Hashing

Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

Improve Retrieval Efficiency

- Sort the list in a specific order before searching
- Approaches
  - Sort after a batch of insertion
    - Insertion time should be small
    - Chance of retrieval is rare
  - Insertion based on some sorting policy
    - Retrieval time should be small

Indexing

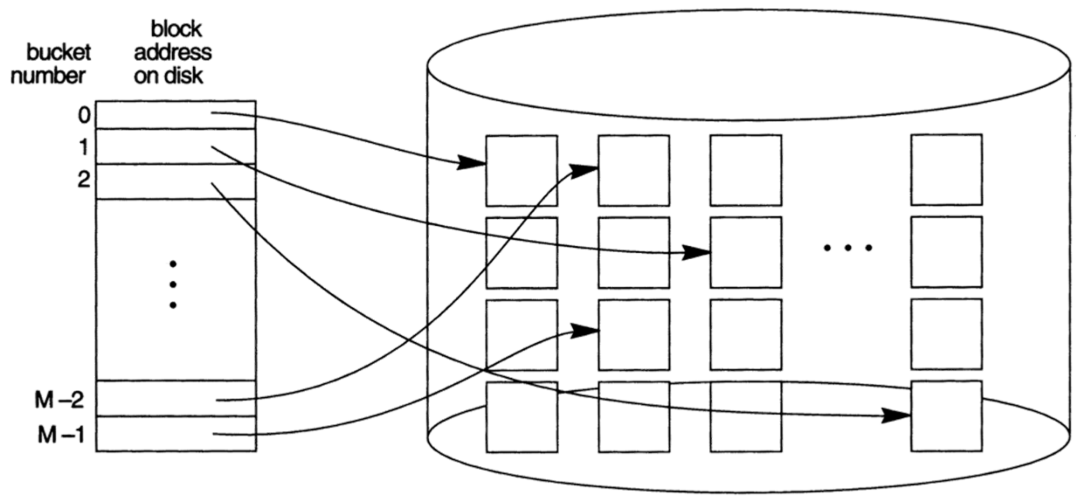
- Balanced (binary) search tree
  - Get, Insert and Delete take O(logn)
- Hashing
  - Get, Insert and Delete take O(1)
  - Static hashing
  - Dynamic hashing

Static Hashing

[redacted]

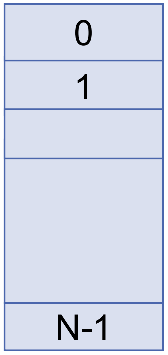
Overview of Hashing

- The file blocks are divided into M equal-sized *buckets*
- The record with hash key value K is stored in bucket I
  - i=h(K), and h is the *hashing function*



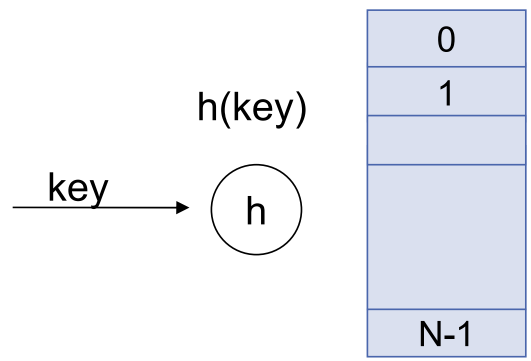
Hash Table

- Hash table (ht)
  - A container stores dictionary pairs
- Hash table is partitioned into b buckets
  - ht[0], ht[1], …, ht[b-1]
  - Each bucket holds s dictionary pairs (slots)
    - Usually s=1 which means each bucket can hold exactly one pair



Hash Function

- The *hash* (*address*) of the pair is determined by a **hash function**, h(k)
  - Hash function maps keys into buckets by returning an integer in the range 0 through b-1



Definitions

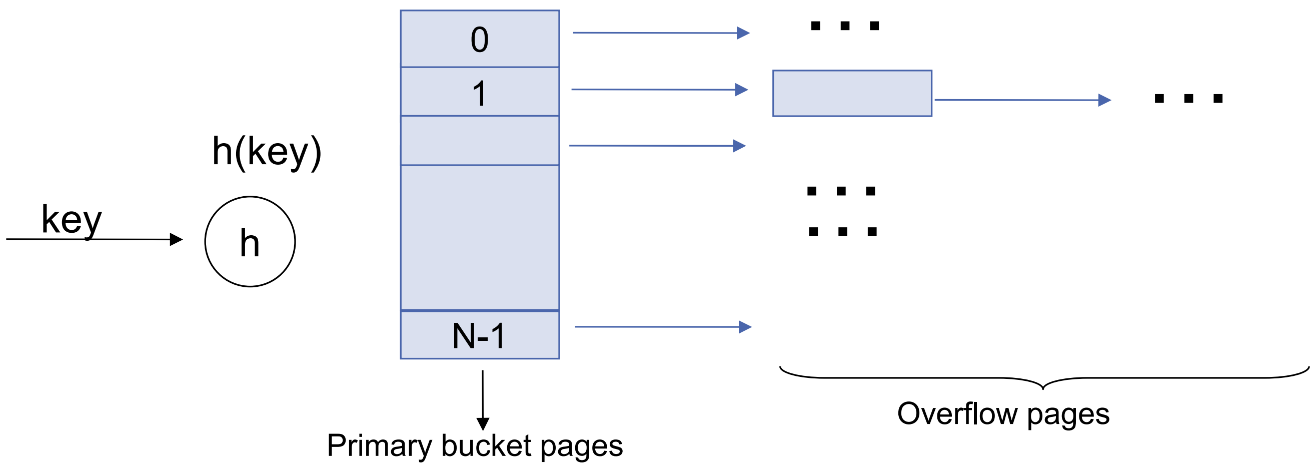
- Key density (*n*/T)
  - *n*: # of pairs in the table
  - T: Total # of possible keys
- Loading density or loading factor ($\alpha = n/(s \times b)$)
  - s: # of pairs in a bucket, b: # of buckets
- Two keys, k_1 and k_2, are said to be *synonyms* w.r.p.t. h, if $h(k_1)=h(k_2)$.

Collisions

- Many keys might be mapped to the same home bucket
- Collision
  - When a key is mapped to a non-empty home bucket
- Overflow
  - When a key is mapped to a full home bucket
- Overflow and collision occur simultaneously when each bucket has 1 slot.

Collisions (Contd.)

- A new record hashes to a bucket that is already full
  - An overflow file is kept for storing such records
  - Overflow records that hash to each bucket can be linked together



Hashing Properties

- If # of slots is small, all operations (search, insert and delete) can be performed in O(1)
- Using leading letter is not a good hash function
  - Keys might bias toward certain buckets
- A good hash function should be
  - Easy to compute
  - Result few collisions

Uniform Hash Function

- A hash function that dose not result in a biased use of the hash table for random keys
- Given a key *k* chosen at random, the probability that *h(k)=i* to be *1/b* for all buckets *i*
- Four popular hash functions
  - Division
  - Mid-Square
  - Folding
  - Digit Analysis

Division

- $h(k) = k \% D$
- Keys are non-negative integer
- The home bucket is obtained by using the modulo (%)
- Bucket address range from 0 to D-1
  - The hash table must have at least b=D buckets
- Using a prime number for D (see textbook)

Mid-Square

- Mid-Square:
  - Squaring the keys
  - Use an appropriate number of bits from the middle of the squared key as bucket address

- If r bits is used, the size of the table is 2^r
  - If there are 64 buckets (2^6), we need middle 6-bits to determine the bucket address

Folding

- The key is partitioned into several parts
- These parts are added together to obtain the key address

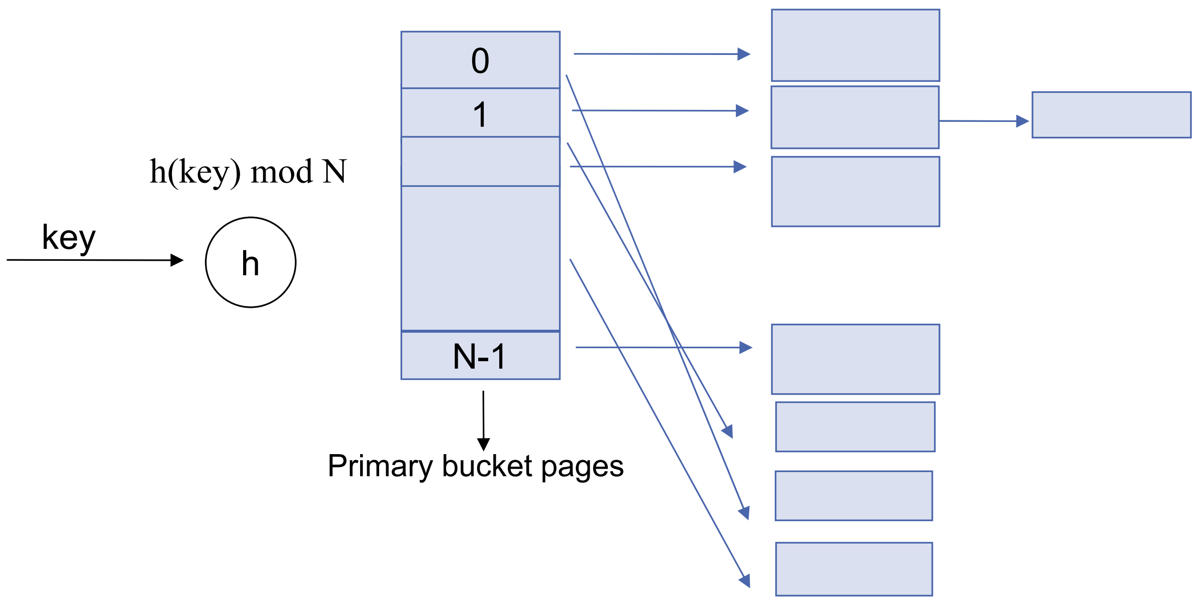
Digit Analysis

- All the keys in the table are known in advance
- Digitals having the most skewed distributions are deleted
- Employ the remaining digits

Dynamic Hashing

Dealing Overflow Problems

- Add overflow pages
- Double the size of the buckets
- Double the number of the buckets and reorganize
- ?



Dynamic Hashing

- Also called Extendable Hashing
- Use the *binary representation* of the hash value h(K) in order to access a *directory*

Directory

- An array of size 2^d where d is called the *global depth*
- Can be stored on disk
- Expand or shrink dynamically
- Entries point to the disk blocks
  - That contain the stored records
  - When an insertion in a disk block that is full
  - The block split into two blocks
  - The records are redistributed among the two blocks
- Updated appropriately

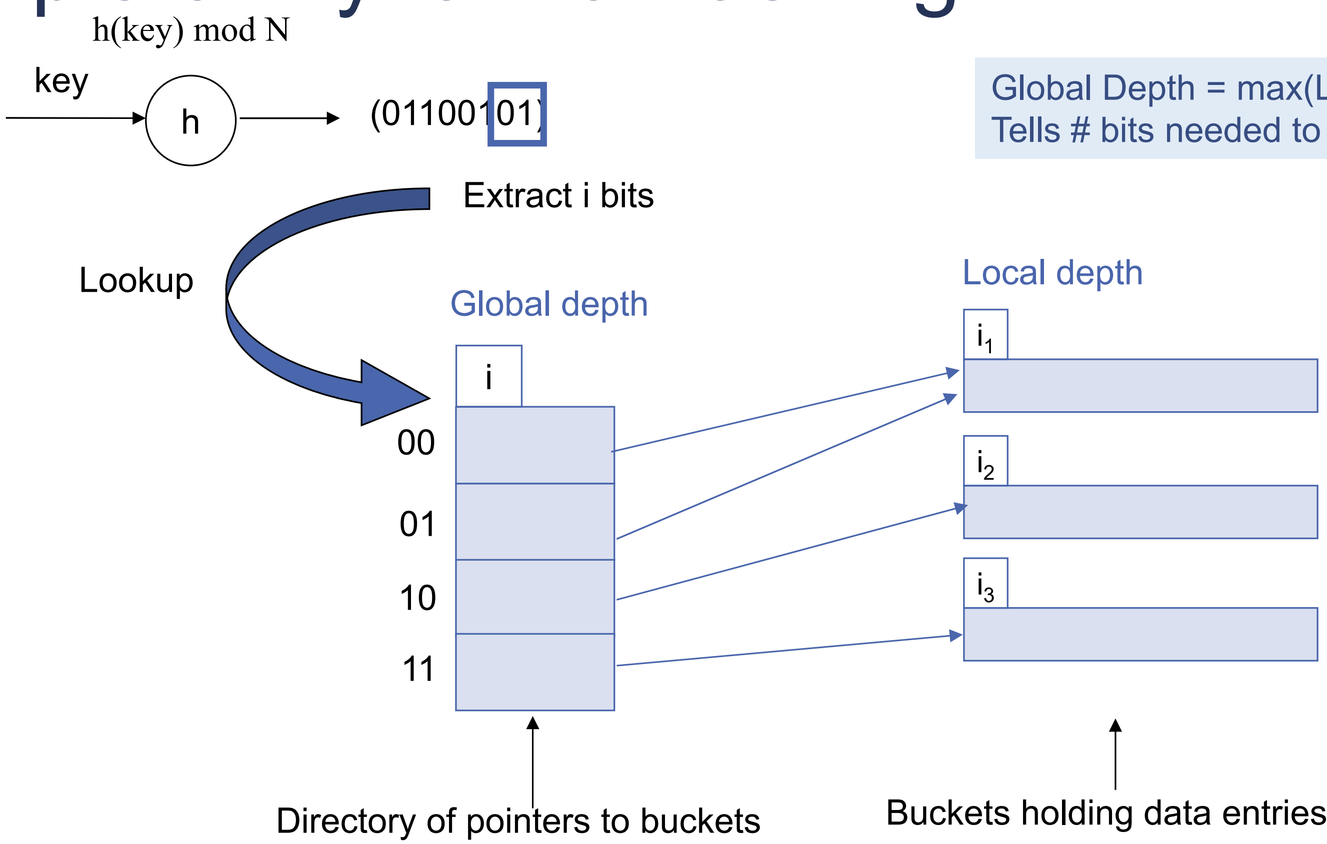
Motivation

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?

- *Idea*: Use *directory of pointers to buckets*
  - Double #buckets by *doubling the directory*
  - *S*plitting just the bucket that overflowed!

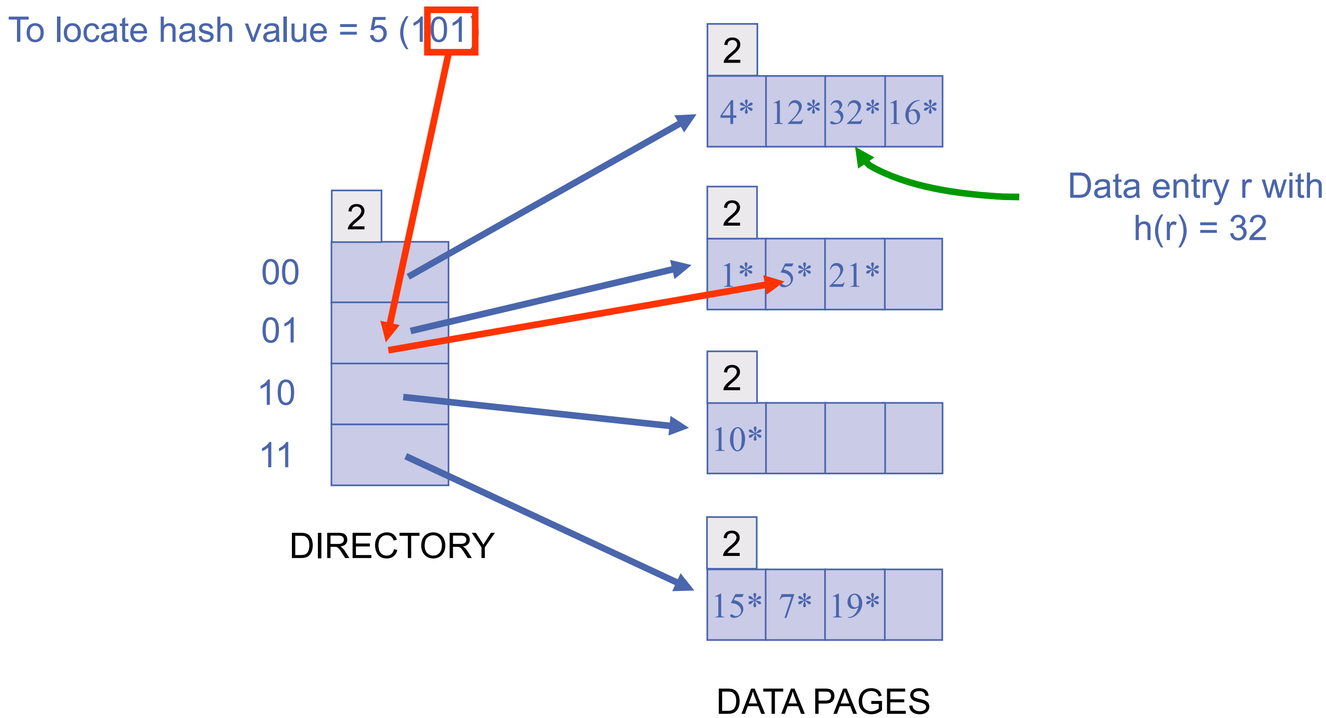
- Directory much smaller than file
  - So doubling it is much cheaper
  - Only one page of data entries is split. *No overflow page*!

Example of Dynamic Hashing

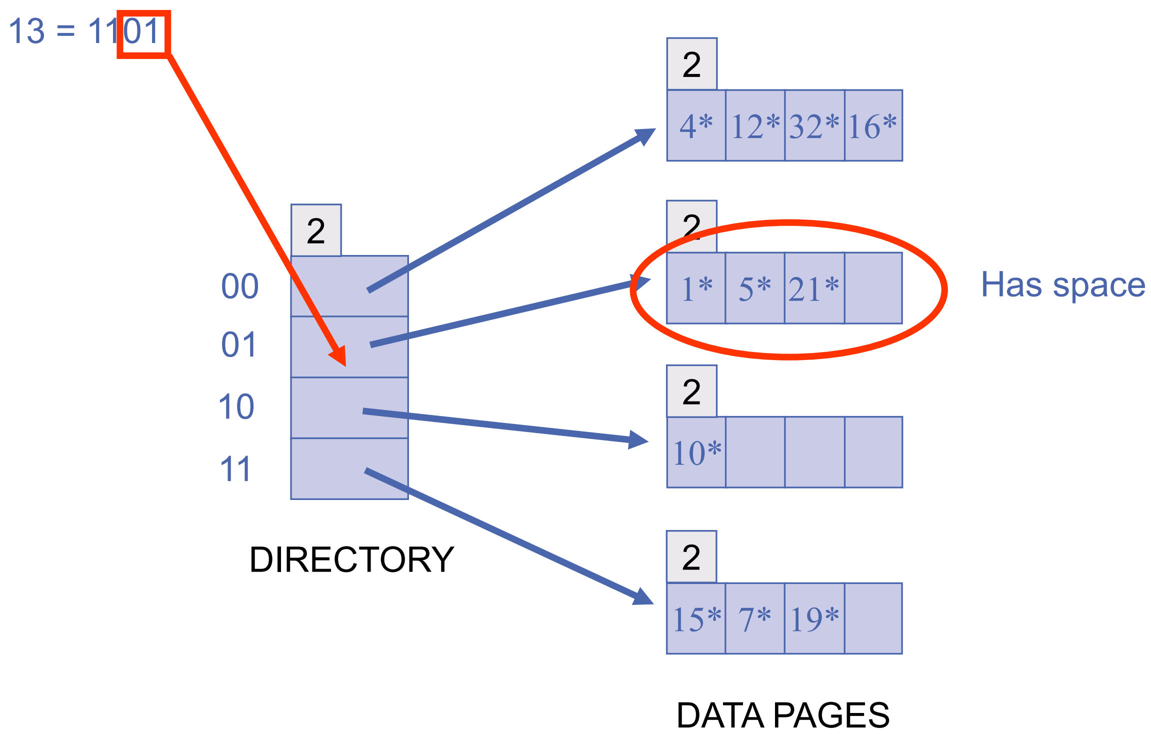


Global Depth = max(Local Depth of all buckets)
Tells # bits needed to determine the address

Dynamic Hashing: Example

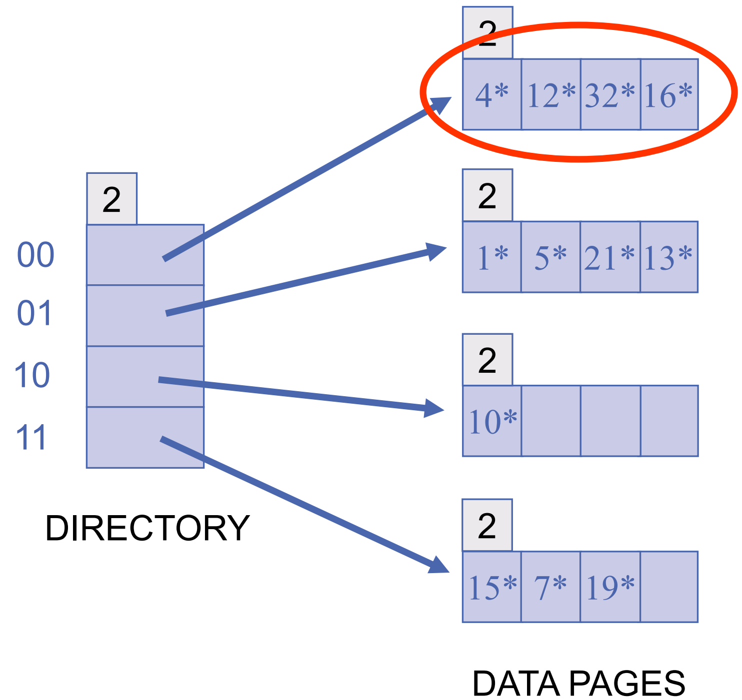


Dynamic Hashing: Insert 13*



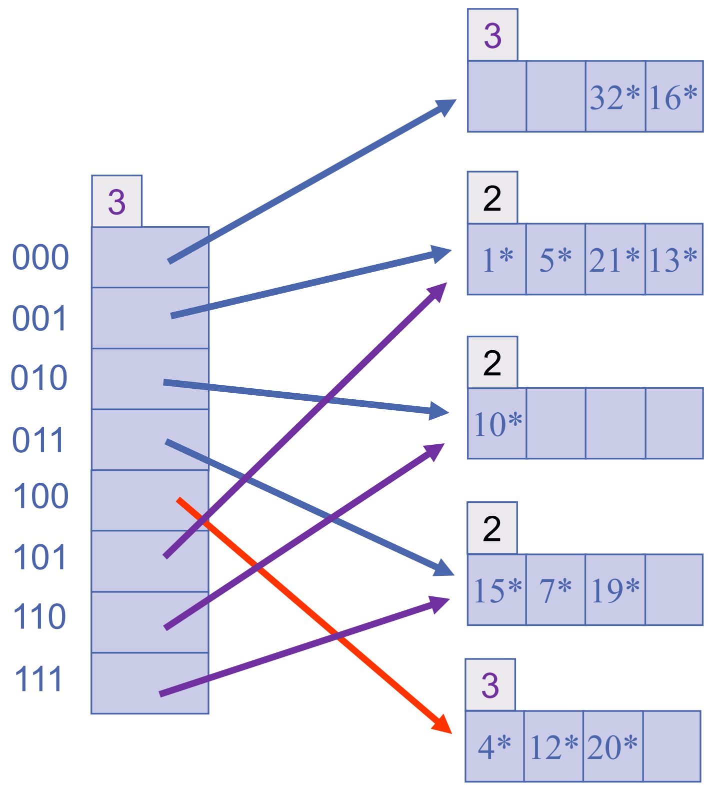
Dynamic Hashing: Insert 20*

20 = 10100



Full, need split. Consider the last three bits

Dynamic Hashing: Insert 20*



Local/Global Depth

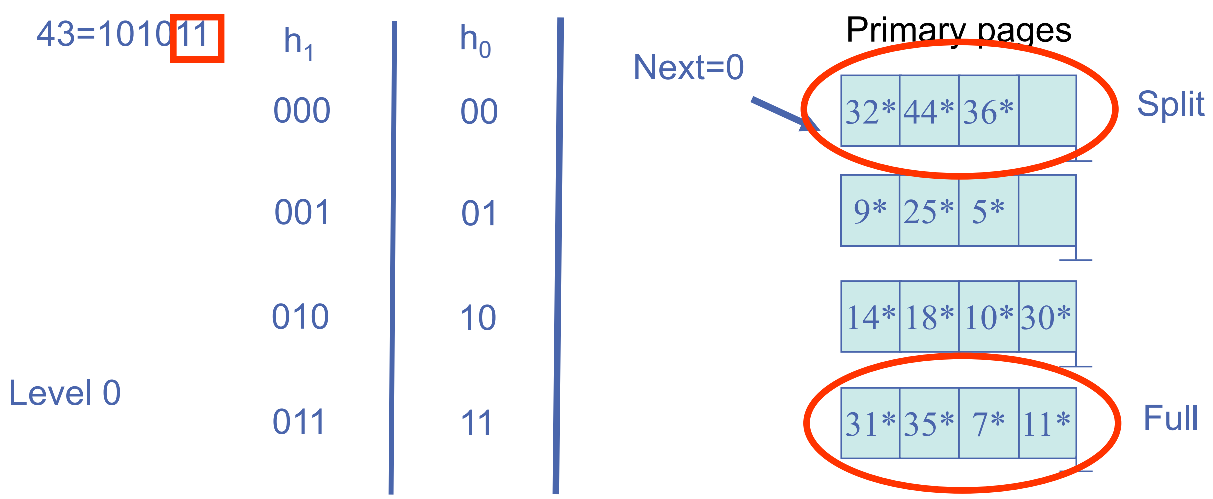
- Initially, all local depths are equal to global depth
  - # of bits need to express the total # of buckets
- While the process of split, if a bucket whose local depth = global depth
  - The directory must be doubled
- Global depth + 1 when the directory doubles
  - Local depth + 1 when a bucket is split

Linear Hashing

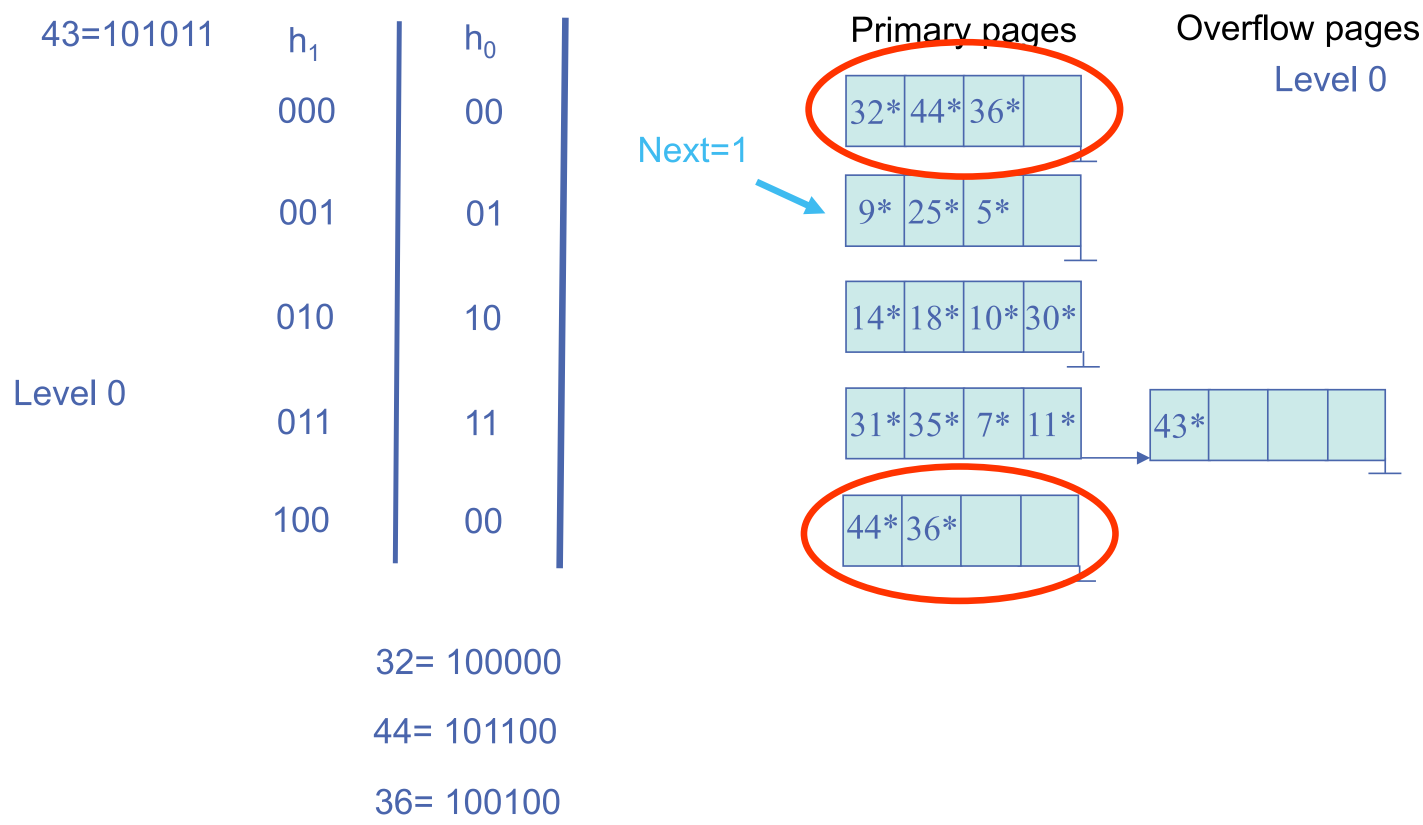
Linear Hashing

- Requires an overflow area
- Blocks are split in linear order
- Round round-robin fashion
  - eventually all buckets are split
- Idea: Use a family of hash functions h0, h1, h2, ...
  - $h_i(key) = h(key) \bmod (2^i N)$; N = initial # buckets
  - If $N = 2^{d0}$, for some d_0, h_i checks the last d_i bits, where $d_i = d_0 + i$.
  - h_{i+1} doubles the range of h_i

Linear Hashing: Insert 43*



Linear Hashing: Insert 43*



Linear Hashing: Insert 29*