Hashing

CMPT 606

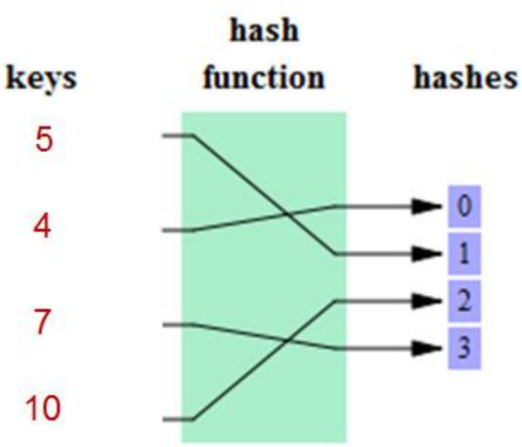
Read Chapter 16 (Hashing Sections)

Outline

1. Extendible Hashing
2. Linear Hashing

Hash Tables in Main Memory

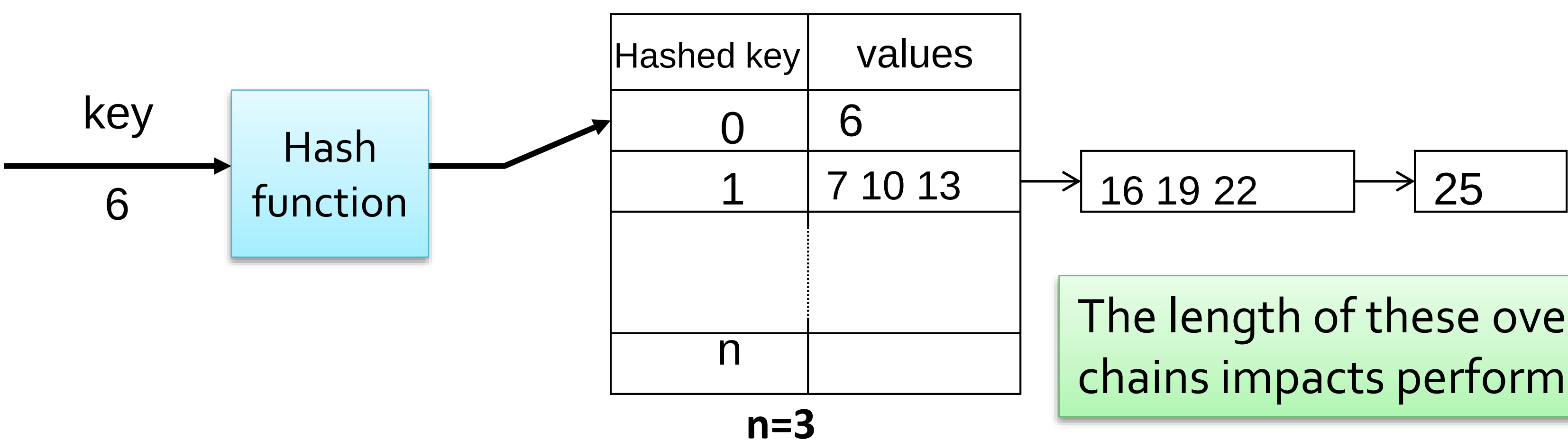
- In a hash table a hash function maps search key values to array elements
  - The array can either contain the data objects, or
  - Linked lists containing data objects (often called ***buckets***)
- Hash functions generate a value between 0 and *B*-1
  - Where *B* is the number of buckets
  - A record with **search key** ***K*** is **stored in bucket** ***h*(*K*)**



- **A typical hash functions is a bit representation of (the Search Key Value modulus the Number of Buckets)**
- **Hash indexes do not support range lookup**

Hash Index

- A hash index stores **key-value pairs** based on a *hash function* => *an array of pointers to buckets*
- Fast data structures that support **O(1)** look-ups
- The hash function often is ***K* mod *B***
  - Where ***K*** is the key value and ***B*** is the number of buckets
- Collision is resolved by placing new values in an **overflow bucket**



The length of these overflow chains impacts performance

Hash Functions

- Hash (any input key) return an **integer representation** of that key. DBMS hash tables use a fast hash function with a low collision rate:

- [CRC-64](#) **(1975)**
  - Used in networking for error detection
- [MurmurHash](#) **(2008)**
  - Designed to a fast, general purpose hash function
- [Google CityHash](#) **(2011)**
  - Designed to be faster for short keys (<64 bytes)
- [Facebook XXHash](#) **(2012)**
  - It is proposed in two flavors, 32 and 64 bits
- [Google FarmHash](#) **(2014)**
  - Newer version of CityHash with better collision rates

Static Hashing vs. Dynamic Hashing

- In static hashing the number of buckets is fixed (assumes you know the number of entries ahead of time)
  - If the file grows some buckets will have overflow chains, reducing efficiency

=> There should be enough buckets so that there are few overflow blocks

- Two versions of dynamic hashing that allow **dynamic File Expansion**
  - Extensible hashing
  - Linear hashing

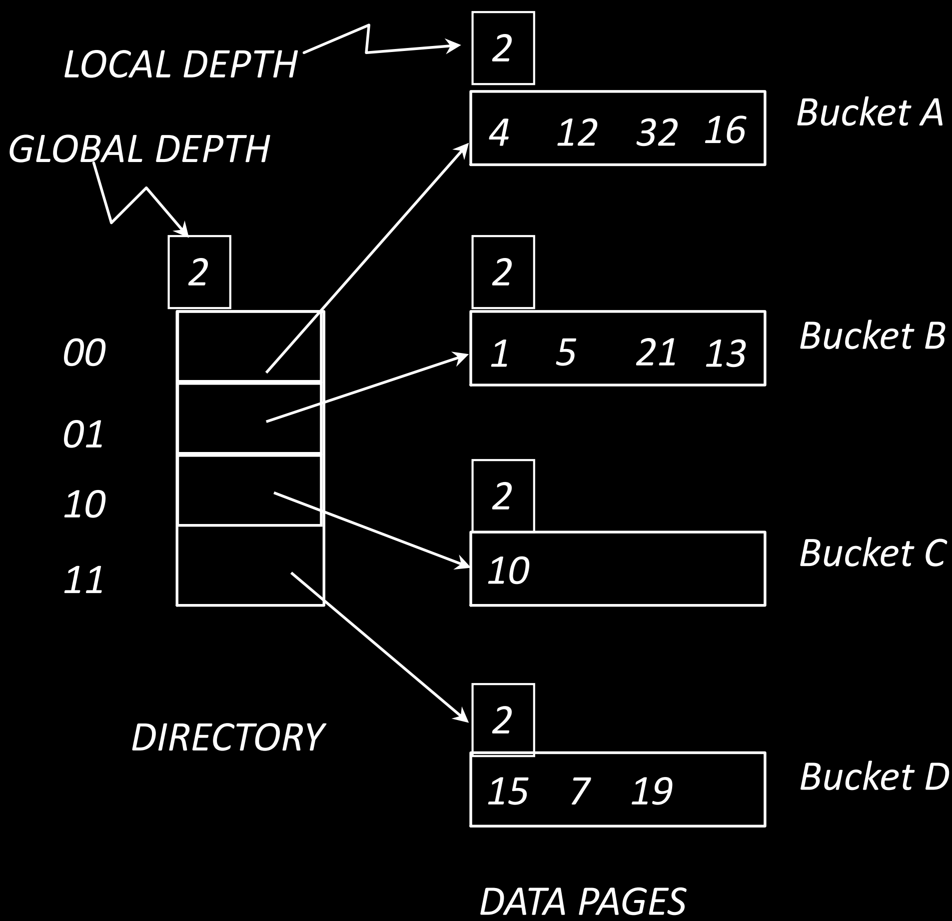
Extendible hashing

Extendible Hashing Structure

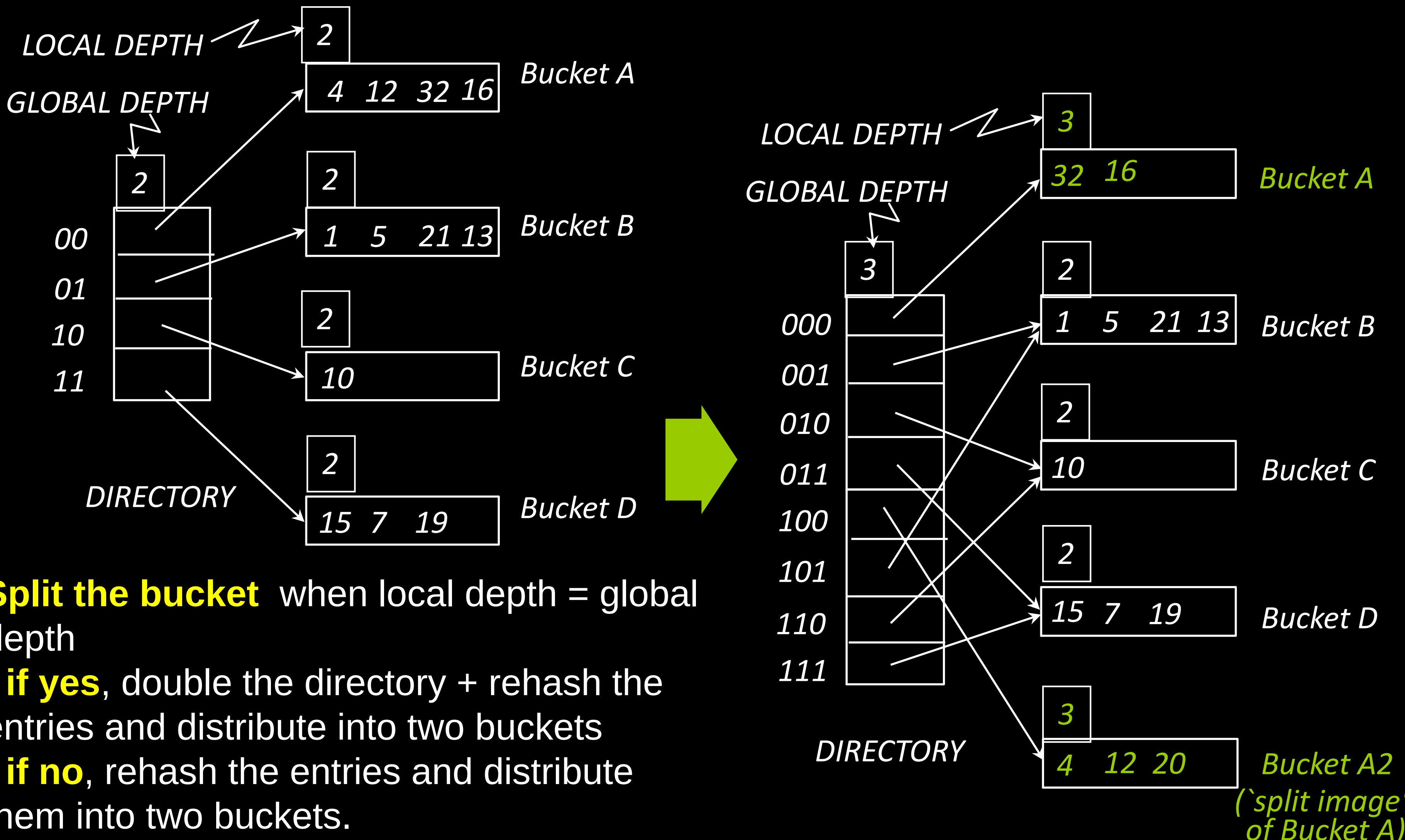
- In extendible hashing there is a **directory** to buckets
  - The directory is an array of pointers to buckets
  - As the array only contains pointers it is relatively small, so it can usually fit in memory
  - The directory size is doubled when overflow occurs
  - New buckets are only created as necessary
- The hash function computes a sequence of bits
  - The directory (and the associated buckets) uses the last i bits of the hash value
  - The directory will have $\mathbf{2}^{i}$ entries
    - When the directory grows, $\boldsymbol{i+1}$ bits of the hash value are used

Extendible Hashing Example

- Directory is array of size 4.
- Assume a bucket capacity of 4
- What info this hash data structure maintains?
  - Global Depth: # of bits needed to tell which bucket an entry belongs to
  - Local Depth: # of bits used to determine if an entry belongs to a specific bucket

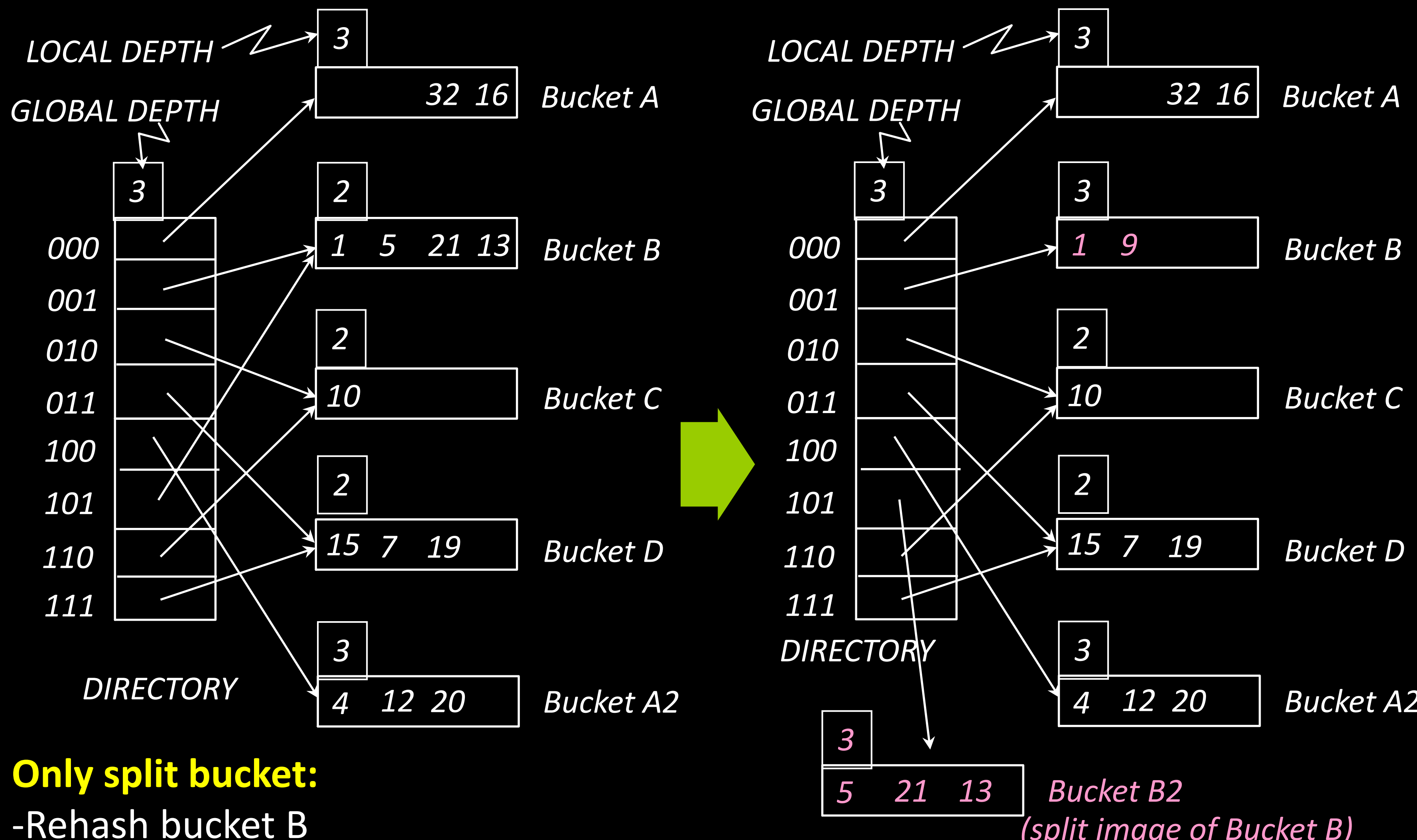


Insert 20 (10100): Double Directory



Split the bucket when local depth = global depth
- if yes, double the directory + rehash the entries and distribute into two buckets
- if no, rehash the entries and distribute them into two buckets.
- increment the local depth.

Insert 9 (1001): only Split Bucket



Only split bucket:

-Rehash bucket B

-Increment local depth

Extendible Hashing - Points to Note

- What is the global depth of directory?
  - Max # of bits needed to tell which bucket an entry belongs to.
- What is the local depth of a bucket?
  - # of bits used to determine if an entry belongs to a specific bucket
- When does bucket split cause directory doubling?
  - Bucket is full & local depth = global depth.
- How to efficiently double the directory?
  - Directory is doubled by copying it over and `fixing' pointers to the splited bucket

Linear Hashing

Linear Hashing (LH)

- Another dynamic hashing scheme, as an alternative to Extendible Hashing.
- What are problems in static/extendible hashing?
  - Static hashing can cause long overflow chains
  - Extendible hashing: data skew causing large directory
    - Data skew = Multiple entries with same hash value
    - If bucket already full of same hash value, will keep doubling forever!
- Is it possible to come up with a more balanced solution?
  - Shorter overflow chains than static hashing
  - No need for directory

=> Linear Hashing is the answer

Linear Hashing Algorithm

- Initial Stage.
  - The initial stage distributes entries into N_o buckets.
  - The hash function to perform is noted h_o
- *Idea*: Use a family of hash functions h_o, h_1, h_2, ...
  - $\mathbf{h_i}(key) = key \bmod (2^i N_o)$; N_o = initial # buckets
  - $\mathbf{h_{i+1}}$ doubles the range of $\mathbf{h_i}$ (similar to directory doubling)

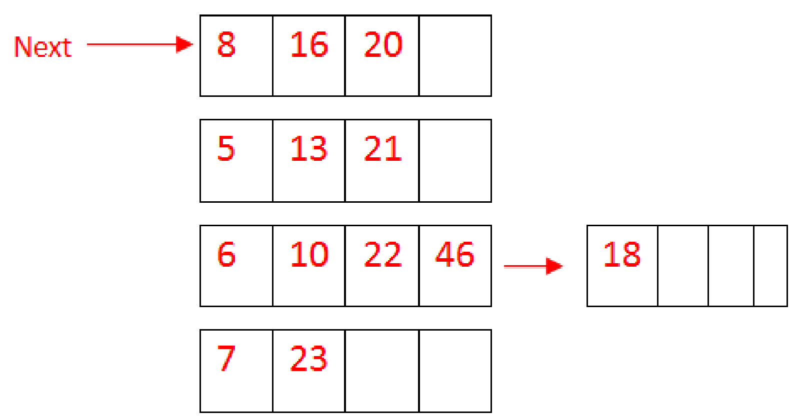
Example

Round = 1

$N_0 = 2$ & $N_1 = 4$

H_1 = key mod N_1

Next = 0



Linear Hashing Verbal Algorithm

- Splitting buckets
  - If a bucket overflows then its primary bucket is chained to an overflow bucket + the bucket **having the next pointer** will get split
    - The bucket to split is the one having the **next pointer** (*not* necessarily the bucket that overflows) – splitting done in round-robin fashion
    - The **next** pointer is moved to the next bucket… and so on until the N^{th} bucket has been split
    - When a bucket is split it entries (including those in overflow buckets) are distributed using h_1
  - To access split buckets the next level hash function (h_1) is applied
  - H_1 maps entries to $2N_0$ buckets
- Alternatively, splitting can be triggered when a fill factor (average occupancy in buckets e.g. 80%) is exceeded

Linear Hashing (Contd.)

- Directory avoided in LH by using overflow buckets, and choosing a bucket to split in round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R (for round R) initial buckets are split.

  - **Search:** To find bucket for data entry ***r*** use $\mathbf{h}_{round}(r)$
    - If $\mathbf{h}_{round}(r)$ in range `*Next* to N_R', *r* belongs here.
    - Else (if it is hashed to bucket **before** Next pointer) then must apply $\mathbf{h}_{round+1}(r)$ to find the bucket

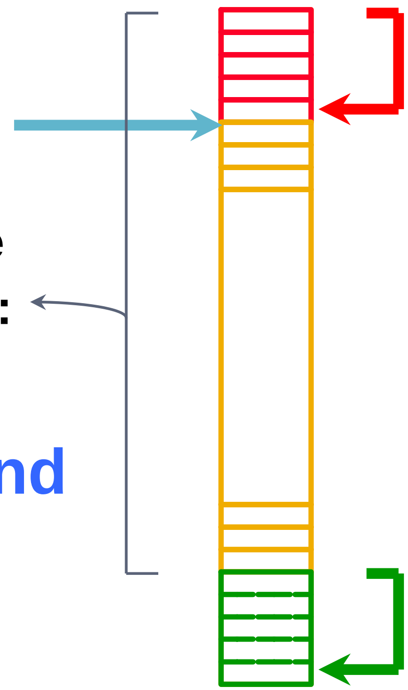
Finding bucket for a search key value

- In the middle of a round.

Bucket to be split

Next

Buckets that existed at the beginning of this round: this is the range of h_{round}

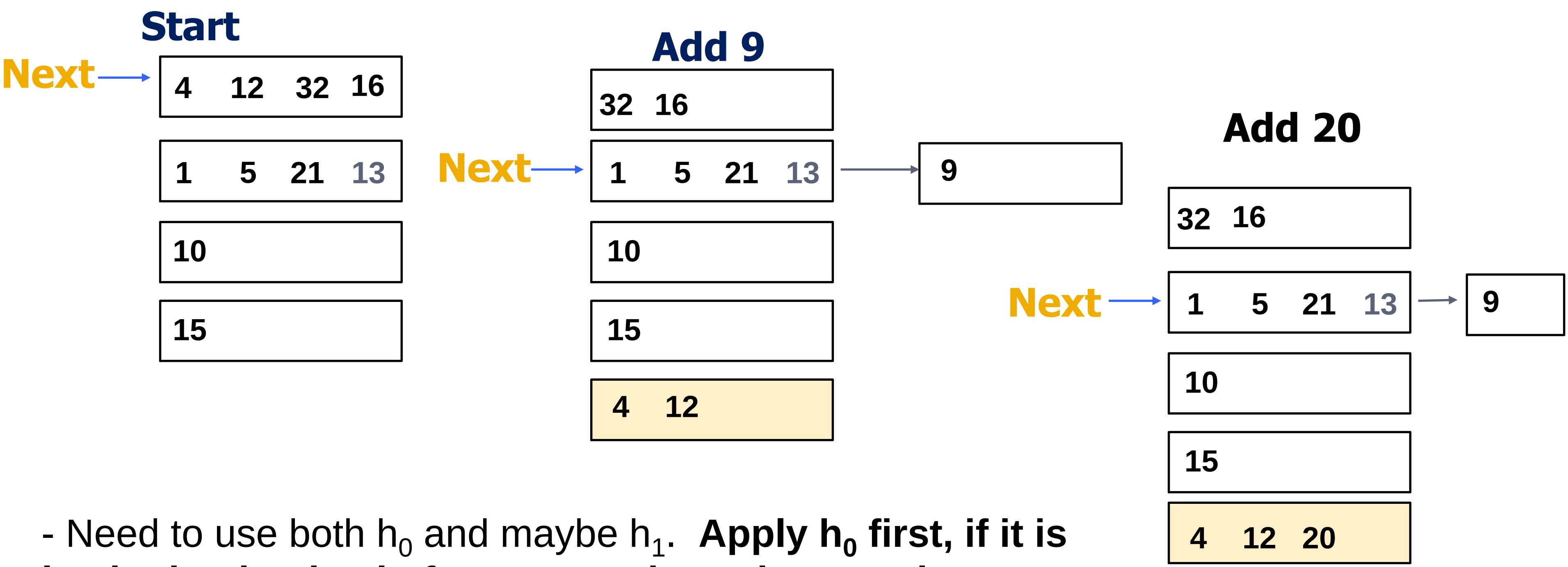


Buckets split in this round: If h_{round} (search key value) is in this range, must use $h_{round+1}$ (search key value) to decide if entry is in `split image' bucket.

buckets created (through splitting of other buckets) in this round

Linear Hashing Example

- Let's start with $\mathbf{N_0 = 4}$ buckets, $\mathbf{H_0}$ **= key mod 4**
  - Start at round_0 with next pointer pointing to bucket B_0
  - Each time any bucket fills, split "next" bucket and redistribute the values using $\mathbf{H_1}$ **($\mathbf{H_1}$ = key mod 8)**



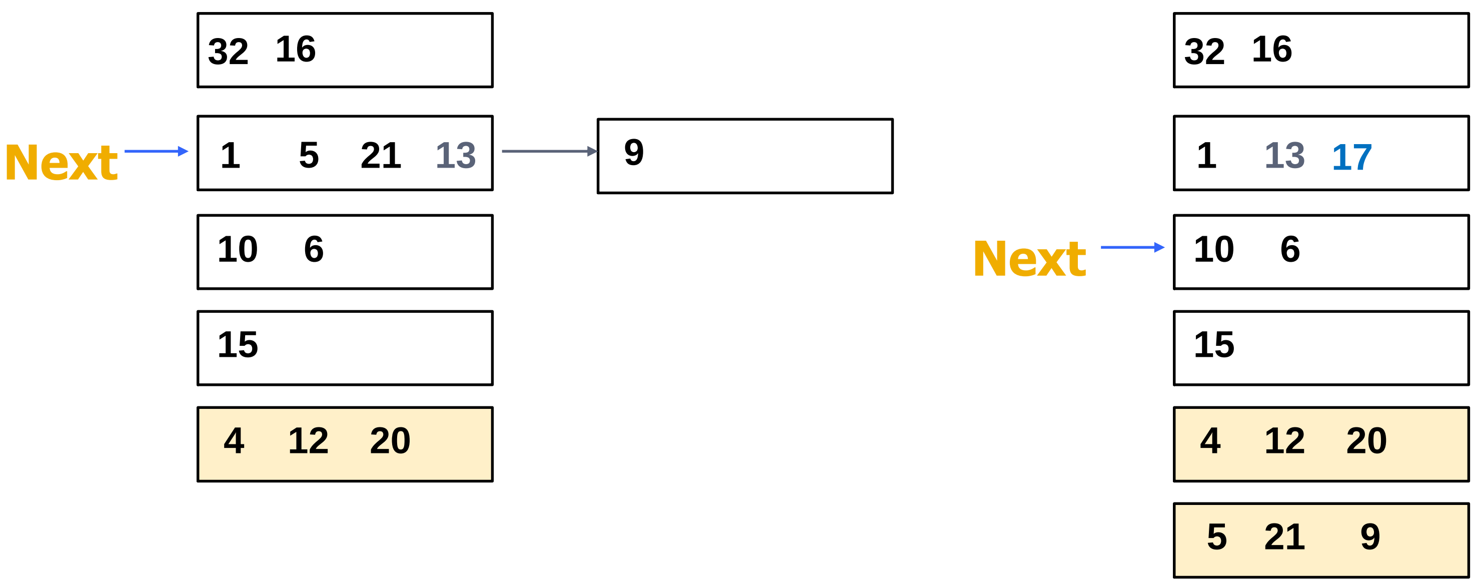
- Need to use both h_0 and maybe h_1. **Apply h_0 first, if it is hashed to bucket before next pointer then use h_1.**

```
If (h0(key) < Next) then Use h1(key) instead
```

Linear Hashing Example (cont)

- Overflow chains do exist, but eventually get split
- Instead of doubling, new buckets added one-at-a-time

Add 6 **Add 17**



Linear Hashing Example

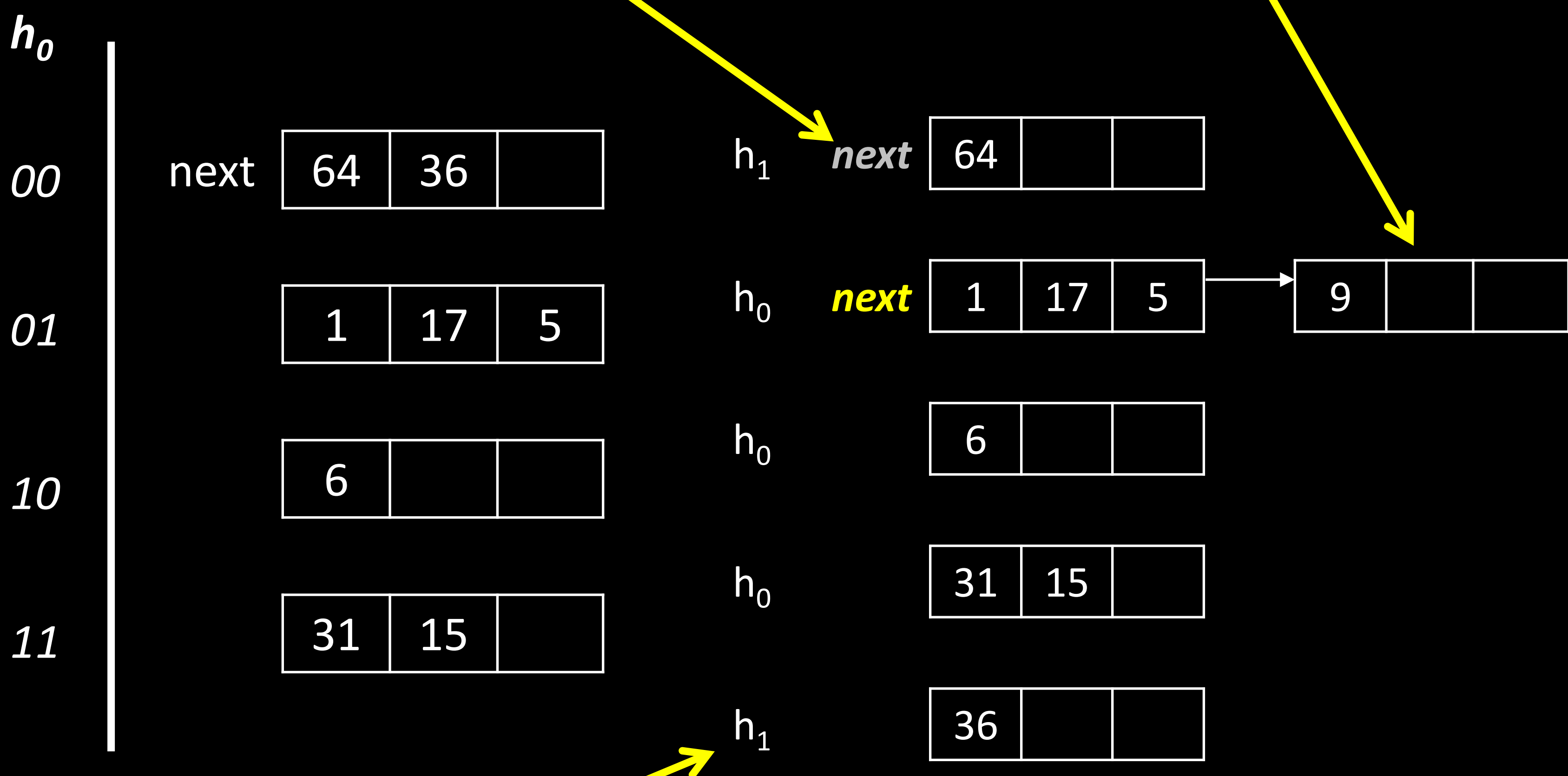
h_0					
00	**next**	**B_0**	64	36	
01		B_1	1	17	5
10		B_2	6		
11		B_3	31	15	

- Initially, we have $N_0 = 4$ buckets
- Assume three entries fit on a bucket
- h_0 = key mod 4
- ***Next*** pointer indicates which bucket is to split next (Round Robin)
- Now consider what happens when 9 (1001) is inserted (which will not fit in the second bucket)

Insert 9 (1001)

- The bucket indicated by **next** (the first one) is split
- **Next** is incremented.

An overflow bucket is chained to the primary bucket to contain the inserted value.
This causes a split to occur.



For subsequent inserts, apply hash function $\mathbf{h_0}$ first, if the key is hashed to bucket before **next** pointer. Then h_1 must be used to insert the new entry.

Insert 8 (1000), 7(111), 18(10010), 14(1100)

$(000)H_1$		B_0	64	**8**					
$(01)H_0$	***next***	B_1	1	17	5	→	9		
$(10)H_0$		B_2	6	**18**	**14**				
$(11)H_0$		B_3	31	15	**7**				
$(100)H_1$		B_4	36						

- Note that the split bucket is not necessarily the overflow bucket. The split bucket is chosen based on round robin.
- Need to use both h_0 and maybe h_1. **Apply h_0 first, if it is hashed to bucket before next pointer. Then use h_1.**

Before Insert 11 (1011)

(000)H_1		B_0	64	8				
(01)H_0	***next***	B_1	1	17	5	9		
(10)H_0		B_2	6	18	14			
(11)H_0		B_3	31	15	7			
(100)H_1		B_4	36					

After Insert 11 (1011)

(000)H_1		B_0	64	8					
(001)H_1	next	B_1	1	17	9				
(10)H_0	next	B_2	6	18	14				
(11)H_0		B_3	31	15	7	→	11		
(100)H_1		B_4	36						
(101)H_1		B_5	5						

Which hash function (h0 or h1) for searching 9?

How about searching for 18?

Inserting 11 caused an overflow of B_3.
This triggered splitting B_1

Insert 32, 16

$(000)H_1$		B_0	64	8	32	→	16		
$(001)H_1$		B_1	1	17	9				
$(010)H_1$	*next*	B_2	18						
$(11)H_0$	***next***	B_3	31	15	7	→	11		
$(100)H_1$		B_4	36						
$(101)H_1$		B_5	5						
$(110)H_1$		B_6	6	14					

Inserting 16 caused an overflow of B_0.
This triggered splitting B_2

Insert 10, 13, 23

$(000)H_1$	***next***	B_0	64	8	32	→	16	
$(001)H_1$		B_1	1	17	9			
$(\mathbf{0}10)H_1$		B_2	18	**10**				
$(\mathbf{0}11)H_0$	*next*	B_3	11					
$(100)H_1$		B_4	36					
$(101)H_1$		B_5	5	**13**				
$(110)H_1$		B_6	6	14				
$(111)H_1$		B_7	31	15	7	→	23	

- Inserting 23 caused an overflow of B_3
- This triggered splitting B_3
- Round_1 is now complete as the 4 buckets of Round_1 were split. Now all 8 buckets are using and h_1 = key mod 8

- Next pointer is reset to 0

- Roud_2 starts and subsequent splits will use h_2 hash function (h_2 = key mod 16)

Extendible and Linear Hashing

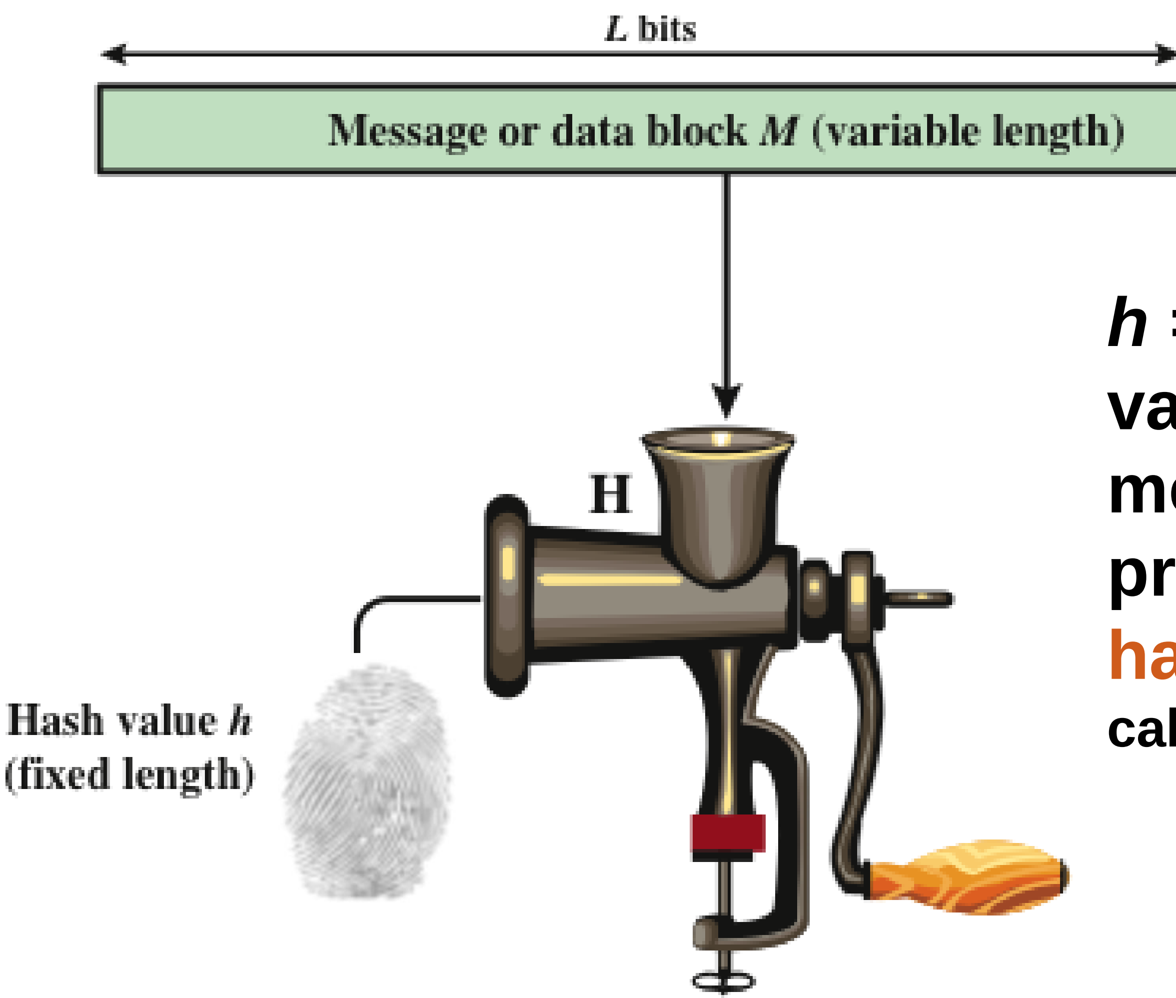
- Linear hashing does not require a dictionary
- Linear hashing may result in less space efficiency because buckets are split before they overflow
- Multiple collisions in one bucket in extensible hashing will result in a large directory
  - Such a directory may not fit on one disk block
- Collisions in linear hashing lead to long overflow chains for the bucket with the collisions
  - Requiring multiple disk reads for that bucket
  - But no increase in the cost of accessing other buckets

Summary

- Use a hash index for point queries only. Use a B-tree for multipoint queries or range queries
- Use clustered index:
  - if your queries need all or most of the table columns
  - if multipoint, range queries or order-by queries are often requested

Appendix - Cryptographic Hashing

Cryptographic Hash Function Analogy



h **= H(*m*). Hash variable size message m to produce a fixed size hash value (sometimes called a *message digest*)**

Cryptographic Hash Functions

- MD5 (Message Digest 5)
  - Produces a 128-bit hash
  - Collisions can be found. An attacker can use them to substitute an authorized message with an unauthorized one.
- SHA1 (Secure Hash Algorithm 1)
  - 160-bit hash
  - Collisions can be found
- SHA2
  - Actually 4 different hash functions: SHA-224, SHA-256, SHA-384, SHA-512
  - Minor attacks, but still good
- SHA3
  - New NIST standard
  - No known attacks

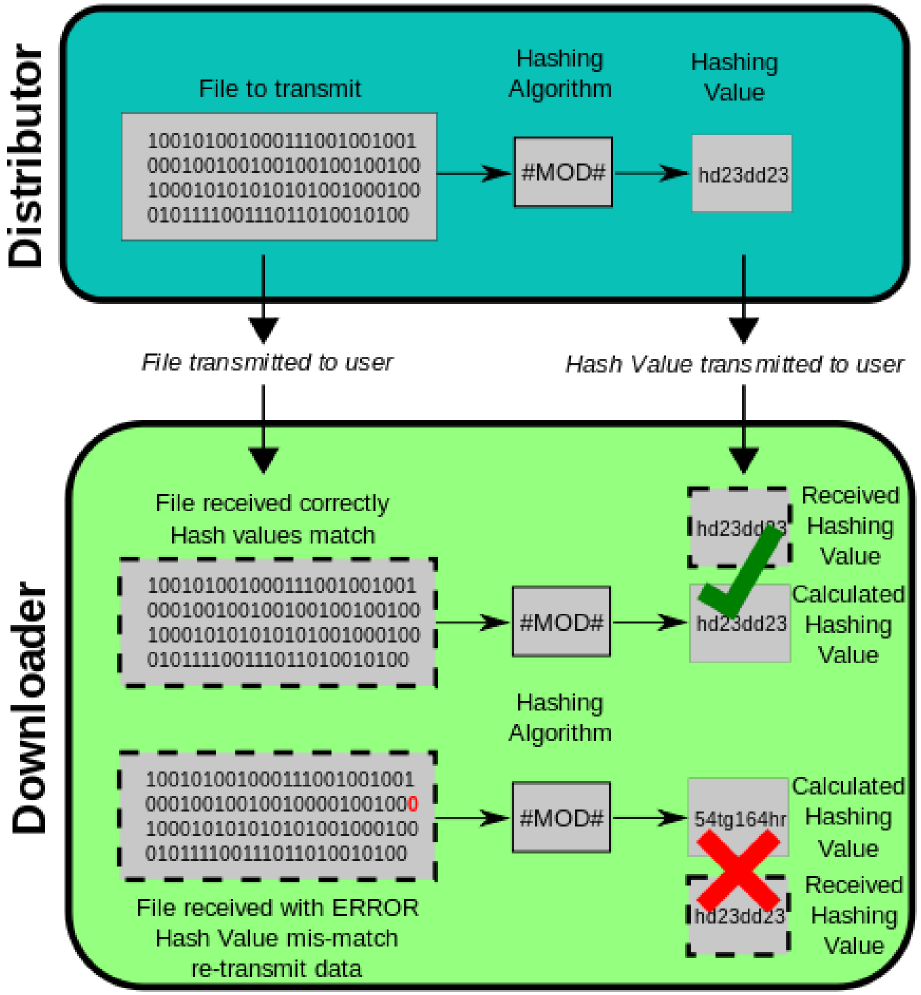
Sample in Python

Data (??-bits) → SHA256 → Hash (256 bits)

```
import hashlib
md = hashlib.sha256(b"The quick brown fox jumps over the lazy dog").hexdigest()
print (md)
```

```
d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

Application 1: File Transmission



https://en.wikipedia.org/wiki/MD5

Application 2: Message Authentication Code (MAC)