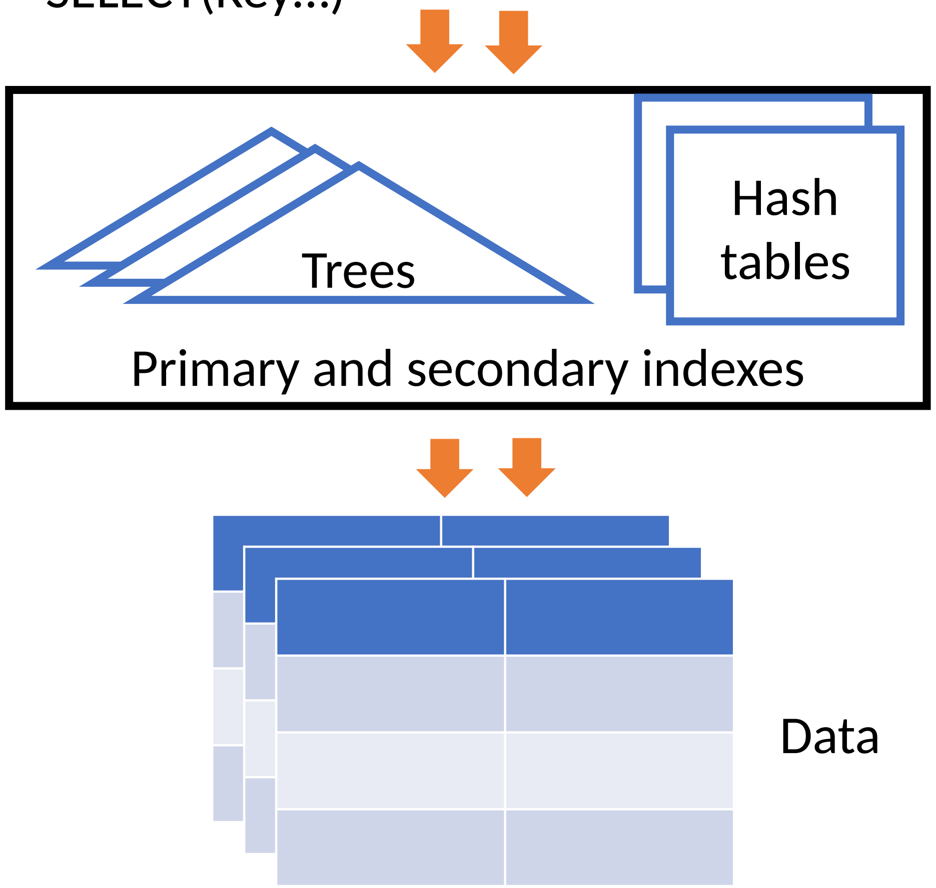
# CSE 541: Database Systems I

Hash Tables

Slides Credit: Tianzheng Wang

# Recap: Indexing

**User queries and transactions:**
UPDATE(Key, record…)
SELECT(Key…)




Alternatives for index entries:
- Key + actual data record
- Key + RID
- Key + list of RIDs

➔ Use a key to find data entry
Can be implemented using different index structures
- Hash tables
  - Equality search only
- Trees and skip lists
  - Equality + range search
  - “Order preserving”

# Hash Tables

Hash table: a collection of buckets, each bucket holds keys (and associated payload) that are hashed into the same bucket using a hash function H

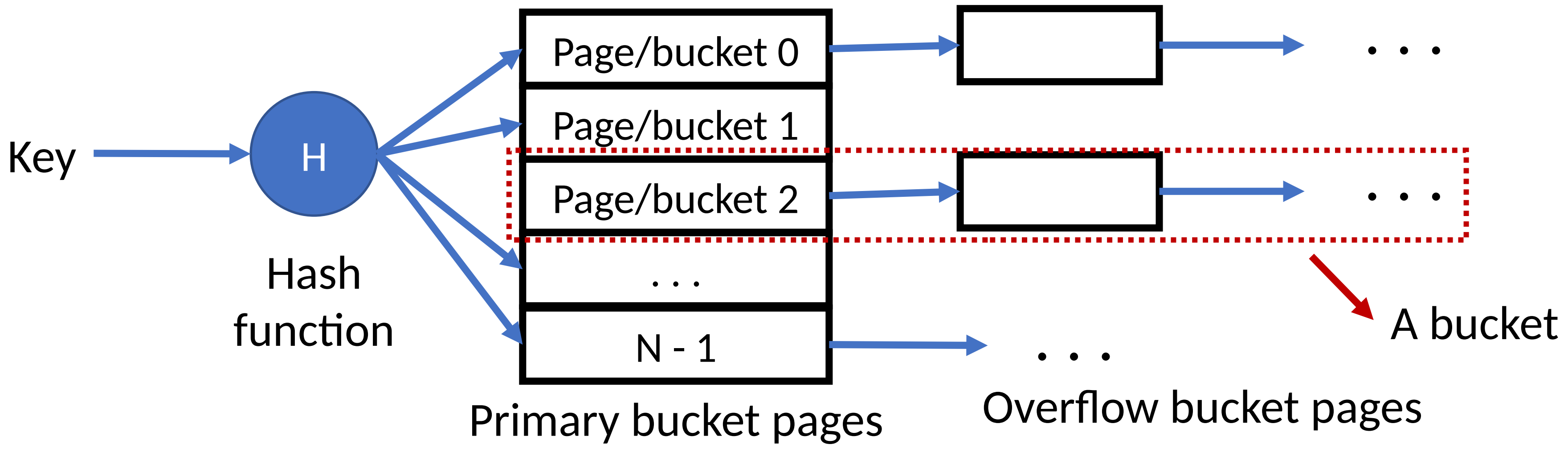
- Hash function maps key to an index into bucket number
  - Collisions: for different keys K1 and K2, H(K1) == H(K2)
  - So each bucket may contain multiple data entries
- Good for point queries
  - Arguably the fastest option
- Typically cannot do range scans
  - Order is usually not preserved by hash functions
  - Possible to have: K1 < K2 but H(K1) > H(K2)

Techniques

- Static hashing
- Dynamic: extendible hashing and linear hashing

# Static Hashing

- Fixed number of buckets
- Each bucket consists of one or more pages
- Each bucket page has fixed capacity
  - No space in single page: create overflow pages chained together per bucket



# Static Hashing

- Hash function must evenly distribute keys into different buckets
  - Usually works well: H(key) = (A * key + B)
  - Bucket number = H(key) mod N

A and B are constants that need to be tuned

N: Hash table size

Main problem: the number of buckets is fixed

- Chains of overflow pages can become long
  - Affect search performance
- Waste space when data size reduced significantly
  - Empty buckets
- High cost to re-organize/expand by increasing the number of buckets: too much data copying

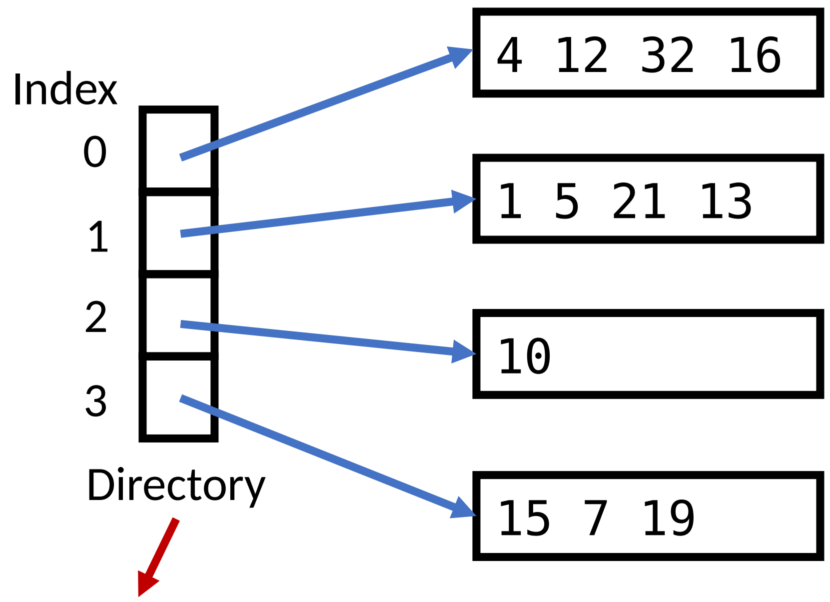
# Extensible Hashing

**Use a level of indirection to easily add more buckets**

- Add a directory of pointers to buckets
  - Much smaller than actual bucket pages, cheap to resize/expand
- Double the directory when a bucket is full
  - Only split buckets that are full
  - No overflow pages

# Extensible Hashing

Bucket: capacity = 4




Array with four elements

- Each element is a pointer to a bucket page (page offset in file)

Find: Apply hash function, modulo directory size (N) to find the right bucket

- N is power of 2, mod N **same as taking the last $\log_2 N$ bits**
- E.g., capacity = 4, take 2 bits

Example:

Assume H(5) = 13 = $11\underline{\mathbf{01}}_2$

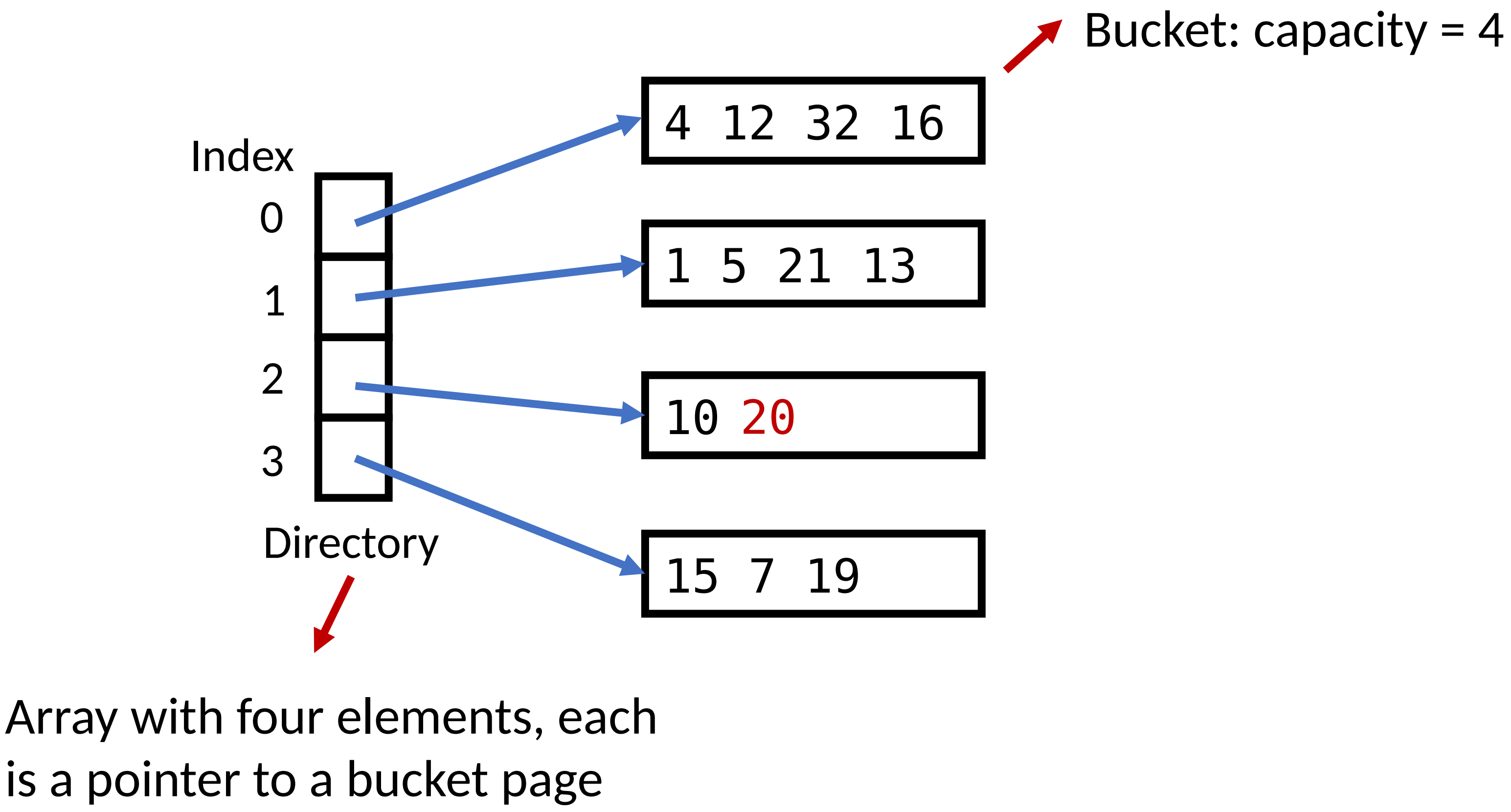
- 13 mod 4 = 1

Insert: Use Find() to arrive at the right bucket first, then add new data entry

- Done if there is enough space
- Split if the bucket is already full

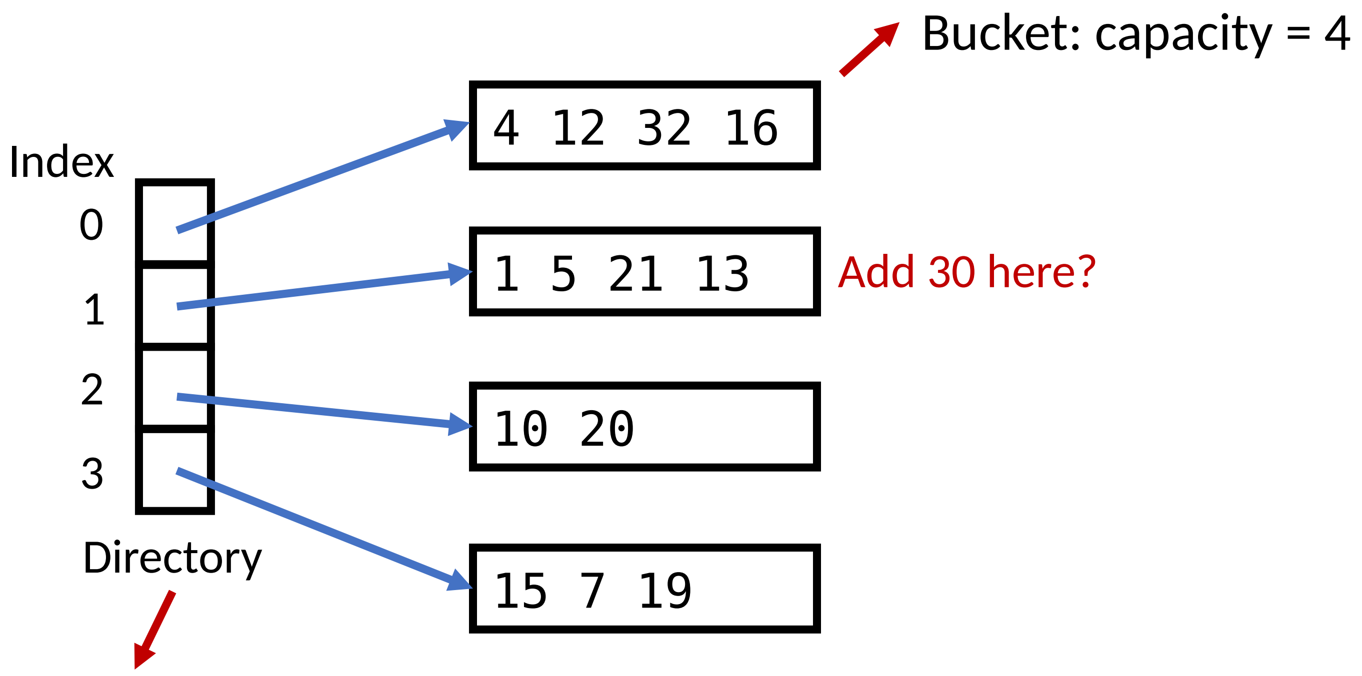
# Extensible Hashing

- Insert 20: assume H(20) mod 4 = 2, bucket 2 has enough space



# Extensible Hashing

- Insert 30: assume H(30) mod 4 = 1, bucket 1 is already full




Array with four elements, each
is a pointer to a bucket page

# Extensible Hashing

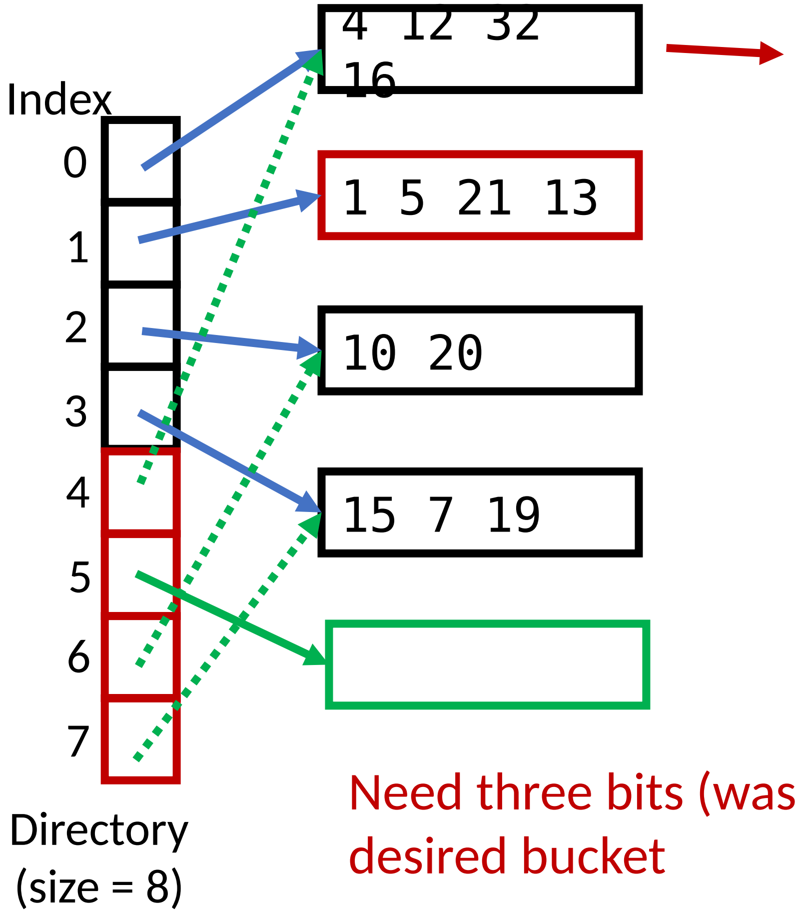
Insert 30: assume H(30) mod 4 = 1, bucket 1 is already full

- Double the directory size to 8

Corresponding elements:

- Buckets 0 and 4
- Buckets 1 and 5
- Buckets 2 and 6
- Buckets 3 and 7

- Old vs. new buckets: differ in the number of bits used




Non-full buckets/not split; two incoming pointers

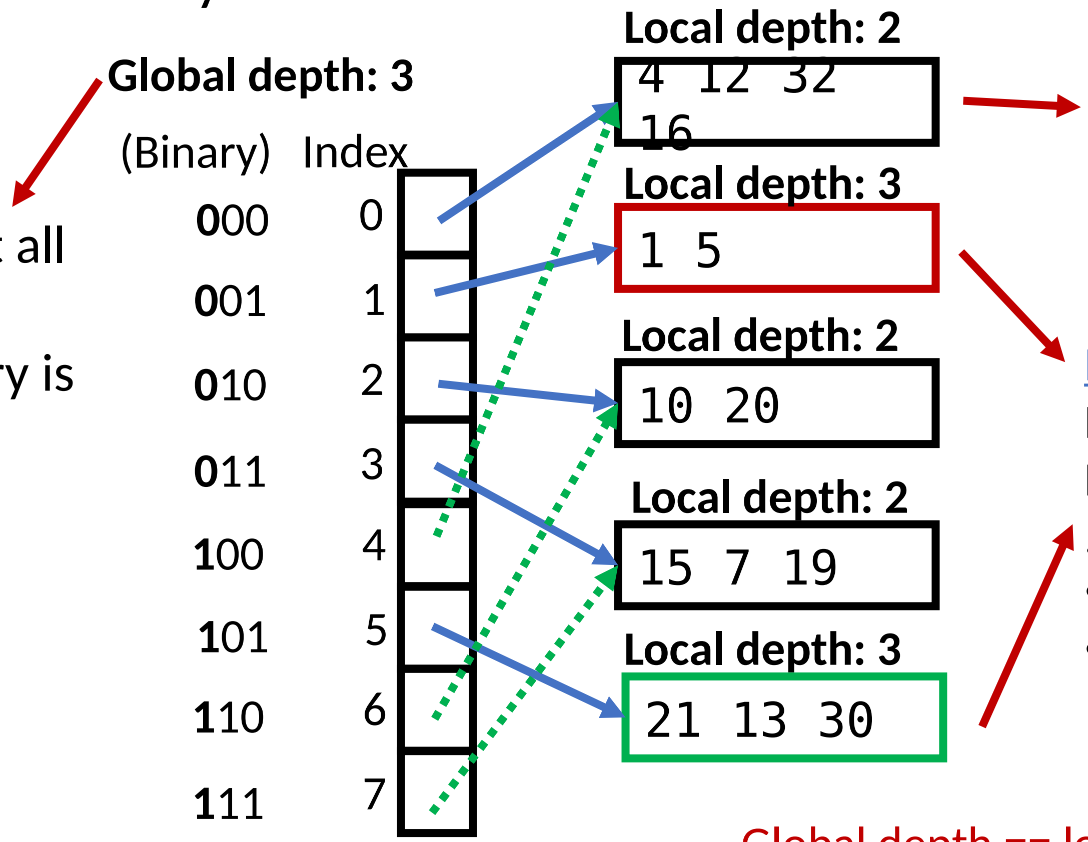
Need three bits (was two) to reach desired bucket

# Extensible Hashing

Insert 30: assume H(30) mod 4 = 1, bucket 1 is already full

- Double the directory size to 8

- # of bits to represent all slots in the directory
- +1 when the directory is doubled




Directory
(size = 8)

Non-full buckets/not split; two incoming pointers

Bucket entries re-arranged:

H(30) mod **8** = 5

H(1) mod **8** = 1

. . .

- Mod 8: use the last three bits of H(key)
- Before split, Mod 4: use the last two bits

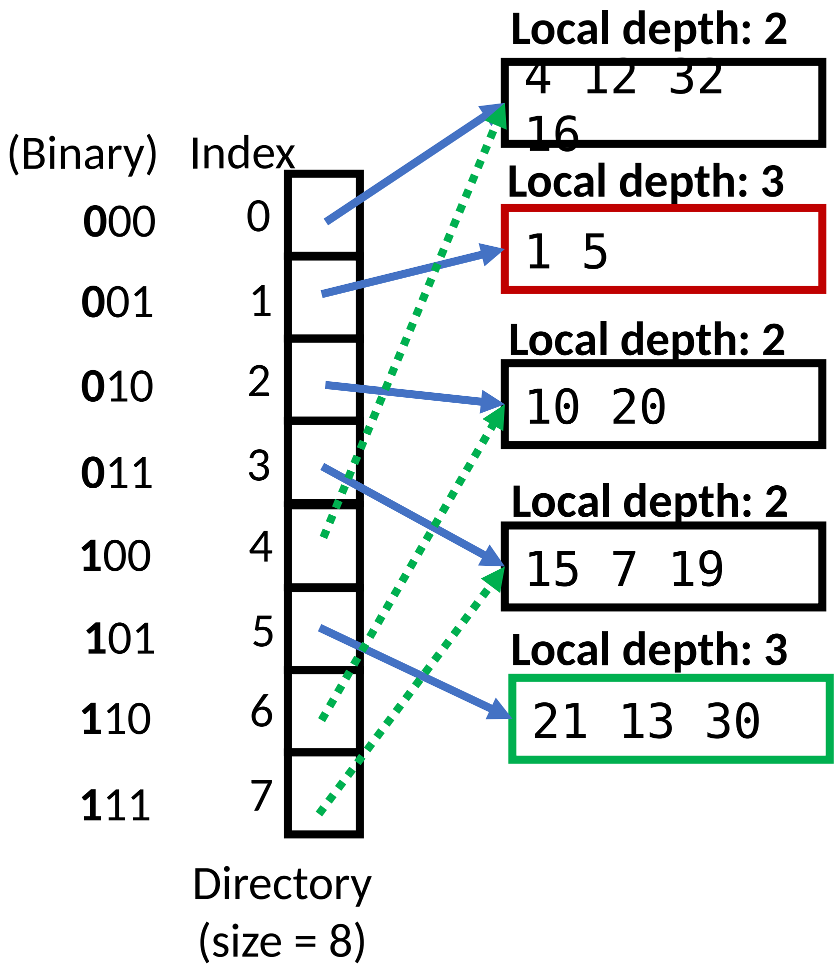
Global depth == local depth and bucket full ➔ must double the directory size

# Extensible Hashing

- Global depth: the last D bits of H(key) used to decide bucket number
  - $2^{\text{Global depth}}$ = Number of total buckets (i.e., directory size)
  - Example: suppose H("abc") = 20,
  - Decimal 20 = binary 10100, global depth = 2

  ➔ Bucket number = 20 mod 4 (or taking the last 2 bits of 20) = 0
  - Increment global depth by 1 whenever directory is doubled
- Directory doubling not always necessary when a bucket is full
  - Only happens if local depth of the bucket == global depth
    - I.e., no available slot in the directory for the new bucket
  - Bucket pages in file are allocated gradually as they are split
- Each directory entry contains a pointer (page ID)
  - Much smaller than the actual file, more chance to fit in memory
  - Search cost: one I/O if directory already in memory, two if not

# Extensible Hashing




Delete:

- Remove entry
- Merge empty bucket
- Decrement local depth
- Halve the directory if every slot points to the same split image

# Linear Hashing

**Alternative to extendible hashing, without using directory**

Basic idea:

- Use a family of hash functions H0, H1, H2, …
  - Each hash function's range is twice that of its predecessor
  - H0 maps to M buckets, H1 maps to 2M buckets
    - E.g., H0: 0-7, H1: 0-15, H2: 0-31, and so on
  - Can be obtained by having a base function H, and define:
    - Hn(key) = H(key) mod ($2^n$ * M)
    - H0(key) = H(key) mod (1 * M)
    - H1(key) = H(key) mod (2 * M)
    - H2(key) = H(key) mod (4 * M)

    ➔ $H_{n+1}$ doubles the range of $H_n$
- Use overflow pages and choose bucket to split round-robin
- Main buckets are stored sequentially in file

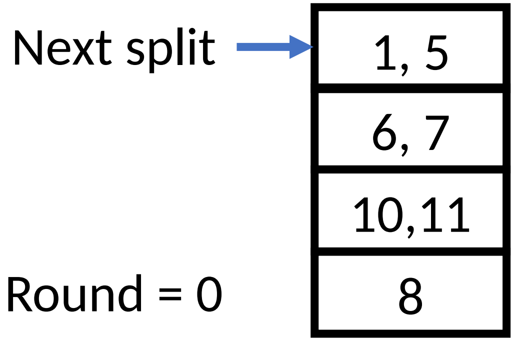
# Linear Hashing

**Rounds of splitting**

- Split buckets in rounds; at round n, use $H_n$ and $H_{n+1}$
- The bucket to split is chosen linearly – one bucket after another, eventually doubling the number of buckets
    - Always start from the **first** bucket

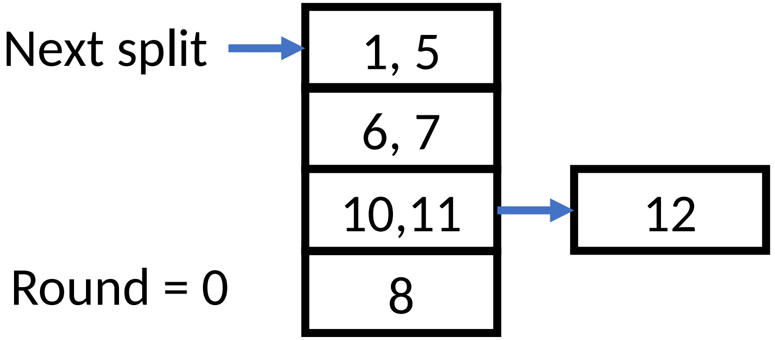
Example: initially four buckets (M = 4), max two elements per bucket

1 Initial state:




**H0 and H1 in use**

2 Insert to the third bucket: overflow




**H0 and H1 in use**

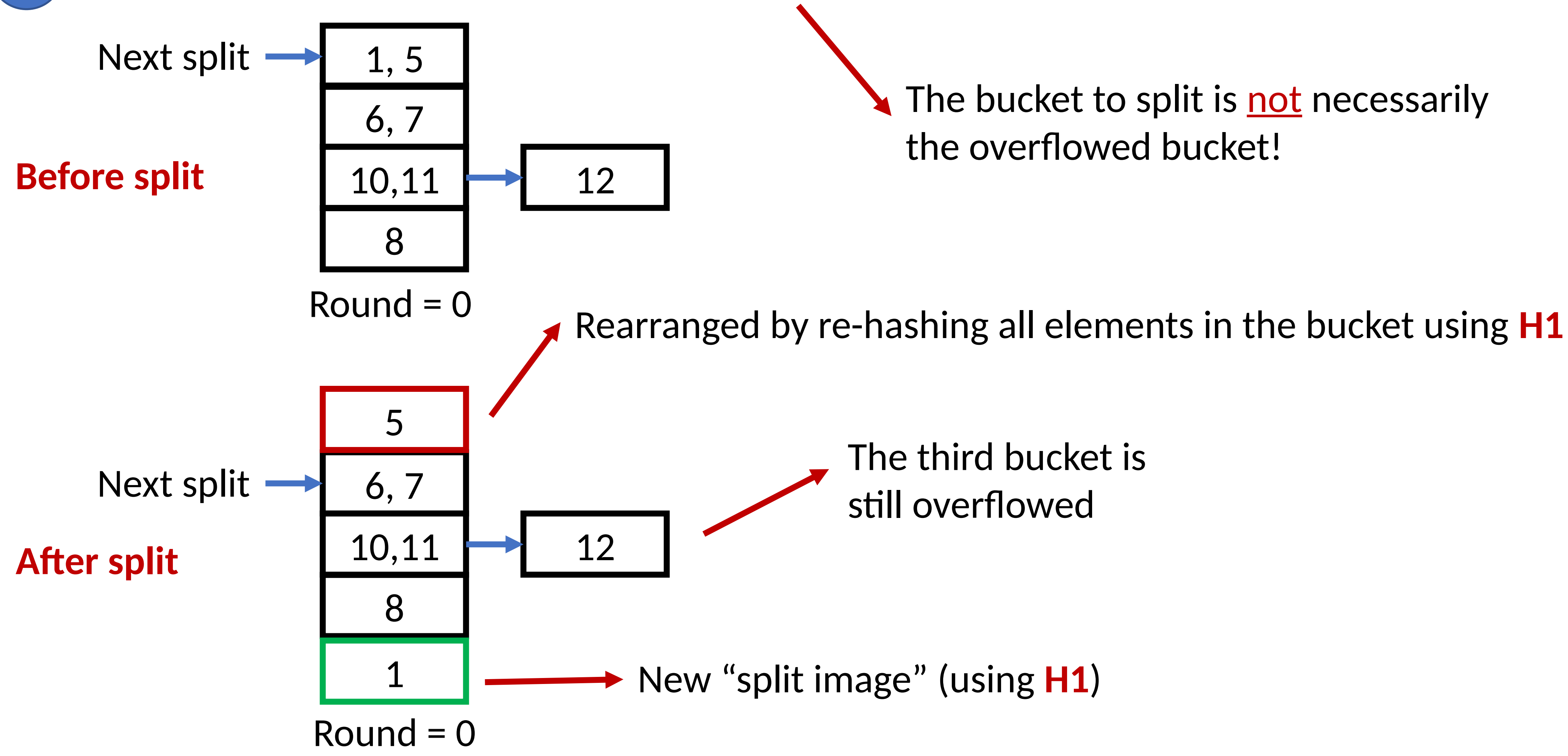
# Linear Hashing

- When should we trigger splits?
- Can use many different criteria
  - E.g., number of buckets with overflows > threshold
  - E.g., longest overflow chain > threshold
  - **E.g., as long as there is any overflow**

Simplistic example policy in the rest of the slide deck

# Linear Hashing

3 Start by splitting the bucket pointed to by [next split] (the first bucket [1, 5])



# Linear Hashing

**Search for Key K:**

- Need two hash function: one for buckets before the split, the other for buckets that are already split

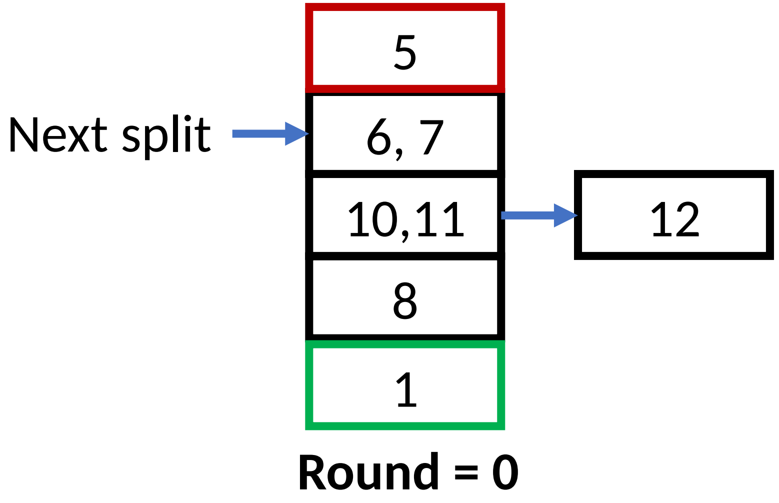
Inputs:

- Key K
- Current split round N

Decide which hash functions to use

Algorithm:

- Do b = $H_n(K)$ first
- If b >= next_split: done (b is the final bucket number)
- Otherwise b = $H_{n+1}(K)$



# Linear Hashing

**Insert Key K:**

- Similar to search: need two hash function: one for buckets before the split, the other for buckets that are already split

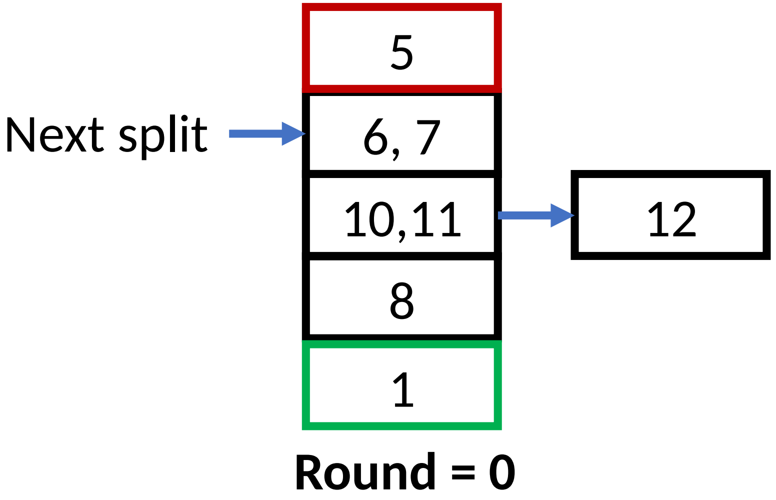
Decide which hash functions to use

Inputs:

- Key K
- Current split round N

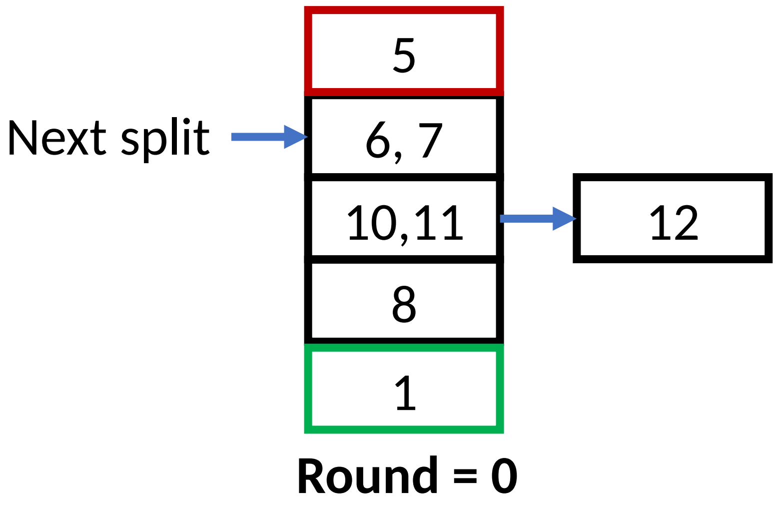
Algorithm:

- Let b = $H_n(K)$
- If b < next_split, let b = $H_{n+1}(K)$
- Insert into bucket b, overflow if needed

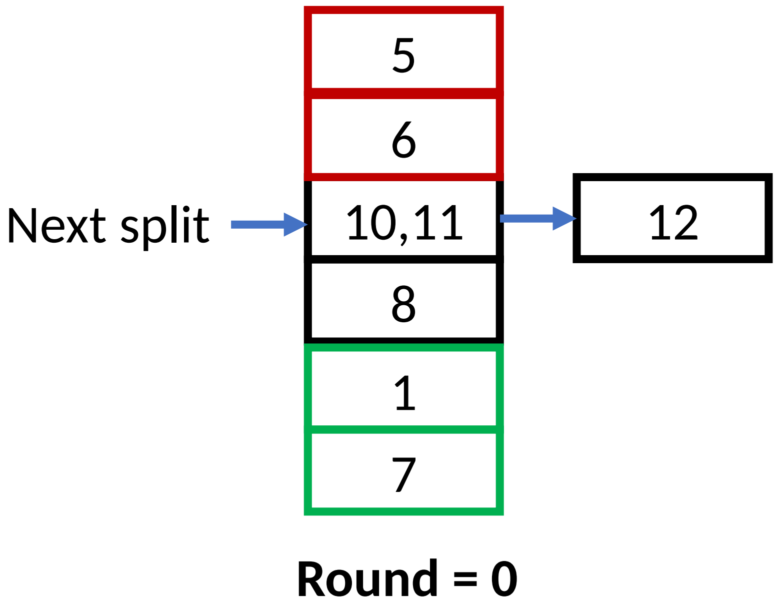


# Linear Hashing

4 Continue to split the second bucket



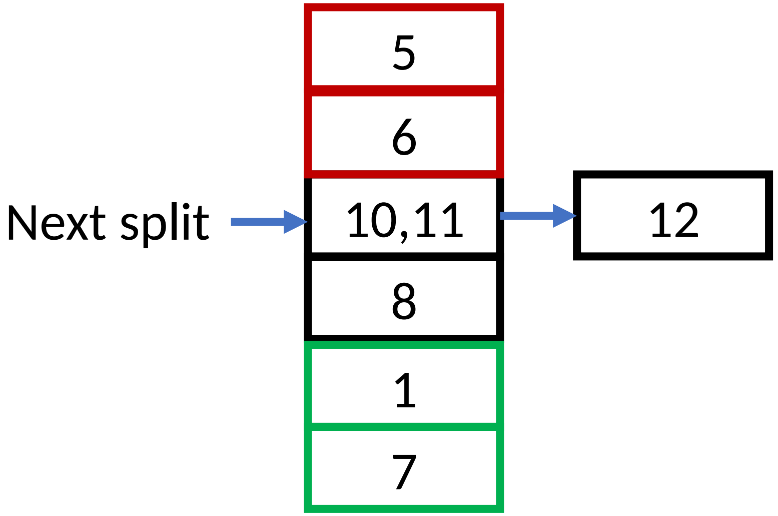

Before split




After split

# Linear Hashing

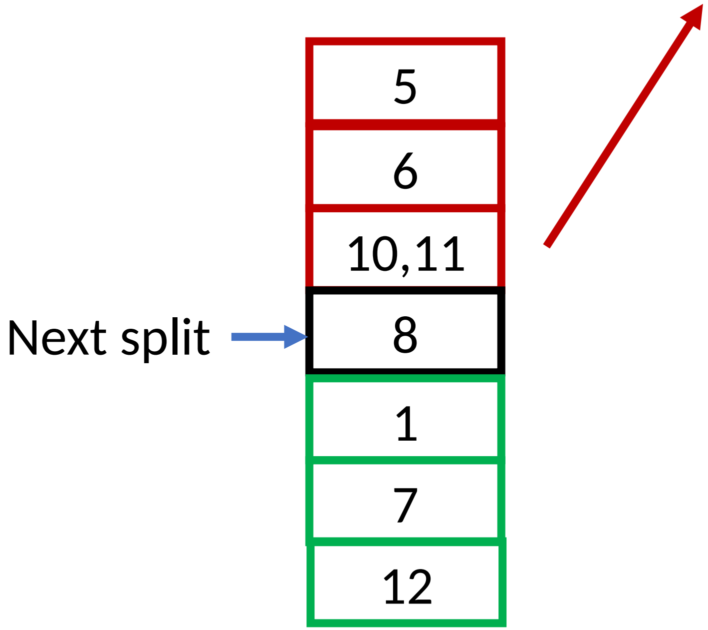
5 Continue to split the third bucket




Round = 0

**Before split**

10, 11, and 12 re-hashed using H1 and placed in the corresponding buckets

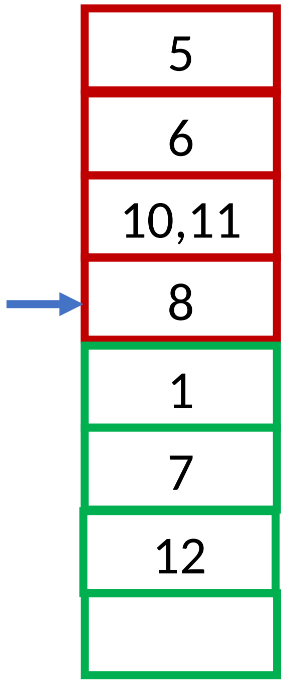



Round = 0

**After split**

# Linear Hashing

6 After all splits

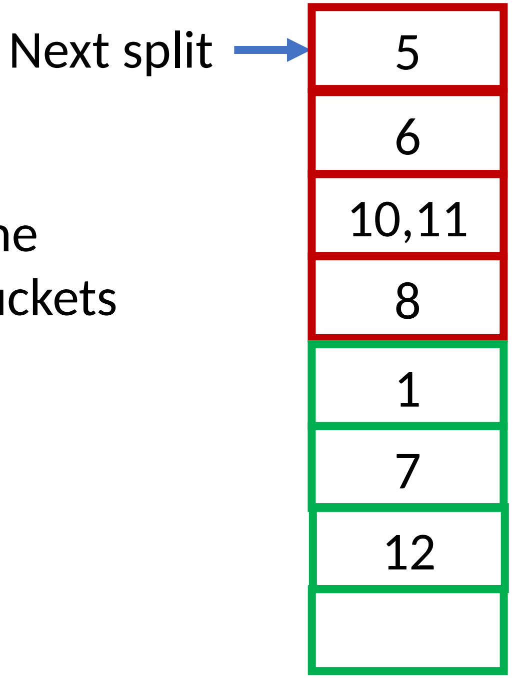



**Round = 0**

7 Final steps at the end of a split round:

- Move next_split pointer back to bucket 0
- Increment round number by 1

I.e., double the number of buckets




**Round = 1**

**H1 and H2 in use**

# Other Issues

- “Skewed data distribution”
  - Good hash function outputs uniformly distributed hash values
  - Typically ‘skewed data distribution’ == hash values are not uniformly distributed
- Space utilization
  - Load factor: number of stored keys vs. capacity
  - Raw utilization: metadata vs. data (key-value) size
- Concurrency
  - Global latch – low concurrency
  - Per-bucket latch
  - Latch-free – resizing is non-trivial

# Summary

- Hash tables
  - Best for point queries, cannot do range search (at least not easily)
  - Hash function maps each key to a bucket
  - Each bucket may store multiple key-value pairs but has limited capacity
- Static hashing: Fixed number of buckets + overflow buckets upon collision
  - Collision: H(K1) == H(K2) but K1 != K2
  - Long overflow chains possible, impact performance
- Dynamic approaches
  - Extendible hashing – use directory to avoid overflow buckets
  - Linear hashing – avoid directory, split buckets round-robin
  - Adaptations for in-memory use
- Other issues: skewness, space utilization and concurrency