CS525: Advanced Database Organization

Notes 5: Indexing and Hashing
Part III: Hash Tables

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

February 26th 2024

Slides: adapted from courses taught by Hector Garcia-Molina, Stanford, Shun Yan Cheung, Emory University, Ellen Munthe-Kaas, Universitetet Oslo, & Andy Pavlo, Carnegie Mellon University

Outline

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level ...
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- `B+-Trees`
- Hash Tables

Hash Tables

- A hash table is a another structure useful for indexes
- A hash table implements an unordered associative array[1] that maps keys to values.
- It uses a hash function to compute an offset into the array for a given key, from which the desired value can be found.

[1] *Associative array: An array that stores element values in association with key values rather than in a strict linear index order.*
Numeric array: An array with a numeric index. Values are stored and accessed in linear fashion.

Hash Table

- Two essential design decisions involved in implementing a hash table:
  - Design Decision 1: Hash function
  - Design Decision 2: Hashing Scheme

Design Decision 1: Hash function

- The hash function is responsible for transforming a potentially large and diverse set of keys into a smaller, more manageable range of indices, often referred to as *buckets* or *slots* in the hash table.
  - *how to map a large key space into a smaller domain.*
  - *used to compute an index into an array of buckets or slots.*
- Need to consider the trade-off between fast execution vs. collision rate.
  - *The choice of hash function involves a trade-off between the speed at which it can compute the hash value and the likelihood of collisions (two different keys mapping to the same index).*
    - Hash function that always returns a constant value would be very fast but result in numerous collisions, which can degrade performance.
    - Conversely, a perfect hashing function with no collisions can be ideal but is often computationally expensive.
  - The ideal design is somewhere in the middle.
    - *It should distribute keys as evenly as possible across the available buckets while being reasonably fast to compute*

Design Decision 2: Hashing Scheme

- The hashing scheme determines how collisions are handled when two keys hash to the same index.
- It defines what happens when multiple keys are assigned to the same bucket.
  - *how to handle key collisions after hashing.*
- Need to consider the trade-off between the need to allocate a large hash table (more buckets) to reduce the chance of collisions vs. executing additional instructions to handle collisions (to get/put keys).

Today's Agenda

- Hash Functions
- Introduction to Indexing using Hashing
- Dynamic Hashing Schemes

Hash Functions

- A hash function takes in any key as its input. It then return an integer representation of that key i.e., the "hash"
- The function's output is deterministic
  - i.e., the same key should always generate the same hash output.
- We do not want to use a cryptographic hash function for DBMS hash tables.
- We want something that is fast and will have a low collision rate.

Example hash function

- A good hash functions is important
  - should be easy to compute (fast)
  - should distribute the search keys evenly - each bucket should have approximately the same number of elements
  - common function examples:
    - integer keys:
      `key value mod array size`
    - character keys:
      `sum characters as integer mod array size`
  - best function varies between different uses

Hash Functions

Hash function	**Year**	**Description**
CRC-64	1975	Cyclic Redundancy Check: Checksum function used for error detection. Not typically used as a general-purpose hash function.
MurmurHash	2008	Fast and general-purpose hash function. Well-suited for a variety of applications, including caching, hashing databases, and bloom filters.
Google CityHash	2011	Fast hash function for short keys (<64 bytes). Often used in applications such as caching and hashing database indexes.
Facebook XXHash	2012	Family of fast and lightweight hash functions. Well-suited for a variety of applications
Google FarmHash	2014	Newer version of CityHash with better collision rates. Often used in applications, such as DB replication and distributed caching.
Facebook XXHash3	2018	Fast and lightweight hash function with excellent collision resistance.

Hashing used as an Indexing Technique

- Hash function `h` is a function that maps a search key to an index between `[0..B-1]`, where `B` is the size of the hash table
- A bucket is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - i.e., a location (slot) in the bucket array
  - we obtain the bucket of an entry from its search-key value using a hash function
- Bucket array is an array of (fixed) size B, where 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.

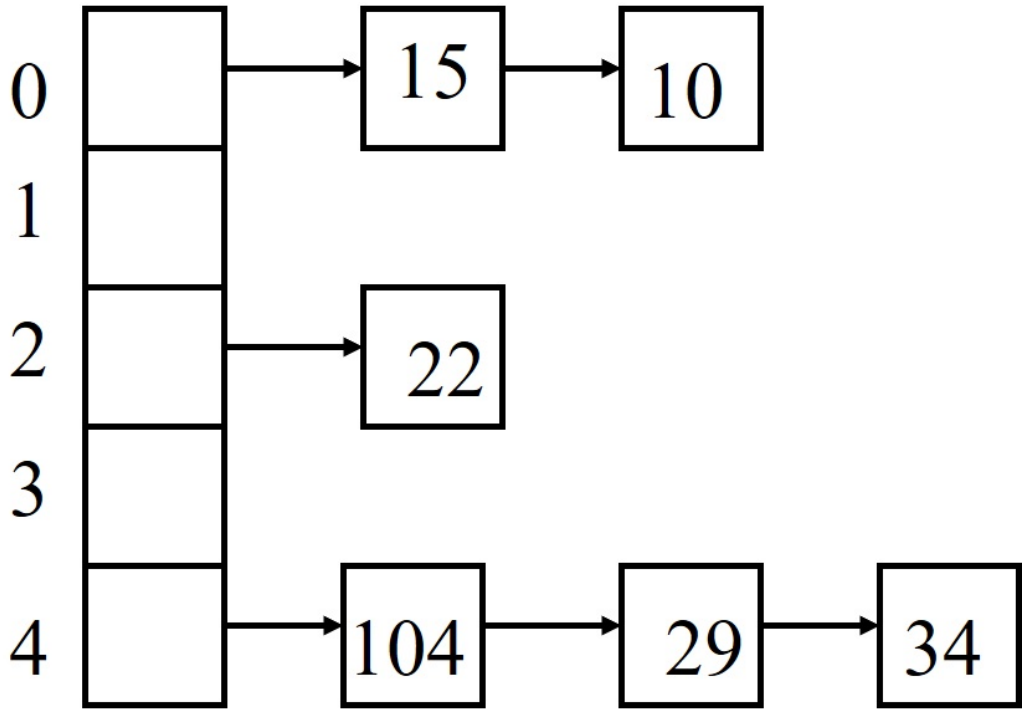
Hashing used as an Indexing Technique

- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- Record with key k is put in the linked list that starts at entry $h(k)$ of B.

Example of Hash Table

$B = 5$

$h(k) = k \bmod 5$

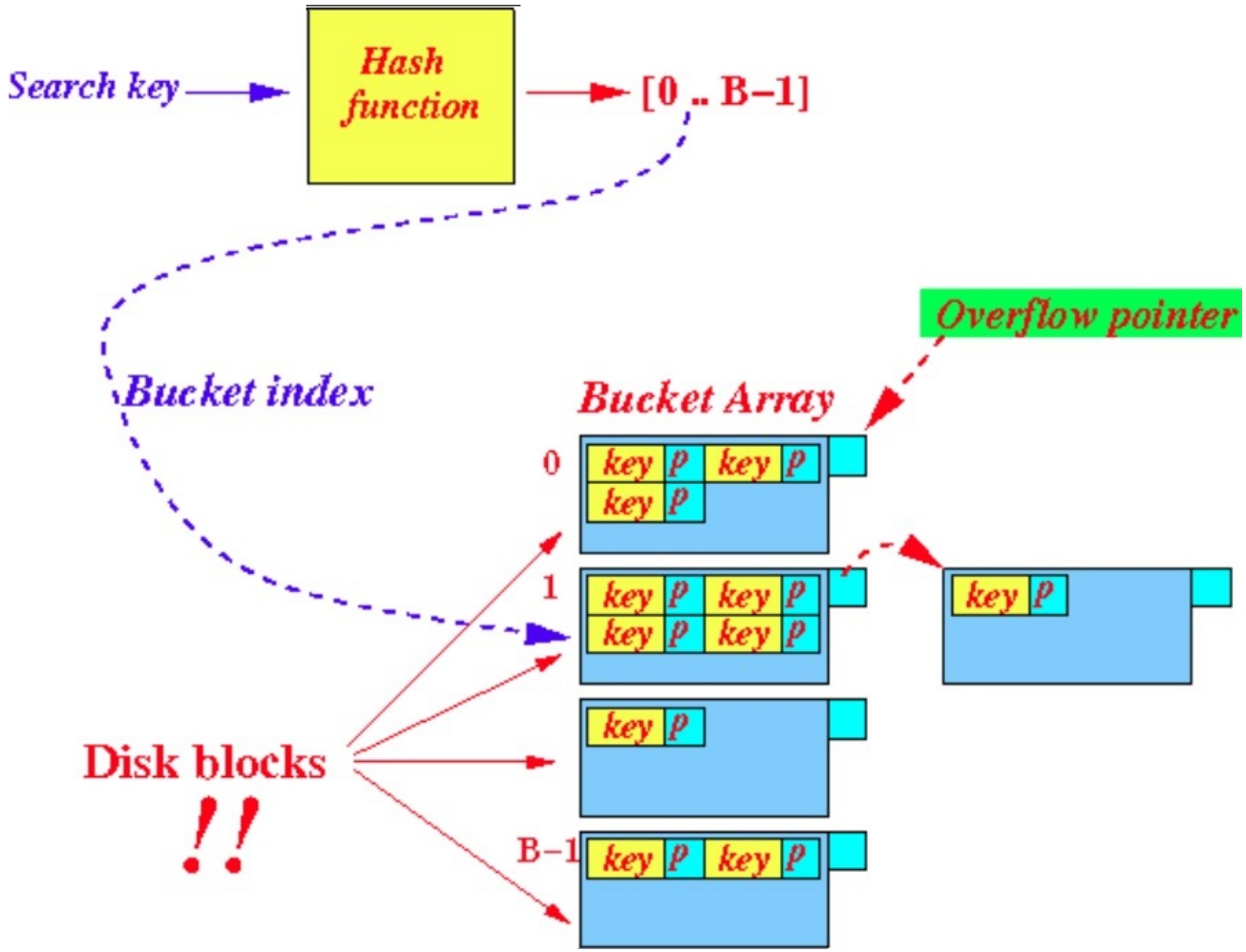


Storing Hash Tables on Disk

- 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
- Bucket array is used to store the data (key-value entries) according to their computed indices
- Records that hash to a certain bucket are put in the corresponding `block`
- Bucket array is stored on disk:
  - `1` bucket = `1` disk block
  - A bucket is a collection of key-value pairs
- If a bucket overflows then start a chain of overflow blocks

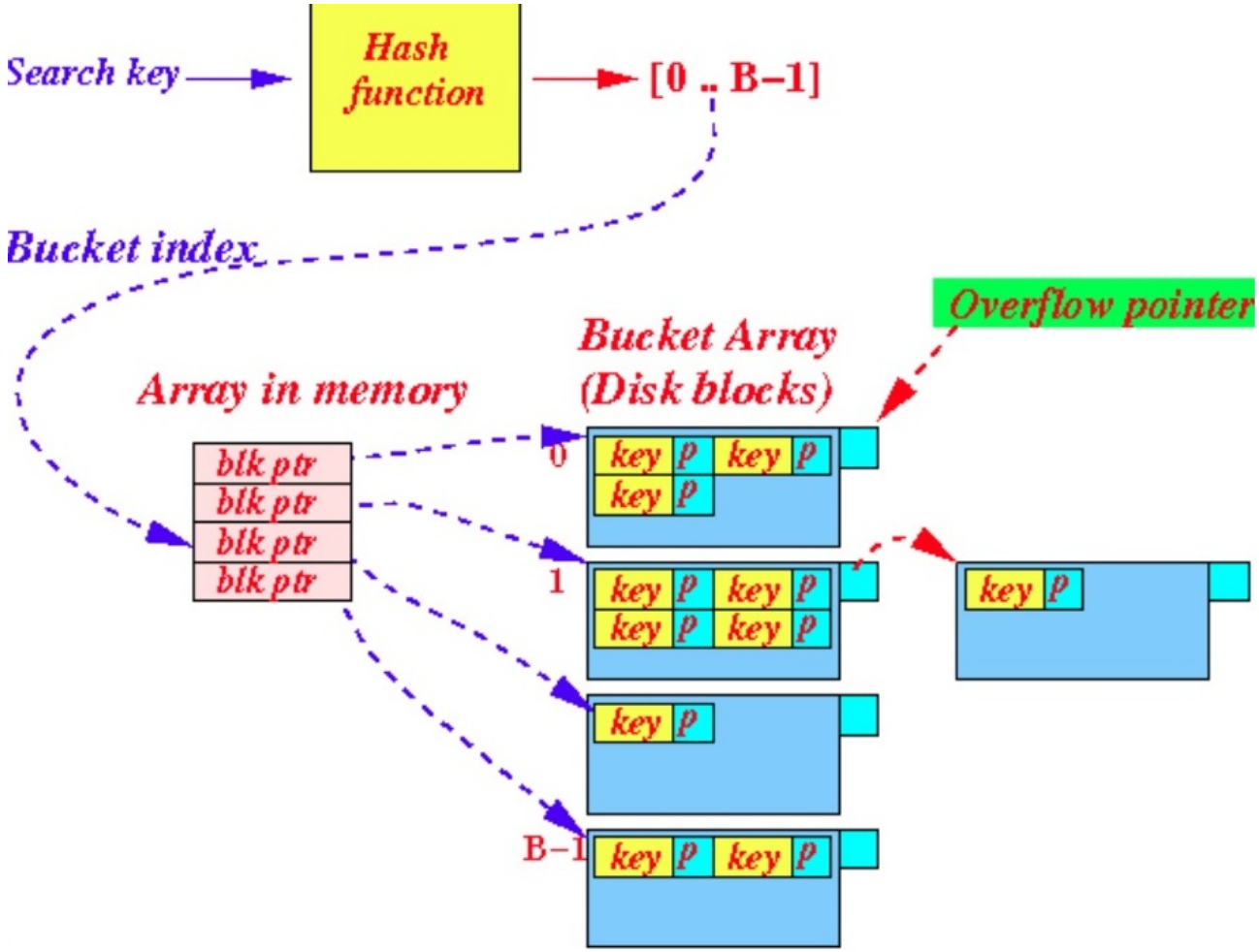
Storing Hash tables on disk

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



Storing Hash tables on disk

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



Storing Hash tables on disk

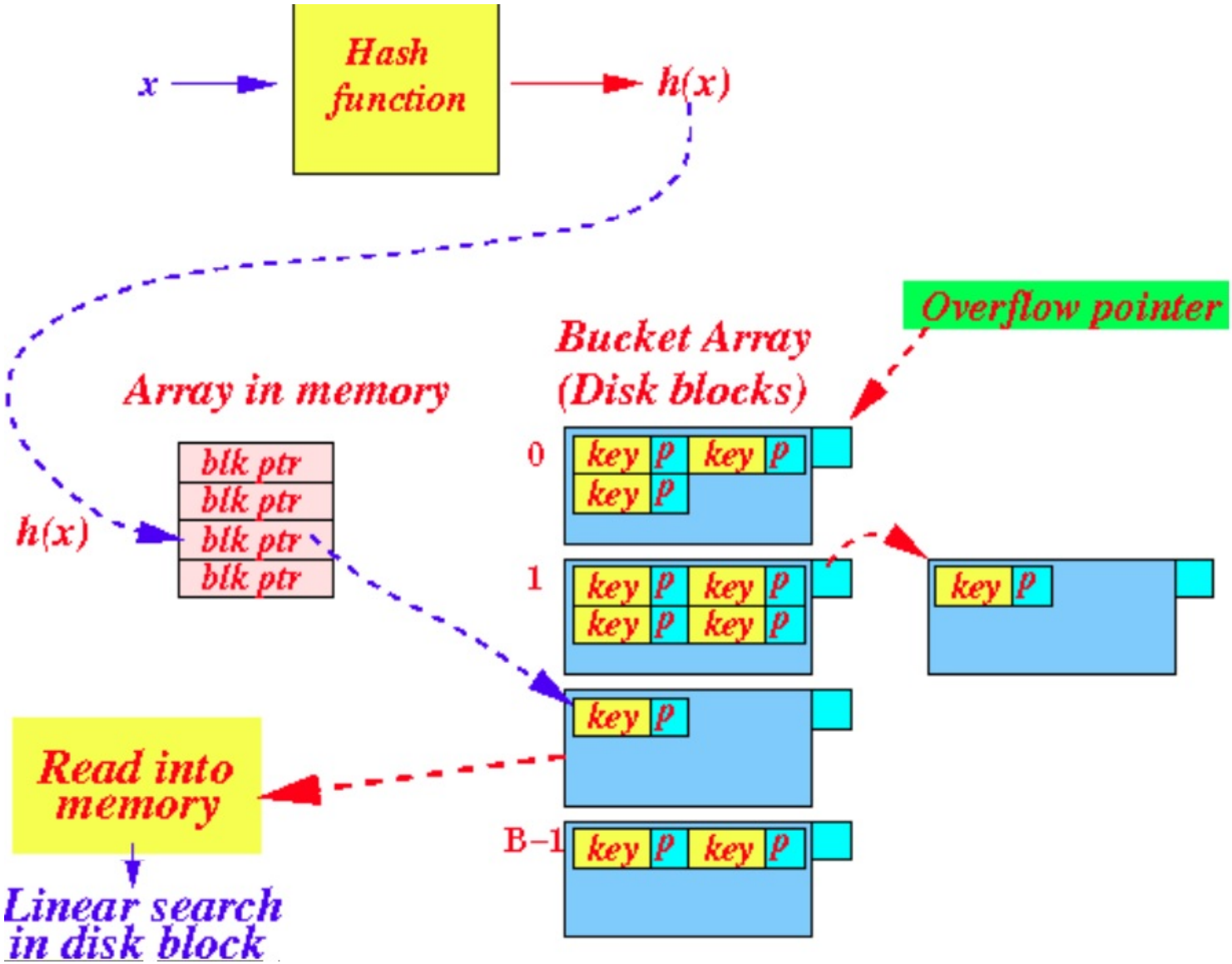
- The main memory array of pointers to blocks is typically called the bucket directory.
- Each entry in the bucket directory points to the first block of the corresponding bucket.
- The blocks themselves are stored on disk, and each block contains a pointer to the next block in the list.

Using a Hash Index

- How to use a Hash index to access a record:

  Given a *search key x*:
  - calculate the hash value of the key *h(x)* and then use it to index into the bucket directory.
    - *the bucket directory will give us the pointer to the first block of the corresponding bucket*
  - read the disk block(s) of of bucket *h(x)* into memory
  - search (a linear search algorithm is sufficient because memory search is fast) the *bucket h(x)* for $\langle$*x,RecordPtr(x)*$\rangle$
  - use *RecordPtr(x)* to access *x* on disk

Using a Hash Index



Inserting into Hash Table

- To insert a record with *key k*:
  - calculate the hash value of the key *h(k)*. We then use the hash value to index into the bucket directory.
  - insert record $\langle$ *k,RecordPtr(k)* $\rangle$ 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

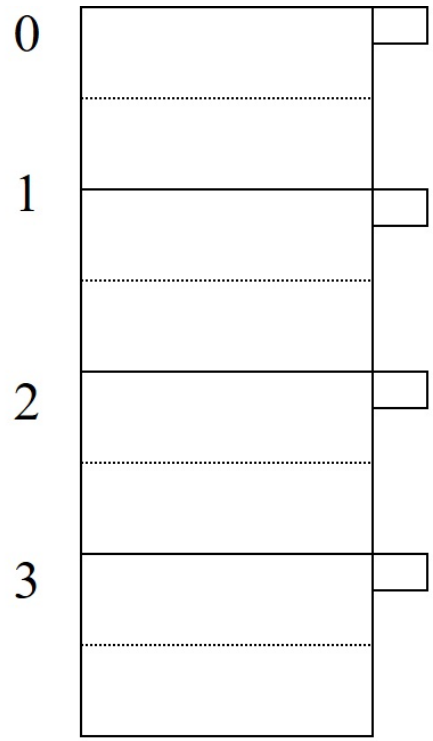
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



Insertion: Example 2 records/bucket

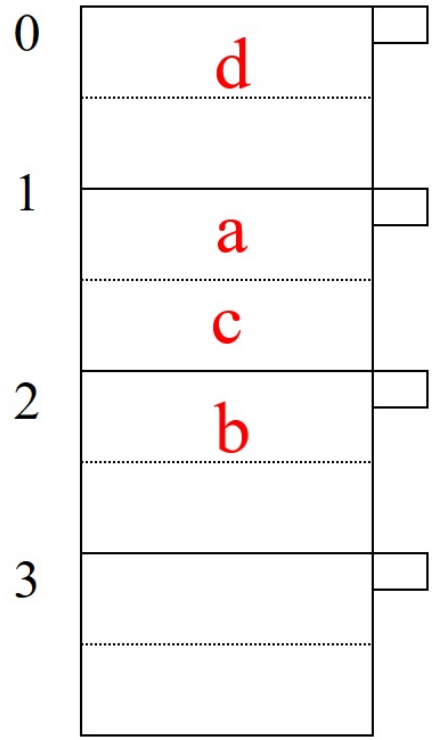
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



Deleting from a 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
    - Delete ⟨*k,RecordPtr(k)*⟩ from bucket
  - Optional
    - Consolidate overflow blocks of a bucket into few blocks

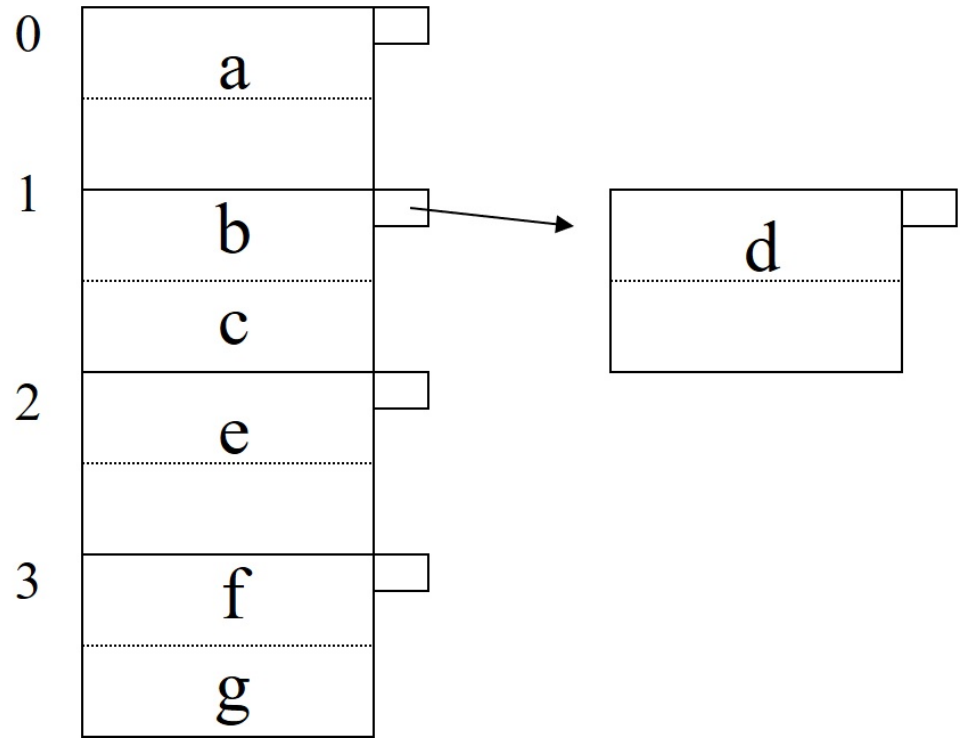
Deletion: Example 2 records/bucket

Delete:

e

f

c

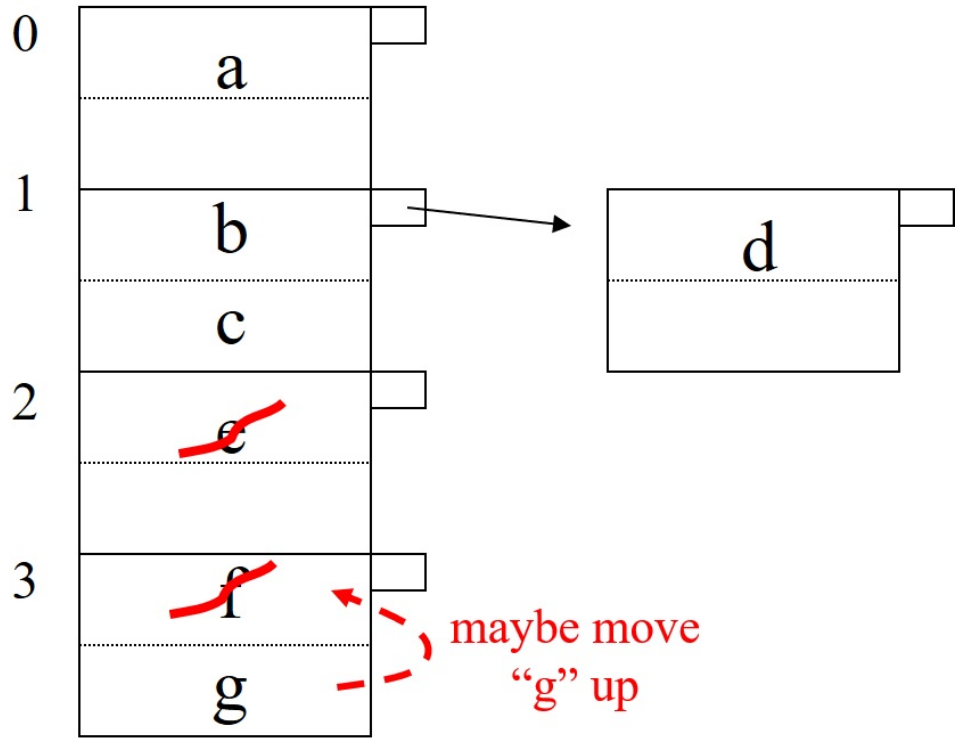


Deletion: Example 2 records/bucket

Delete:

e

f



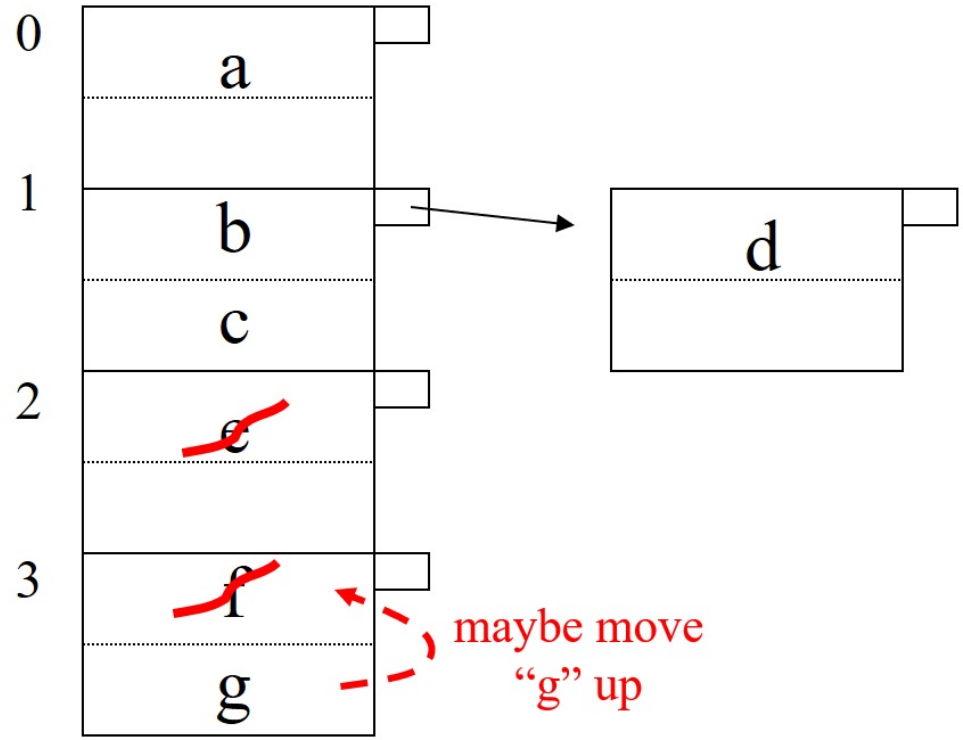
Deletion: Example 2 records/bucket

Delete:

e

f

c



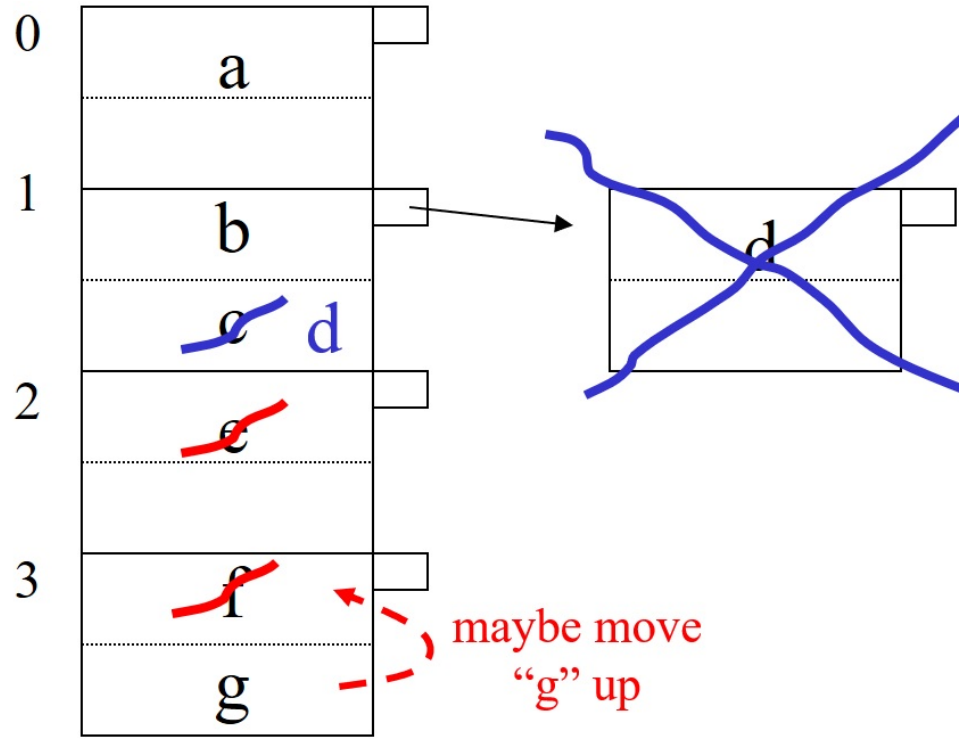
Deletion: Example 2 records/bucket

Delete:

e

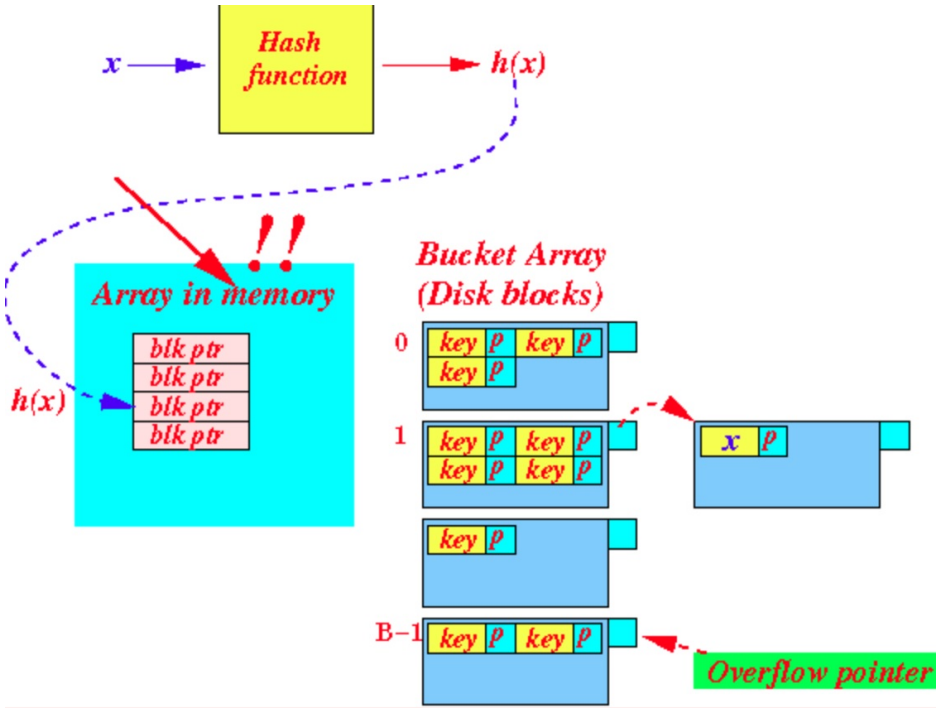
f

c



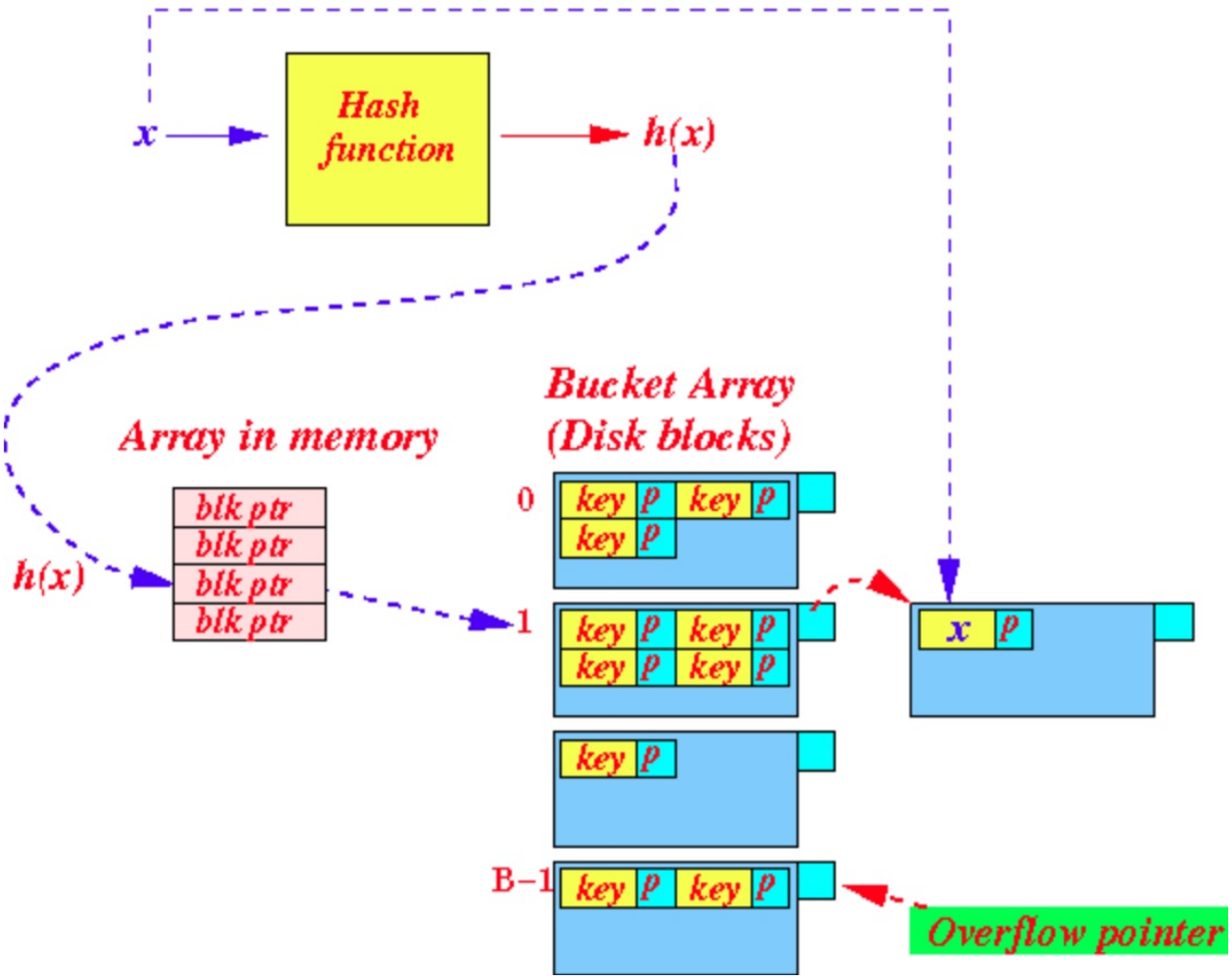
Performance of Hash Tables

- **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
    - *the access time to main memory is much faster than the access time to disk.*



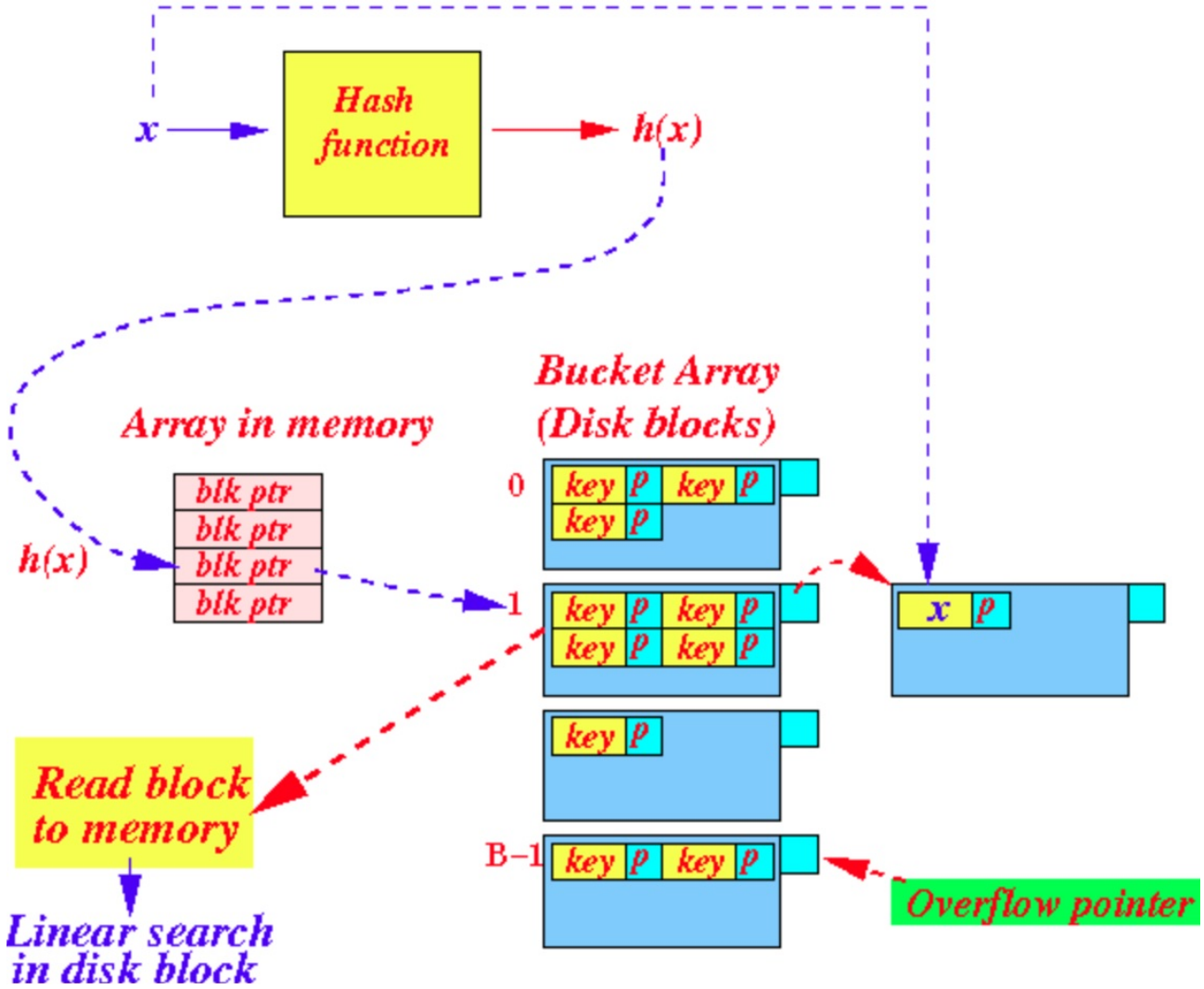
Performance of Hash Tables

- Suppose we look up using `key x`



Performance of Hash Tables

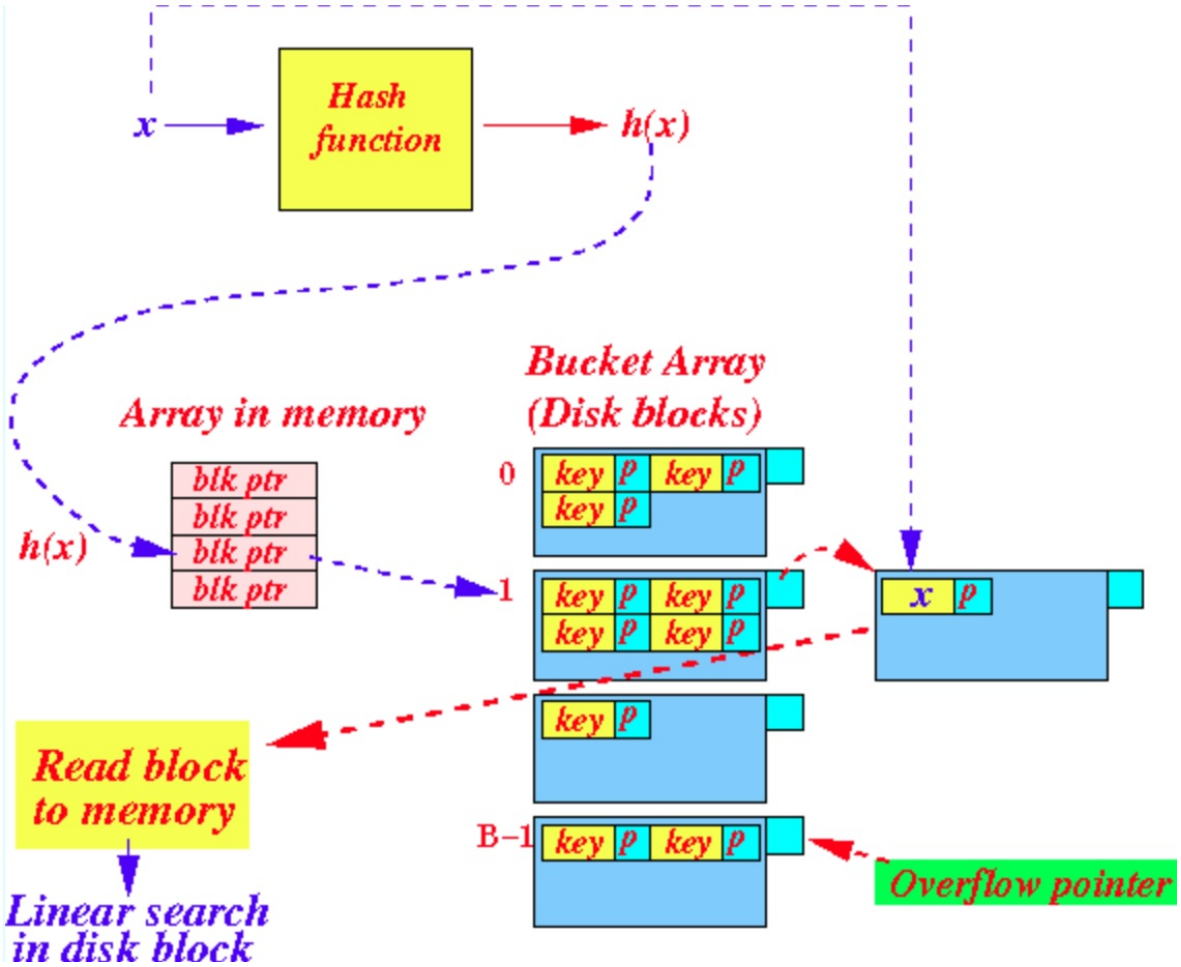
- We read in the `first index block` into the memory



- The search for *`key x`* results in failure (not found)

Performance of Hash Tables

- We read in the `next (overflow) index block` into the memory



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

Conclusion: Performance of Hash Tables

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

Optimal performing hash index

- The fastest hash index is when *no overflow* blocks are used
  - *1 block access* to read the index block (and find the record address of the data block)
  - *1 block access* to read the data block (containing the searched data) into memory
- Best performance of hash index
  - Total: *2 disk blocks accesses* to lookup any record

Hash index vs ordered preserving index (B^+-tree)

- Hash index is generally faster (less disk accesses) than ordered index (B^+-tree) for operations such as equality joins and lookups.
- Hash index cannot support ordered searches, like:

```
SELECT fname, lname
FROM   employee
WHERE  Salary >  100000;
```

  - *hash indexes do not store the data in any particular order*

How do we cope with growth?

- Databases grow or shrink with time.
- Overflows and periodic reorganizations with a new hash functions.
- Dynamic hashing (`B` is allowed to vary)
  - Extendible hashing
  - Linear hashing

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
  - expensive, disrupt normal operations.
- Solution: Dynamic hashing

Dynamic Hashing

- Hard to keep all elements within one bucket-block if file grows and hash table is static
  - In static hashing, function h maps search key values to a fixed set of B bucket addresses.
- Dynamic Hashing: hashing techniques that allow the size of the `hash table` to change with relative low cost
- We will discuss two `dynamic` hashing techniques where
  - The size of the hash table can increase (and decrease)
  - When the size of hash table is changed:
    - Only a *fraction* of the existing search keys needs to be `re-mapped (re-hashing)`

Dynamic Hashing Techniques

- Extendible hashing
  - Can eliminate the need/use of *overflow block* completely
  - However, the size of the *hash table* increases exponentially.
- Linear hashing
  - The size of the *hash table* increases linearly
  - However, cannot eliminate the need/use of *overflow block*

Extendible Hash Tables

- Hash function generates values over a large range, typically $n\text{-}bit$ integers, with $n\text{-}bit=32$.
- At any time, use only a prefix of the hash function to index into a table of buckets addresses (Bucket Index/Bucket address table/Directory).
  - Directories: Store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when directory expansion takes place.
    - Each table entry points to one bucket, i.e., contains a bucket pointer.
  - Buckets: Are used to hash the actual data.

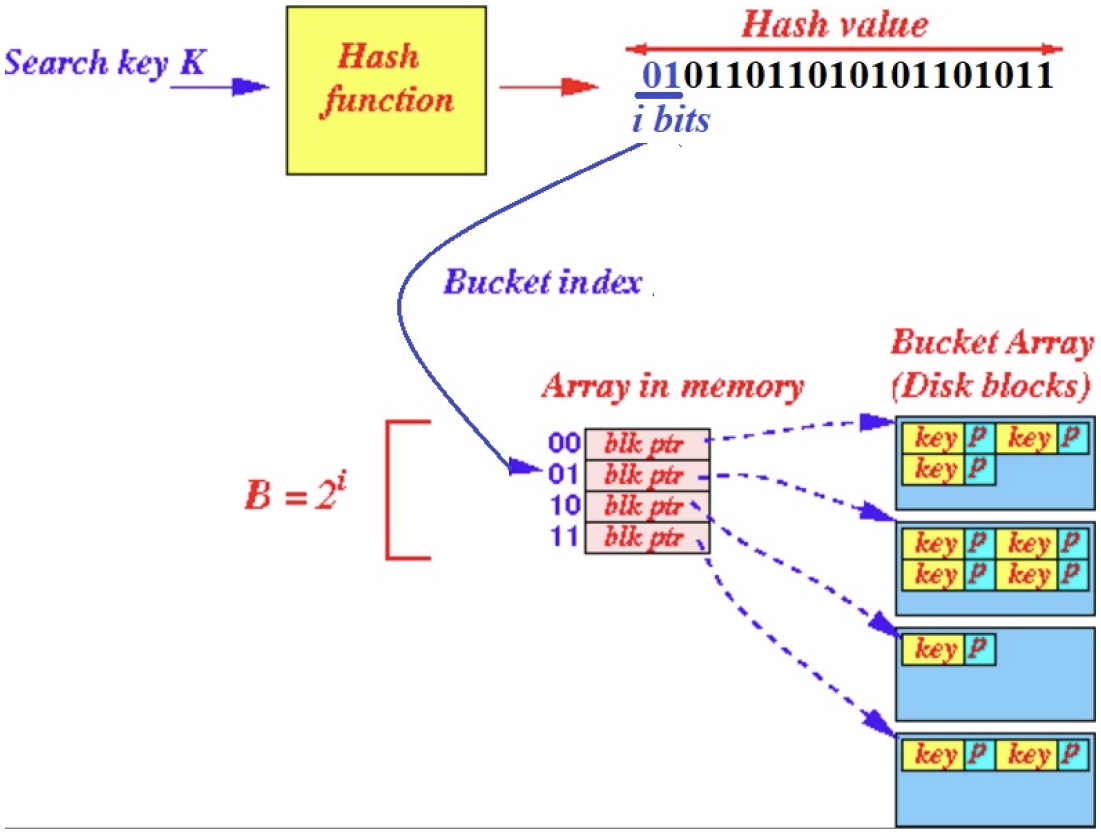
Extendible Hash Tables

- Splits buckets instead of letting chains to grow forever.
- Let the length of the prefix be i-bit, $0 < i \leq n$
- Bucket address table size = 2^i. Initially $i = 1$.
  - `hash function` computes a sequence of n-bit, but only **first** i **leftmost bits** are used at any time to index into the `bucket array`
- Value of i grows and shrinks as the size of the database grows and shrinks.
- Multiple entries in the bucket address table may point to a bucket.
  - Certain buckets may share a `block` if small enough to reduce space - if so, block header contains an indication of this
- Thus, actual number of buckets is $< 2^i$ because of some buckets may be shared or empty.

Extendible Hash Tables

- Dynamic adaptation
  - The table dynamically adjusts its size by splitting buckets when full and merging them when empty, improving average case performance compared to static hash tables.
- Localized updates:
  - Reshuffling during splits and merges affects only the involved buckets, minimizing data movement overhead.
- Reshuffling bucket entries on split and increase the number of bits to examine.

Hash Function used in Extendible Hashing



- The `bucket index` consists of the **first** i **leftmost 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 Extendible hashing

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

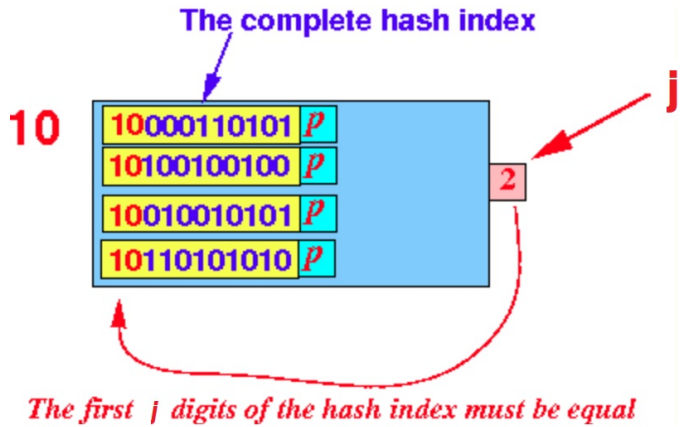
Parameters used in Extendible Hashing

- There are 2 integers used in `Extendible Hashing`
  1) Global parameter i:
     - Associated with the Bucket Index/Directories.
     - Denote 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 Extendible Hashing

2) The bucket label parameter j:

- Associated with the buckets.
- Denote the number of bits of hash value used to determine membership in a Bucket.



- Always less than or equal to the Global parameter.
  - `global parameter` $i \geq$ `any bucket label parameter` j

Inserting into Extendible Hash Table

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

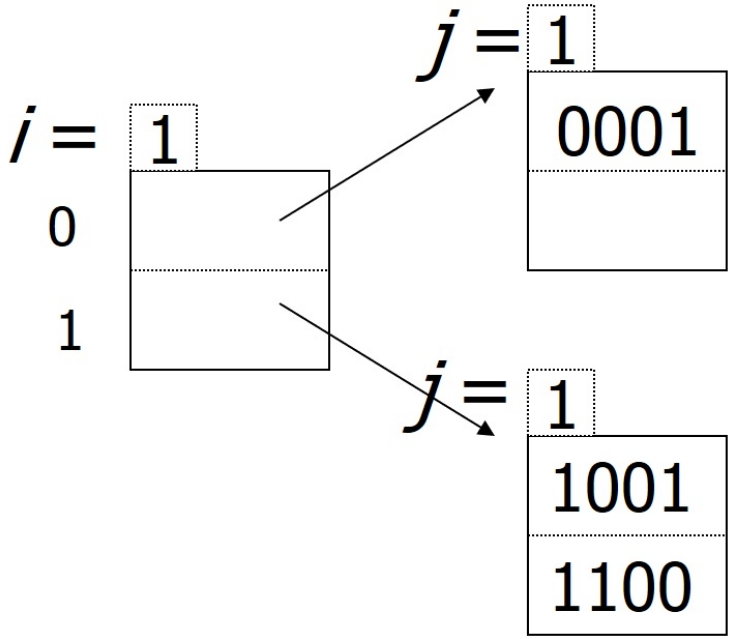
Insertion: Case 1: $j < i$

- split *block* x in two blocks
- distribute records in x to the two new blocks based on value of their (*j+1*)-st leftmost bits
- update header of each new block to *j+1*
- adjust pointers in the hash buckets so that entries that used to point to x; now point either to x or the new block, depending on their (*j+1*)-st leftmost bits
- repeat this process if still no room in appropriate block for new record.

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

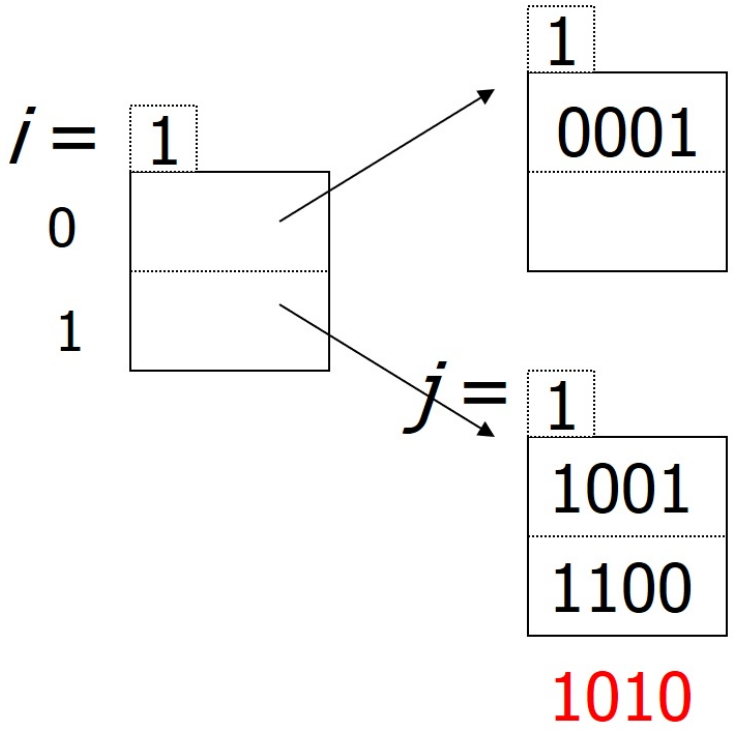
- increment *`i`* by 1
- double length of the hash buckets.
- in the new hash buckets, entry indexed by both *`w0`* and *`w1`* each point to same `block` that old entry *`w`* pointed to (`block` is shared)
- apply `case 1` to split *`block x`*

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



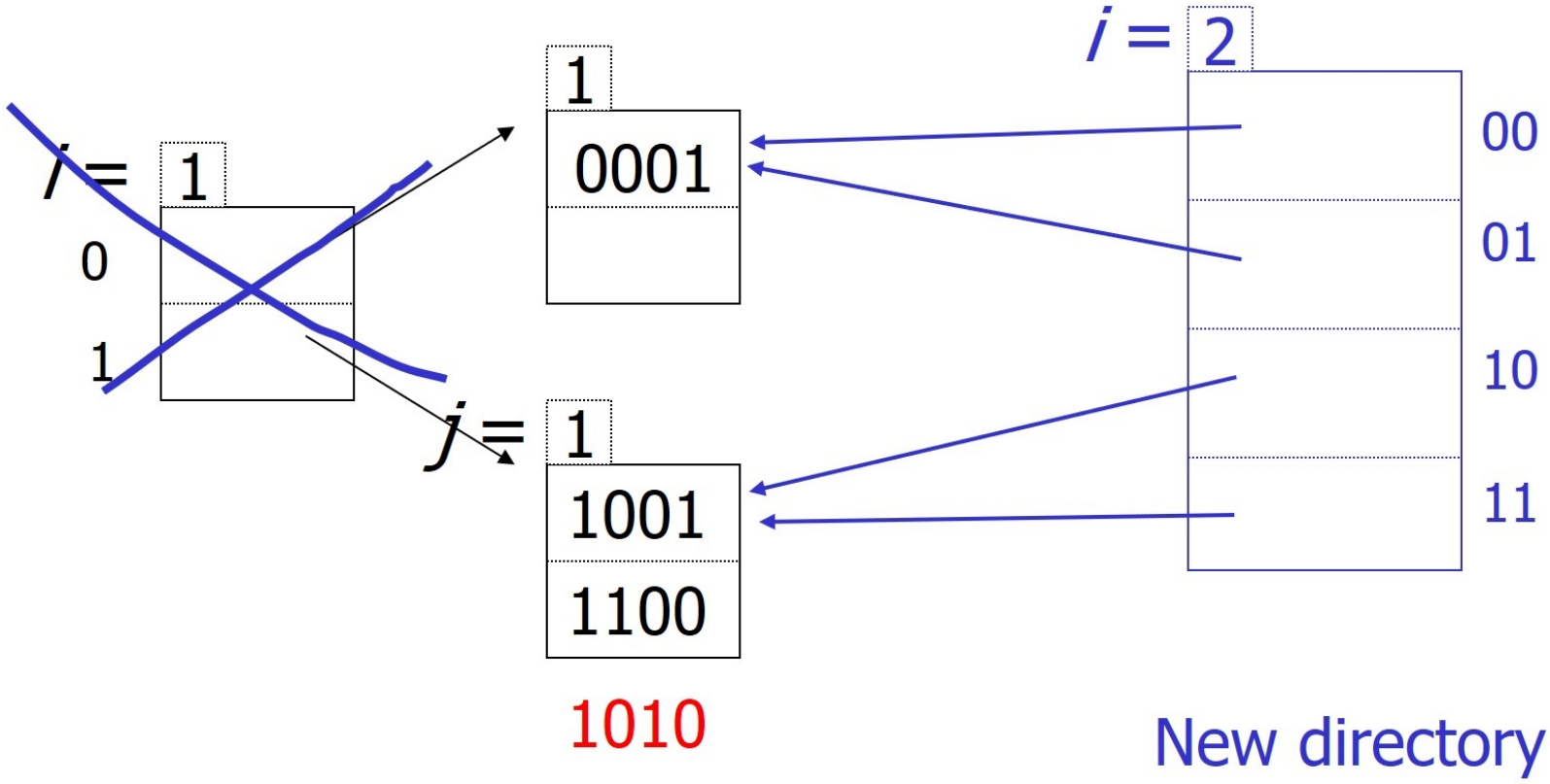
Insert h(key)=1010

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



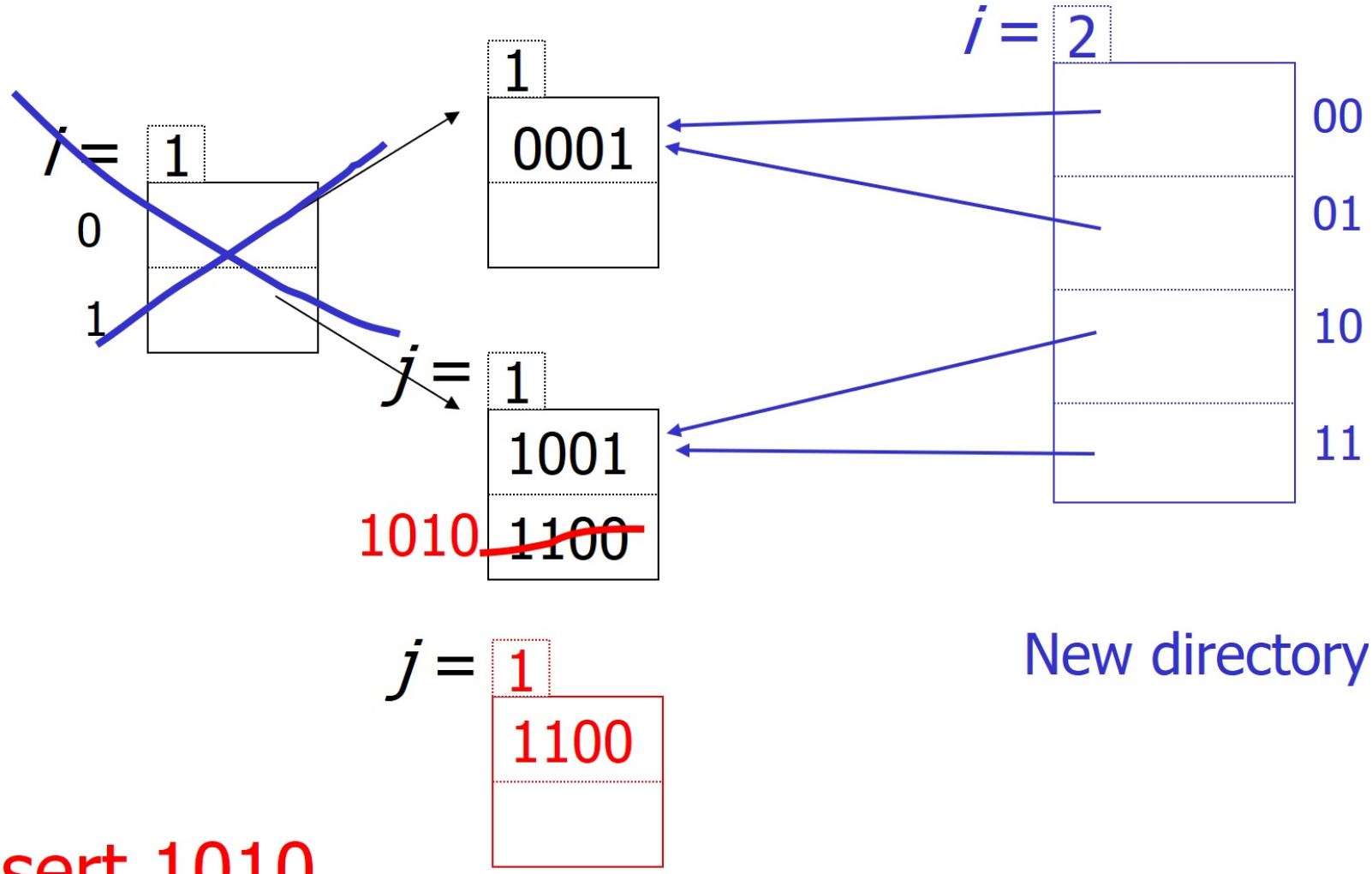
Insert h(key)=1010

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

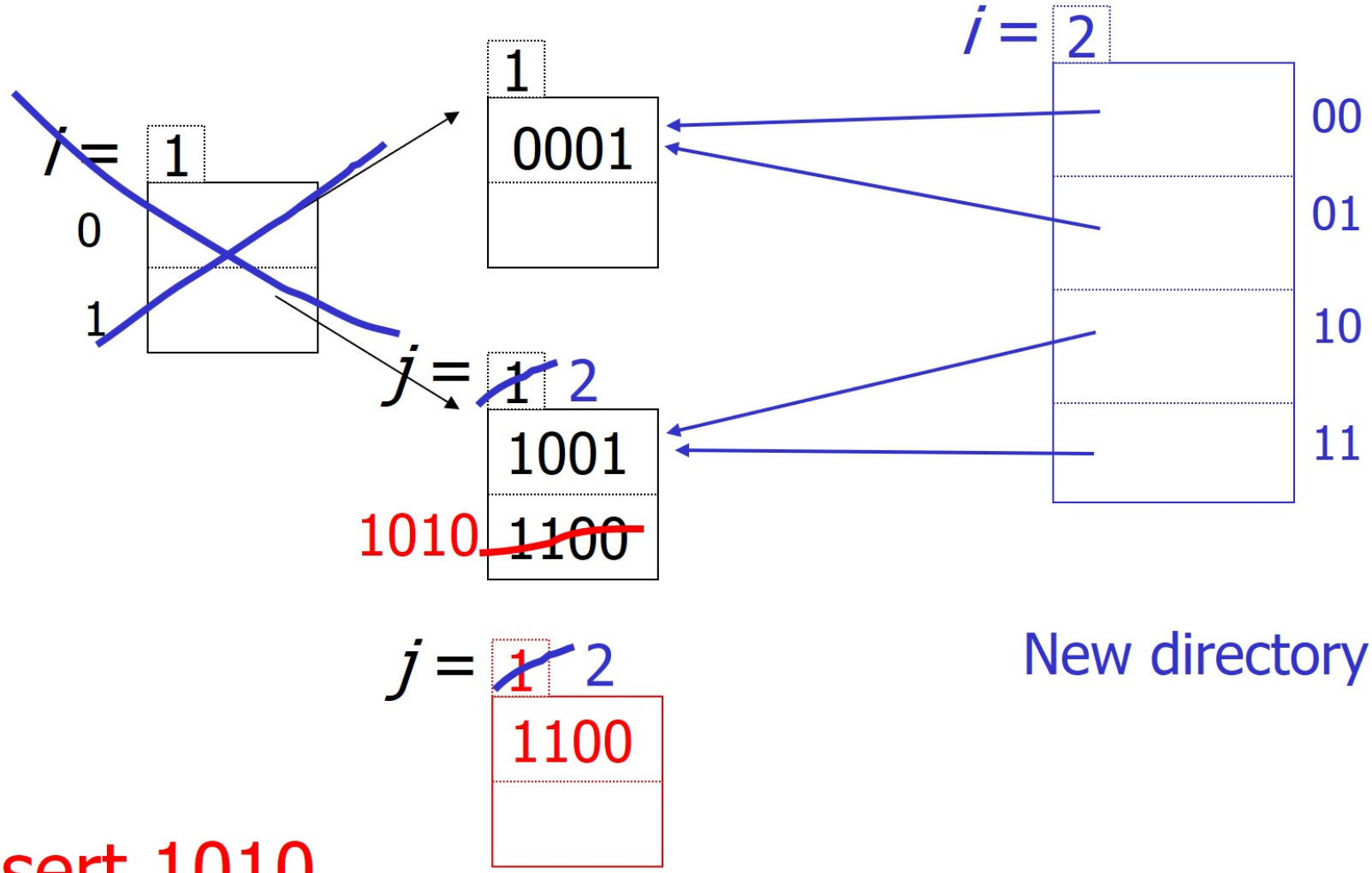


Insert h(key)=1010

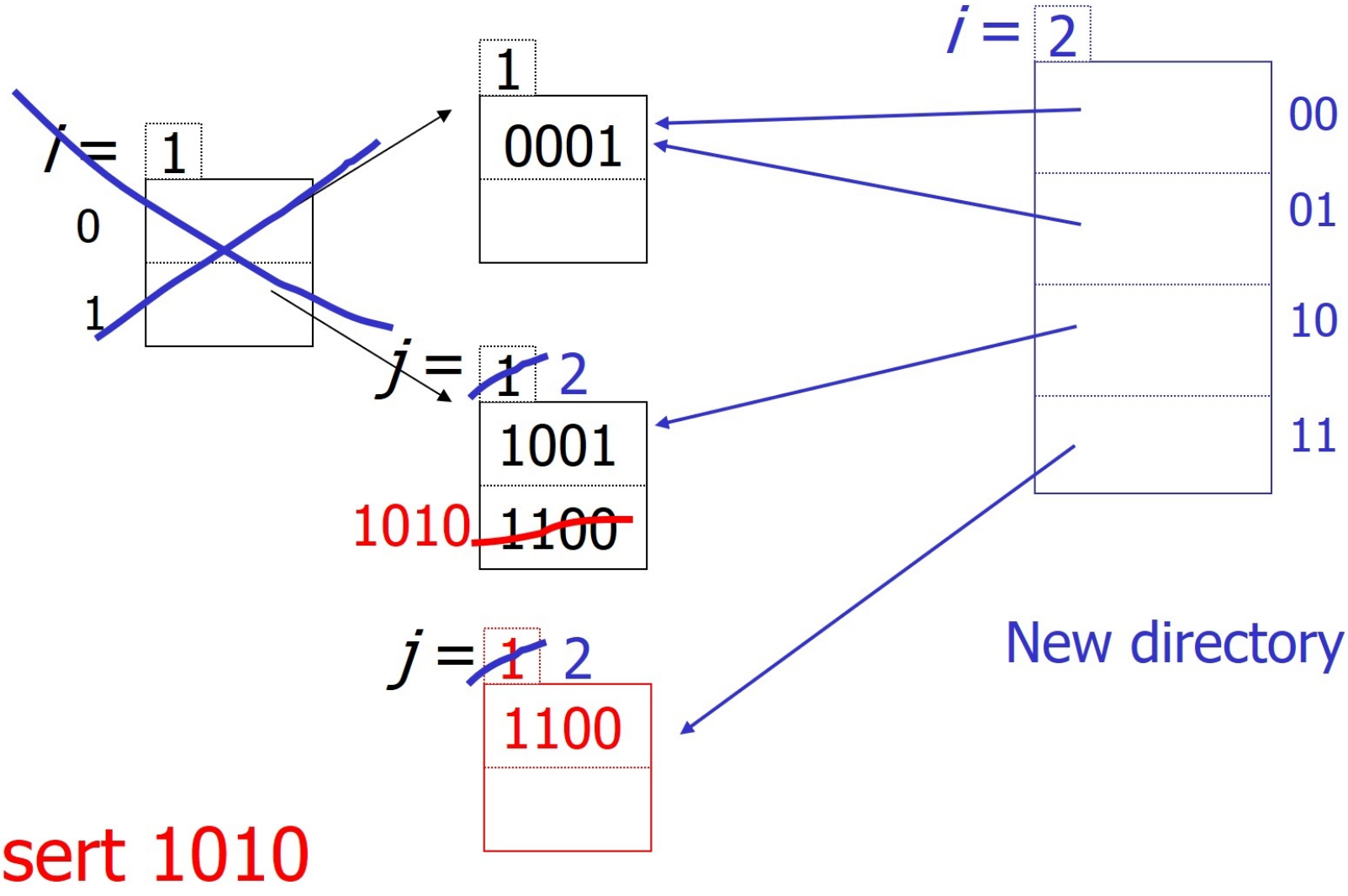
Example: h(k) is 4 bits; 2 keys/bucket



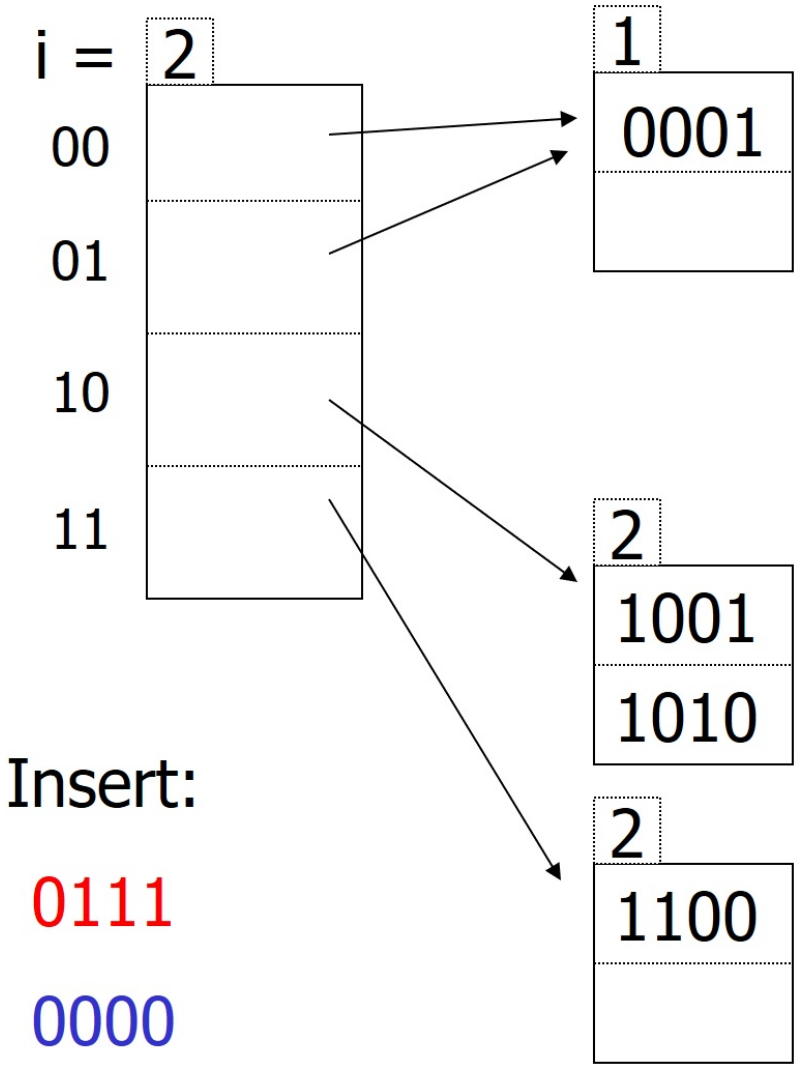
Example: h(k) is 4 bits; 2 keys/bucket



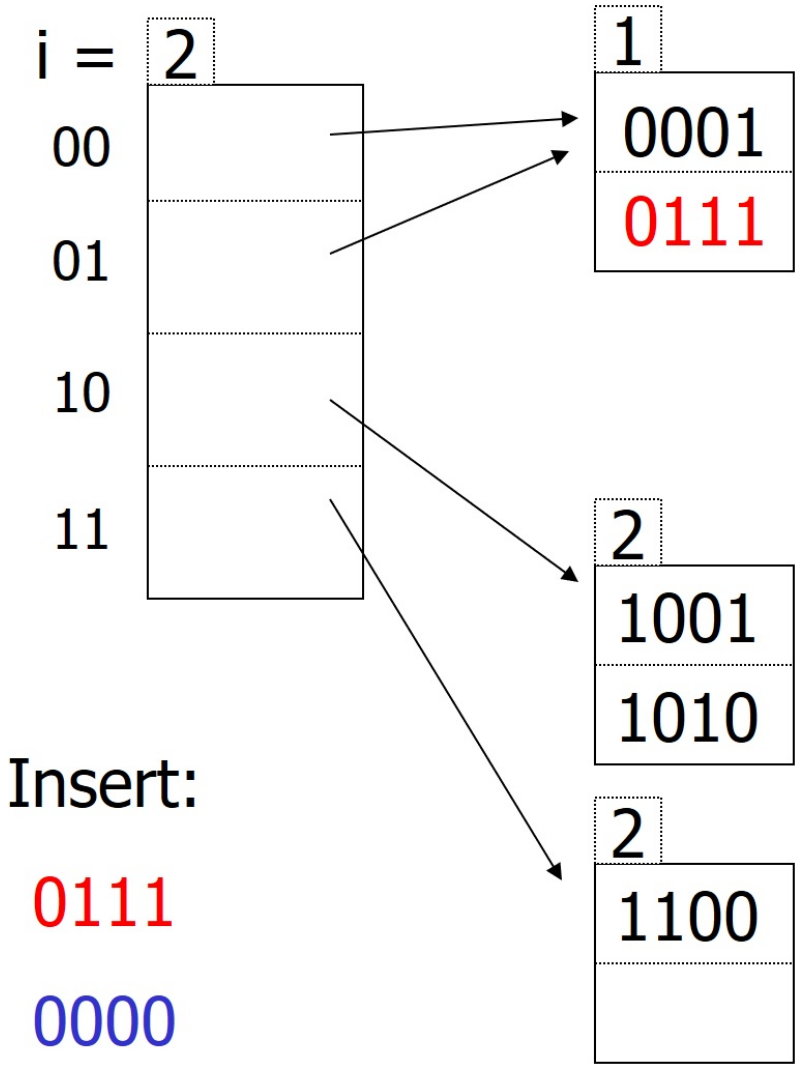
Example: h(k) is 4 bits; 2 keys/bucket



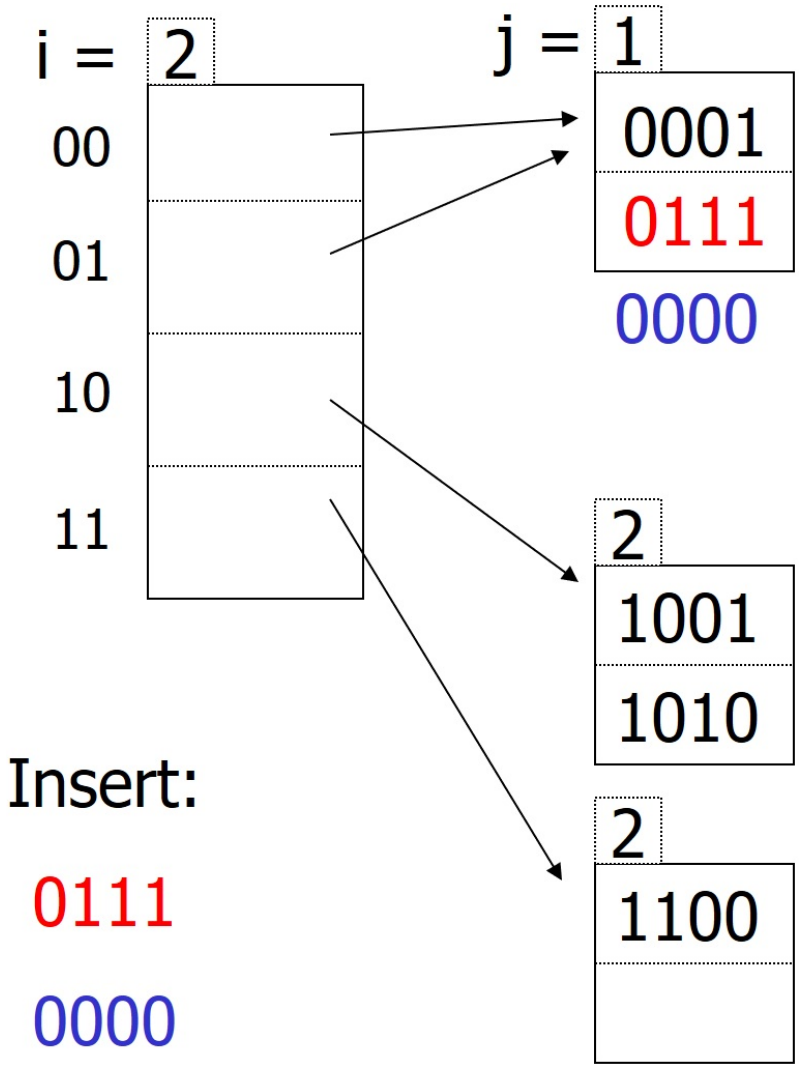
Example: h(k) is 4 bits; 2 keys/bucket



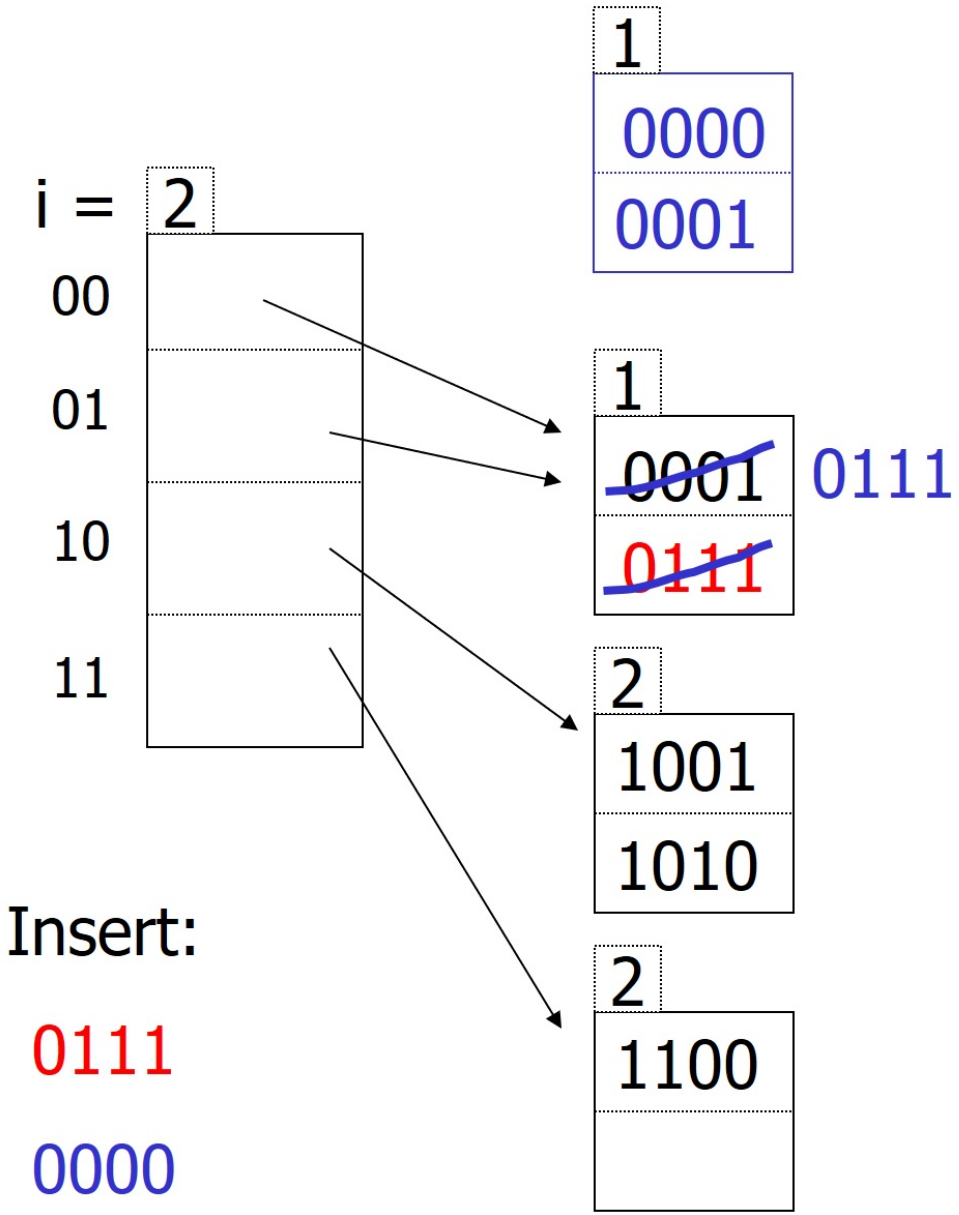
Example: h(k) is 4 bits; 2 keys/bucket



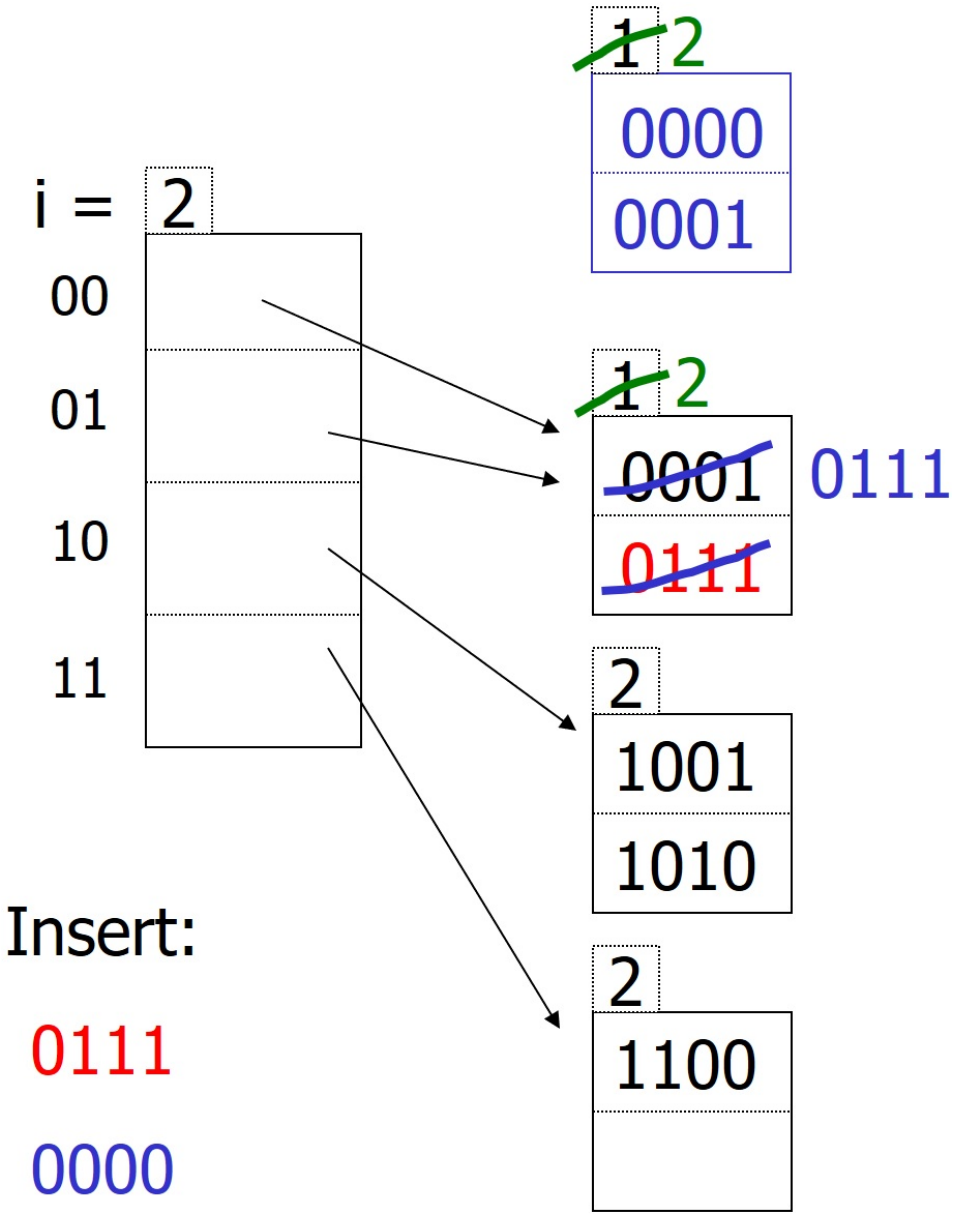
Example: h(k) is 4 bits; 2 keys/bucket



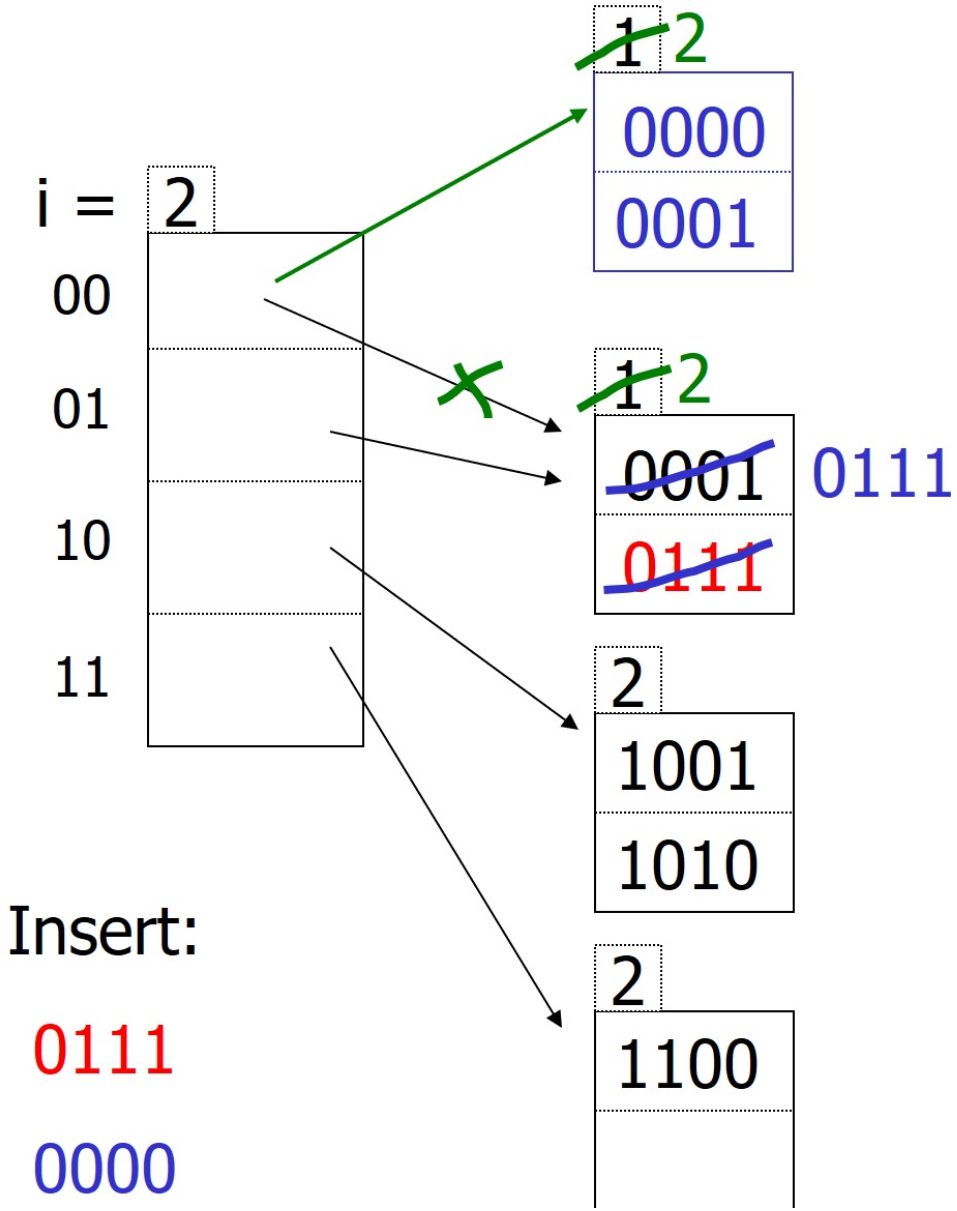
Example: h(k) is 4 bits; 2 keys/bucket



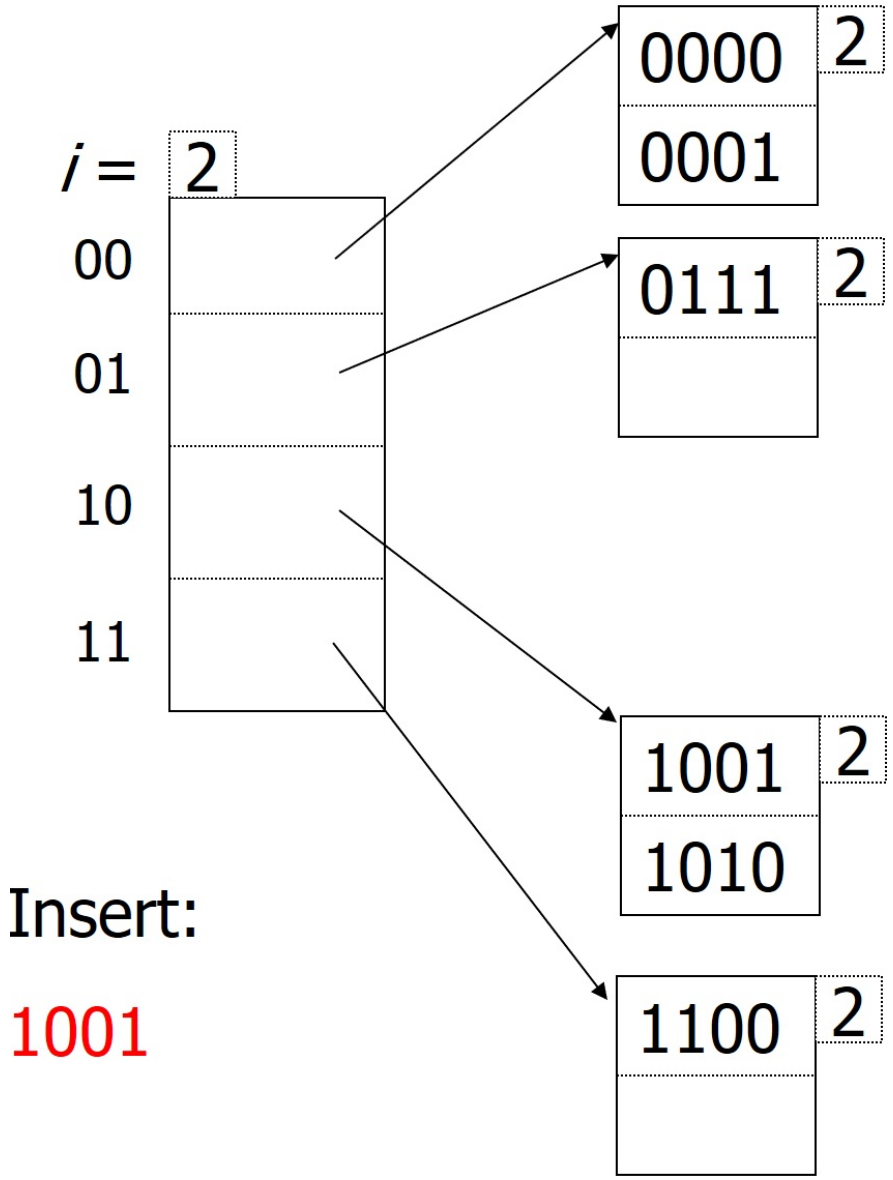
Example: h(k) is 4 bits; 2 keys/bucket



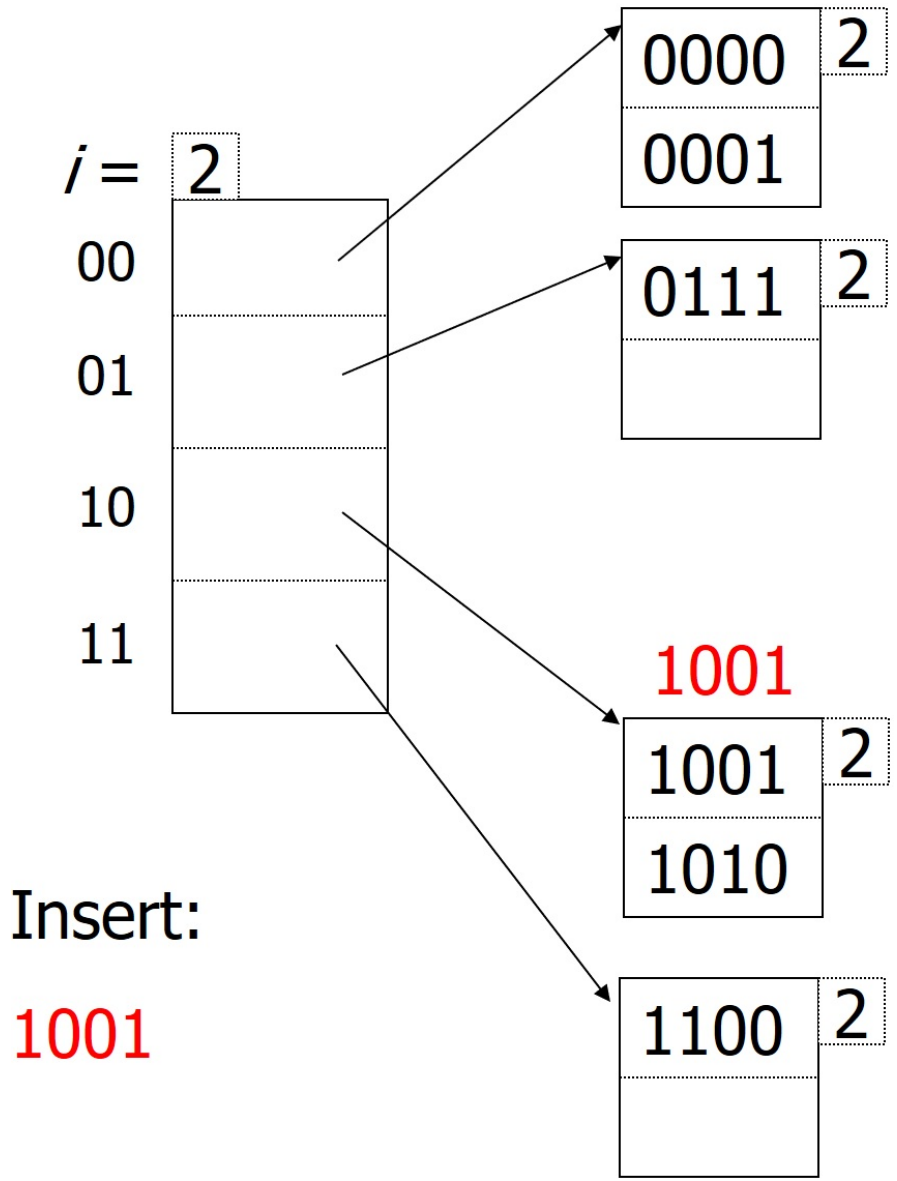
Example: h(k) is 4 bits; 2 keys/bucket



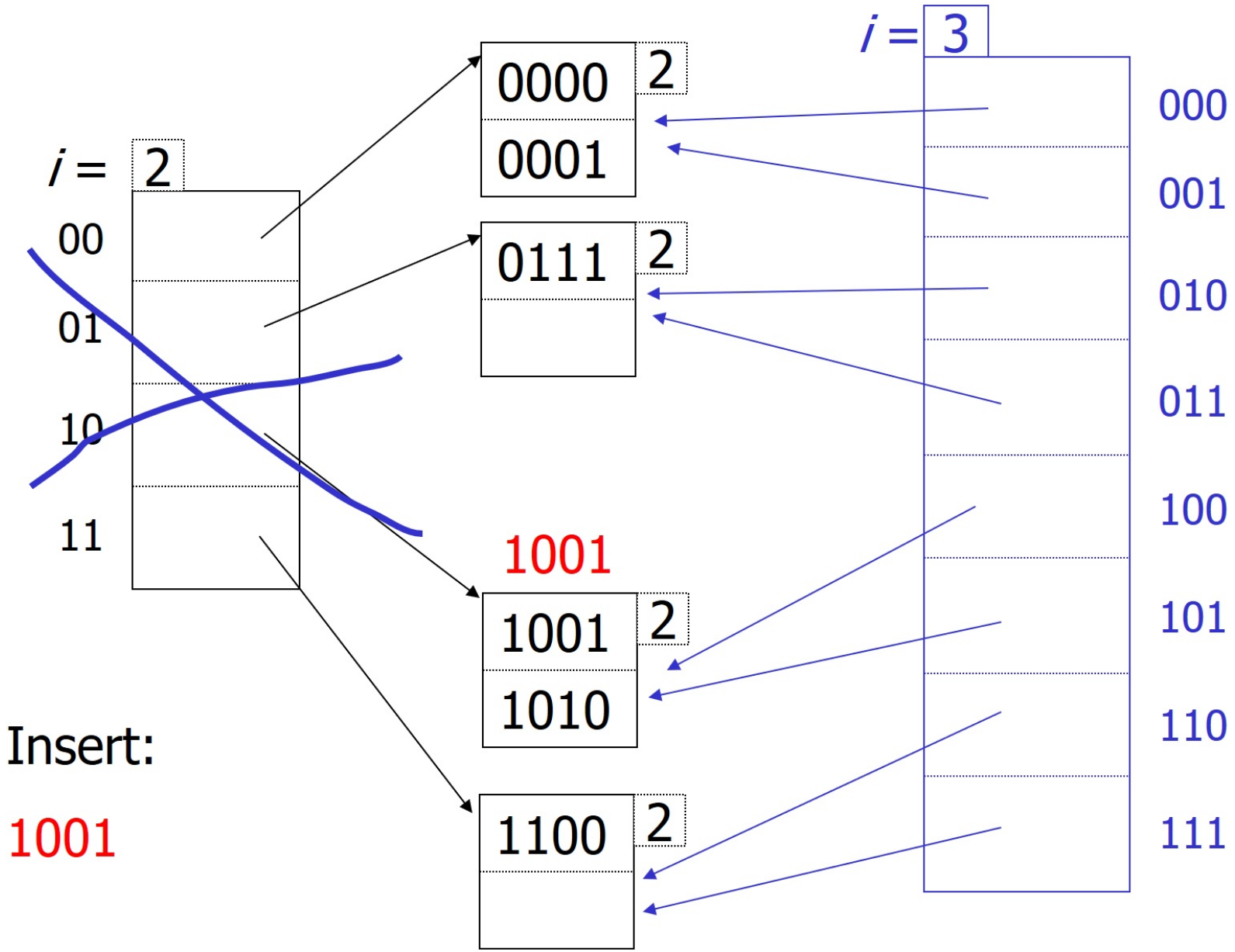
Example: h(k) is 4 bits; 2 keys/bucket



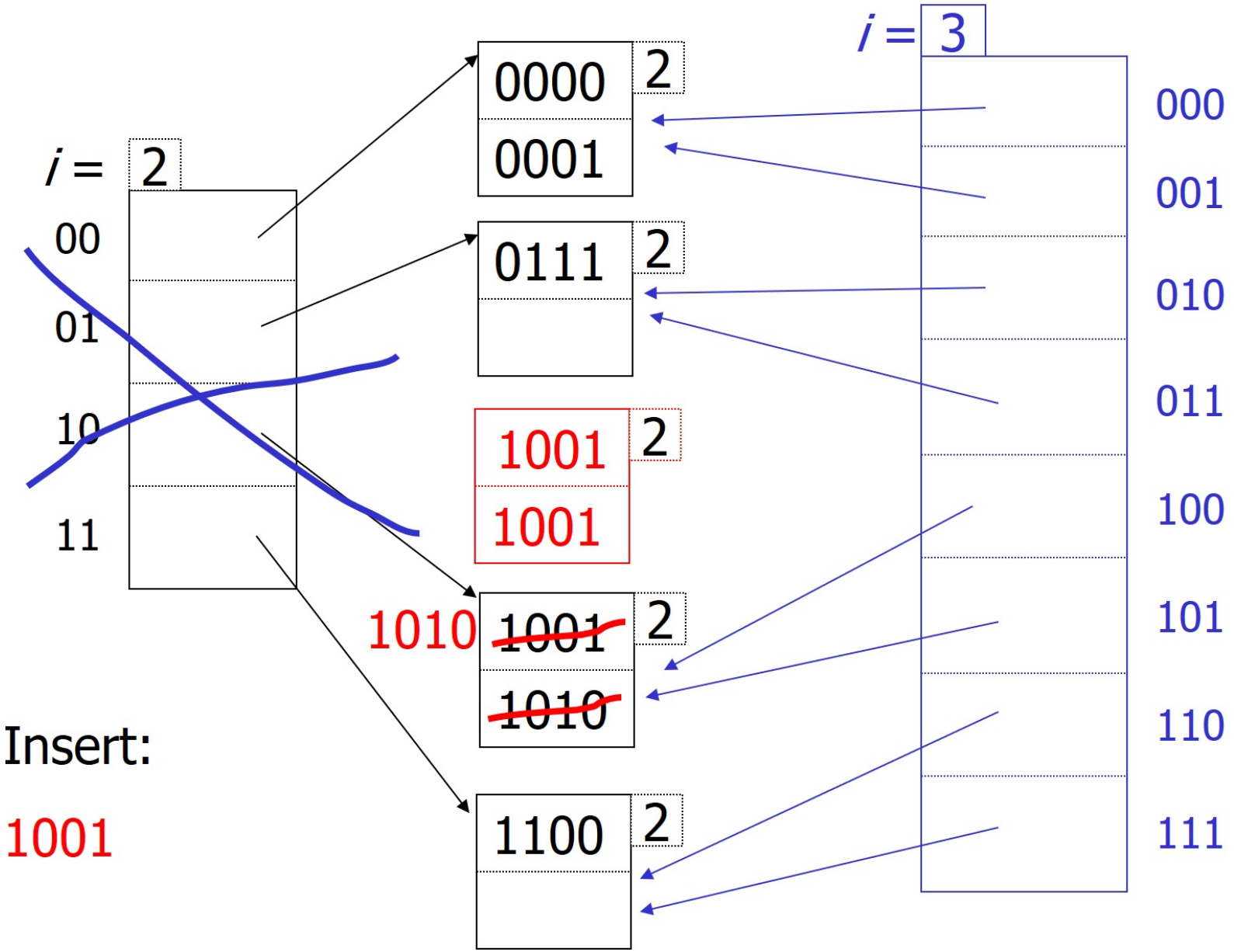
Example: h(k) is 4 bits; 2 keys/bucket



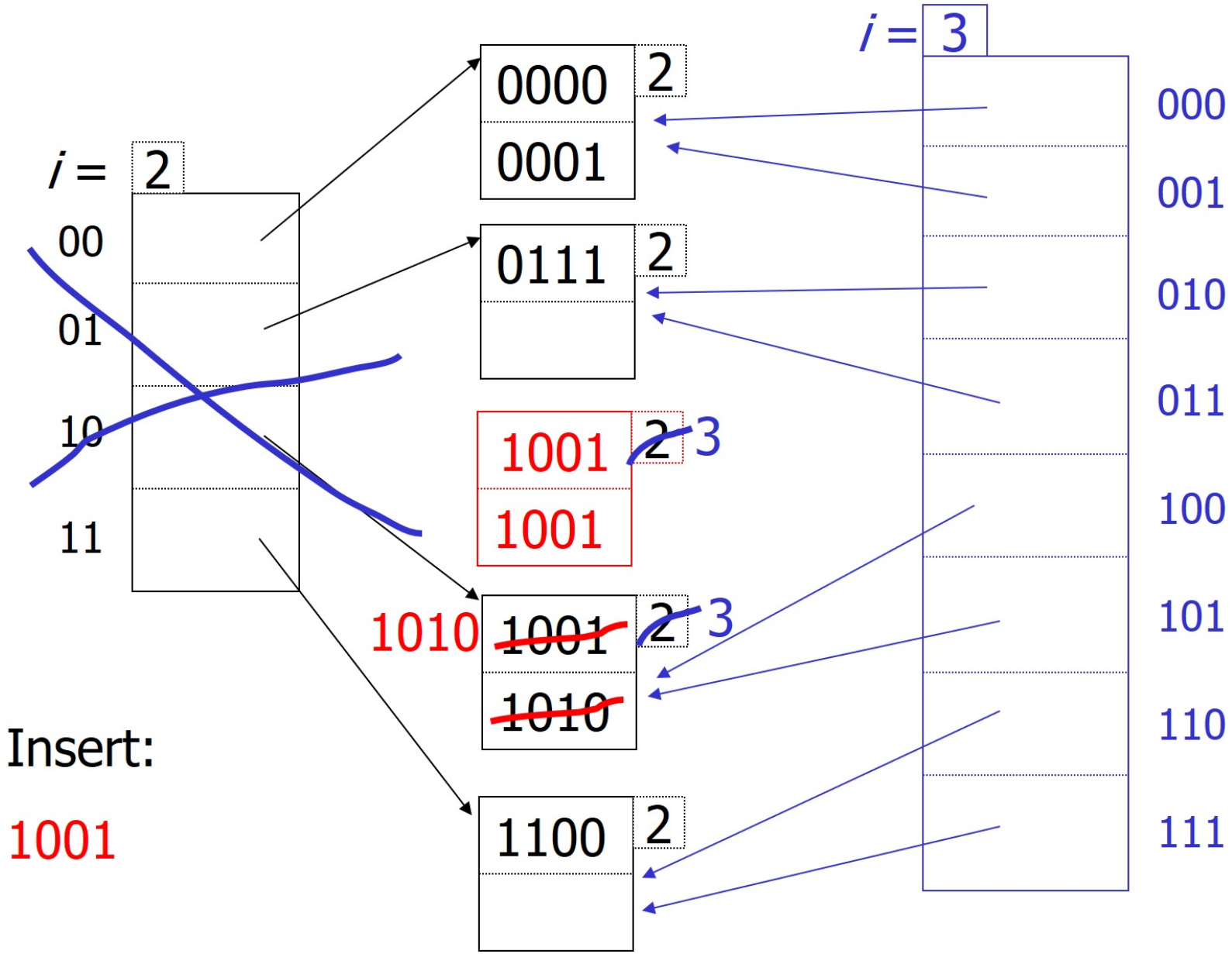
Example: h(k) is 4 bits; 2 keys/bucket



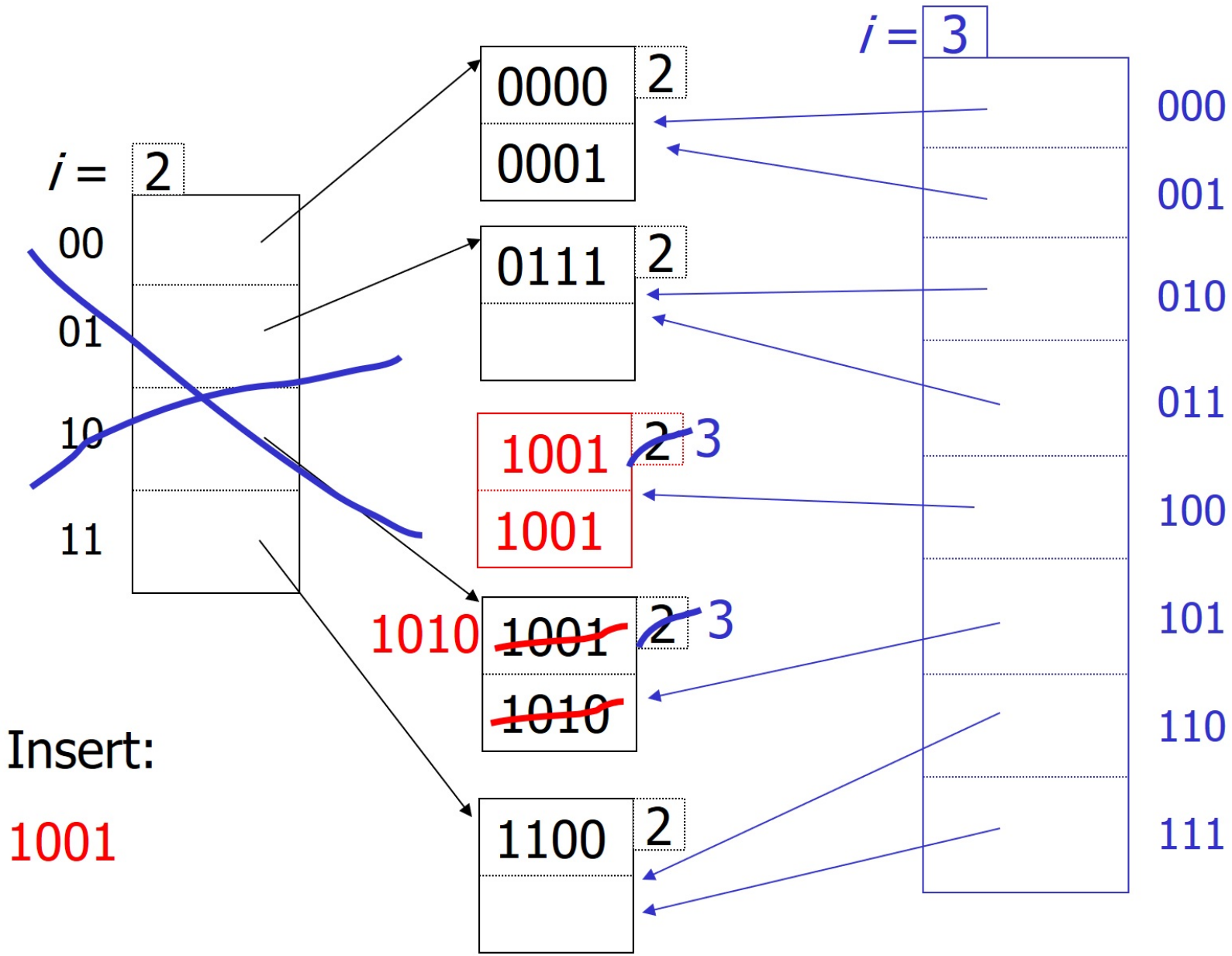
Example: h(k) is 4 bits; 2 keys/bucket



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



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

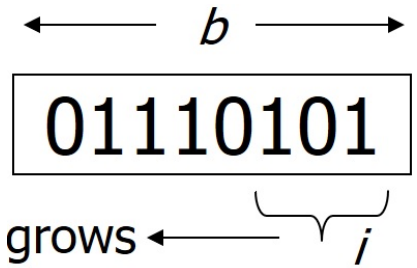


Summary: Extendible Hashing

- ☺ Hash performance does not degrade with growth of file
  - Still one block per bucket to lookup a record (fast)
- ☺ Minimal space overhead
- ☹ Extra level of indirection to find desired record.
- ☹ Doubling size
  - the size of the hash table will double each time when we extend the table
  - much work to be done, especially if i is large
  - as size increases, it may no longer fit in memory
- ☹ Multiple entries with same hash value cause problems
  - Unfixable issue for duplicates
  - Results in Extendible hashing directory exploding!
- Better solution:
  - linear hashing: is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows.

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`
  - will have higher search overhead than Extendible Hashing
- 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
- i: number of bits used in the hash value
  - i is a derived parameter: $i = \lceil \log_2 n \rceil$
- 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)

Important property help to understand Linear Hashing

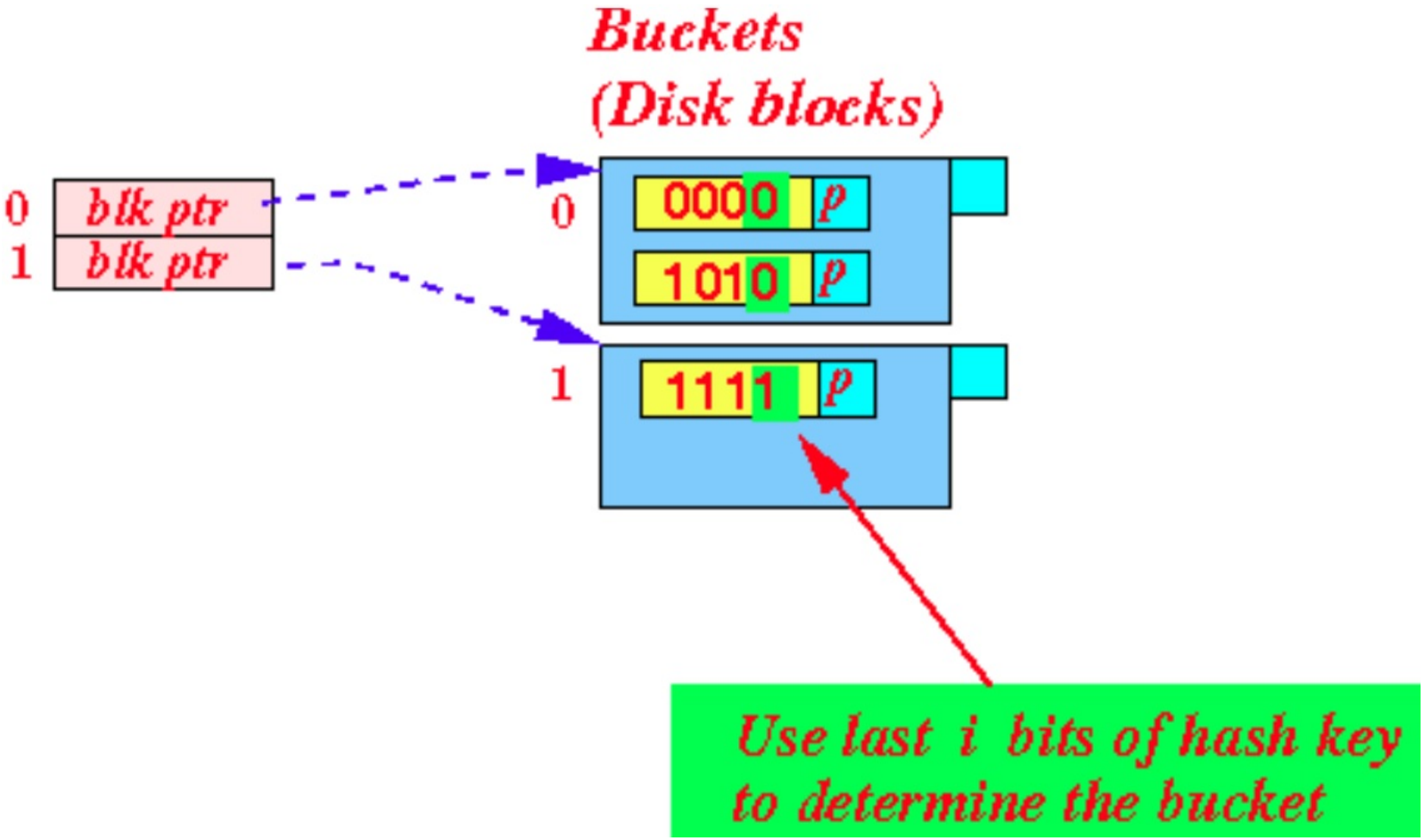
- When the number $(n-1)$ is written as i bits `binary number`, the `first bit` in the `binary number` is always "1"

```
 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"

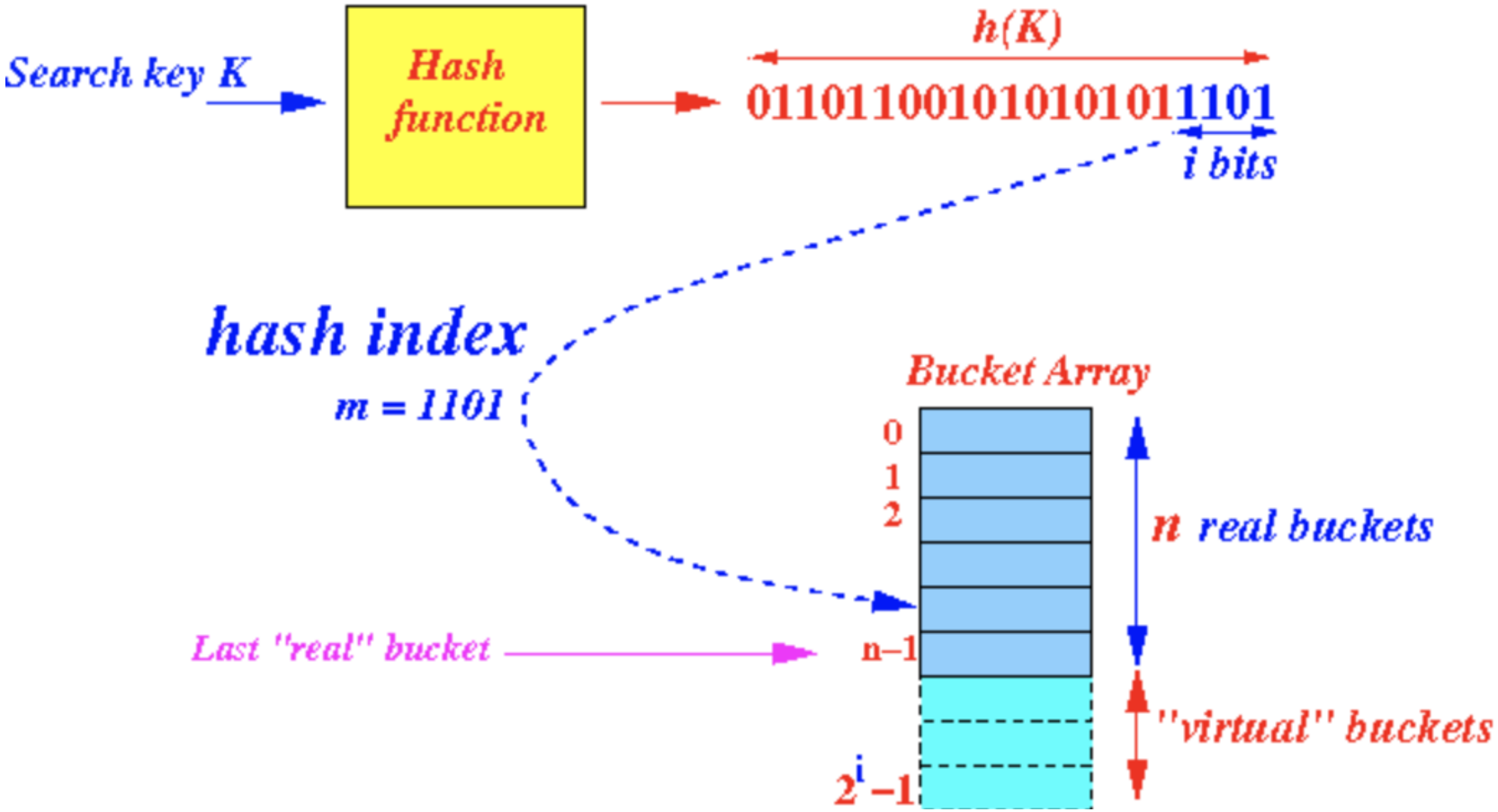
Example of parameters in the Linear hashing method

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



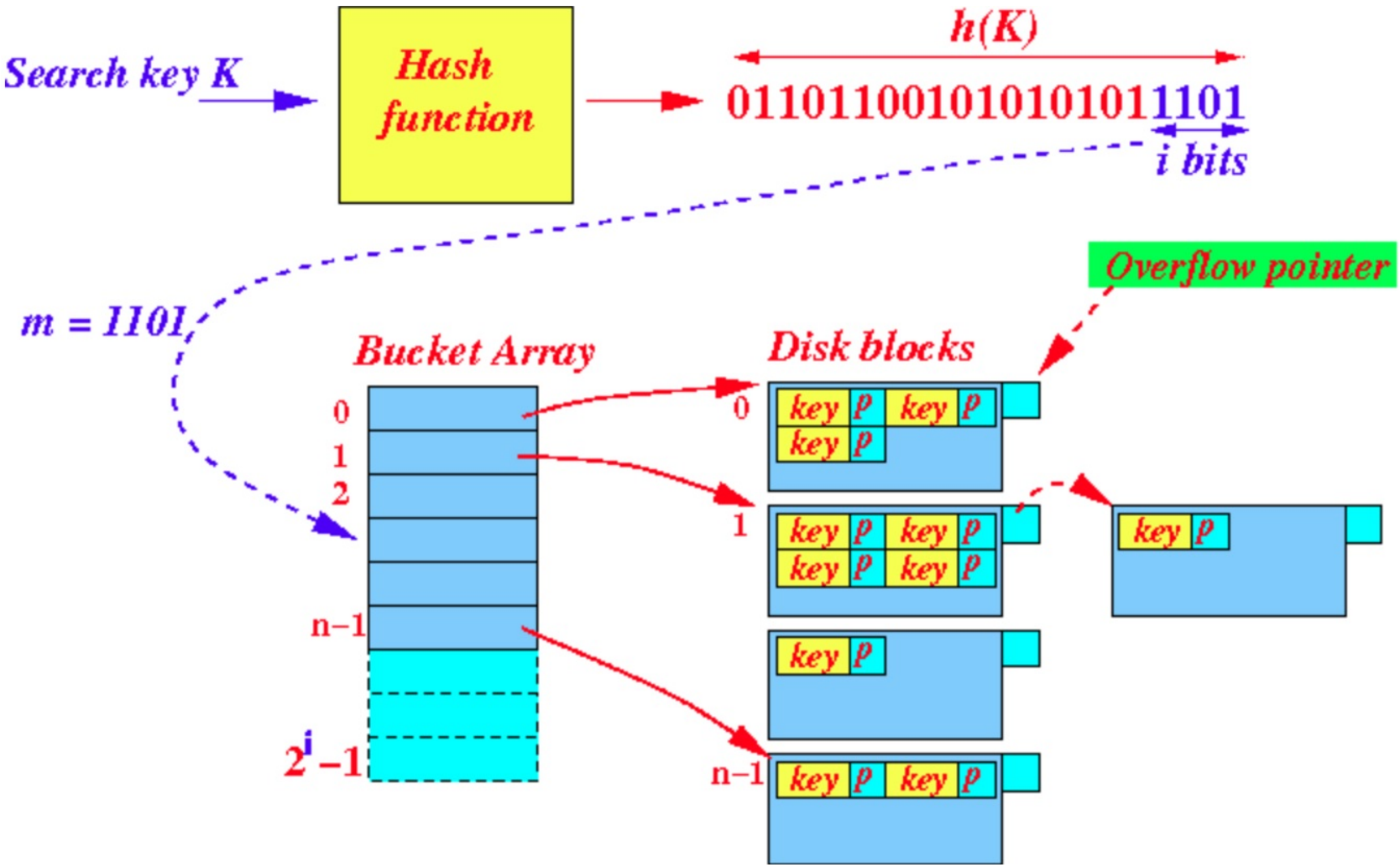
Linear Hashing technique

- Hash function used in Linear Hashing



Linear Hashing technique

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

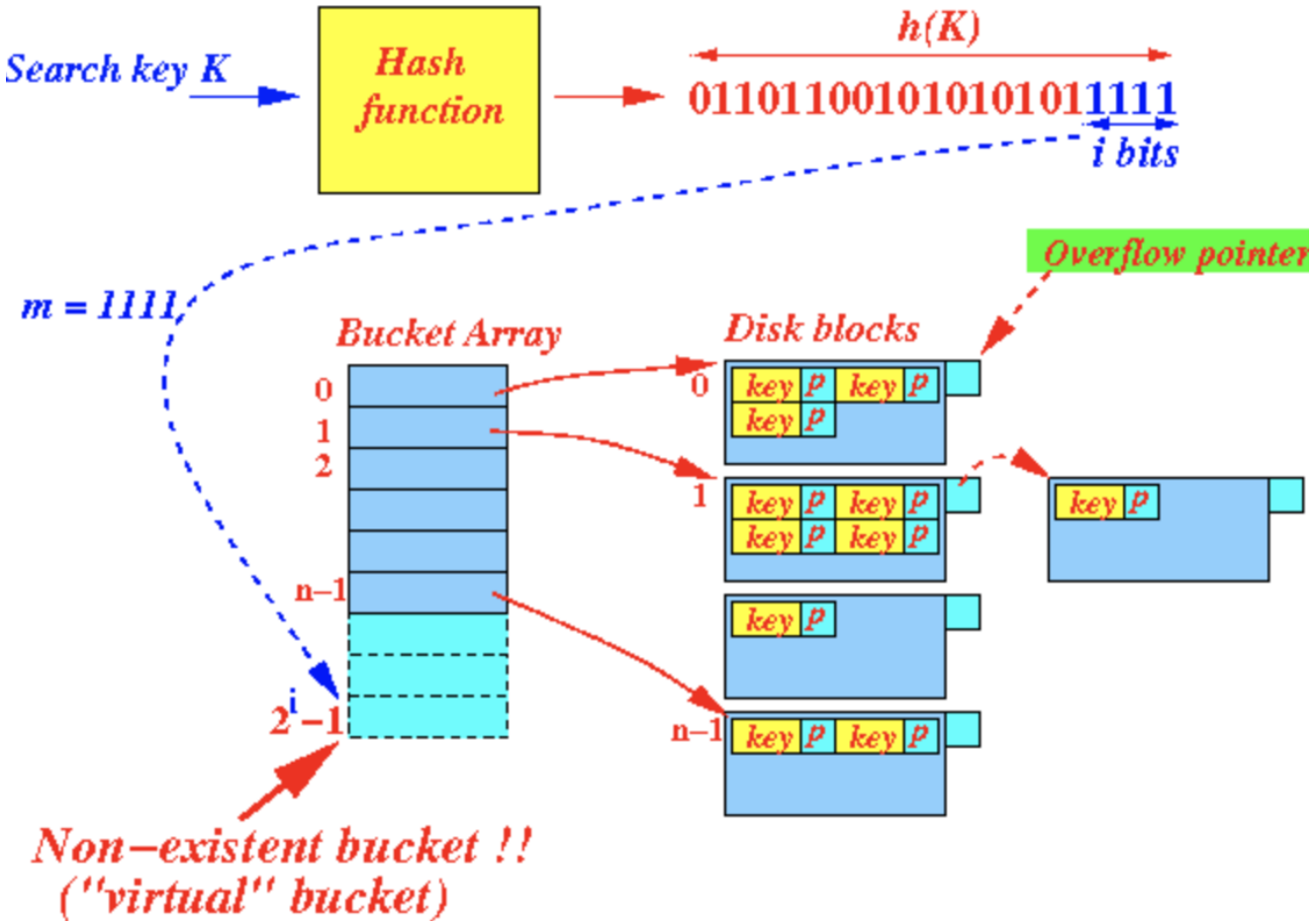


Note

- 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`:



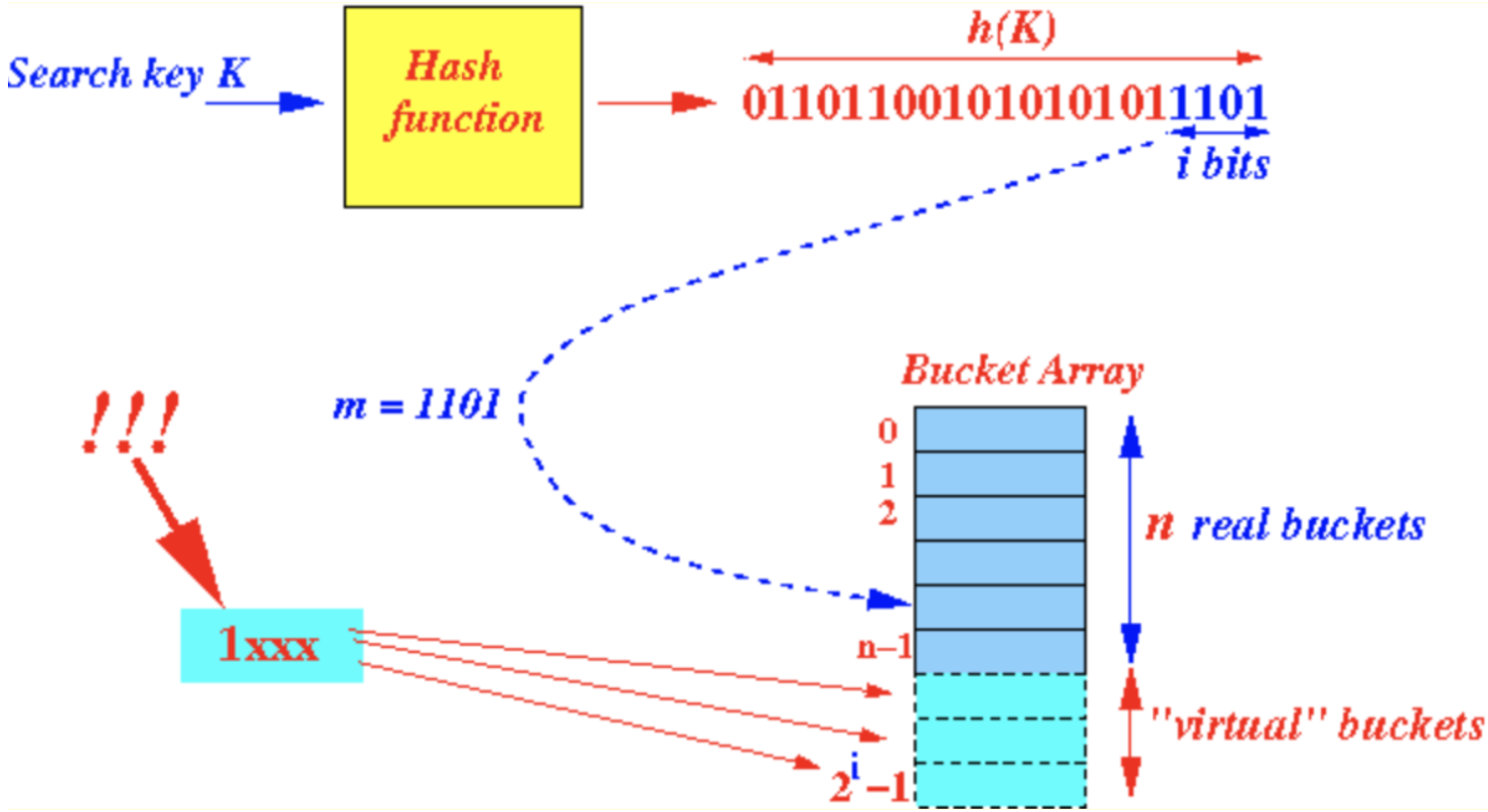
- Conclusion: We will need to map the `non-existing` buckets to an existing

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:

$$n - 1 = 1xxxx\ldots$$

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



Map the `non-existing` buckets to an existing

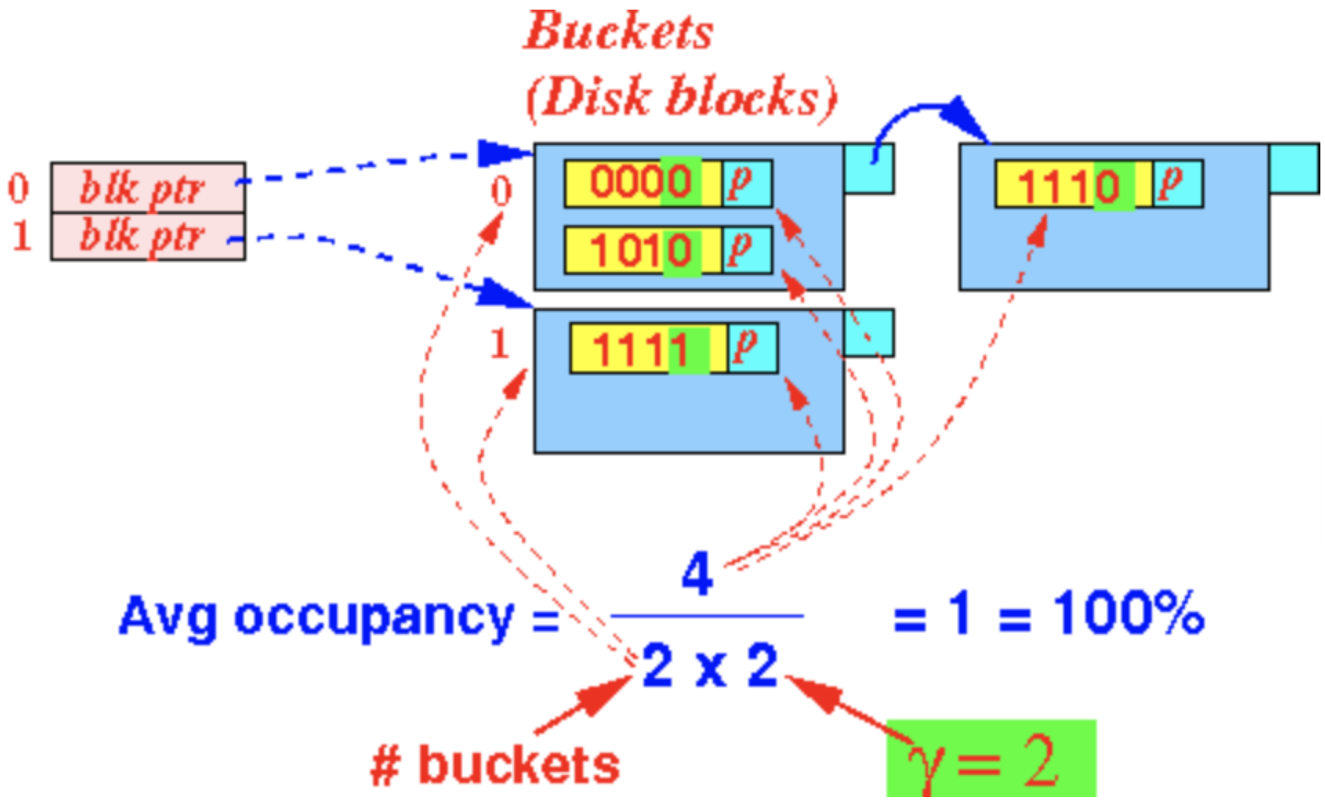
- When we change the first bit of a **`non-existent bucket index`** from 1 to 0
  - The result index identifies a **`real bucket`** (because the last bucket is $(n-1)$ starts with a 1 bit)

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 buckets` $> \tau$`) then` n`++`
- How to determine average occupancy of buckets:
  - n: number of buckets in use
  - r: total number of search keys stored in the hash buckets
  - γ: block size (# search keys that can be stored in 1 block)
  - Computing the average occupancy of buckets:
    - 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

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 $a_1 a_2 \ldots a_i$ into `binary`
  - `If` $m < n$, then `bucket` m exists – insert $\langle k, recordPtr(k) \rangle$ into that `bucket`. If necessary, use an `overflow block`
  - `If` $m \geq n$, then `bucket` m does not (yet) exist, so insert $\langle k, recordPtr(k) \rangle$ into `bucket` whose index corresponds to $0 a_2 \ldots a_i$. If necessary, use an `overflow block`

Inserting into Linear Hash Tables

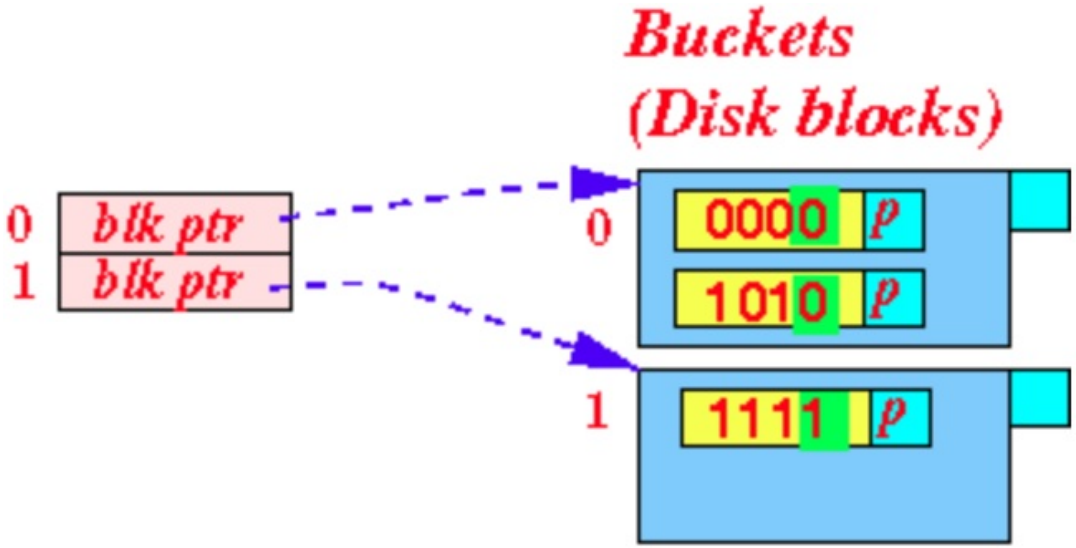
- Check if we need to adjust n
  - Compare `average occupancy` to `threshold`
  - If exceeds `threshold` then add a new `bucket` and rearrange records
    - We need to move some search keys into this new bucket
  - If number of buckets exceeds 2^i, then increment i by `1`

Inserting into Linear Hash Tables: Example

- Parameters:
  - `Max # search keys in 1 block (`γ`) = 2`
  - `Threshold avg occupancy (`τ`) = 0.85`

Example Linear Hashing

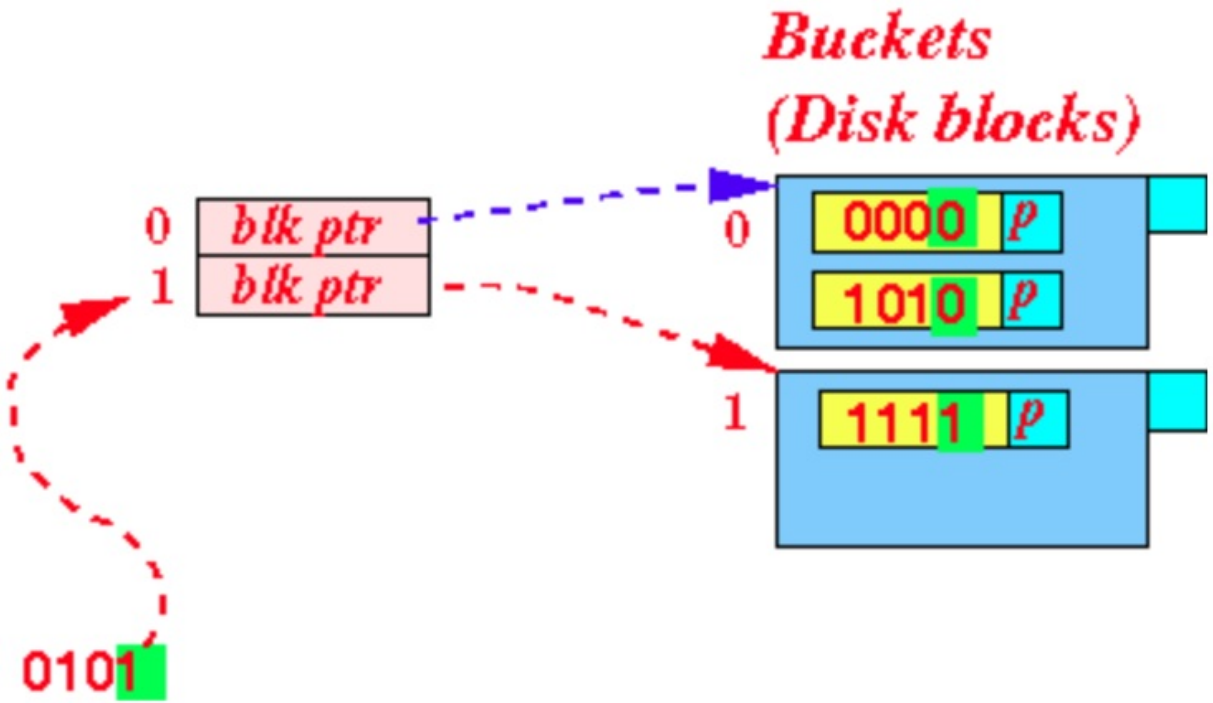
- Initial State: *n=2, i=1, r=3*



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

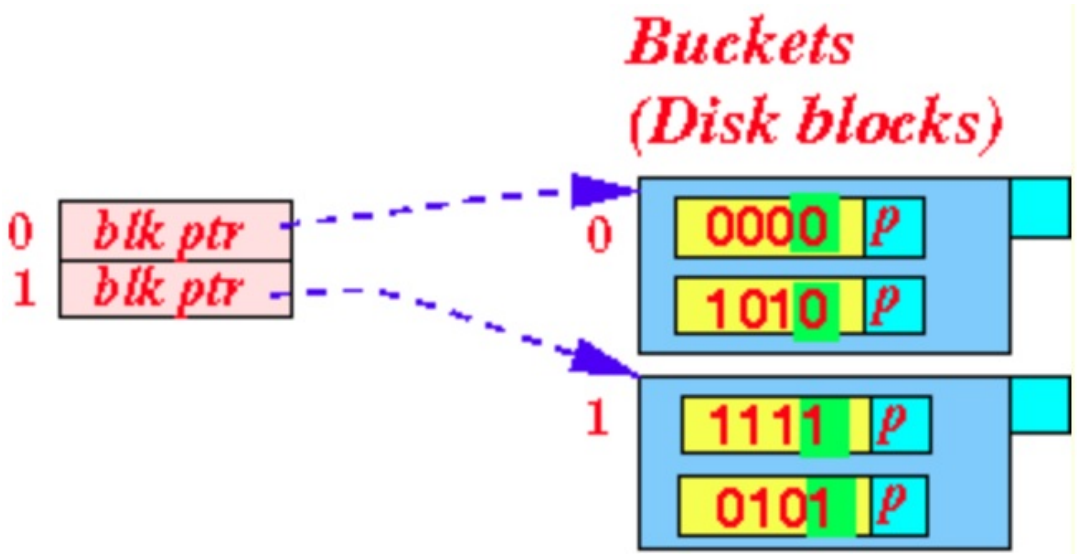
Insert search key *k* such that *h(k) = 0101*

- Insert 0101
- *n=2, i=1, r=3*



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

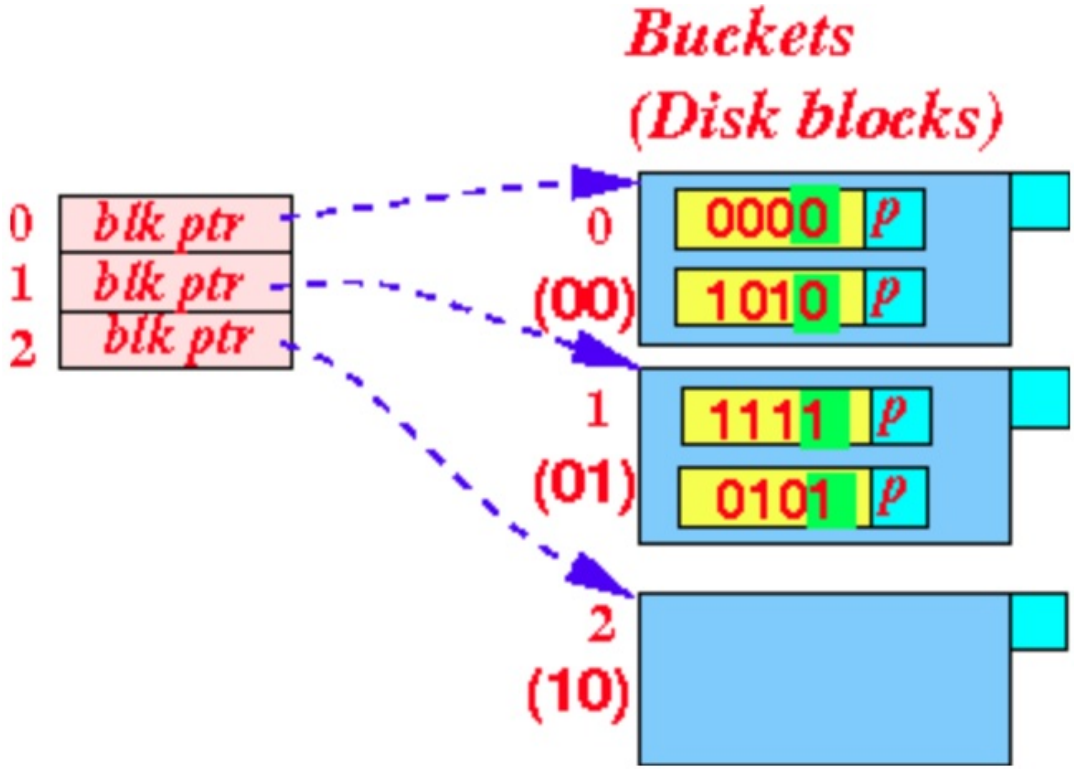
- *n=2, i=1, r=4*



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

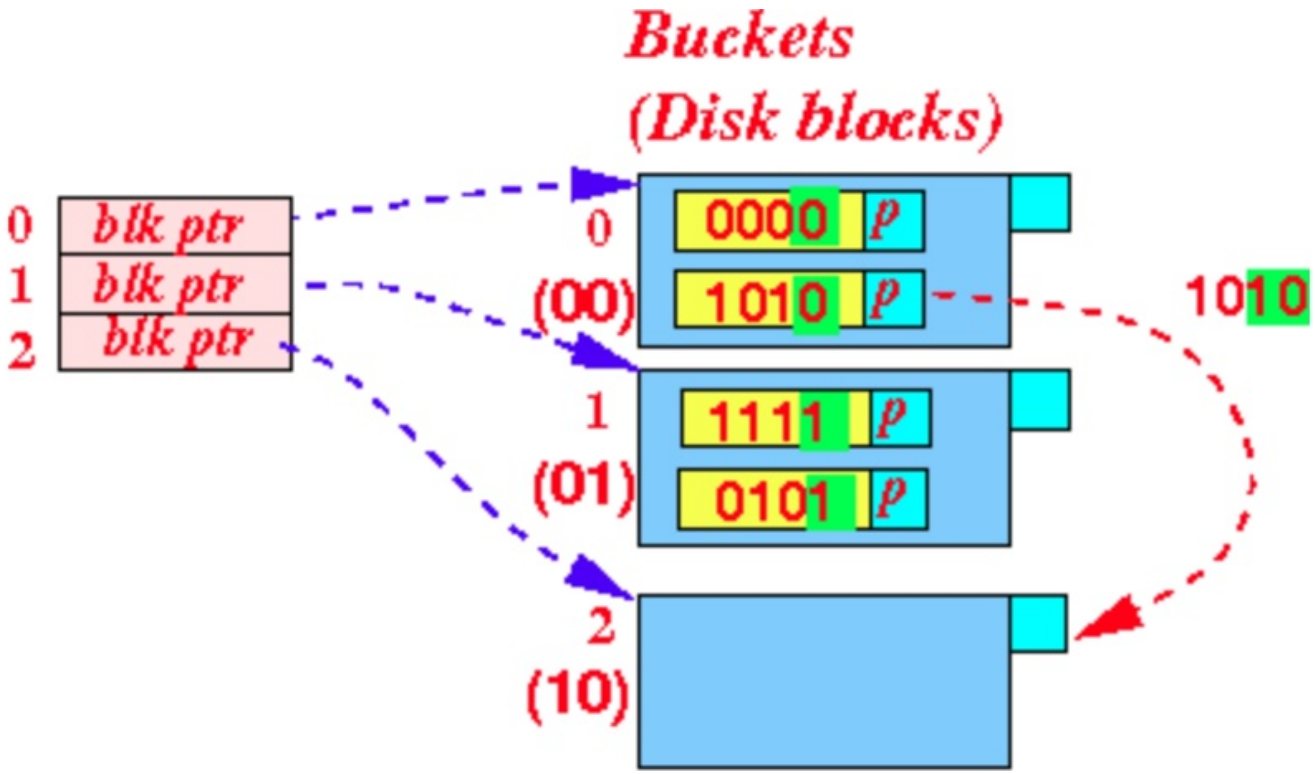
Insert search key k such that $h(k) = 0101$: result

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



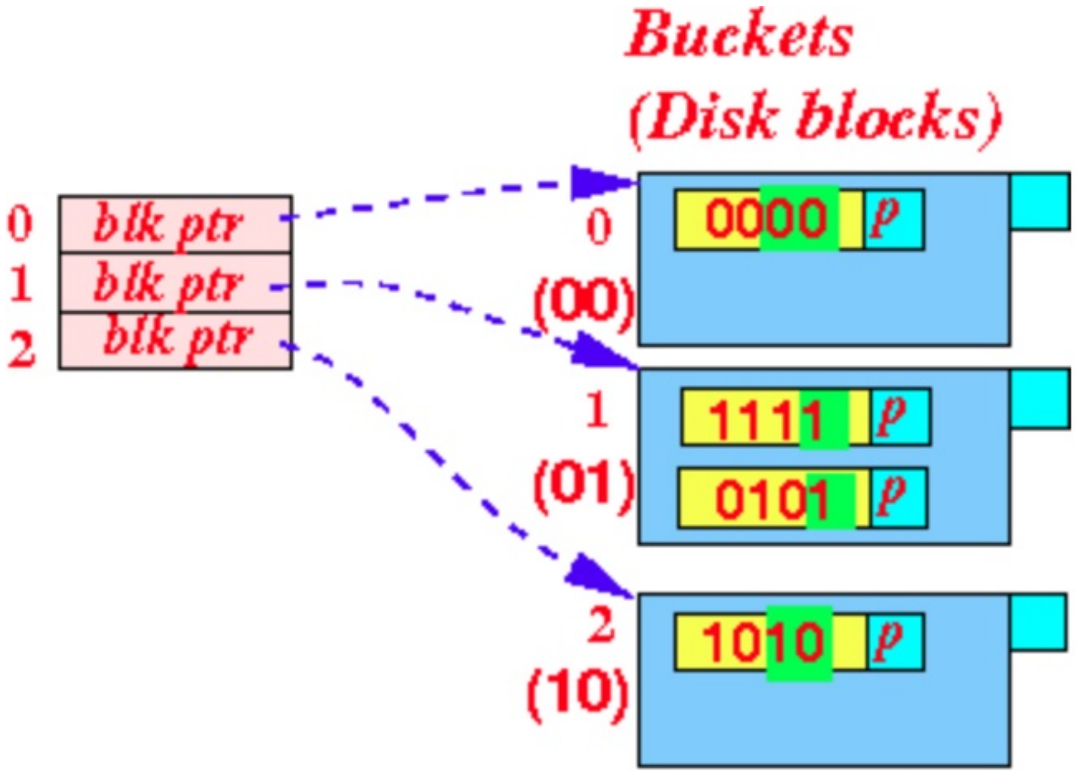
Insert search key k such that $h(k) = 0101$: result

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



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

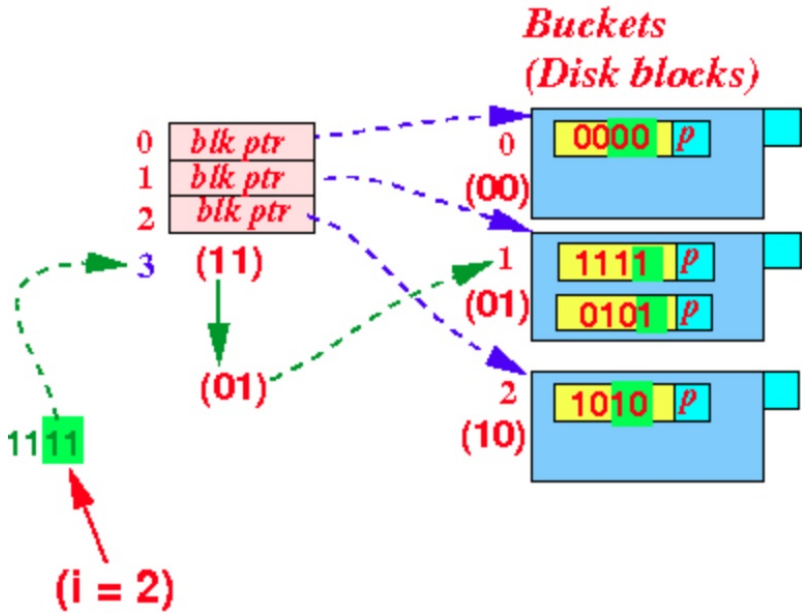
- *n=3, i=2, r=4*
- Transfer search keys from `bucket 00` to the newly created `bucket 10`



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

Notice that

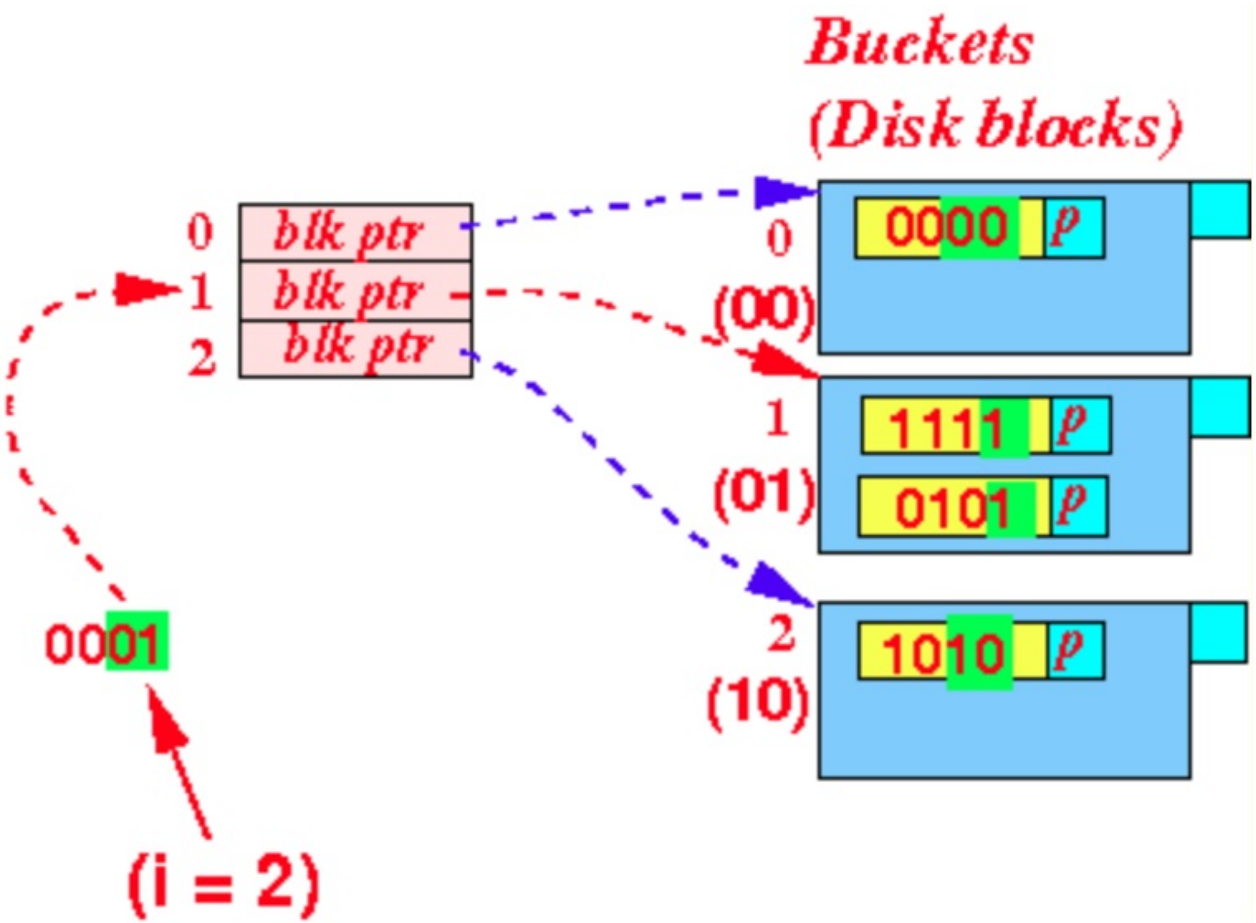
- $n = 3$ (changed)
- $i = 2$ (changed)
- We can `find` 1111 as follows



- Explanation
  - 1111 will lead to the bucket 11 (using the *last i (=2) bits*) which is a `non-existent` bucket
  - The search algorithm will use the `real bucket` 01 instead.

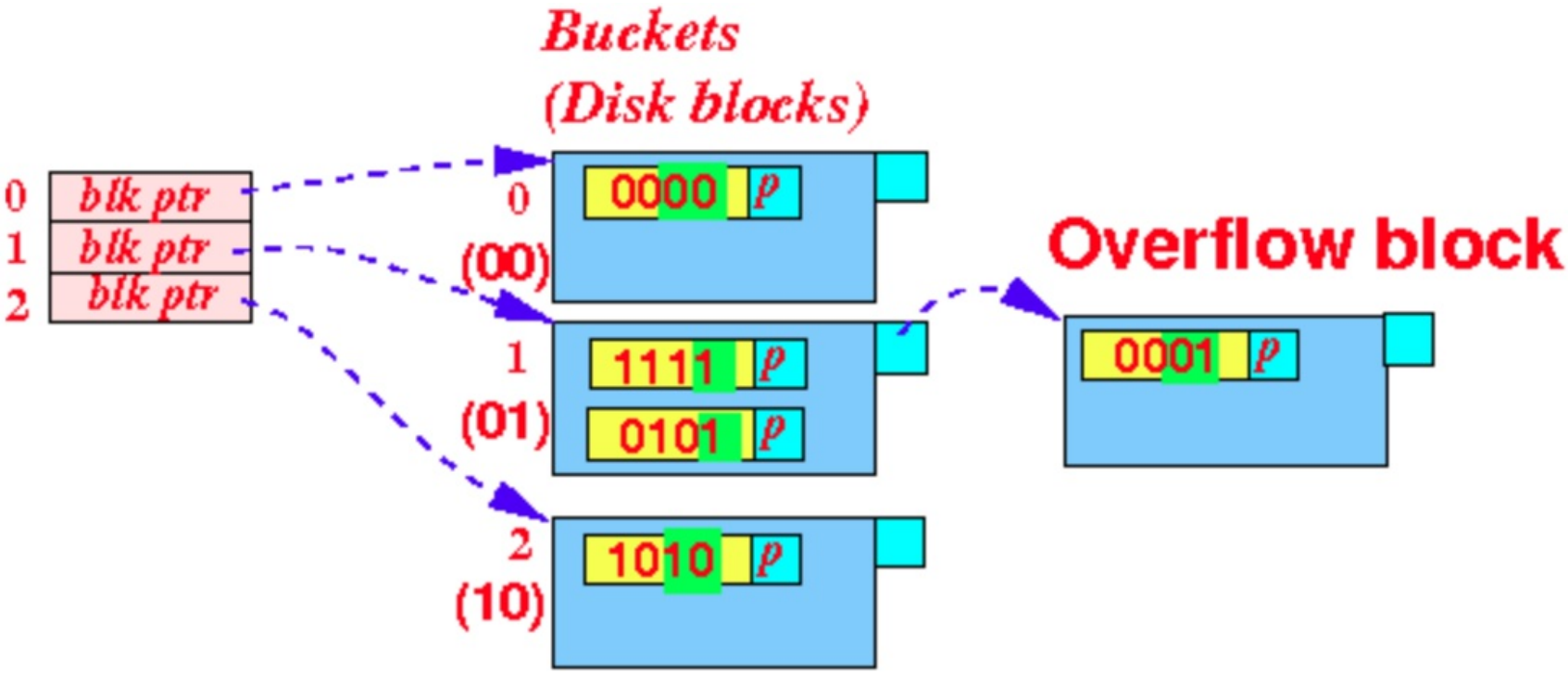
Insert search key k such that $h(k) = 0001$

- Insert 0001
- n=3, i=2, r=4



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

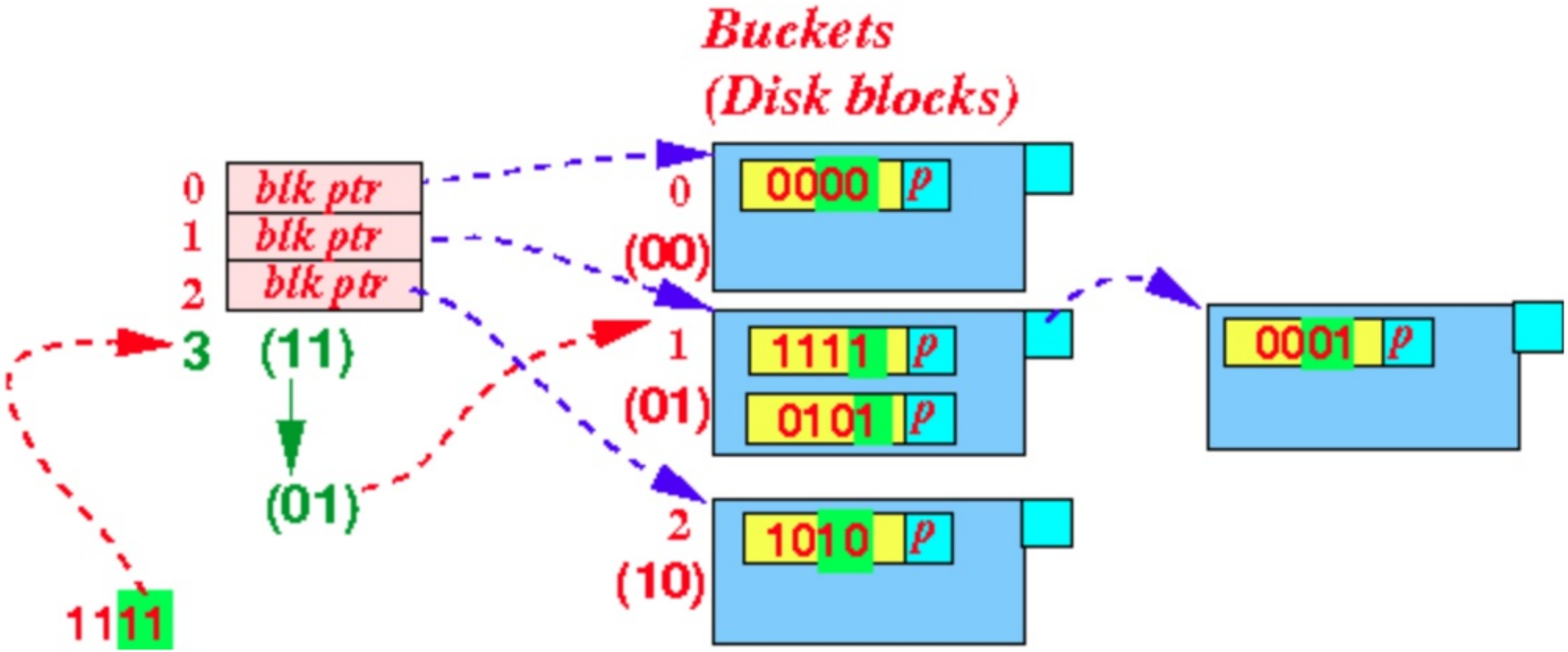
- *n=3, i=2, r=5*



- 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 to add another bucket`

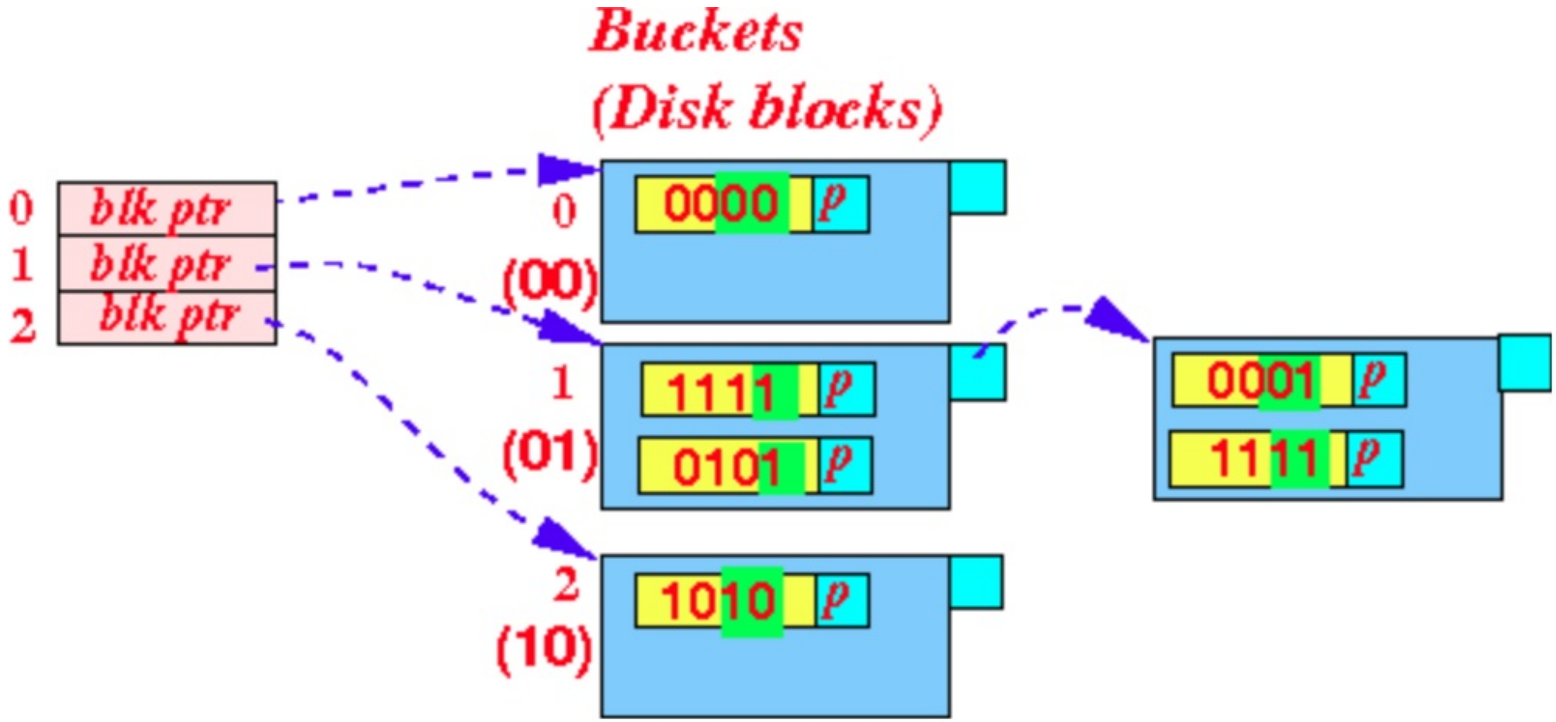
Insert search key k such that $h(k) = 1111$

- Insert 11111
- $n=3$, $i=2$, $r=5$



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

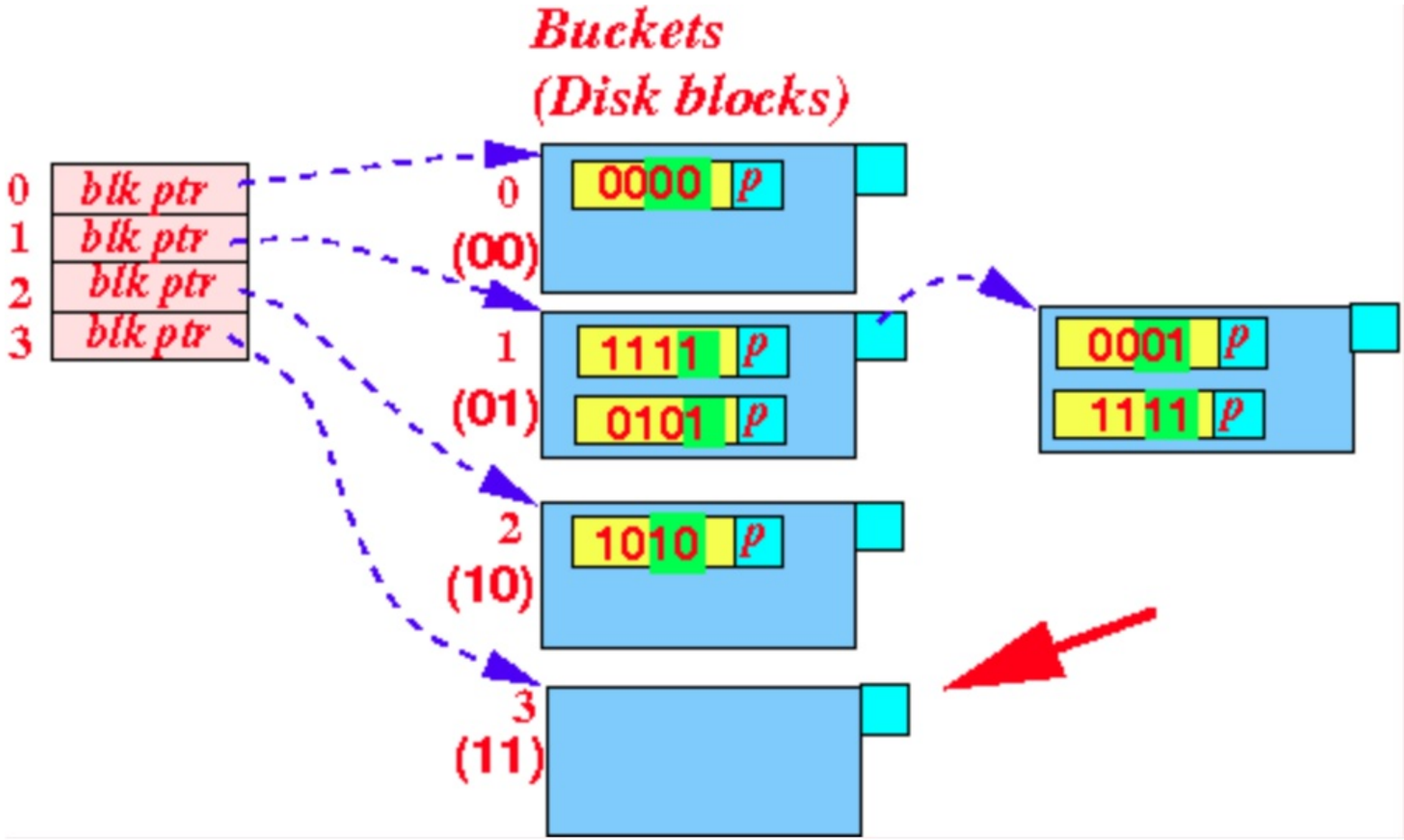
- n=3, i=2, r=6



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

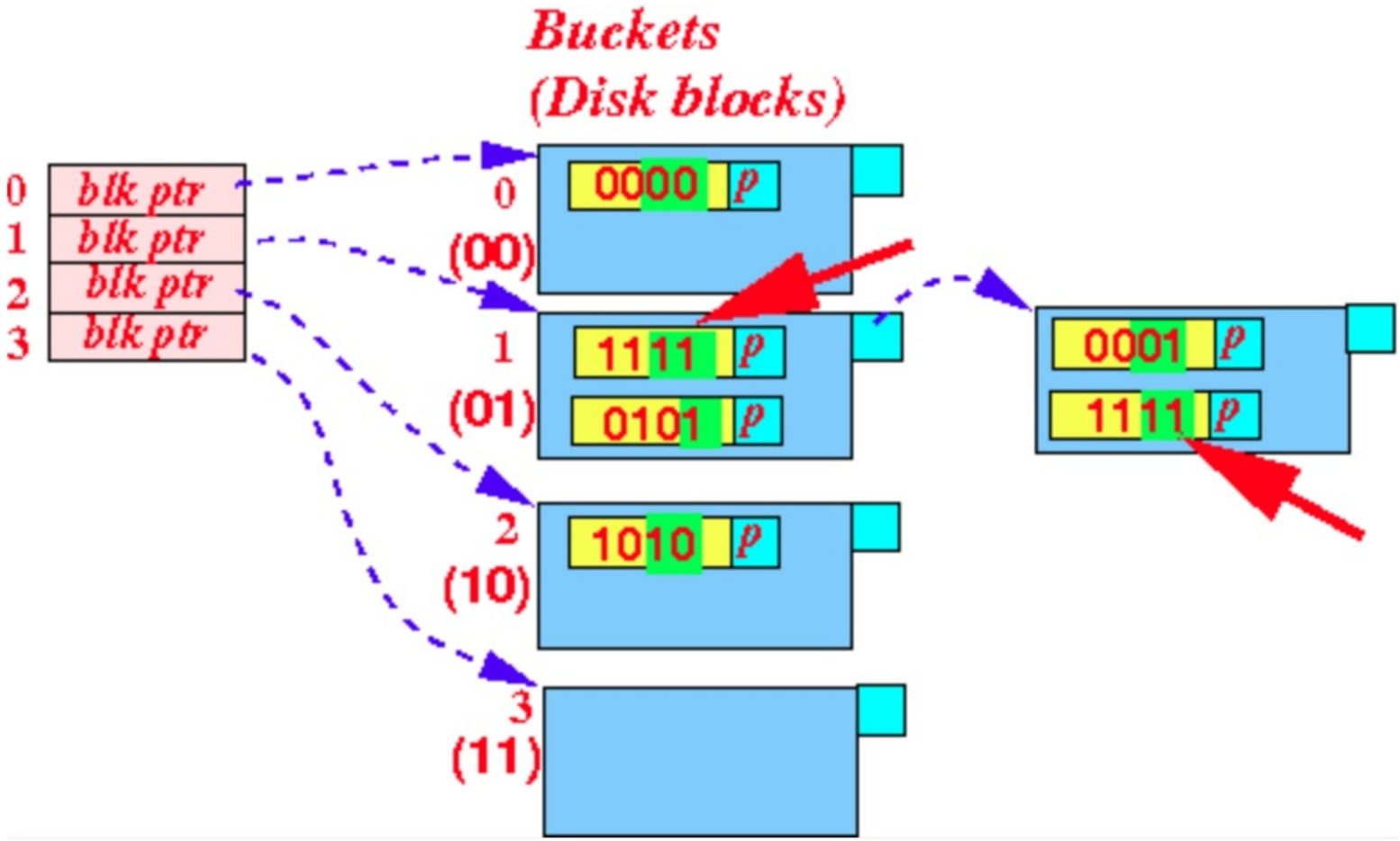
Insert search key k such that $h(k) = 1111$: Result

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



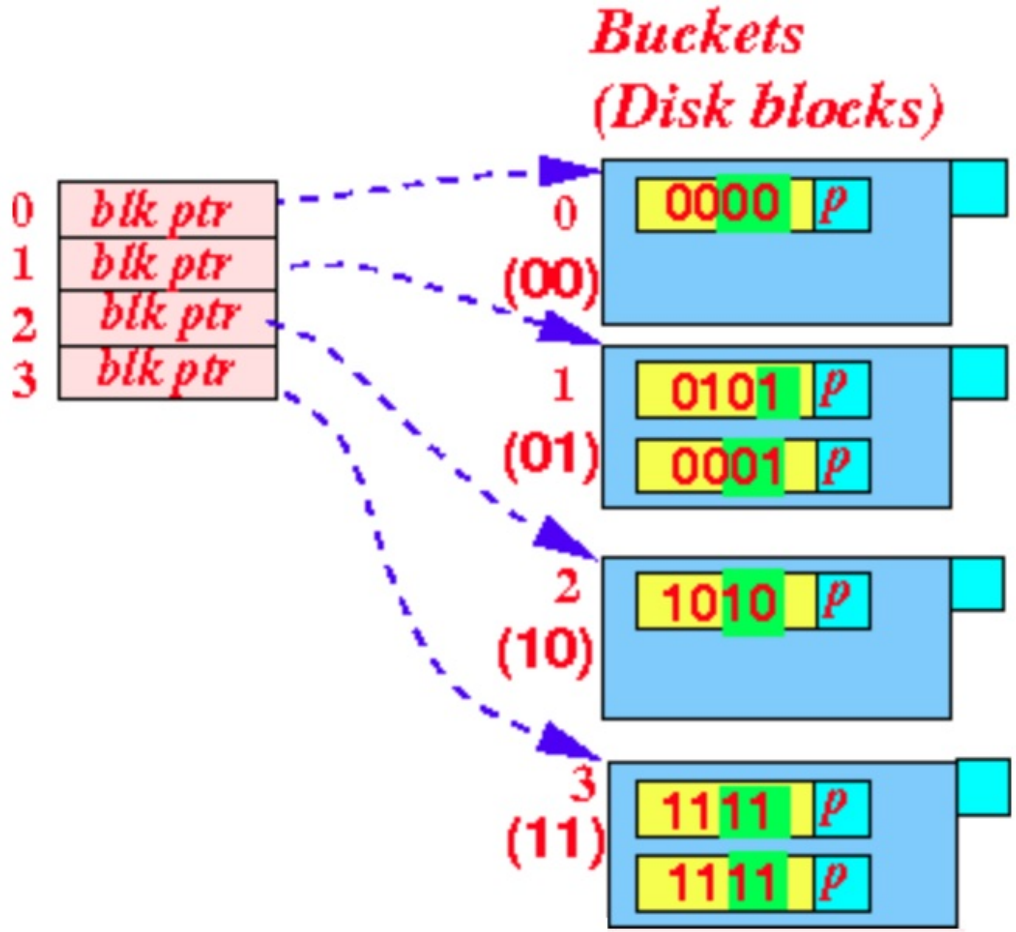
Insert search key k such that $h(k) = 1111$: Result

- 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

- ☺ manage growing files
- ☺ no indirection like Extendible hashing

- ☹ can still have overflow chains
- ☹ some work moving elements from "re-directed" bucket when creating a new

Conclusion

- Fast data structures that support `O(1)` look-ups that are used all throughout the DBMS internals.
  - Trade-off between speed and flexibility.
- Hash tables are usually not what you want to use for a table index

Comparing Index Approaches

- Hashing good for probes given key. (`Point Queries`)

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

- Order Preserving Indexes like `B+-trees` good for Range Searches. (`Range Queries`)

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

Indexes in SQL

- Syntax

```
CREATE INDEX name ON relation_name (attribute)
```

```
CREATE UNIQUE INDEX name
          ON relation_name (attribute)
-- defines a candidate key
```

```
DROP INDEX name
```

- Note: cannot specify
  - type of index, e.g., B-tree, hashing, etc.
  - parameters such as load factor, hash size, etc.
  - at least in standard SQL
  - Vendor specific extensions allow that

Indexes in SQL: Postgres Example

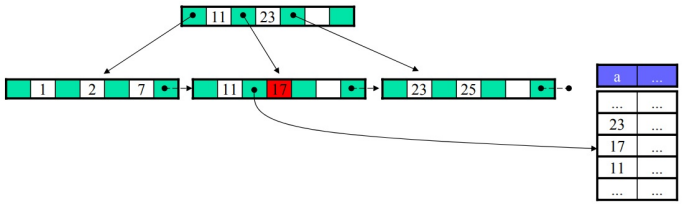
```
-- Create an index named "my_index" on the "your_table"
    table
-- using the B-tree index type & specific storage parameters
CREATE INDEX my_index
ON your_table
USING btree  -- Specify the index type (e.g., B-tree)
WITH (fillfactor = 70, pages = 1000);  -- Set storage
    parameters
```

Using Indexes in Queries

- Indexes are used to quickly find a record
- Some queries may even be solved without reading the records from disk,
  e.g., find the number of elements
  - count the number of pointers in the index (dense)
- Index structures discussed so far are one dimensional
  - one single search key
  - good for queries like

```
SELECT ... FROM R WHERE a = 17
```

e.g.: using a B+-tree index on a



  - what about

```
SELECT ... FROM R WHERE a = 17 AND b < 5
```

Using Indexes in Queries

- Strategy 1:

```
SELECT ... FROM R WHERE a = 17 AND b < 5
```

  - use one index, say on `a`
  - find and *retrieve* all records where `a=17` using the index
  - search these records for `b` values less than `5`

☺ simple and easy approach

☹ may read a lot of records not needed from disk

Using Indexes in Queries

- Strategy 2:

```
SELECT ... FROM R WHERE a = 17 AND b < 5
```

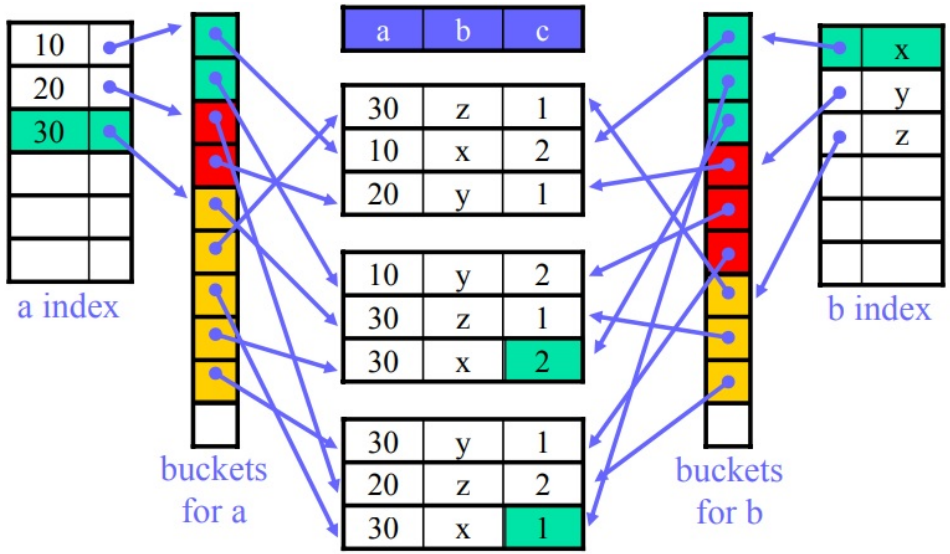
  - use two dense indexes on `a` and `b`
  - find all pointers in the first index where `a=17`
  - find all pointers in the second index where `b < 5`
  - manipulate pointers - compare (intersect) pointers and retrieve the records where the pointers match

☺ may reduce data block accesses compared to strategy 1
☹ search two indexes (or more if more conditions)
☹ cannot sort records on two attributes (for two indexes)

Using Indexes in Queries

- Example - Strategy 2, using pointer buckets:

```
SELECT c FROM R WHERE a=30 AND b='x'
```



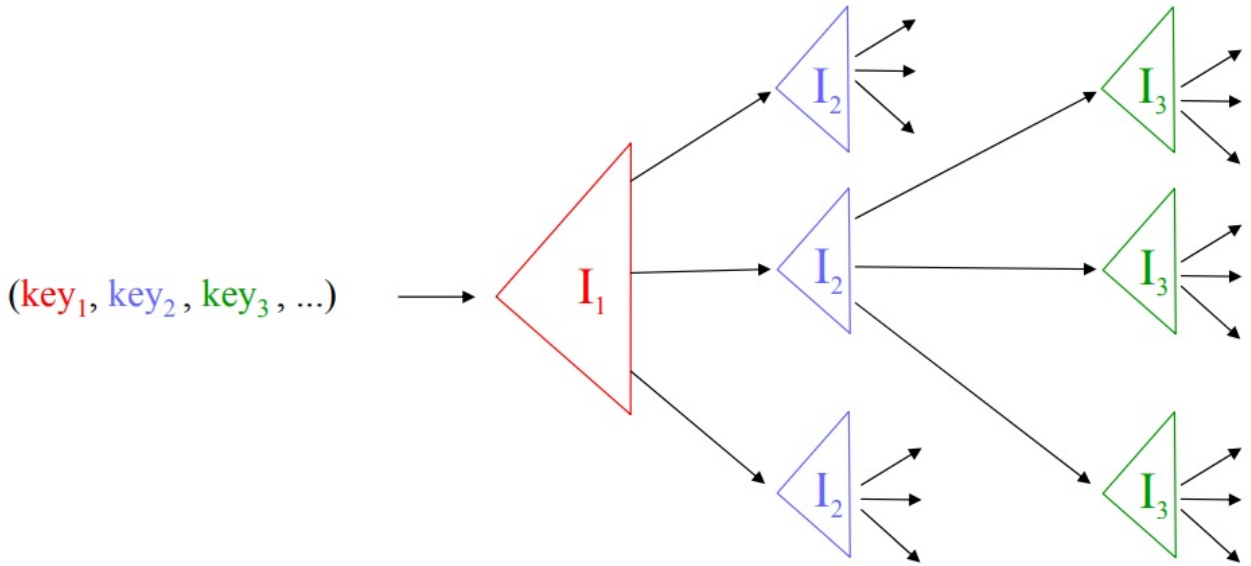
- find records where `a` is `30` using index on `a`
- find records where `b` is '`x`' using index on `b`
- have two set of record-pointers
- compare (intersect) pointers and retrieve the records requested
- select specified attributes

Using Indexes in Queries

- *One dimensional* indexes can be uses many places, but in case of operations with several search conditions, it may be
  - many indexes to search
  - hard to keep efficient record order on disk for several indexes
  - ...

Using Indexes in Queries

- Strategy 3: *Multiple Key Index*
- One simple tree-like approach:

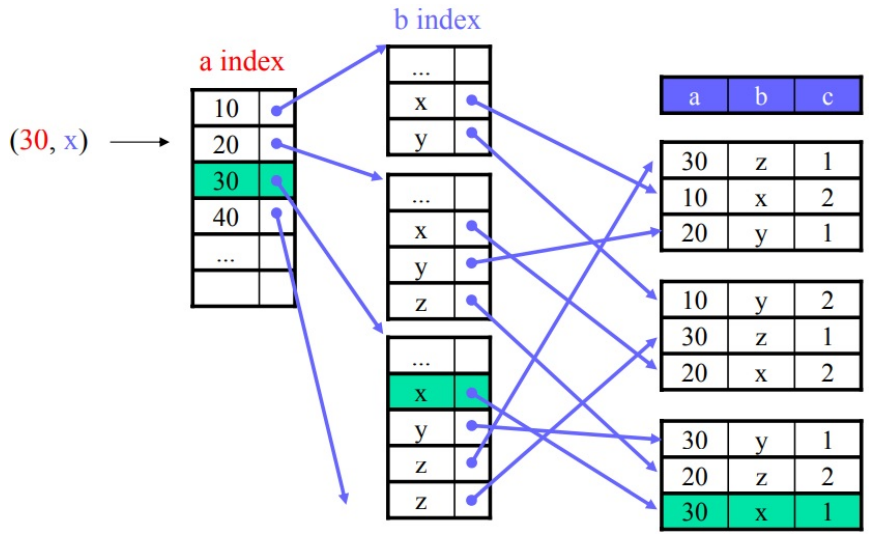


  - Each level as an index for one of the attributes.
  - Works well for partial matches if the match includes the first attributes.

Using Indexes in Queries

- Example - Strategy 3, *Multiple Key*, dense index

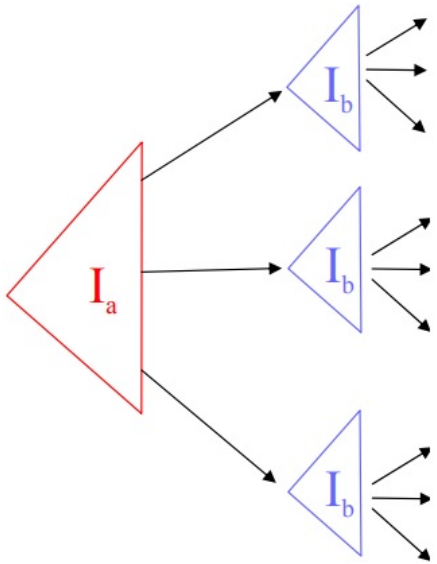
```
SELECT c FROM R WHERE a=30 AND b='x'
```



- `search key = (30, x)`
- read *`a-dimension`*
- search for `30`, find corresponding *`b-dimension`* index
- search for `x`, read corresponding disk block and get record
- select requested attributes

Using Indexes in Queries: Multiple Key index

- For which queries is this index good?



☺ find records where `a = 10 AND b = 'x'`
☺ find records where `a = 10 AND b` $\geq$ `'x'`
☺ find records where `a = 10`
☹ find records where `b = 'x'`
? find records where `a` $\geq$ `10 AND b = 'x'`

- may search several indexes in next dimension
- better if dimensions changed order

Next

Even more index structures