Hashing

BBM371 Data Management

Motivation

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires O(n) time
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires O(log n) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an O(1), that is, constant time search?
    - We can do it *if* the array is organized in a particular way

Hashing

- Suppose we were to come up with a “magic function” that, given a value to search for, would tell us exactly where in the array to look
  - If it’s in that location, it’s in the array
  - If it’s not in that location, it’s not in the array
- This function would have no other purpose
- If we look at the function’s inputs and outputs, they probably won’t “make sense”
- This function is called a hash function because it “makes hash” of its inputs

Hash Function

- Hash function h:
  - Mapping from U to the slots of a hash table $T[0..m-1]$.

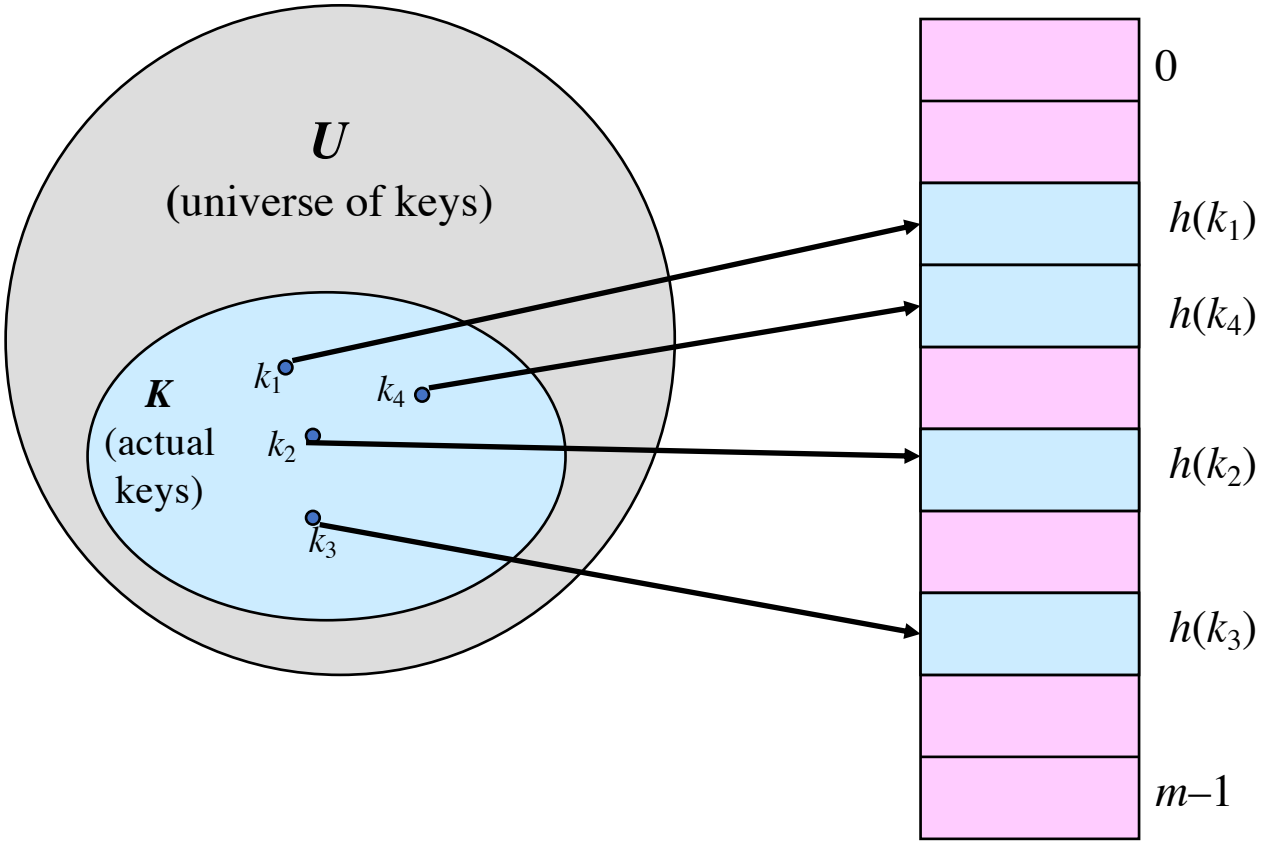
    $h : U \rightarrow \{0,1,\ldots, m-1\}$

- With arrays, key k maps to slot $A[k]$.

- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.

- $h[k]$ is the *hash value* of key k.

Hashing (cont'd)



Finding the Hash Function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function ☹
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash function would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to *start* looking!

Example Hash Function

- Map a key *k* into one of the *m* slots by taking the remainder of *k* divided by *m* (Division Method). That is,

$$h(k) = k \bmod m$$

h(ssn) = ssn mod 100 (i.e., the last two digits)

e.g., if ssn = 10123411 then h(10123411) = 11

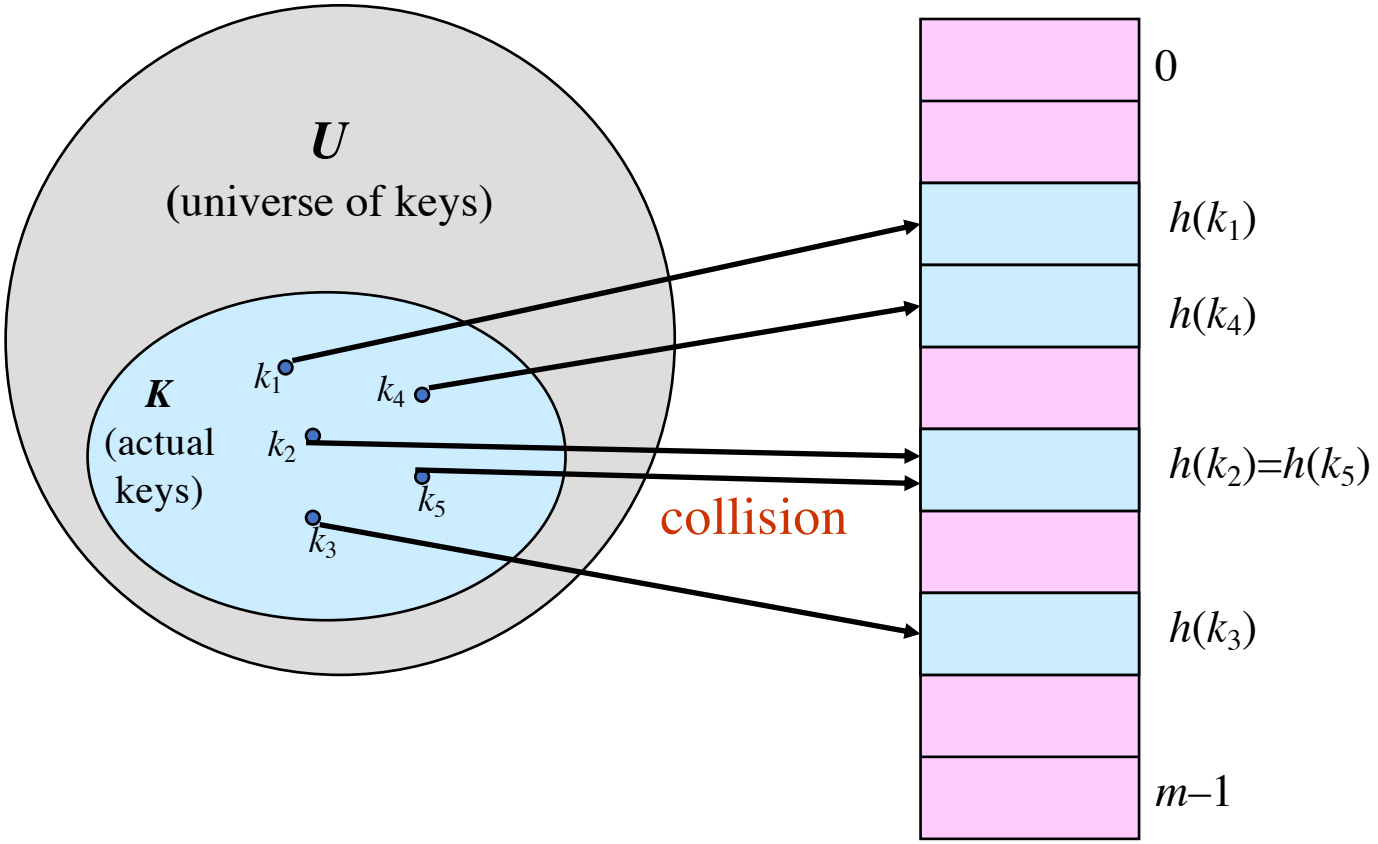
Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.
- Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0$ = 141,764,947.

Collision

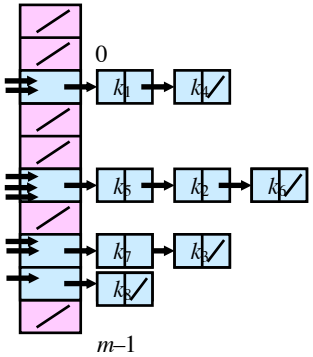
- When two values hash to the same array location, this is called a collision
- Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Collisions

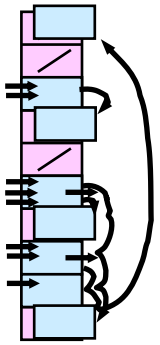

U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

Handling Collisions

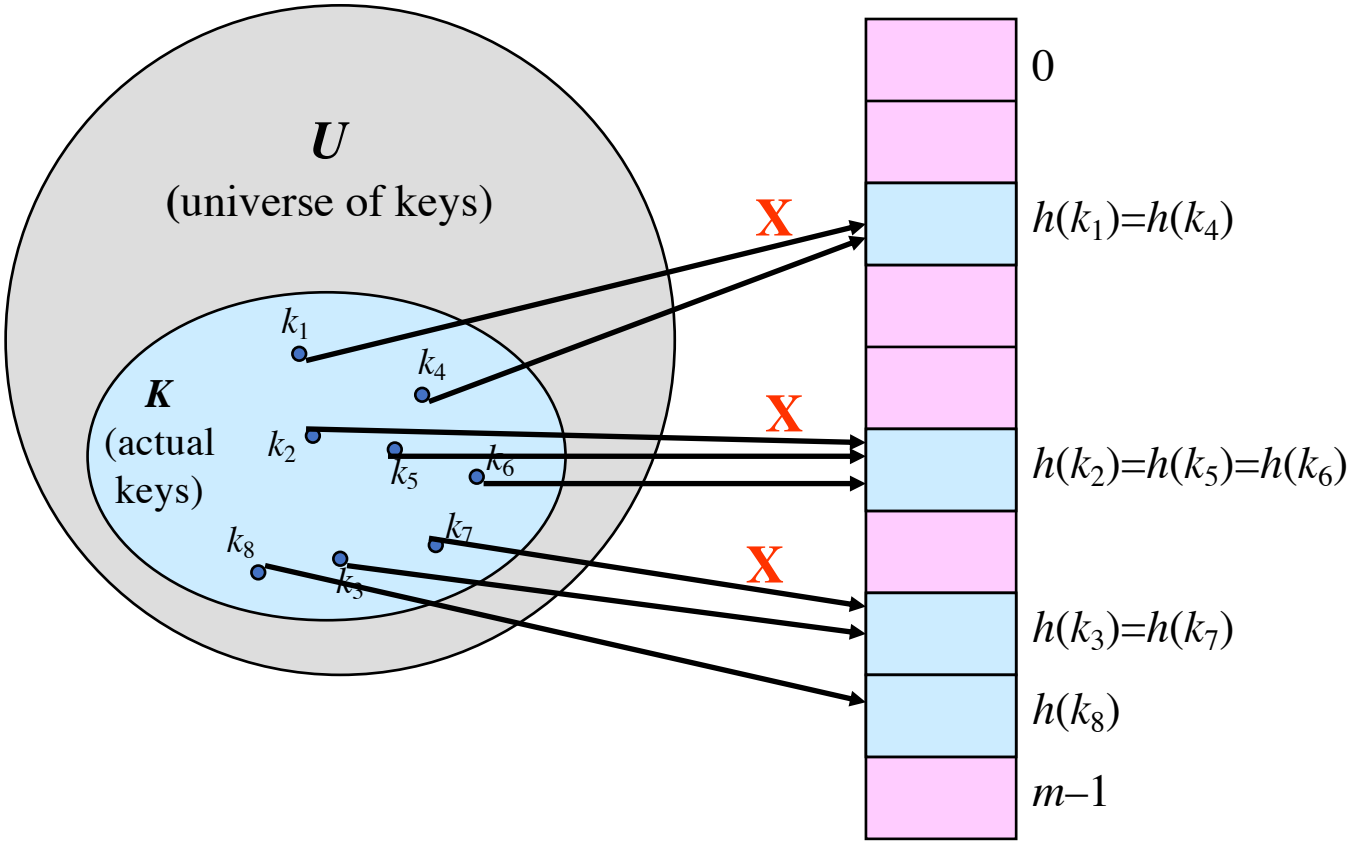
- Chaining
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



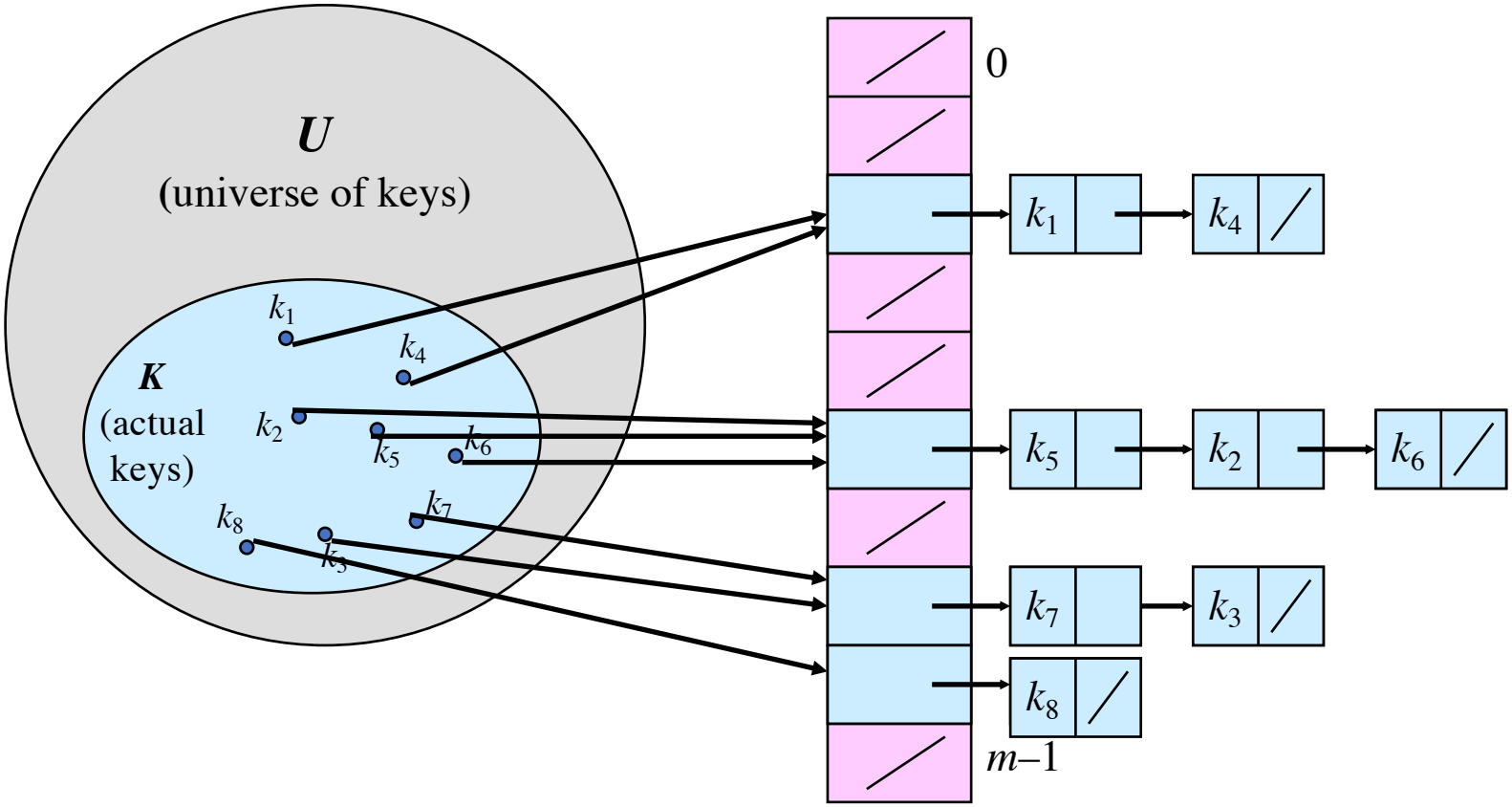
- Open Addressing
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Chaining


U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_6
k_8
k_3
k_7
X
X
X
0
$h(k_1)=h(k_4)$
$h(k_2)=h(k_5)=h(k_6)$
$h(k_3)=h(k_7)$
$h(k_8)$
$m-1$

Chaining (cont'd)

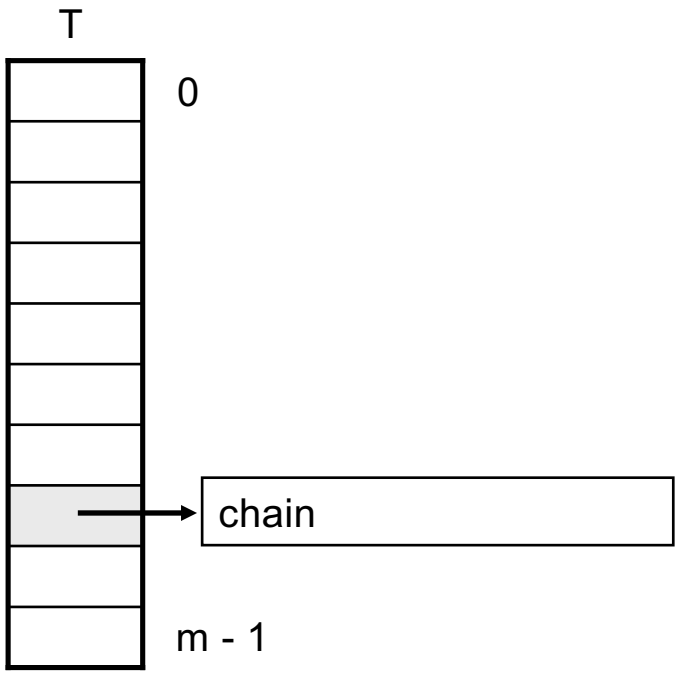


Dictionary Operations with Chaining

- Chained-Hash-Insert (*T, x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – *O*(1).
- Chained-Hash-Delete (*T, x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – proportional to length of list with singly-linked lists. *O*(1) with doubly-linked lists.
- Chained-Hash-Search (*T, k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity – proportional to length of list.

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function



Open Addressing

- If collision occurs, open addressing scheme **probes** for some other empty (or open) location in which to place the item.

- The sequence of locations that we examine is called the **probe sequence.**

- There are different open-addressing schemes:
  - *Linear Probing*
  - *Quadratic Probing*
  - *Double Hashing*

Linear Probing

- In linear probing, we search the hash table sequentially starting from the original hash location.
  - If a location is occupied, we check the next location
  - We wrap around from the last table location to the first table location if necessary.

Linear Probing: Example

- Table Size is 11 (0..10)
- Hash Function: h(x) = x mod 11
- Insert keys:
  - 20 mod 11 = 9
  - 30 mod 11 = 8
  - 2 mod 11 = 2
  - 13 mod 11 = 2 ➔ 2+1=3
  - 25 mod 11 = 3 ➔ 3+1=4
  - 24 mod 11 = 2 ➔ 2+1, 2+2, 2+3=5
  - 10 mod 11 = 10
  - 9 mod 11 = 9 ➔ 9+1, 9+2 mod 11 =0

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Linear Probing: Clustering Problem

- One of the problems with linear probing is that table items tend to **cluster** together in the hash table.
  - This means that the table contains groups of consecutively occupied locations.
- This phenomenon is called **primary clustering**.
- Clusters can get close to one another, and merge into a larger cluster.
  - Thus, the one part of the table might be quite dense, even though another part has relatively few items.
- Primary clustering causes long probe searches and therefore decreases the overall efficiency.

Quadratic Probing

- Primary clustering problem can be almost eliminated if we use **quadratic probing** scheme.

- In quadratic probing,
  - We start from the original hash location **`i`**
  - If a location is occupied, we check the locations **`i+1`**^{2} **`, i+2`**^{2} **`, i+3`**^{2} **`, i+4`**^{2} **`...`**
  - We wrap around from the last table location to the first table location if necessary.

Quadratic Probing: Example

- Table Size is 11 (0..10)
- Hash Function: h(x) = x mod 11
- Insert keys:
  - 20 mod 11 = 9
  - 30 mod 11 = 8
  - 2 mod 11 = 2
  - 13 mod 11 = 2 ➔ $2+1^2=3$
  - 25 mod 11 = 3 ➔ $3+1^2=4$
  - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
  - 10 mod 11 = 10
  - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

Double Hashing

- Double hashing also reduces clustering.
- In linear probing and quadratic probing , the probe sequences are independent from the key.
- We can select increments used during probing using a second hash function. The second hash function h_2 should be:

  $h_2(key) \neq 0$

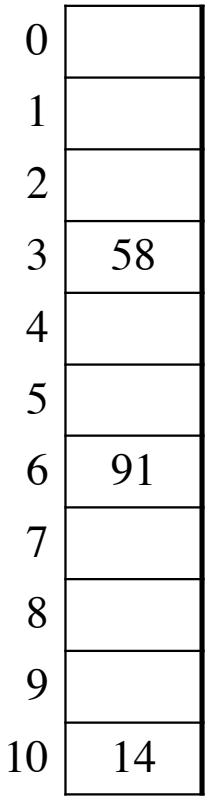
  $h_2 \neq h_1$

- We first probe the location $h_1(key)$
  - If the location is occupied, we probe the location $h_1(key)+h_2(key)$, $h_1(key)+(2*h_2(key))$, ...

Double Hashing: Example

- Table Size is 11 (0..10)
- Hash Functions: $h_1(x) = x \bmod 11$

  $h_2(x) = 7 - (x \bmod 7)$
- Insert keys:
  - 58 mod 11 = 3
  - 14 mod 11 = 3 ➔ 3+7=10
  - 91 mod 11 = 3 ➔ 3+7, 3+2*7 mod 11=6



Open Addressing: Retrieval & Deletion

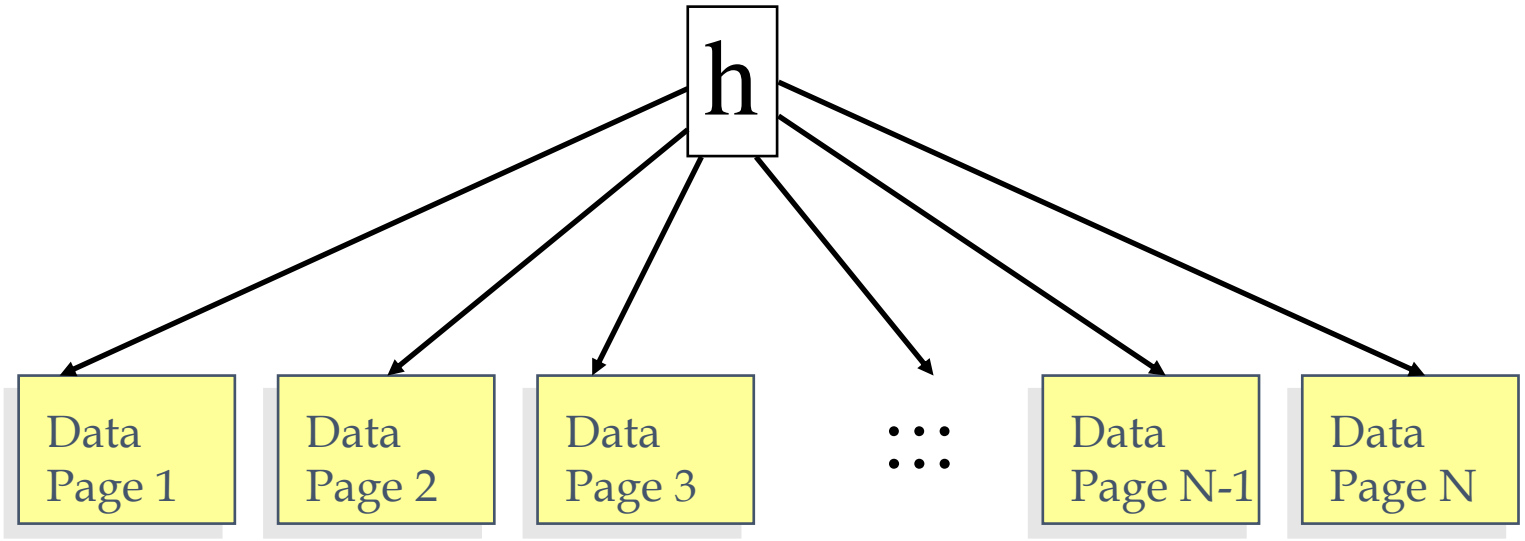
- In open addressing, to find an item with a given key:
  - We probe the locations (same as insertion) until we find the desired item or we reach to an empty location.
- Deletions in open addressing cause complications:
  - We CANNOT simply delete an item from the hash table because this new empty (deleted locations) cause to stop prematurely (incorrectly) indicating a failure during a retrieval.
  - ***Solution***: We have to have three kinds of locations in a hash table: ***Occupied, Empty, Deleted***.
  - A deleted location will be treated as an occupied location during retrieval and insertion.

Hashing Techniques

- Static Hashing
- Dynamic Hashing
  - Linear
  - Extendable

Static Hashing

- With hash based indexing, we assume that we have a function h, which tells us where to place any given record.
- E.g., **page_number = h(value) mod N**, N should be prime

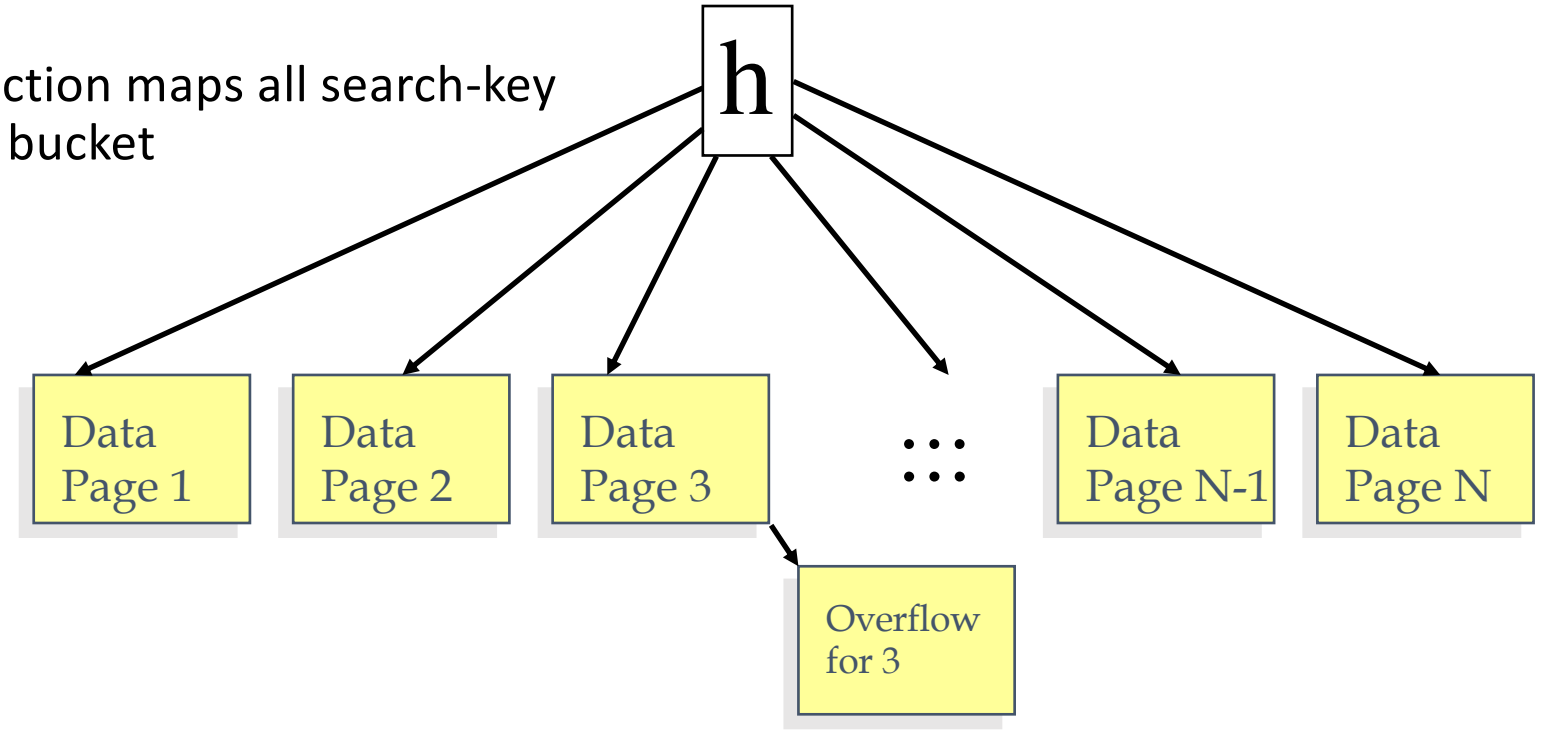


Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Static Hashing: Overflow

- Insertion may cause overflow. The solution is to create chains of overflow pages.
- The worst hash function maps all search-key values to the same bucket



Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

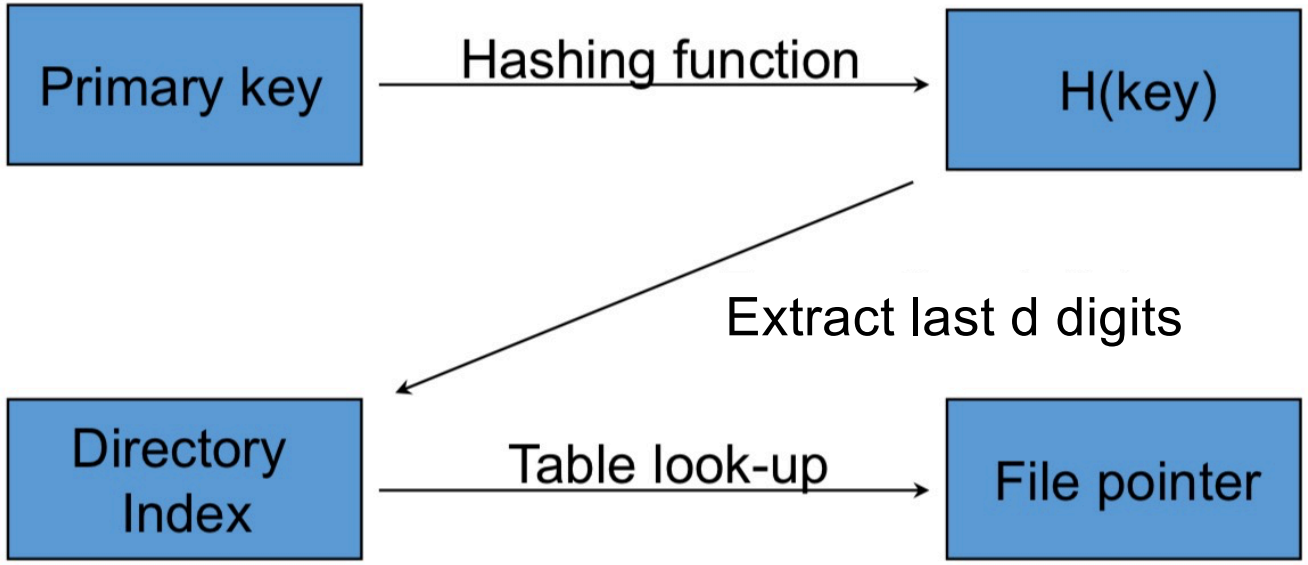
Dynamic Hashing

- Dynamic = Changing number of Buckets B dynamically
- Two methods
  - Extendible (or Extensible) Hashing: Grow B by doubling it
  - Linear Hashing: Grow B by incrementing it by 1
- To save storage space both methods can choose to shrink B dynamically
- Must avoid oscillations when removes and additions are both common.

Extendible Hashing

- ***Idea***: Use *directory of pointers to buckets*
- double # of buckets B by *doubling the directory,* splitting just the bucket that overflowed!
- Directory much smaller than file, so doubling it is much cheaper.
- Only one page of data entries is split. *No overflow blocks.*
- Trick lies in how hash function is adjusted!

Extendible Hashing Overview

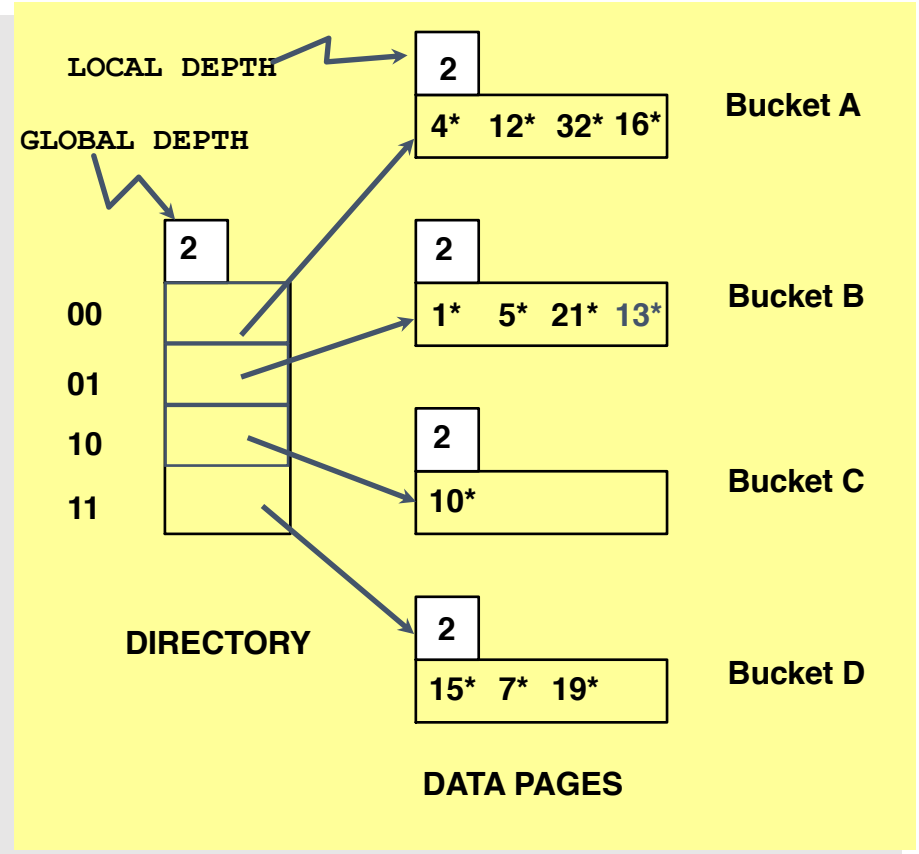


Extendible Hashing

- h(k) maps keys to a fixed address space
- File pointers point to blocks of records known as buckets,
  - where an entire bucket is read by one physical data transfer, buckets may be added to or removed from the file dynamically
- The (last) d bits are used as an index in a directory array containing 2d entries, which usually resides in primary memory
- The value d, the directory size (2d), and the number of buckets change automatically as the file expands and contracts

Example

- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
- *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
- *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.

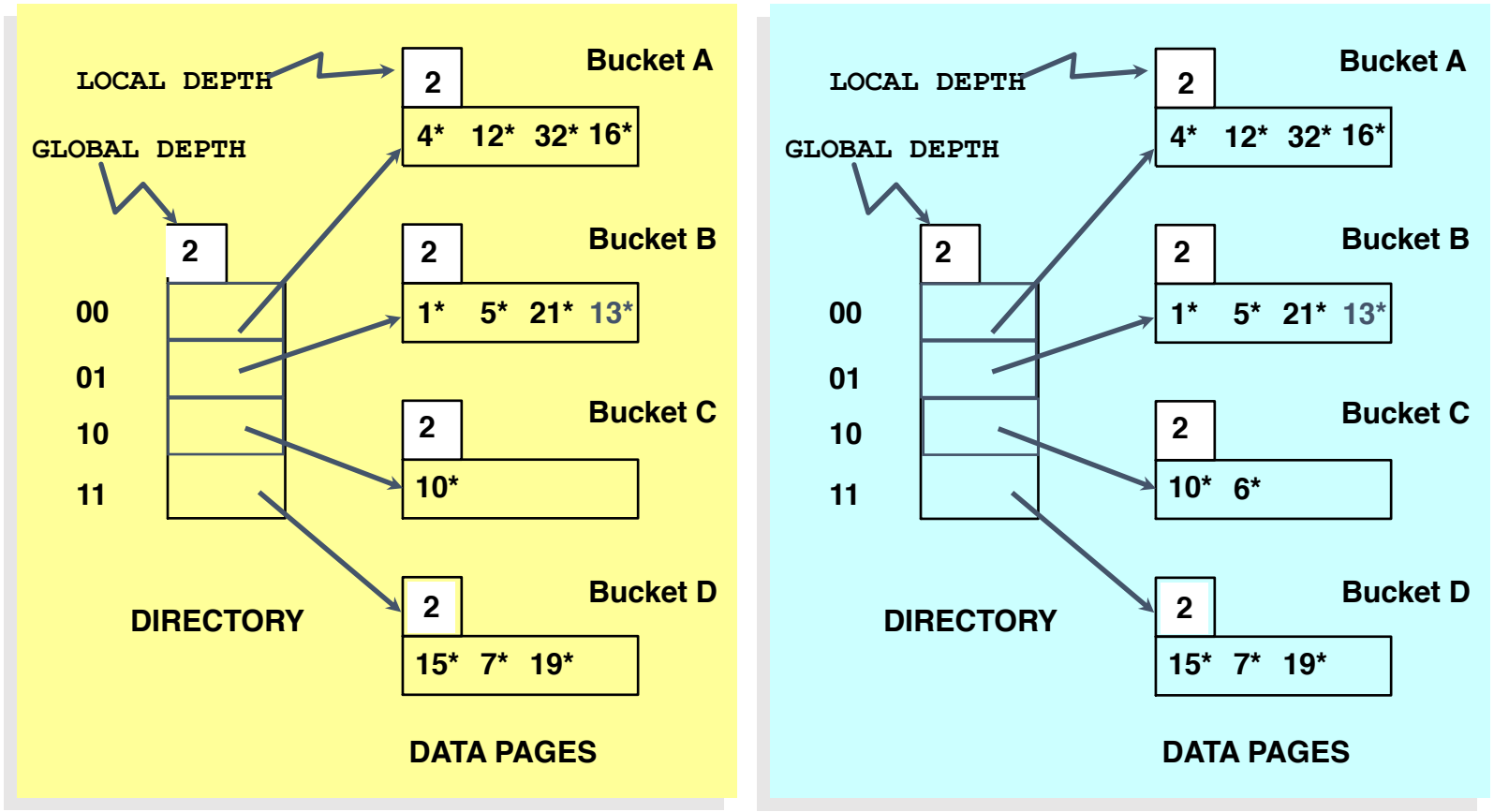


Insert an Item

- Locate the bucket
  - If there is space in bucket, insert the item.
  - If bucket is full, *split* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory.
  - If insert causes *local depth* to become > *global depth*
    - directory is doubled by *copying it over* and `fixing' pointer to split image page.

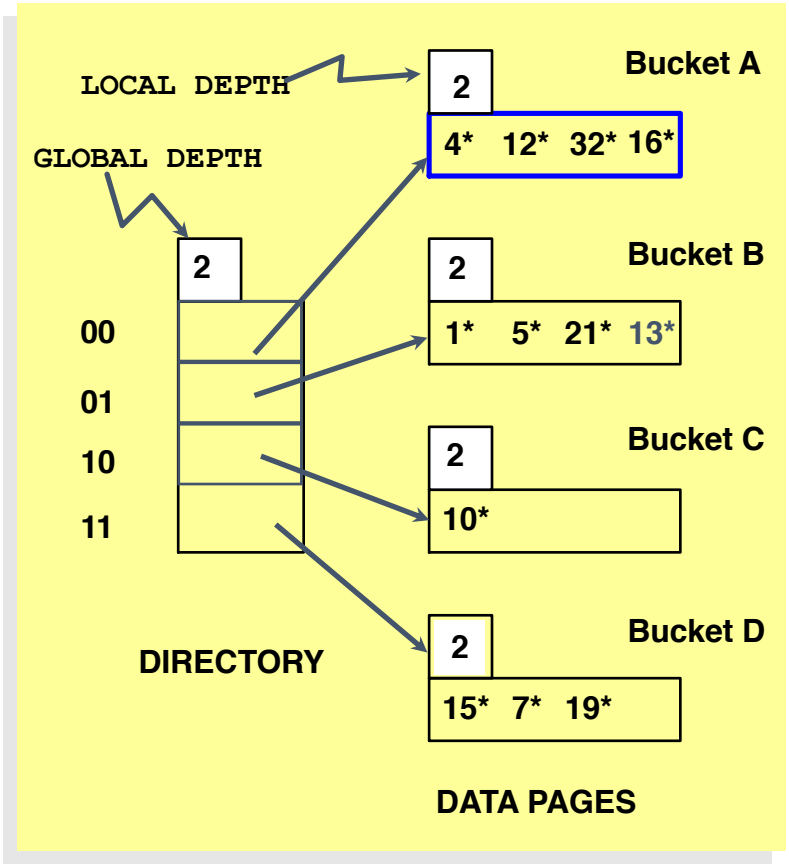
Insert h(r) = 6 (The Easy Case)

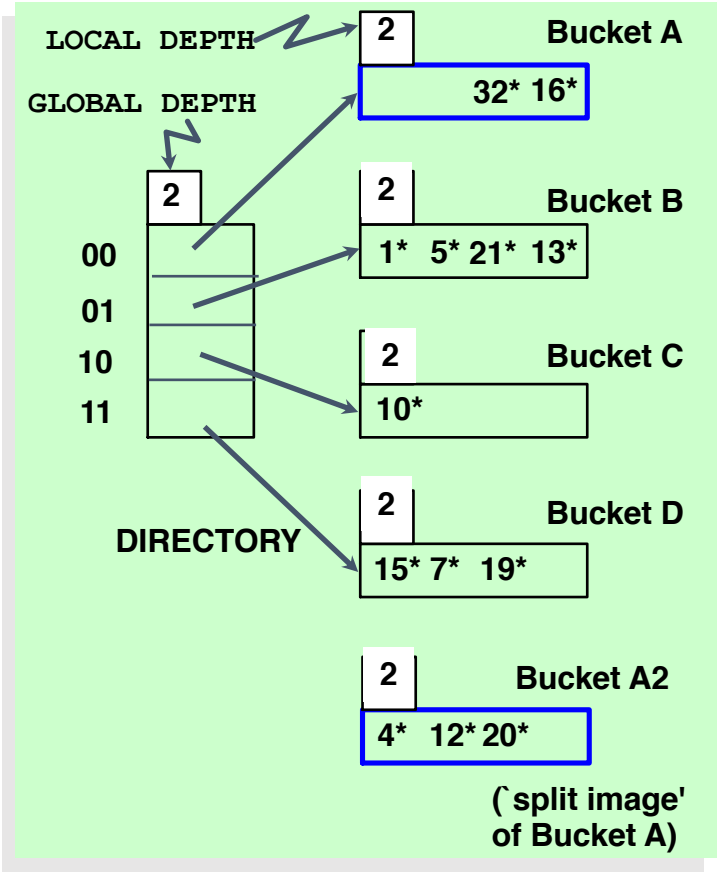
6 = binary 00110



Insert h(r) = 20 (Causes Doubling)

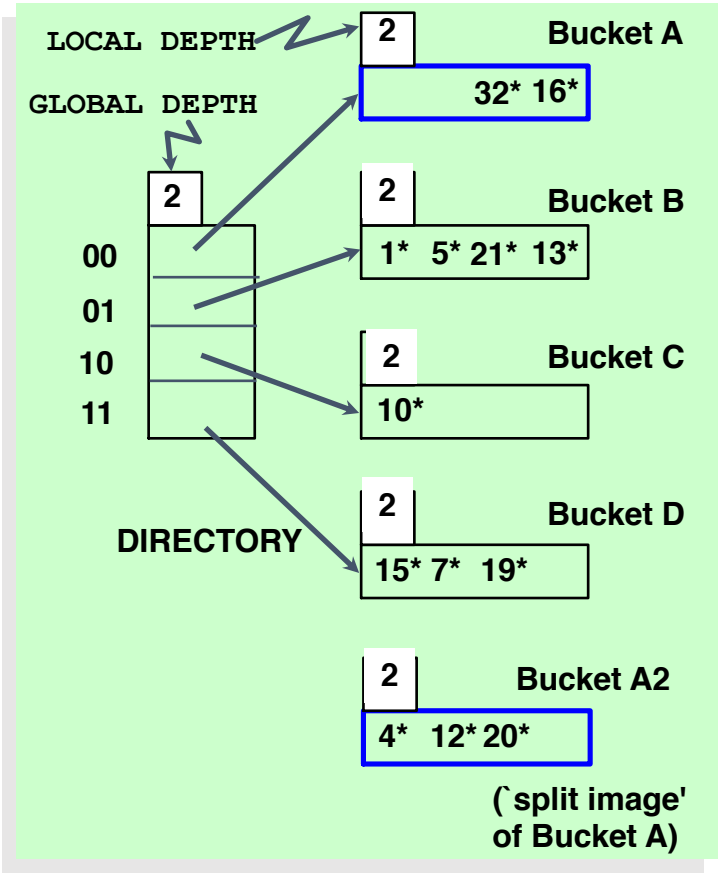
20 = binary 10100

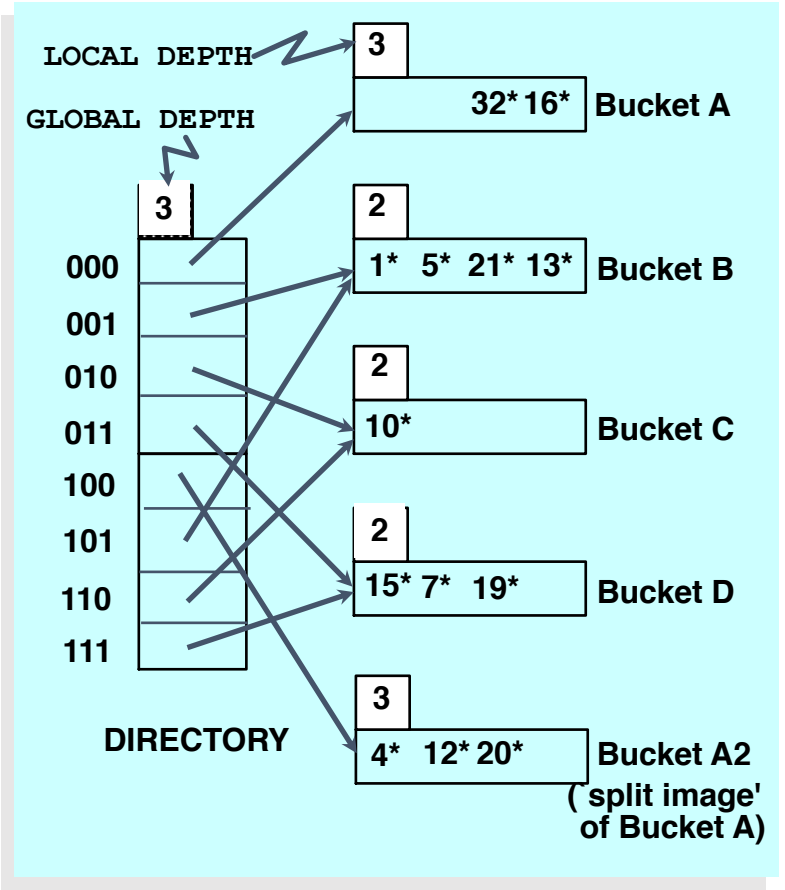




Insert h(r) = 20 (Causes Doubling)

20 = binary 10100





Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.

- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, we can halve directory (this is rare in practice).

Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory.
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where *di* = *d0* + *i*.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Linear Hashing: Bucket Split

- When the first overflow occurs (it can occur in any bucket), bucket 0, which is pointed by p, is split (rehashed) into two buckets:
  - The original bucket 0 and a new bucket m.
- A new empty page is also added in the overflown bucket to accommodate the overflow.
- The search values originally mapped into bucket 0 (using function h0) are now distributed between buckets 0 and m using a new hashing function h1.

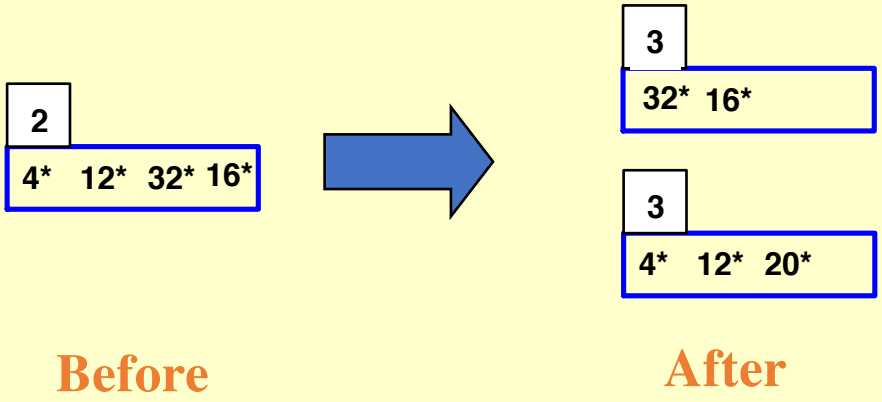
Linear Hashing: Insertion

- Locate bucket to insert
- If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- A split occurs in case of overflow
- Since buckets are split round-robin, long overflow chains don't develop!

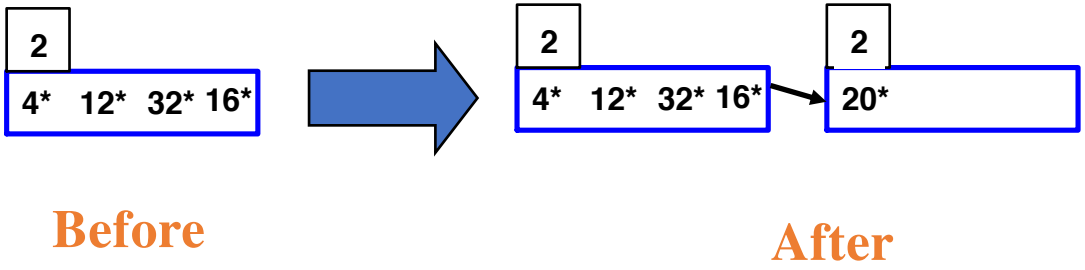
Linear Hashing: Background

Insert 20

We have seen what it means to split a bucket…



We have seen what it means to add an overflow page to a bucket…

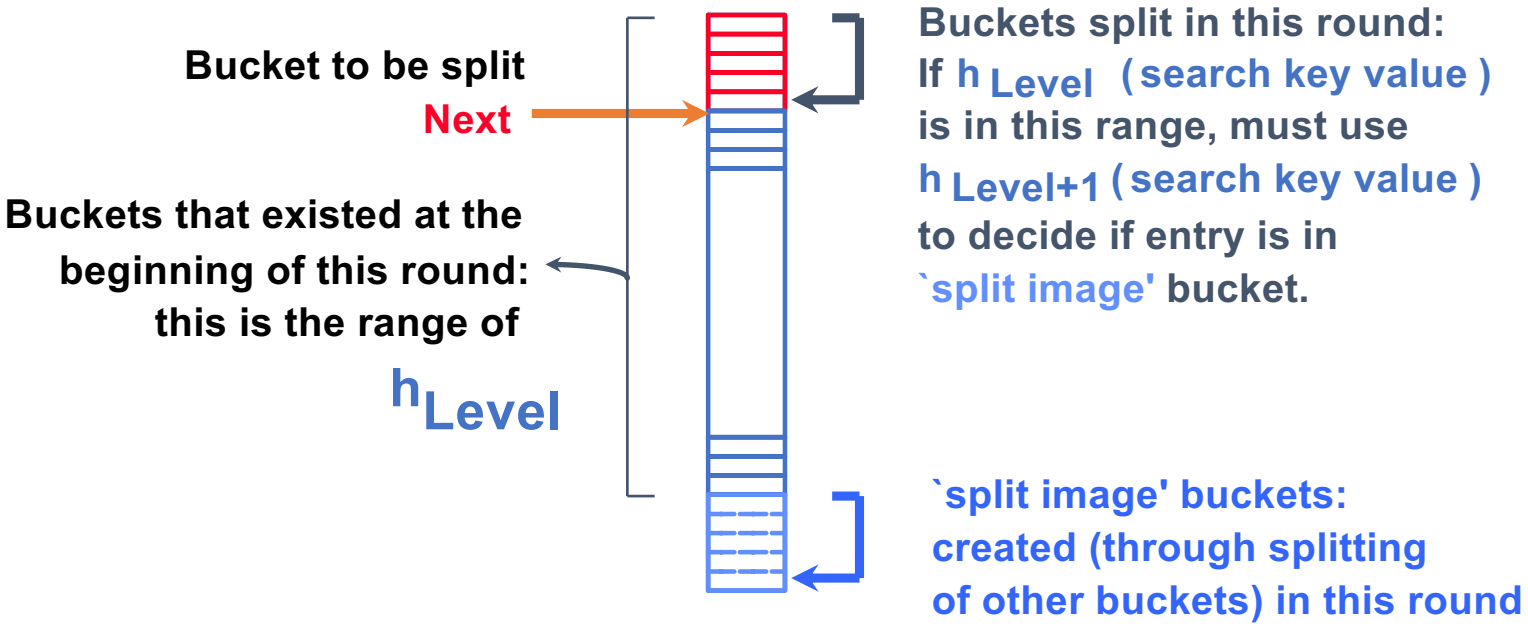


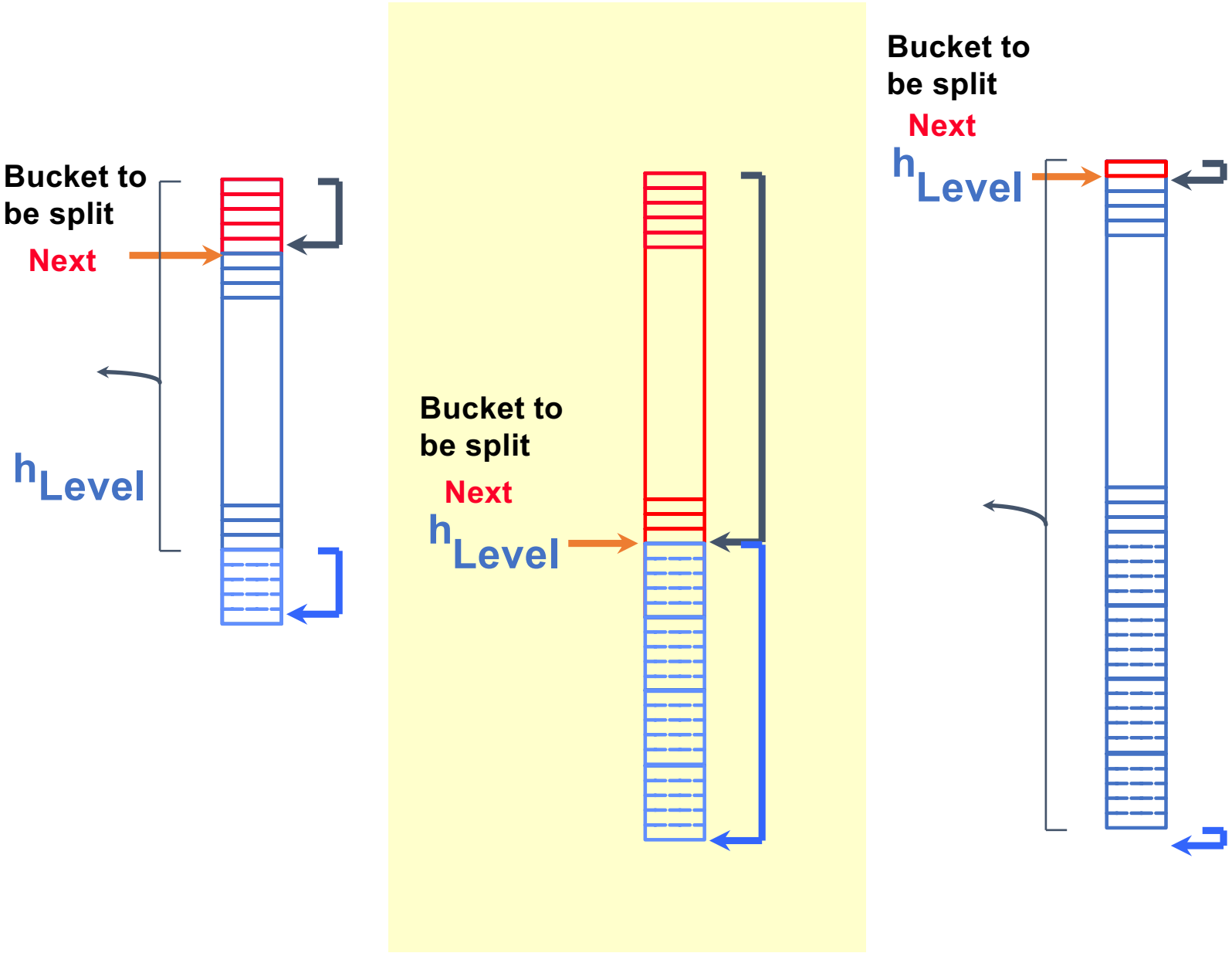
Triggering Splits

- A split performed whenever a bucket overflow occurs is an **uncontrolled** split.
- Let l denote the Linear Hash- ing scheme's load factor, i.e., $l = S / b$ where S is the total number of records and b is the number of buckets used.
- The load factor achieved by uncontrolled splits is usually between 50–70%, depending on the page size and the search value distribution.
- In practice, higher storage utilization is achieved if a split is triggered not by an overflow, but when the load factor l becomes greater than some upper threshold, which is called **controlled** split.

Overview of Splitting as Rounds

- Splits occur in a round robin fashion, i.e., as rounds.

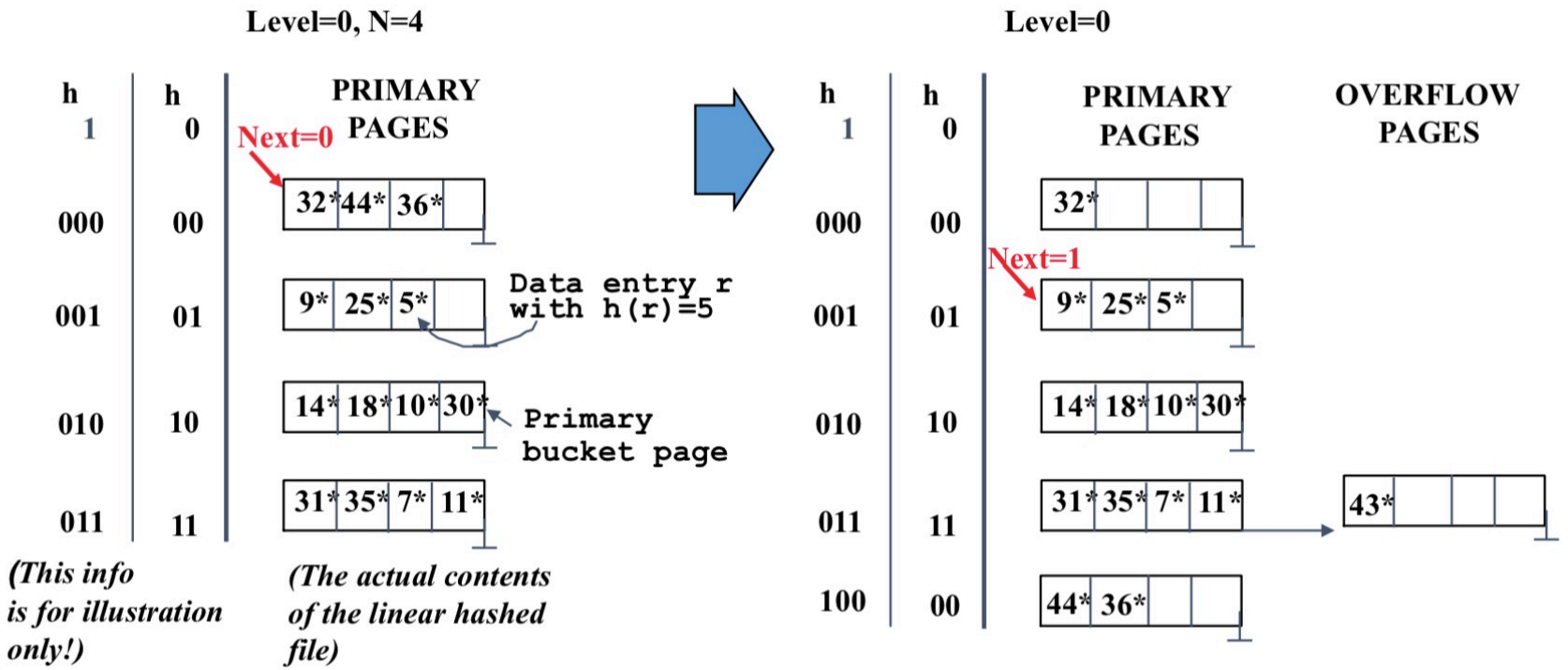



Bucket to
be split
Next
h Level
Bucket to
be split
Next
h Level
Bucket to
be split
Next
h Level

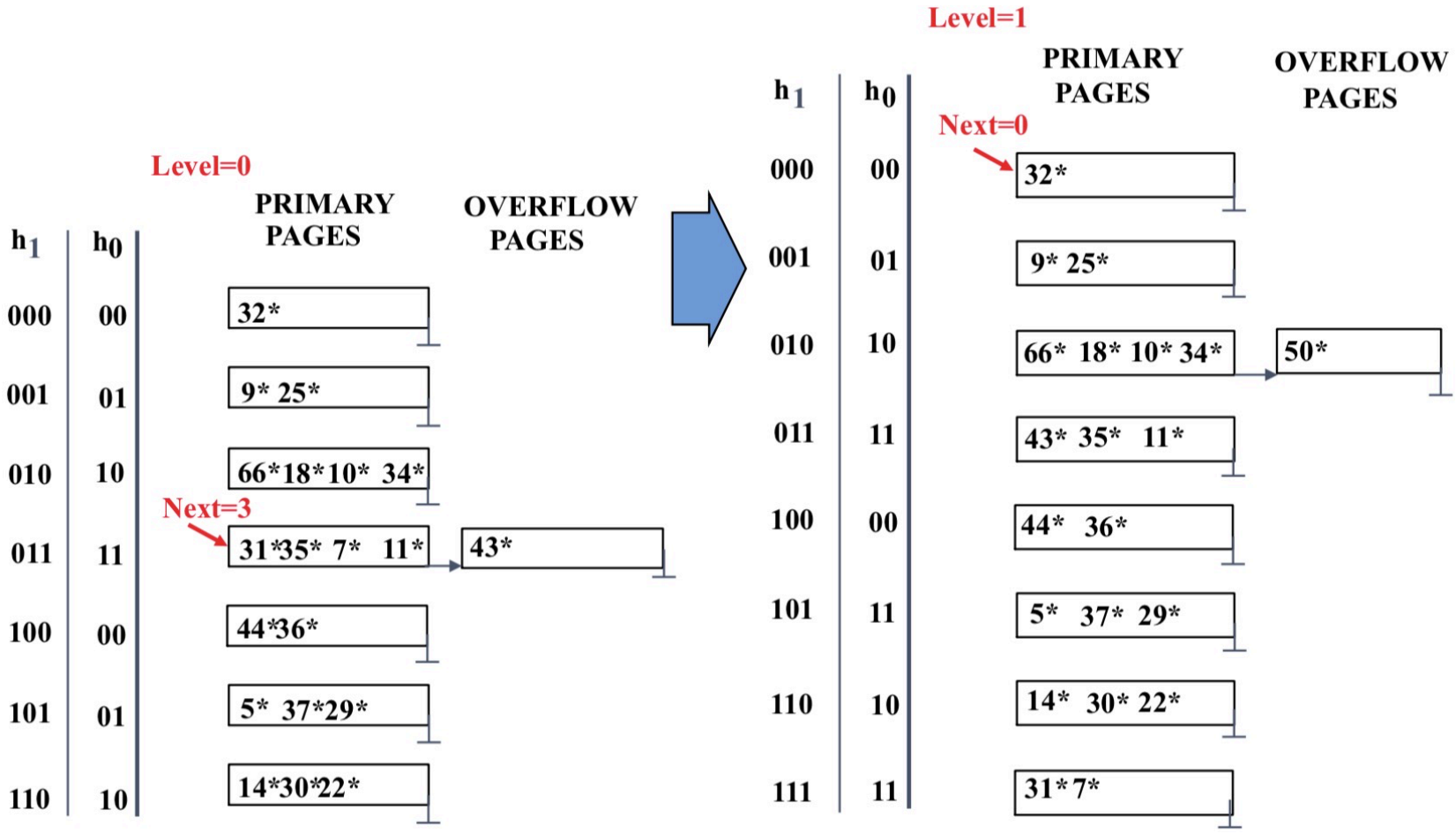
Linear Hashing: Example

On split, $h_{Level+1}$ is used to re-distribute entries.

insert 43 → adds overflow → triggers split → increments Next



Example: End of Round



Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
- Directory to keep track of buckets, doubles periodically.
- Can get large with skewed data; additional I/O if this does not fit in main memory.