MC 302 – DBMS: Hashing

Goonjan Jain
Department of Applied Mathematics
Delhi Technological University

Outline

- hashing
- extendible hashing
- linear hashing
- Hashing vs B-trees

Hashing

- Primary file organization
- Very fast access to records on search condition
- Search condition must be on a single field – hash field or hash key
- Hash function/ randomizing function – applied on hash field, yields address of disk block

Static Hashing

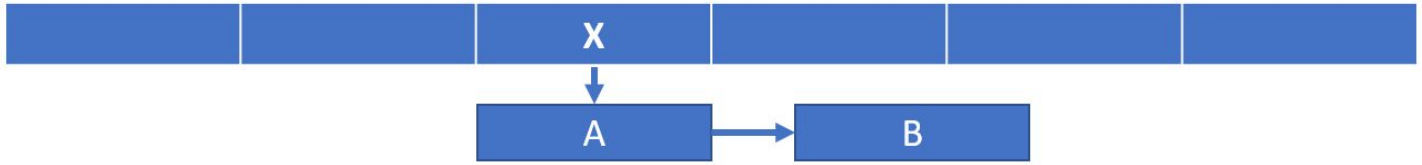
- Fixed number of buckets
- Drawback for dynamic files
- Types:
  - Internal Hashing
  - External Hashing

Internal Hashing

- For internal files
- implement a hash table using array of records
- M slots addressed as 0 to M-1
- Choose a hash function, transform hash field into an integer from 0 to M-1
- Most common hash function h(k) = k mod M
- Problems:
  - No guarantee that different values will hash to different addresses

Collision

- Hash field value hashed to an address already occupied
- Collision resolution: find another position
- Collision Resolution Techniques:
  - Open addressing:
    - Starting from the hashed address, check subsequent positions till an unused position is found
  - Chaining:
    - Place new value in an unused overflow location
    - Set a pointer from the address to the overflow location



  - Multiple Hashing:
    - Apply a second hash function

External Hashing

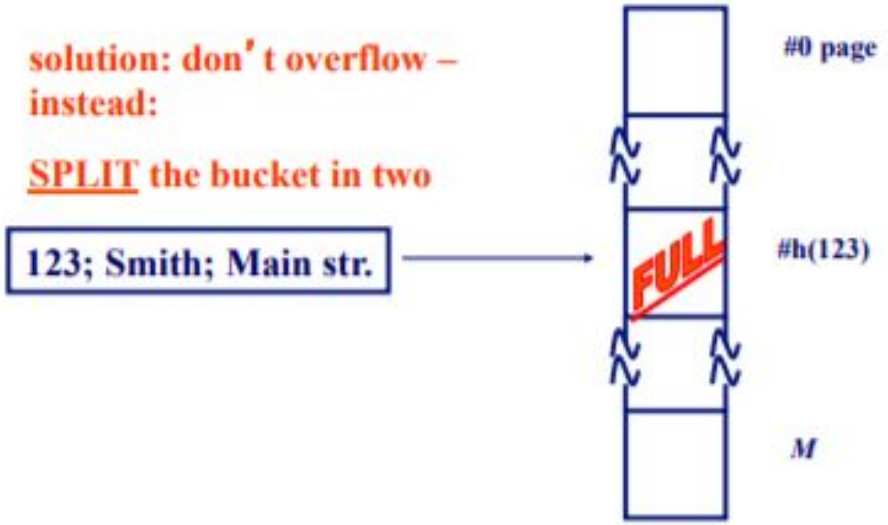
- Target address space is made of buckets,
  - each bucket can hold multiple values
- Bucket is
  - 1 disk block, or
  - Cluster of contiguous blocks
- Hash function maps key to a relative bucket number
- Collision problem is less severe
- If a bucket if full, a variation of chaining can be used-
  - Bucket points to record pointers – block address and record position

Problem with (static) hashing

- Overflow
- Underflow
- Sol: Dynamic Hashing

Dynamic Hashing

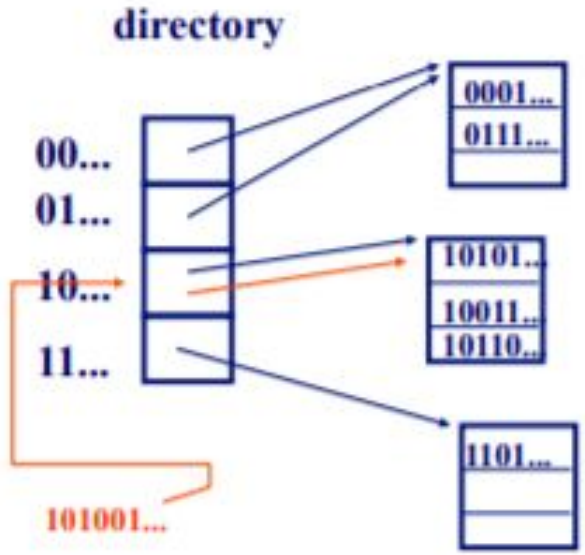
- idea: shrink / expand hash table on demand..
- ..dynamic hashing
- Details: how to grow gracefully, on overflow?
- Many solutions – One of them: ‘extendible hashing’



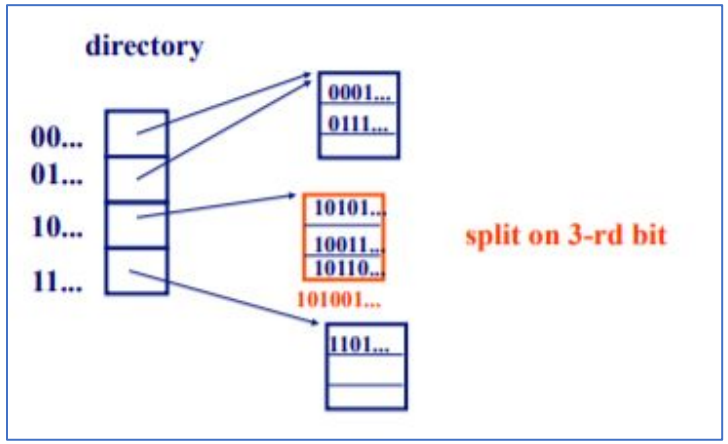
Extendible Hashing

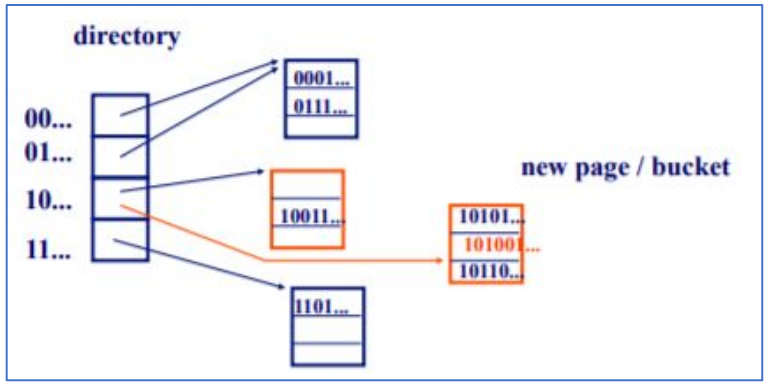
- keep a directory, with pointers to hash-buckets
- Uses-
  - An array of 2^d buckets. d is the global depth
  - Directory
- First d bits of hash value is used as index for directory entry
- Entry in director determines the bucket address
- Each bucket stores
  - Local depth d' – number of bits on which bucket contents are based
- Q: how to divide contents of bucket in two?
- A: hash each key into a very long bit string; keep only as many bits as needed

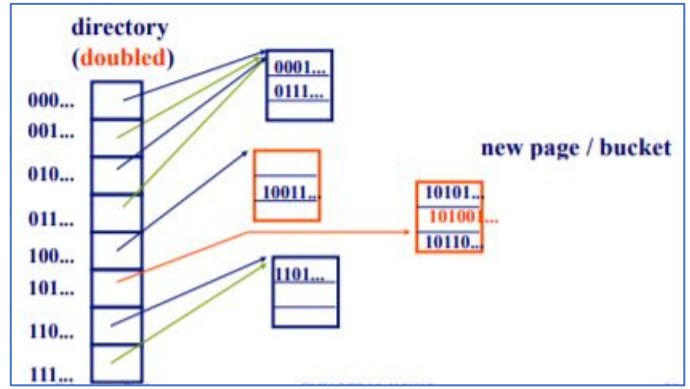
Extendible Hashing



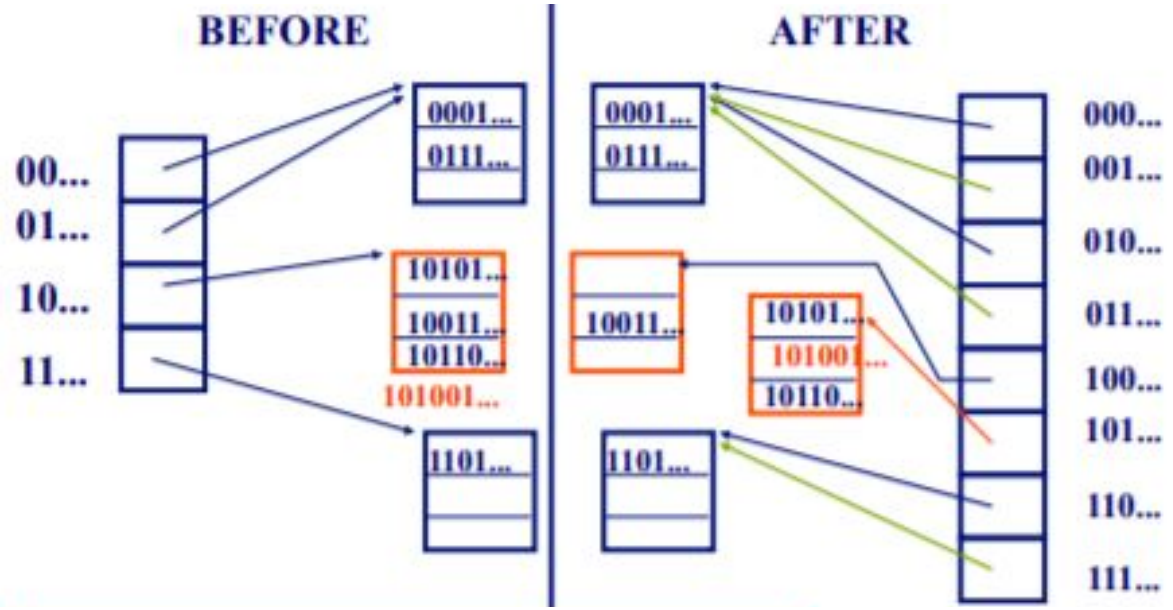
Extendible Hashing



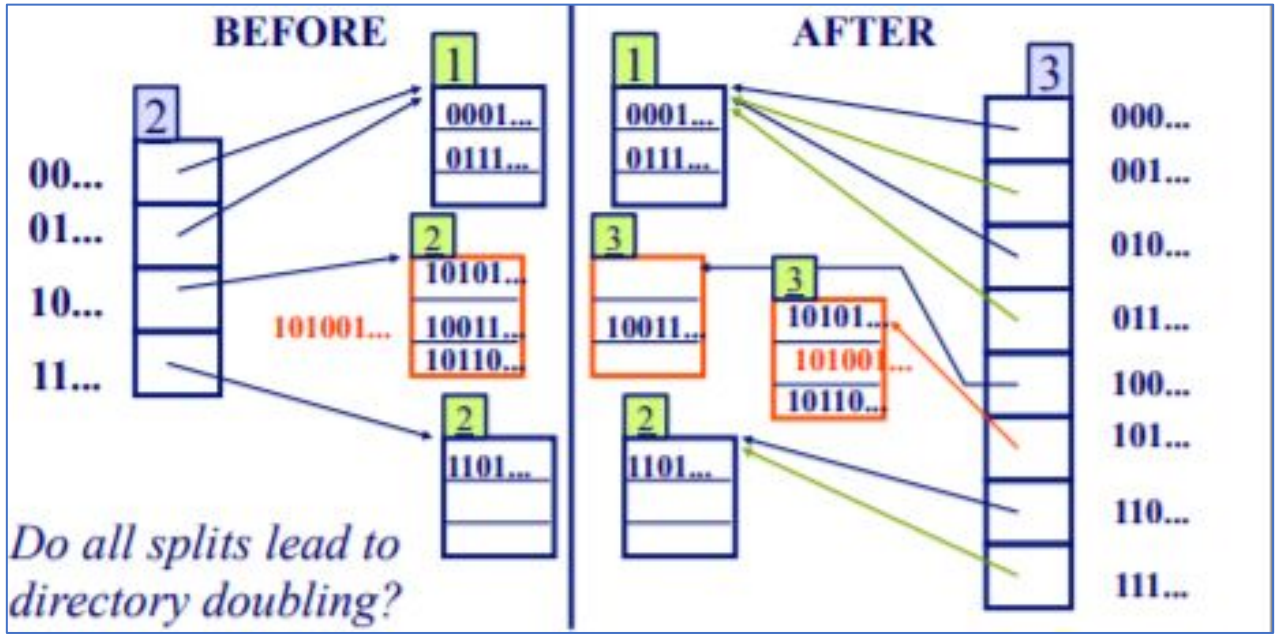




Extendible Hashing



Extendible Hashing



Extendible Hashing – Directory Doubling

- bucket overflows, 2 cases –
  - If local depth, d’ = global depth, d
    - Double the directory
  - If d’< d
    - Split the bucket, no need of directory doubling

Extendible Hashing

- Advantages:
  - Performance of file does not degrade with increase in size
  - Space overhead for directory is negligible
  - Splitting causes minor reorganization
- Disadvantages:
  - Directory must be searched before bucket. 2 block access instead of 1

Linear hashing

- Motivation: extendible hashing needs directory which doubles
- Q: can we do something simpler, with smoother growth?
- A: split buckets from left to right, regardless of which one overflowed

  Initially: h(x) = x mod N (N=3 here)

  Assume capacity: 2 records / bucket

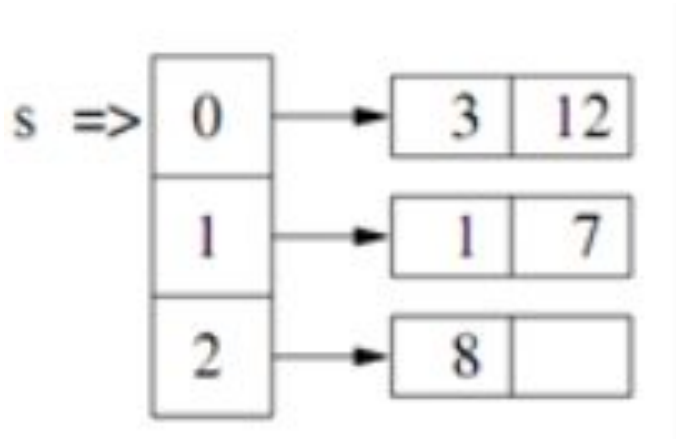
  Use two hash functions:

  $h_0(x)$ = x mod N (N=3 here) - for unsplit buckets

  $h_1(x)$ = x mod (2*N) (N=3 here) - for the splitted ones

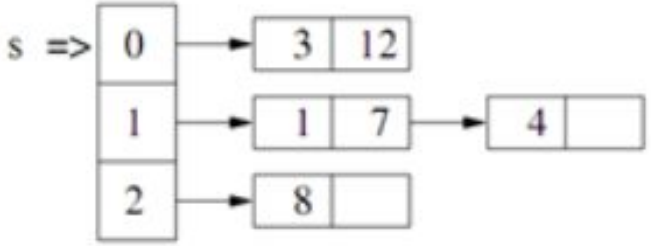
Linear Hashing Example

- In the following M=3 (initial # of buckets)
- Each bucket has 2 keys. One extra key for overflow.
- s is a pointer, pointing to the split location. This is the place where next split should take place.
- Insert Order: 1,7,3,8,12,4,11,2,10,13
- After insertion till 12:

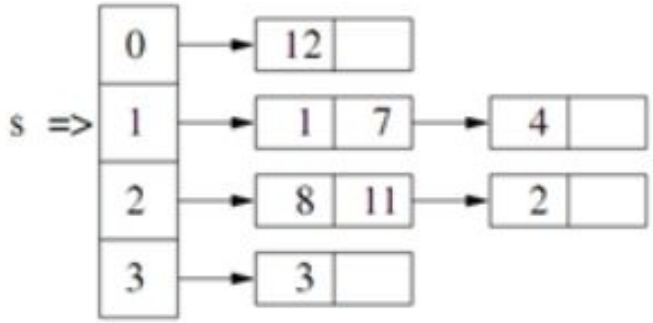


Linear Hashing example

- When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.

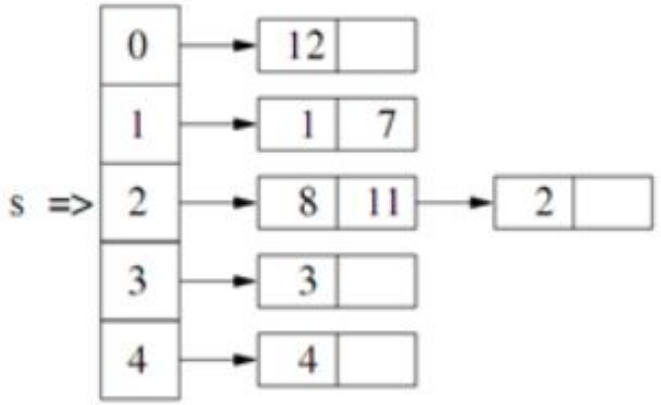


- split bucket 0 and rehashed all keys.
- Placed 3 to new bucket as (3 mod 6 = 3) and (12 mod 6 = 0).
- Then 11 and 2 are inserted.
- *s* is pointing to bucket 1, hence split bucket 1 by re- hashing it.

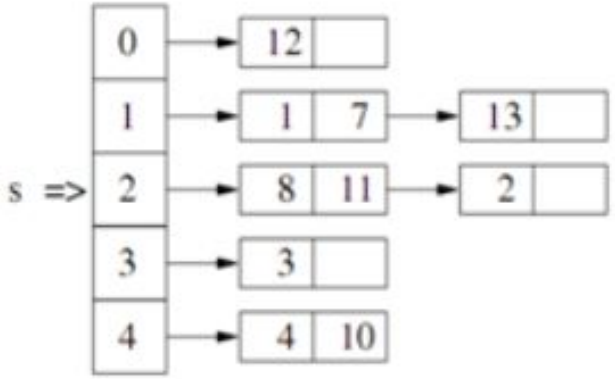


Linear Hashing Example

- After split:

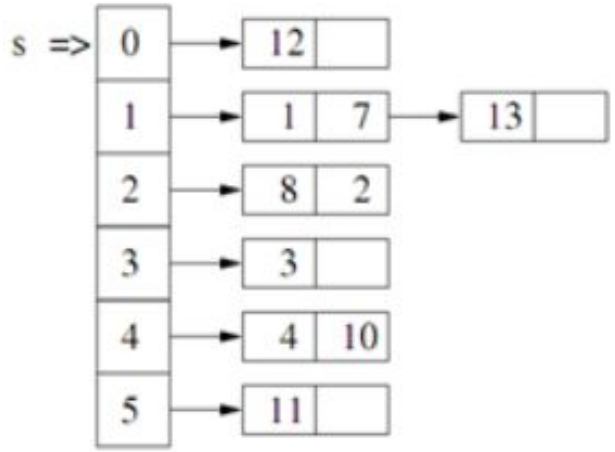


- Insertion of 10:
  - (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using $h_1(10) = 10 \bmod 6 = 4^{th}$ bucket
- For 13
  - same bucket
  - Overflow
  - split 2nd bucket.



Linear Hashing Example

- Final Hash Table:



- *s* is moved to the top again as one cycle is completed and *level* is incremented.

Linear hashing - Searching

- Algo to find key ‘k’:

  compute b= h_0(k) // original slot

  if b < s // has already split

    compute b= h_1(k)

  search bucket b

Linear hashing - Deletion

- Inverse of insertion
- If underflow, contract
- If the last bucket is empty,
  - remove it and
  - Decremented s.
- If s is 0 and the last bucket becomes empty,
  - s is made to point to bucket (n/2)-1, where n is the current number of buckets,
  - Level is decremented, and
  - the empty bucket is removed.

B+ Trees vs Hashing

B+ Trees	Hashing

• Speed on Search
  • Exact match queries, worst case
  • Range queries
  • Nearest-neighbor queries
• Speed on insertion + deletion
• Smooth growing and shrinking (no-reorg)
	• Speed
	  • On exact match queries, on the average