Índices basados en Hashing

Clase 06

IIC 3413

Prof. Cristian Riveros

Clase pasada: Eficiencia de B+-trees

Considere:

- T: el número de tuplas.
- B: el tamaño de una página.

El costo de cada operación (en I/O):

Busqueda: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$

Insertar: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$

Eliminar: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$

Costo depende logaritmicamente en base B!

Clase pasada: Duplicados en B+-trees

Suposición anterior: **NO** hay data-entries duplicados.

Si consideramos **duplicados**, tenemos varias opciones . . .

- usar páginas con **overflow** , o
- data entries extendidos con una llave compuesta $(k, \texttt{id})$, o
- permitir duplicados y flexibilizar los intervalos del directorio:

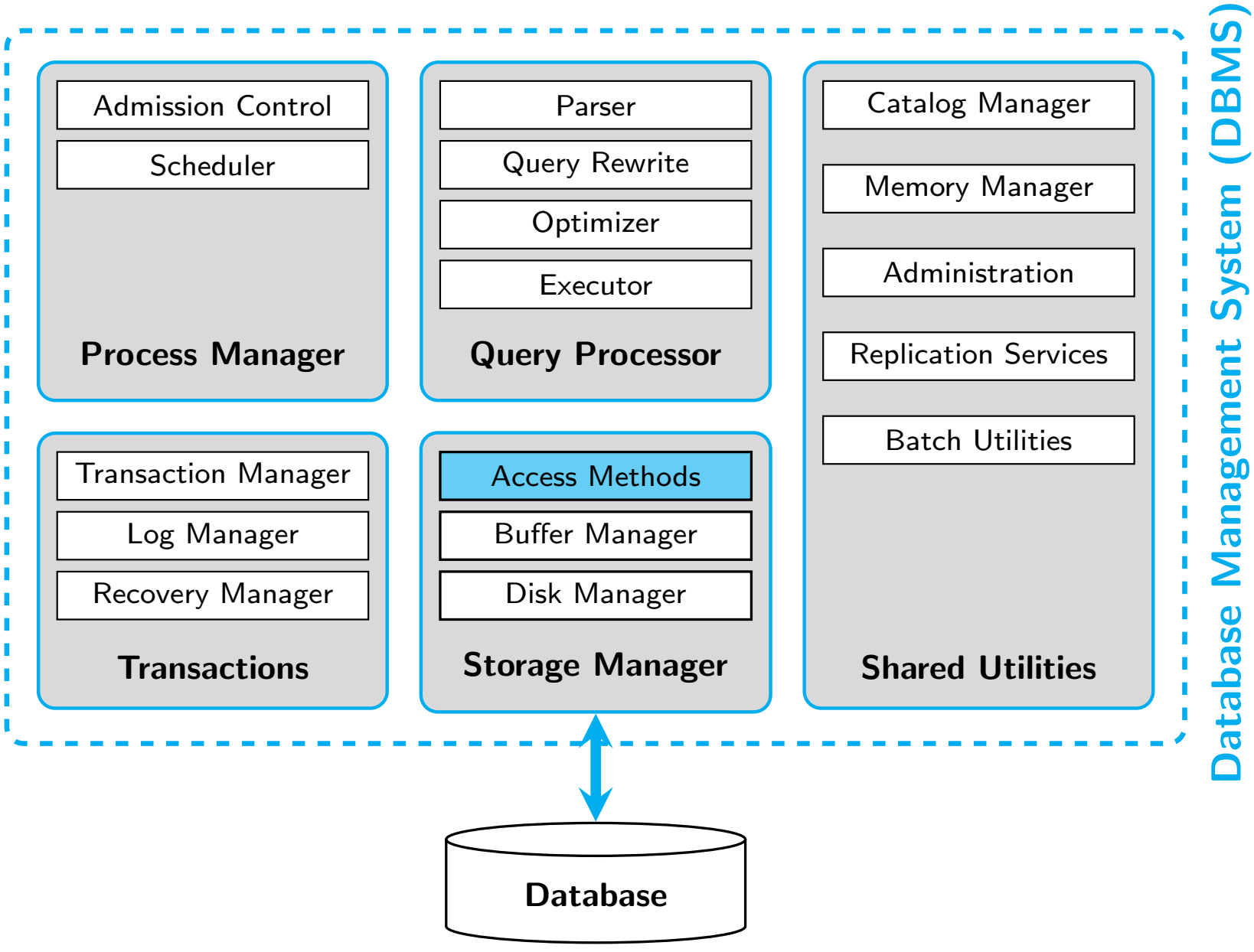
$$k_i \leq k < k_{i+1} \quad \Rightarrow \quad k_i \leq k \leq k_{i+1}$$

Clase pasada: Optimizaciones a B+-trees y otros

Varias posibles optimizaciones para B+-trees:

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

Índices basados en Hashing



Índices basados en Hashing

Eficientes para consultas de valores (value query):

```
SELECT   *
FROM     Players
WHERE    pAge = 30
```

Usado para la implementación de operadores relacionales.

- Ejemplo: Index-join o Hash-join.

Soportado en la mayoría de los motores de BD.

Hash index vs B+-trees

- B+-trees soporta range queries.
- Hash index es más eficiente para value queries.

Outline

Static hashing

Extendable Hashing

Outline

Static hashing

Extendable Hashing

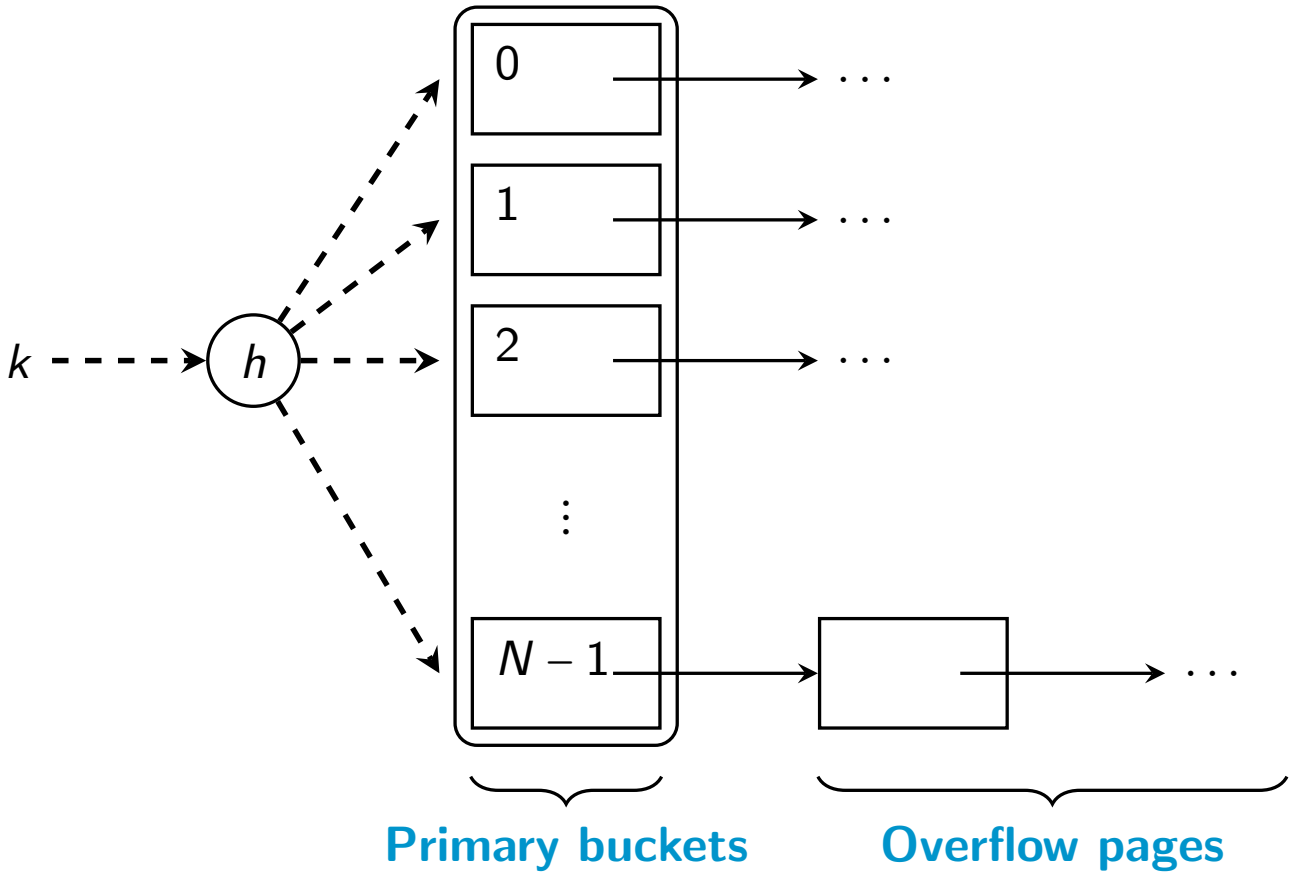
Estructura de una tabla de hash

Para un relación R y atributo X:

- N de páginas en disco (**buckets**).
- Cada bucket cuenta con una lista ligada de overflow pages.
- Usamos una **función de hash** h tal que:

$$h : \text{datatype}(X) \rightarrow [0, \ldots, N-1]$$

Estructura de una tabla de hash



Operaciones en static hashing

Para las operaciones `search(key k)`, `insert(key k)`, `delete(key k)`:

1. Computar el valor $h(k)$.
2. Acceder el **bucket** en la posición $h(k)$.
3. Buscar, insertar, o eliminar
   la tupla en la página $h(k)$ o en sus **overflow pages**.

Costo de cada operación: $1 + \#(\text{overflow pages})$.

Funciones de hash y colisiones

Colisión: valores k_1 y k_2 tal que $h(k_1) = h(k_2)$.

La idea es encontrar una función de hash h,
tal que, para toda secuencia de valores:

$$k_1, k_2, \ldots, k_n$$

el número de colisiones sea mínimo.

Buscamos que la función de hash se comporte
de forma **aleatoria** sobre $k_1, \ldots, k_n$

Ejemplo de funciones de Hash

- División:

$$h(k) = k \mod N$$

- Multiplicación:

$$\lfloor N \cdot \underbrace{(Q \cdot k - \lfloor Q \cdot k \rfloor)}_{\text{P. decimal de } Q \cdot k} \rfloor$$

Para cualquier $Q \in \mathbb{Q}$, como por ejemplo:

$$Q = \frac{2}{\sqrt{5}+1}$$

(inverso del golden ratio).

Para más información sobre funciones de hash
véase el curso de Estructura de Datos o Análisis de Algoritmos.

Costo de operaciones en Hash Index

Dado:

- una buena selección de la función de hash h y
- $\#(\text{data-entries}) \propto B \cdot N$
  (con B el tamaño de una página y N el número de buckets).

se tiene que el costo de cada operación es constante.

$$\text{Costo de cada operación} \approx \frac{\#(\text{data-entries})}{B \cdot N}$$

¿cuál es el problema de este costo?

Hashing dinámico al rescate

Idea (intuitiva)

Aumentar N (#(buckets))
dependiendo del número de data-entries.

Dos soluciones de **hashing dinámico**:

- Extendable hashing. ✓
- Linear hashing. ✗

Outline

Static hashing

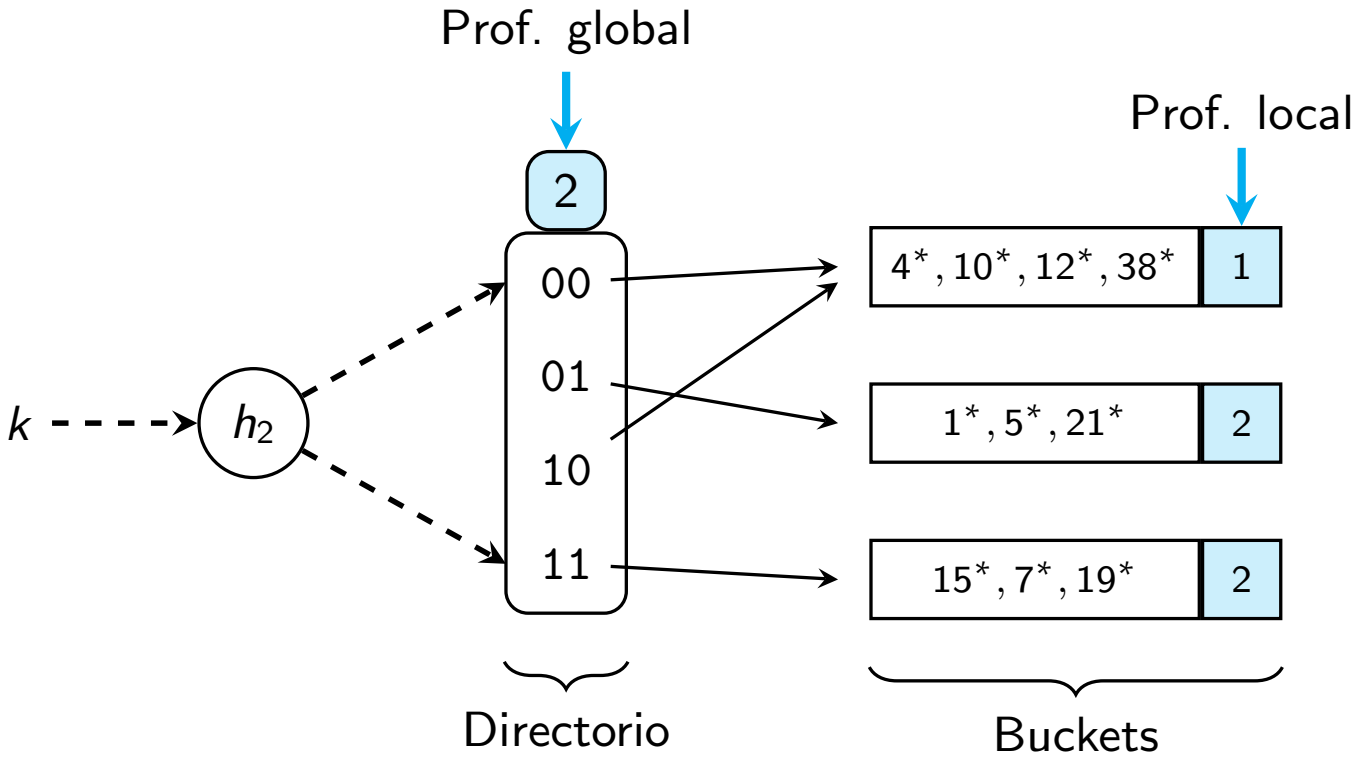
Extendable Hashing

Extendable Hashing

Idea (intuitiva)

1. Usar un **directorio** de buckets principales.
2. **Duplicar** el tamaño del direc. cuando haya un **overflow** de un bucket.

Extendable Hashing



- h_n considera los últimos n bits de k.

Profundidad global y profundidad local

Profundidad **global** (n):

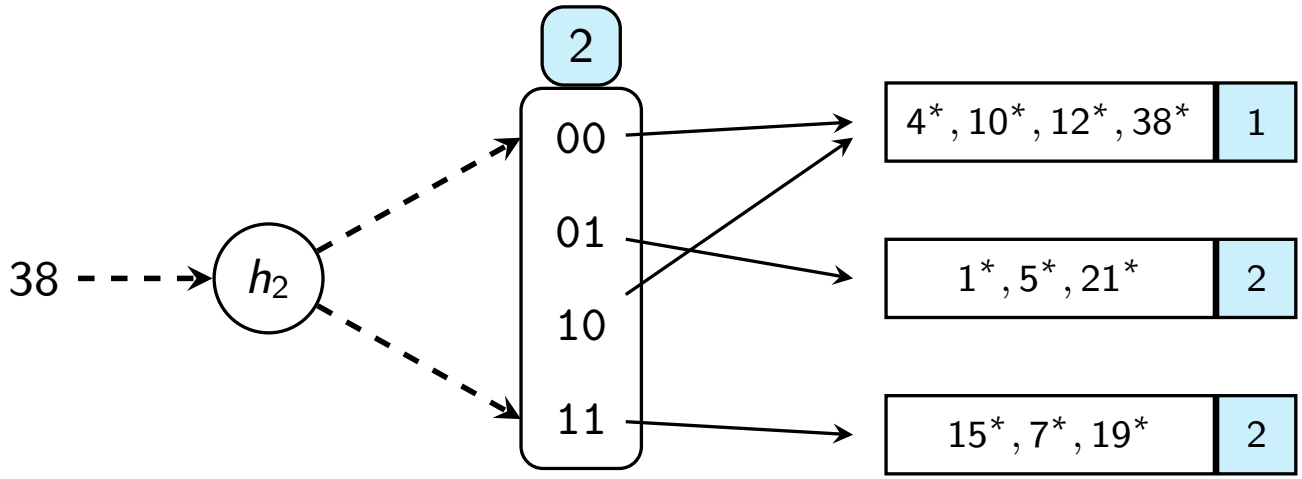
- La función h_n considera los n-últimos bits.
- El directorio es de tamaño 2^n.

Profundidad **local** (d) para cada bucket:

- $h_d(k) = h_d(k')$ para todo k, k' en el mismo bucket.
- Todos los elementos del bucket coinciden en los últimos d bits.

Search en Extendable Hashing

Buscar el elemento 38. (38 = 100110 y $h_2(38) = 10$)



Insertar en Extendable Hashing

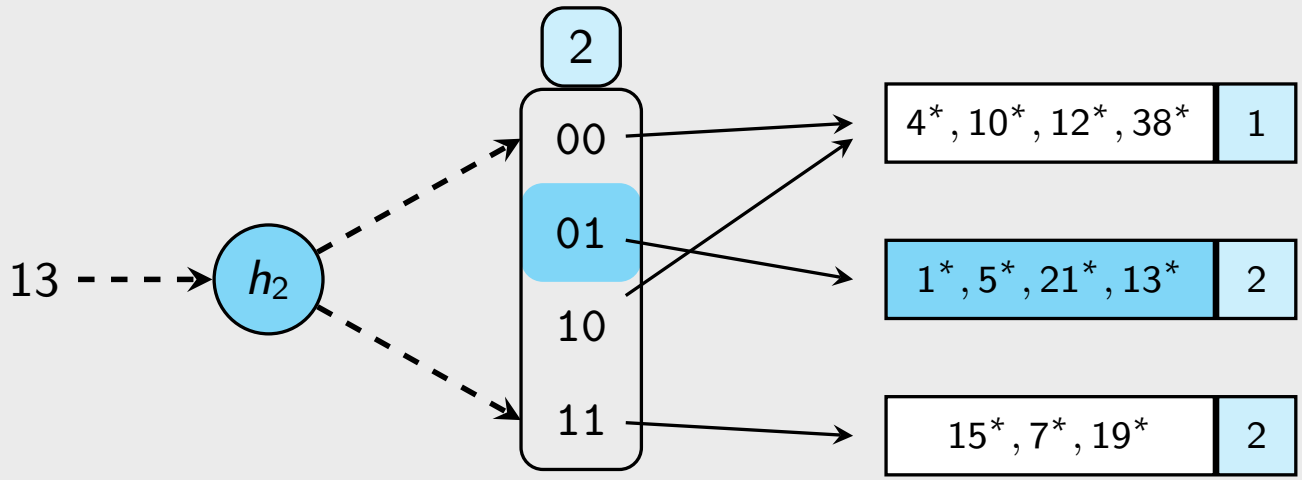
Para insertar una llave k con profundidad global n:

1. Computamos $h_n(k)$ (los últimos n bits de k).
2. Buscamos en el directorio el puntero p para la entrada $h_n(k)$.
3. Seguimos el puntero p al bucket B correspondiente.
4. Si hay espacio en B, insertamos k en B.

Insertar en Extendable Hashing

Ejemplo

Insertamos $k = 13$. $(13 = 1101$ y $h_2(13) = 01)$



Insertar en Extendable Hashing

Para insertar una llave k con profundidad global n:

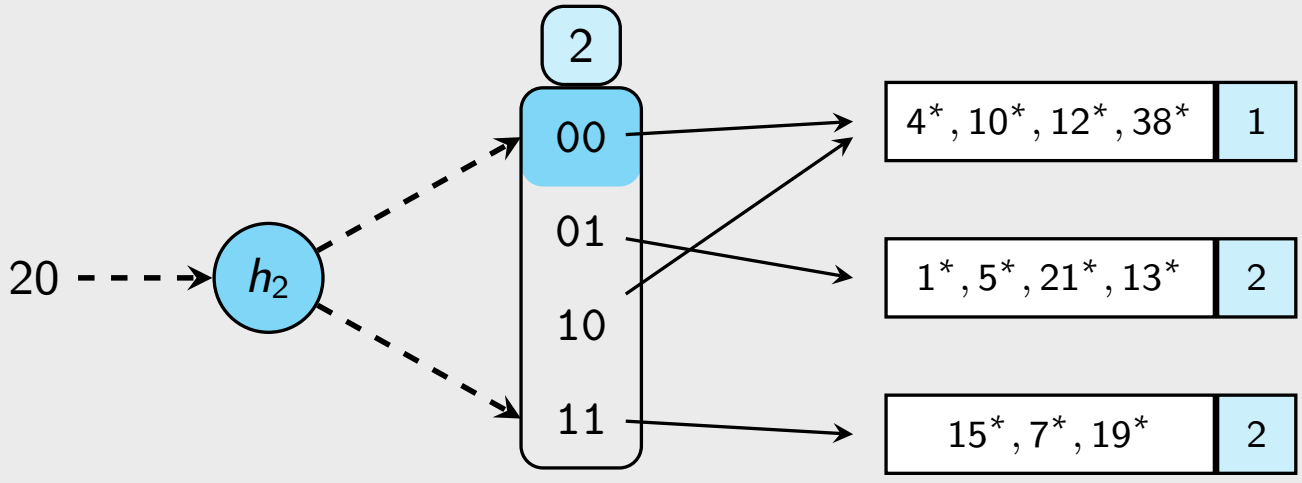
1. Computamos $h_n(k)$ (los últimos n bits de k).
2. Buscamos en el directorio el puntero p para la entrada $h_n(k)$.
3. Seguimos el puntero p al bucket B correspondiente.
4. Si hay espacio en B, insertamos k en B.
5. Si NO hay espacio en B con profundidad local d:
    - Hacemos un **bucket split** de B.

Bucket Split

Dado un bucket B con profundidad local d,
dividimos sus elementos en dos buckets según el bit $d + 1$.

Ejemplo

Insertamos $k = 20$. $(20 = 10100$ y $h_2(20) = 00)$



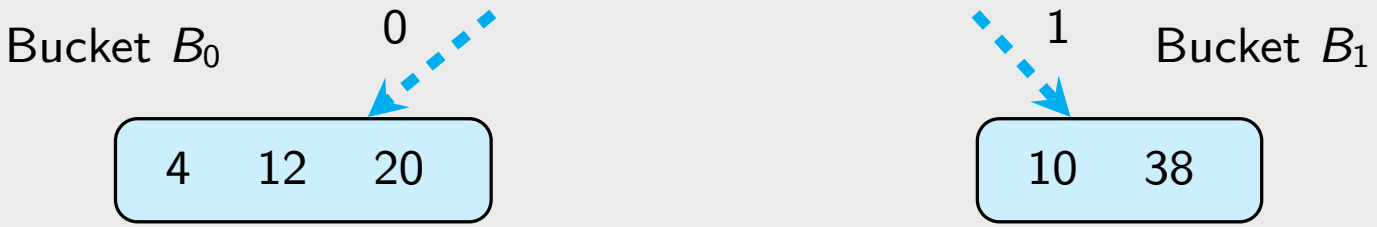
Bucket Split

Dado un bucket B con profundidad local d,
dividimos sus elementos en dos buckets según el bit $d+1$.

Ejemplo

Dividimos el bucket 00 en dos:

$$
\begin{aligned}
4 &= 1\underline{0}0 \\
10 &= 10\underline{1}0 \\
12 &= 11\underline{0}0 \\
38 &= 100\underline{1}10 \\
20 &= 101\underline{0}0
\end{aligned}
$$



Insertar en Extendable Hashing

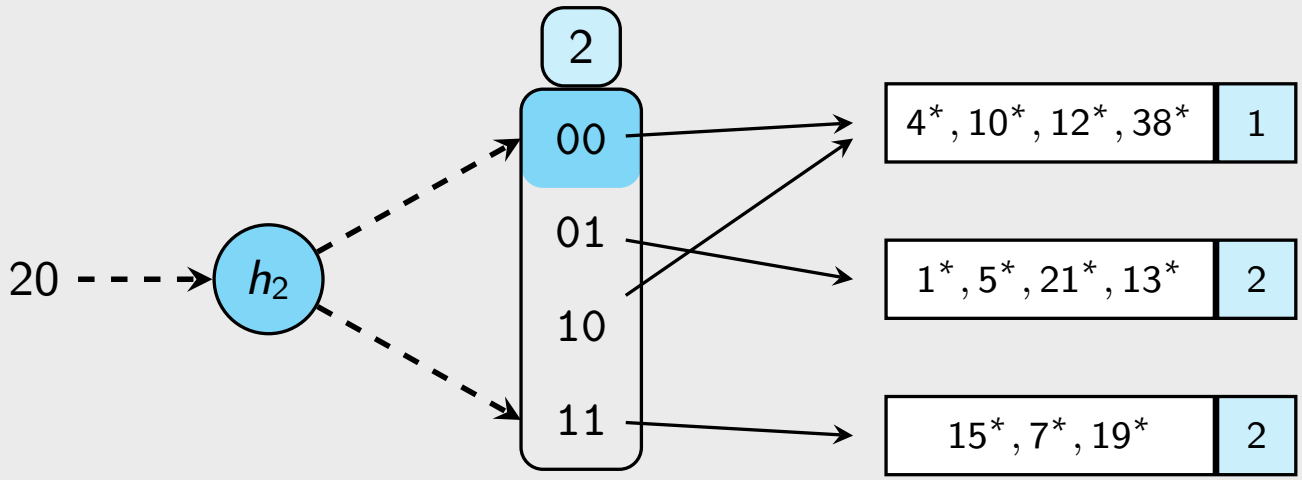
Para insertar una llave k con profundidad global n:

1. Computamos $h_n(k)$ (los últimos n bits de k).
2. Buscamos en el directorio el puntero p para la entrada $h_n(k)$.
3. Seguimos el puntero p al bucket B correspondiente.
4. Si hay espacio en B, insertamos k en B.
5. Si NO hay espacio en B con profundidad local d:
   - Hacemos un bucket split de B.
   - **Aumentamos** la profundidad local de B_0 y B_1 a $d+1$.
   - Si $n > d$, **insertamos** B_0 y B_1 en el directorio.

Insertar B_0 y B_1 en el directorio

Ejemplo (continuación)

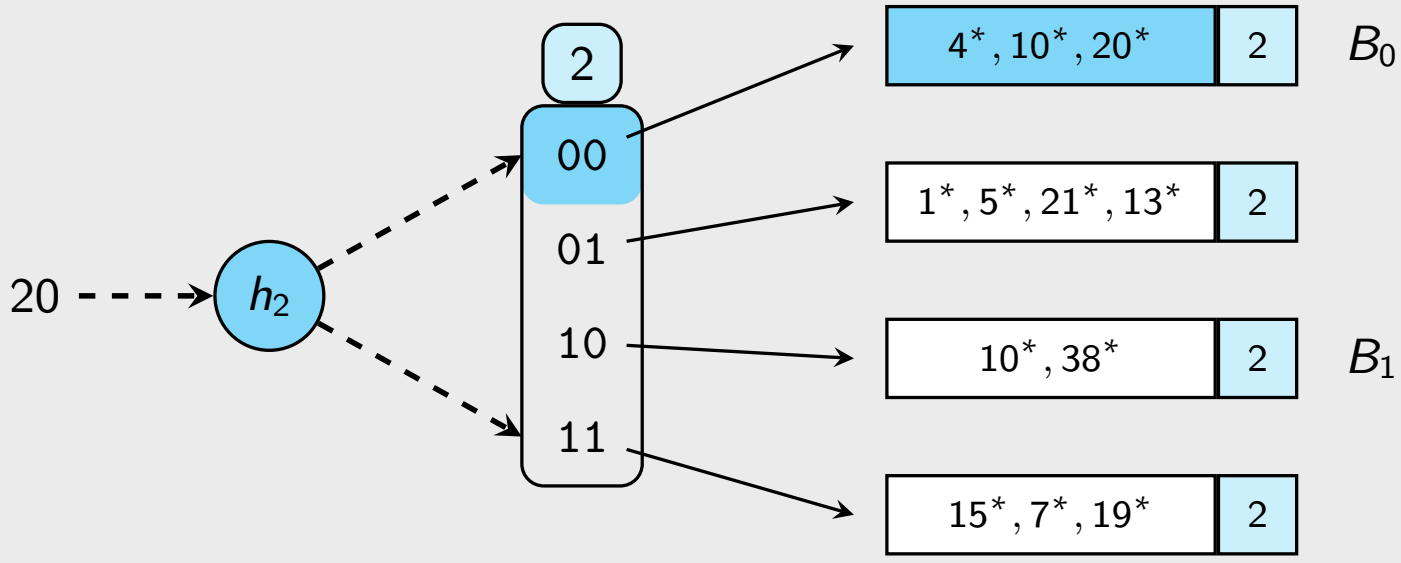
Insertamos $k = 20$. ($20 = 10100$ y $h_2(20) = 00$)



Insertar B_0 y B_1 en el directorio

Ejemplo (continuación)

Insertamos $k = 20$. ($20 = 10100$ y $h_2(20) = 00$)



Insertar en Extendable Hashing

Para insertar una llave k con profundidad global n:

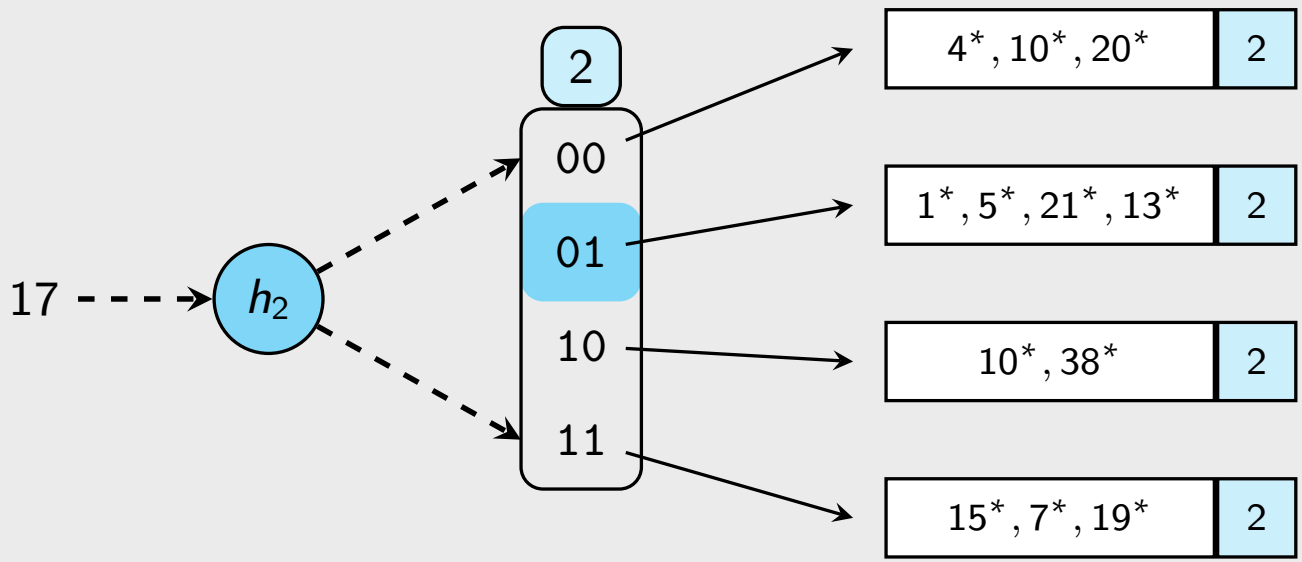
1. Computamos $h_n(k)$ (los últimos n bits de k).
2. Buscamos en el directorio el puntero p para la entrada $h_n(k)$.
3. Seguimos el puntero p al bucket B correspondiente.
4. Si hay espacio en B, insertamos k en B.
5. Si NO hay espacio en B con profundidad local d:
   - Hacemos un bucket split de B.
   - Aumentamos la profundidad local de B_0 y B_1 a $d+1$.
   - Si $n > d$, insertamos B_0 y B_1 en el directorio.
   - Si $n = d$, hacemos un **doblamiento del directorio**.

Doblamiento del directorio

Dado un bucket B con profundidad local $d = n$,
hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Ejemplo

Insertamos $k = 17$. $(17 = 10001$ y $h_2(17) = 01)$



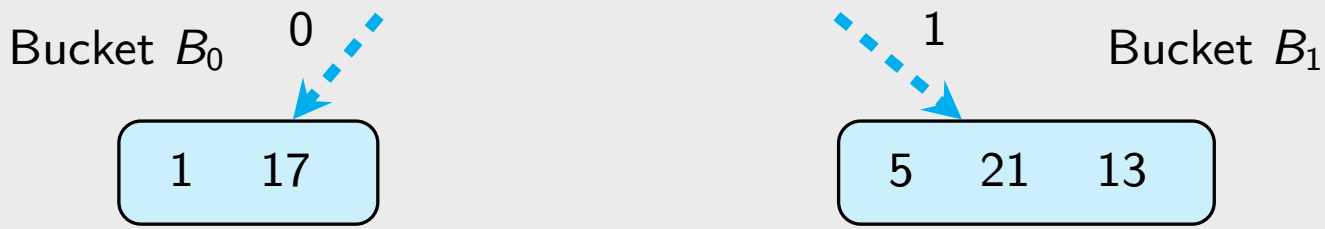
Doblamiento del directorio

Dado un bucket B con profundidad local $d = n$,
hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Ejemplo

Dividimos el bucket 01 en dos:

$$
\begin{aligned}
1 &= 0\underline{0}1 \\
5 &= 1\underline{0}1 \\
21 &= 10\underline{1}01 \\
13 &= 1\underline{1}01 \\
17 &= 10\underline{0}01
\end{aligned}
$$

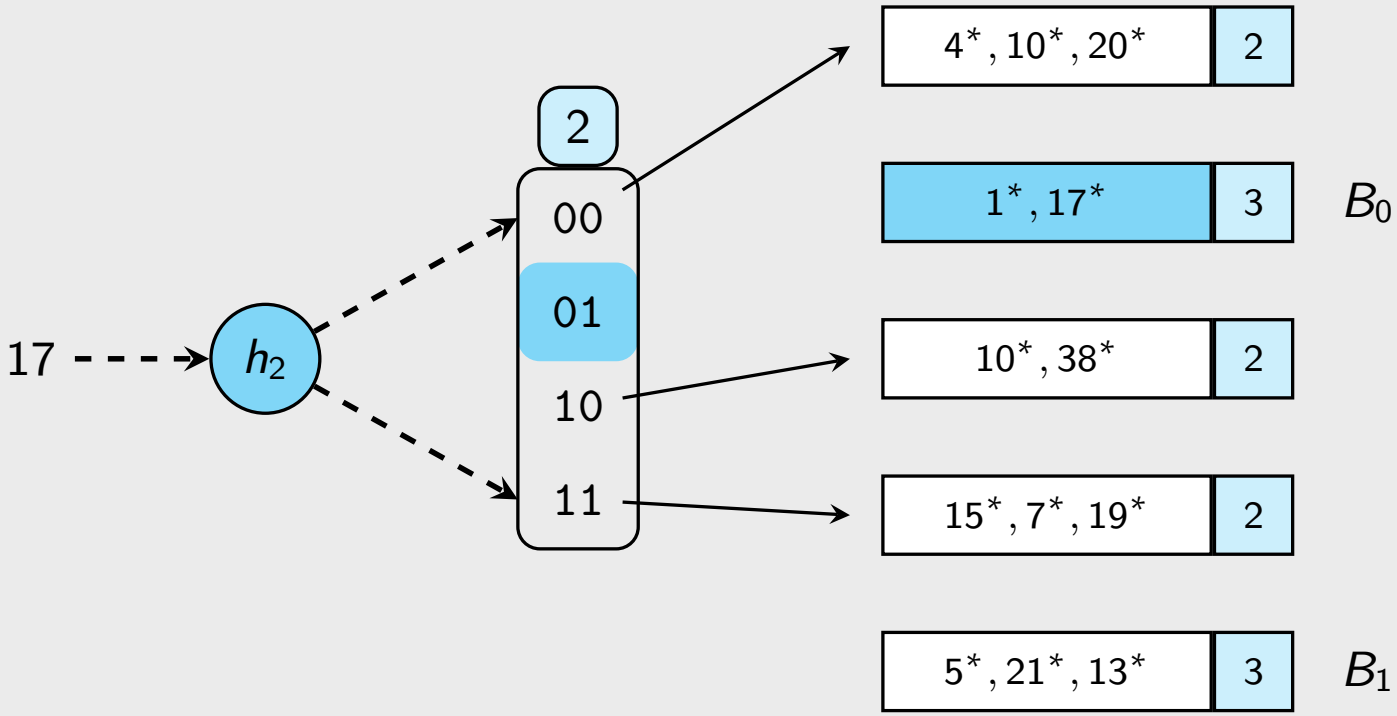


Doblamiento del directorio

Dado un bucket B con profundidad local $d = n$,
hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Ejemplo

Insertamos B_0 y B_1 en el directorio.

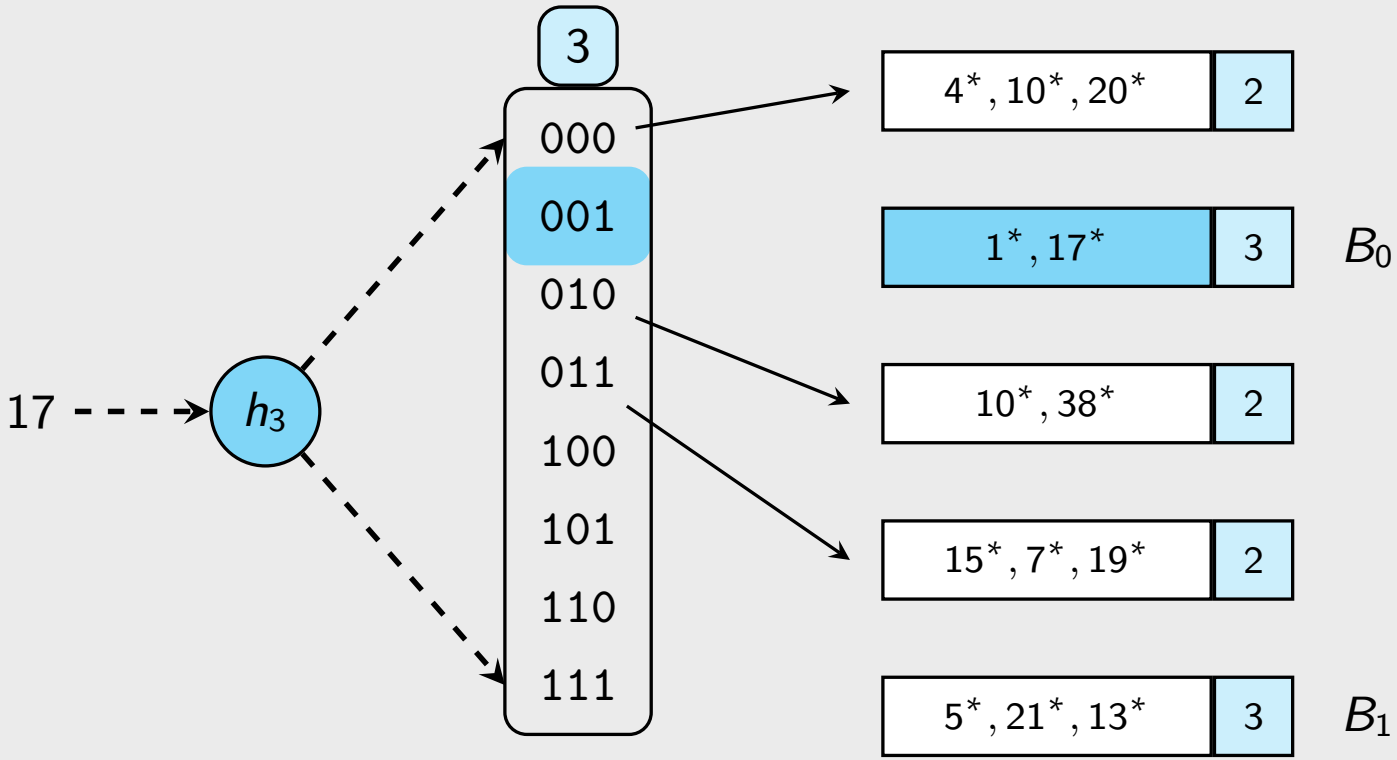


Doblamiento del directorio

Dado un bucket B con profundidad local $d = n$,
hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Ejemplo

Duplicamos el directorio:

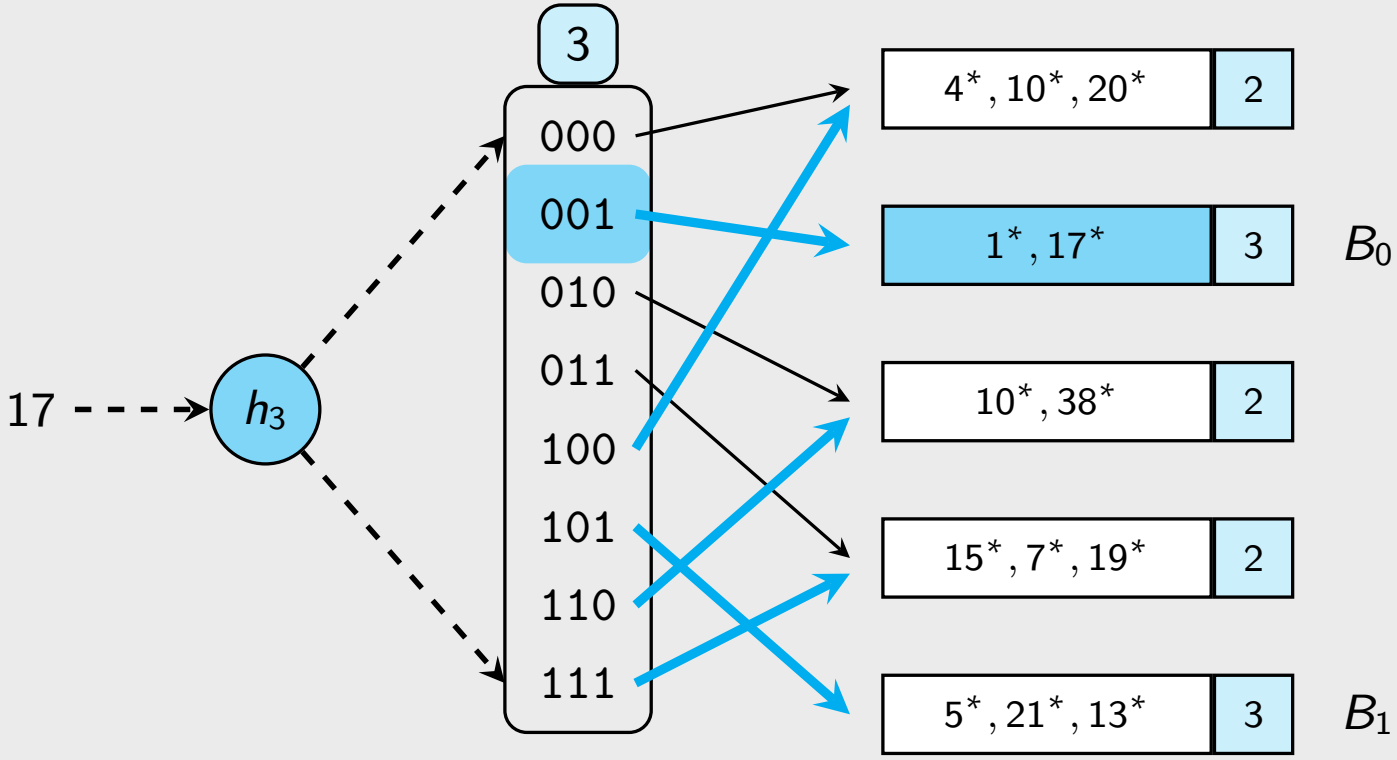


Doblamiento del directorio

Dado un bucket B con profundidad local $d = n$,
hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Ejemplo

Duplicamos el directorio:



Uso de overflow pages

- En principio, extendable hashing no hace uso de overflow pages, pero...
- en algunos casos es razonable usarlas.

¿en qué circunstancias sería prudente usar overflow pages?

Eliminación en Extendable Hashing

- Para eliminar, uno busca el elemento $k*$ y lo elimina del bucket.
- Si el bucket queda vació, extendable hashing hace **merge** de buckets.
  - ¿cuándo ocurre una disminución de la profundidad **local**?
  - ¿cuándo ocurre una disminución de la profundidad **global**?
- En general, **merge** de bucket es poco usado.

¿por qué?

Eficiencia de Extendable Hashing

Considere:

- D: tamaño (en páginas) del directorio.

El costo de cada operación (en I/O):

Search: $\mathcal{O}(1)$

Insertar: $\mathcal{O}(D)$