EECS 204002

Data Structures 資料結構

Prof. REN-SONG TSAY 蔡仁松 教授

NTHU

CH. 8 HASHING

8.1

Motivation

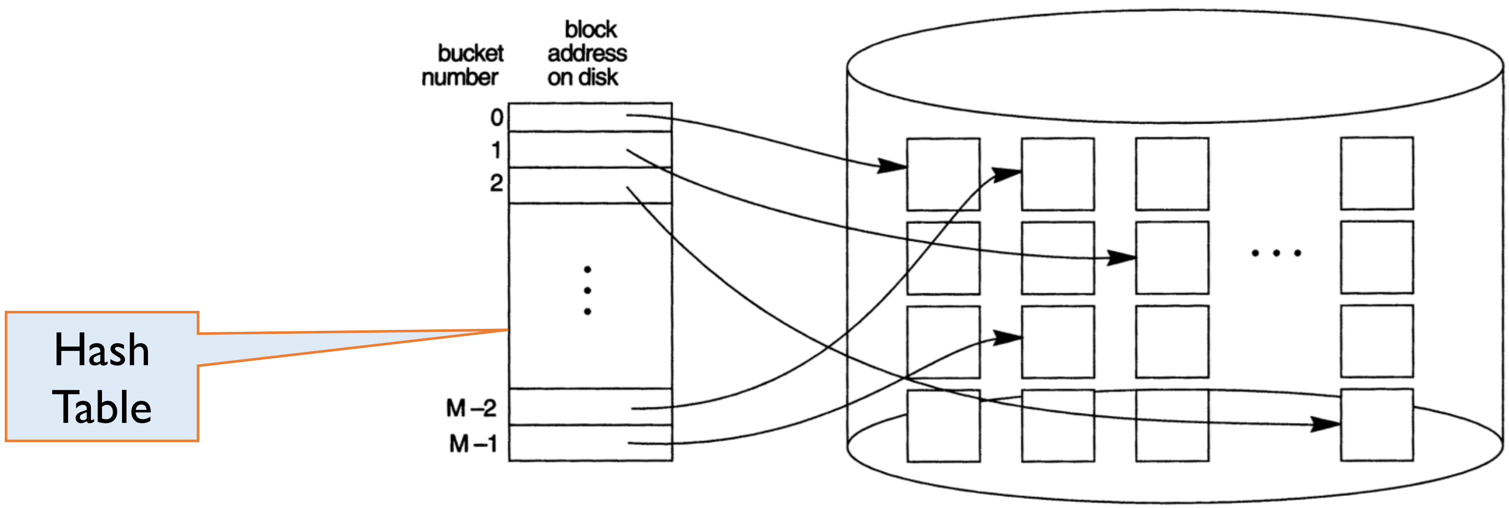
- Operations in a dictionary
  - Get, Insert and Delete
- **Binary search tree**
  - Get, Insert and Delete take $O(n)$
- **Balanced binary search tree (AVL tree)**
  - Get, Insert and Delete take $O(\log n)$
- **Hashing**
  - Get, Insert and Delete take $O(1)$
  - Static hashing
  - Dynamic hashing

8.2

Static Hashing

Overview of Hashing

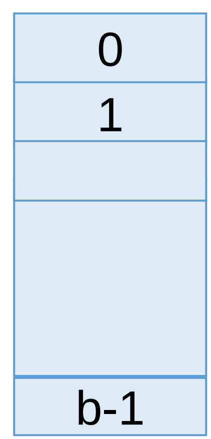
- The file blocks are divided into M equal-sized *buckets*
- The record with hash key value K is stored in bucket i
  - $i = h(K)$, and h is the *hashing function*



8.2.1

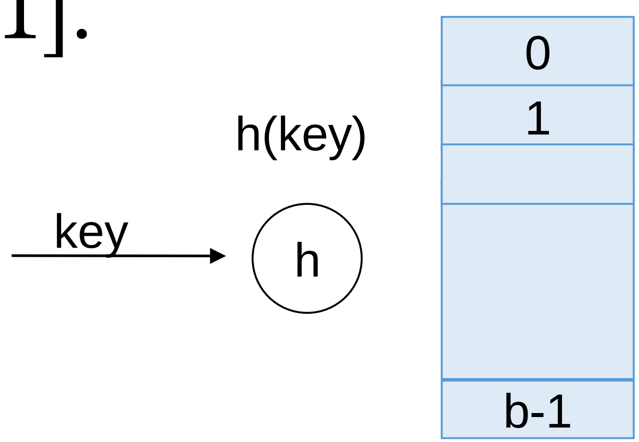
Hash Tables

- **Hash table** (ht) pair=[record, key]
  - A container stores dictionary pairs.
- Hash table is partitioned into $\boldsymbol{b}$ **buckets**
  - $ht[0], ht[1], \dots, ht[b-1]$
  - Each bucket holds $\boldsymbol{s}$ dictionary pairs (**slots**)
    - Usually $s = 1$, i.e. each bucket can hold exactly one pair.



Hash Function

- The *hash (bucket address)* of a pair with key k is determined by a **hash function**, $h(k)$.
- Hash function maps keys into buckets by returning an integer in the range $[0, b-1]$.



Definitions

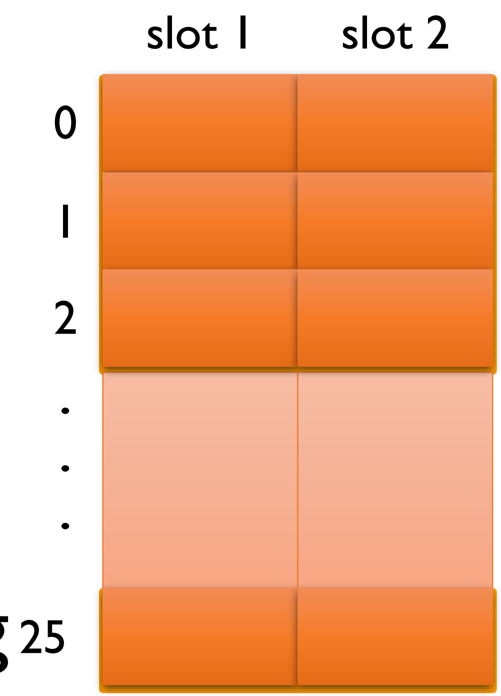
- *Key density* (n/T)
  - n: # of pairs in the table
  - T: Total # of possible keys
- *Loading density* or *loading factor*
  - $\alpha = n/(s \cdot b)$
- Two keys, k_1 and k_2, are said to be *synonyms* w.r.t. h, if $h(k_1) = h(k_2)$.

Definitions

- Many keys might be mapped to the same home bucket (*synonyms*)
- *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.

Example

- Given a set of 8 keys (n=8) {GA, D, A, G, L, A2, A1, A3}.
- Consider a ht with $b = 26$ and $s = 2$.
  - $\alpha = \frac{n}{s \cdot b} = \frac{8}{2*26} = 0.154$
- The hash function maps each key into a bucket using its leading letter.
  - Represent A – Z as $0 - 25$



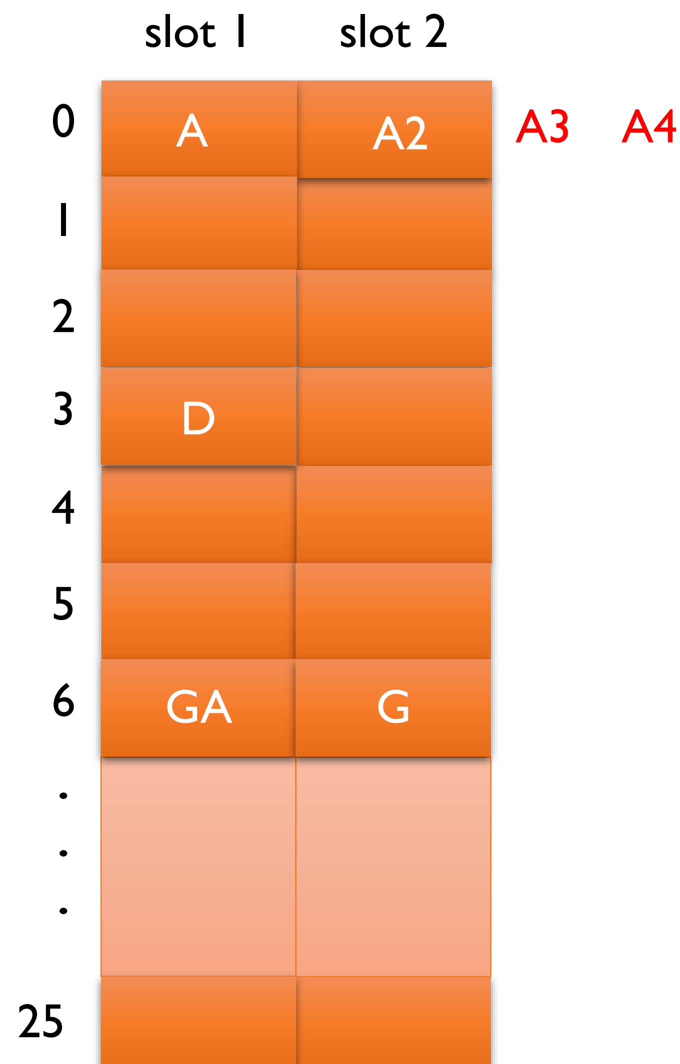
Example (cont'd)

GA, D, A, G, A2 A3 A4

Collision

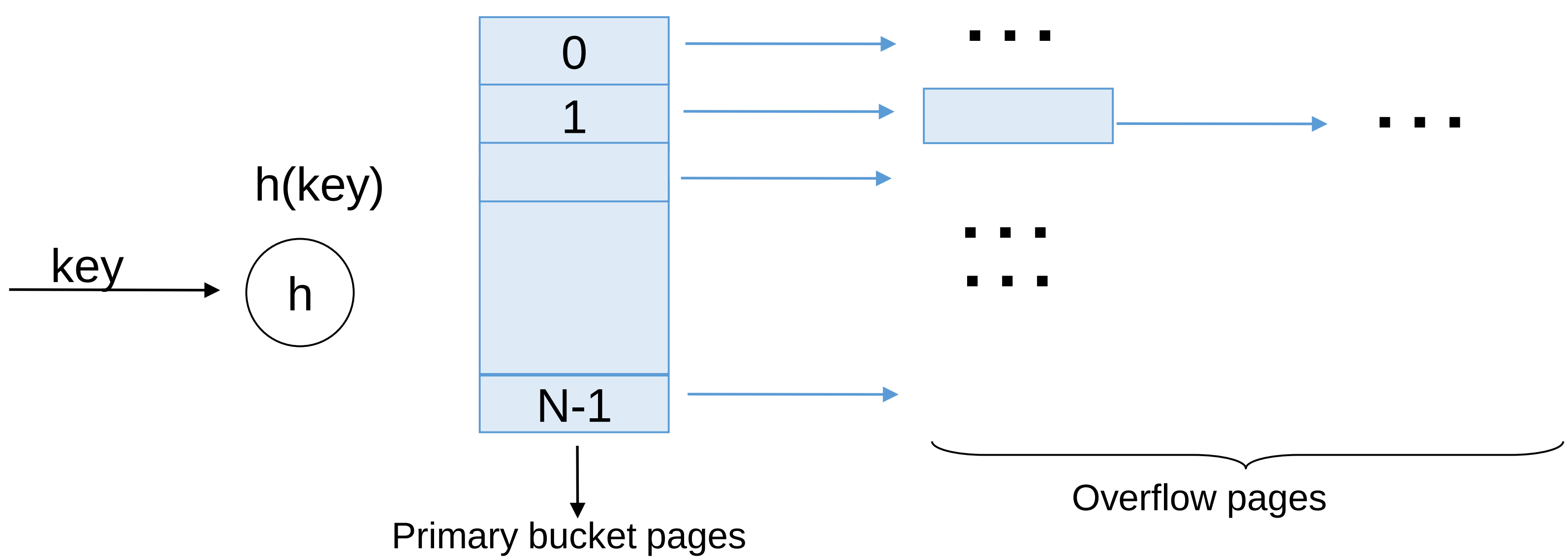
Overflow

mapping	
A	0
D	3
G	6



Overflow

- 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 the # 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**
  - **Few collisions**

8.2.2

Uniform Hash Function

- A hash function that does not result in a biased use of the hash table for **random keys**.
- Given a key k chosen at random, $\text{probability}[h(k) = i\,] = \frac{1}{b}, \forall\, i.$
- 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 (%) operator.
- Bucket address range from 0 to $D - 1$,
  - hash table must have at least $b = D$ buckets.
- Using a prime number for D (see textbook).
- Ex: $h(k = 219) = 219\%8 = 3$

8.2.2.2

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 8 buckets (2^3), we need the middle 3-bits to determine the bucket address

key=219 219^2=47961=1011 $\boxed{101}$1 0101 1001 r=3

h(219)=5

8.2.2.3

Folding

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

k=12320324111220

\+ \+ \+ \+ = 699

8.2.2.4

Digit Analysis

- All the keys in the table are known in advance
- Represent each key as a number in radix r
- Digits having the most skewed distributions are deleted
- Employ the remaining digits
- Example: 100 buckets = 2 digits
  - $m = 10^5$ ➡delete 3 digits

$\boldsymbol{k_1}$=	$\boldsymbol{d_{11}}$	$\boldsymbol{d_{12}}$	…	$\boldsymbol{d_{1n}}$
$\boldsymbol{k_2}$=	$\boldsymbol{d_{21}}$	$\boldsymbol{d_{22}}$	…	$\boldsymbol{d_{2n}}$
…				
$\boldsymbol{k_m}$=	$\boldsymbol{d_{m1}}$	$\boldsymbol{d_{m2}}$	…	$\boldsymbol{d_{mn}}$

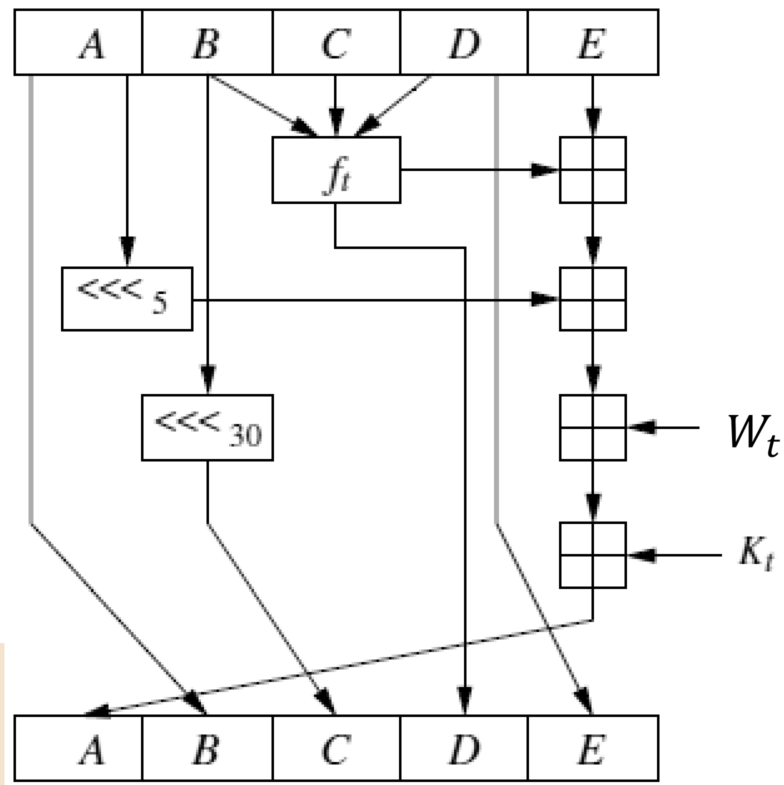
8.2.3

The Secure Hash Algorithm (SHA)

- 4 runs * 20 steps

t	**Step #, $0 \leq t \leq 79$**
f_t	*logical function*
$<<<_k$	Circular left shift k bits
W_t	A 32-bit value derived from M_i
K_t	A constant

$$f_t(B,C,D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq t \leq 19 \\ B \oplus C \oplus D & \text{if } 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq t \leq 59 \\ B \oplus C \oplus D & \text{if } 60 \leq t \leq 79 \end{cases}$$



SHA-1: a Merkle-Damgard Hash Function

- Padding: Given an m-bit message, a single bit "1" is appended as the $(m+1)$-th bit and then $(448-(m+1)) \bmod 512$ (between 0 and 511) zero bits are appended. As a result, the message becomes 64-bit short of being a multiple of 512 bits long.
- Merkle-Damgard Strengthening: append the 64-bit representation of the original length of m, making the result a multiple of 512 bits long.
- Divide the result into 512-bit blocks, denoted by $M_1, M_2, \ldots, M_l$.

m bits	1 bit		64 bits
message	1	000…0	m

SHA-1

- The internal state of SHA-1 is composed of five 32-bit words A, B, C, D and E, used to keep the 160-bit chaining value h_i .
- Initialization: The initial value (h_0) is (in hexadecimal)
  - A_0 = 67452301x
  - B_0 = EFCDAB89x
  - C_0 = 98BADCFEx
  - D_0 = 10325476x
  - E_0 = C3D2E1F0x .
- Compression: For each block, the compression function $h_i = H(h_{i-1}, M_i)$ is applied on the previous value of $h_{i-1} = (A, B, C, D, E)$ and the message block.
- Output: The hash value is the 160-bit value $h_l = (A, B, C, D, E)$.

The Compression Function H

- Divide M_i into 16*32-bit words:
  - $W_0, W_1, W_2, \ldots, W_{15}$.
- for t = 16 to 79 compute $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1$.
  - Remark: The one-bit rotate in computing W_t was not included in SHA, and is the only difference between SHA and SHA-1.
- Set $h_0 = (A_0, B_0, C_0, D_0, E_0)$.
- For t = 0 to 79 do
  - $T = A_t \lll 5 + f_t(B_t, C_t, D_t) + E_t + W_t + K_t$.
  - $E_{t+1} = D_t, D_{t+1} = C_t, \; C_{t+1} = B_t \lll 30, B_{t+1} = A_t, \; A_{t+1} = T$.
- Output $A = A_0 + A_{80}, B = B_0 + B_{80}, C = C_0 + C_{80}, D = D_0 + D_{80}$, and $E = E_0 + E_{80}(modulo\ 232)$.
- The function f_t and the values K_t used above are:

	$f_t(X, Y, Z) =$	$K_t =$
0 ≤ t ≤ 19	XY ∨ (¬X)Z	5A827999
20 ≤ t ≤ 39	X ⊕ Y ⊕ Z	6ED9EBA1
40 ≤ t ≤ 59	XY ∨ XZ ∨ YZ	8F1BBCDC
60 ≤ t ≤ 79	X ⊕ Y ⊕ Z	CA62C1D6

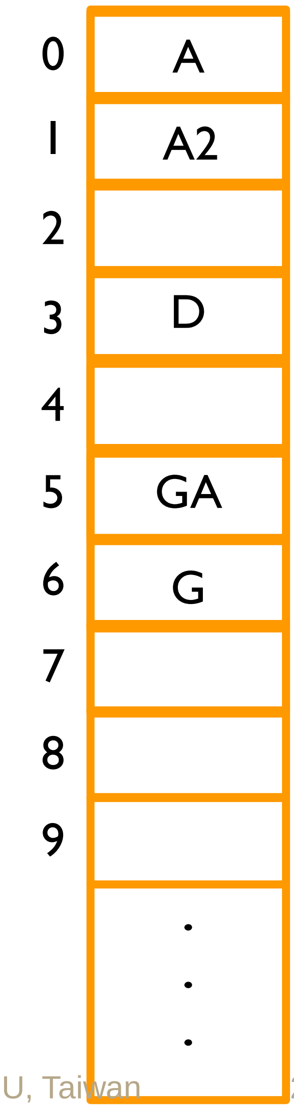
8.2.4

Overflow Handling

- Open addressing
  - Linear probing
  - Quadratic probing
  - Rehashing
  - Random probing
- Chaining

Linear Probing: Insert

- Find the closest unfilled bucket.
- To insert a key k.
  - Compute $h(k)$.
  - Check the hash table buckets in the order $h_t[h(k)], h_t[(h(k)+1)\%b], \ldots, h_t[(h(k)+j)\%b]$ until an empty bucket is found.
  - If no empty bucket is found, double the size of h_t.
- e.g. GA, D, A, G, A2



Linear Probing: Search

- Searching for a key k.
  - Compute $h(k)$.
  - Examine the hash table buckets in the order $h_t[h(k)], h_t[(h(k)+1)\%b], \ \ldots, h_t[(h(k)+j)\%b]$ until:
    - $h_t[(h(k)+j)\%b]$ has the same key. Found!
    - $h_t[(h(k)+j)\%b]$ is empty. Not found!
    - Go back to starting point. Not found!
- Disadvantage:
  - Keys tend to cluster together.

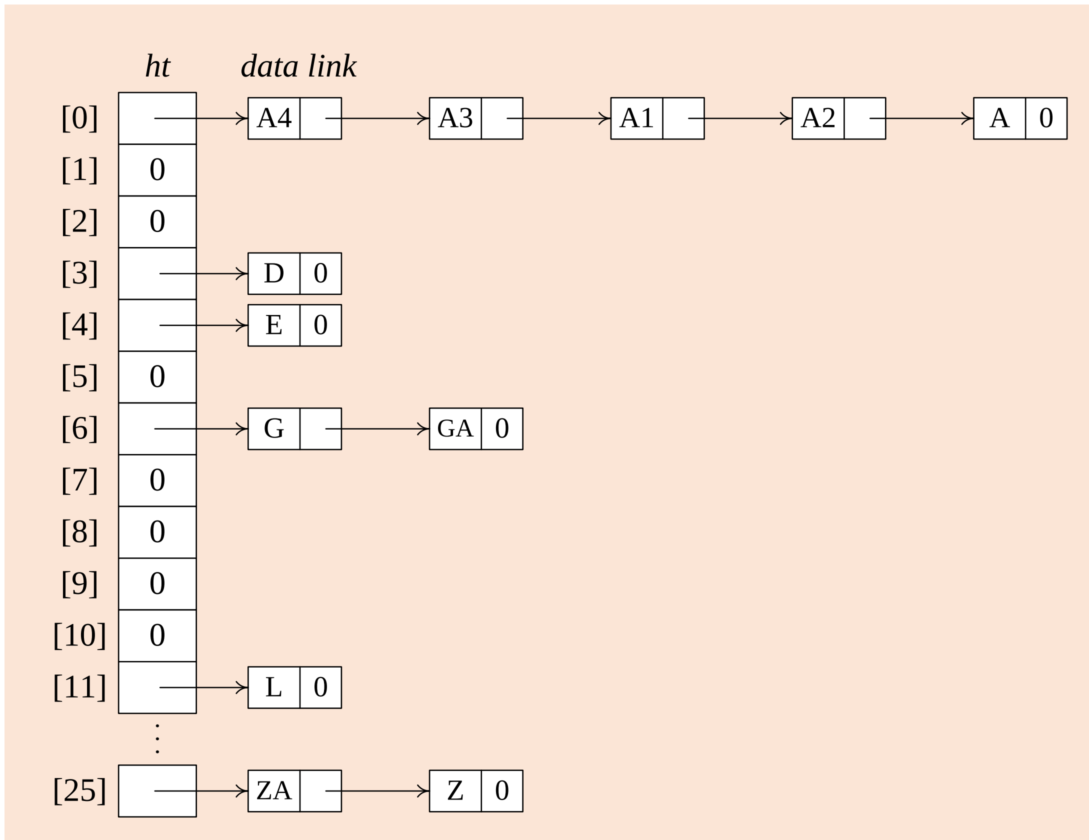
Others

- Quadratic probing:
  - Compute $h(k)$.
  - Examine buckets at $h(k), (h(k) + i^2)\%b$, and $(h(k) - i^2)\%b, 1 \leq i \leq (b-1)/2$.
- Rehashing:
  - A series of hashing functions $h_1, h_2, \ldots, h_n$.
  - Bucket is searched by $h_1, h_2, \ldots, h_n$.

8.2.4.2

Chaining

- Use chained hash table to solve collisions
- Each bucket holds a list of keys (key chain)



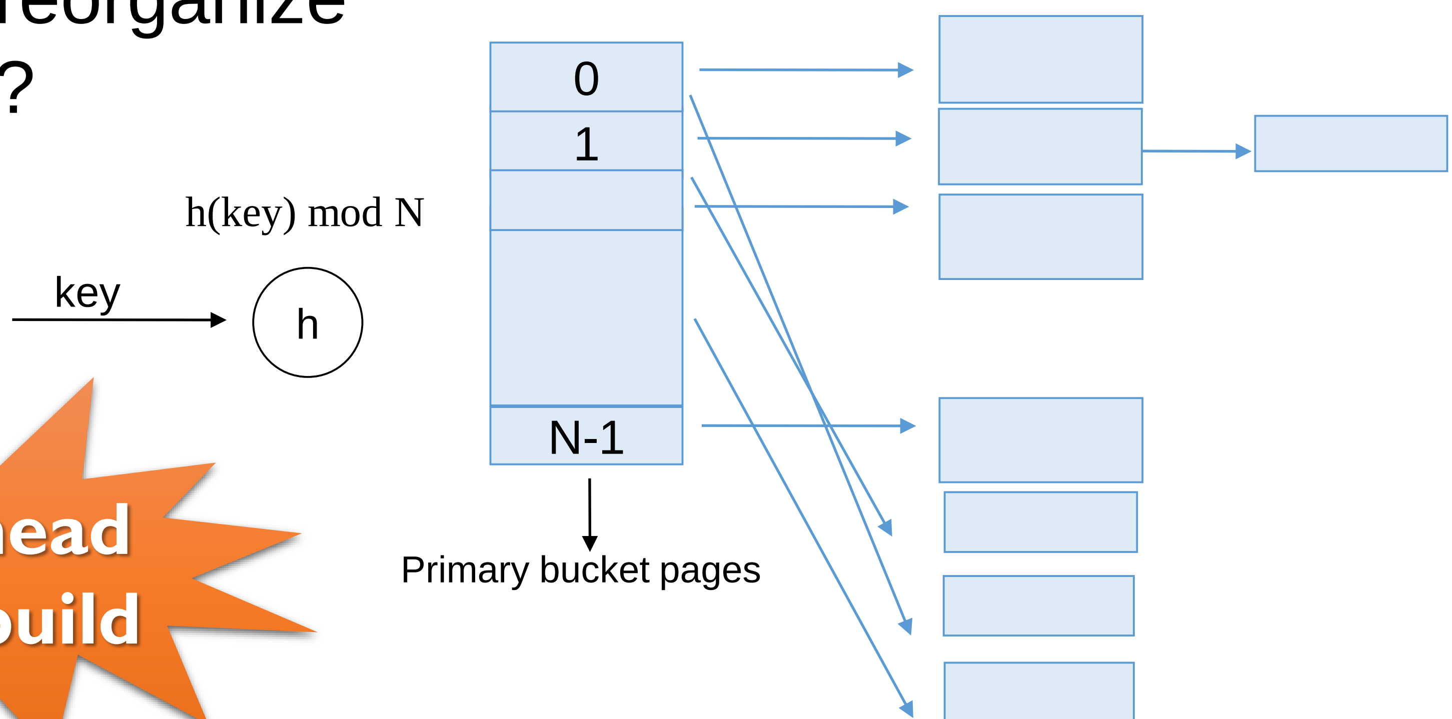
8.3

Dynamic Hashing

8.3.1

Manage Overflow Problems

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



Dynamic Hashing

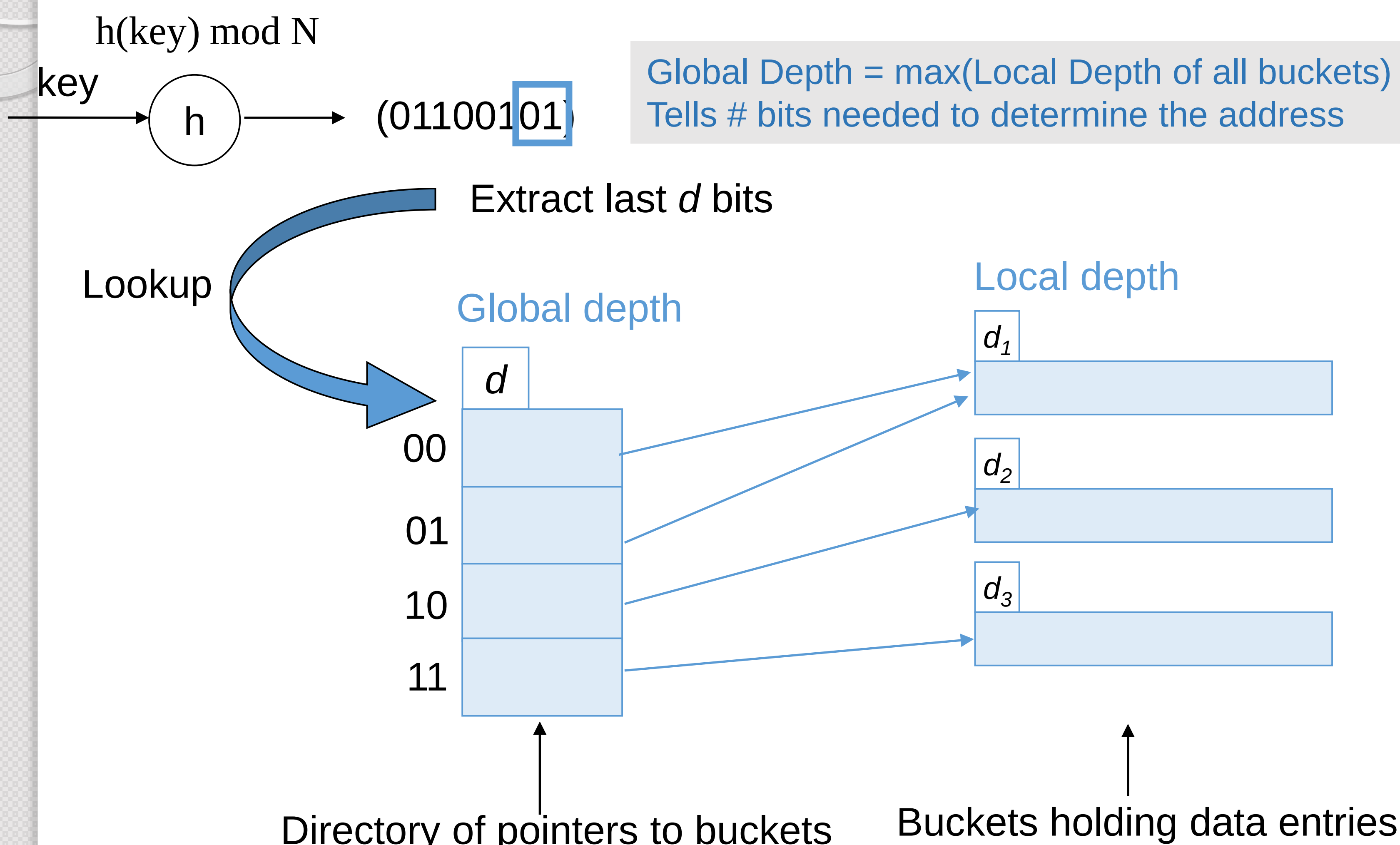
- Also called Extendible Hashing
- *Idea*: Use *directory of pointers to buckets*
  - Use the *binary representation* of the hash value $h(K)$ in order to access a *directory*
  - Double #buckets by doubling the directory
  - Splitting just the bucket that is overflowed!
- Directory is much smaller than bucket file
  - Much cheaper to double the directory
  - Split only the page of data entries. *No overflow page!*

Directory

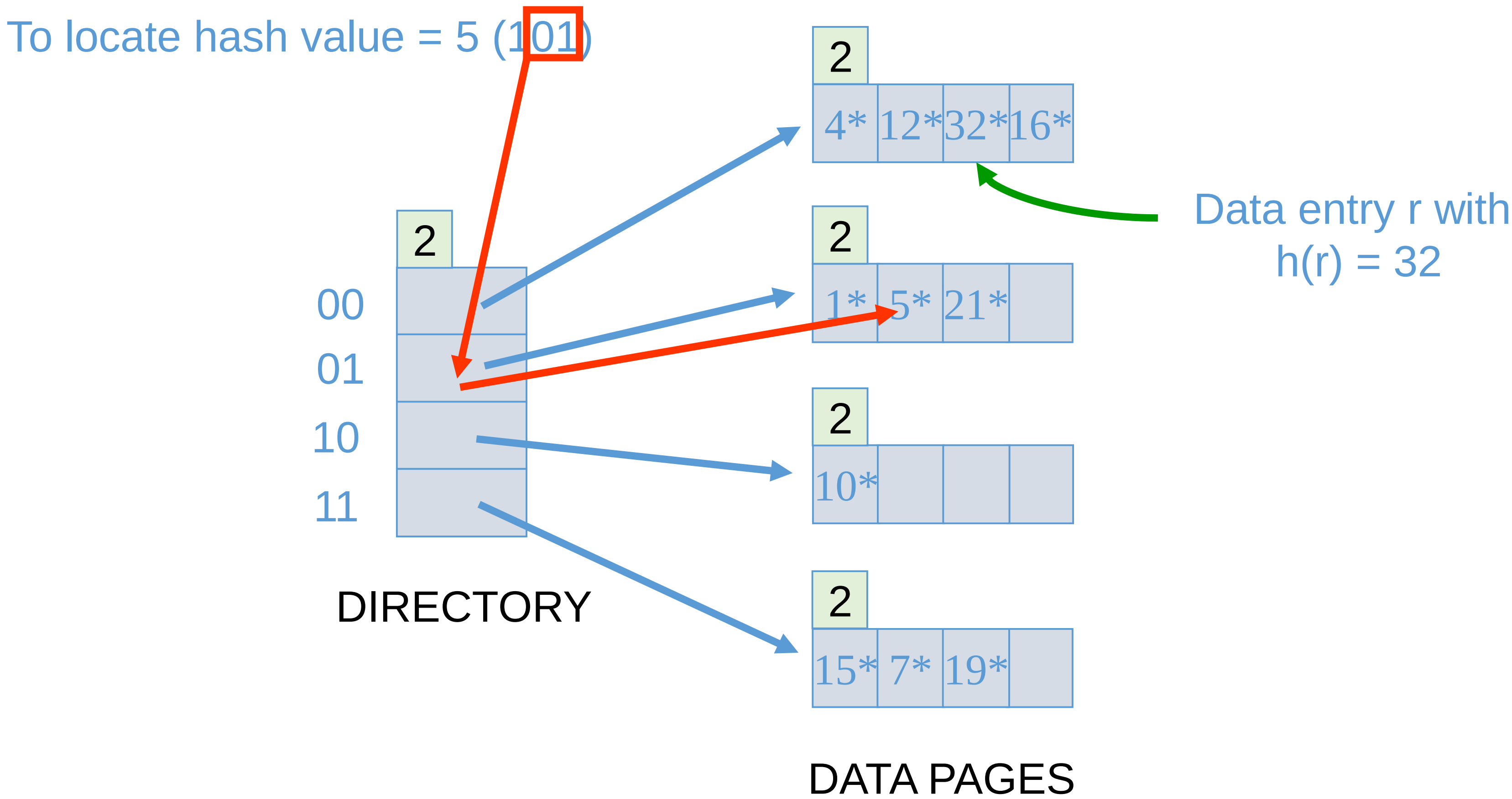
- An array of size 2^d where d is called the *global depth*
- Expand or shrink dynamically
- Entries point to the buckets
  - That contain the stored records
  - When an insertion in a bucket that is full the bucket splits into two buckets
    - The records are redistributed among the two buckets
- Update directory appropriately

Example of Dynamic Hashing

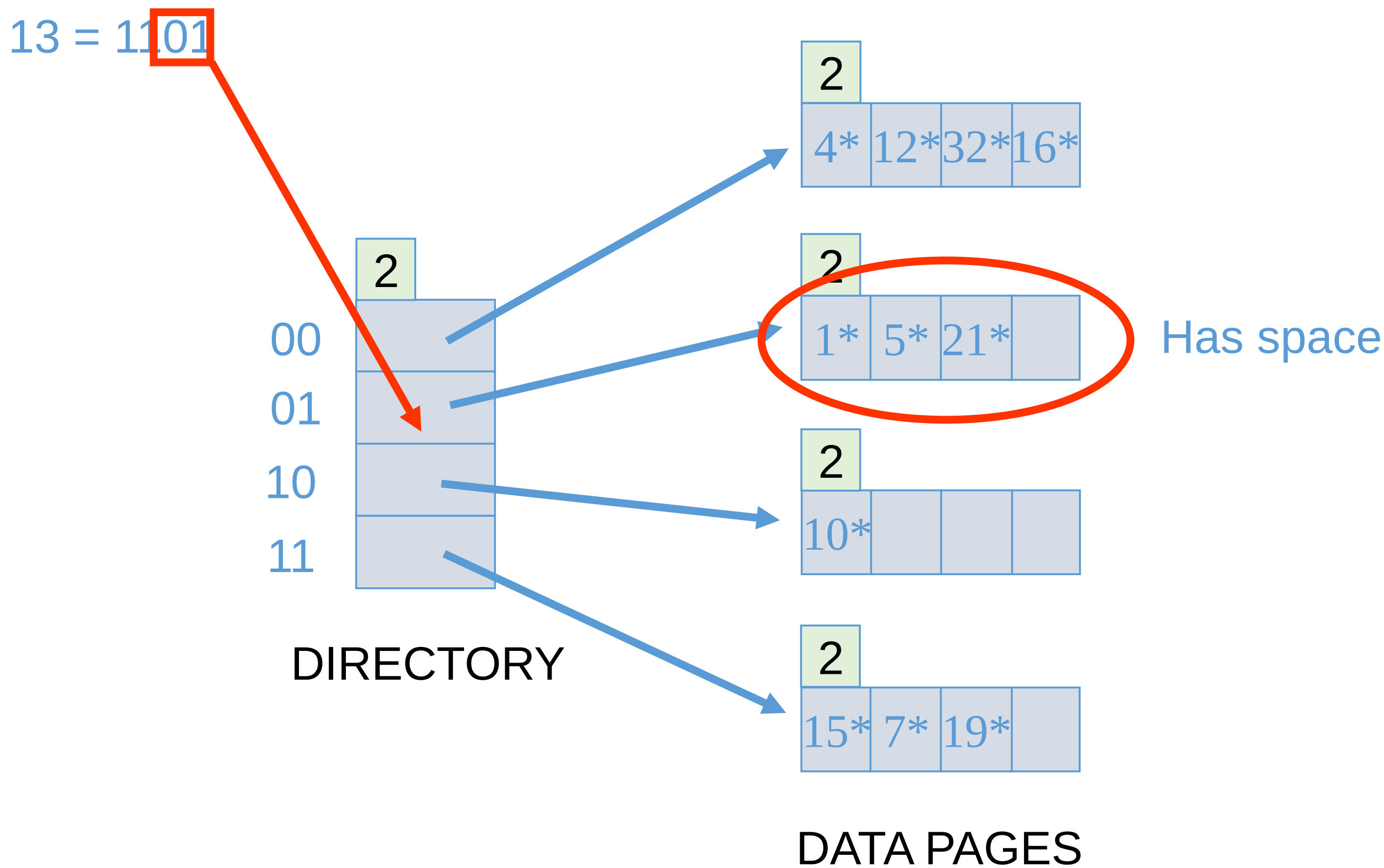
Global Depth = max(Local Depth of all buckets)
Tells # bits needed to determine the address



Dynamic Hashing: Example

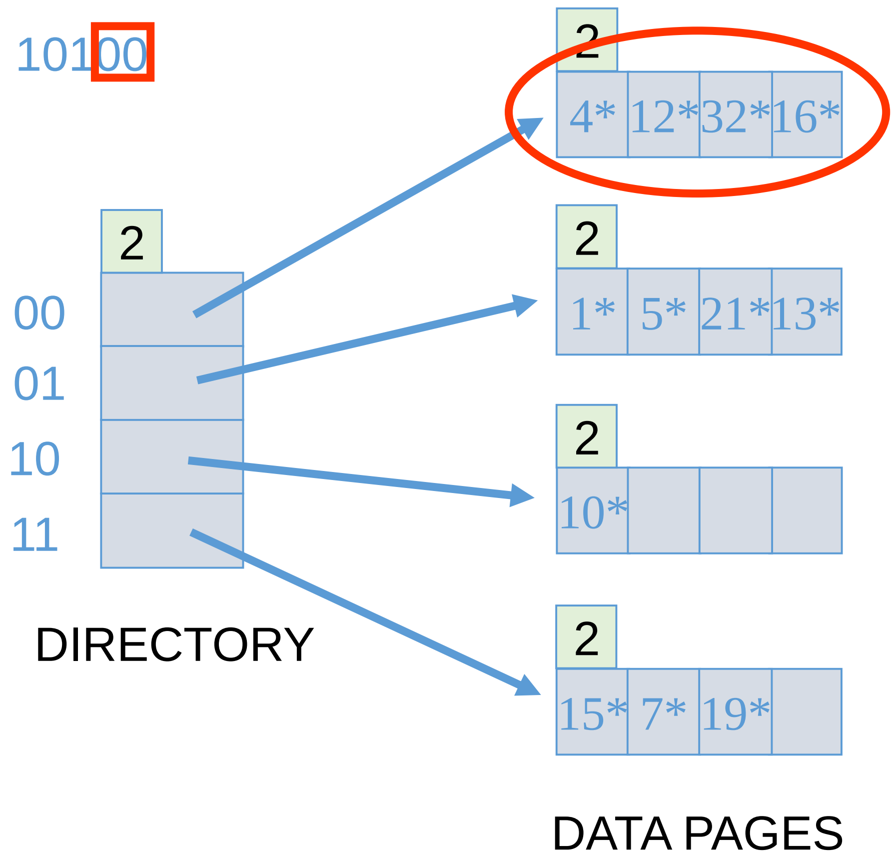


Dynamic Hashing: Insert 13*



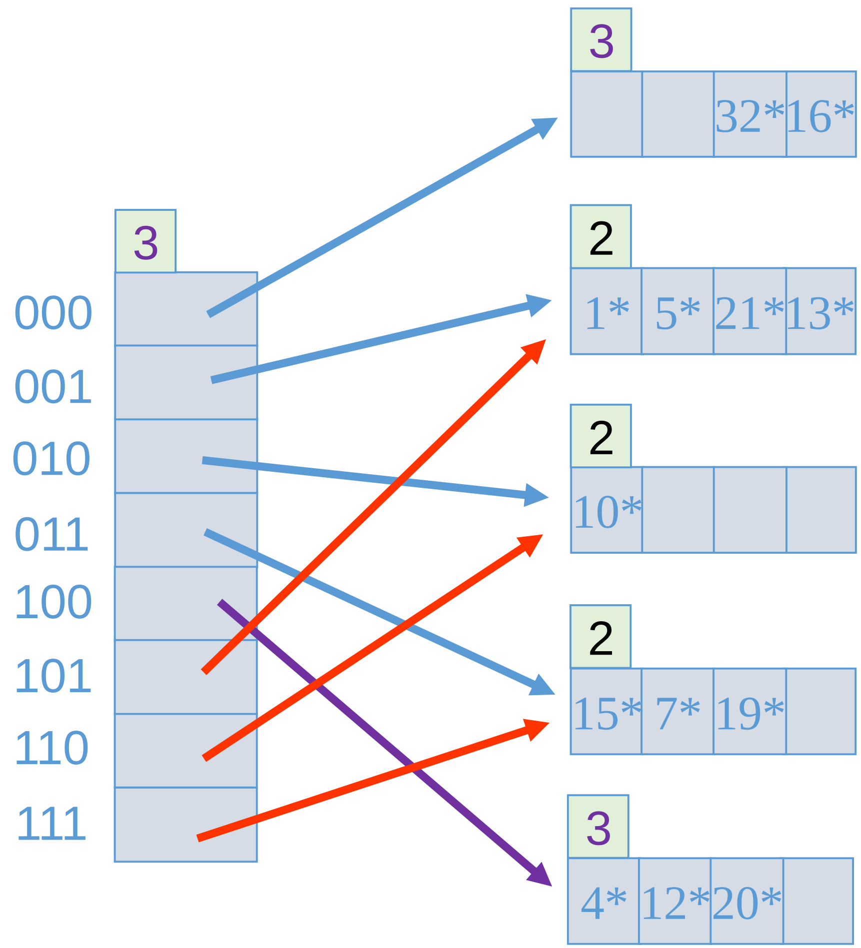
Dynamic Hashing: Insert 20*

20 = 10100



Full, need split.
Consider the last three bits

Dynamic Hashing: Insert 20*

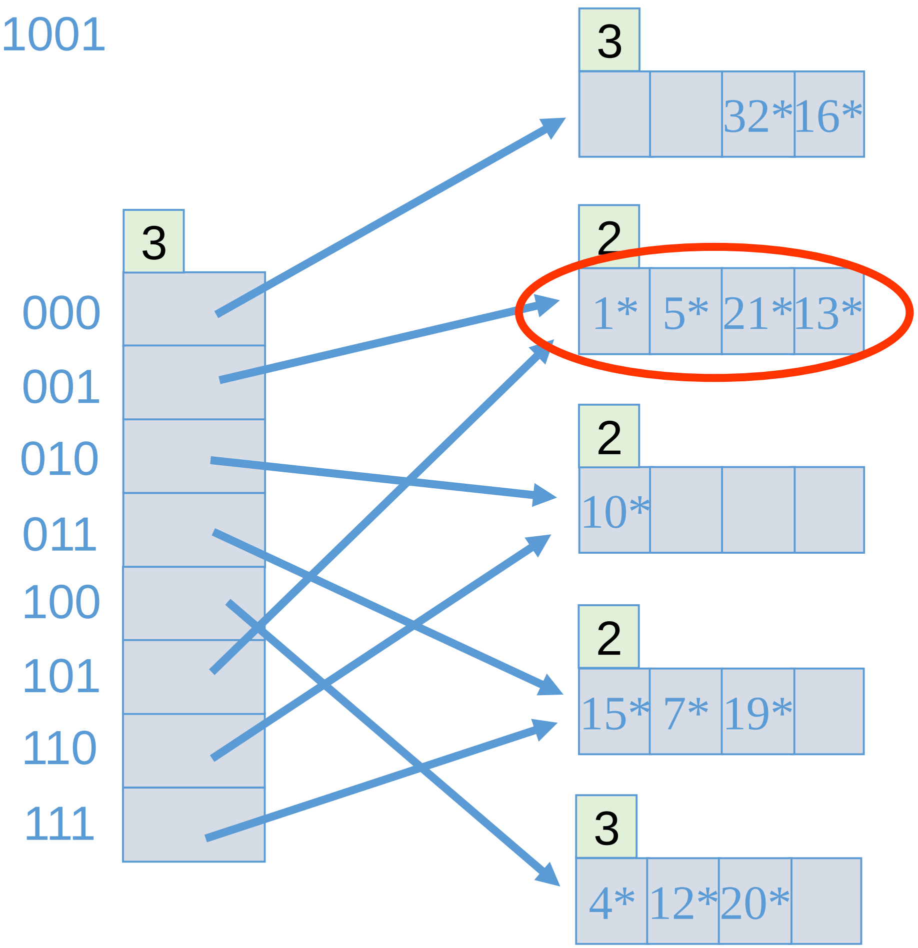


32=10 0000
16=01 0000

4=00 0100
12=00 1100
20=01 0100

Dynamic Hashing: Insert 9*

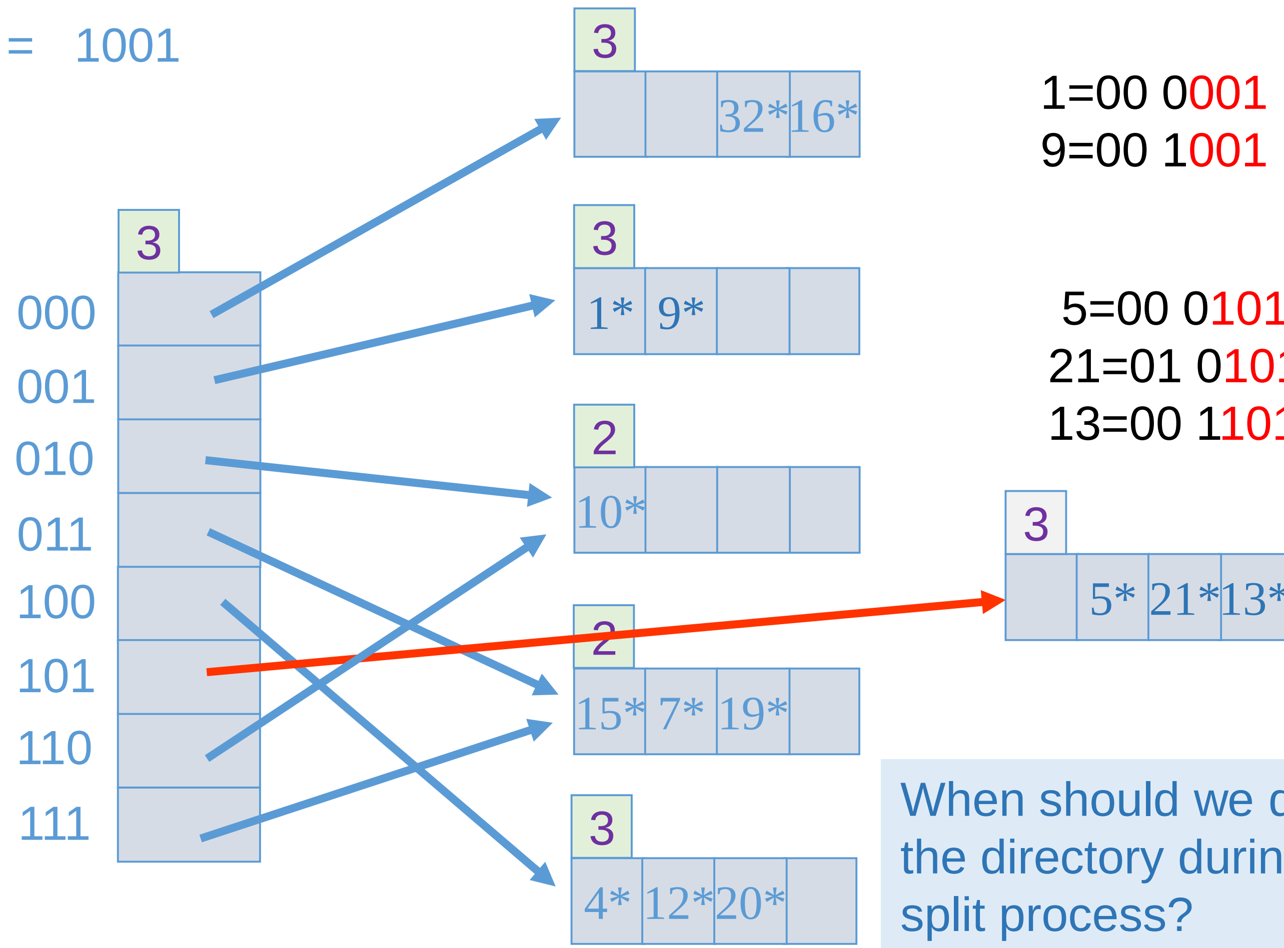
9 = 1001



Need split but directory doubling is not necessary

Dynamic Hashing: Insert 9*

9 = 1001



1=00 0001
9=00 1001

5=00 0101
21=01 0101
13=00 1101

When should we double the directory during the split process?

When to Double Directory

- Initially, all local depths are equal to global depth
  - # of bits need to express the total # of buckets
- During the process of split, if the 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

8.4

Bloom Filters

Introduction

- Generalize the hashing ideas
  - $h(k_1) = h(k_2) \Rightarrow k_1 ?\, k_2$
  - $h(k_1) \neq h(k_2) \Rightarrow k_1 \neq k_2$
- Approximate set membership problem
- Trade-off between the space and the false positive probability

Approximate Set Membership Problem

- Given a set
  S $= \{s_1, s_2, \dots, s_n\} \subseteq U$ (Universe)
- Want to check if "x is an element of S"
- Approximated approach
  - $h(S) = h(s_1) \vee h(s_2) \vee \cdots \vee h(s_n)$
  - $h(x) \wedge h(S) = \begin{cases} 1 & x ? S \quad \text{false positive} \\ 0 & x \notin S \quad \text{sure exclusion} \end{cases}$
  - Take little space

Bloom Filters

1. An n-bit array $A[n]$, initially set to 0
2. k independent random hash functions

   $$h_1, \dots, h_k \colon U \Rightarrow \{0, 1, \dots, n-1\}$$
3. $\forall\, s \in S,\ \ A[h_i(s)] = 1$ for $1 \le i \le k$

To check if $x \in S$, calculate

$$\bigcap_{1 \le i \le k} A[h_i(x)] = \begin{cases} 1 & x ?\, S \quad \text{assume } x \in S \\ 0 & x \notin S \quad \text{sure exclusion} \end{cases}$$

BF Design Consideration

- Choose n (filter size in bits).
  - Use as much memory as is available.
- Pick k (number of hash functions).
  - k too small ➡ high probability for different keys to have same signature.
  - k too large ➡ soon to fill ones in the filter
- Select the k hash functions.
  - Hash functions should be relatively independent.

Example: $n = 11, k = 2$

1	0	1	1	0	1	0	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10

- $h_1(s) = s \bmod n$.
- $h_2(s) = 2(s+1) \bmod n$.

	Student ID	h_1	h_2	
my class $m = 3$	105021121	7	5	
	210510215	3	8	
	106000103	0	2	
	107062601	8	7	false positive
	104034052	1	4	not in class

The Probability of a False Positive

- We assume the hash function are random.
- After all the elements of S are hashed into the bloom filters, the probability that a specific bit is still 0 is

$$p = (1 - 1/n)^{km} \approx e^{-km/n}$$

Note:

$$e^{-x} = 1 - x + \frac{x^2}{2} - \cdots \approx 1 - x$$

$$(e^{-x})^{km} \approx (1 - x)^{km}$$

Optimal n & k

- The probability of a false positive f is
$$f = (1-p)^k \approx (1-e^{-km/n})^k$$
- To find the optimal k to minimize f.

Minimize f iff minimize $g = \ln(f)$

$$\frac{dg}{dk} = \ln(1-e^{-km/n}) + \frac{km}{n}\frac{e^{-\frac{km}{n}}}{(1-e^{-km/n})}$$

$$\Rightarrow k = \ln(2) * (n/m),$$

$$\Rightarrow f = (1/2)^k = (0.6185)^{n/m}$$

The false positive probability falls exponentially in n/m, the number bits used per item !!

Conclusion

- A Bloom filters is like a hash table, and simply uses one bit to keep track whether an item hashed to the location.
- If $k = 1$, it's equivalent to a hashing based fingerprint system.
- If $n = cm$ for small constant c, such as $c = 8$, then $k = 5$ or 6, the false positive probability is just over 2% .
- It's interesting that when k is optimal

  $k = \ln(2) * (n/m)$, then $p = 1/2$.

  An optimized Bloom filters looks like a random bit-string