CSE462/562: Database Systems (Spring 22)

Lecture 8: Hash Index

2/24/2022

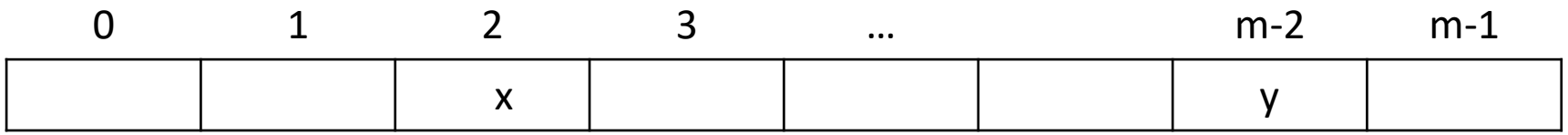
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Hashing basics

- Hash function $h: U \rightarrow [m]$
  - U: key domain, $[m] = \{0,1,2, \ldots m-1\}$
  - *Deterministic*
  - Examples:
    - Multiplicative hashing for integers: $h(x) = \lfloor m \cdot frac(x * a) \rfloor$
      - a: a real number with a good mixture of 0s and 1s
      - $frac(y)$: the fractional part of a real number
      - can be efficiently implemented as $h(x) = \left(\frac{ax}{2^q}\right) \ mod \ m$ for appropriately chosen integers a, q, m
    - String hashing: SHA-1, MD5
      - often available off the shelf
      - can combine a salt to create different hash functions
        - e.g., SHA-1(concat(a, s)) for some randomly chosen string a
      - not that secure, but works for efficiency

Hashing basics

- (In-memory) hash table
  - With a hash function $h: U \rightarrow [m]$

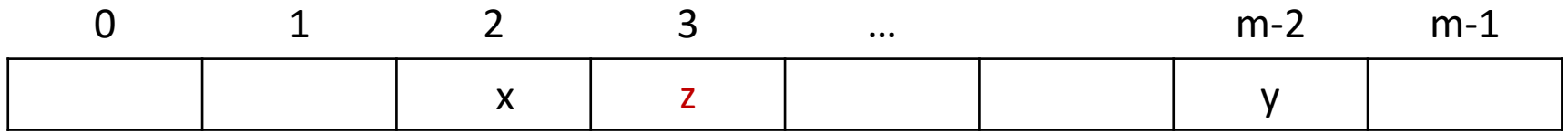


h(x) = 2
h(y) = m-2

Hashing basics

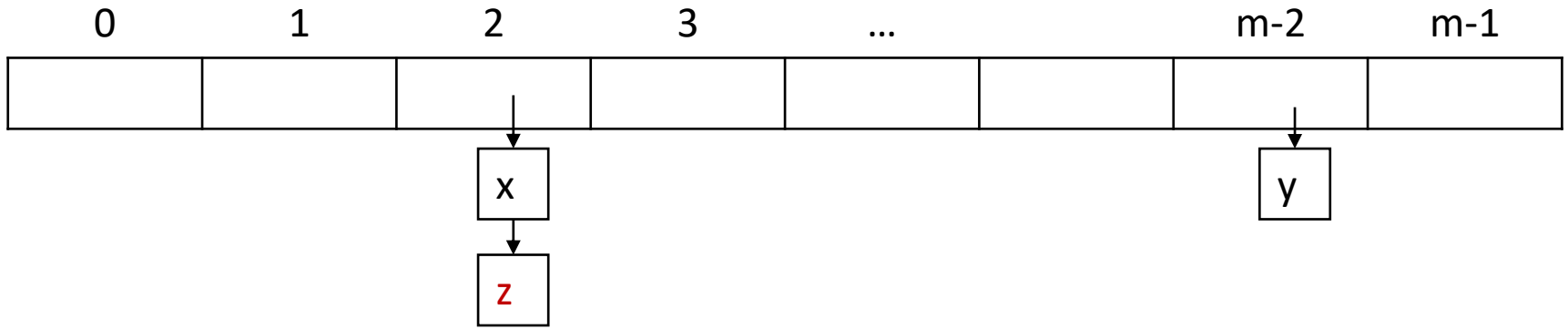
- (In-memory) hash table
  - With a hash function $h: U \to [m]$
  - How to handle collision?
    - Closed hashing vs open hashing
    - Sometimes also called open addressing vs closed addressing

closed hashing with linear probing



h(x) = 2
h(y) = m-2
h(z) = 2

open hashing with linked list



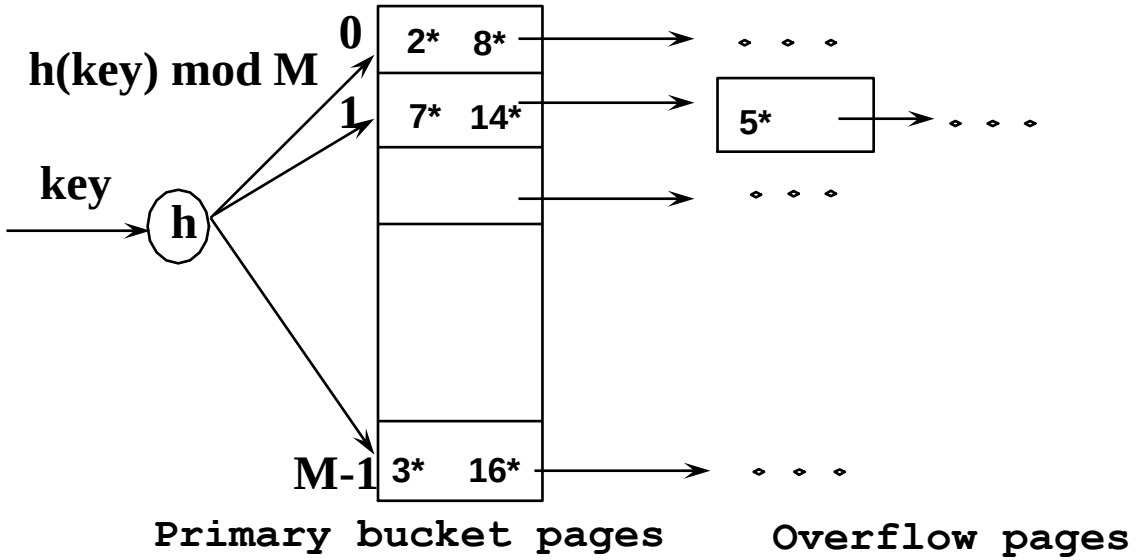
Hash-based index

- Hash-based index are best for equality searches
  - Does not support range searches
- Difference from in-memory hash table
  - Page oriented: multiple data entries per hash bucket
  - Usually don't use closed hashing -- hard to physically delete a data entry
  - Rehashing is very expensive! (reading + writing all the pages)

- Static vs dynamic hashing techniques
  - Static: fixed hash value domain $[m]$
    - Easy to implement, no rehashing overhead
    - Inefficient if number of records is large
  - Dynamic: grow hash value domain over time
    - Sometimes need to rehash, which are expensive
      - How to amortize cost?
    - Scales gracefully with number of records if choice of hash function is good

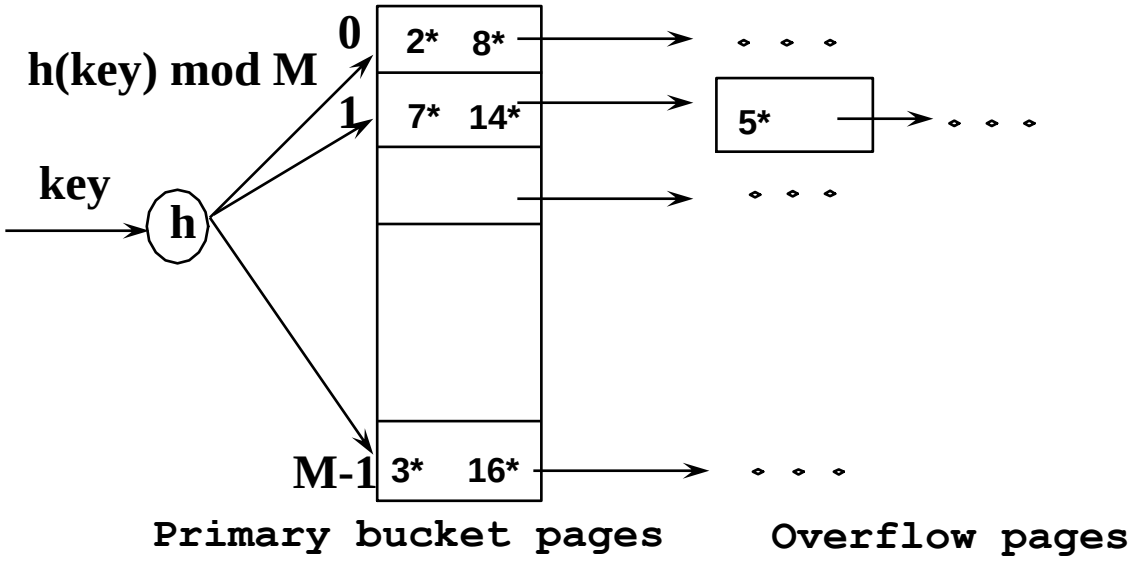
Static hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- $h(k)\ mod\ M$ = bucket to which data entry with key k belongs. (M = # of buckets)



Static hashing

- Buckets contain *data entries*.
- Hash function works on *search key* field(s) of record *r.*
  - Use its value MOD N to distribute values over range 0 ... N-1.
- Long overflow chains can develop over time and degrade performance.
  - ***Extendible*** and ***Linear Hashing***: dynamic techniques to fix this problem.



Extendible hashing

- When the primary page of a bucket gets full,
  - why not doubling the number of buckets and rehash?
  - reading and writing all the pages are very expensive!

- Idea: use *directory of pointers* to buckets (these pointers are page numbers)
  - To double number of buckets, only need to double the directory size
  - Only split the bucket about to overflow. *No overflow page!*
- Why this works?
  - Directory is much smaller than the data files
  - Uses a family of hash functions $h_D: U \rightarrow [2^D]$
    - Trick is how to switch from h_D to h_{D+1} without rehashing for doubling number of buckets

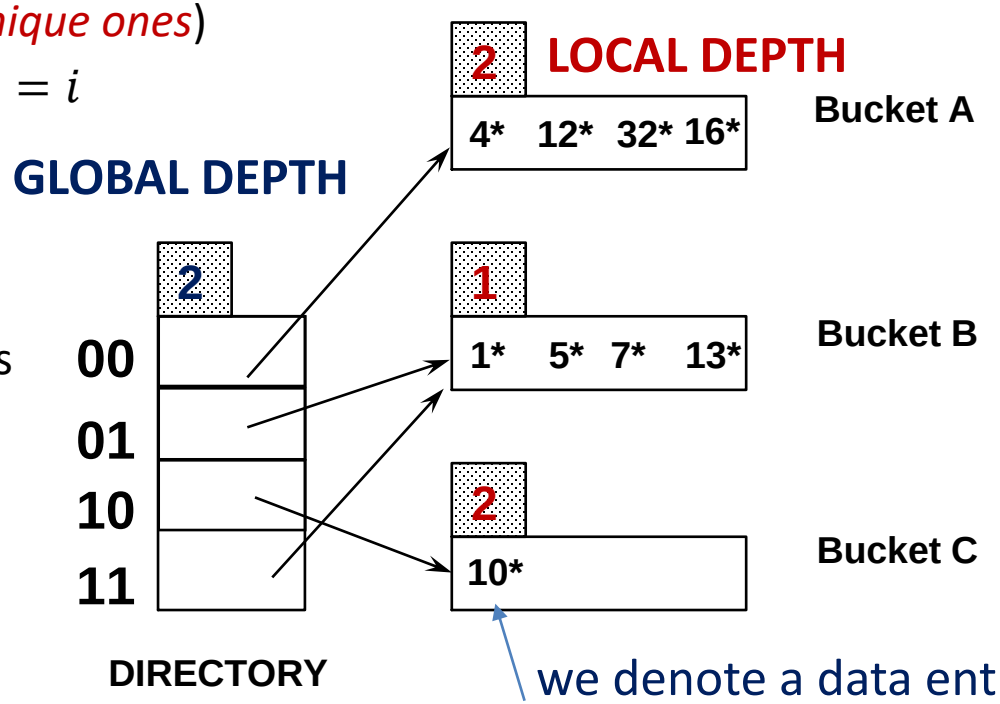
Extendible hashing example

- Hash function $h: U \rightarrow [2^{32}]$ (or $[2^{64}]$ depending on the type of hash value)
  - Define $h_D(k) = h(k)\ mod\ 2^D$ -- therefore $h_D: U \rightarrow [2^D]$
  - Essentially taking *the lowest i bits of the key hash* as the hash value

- Directory: an array of pointers (page numbers) of size 2^D
  - D: global depth
  - Each points to a bucket p_i (*not necessarily unique ones*)
    - A data entry with key k is in p_i iff $h_D(k) = i$

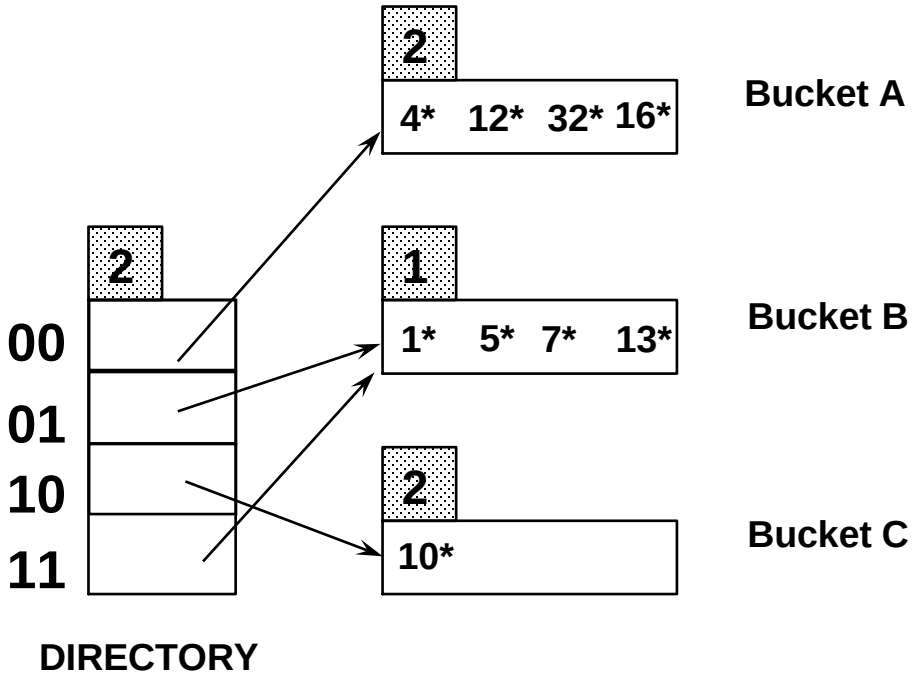
- Each bucket has a local depth d_i
  - Can be used to determine whether this bucket is currently shared by two hash values
    A bucket is not shared iff D == d_i

  - $h_D(k)$ may be different in a bucket
    - Question: what's in common?
    - *$h_{d_i}(k)$ are always the same*



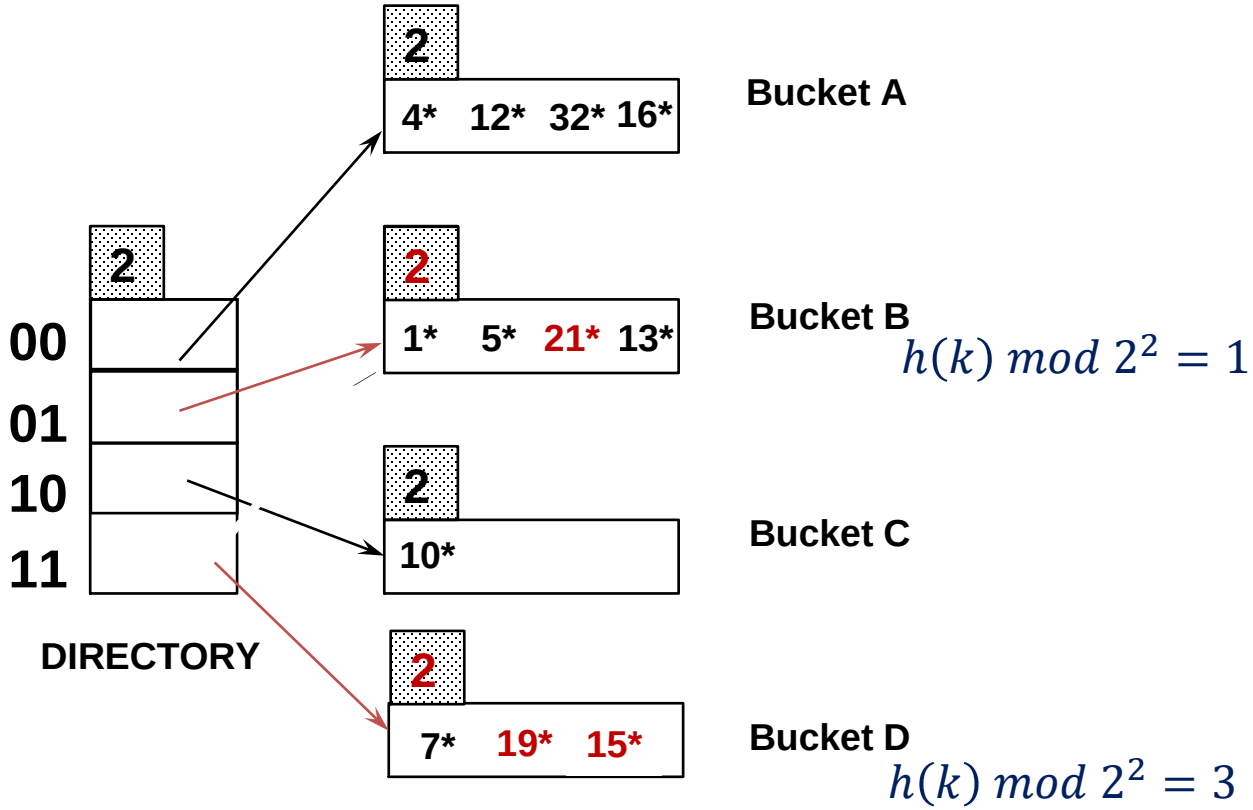
Handling inserts in extendible hashing

- Find the bucket p_i where the insertion belongs
- If there's room, insert it into p_i
- If not, split p_i before insertion
  - increment the local depth d_i
  - allocate a new page with the same (new) local depth
  - redistribute the data entries with the new page
  - double the global depth if local depth is now larger than global depth
    - also duplicate the old directory if global depth is doubled
  - set the pointer for the new page in the directory



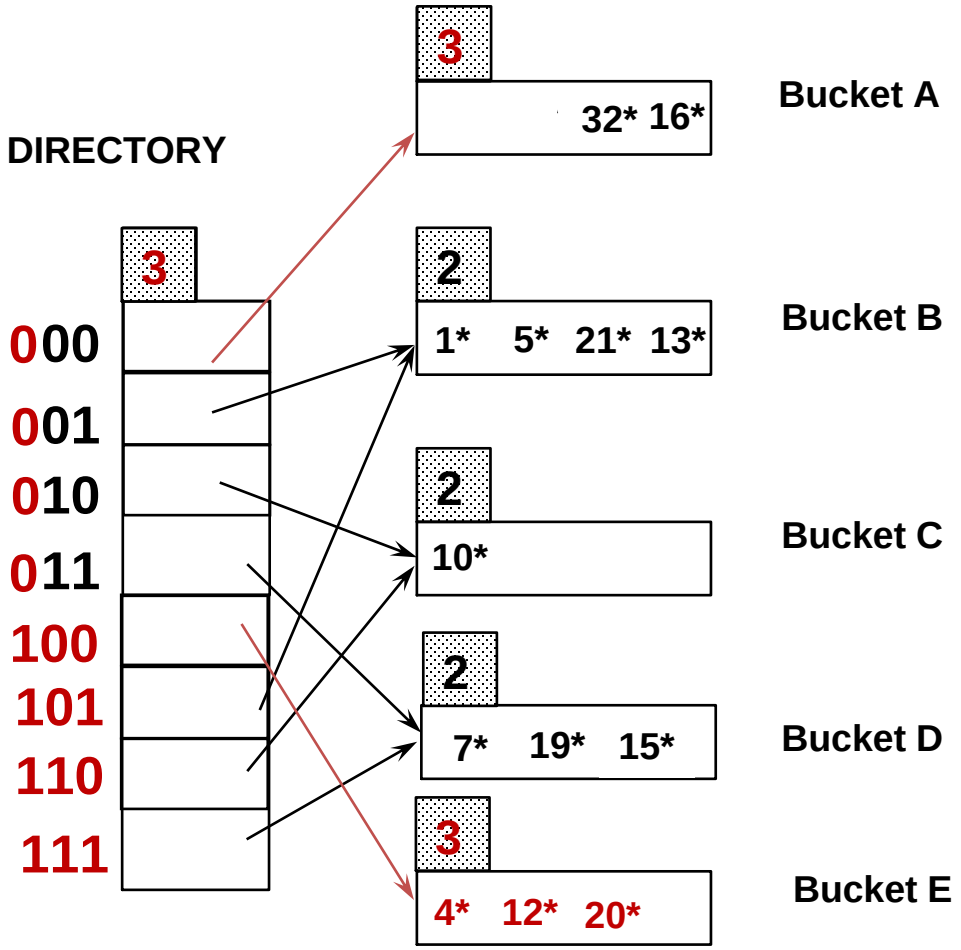
Example: inserting $21^*, 19^*, 15^*$

- $21 = (10101)_2$
- $19 = (10011)_2$
- $15 = (01111)_2$



Example: inserting 20^* (causing doubling)

- $20 = (10100)_2$
- Last **2** bits (00) tell us *20** belongs to A or E
  Last **3** bits needed to tell which.
  - *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.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page.



Notes on extendible hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - If the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
    - *What if we still don't have space after split?*
- **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, can halve directory.

Linear hashing

- Another dynamic hashing scheme that handles long overflow chains without using a directory.

- Page to split is chosen in a *round-robin* fashion, not where it will overflow
  - LH allows using *temporary* overflow pages
  - If the hash values are reasonably uniform -- overflows will be resolved quickly

Handling insertion in linear hashing

- Also uses a family of hash functions
  - $h_i(k) = h(k)\ mod\ (2^i N)$
  - Initial size N does not need to be power of 2
- Proceeds in "rounds".
  Current round number is called level $l\ >=\ 0$
- There are $N_l = N * 2^l$ buckets at the beginning
  - *Next* initially set to 0
  - Invariant:
    - Buckets [0, Next) has been split in this round
    - Buckets [Next, N_l) are to be split in this round
- On insert
  - If the bucket for insertion is full
    - Add an overflow page and insert the data entry
    - Split Next bucket and increment Next
  - Use h_{l+1} to redistribute entries for a split bucket
- Round ends when $Next = N_l$
  - Start a new round, $Next \leftarrow 0, l \leftarrow l + 1$

$N\ = 4, l = 0$

Next = 0

h_1	h_0
000	00
001	01
010	10
011	11

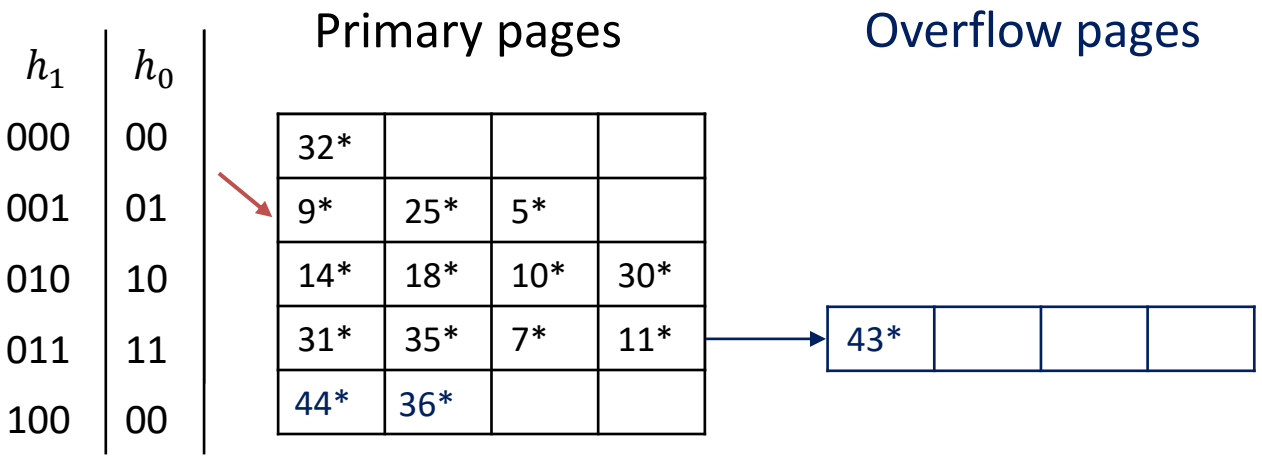
Primary pages

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

Example: insert 43^* $(101011)_2$

$N = 4, l = 0$

Next = 1

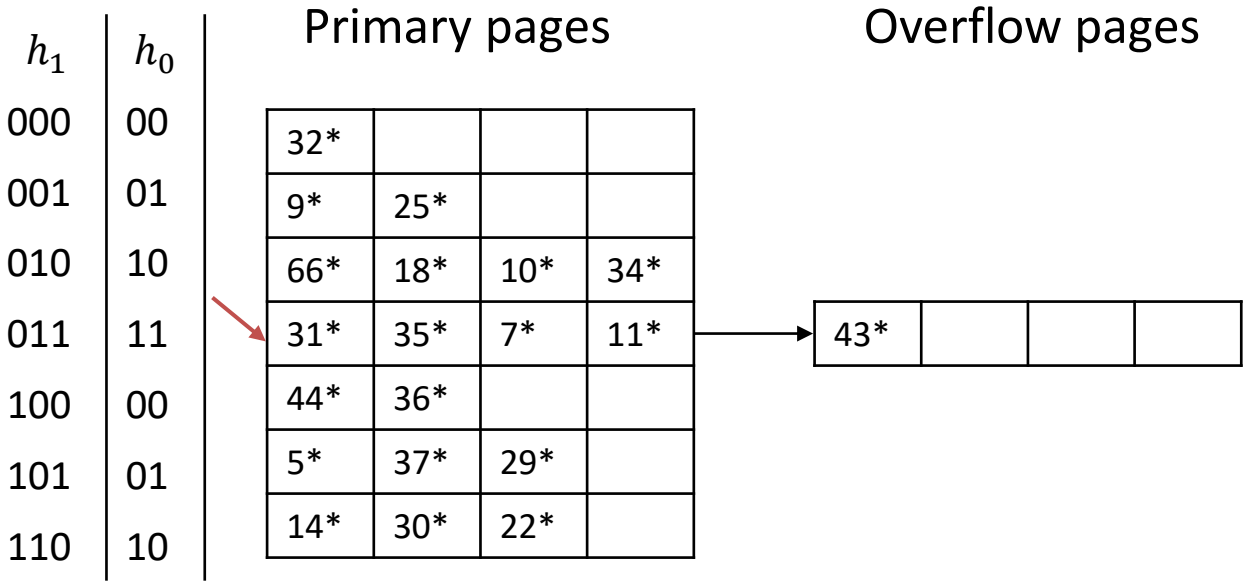


Example: end of a round

Insert 50^* $(110010)_2$

$N = 4, l = 0$

Next = 3

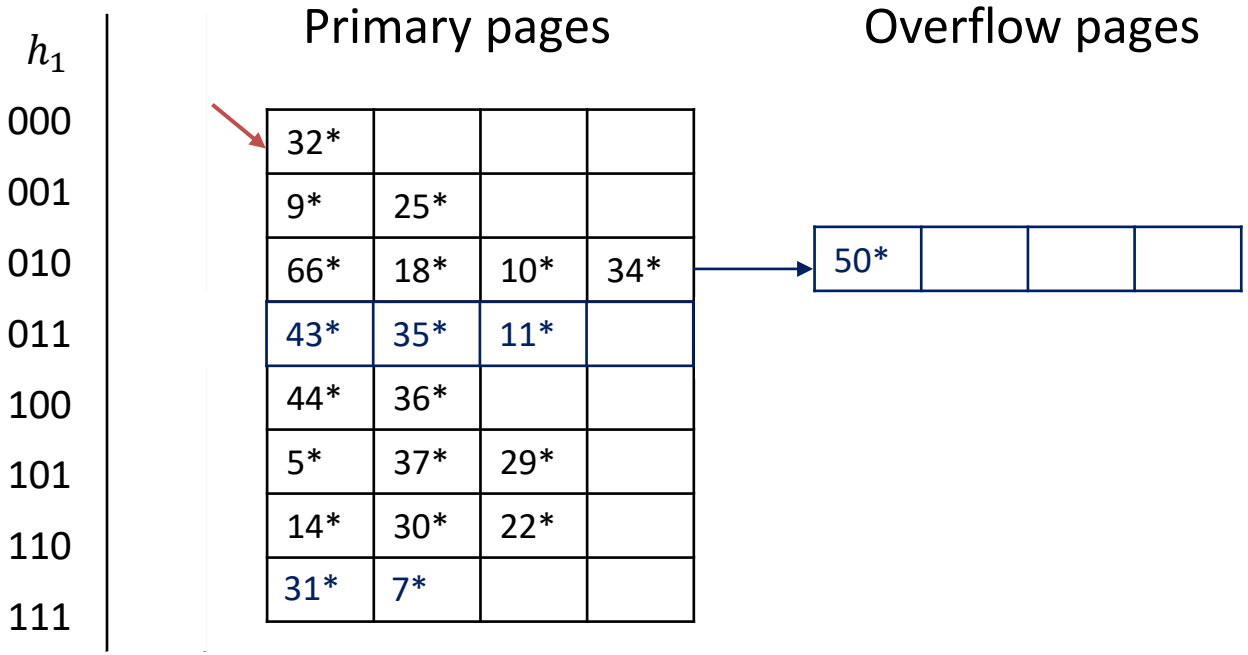


Example: end of a round (cont'd)

Insert 50^* $(110010)_2$

$N = 4, l = 1$

Next = 0



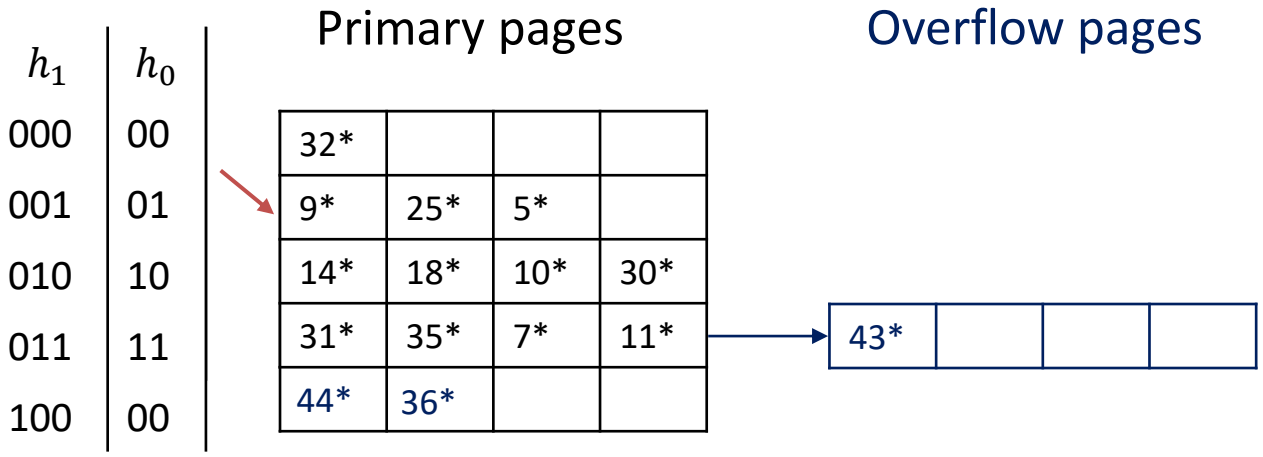
Linear Hashing Search Algorithm

- To find the bucket for a data entry k^*
  - Compute $h_l(k) = h(k)\ mod\ (2^l N)$
    - If $h_l(k) \geq Next$
      - Bucket $h_l(k)$ is the bucket for k^* (because it hasn't been split in this round)
    - Otherwise,
      - k^* could belong to either bucket $h_l(k)$ or bucket $h_l(k) + 2^l N$
      - Compute $h_{l+1}(k)$ to find out

$$N = 4, l = 0$$

Next = 1

Find 32^*?
Find 14^*?
Find 43^*?



Notes on linear hashing

- If hash values are skewed
  - because of key skew or bad hash function
  - *then will still have long overflow chains*
- Delete: the reverse of insertion algorithm

Summary

- This lecture
  - Hashing basics
  - Static hashing
  - Dynamic hashing
    - Extendible hashing
    - Linear hashing

- Next lecture
  - How to choose a good hash function
  - Hash-based sketches