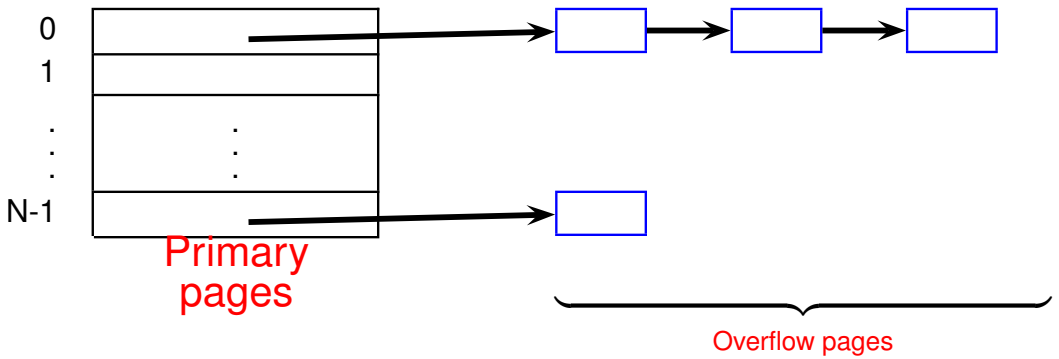
CS3223 Lecture 3
Hash-based Indexing

Hashed-Based Indexing

- Used for equality queries but not for range queries
- **Hashing Techniques**:
  - Static hashing
  - Dynamic hashing
    - ★ Linear hashing
    - ★ Extendible hashing
    - ★ etc

Static Hashing

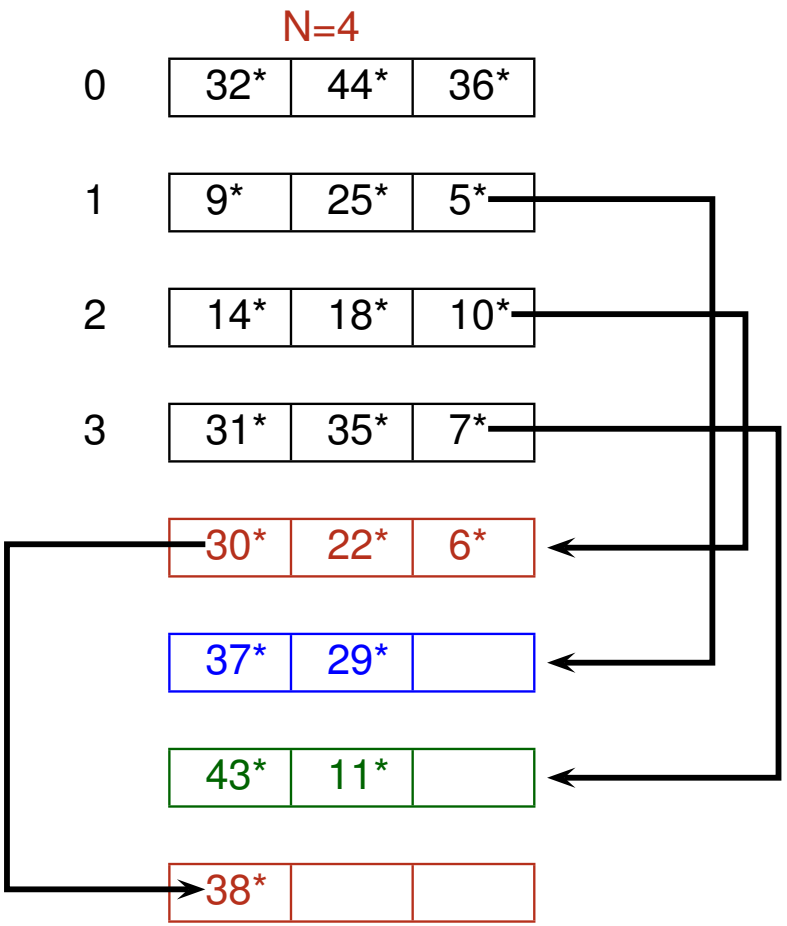
- Data is stored in N buckets $B_0, B_1, \cdots, B_{N-1}$
  - N is fixed at creation time
- Hashing function $h(.)$ is used to identify the bucket to store a record
  - A record with search key k is inserted into bucket B_i, where $i = h(k) \ mod \ N$
  - $h(k)$ maps the search key value into a bit string
- Each bucket consists of one primary data page & a chain of zero or more overflow data pages



Static Hashing: Example

v^* denote a data entry e with $h(e.key) = v$

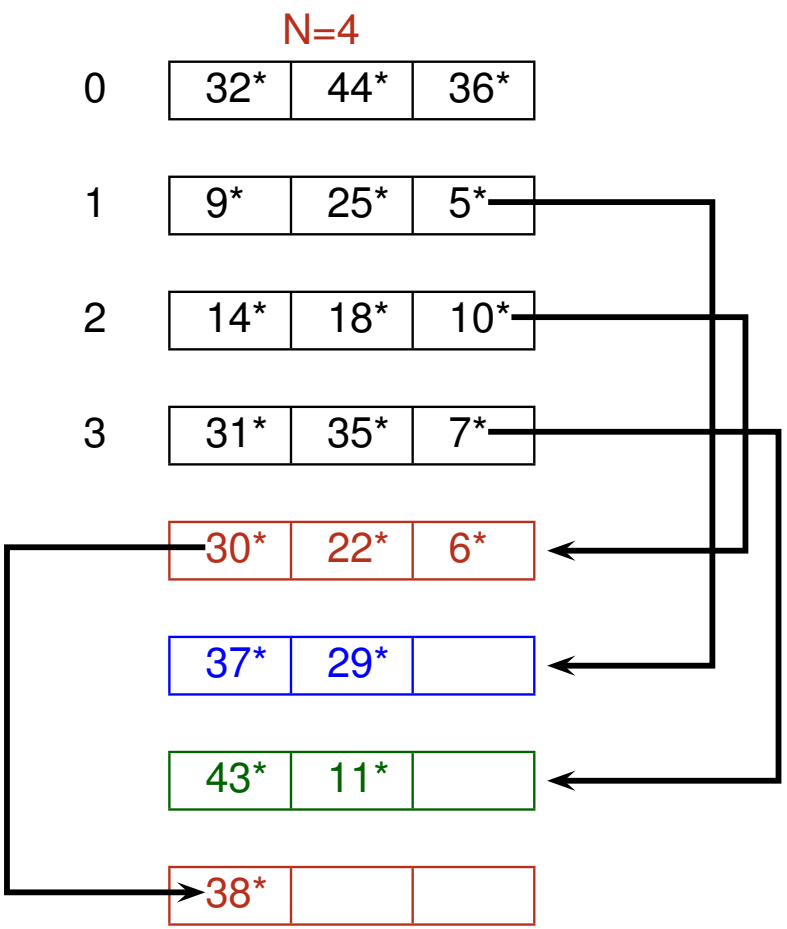
$e.key$ denote the index's search key value of e

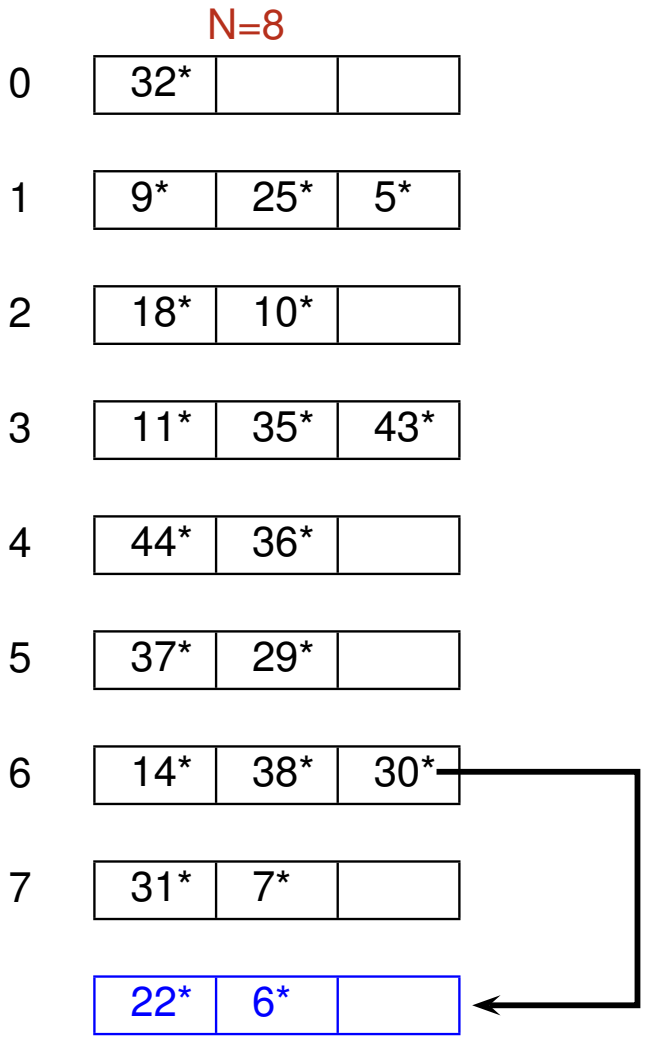


Static Hashing: Example

v^* denote a data entry e with $h(e.key) = v$

$e.key$ denote the index's search key value of e





Linear Hashing

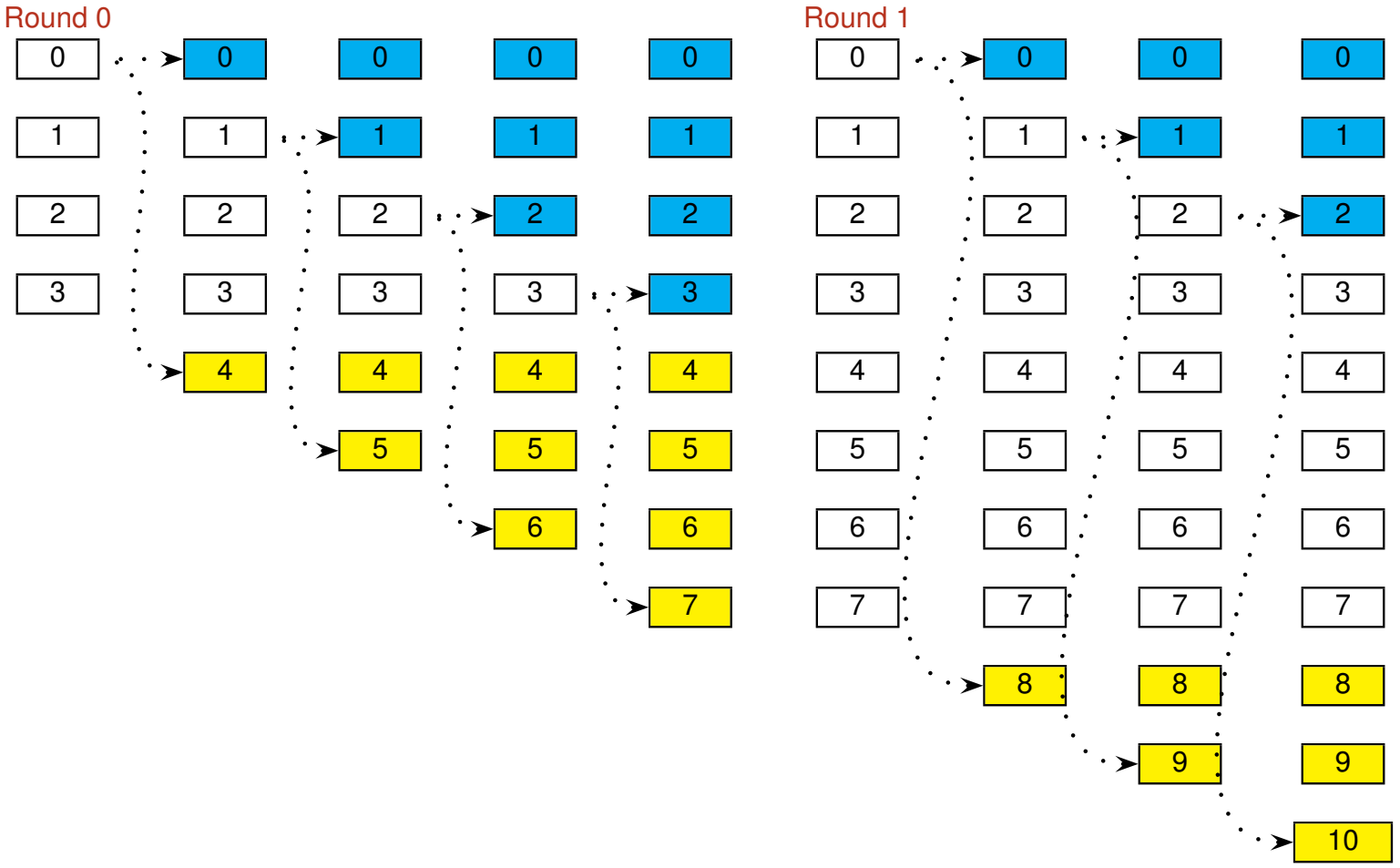
- Dynamic hashing technique
- Hash file grows linearly by systematic splitting of buckets
- Overflow pages are needed since an overflowed bucket might not be split immediately
- Each bucket consists of a primary data page & a chain of zero or more overflow pages
- An insertion into a bucket B_i overflows if all the pages in B_i (i.e., primary & overflow pages) are full

Linear Hashing (cont.)

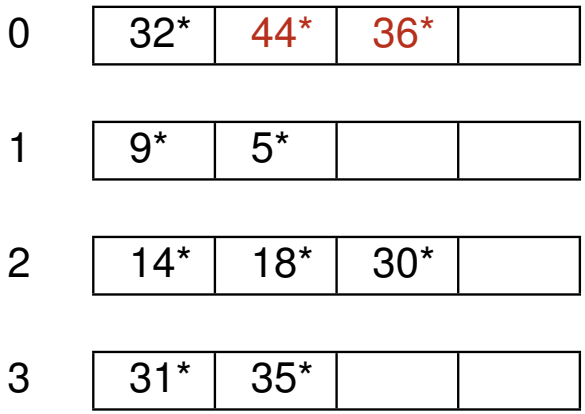
- Assume initial file size of N_0 buckets
  - Buckets $B_0, B_1, \cdots B_{N_0-1}$
- File grows linearly by splitting buckets in rounds
  - At each round, buckets are split sequentially: B_i is split before B_{i+1}
- How to split a bucket B_i?
  - Add a new bucket B_j (known as split image of B_i)
  - Redistribute entries in B_i between B_i & B_j
- File size increases by one bucket after each bucket split
- At the end of one round of splitting (i.e., every bucket at the start of the round has been split), file size is doubled
- Let N_i denote the file size at the beginning of round i (i=0,1,$\cdots$)
  - $N_i = 2^i N_0$
- At the end of round i, N_i new buckets are added: $B_{N_i}, B_{N_i+1}, \cdots, B_{2N_i-1}$
  - In round i, the split image of B_j is B_{N_i+j} for $j \in [0, N_i - 1]$

Splitting Buckets

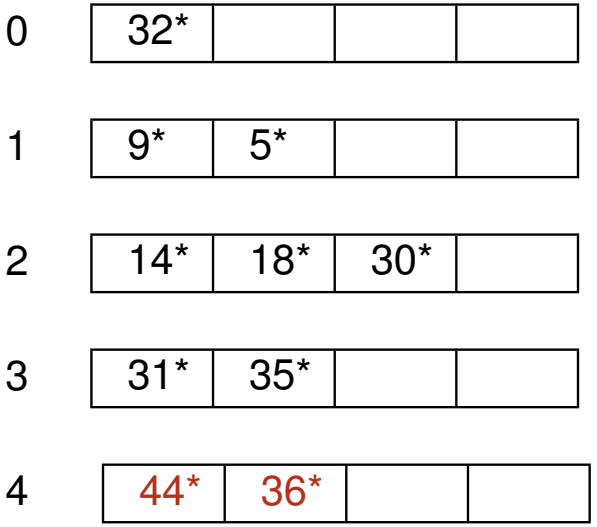
- Assume $N_0 = 4$
- In round 0, split image of bucket B_i is B_{4+i}, for $i \in [0,3]$
- In round 1, $N_1 = 8$ & split image of bucket B_i is B_{8+i}, for $i \in [0,7]$



Dynamic Hashing



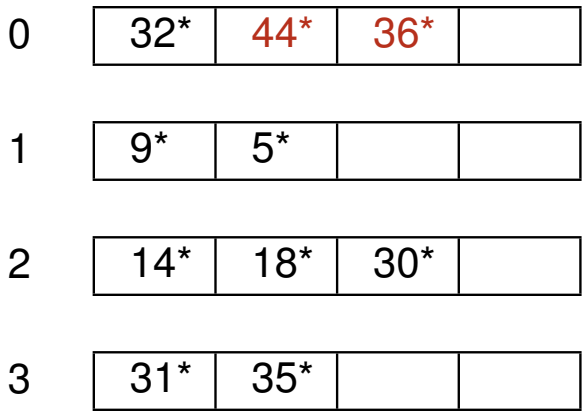
Before B_0 was split in round 0



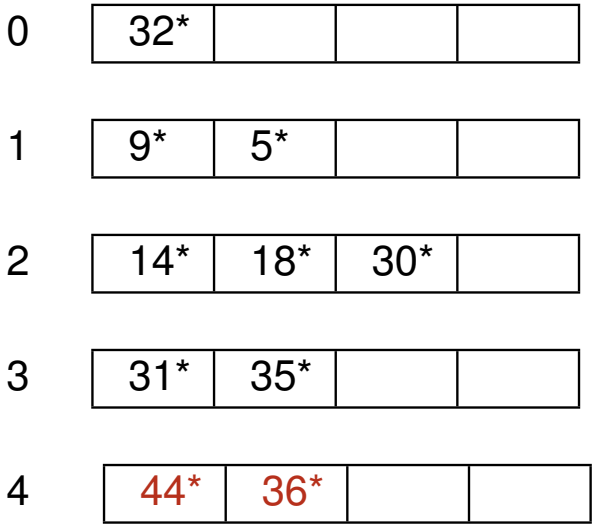
After B_0 was split in round 0

- Recall that v^* denote a data entry e with $h(e.key) = v$ and $e.key$ denote the index's search key value of e
- Each round uses two hash functions: functions h_i and h_{i+1} for round i
  - $h_i(k) = h(k)\ mod\ N_i$
  - B_x is the bucket for search key k if B_x had not been split, where $x = h_i(k)$
  - B_y is the bucket for search key k if B_x had been split, where $y = h_{i+1}(k)$
- Keep track of which bucket to be split next using variable next

Redistributing Entries



h(e.key)	
32	(100000_2)
44	(101100_2)
36	(100100_2)
9	(001001_2)
5	(000101_2)
14	(001110_2)
18	(010010_2)
30	(011110_2)
31	(011111_2)
35	(100011_2)

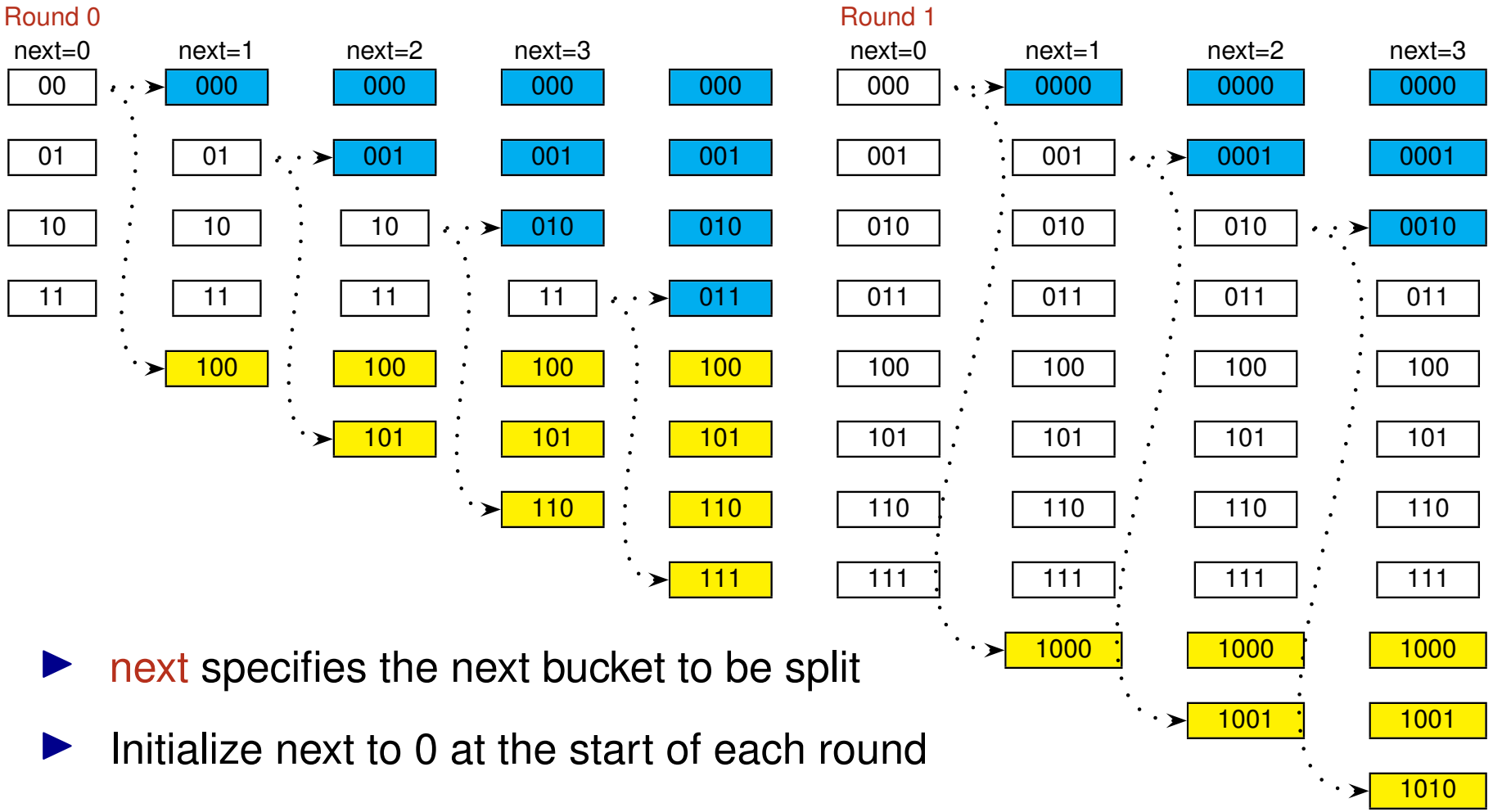


Before B_0 was split in round 0

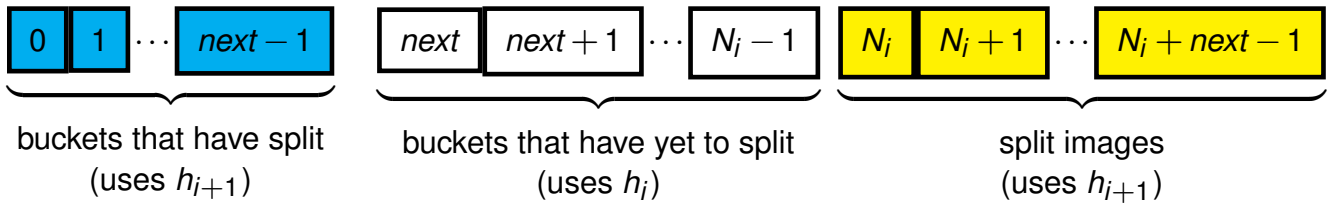
After B_0 was split in round 0

- For convenience, $N_0 = 2^m$ for $m \in \{0, 1, \cdots\}$
- $N_i = 2^i N_0 = 2^{m+i}$
- $h_i(k) = h(k)$ *mod* N_i = value of last (m+i) bits of h(k)
- Consider the splitting of bucket B_j in round i
  - Before splitting, all entries e in B_j have $h_i(e.key) = j$ (i.e., same last (m+i) bits)
  - After splitting, e remains in B_j iff the last (m+i+1) bit of $h(e.key)$ is 0

Splitting Buckets



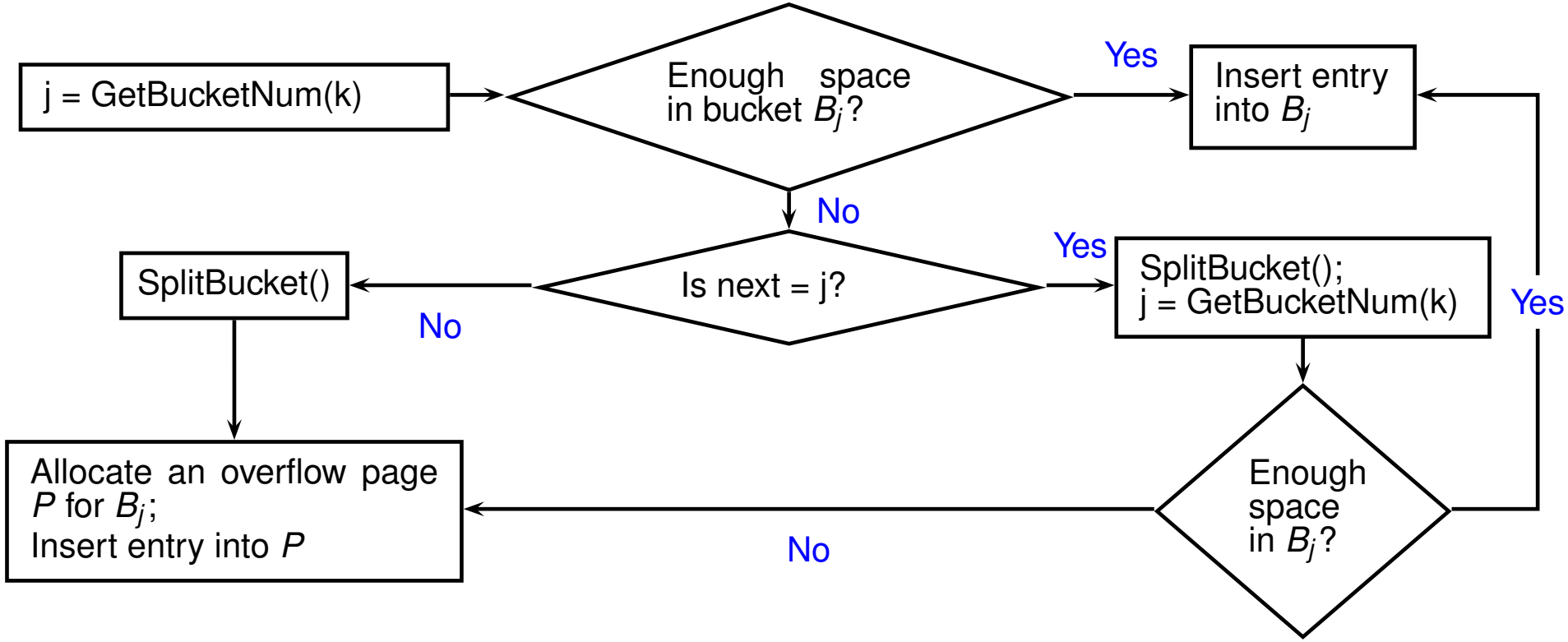
- next specifies the next bucket to be split
- Initialize next to 0 at the start of each round
- Buckets (in round i) can be classified into three regions:



When to split a bucket?

- The time to split the next bucket can be decided with various criteria:
  - Split whenever some bucket overflows
  - Split whenever space utilization of file is above some threshold
  - etc.
- Overflow pages are needed since an overflowed bucket might not be split immediately
- We shall assume that a bucket split is triggered whenever some bucket overflows
  - Bucket B_j overflows if an entry is to be inserted into B_j and all the pages in B_j (i.e., primary & overflow pages) are full
- We use **level** to denote the splitting round number

Inserting data entry with search key k



- **GetBucketNum(k)** returns bucket # where entry with search key k is located

$$GetBucketNum(k) = \begin{cases} h_{level}(k) & \text{if } h_{level}(k) \geq next, \\ h_{level+1}(k) & \text{otherwise.} \end{cases}$$

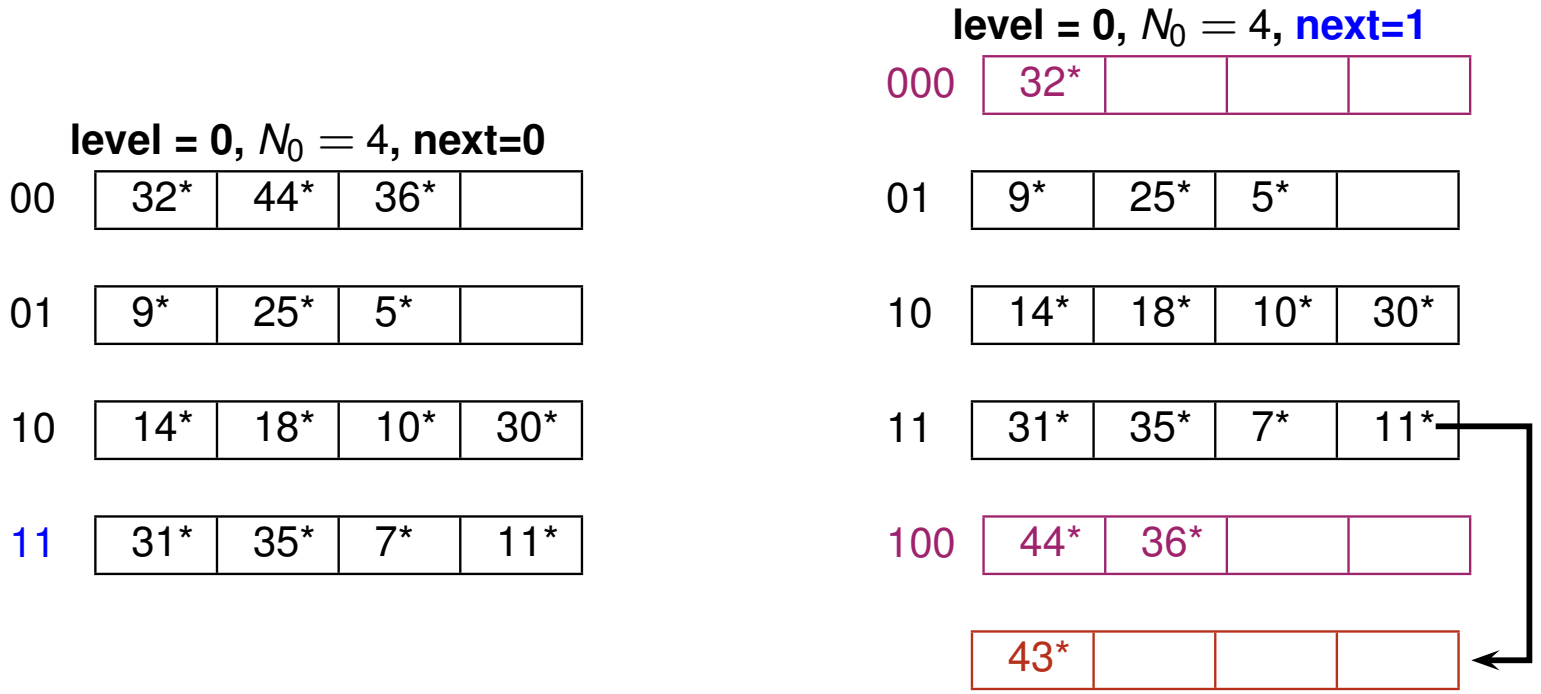
- **SplitBucket()** splits bucket B_{next}
  1. Redistribute the entries in B_{next} into $B_{next+N_{level}}$ using $h_{level+1}()$
  2. next = next + 1
  3. if ($next = N_{level}$) then { level = level + 1; next = 0 }

Linear Hashing: Inserting 43* (101011)

level = 0, $N_0 = 4$**, next=0**

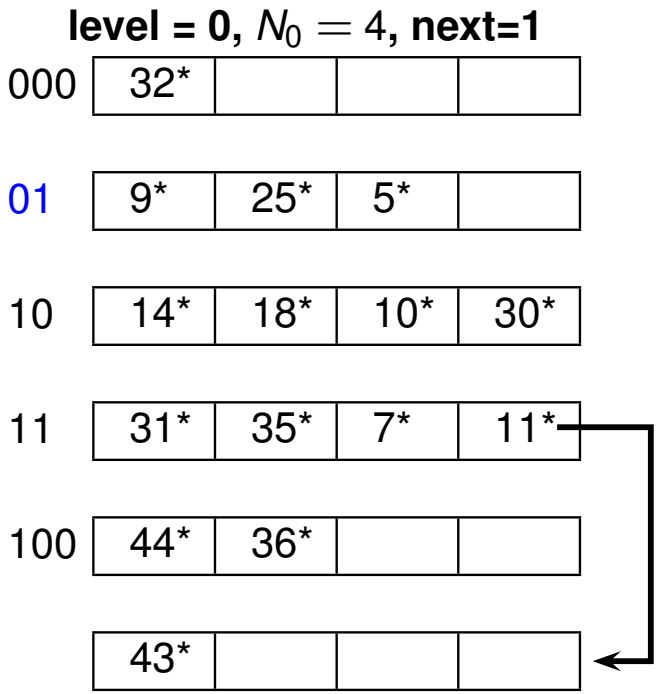
00	32*	44*	36*	
01	9*	25*	5*	
10	14*	18*	10*	30*
11	31*	35*	7*	11*

Linear Hashing: Inserting 43* (101011)

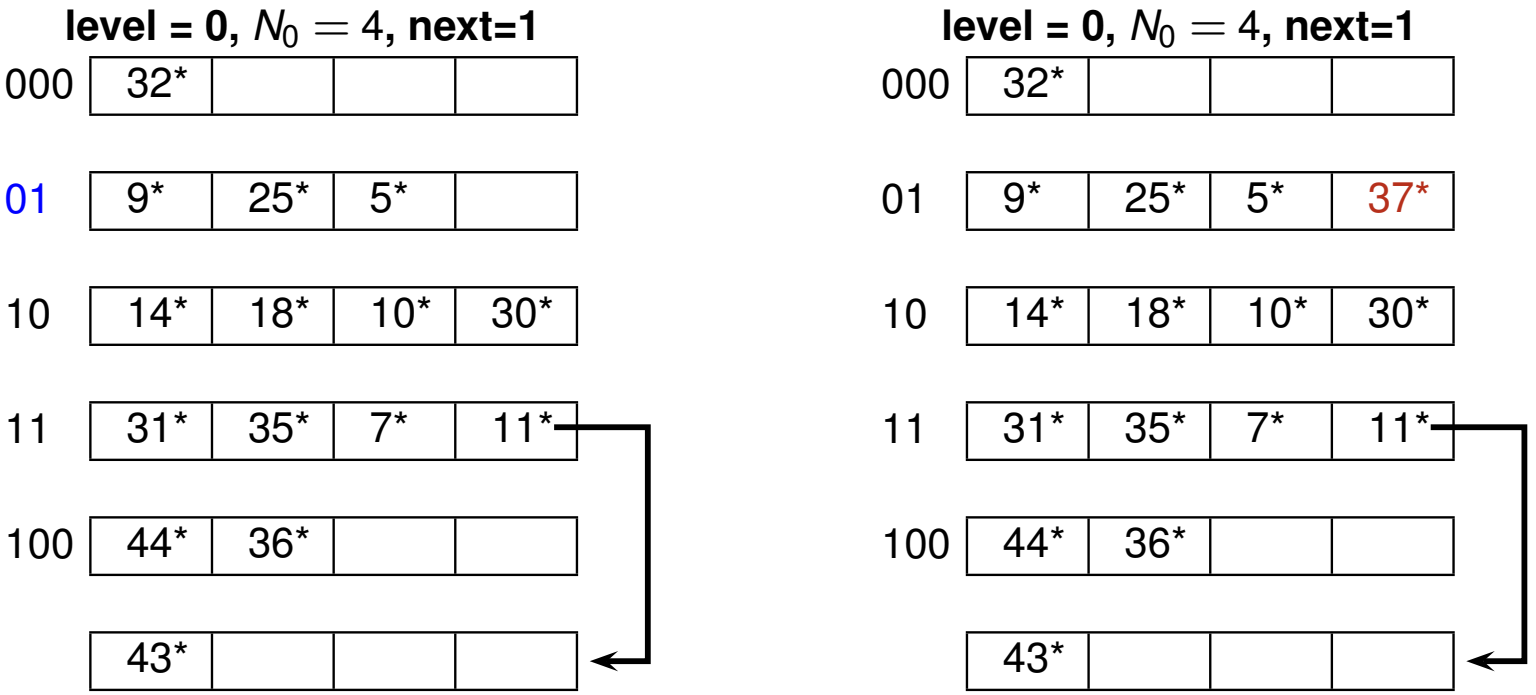


- Bucket B_{11} overflows
- Split bucket B_{00} (32 = 100000, 36 = 100100, 44 = 101100)
- Increment next to 1
- Insert 43* into overflow page

Linear Hashing: Inserting 37* (100101)

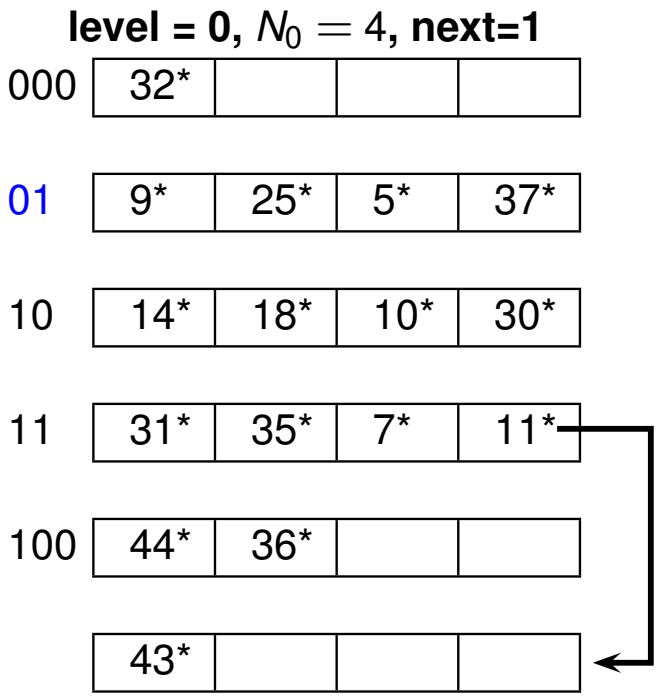


Linear Hashing: Inserting 37* (100101)

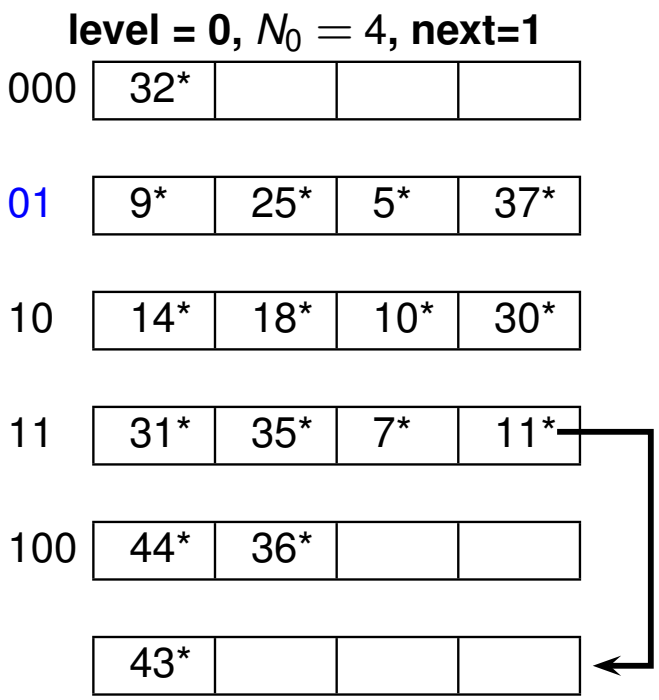


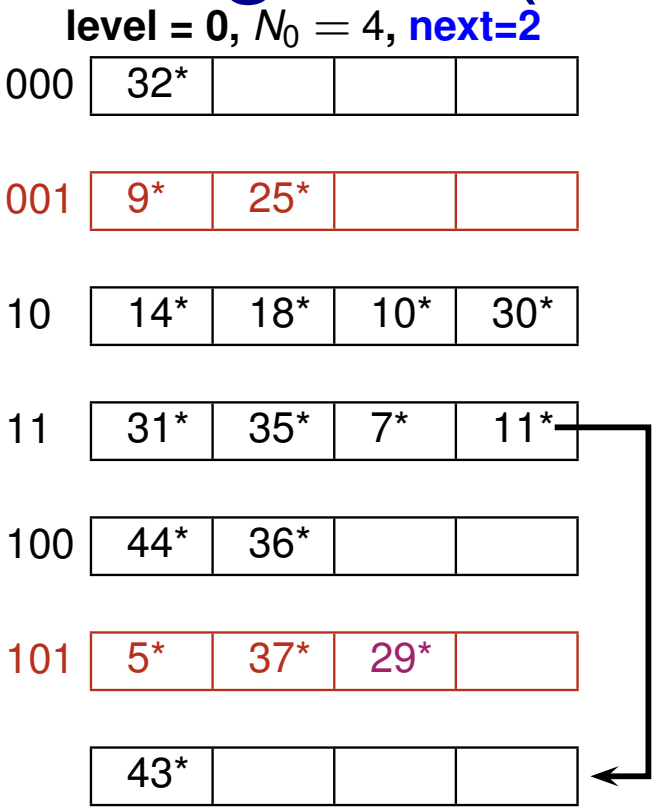
- No overflow, no splitting
- Insert 37* into bucket B_{01}

Linear Hashing: Inserting 29* (011101)



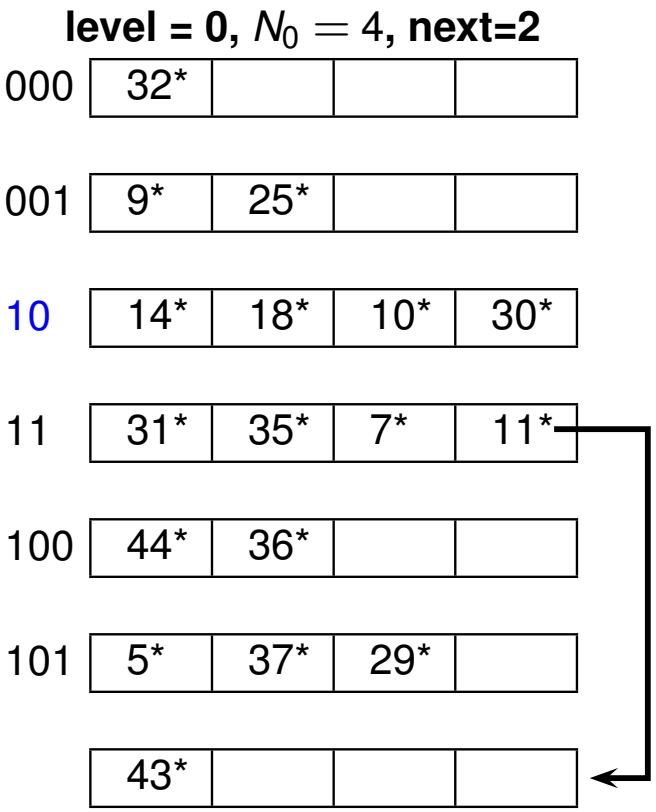
Linear Hashing: Inserting 29* (011101)



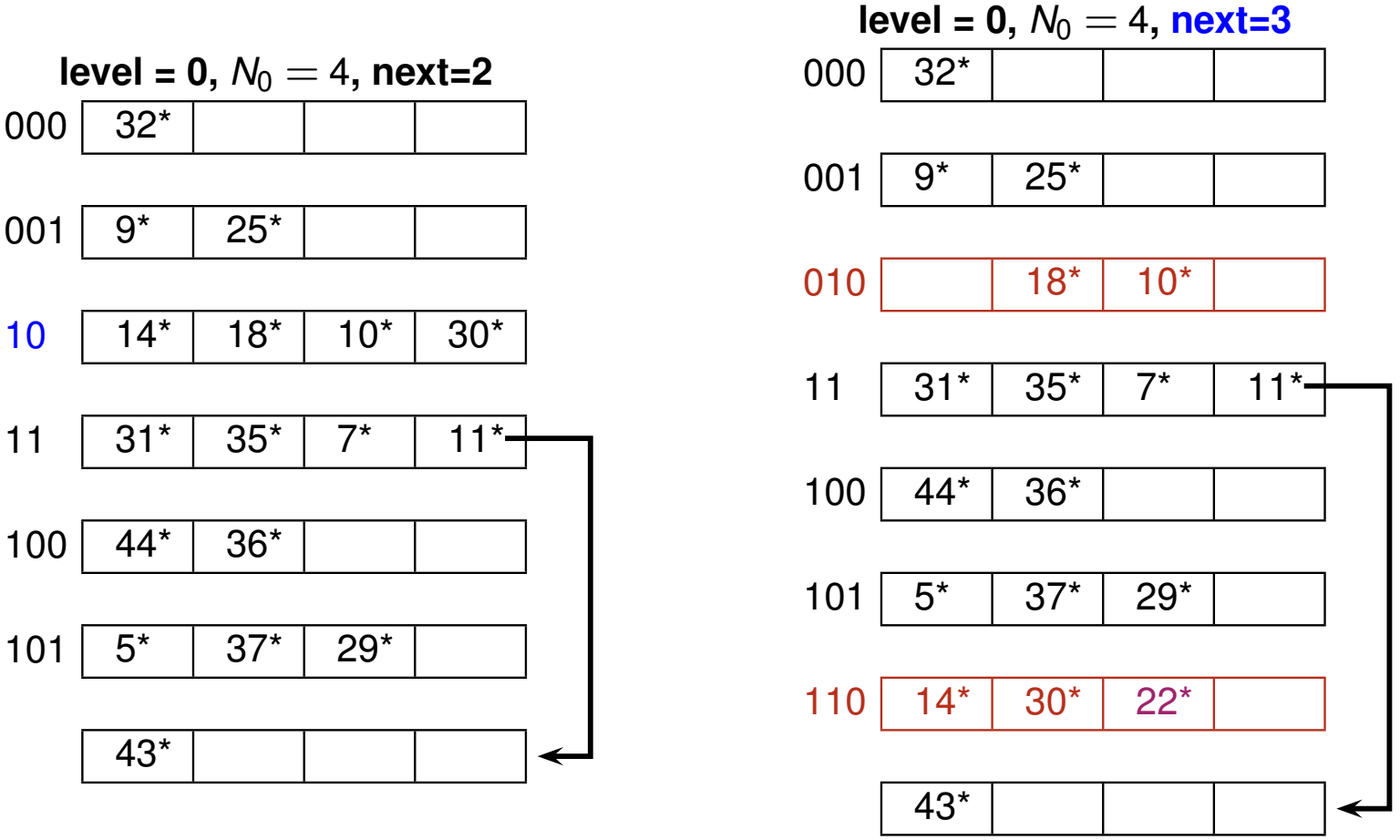


- Bucket B_{01} overflows
- Split bucket B_{01} (5 = 101, 9 = 1001, 25 = 11001, 37 = 100101)
- Increment next to 2
- Insert 29* into bucket B_{101}

Linear Hashing: Inserting 22* (010110)

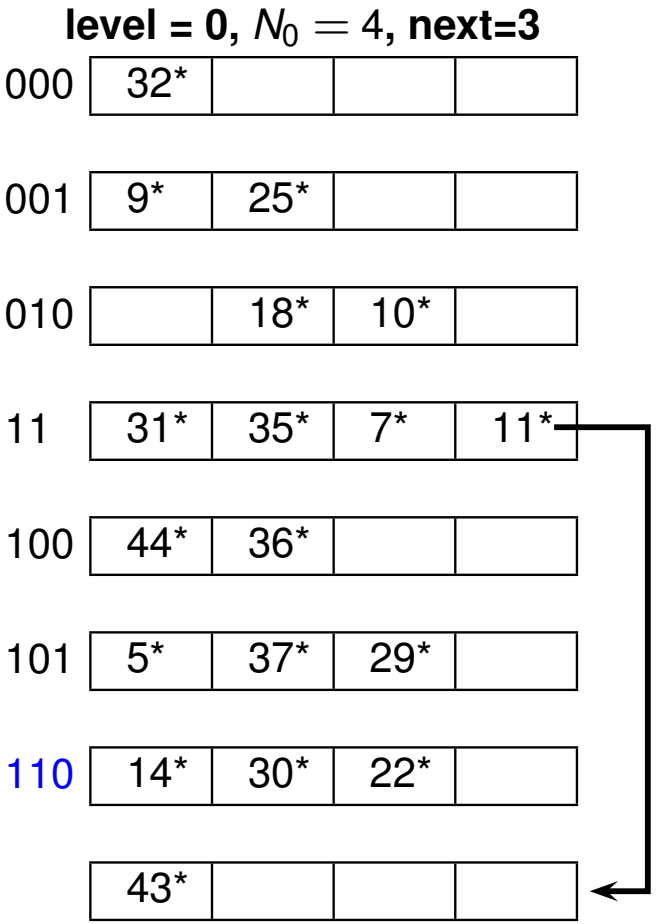


Linear Hashing: Inserting 22* (010110)

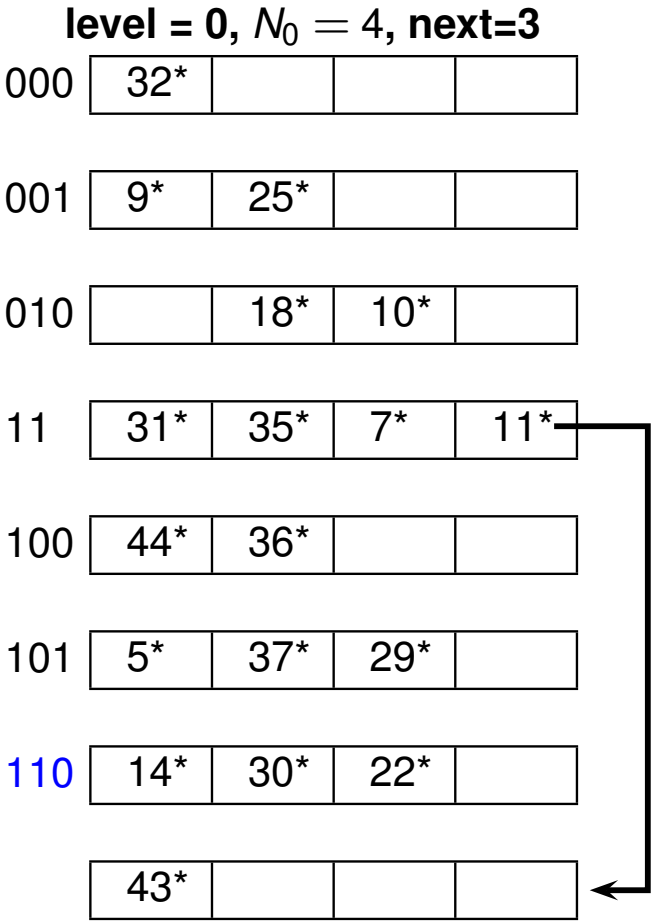


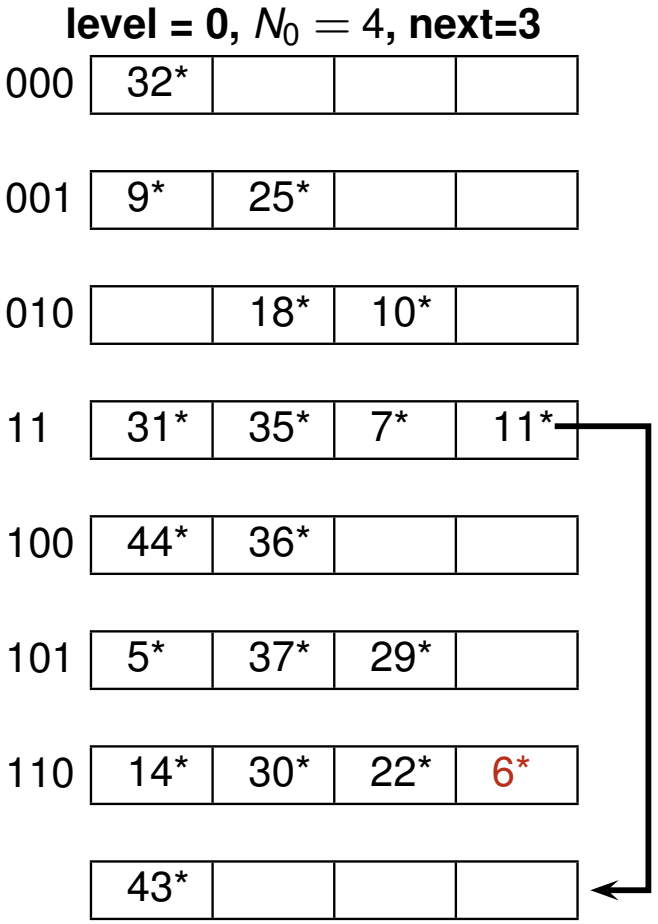
- Bucket B_{10} overflows
- Split bucket B_{10} (10 = 1010, 14 = 1110, 18 = 10010, 30 = 11110)
- Increment next to 3
- Insert 22* into bucket B_{110}

Linear Hashing: Inserting 6* (000110)



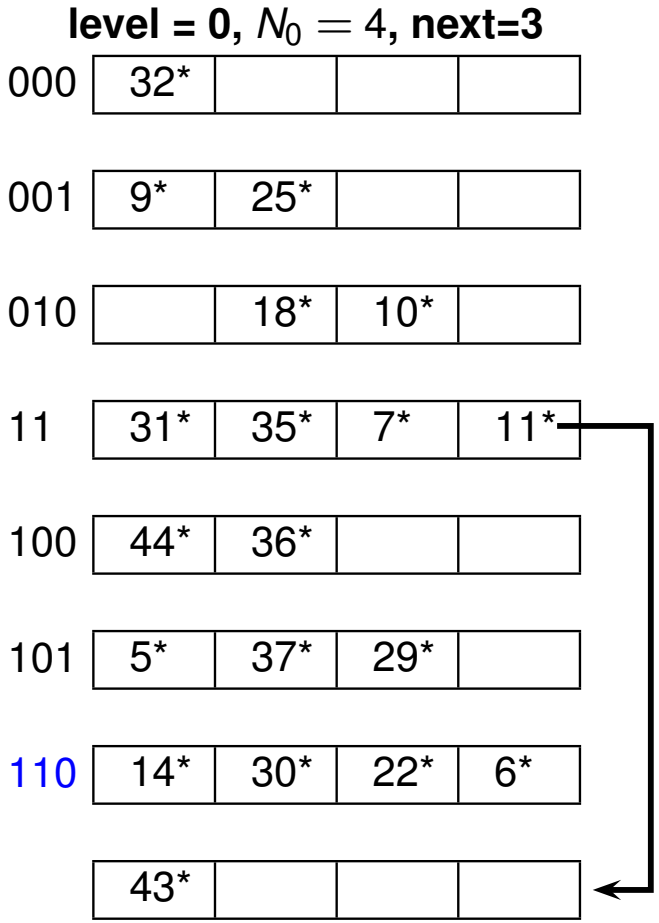
Linear Hashing: Inserting 6* (000110)



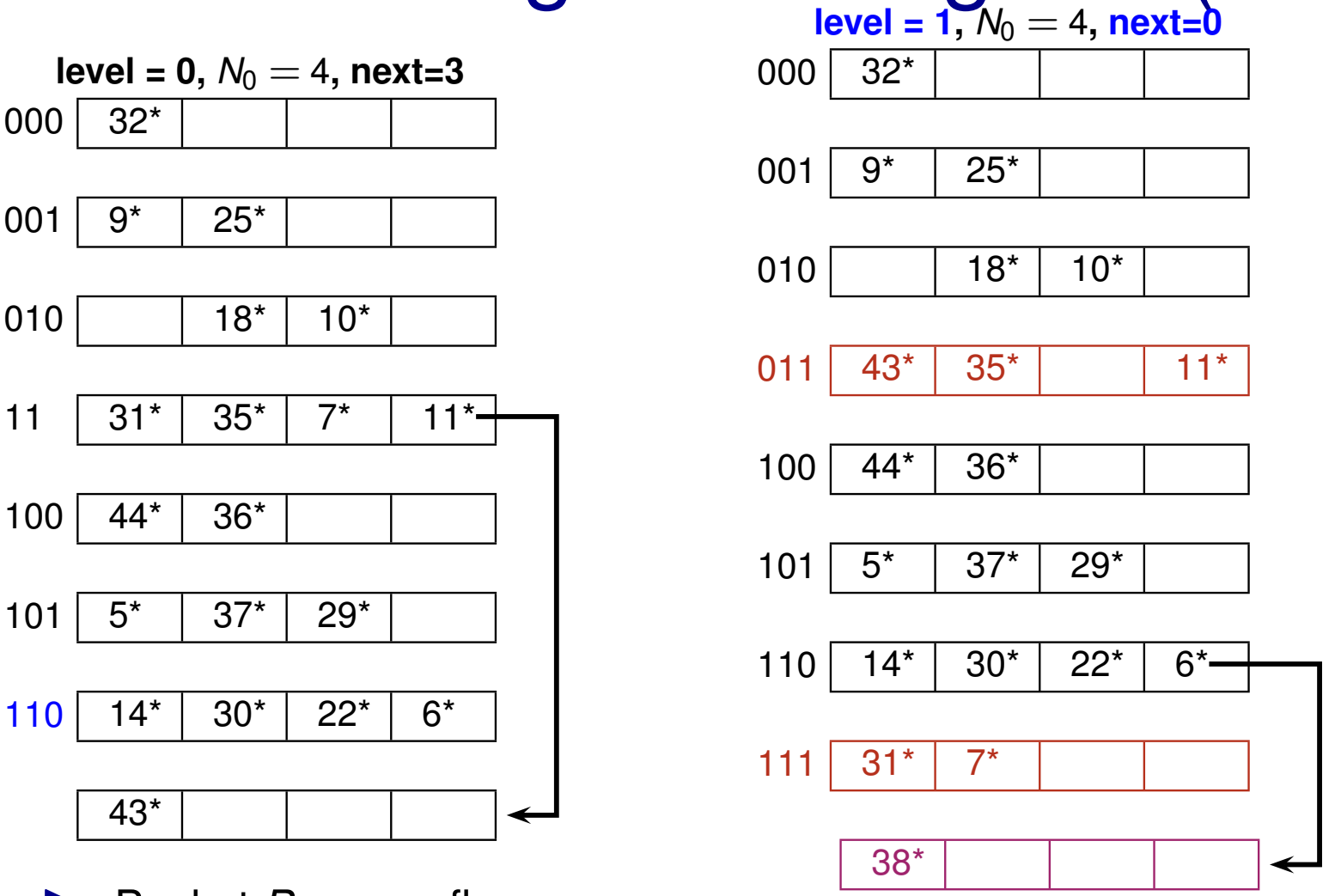


- No overflow, no splitting
- Insert 6* into bucket B_{110}

Linear Hashing: Inserting 38* (100110)

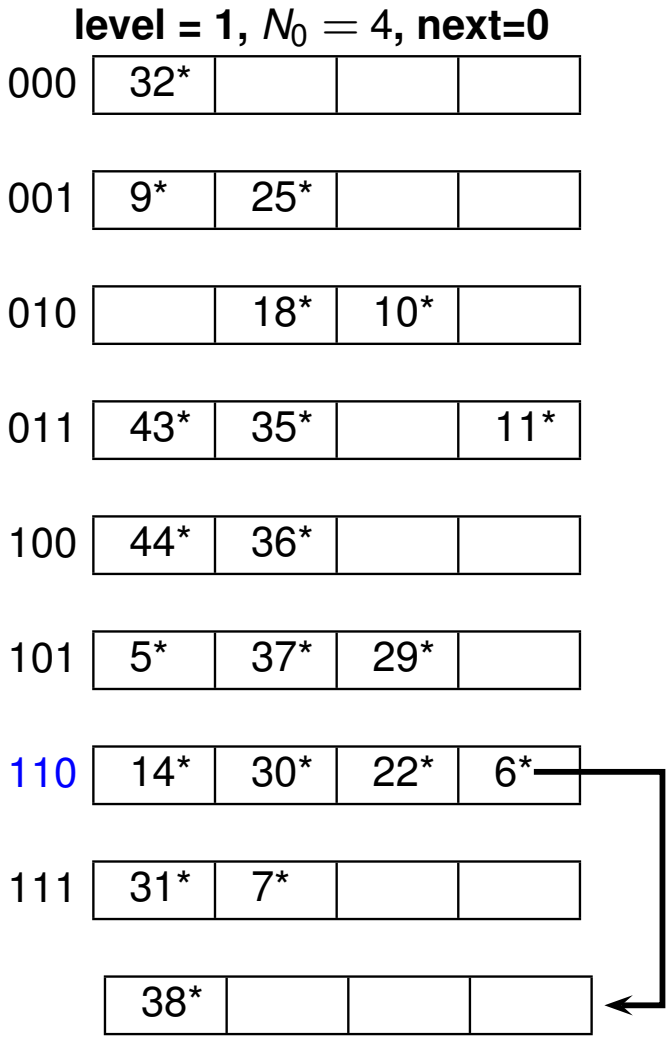


Linear Hashing: Inserting 38* (100110)

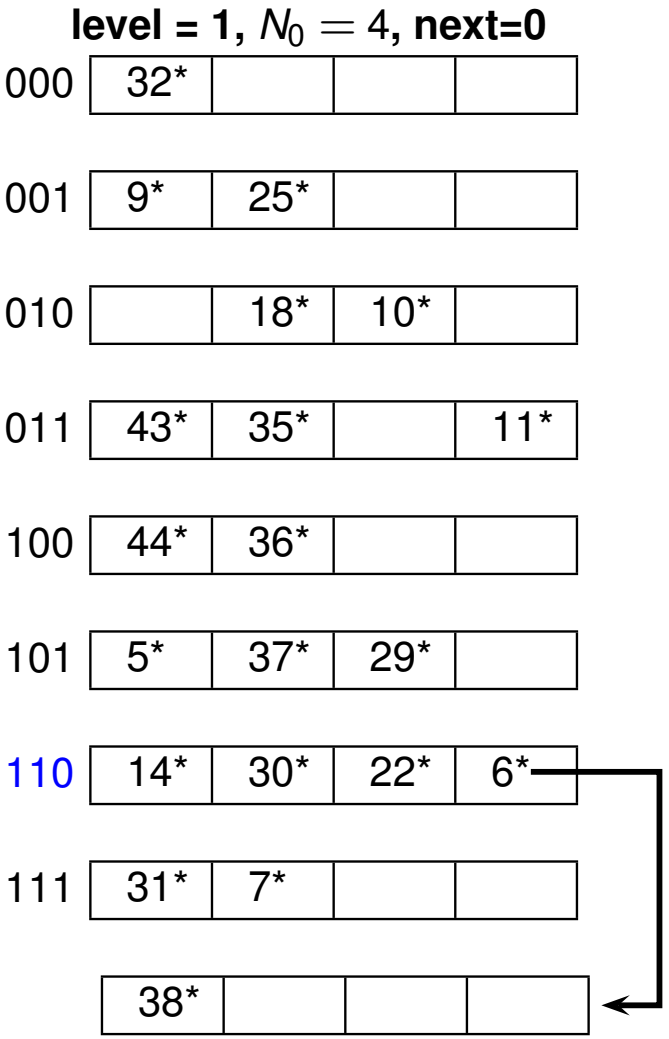


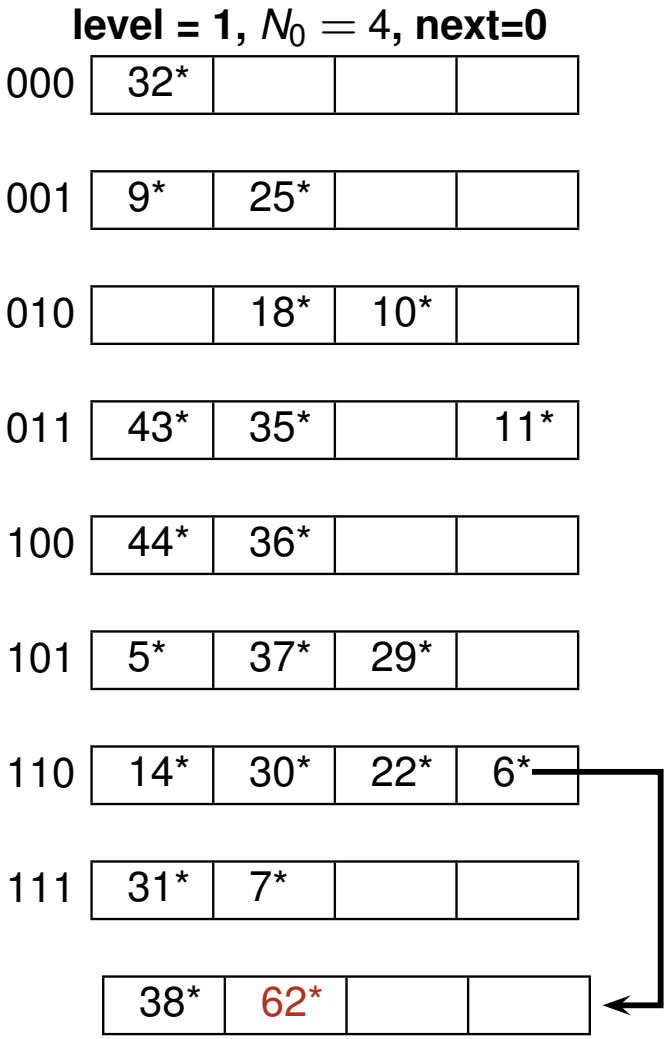
- Bucket B_{110} overflows
- Split bucket B_{11} (7 = 111, 11 = 1011, 31 = 11111, 35 = 100011, 43 = 101011)
- Increment level to 1; reset next to 0
- Insert 38* into overflow page

Linear Hashing: Inserting 62* (111110)



Linear Hashing: Inserting 62* (111110)

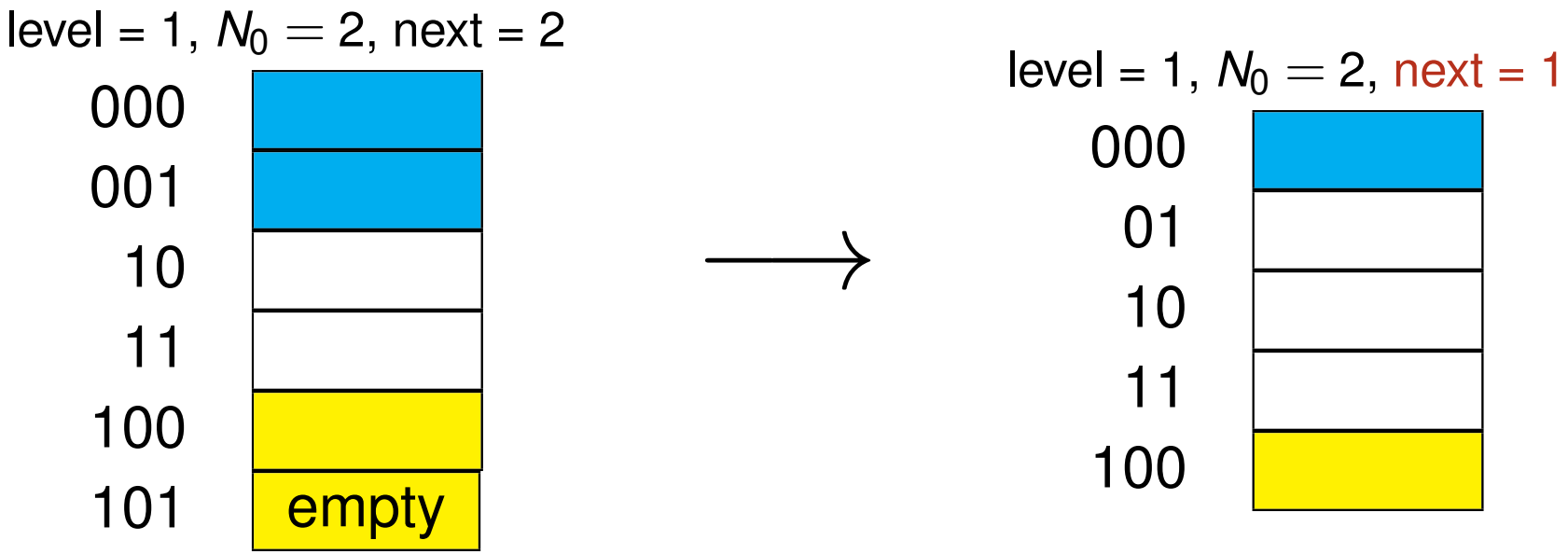




- No overflow, no splitting
- Insert 62^* into bucket B_{110}

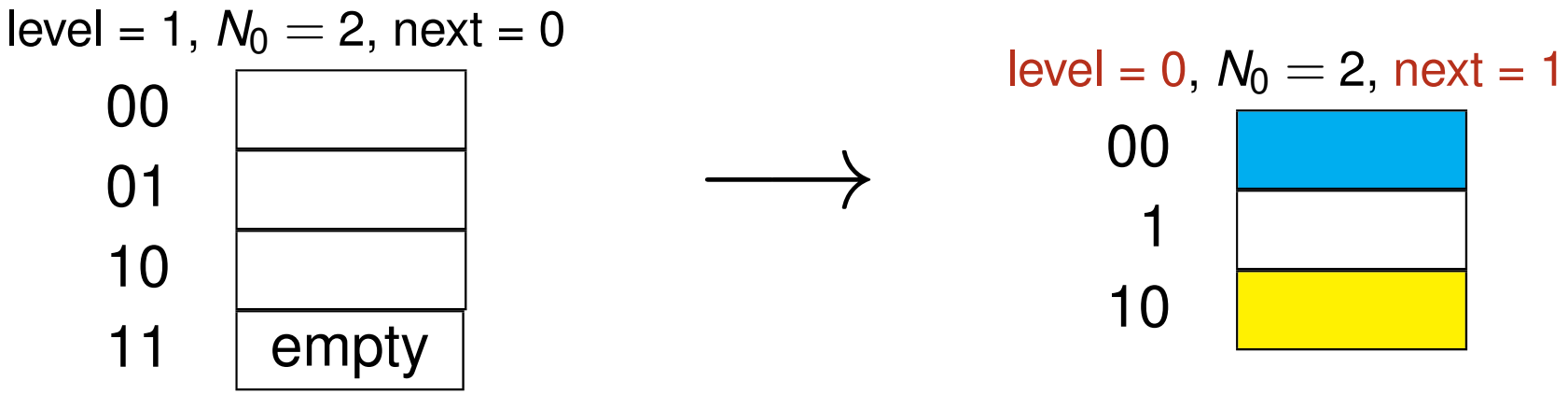
Linear Hashing: Deletion

- Locate bucket and delete entry
- If the last bucket $B_{N_{level}+next-1}$ becomes empty, it can be removed
- **Case 1**: If $next > 0$
  - Decrement `next` by one



Linear Hashing: Deletion (cont.)

- **Case 2**: If $(next = 0)$ and $(level > 0)$
  - Update `next` to point to the last bucket in previous level $B_{N_{level-1}-1}$
  - Decrement `level` by one



Linear Hashing: Performance

- One disk I/O unless the bucket has overflow pages
  - On average 1.2 disk I/O for uniform data distribution
  - Worst case: I/O cost is linear in the number of data entries
- Poor space utilization with skewed data distribution

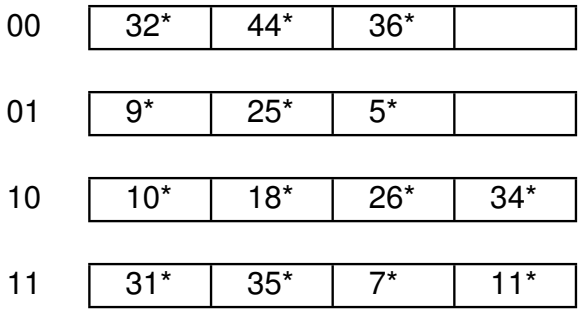
Linear Hashing: Summary

- File grows dynamically by splitting buckets linearly in rounds
  - Each bucket split adds one new bucket to file
- The bucket B_j for a search key k is determined using two hash functions (dependent on round)
  - Assume initial file has 2^m buckets & current round number is level
  - j = last (m+level) bits of h(k) if B_j has not yet been split in current round; otherwise, j = last (m+level+1) bits of h(k)
- In this course, we assume a bucket split is triggered whenever some bucket overflows due to an insertion
- Overflow pages are required for an overflowed bucket if it is not the next bucket to be split

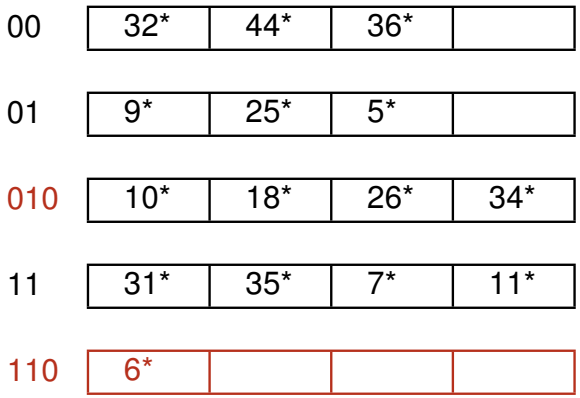
Another Idea for Dynamic Hashing

- Similar to Linear Hashing
  - Number of buckets grow dynamically
  - Uses some number of the least significant bits of h(k) to determine the bucket address for search key k
- But adds a new bucket (as split image) whenever an existing bucket overflows
  - No overflow pages! (except when the number of collisions exceed page capacity)
    - ★ Two data entries collide if they have the same h(.) value

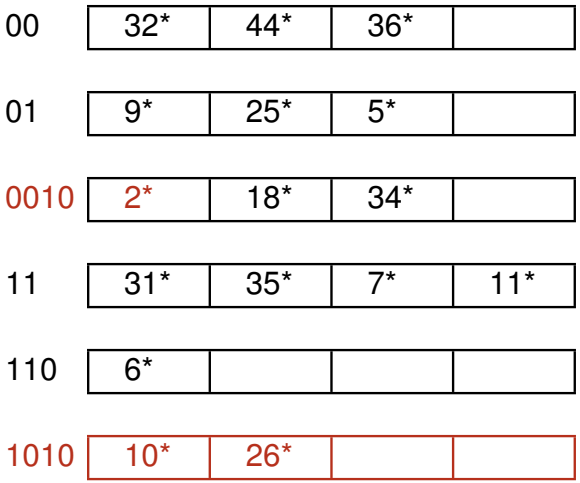
Another Idea for Dynamic Hashing (cont.)



Initial state with 4 buckets



After insertion of 6^*

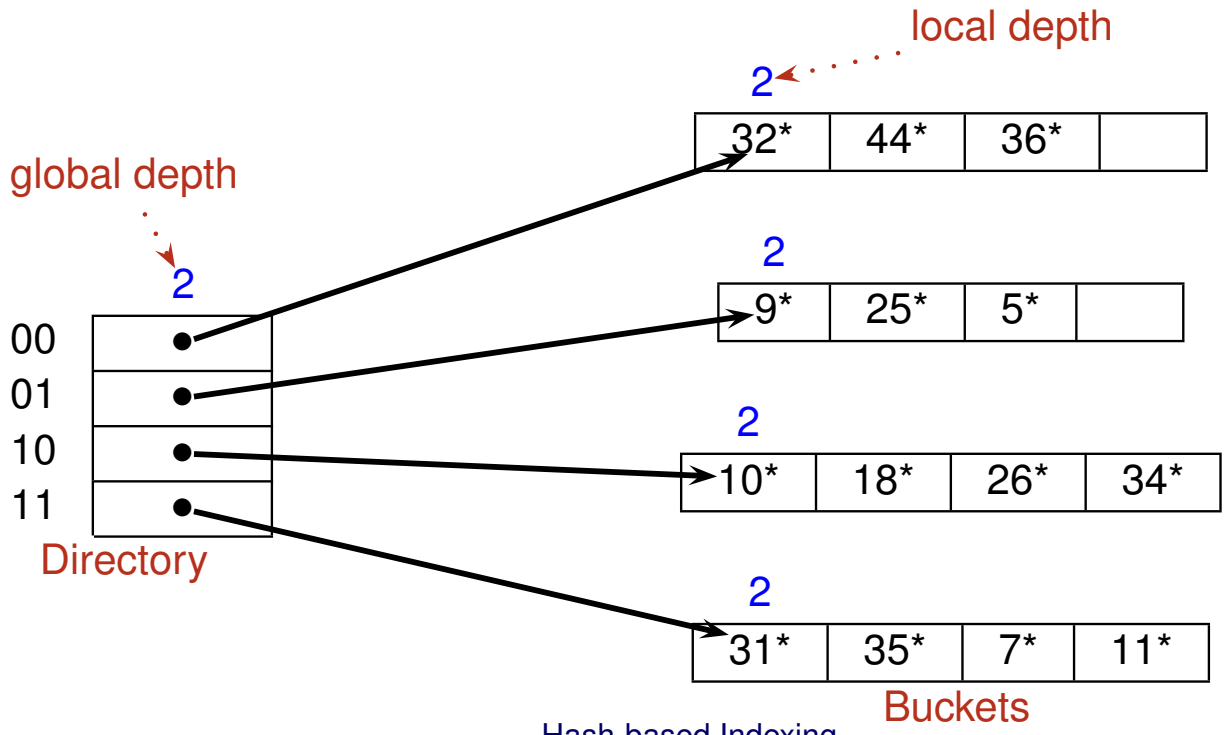


After insertion of 2^*

- The order in which buckets are split is random
- How to keep track of the number of last bits used as bucket addresses for different buckets?
- How to locate a bucket given its bucket address?

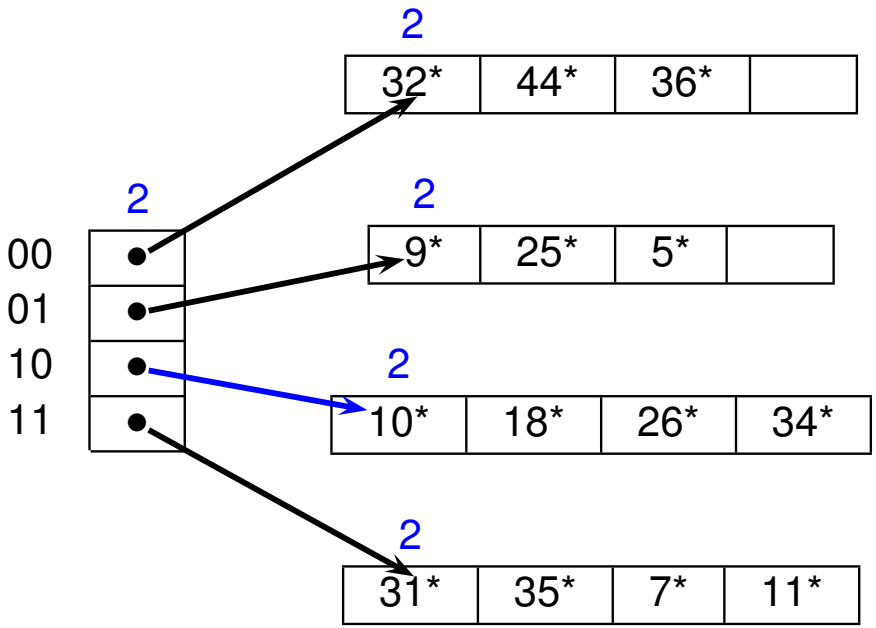
Extendible Hashing

- Uses a directory of pointers to buckets
  - Directory expands dynamically as buckets overflow
- Directory has 2^d entries
  - d is called global depth of the hashed file
  - Each directory entry has a unique d-bit address $b_d b_{d-1} \cdots b_2 b_1$
  - Two directory entries are said to correspond if their addresses differ only in the d^{th} bit (i.e., b_d); such entrie are called corresponding entries
  - Each bucket maintains a local depth (denoted by $\ell \in [0, d]$)
    - ★ All entries in a bucket with local depth ℓ have the same last ℓ bits in h(.)



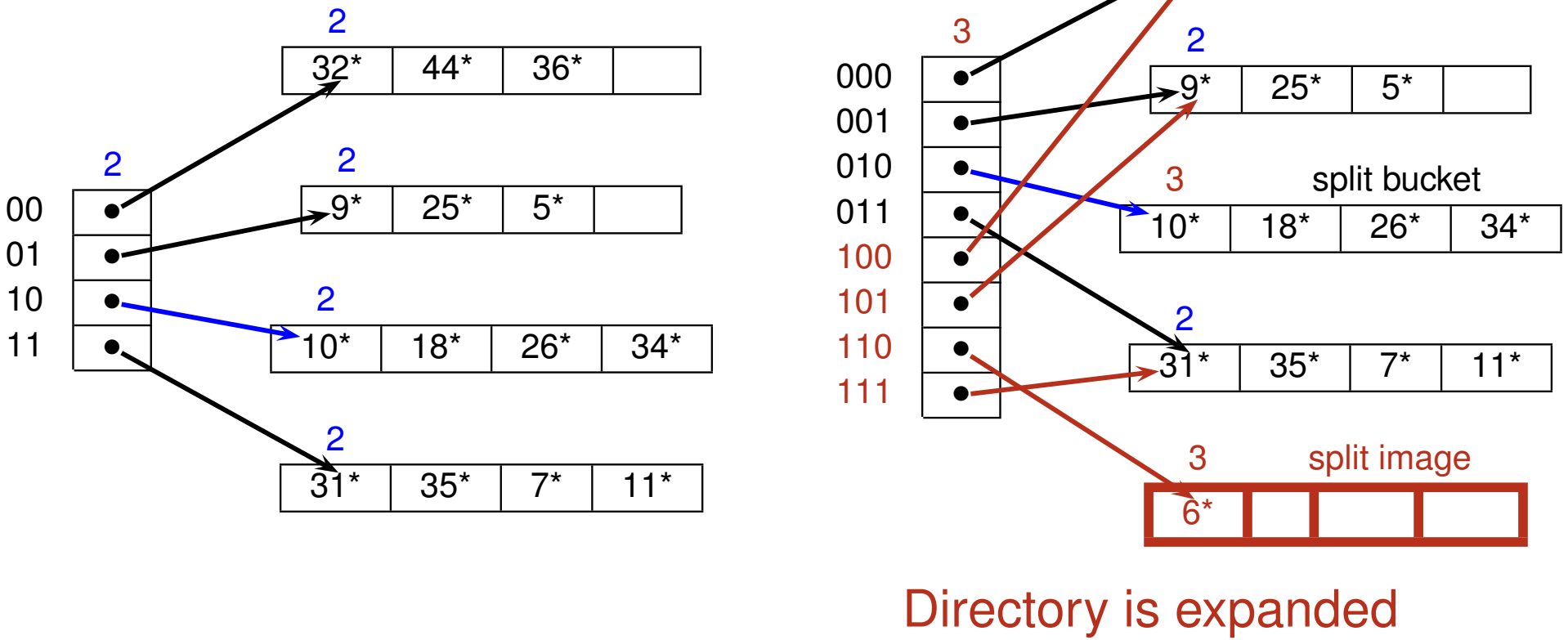
Handling Bucket Overflow (Case 1)

- When a bucket overflows, it is split
  - Allocate a new bucket called its split image
  - Redistribute entries (including new entry) between split bucket & its split image
- Case 1: Split bucket's local depth is equal to global depth
- Example: Inserting 6* (110)



Handling Bucket Overflow (Case 1)

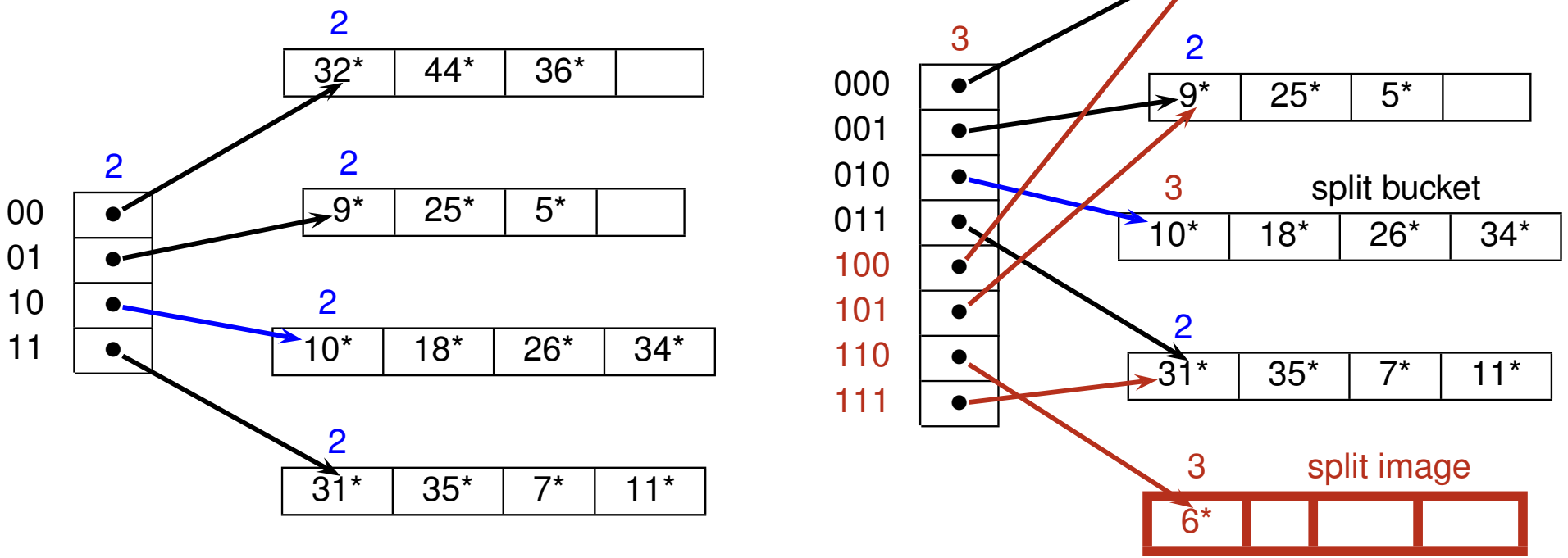
- Initially, all buckets have local depth $\ell = d$
- A bucket split causes directory to double only if bucket's $\ell = d$
  - d is incremented by one whenever the directory is doubled
- ℓ is incremented by one for a bucket whenever it splits
- A split bucket & its image have the same local depth



Directory is expanded

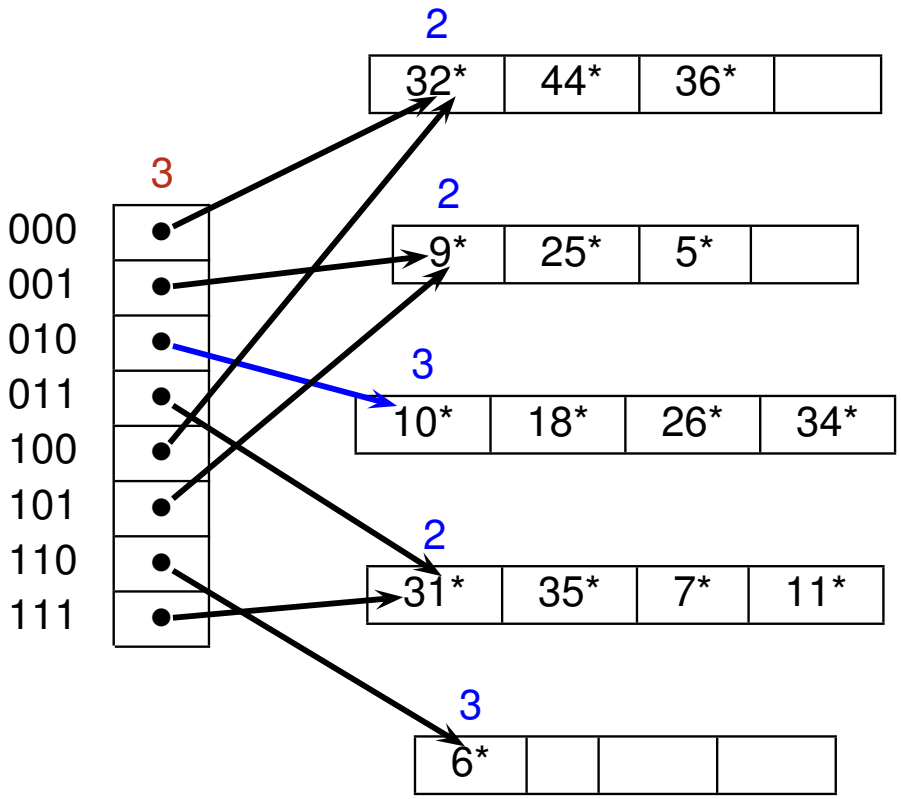
Handling Bucket Overflow (Case 1)

- When the directory is doubled,
  - Each new directory entry (except for the entry for the split image) points to the same bucket as its corresponding entry
- Number of directory entries pointing to a bucket = $2^{d-\ell}$
- Example: Inserting 6* (110)



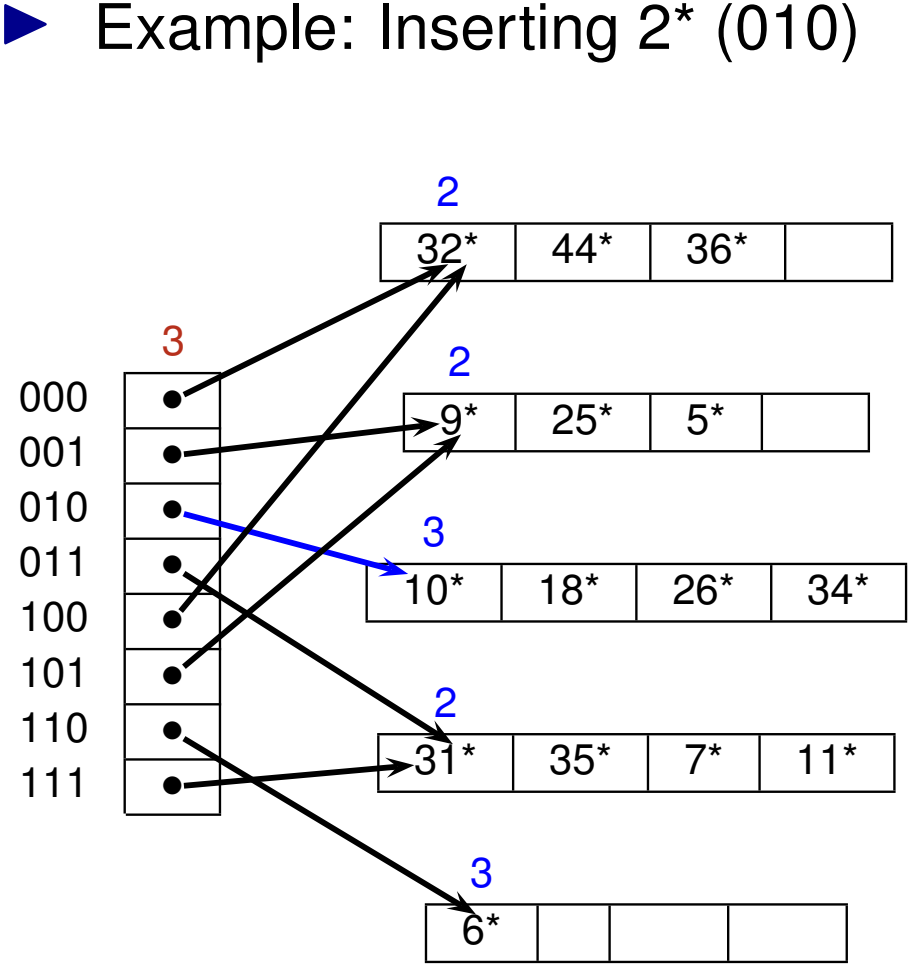
Another Bucket Overflow Example (Case 1)

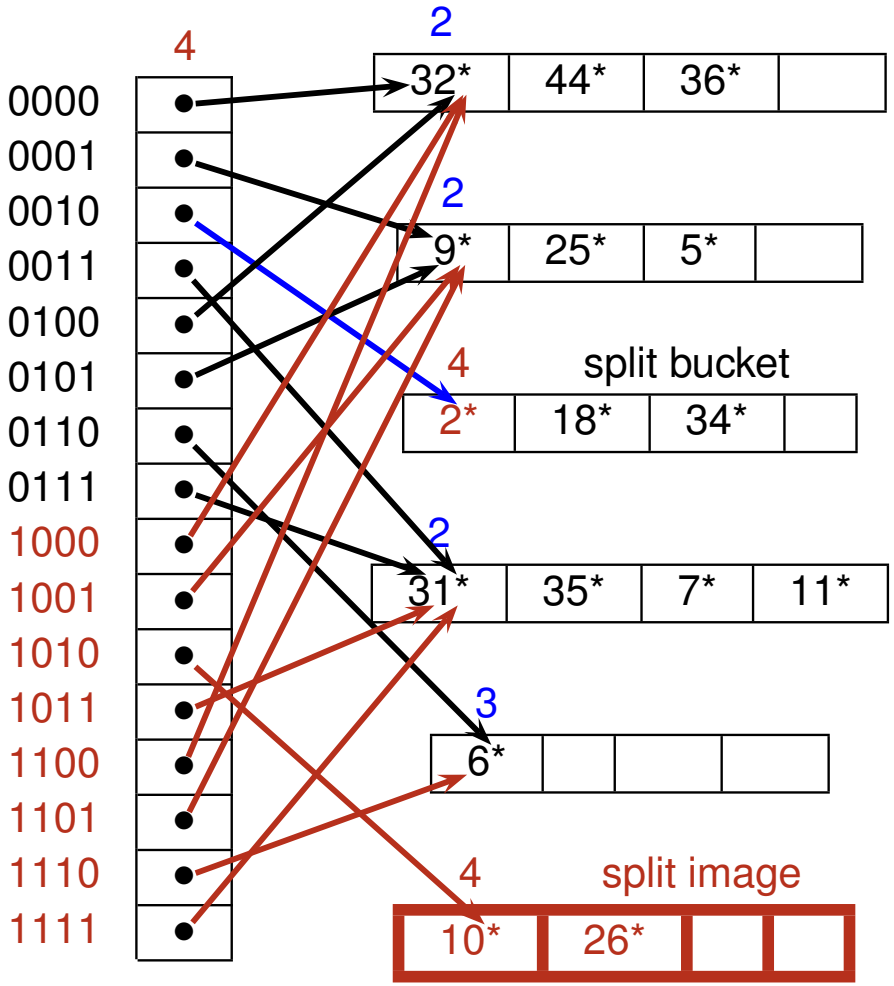
- Example: Inserting 2* (010)



Another Bucket Overflow Example (Case 1)

- Example: Inserting 2* (010)

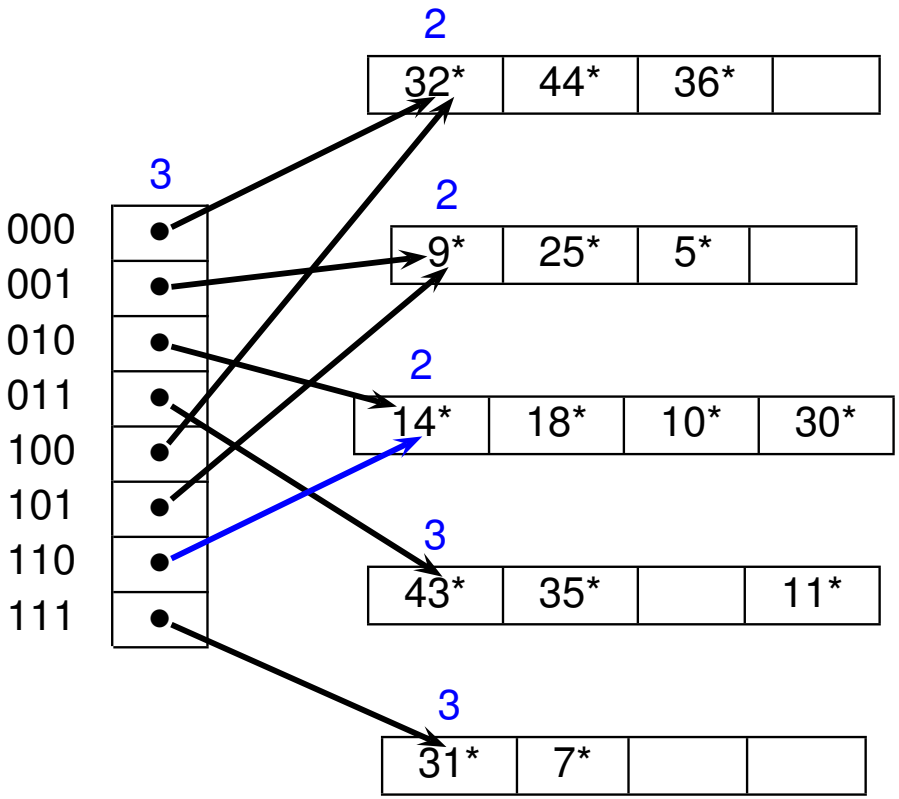




Directory is expanded

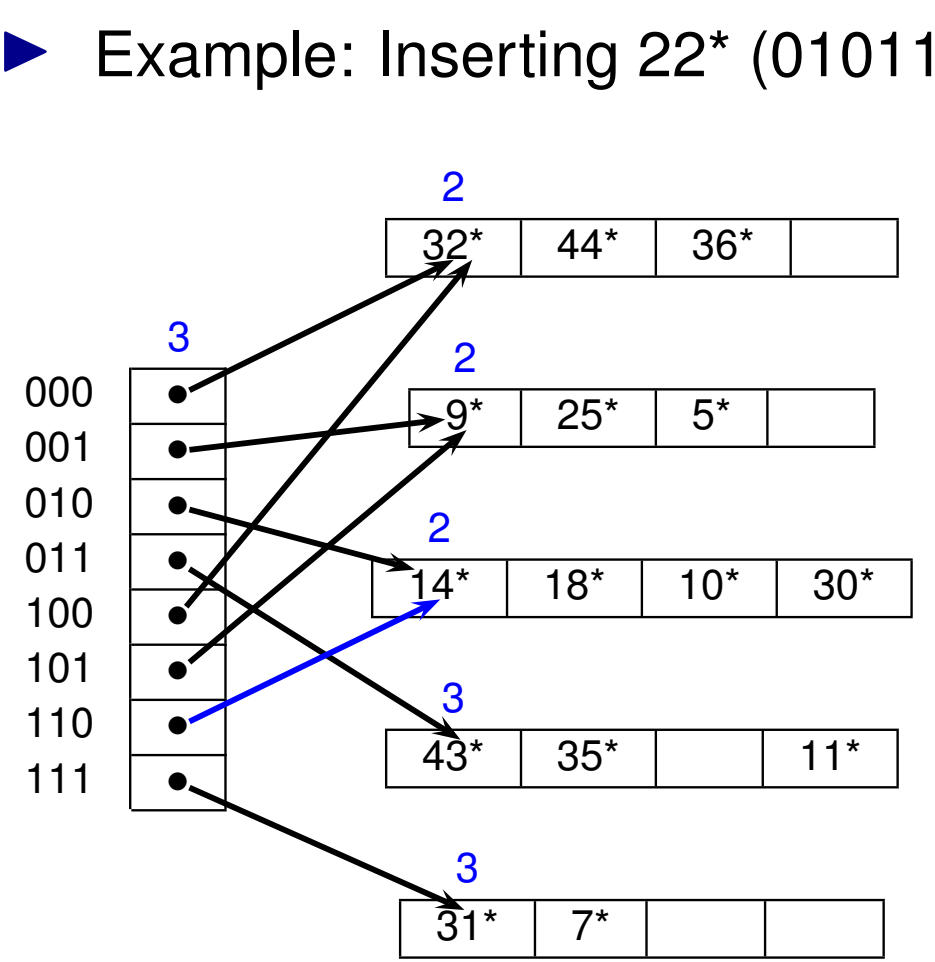
Handling Bucket Overflow (Case 2)

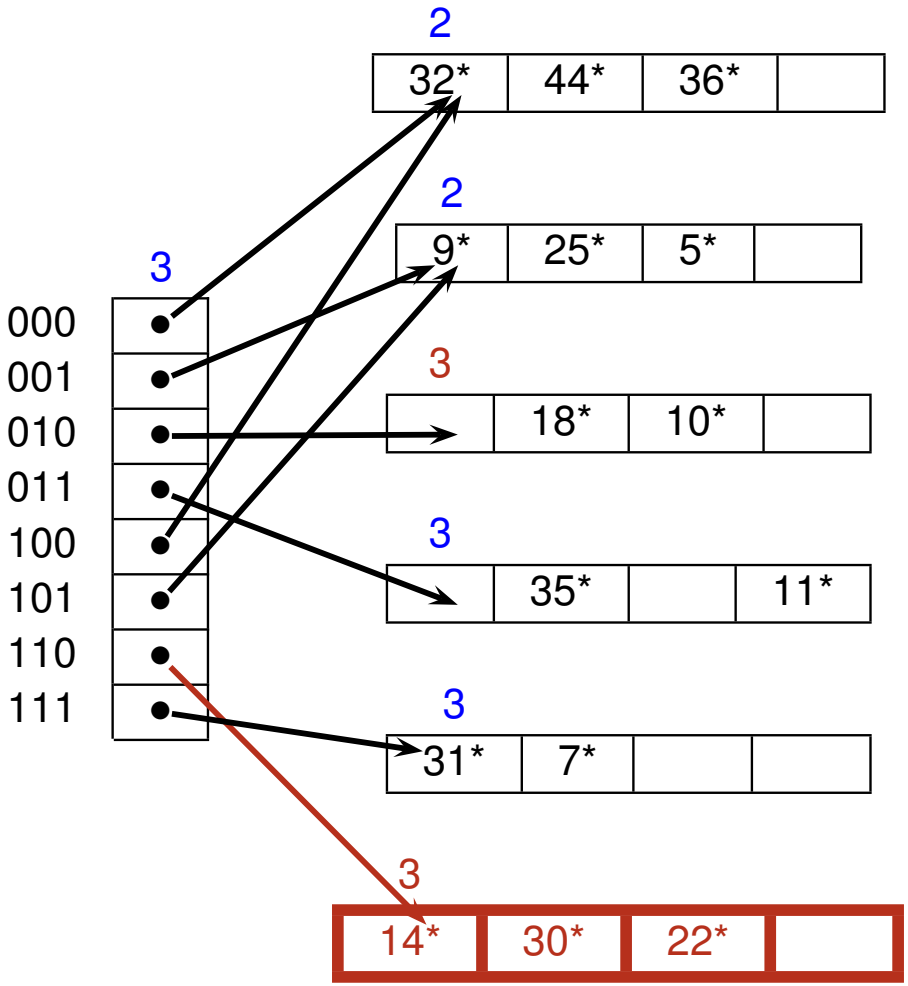
- Case 2: Split bucket's local depth $<$ global depth
- Example: Inserting 22* (010110)



Handling Bucket Overflow (Case 2)

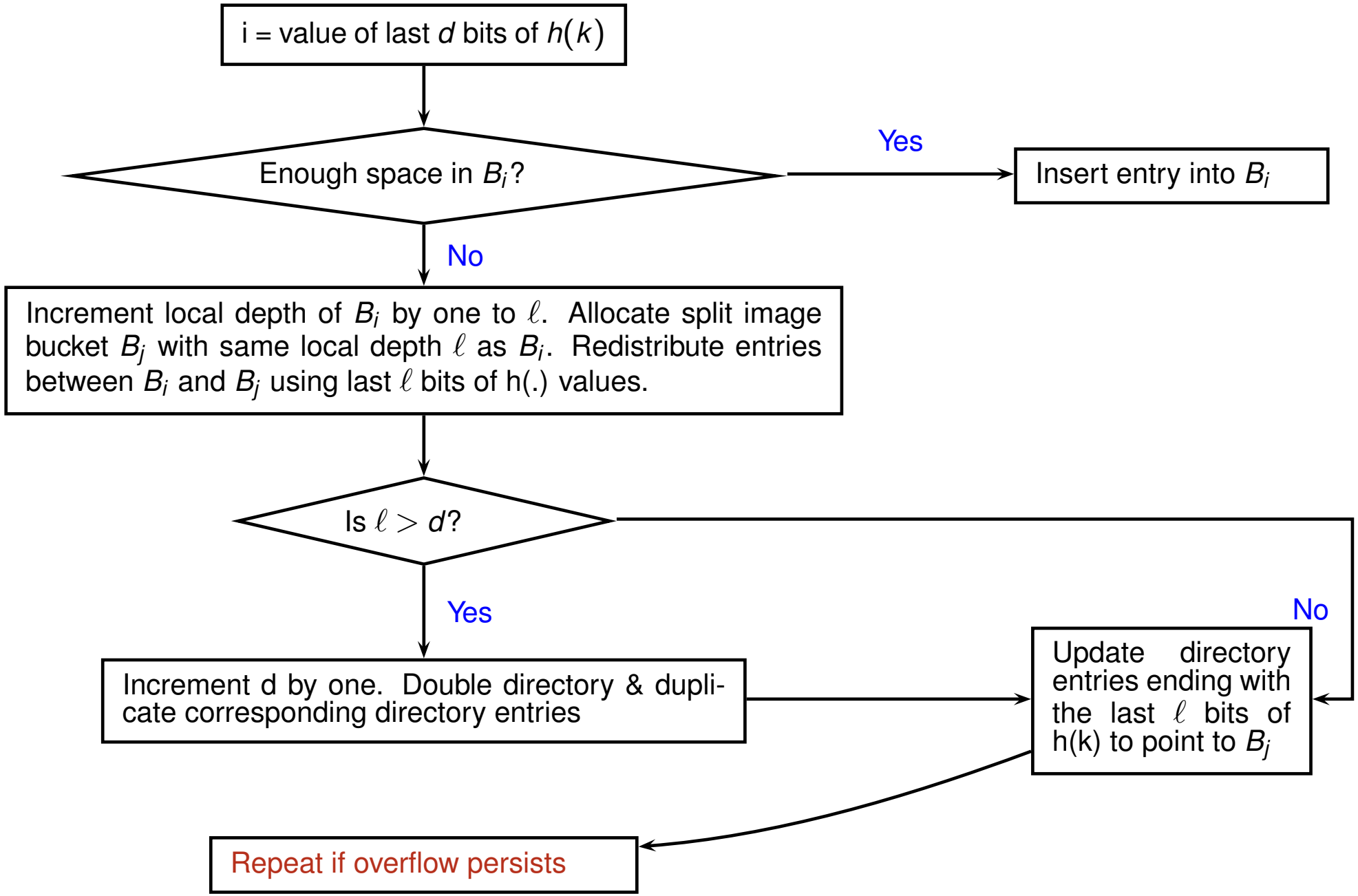
- Case 2: Split bucket's local depth $<$ global depth
- Example: Inserting 22* (010110)





Directory is not expanded

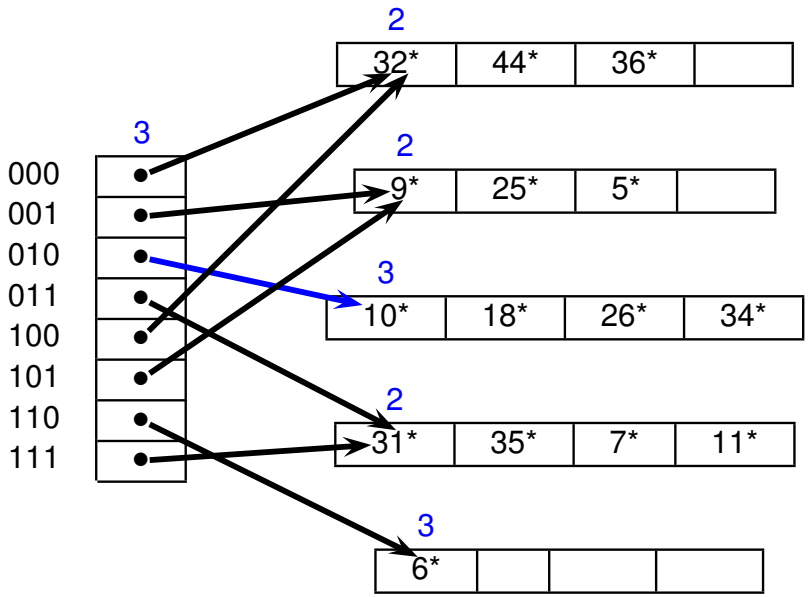
Inserting data entry with search key k

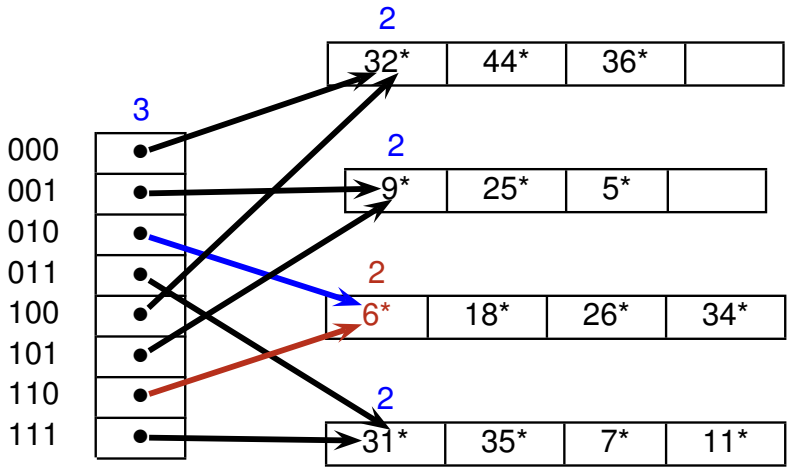


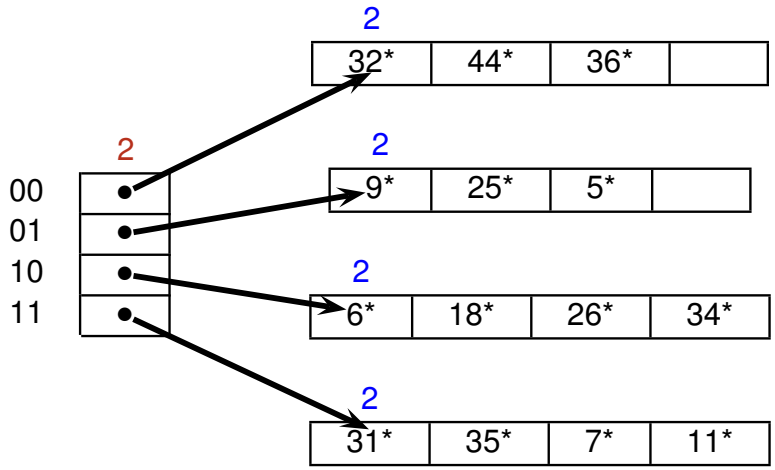
Extendible Hashing: Deletion

- Locate bucket B_i containing entry & delete entry
- If B_i becomes empty, B_i can be merged with the bucket B_j where i & j differs only in the ℓ^{th} bit
  - B_i is deallocated
  - ℓ is decremented by one
  - Directory entries that point to B_i are updated to point to B_j
- More generally, B_i & B_j (where i & j differs only in ℓ^{th} bit) can be merged if their entries can fit within a bucket
- If each pair of corresponding entries point to the same bucket, directory can be halved
  - d is decremented by one

Example: Deleting 10* (1010)







Extendible Hashing: Performance

- At most 2 disk I/Os for equality selection
  - At most one disk I/O if directory fits in main memory
- Handling collisions:
  - Two data entries collide if they have the same hashed value
  - Overflow pages are needed when the number of collisions exceeds page capacity