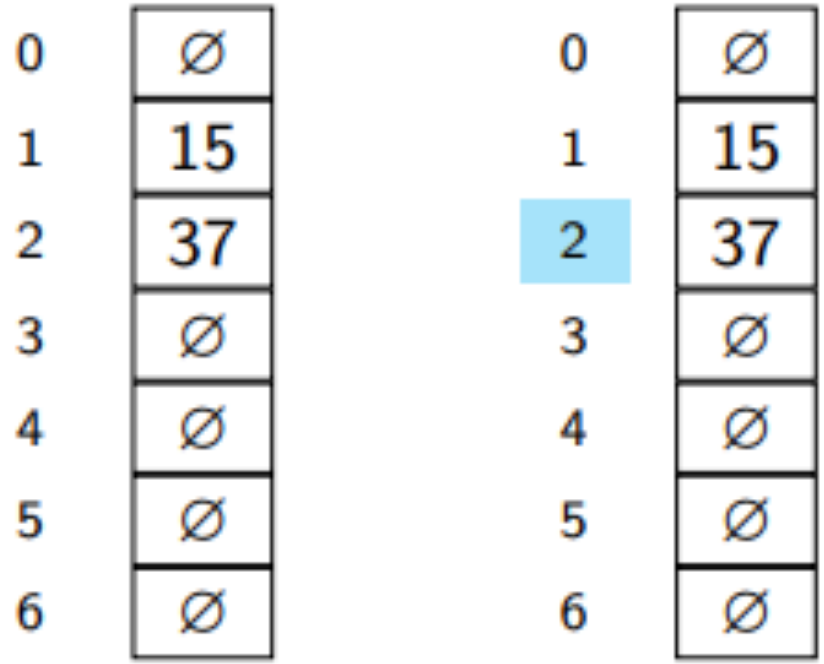


La complejidad de estas operaciones depende de que tan largas sean las listas.
Una buena función de hash repartiría las llaves de manera mas o menos homogénea

Tablas de Hash - Inserciones

Volvamos a la inserción con colisión del 51

Su valor de hash es h(51) = 51 mod 7 = 2



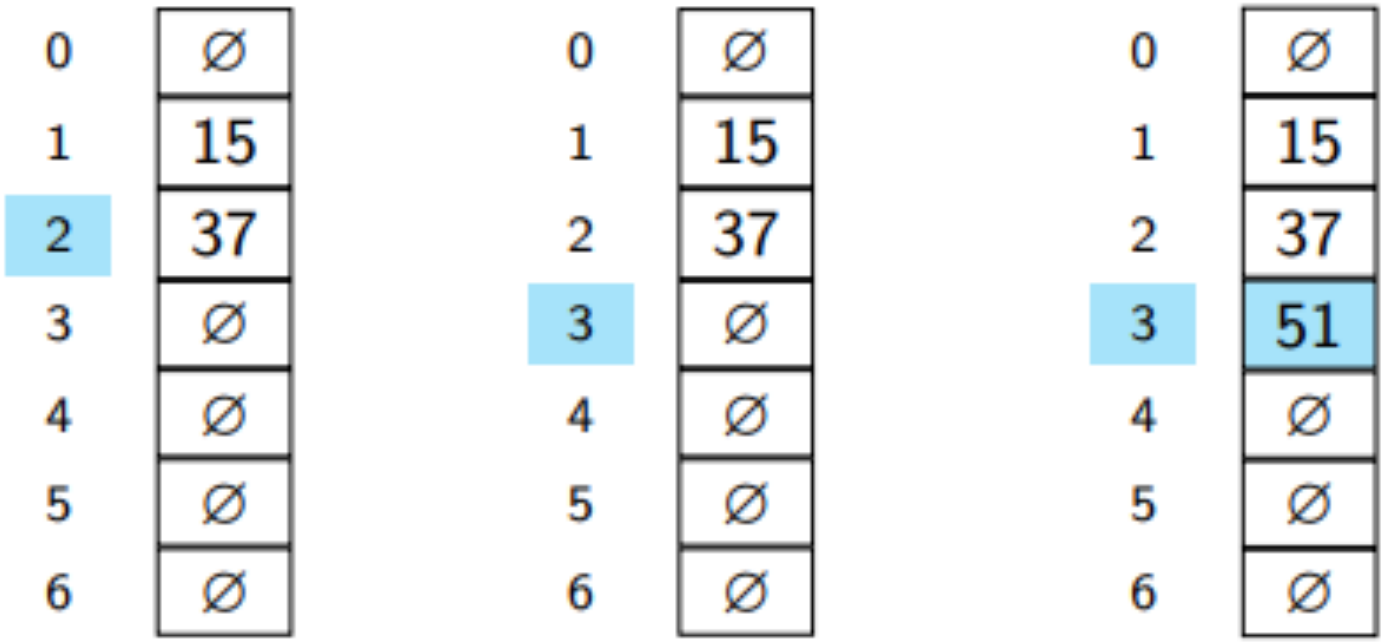
¿Alguna alternativa al encadenamiento?

Tablas de Hash - Inserciones

Segunda propuesta: **direccionamiento abierto**

- Buscamos sistemáticamente una celda vacía
- Puede producir nuevas colisiones no previstas por h

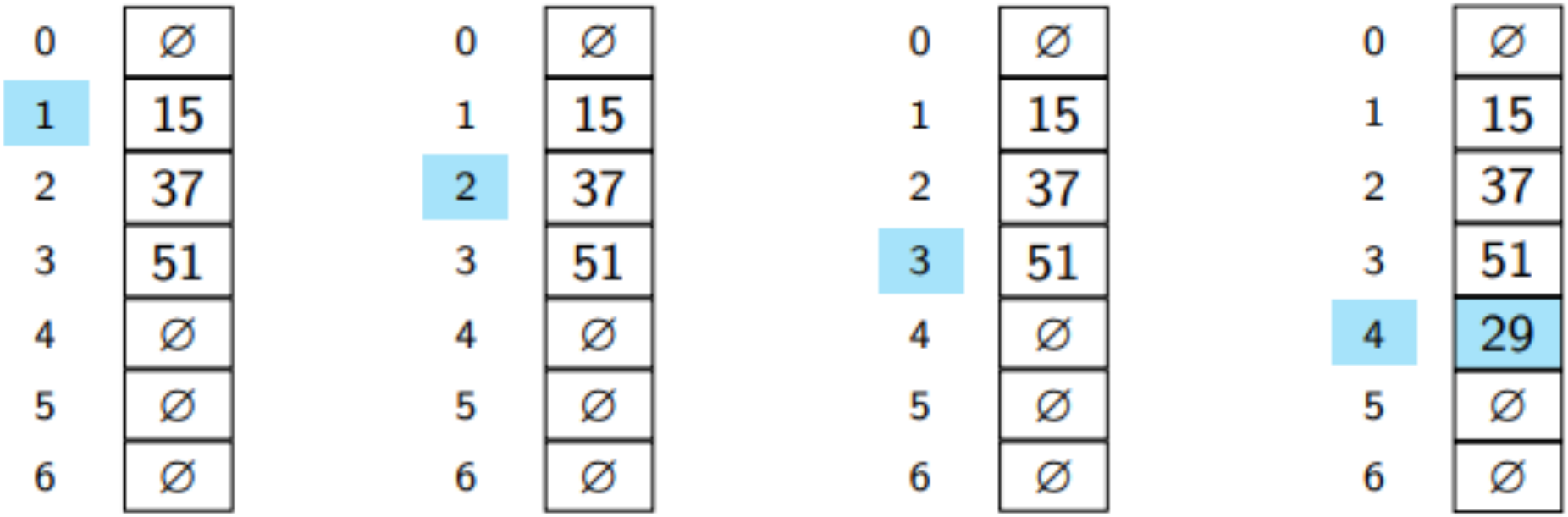
Una forma de buscar: el **sondeo lineal** inserta en la primera celda vacía a la derecha de la colisión.



Tablas de Hash - Inserciones

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

Su valor de hash es h(29) = 29 mod 7 = 1

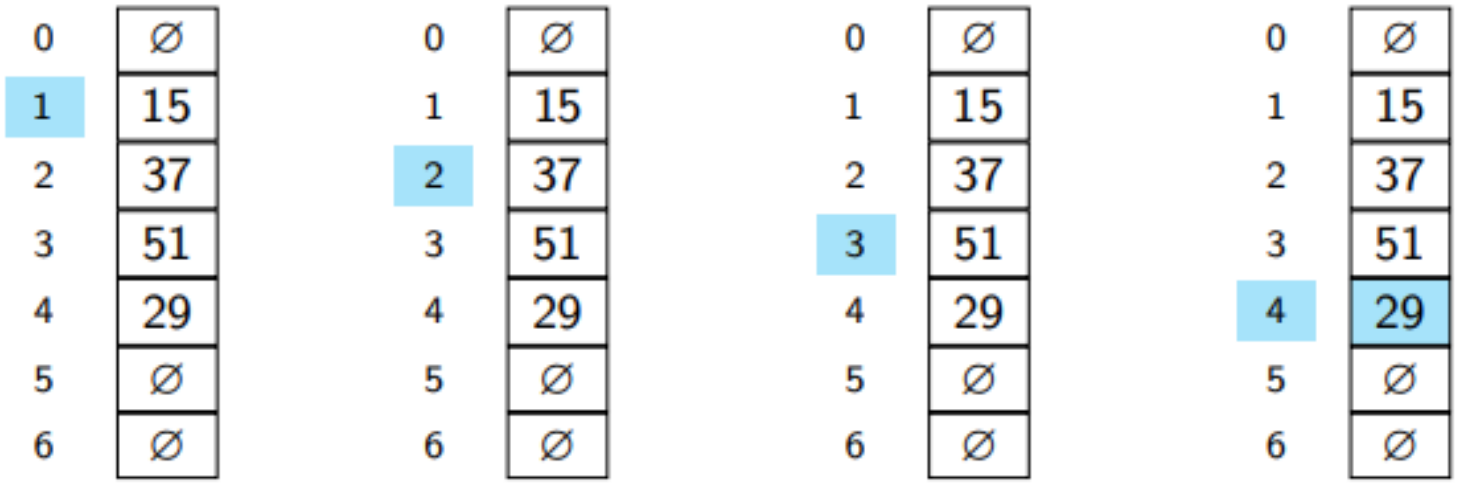


Tablas de Hash – Búsqueda

Si las inserciones son con sondeo lineal, la búsqueda debe tenerlo en cuenta:

- No necesariamente k esta guardado en A[h(k)]
- Debemos revisar esa celda, y si no corresponde, buscar hacia adelante

Por ejemplo, al buscar la llave 29 comenzamos la búsqueda en la pos. 1:



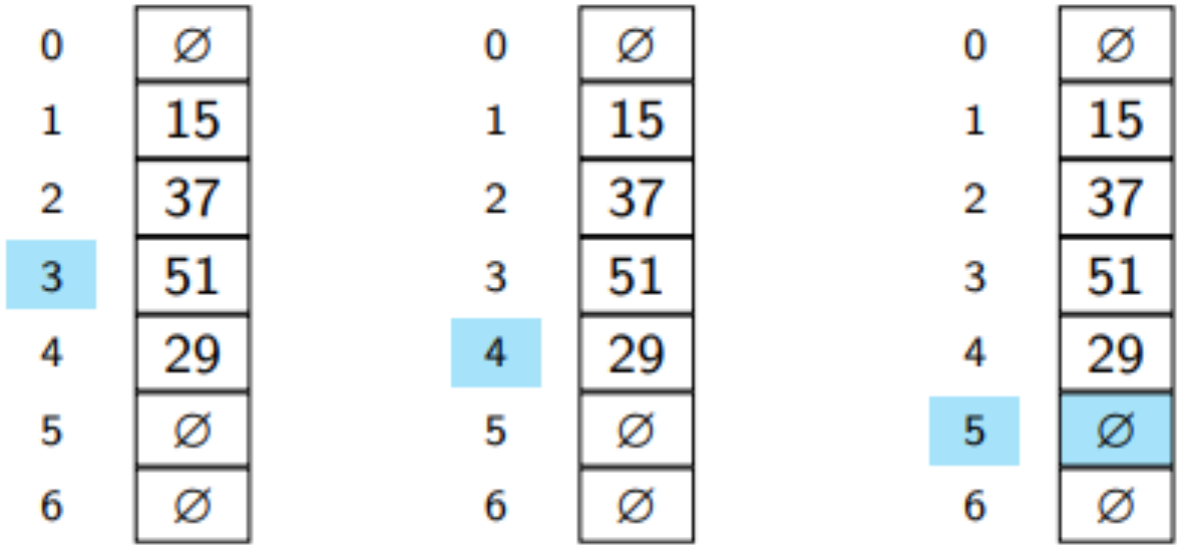
La búsqueda sigue la misma secuencia que la inserción

Tablas de Hash - Búsqueda

¿Como detectamos si la llave no está?

- Comenzamos la búsqueda en A[h(k)]
- Si al buscar a la derecha llegamos a un ∅ significa que no esta

Por ejemplo, al buscar la llave 10, tal que h(10) = 10 mod 7 = 3



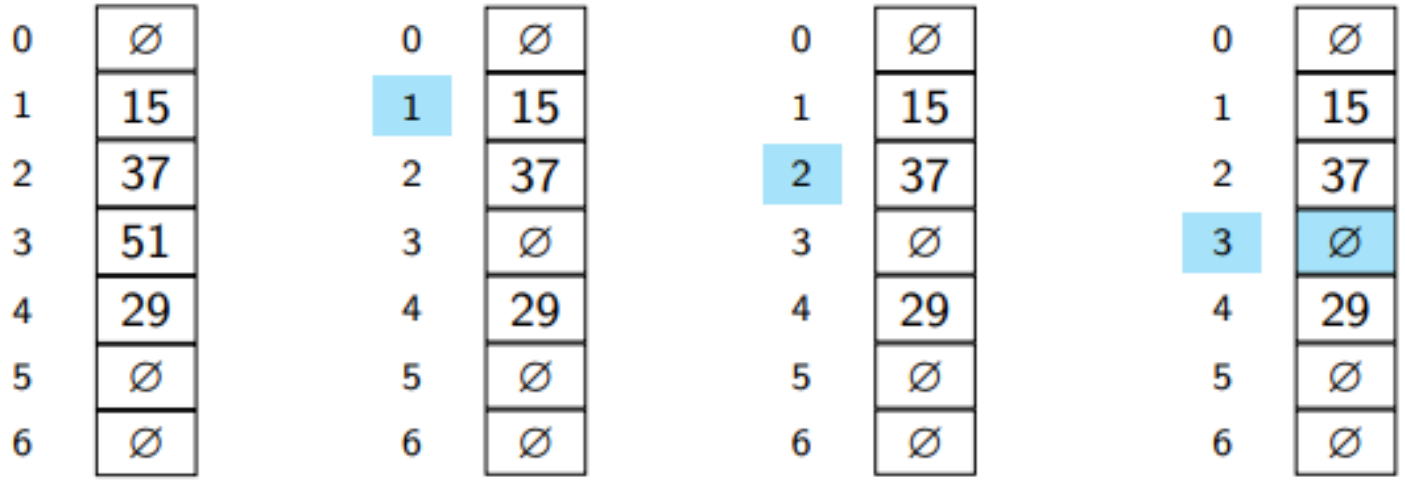
Concluimos que 10 no está almacenada

Tablas de Hash - Eliminar

Para eliminar llaves guardadas tenemos un problema

- Si borramos una llave, la reemplazamos por ∅

Por ejemplo, si borramos el 51 y buscamos el 29 con h(29) = 29 mod 7 = 1:



Concluimos que 29 no esta almacenado...

Si necesitamos eliminación, es mejor usar encadenamiento

Tablas de Hash – Otros sondeos

Sondeo lineal: Si h(k) = H, para alguna constante d buscamos en

H, H + d, H + 2d, . . .

Se debe cumplir d = 1 o que d y m son primos relativos

Sondeo cuadrático: Si h(k) = H, buscamos en

H, H + 1, H + 4, H + 9, . . .

Doble hashing: Usamos dos funciones de hash h1 y h2 y buscamos en

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

Todos ellos presentan el problema de la eliminación

Tablas de Hash – Factor de carga

Dado que las colisiones impactan la tabla, nos interesa medir cuantos datos tenemos almacenados.

Dada una tabla de hash A de tamaño m con n valores almacenados, se define su **factor de carga** como

$$\lambda = n/m$$

El factor de carga es una medida de que tan llena esta la tabla.

Según la estrategia de resolución de colisiones:

Encadenamiento: es aceptable $\lambda \approx 1$

Direccionamiento abierto: $\lambda > 0.5$ resulta en inserciones y búsquedas muy lentas

Tablas de Hash – 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 O(n) para n datos insertados
- No obstante, es infrecuente

Índices

Índices

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

A un índice yo le indico que quiero las tuplas con cierto valor en un atributo (o con un valor en cierto rango)

El índice encontrará rápidamente las tuplas que cumplan con la condición

(Idealmente un índice debe caber en RAM)

Consultas a Optimizar

Consultas **por valor**

```
SELECT *
FROM Table
WHERE Table.value = ‘value’
```

Consultas **por rango**

```
SELECT *
FROM Table
WHERE Table.value >= ‘value’
```

Índices

CREATE INDEX <nombre índice> ON table <atributo>

Las llaves primarias están indexadas por defecto

Índices

Flujo

Supongamos que tengo la tabla:

Usuarios(uid: int PRIMARY KEY,
nombre: text,
edad: int)

en donde tenemos indexada la tabla por la primary key, que es el atributo uid

Índices

Flujo

El índice normalmente es una colección de archivos que puede ir completo en memoria

Si hacemos la consulta:

SELECT * FROM Usuarios WHERE uid = 104

esta se resolverá con el índice, así evitaremos recorrer toda la tabla

Índices

Definiciones

Search Key: es el parámetro con el que busco algo en mi índice (por ejemplo, el uid de una tabla de usuarios)

Data Entry: la tupla que nos retorna el índice después de preguntar por una Search Key en particular

Por ahora:
Data Entry = mi tupla

Ejemplo: Apliquemos EDD a BDD

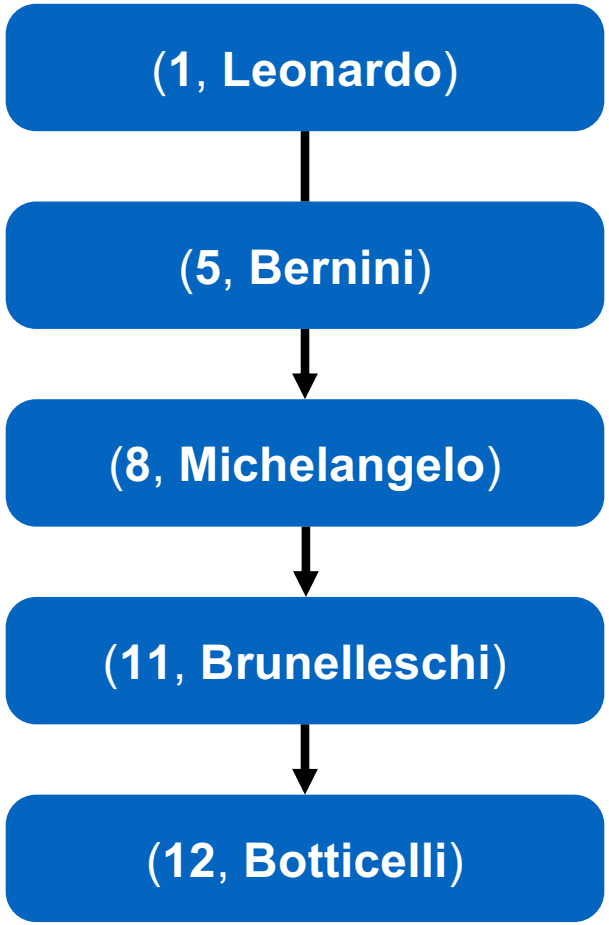
Mi esquema: **Actores(aid, nombre)**

Instancia = una lista de tuplas en Python:

```
t = [
     (1, Leonardo),
     (5, Bernini),
     (8, Michelangelo),
     (11, Brunelleschi),
     (12, Botticelli)
    ]
```

Ejemplo: Apliquemos EDD a BDD

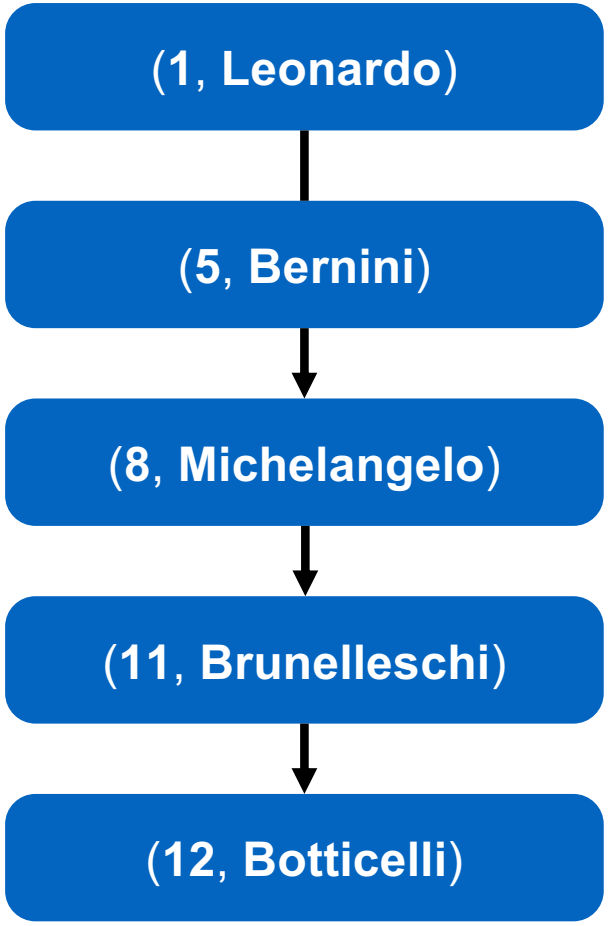
Lista ligada (versión básica)



Ejemplo: Apliquemos EDD a BDD

Lista ligada (versión básica)

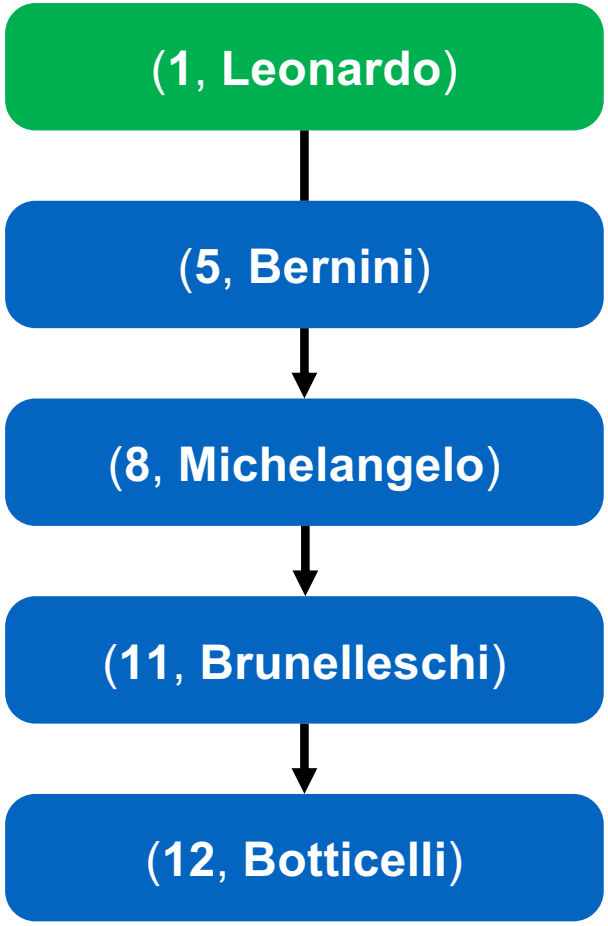
Supongamos que queremos el artista con identificador 11



Ejemplo: Apliquemos EDD a BDD

Lista ligada (versión básica)

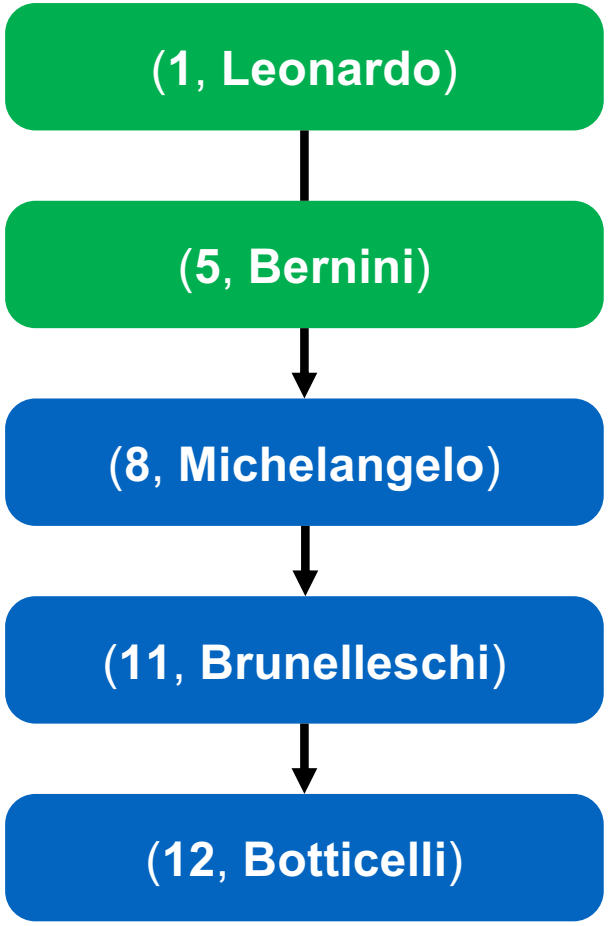
Supongamos que queremos el artista con identificador 11



Ejemplo: Apliquemos EDD a BDD

Lista ligada (versión básica)

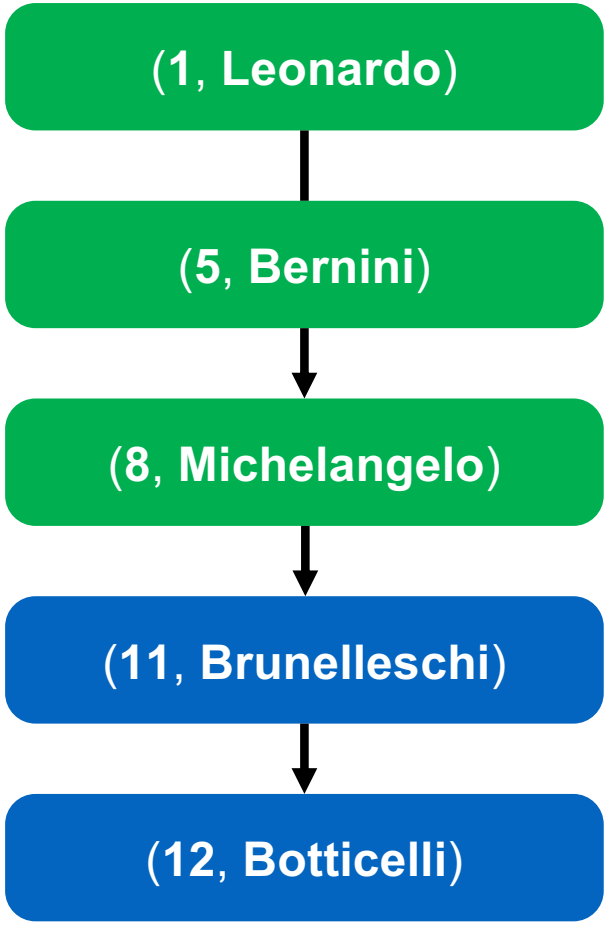
Supongamos que queremos el artista con identificador 11



Ejemplo: Apliquemos EDD a BDD

Lista ligada (versión básica)

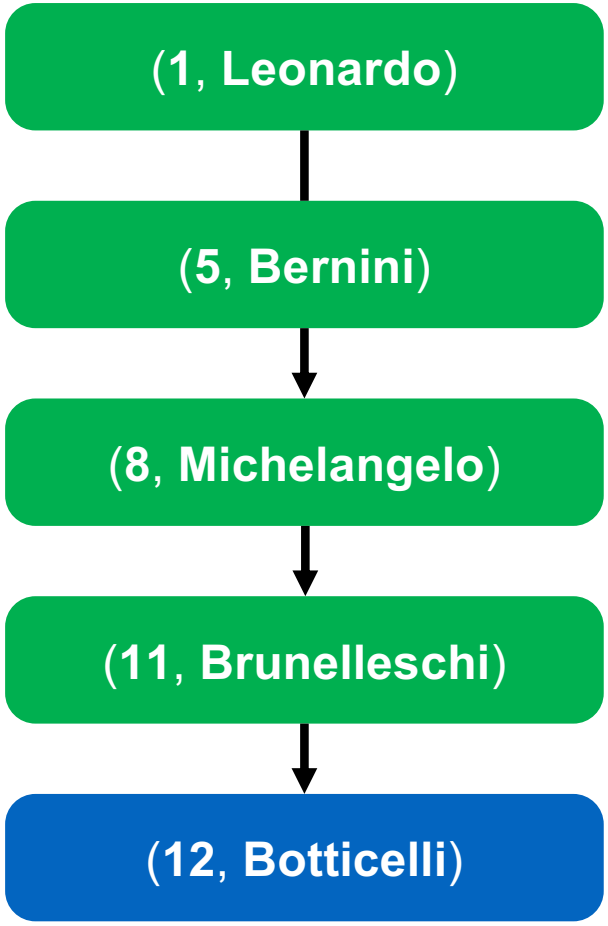
Supongamos que queremos el artista con identificador 11



Ejemplo: Apliquemos EDD a BDD

Lista ligada (versión básica)

Supongamos que queremos el artista con identificador 11



Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

Iniciamos un número fijo de casilleros (buckets)

Cada tupla va a parar a un casillero según su id

Para esto utilizamos una función de **hash.**

Una función de hash es simplemente una función h que lleva elementos de un conjunto U a otro conjunto M (los casilleros/buckets)

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

En el ejemplo, podemos tener una función de **hash** que recibe el id del artista y retorna un número de casillero

Si tenemos 5 casilleros, una posible función es módulo 5 (retorna el resto de la división por 5)

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash



Función hash:
módulo 5 (aid % 5)

Bucket	
0	
1	
2	
3	
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

(1, **Leonardo**)



Función hash:
módulo 5 (aid % 5)

Bucket	
0	
1	
2	
3	
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

(5, **Bernini**)



Función hash:
módulo 5 (aid % 5)

Bucket	
0	
1	(1, **Leonardo**)
2	
3	
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash



Función hash:
módulo 5 (aid % 5)

Bucket	
0	(5, Bernini)
1	(1, Leonardo)
2	
3	
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

(8, Michelangelo)

(11, Brunelleschi)

(12, Botticelli)



Función hash:
módulo 5 (aid % 5)

Bucket	
0	(5, Bernini)
1	(1, Leonardo)
2	
3	
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash



Función hash:
módulo 5 (aid % 5)

Bucket	
0	(5, Bernini)
1	(1, Leonardo) (11, Brunelleschi)
2	(12, Botticelli)
3	(8, Michelangelo)
4	

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

Si queremos buscar el artista con identificador 12, ¿cómo lo hacemos ahora? ¿en cuántos pasos lo obtenemos?

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

aid = 12



Función hash:
módulo 5 (aid % 5)

Bucket	
0	(5, Bernini)
1	(1, Leonardo) (11, Brunelleschi)
2	(12, Botticelli)
3	(8, Michelangelo)
4	

Computamos el hash del número 12

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

aid = 12



Función hash:
módulo 5 (aid % 5)

Bucket	
0	(5, Bernini)
1	(1, Leonardo) (11, Brunelleschi)
2	(12, Botticelli)
3	(8, Michelangelo)
4	

Y en 1 paso encontramos el casillero que le corresponde

Ejemplo: Apliquemos EDD a BDD

Tablas de Hash

Notamos que hay colisiones, esto es, dos valores que van a parar al mismo casillero

En general, suponemos que las funciones de hash distribuyen uniforme

También suponemos que hay suficientes casilleros para que la búsqueda se haga en 1 paso (o no mucho más)

Hash Index

Hash Index

La idea es replicar el concepto de las tablas de hash pero en un sistema de bases de datos

Aquí nuestros casilleros serán páginas del disco duro

En cada página caben muchas tuplas

Hash Index

Recordemos que queremos encontrar tuplas para consultas de igualdad de forma rápida

```
SELECT * FROM Artistas A WHERE A.aid = 12
```

En este caso, rápido significa hacer la menor cantidad de I/O posible

Hash Index

Ejemplo

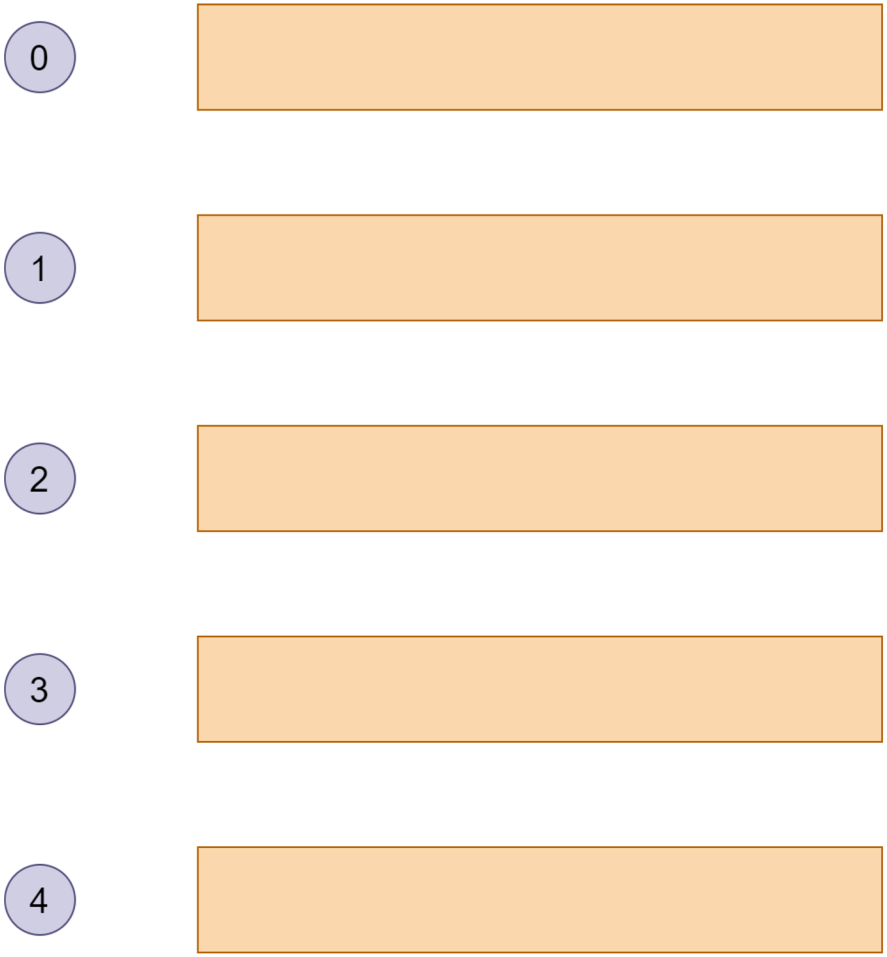
Vamos a retomar el ejemplo de los artistas, pero ahora insertando las tuplas a una base de datos

Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto

Hash Index

Ejemplo

Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

Ejemplo

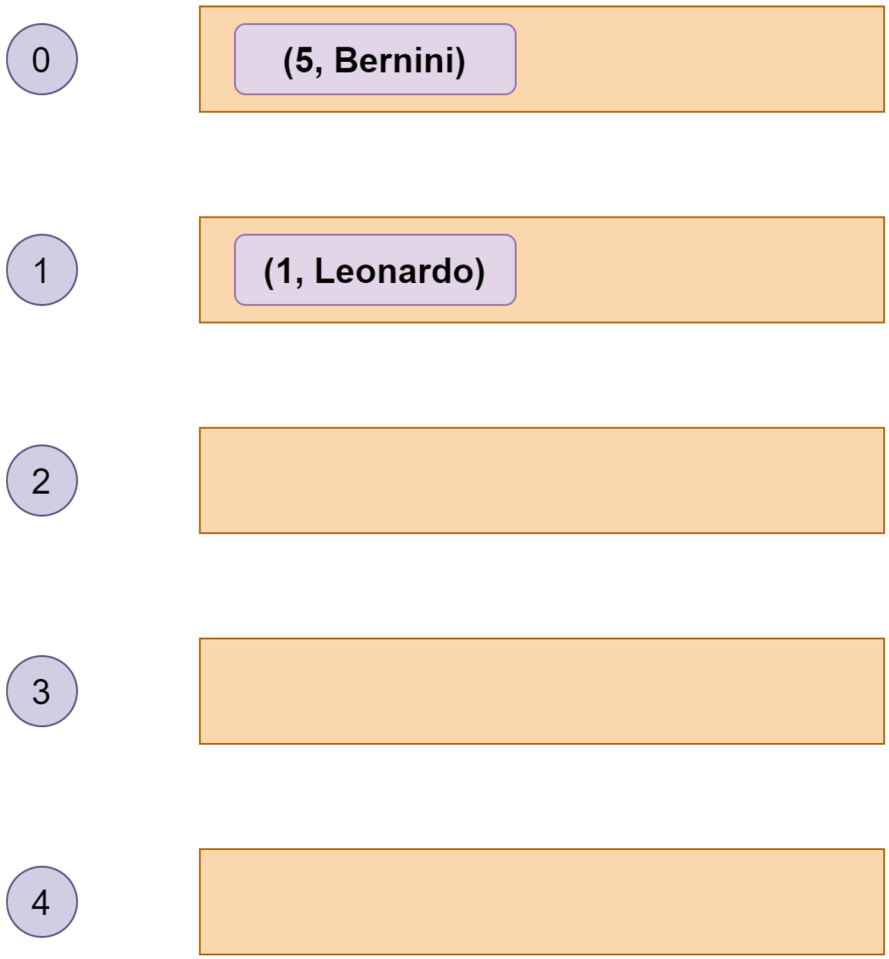
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

Ejemplo

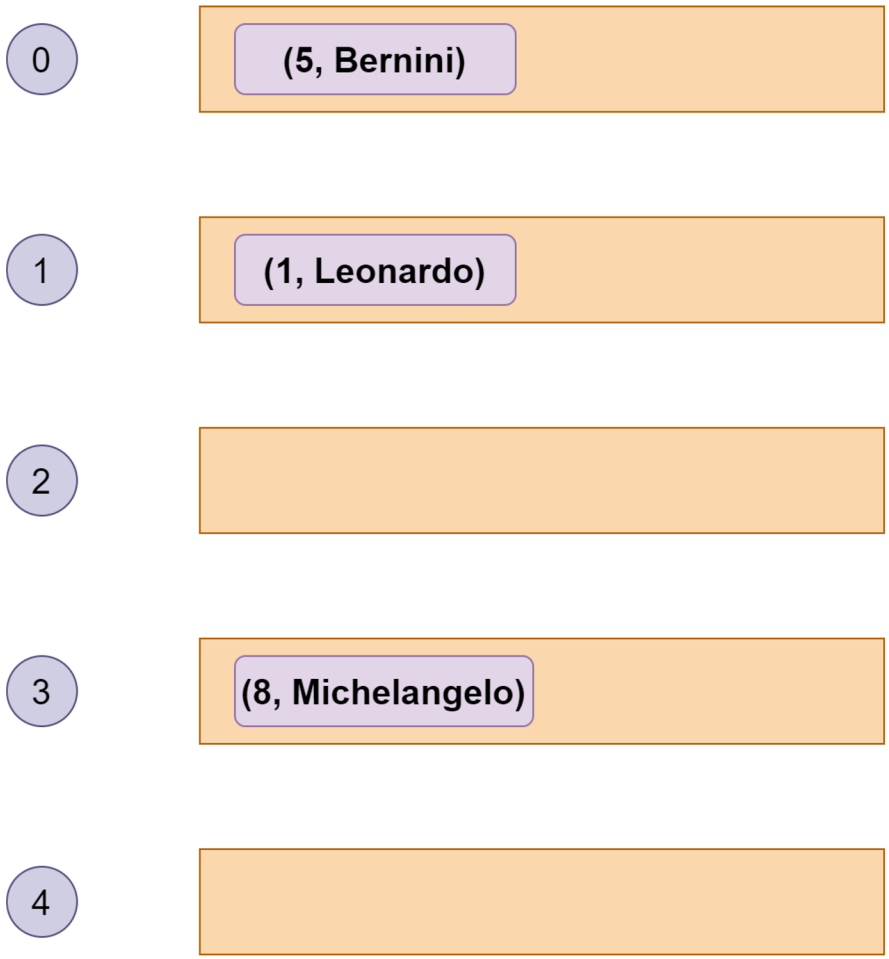
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

Ejemplo

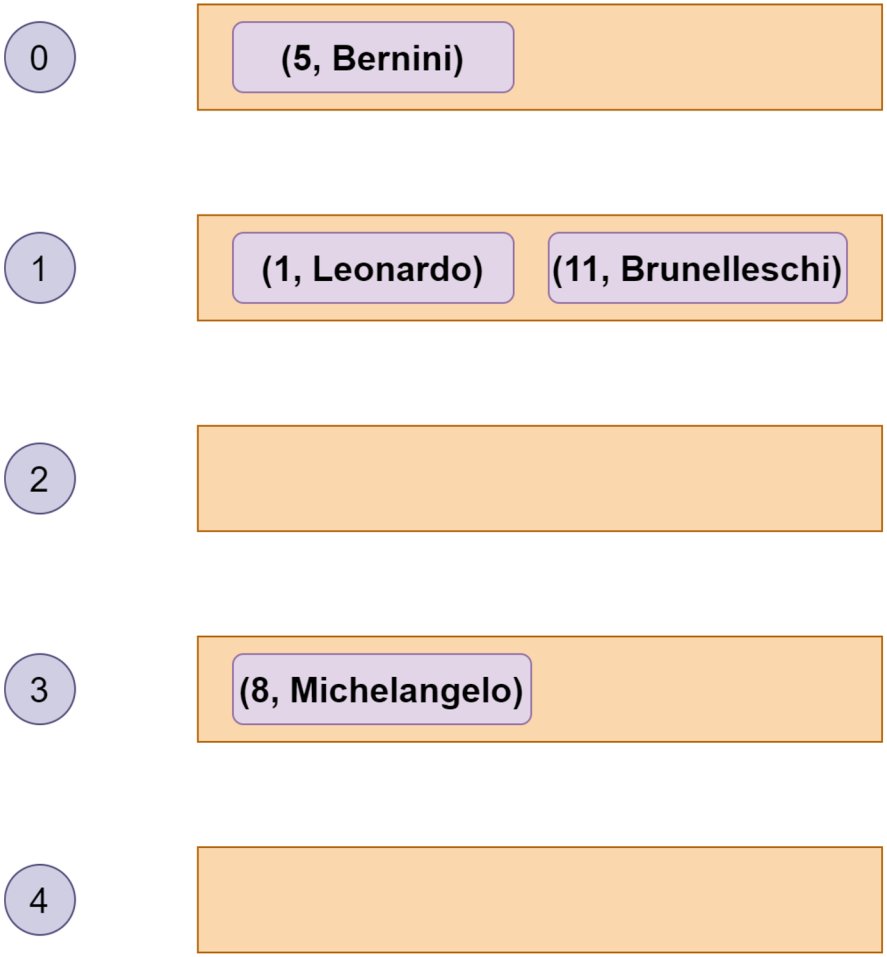
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

Ejemplo

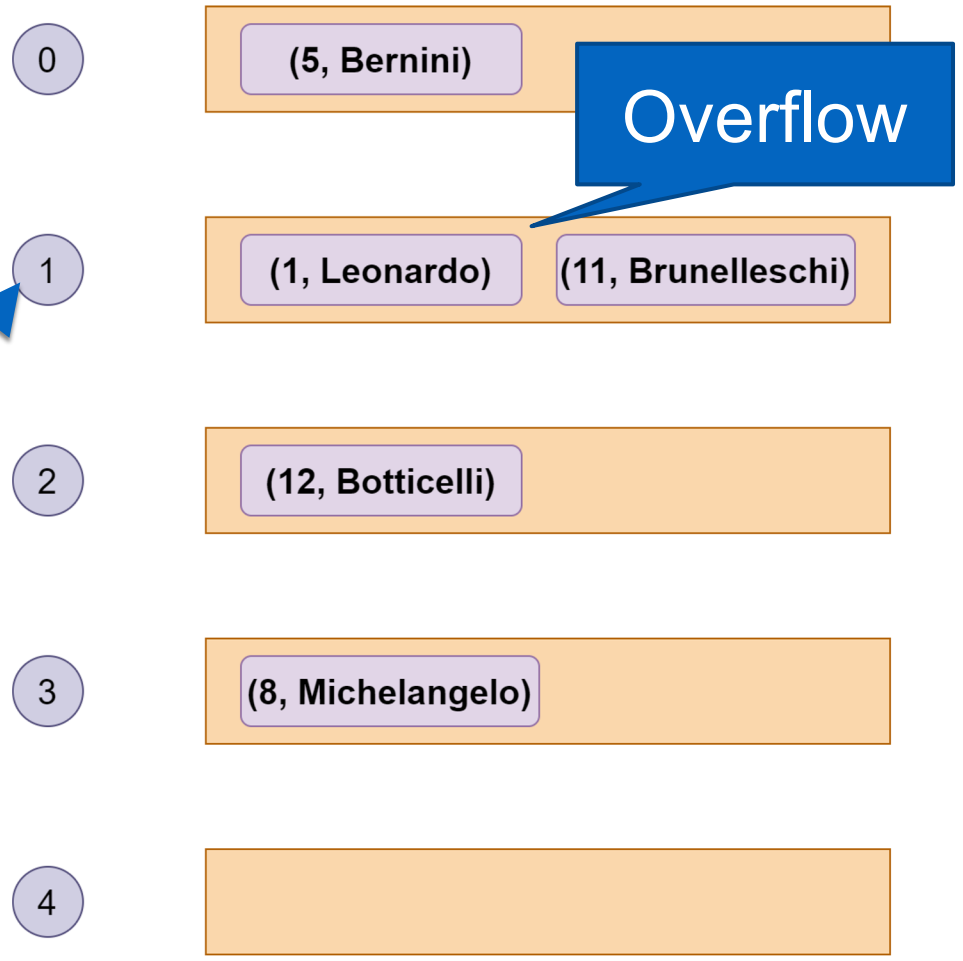
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

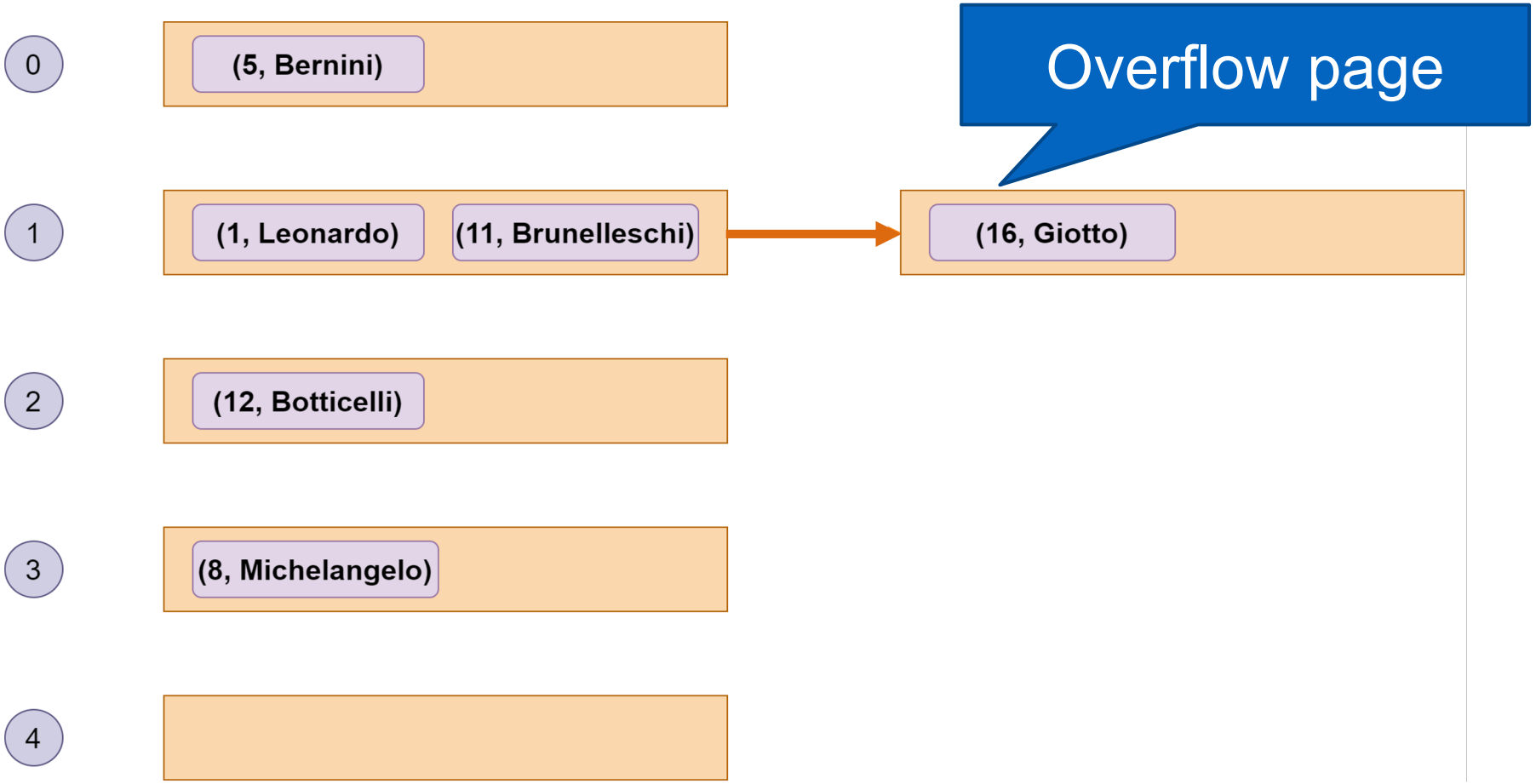
Ejemplo

Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



Hash Index

Ejemplo



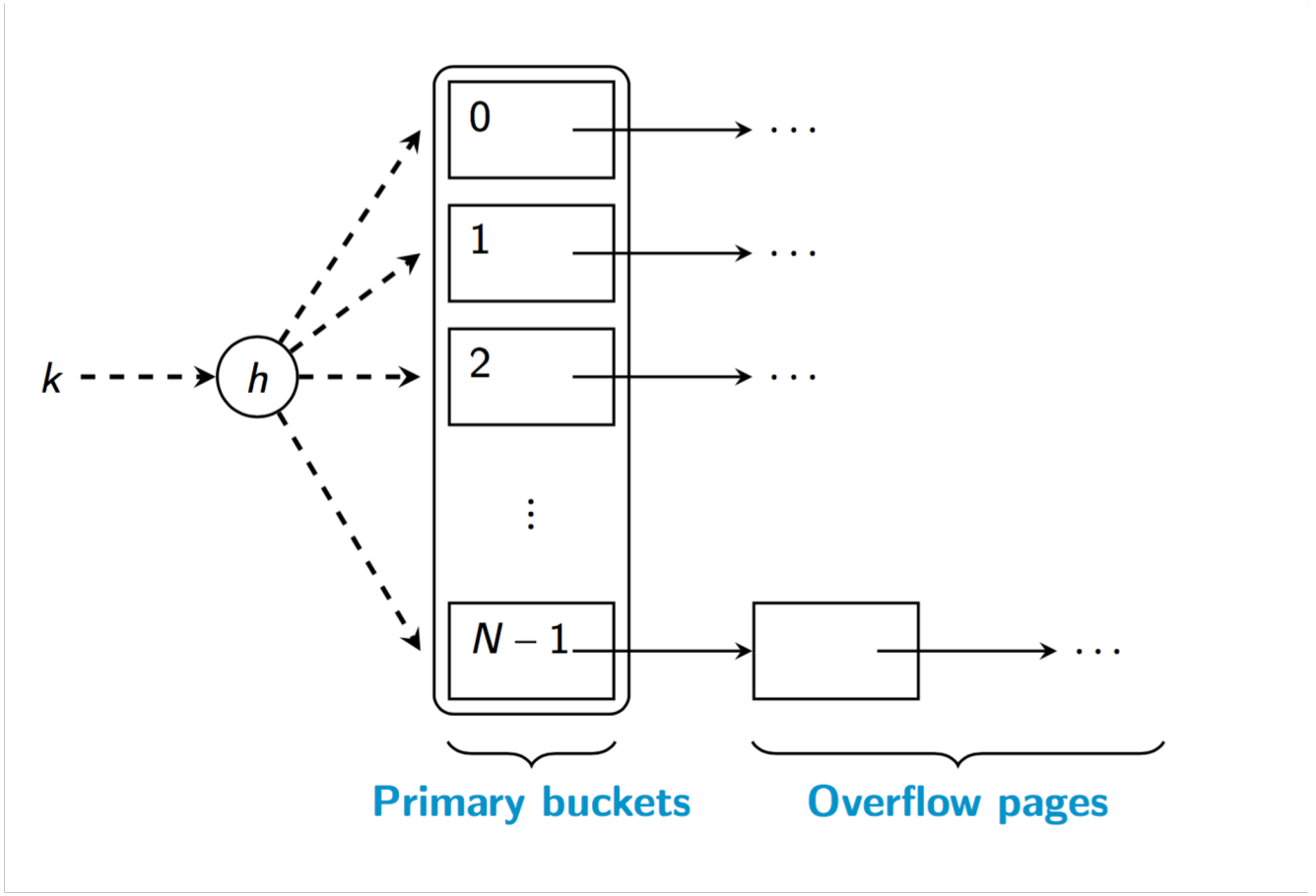
Hash Index

Para una relación **R** y un atributo **A**

- N páginas de disco sirven de **buckets**
- Cada bucket cuenta con una lista ligada de **overflow pages**
- Usamos una función de hash **h** que me indicará que bucket le corresponde a cada valor:

$$\mathbf{h}: \text{dominio}(A) \rightarrow [0, \ldots, N - 1]$$

Hash Index



Hash Index

Para buscar un elemento dada una search key **k**:

- Se computa el valor de **h(k)**
- Se accede al bucket correspondiente
- Se busca el elemento en la página asociada o una de sus overflow pages

costo $O(1)$

Caso ideal: costo 0

Peor caso: costo $O(\lceil \frac{|R|}{B} \rceil)$

Cuando todo va a un casillero!

Hash Index

¿Cuándo **no** nos conviene utilizar un Hash Index?

SELECT * FROM Artistas A WHERE A.aid > 12

Para estimar el número de accesos al disco necesarios para esta consulta, calculamos el número de tuplas que satisfacen la condición, dividido por el número de tuplas por bloque:

$$\left\lceil \frac{n}{B} \right\rceil \quad \text{(o también)} \quad \left\lceil \frac{|R|}{B} \right\rceil$$

Aquí, n es el número de registros donde A.aid > 12 y B es la capacidad de registros por bloque.

Los índices hash son ineficientes para consultas que involucran condiciones de rango, como A.aid > 12, ya que están optimizados para búsquedas de igualdad.

Arboles

Arboles

Un **árbol binario de búsqueda** (ABB) es una estructura de datos que almacena pares (llave, valor) asociándolos mediante punteros según una estrategia recursiva:

1. Un ABB tiene un nodo que contiene una tupla (llave, valor)

2. El nodo puede tener hasta dos ABB's asociados mediante punteros:

- Hijo izquierdo
- Hijo derecho

y que además, satisface la propiedad ABB: las llaves menores que la llave del nodo están en el sub-árbol izquierdo, y las llaves mayores, en el sub-árbol derecho.

Arboles

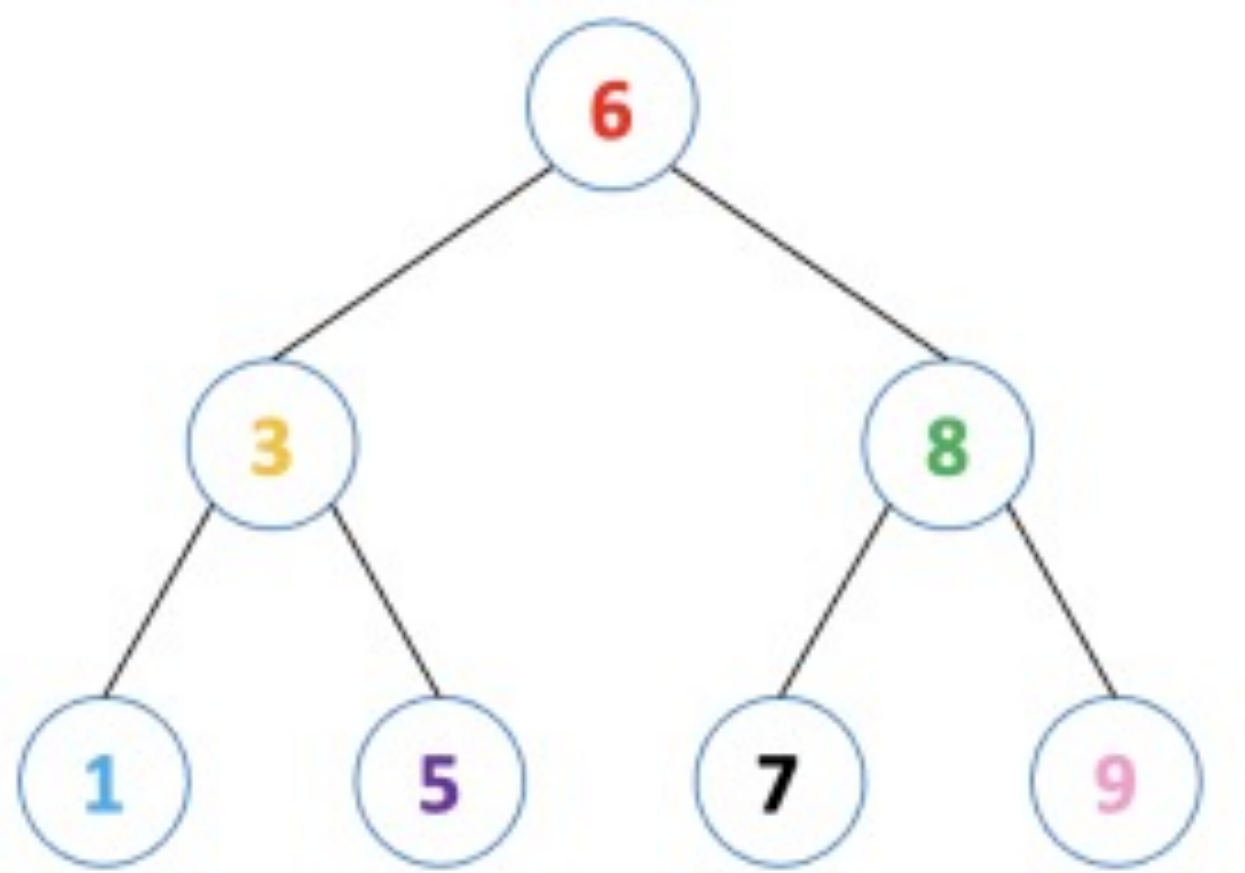
Un árbol binario (de búsqueda o no) cumple que:

- Cada nodo x tiene a lo más un padre x.p
- El nodo sin padre se conoce como raíz
- Cada nodo x tiene hasta dos punteros que (apuntan) a subárboles:

- x.left es un puntero al hijo izquierdo
- x.right es un puntero al hijo derecho
- x.p es puntero al padre (si tiene)

- Un nodo sin punteros descendentes, i.e. sin hijos, se conoce como hoja

Arboles- Ejemplo



B+ Tree Index

B+ Tree Index

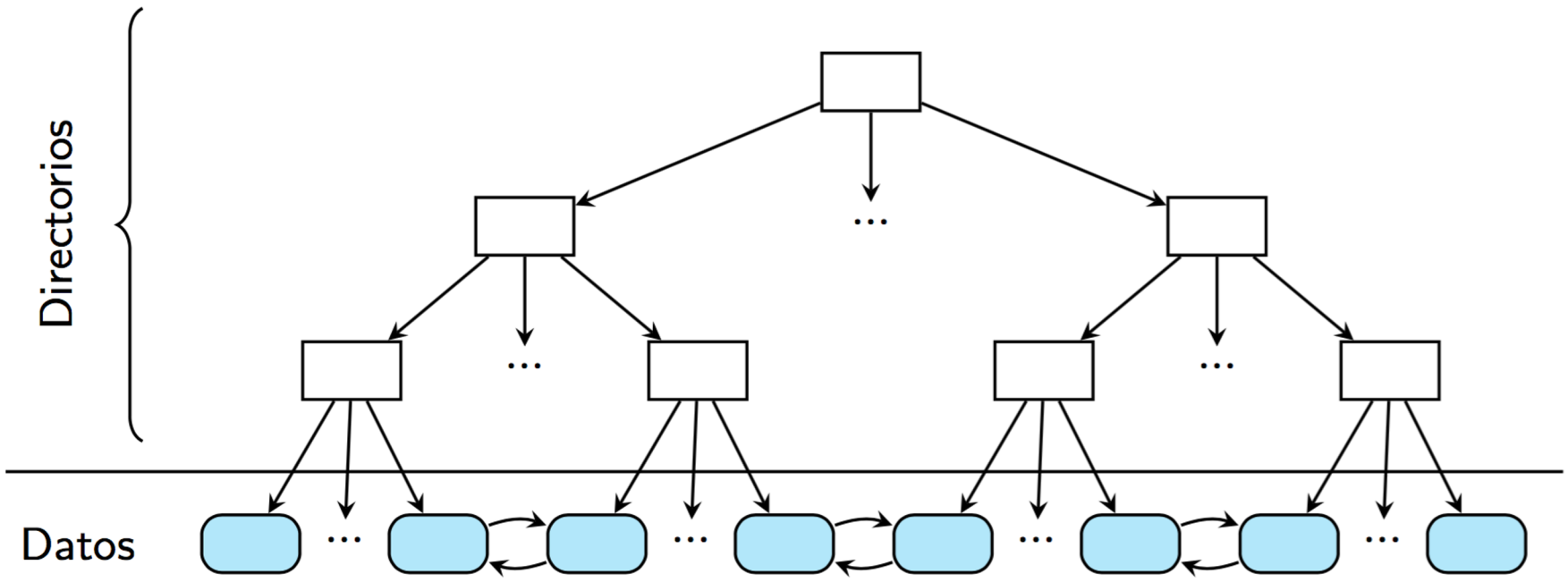
"*B+ Trees are by far the most important access path structure in database and file systems*",
Gray y Reuter (1993).

B+ Tree Index

Es un índice basado en un árbol:

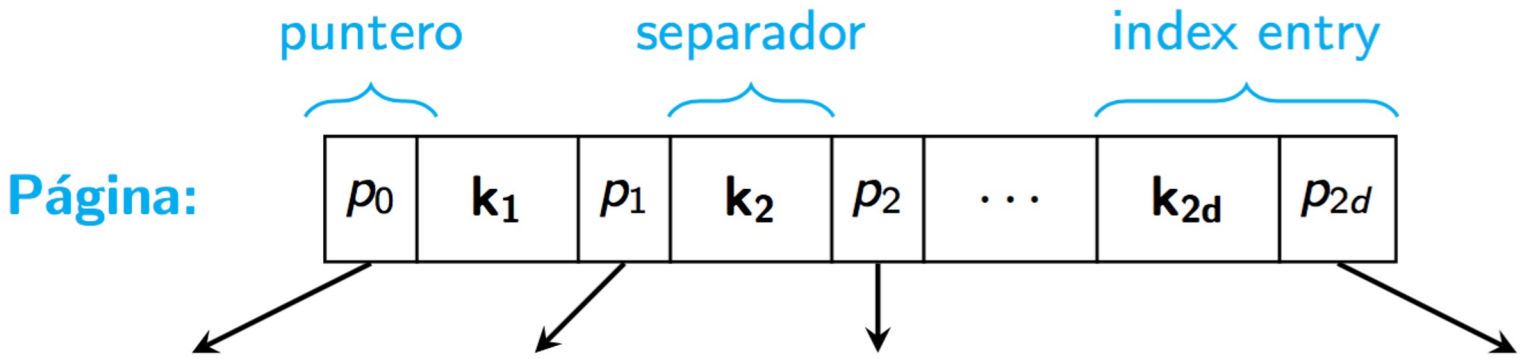
- Las páginas del disco representan nodos del árbol
- **Las tuplas** están en las hojas
- Provee un tiempo de búsqueda logarítmico
- Se comporta bien en consultas de rango
- Es el índice que usa Postgres sobre las llaves primarias (y en general, todos los sistemas)

B+ Tree Index



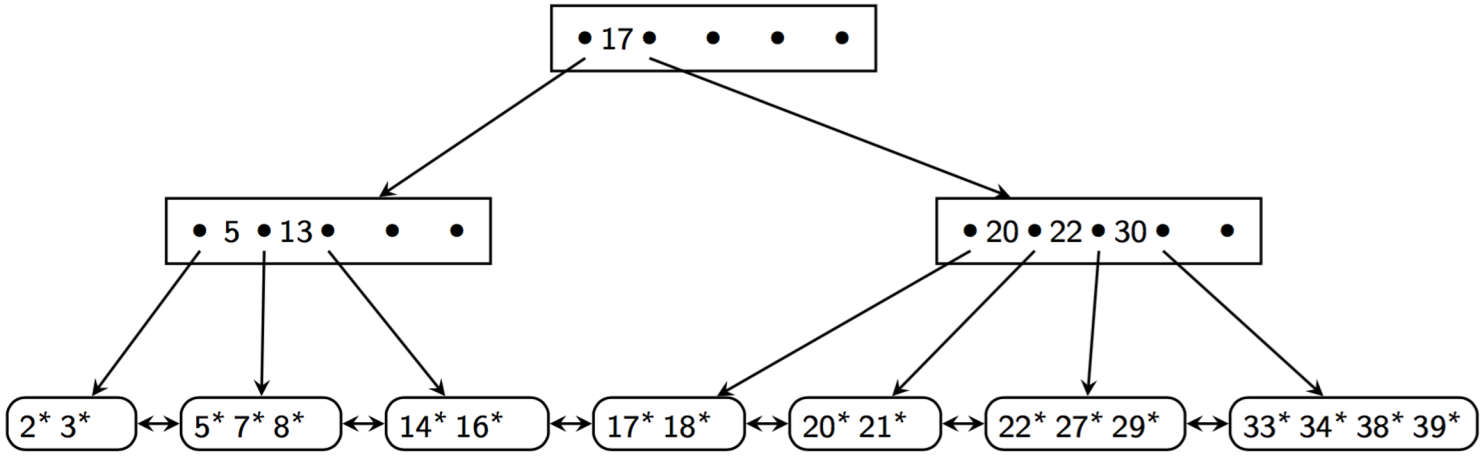
B+ Tree Index

Cada nodo del directorio tiene la siguiente forma:



B+ Tree Index

En los directorios, los nodos tienen una **Search Key** y un puntero antes y después de ella

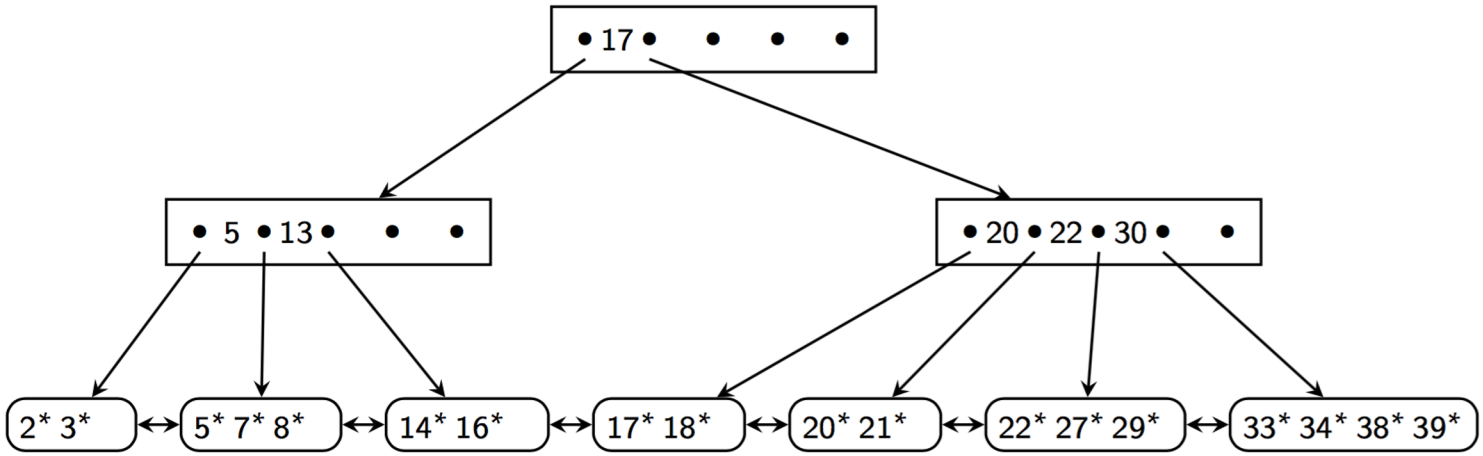


B+ Tree Index

Ejemplo: consulta por un valor

Supongamos que este índice también almacena artistas que están indexados por su aid

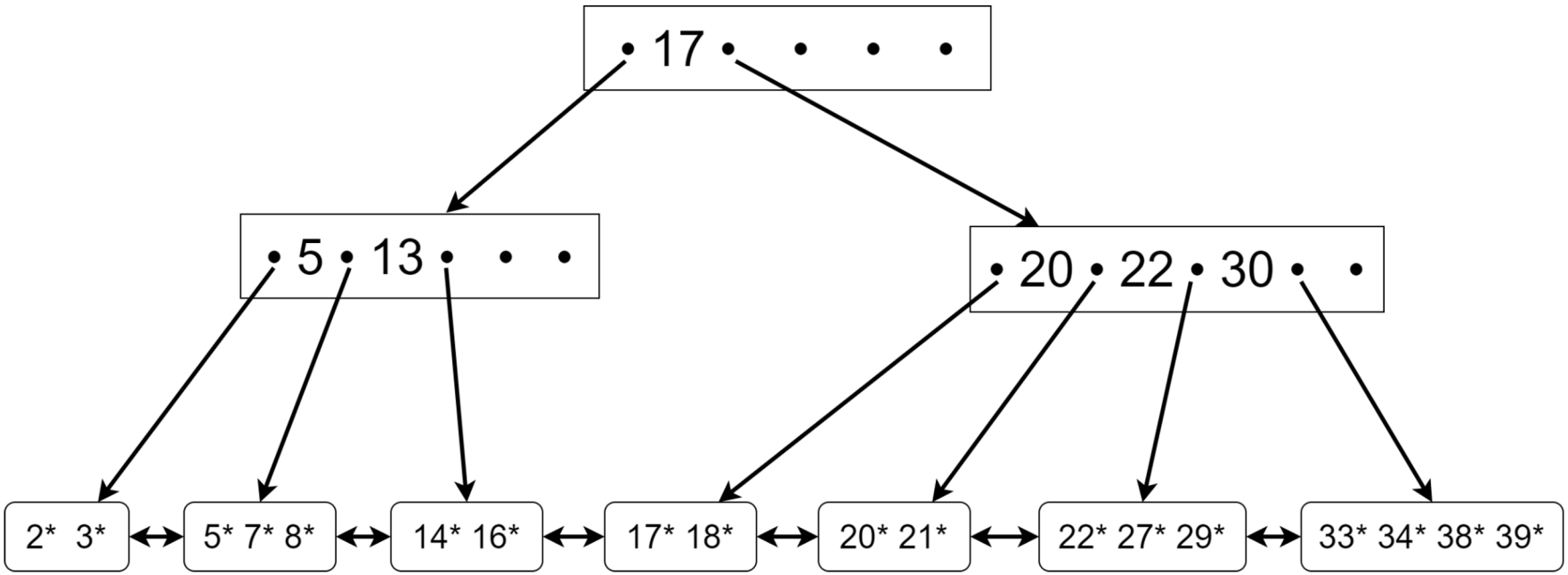
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

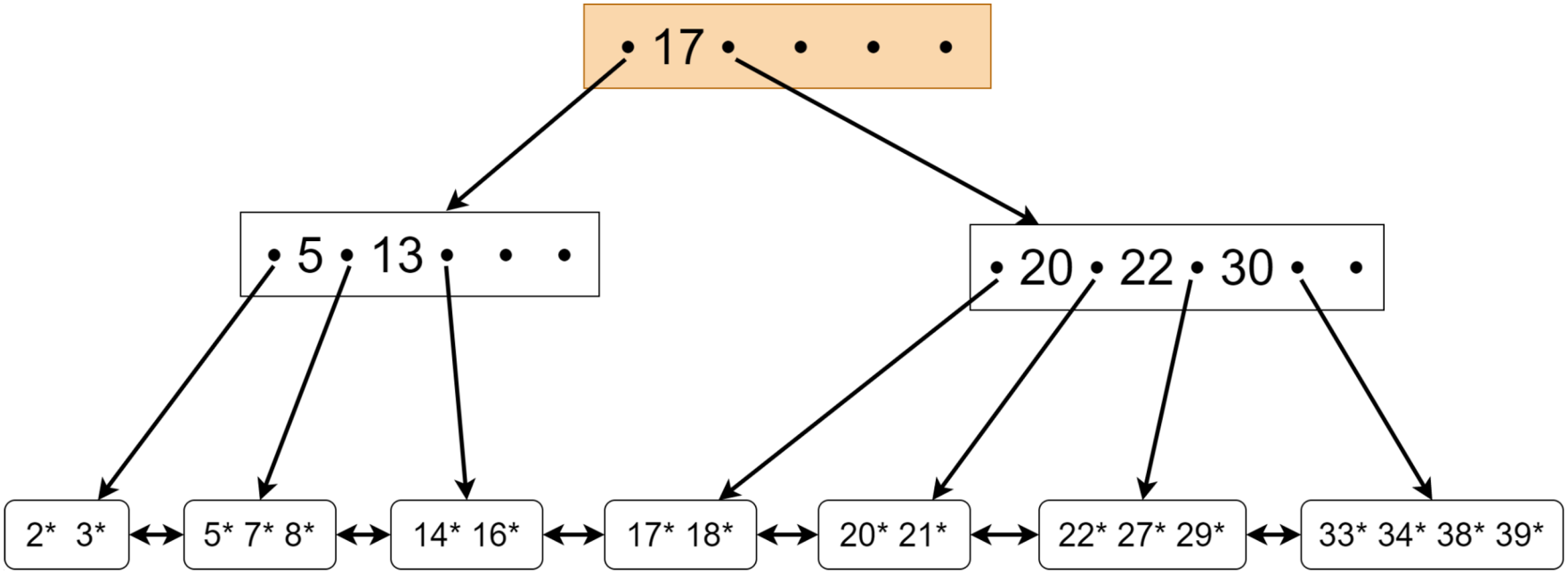
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

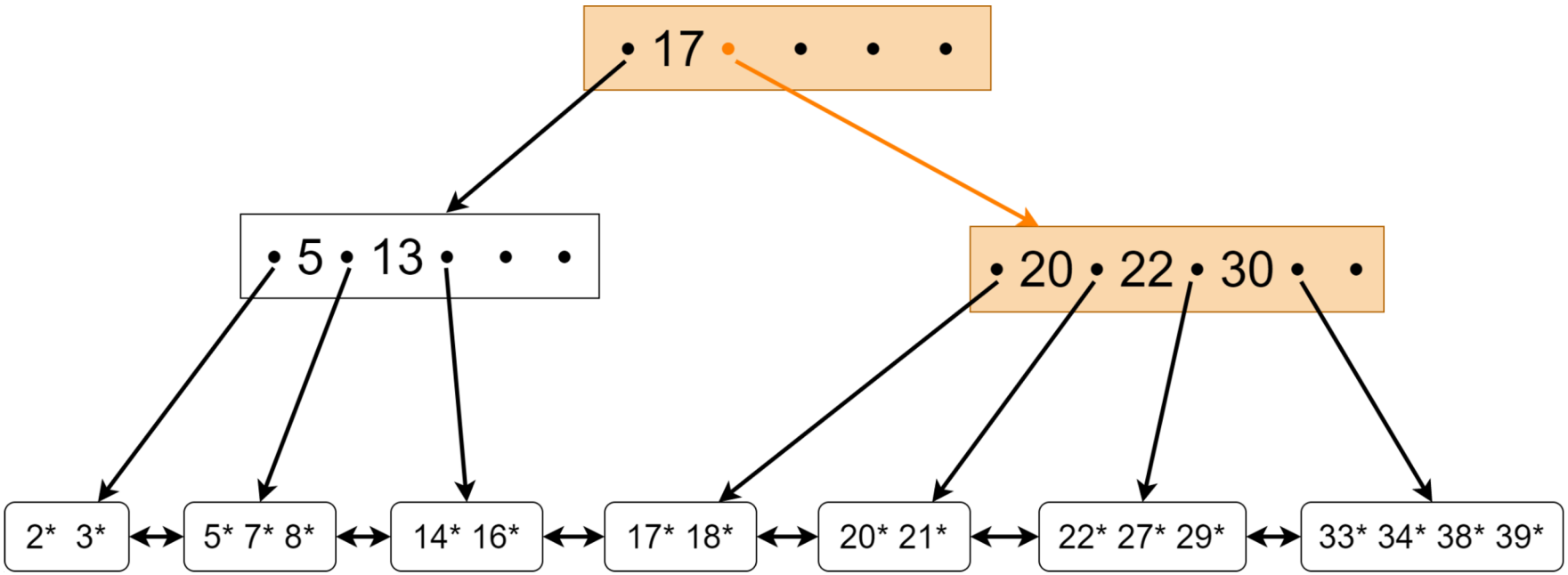
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

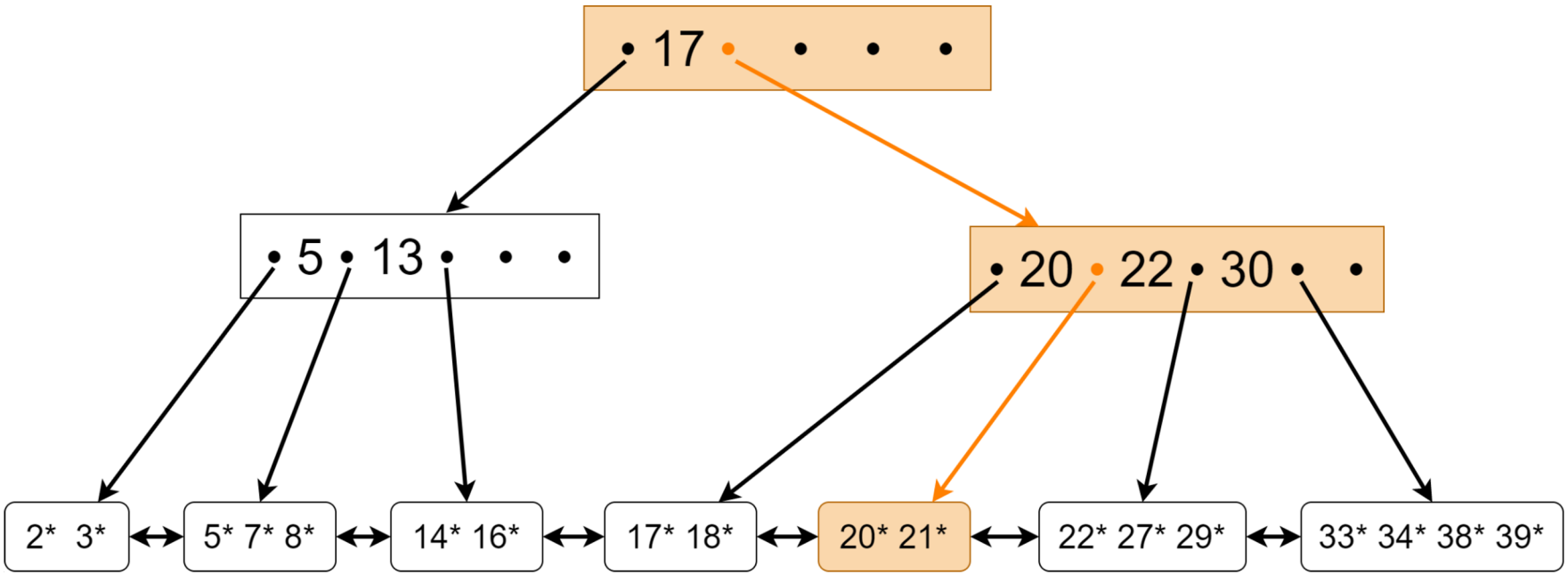
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

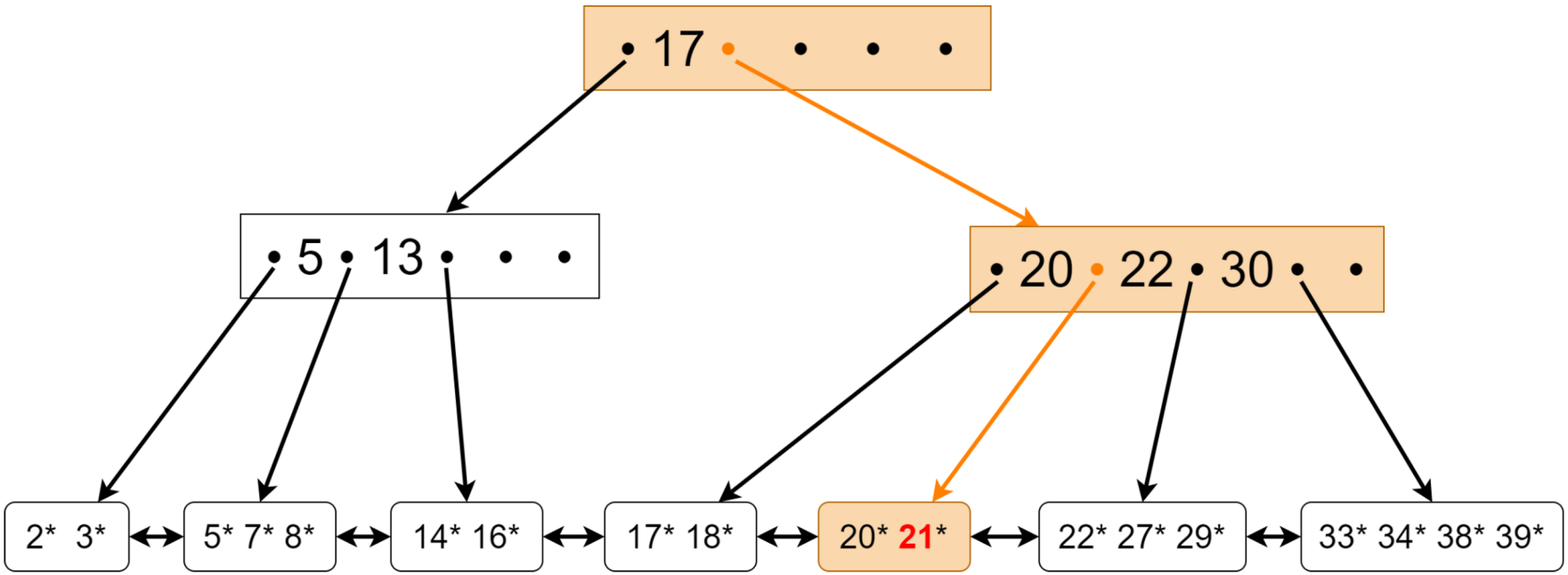
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

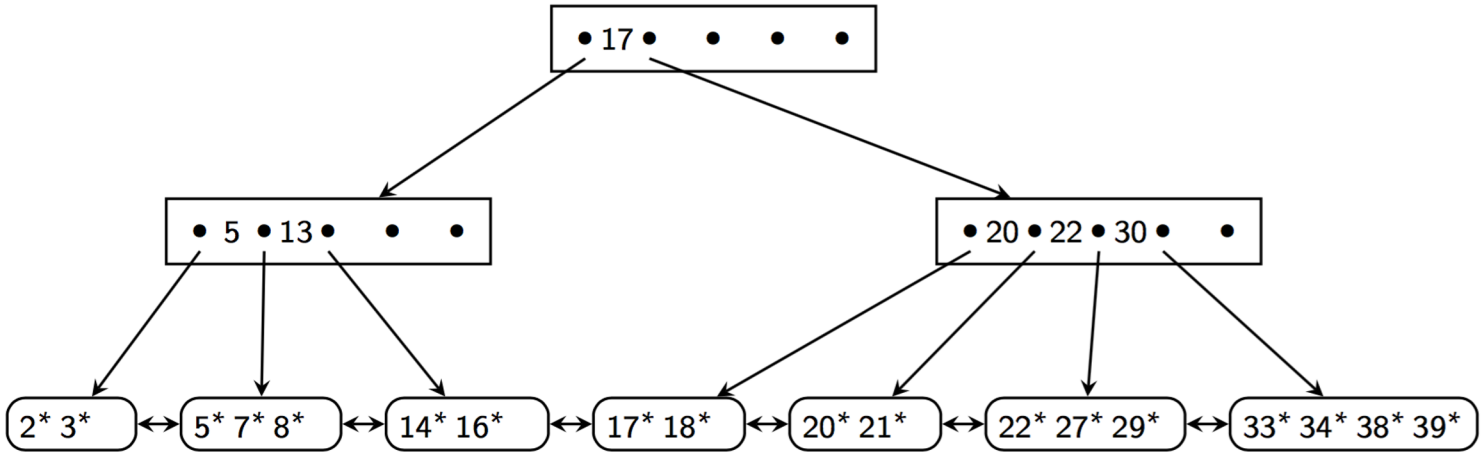
Queremos el artista con aid = 21



¿Y por qué este índice me sirve en consultas de rango?

B+ Tree Index

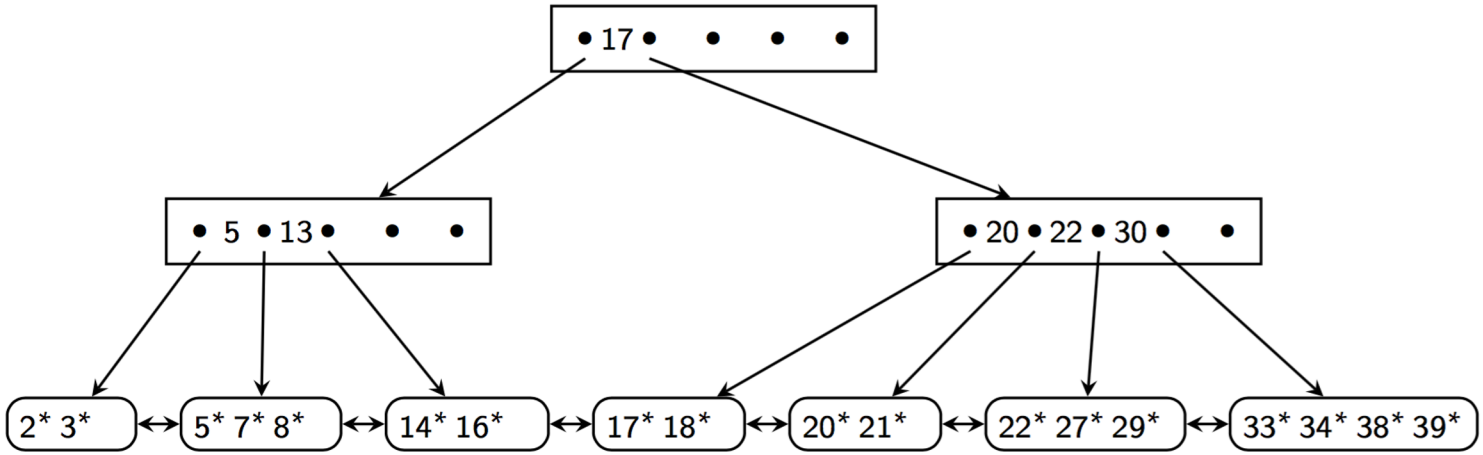
Sabemos que las tuplas van a estar ordenadas según el atributo indexado, y además las hojas están encadenadas unas con otras



B+ Tree Index

Ejemplo: consulta de rango

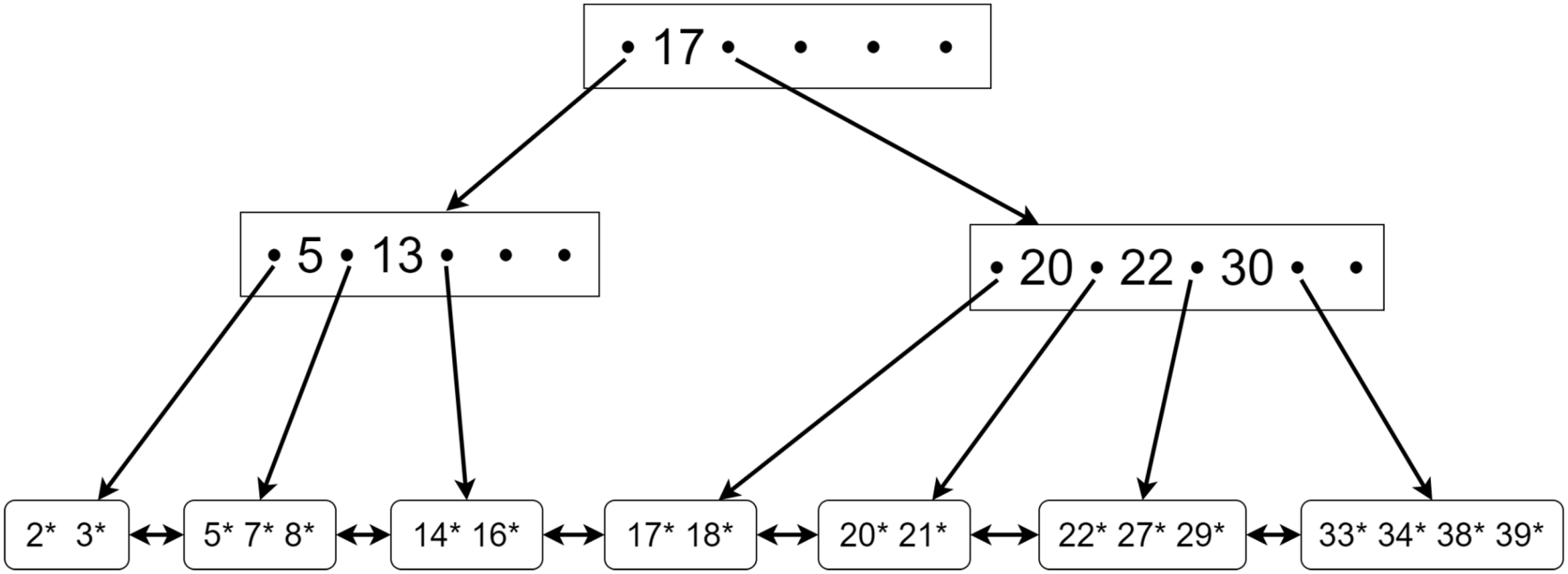
Ahora queremos hacer la consulta de todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

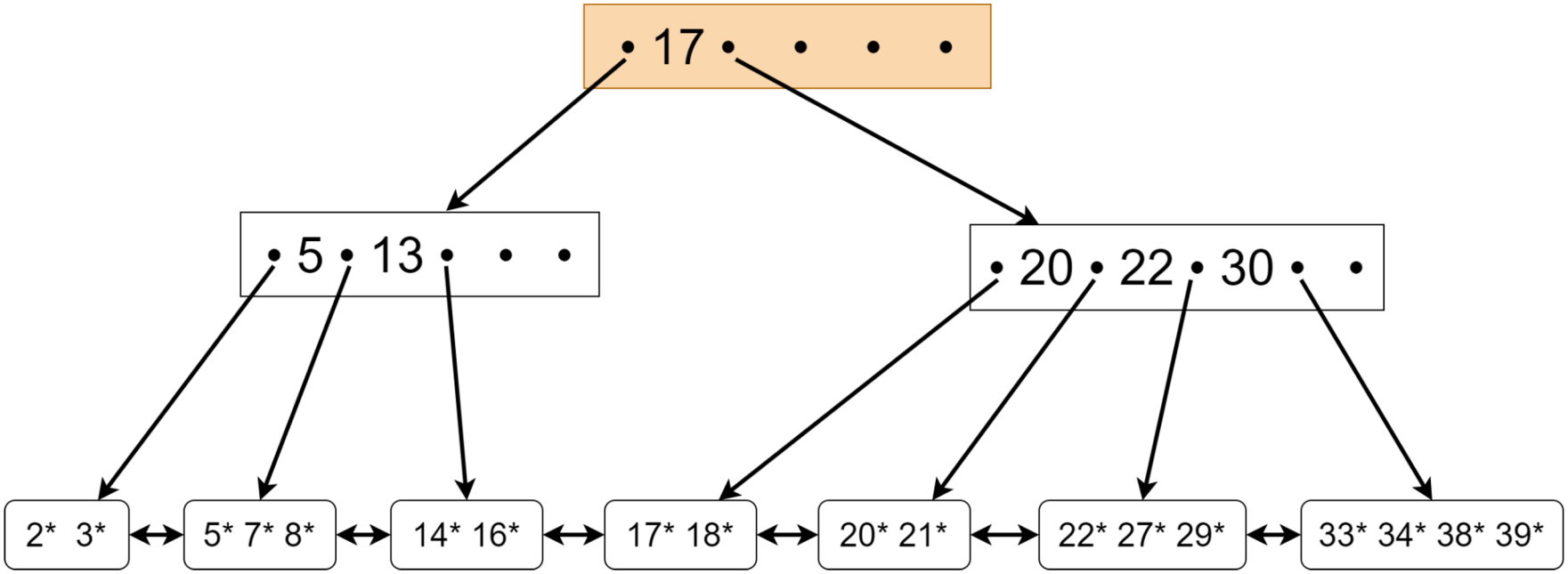
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

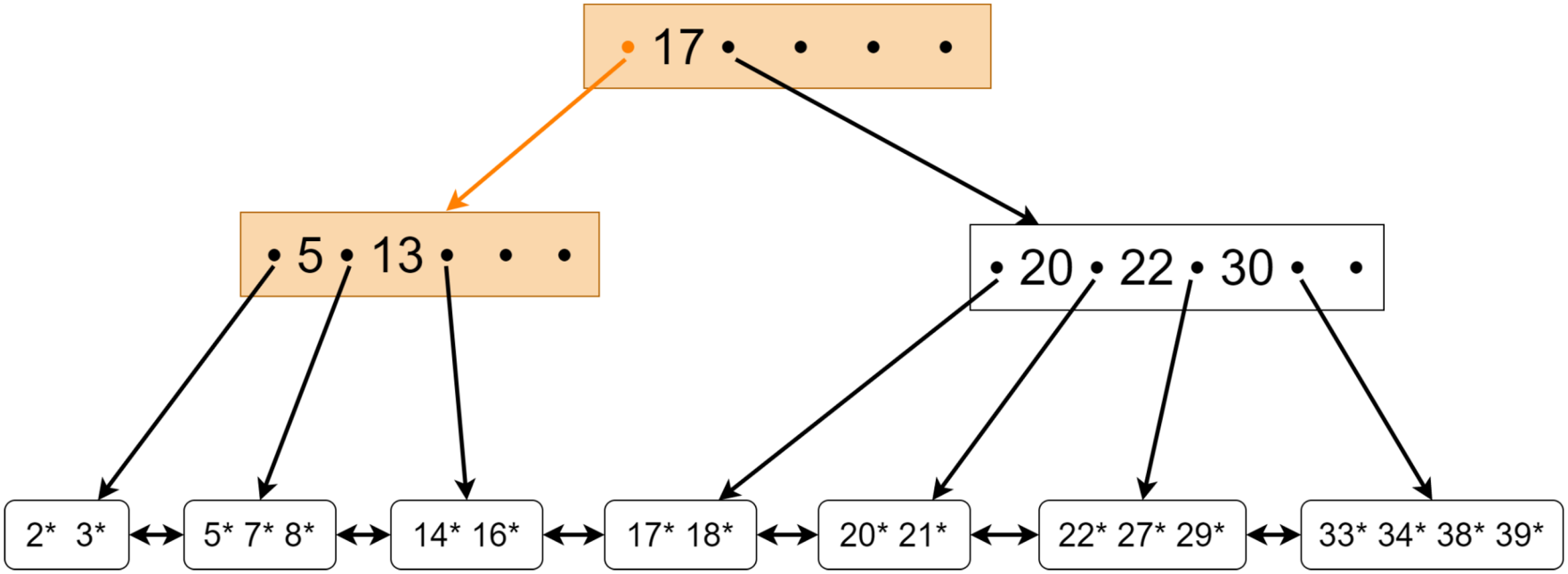
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

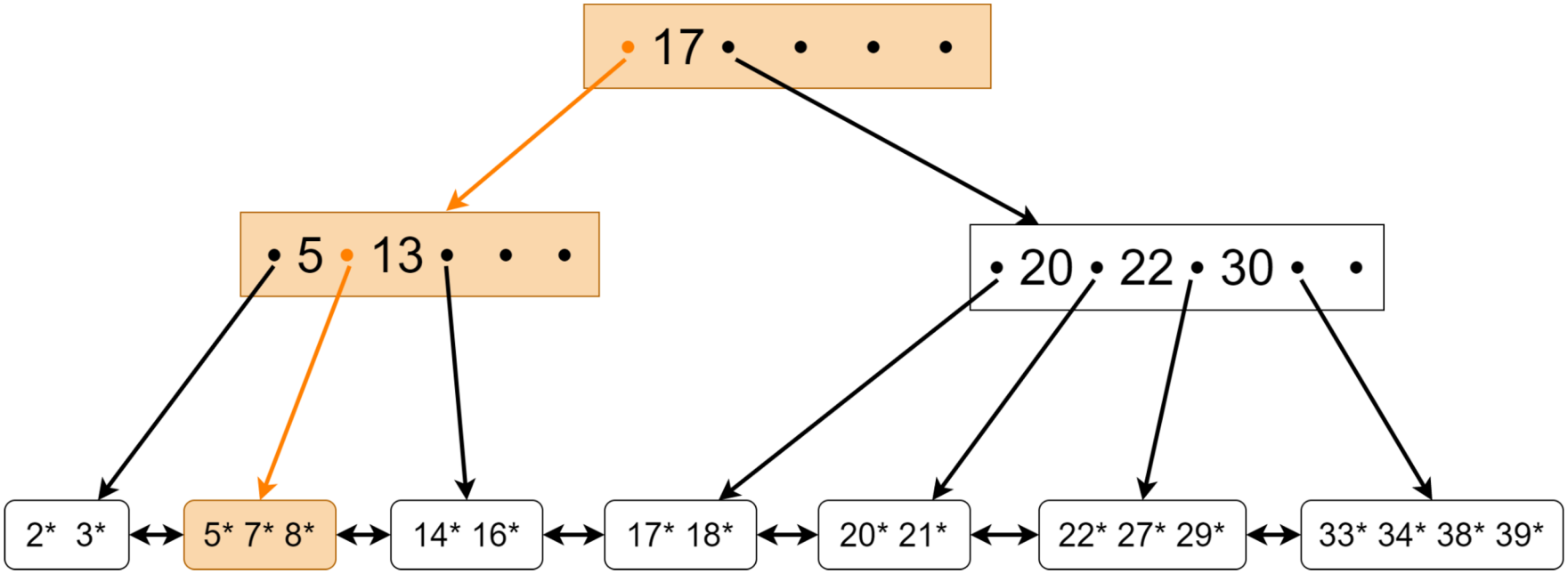
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

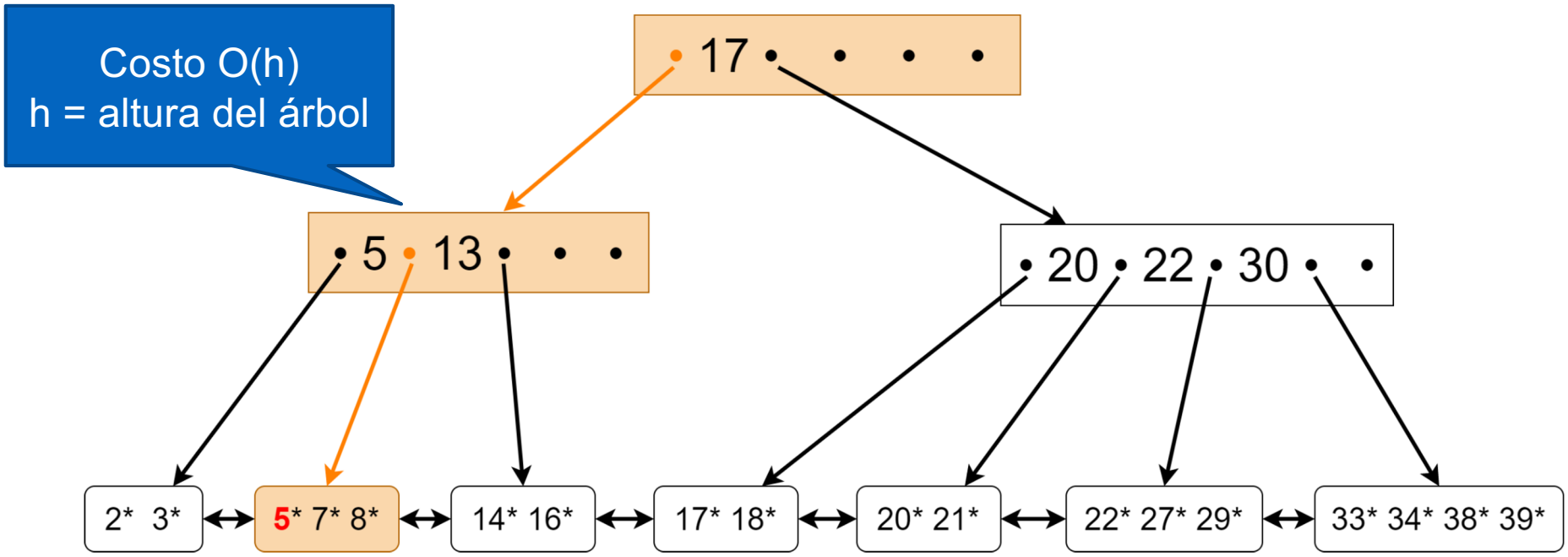
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

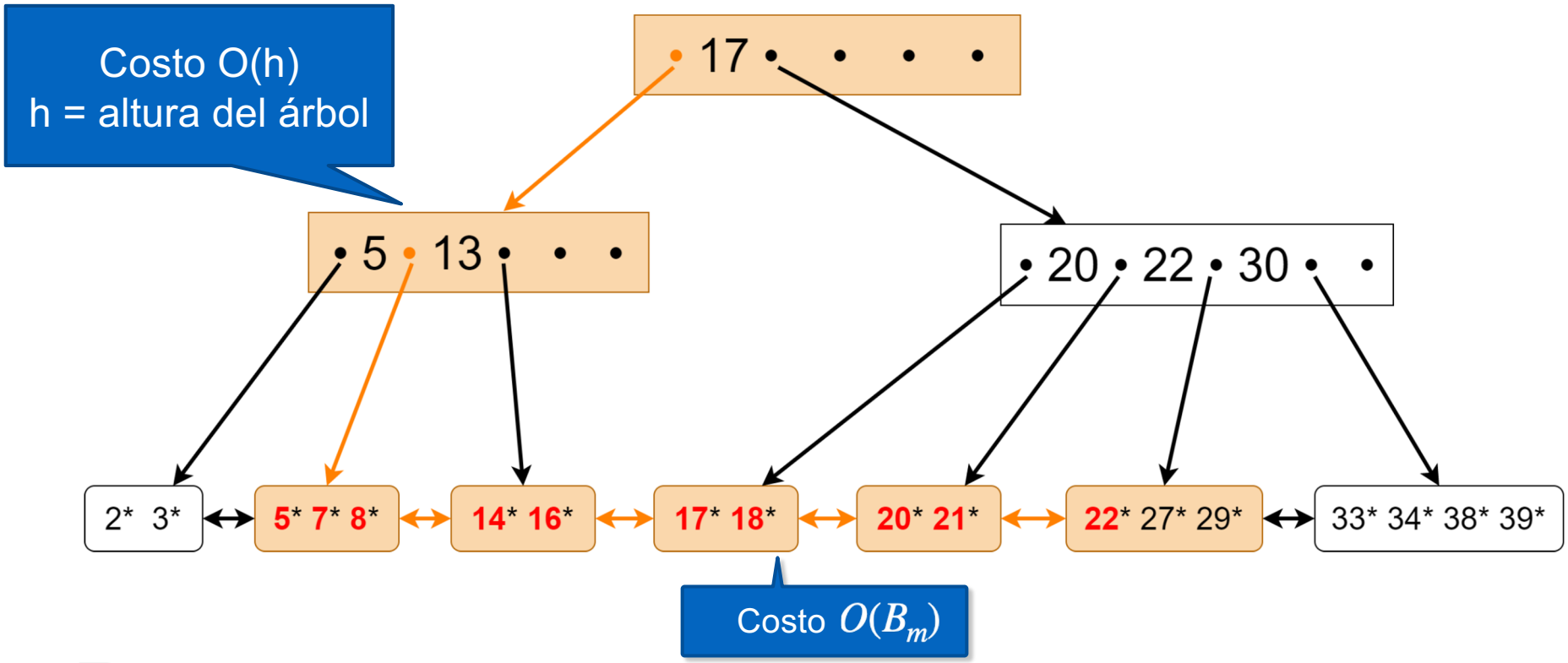
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

Todos los Artistas con aid entre 5 y 26



B_m : nr de bloques con récords que calzan búsqueda

B+ Tree Index

Importante: para facilitar inserción de datos nuevos los B+ trees no tienen nodos completamente llenos

Típicamente se dice que las hojas están ocupadas a un porcentaje; e.g. 60% -- quiere decir si caben 100 tuplas hay solo 60 en una página de la hoja

B+ Tree Index

El costo de búsqueda es logarítmico $\sim log(|R|)$

Si **B** es el tamaño de una página en tuplas, el costo de búsqueda es:

$$O(log_B(\left\lceil \frac{|R|}{B} \right\rceil))$$

¿Qué pasa si el índice lo mantenemos en memoria RAM?

Tipos de valores a guardar

Un índice me puede retornar:

- *La tupla* que estoy buscando
- Un *puntero* o lista de punteros a las páginas del disco duro que tienen las tuplas que estoy buscando

Clustered y Unclustered Index

Data entry:
La tupla misma

Hablaremos de un índice **Clustered** si al preguntarle por un valor me retorna una tupla

Hablaremos de un índice **Unclustered** si me retorna un puntero

Data entry:
Puntero a la página
con mi tupla

Clustered y Unclustered Index

Clustered Index se conoce como índice primario

Unclustered Index se conoce como índice secundario

Clustered y Unclustered Index

Considere Empleados(id, nombre, edad)

Un índice primario en general se aplica sobre la primary key, por ejemplo el id de los empleados

Un índice secundario se aplica sobre un atributo como edad (para ordenar por edad más rápido)

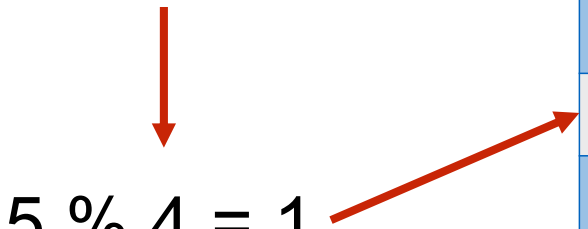
Buscando una tupla con distintos tipos de índices

Hash Index clustered

R(a,b,c) … índice sobre a

a = 5

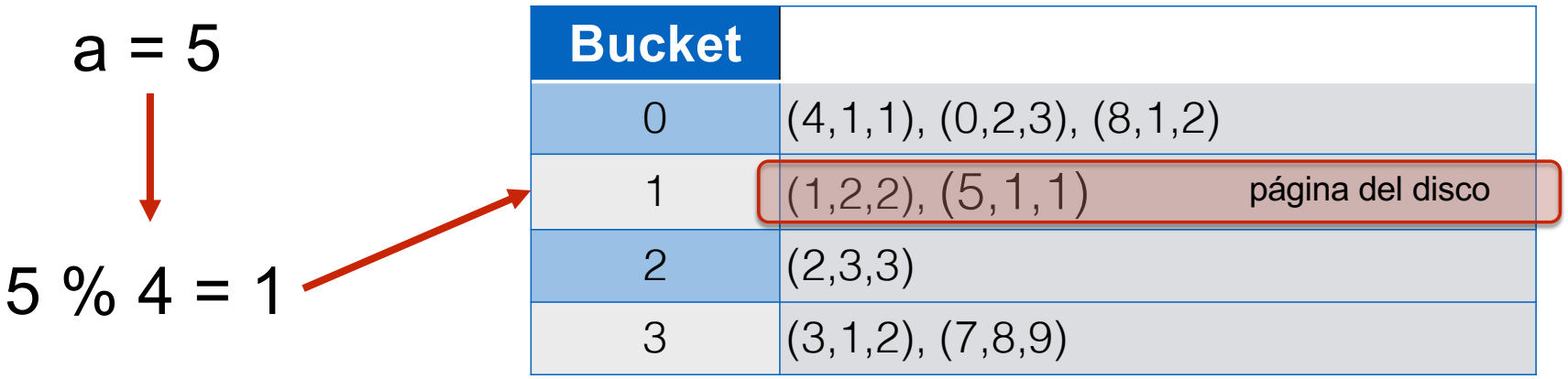
5 % 4 = 1



Bucket	
0	(4,1,1), (0,2,3), (8,1,2)
1	(1,2,2), (5,1,1)
2	(2,3,3)
3	(3,1,2), (7,8,9)

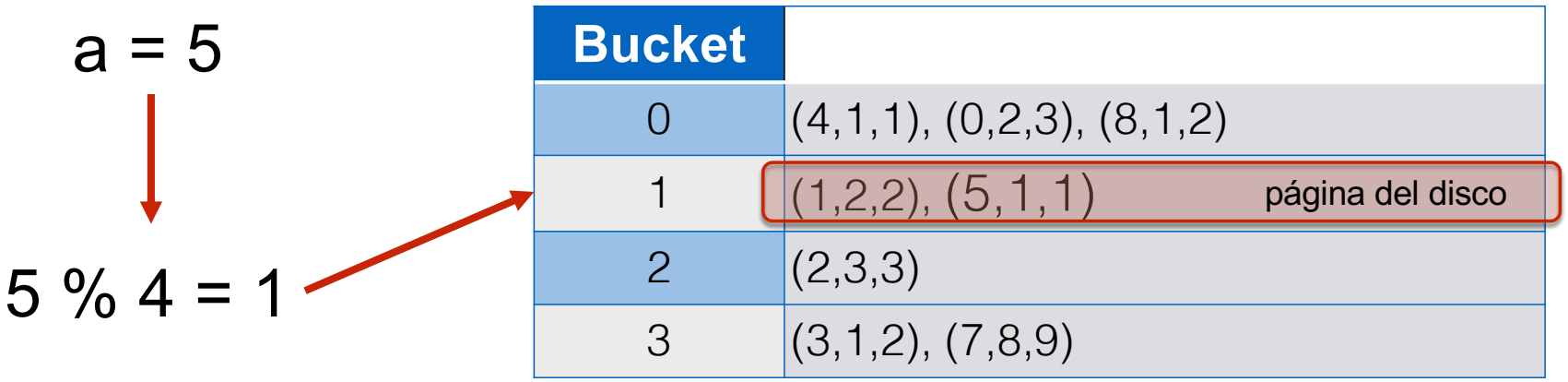
Hash Index clustered

R(a,b,c) … índice sobre a



Hash Index clustered

R(a,b,c) … índice sobre a



#(I/O) = 1 (el bucket es la página con mi tupla)

Hash Index unclustered

R(a,b,c) … índice sobre a

a = 5

5 % 4 = 1

Bucket	
0	(4,p1), (0,p2), (8,p3)
1	(1,p1), (5,p1)
2	(2,p2)
3	(3,p2), (7,p3)

Datos en el disco:

p1	p2	p3
(4,1,1), (1,2,2), (5,1,1)	(0,2,3), (3,1,2), (2,3,3)	(7,8,9), (8,1,2)

Hash Index unclustered

R(a,b,c) … índice sobre a

a = 5

5 % 4 = 1

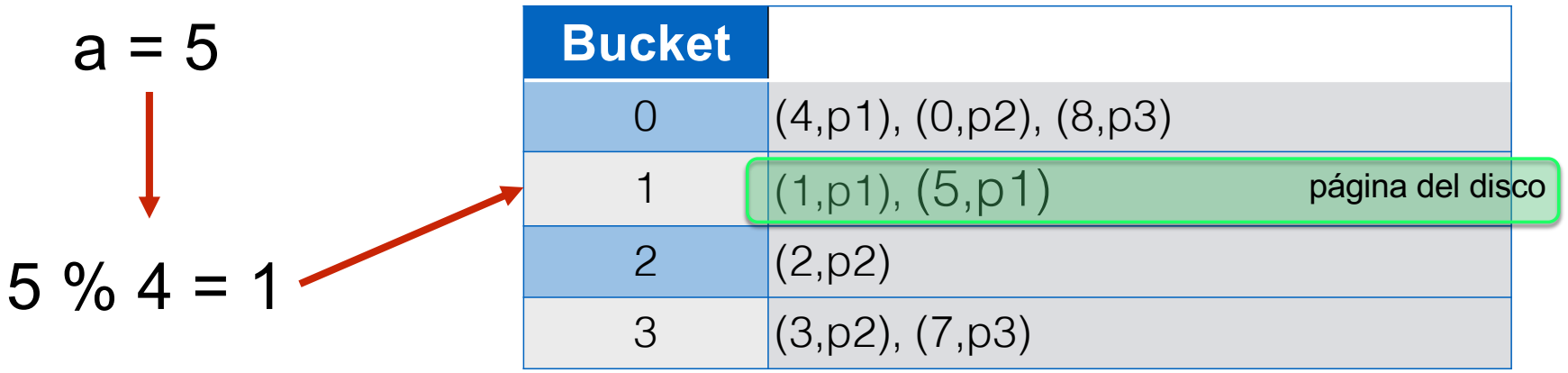
Bucket	
0	(4,p1), (0,p2), (8,p3)
1	(1,p1), (5,p1) página del disco
2	(2,p2)
3	(3,p2), (7,p3)

Datos en el disco:

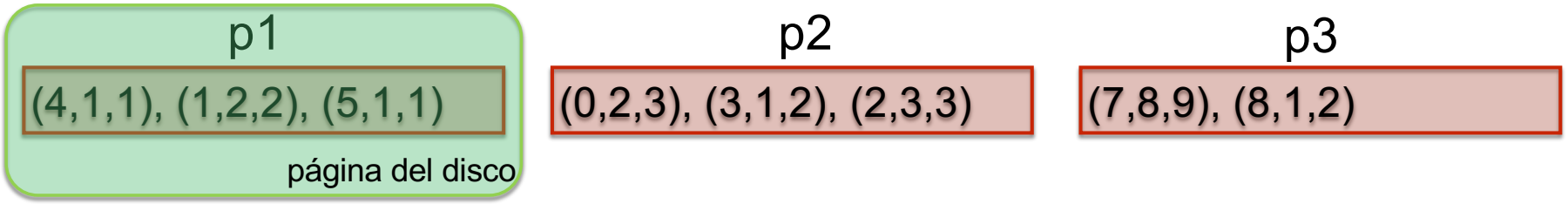
p1	p2	p3
(4,1,1), (1,2,2), (5,1,1)	(0,2,3), (3,1,2), (2,3,3)	(7,8,9), (8,1,2)

Hash Index unclustered

R(a,b,c) … índice sobre a



Datos en el disco:



Hash Index unclustered

R(a,b,c) … índice sobre a

a = 5

5 % 4 = 1

Bucket	
0	(4,p1), (0,p2), (8,p3)
1	(1,p1), (5,p1) página del disco
2	(2,p2)
3	(3,p2), (7,p3)

Datos en el disco:

p1

(4,1,1), (1,2,2), (5,1,1)

página del disco

p2

(0,2,3), (3,1,2), (2,3,3)

p3

(7,8,9), (8,1,2)

#(I/O) = 2

Hash Index unclustered

R(a,b,c) … índice sobre a

Considerar: página del índice vs página con tuplas



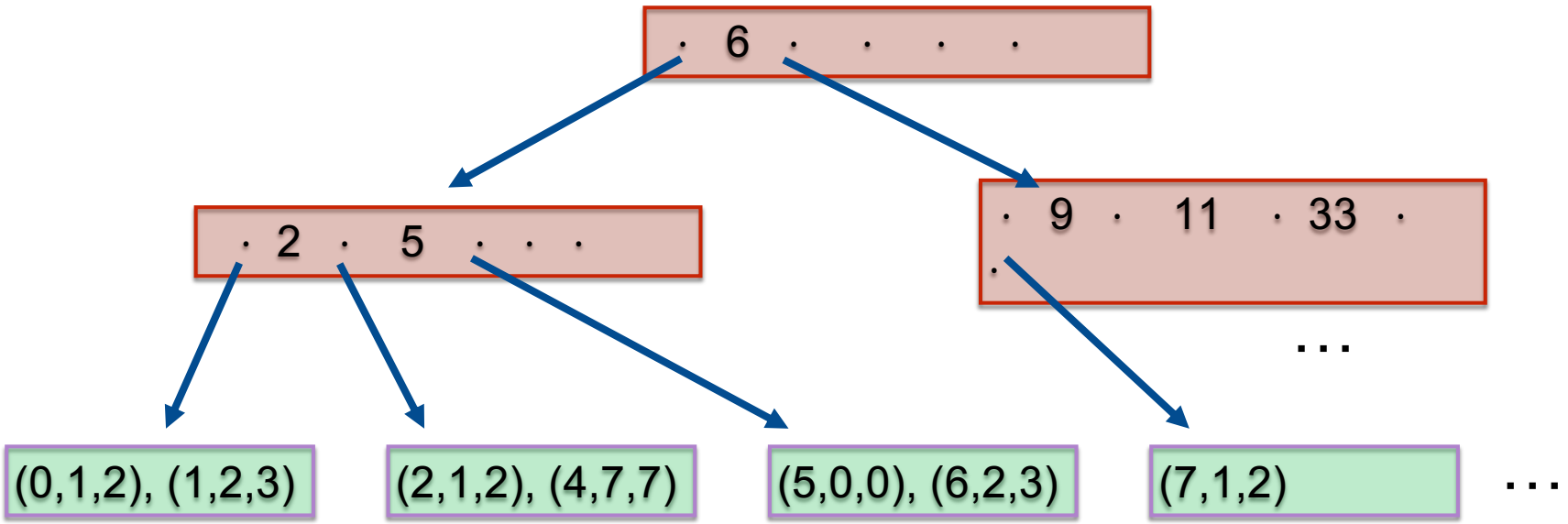
puntero p1 … un long/8 bytes

tupla … puede ser 20 longs/160 bytes

¿Dónde cabe más? (P>B)

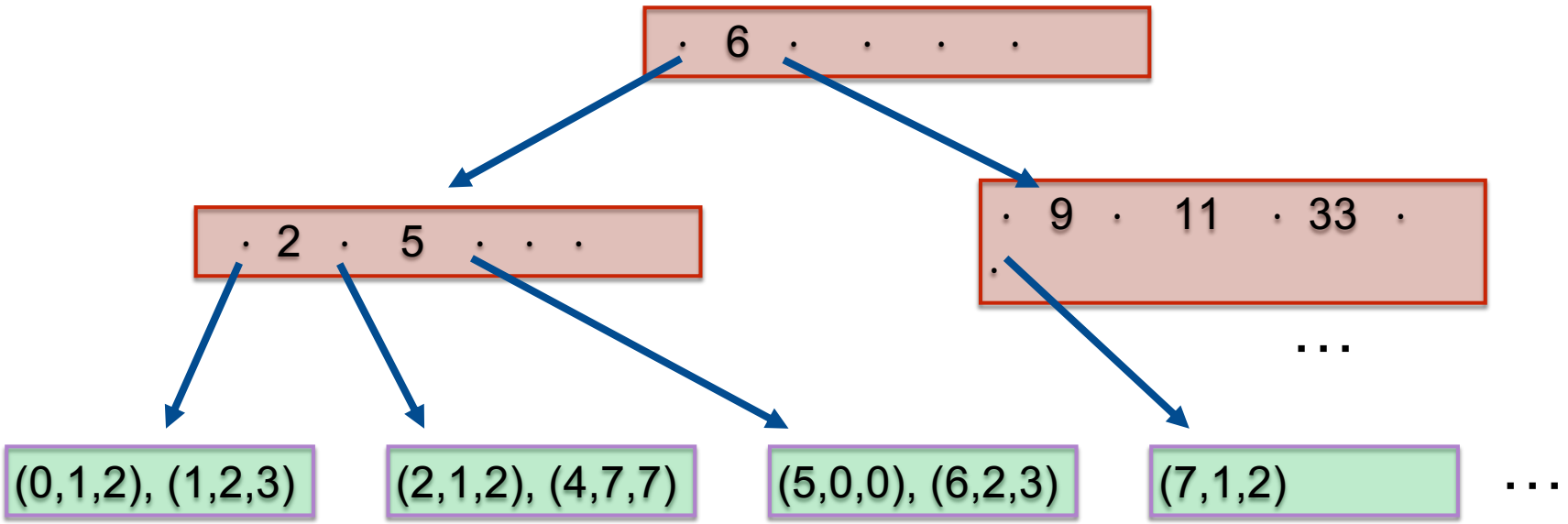
B+ tree clustered

R(a,b,c) … índice sobre a



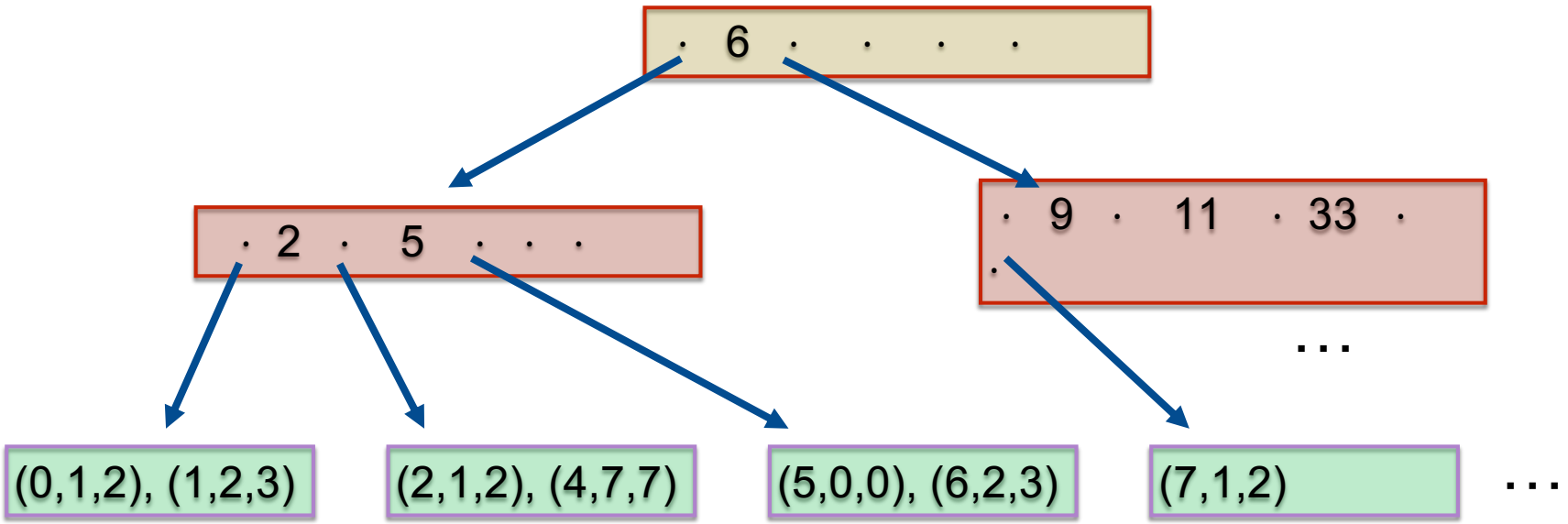
B+ tree clustered

R(a,b,c) … índice sobre a … a = 5



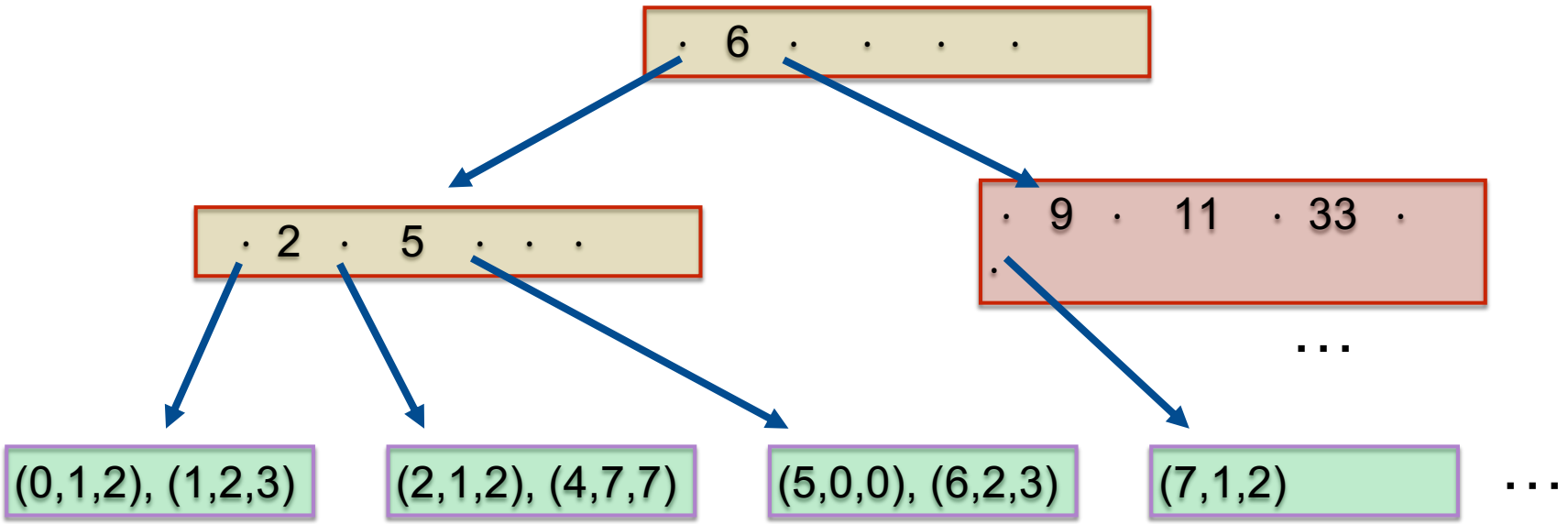
B+ tree clustered

R(a,b,c) … índice sobre a … a = 5



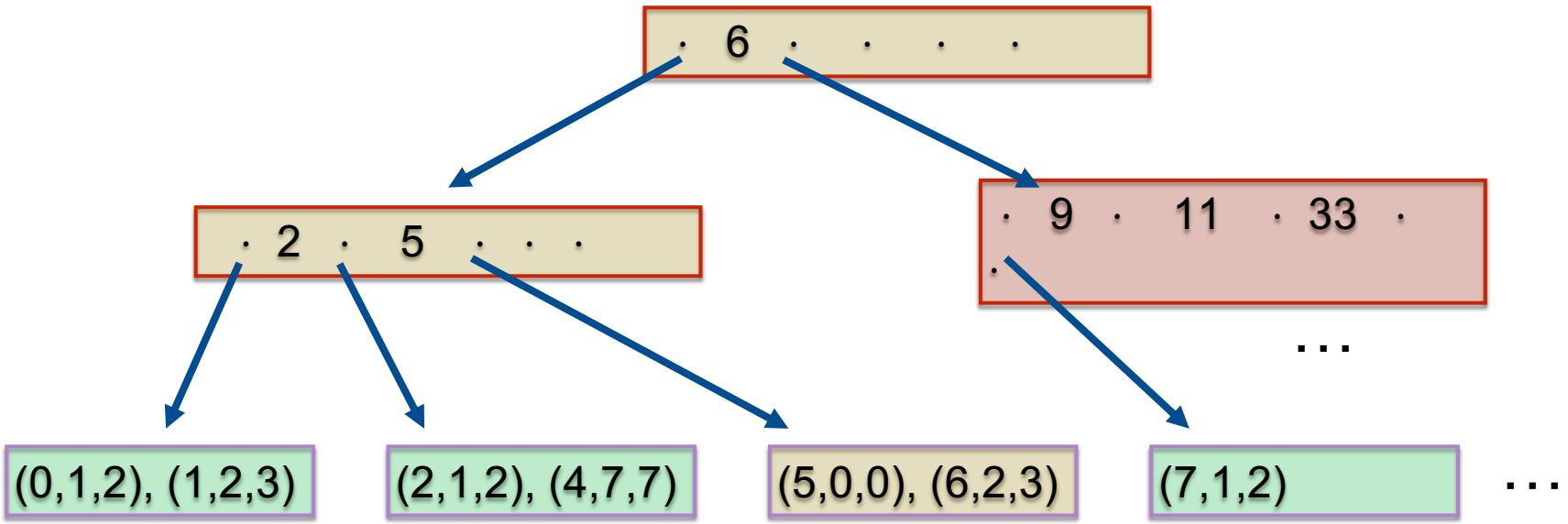
B+ tree clustered

R(a,b,c) … índice sobre a … a = 5



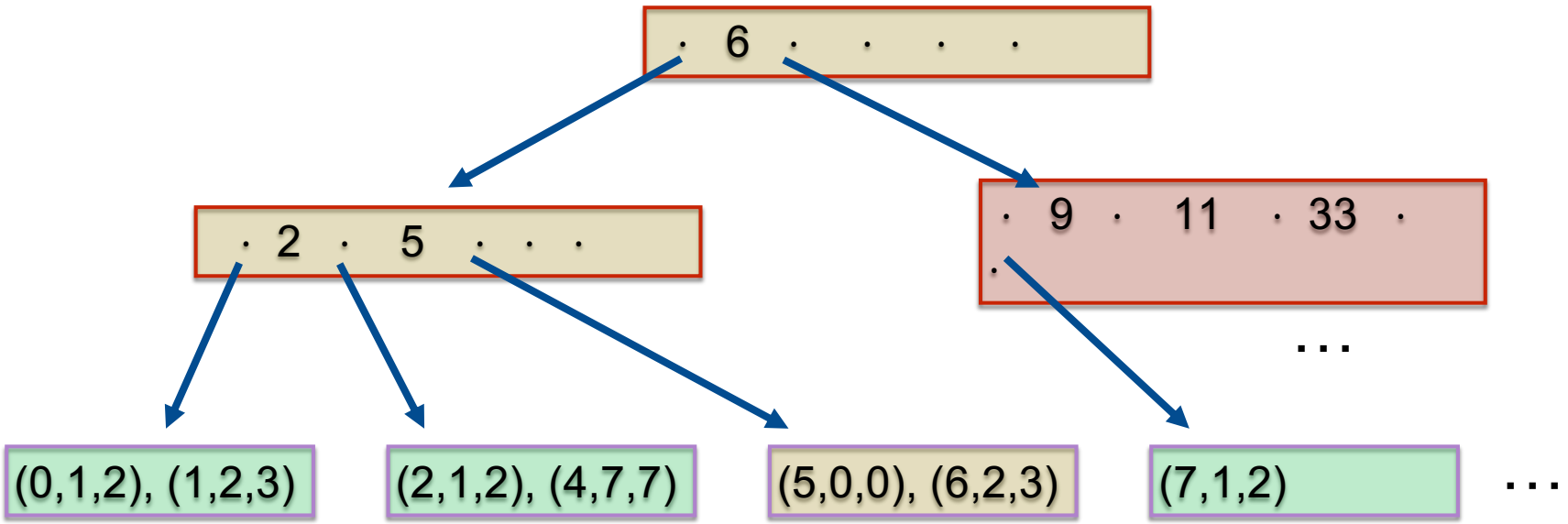
B+ tree clustered

R(a,b,c) … índice sobre a … a = 5



B+ tree clustered

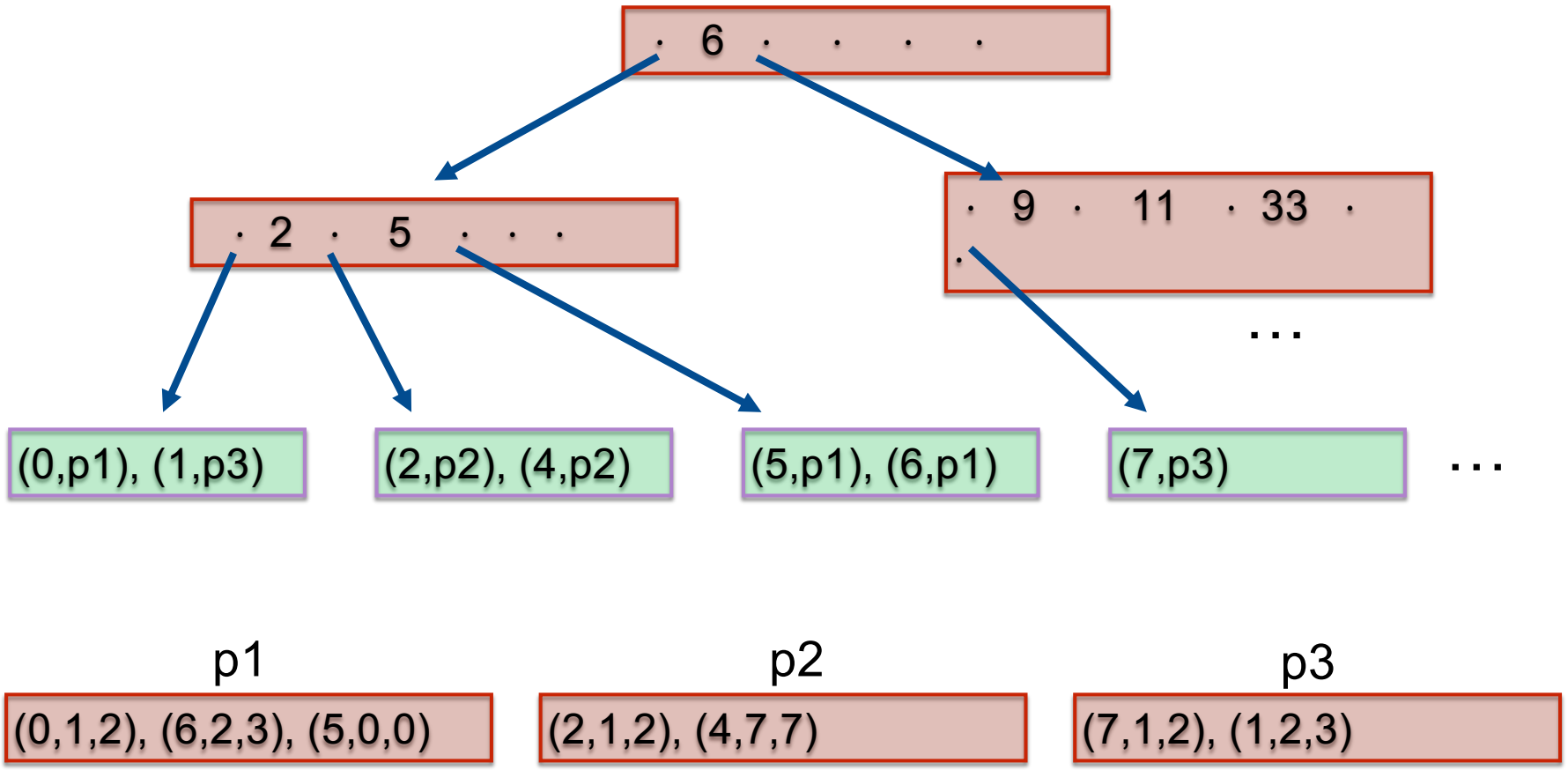
R(a,b,c) … índice sobre a … a = 5



#(I/O) = h … altura del árbol (los datos están en hojas)

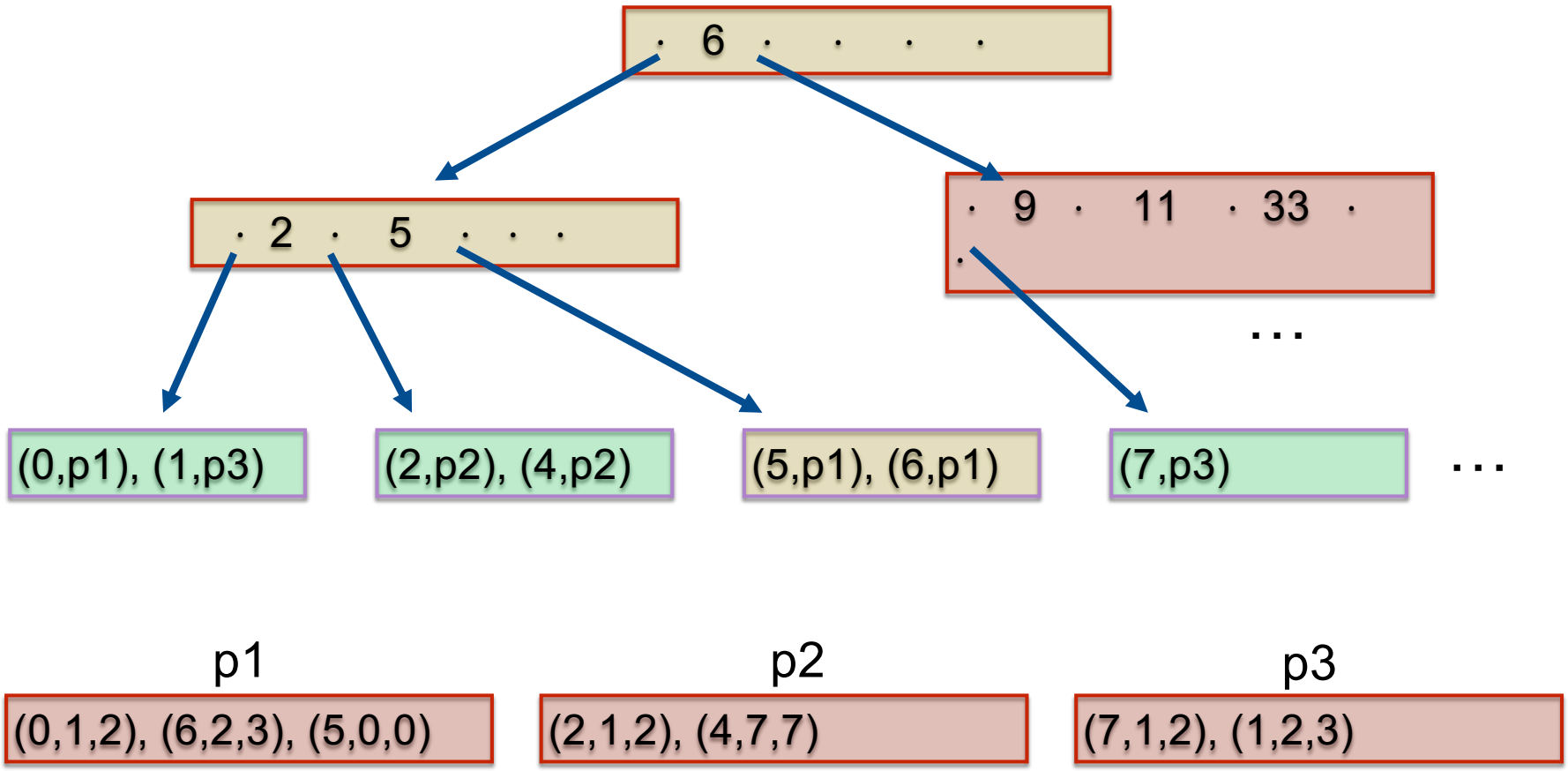
B+ tree unclustered

R(a,b,c) … índice sobre a



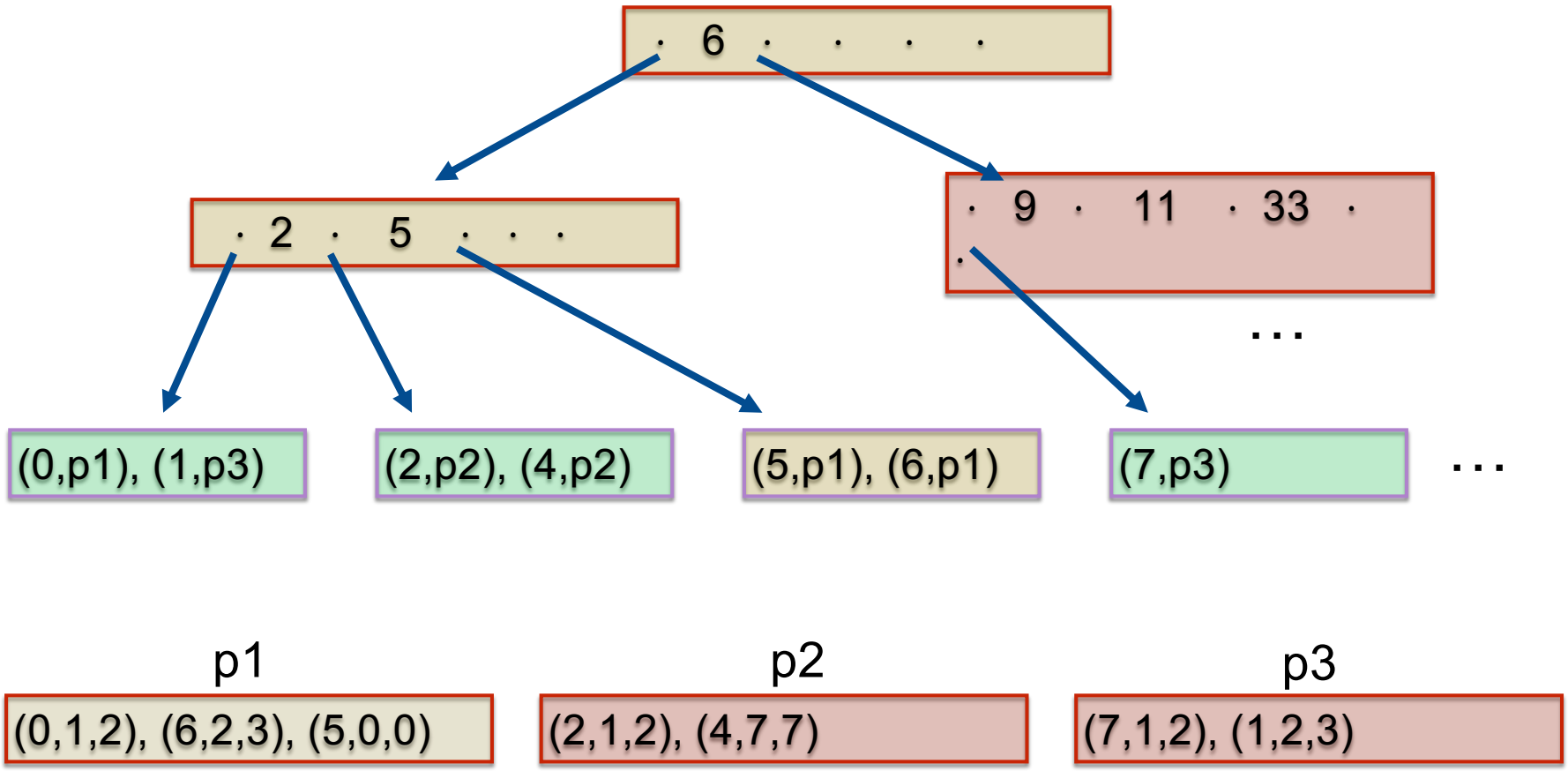
B+ tree unclustered

R(a,b,c) … índice sobre a … a = 5



B+ tree unclustered

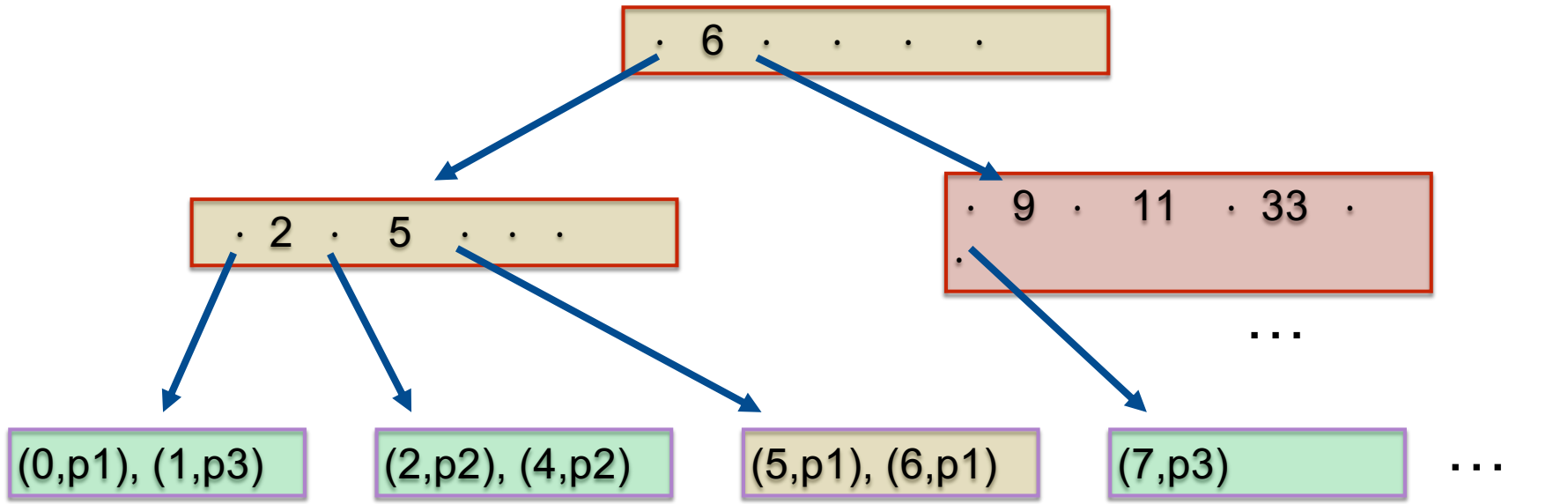
R(a,b,c) … índice sobre a … a = 5



B+ tree unclustered

R(a,b,c) … índice sobre a … a = 5

Costo *h+1*



p1

(0,1,2), (6,2,3), (5,0,0)

p2

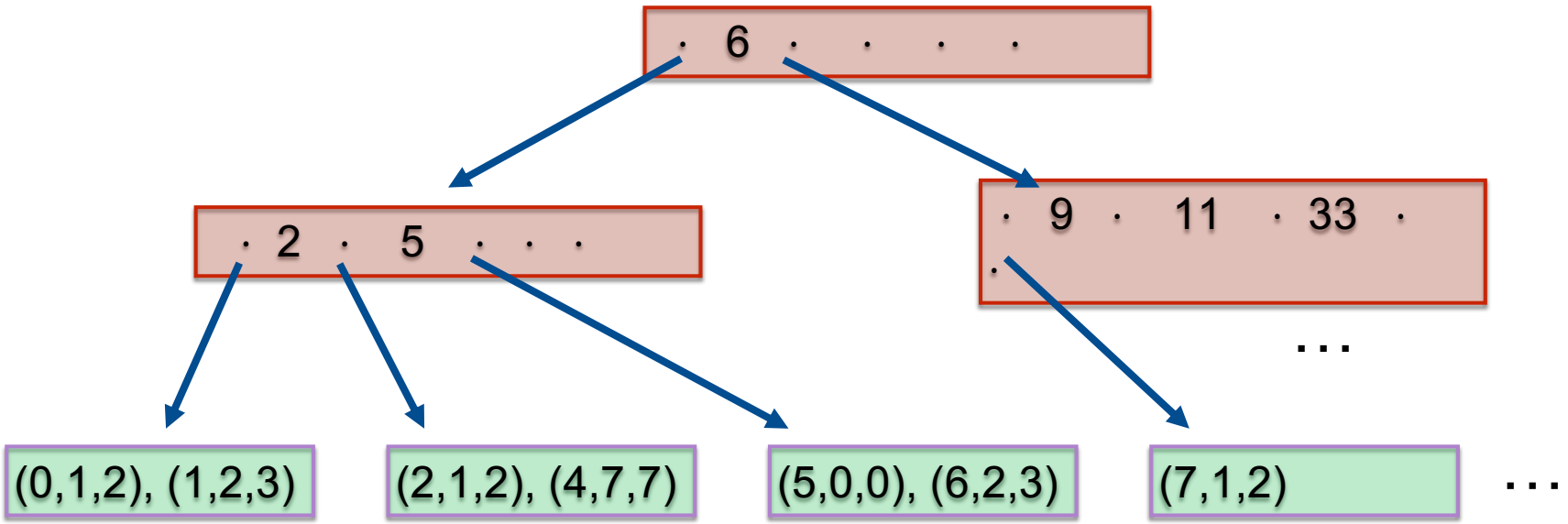
(2,1,2), (4,7,7)

p3

(7,1,2), (1,2,3)

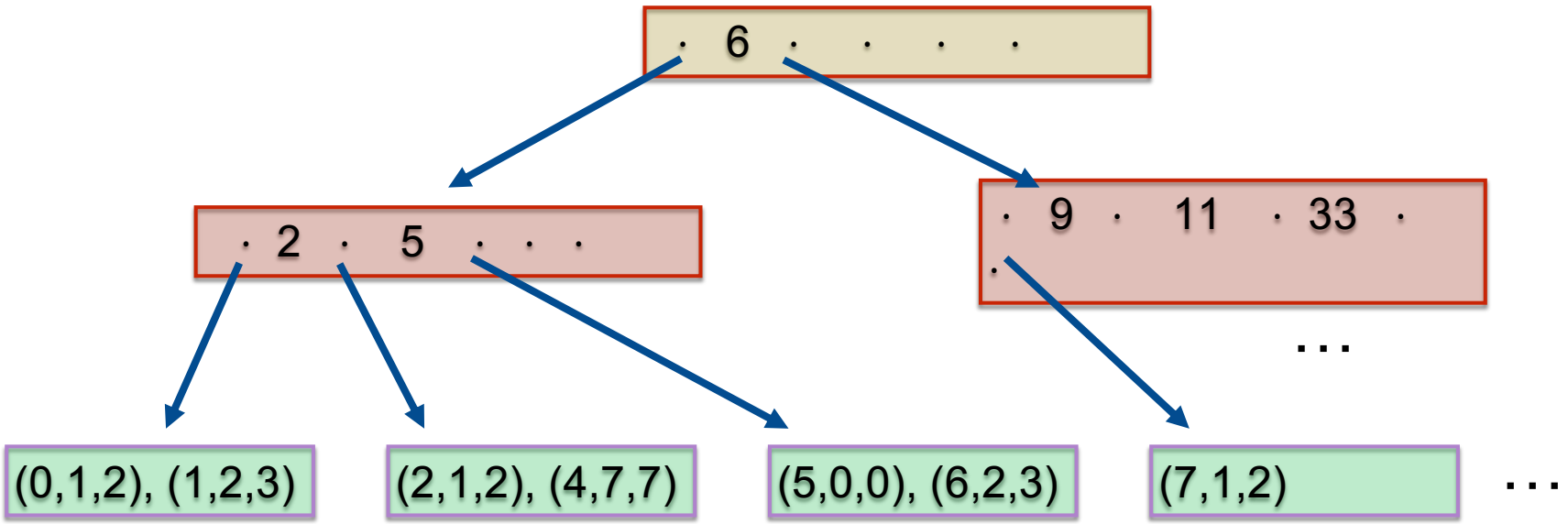
B+ tree clustered

R(a,b,c) … índice sobre a … a > 5



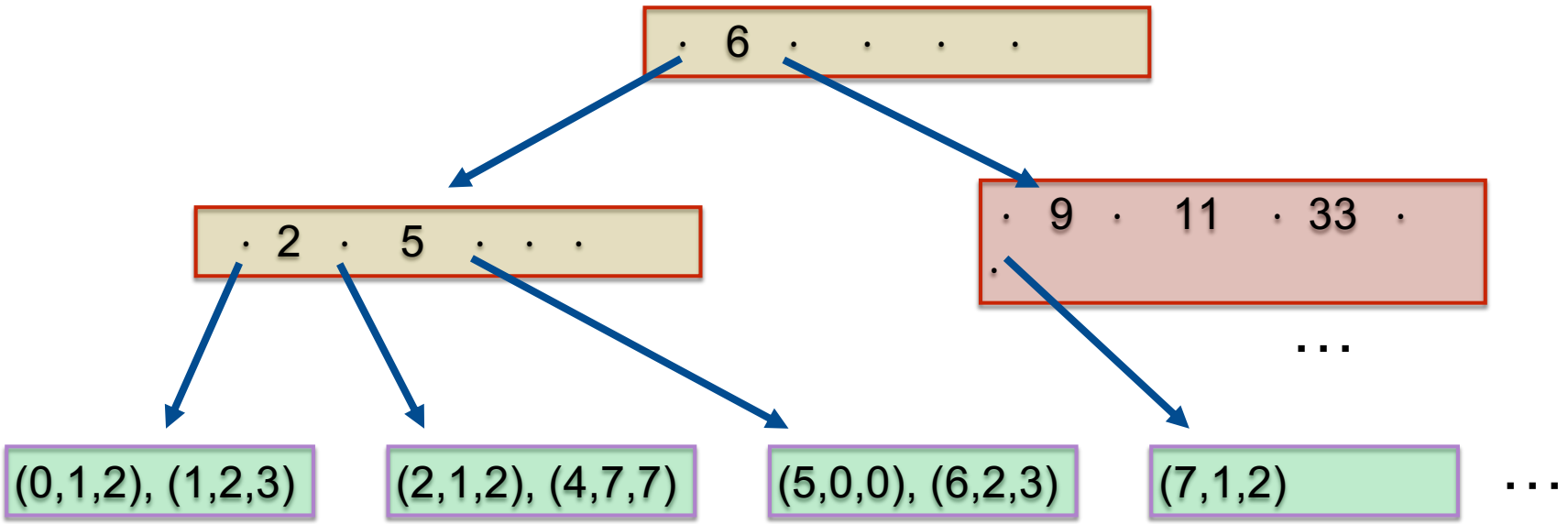
B+ tree clustered

R(a,b,c) … índice sobre a … a > 5



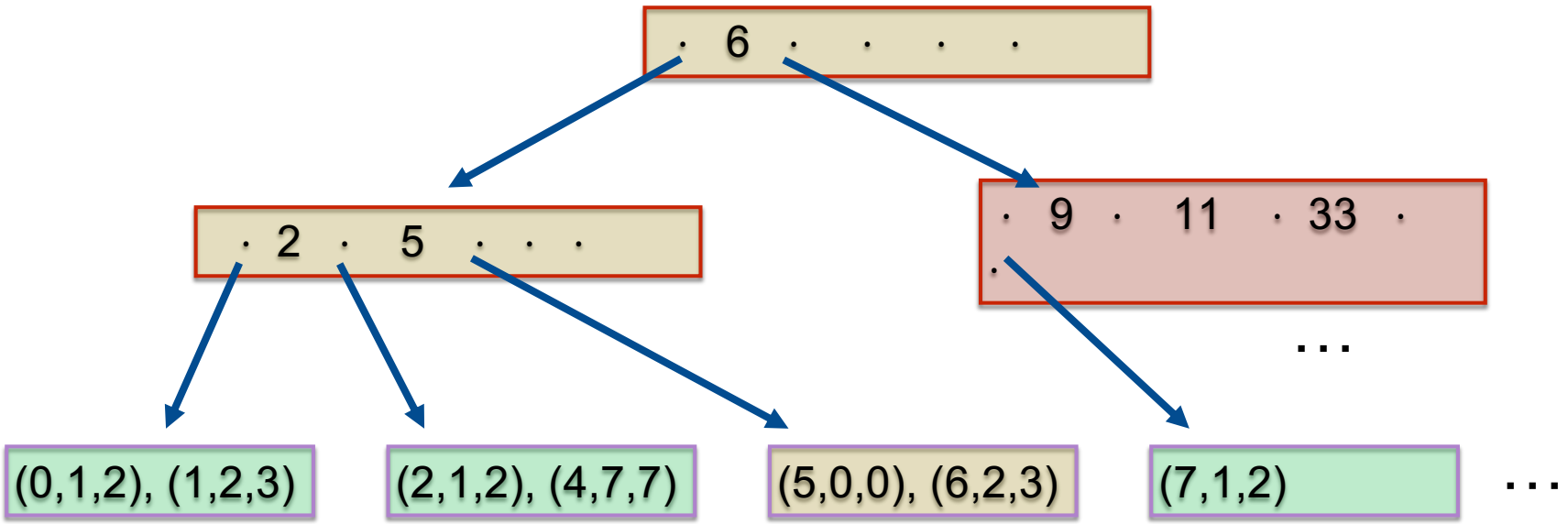
B+ tree clustered

R(a,b,c) … índice sobre a … a > 5



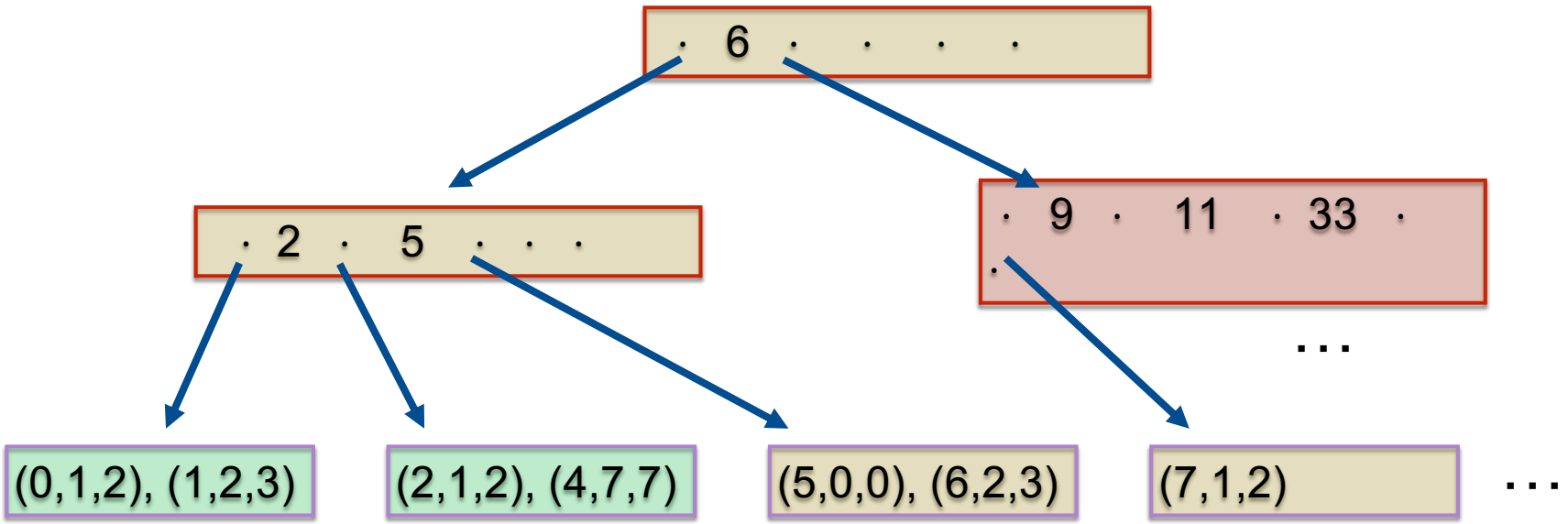
B+ tree clustered

R(a,b,c) … índice sobre a … a > 5



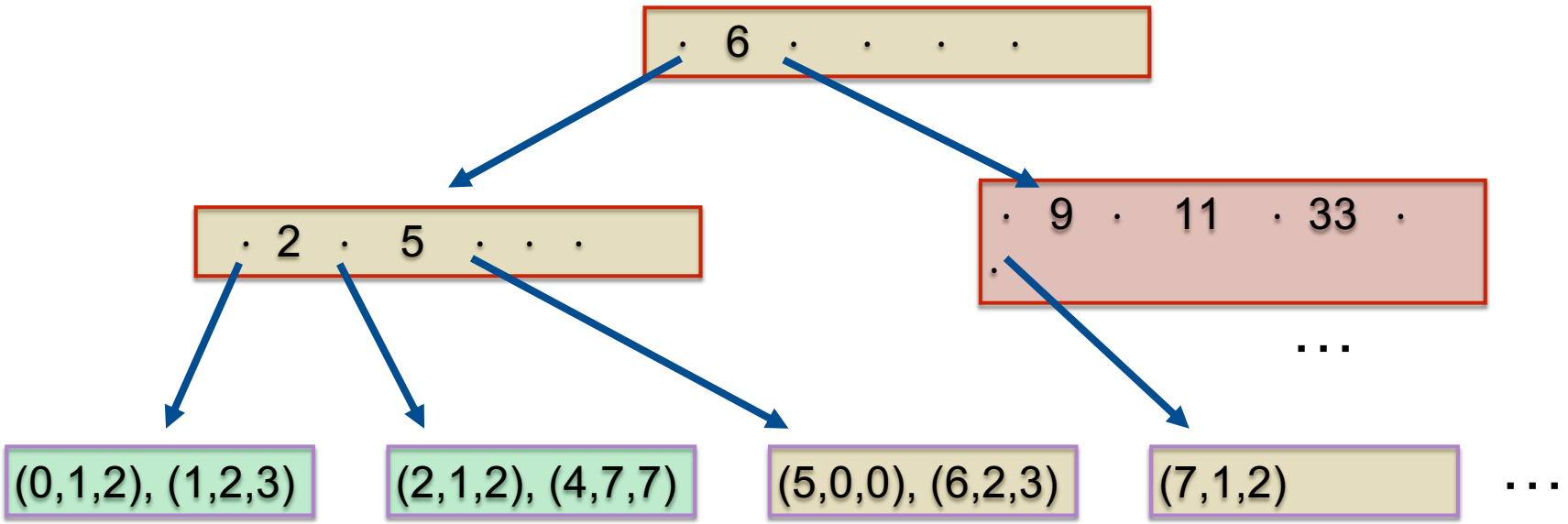
B+ tree clustered

R(a,b,c) … índice sobre a … a > 5



B+ tree clustered

R(a,b,c) … índice sobre a … a > 5

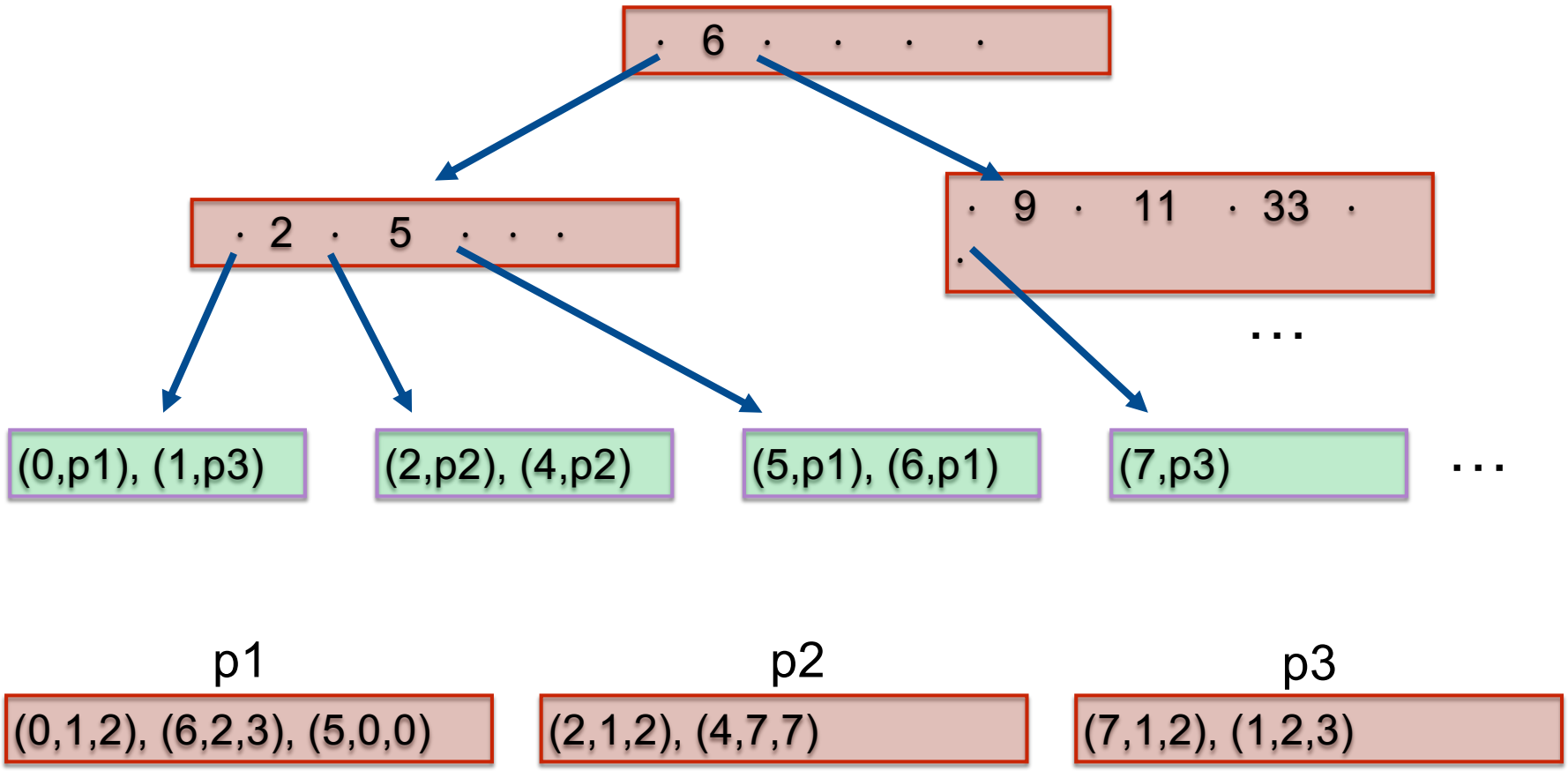


#(I/O) = $h + B_m$

B_m : nr de bloques con tuplas que calzan búsqueda

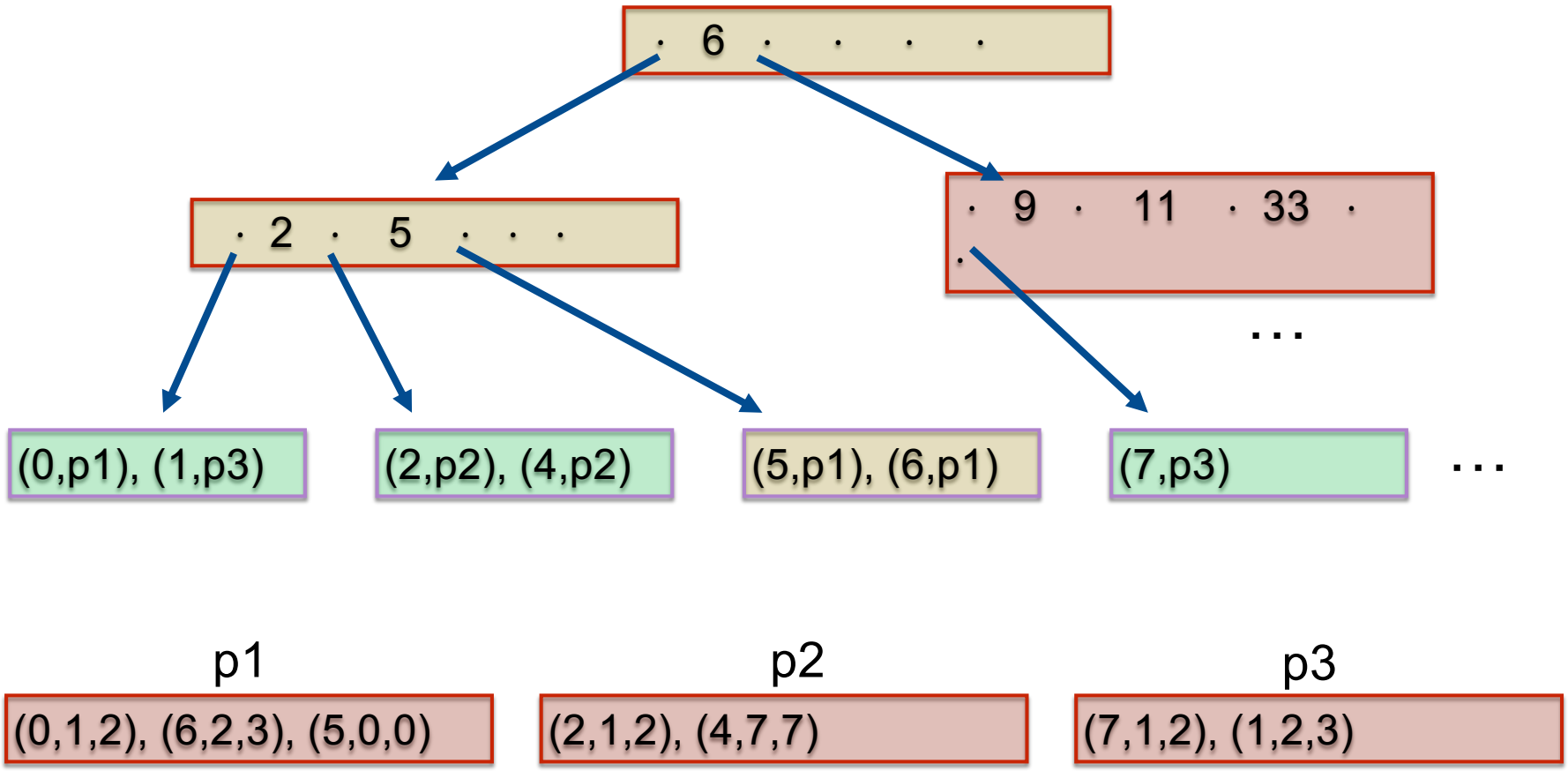
B+ tree unclustered

R(a,b,c) … índice sobre a … a > 5



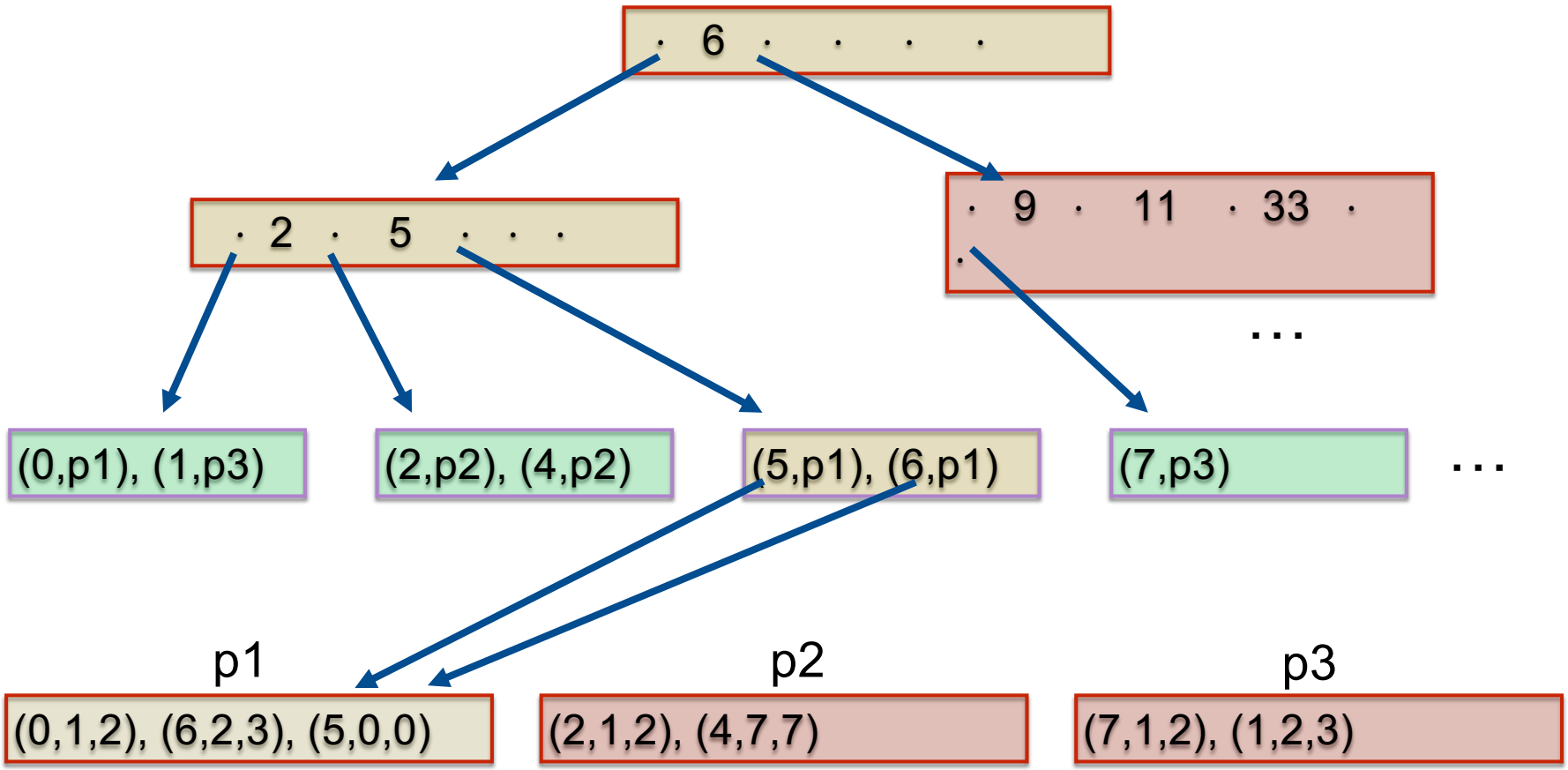
B+ tree unclustered

R(a,b,c) … índice sobre a … a > 5



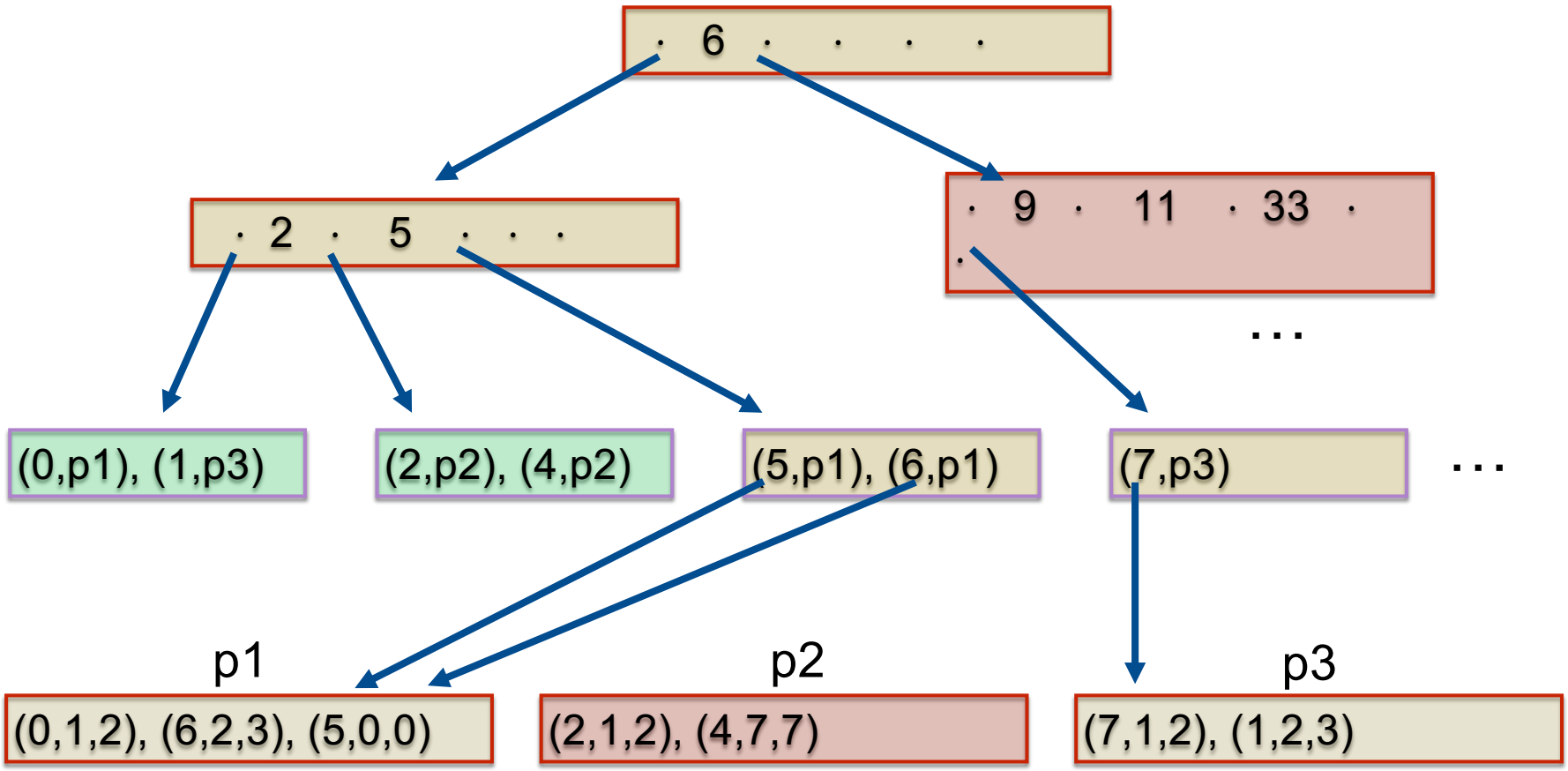
B+ tree unclustered

R(a,b,c) … índice sobre a … a > 5



B+ tree unclustered

R(a,b,c) … índice sobre a … a > 5

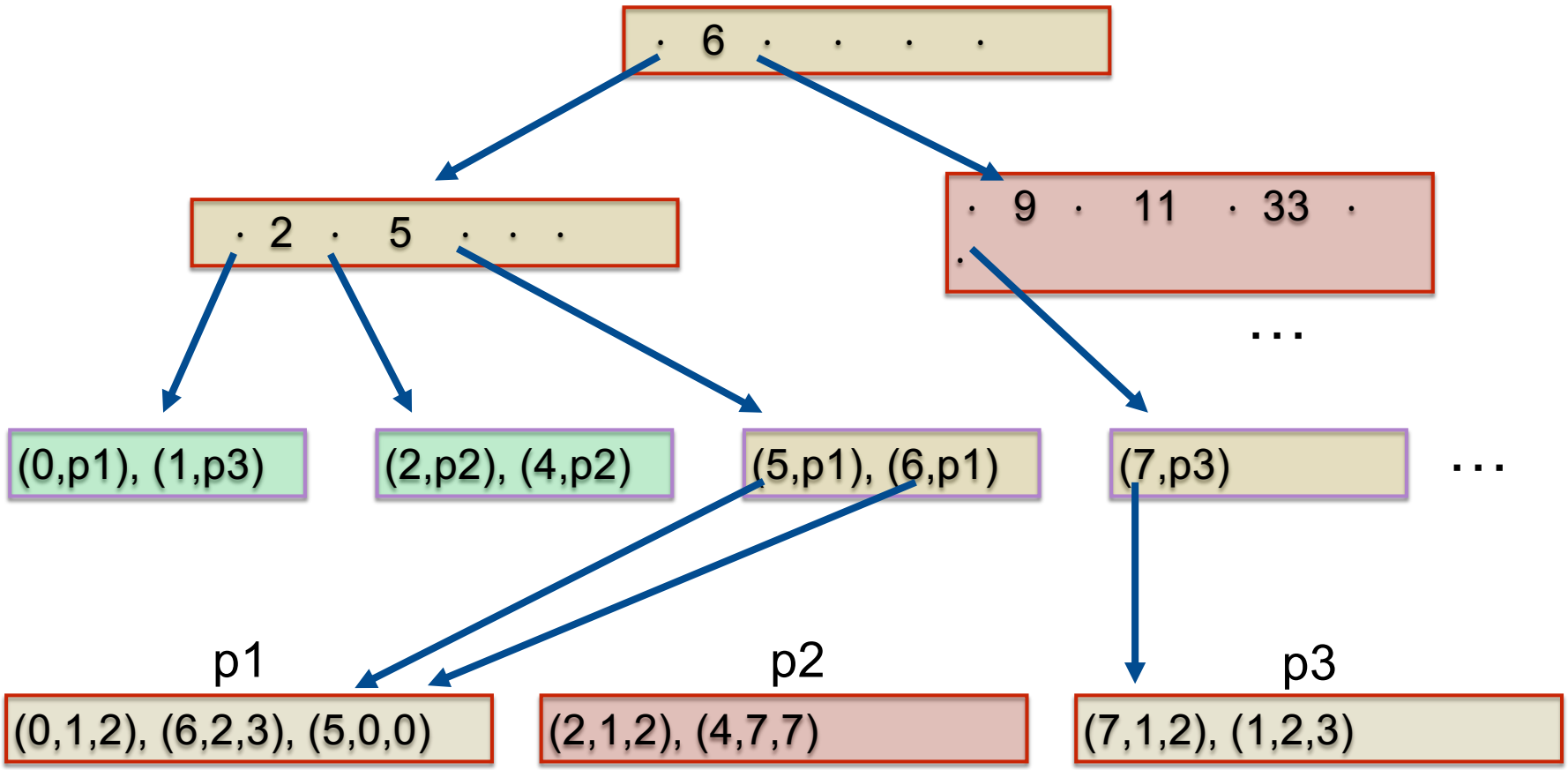


B+ tree unclustered

$$\#(I/O) = h + \left\lceil \frac{n}{P} \right\rceil + n$$

n : número de tuplas que calzan con búsquedas
P : Cantidad de punteros por bloque

R(a,b,c) … índice sobre a … a > 5



B+ tree unclustered

R(a,b,c):

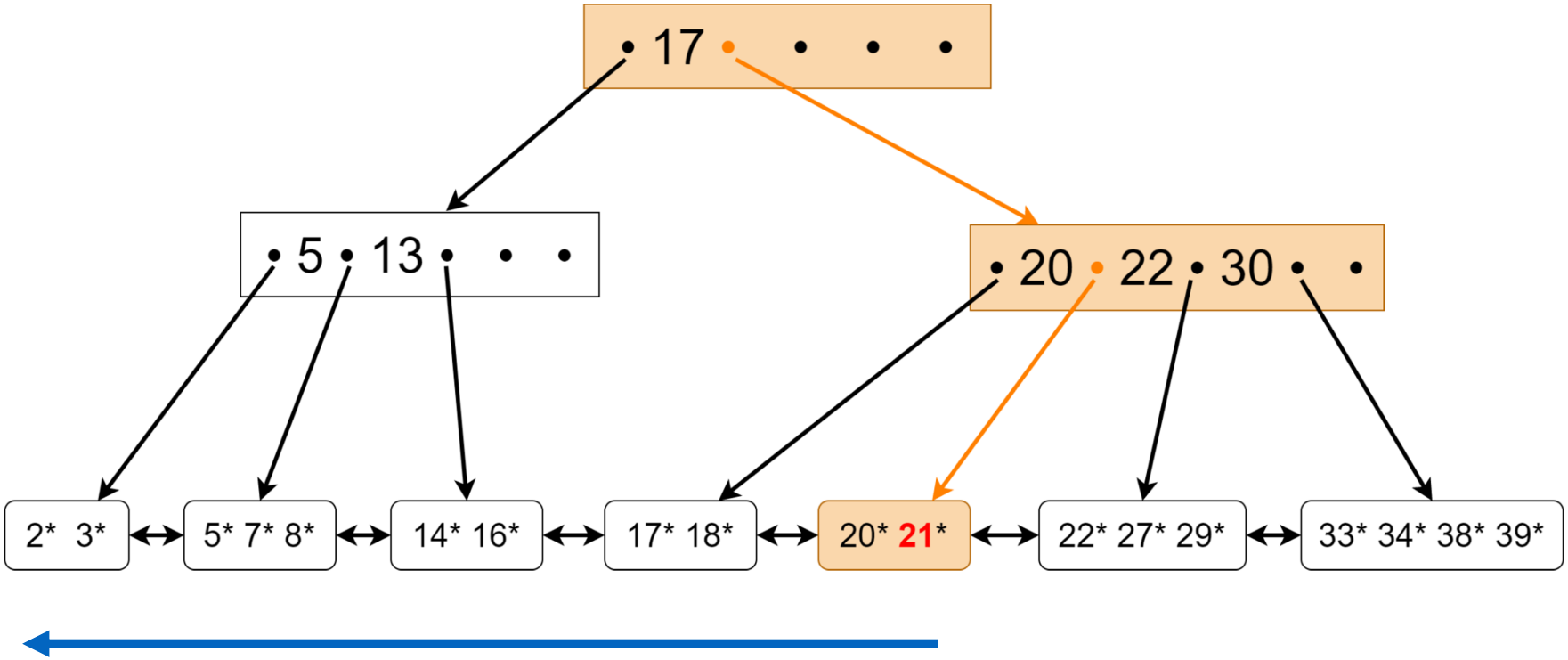
- 1M tuplas
- valor de a distribuido uniformemente en [0,1.000.000]
- B tuplas por página
- P > B punteros en una página (hojas a 100% ocupación)
- h altura del B+ tree

SELECT * FROM R WHERE a<21

Costo:

- Buscar a = 21 … h
- Scan hacia la izquierda

B+ tree unclustered



B+ tree unclustered

R(a,b,c):

- 1M tuplas
- valor de a distribuido uniformemente en [0,1.000.000]
- B tuplas por página
- P > B punteros en una página (hojas a 100% ocupación)
- h altura del B+ tree

Costo:

- Buscar a = 21 -> h
- Por uniformidad + a es llave para a = 21 -> 21 tuplas
- Cúantos punteros hay en una página? -> 21/P
- Más buscar la página en el disco (unclustered) -> 21

$$h + (\left\lceil \frac{21}{P} \right\rceil - 1) + 21$$

B+ tree unclustered

R(a,b,c):

- 1M tuplas
- valor de a distribuido uniformemente en [0,1.000.000]
- B tuplas por página
- P > B punteros en una página (hojas a 100% ocupación)
- h altura del B+ tree

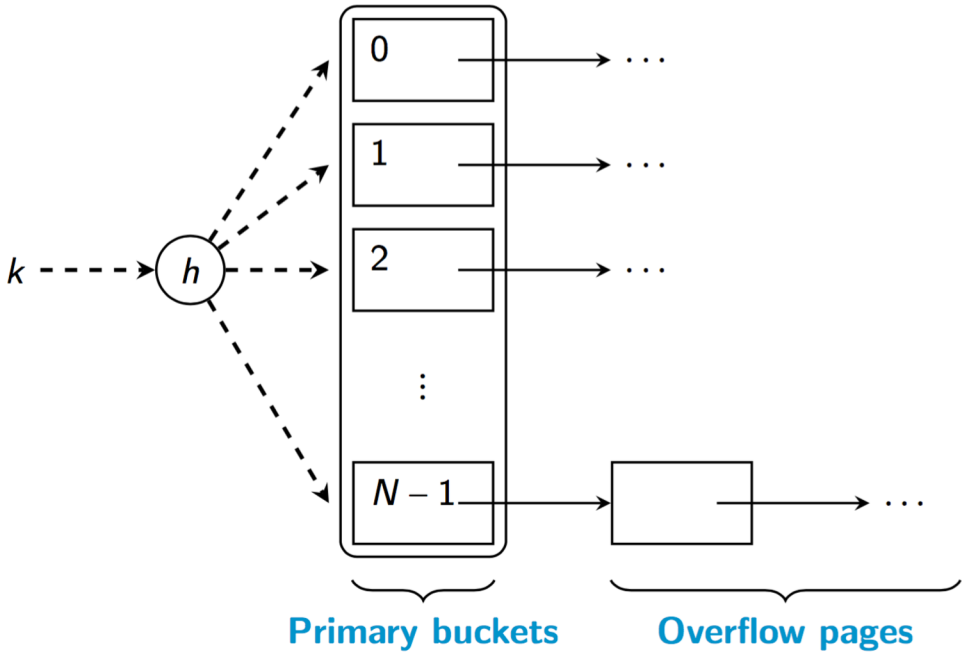
Costo: $h + (\left\lceil \frac{21}{P} \right\rceil - 1) + 21$

Bajar hacia a = 21

Scan a izquierda hay P punteros/página

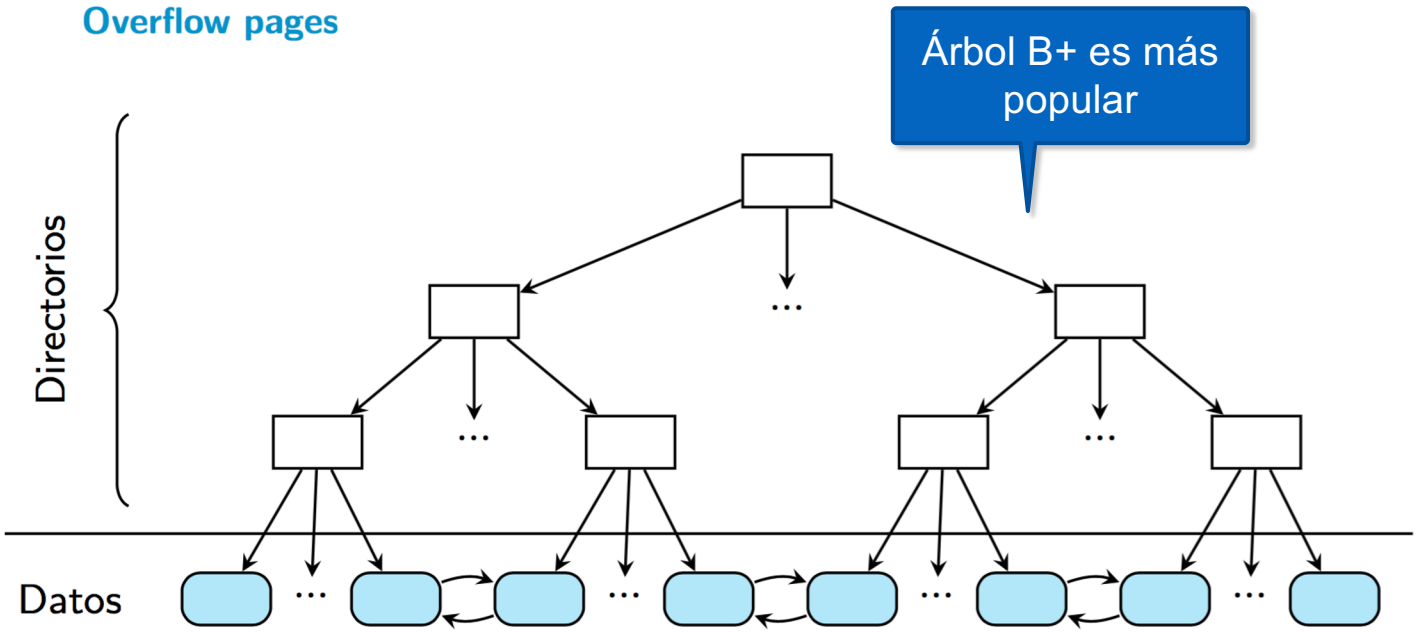
Para cada a recibo puntero, no la tupla

Índices: Hash vs. Árbol B+



Levemente más eficiente para búsquedas exactas asumiendo una función de hash ideal. Muy ineficientes para búsqueda de rango.

Mucho más eficiente para búsquedas por rango



Recursos para estudiar

Database Management Systems, 3rd edition, de Raghu Ramakrishnan y Johannes Gehrke. Part III