

Bases de Datos

Clase 11: Fundamentos de EDD y Almacenamiento e
Índices

Agenda

Costo de consulta

EEDD

Índices

Hash/B+

Clustered/Unclustered

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



Memoria
Secundaria

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

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



Memoria
Principal

Los accesos a memoria principal son despreciables

- 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 ?

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

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

Costo de consulta	EEDD	Índices	Hash/B+	Clustered/Unclustered
-------------------	-------------	---------	---------	-----------------------

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.

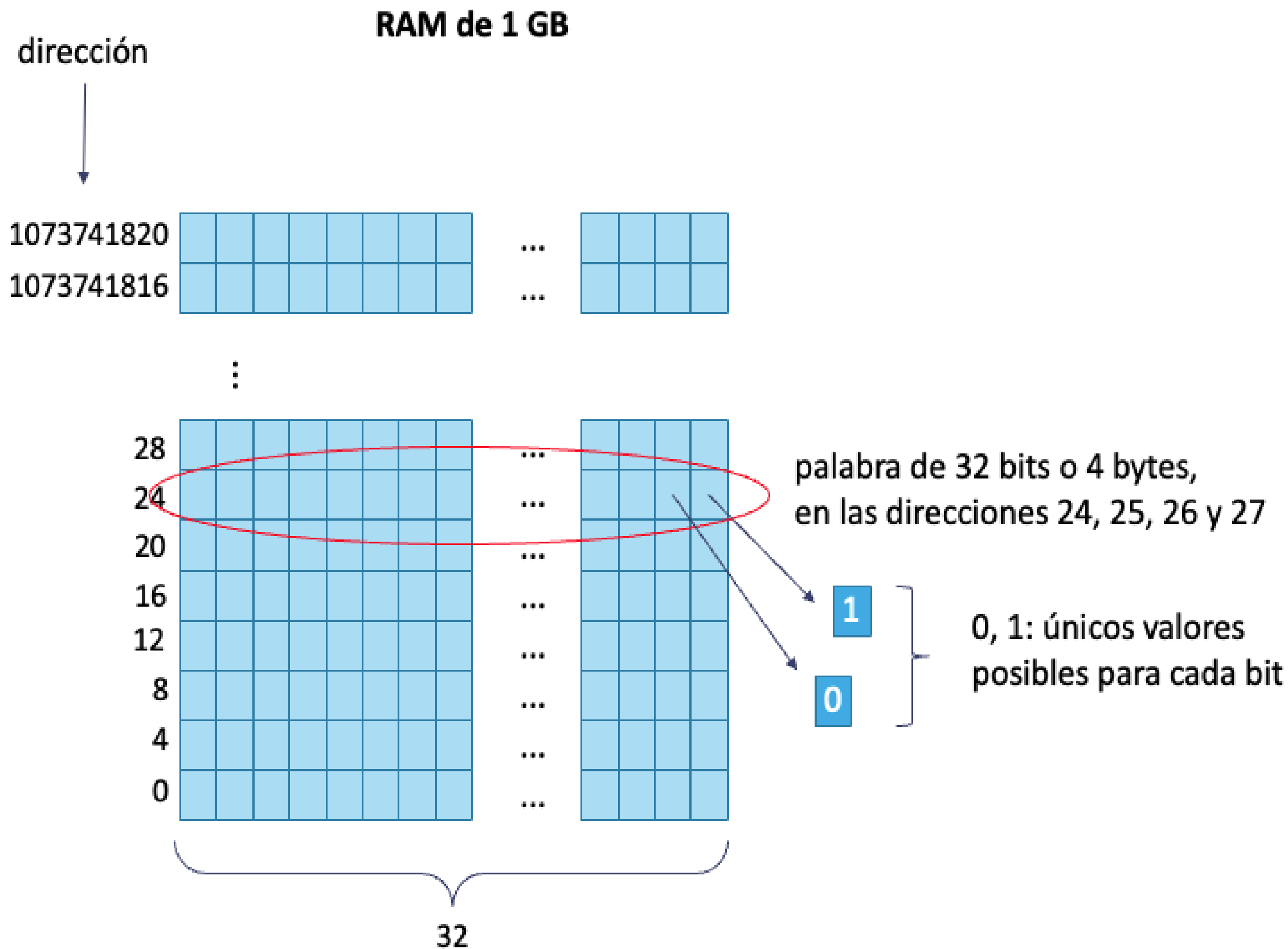
La memoria principal de un computador (RAM) puede ser imaginada como una gran tabla, arreglo o matriz de bits:

- Cierta 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

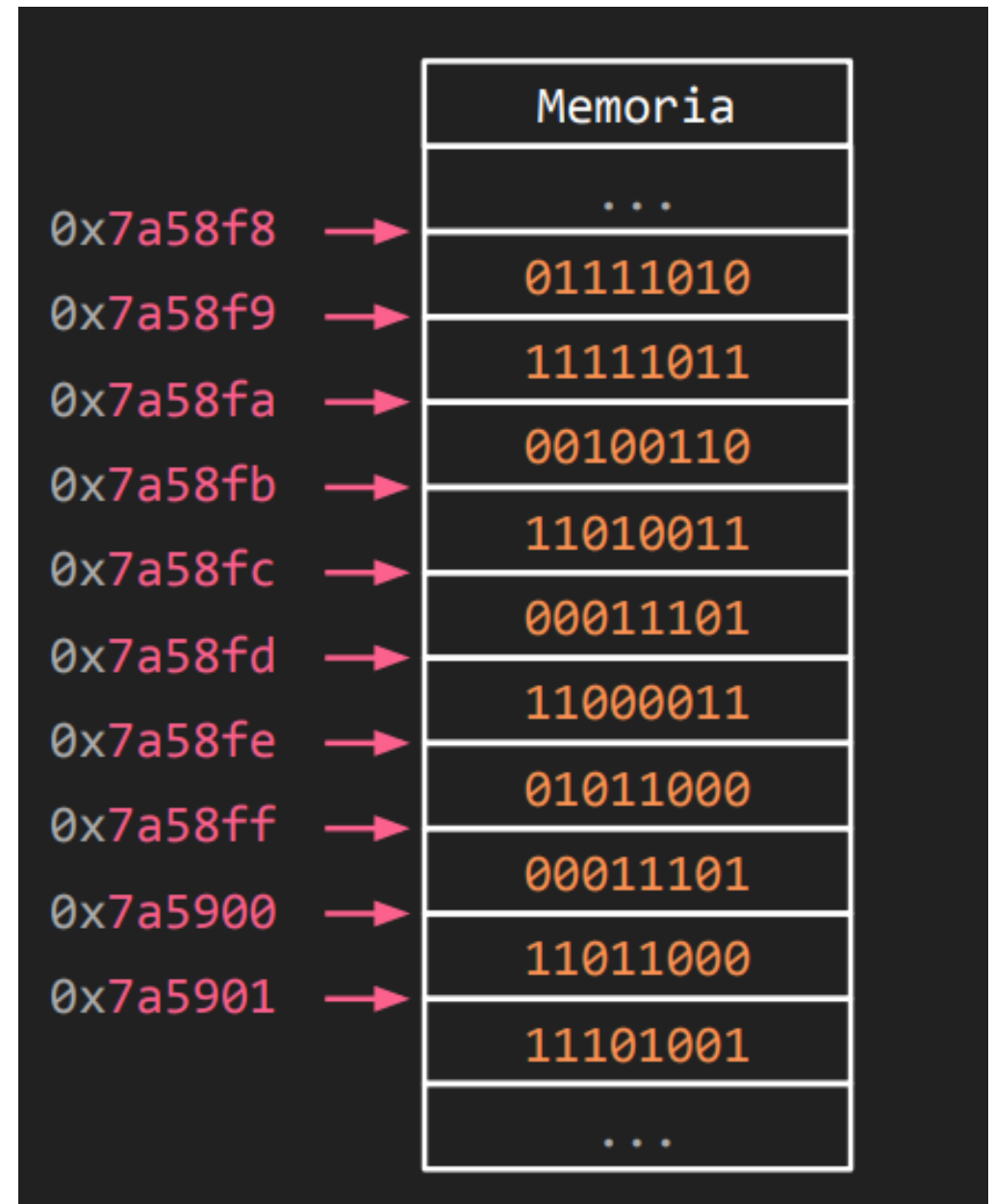


Memoria RAM y Punteros

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



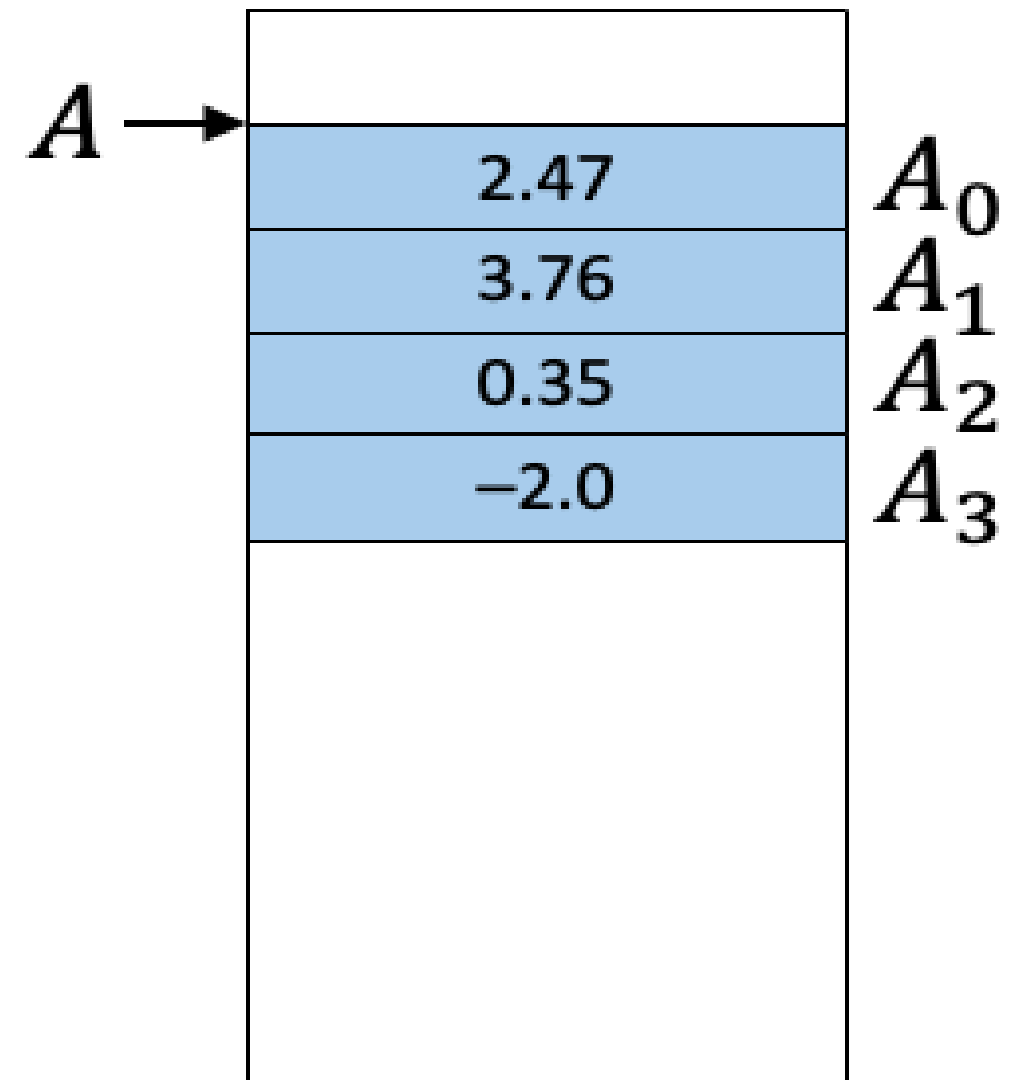
Un **puntero** es una variable cuyo **valor** es la **dirección** de memoria de otra variable.

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



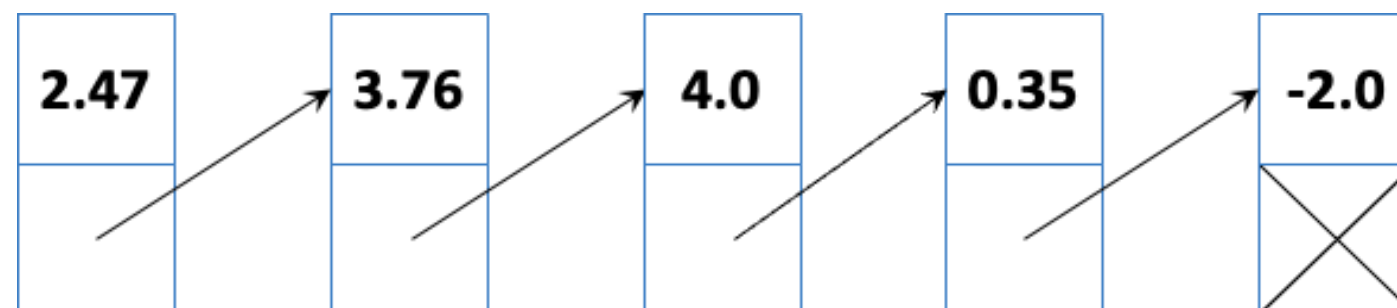
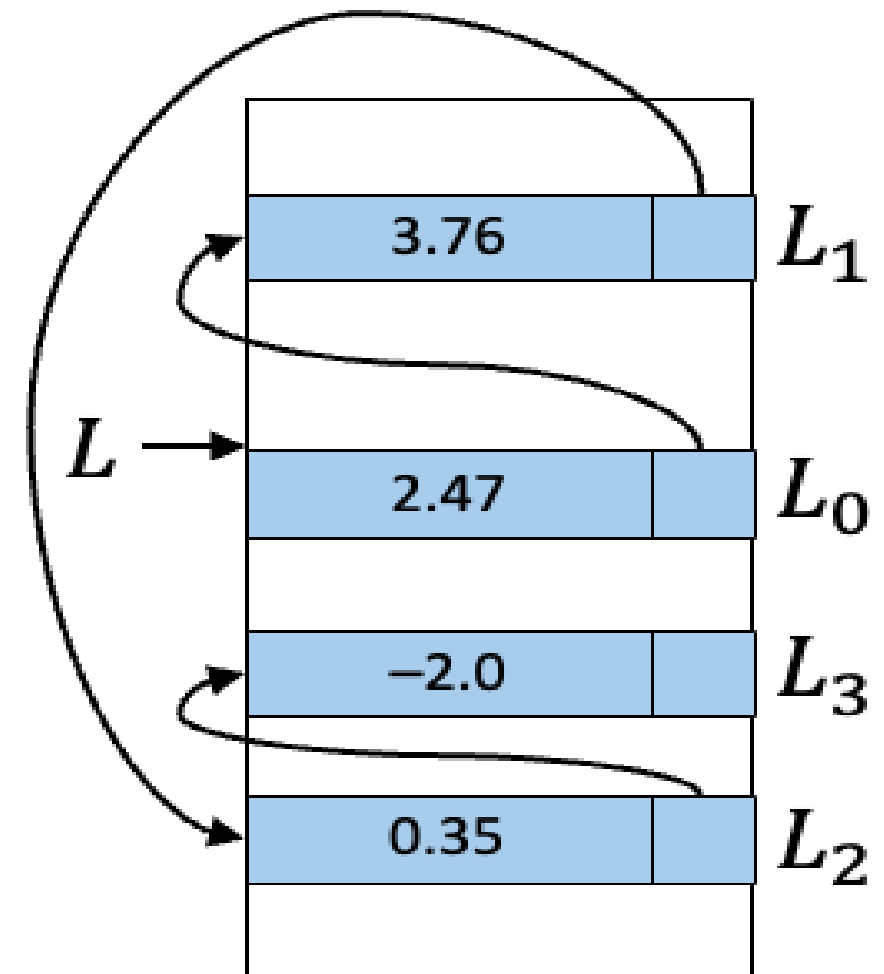
	2.47	3.76	0.35	-2.0	
--	------	------	------	------	--

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**



Costo de consulta	EEDD	Índices	Hash/B+	Clustered/Unclustered
-------------------	-------------	---------	---------	-----------------------

Tablas de Hash

Funciones de Hash

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

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

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

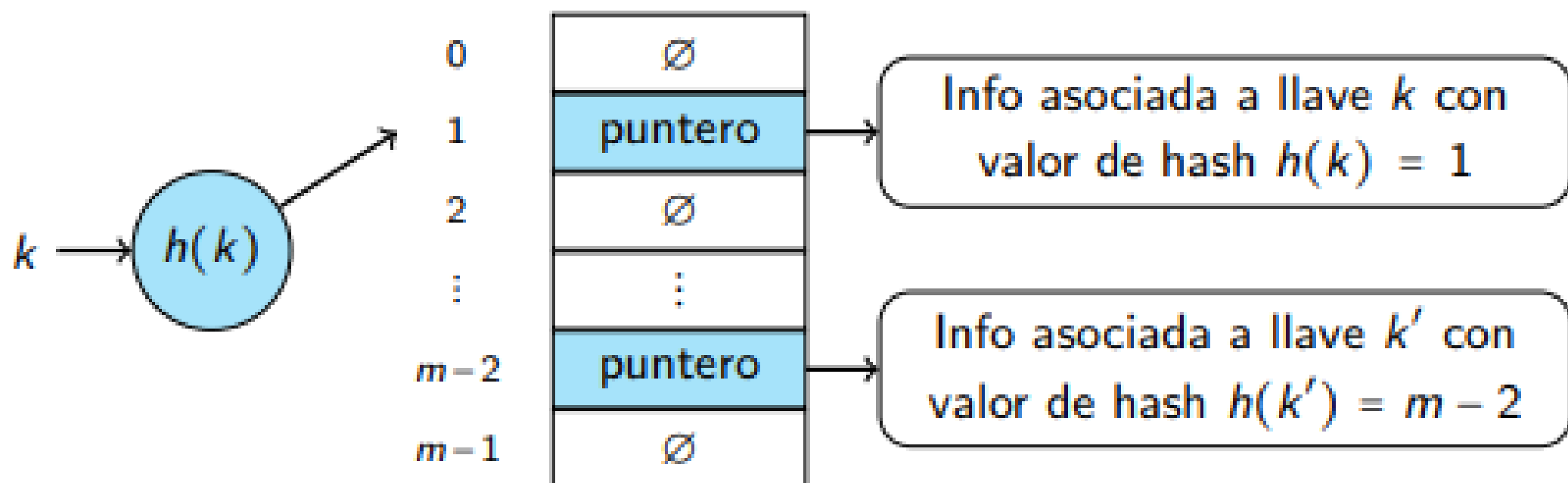
- Una función de hash nos permite mapear un espacio de llaves a otro mas pequeño (con m razonable)
- Una función de hash no necesariamente es inyectiva,
- Si $m < |K|$, no puede ser inyectiva.
- En la práctica, $m \ll |K|$

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

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

Diremos que tal A es de tamaño m .



Tablas de Hash

hashing general :

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

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

HashDelete (A, k):
 $A[h(k)] = \emptyset$

Pero sabemos que h no necesariamente es inyectiva

Es decir, puede ocurrir una colisión $h(k_1) = h(k_2)$ para $k_1 \neq k_2$

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 $h(k) \in \{0, \dots, m - 1\}$ para todo k . Todas las llaves con el mismo resto al dividir entre m generan una colisión, i.e.

$$h(k_1) = h(k_2) \Leftrightarrow k_1 \equiv_m k_2$$

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

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

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

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 $h(15) = 15 \bmod 7 = 1$

0	Ø
1	Ø
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø

Tablas de Hash - Inserciones

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

0	Ø
1	Ø
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø

0	Ø
1	Ø
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø

0	Ø
1	15
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø

Tablas de Hash - Inserciones

Ahora insertamos la llave 37:

Su valor de hash es $h(37) = 37 \bmod 7 = 2$

0	∅
1	15
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

Tablas de Hash - Inserciones

Ahora insertamos la llave 51

Su valor de hash es $h(51) = 51 \bmod 7 = 2$

0	∅	0	∅
1	15	1	15
2	37	2	37
3	∅	3	∅
4	∅	4	∅
5	∅	5	∅
6	∅	6	∅

¿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

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

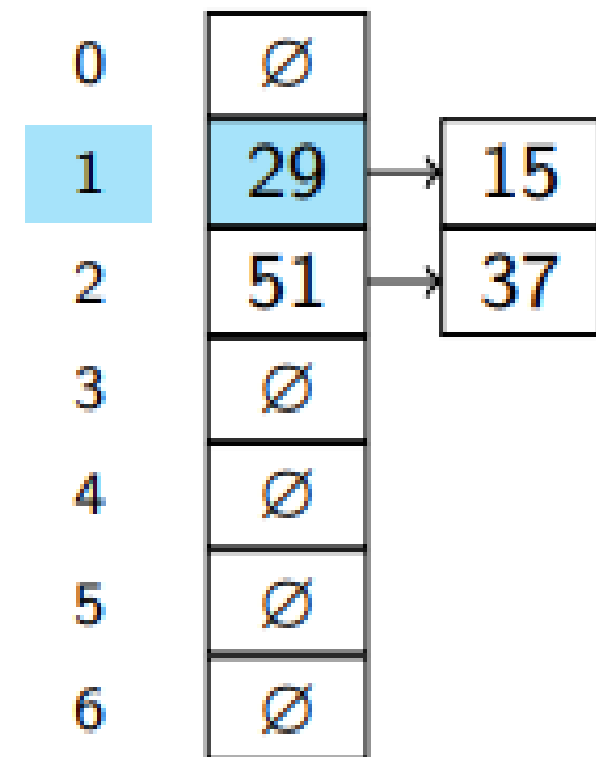
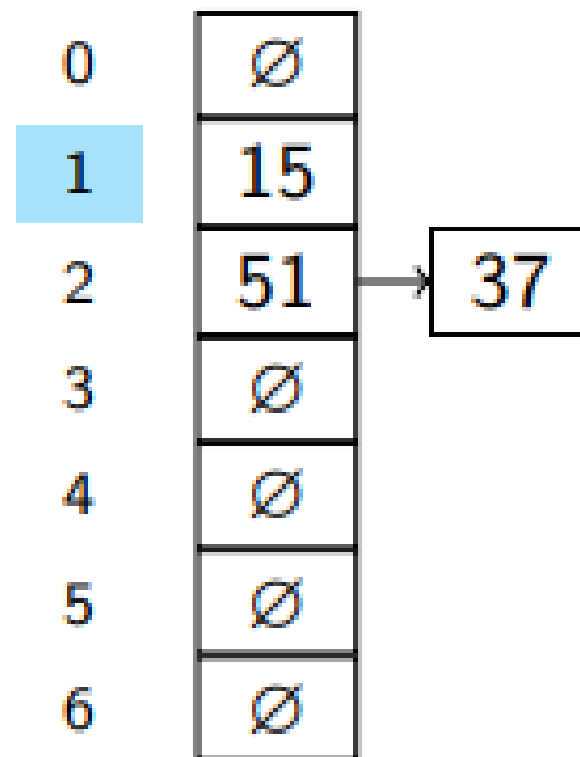
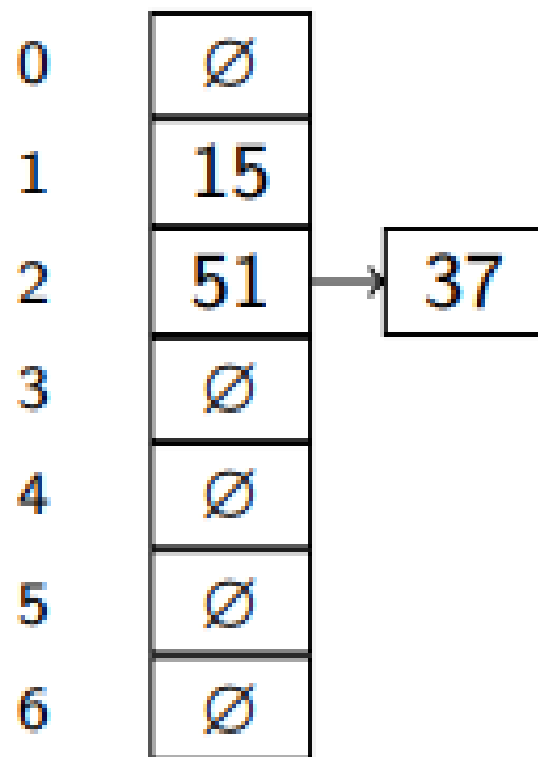
0	∅
1	15
2	51
3	∅
4	∅
5	∅
6	∅

→ 37

Tablas de Hash - Inserciones

Al insertar la llave 29 seguimos la misma idea

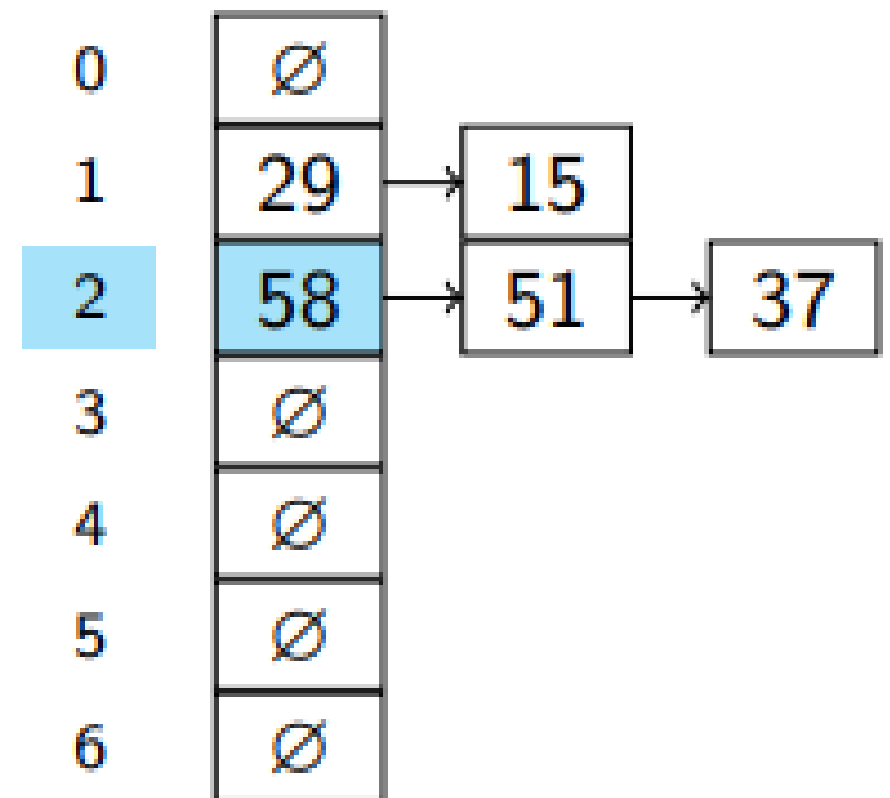
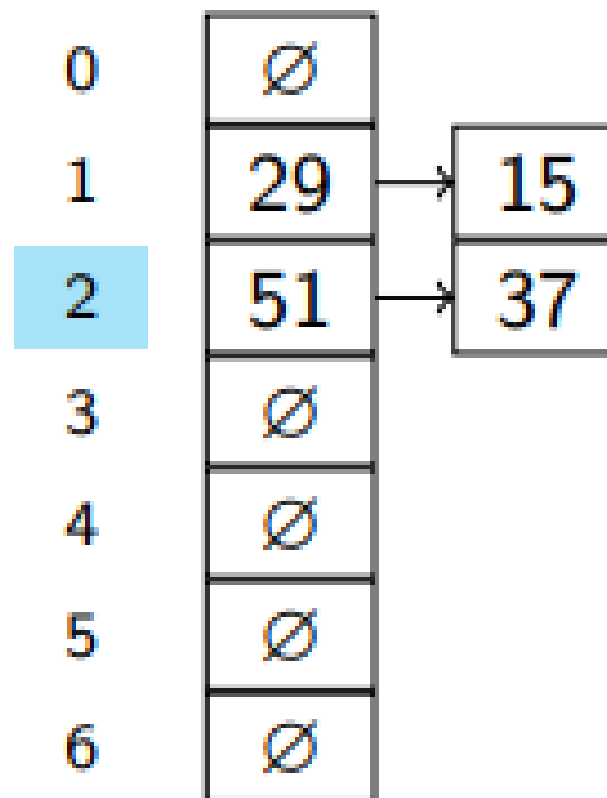
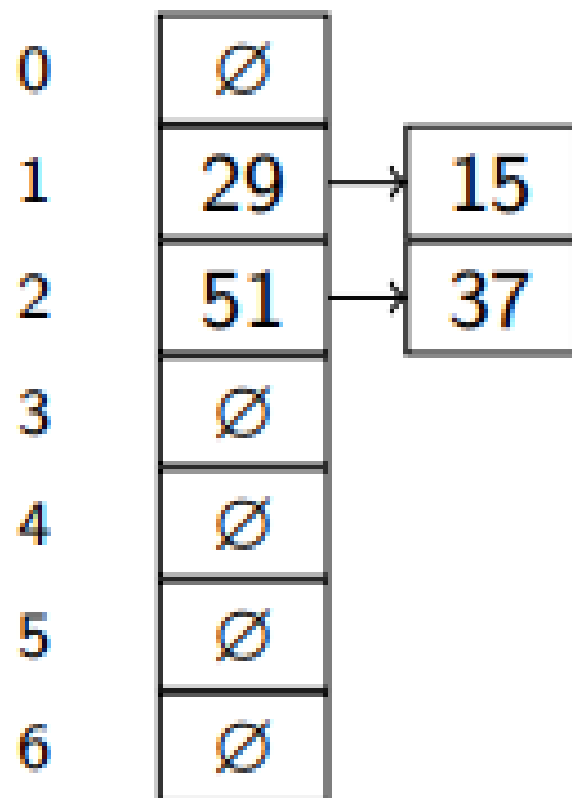
Su valor de hash es $h(29) = 29 \bmod 7 = 1$



Tablas de Hash - Inserciones

Al insertar la llave 58 seguimos la misma idea

Su valor de hash es $h(58) = 58 \bmod 7 = 2$



Tablas de Hash - Inserciones

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

ChainedHashSearch (A, k):

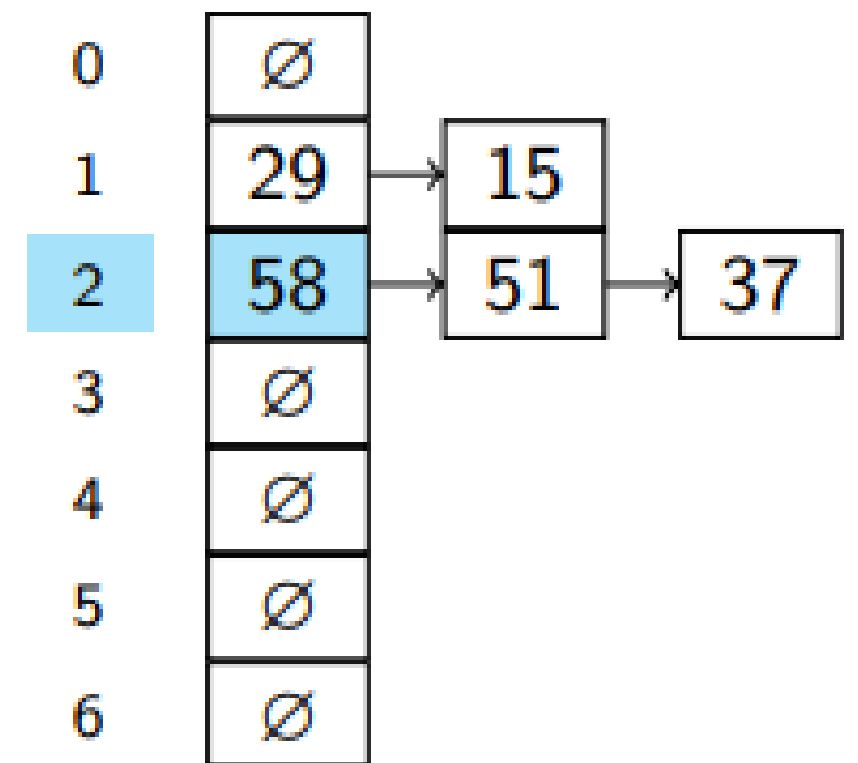
Buscar llave k en $A[h(k)]$

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 – 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

Costo de consulta	EEDD	Índices	Hash/B+	Clustered/Unclustered
Árboles				

Árboles

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

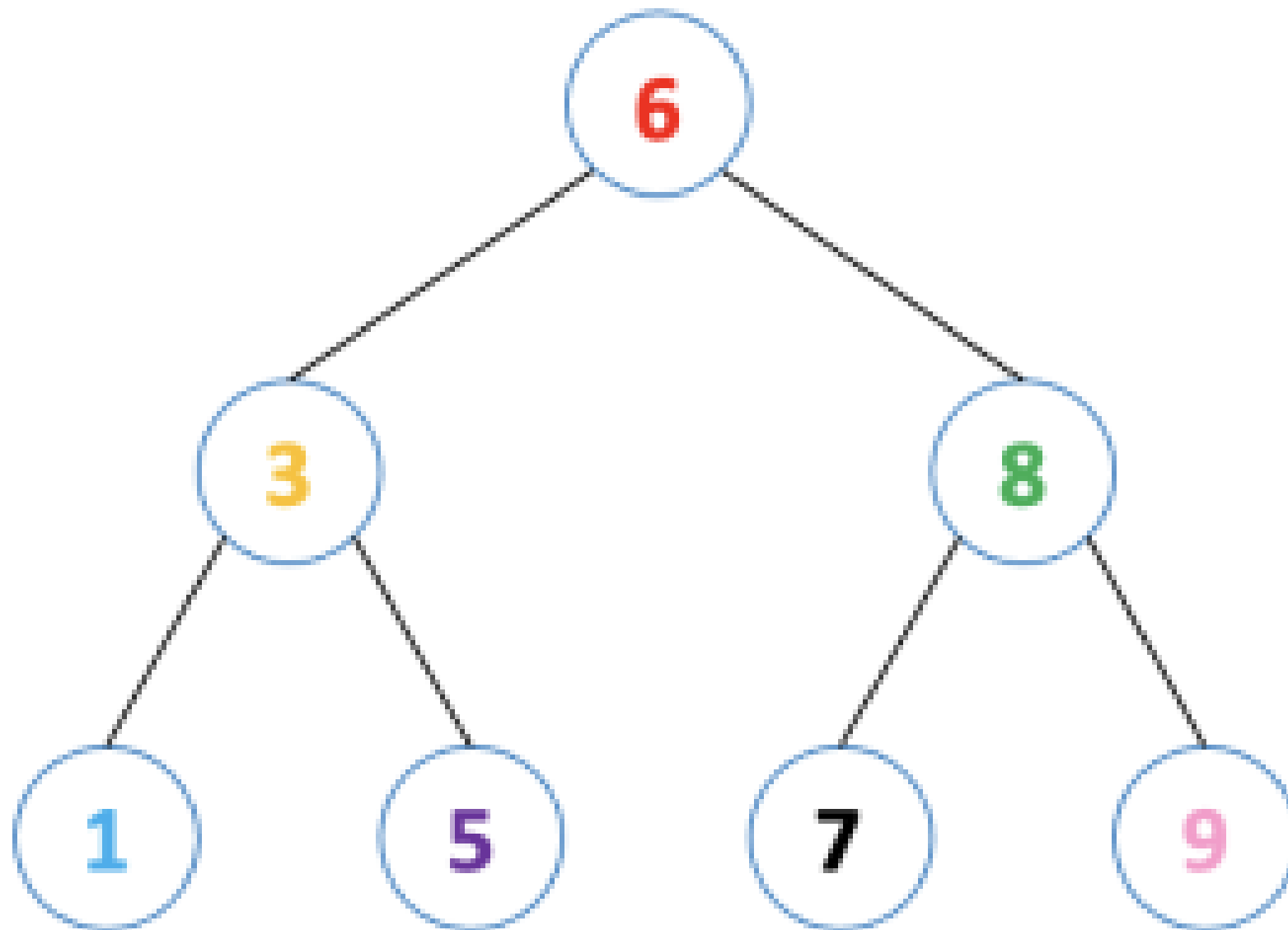
y que además, satisface la propiedad ABB: las llaves menores que la llave del nodo están en el sub-árbol izquierdo, y las llaves mayores, en el sub-árbol derecho.

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

Árboles- Ejemplo



Costo de consulta	EEDD	Índices	Hash/B+	Clustered/Unclustered
B+ Tree Index				

B+ Tree Index

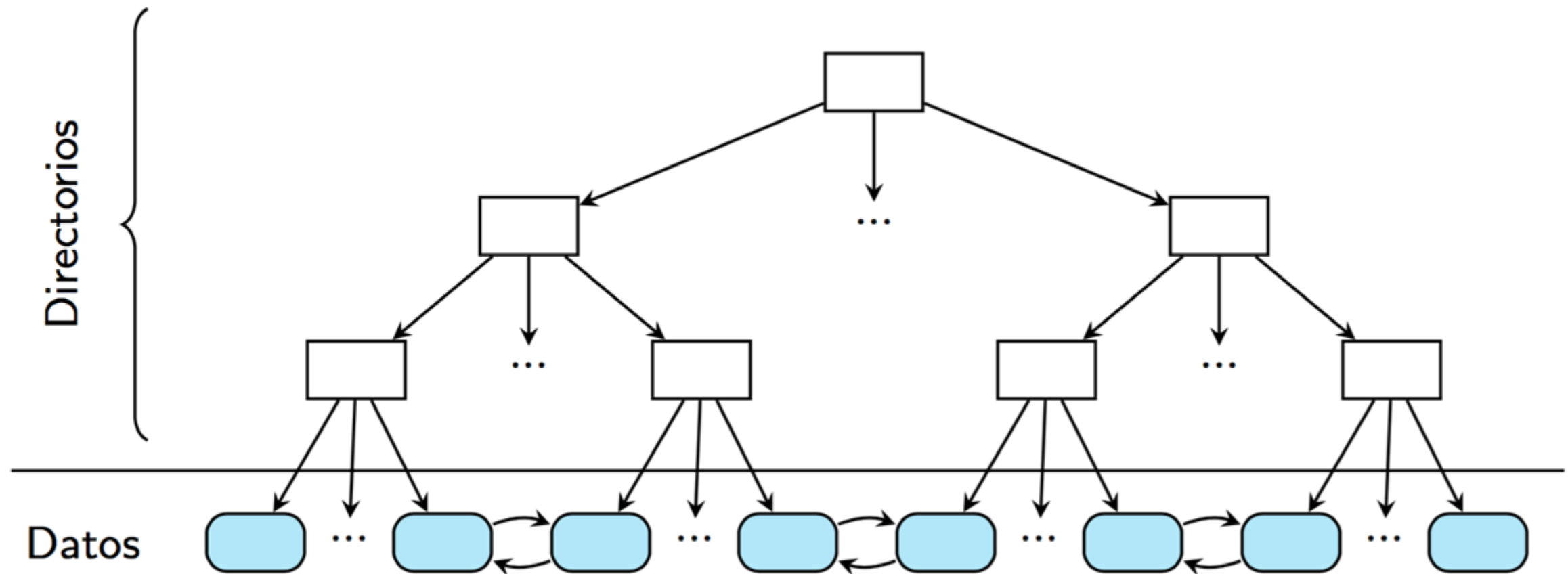
*“B+ Trees are by far the most important access path structure in database and file systems”,
Gray y Reuter (1993).*

B+ Tree Index

Es un índice basado en un árbol:

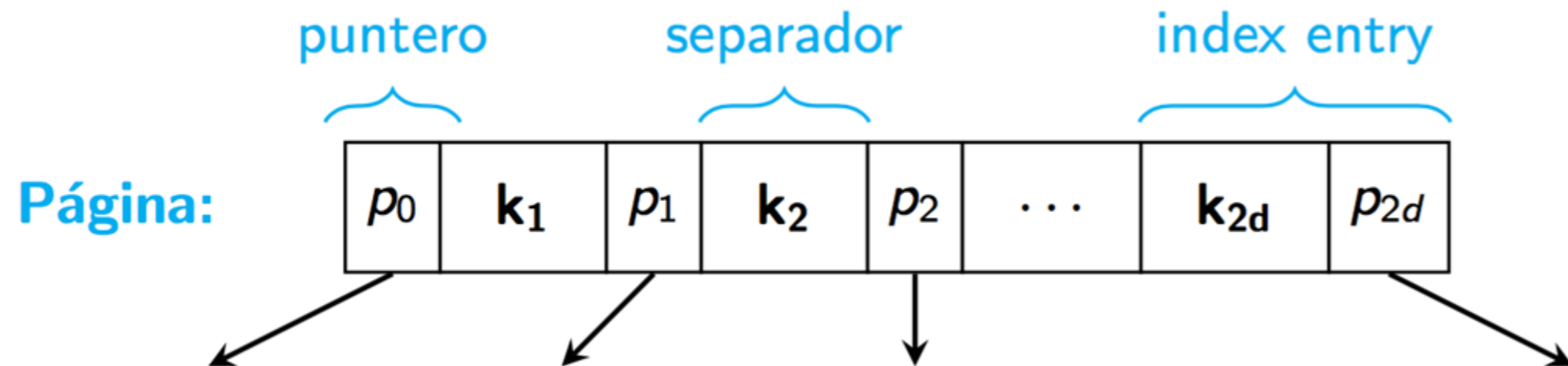
- Las páginas del disco representan nodos del árbol
- **Las tuplas** están en las hojas
- Provee un tiempo de búsqueda logarítmico
- Se comporta bien en consultas de rango
- Es el índice que usa Postgres sobre las llaves primarias (y en general, todos los sistemas)

B+ Tree Index



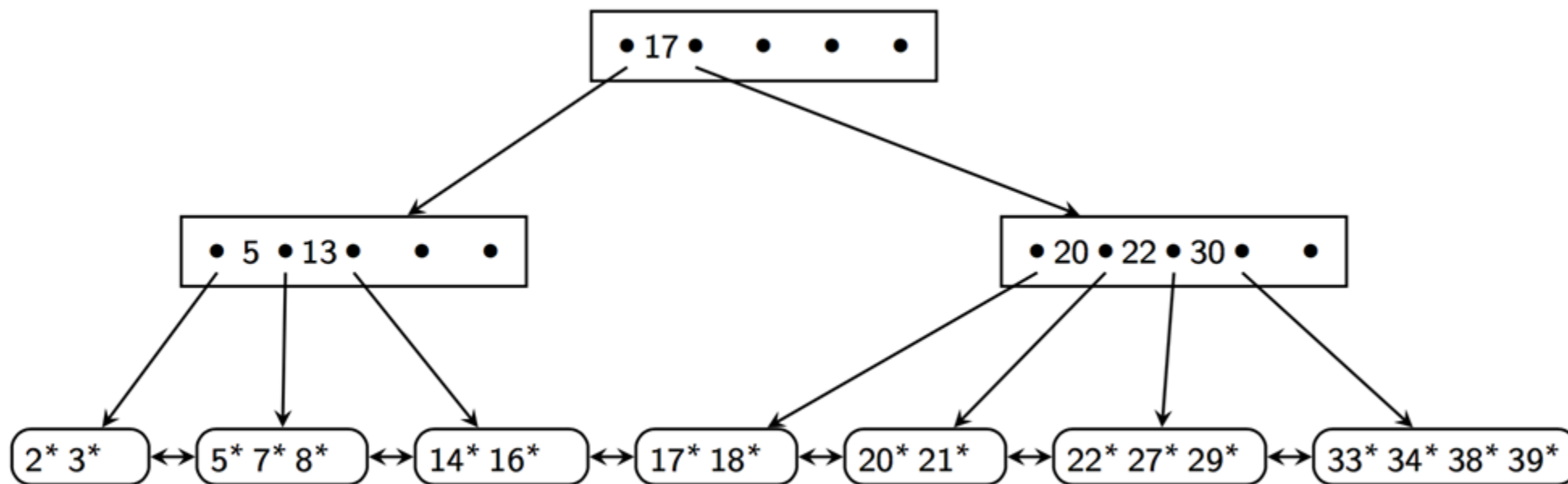
B+ Tree Index

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

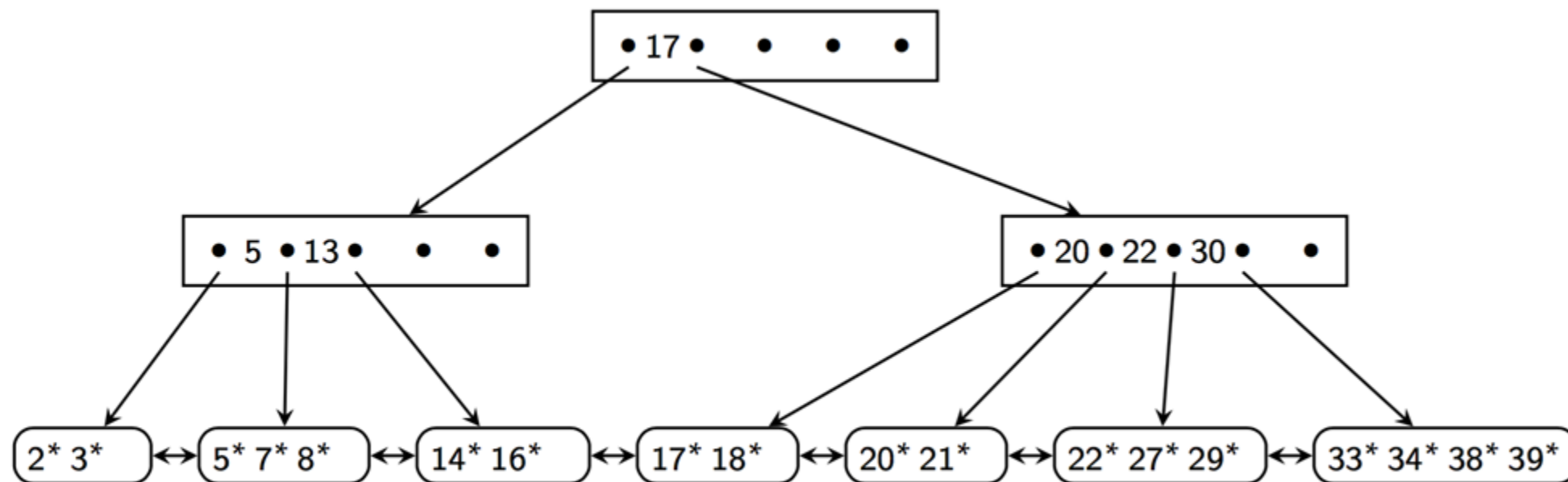


B+ Tree Index

Ejemplo: consulta por un valor

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

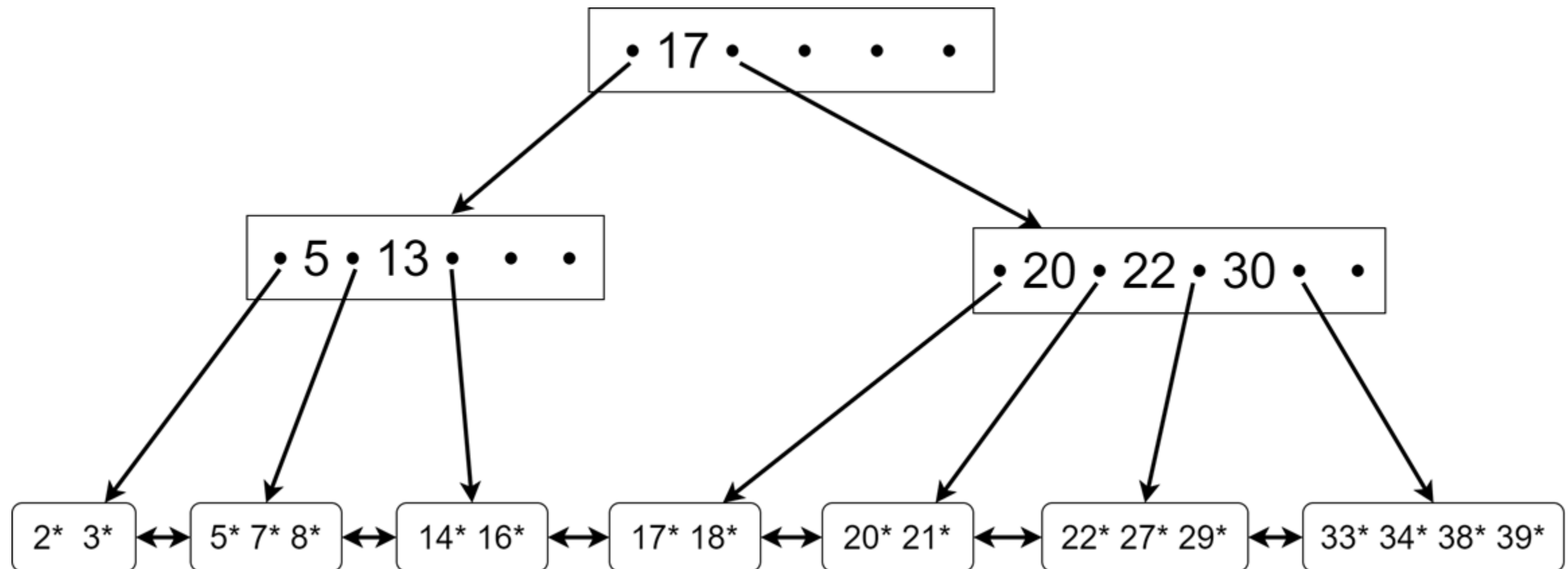
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

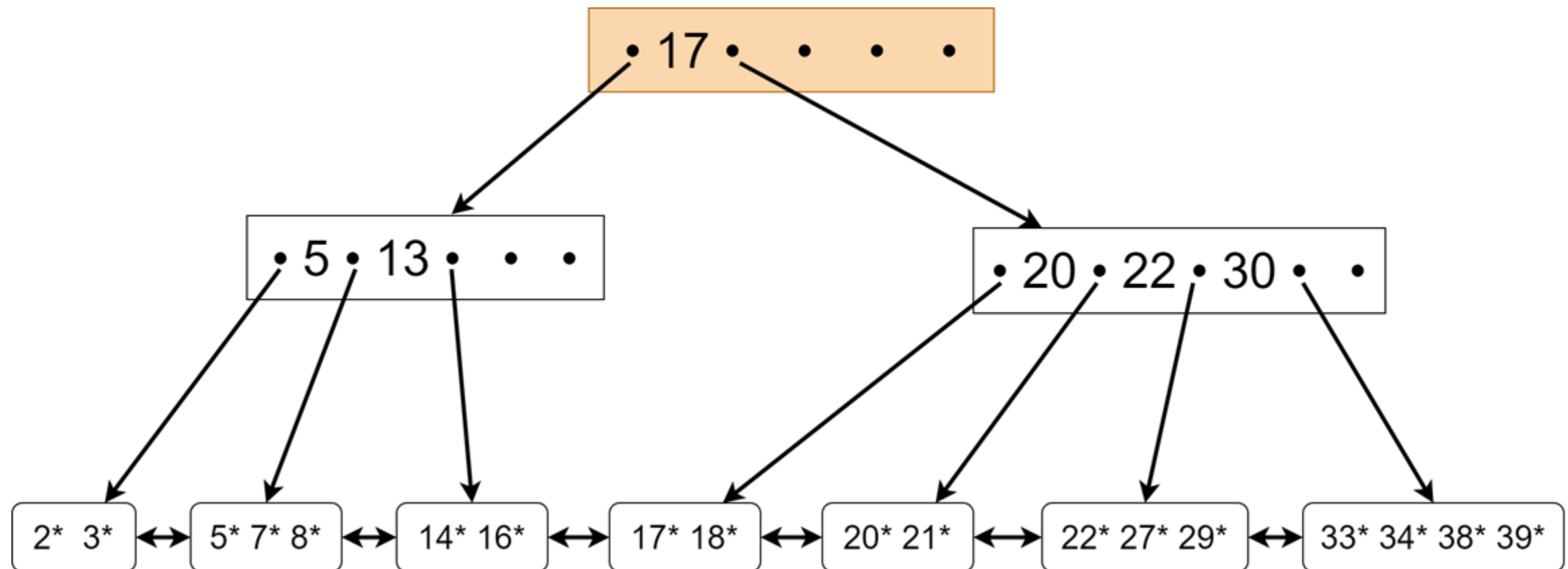
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

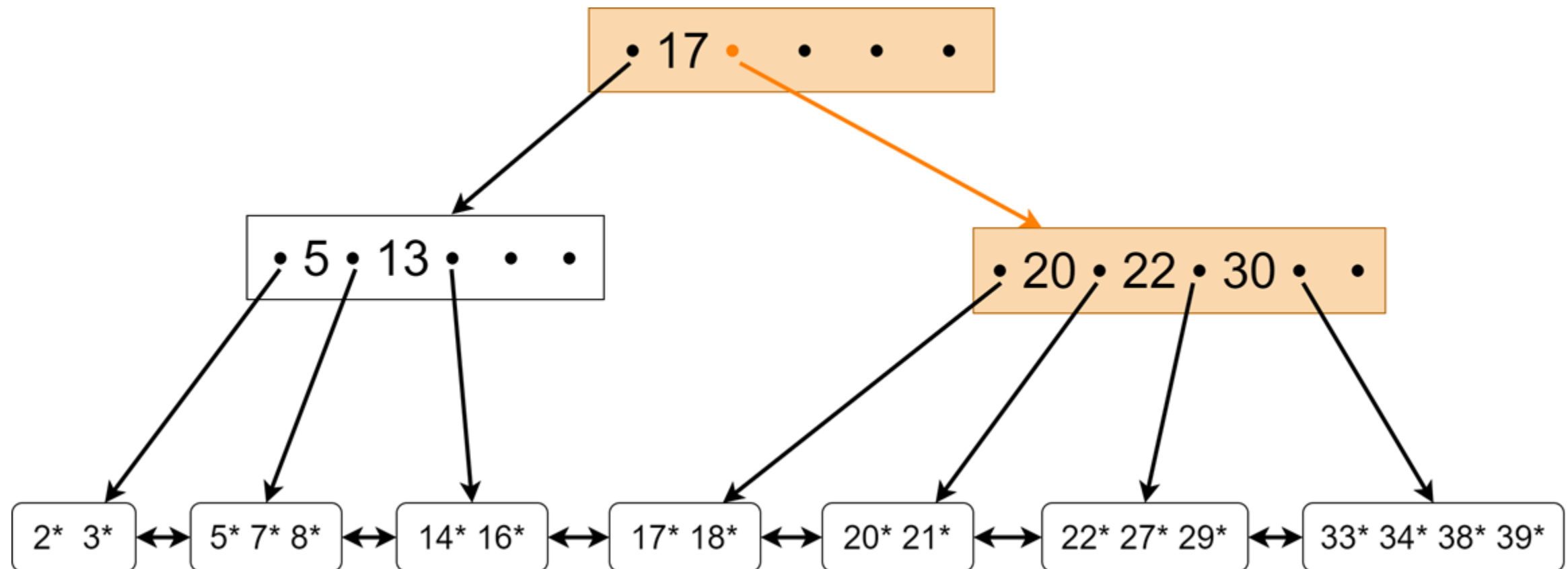
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

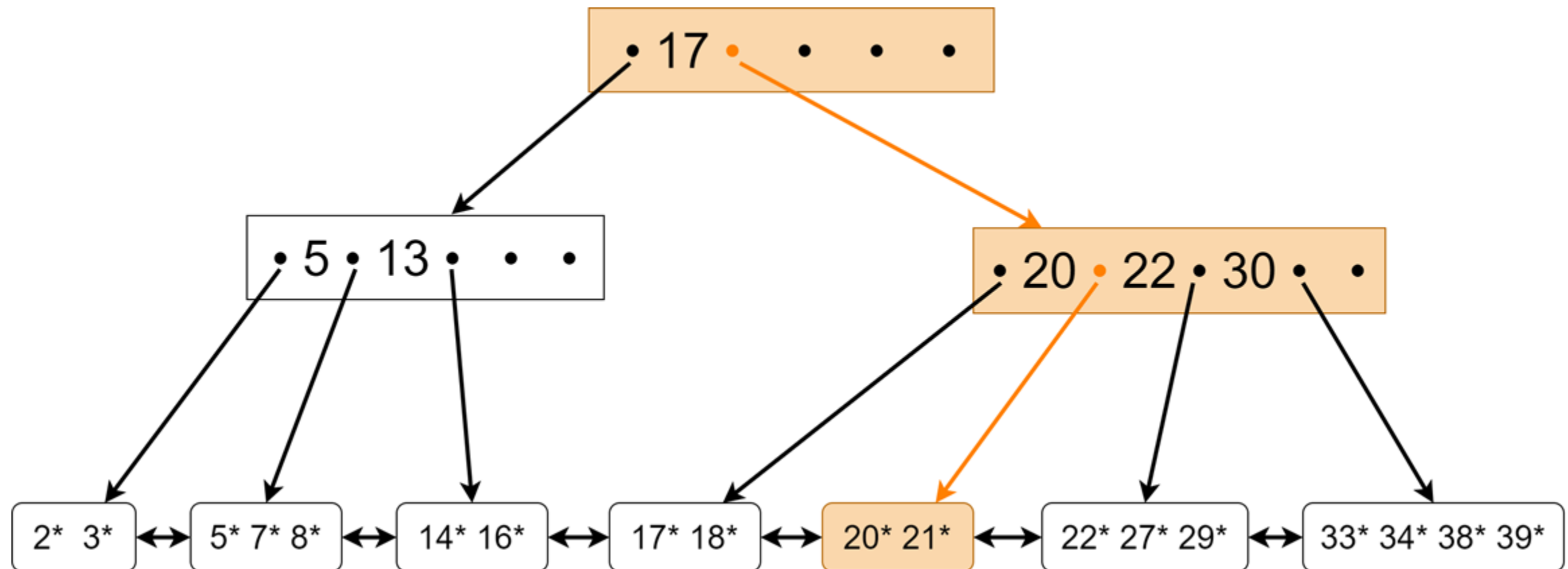
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

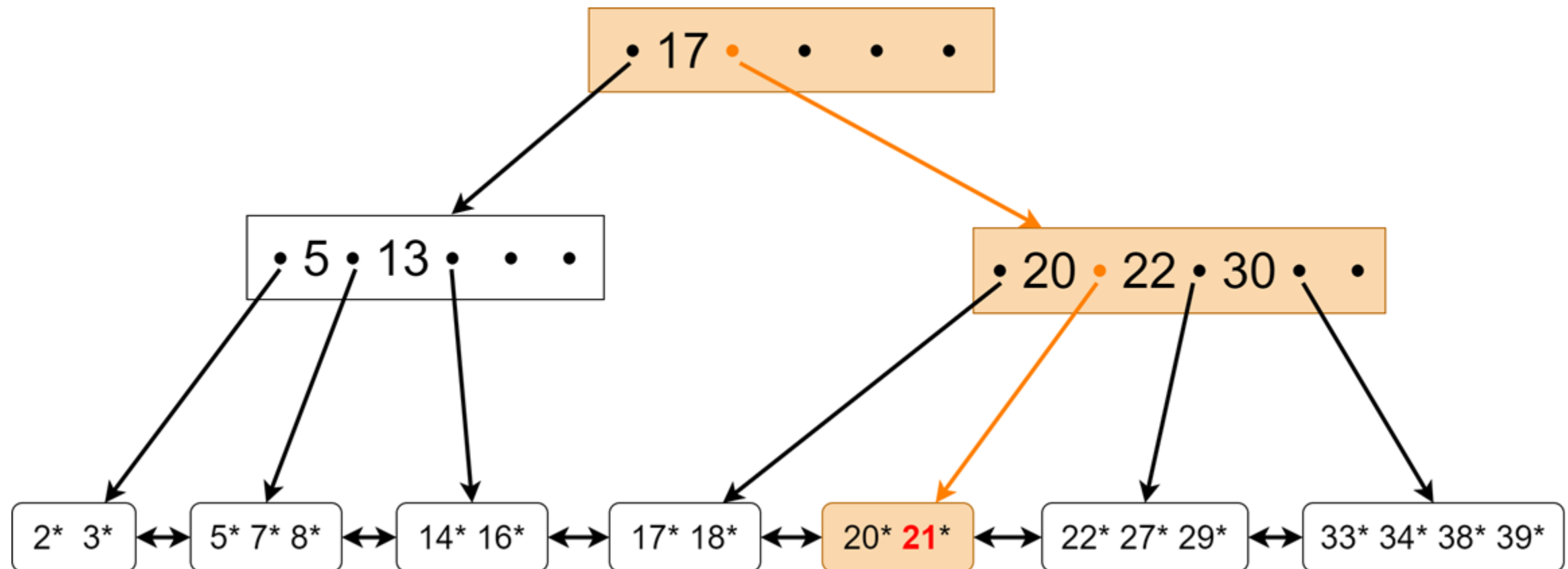
Queremos el artista con aid = 21



B+ Tree Index

Ejemplo: consulta por un valor

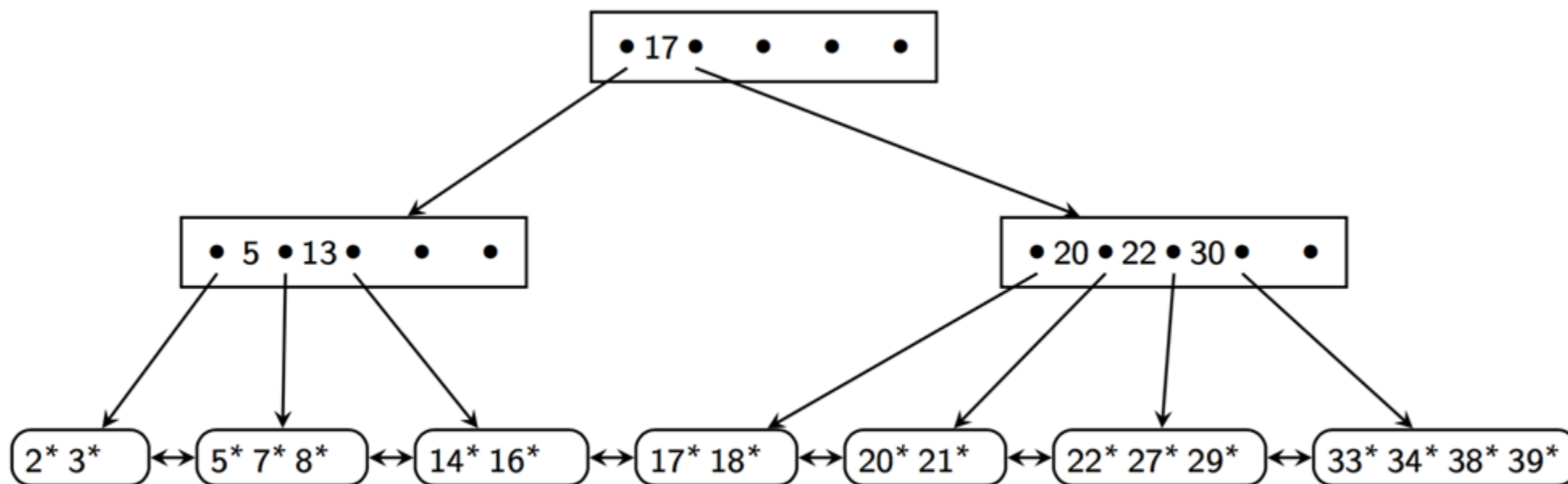
Queremos el artista con aid = 21



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

B+ Tree Index

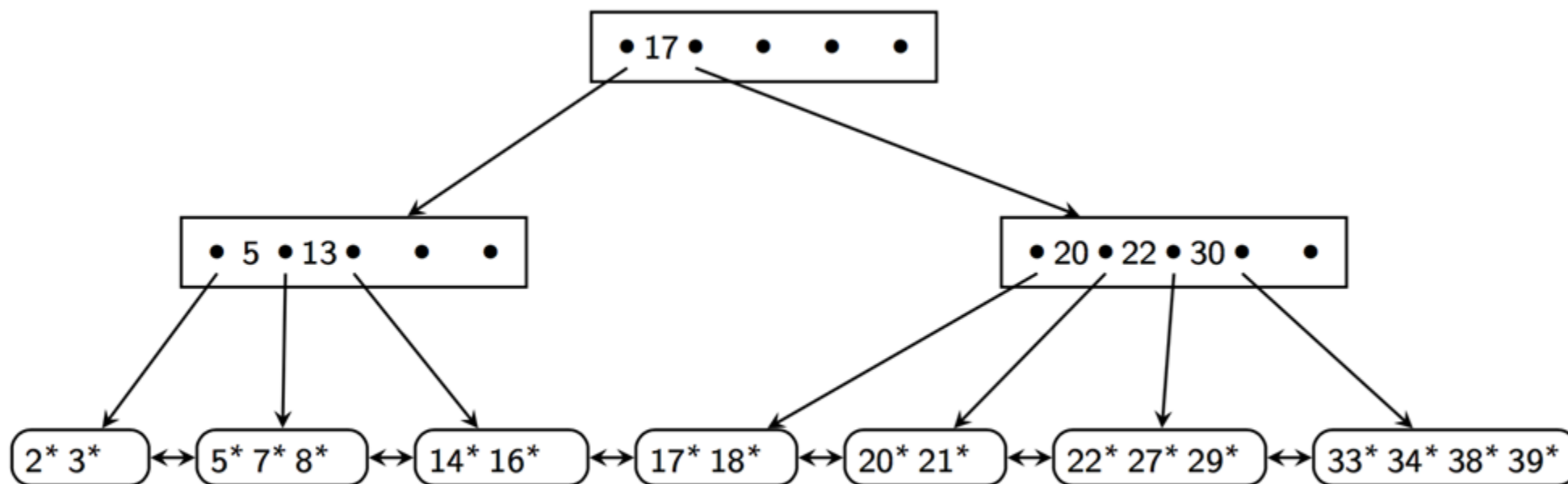
Sabemos que las tuplas van a estar ordenadas según el atributo indexado, y además las hojas están encadenadas unas con otras



B+ Tree Index

Ejemplo: consulta de rango

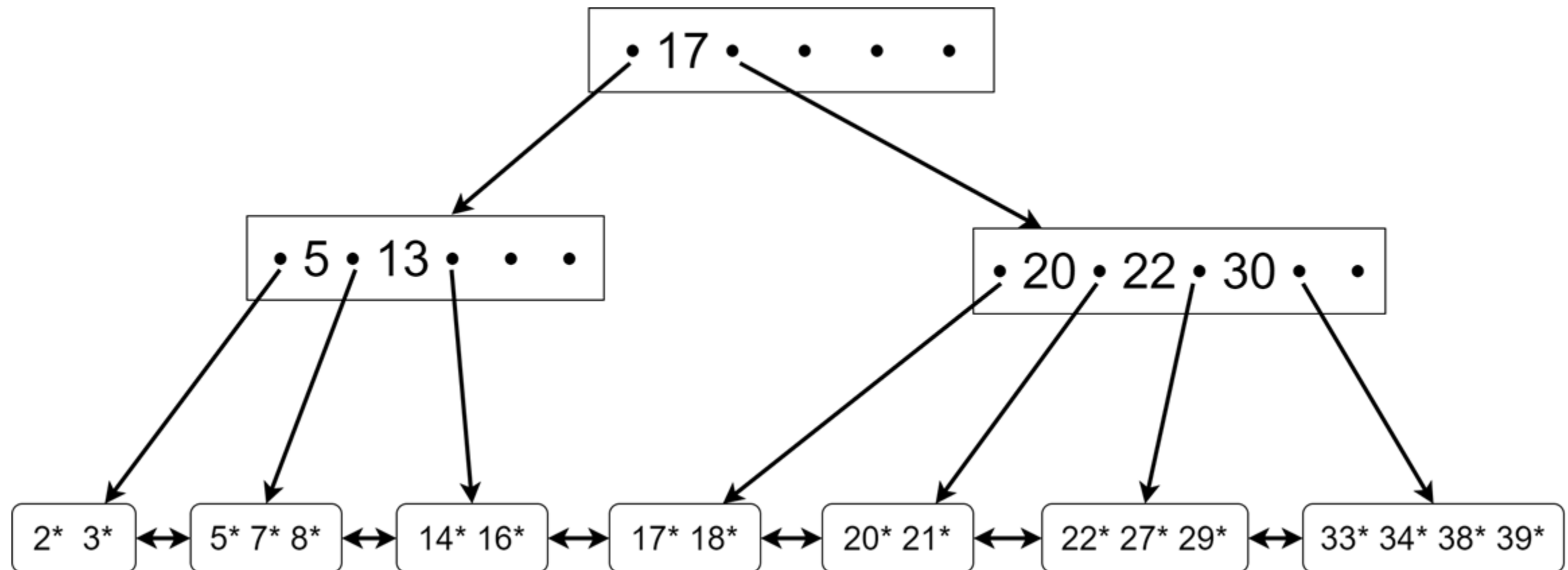
Ahora queremos hacer la consulta de todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

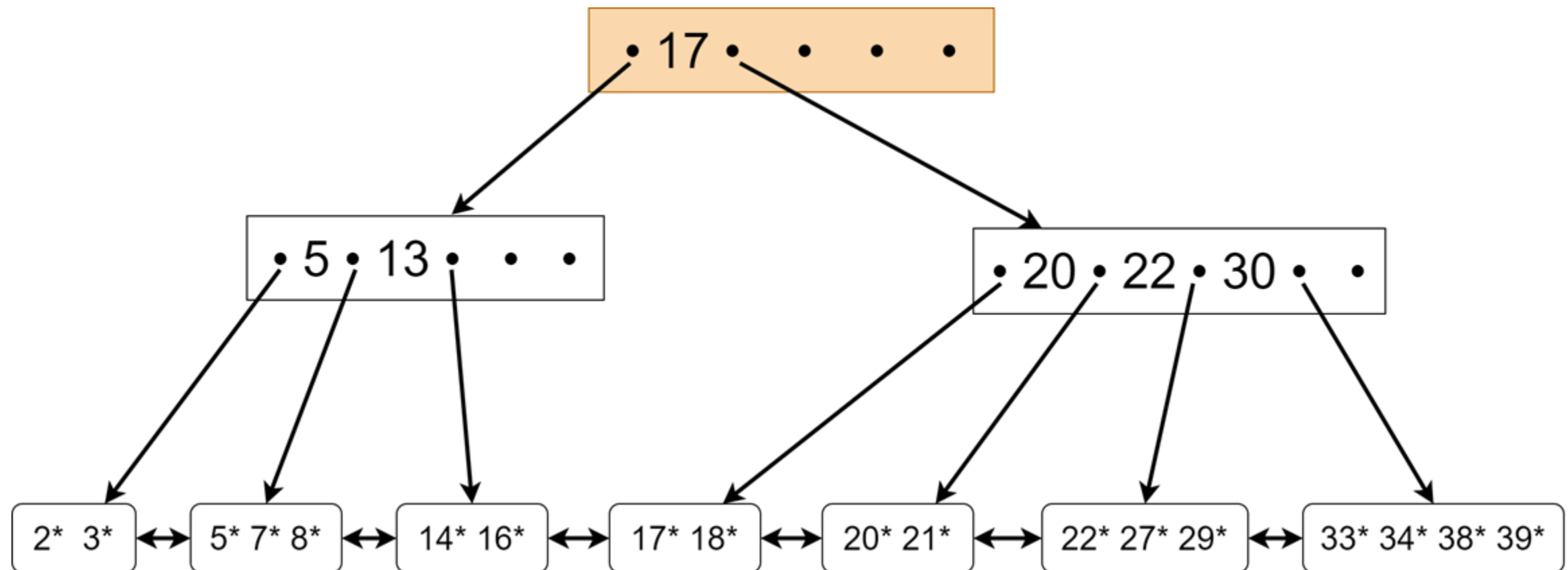
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

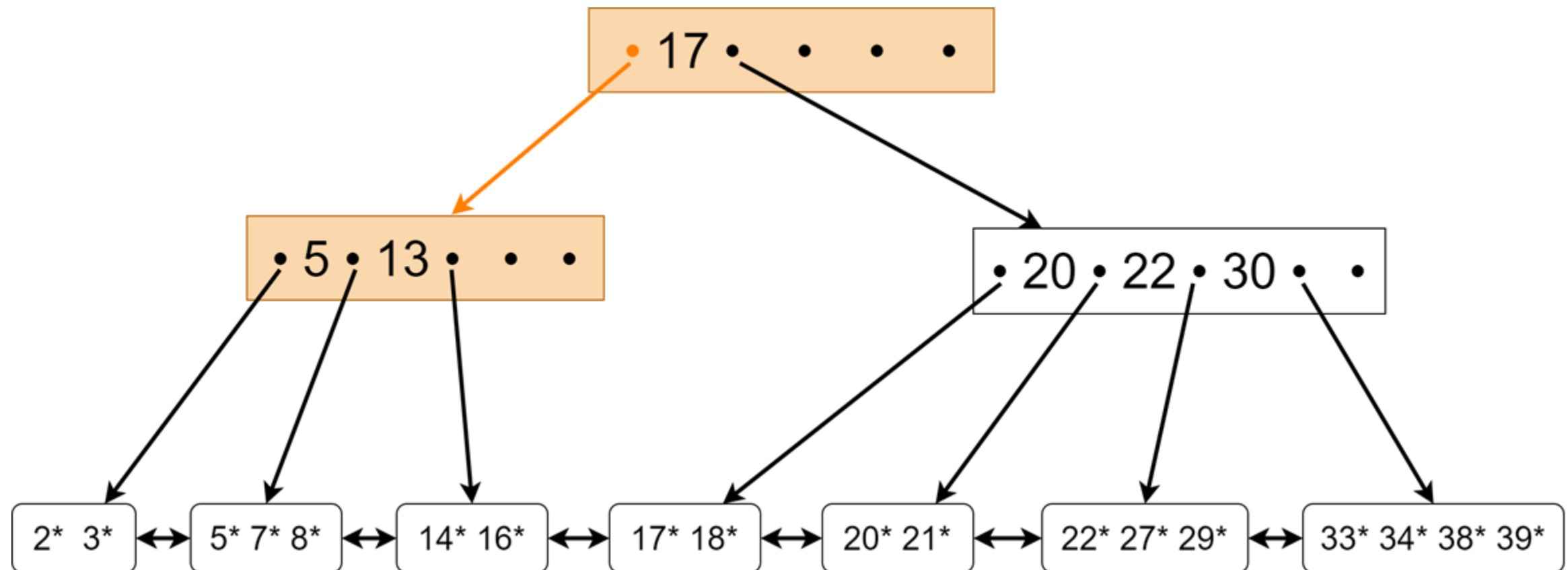
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

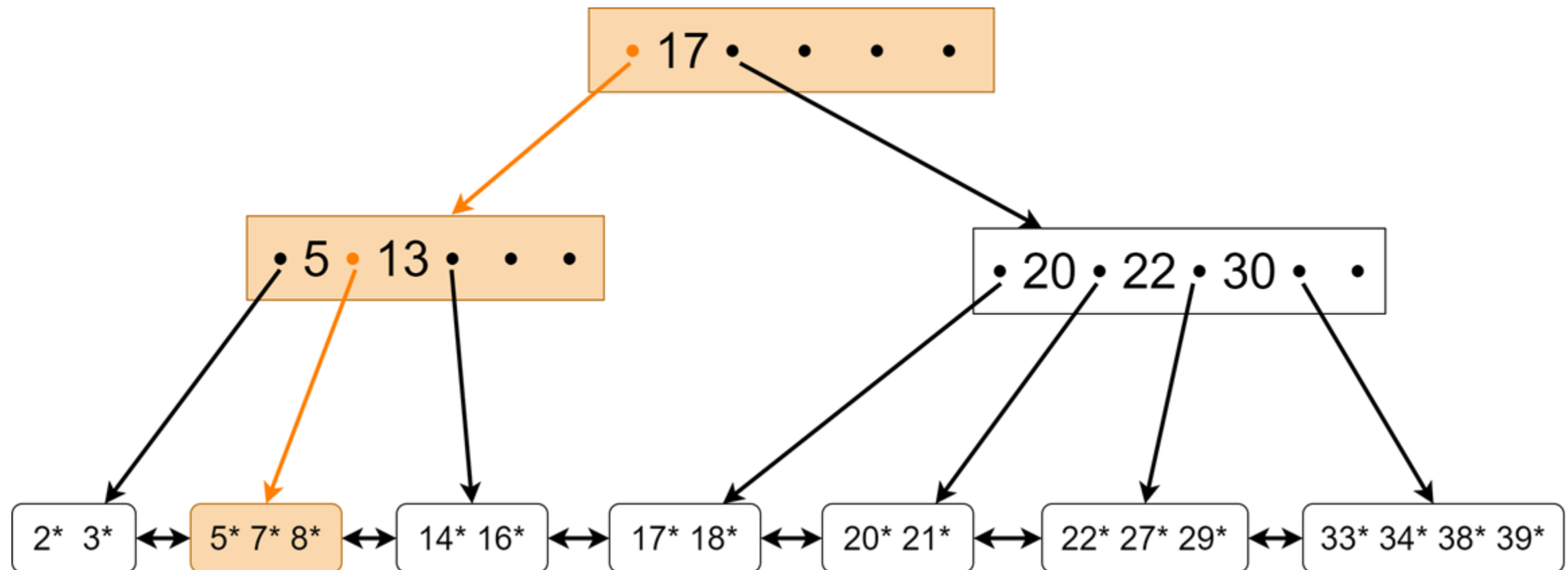
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

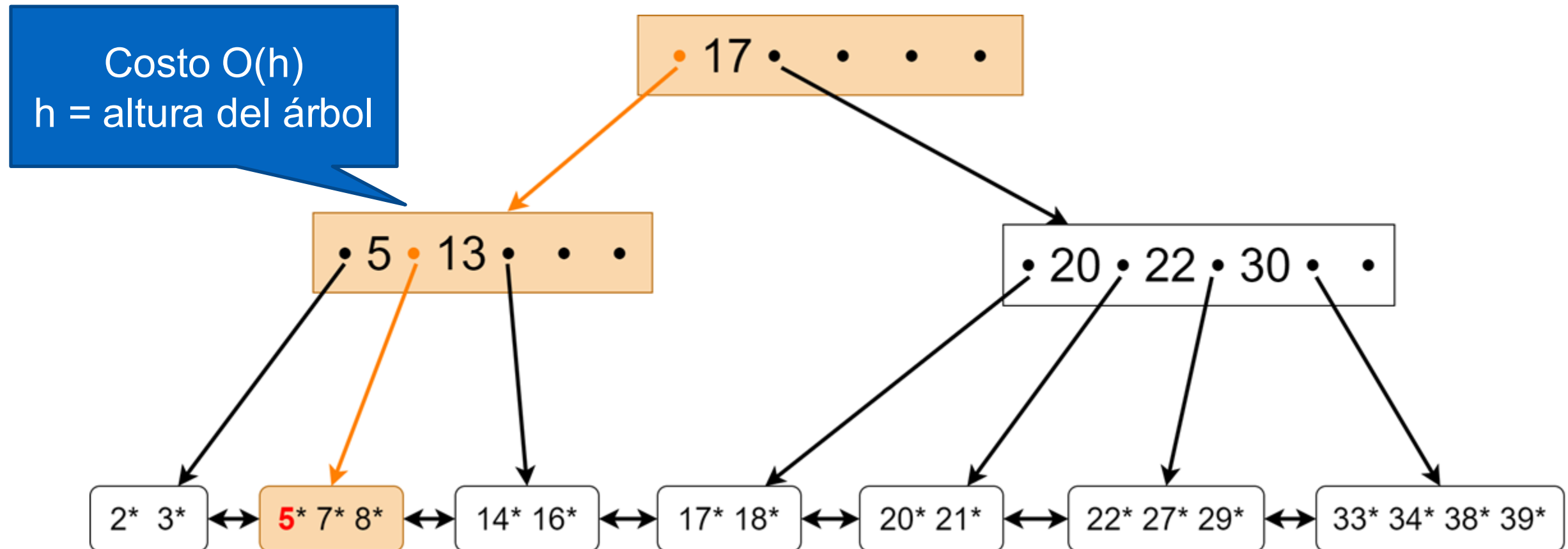
Todos los Artistas con aid entre 5 y 26



B+ Tree Index

Ejemplo: consulta de rango

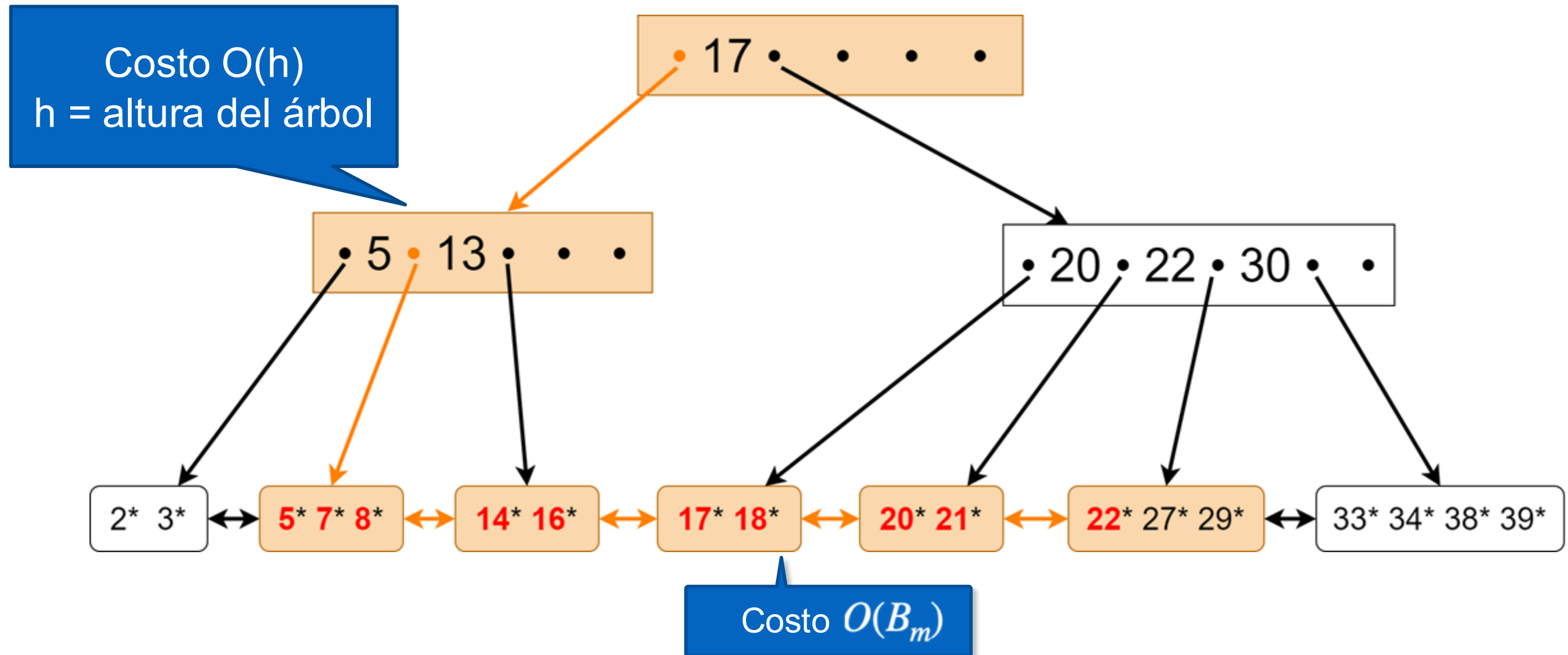
Todos los Artistas con aid entre 5 y 26



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

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
```

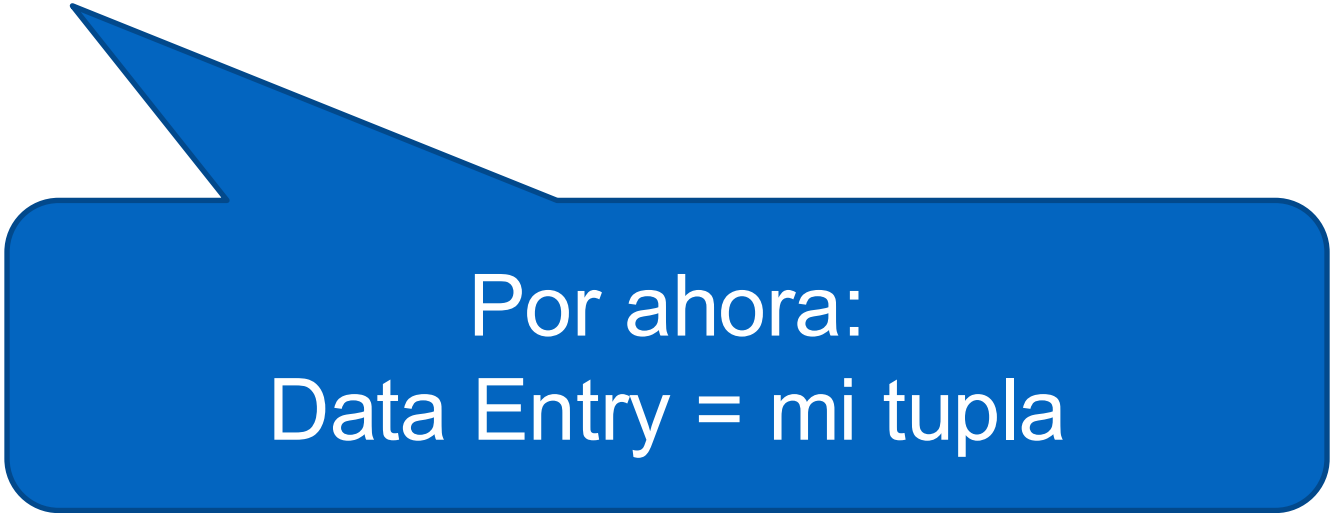
esta se resolverá con el índice, así evitaremos recorrer toda la tabla

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

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)

MÉTODO 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

MÉTODO 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

MÉTODO Hash Index

Ejemplo

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

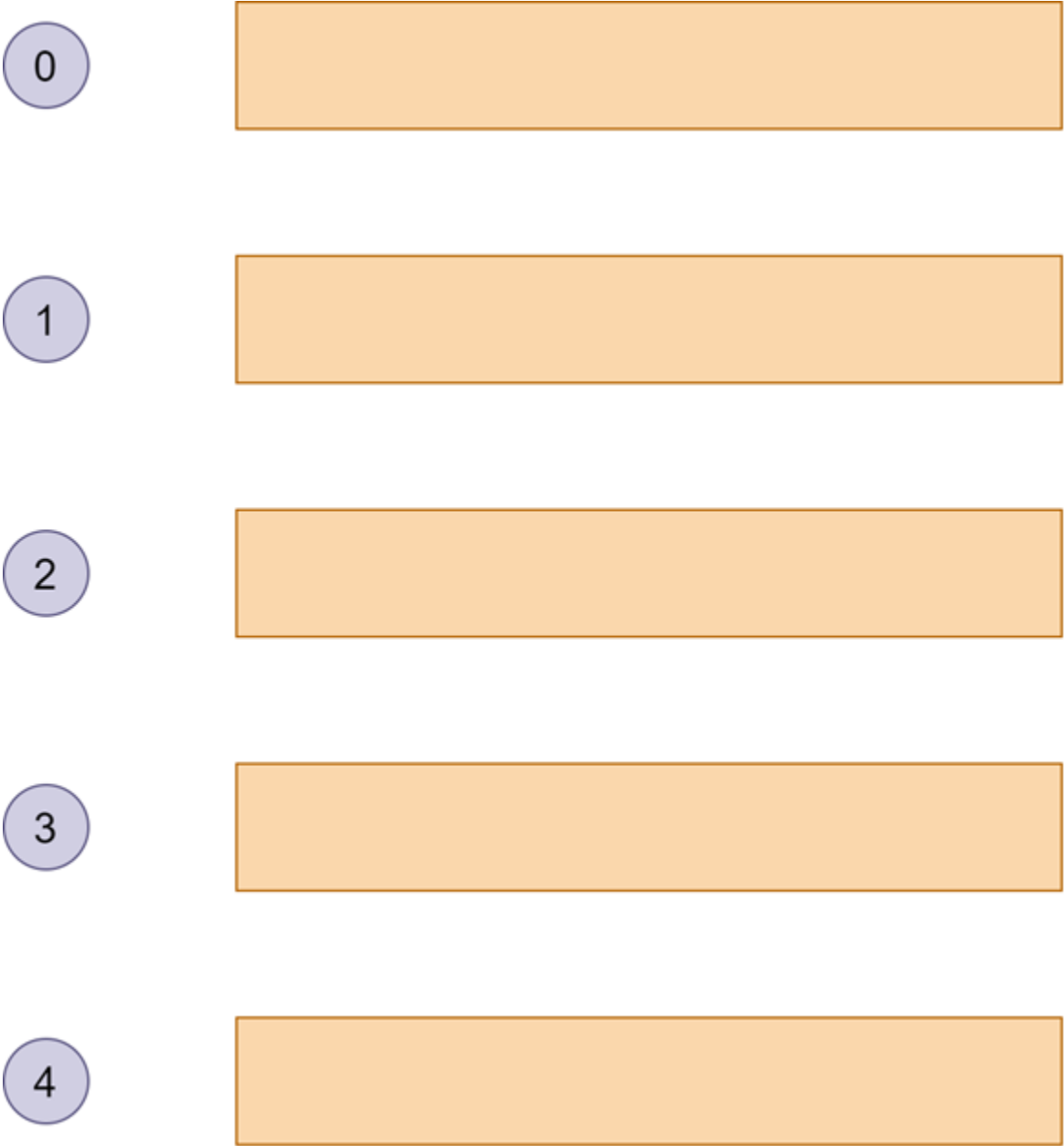
aid	nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto

```
CREATE INDEX id_hash ON Artista (aid) USING HASH
```

MÉTODO Hash Index

Ejemplo

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



MÉTODO Hash Index

Ejemplo

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

0

1

(1, Leonardo)

2

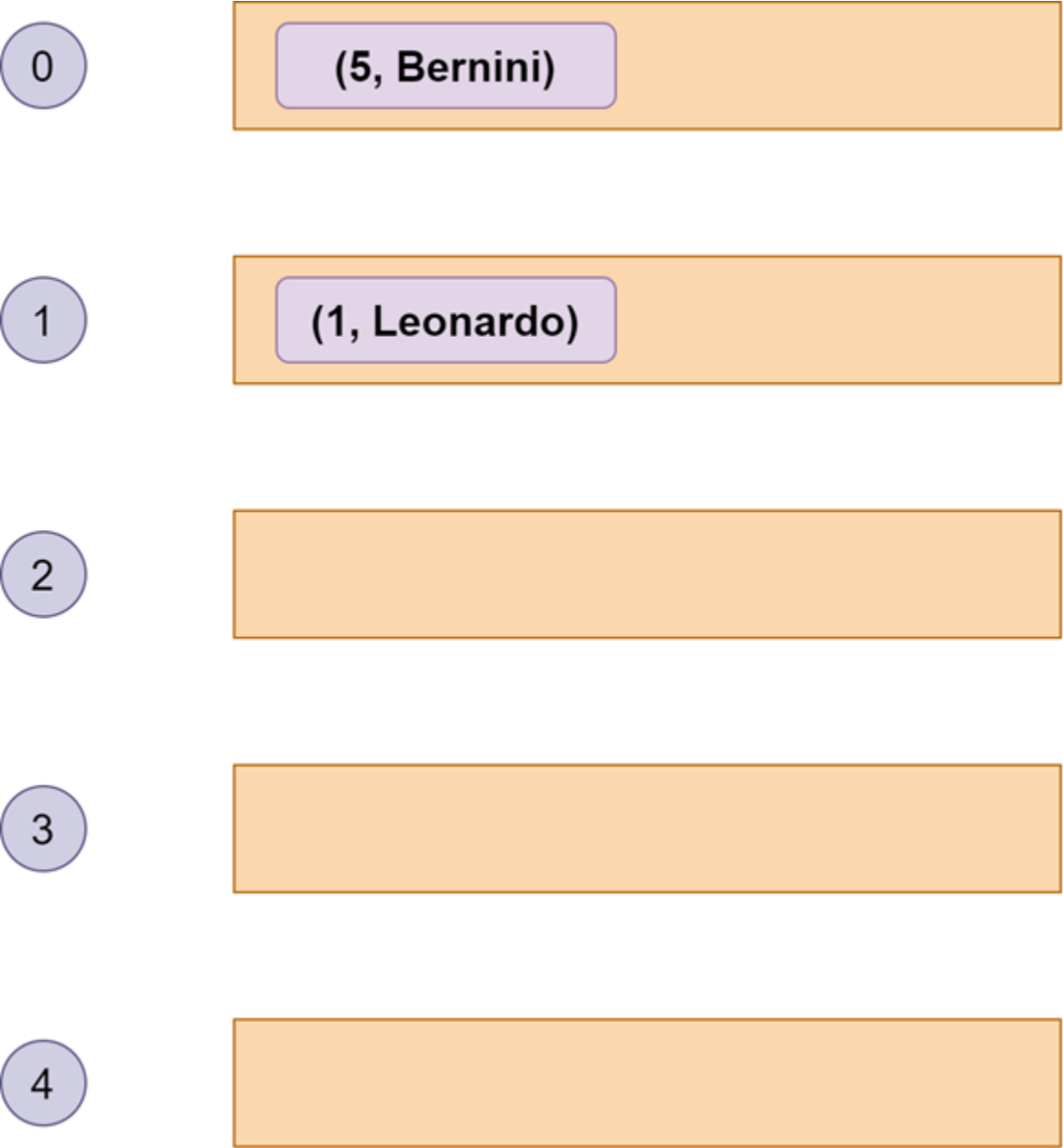
3

4

MÉTODO Hash Index

Ejemplo

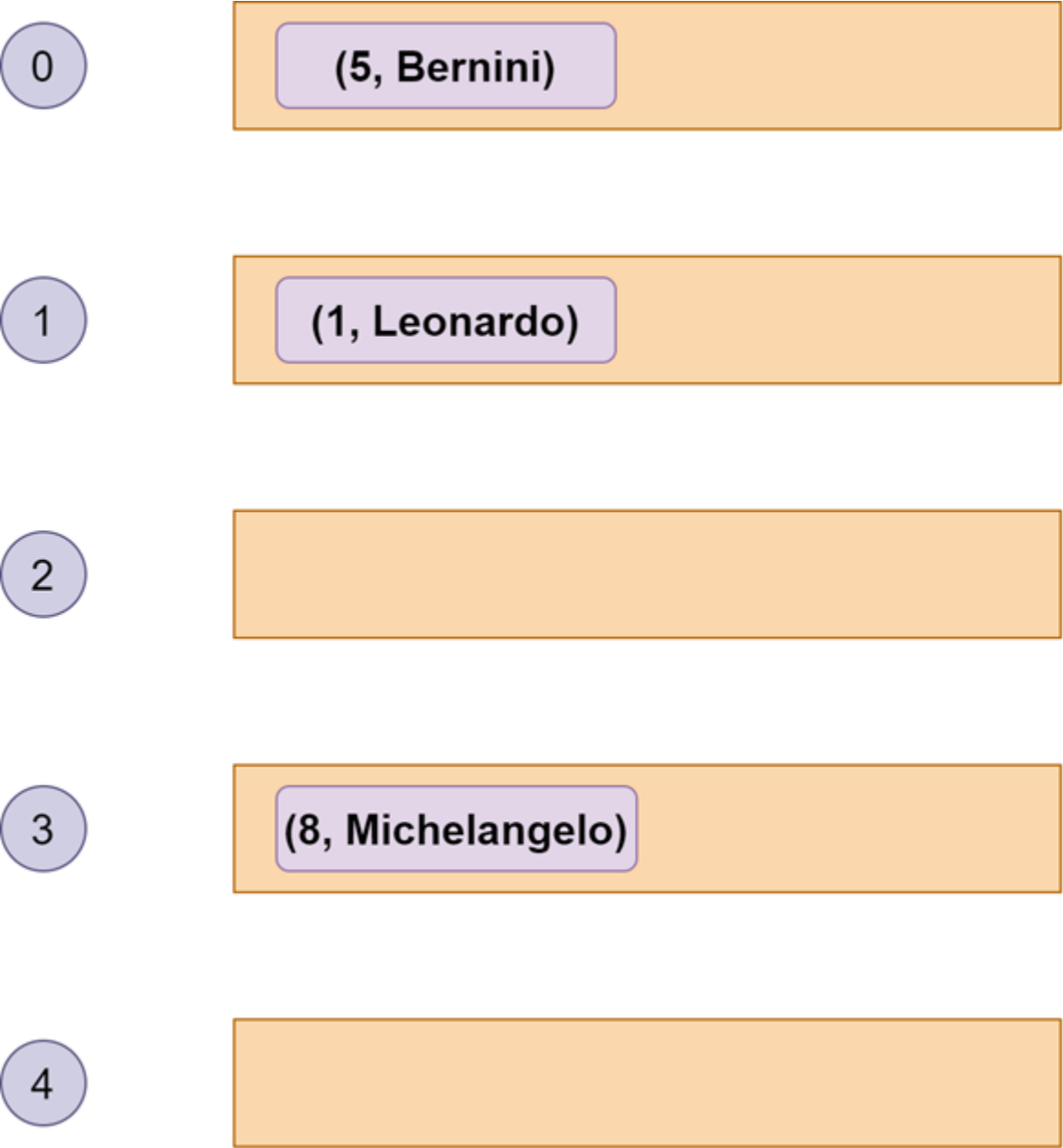
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



MÉTODO Hash Index

Ejemplo

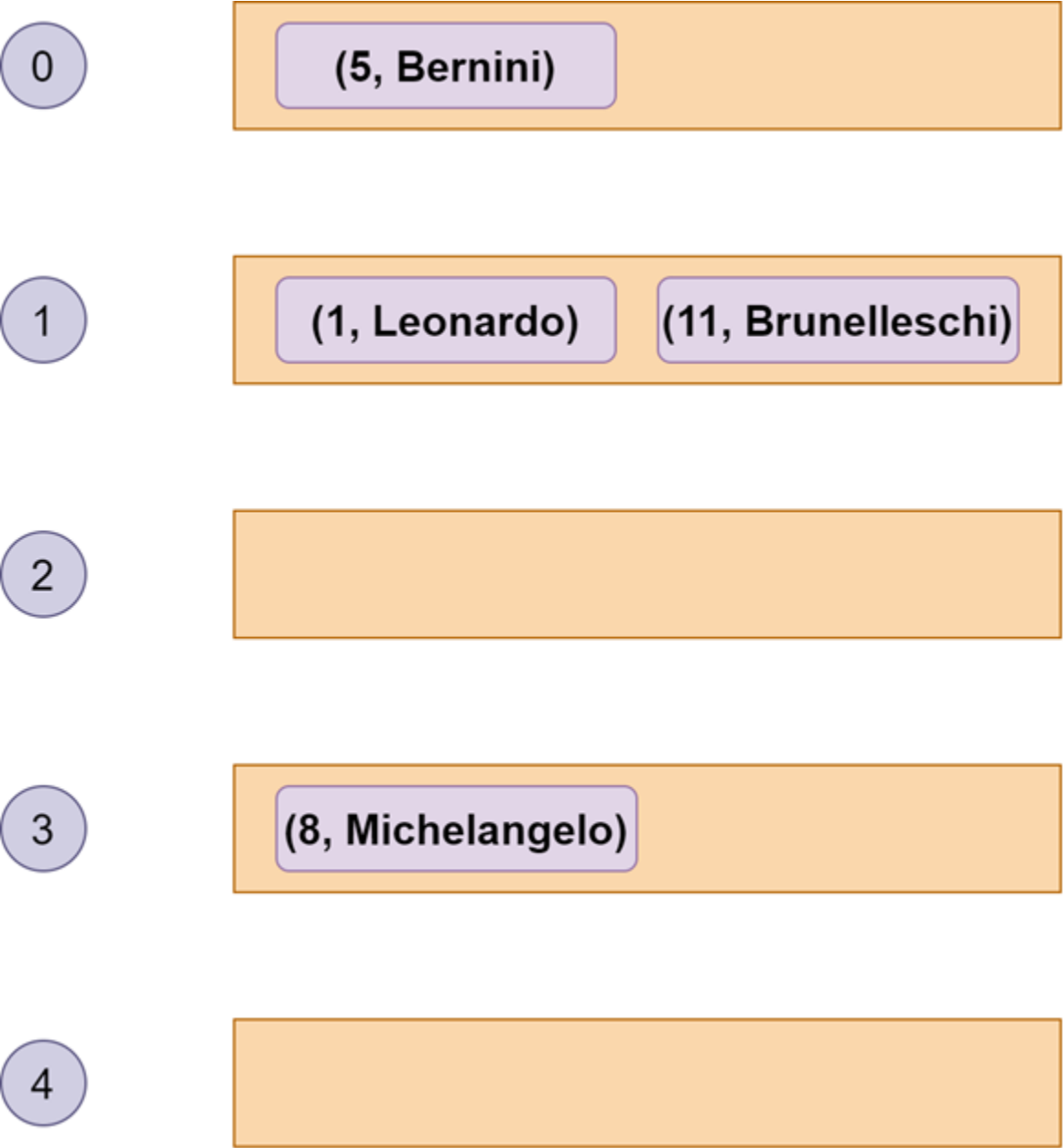
Artista.aid	Artista.nombre
1	Leonardo
5	Bernini
8	Michelangelo
11	Brunelleschi
12	Botticelli
16	Giotto



MÉTODO Hash Index

Ejemplo

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



MÉTODO Hash Index

Ejemplo

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

0

(5, Bernini)

Overflow

1

(1, Leonardo)

(11, Brunelleschi)

2

(12, Botticelli)

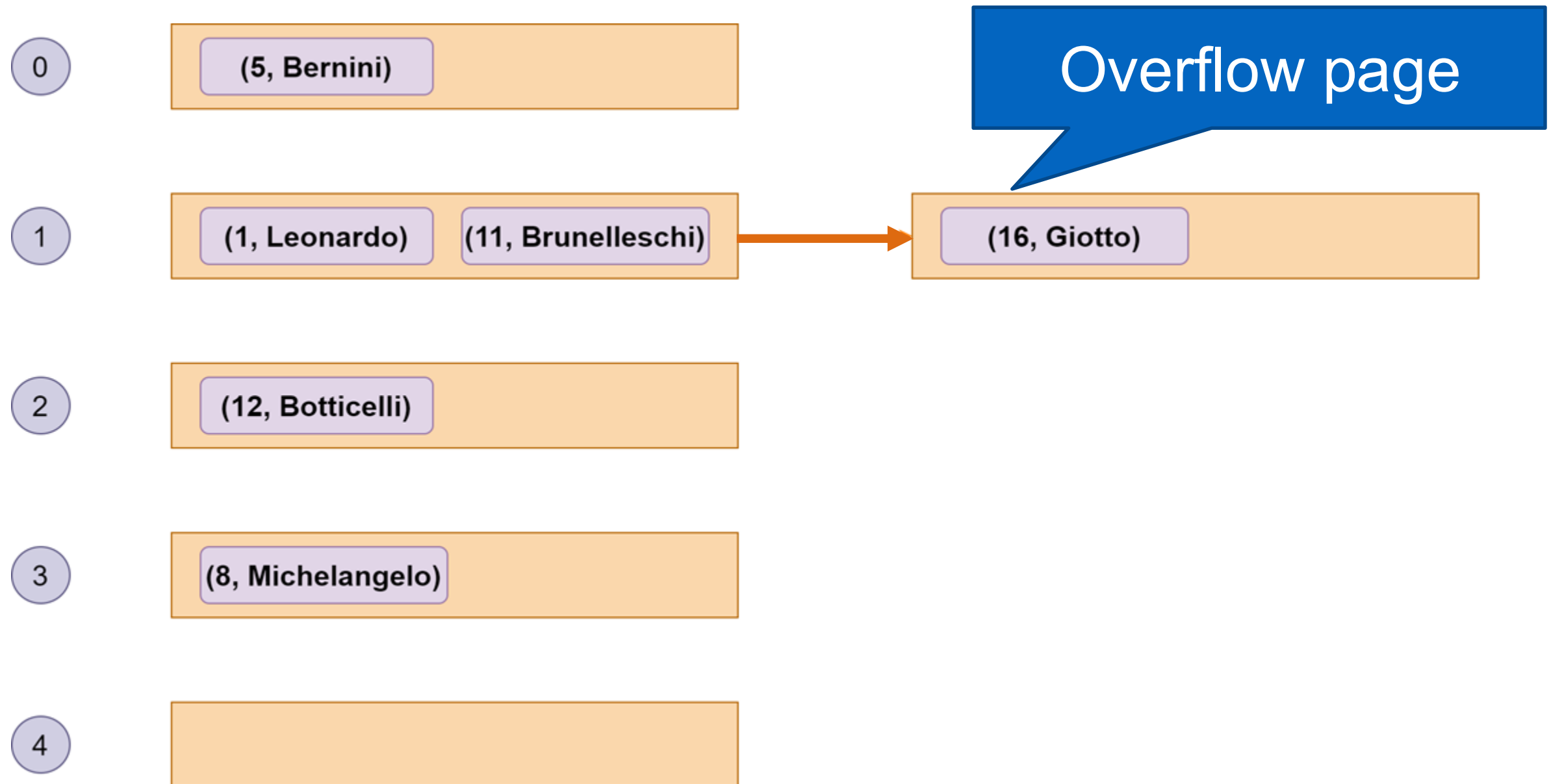
3

(8, Michelangelo)

4

MÉTODO Hash Index

Ejemplo



Clustered y Unclustered Index

Data entry:
La tupla misma

Hablaremos de un índice **Clustered** si al preguntarle por un valor me retorna una tupla. Las llaves están ordenados físicamente.

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

Considere Empleados(id, nombre, edad)

Un índice primario (clustered) en general es el deafault sobre la primary key

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

Sólo puede haber UN índice Clustered por tabla

Costo de consulta	EEDD	Índices	Hash/B+	Clustered/Unclustered
-------------------	------	---------	---------	-----------------------

Buscando una tupla con
distintos tipos de índices

Hash Index clustered

$R(\underline{a}, b, c)$... índice sobre a

Datos en la hoja

$a = 5$



$5 \% 4 = 1$



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

Hash Index clustered

$R(\underline{a}, b, c)$... índice sobre a

$a = 5$



$5 \% 4 = 1$



Bucket	
0	(0,p2), (4,p1), (8,p3)
1	(1,2,2), (5,1,1) página del disco
2	(2,3,3)
3	(3,1,2), (7,8,9)

Hash Index clustered

$R(\underline{a}, b, c)$... índice sobre a

$a = 5$

$5 \% 4 = 1$

Bucket	
0	(0,p2), (4,p1), (8,p3)
1	(1,2,2), (5,1,1) página del disco
2	(2,3,3)
3	(3,1,2), (7,8,9)

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

Hash Index unclustered

$R(\underline{a}, b, c)$... índice sobre a

$a = 5$



$5 \% 4 = 1$



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

Datos en el disco:

p1

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

p2

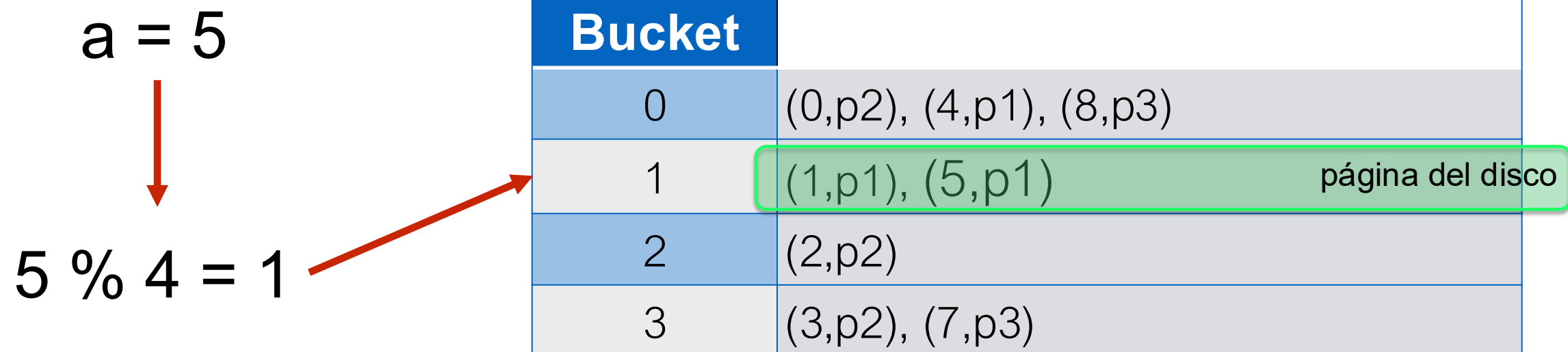
(0,2,3), (3,1,2), (2,3,3)

p3

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

Hash Index unclustered

$R(\underline{a}, b, c)$... índice sobre a



Datos en el disco:

p1

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

p2

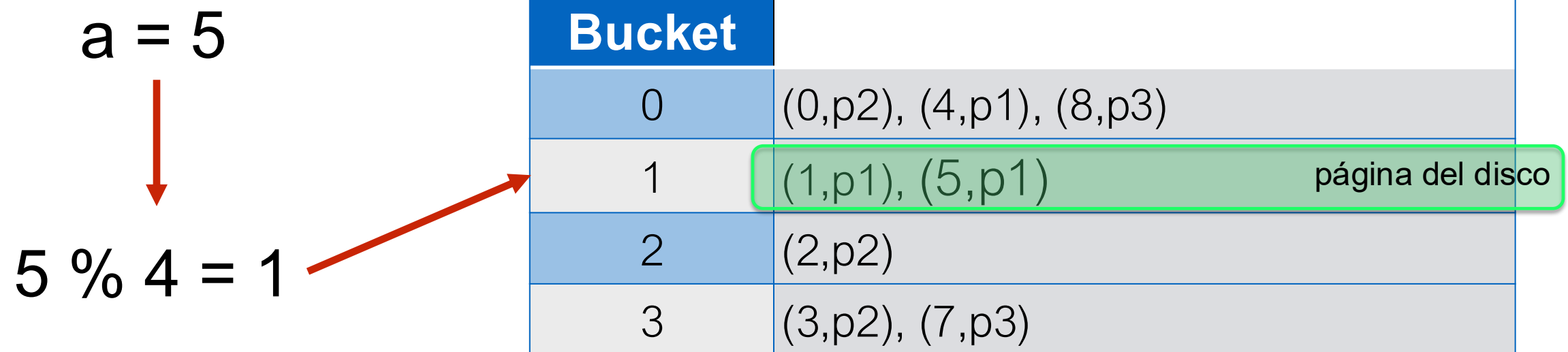
(0,2,3), (3,1,2), (2,3,3)

p3

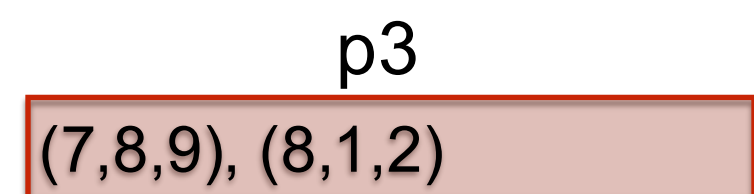
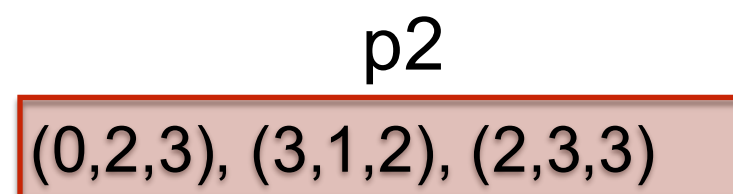
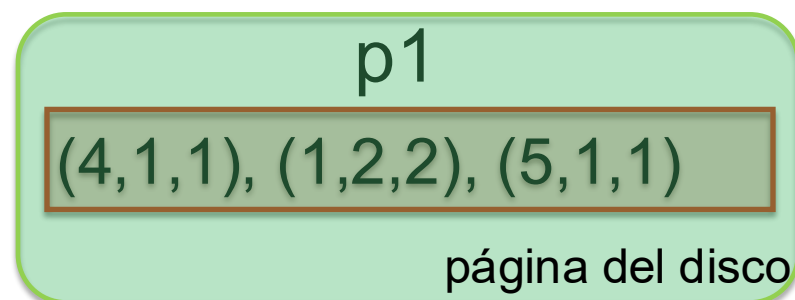
(7,8,9), (8,1,2)

Hash Index unclustered

$R(\underline{a}, b, c)$... índice sobre a

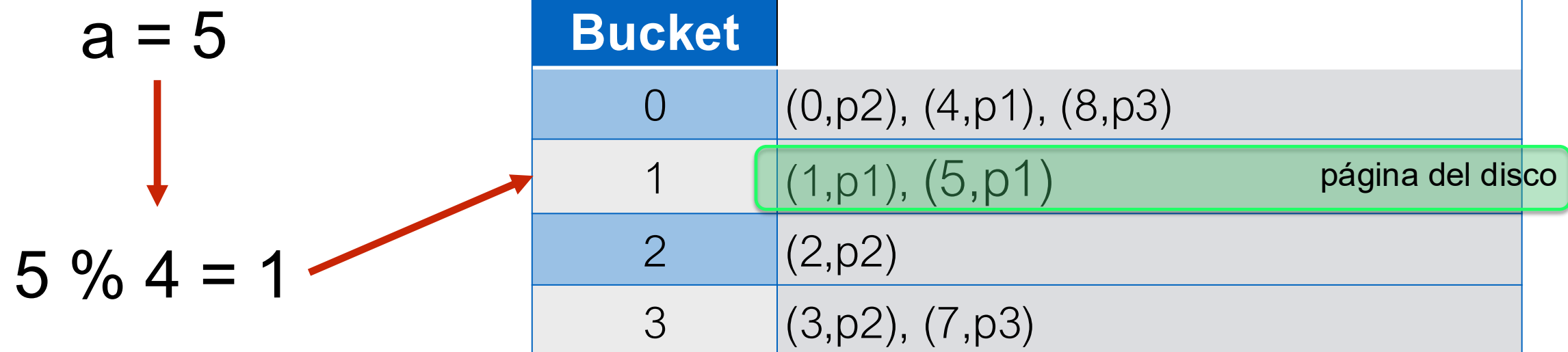


Datos en el disco:

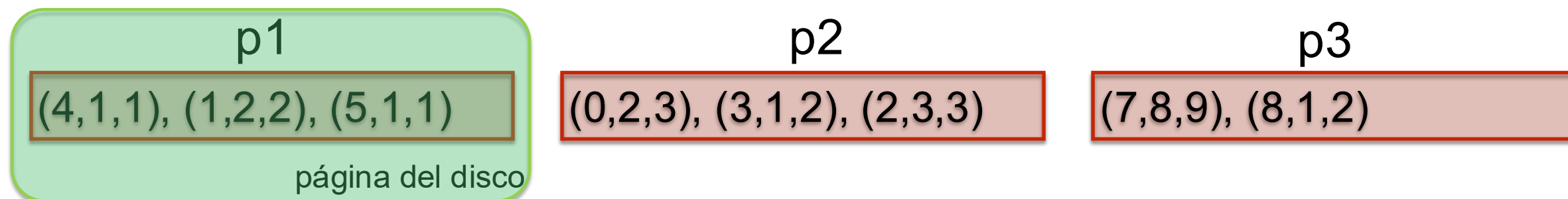


Hash Index unclustered

$R(\underline{a}, b, c)$... índice sobre a



Datos en el disco:



$\#(I/O) = 2$

Hash Index unclustered

$R(\underline{a}, b, c)$... índice sobre a

Considerar: página del índice vs página con tuplas

P punteros/página

$(1, p1), (2, p7), \dots$

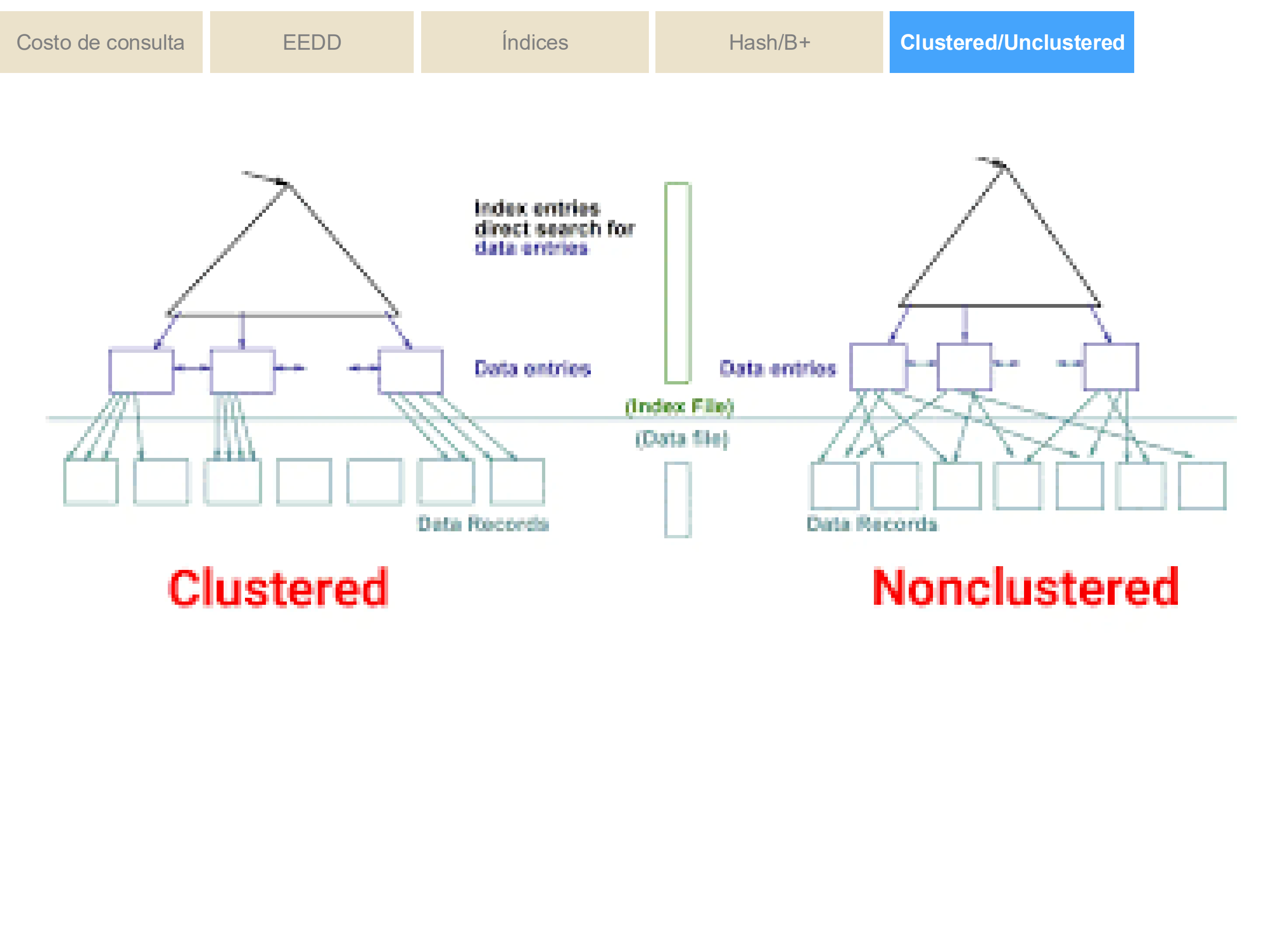
B tuplas/página

$(1, 2, 3), (7, 3, 1), \dots$

puntero $p1$... un long/8 bytes

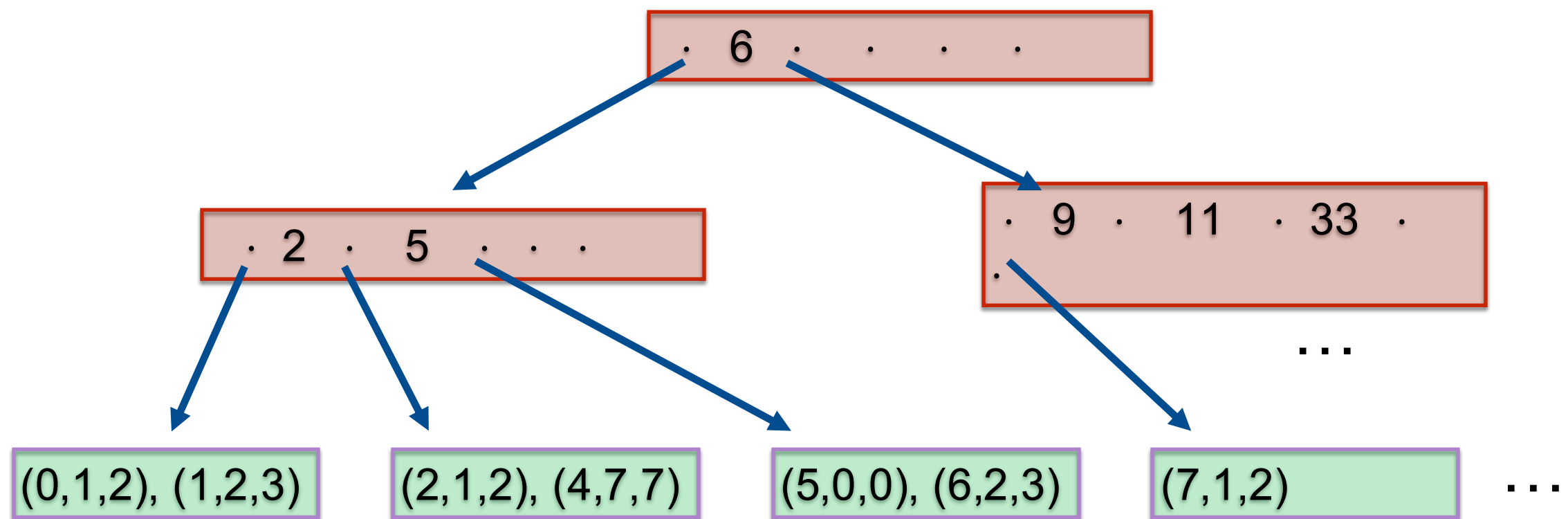
tupla ... puede ser 20 longs/160 bytes

¿Dónde cabe más? ($P > B$)



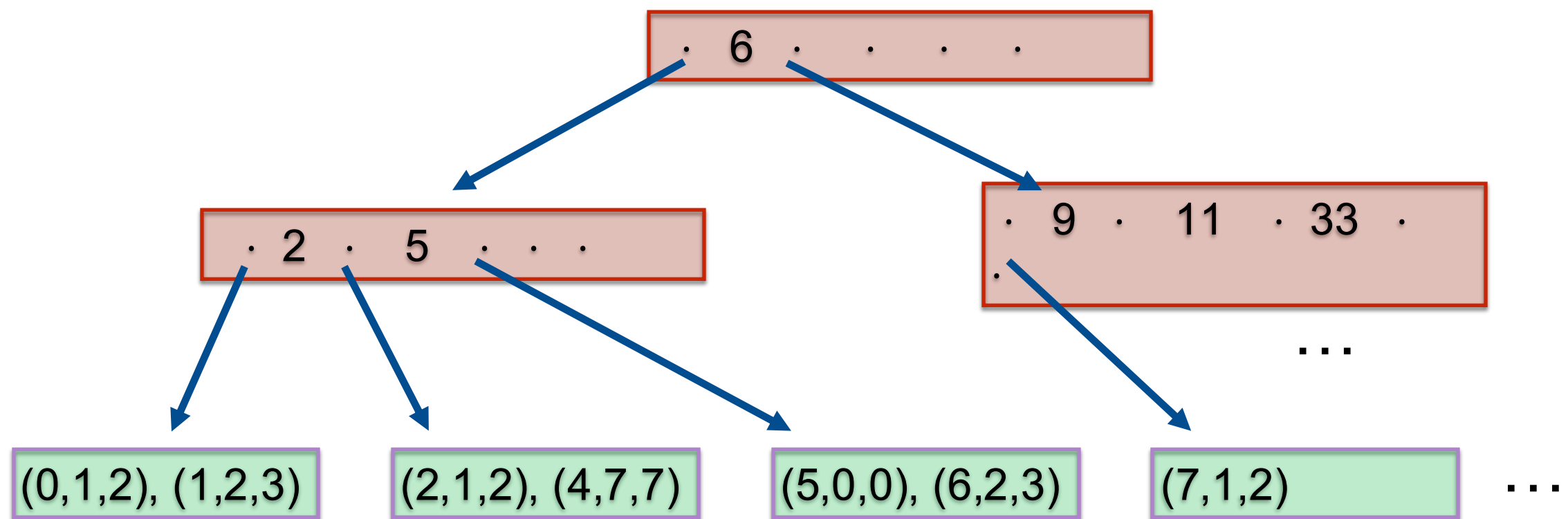
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a



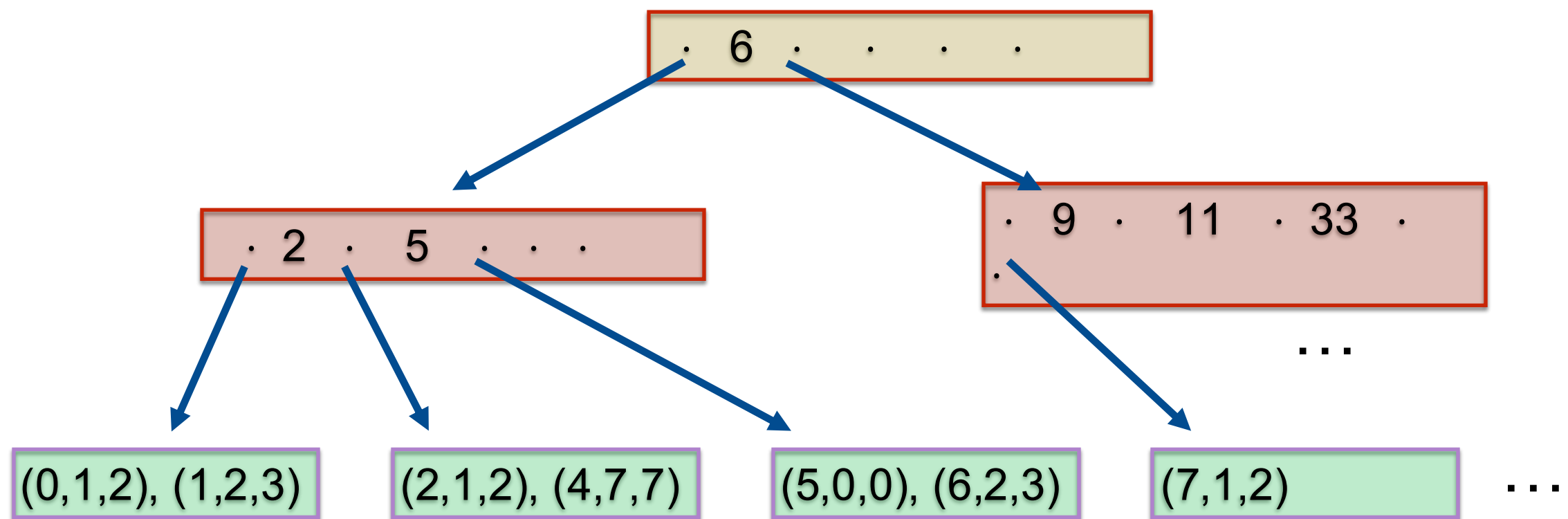
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



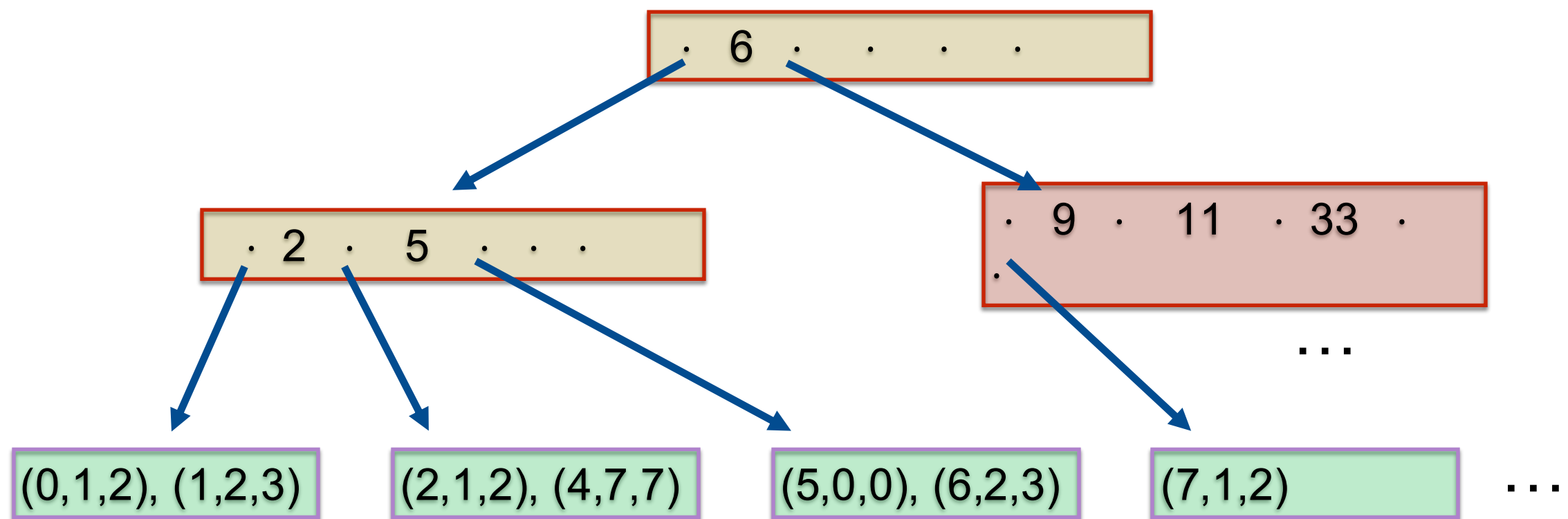
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



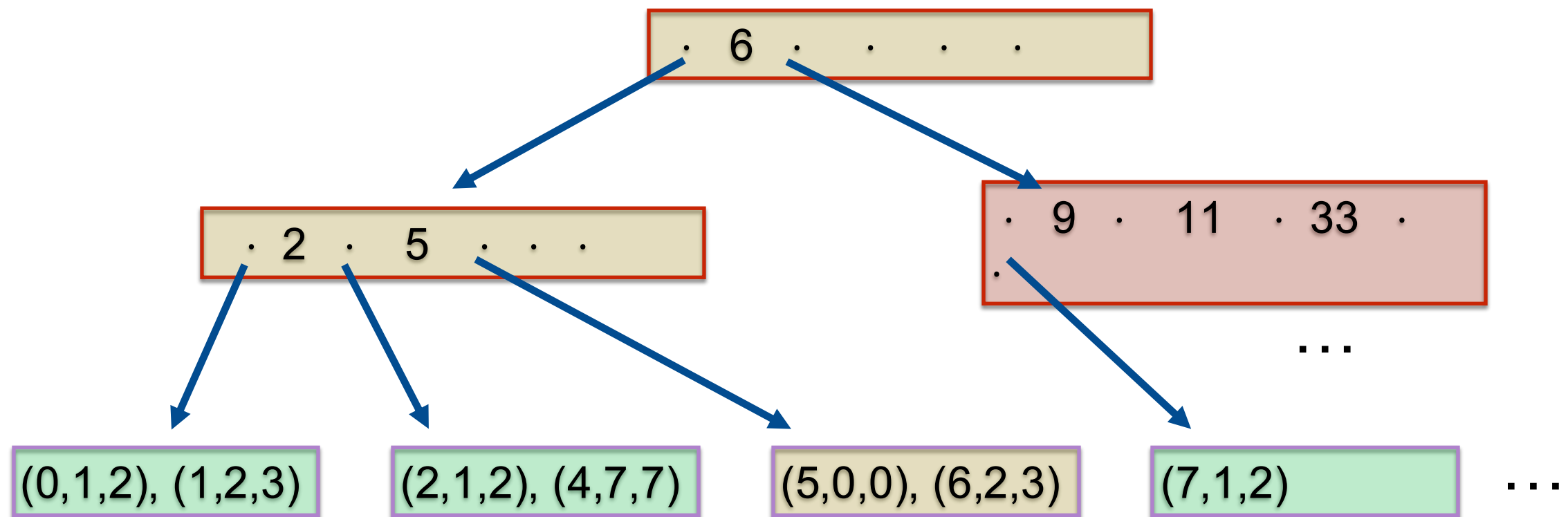
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



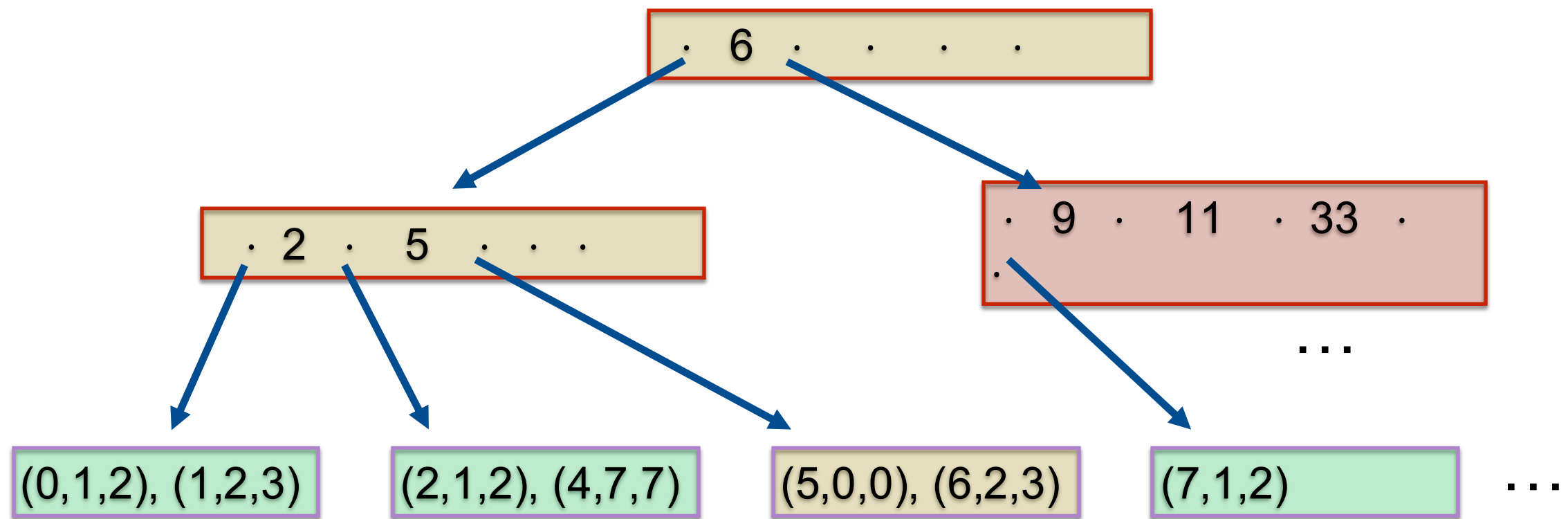
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



B+ tree clustered

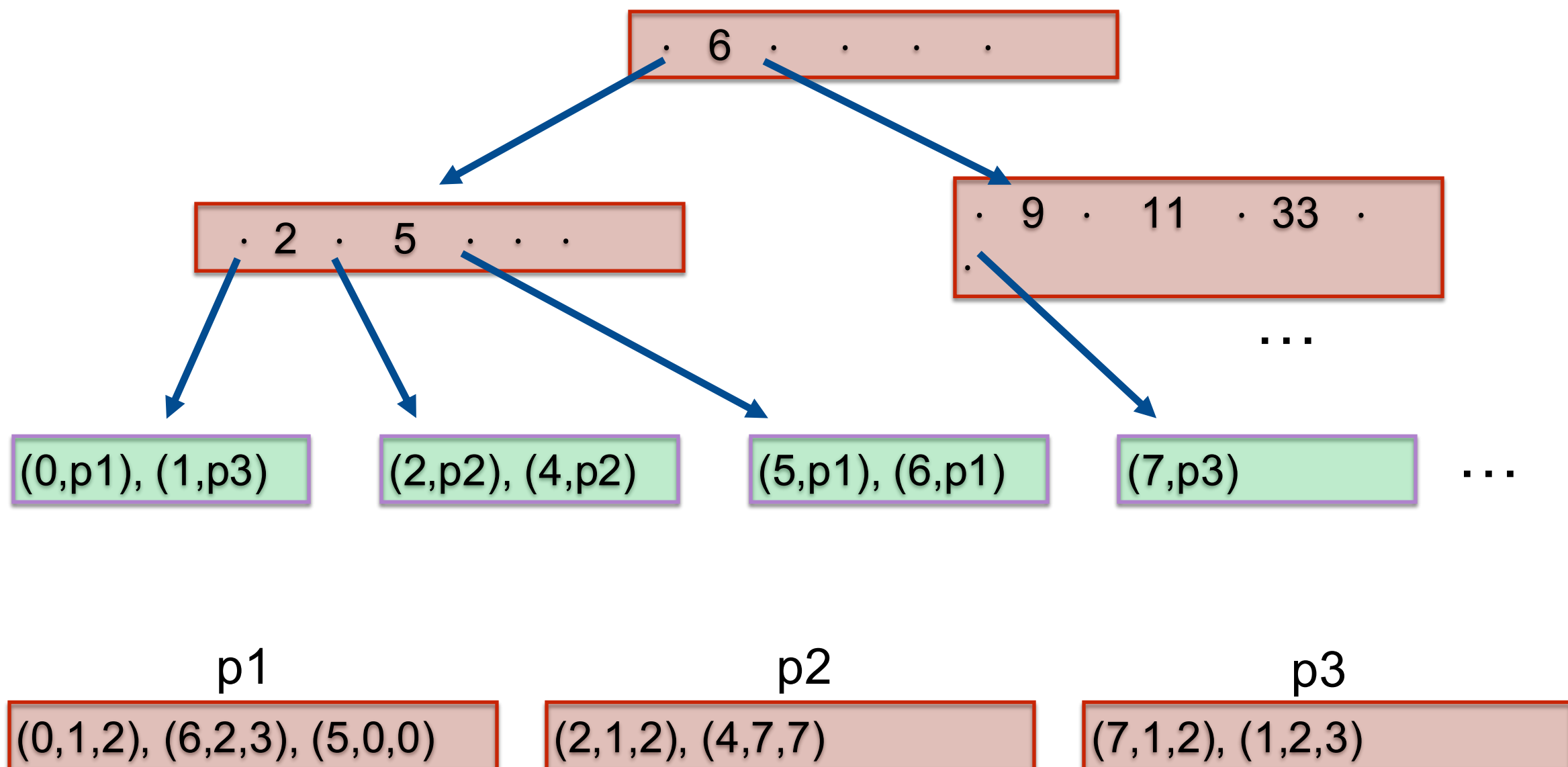
$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



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

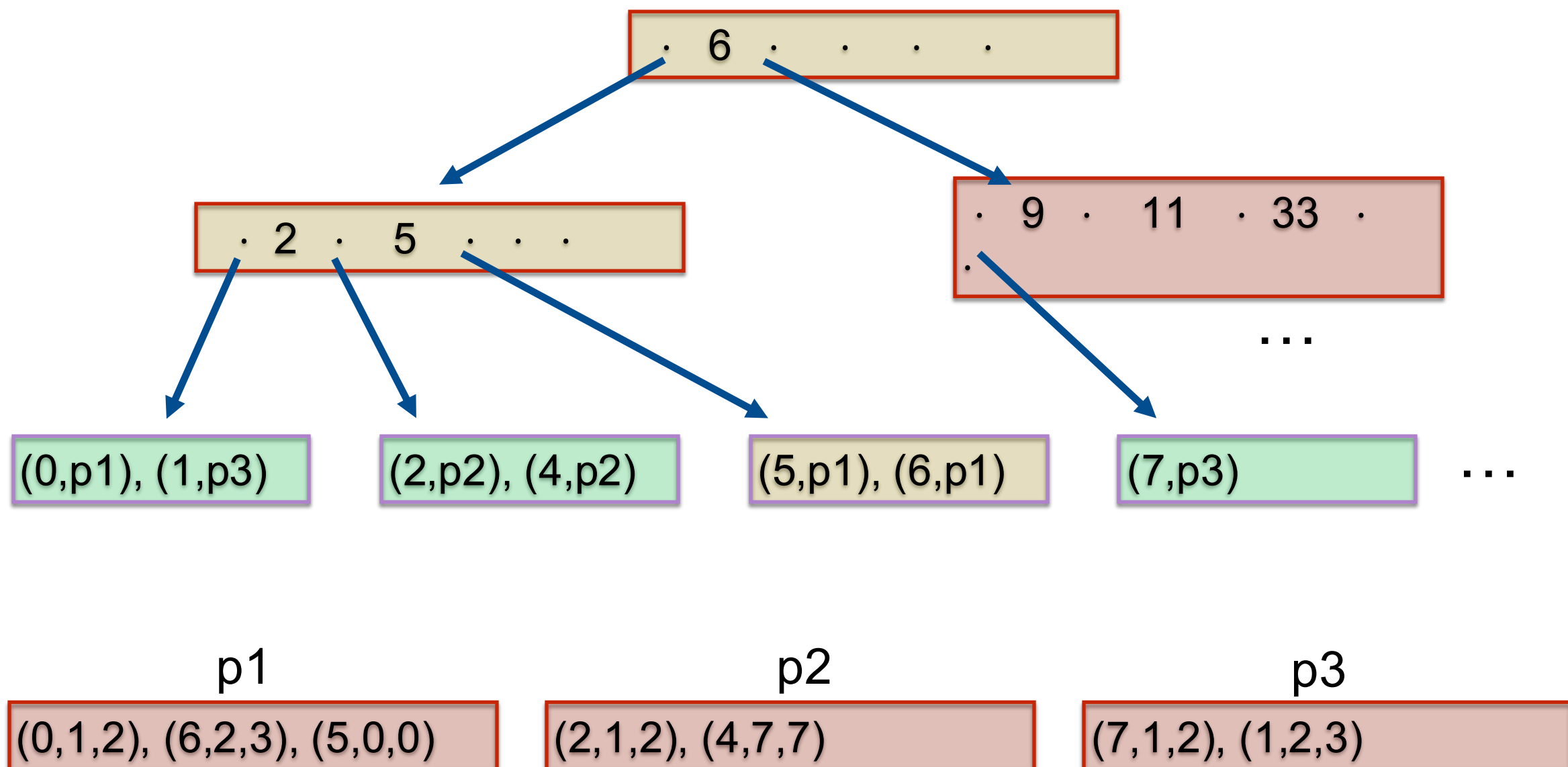
B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a



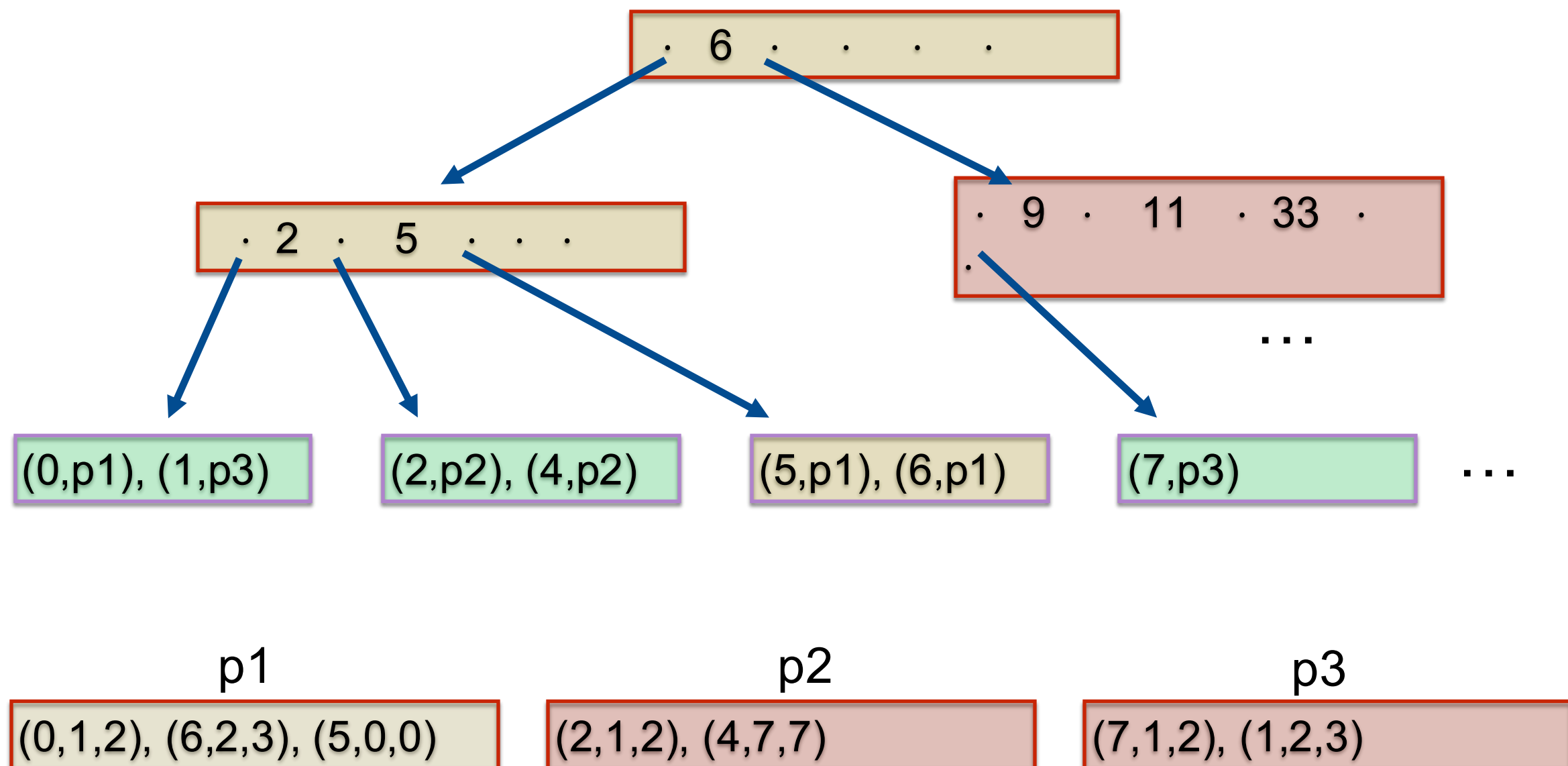
B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



B+ tree unclustered

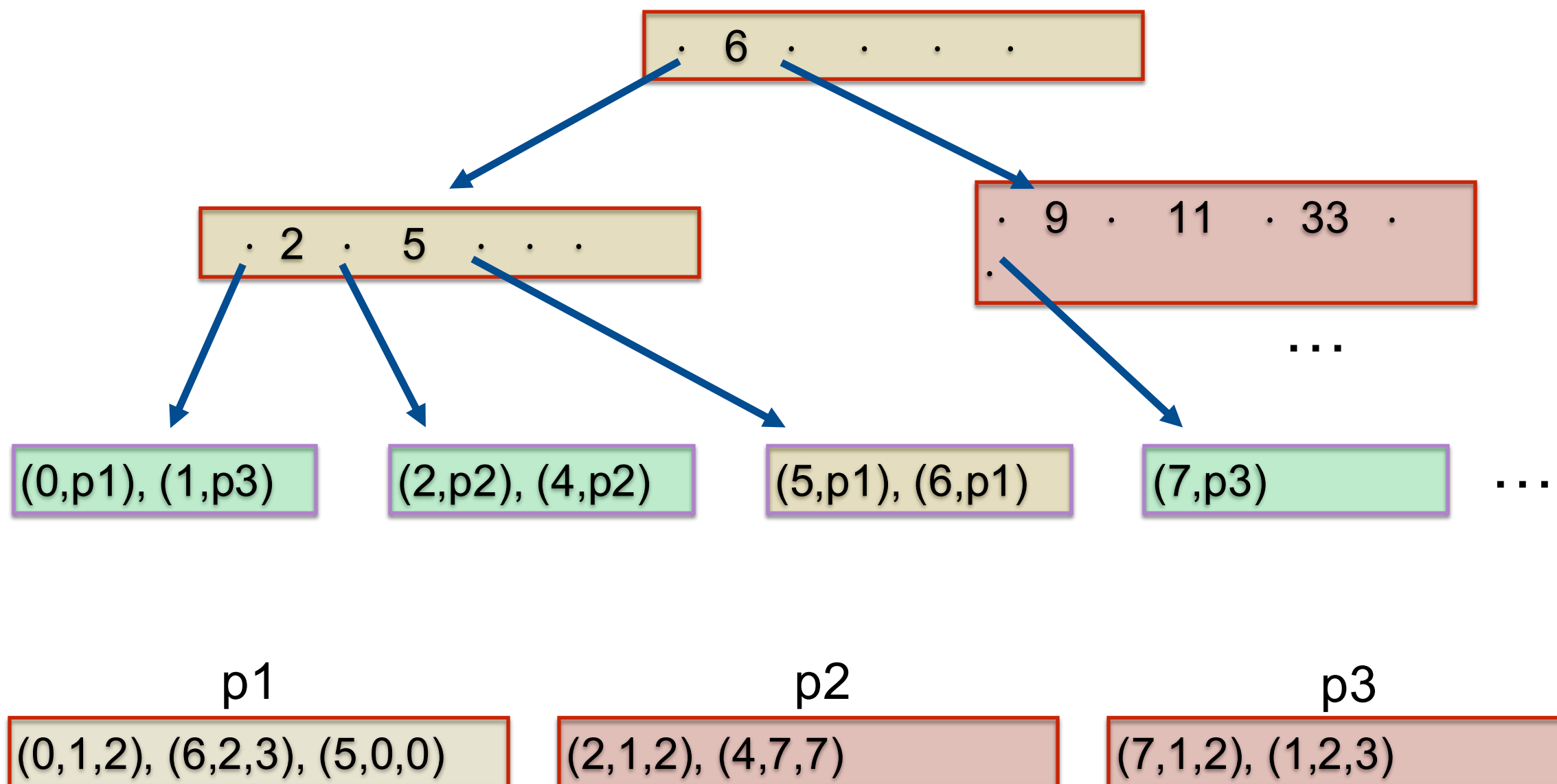
$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$



B+ tree unclustered

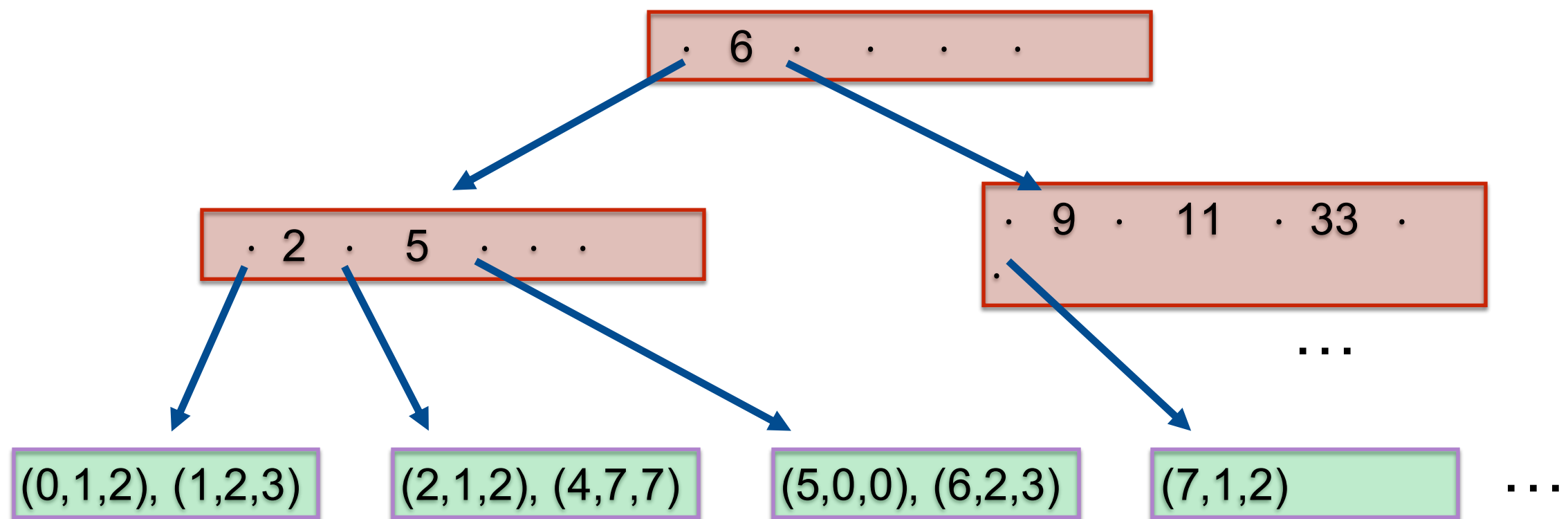
$R(\underline{a}, b, c)$... índice sobre a ... $a = 5$

Costo $h+1$



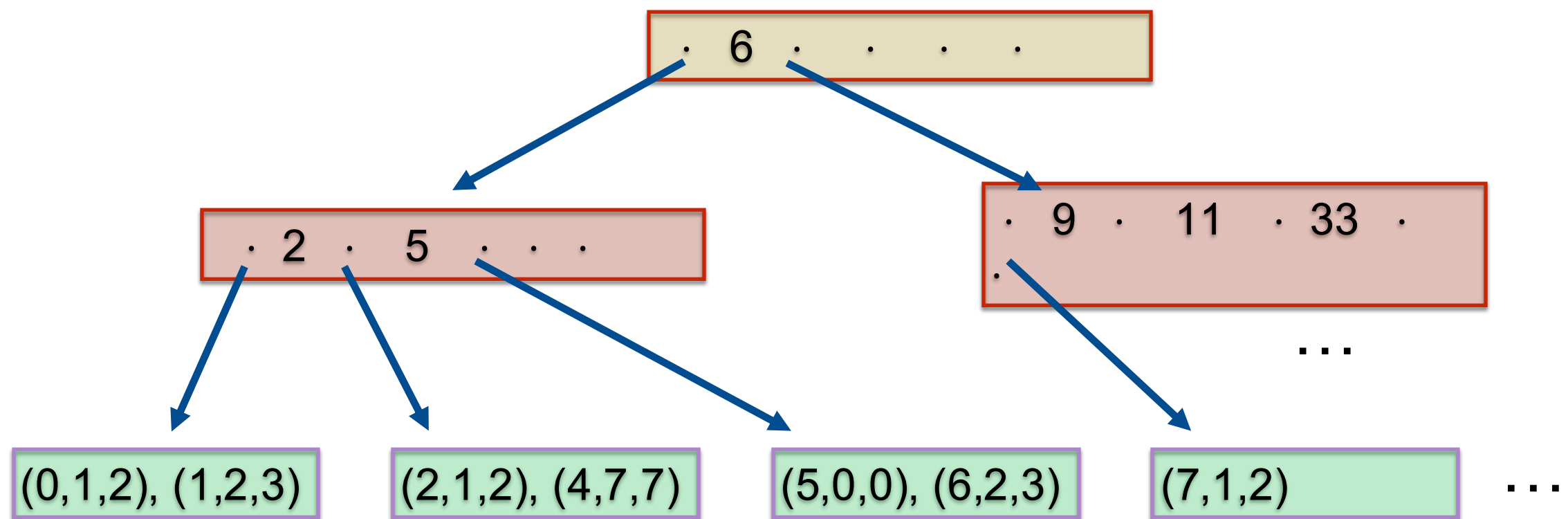
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



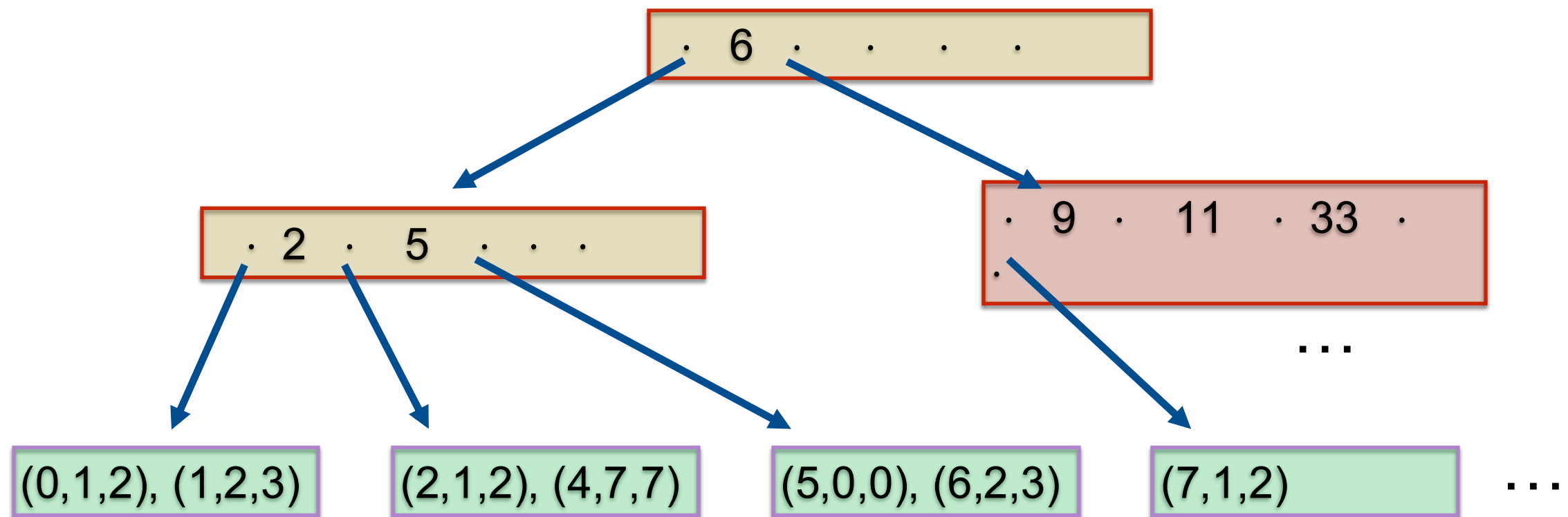
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



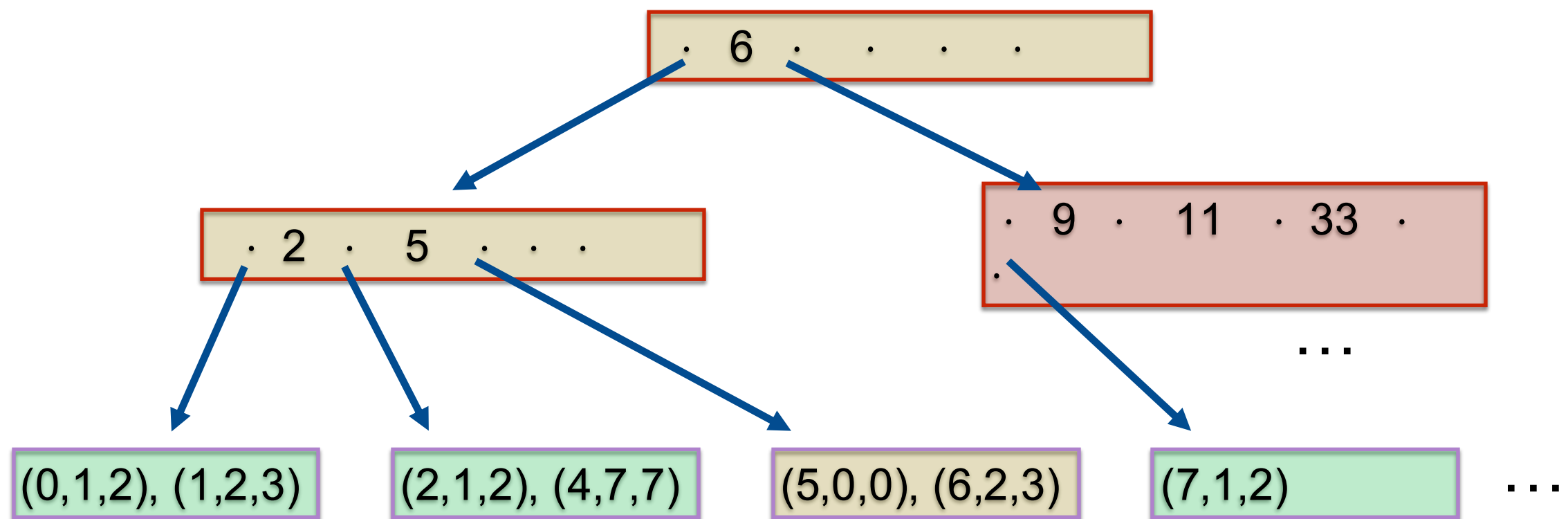
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



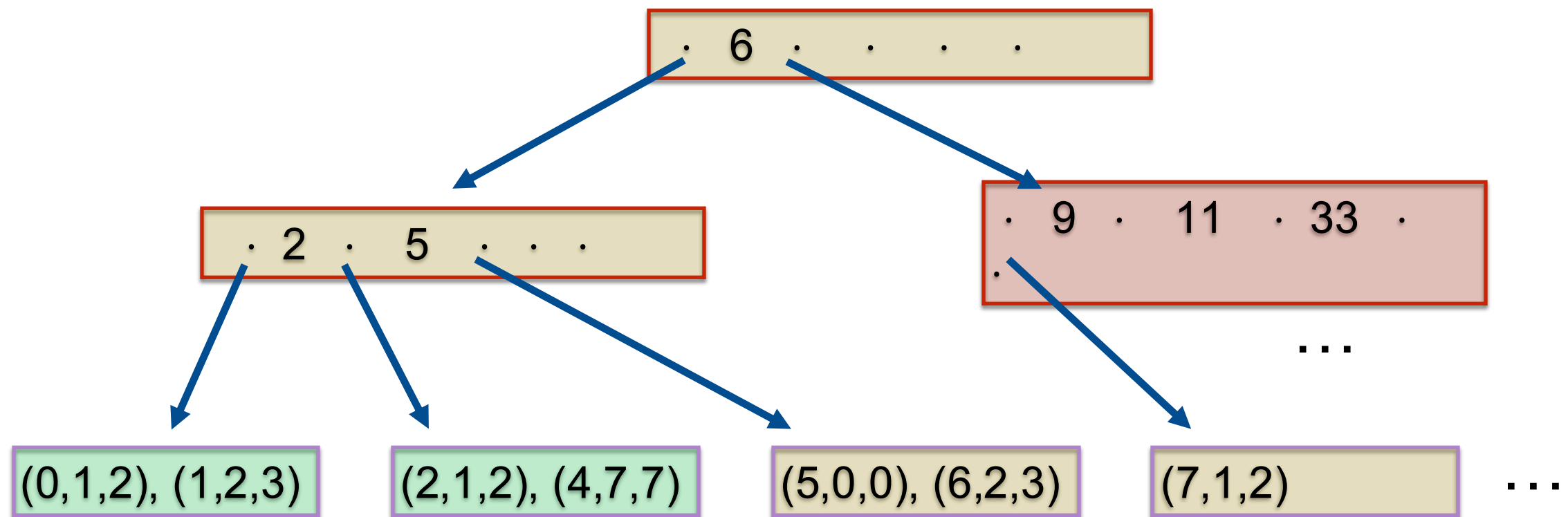
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



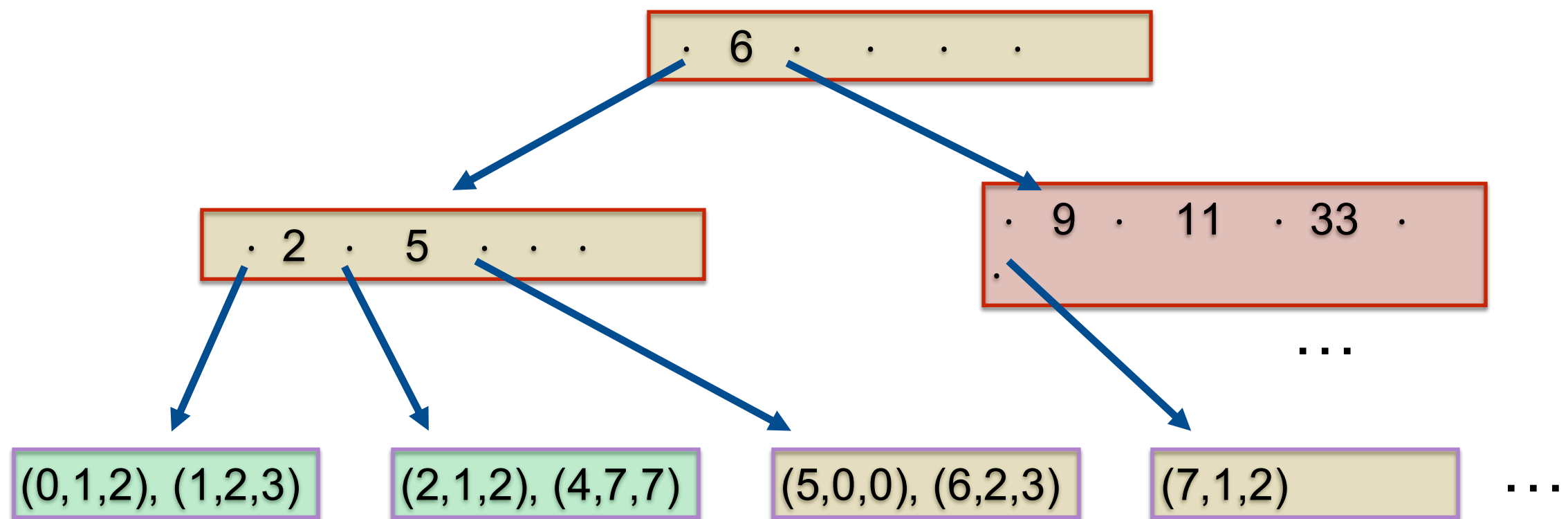
B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



B+ tree clustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$

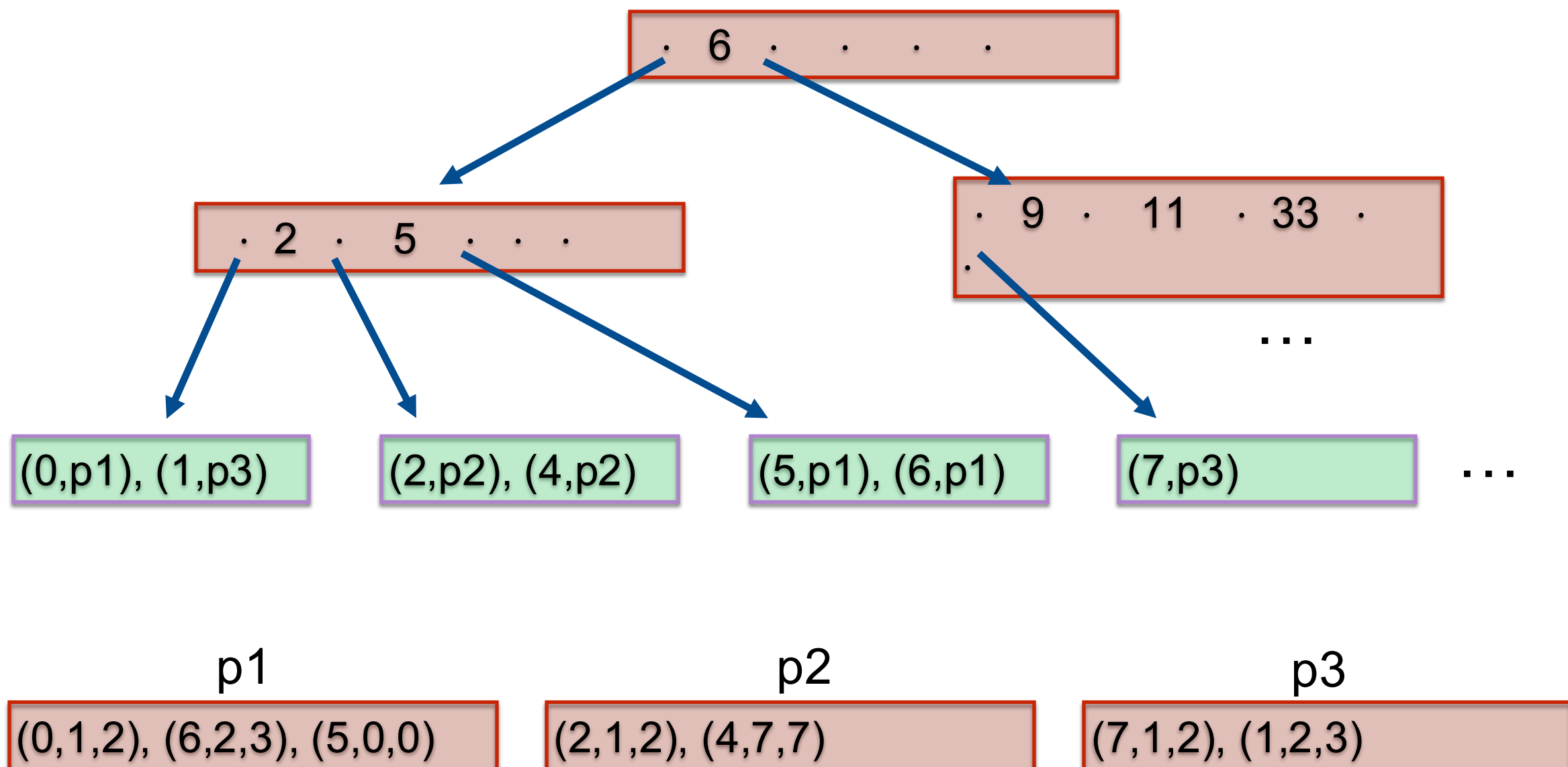


$$\#(I/O) = h + B_m$$

B_m : nr de bloques con tuplas que calzan búsqueda

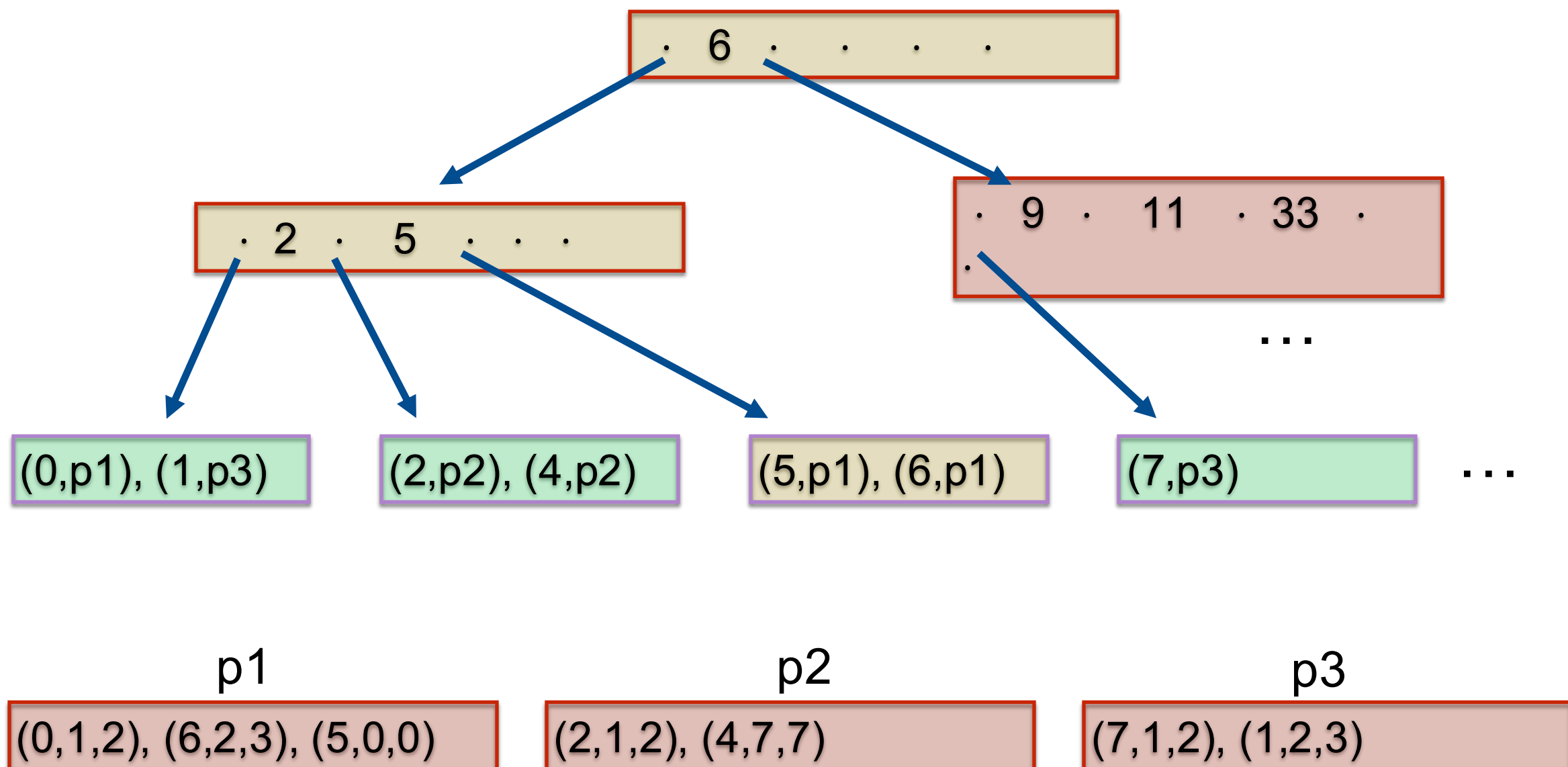
B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



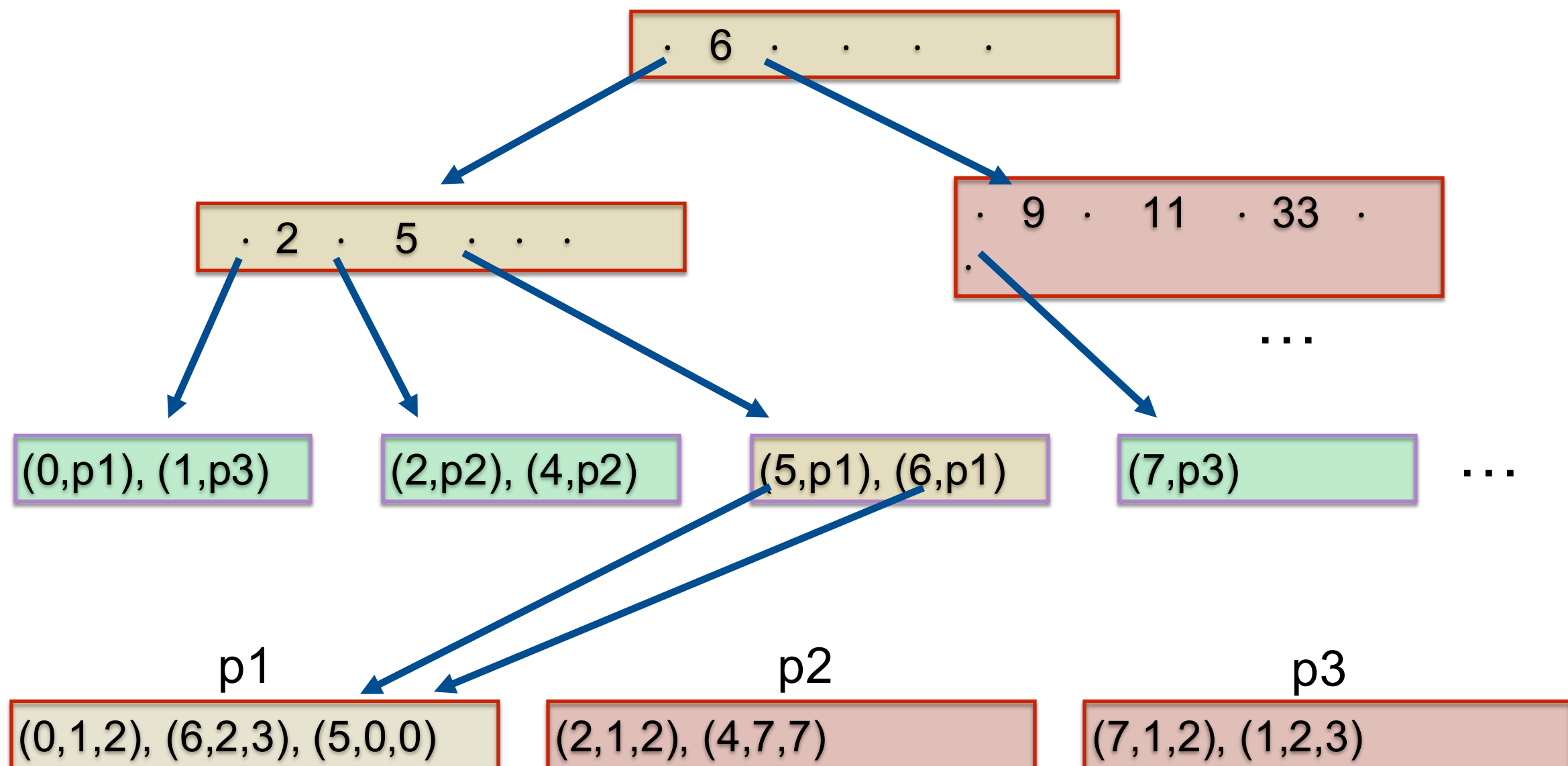
B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



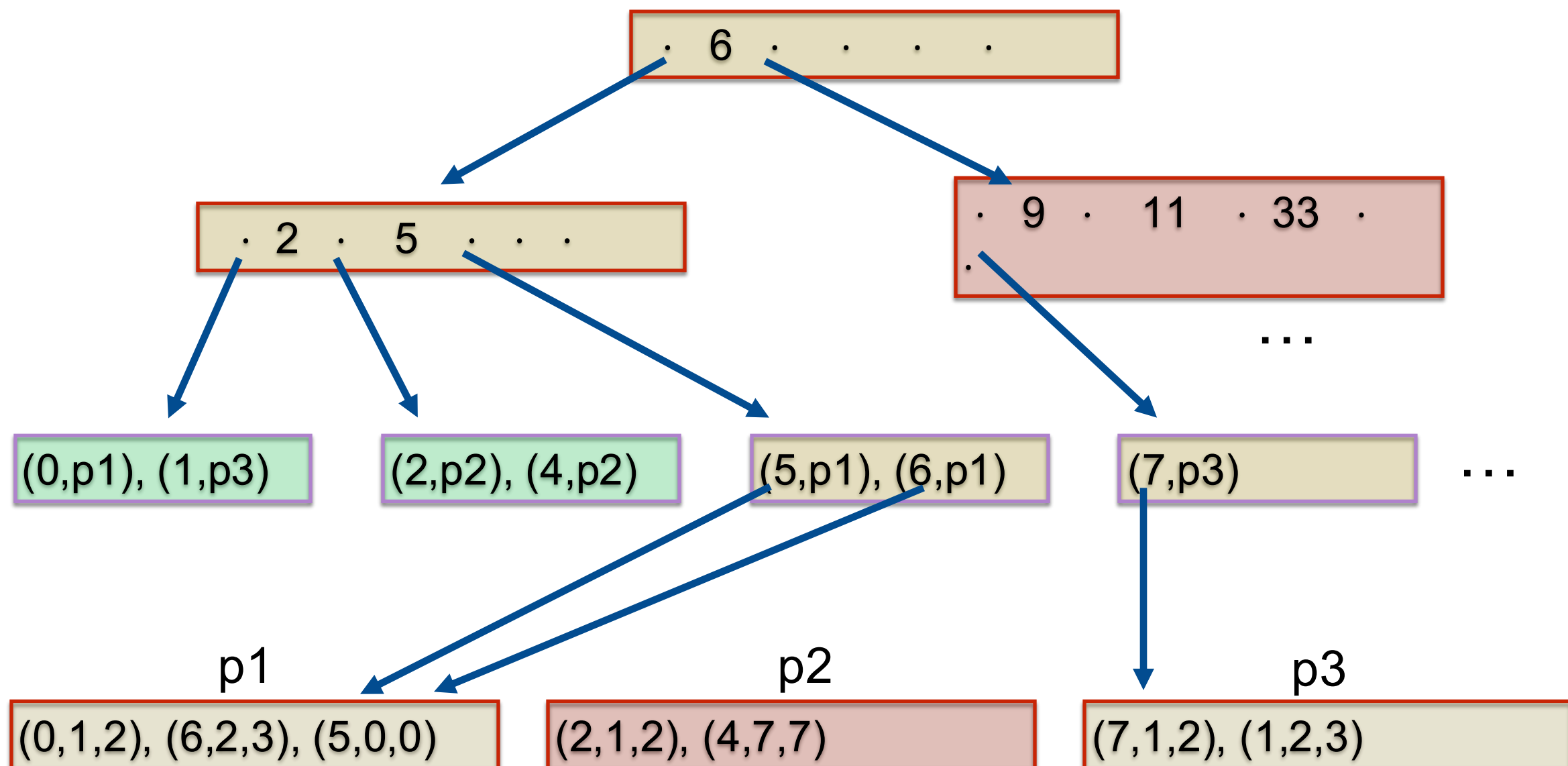
B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$



B+ tree unclustered

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$

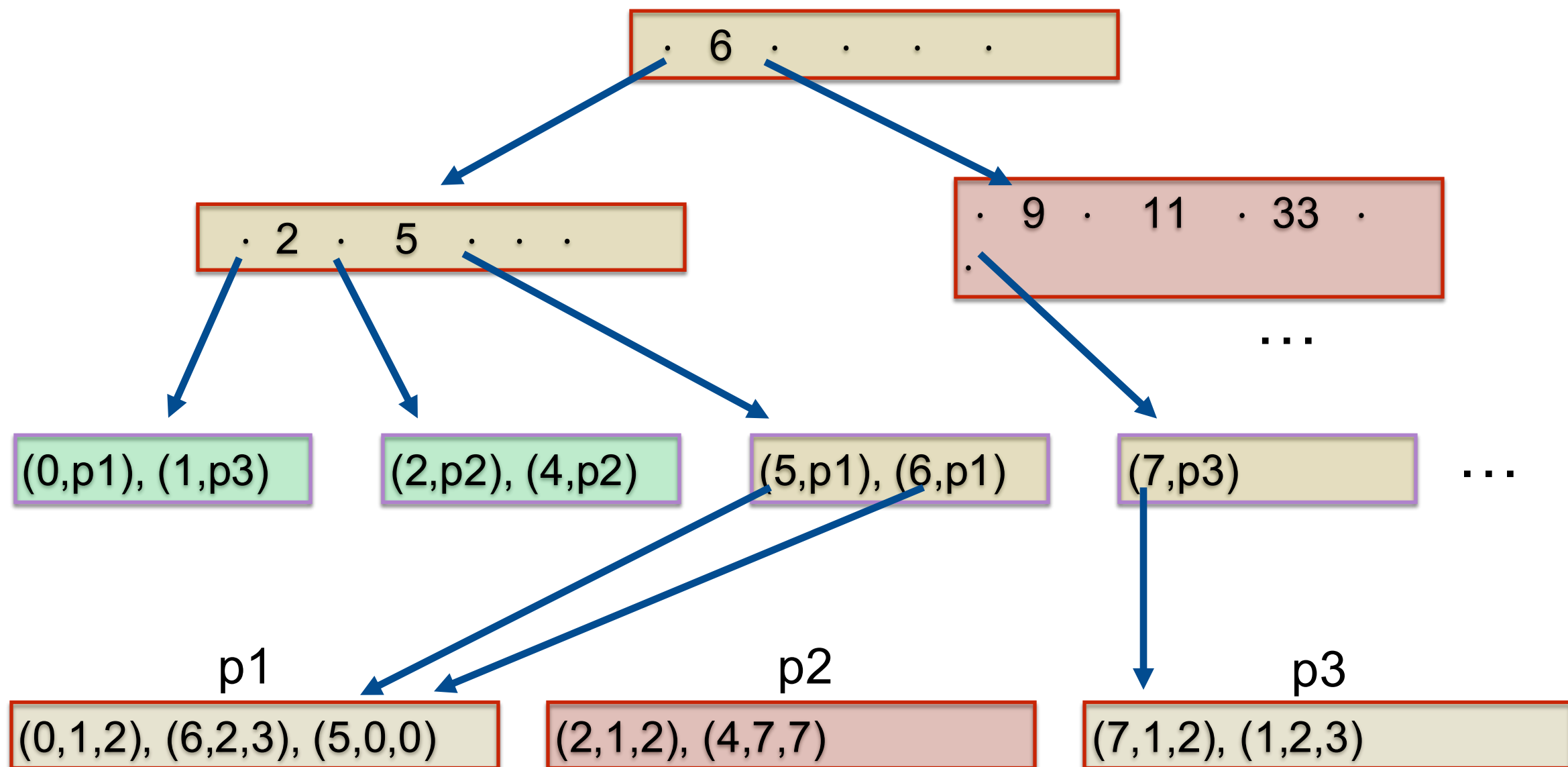


B+ tree unclustered

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

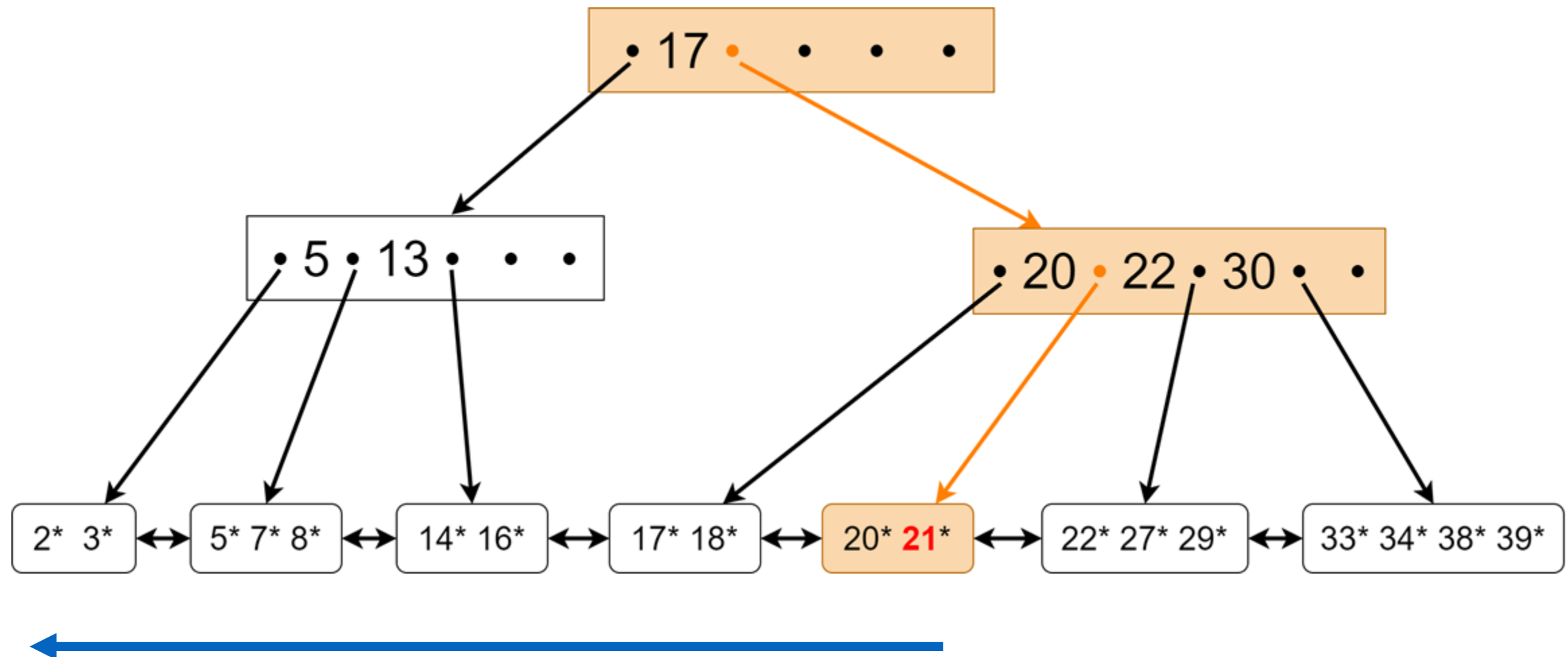
n : número de tuplas que calzan con búsquedas
 P : Cantidad de punteros por bloque

$R(\underline{a}, b, c)$... índice sobre a ... $a > 5$

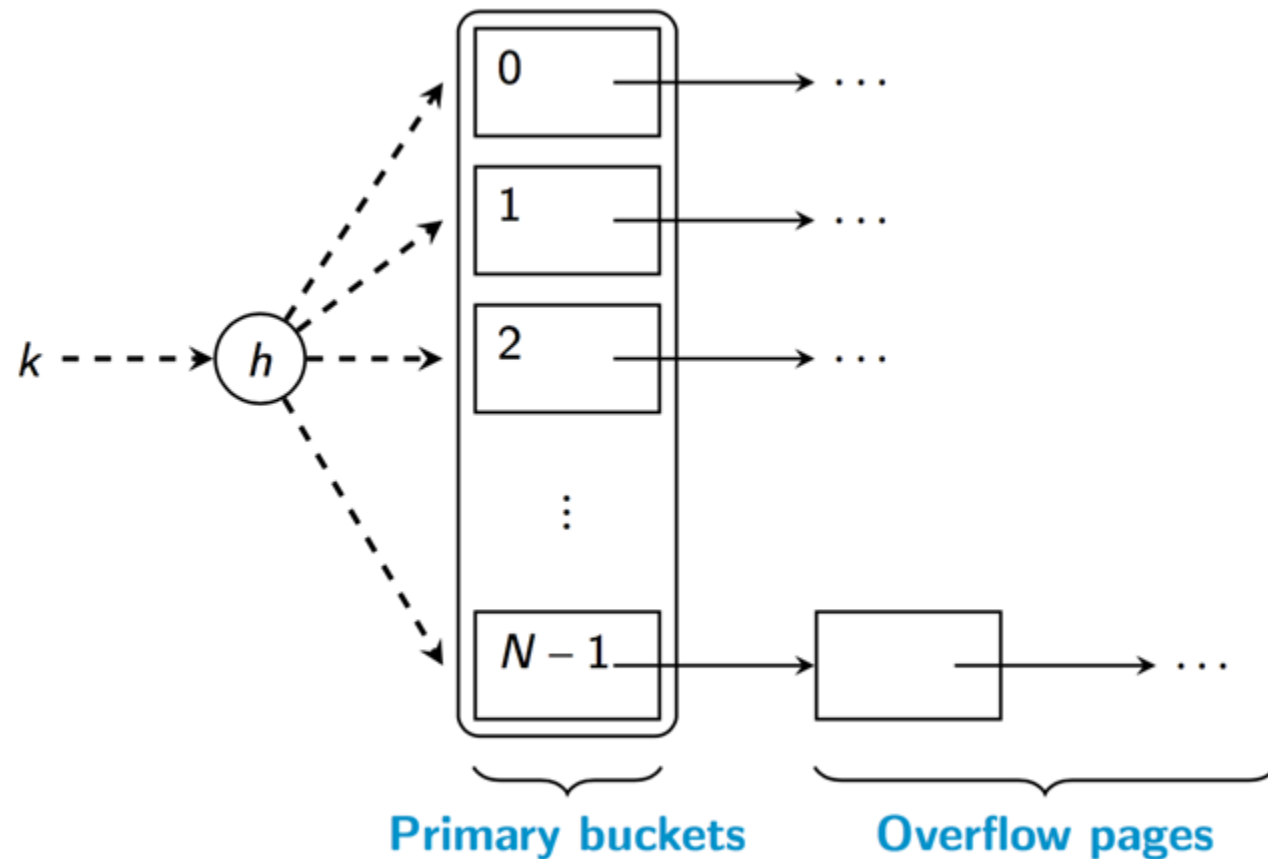


B+ tree unclustered

SELECT * FROM R WHERE a < 21



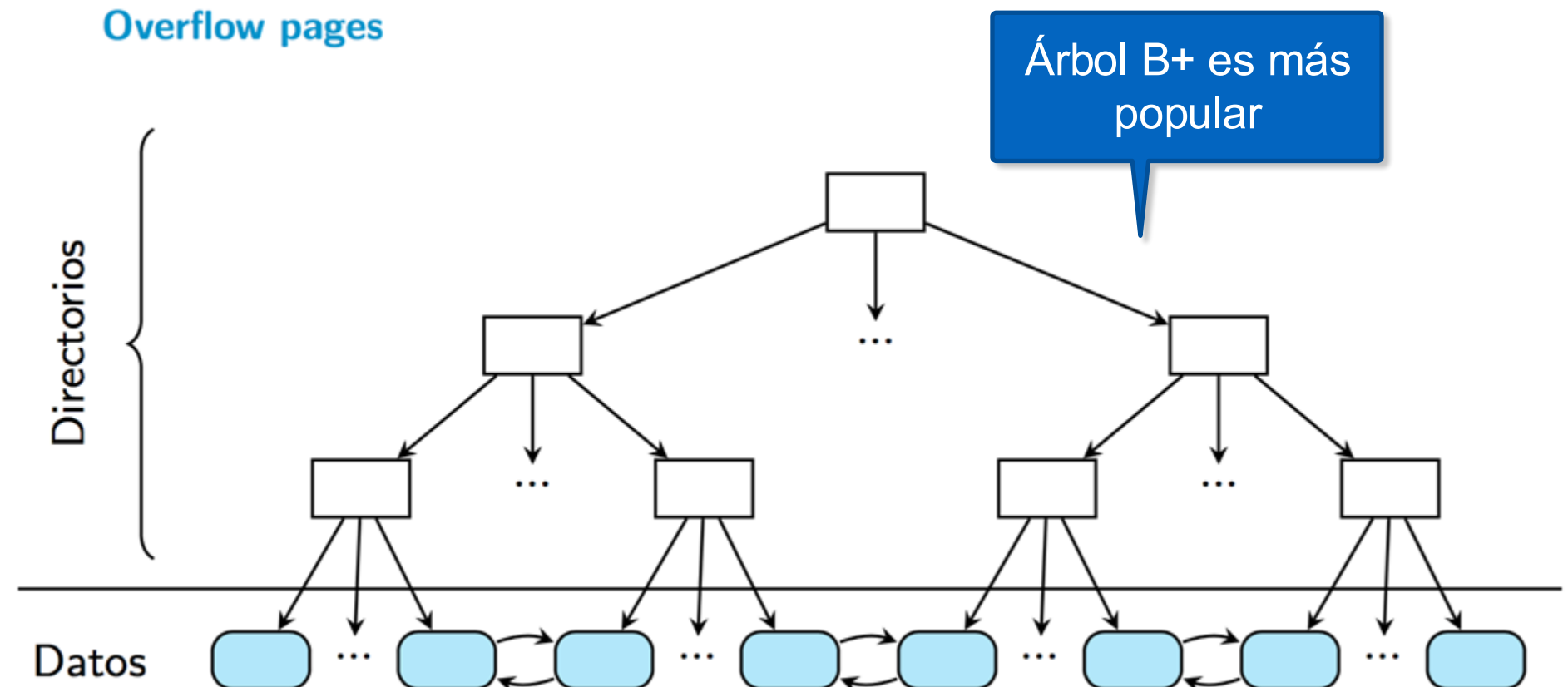
Índices: Hash vs. Árbol B+



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

La selección de la función de hash está disponible solo en algunos DBMS

Mucho más eficiente para búsquedas por rango



Recursos para estudiar

[Database Management Systems, 3rd edition, de Raghu Ramakrishnan y Johannes Gehrke.](#) Part III