# Databases
# Lab

## Hash Table and Inverted Indexes

Flavio Bertini

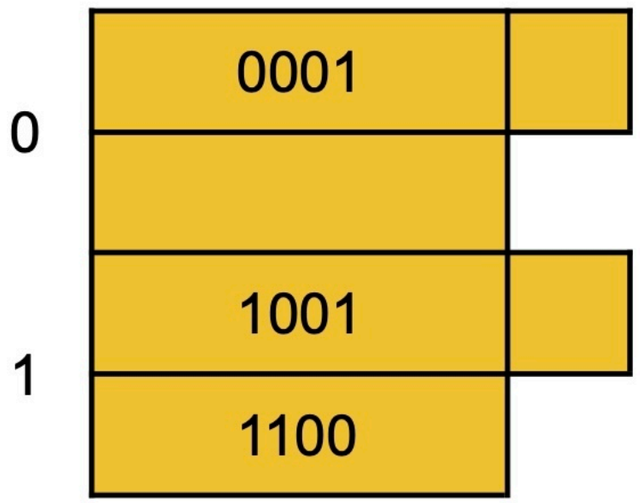
flavio.bertini@smartdata.cs.unibo.it

# Hash-based Indexes

- Hashing maps a search key directly to the pid of the containing page/page-overflow chain.
- Hash-based indexes are **best for equality search**. They do not support efficient range search.
- As with sorting-based indexing, there are static and dynamic hashing techniques.
  - **Static hashing**: for **fixed-size non-mutable data** (e.g., single session CD-ROM).
  - **Extensible and linear hashing**: when **both data and data sizes** could **vary in time**.
- As for the tree-based indexes for secondary memory, hash indexes **use blocks for storing buckets**.

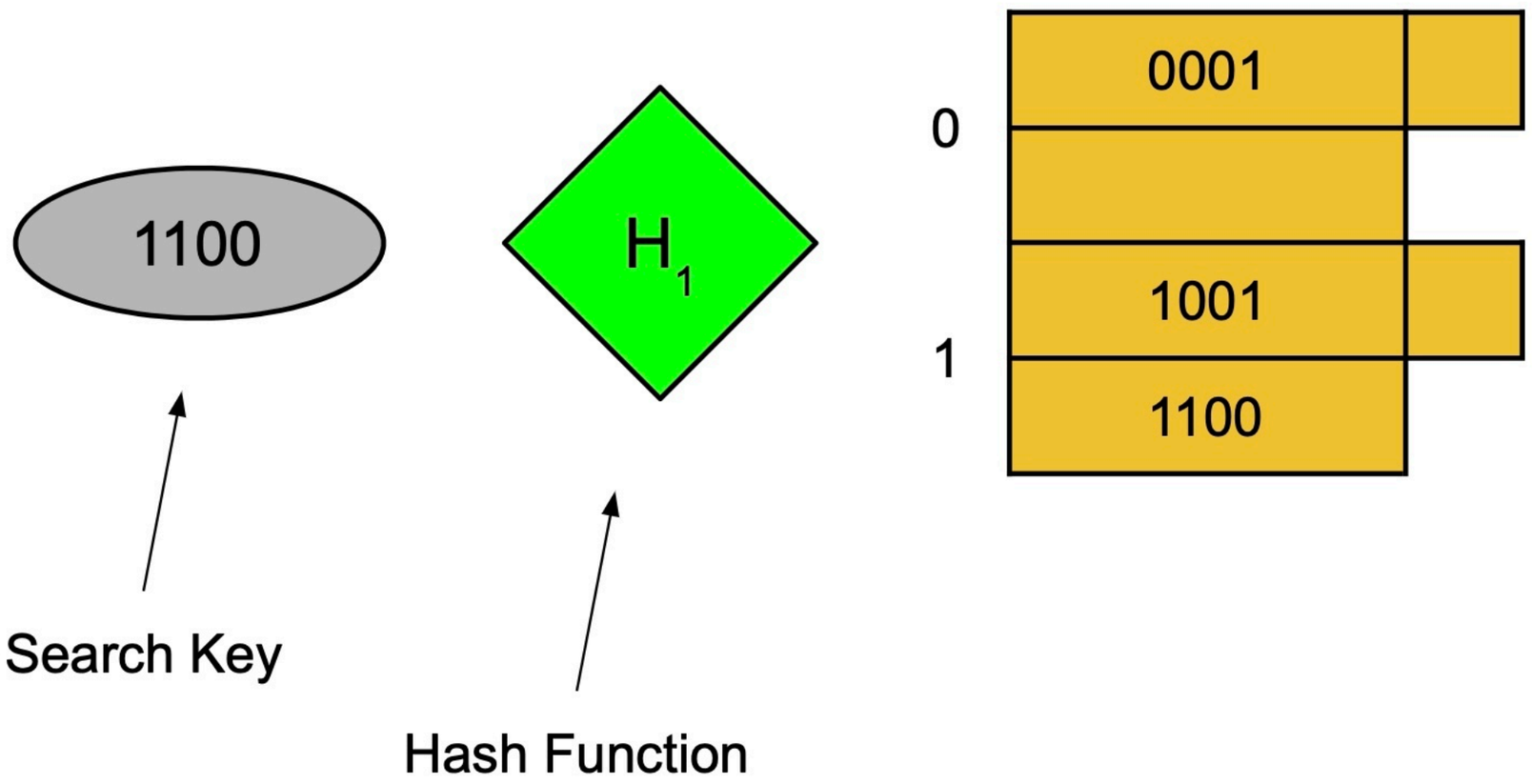
# Static Hashing

- Static hashing involves **N buckets** and a hash function that maps the search key in the range of 0 to N-1.
- In the following, we will use **$H_i$ hash functions**, which return the first (or last) ***i* bits of the binary encoding** of the search key.
  - If a record has search key K, then we store the record into the bucket numbered H(K).
- We will show the keys already encoded, as in the following example (by using the most significant bit). In this case, $i = 1$, $N = 2^i = 2$, and each bucket contains one block.



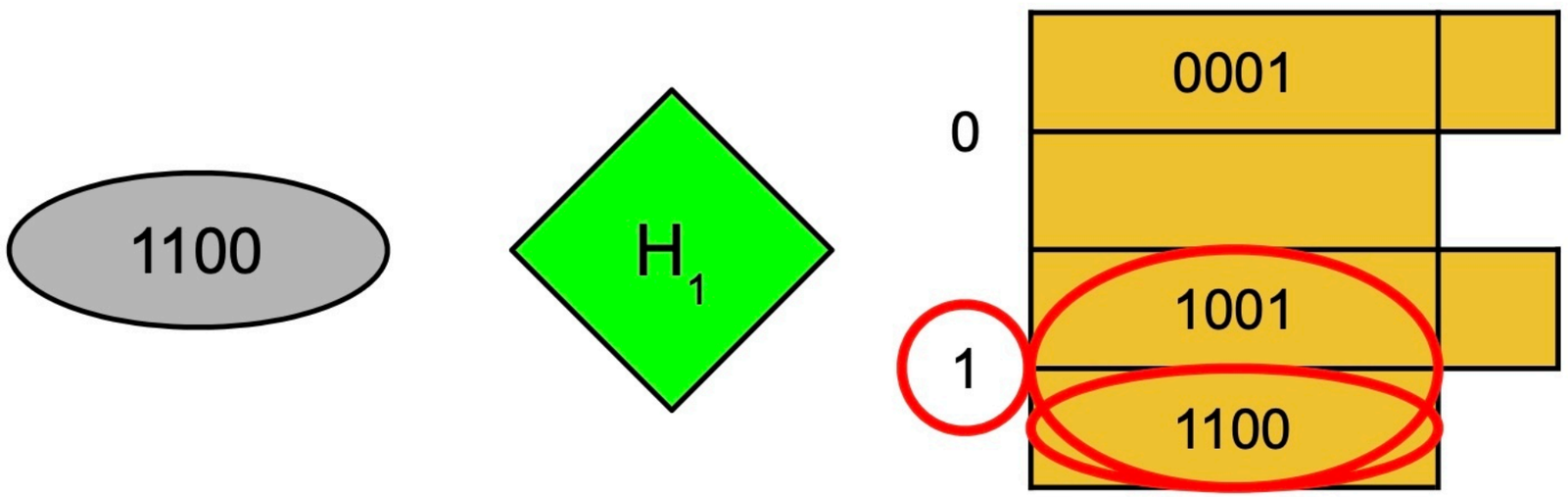
# Static Hashing: Searching (1)

- The hash function computes the address of the bucket where the data record with a given search key ($K_2$=1100) is located (if any).



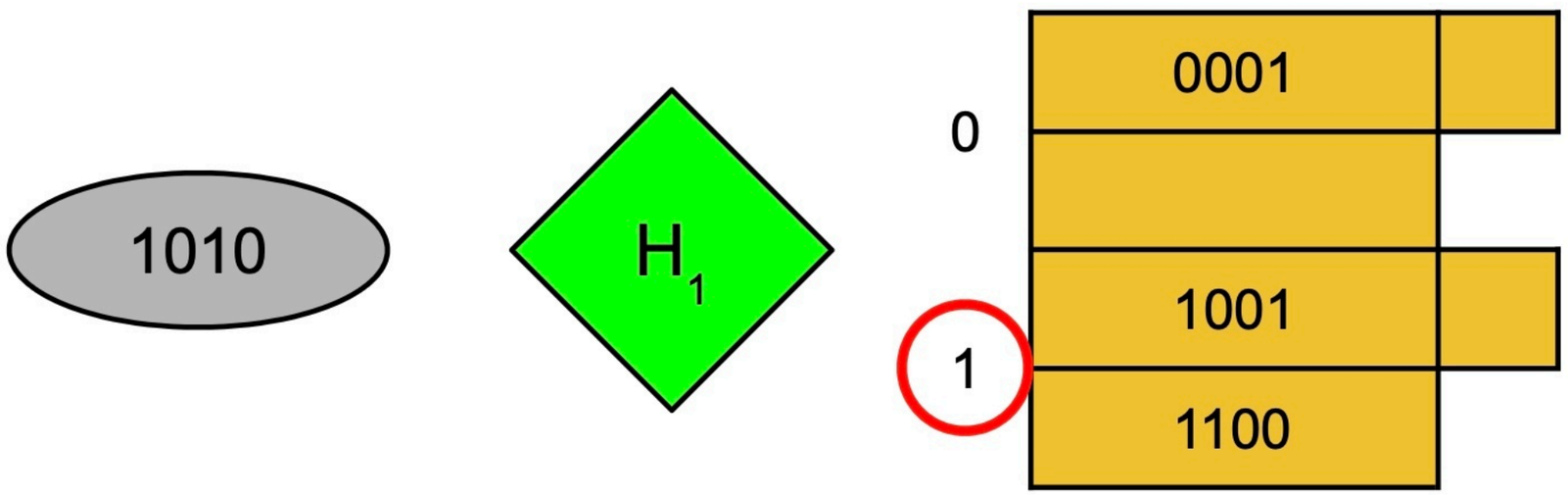
# Static Hashing: Searching (2)

- The hash function computes the address of the bucket where the data record with a given search key ($K_2$=1100) is located (if any).



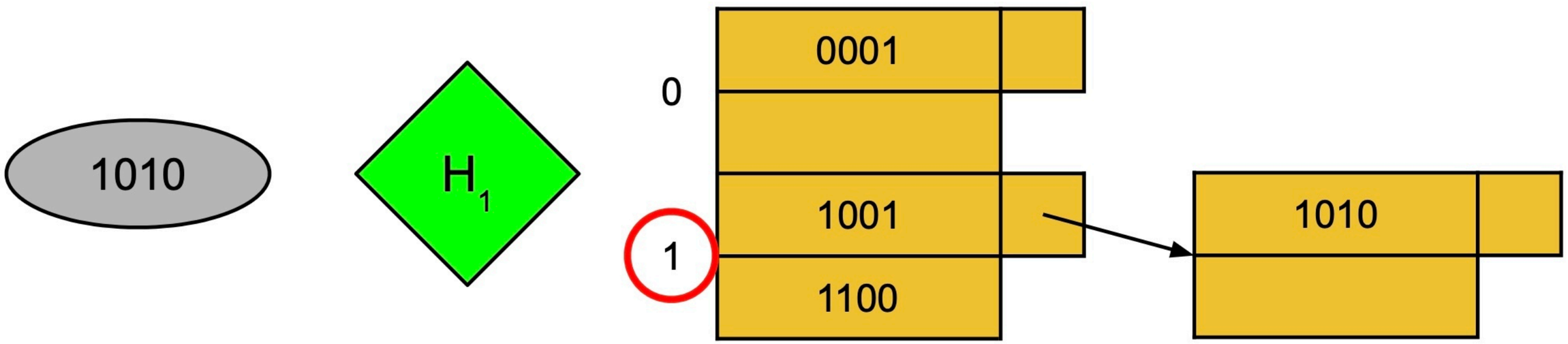
# Static Hashing: Insertion (1)

- Static hashing can not change the number of buckets. It can vary the size of the buckets, using a **chain of blocks of overflow**.



# Static Hashing: Insertion (2)

- Static hashing can not change the number of buckets. It can vary the size of the buckets, using a **chain of blocks of overflow**.



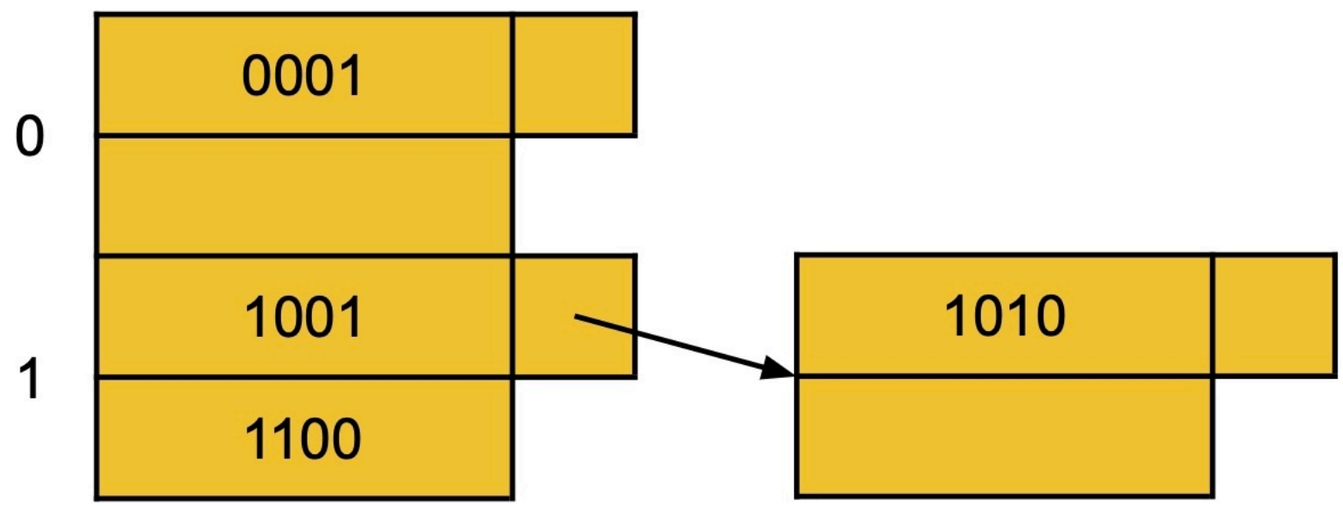
# How many buckets are there?

A. 0

B. 1

C. ✓ 2

D. 3



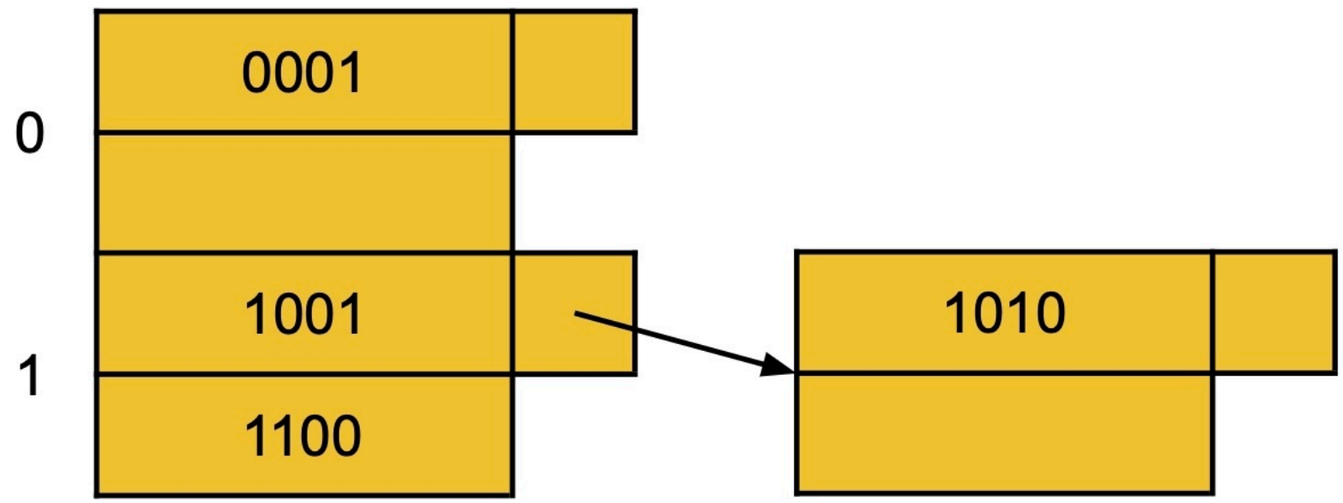
# How many blocks are there?

A. 0

B. 1

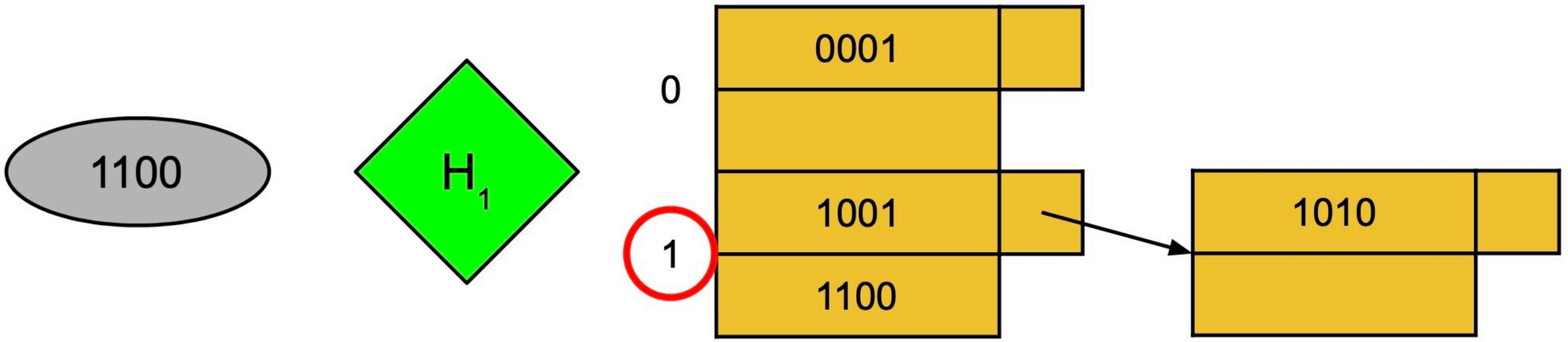
C. 2

D. 3 ✓



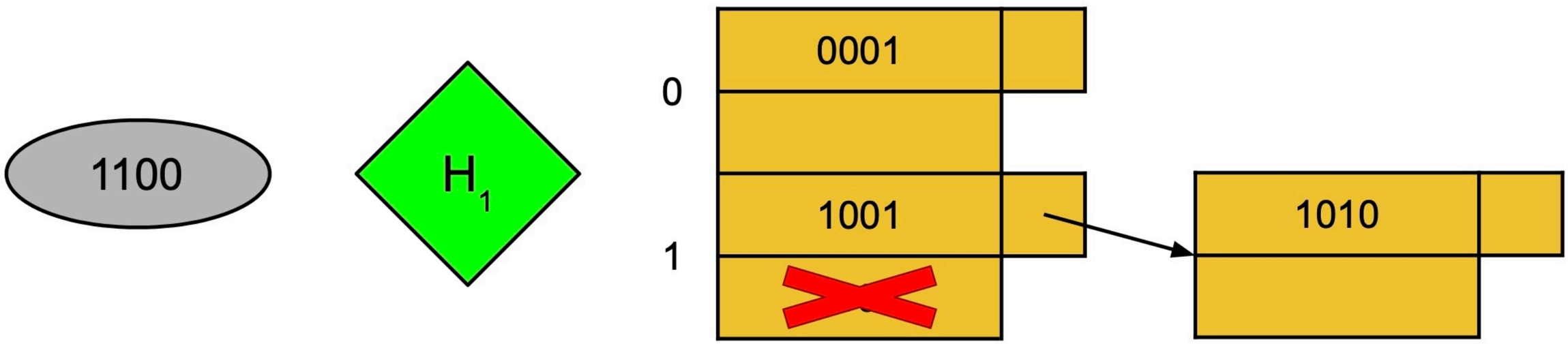
# Static Hashing: Deletion (1)

- Long chains of overflow blocks degrade the performance.
- Deleting keys can lead to the deletion of overflow blocks.



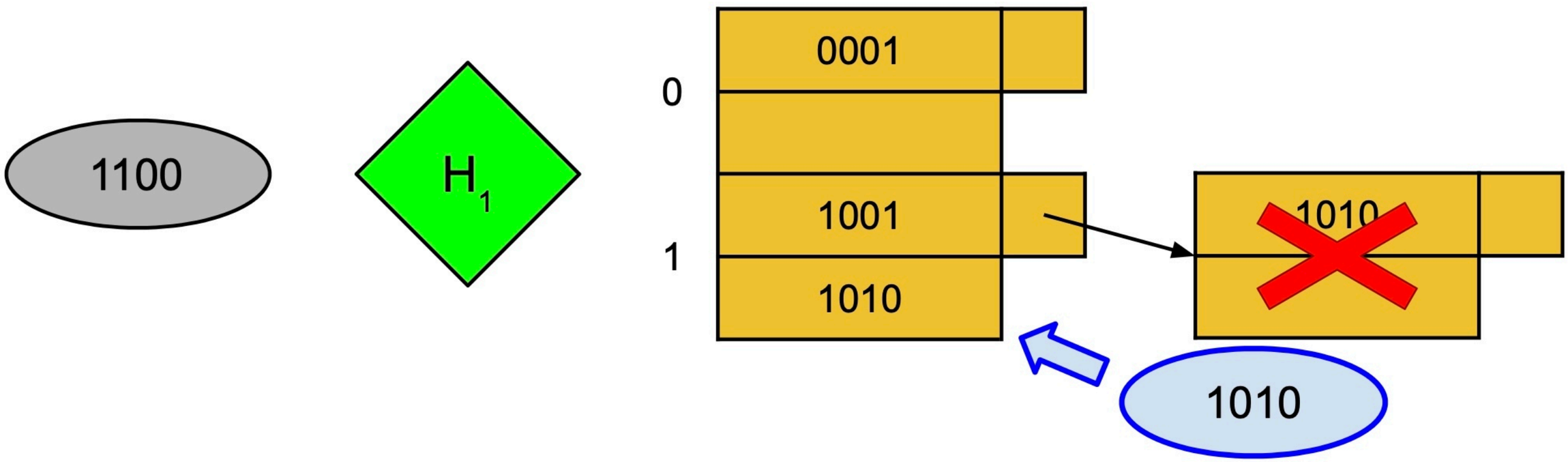
# Static Hashing: Deletion (2)

- Long chains of overflow blocks degrade the performance.
- Deleting keys can lead to the deletion of overflow blocks.



# Static Hashing: Deletion (3)

- Long chains of overflow blocks degrade the performance.
- Deleting keys can lead to the deletion of overflow blocks.



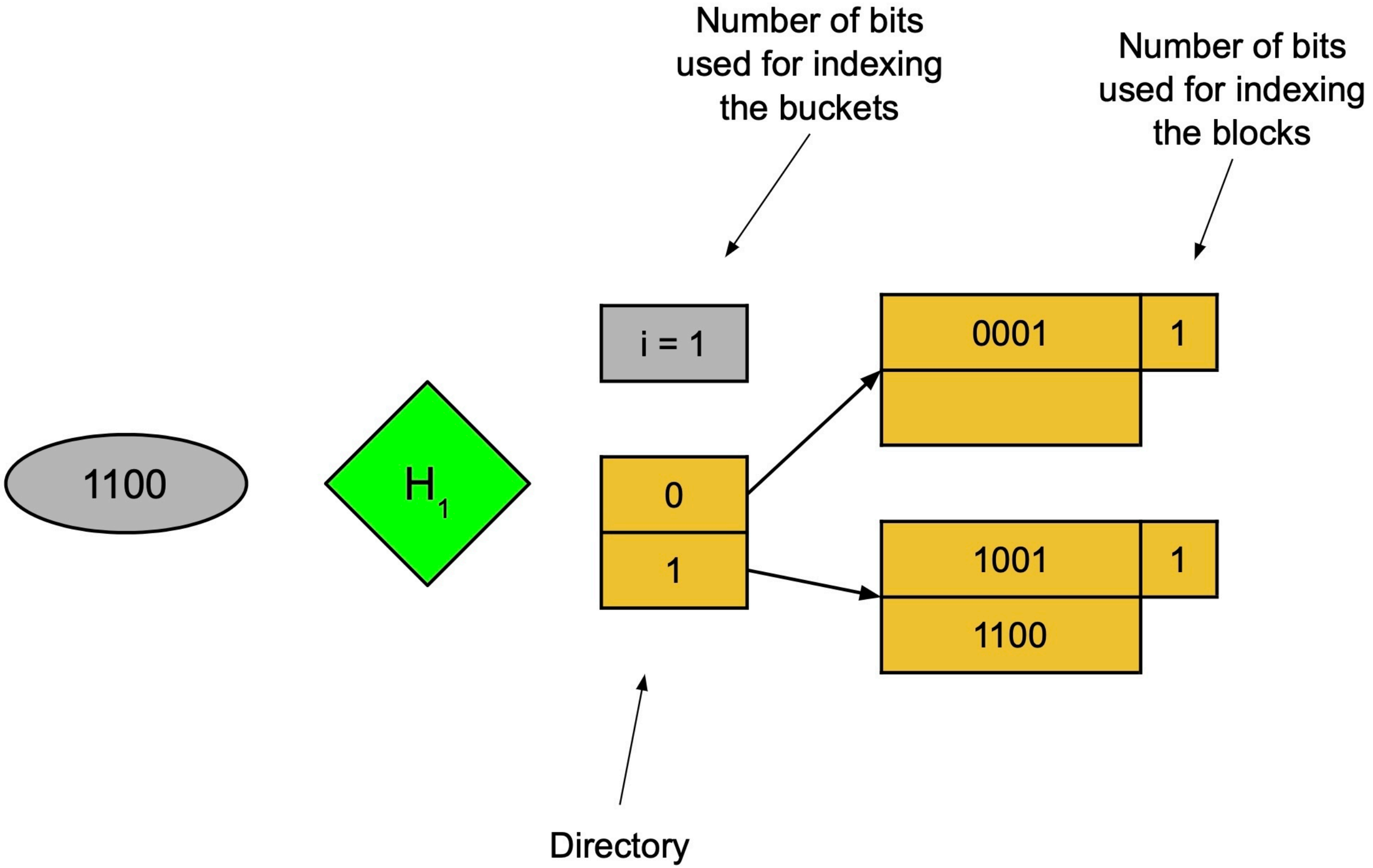
# Efficiency of Static Hashing Indexes

- Ideally, there are enough buckets that most of them fit on one block. If so, the lookup requires only one disk access.
- **Long chains of overflow blocks cause a quick performance degradation** (e.g., taking at least one disk access per block). Efficiency depends on:
  - The ratio between the size of the index and the data, namely the number of buckets.
  - Values distribution of the search key with respect to the hash function.
- There is a good reason to try to keep the number of blocks per bucket low. The **hash function can be dynamically changed** to adapt the number of buckets to the size of the data file.
- We can use the **extendible hashing**, which in case of need allows the doubling of the number of buckets, and the **linear hashing**, which allows the addition of a bucket.

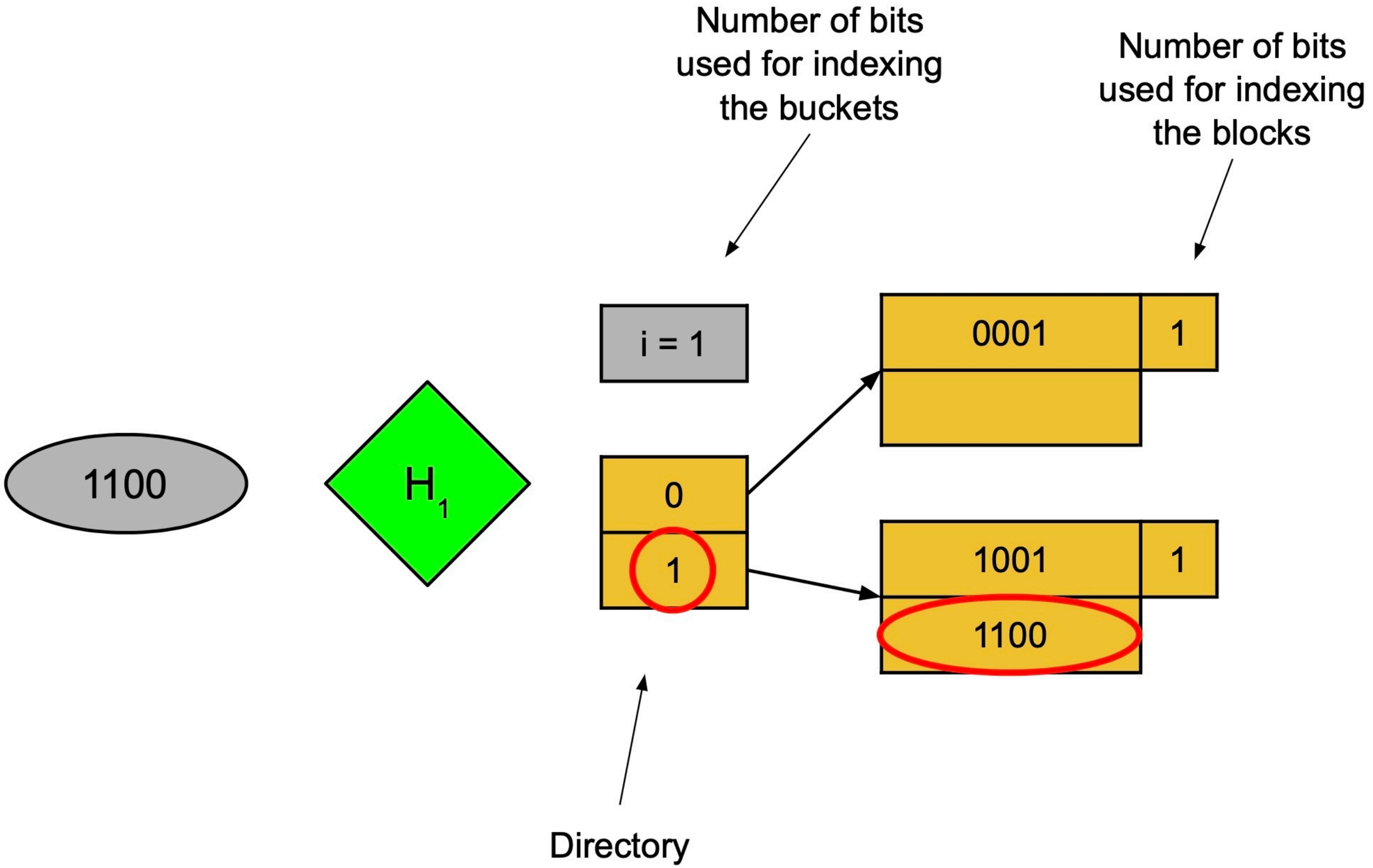
# Extendible Hashing

- There is a level of indirection introduced for the buckets. In particular, extendible hashing uses a **directory of pointers to blocks**.
- The directory of pointers can grow and its length is always a power of 2. By **doubling the directory** (i.e., a growing step), **the number of buckets doubles**.
- There does not have to be a data block for each bucket; certain **buckets can share a block**.
- Extendible hashing does **not use overflow blocks**.
- The hash function returns the ***i* most significant bits** of the binary encoding of the search key.
- Each block has a variable that indicates **how many bits are used to indexing** it.

# Extendible Hashing: Searching (1)



# Extendible Hashing: Searching (2)

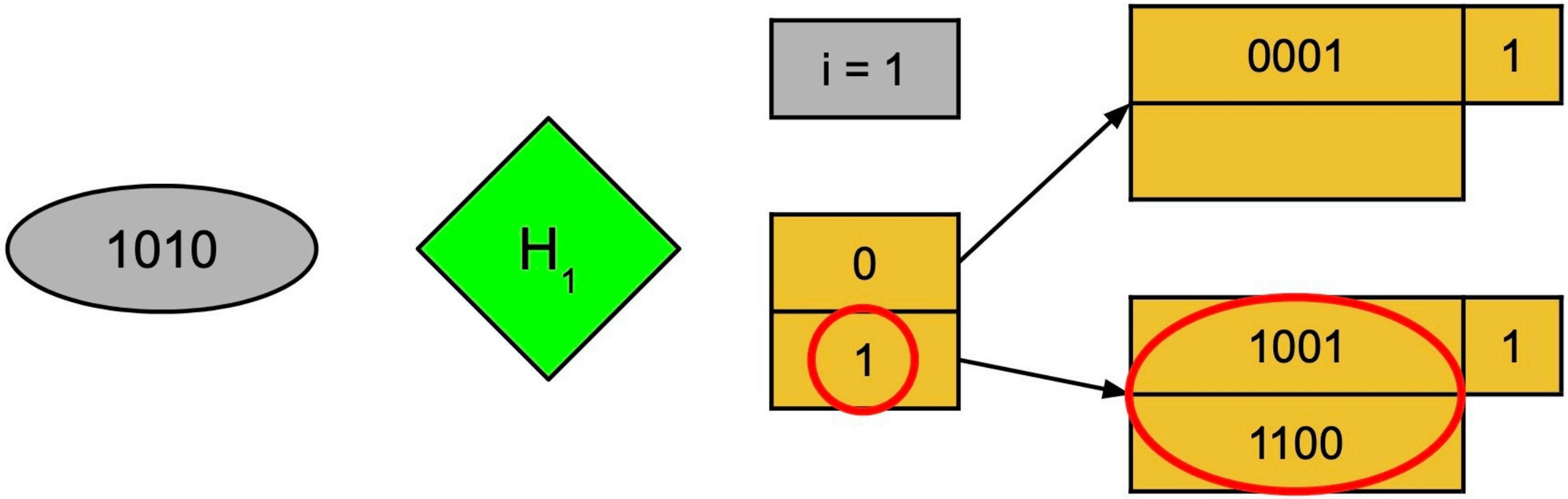


# Extendible Hashing: Insertion Steps

- To insert a data record with search key K, we take the first i bits of $H_i(K)$ bit sequence to identify the bucket in the directory. If there is room in the block, we insert the data record.
- If there is no room, there are two possibilities, depending on the number j, which indicates the current number of bit to indexing the involved block B.
- j < i
  1.1. The block B is split into two.
  1.2. The data records in B are distributed the two new blocks, based on the value of their (j+1) bits.
  1.3. The value of j is increased by 1.
  1.4. The directory is updated with the pointer to the new block.
- j = i
  2.1. The value of i is increased by 1.
  2.2. We double the directory so that an entry indexed by the bit sequence w of i bits produces two entries w0 and w1.
  2.3. Let's proceed as in the previous case.

# Extendible Hashing: Insertion (1)

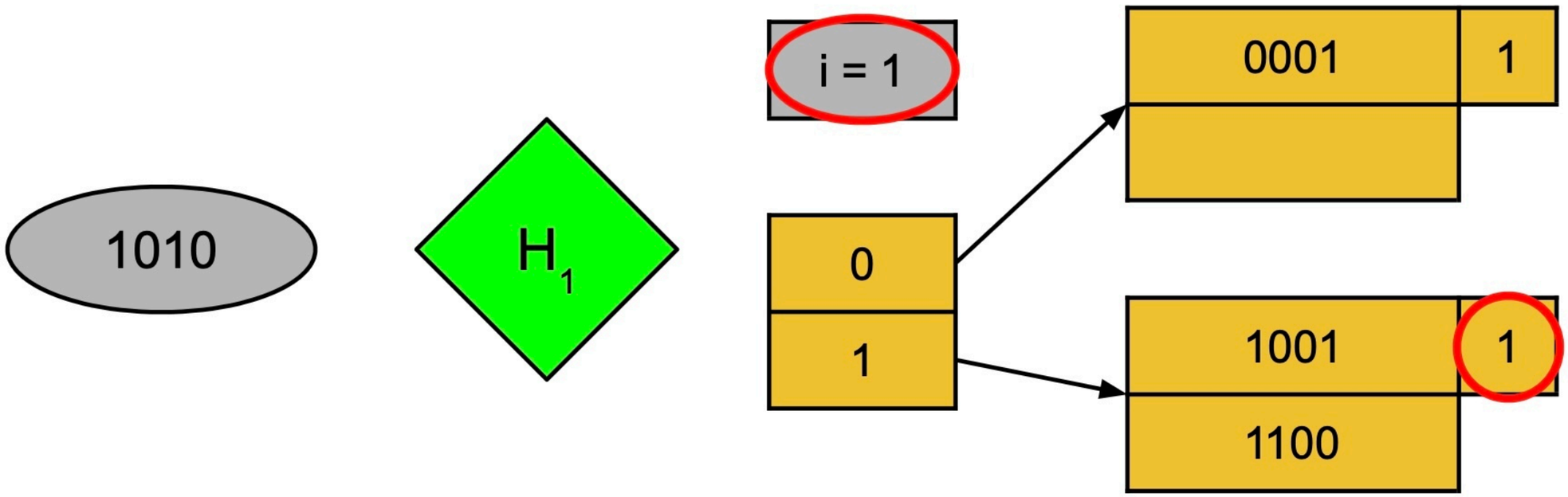
We are looking for the block using the search key $K_2$=1010.




**There is no room!**

# Extendible Hashing: Insertion (2)

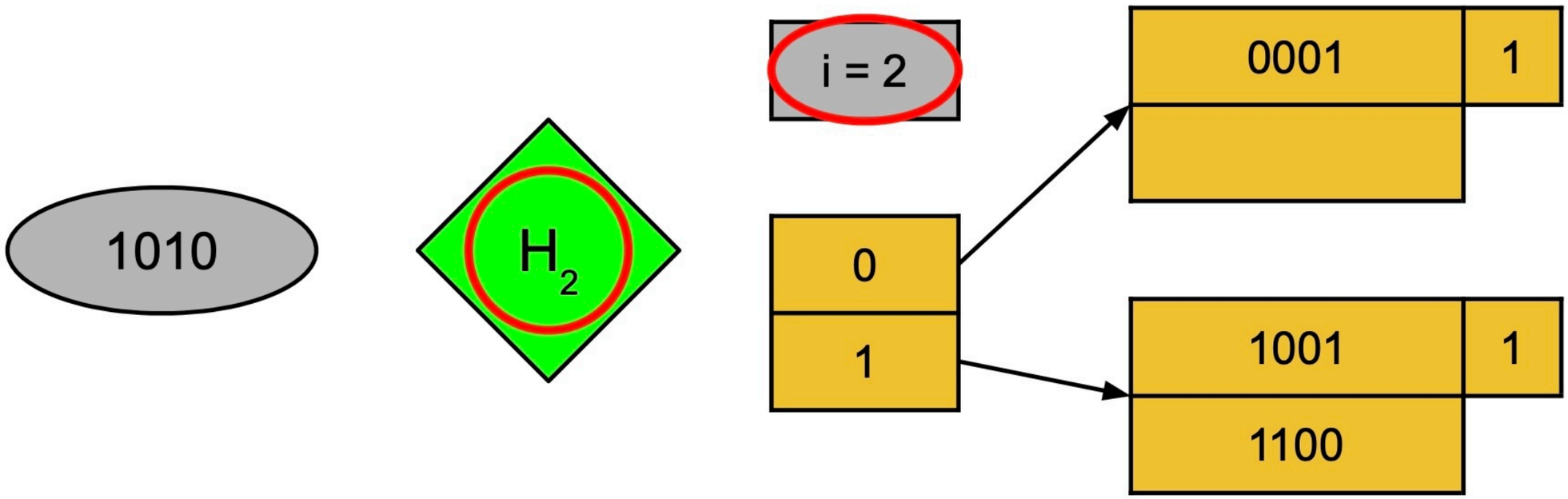
Let's check the values of i and j.




**j = i**

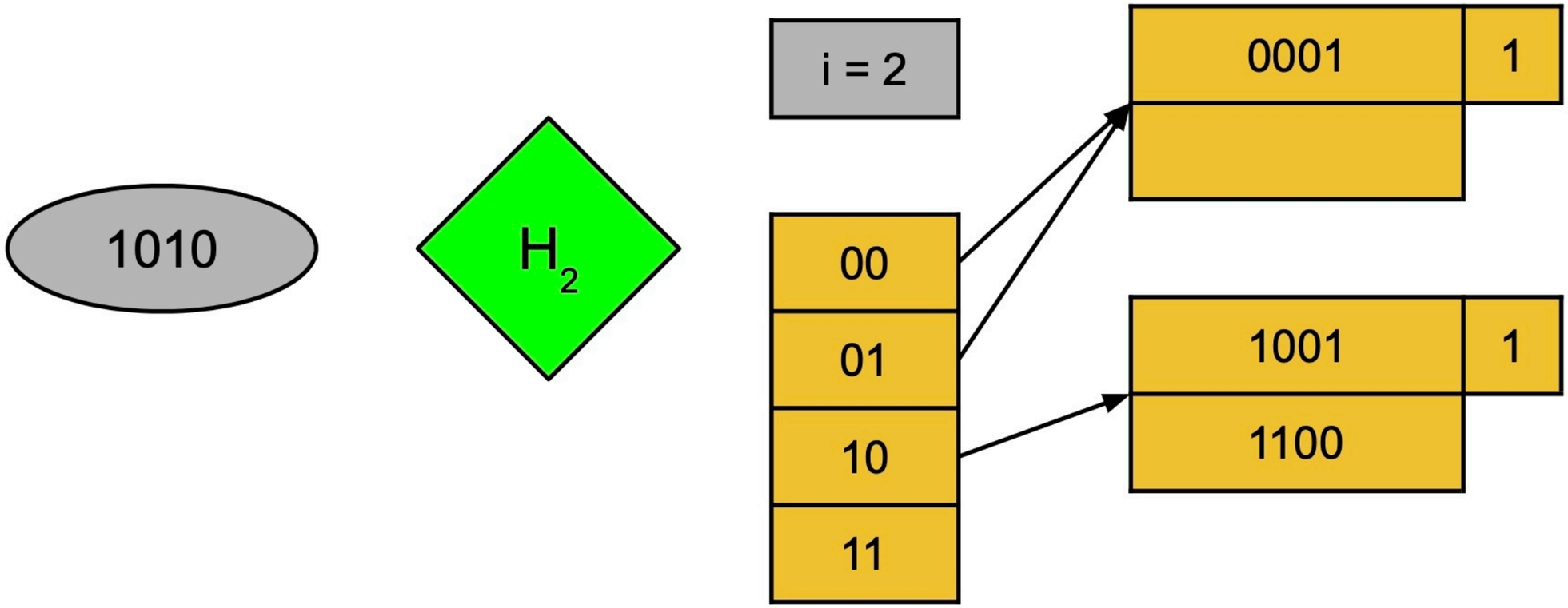
# Extendible Hashing: Insertion (3)

2.1. The value of i is increased by 1.



# Extendible Hashing: Insertion (4)

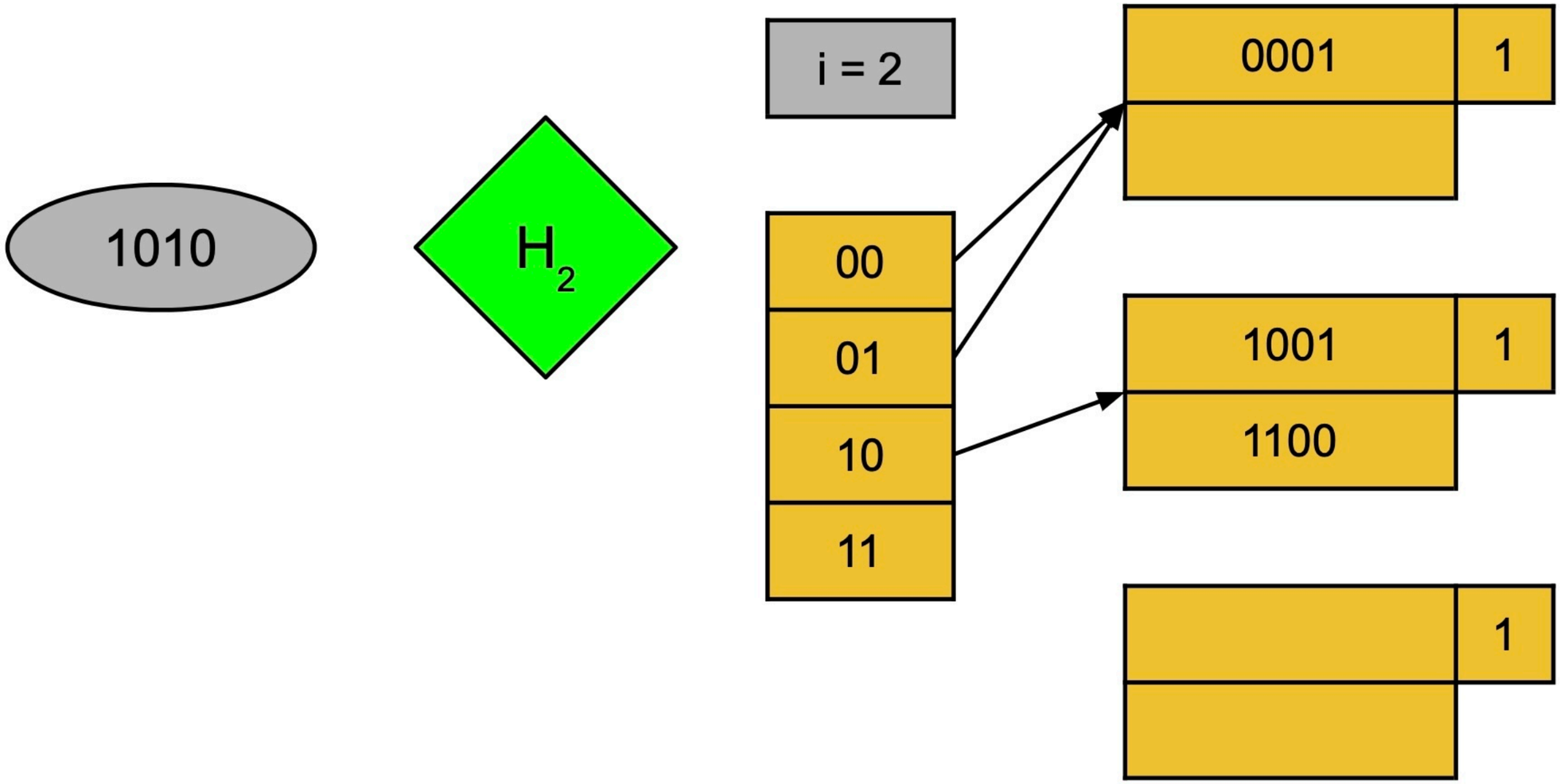
2.2. We double the directory so that an entry indexed by the bit sequence w of i bits produces two entries w0 and w1.




**j < i**

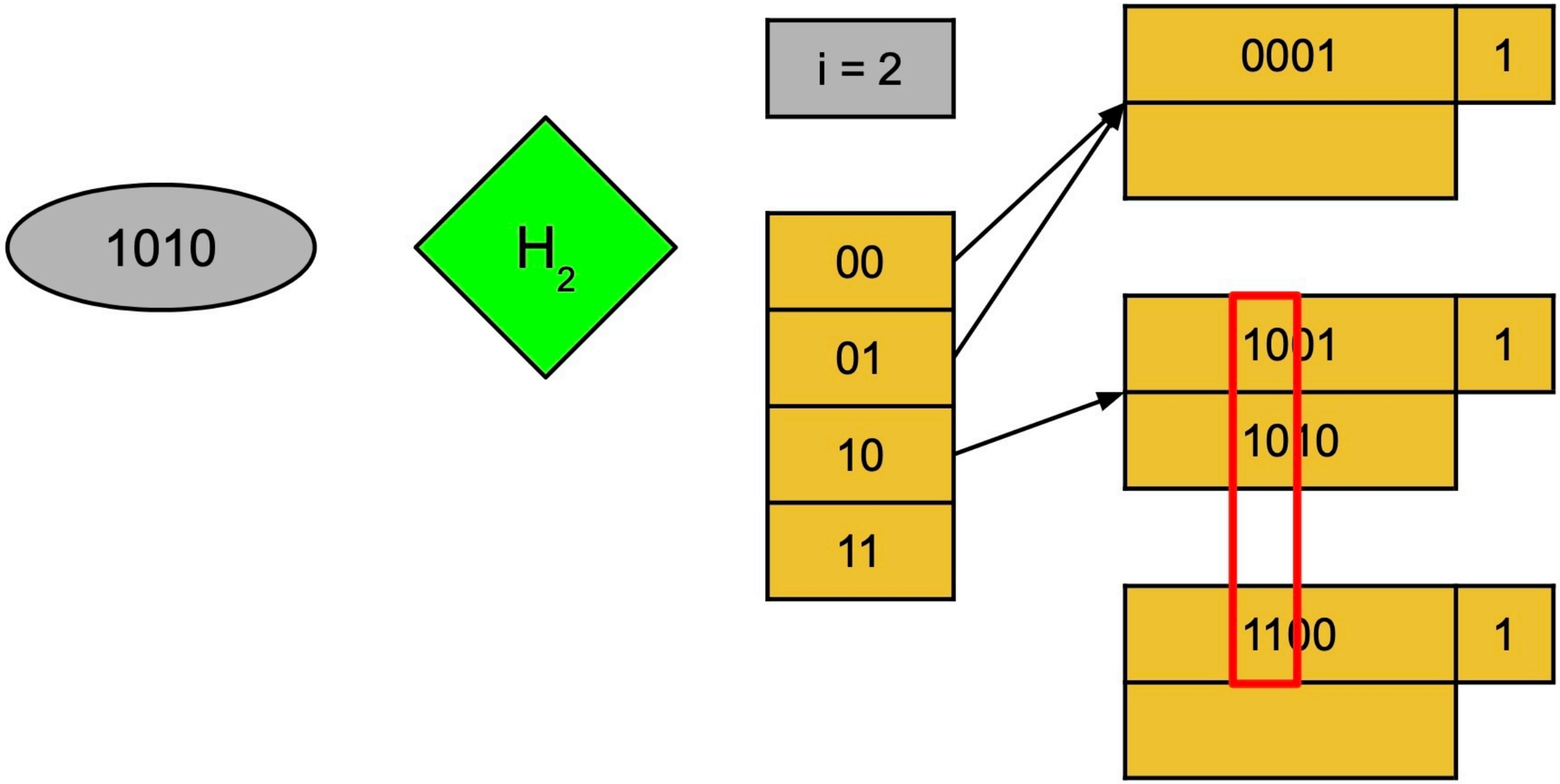
# Extendible Hashing: Insertion (5)

1.1. The block B is split into two.



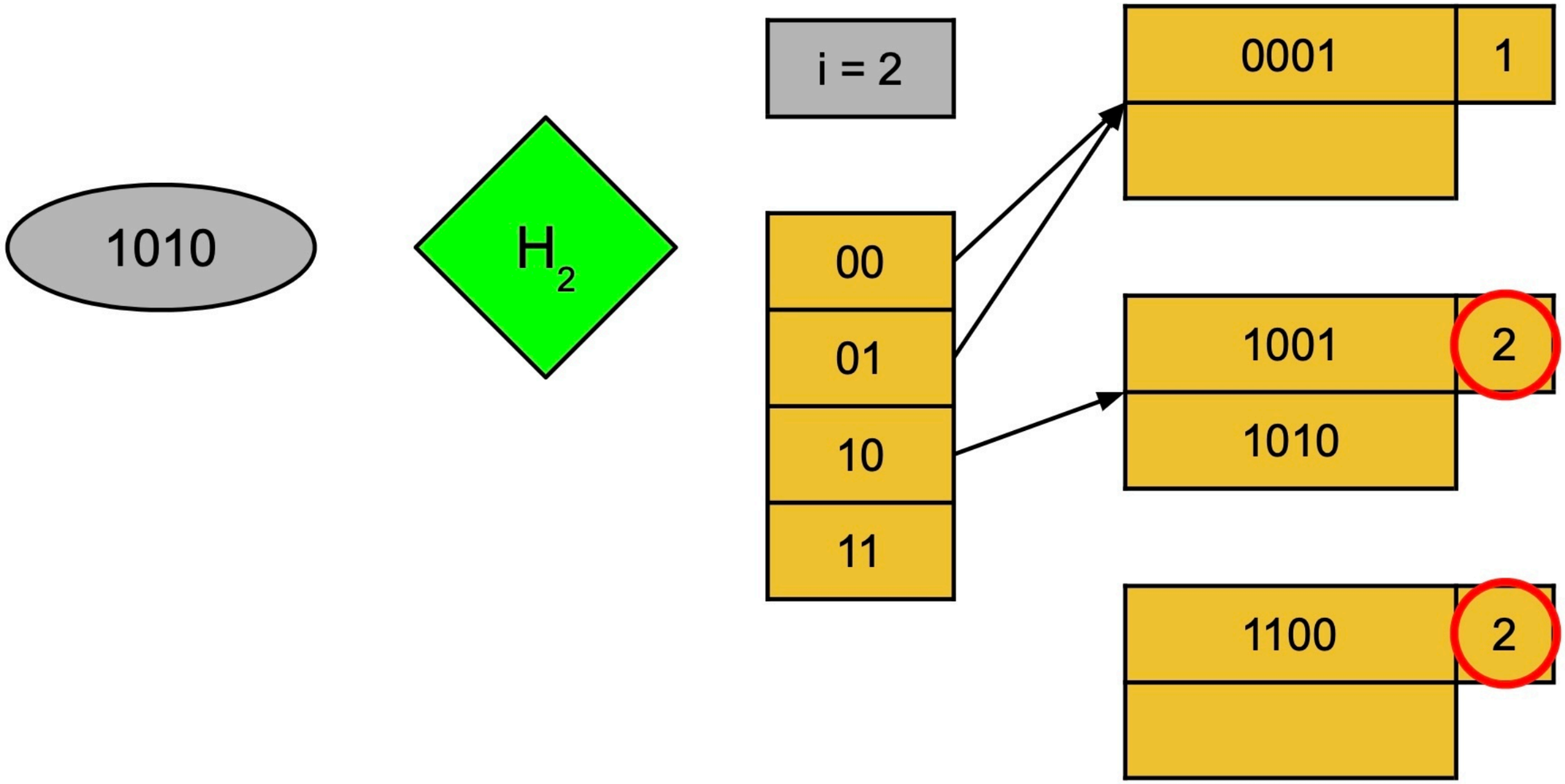
# Extendible Hashing: Insertion (6)

1.2. The data records in B are distributed using (j+1) bits.



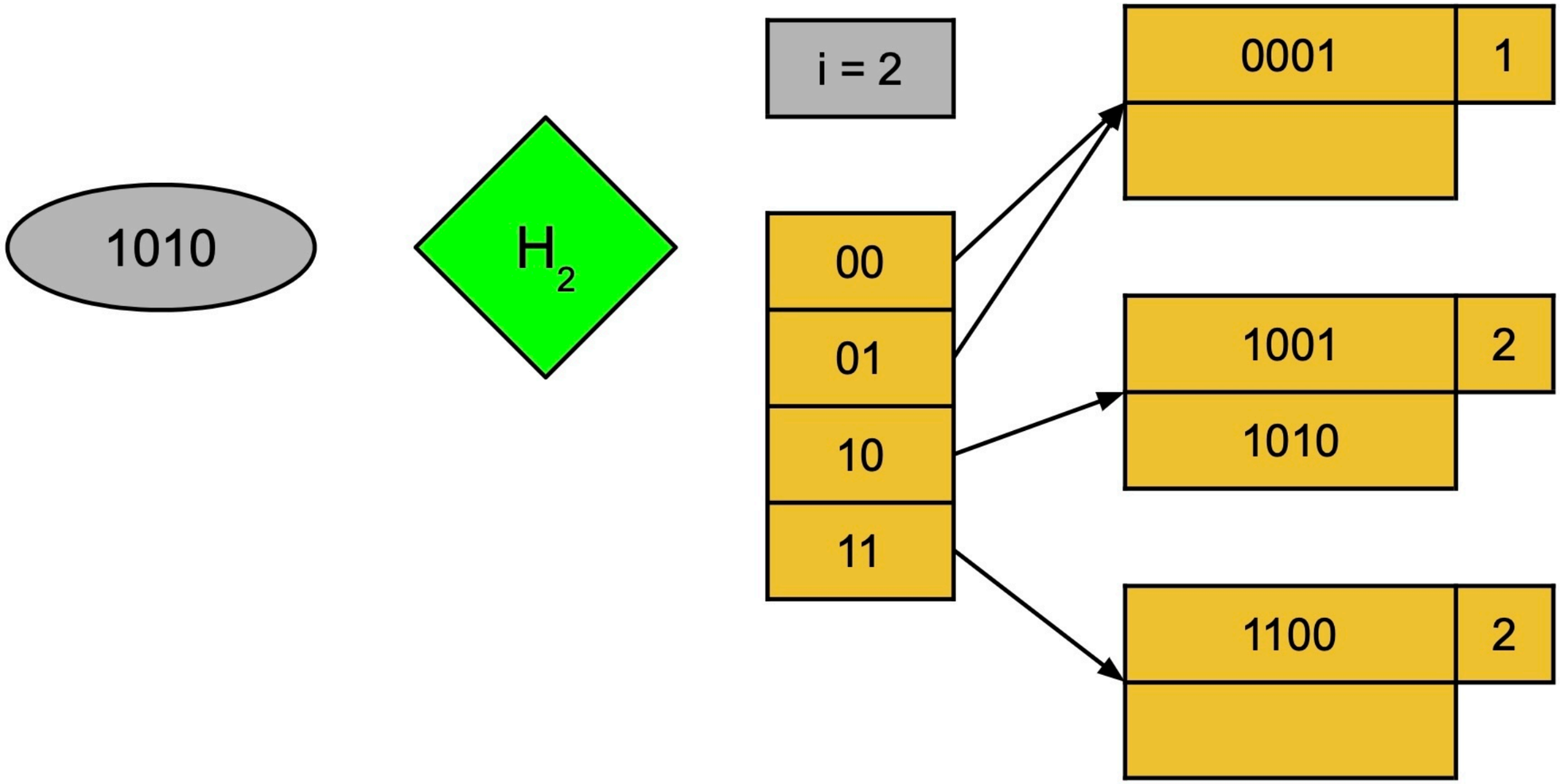
# Extendible Hashing: Insertion (7)

1.3. The value of j is increased by 1.



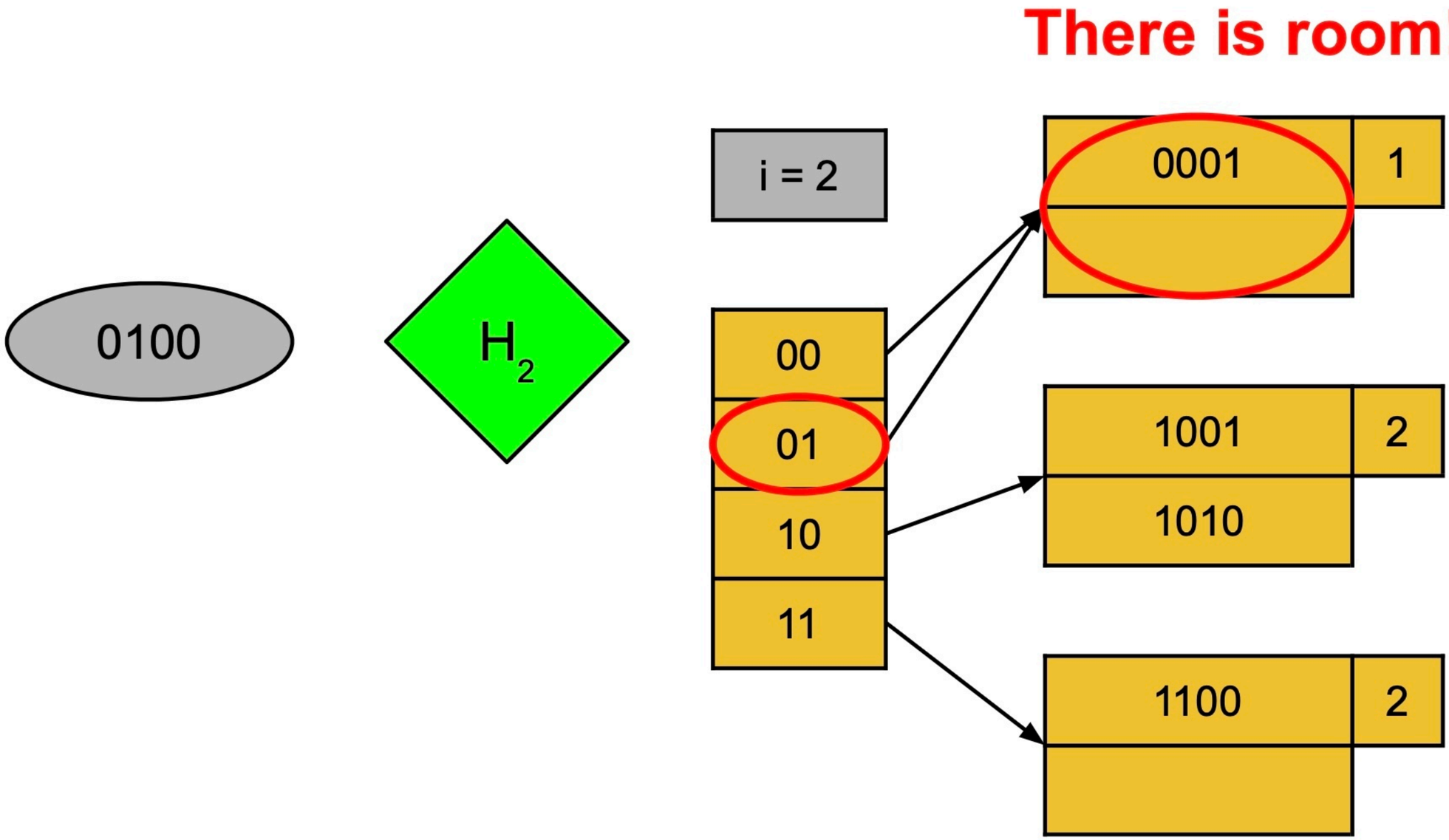
# Extendible Hashing: Insertion (8)

1.4. The directory is updated with the pointer to the new block.



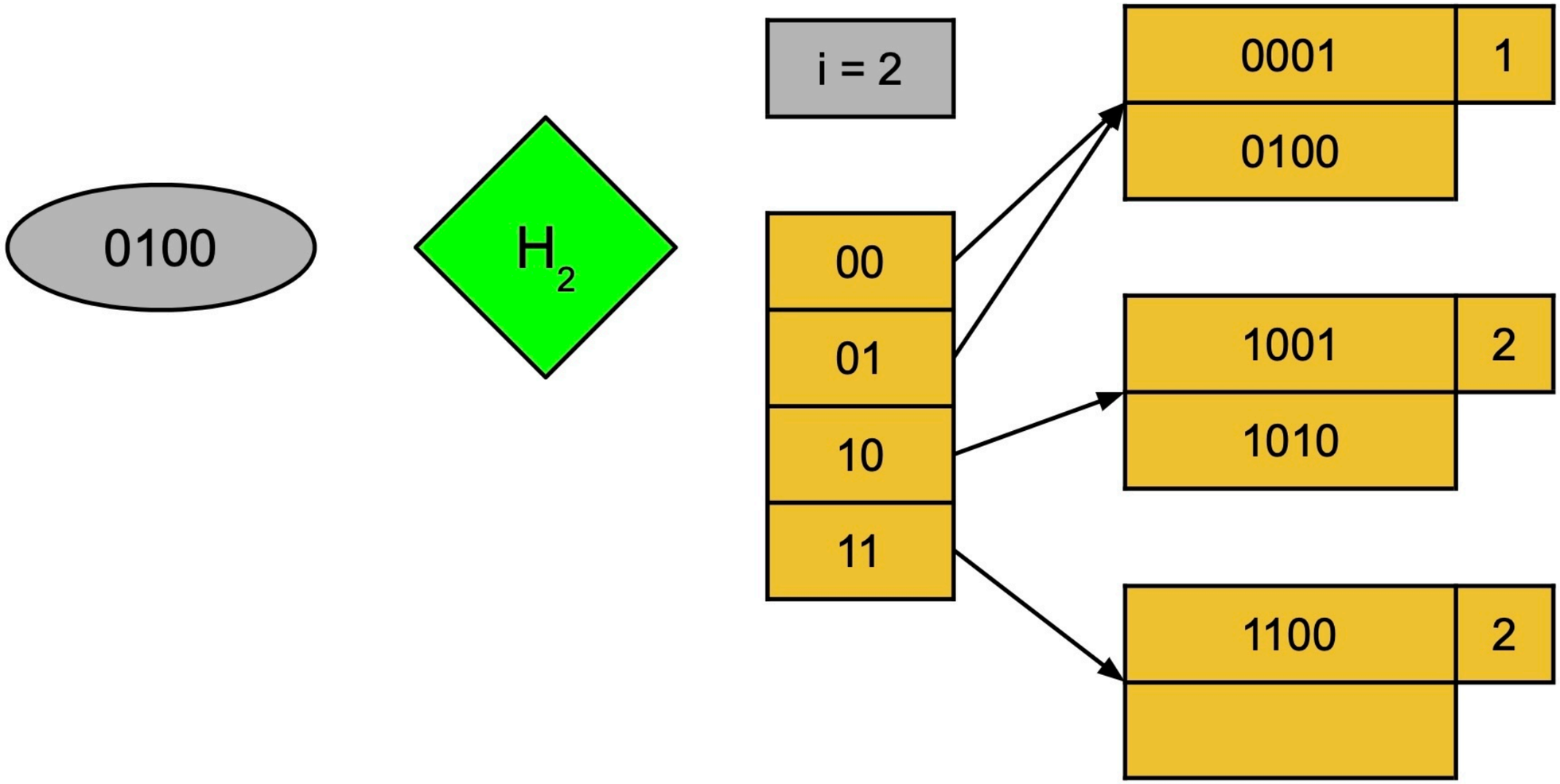
# Extendible Hashing: Insertion (9)

We are looking for the block using the search key $K_2$=0100.



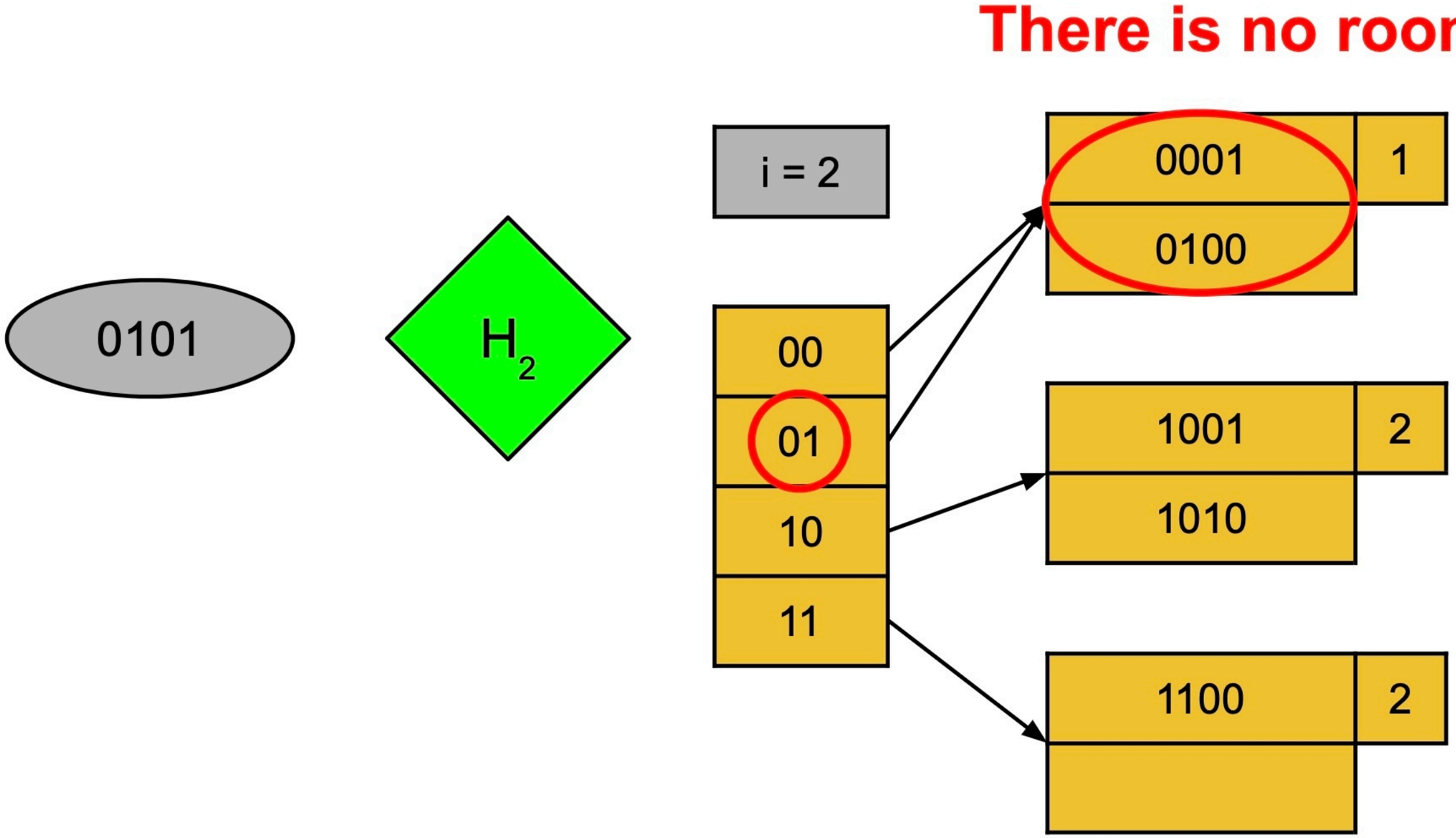
# Extendible Hashing: Insertion (10)

We are looking for the block using the search key $K_2$=0100.



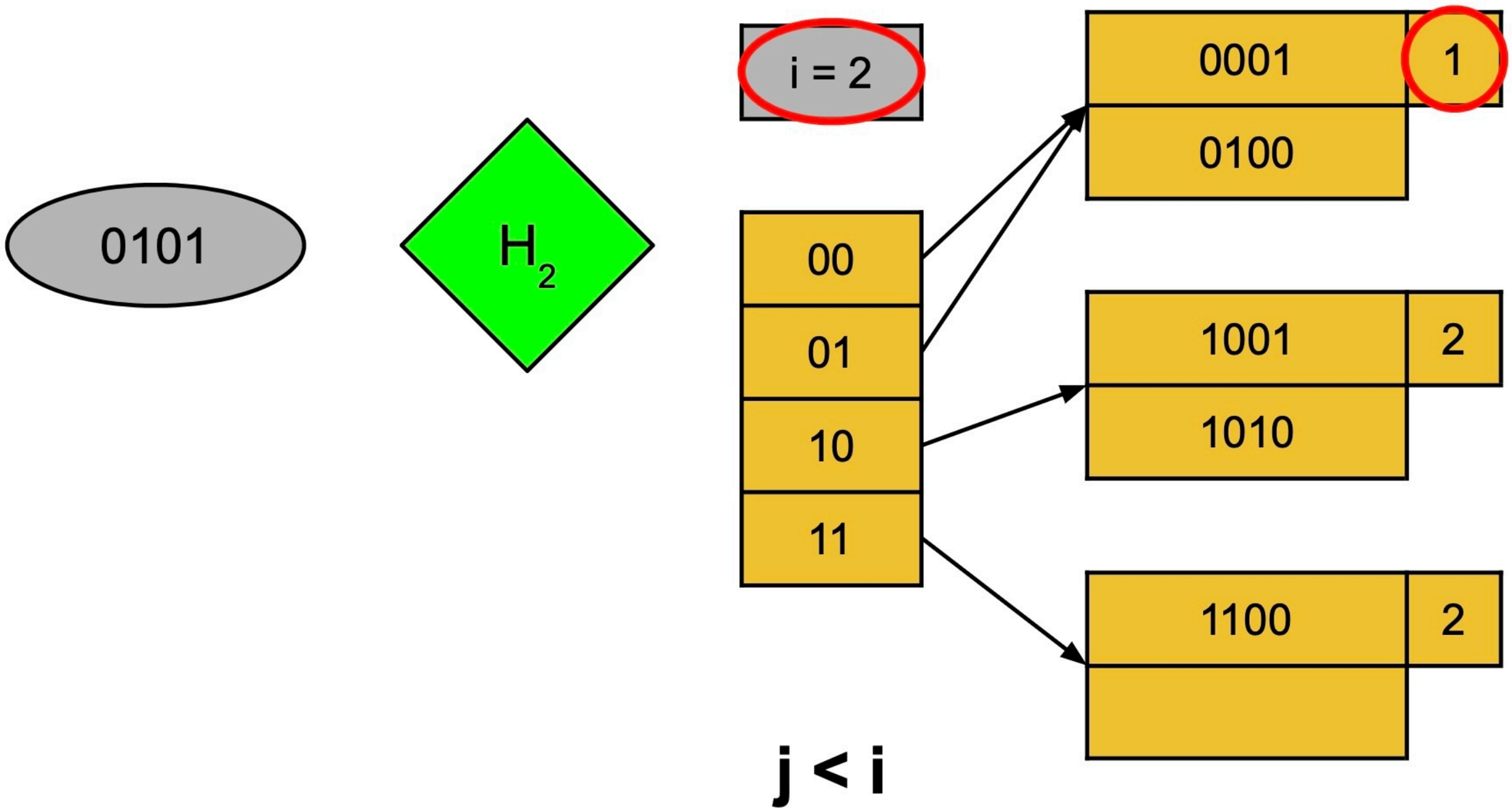
# Extendible Hashing: Insertion (11)

We are looking for the block using the search key $K_2$=0101.



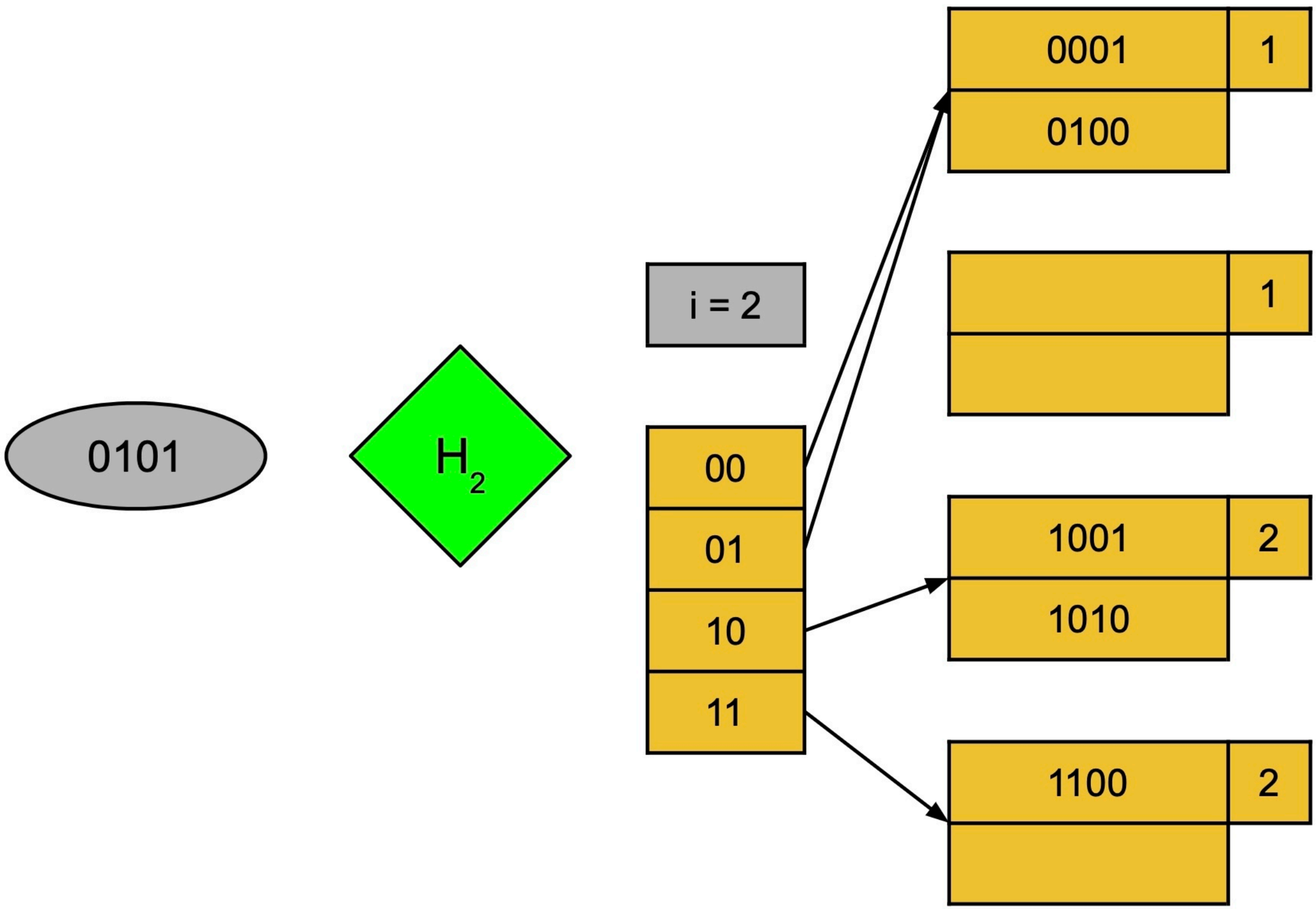
# Extendible Hashing: Insertion (12)

Let's check the values of i and j.



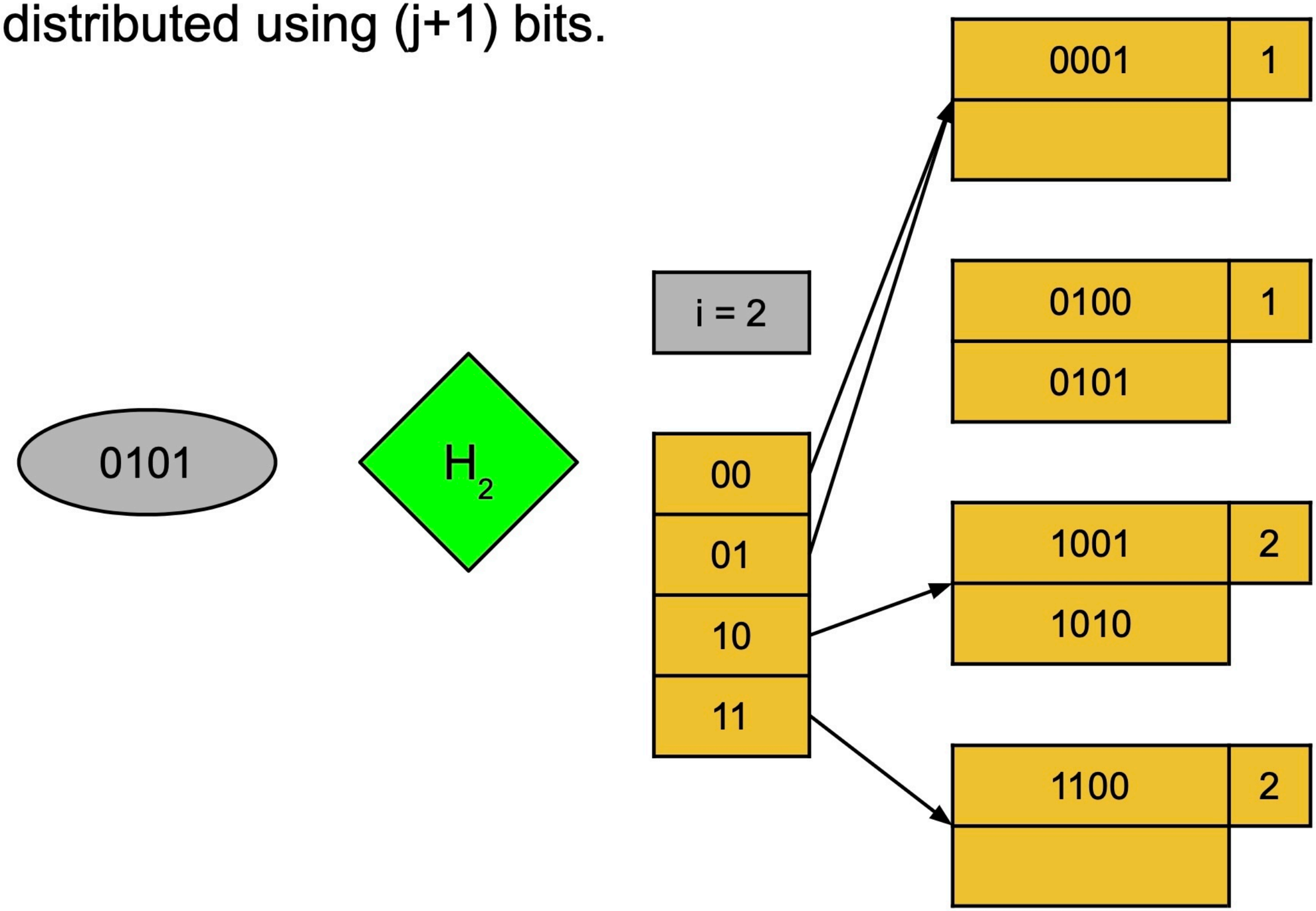
# Extendible Hashing: Insertion (13)

1.1. The block B is split into two.



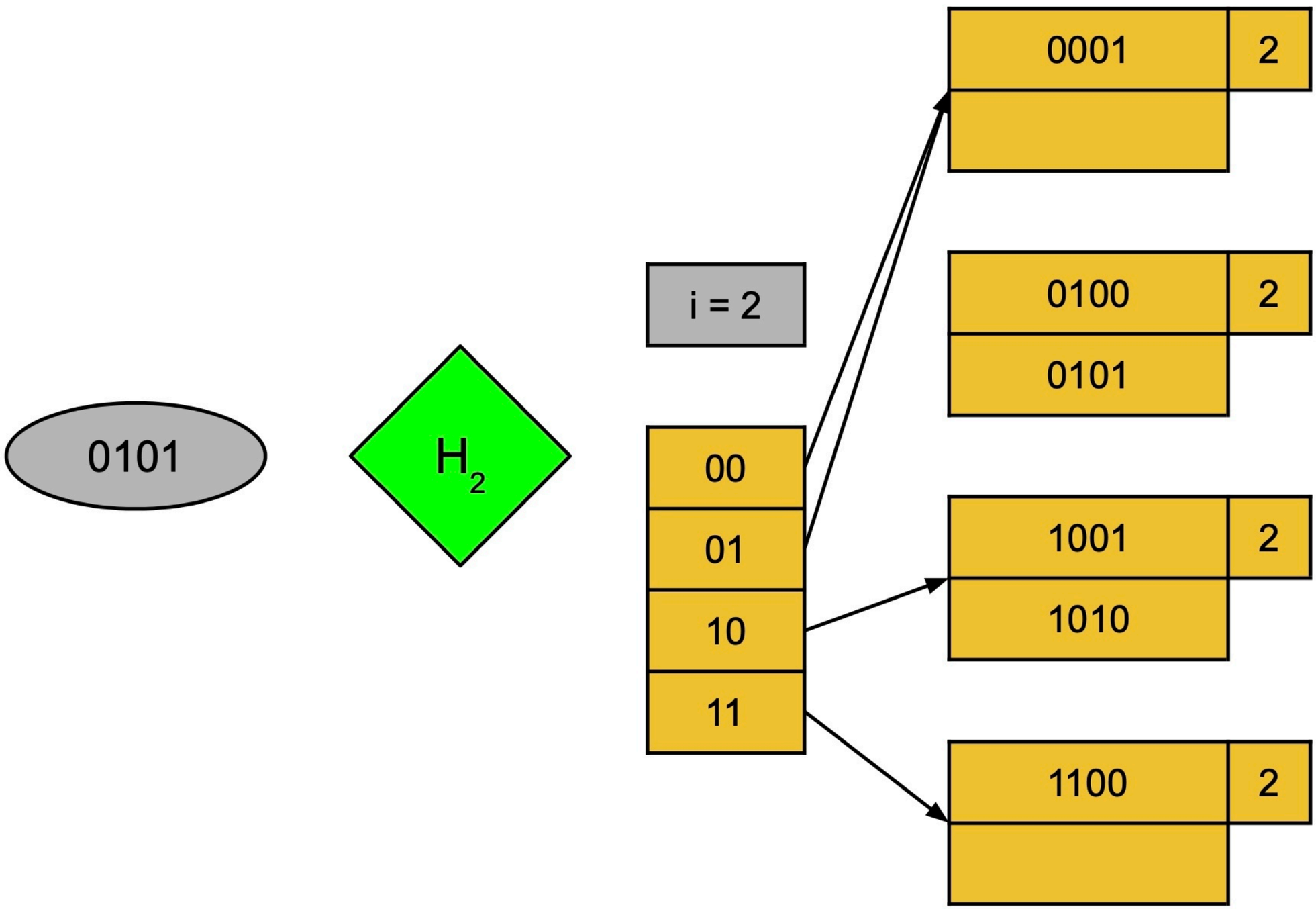
# Extendible Hashing: Insertion (14)

1.2. The data records in B are distributed using (j+1) bits.



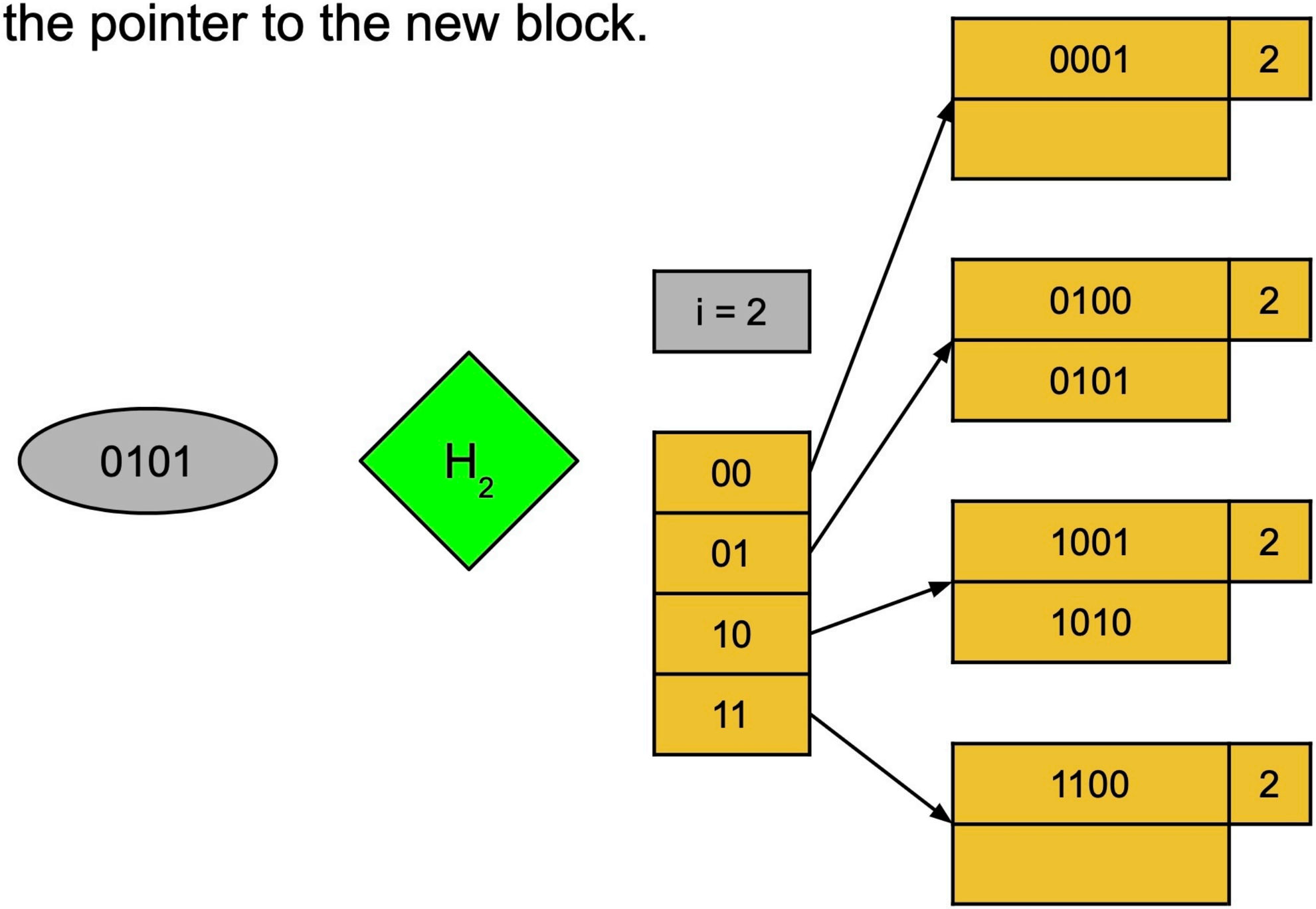
# Extendible Hashing: Insertion (15)

1.3. The value of j is increased by 1.



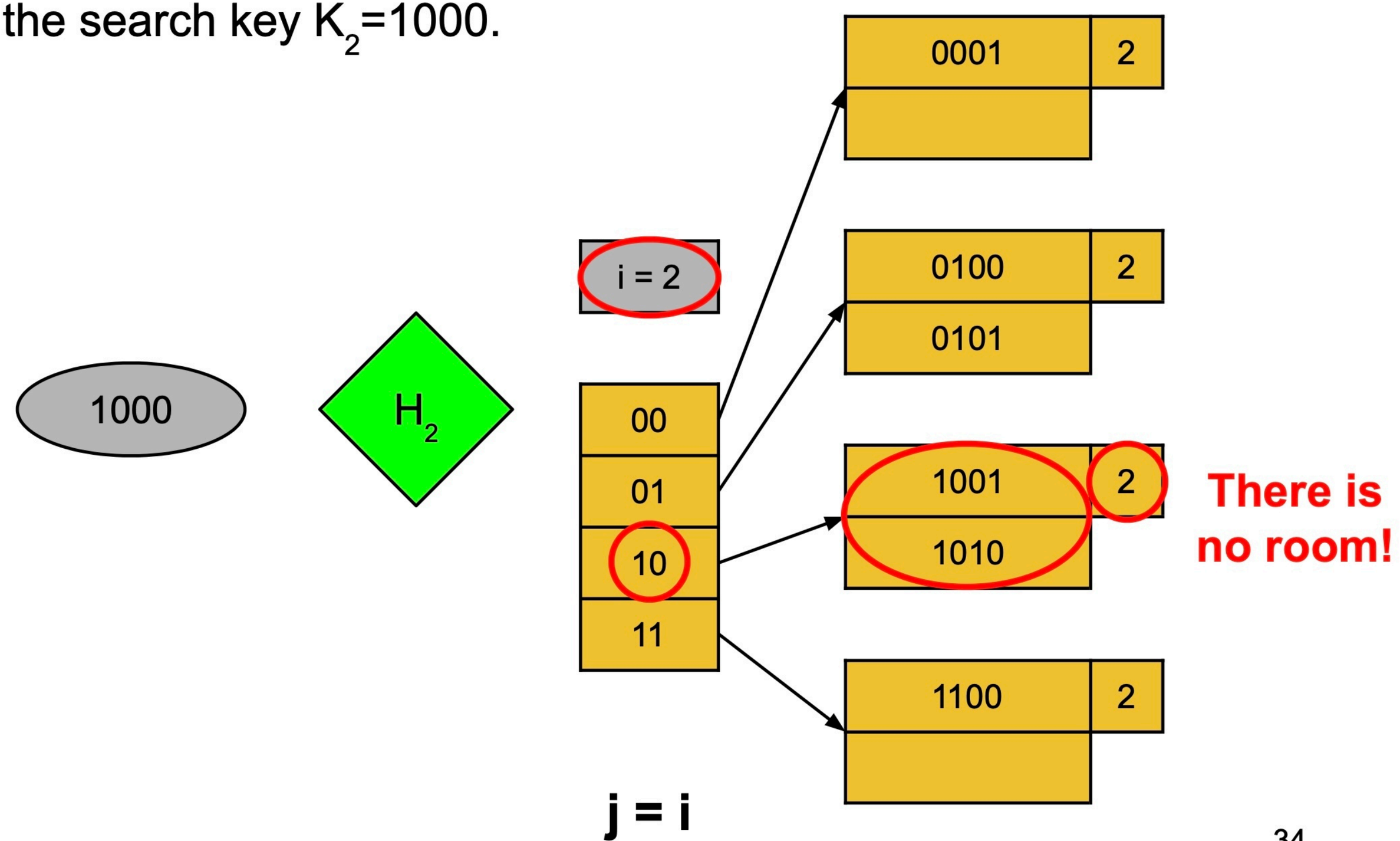
# Extendible Hashing: Insertion (16)

1.4. The directory is updated with the pointer to the new block.



# Extendible Hashing: Insertion (17)

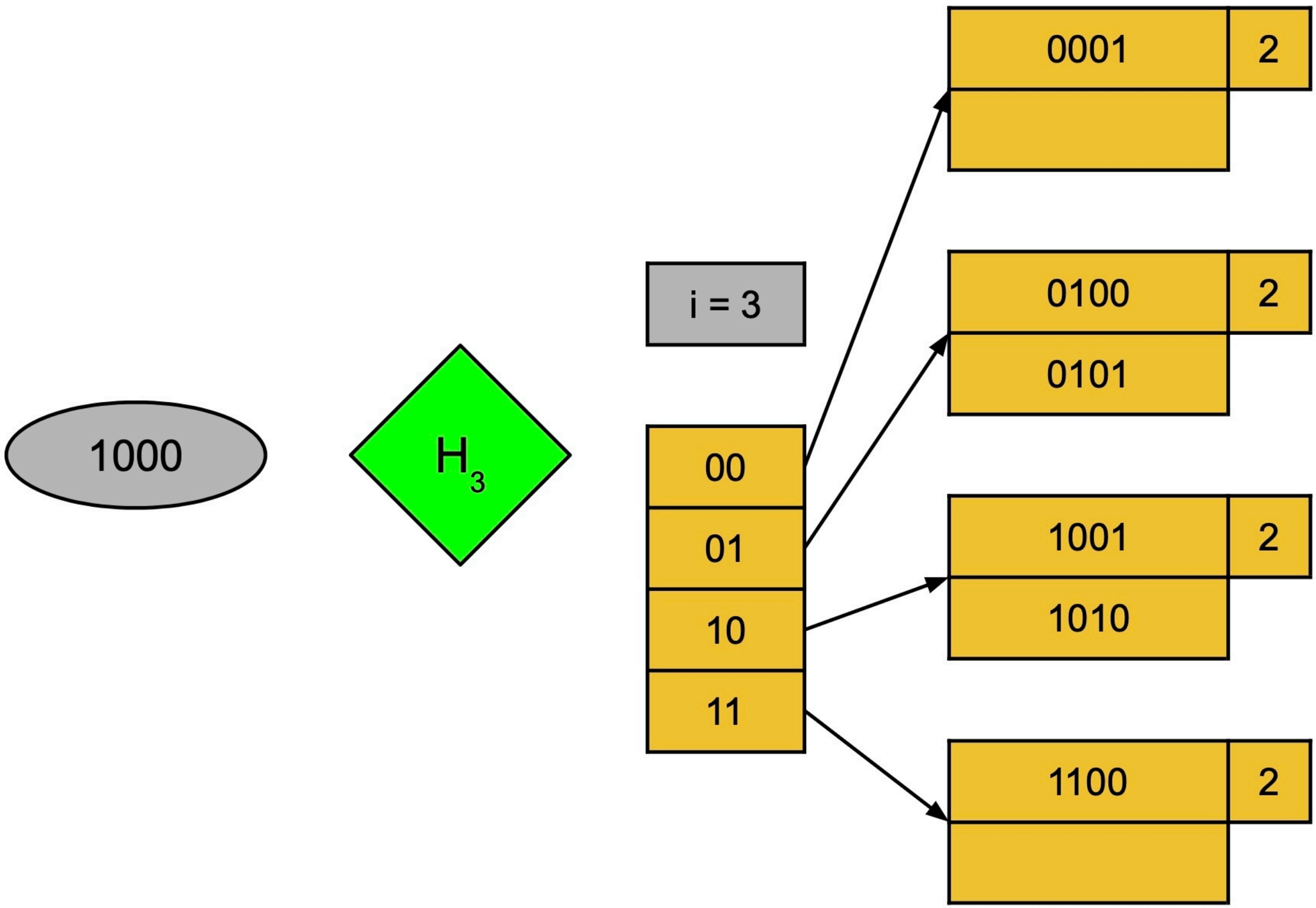
We are looking for the block using the search key $K_2$=1000.




There is no room!

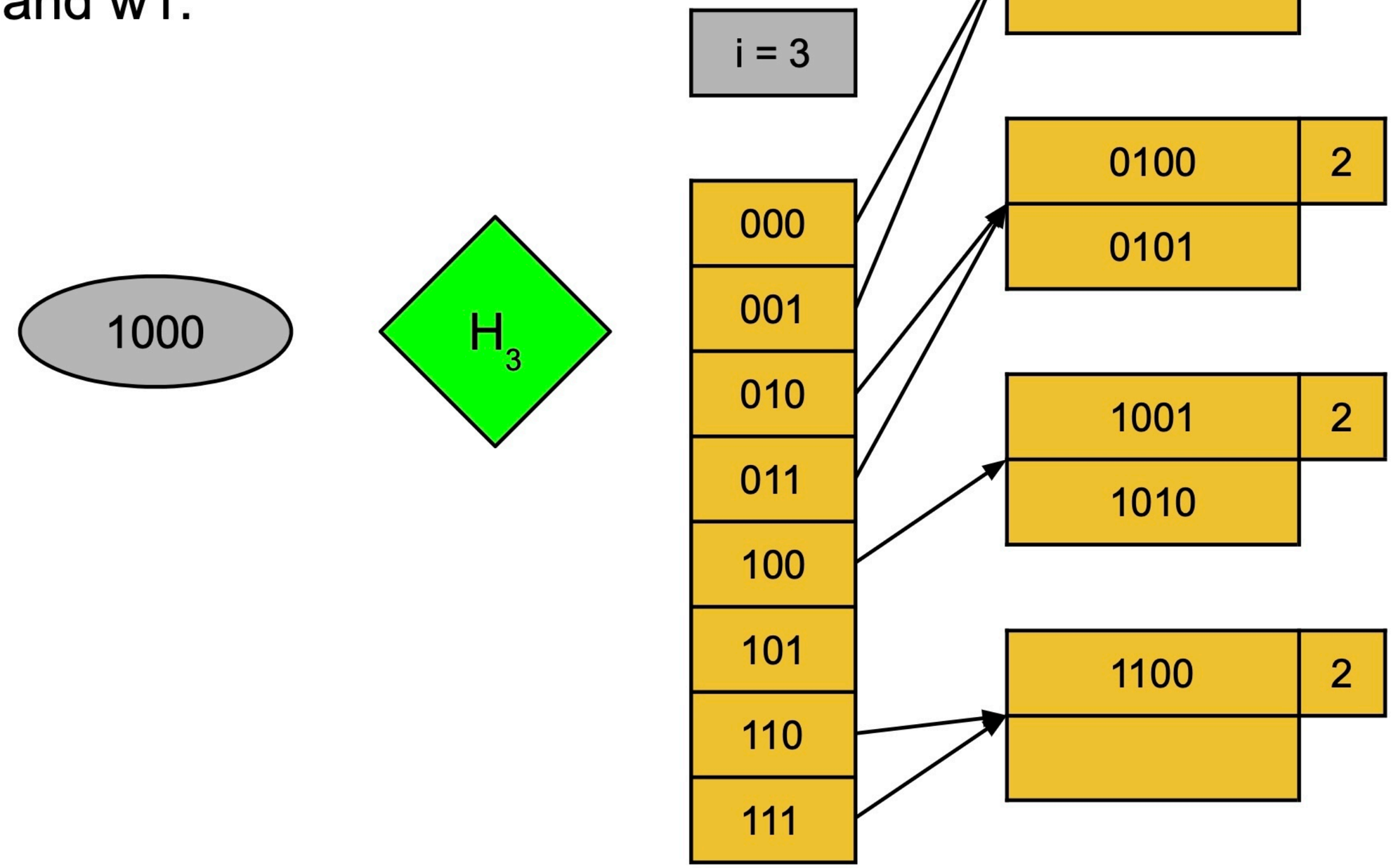
# Extendible Hashing: Insertion (18)

2.1. The value of i is increased by 1.



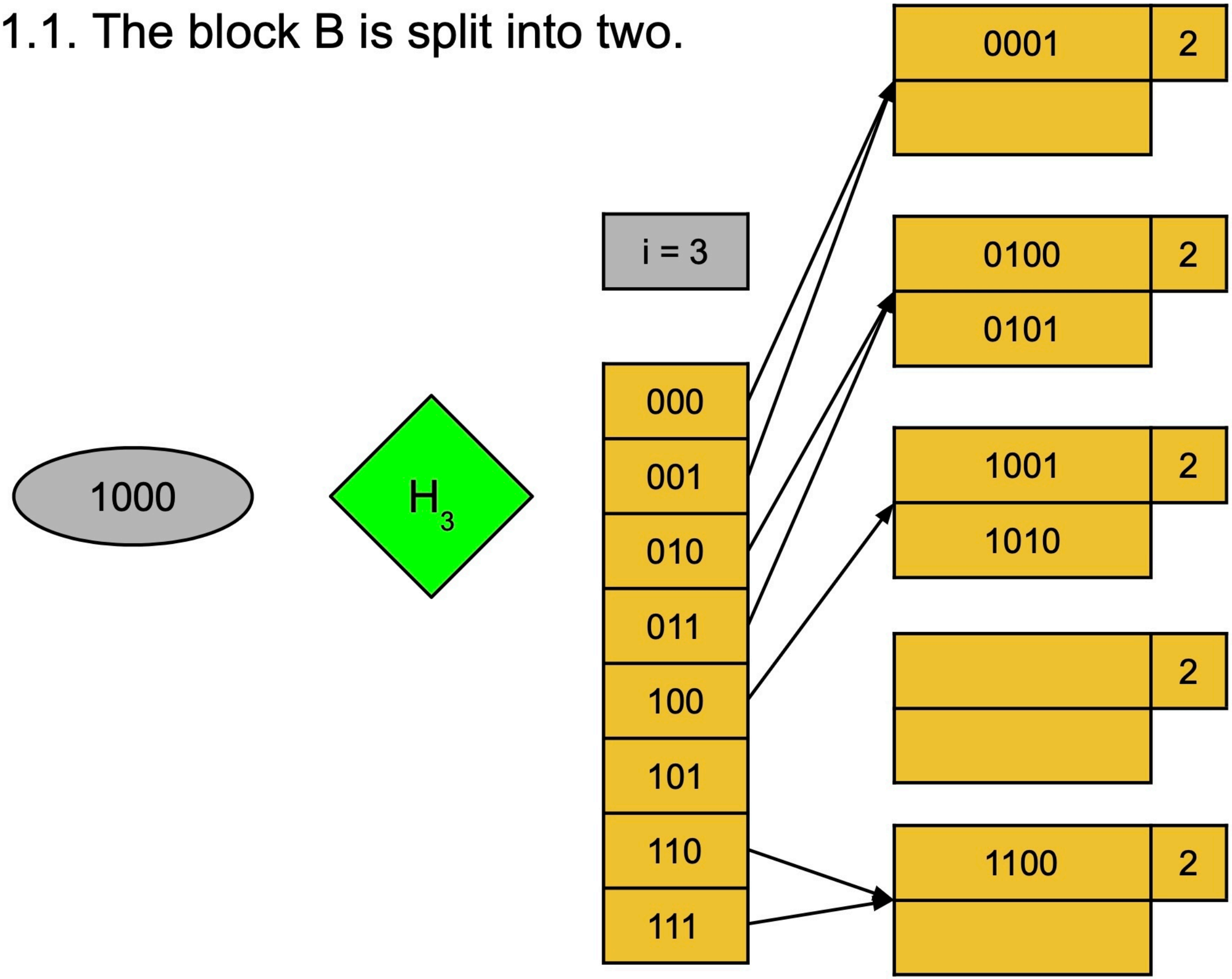
# Extendible Hashing: Insertion (19)

2.2. We double the directory so that an entry indexed by the bit sequence w of i bits produces two entries w0 and w1.



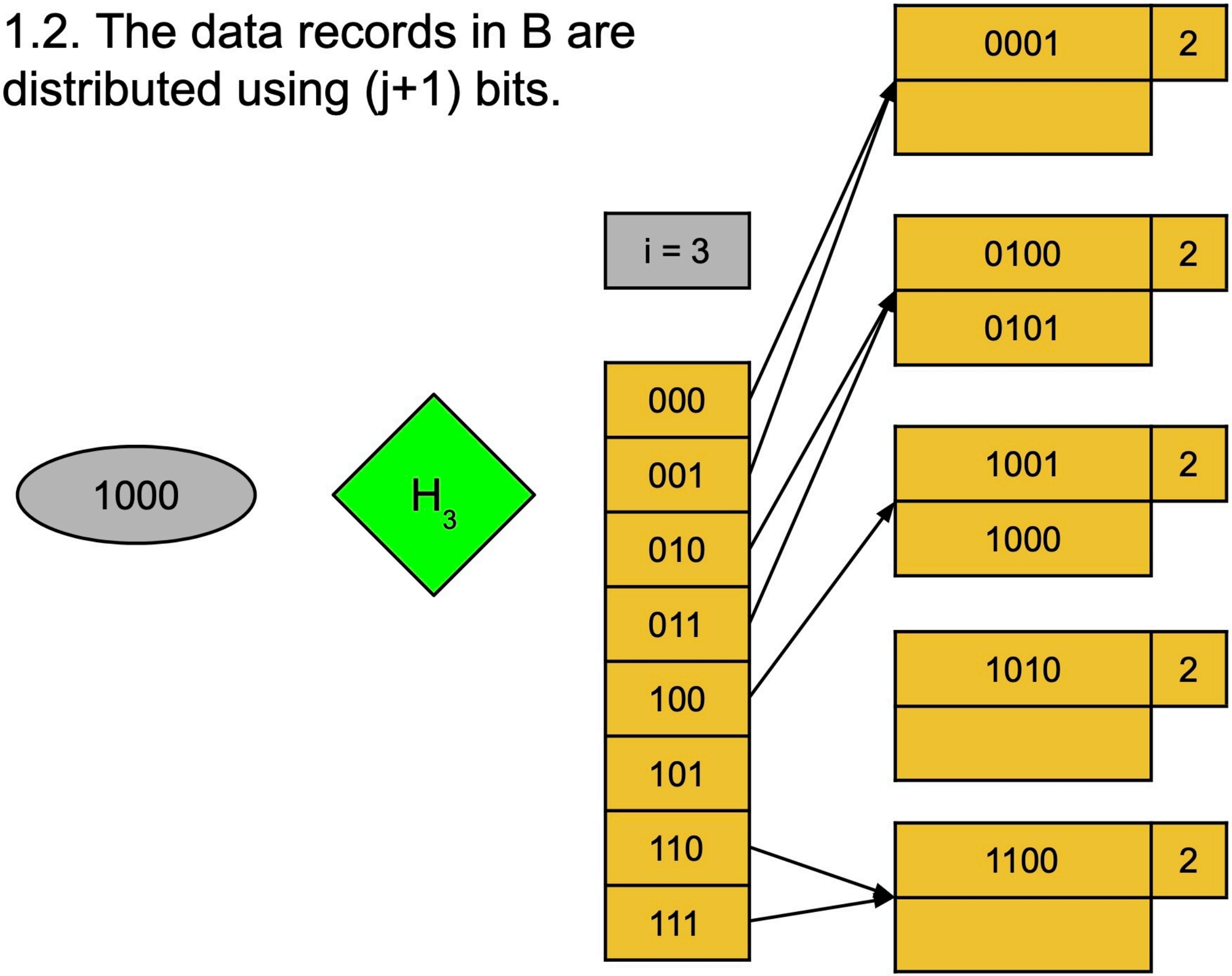
# Extendible Hashing: Insertion (20)

1.1. The block B is split into two.



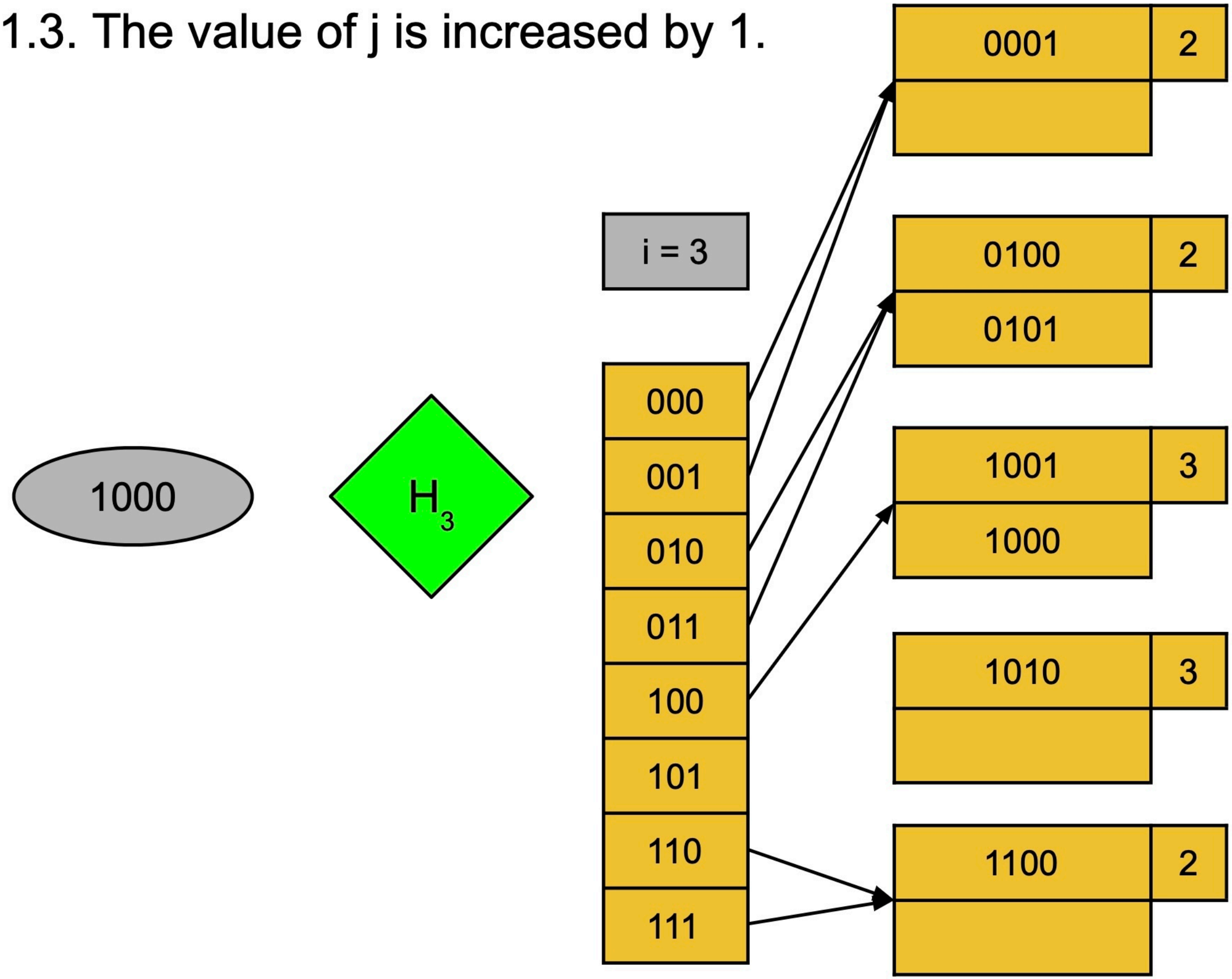
# Extendible Hashing: Insertion (21)

1.2. The data records in B are distributed using (j+1) bits.



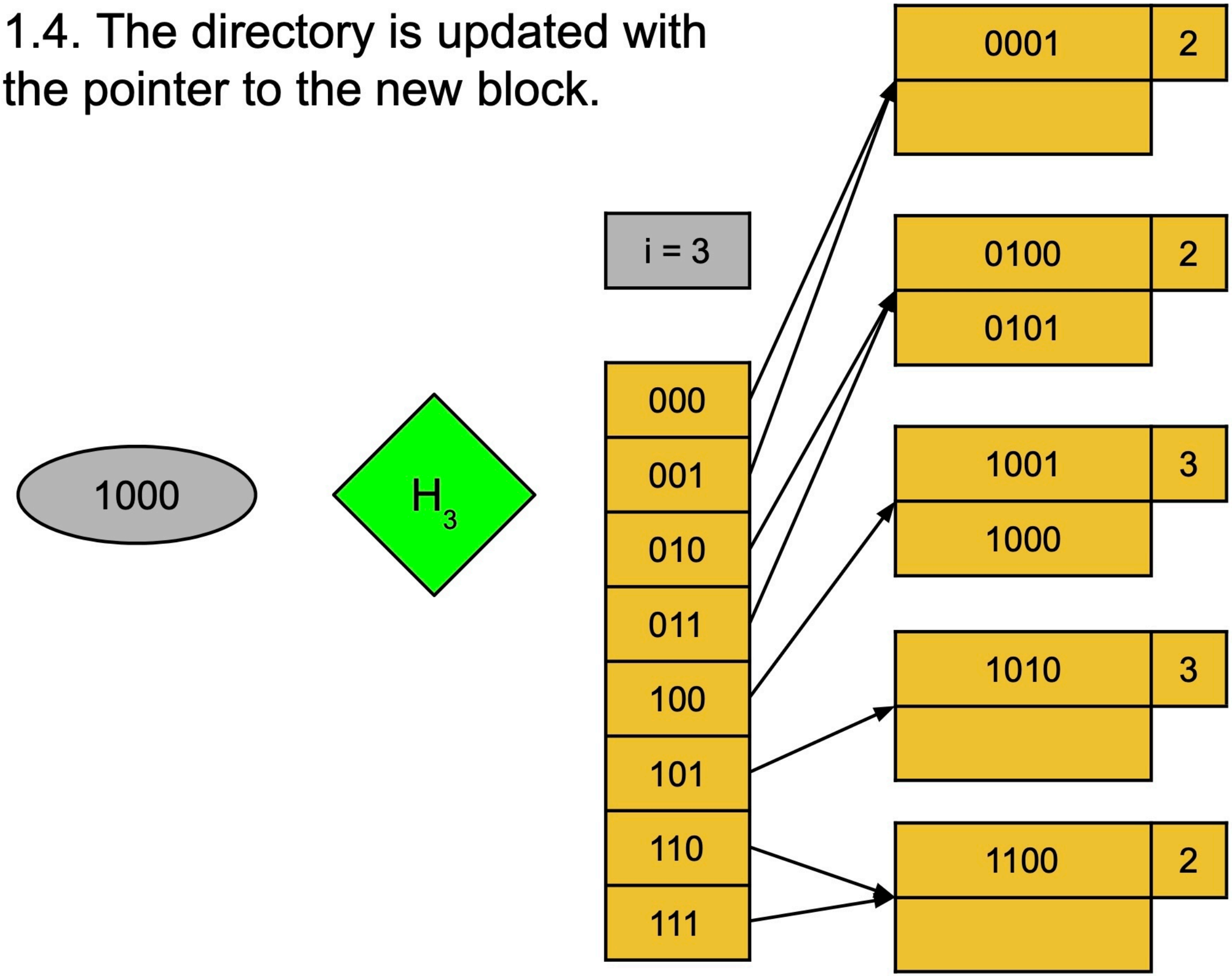
# Extendible Hashing: Insertion (22)

1.3. The value of j is increased by 1.



# Extendible Hashing: Insertion (23)

1.4. The directory is updated with the pointer to the new block.



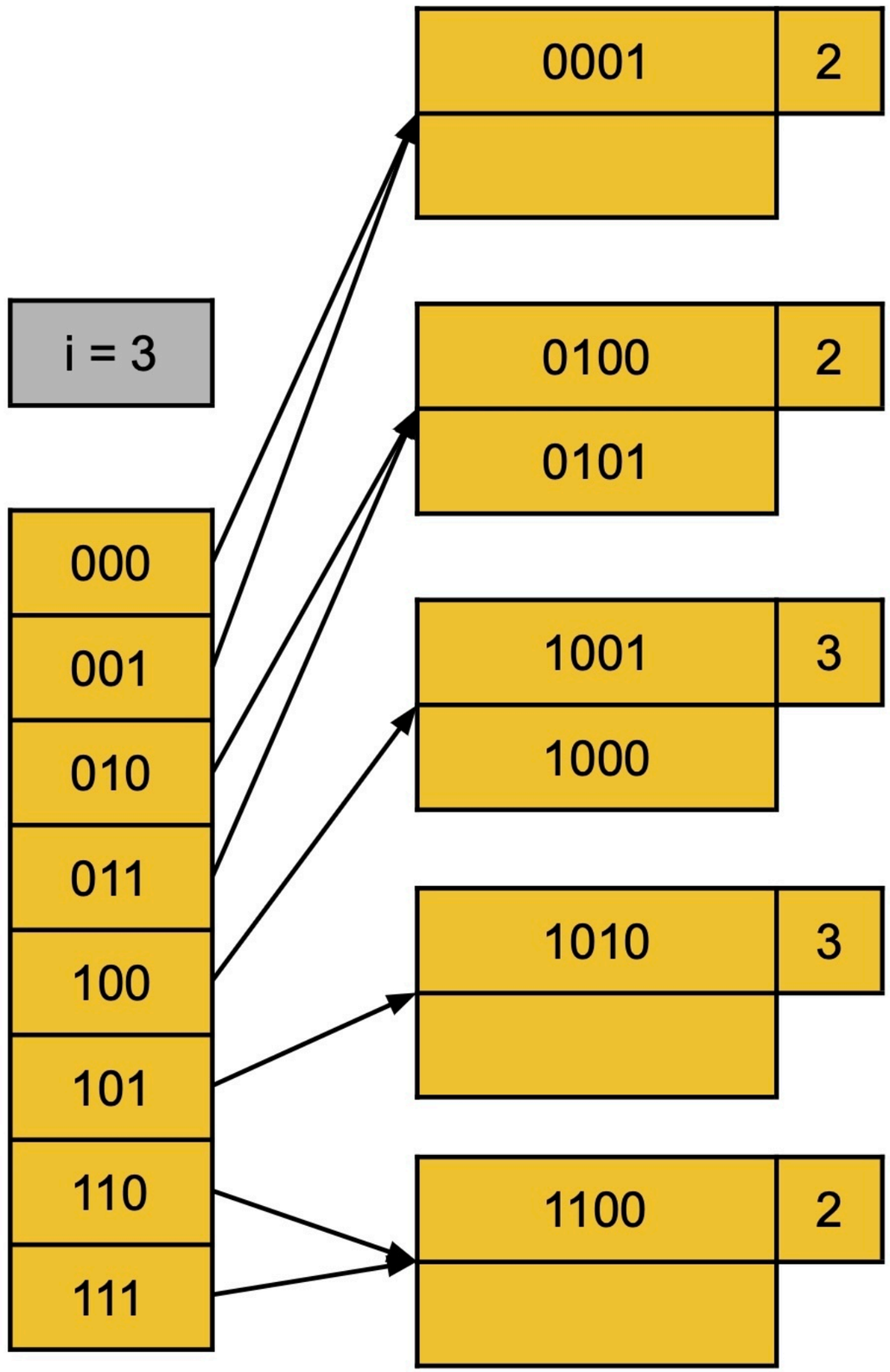
# How many buckets are there?

A. 3

B. 5

C. 6

D. 8 ✓



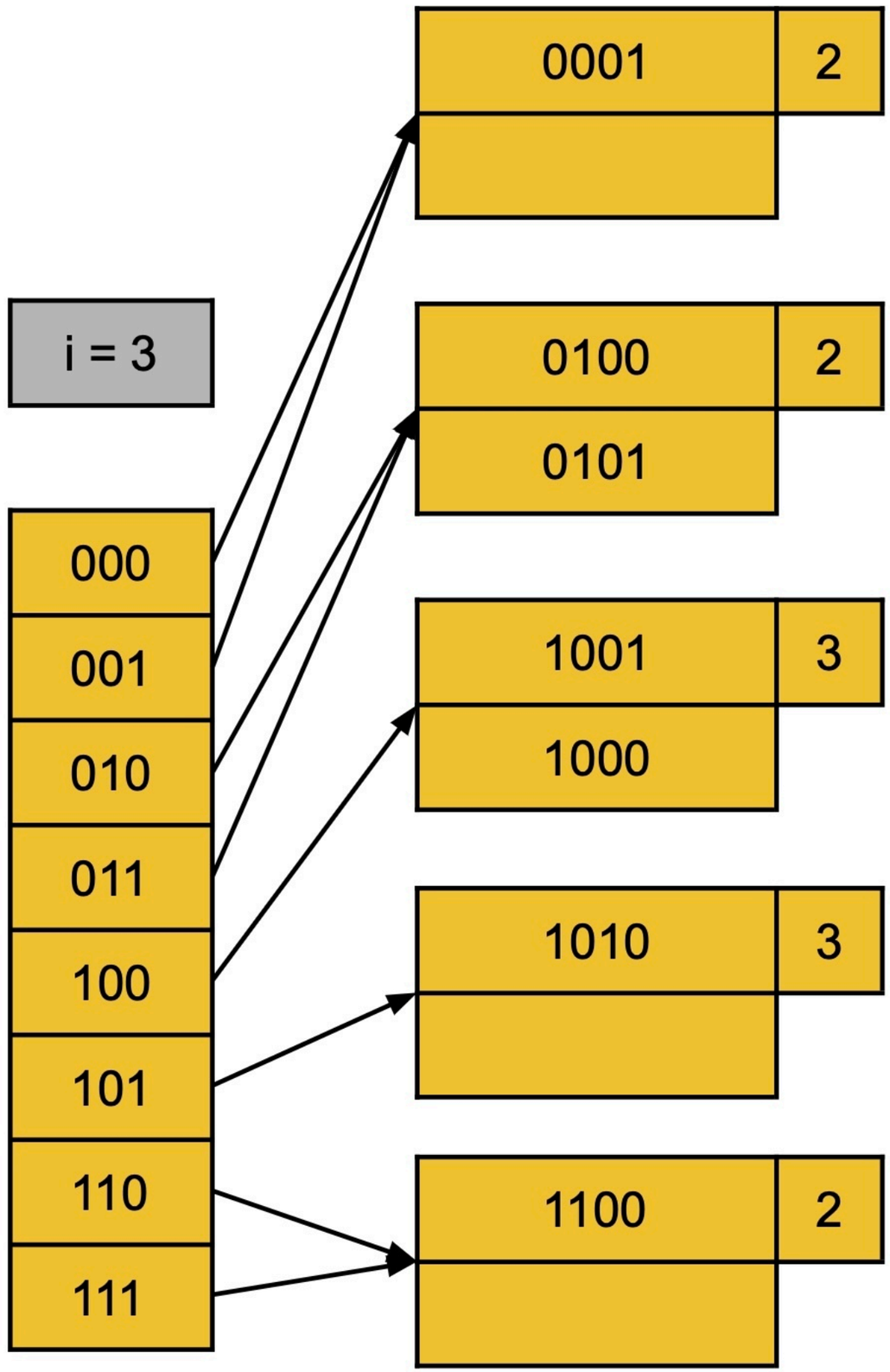
# How many blocks are there?

A. 3

B. 5 ✓

C. 6

D. 8



# Extendible Hashing: Observations

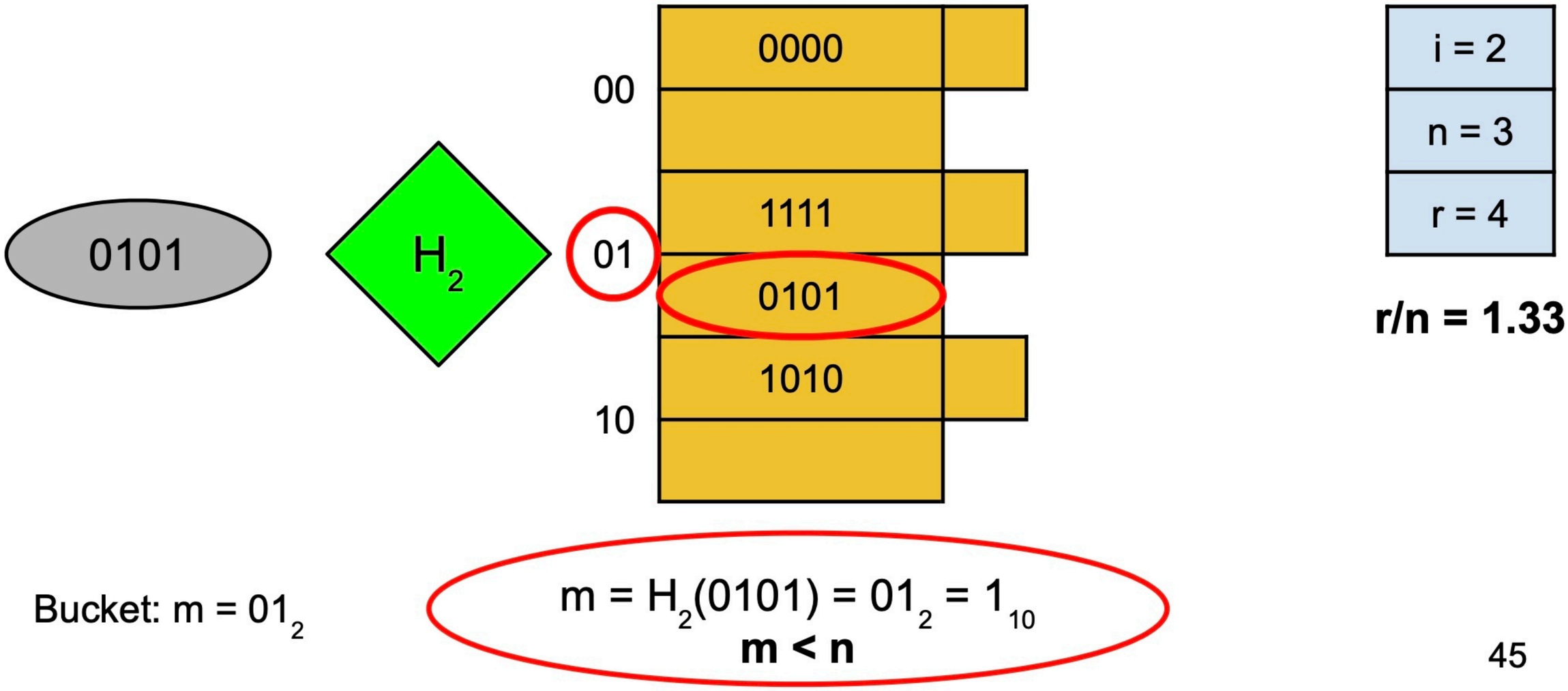
- There being no overflow blocks, **if directory fits in main memory**, equality search answered with **one disk access**.
- **Splitting causes minor reorganizations** since only the data records in one block are redistributed to the two new blocks.
- **If the redistribution after a split is unlucky**, it may be necessary to split one of the newly **divided blocks again**. For example, think of records that start with the same sequence of bits.
- The directory grows in spurts, and, if the distribution of hash values is unbalanced, **the directory can unnecessarily grow large**.
- Moreover, the **exponential growth of the directory** risks complicating its management and storage in the main memory.

# Linear Hashing

- Linear hashing avoids the need for a directory, yet **handles the problem** of long chains of **overflow blocks**.
- The number of **buckets is growing one by one**.
- **Overflow blocks are permitted**.
- The number of **buckets increases** linearly. In particular, the increase occurs **when an overflow block is created** or when **a given threshold is overcome**.
- The threshold is the **record-bucket ratio** and represents the load factor. We shall adopt the policy of choosing the number of buckets n so that there are no more than **1.7*n records** in the file.
- The hash function $H_i(K)$ returns the **i less significant bits** of the binary encoding of the search key.

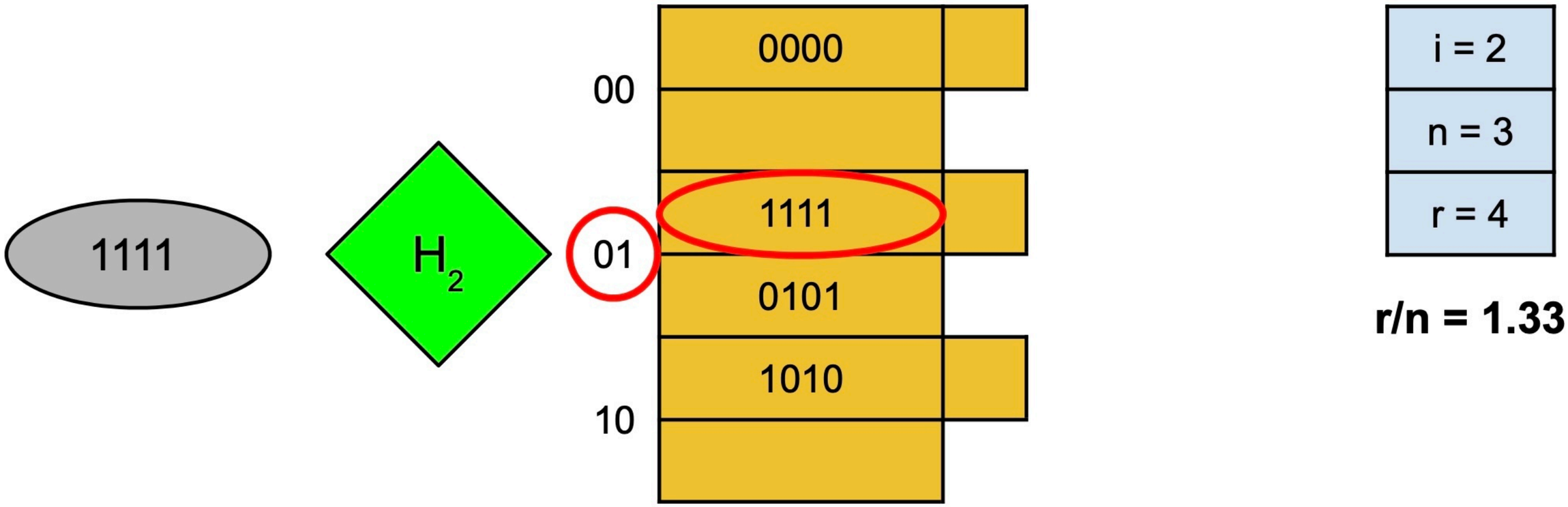
# Linear Hashing: Searching (1)

- We are looking for using the search key $K_2$=0101.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.



# Linear Hashing: Searching (2)

- We are looking for using the search key $K_2$=1111.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.




Bucket: $(m - 2^{i-1}) =$

$(3 - 2^{2-1}) = 1_{10} = 01_2$

$m = H_2(1111) = 11_2 = 3_{10}$

**$m \geq n$**

# Linear Hashing: Insertion Steps (1)

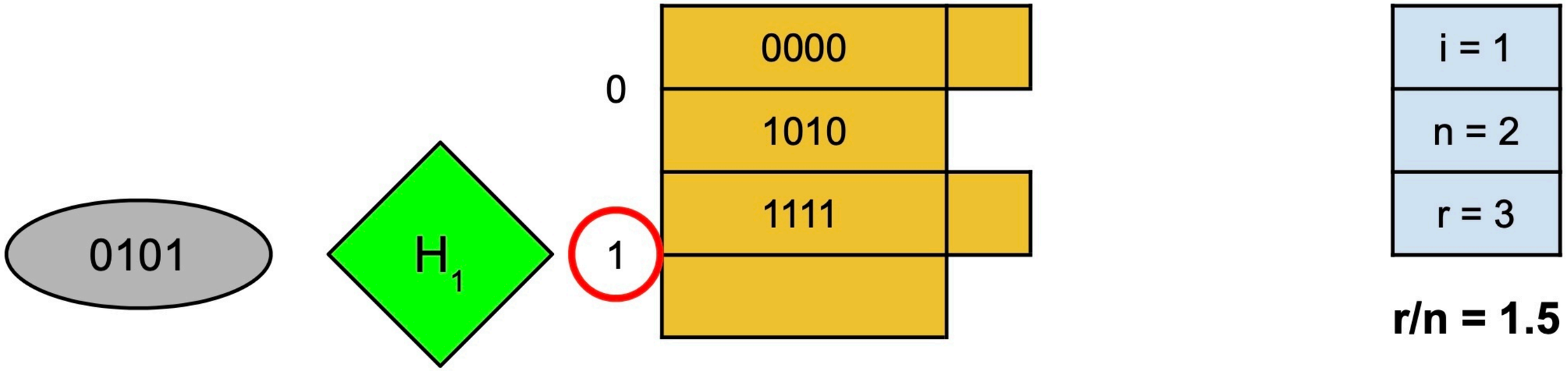
- Let's count the number of records (r) and the number of buckets (n).
- If the ratio r/n exceeds 1.7, we split a bucket adding the bucket (n+1)-th.
- While using the hashing function $H_i$, all the buckets up to the $(2^{i-1})$-th are split following the order and regardless of the bucket that caused the division.
- If n exceeds $2^i$, the hash function is switched to i+1 and the splitting starts again from the first bucket.

# Linear Hashing: Insertion Steps (2)

- If $H_i(K) = m < n$, then:
  - The search key can be inserted into the m-th bucket. If there is no room in the designated bucket, we create an overflow block.
- If $H_i(K) = m \geq n$, then:
  - The search key can be inserted into the $(m - 2^{i-1})$-th bucket. If there is no room in the designated bucket, we create an overflow block.
- Increase r and if r/n > 1.7, then:
  1. If $n = 2^i$, then increase i by 1.
  2. Add the n-th bucket.
  3. If the binary representation of the number of the bucket we added is $1a_2a_3 \ldots a_i$, then we split the bucket numbered $0a_2a_3 \ldots a_i$. We move into the n-th bucket all the records from the bucket $0a_2a_3 \ldots a_i$ having the i-th rightmost bit equals to 1.
  4. Increase n by 1.

# Linear Hashing: Insertion (1)

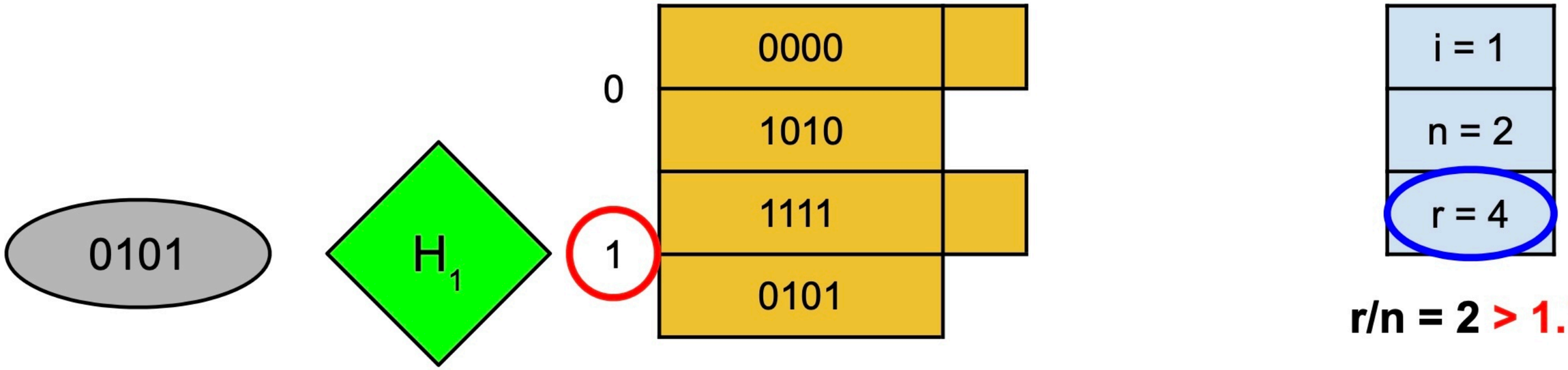
- We are looking for using the search key $K_2$=0101.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.




$m = H_2(0101) = 1_2 = 1_{10}$
**m < n**

# Linear Hashing: Insertion (2)

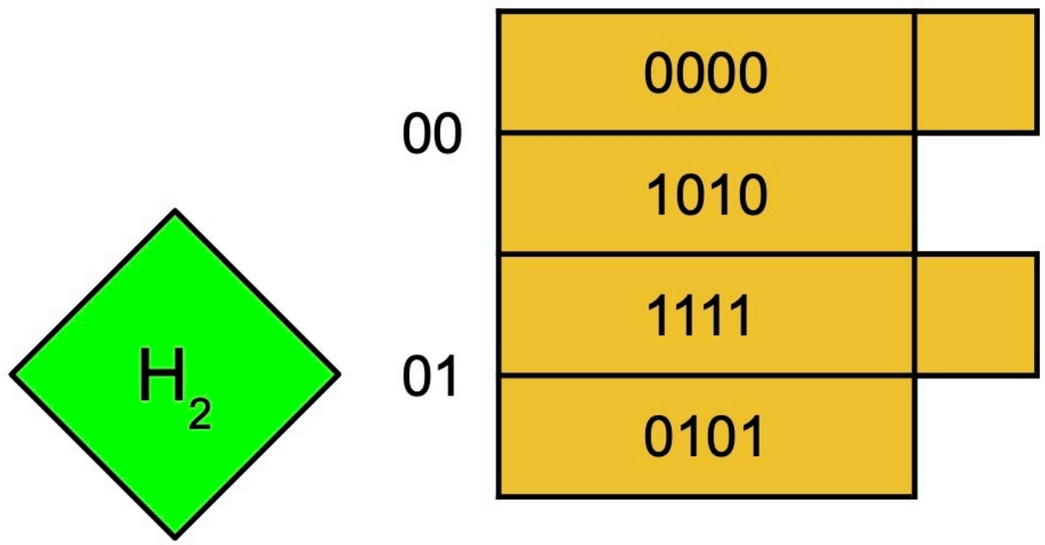
- We are looking for using the search key $K_2$=0101.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.




$m = H_2(0101) = 1_2 = 1_{10}$
**m < n**

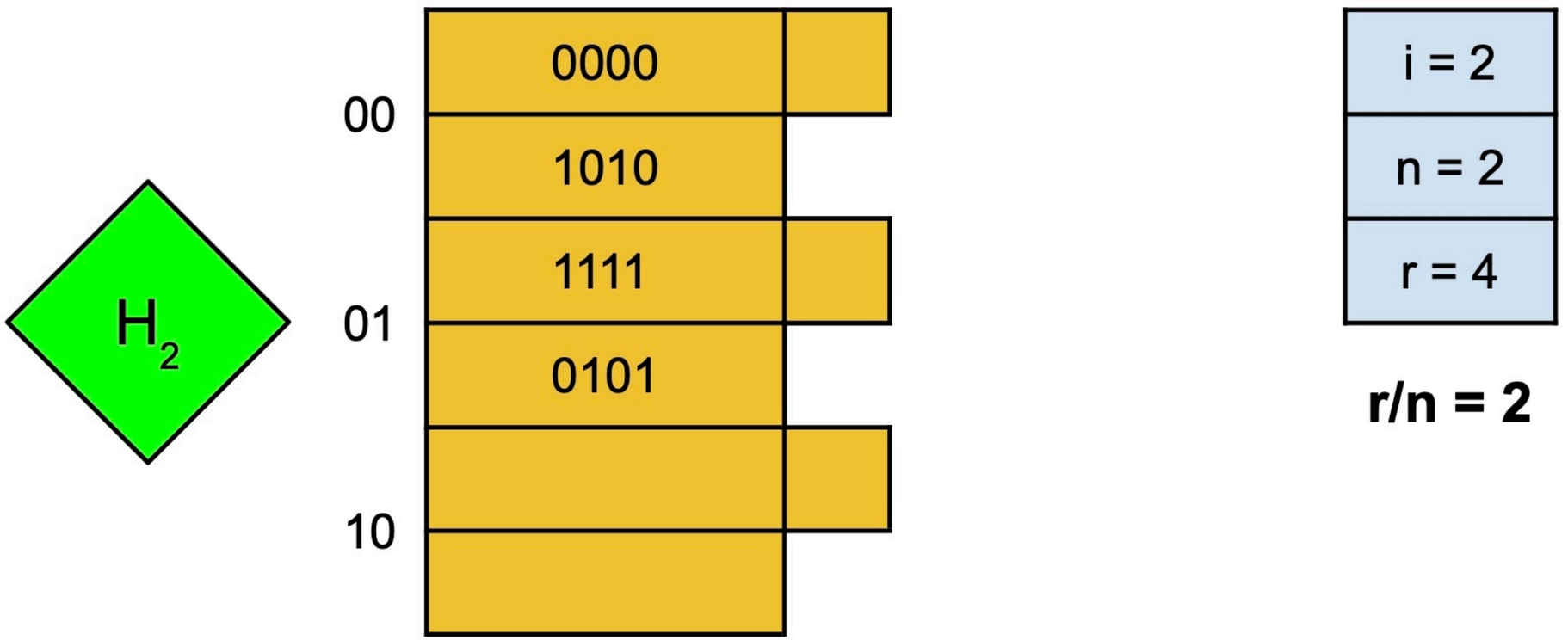
# Linear Hashing: Load Balancing (1)

1. If $n = 2^i$, then increase i by 1.



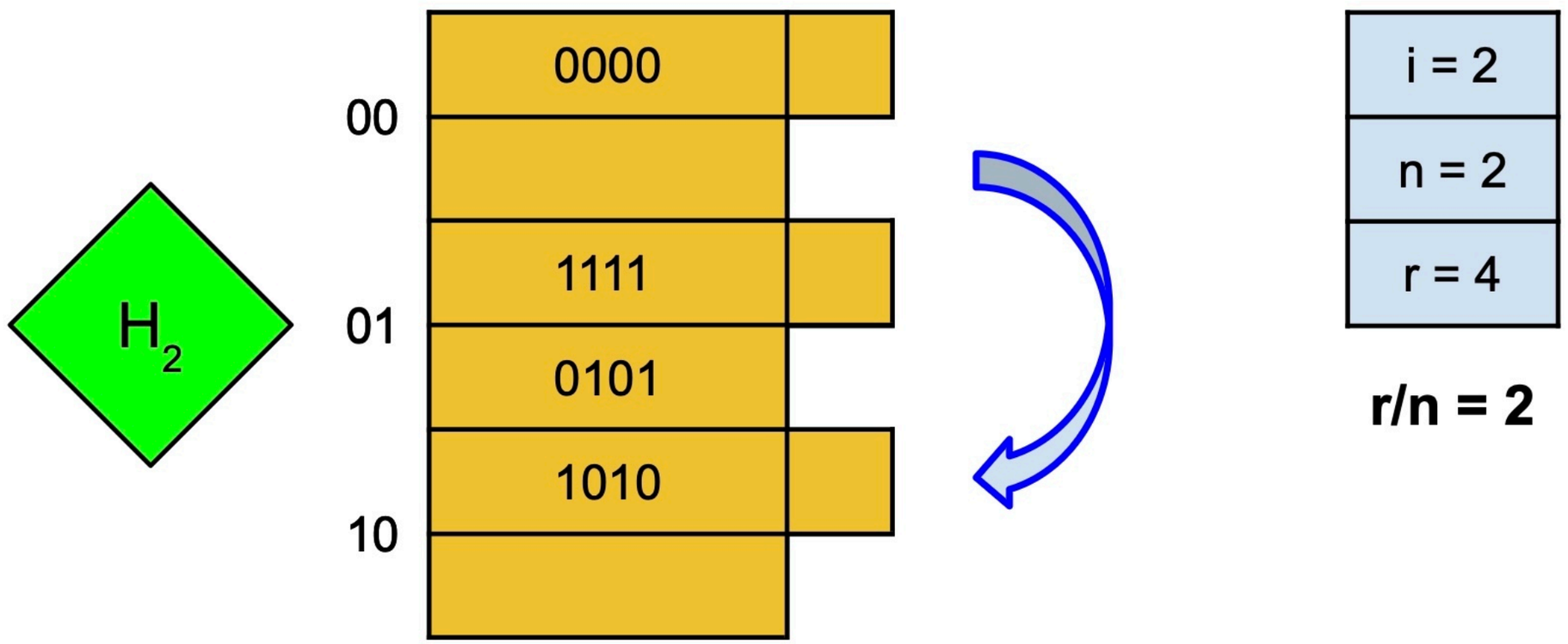
# Linear Hashing: Load Balancing (2)

2. Add the n-th bucket.



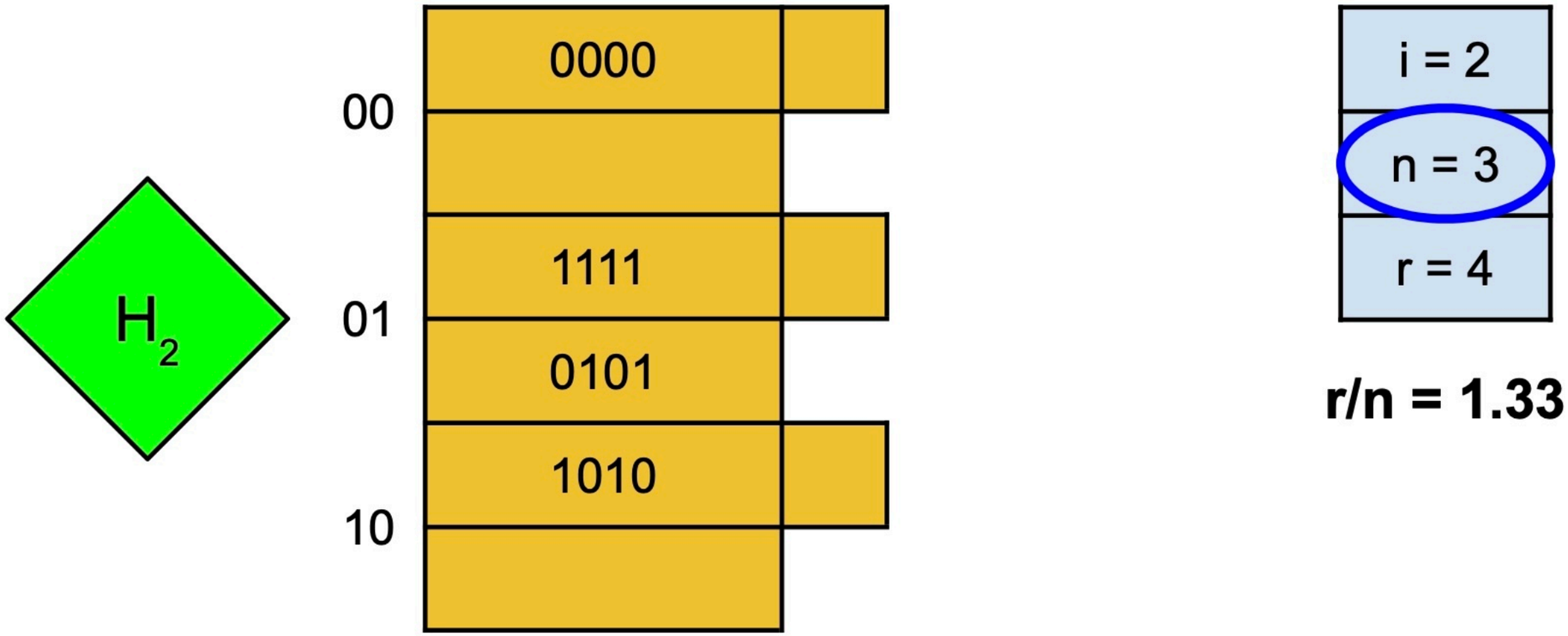
# Linear Hashing: Load Balancing (3)

3. We move into the n-th bucket all the records from the bucket $0a_2a_3 \ldots a_i$ having the i-th rightmost bit equals to 1.
   - $n = 2_{10} = 10_2$ ($\equiv 1a_2a_3 \ldots a_i$) $\rightarrow 10_2$ identifies the new bucket.
   - We move the data records from $00_2$ ($\equiv 0a_2a_3 \ldots a_i$) to $10_2$.



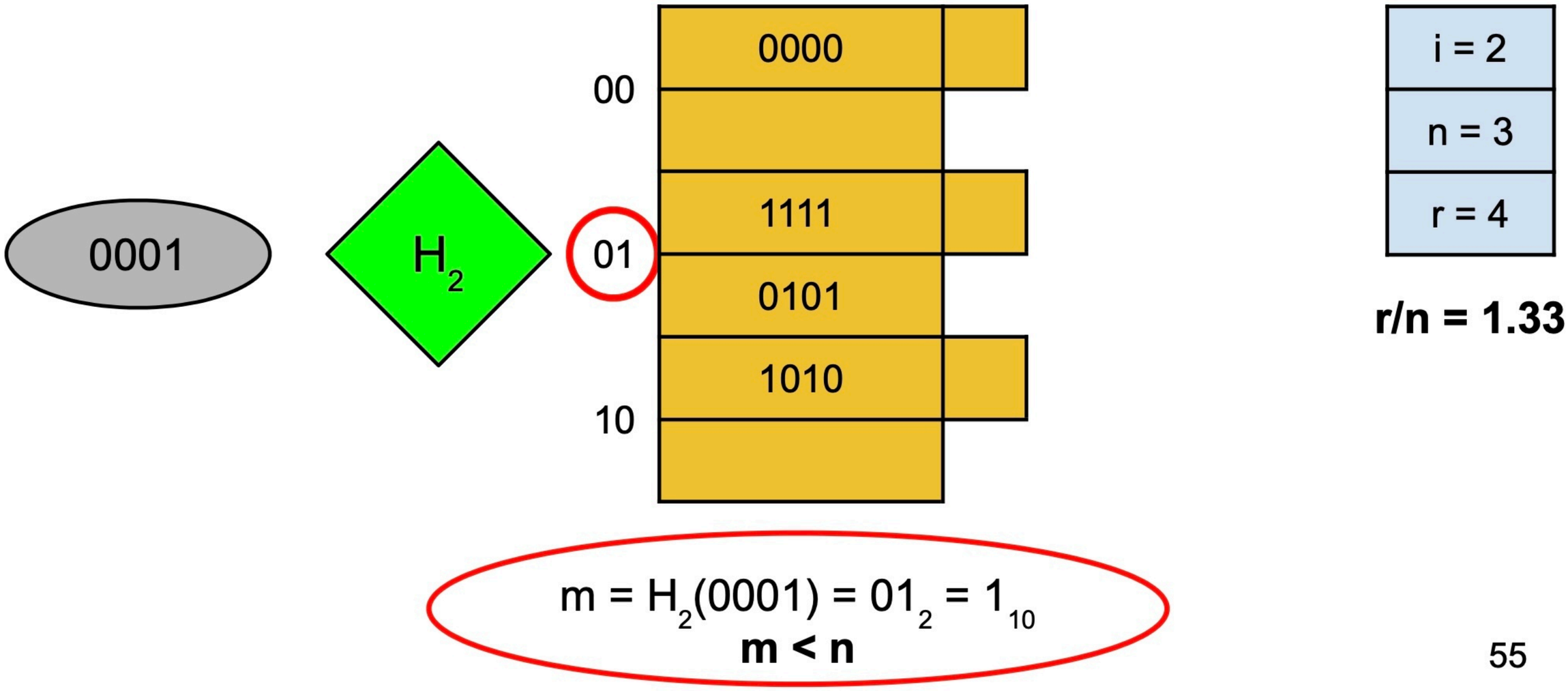
# Linear Hashing: Load Balancing (4)

4. Increase n by 1.



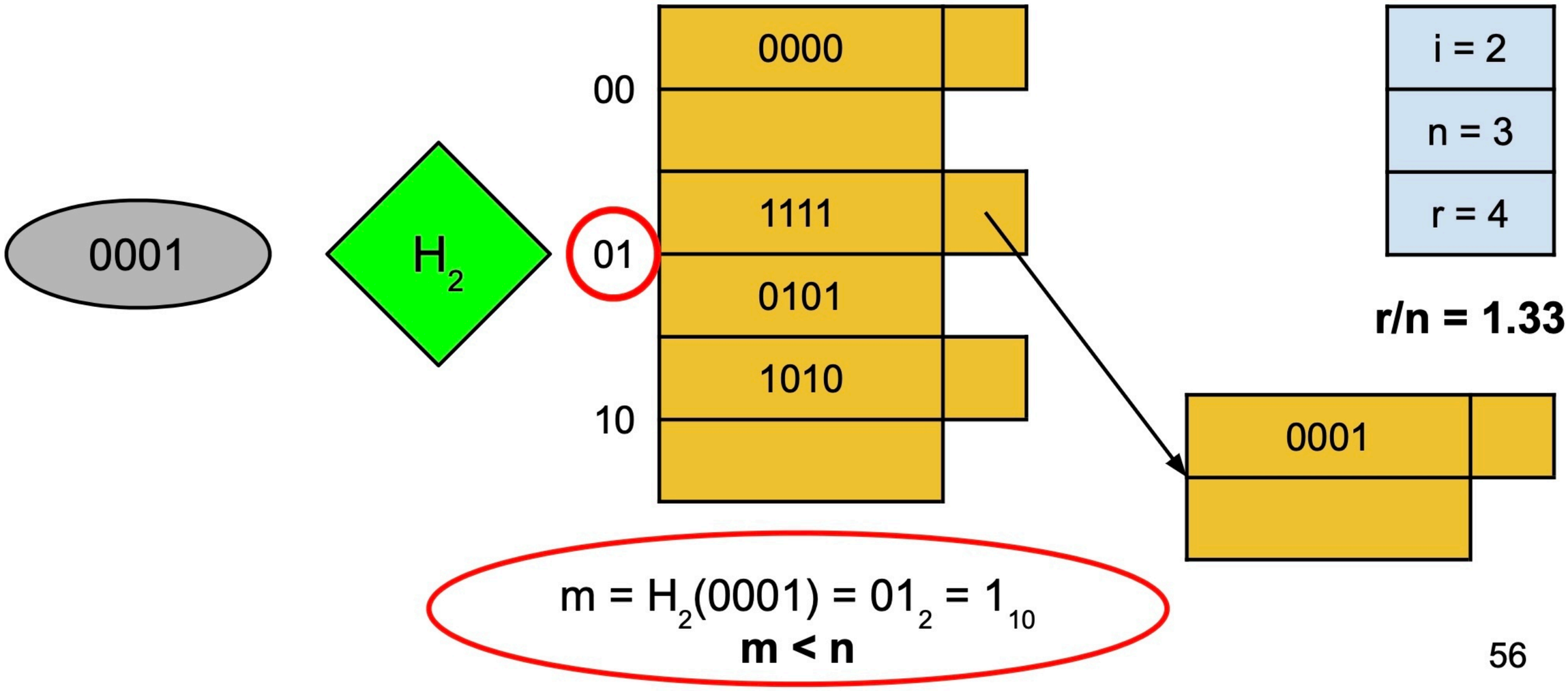
# Linear Hashing: Insertion (1)

- We are looking for using the search key $K_2$=0001.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.



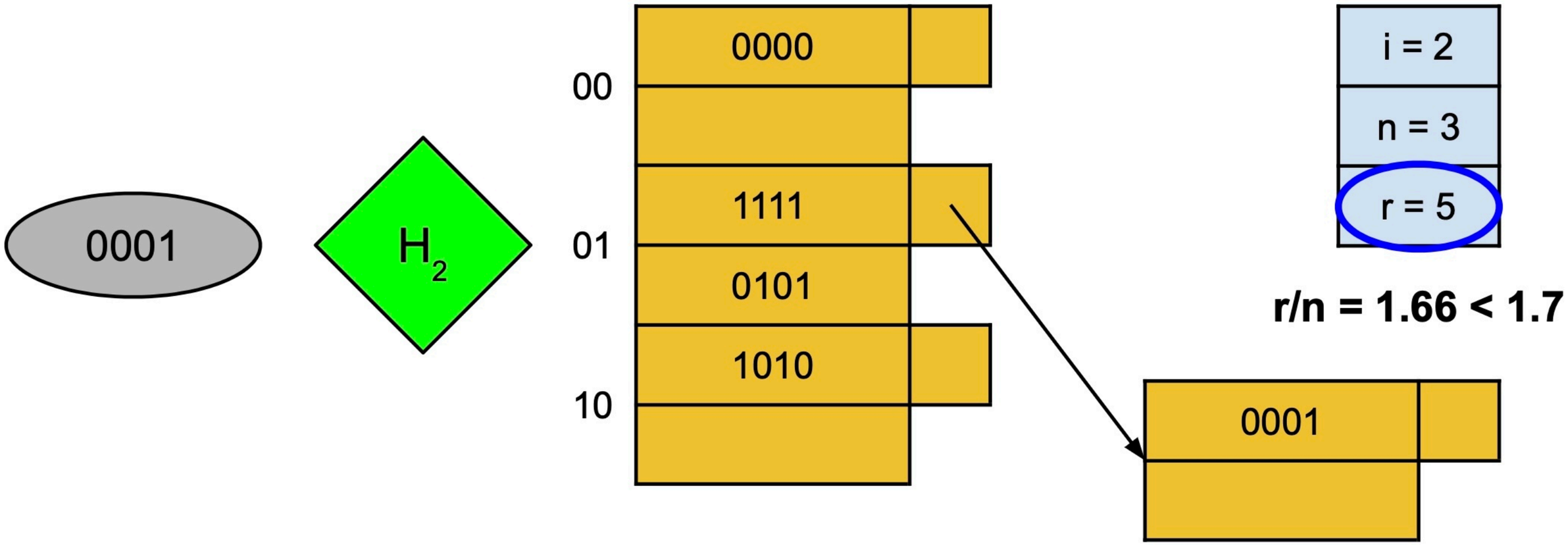
# Linear Hashing: Insertion (2)

- We are looking for using the search key $K_2$=0001.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.



# Linear Hashing: Insertion (3)

- We are looking for using the search key $K_2$=0001.
- Let n be the number of buckets (where $2^{i-1} < n \leq 2^i$).
  - If $H_i(K) = m < n$, the search key is in the bucket m.
  - If $H_i(K) = m \geq n$, the search key is in the bucket $(m - 2^{i-1})$.

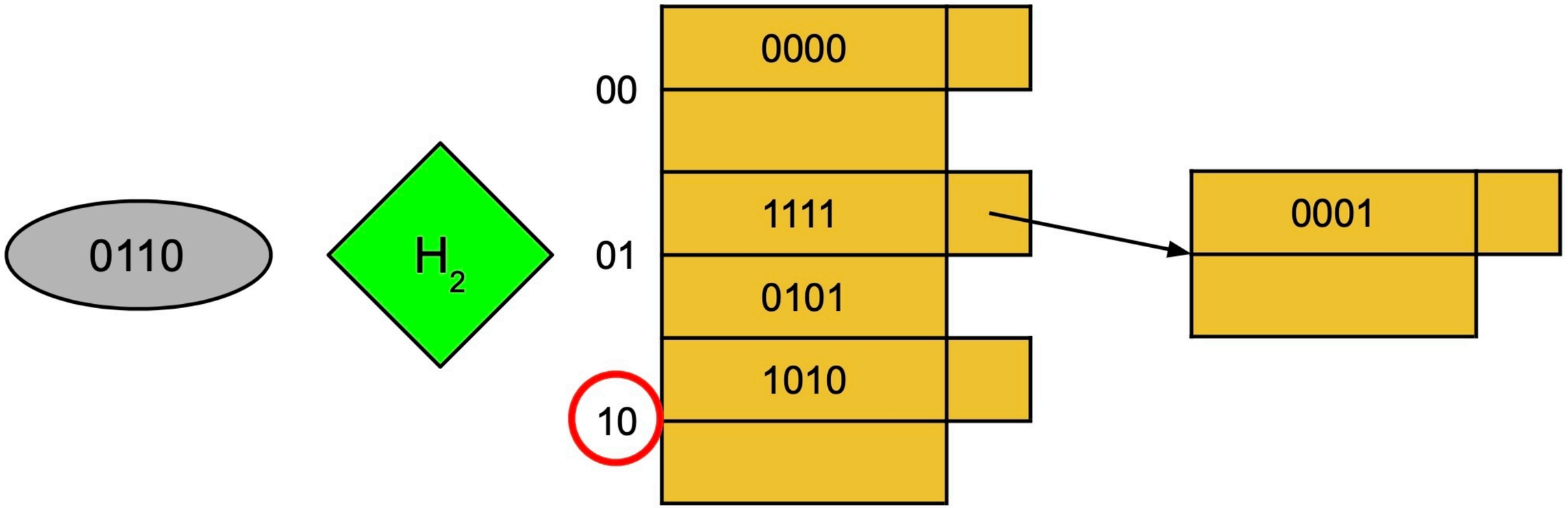


# Linear Hashing: Insertion (4)

- We are looking for using the search key $K_2$=0110.

| i = 2 |
|---|
| n = 3 |
| r = 5 |

**r/n = 1.66**

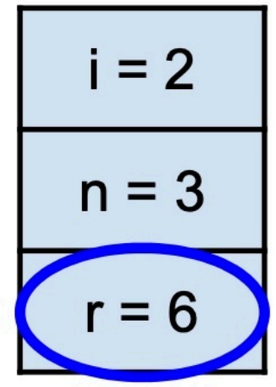



$m = H_2(0110) = 10_2 = 2_{10}$

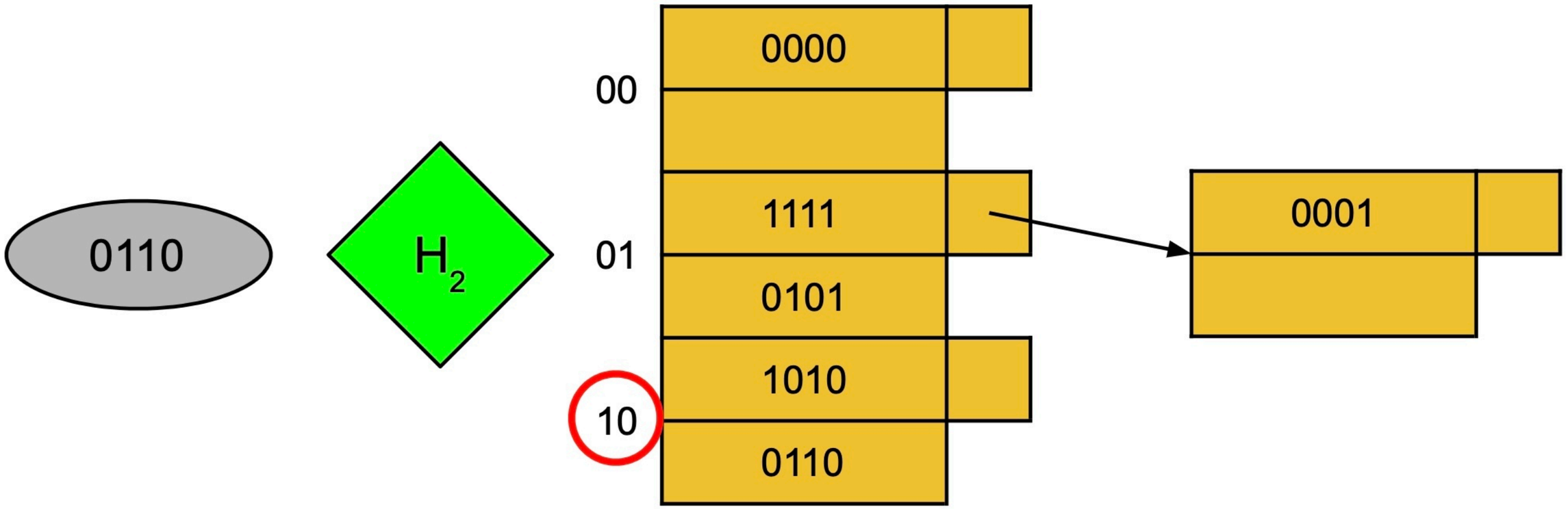
**m < n**

# Linear Hashing: Insertion (5)

- We are looking for using the search key $K_2$=0110.




**r/n = 2 > 1.7**




$m = H_2(0110) = 10_2 = 2_{10}$

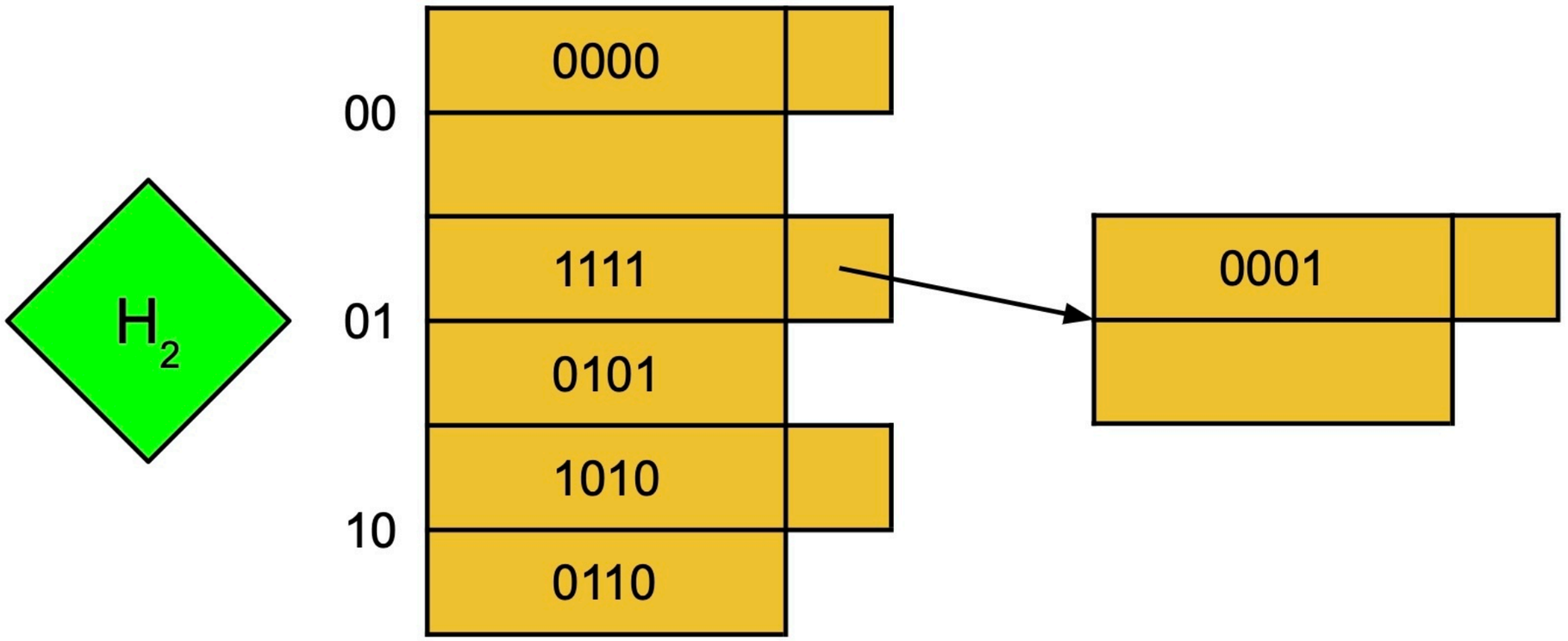
**m < n**

# Linear Hashing: Load Balancing (1)

1. Since $n \neq 2^i$, we don't need to increase i.

| i = 2 |
|---|
| n = 3 |
| r = 6 |

**r/n = 2**

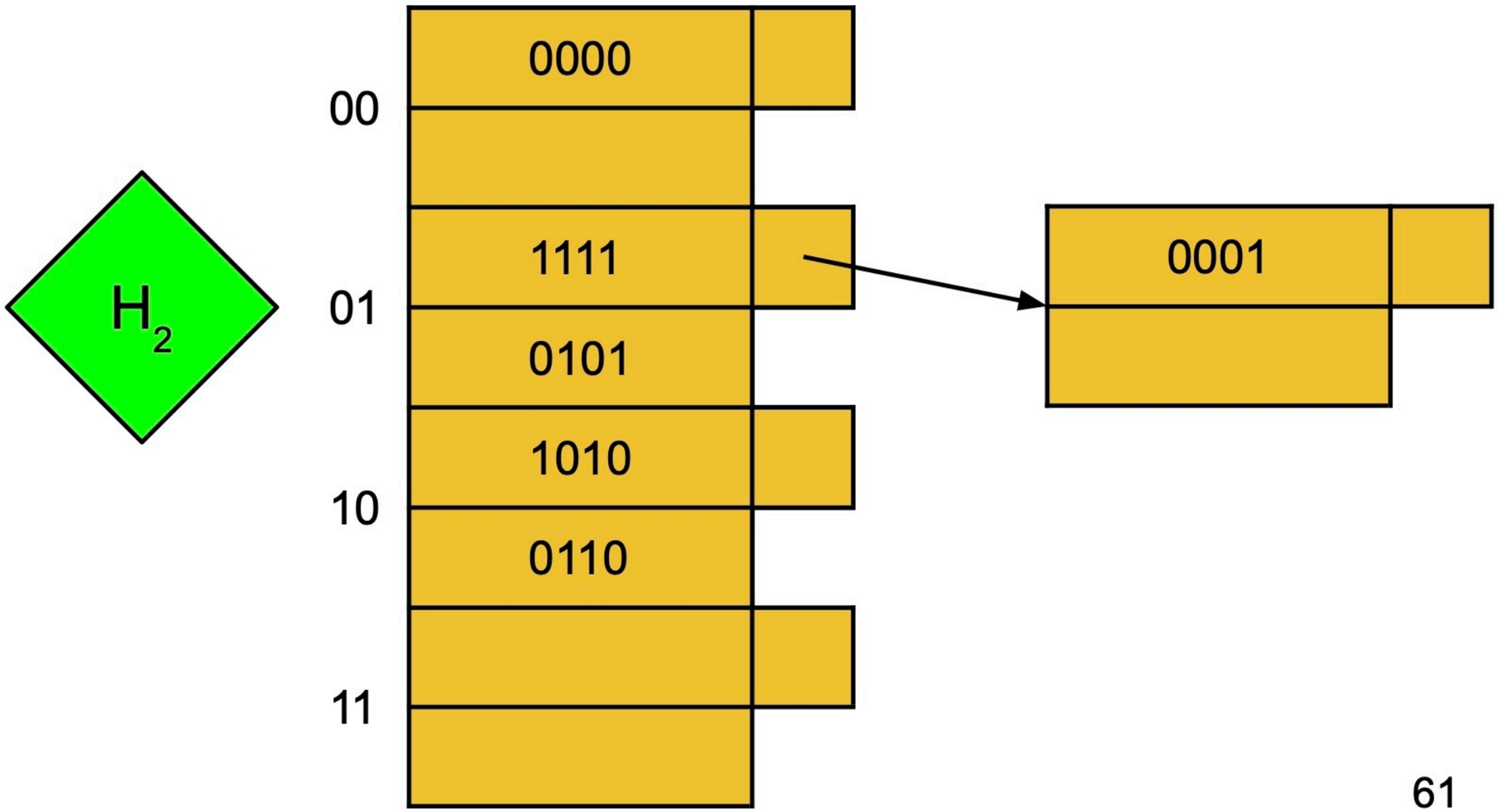


# Linear Hashing: Load Balancing (2)

2. Add the n-th bucket.

| i = 2 |
|---|
| n = 3 |
| r = 6 |

**r/n = 2**

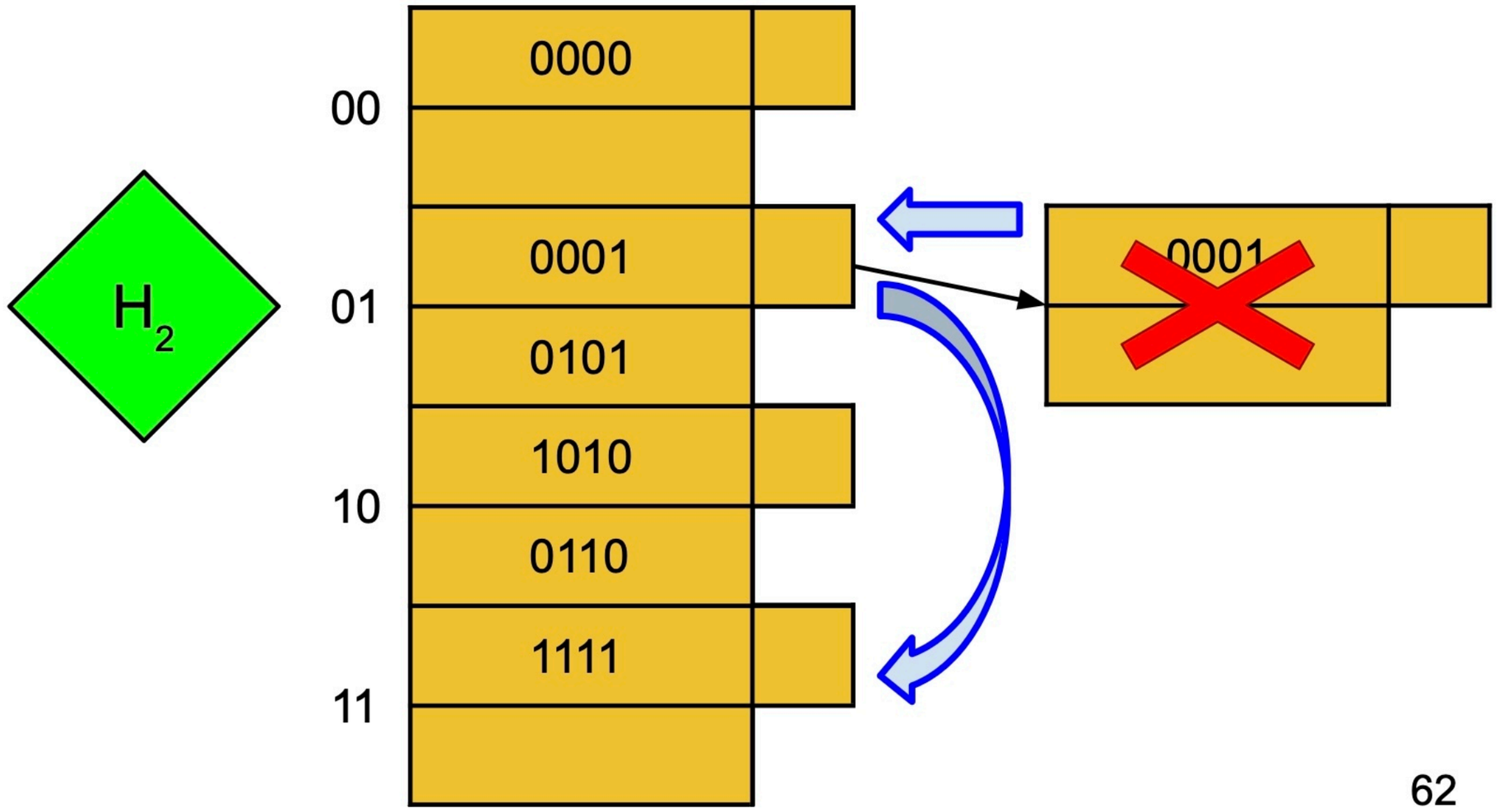


# Linear Hashing: Load Balancing (3)

3. We move into the n-th bucket all the records from the bucket $0a_2a_3 \ldots a_i$ having the i-th rightmost bit equals to 1.
   - $n = 3_{10} = 11_2$ ($\equiv 1a_2a_3 \ldots a_i$).
   - Move from $01_2$ ($\equiv 0a_2a_3 \ldots a_i$) to $11_2$.

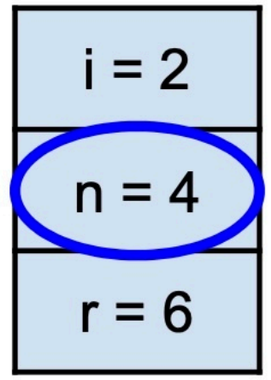
| i = 2 |
|---|
| n = 3 |
| r = 6 |

**r/n = 2**

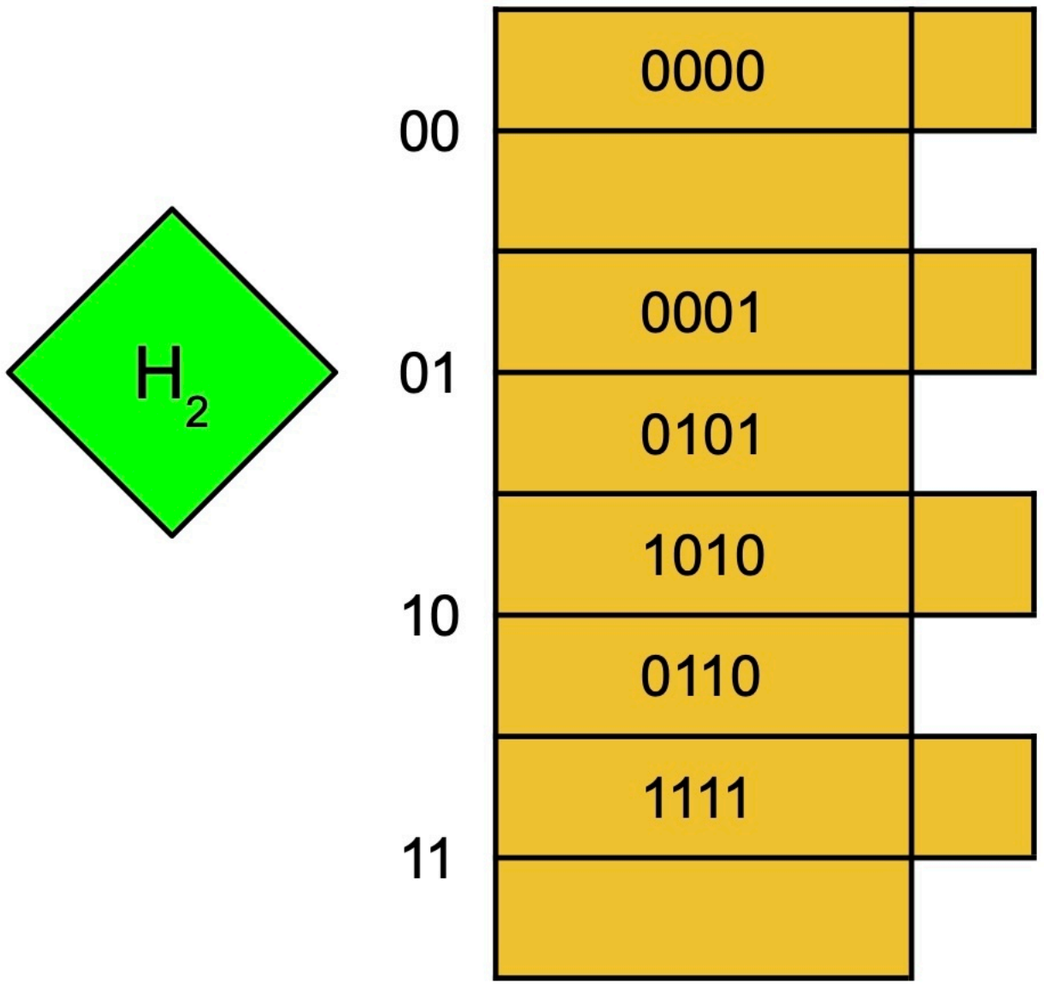


# Linear Hashing: Load Balancing (4)

4. Increase n by 1.




**r/n = 1.5**



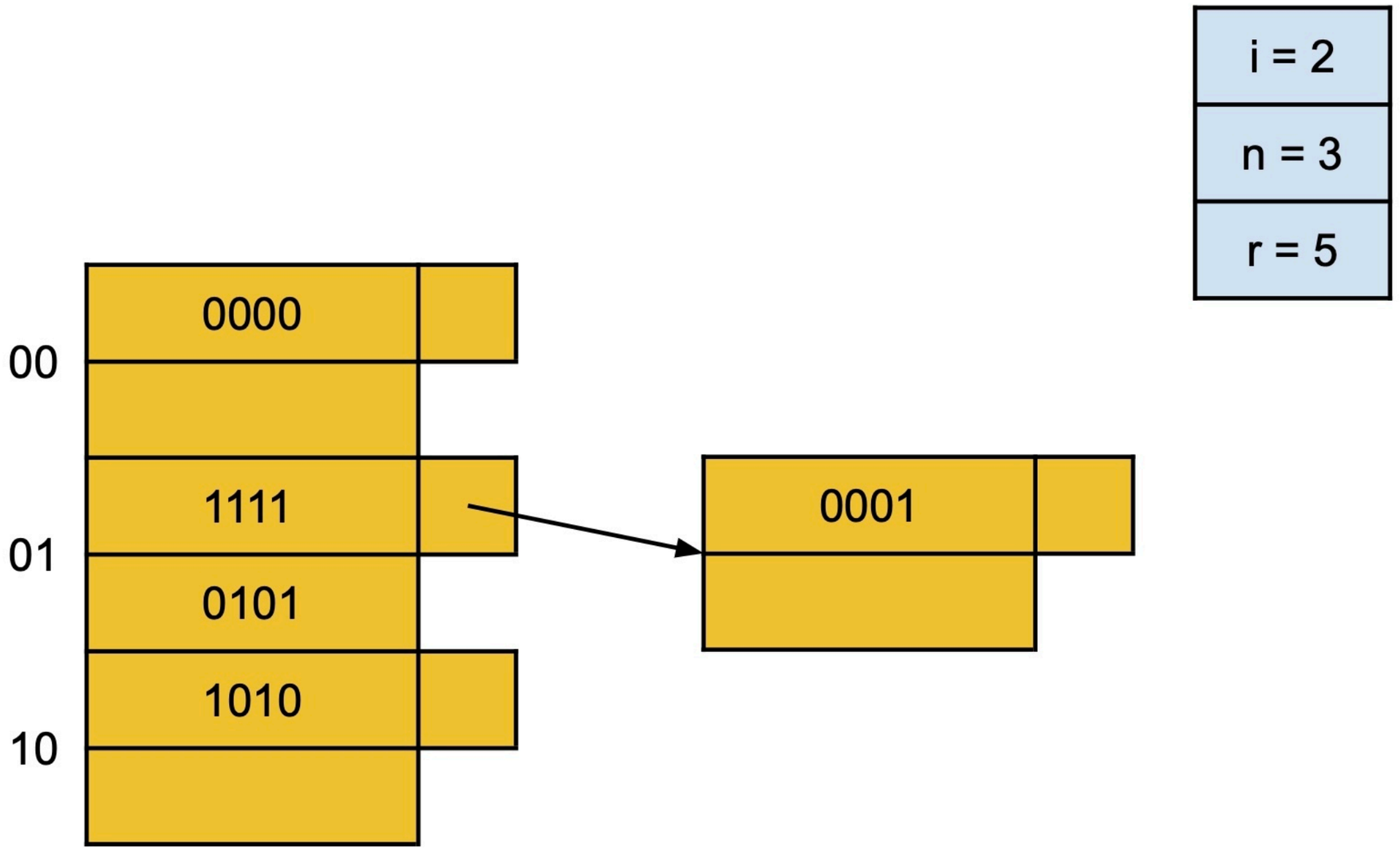
# How many buckets are there?

A. 2

B. 3 ✓

C. 4

D. 5



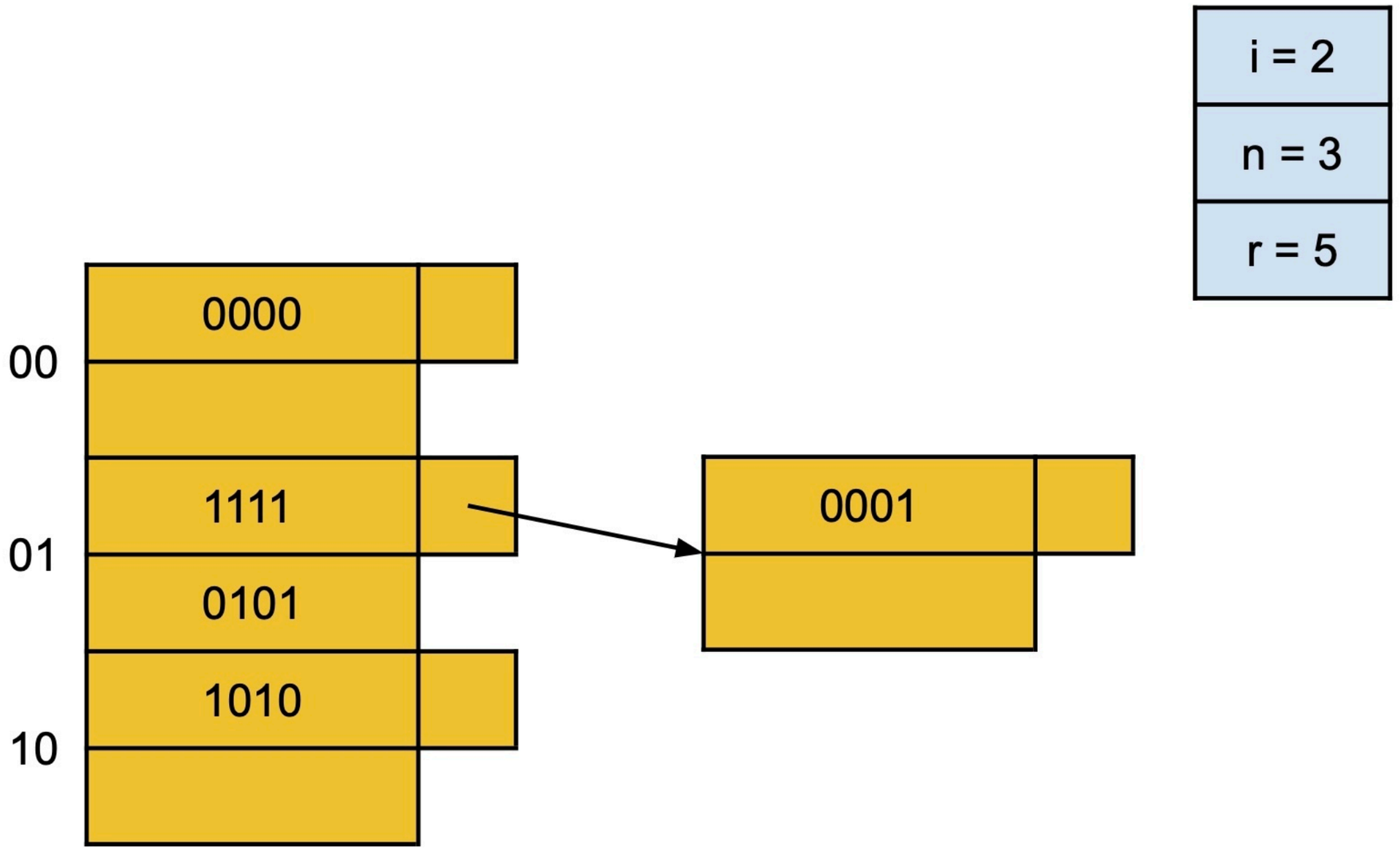
# How many blocks are there?

A. 2

B. 3

C. ✓ 4

D. 5



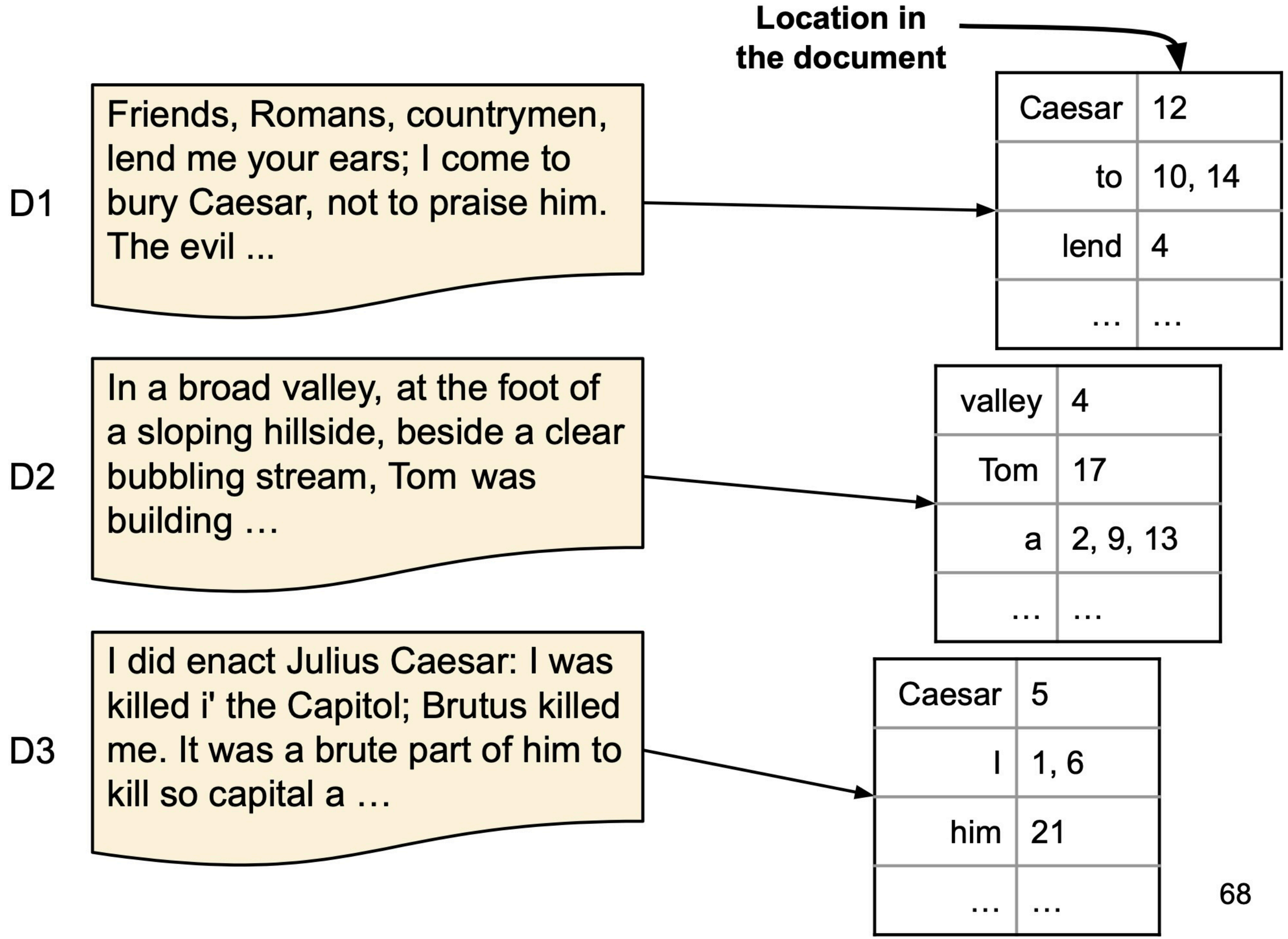
# Document Retrieval

- Information retrieval is **finding material** (usually documents) of an **unstructured nature** (usually text) that satisfies an **information need** from within **large collections** (stored on computers and available online).
- The **retrieval of documents** given keywords has become one of the **largest database problems**.
- We have seen how to efficiently retrieve **data records** (i.e., the entries of a table) containing one or more values of interest.
- Similarly, we can use indexes to efficiently retrieve **documents containing the text** of interest.

# Inverted Indexes

- Inverted indexes are used to efficiently retrieve unstructured documents through full-text queries.
- In particular, there are two kinds of queries we are interested in:
  1. Return all the documents that contain **a given set of keywords** $K_1$, $K_2$, … $K_n$.
  2. Return all the documents that contain **a given sequence of keywords** $K_1$, $K_2$, … $K_n$.
- Inverted indexes support these type of operations because instead of creating a separate index for each attribute (i.e., for each word), they represent all the documents in which a specific word appears.

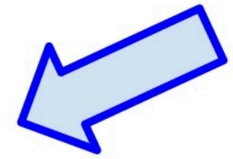
# Building an Inverted Index (1)



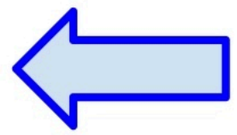
# Building an Inverted Index (2)

We create an inverted index, consisting of a dictionary with pairs (document, position).

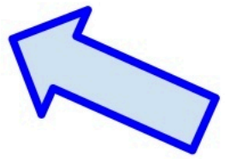
| | |
|---|---|
| a | (D2,2), (D2,9), (D2,13) |
| Brutus | (D3,12) |
| … | … |
| Caesar | (D1,12), (D3,5) |
| clear | (D2,14) |
| him | (D3,21) |
| I | (D3,1), (D3,6) |
| … | … |
| valley | (D2, 4) |



| | |
|---|---|
| Caesar | 12 |
| to | 10, 14 |
| lend | 1 |
| … | … |



| | |
|---|---|
| valley | 4 |
| Tom | 17 |
| a | 2, 9, 13 |
| … | … |



| | |
|---|---|
| Caesar | 5 |
| I | 1, 6 |
| him | 21 |
| … | … |

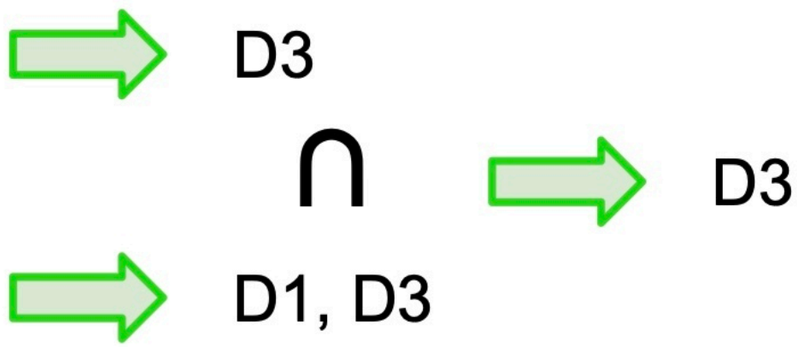
# Linguistic Preprocessing

- To gain the speed benefits of indexing at retrieval time and to improve the accuracy of results, there are several techniques, such as:
  - **Token normalization** - The words are normalized so that matches occur despite superficial differences in the character sequences. For example, the word “Windows” might be transformed into “windows”.
  - **Stemming** - We remove suffixes to find the “stem” of each word, before entering its occurrence into the index. For example, plural nouns can be treated as their singular versions or the words “fishing”, “fished”, and “fisher” might be reduced to the stem “fish”.
  - **Stop words** - The most common words, such as “the” or “and”, are called stop words and often are excluded from the inverted index. They appear in too many documents to make them useful and eliminating stop words also reduces the size of the index.

# Queries with Inverted Index (1)

Return all the documents that contain both the words “Brutus” and “Caesar”.

| | |
|---|---|
| a | (D2,2), (D2,9), (D2,13) |
| Brutus | (D3,12) |
| … | … |
| Caesar | (D1,12), (D3,5) |
| clear | (D2,14) |
| him | (D3,21) |
| I | (D3,1), (D3,6) |
| … | … |
| valley | (D2, 4) |




I did enact Julius **Caesar**: I was killed i' the Capitol; **Brutus** killed me. It was a brute part of him to kill so capital a …

# Queries with Inverted Index (2)

Return all the documents that contain the sequence “a valley”.

| | |
|---|---|
| a | (D2,2), (D2,9), (D2,13) |
| Brutus | (D3,12) |
| … | … |
| Caesar | (D1,12), (D3,5) |
| clear | (D2,14) |
| him | (D3,21) |
| I | (D3,1), (D3,6) |
| … | … |
| valley | (D2, 4) |

(D2,2),
(D2,9),
(D2,13)

∩



D2: ~~2, 4~~ ~~9, 4~~ ~~13, 4~~

(D2,4)

# Queries with Inverted Index (3)

Return all the documents that contain the sequence “a clear”.

| | |
|---|---|
| a | (D2,2), (D2,9), (D2,13) |
| Brutus | (D3,12) |
| … | … |
| Caesar | (D1,12), (D3,5) |
| clear | (D2,14) |
| him | (D3,21) |
| I | (D3,1), (D3,6) |
| … | … |
| valley | (D2, 4) |

(D2,2),
(D2,9),
(D2,13)

∩

2, 14
D2: 9, 14
13, 14

(D2,14)

In a broad valley, at the foot of a sloping hillside, beside **a clear** bubbling stream, Tom was building …