CS525: Advanced Database Organization

Notes 4: Indexing and Hashing
Part III: Hashing and more

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

September, 18th, 2018

Slides: adapted from a courses taught by Hector Garcia-Molina, Stanford, Shun Yan Cheung, Emory University, Jennifer Welch, & Elke A. Rundensteiner, Worcester Polytechnic Institute

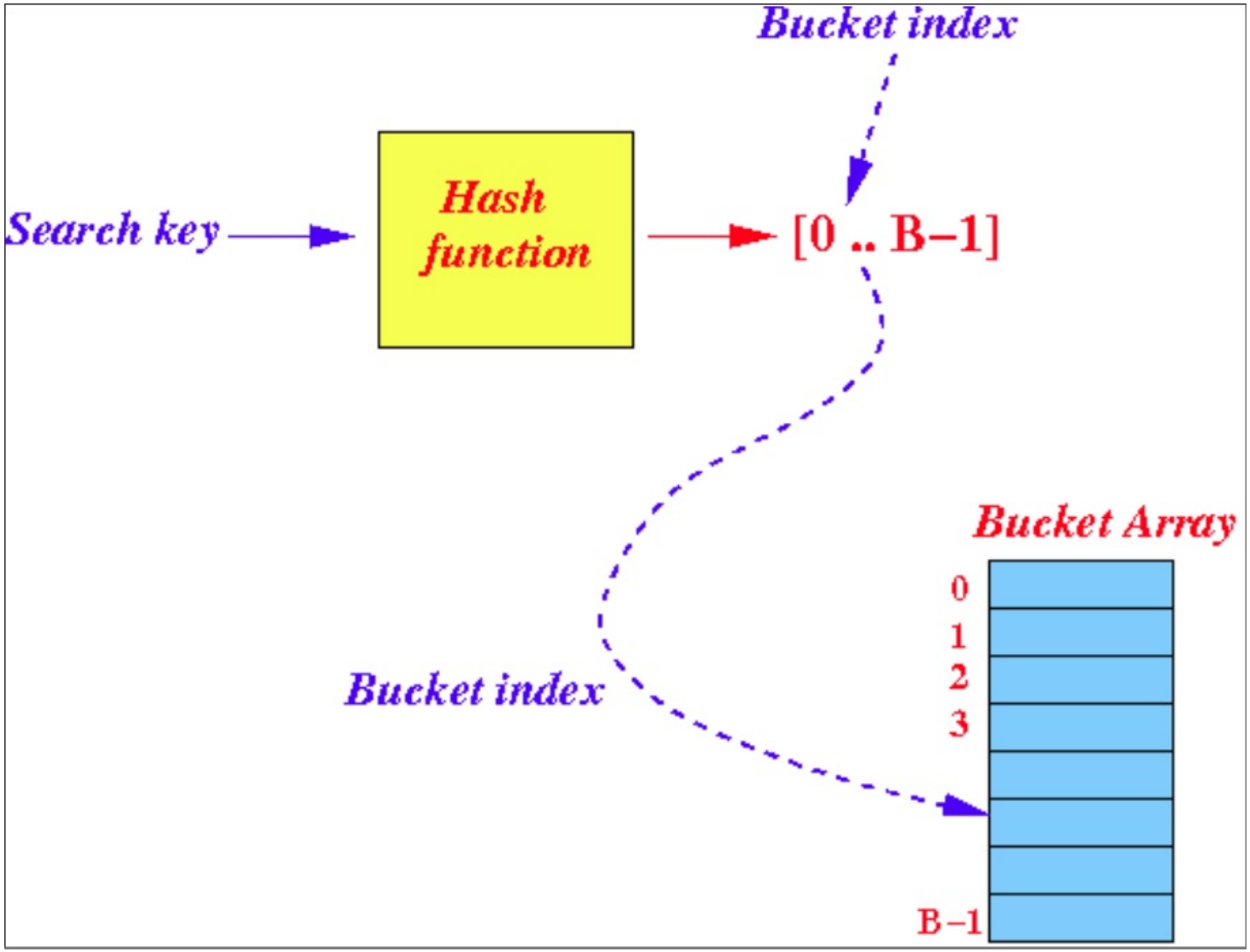
Outline

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level ...
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- B^+-Trees
- Hashing schemes

Hash Function

- **Hash function**
  - a function that maps a search key to an index between `[0..B-1]`, where `B` is the size of the hash table (number of buckets)
- **Bucket**
  - a location (slot) in the bucket array
- **Bucket array**
  - An array of (fixed) size B
  - Each cell of the bucket array is called a `bucket` and holds a pointer to a linked list, one for each bucket of the array.
- The integer between `[0..B-1]` is used as an index for the bucket array
- Record with key k is put in the linked list that starts at entry `h(k)` of B.

The hashing technique



Different search keys can be hashed into the same hash bucket

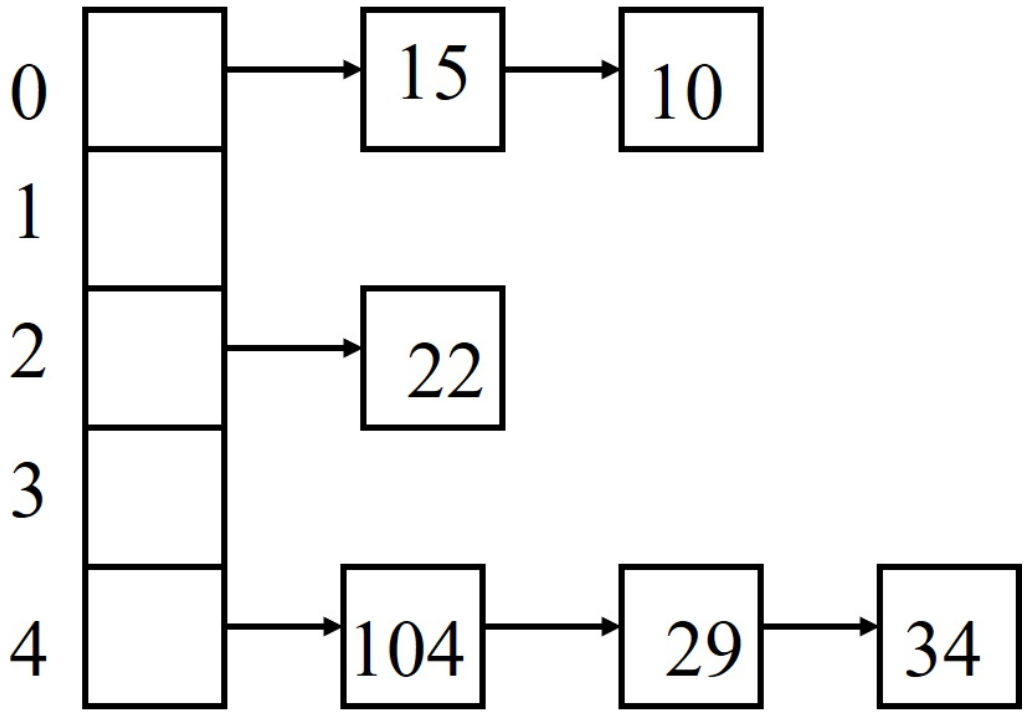
Example hash function

- Typical hash functions perform computation on the internal binary representation of the search-key.
- For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned
  - Key = $x_1 x_2 \ldots x_n$, n bytes character string
  - Have B buckets
  - h
    - add $x_1 + x_2 + \ldots + x_n$
    - compute sum modulo B
  - This may not be best function
  - Read Knuth Vol. 3, chapter 6.4 if you really need to select a good function.
- Good hash function
  - Expected number of keys/bucket is the same for all buckets

Example of Hash Table

$B = 5$

$h(k) = k \bmod 5$



Within a bucket

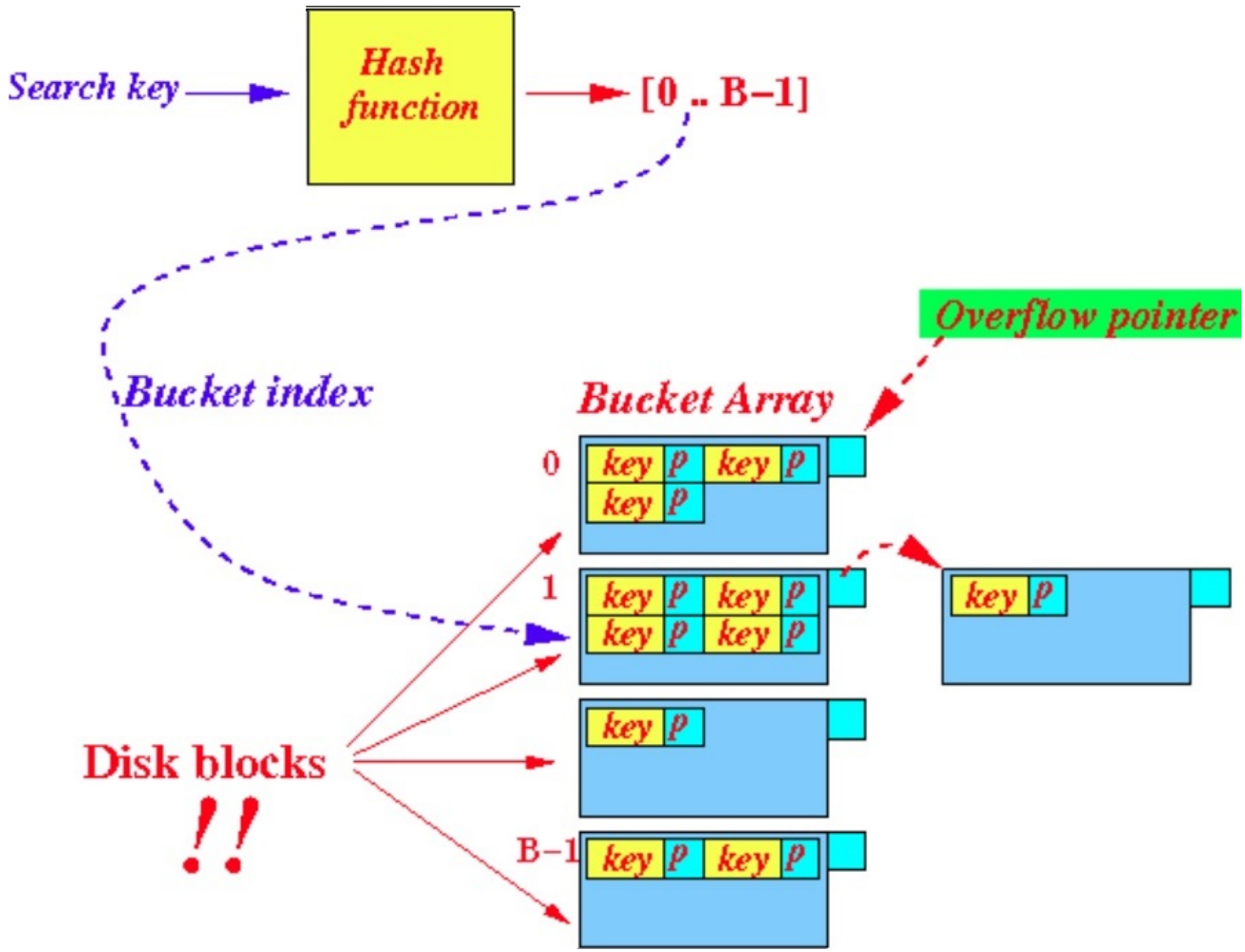
- Do we keep keys sorted?
- Yes, if CPU time critical & Inserts/Deletes not too frequent

Secondary-Storage Hash Tables

- A hash table holds a very large number of records
  - must be kept mainly in secondary storage
- `Bucket array` contains `blocks`, not pointers to linked lists
- Records that hash to a certain bucket are put in the corresponding `block`
- One bucket will contain `n` (search key, block pointer)
- If a `bucket` overflows then start a chain of overflow blocks

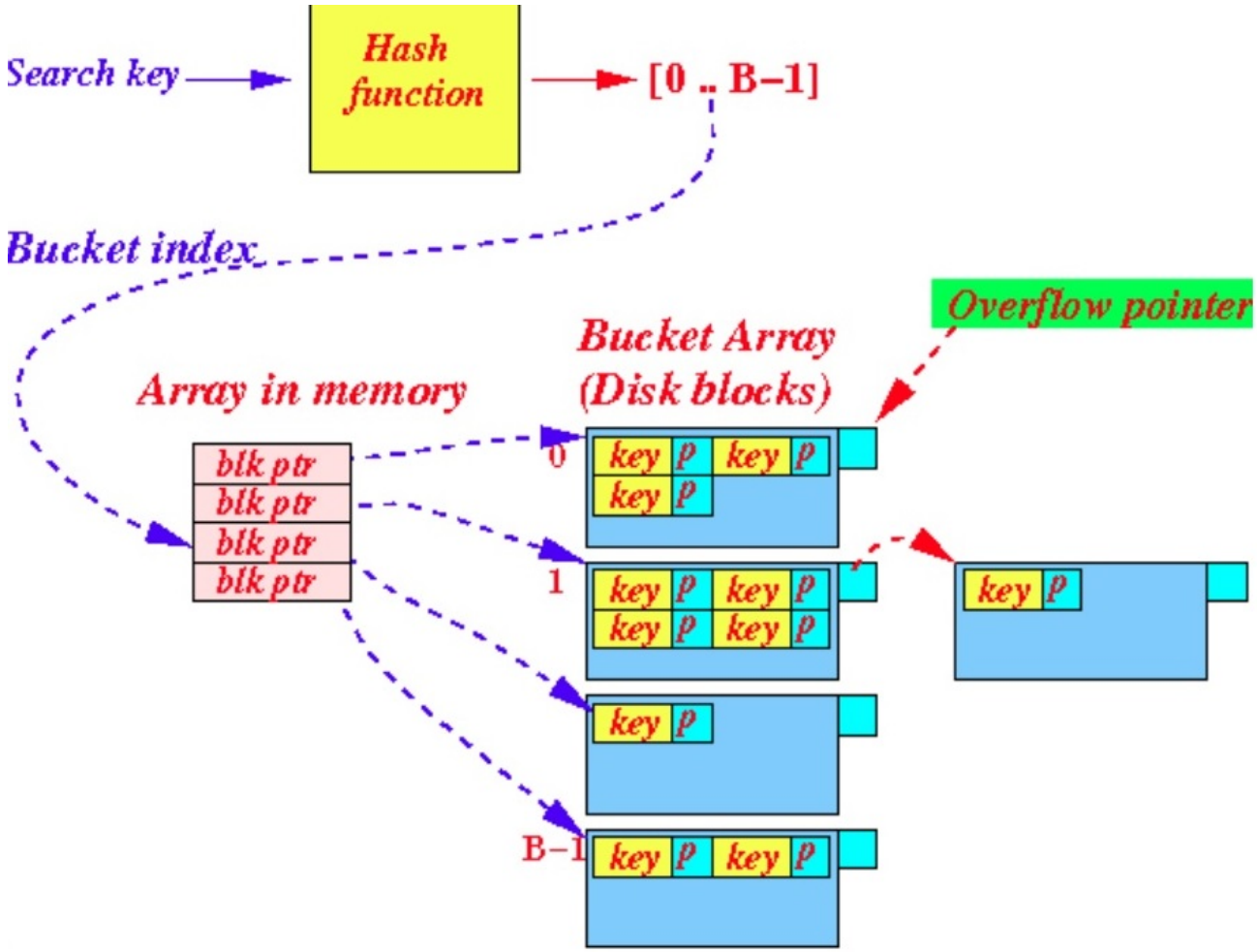
Storing Hash tables on disk

Logically, the Hash Table is stored as follows: We assume that the location of the first `block` for any `bucket i` can be found given `i`



Storing Hash tables on disk

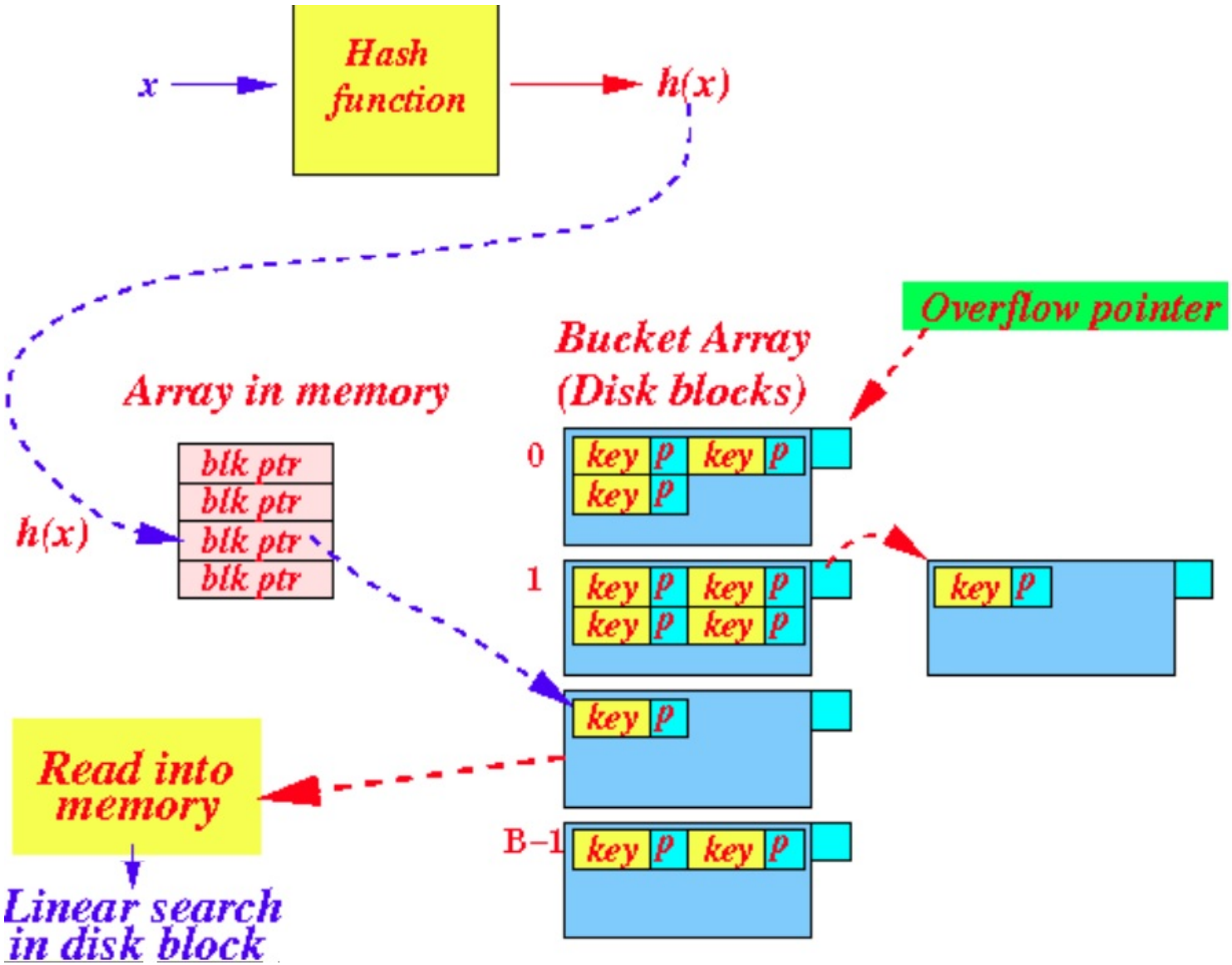
Physically, the Hash table is stored as follows: e.g., there might be a main-memory array of pointers to blocks, indexed by the bucket number



Using a Hash Index

- How to use a Hash index to access a record: Given a *search key* x:
  - Compute the hash value $h(x)$
  - Read the disk block pointed to by the block pointer in *bucket* $h(x)$ into memory
  - Search the *bucket* $h(x)$ for $(x, ptr(x))$
  - Use $ptr(x)$ to access x on disk

Using a Hash Index



Insertion into Static Hash Table

- To insert a record with *key k*:
  - compute *h(k)*
  - insert record *(k, recordPtr(k))* into one of the blocks in the chain of blocks for bucket number *h(k)*, adding a new block to the chain if necessary

Insertion: Example 2 records/bucket

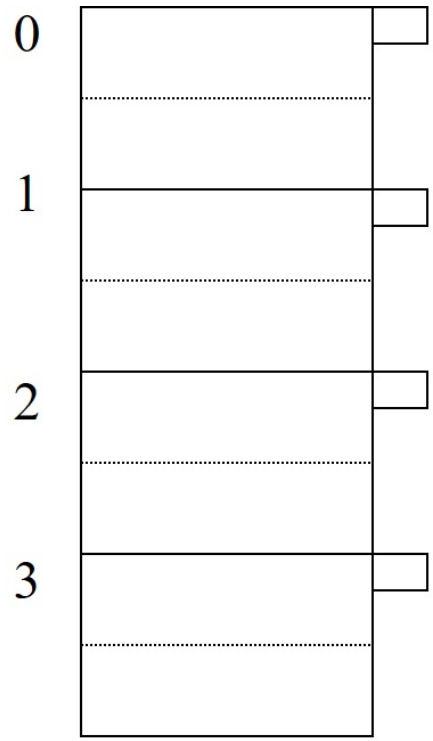
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



Insertion: Example 2 records/bucket

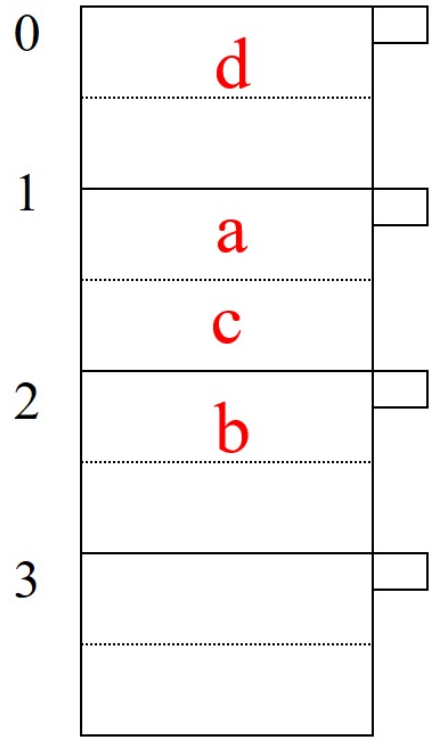
INSERT:

$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$



Deletion from a Static Hash Table

- To delete a record with *key* *K*:
  - Go to the `bucket` numbered *h(K)*
  - Search for records with *key* *K*, deleting any that are found
  - Possibly condense the chain of `overflow blocks` for that `bucket`

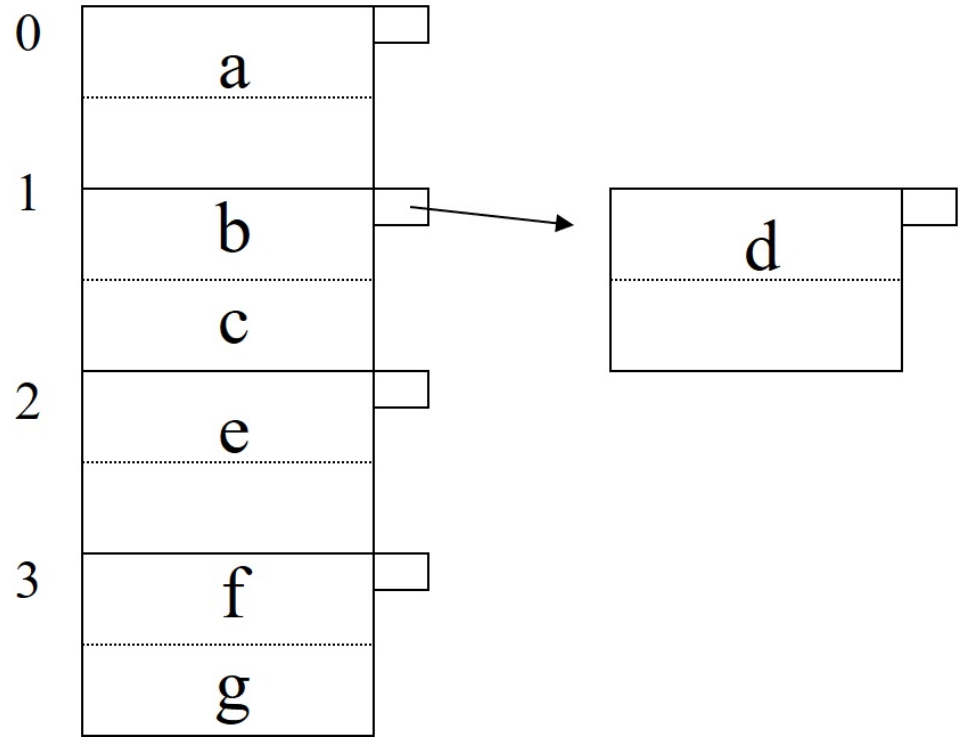
Deletion: Example 2 records/bucket

Delete:

e

f

c

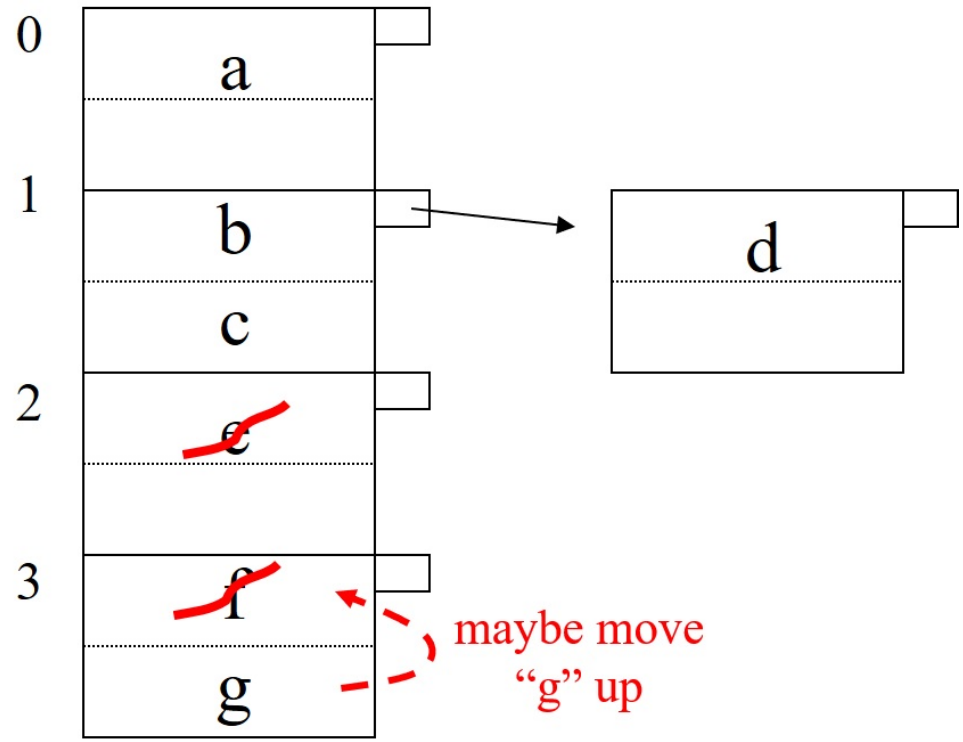


Deletion: Example 2 records/bucket

Delete:

e

f



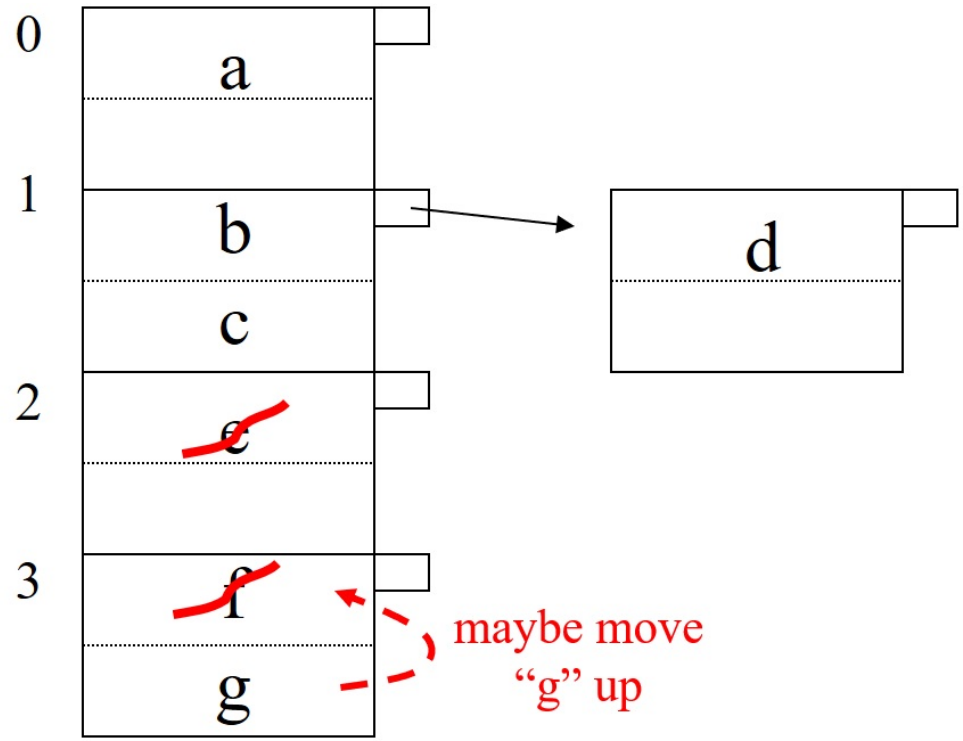
Deletion: Example 2 records/bucket

Delete:

e

f

c



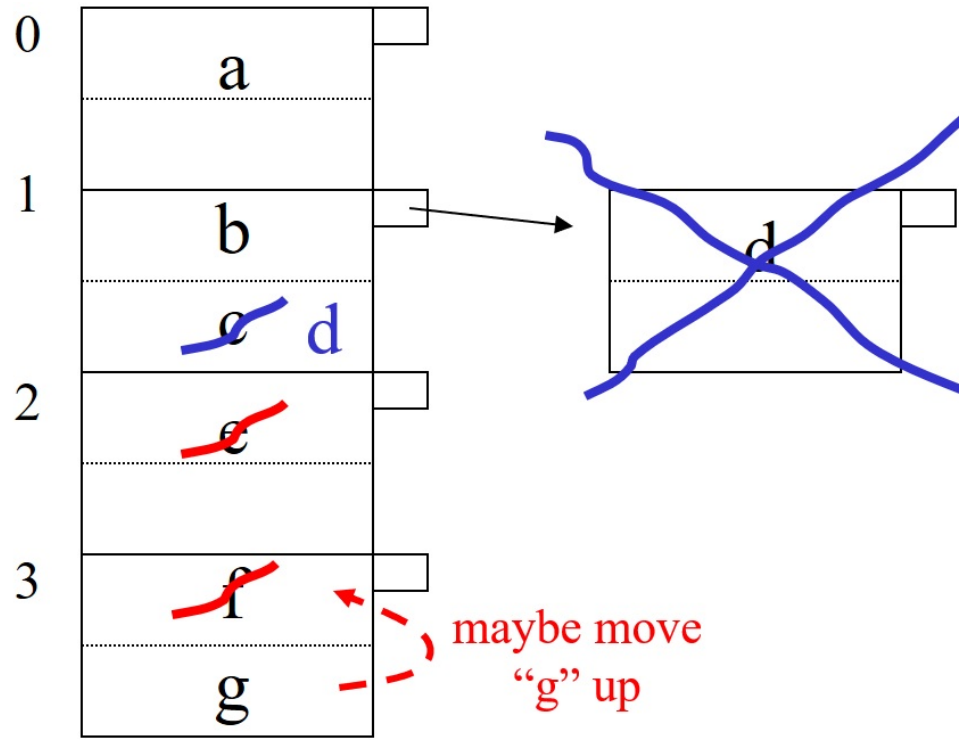
Deletion: Example 2 records/bucket

Delete:

e

f

c

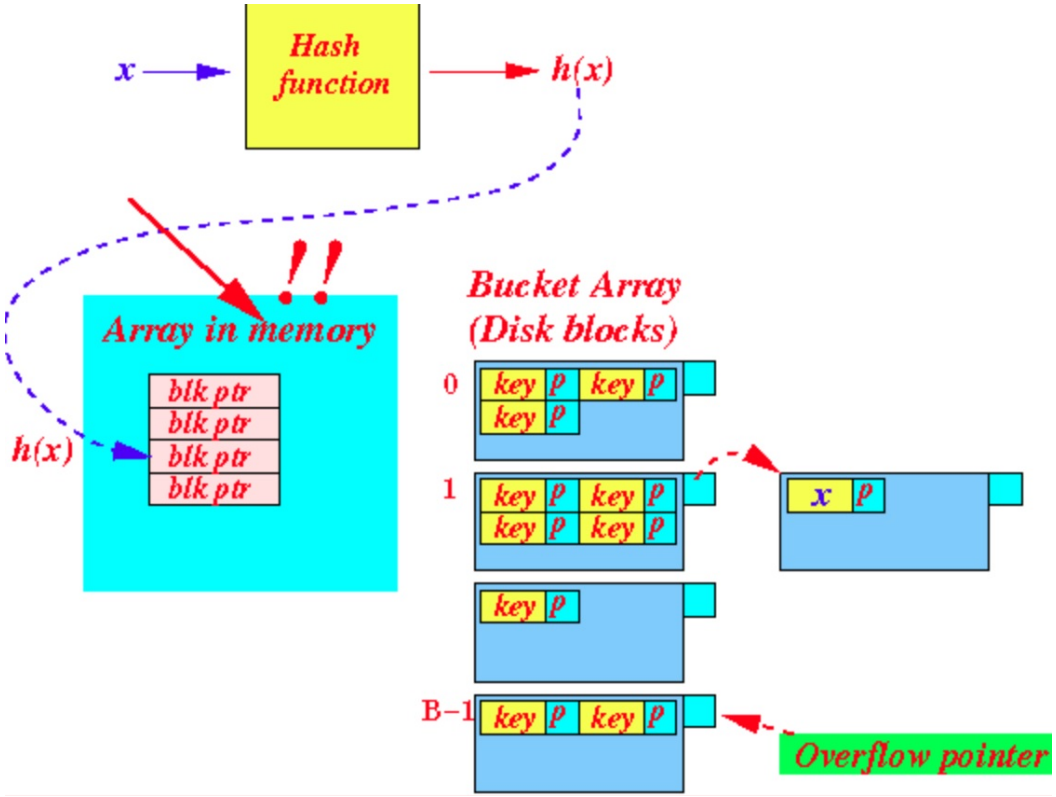


Rule of thumb

- Try to keep space utilization between 50% and 80%
- $\textit{Utilization} = \frac{\#\ \textit{keys}}{\textit{total}\ \#\ \textit{keys}\ \textit{that}\ \textit{fit}}$
- If $< 50\%$, wasting space
- If $> 80\%$, overflows significant
  - depends on how good hash function is and on # keys/bucket

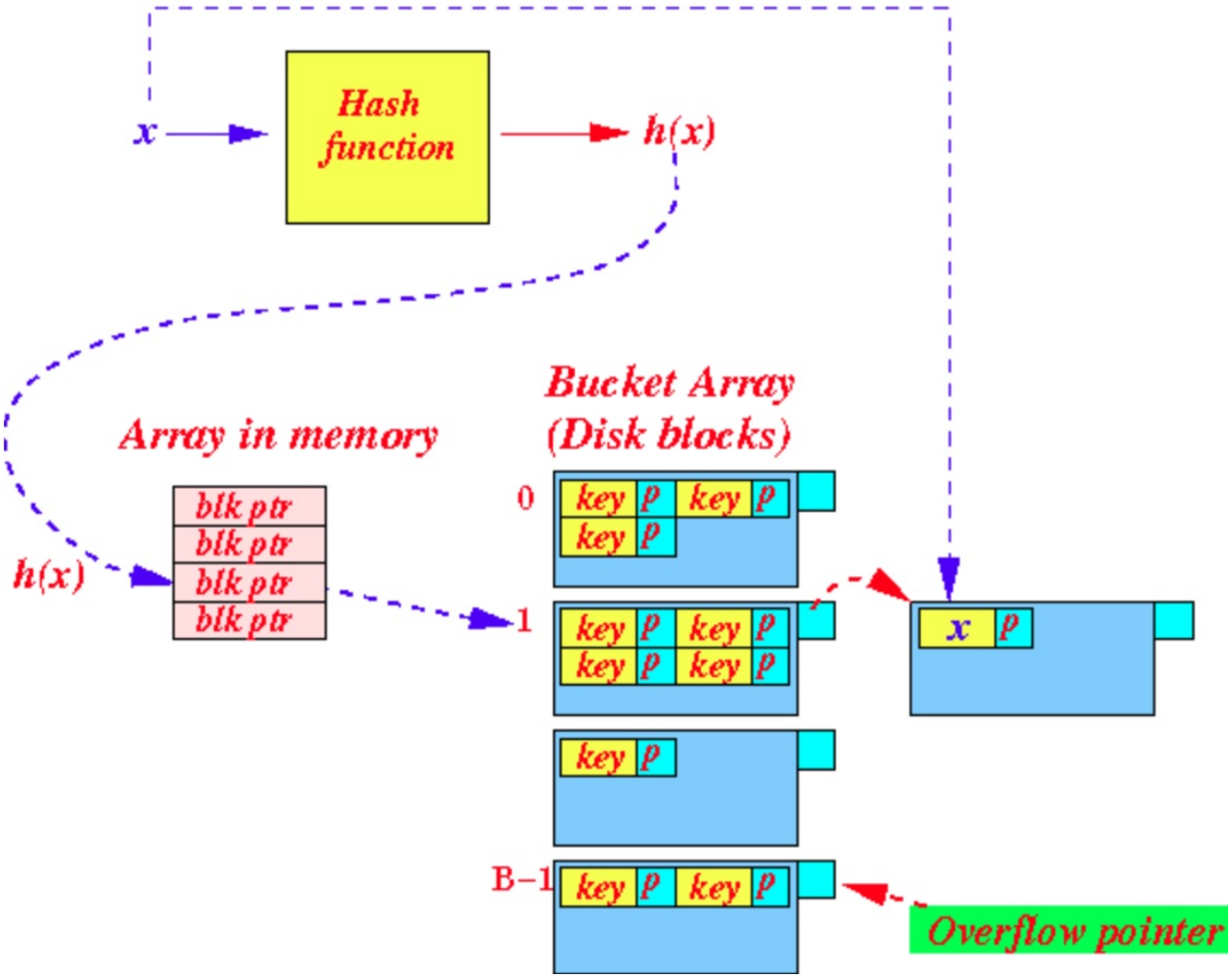
Performance of Static Hash Tables

- **fact**:
  - The array of block pointer is small enough to be stored entirely in main memory
  - Therefore, we disregard the access time to a block pointer



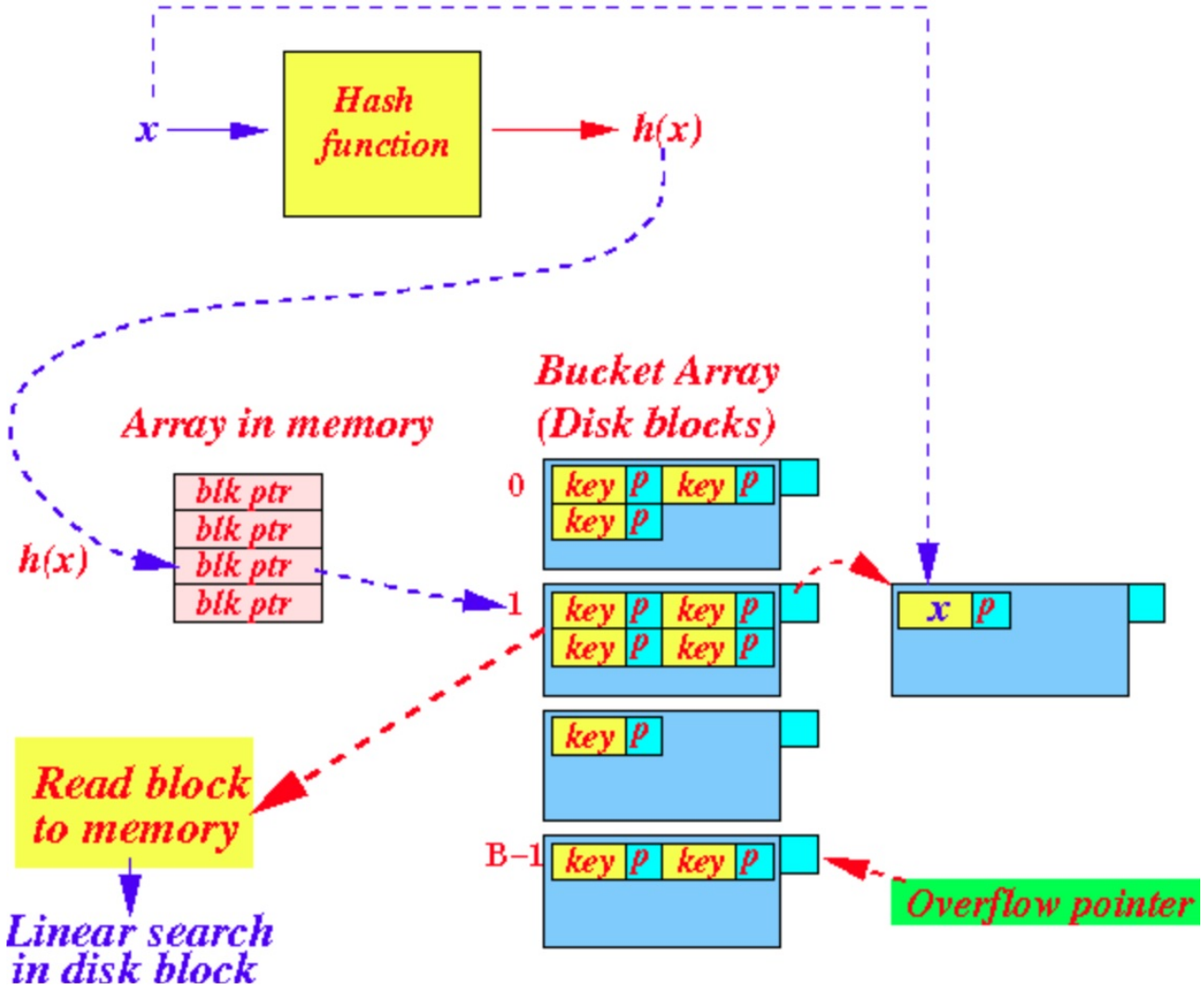
Performance of Static Hash Tables

- Suppose we look up using `key x`



Performance of Static Hash Tables

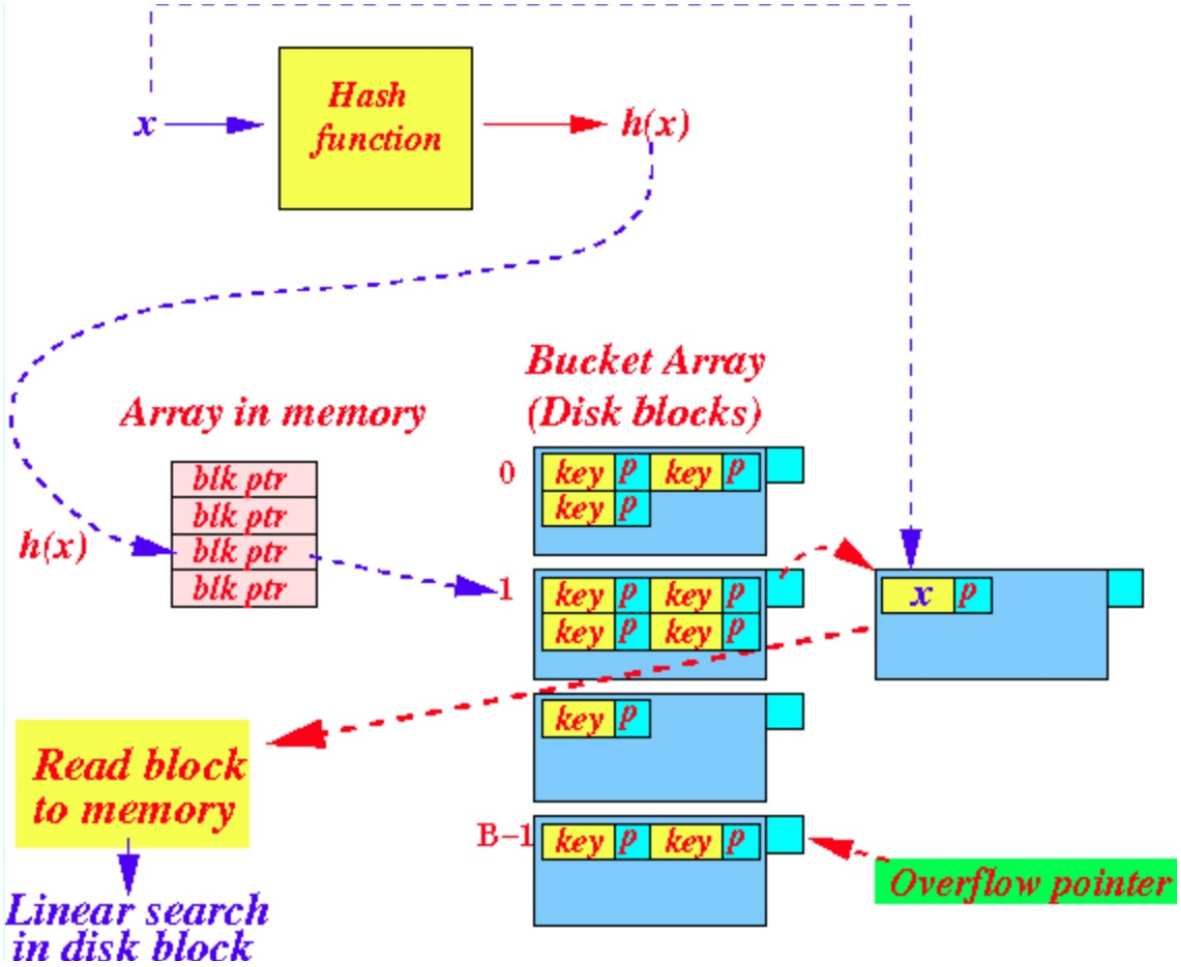
- We read in the `first index block` into the memory



- The search for `key x` fails

Performance of Static Hash Tables

- We read in the `next index block` into the memory



- The search for `key x` succeeds. We use the corresponding `block/record pointer` to access the data

Performance of Static Hash Tables

- Performance of a `hash index` depends of the number of `overflow blocks` used

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing (B is allowed to vary)
  - Extensible hashing
  - Linear

Problem with Hash Tables

- When many keys are inserted into the hash table, we will have many `overflow blocks`
  - `Overflow blocks` will require more `disk block read operations` and slow performance
- We can reduce the `# overflow blocks` by increasing the size `(B)` of the `hash table`
- The `Hash Table size (B)` is hard to change
  - Changing the hash table size will usually require "`Re-hashing`" all keys in the `hash table` into a new table size

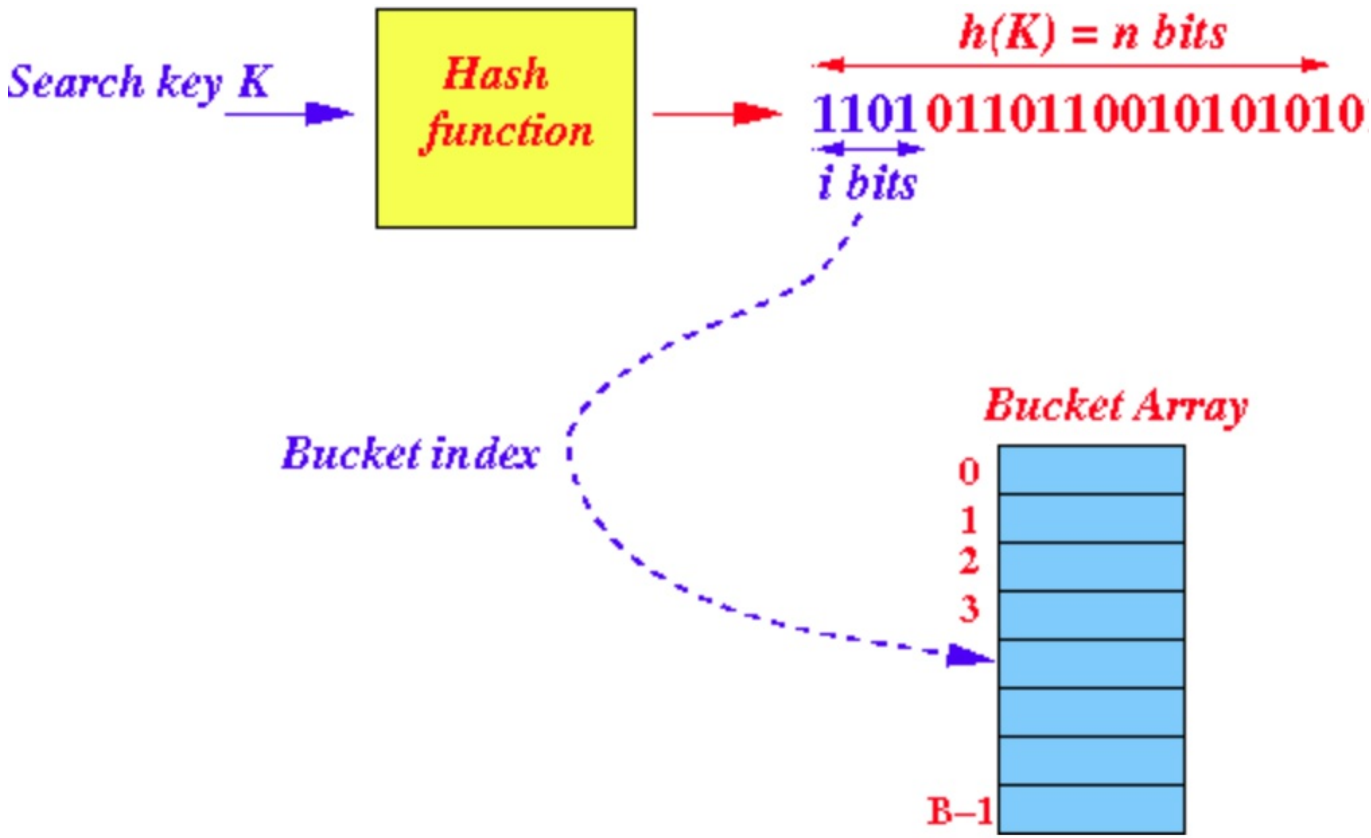
Dynamic hashing

- hashing techniques that allow the size of the `hash table` to change with relative low cost
  - Extensible hashing
  - Linear

Extensible Hash Tables

- Each `bucket` in the `bucket array` contains a `pointer to a block`, instead of a `block` itself
- `Bucket array` can grow by `doubling in size`
- Certain buckets can share a `block` if small enough
- `hash function` computes a sequence of n `bits`, but only **first** i **bits** are used at any time to index into the `bucket array`
- Value of i can increase (corresponds to `bucket array` doubling in size)

Hash Function used in Extensible Hashing



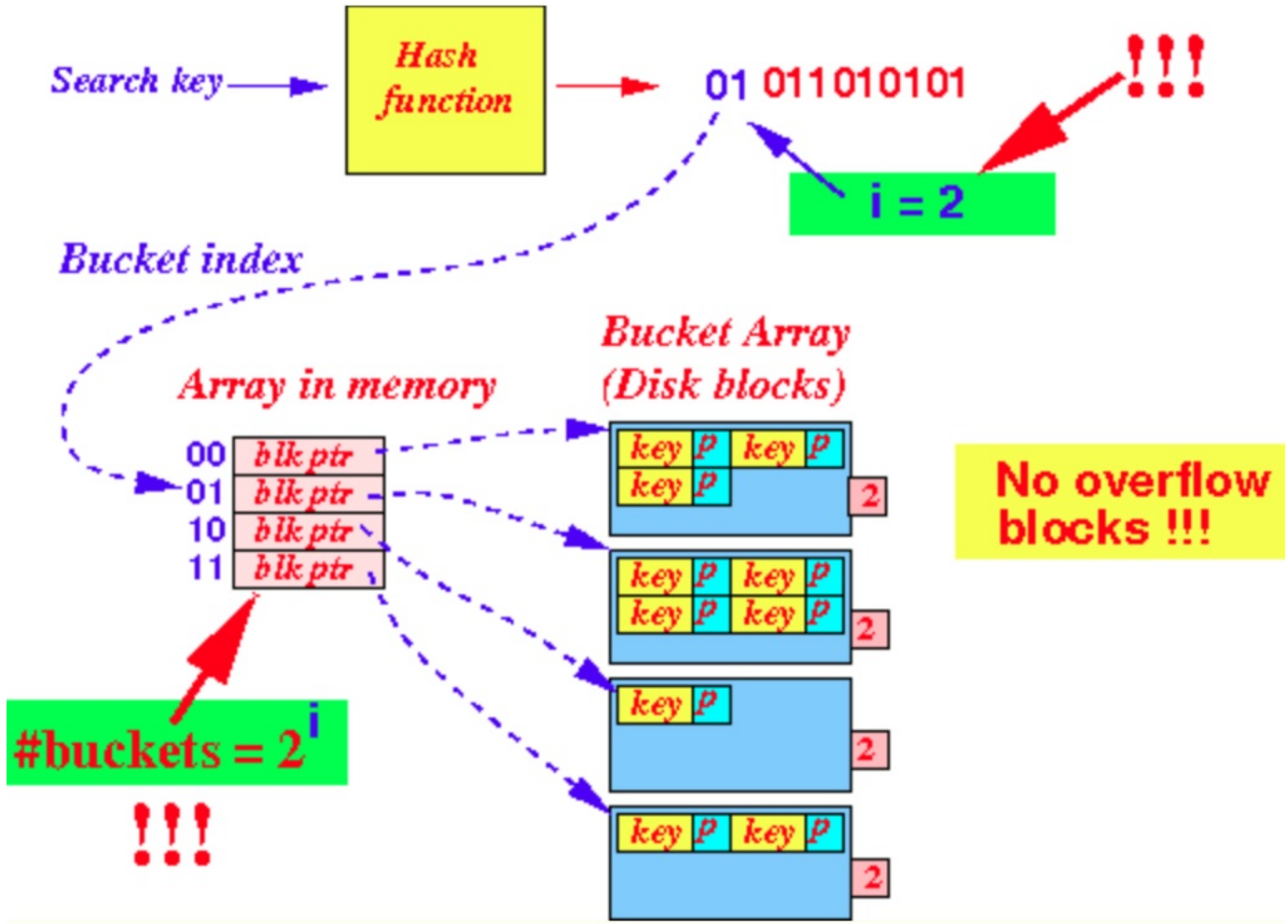
- The `bucket index` consists of the **first** *i* **bits** in the `hash function` value
  - The number of `bits` *i* is dynamic. (You can use the **last** *i* **bits** instead of the **first** *i* **bits**)

New things in extensible hashing

- Each `bucket` consists of:
  - Exactly 1 `disk block`. (there are no `overflow blocks`)
- Each bucket (disk block) contains an integer indicating
  - The number bits of the `hash function value` used to hash the `search keys` into the `bucket`

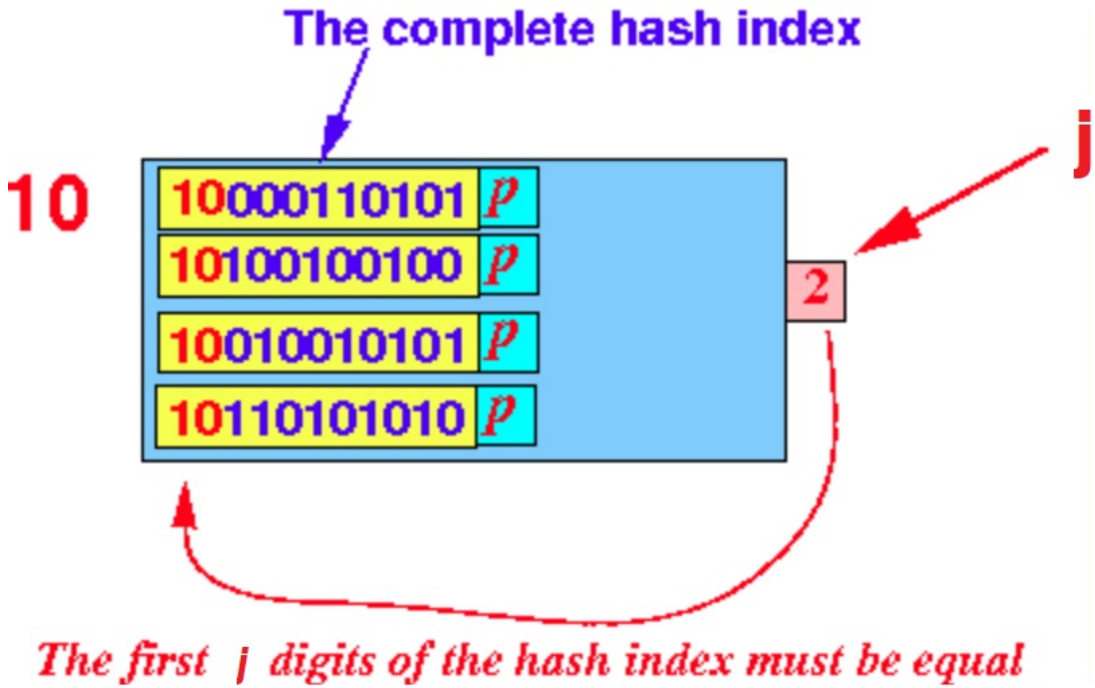
Parameters used in Extensible Hashing

- There are 2 integers used in Extensible Hashing
  1) Global parameter i: the number of bits used in the hash (key) to lookup a (hash) bucket
     - Control the number of buckets (2^i) of the hash index



Parameters used in Extensible Hashing

2) The bucket label parameter j: number of bits of hash value used to determine membership in a Bucket



- The following property holds:
  global parameter $i \geq$ any bucket label parameter j

Inserting into Extensible Hash Table

- To insert record with `key K`
  - compute `h(K)`
  - go to `bucket indexed by first` i `bits of h(K)`
  - follow the pointer to get to a `block B`
  - if there is a room in `B`, insert record. Done
  - else, there are two possibilities, depending on the number j
    - `Case 1:` $j < i$
    - `Case 2:` $j = i$

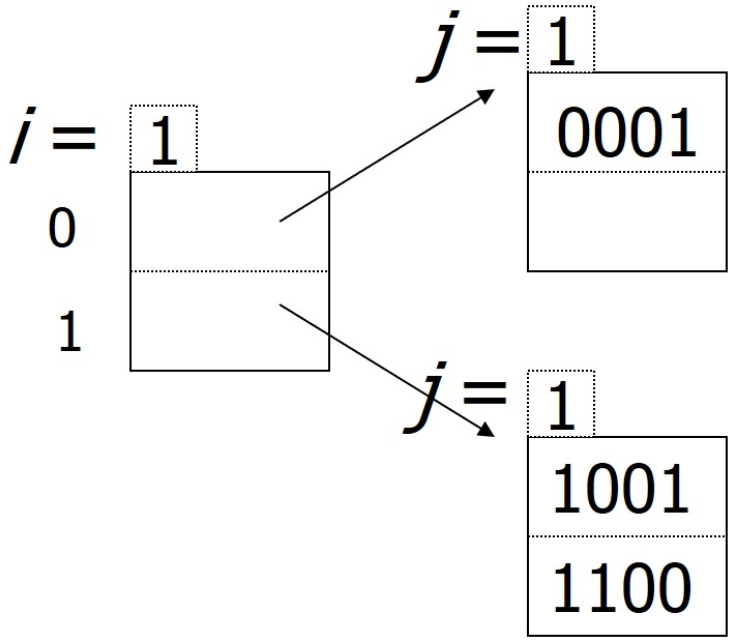
Insertion: Case 1: $j < i$

- split `block B` in two
- distribute records in `B` to the 2 new blocks based on value of their $(j+1)$`-st bit`
- update header of each new block to $j+1$
- adjust pointers in `bucket array` so that entries that used to point to `B` now point either to `B` or the new `block`, depending on their $j+1$`-st bit`
- if still no room in appropriate `block` for new record then `repeat` this process

Insertion: `Case 2:` $j = i$

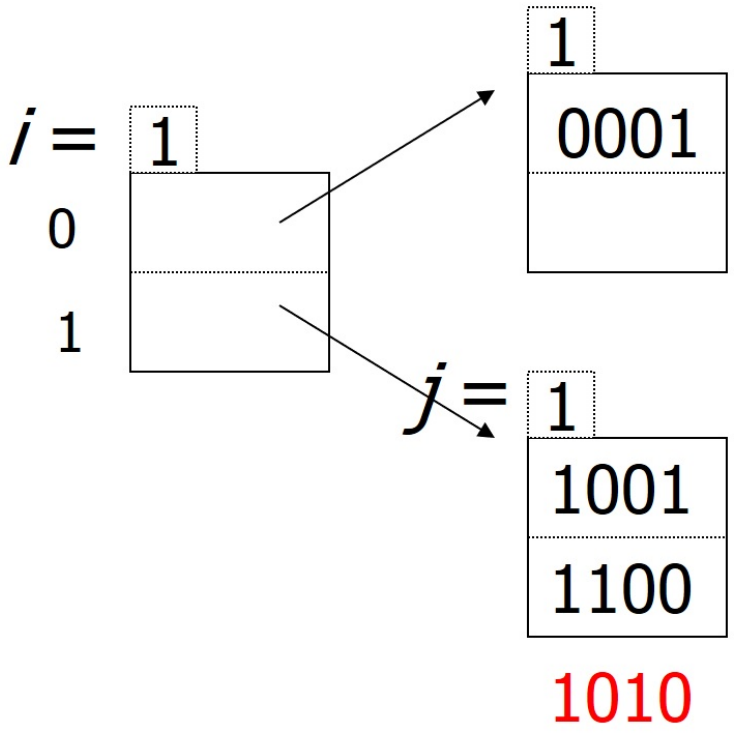
- increment `i` by 1
- double length of `bucket array`
- in the new bucket array, entry indexed by both $w0$ and $w1$ each point to same `block` that old entry w pointed to (`block` is shared)
- apply `case 1` to split `block B`

Example: h(k) is 4 bits; 2 keys/bucket



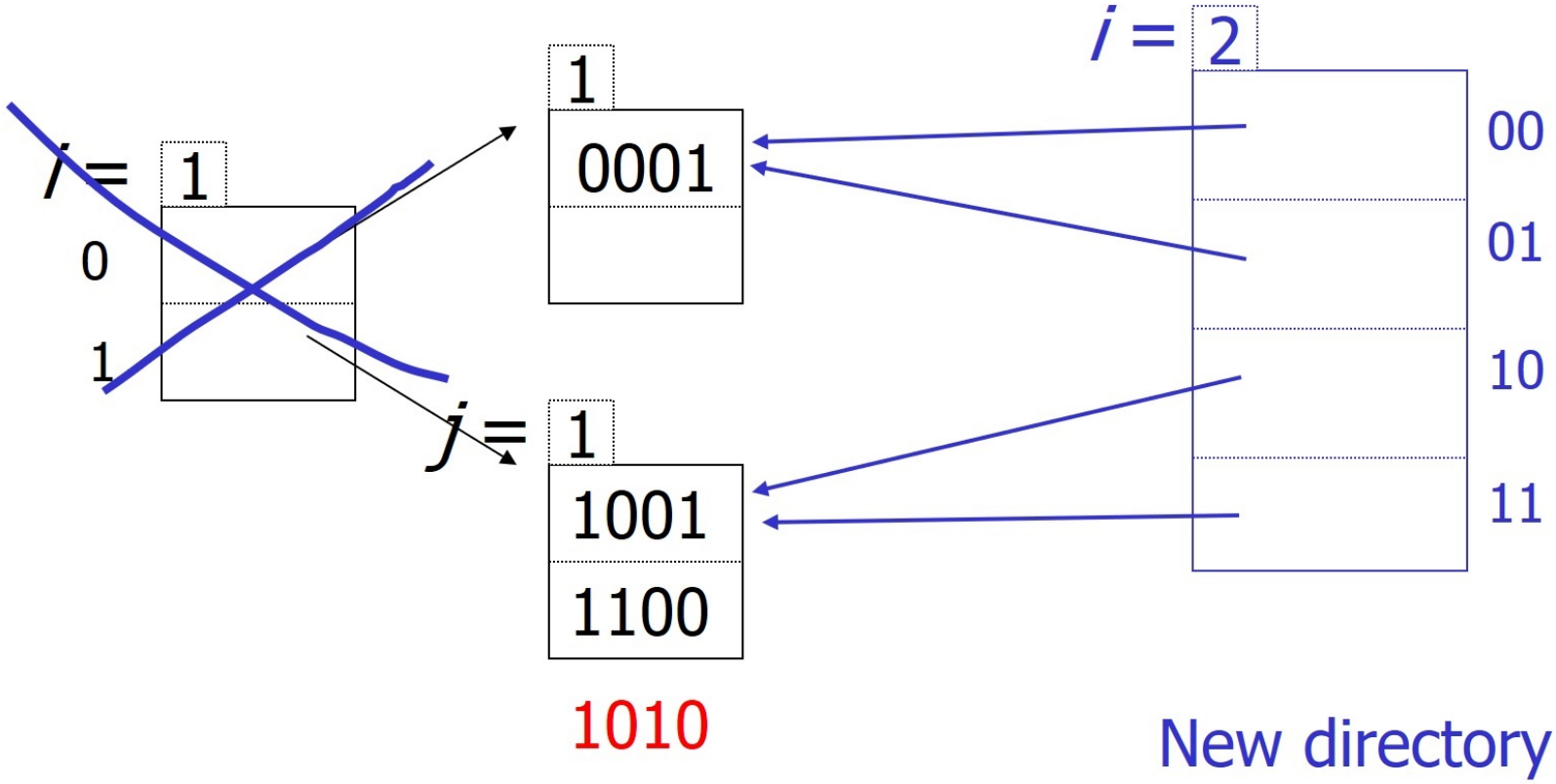
Insert h(key)=1010

Example: h(k) is 4 bits; 2 keys/bucket



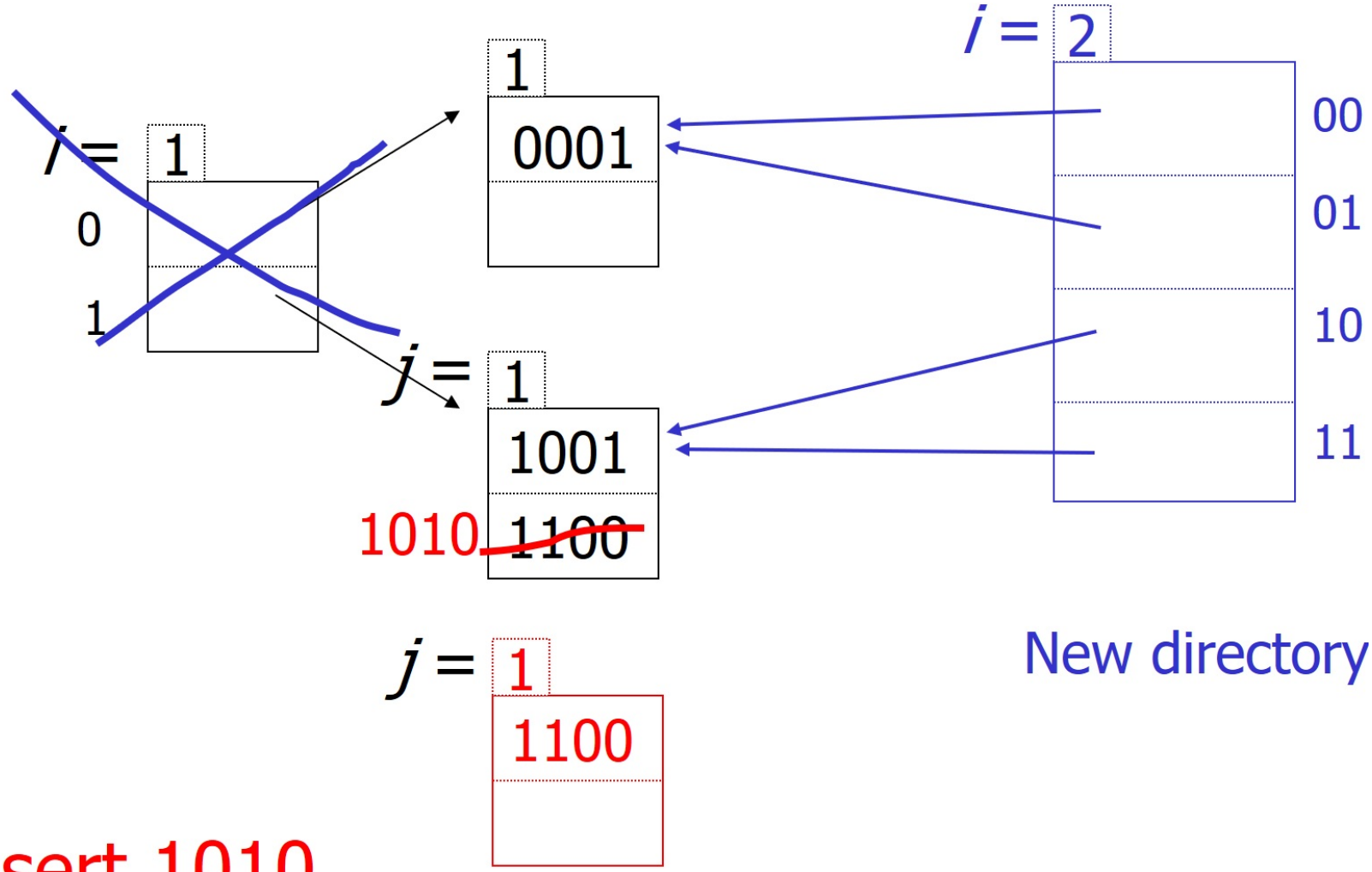
Insert h(key)=1010

Example: h(k) is 4 bits; 2 keys/bucket

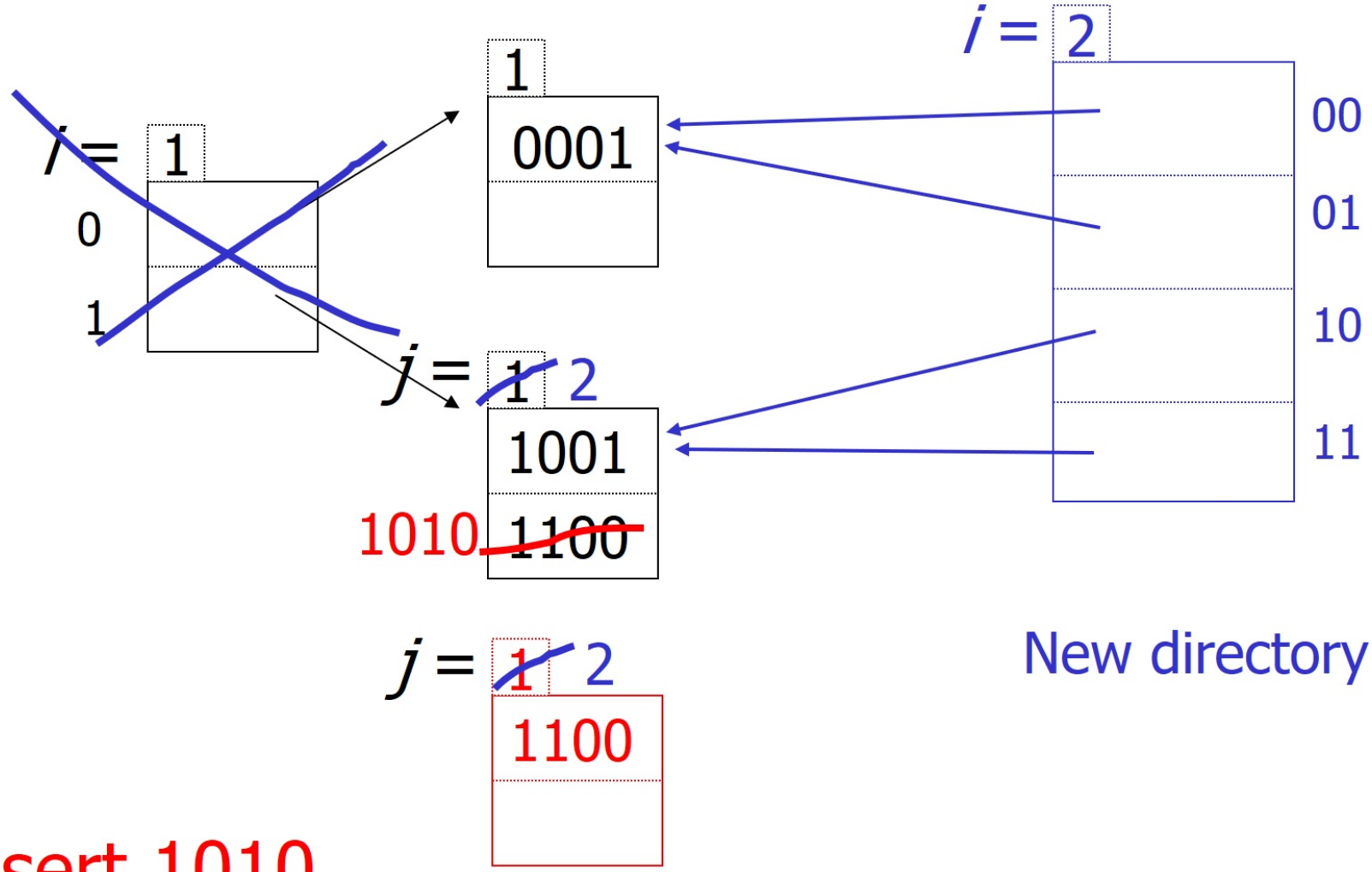


Insert h(key)=1010

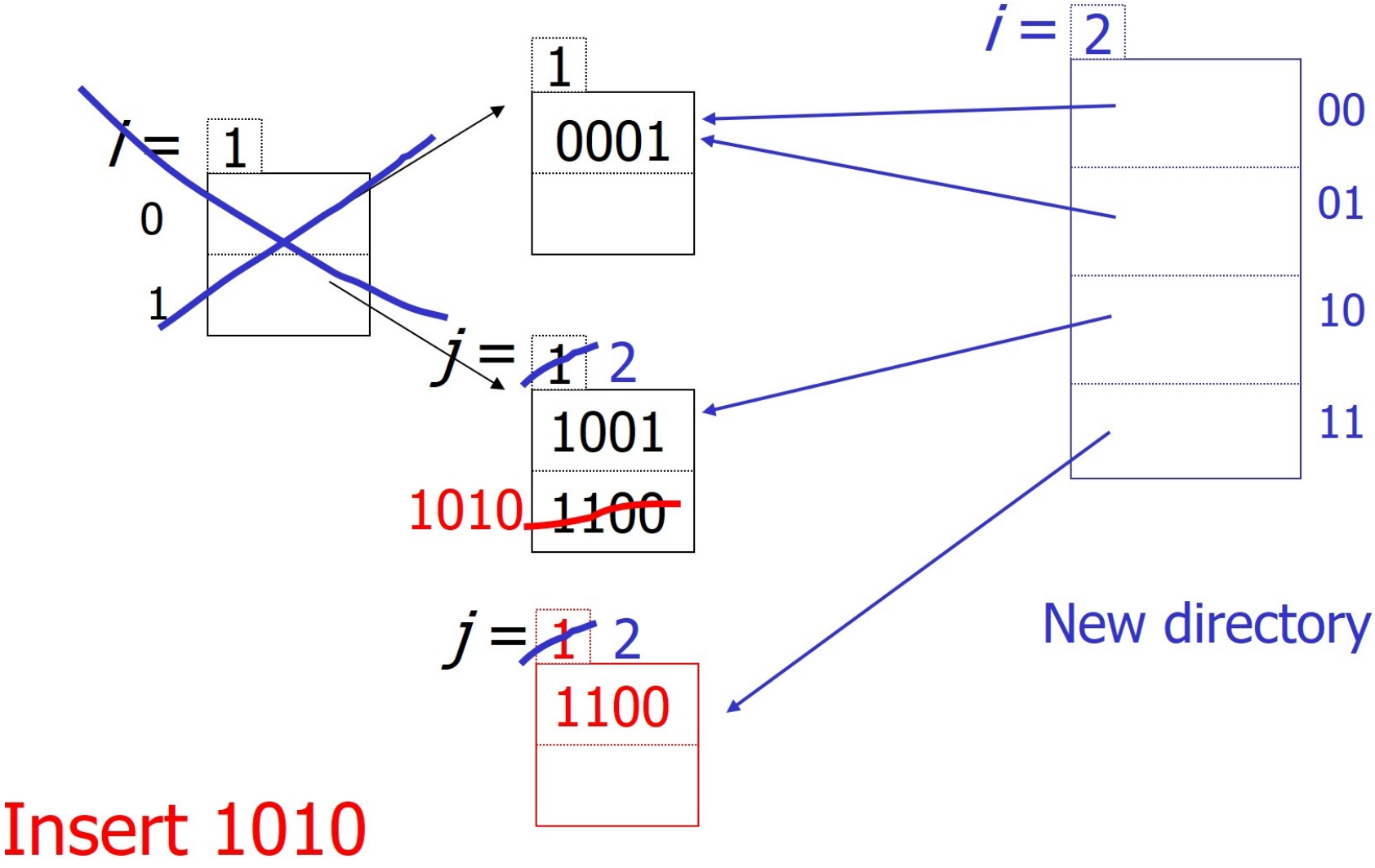
Example: h(k) is 4 bits; 2 keys/bucket



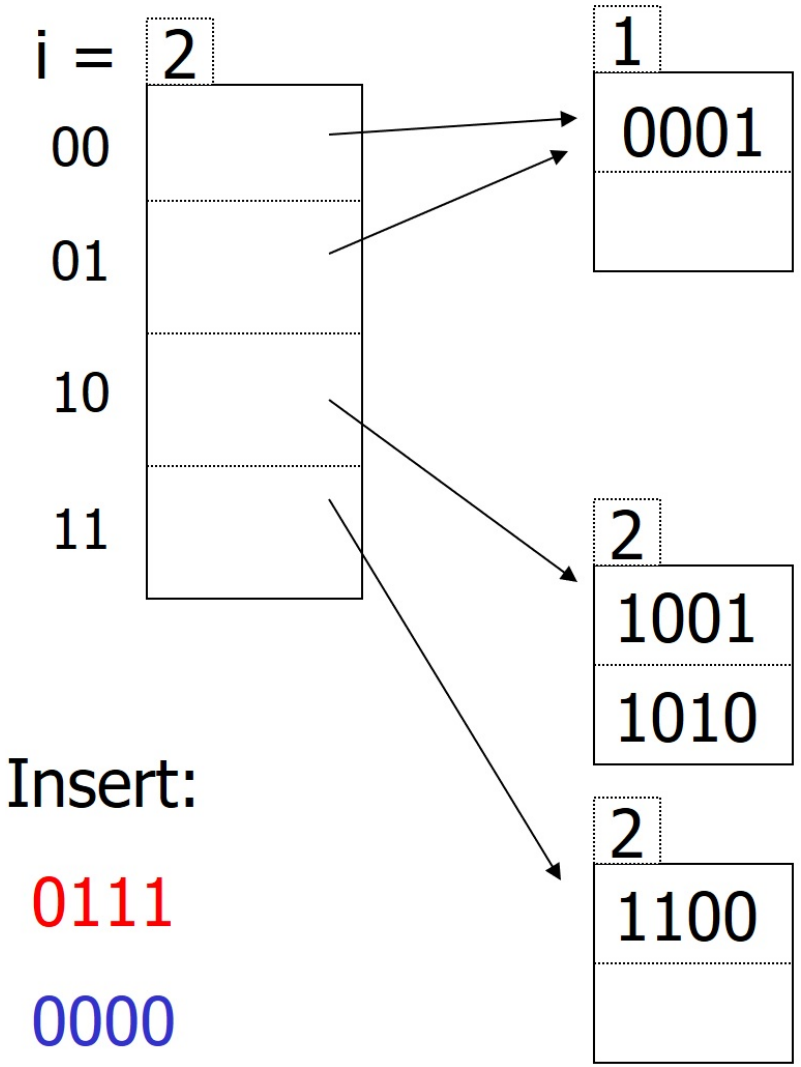
Example: h(k) is 4 bits; 2 keys/bucket



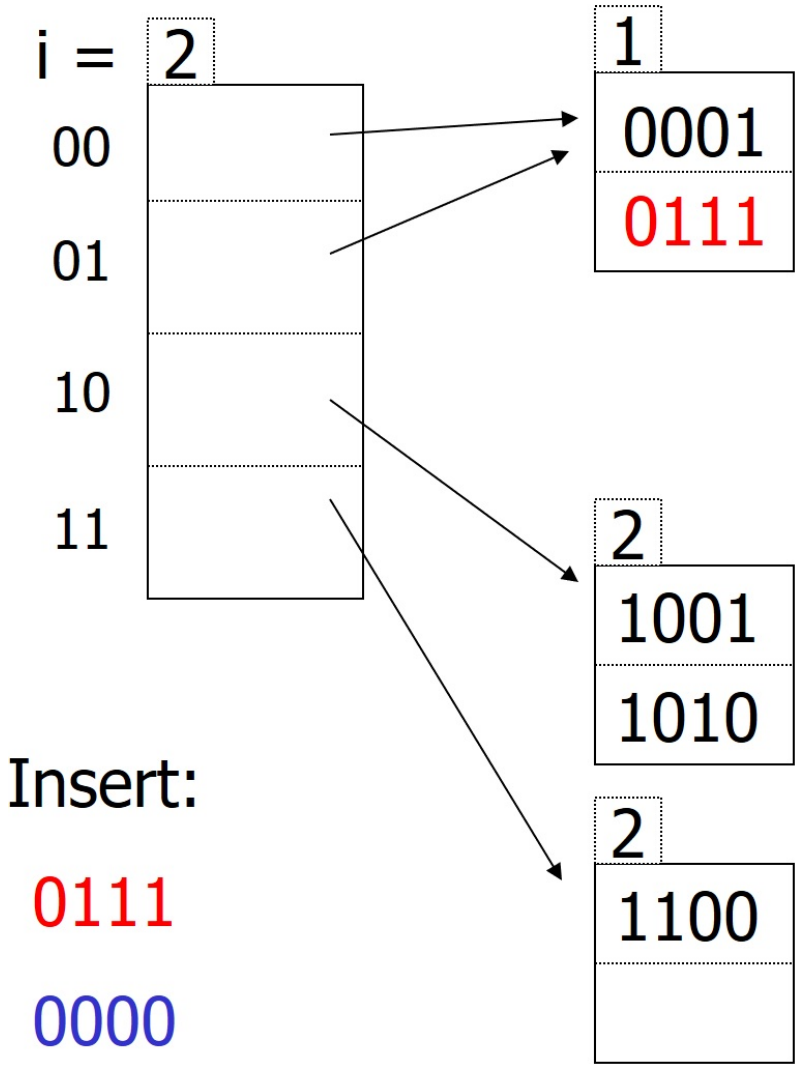
Example: h(k) is 4 bits; 2 keys/bucket



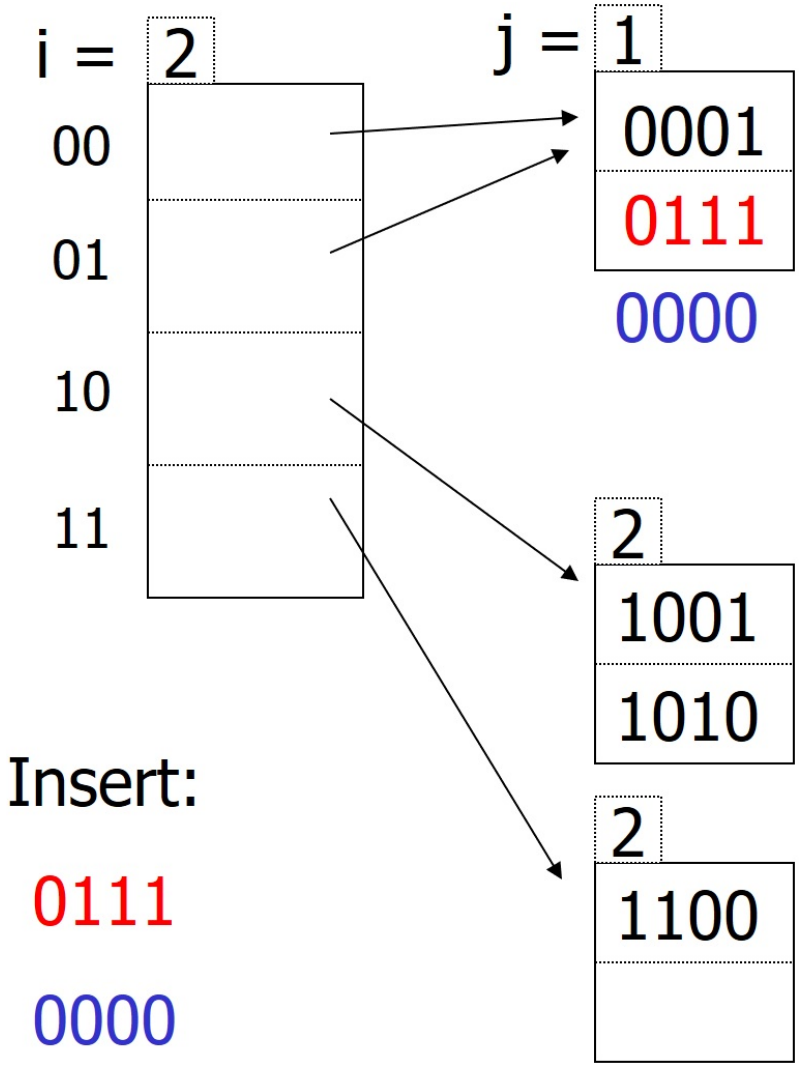
Example: h(k) is 4 bits; 2 keys/bucket



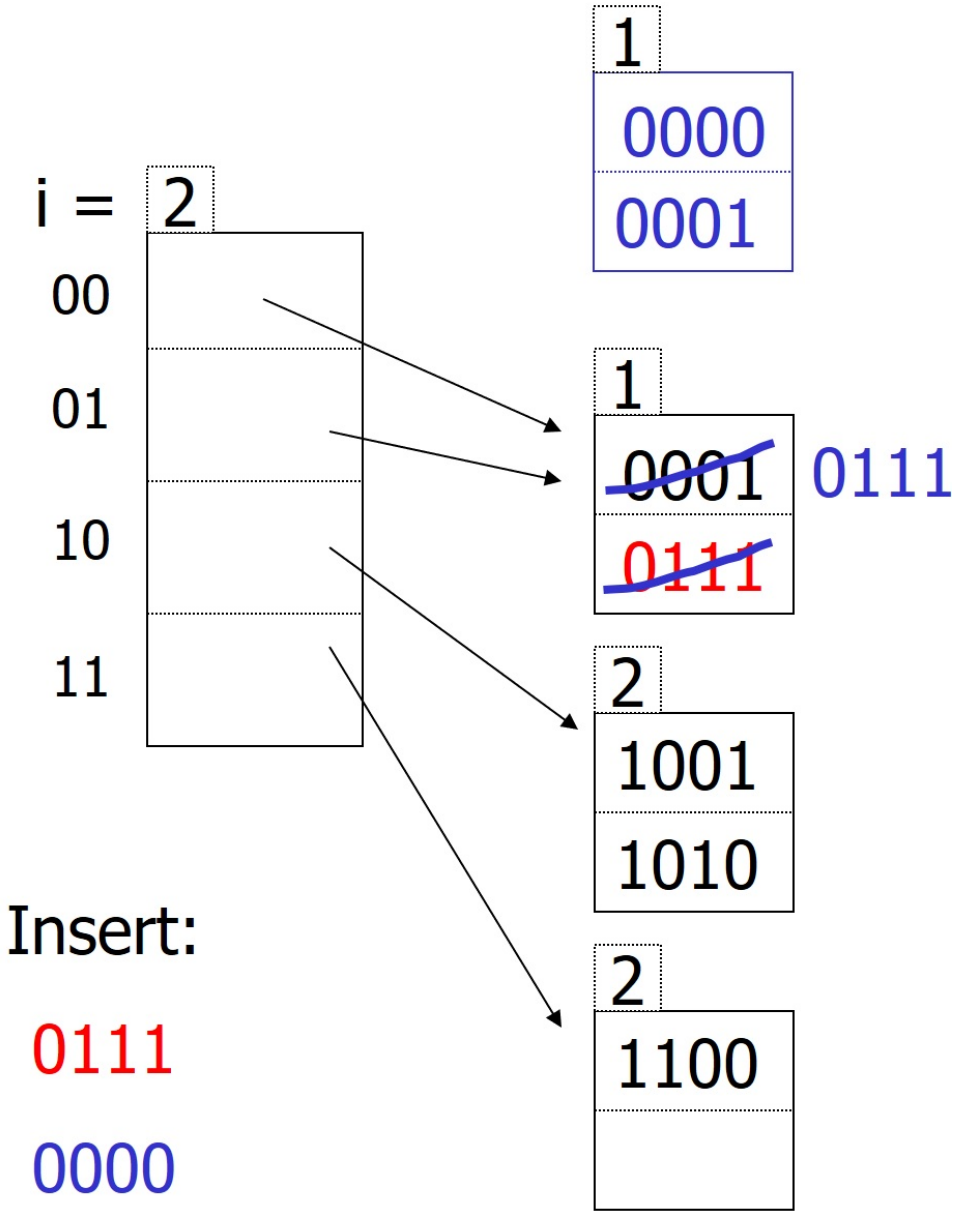
Example: h(k) is 4 bits; 2 keys/bucket



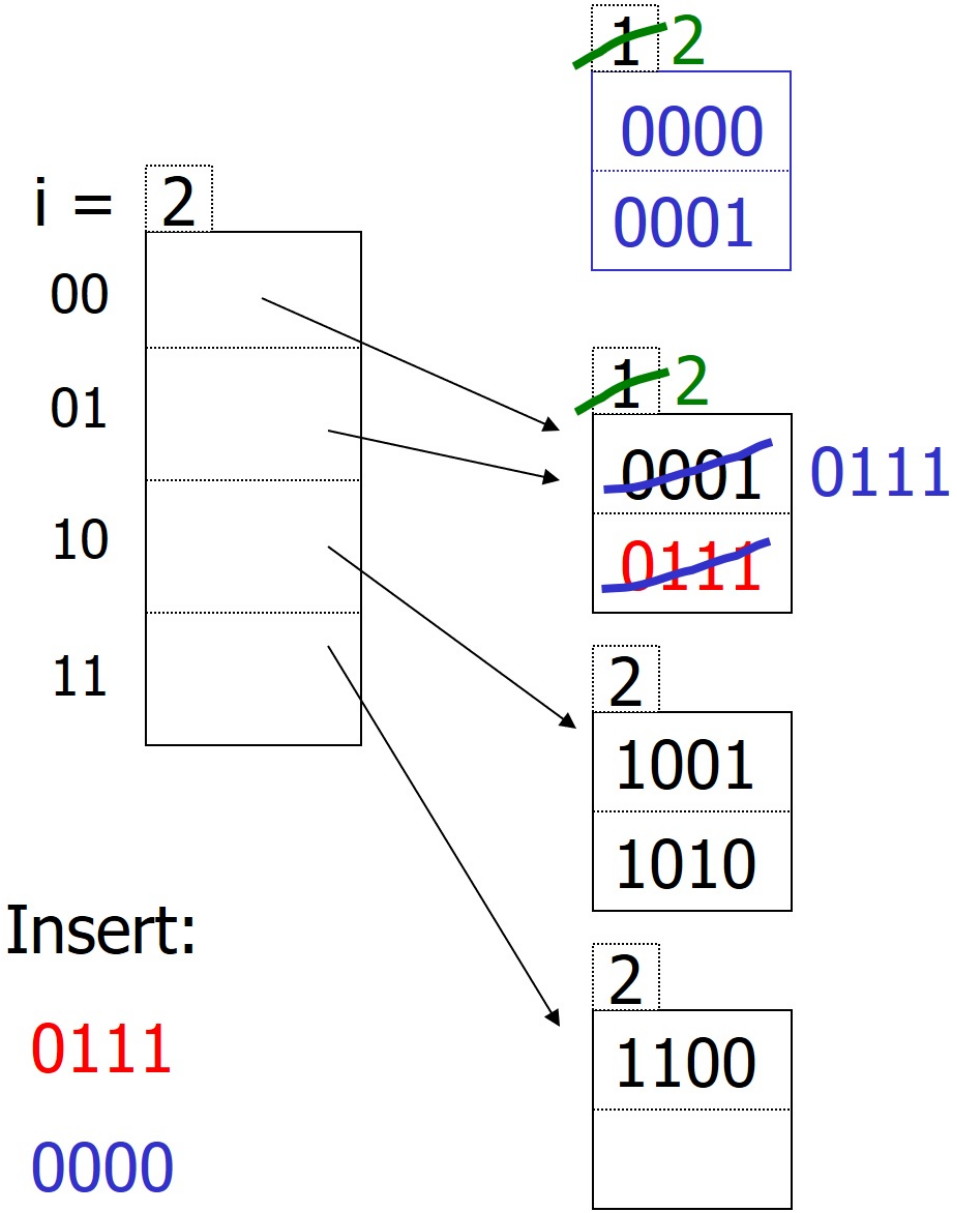
Example: h(k) is 4 bits; 2 keys/bucket



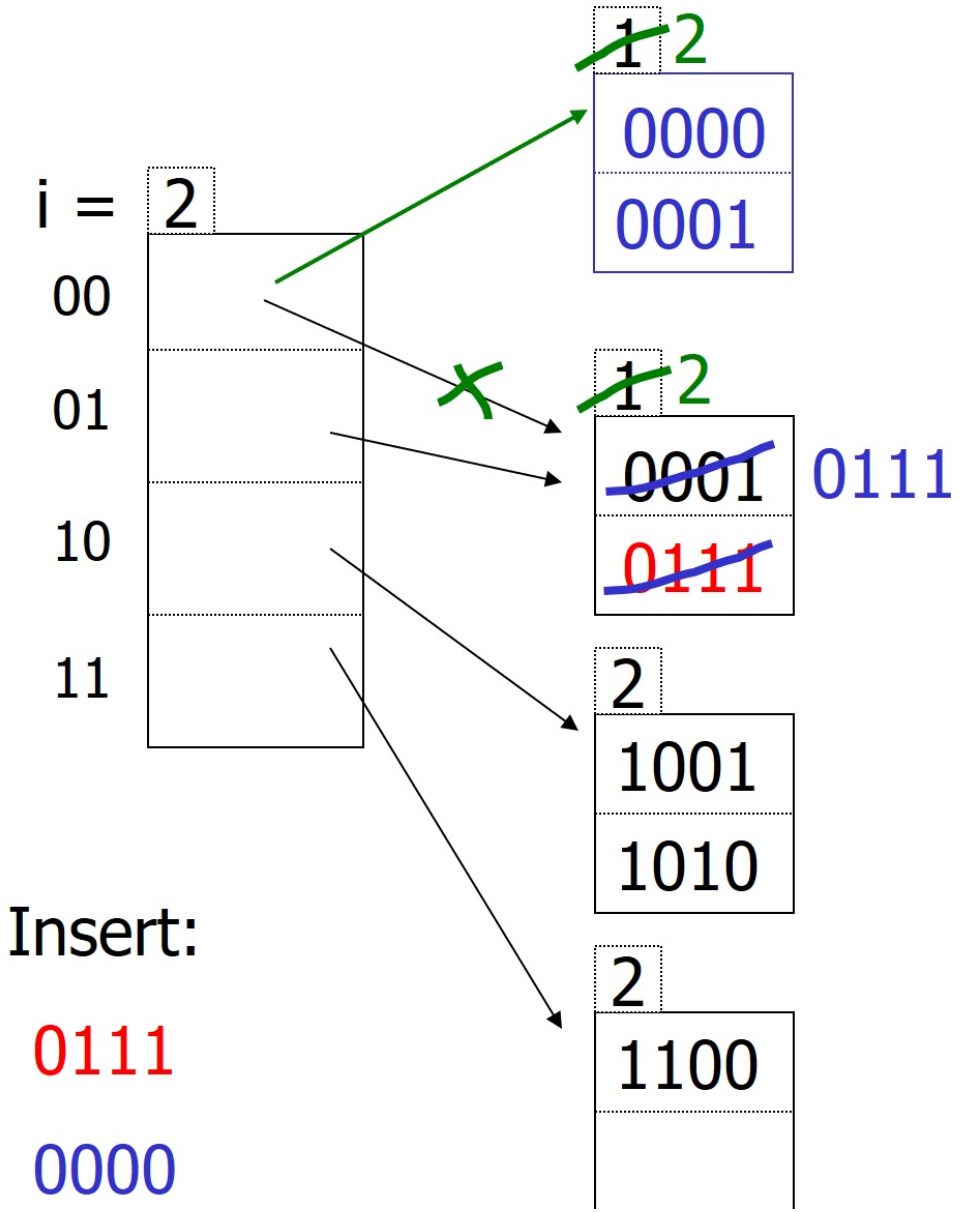
Example: h(k) is 4 bits; 2 keys/bucket



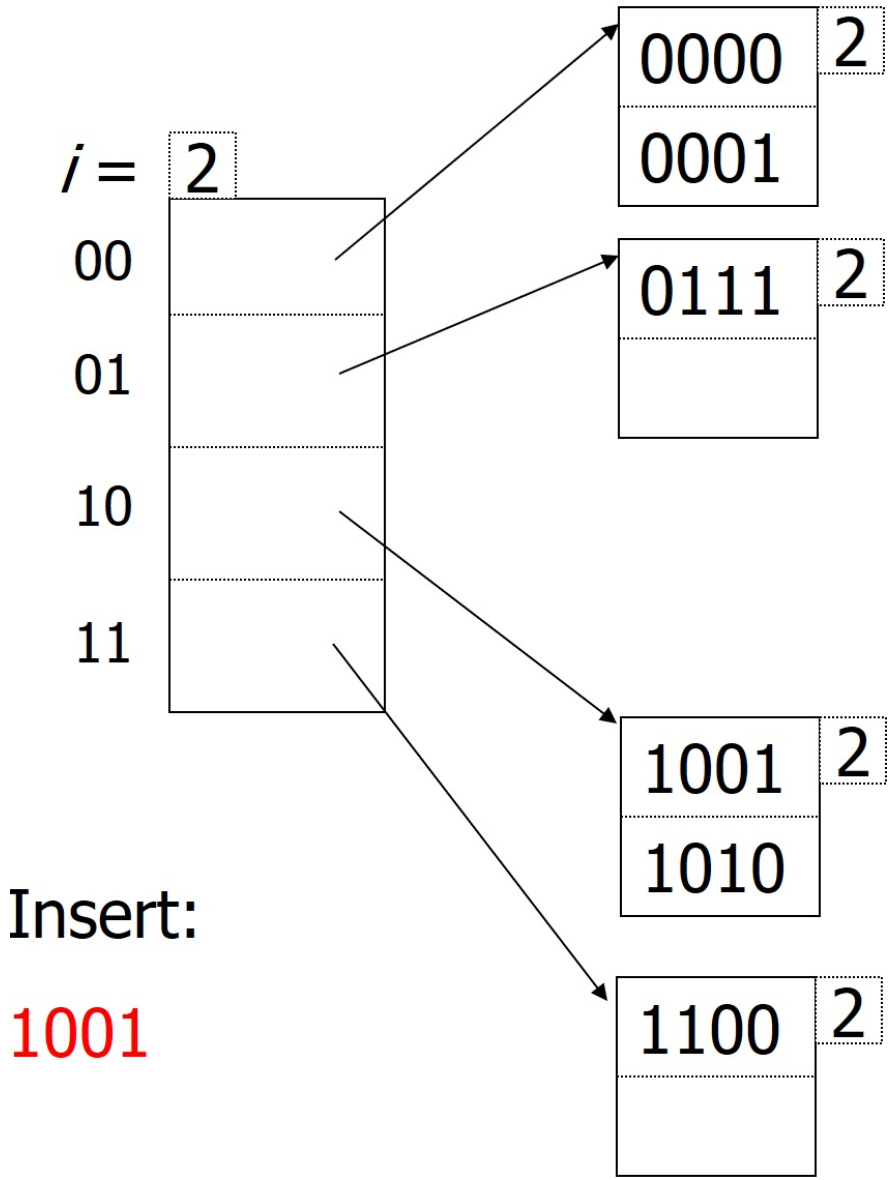
Example: h(k) is 4 bits; 2 keys/bucket



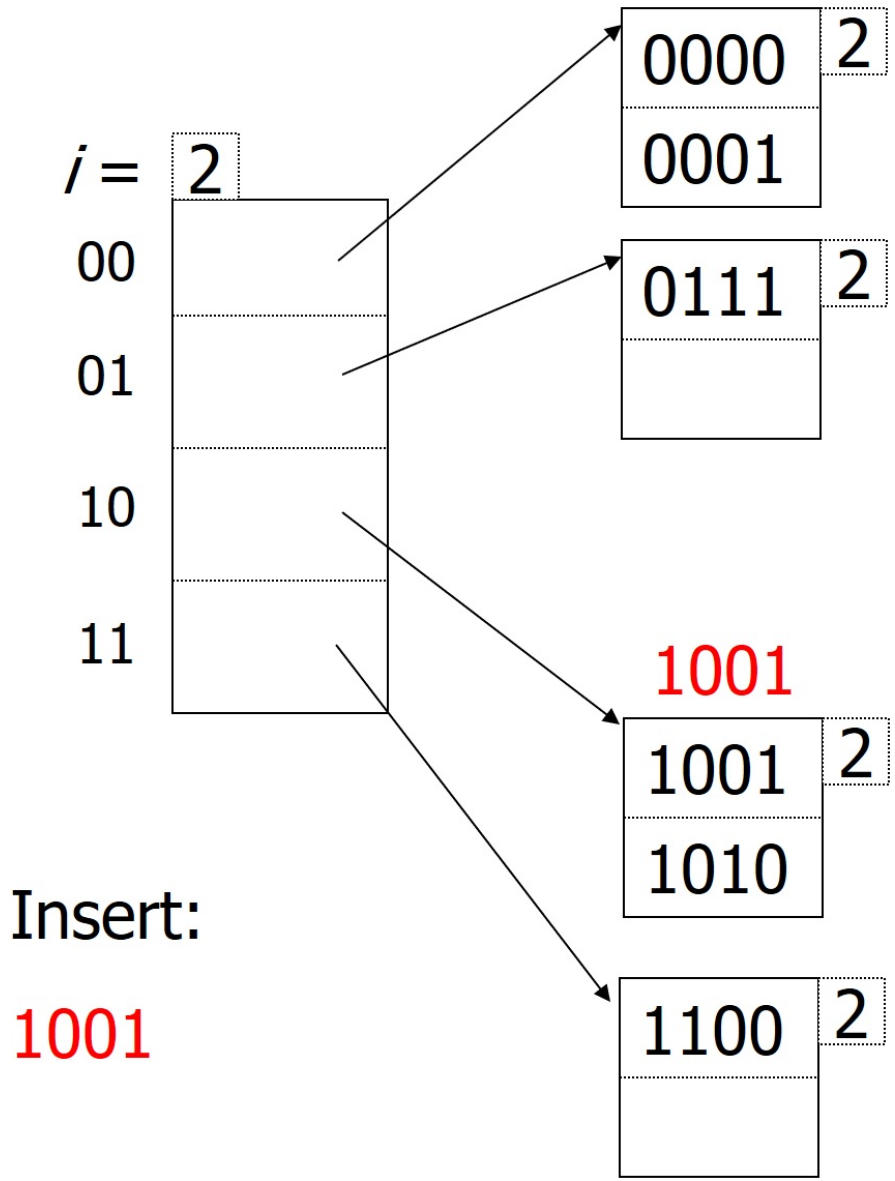
Example: h(k) is 4 bits; 2 keys/bucket



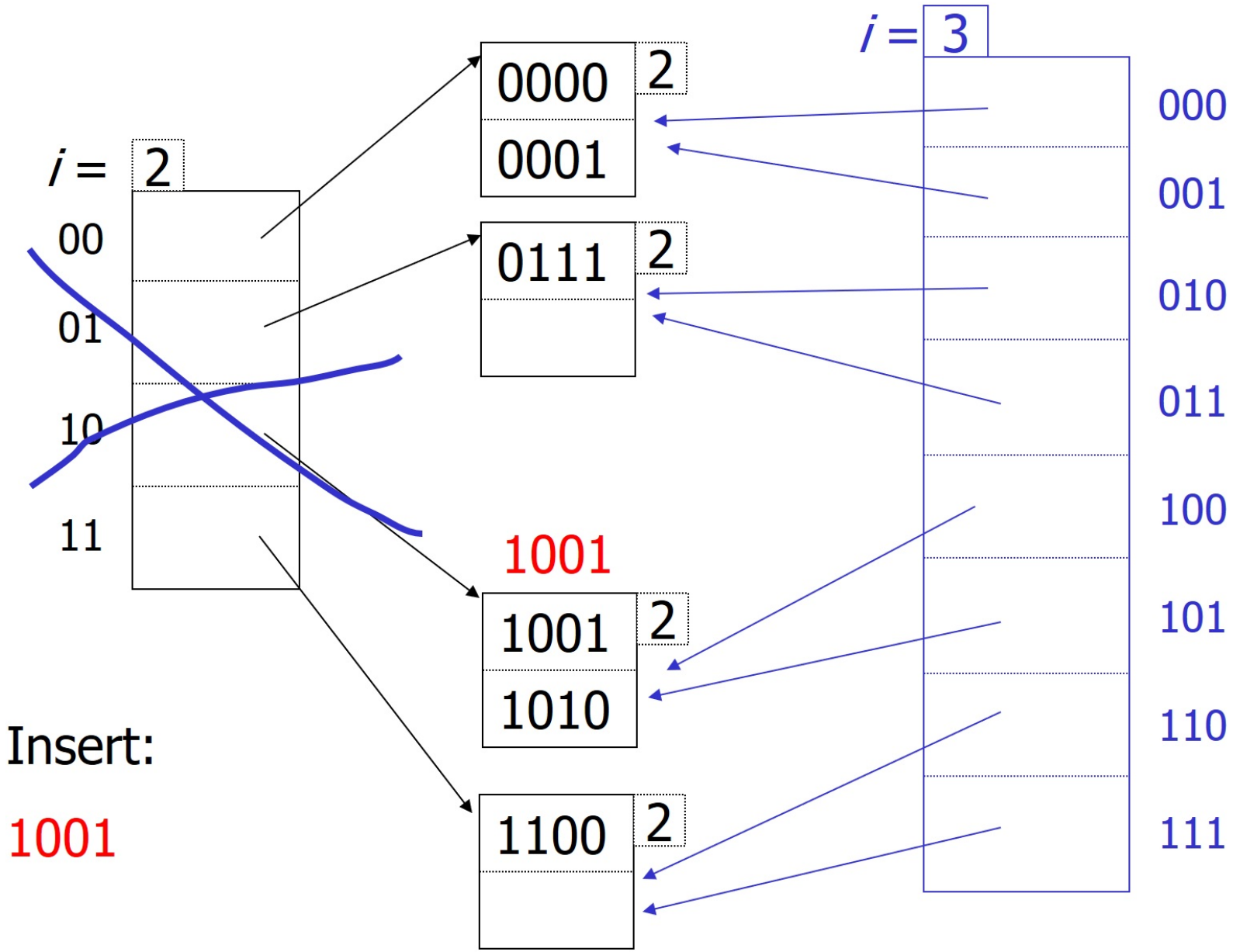
Example: h(k) is 4 bits; 2 keys/bucket



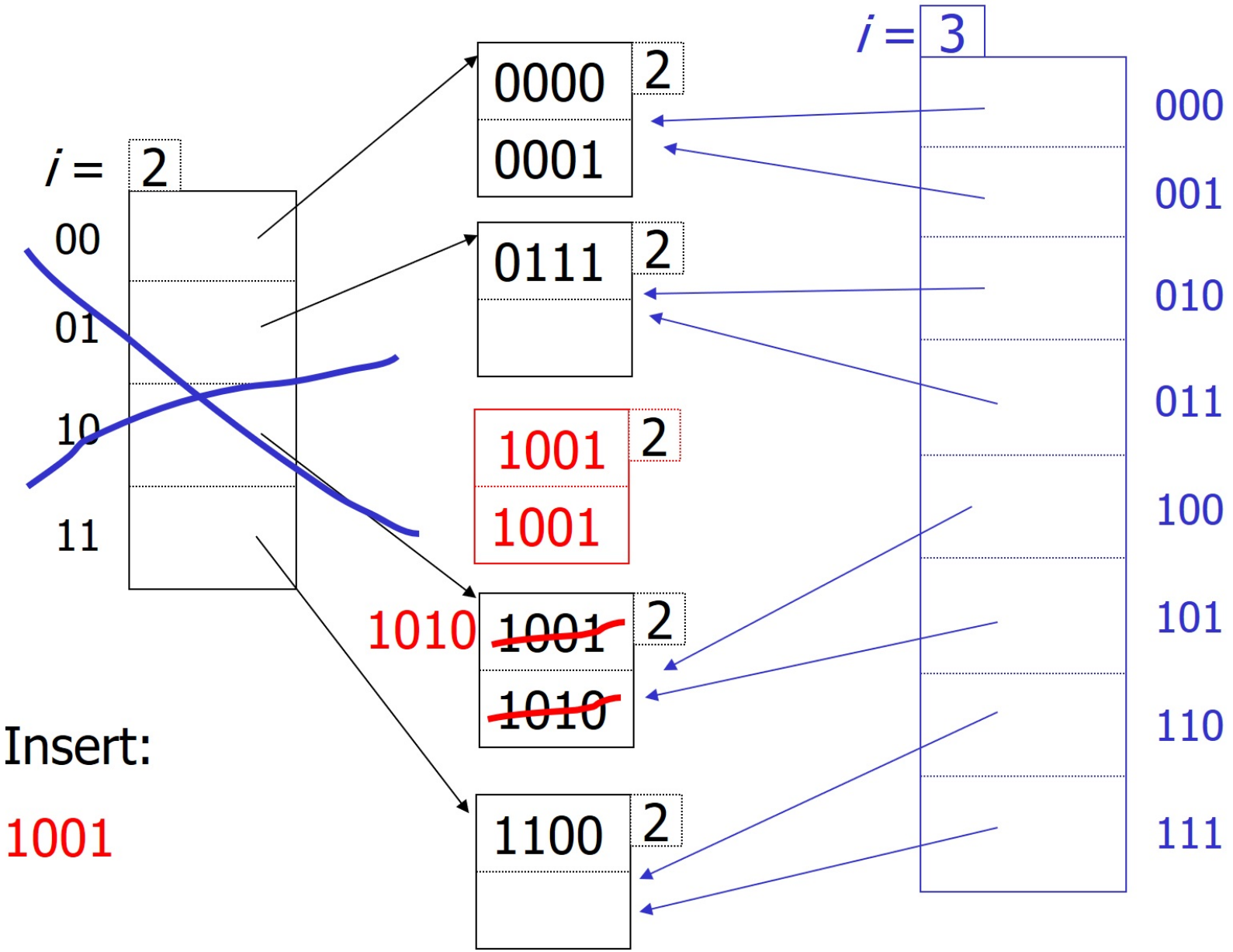
Example: h(k) is 4 bits; 2 keys/bucket



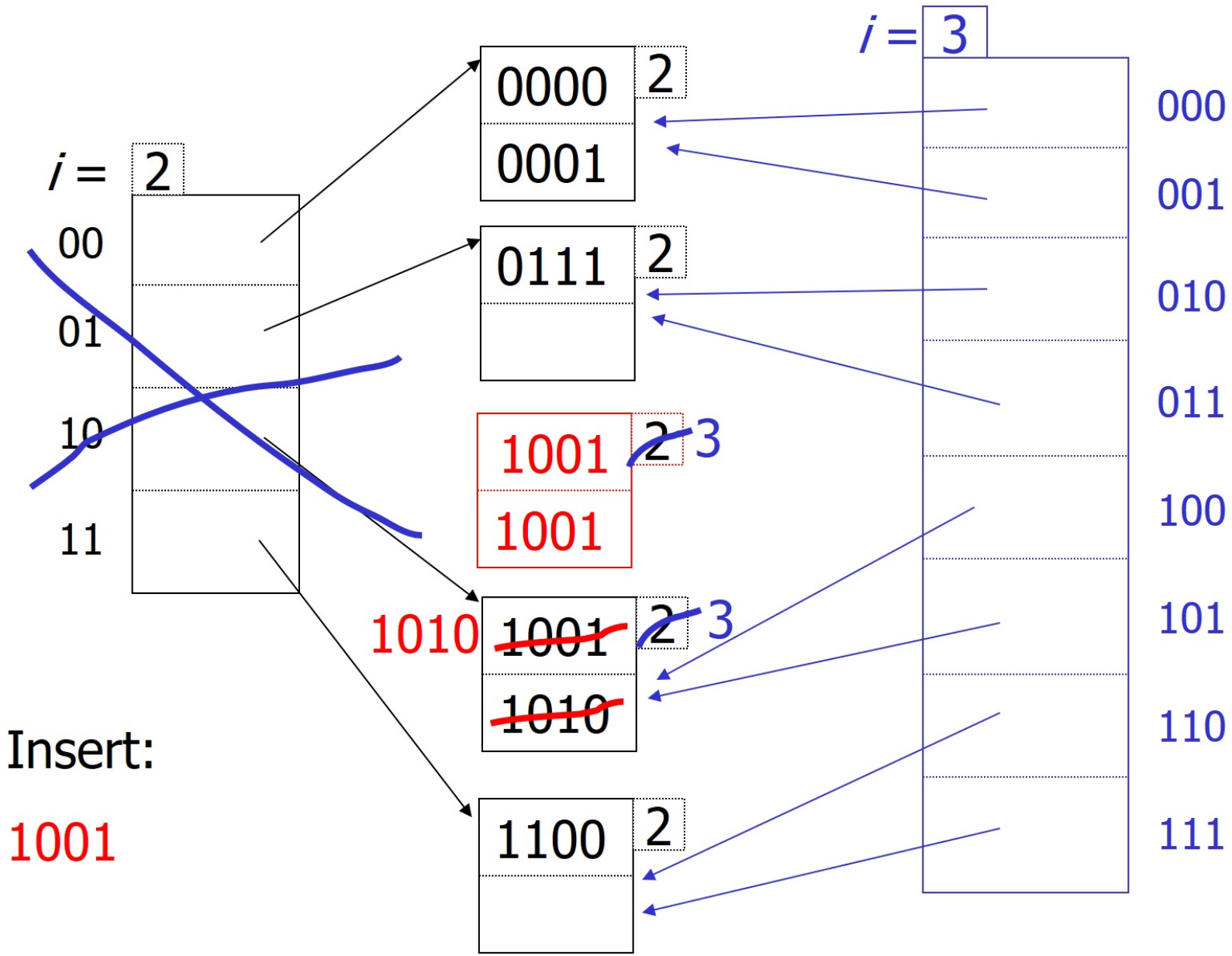
Example: h(k) is 4 bits; 2 keys/bucket



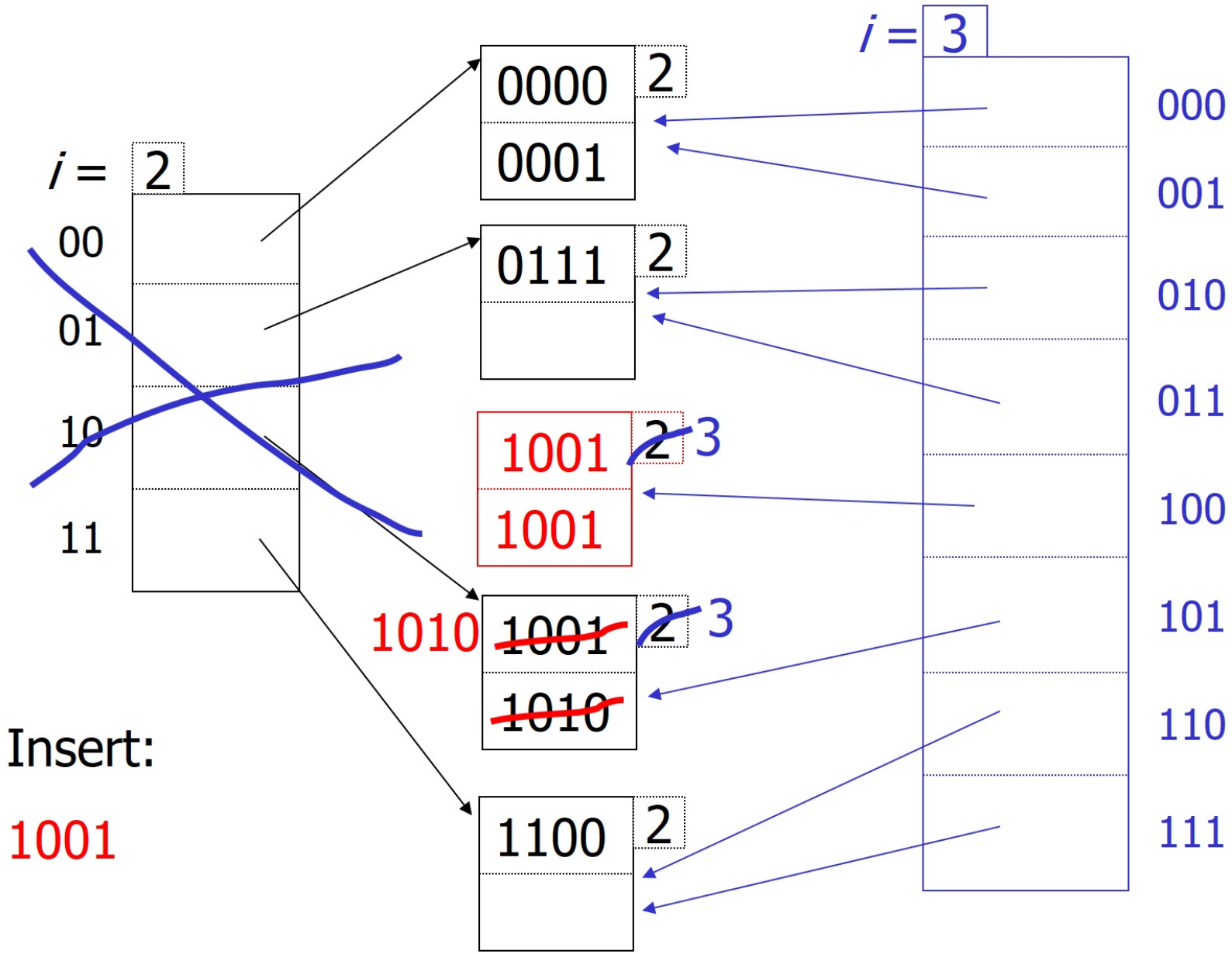
Example: h(k) is 4 bits; 2 keys/bucket



Example: h(k) is 4 bits; 2 keys/bucket



Example: h(k) is 4 bits; 2 keys/bucket



Extensible hashing: deletion

- No merging of blocks
- Merge blocks and cut directory if possible (Reverse insert procedure)

Summary: Extensible hashing

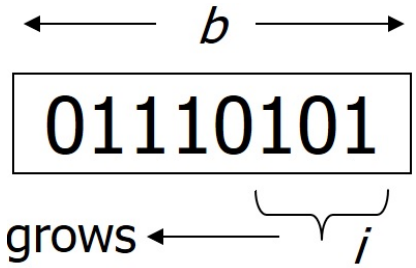
+ Can handle growing files
  - with less wasted space
+ Always access `1 hash table block` to lookup a record

- Indirection (Not bad if directory in memory)
- The size of the `hash table` will double each time when we extend the table (Exponential rate of increase)

- Better solution:
  - Increase the hash table size linearly
  - Linear hashing (discussed next)

Linear Hash Tables

- Another dynamic hashing scheme
- Two ideas:
  a) Use i `low order` bits of hash



  b) File grows linearly

Linear Hash Tables: Properties

- The growth rate of the bucket array will be linear (hence its name)
- The decision to increase the size of the bucket array is flexible
- A commonly used criteria is:
  - If (the average occupancy per bucket $>$ some threshold) then split one bucket into two
- Linear hashing uses overflow buckets
- Use the i low-order bits from the result of the hash function to index into the bucket array

Parameters used in Linear hashing

- n: the number of buckets that is currently in use
- There is also a derived parameter i: $i = \lceil \log_2 n \rceil$
- The parameter i is the number of bits needed to represent a `bucket index` in `binary` (the number of bits of the hash function that currently are used):

```
#buckets    bucket indexes
   n            used              i = [log(n)]
-----------------------------------------------
   1                0             1 bit     // 1 bucket  --> bucket 0
   2              0   1           1 bit     // 2 buckets --> bucket 0 and 1
   3            00 01 10          2 bits
   4            00 01 10 11       2 bits
   5          000 001 .. 100      3 bits
   6          000 001 .. 101      3 bits
   7          000 001 .. 110      3 bits
   8          000 001 .. 111      3 bits
   9        0000 0001 .. 1000     4 bits
   ....
```

- Note: The n `buckets` are number as $0, 1, 2, \ldots, (n-1)$ (in binary)

An important property help to understand Linear Hashing

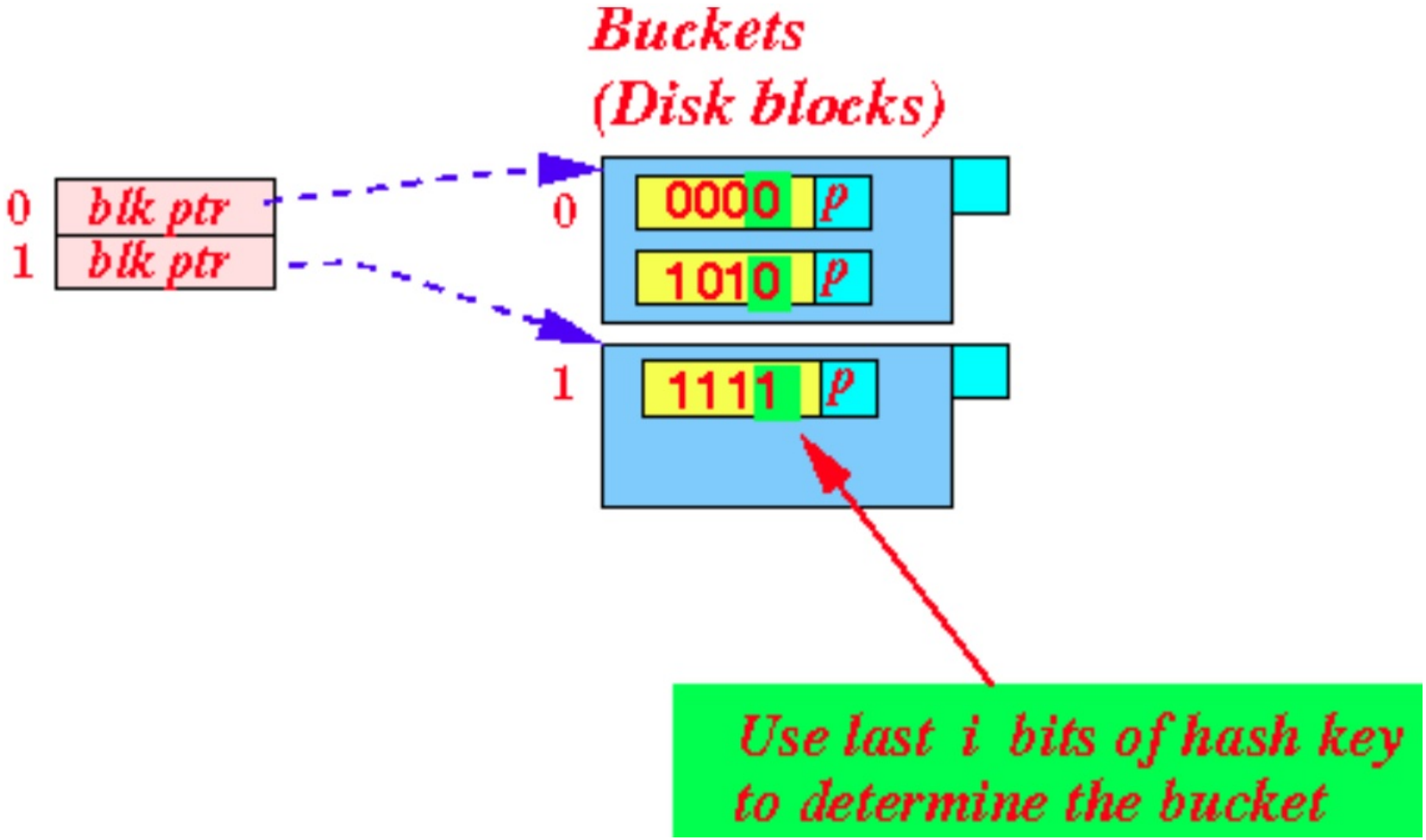
- When the number $(n-1)$ is written as i bits `binary number`, the `first bit` in the `binary number` is always "1"

n	n-1	in binary	i = ⌈log(n)⌉
2	1	1	1 bits
3	2	10	2 bits
4	3	11	2 bits
5	4	100	3 bits
6	5	101	3 bits
7	6	110	3 bits
8	7	111	3 bits
...		^	
		|	
		first bit = 1	

- Consequently: For any number x: $(n-1) < x < 2^i - 1$, when x is written as i `bits binary number`, the `first bit` in the `binary number` (for x) is always "1"

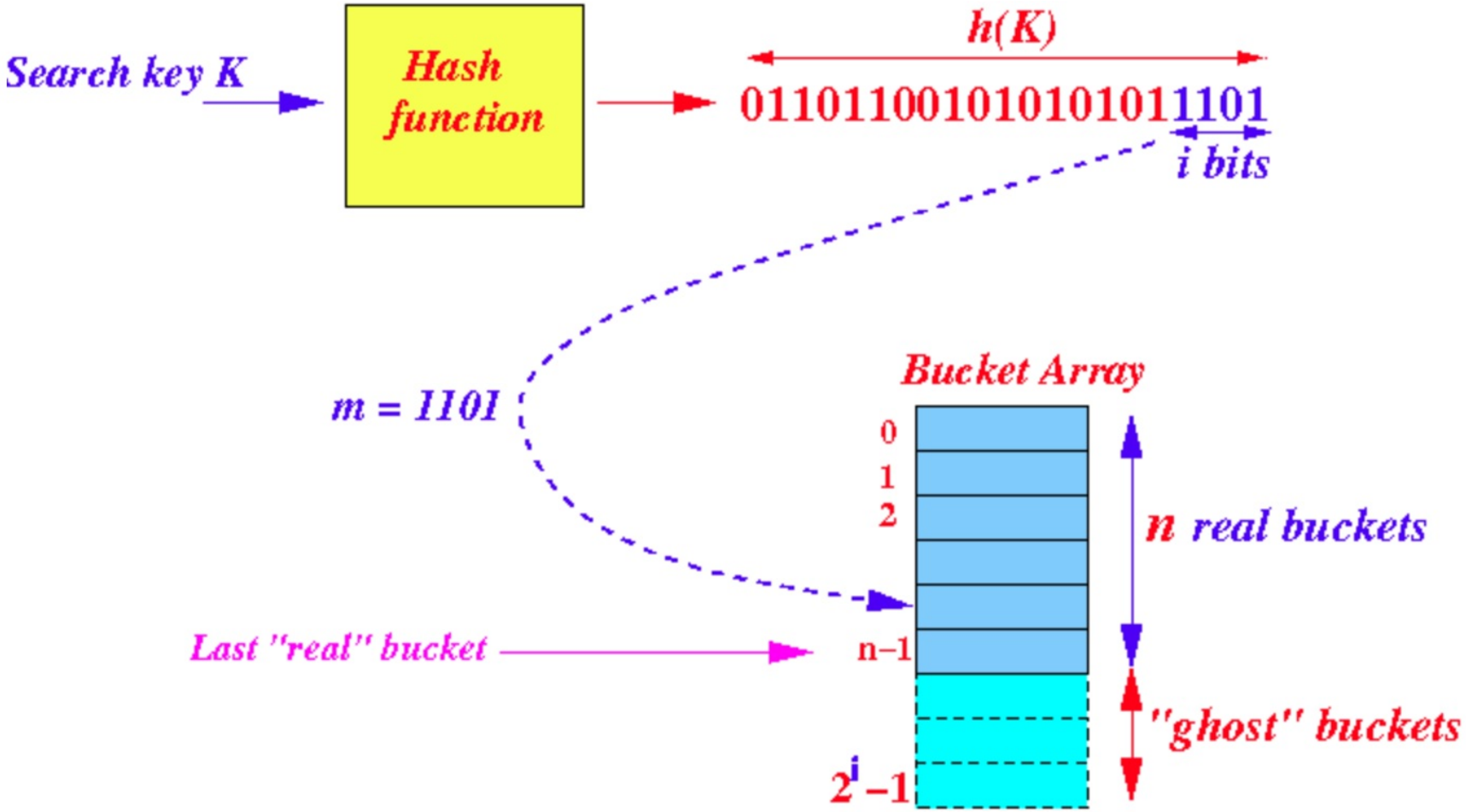
Example of parameters in the Linear hashing method

- $n = 2$ (2 buckets in use, bucket indexes: 0..1)
- $i = 1$ (1 bit needed to represent a bucket index)
- Suppose the number of records $r = 3$



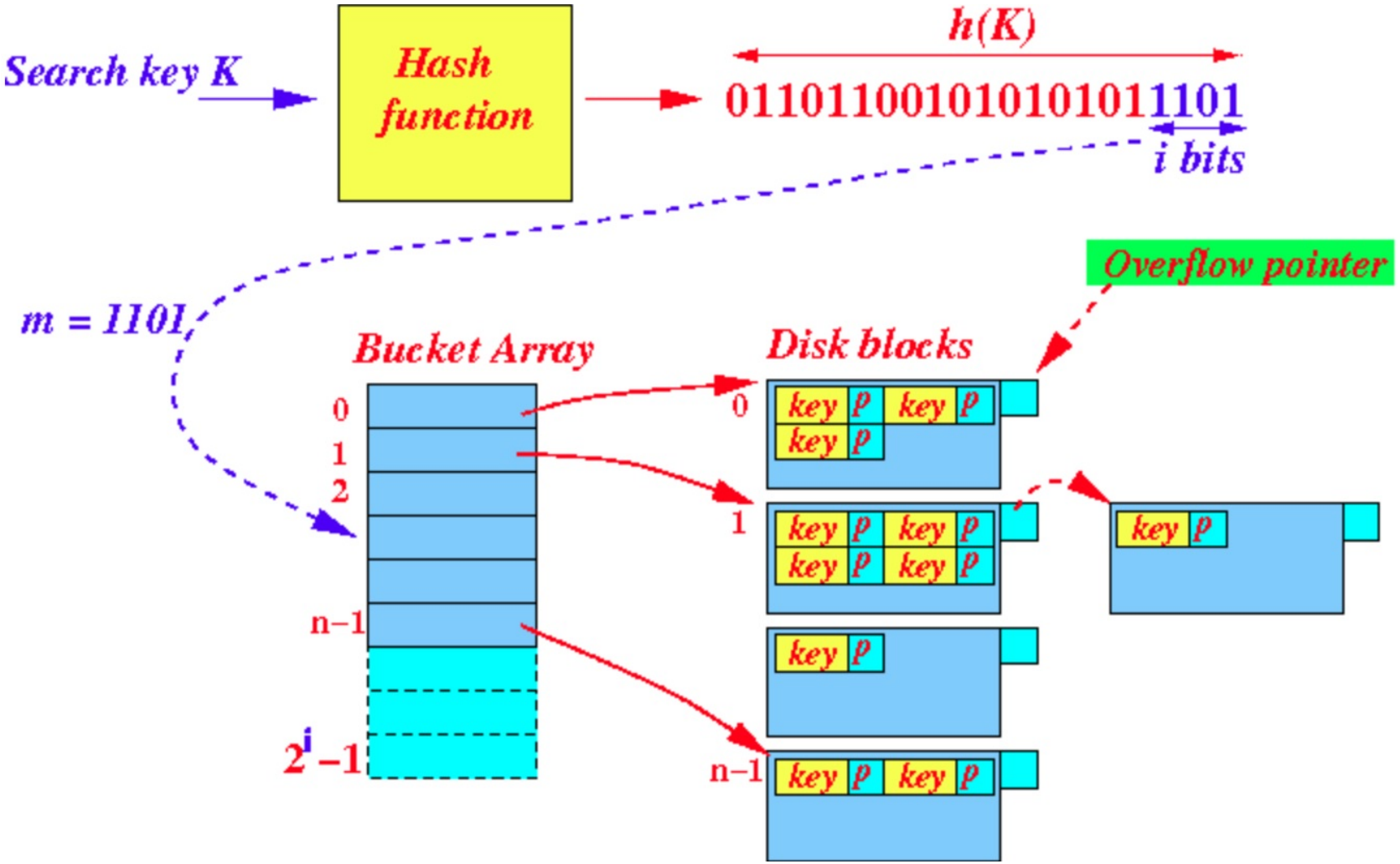
Linear Hashing technique

- Hash function used in Linear Hashing



Linear Hashing technique

- A bucket in Linear Hashing is a chain of disk blocks

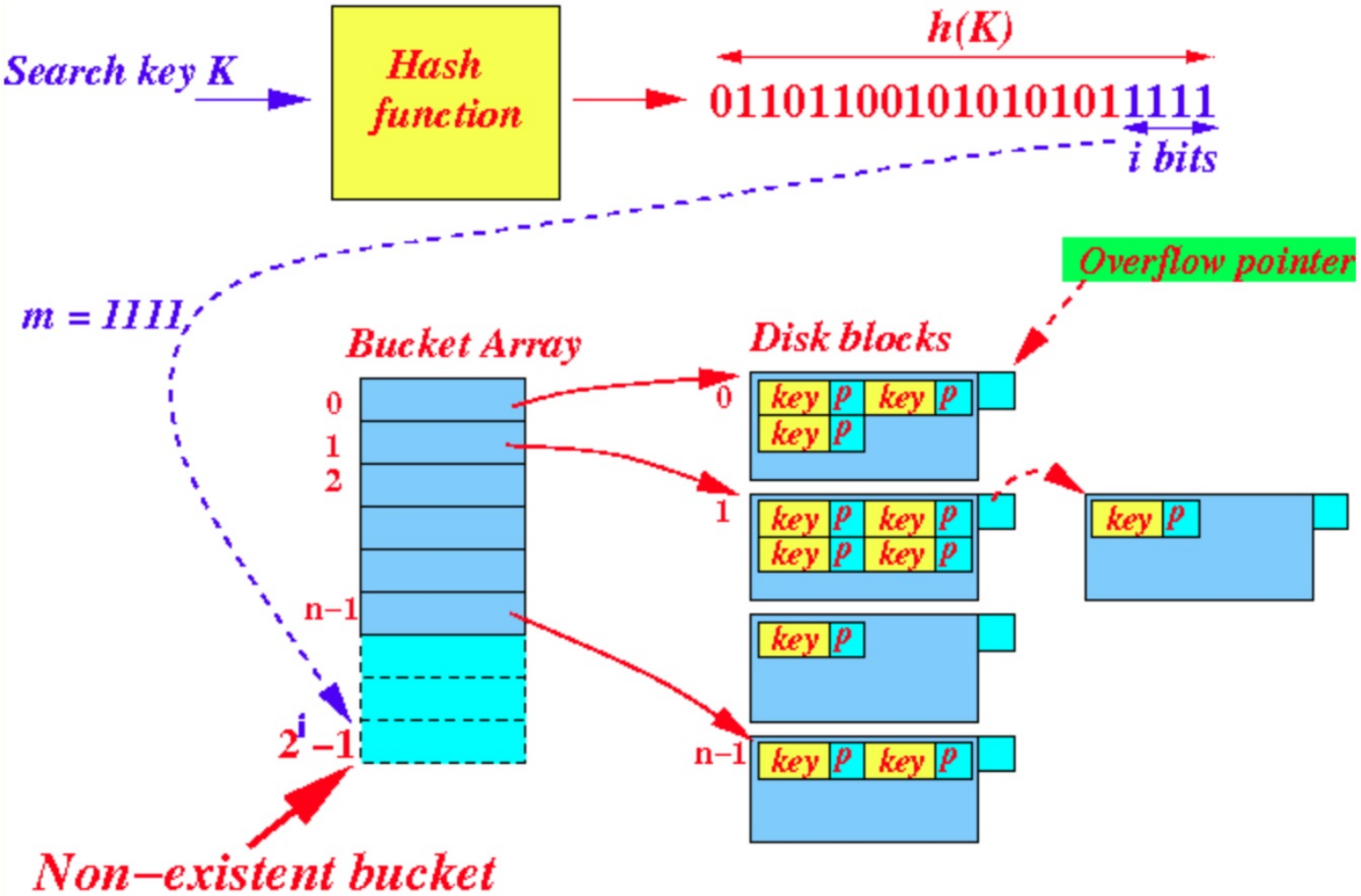


Note

- There are only n buckets in use
- However:
  - A hash key value consists of i bits
  - A hash key value can address: 2^i buckets
- And: $n \leq 2^i$

Note

- $\Rightarrow$ `hash key value` that is $> (n-1)$ will lead to `non-existent bucket (ghost buckets :)`



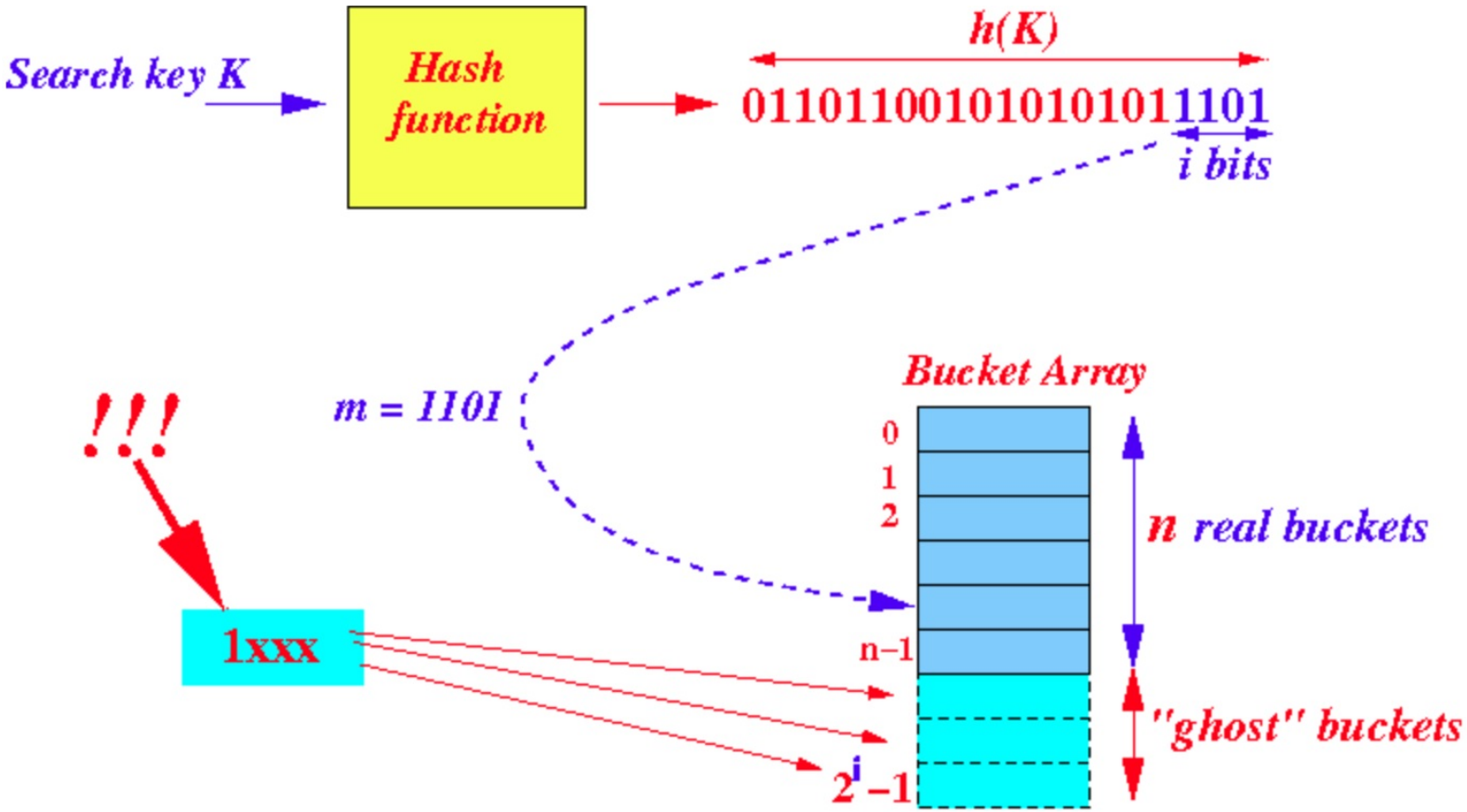
- Conclusion: Need to map the `non-existing` buckets to an existing

Map the non-existing buckets to an existing

- Recall that the first bit of the parameter $n-1$ written in binary must be equal to 1:

$$n-1 = 1xxxxx\ldots$$

- $\Rightarrow$ The non-existent buckets must have as first bit the binary number 1



Map the `non-existing` buckets to an existing

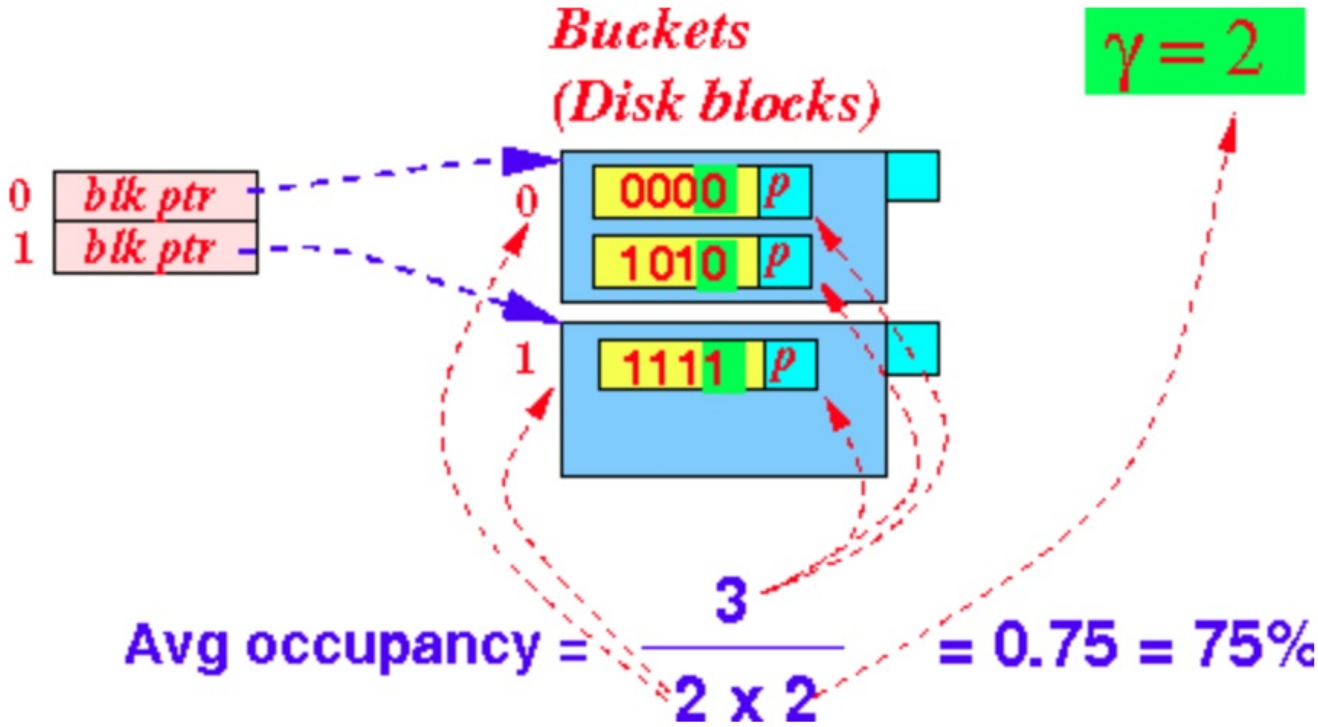
- When we change the first bit of a `non-existent bucket index` from 1 to 0
  - The result index identifies a `real bucket` (because the last bucket is $(n-1)$ starts with a 1 bit)

Criteria to increase n in Linear Hashing

- Commonly used criteria to adjust (increase) the number of buckets n in Linear Hashing:
  - `if (Avg occupancy of a bucket` $> \tau$`) then` $n++$
- How to determine average occupancy of a bucket:
  - n: current number of buckets in use
  - r: current number of search keys stored in the buckets
  - γ: block size (# search keys that can be stored in 1 block)
  - Computation:
    - Max # search keys in 1 block $= \gamma$
    - Max # search keys in n blocks $= n \times \gamma$
    - We have a total of: r search keys in n blocks
    - `Avg occupancy` $= \frac{r}{n \times \gamma}$

Example

$$\texttt{Avg occupancy} = \frac{r}{n \times \gamma}$$



Increase criteria in Linear hashing

- `if` $(\frac{r}{n \times \gamma} > \tau)$ `then` $n++$

Inserting into Linear Hash Tables

- To insert record with `key` K, with `last` i bits of $h(K)$ being $a_1 a_2 \ldots a_i$:
  - Let m be the integer represented by $a_1 a_2 \ldots a_i$ in `binary`
  - `If` $m < n$, then `bucket` m exists – put record in that `bucket`. If necessary, use an `overflow block`
  - `If` $m \geq n$, then `bucket` m does not (yet) exist, so put record in `bucket` whose index corresponds to $0 a_2 \ldots a_i$

Inserting into Linear Hash Tables

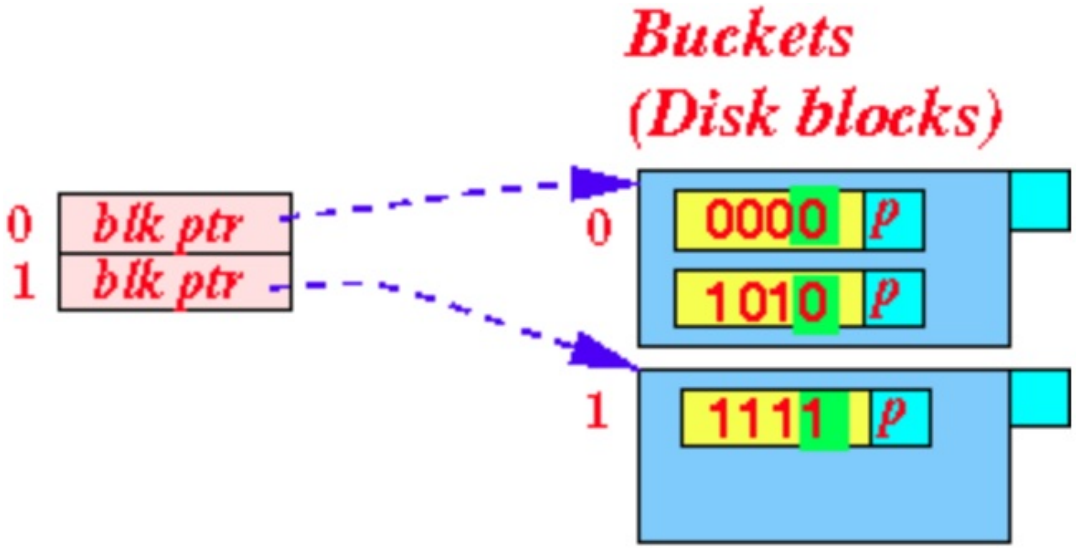
- Check if we need to adjust n
  - Compare `average occupancy` to `threshold`
  - If exceeds `threshold` then add a new `bucket` and rearrange records
    - We need to move some search keys into this new bucket
  - If number of buckets exceeds 2^i, then increment i by 1

Inserting into Linear Hash Tables: Example

- Parameters:
  - `Max # search keys in 1 block` $(\gamma) = 2$
  - `Threshold avg occupancy` $(\tau) = 0.85$

Example Linear Hashing

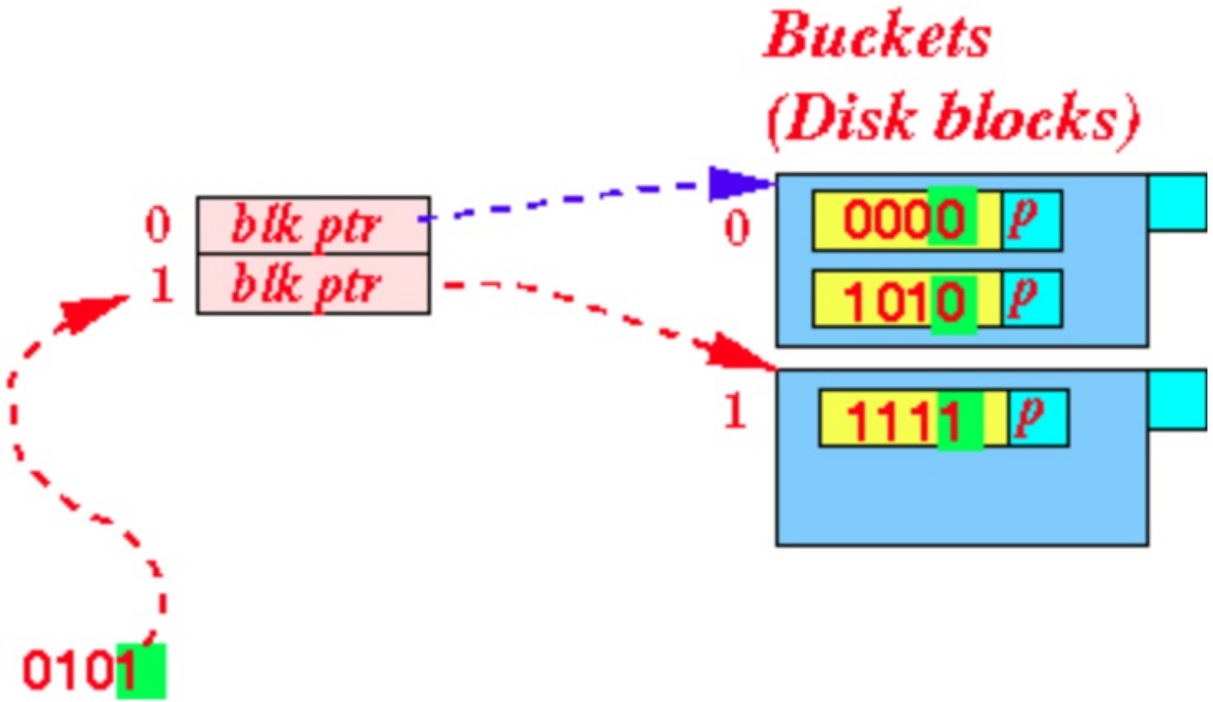
- Initial State: $n = 2,\ i = 1,\ r = 3$



- Average occupancy: $\frac{r}{n \times \gamma} \Rightarrow \frac{3}{2 \times 2} = 0.75 < 0.85$

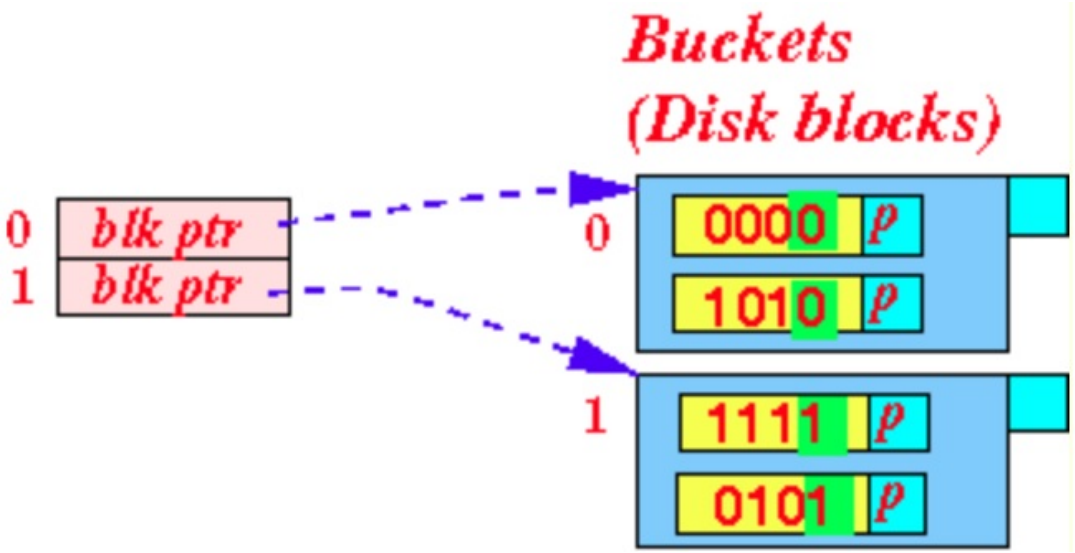
Insert search key K such that h(K) = 0101

- `Insert 0101`
- $n = 2,\ i = 1,\ r = 3$



Insert search key K such that h(K) = 0101: result

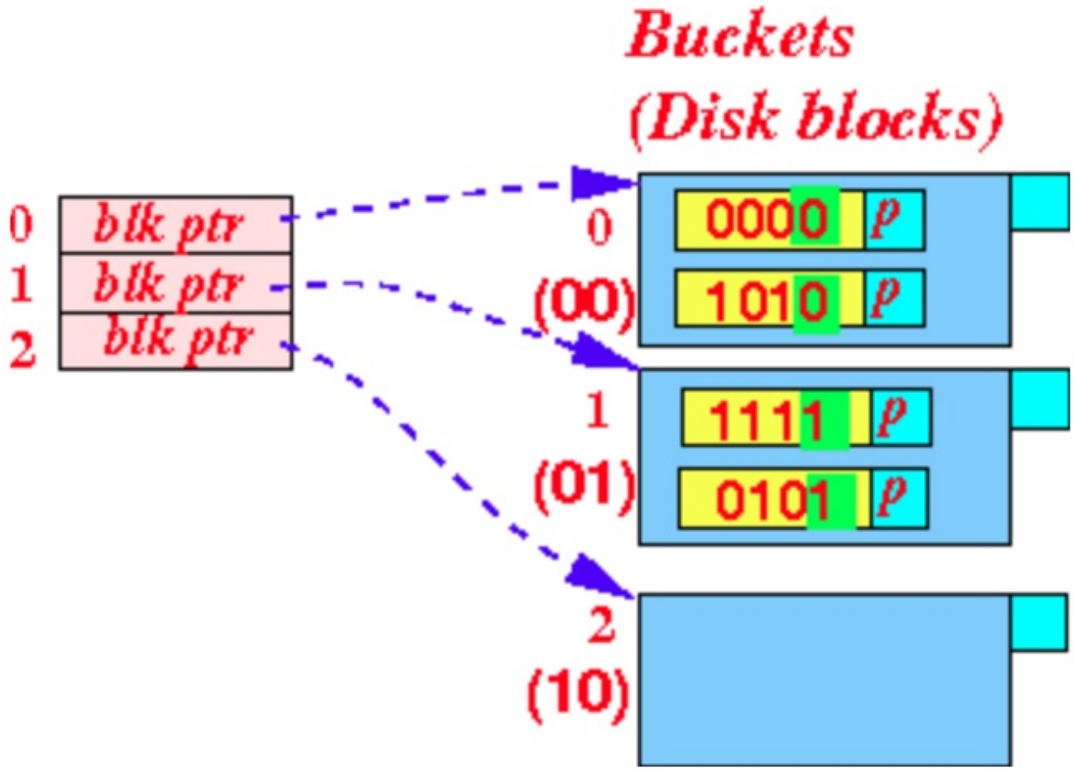
- $n = 2,\ i = 1,\ r = 4$



- `Average occupancy:` $\frac{r}{n \times \gamma} \Rightarrow \frac{4}{2 \times 2} = 1 > 0.85$. We must add an new bucket

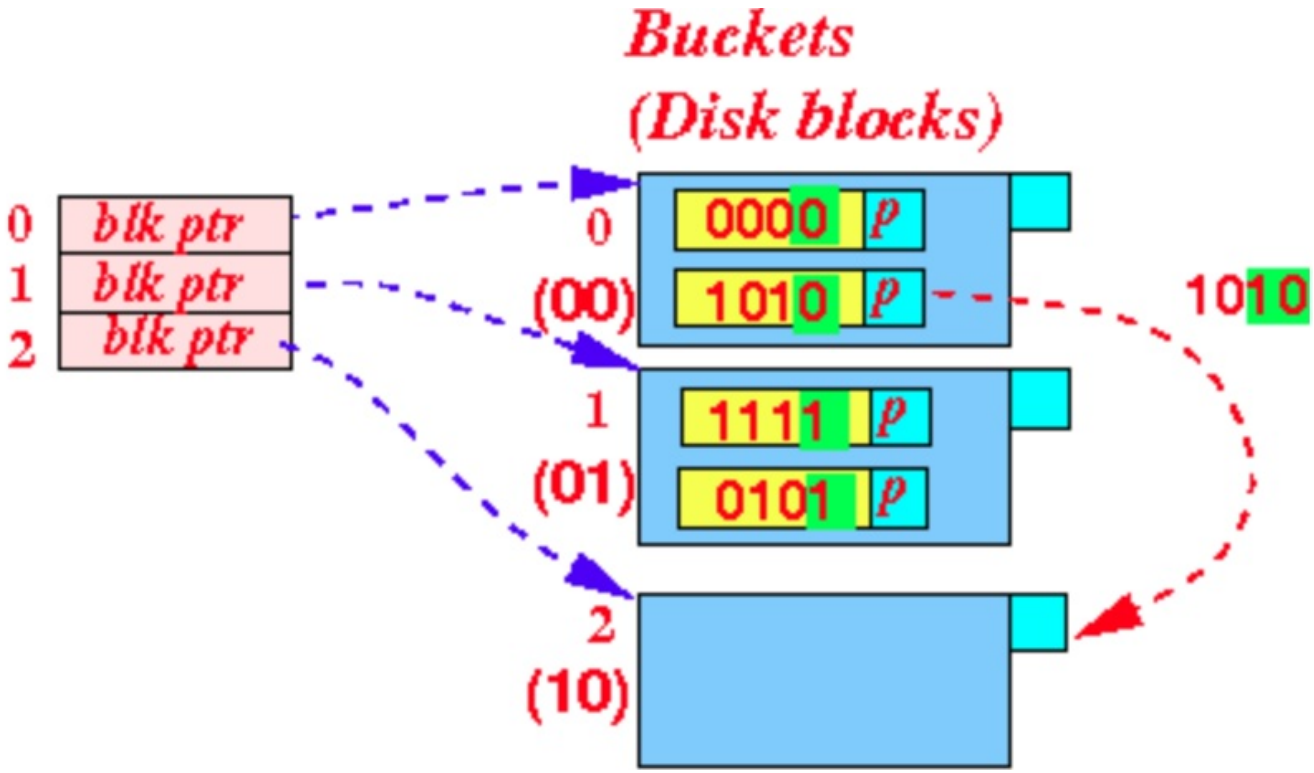
Insert search key K such that h(K) = 0101: result

- Add `bucket 2 (= 10 (binary))`
- $n = 3,\ i = 2,\ r = 4$



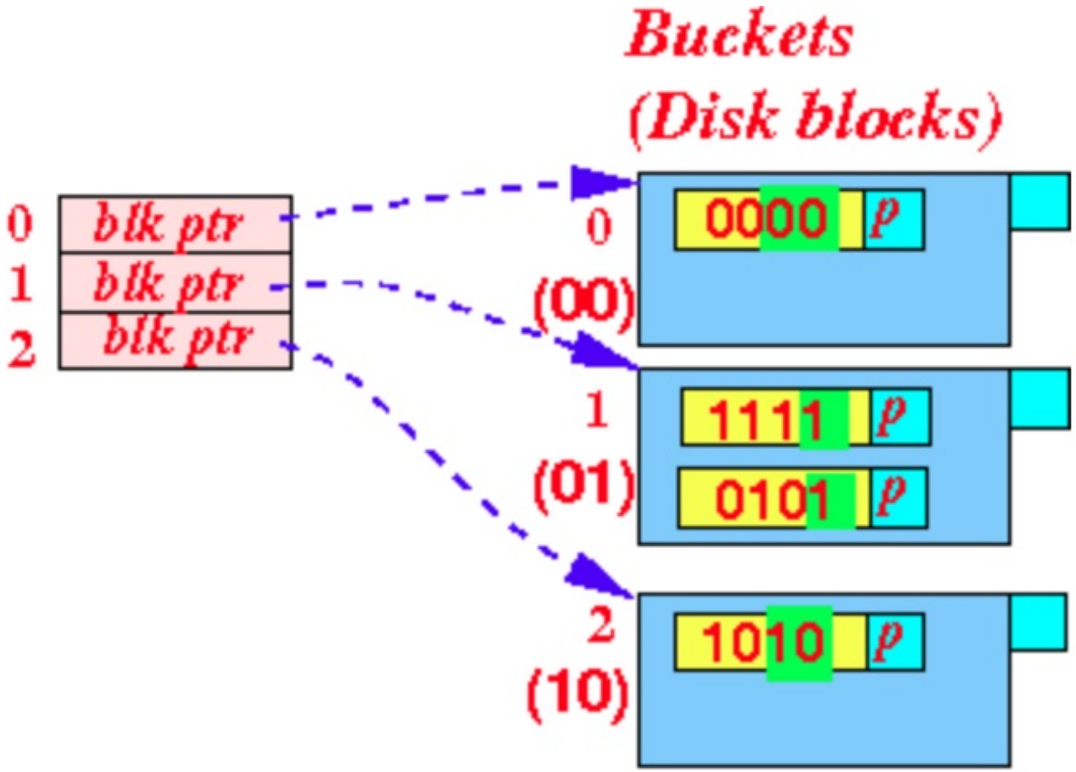
Insert search key K such that h(K) = 0101: result

- $n = 3$, $i = 2$, $r = 4$
- Transfer search keys from `bucket` 00 to the newly created `bucket` 10



Insert search key K such that h(K) = 0101: result

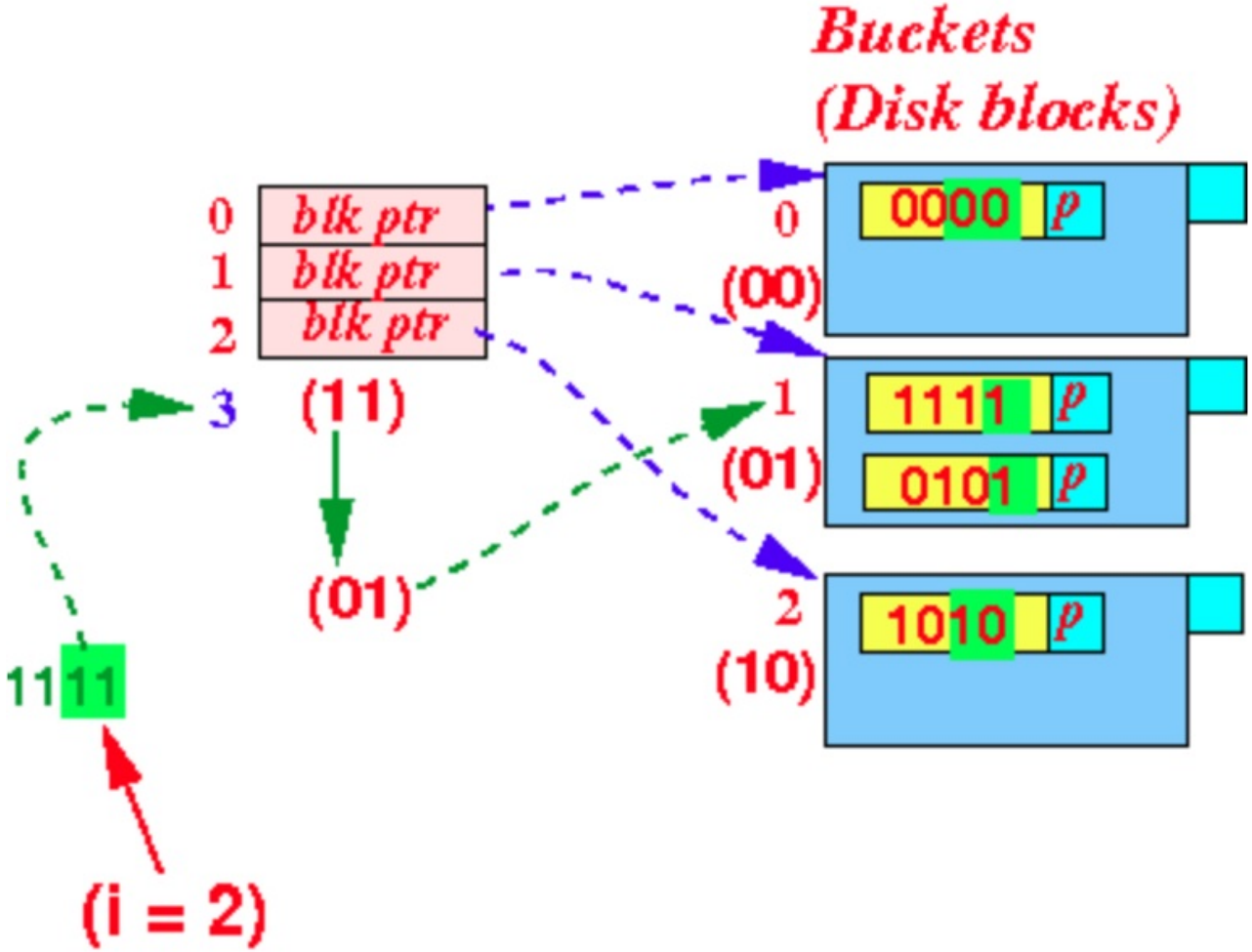
- $n = 3$, $i = 2$, $r = 4$
- Transfer search keys from `bucket` 00 to the newly created `bucket` 10



- `Average occupancy:` $\frac{r}{n \times \gamma} \Rightarrow \frac{4}{3 \times 2} = 0.67 < 0.85$

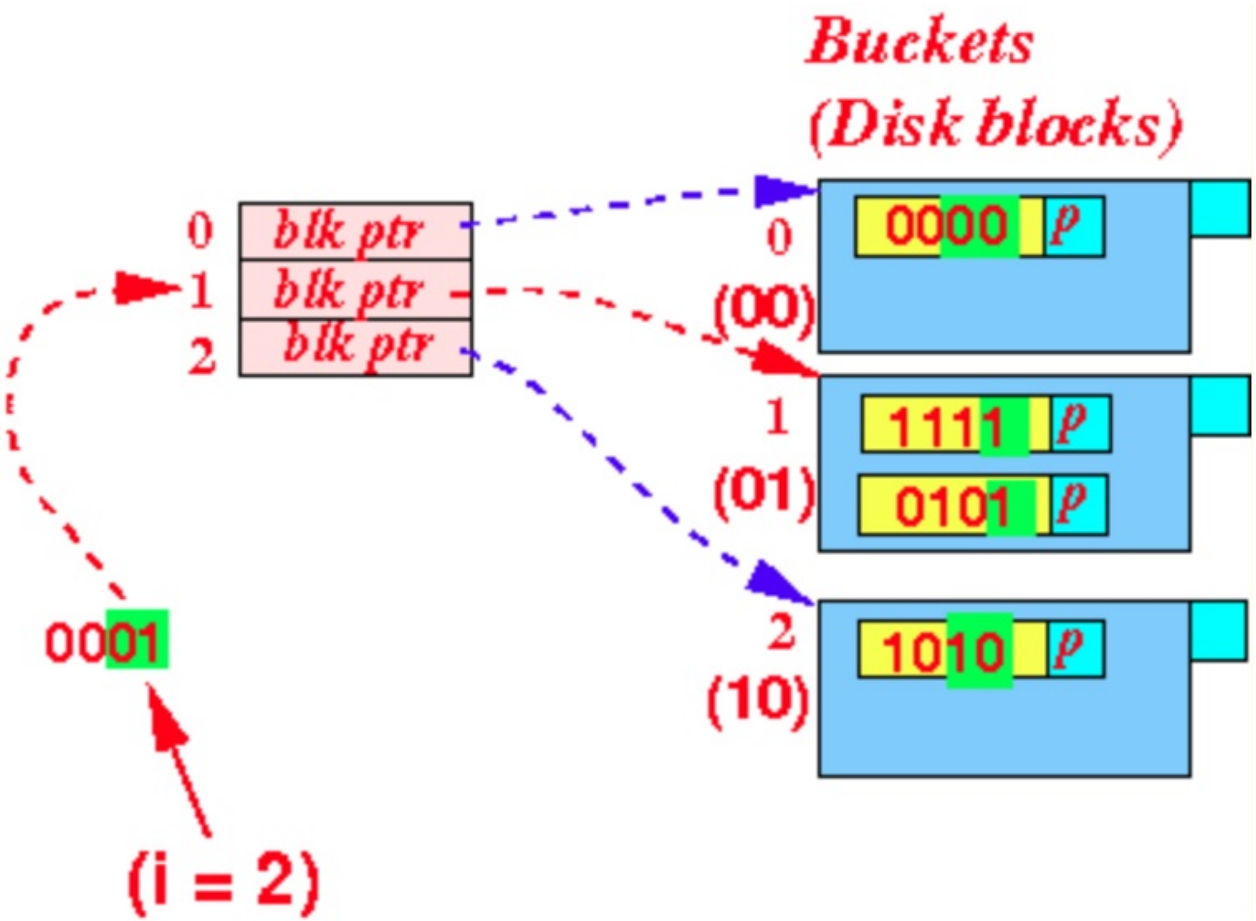
Notice that

- $n = 3$ (changed)
- $i = 2$ (changed)
- We can `find` 1111 as follows



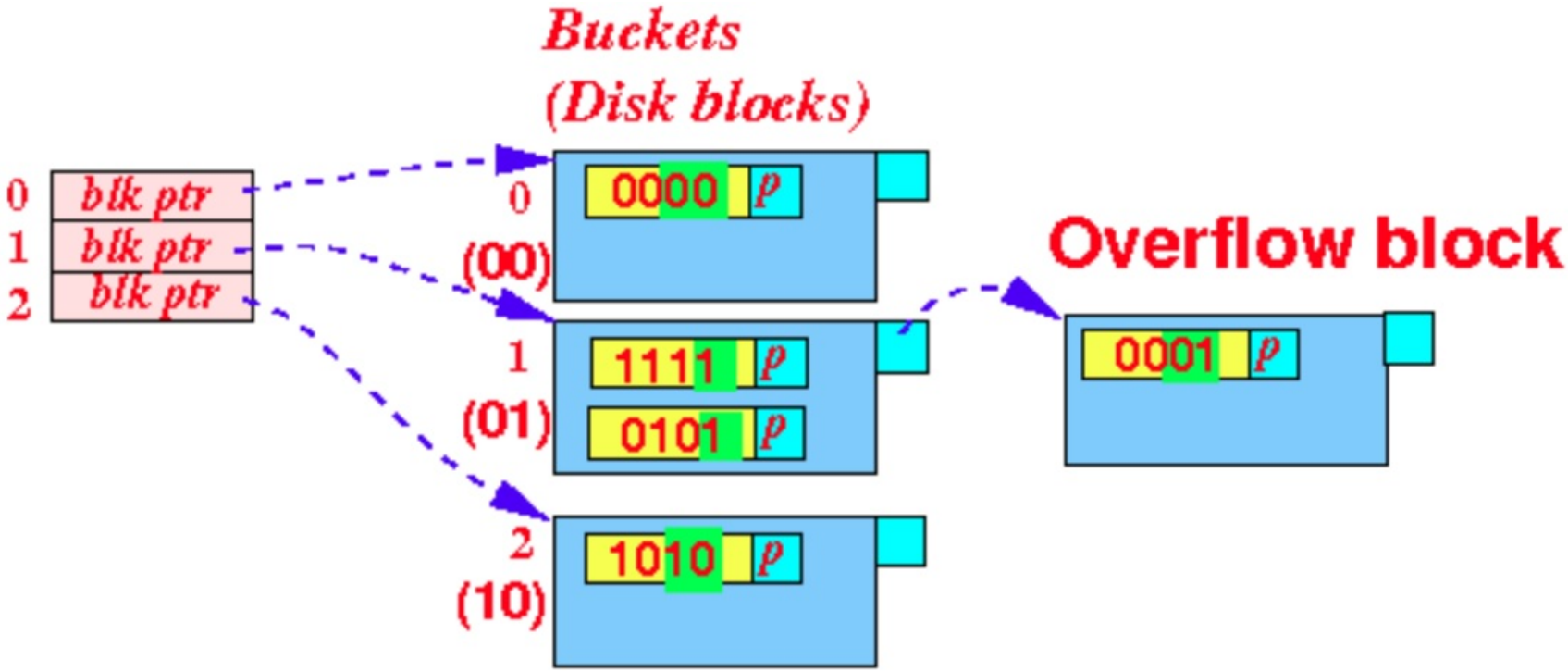
Insert search key K such that h(K) = 0001

- `Insert 0001`
- $n = 3,\ i = 2,\ r = 4$



Insert search key K such that h(K) = 0001: result

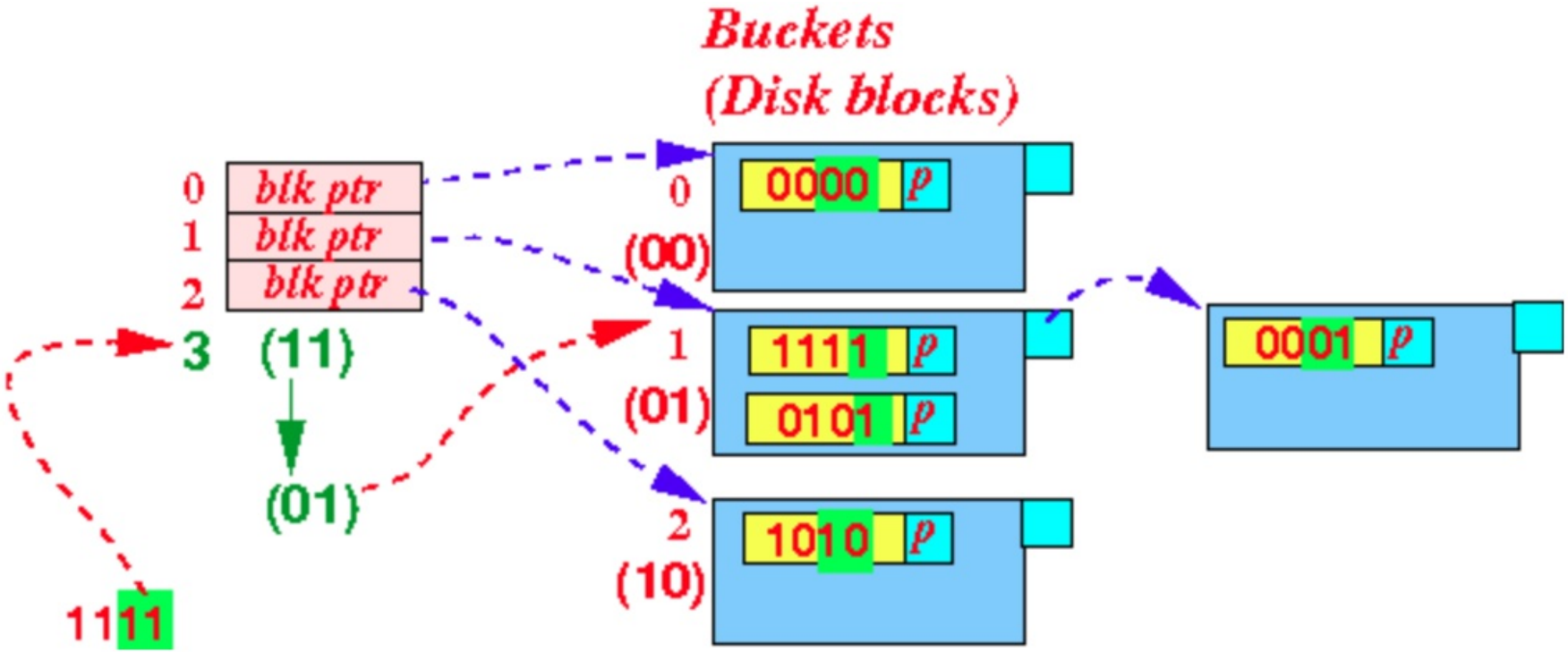
- $n = 3,\ i = 2,\ r = 5$



- So: `Linear Hashing uses overflow blocks`
- `Average occupancy:` $\frac{r}{n \times \gamma} \Rightarrow \frac{5}{3 \times 2} = 0.83 < 0.85.$ `No need too add another bucket`

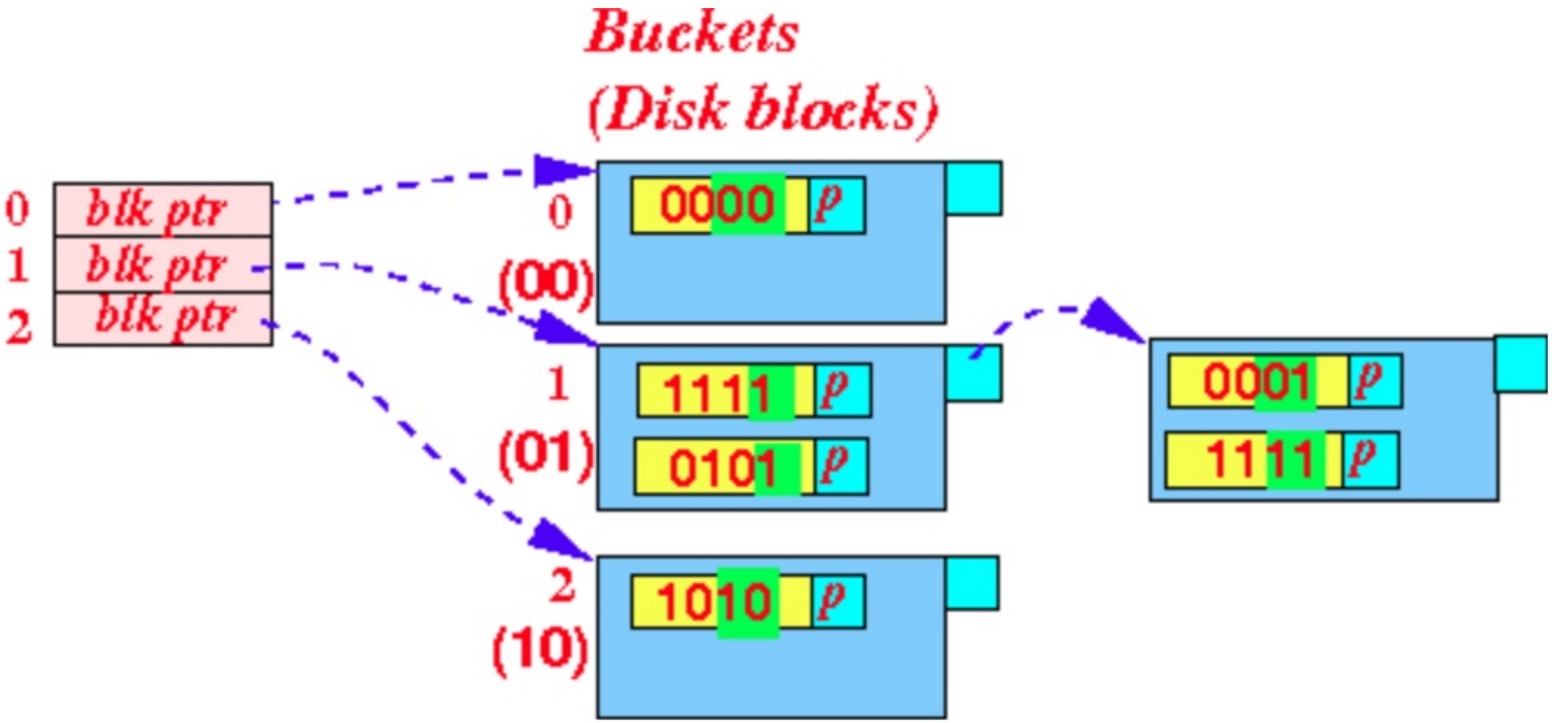
Insert search key K such that h(K) = 1111

- `Insert 1111`
- $n = 3,\ i = 2,\ r = 5$



Insert search key K such that h(K) = 1111: Result

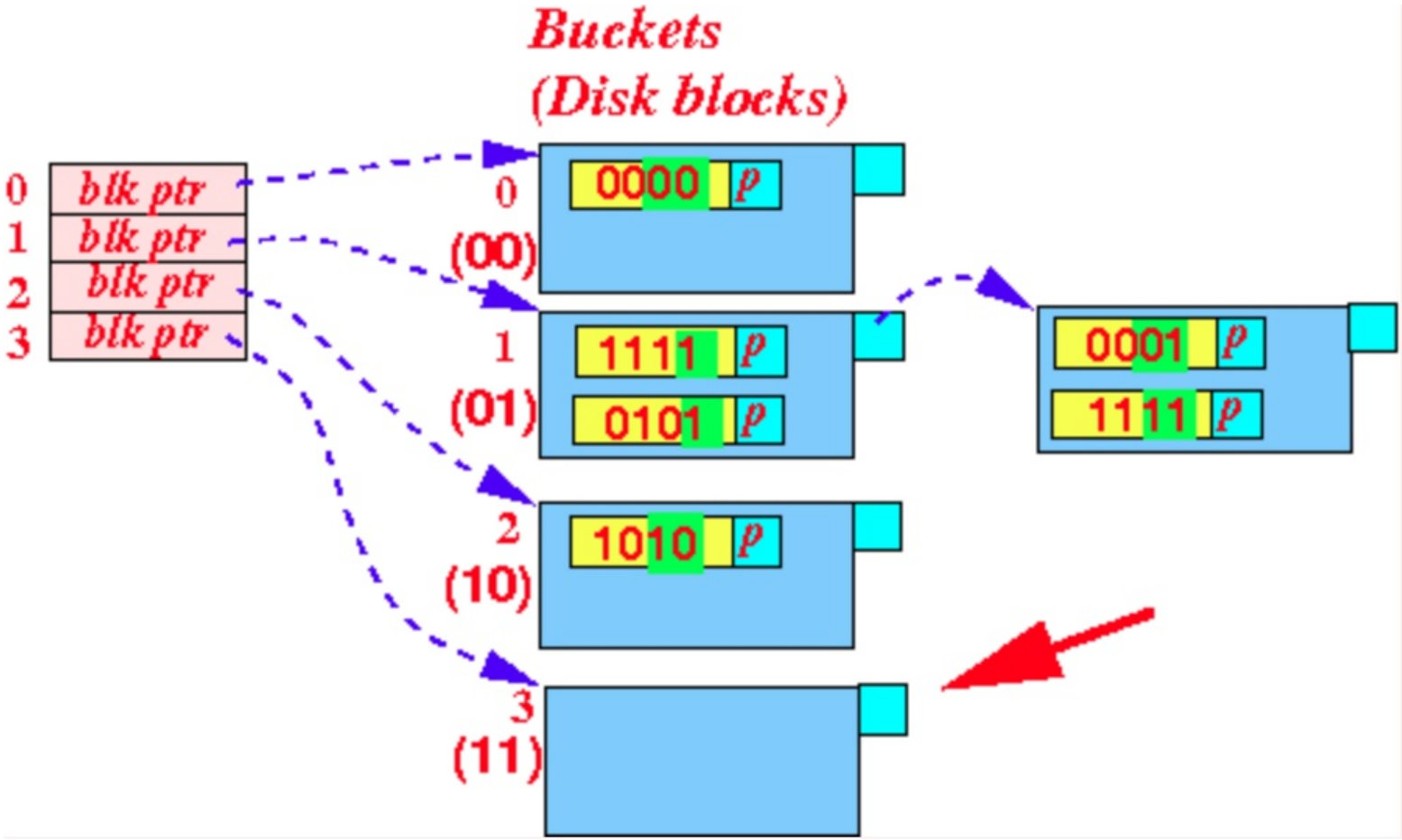
- $n = 3,\ i = 2,\ r = 6$



- Average occupancy: $\frac{r}{n \times \gamma} \Rightarrow \frac{6}{3 \times 2} = 1 > 0.85$. We must add an new bucket

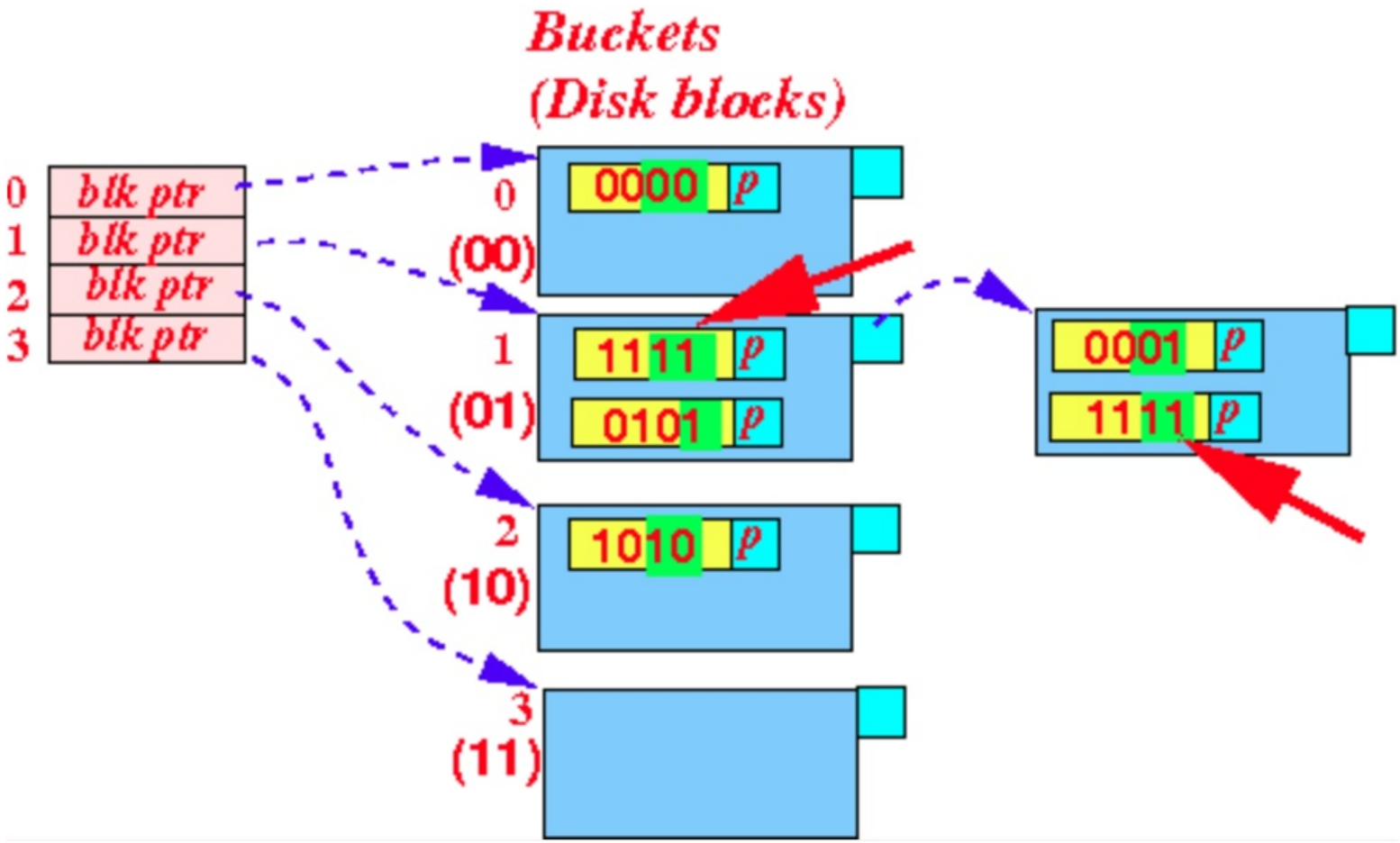
Insert search key K such that h(K) = 1111: Result

- Add `bucket 3 (= 11 (binary))`
- $n = 4,\ i = 2,\ r = 6$



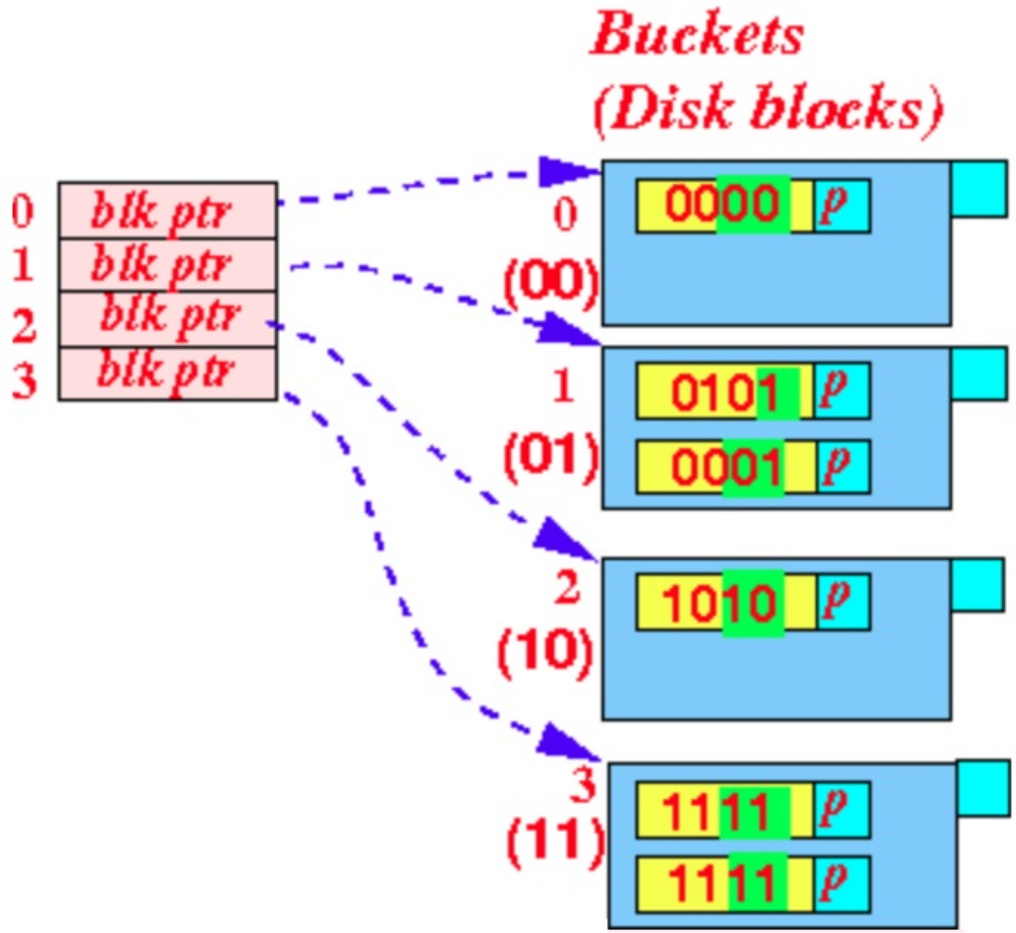
Insert search key K such that h(K) = 1111: Result

- Transfer `search keys` from `bucket 01` to the newly created `bucket 11`
- $n = 4$, $i = 2$, $r = 6$



Insert search key K such that h(K) = 1111: Result

- Transfer `search keys` from `bucket 01` to the newly created `bucket 11`



- `Average occupancy:` $\frac{r}{n \times \gamma} \Rightarrow \frac{6}{4 \times 2} = 0.75 < 0.85$

Summary: Linear hashing

+ Can handle growing files
  - with less wasted space
  - with no full reorganizations
+ No indirection like `extensible hashing`

- Can still have overflow chains

Comparing Index Approaches

- Hashing good for probes given key.

```
SELECT ...
FROM r
WHERE r.A = 5;
```

- Sequential Indexes and B^+-trees good for `Range Searches`

```
SELECT ...
FROM r
WHERE r.A > 5;
```