Hashing

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Hashing

- A technique that enable use to perform search, insert, and delete operations in $O(1)$ expected time.
- Relies on a formula called the hash function
- Two categories
  - Static hashing
    - The hash table is fixed-sized.
  - Dynamic/Extendible hashing
    - Can accommodate dynamically increasing and decreasing file size without penalty

Static Hashing -- Hash Tables

- Keys are stored in a fixed size table called a hash table.
  - b buckets, $ht[0], \ldots, ht[b\text{-}1]$
  - Each bucket consists of s slots.
  - The hash function, h, is used to map keys into buckets.
    - $h(k)$ is the hash or home address of k.
    - $h(k)$ is an integer in the range 0 through b-1.
    - T: the total # of possible keys
    - n: # of keys in the table

Static Hashing -- Hash Tables (contd.)

- The key density of a hash table
  - The ratio n/T
  - Usually very small
- The loading density or loading factor of a hash table
  - $\alpha = n/(sb)$
- Since b is usually much less than T, h maps several different keys into the same bucket.
  - k_1 and k_2 are synonyms with respect to h if $f(h_1) = f(h_2)$

Static Hashing -- Hash Tables (contd.)

- We enter distinct synonyms into the same bucket as long as the bucket has slots available.
- An overflow occurs when we hash a new key, i, into a full bucket.
- A collision occurs when we hash two nonidentical identifiers into the same bucket.

- When no overflow occurs, the time required to insert, delete, or search using hashing depends only on the time required to compute the hash function and to search one bucket.

Static Hashing -- Hash Functions

- Independent of n
- As s is usually small, the search within a bucket is carried out using a sequential search.

- We would like to choose a hash function that is both easy to compute and results in very few collisions.
- An uniform hash function
  - For a randomly chosen key, k, the probability that $h(k) = i$ is $1/b$ for all buckets i.

Static Hashing -- Hash Functions

- Not a data type for which arithmetic operations are defined
  - First convert the key into an integer (say) and then perform arithmetic on the obtained integer.
- Division
  - Assumption: No negative keys!
  - Using the modulus (%) operator to obtain the home bucket
  - $h(k) = k \% D$
    - The hash table must have at least $b = D$ buckets.
  - A good choice for D
    - D is a prime number such that M does not divide $r^k \pm a$ where k and a are small and r is the radix of the character set. [Knuth]
    - Be odd; set b equal to the divisor D

Static Hashing -- Hash Functions (contd.)

- Mid-Square
  - Assumption: Integer keys
  - Squaring the identifier and then using an appropriate # of bits/digits from the middle of the square to obtain the bucket address
  - Ex. $h(k)$ = central two digits of k^2, k = 355, k^2 = $12\underline{60}25$

Static Hashing -- Hash Functions (contd.)

- Folding
  - Partition the key k into several parts and then add the parts together to obtain the hash address
  - Ex. k = 12320324111220, a partition is three-digit long => P_1=123, P_2=203, P_3=241, P_4=112, P_5=20
  - Shift folding
    - Ex. Align P_1 through P_4 with P_5 and add $\Rightarrow$
      - $h(k) = \sum_{i=1}^{5} P_i$ = 123 + 203+241+112+20 = 699
  - Folding at the boundaries
    - Reverse every other partition before adding
    - Ex. P_2=302 and P_4=211 $\Rightarrow$ 897

Static Hashing -- Hash Functions (contd.)

- Digit analysis
  - Useful with static files
    - All the keys are known in advance.
  - Select and shift digits or bits of the original identifier
  - Digits having the most skewed distribution are deleted.
  - The number of remaining digits are small enough to give an address in the range of the hash table.

Static Hashing -- Hash Functions (contd.)

- Converting keys to integers
  - Consider only the conversion of strings into non-negative integers here.
  - Ex. p. 400, Program 8.1
  - Ex. p. 401, Program 8.2

Overflow Handling – Open Addressing

- Linear probing
  - aka. Linear open addressing
  - Search the hash table buckets in the order $ht[(h(k) + i) \% b]$, $0 \leq i \leq b-1$
  - Terminates when we reach the first unfilled bucket
  - Ex. p. 401, Example 8.4, p. 402, Fig. 8.2 & 8.3
  - Keys tend to cluster together.
    - Adjacent clusters tend to coalesce and thus increasing the search time.
  - When it is used together with a uniform hash function, the expected average number of key comparisons to look up a key is approximately $(2-\alpha)/(2-2\alpha)$.

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + 116 + 105 + 111 + 110	870	12

Figure 8.2: Additive transformation

[0]	function
[1]	
[2]	for
[3]	do
[4]	while
[5]	
[6]	
[7]	
[8]	
[9]	else
[10]	
[11]	
[12]	if

Figure 8.3: Hash table with linear probing (13 buckets, one slot per bucket)

Overflow Handling – Open Addressing (contd.)

- Quadratic probing
  - Some improvement in the growth of clusters and hence in the average # of comparisons needed
  - Examining buckets $h(k)$, $(h(k) + i^2)$ % b, $(h(k) - i^2)$ % b for $1 \leq i \leq (b\text{-}1)/2$
- Rehashing
  - Using a series of hash functions $h_1, h_2, \ldots, h_m$
  - Buckets $h_i(k)$, $1 \leq i \leq m$ are examined in that order.
- Random probing
  - Examining the buckets in the order $h(k)$, $(h(k) + s(i))$ % b, $1 \leq i \leq b\text{-}1$ where $s(i)$ is a pseudo random number.

Overflow Handling – Chaining

- The poor performance of linear probing and its variations
  - The search for a key involves comparison with keys that have different hash values.
- Maintaining lists of keys, one list per bucket, each list containing all the synonyms for that bucket
  - A search involves computing the hash address $h(k)$ and examining only those keys in the list for $h(k)$.
- The expected average # of key comparisons for a successful search $\approx 1+\alpha/2$

Overflow Handling (contd.)

- With a uniform hash function, performance depends only on the method used to handle overflows.
  - In practice, different hash functions result in different performance.
- The worst-case number of comparisons needed for a successful search remains $O(n)$ regardless of whether open addressing or chaining is used.

Dynamic Hashing

- Motivation
  - For good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a threshold
  - Aiming to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket
  - Its objective: Providing acceptable hash table performance on a per operation basis
- $h(k, p)$ – the integer formed by the p least significant bits of $h(k)$

Dynamic Hashing Using Directories

- A directory, d, of pointers to buckets
  - The size of d depends on the directory depth
    - # of bits of $h(k)$ used to index d
  - Given the directory depth t, the size of d is 2^t and the # of buckets is at most equal to the size of d
  - Ex. p. 411, Fig. 8.7, p. 412, Fig. 8.8
- To search for a key k, examine the bucket pointed to by $d[h(k, t)]$, where t is the directory depth

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure 8.7: An example hash function

Dynamic Hashing Using Directories – Overflow Resolution (contd.)

- Determine the least u such that $h(k, u)$ is not the same for all keys in the overflowed bucket.
  - Case 1: The least u is greater than the directory depth
    - Increase the directory depth to this least u value.
      - This requires us to increase the directory size but not the # of buckets.
    - Ex. p. 412, Fig. 8.8 (insert C5 into (a), insert C1 into (b))
  - Case 2: The current directory depth is greater than or equal to u
    - Some of the other pointers to the split bucket must be updated to point to the new bucket.
    - Ex. p. 412, Fig. 8.8(b) (insert A4 into (b))

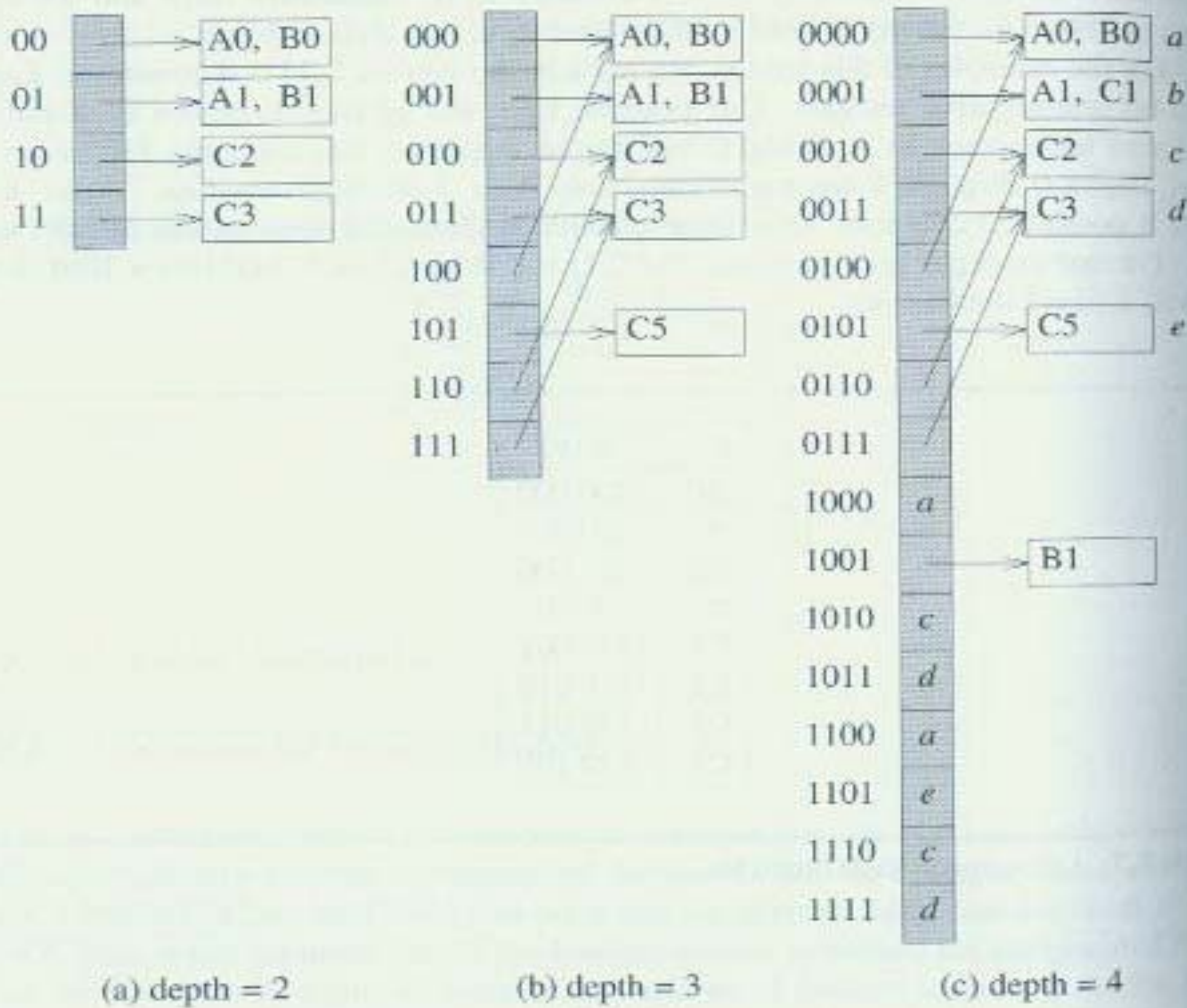


Figure 8.8: Dynamic hash tables with directories

Dynamic Hashing Using Directories – Overflow Resolution (contd.)

- Deletion from a dynamic hash table with a directory is similar to insertion.

Directoryless Dynamic Hashing

- aka. linear dynamic hashing
  - No dynamic increase of size
- Two variables q and r to keep track of active buckets
  - $0 \leq q < 2^r$
  - Only buckets 0 through $2^r + q - 1$ are active
    - The remaining on a chain are overflow buckets
  - indexed using
    - $h(k, r+1)$: buckets 0 through $q - 1$ as well as buckets 2^r through $2^r + q - 1$
    - $h(k, r)$: the remaining active buckets

Directoryless Dynamic Hashing (contd.)

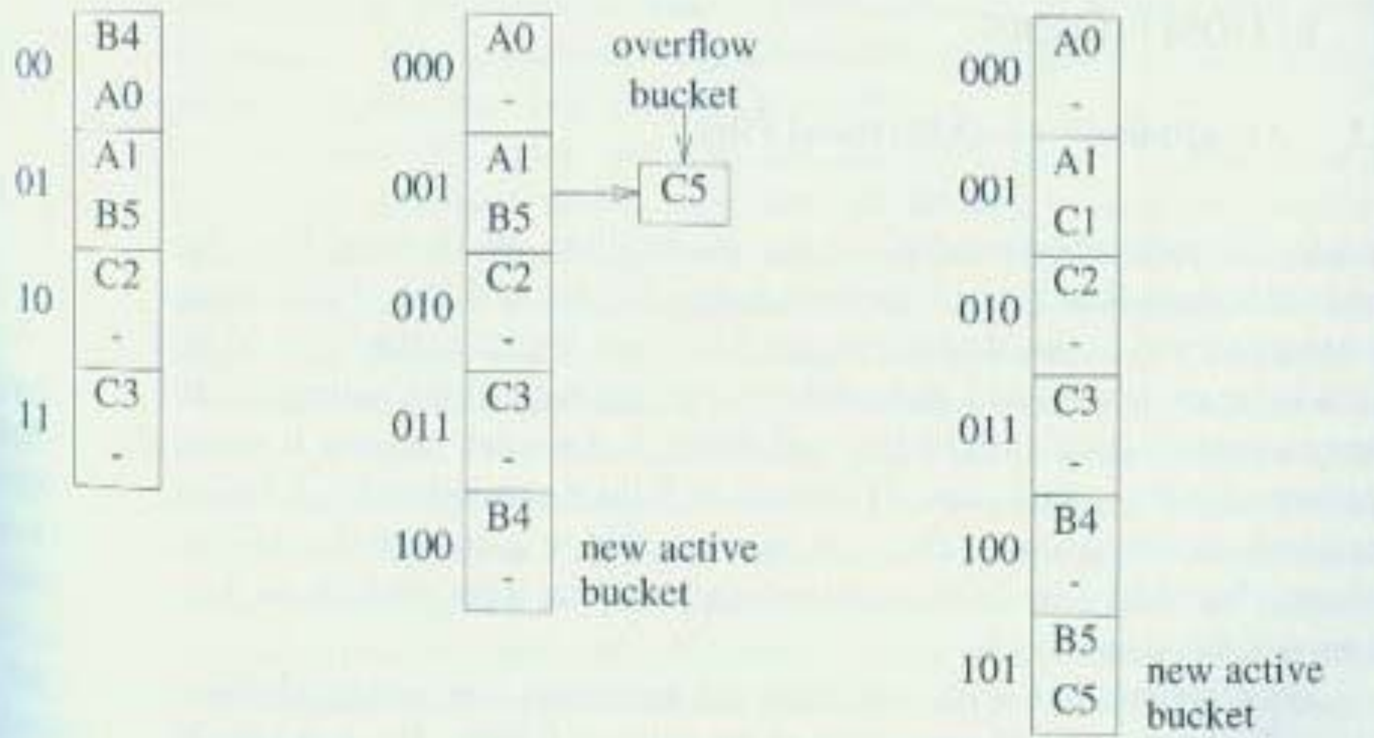
- The steps used to search for k (p. 415, Program 8.5)
  - Step 1: Compute $h(k, r)$; if $h(k, r) < q$, goto Step 2; otherwise, go to Step 3.
  - Step 2: k is in a chain indexed using $h(k, r + 1)$ if present
  - Step 3: The chain to examine is given by $h(k, r)$.

Directoryless Dynamic Hashing – Overflow Resolution

- Step 1: Activate bucket $2^r + q$
- Step 2: Reallocate the entries in the chain q between q and the newly activated bucket $2^r + q$ and increment q by 1
  - If $q = 2^r$, increment r by 1 and reset q to 0. The reallocation is done by using $h(k, r + 1)$.
- Ex. To insert C5 into the table of Fig. 8.9(a)
- Ex. To insert C1 into the table of Fig. 8.9(b)

if ($h(k,r) < q$) search the chain that begins at bucket $h(k,r+1)$;
else search the chain that begins at bucket $h(k,r)$;

Program 8.5: Searching a directoryless hash table



(a) $r = 2, q = 0$ (b) Insert C5, $r = 2, q = 1$ (c) Insert C1, $r = 2, q = 2$

Figure 8.9: Inserting into a directoryless dynamic hash table