

# Searching and Hashing

# Sequential Search

**Property:** Sequential search (array implementation) uses  $N+1$  comparisons for an unsuccessful search (always).

**Unsuccessful Search:**

→  $\Theta(n)$

**Successful Search:** *item* is in any location of the array

→  $O(n)$

→  $\Omega(1)$

**Property:** Sequential search (array implementation) uses about  $n/2$  comparisons for a successful search (on the average).

$$\frac{\sum_{i=1}^n i}{n} = \frac{(n^2 + n) / 2}{n}$$

**Average-Case:** The number of key comparisons  $1, 2, \dots, n$

→  $O(n)$

# Binary Search – Analysis

- **Property:** Binary search never uses more than  $\log_2 N + 1$  comparisons for either successful or unsuccessful search.
- For an unsuccessful search:
  - The number of iterations in the loop is  $\lfloor \log_2 n \rfloor + 1$   
 $\rightarrow \Theta(\log_2 n)$
- For a successful search:
  - The number of iterations is  $\lfloor \log_2 n \rfloor + 1$   $\rightarrow O(\log_2 n)$   
 $\rightarrow \Omega(1)$

– **Average-Case:** The avg. # of iterations  $< \log_2 n \rightarrow O(\log_2 n)$

0 1 2 3 4 5 6 7  $\leftarrow$  an array with size 8

3 2 3 1 3 2 3 4  $\leftarrow$  # of iterations

The average # of iterations =  $21/8 < \log_2 8$

$$T(n) = T(n/2) + 1 \text{ with } T(1) = 1$$

$$\rightarrow O(\log_2 n)$$

# Interpolation Search

- One improvement possible in binary search is to try to guess more precisely where the key being sought falls within the current interval of interest (rather than blindly using the middle element at each step), i.e., looking up a number in telephone directory.
- Note that  $x = (l+r) \text{ div } 2$  is derived from the expression
$$x = l + \frac{1}{2} (r - l)$$
- In the interpolation search
$$x = l + [(v - a[l].key) / (a[r].key - a[l].key)] * (r - l)$$
might be a better guess.
- This assumes numerical evenly distributed key values.

# Interpolation Search

## Example:

A A A C E E E G H I L M N P R S X

- If look for  $v = M$ , the first table position examined would be 9, since

$$x = l + [(v - a[l].key) / (a[r].key - a[l].key)] * (r-l)$$

$$x = 1 + [(13 - 1) / (24 - 1)] * (17-1) = 9.3...$$

I L M N P R S X

M N P R S X

**Property:** Interpolation search uses fewer than  $lglgn + 1$  comparisons for both successful and unsuccessful search, in files of **random** keys.  $\rightarrow O(lglgn)$

$$lglgN < 5 \text{ if } N = 10^9.$$

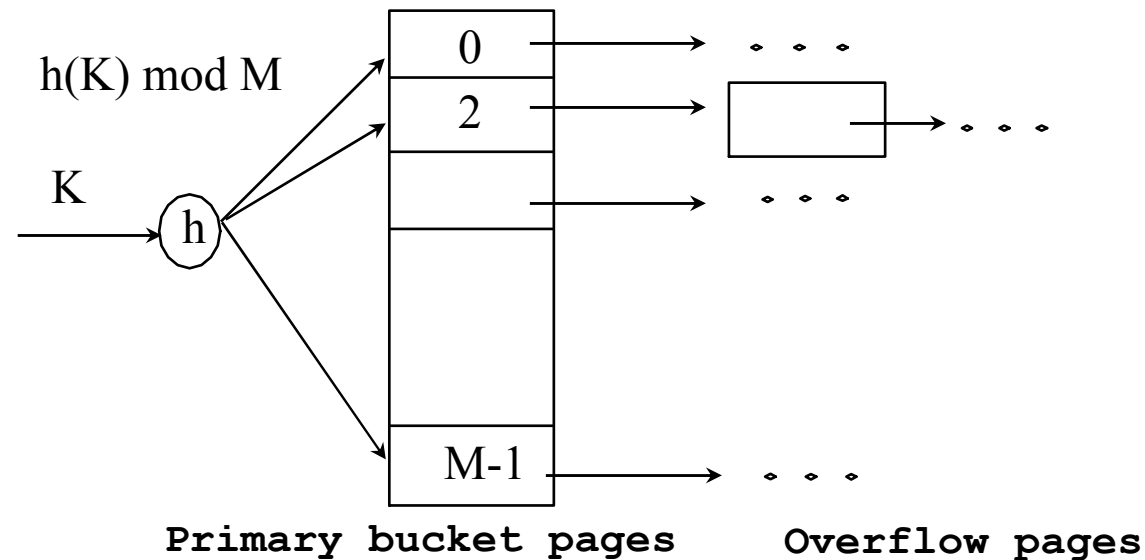
**Suggestion:** Interpolation search certainly should be considered for **large files**, for applications where **comparisons** are particularly **expensive**, or for **external methods** where very high access costs are involved.

# Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- The file blocks are divided into **M equal-sized buckets**, numbered  $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$ . Typically, a bucket corresponds to one (or a fixed number of) disk block.
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**,  $h(K)$ .
- The record with hash key value  $K$  is stored in  $\text{bucket}_i$ , where  $i=h(K)$ .
- **Hash function** is used to locate records for **access, insertion** as well as **deletion**.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

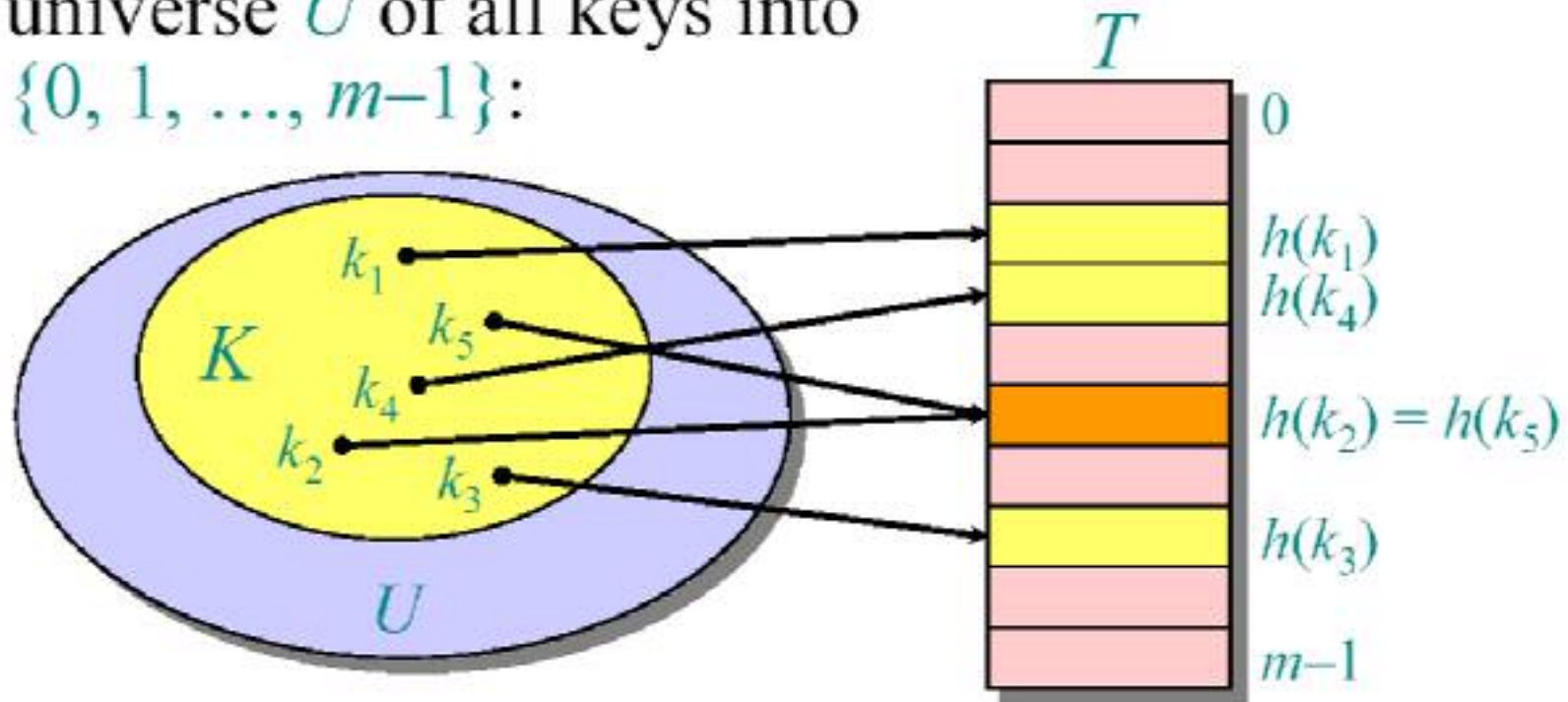
# Static Hashing

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



# Hash functions

**Solution:** Use a *hash function*  $h$  to map the universe  $U$  of all keys into  $\{0, 1, \dots, m-1\}$ :



When a record to be inserted maps to an already occupied slot in  $T$ , a *collision* occurs.

# Static Hashing

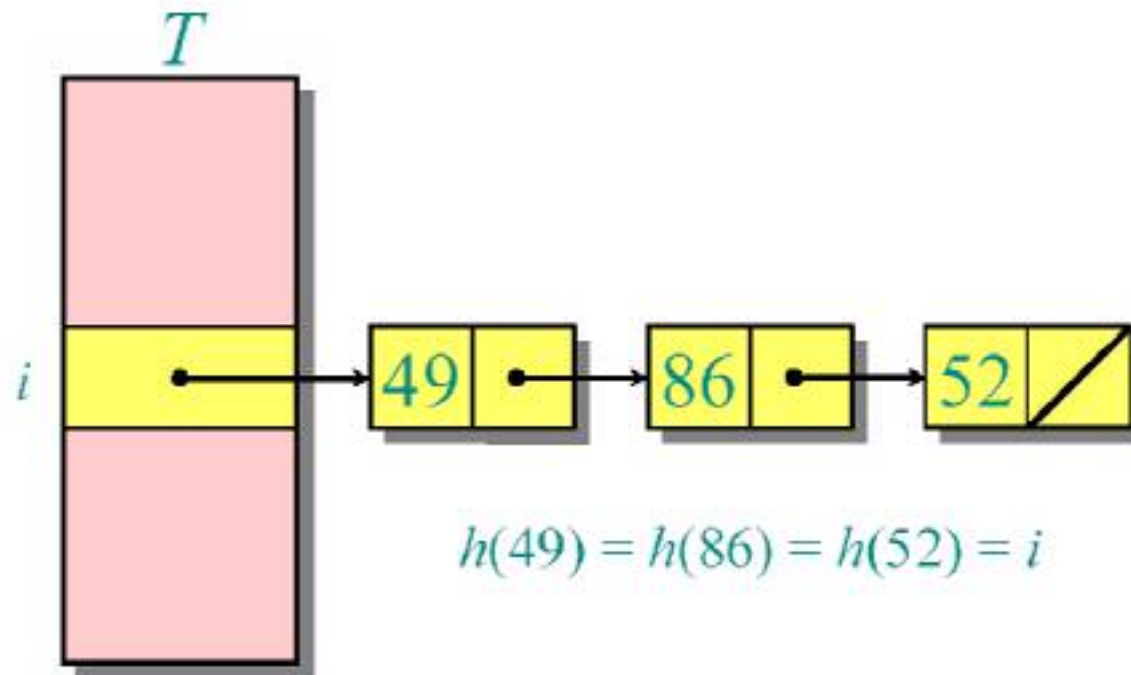
- One of the file fields is designated to be the hash key,  $K$ , of the file.
- **Collisions** occur when a new record hashes to a bucket that is already full.
- An **overflow file** is kept for storing such records. Overflow records that hash to each bucket can be linked together.
- To **reduce overflow records**, a hash file is typically kept 70-80% full.
- The hash function  $h$  should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.

## Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

# Resolving collisions by chaining

- Records in the same slot are linked into a list.



# Analysis of chaining

We make the assumption of *simple uniform hashing*:

- Each key  $k \in K$  of keys is equally likely to be hashed to any slot of table  $T$ , independent of where other keys are hashed.

Let  $n$  be the number of keys in the table, and let  $m$  be the number of slots.

Define the *load factor* of  $T$  to be

$$\alpha = n/m$$

= average number of keys per slot.

# Search cost

Expected time to search for a record with a given key =  $\Theta(1 + \alpha)$ .

*apply hash  
function and  
access slot*

*search  
the list*

Expected search time =  $\Theta(1)$  if  $\alpha = O(1)$ ,  
or equivalently, if  $n = O(m)$ .

# Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

- The probe sequence  $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$  should be a permutation of  $\{0, 1, \dots, m-1\}$ .
- The table may fill up, and deletion is difficult (but not impossible).

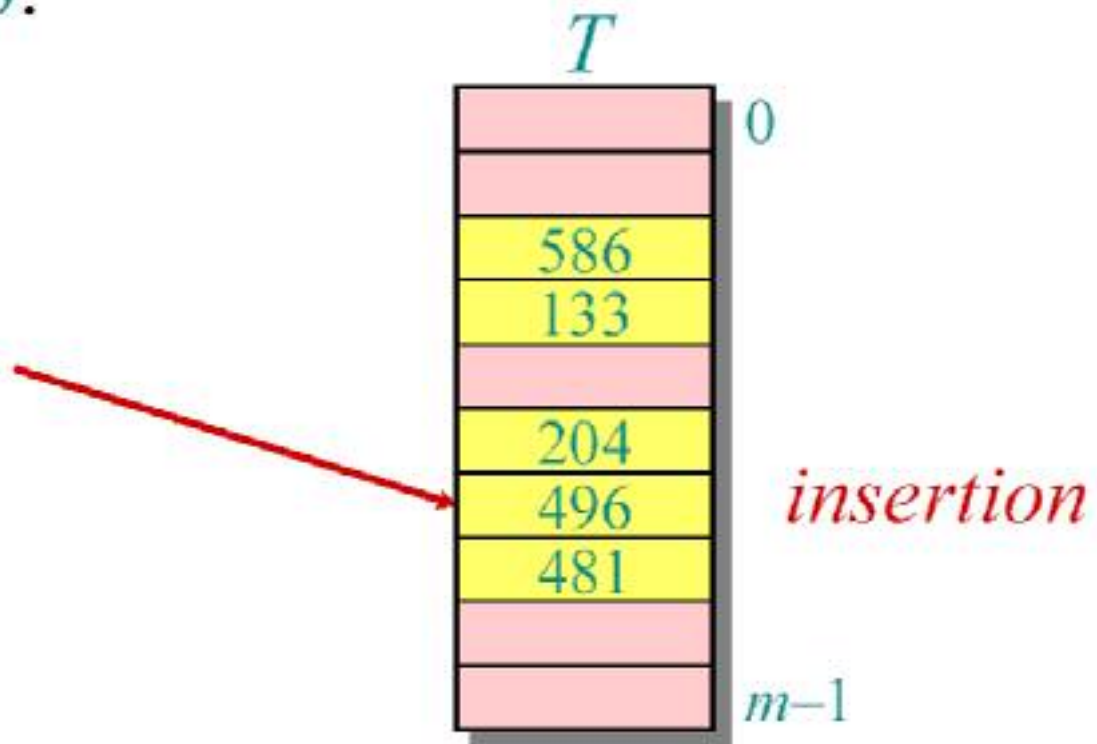
# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496,0)$

1. Probe  $h(496,1)$

2. Probe  $h(496,2)$



# Example of open addressing

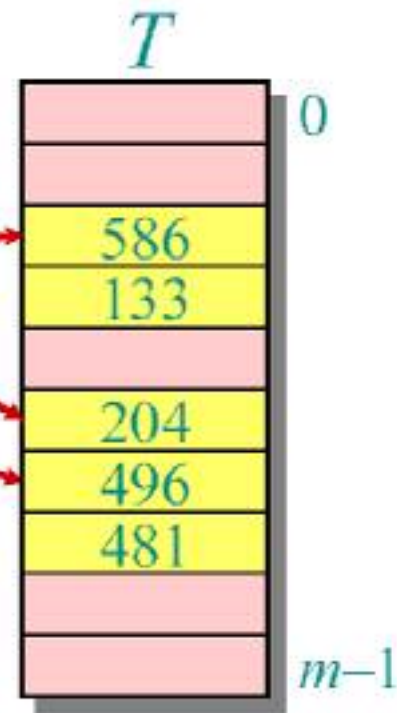
Search for key  $k = 496$ :

0. Probe  $h(496,0)$

1. Probe  $h(496,1)$

2. Probe  $h(496,2)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.



# Probing strategies

## Linear probing:

Given an ordinary hash function  $h'(k)$ , linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m.$$

This method, though simple, suffers from **primary clustering**, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

# Linear Probing: Example

0	
1	
2	
3	3
4	13
5	5
6	6
7	23
8	15
9	

$$h(k,i) = (h'(k) + i) \bmod M$$

Ex:  $M = 10$

Input =  $\langle 5, 3, 6, 13, 23, 15 \rangle$

$$h(3,0) = (h'(3) + 0) \bmod 10 = 3$$

$$h(13,0) = (h'(13) + 0) \bmod 10 = 3 ?$$

$$h(13,1) = (h'(13) + 1) \bmod 10 = 4$$

$$h(5,0) = (h'(5) + 0) \bmod 10 = 5$$

$$h(6,0) = (h'(6) + 0) \bmod 10 = 6$$

$$h(23,0) = (h'(23) + 0) \bmod 10 = 3 ?$$

$$h(23,1) = (h'(23) + 1) \bmod 10 = 4 ?$$

---

$$h(23,4) = (h'(23) + 4) \bmod 10 = 7$$

# Quadratic Probing

- Uses a hash function of the form

$$h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod M$$

Where  $h'$  is an auxiliary hash function,  $c_1$  and  $c_2 \neq 0$  are auxiliary constants and  $i = 0, 1, \dots, M-1$ .

- Initial position probed is  $T[h'(k)]$ .
- If two keys have the same initial probe position, then their probe sequences are the same, since  $h(k_1,0) = h(k_2,0)$  implies  $h(k_1,i) = h(k_2,i)$ . This leads to a milder form of clustering, **secondary clustering**.

# Quadratic Probing: Example

0	12
1	
2	2
3	3
4	
5	22
6	
7	
8	18
9	

$$h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod M$$

**Ex:**  $M = 10, c_1 = 2, c_2 = 1$

Input =  $\langle 2, 3, 22, 12, 18 \rangle$

$$h(2,0) = (h'(2) + 2*0 + 1*0) \bmod 10 = 2$$

$$h(3,0) = (h'(3) + 2*0 + 1*0) \bmod 10 = 3$$

$$h(22,0) = (h'(22) + 2*0 + 1*0) \bmod 10 = 2 ?$$

$$h(22,1) = (h'(22) + 2*1 + 1*1) \bmod 10 = 5$$

$$h(12,0) = (h'(12) + 2*0 + 1*0) \bmod 10 = 2 ?$$

$$h(12,1) = (h'(12) + 2*1 + 1*1) \bmod 10 = 5 ?$$

$$h(12,2) = (h'(12) + 2*2 + 1*4) \bmod 10 = 0$$

$$h(18,0) = (h'(18) + 2*0 + 1*0) \bmod 10 = 8$$

# Probing strategies

## Double hashing

Given two ordinary hash functions  $h_1(k)$  and  $h_2(k)$ , double hashing uses the hash function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

This method generally produces excellent results, but  $h_2(k)$  must be relatively prime to  $m$ . One way is to make  $m$  a power of 2 and design  $h_2(k)$  to produce only odd numbers.

Double hashing represents an improvement, since each  $(h_1(k), h_2(k))$  pair yields a distinct probe sequence.

0	
1	79
2	
3	
4	
5	96
6	
7	
8	
9	14
10	
11	
12	

# Double Hashing

$$h(k,i) = (h_1(k) + i * h_2(k)) \bmod M$$

Either  $M = 2^d$  and design  $h_2$  so that it produces odd numbers or  $M$  is prime and  $h_2$  produces positive integer less than  $M$ .

Ex:  $M = 13$ ,  $h_1(k) = k \bmod M$ ,  $h_2(k) = 1 + (k \bmod M')$ .

Input =  $\langle 96, 79, 14 \rangle$

$$h_1(96,0) = 96 \bmod 13 = 5$$

$$h_1(79,0) = 79 \bmod 13 = 1$$

$$h_1(14,0) = 14 \bmod 13 = 1 ?$$

$$h_2(14) = 1 + 14 \bmod 11 = 4$$

$$h(14, 1) = (h_1(14) + i * h_2(14)) \bmod 13 = 1 + 1 * 4 = 5 ?$$

$$h(14, 2) = (1 + 2 * 4) \bmod 13 = 9$$

# Analysis of open addressing

We make the assumption of *uniform hashing*:

- Each key is equally likely to have any one of the  $m!$  permutations as its probe sequence.

**Theorem.** Given an open-addressed hash table with load factor  $\alpha = n/m < 1$ , the expected number of probes in an unsuccessful search is at most  $1/(1-\alpha)$ .

# Proof of the theorem

*Proof.*

- At least one probe is always necessary.
- With probability  $n/m$ , the first probe hits an occupied slot, and a second probe is necessary.
- With probability  $(n-1)/(m-1)$ , the second probe hits an occupied slot, and a third probe is necessary.
- With probability  $(n-2)/(m-2)$ , the third probe hits an occupied slot, etc.

Observe that  $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$  for  $i = 1, 2, \dots, n$ .

# Proof (continued)

Therefore, the expected number of probes is

$$\begin{aligned} & 1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \dots \left( 1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right) \\ & \leq 1 + \alpha (1 + \alpha (1 + \alpha (\dots (1 + \alpha) \dots))) \\ & \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots \\ & = \sum_{i=0}^{\infty} \alpha^i \\ & = \frac{1}{1-\alpha} \cdot \blacksquare \end{aligned}$$

*The textbook has a more rigorous proof.*

# Implications of the theorem

- If  $\alpha$  is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is  $1/(1-0.5) = 2$ .
- If the table is 90% full, then the expected number of probes is  $1/(1-0.9) = 10$ .

# Dynamic Hashing Techniques

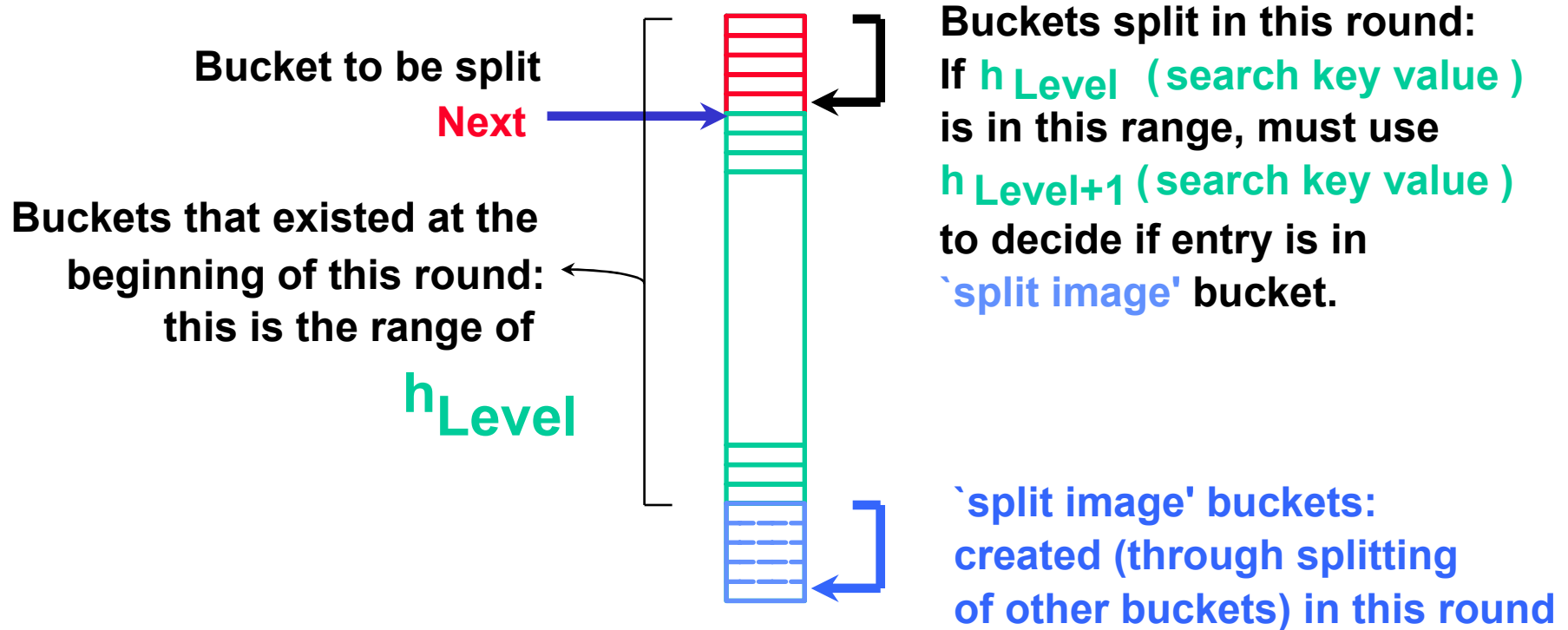
- Hashing techniques are adapted to allow the **dynamic growth** and **shrinking** of the number of file records.
- These techniques include the following: **dynamic hashing** , **extendible hashing** , and **linear hashing**.
- These hashing techniques use the binary representation of the hash value  $h(K)$ .
- In **dynamic hashing** the directory is a binary tree.
- In **extendible hashing** the directory is an array of size  $2^d$  where  $d$  is called the **global depth**.

# Linear Hashing

- This is another dynamic hashing scheme, an alternative to [Extendible Hashing](#).
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- Idea: Use a family of hash functions  $\mathbf{h}_0, \mathbf{h}_1, \mathbf{h}_2, \dots$ 
  - $\mathbf{h}_i(\text{key}) = \mathbf{h}(\text{key}) \bmod (2^i N)$ ;  $N = \text{initial \# buckets}$
  - $\mathbf{h}$  is some hash function
  - If  $N = 2^{d_0}$ , for some  $d_0$ ,  $\mathbf{h}_i$  consists of applying  $\mathbf{h}$  and looking at the last  $d_i$  bits, where  $d_i = d_0 + i$ .
  - $\mathbf{h}_{i+1}$  doubles the range of  $\mathbf{h}_i$  (similar to directory doubling)

# Overview of LH File

- In the middle of a round.




# A sample problem in LH

00:	56	64	
01:			
10:			
11:	67	43	79

15	27	19
----	----	----



56: 011 1000

67: 100 0011

43: 010 1011

79: 100 1111

15: 000 1111

19: 001 0011

27: 001 1011

64: 100 0000

12: 000 1100

33: 010 0001

57: 011 1001


65: 100 0001

# Example: LH

00:	56	64	12
01:	33		
10:			
11:	67	43	79

15	27	19
----	----	----



56: 011 1000

67: 100 0011

43: 010 1011

79: 100 1111

15: 000 1111

19: 001 0011

27: 001 1011

64: 100 0000

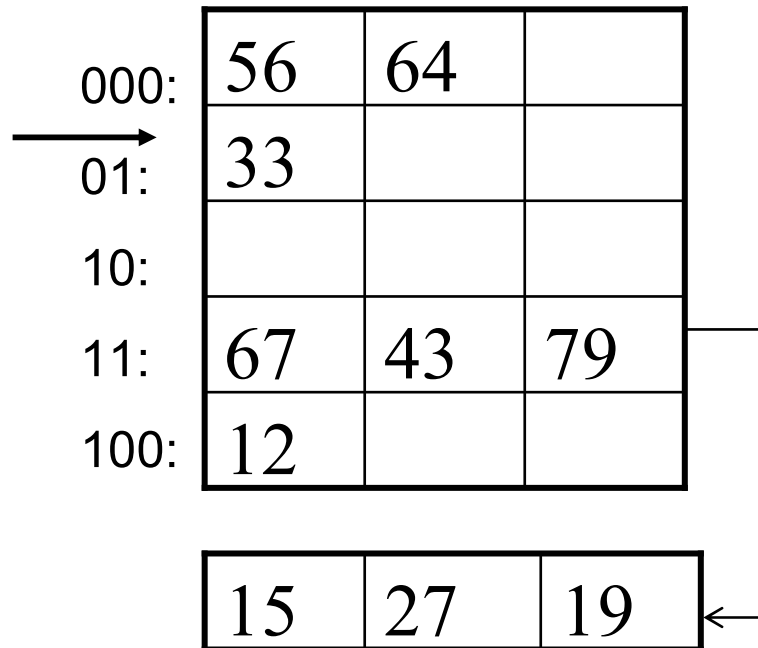
12: 000 1100

33: 010 0001

57: 011 1001

65: 100 0001

# Example: LH



56: 011 1000

67: 100 0011

43: 010 1011

79: 100 1111

15: 000 1111

19: 001 0011

27: 001 1011

64: 100 0000

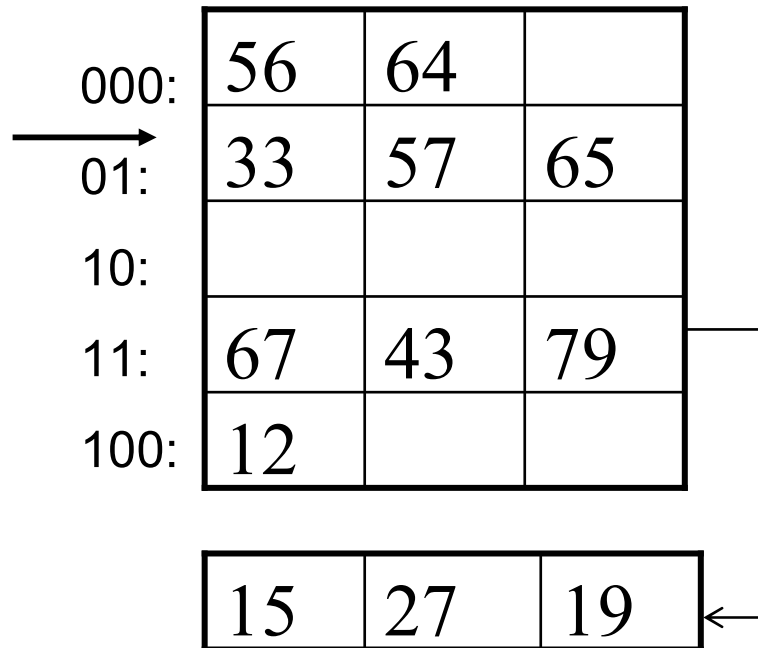
12: 000 1100

33: 010 0001

57: 011 1001

65: 100 0001

# Example: LH



56: 011 1000

67: 100 0011

43: 010 1011

79: 100 1111

15: 000 1111

19: 001 0011

27: 001 1011

64: 100 0000

12: 000 1100

33: 010 0001

57: 011 1001

65: 100 0001

# Example: LH

000:	56	64	
001:	33	57	65
→ 10:			
11:	67	43	79
100:	12		
101			

15	27	19
----	----	----

56: 011 1000

67: 100 0011

43: 010 1011

79: 100 1111

15: 000 1111

19: 001 0011

27: 001 1011

64: 100 0000

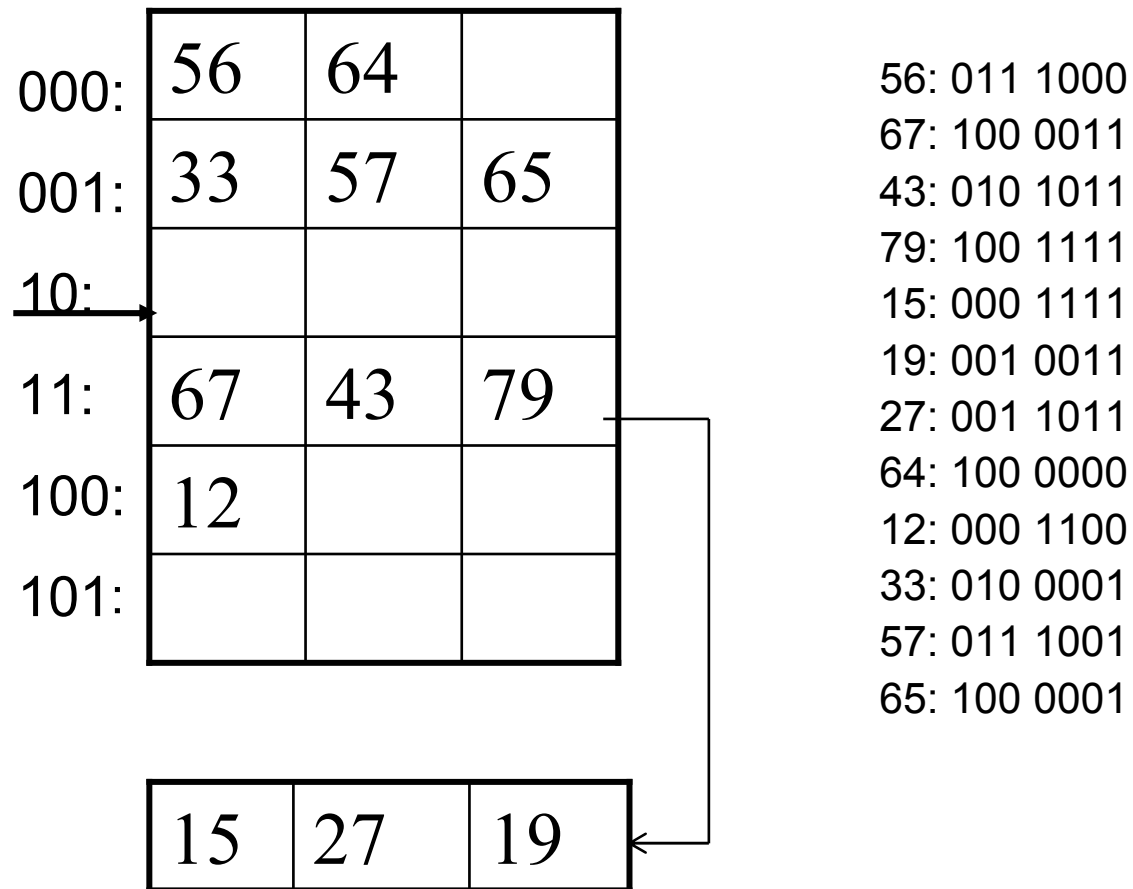
12: 000 1100

33: 010 0001

57: 011 1001

65: 100 0001

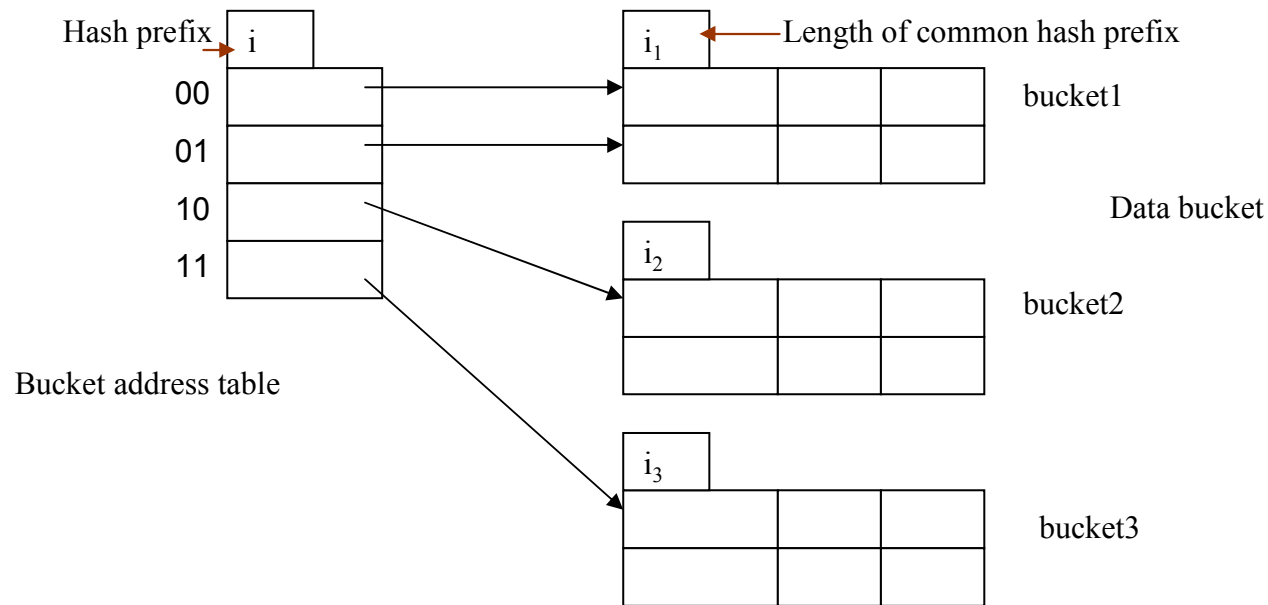
# Example: LH



# LH

- **Linear Hashing** avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - **Space utilization** could be lower than Extendible Hashing, since splits not concentrated on 'dense' data areas.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!

# General Extendable Hash Structure

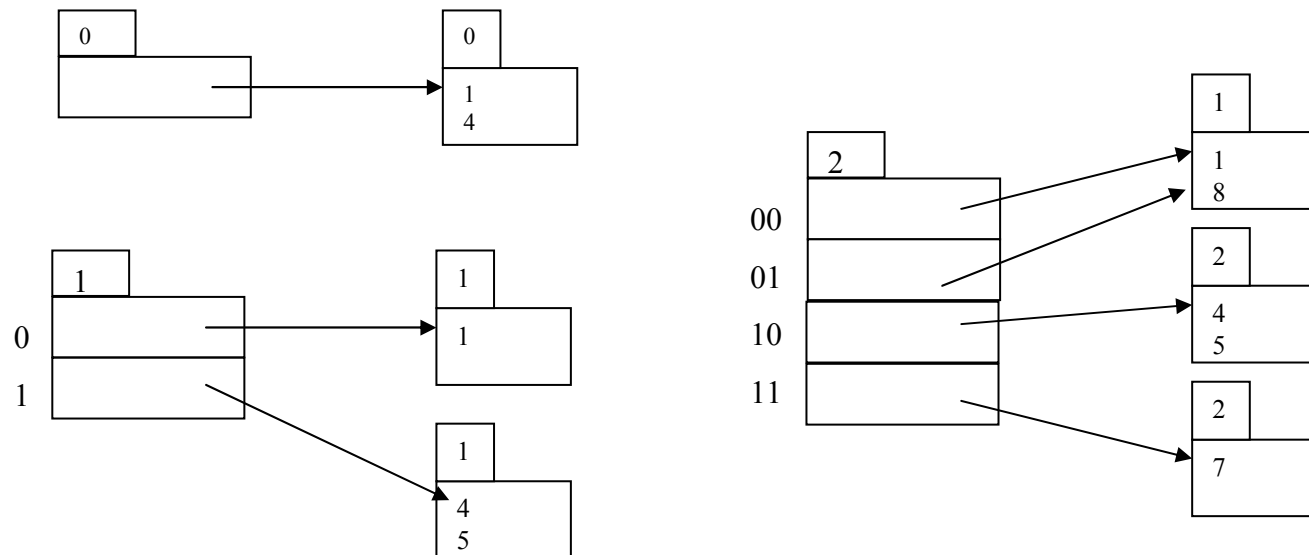


- Hash function returns **b** bits
- Only the prefix **i** bits are used to hash the item
- There are  $2^i$  entries in the bucket address table
- Let  $i_j$  be the length of the common hash prefix for data bucket **j**, there is  $2^{(i-i_j)}$  entries in bucket address table points to **j**.
- In this structure,  $i_2 = i_3 = i$ ,  $2^0 = 1$  entry, whereas  $i_1 = i - 1$ ,  $2^1 = 2$  entries.

# Example: Extendable Hashing

- Example 5: Suppose the hash function is  $h(x) = x \bmod 8$  and each bucket can hold at most two records. Show the extendable hash structure after inserting 1, 4, 5, 7, 8, 2, 20.

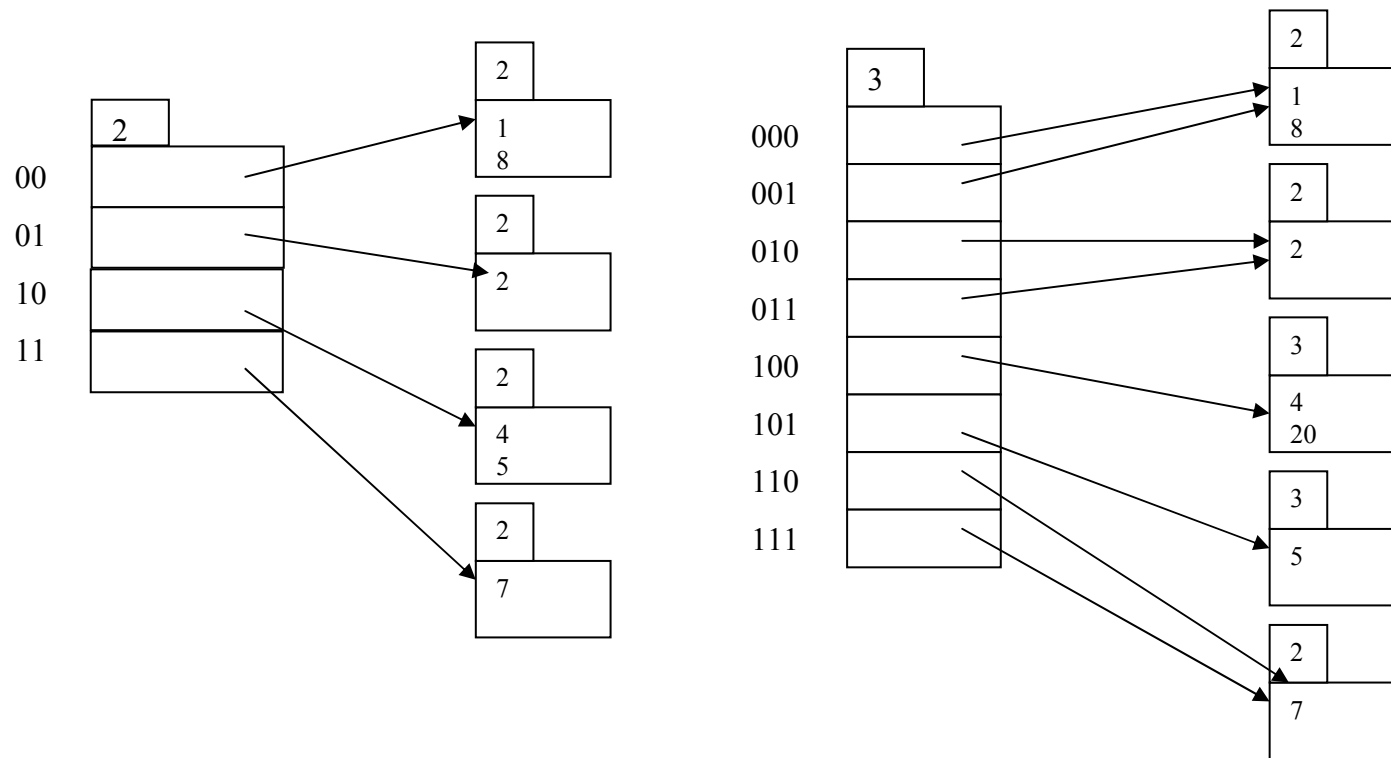
1	4	5	7	8	2	20
001	100	101	111	000	010	100



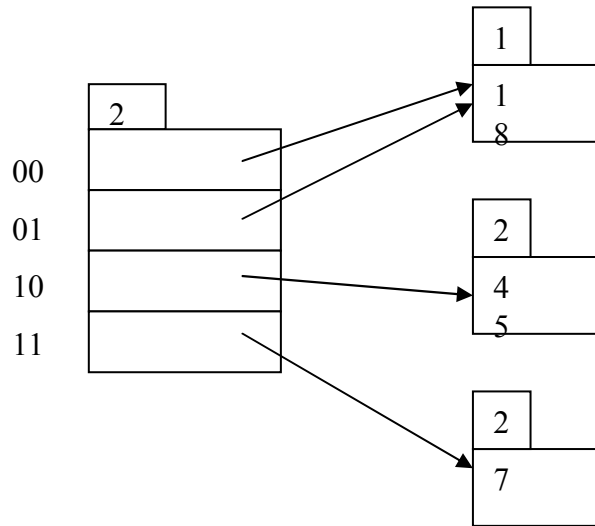
# Example: Extendable Hashing

inserting 1, 4, 5, 7, 8, 2, 20

1	4	5	7	8	2	20
001	100	101	111	000	010	100



# Example: Extendable Hashing



Suppose the hash function  $h(x) = x \bmod 8$ , each bucket can hold at most 2 records. Show the structure after inserting “20”

# Example: Extendable Hashing

