Chapter 8
Hashing

Yi-Fen Liu
Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU
- and some supplement from the slides of Prof. C. Y. Tang and J. S. Roger, NTU

Outline

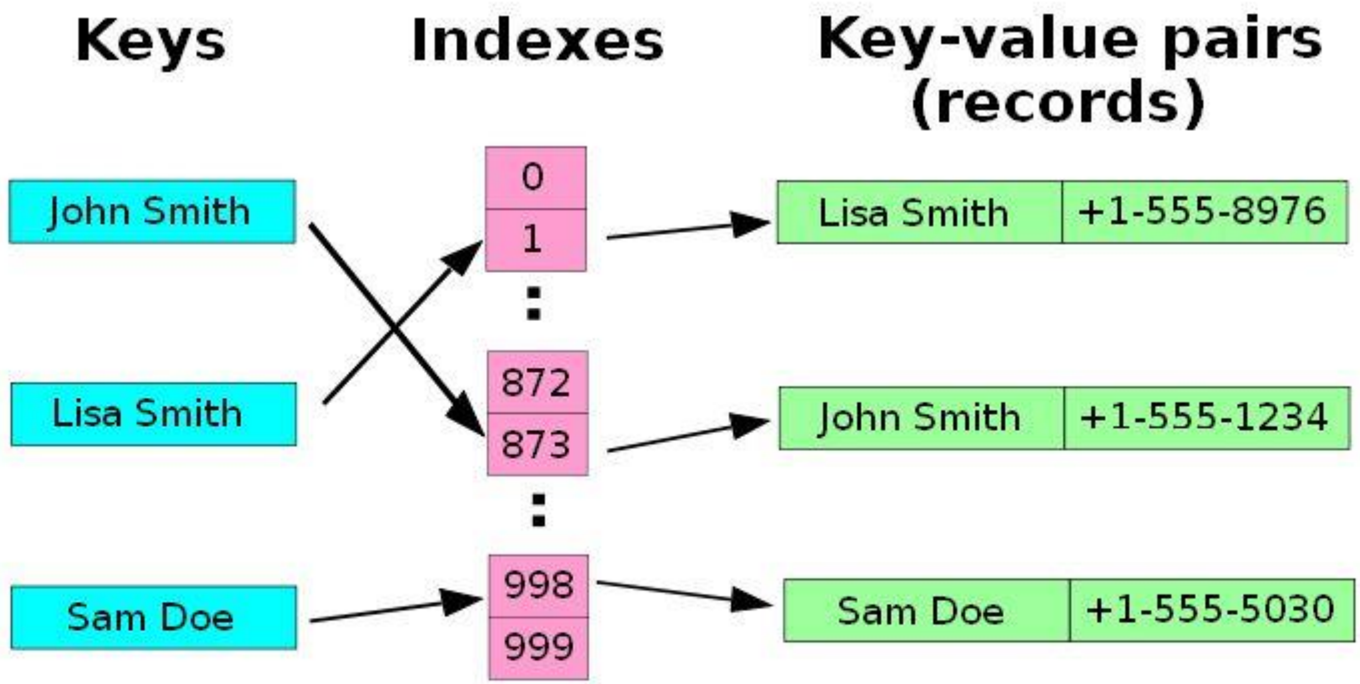
- Introduction
- Static Hashing
  - Hash Tables
  - Hashing Functions
    - Mid-square
    - Division
    - Folding
    - Digit Analysis
  - Overflow Handling
    - Linear Open Addressing, Quadratic probing, Rehashing
    - Chaining

INTRODUCTION

Concept of Hashing

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array

Example



A small phone book as a hash table.

(Figure is from Wikipedia)

Dictionaries

- Collection of pairs.
  - (key, value)
  - Each pair has a unique key.
- Operations.
  - Get(theKey)
  - Delete(theKey)
  - Insert(theKey, theValue)

Just An Idea

- Hash table :
  - Collection of pairs,
  - Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
  - Worst-case time for Get, Insert, and Delete is O(size).
  - Expected time is O(1).

Search vs. Hashing

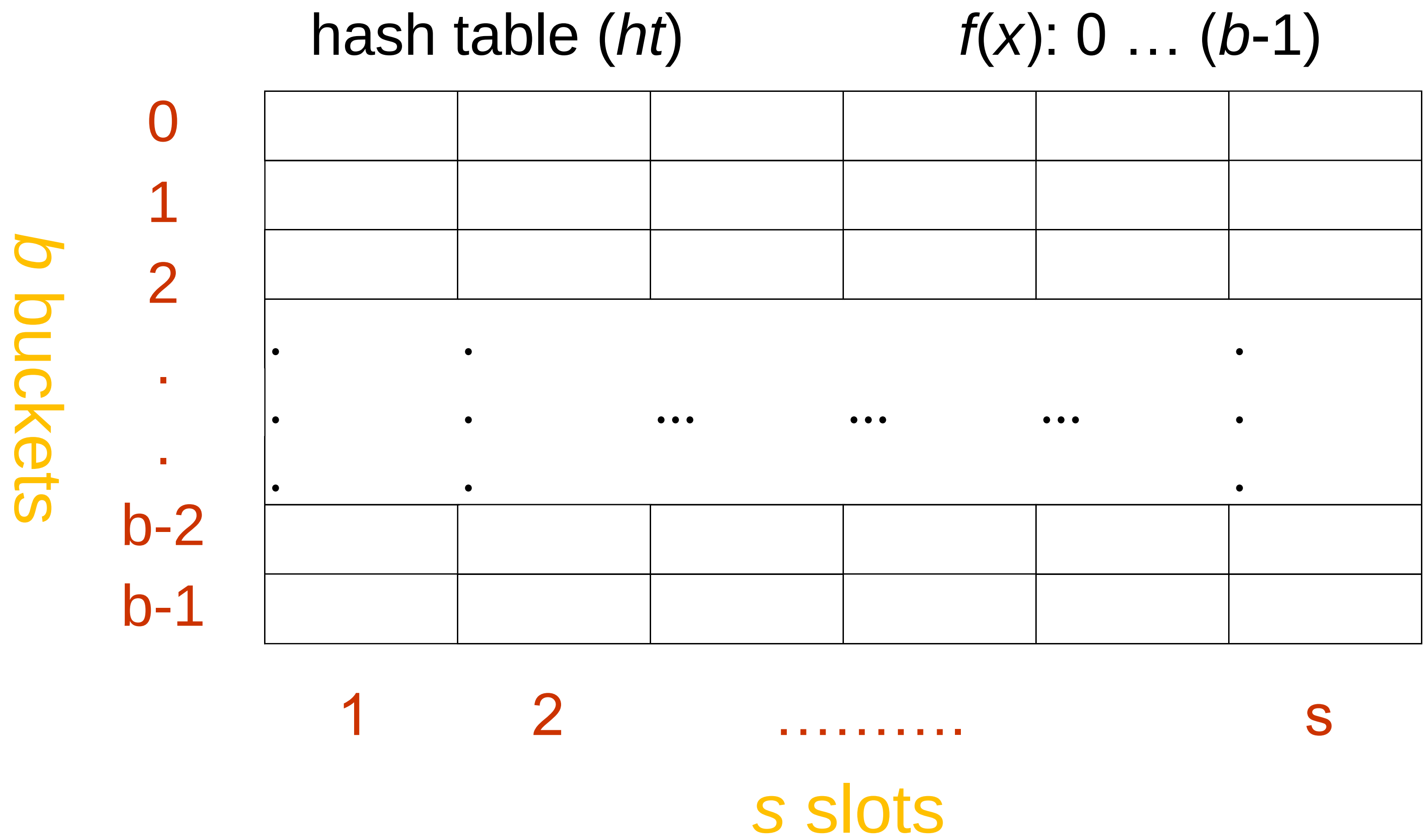
- Search tree methods: key comparisons
  - Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
- Types
  - Static hashing (section 8.2)
  - Dynamic hashing (section 8.3)

HASH TABLE

Hash Tables (1)

- In static hashing, we store the identifiers in a fixed size table called a *hash table*
- Arithmetic function, *f*
  - To determine the address of an identifier (key), *x*, in the table
  - $f(x)$ gives the hash, or home address, of *x* in the table
- Hash table, *ht*
  - Stored in sequential memory locations that are partitioned into *b* buckets, $ht[0]$, …, $ht[b-1]$.
  - Each bucket has *s* slots

Hash Tables (2)



Hash Tables (3)

- The *identifier density* of a hash table is the ratio *n/T*
  - *n* is the number of identifiers in the table
  - *T* is possible identifiers
- The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$
  - *s* is the number of slots
  - *b* is the number of buckets

Hash Tables (4)

- Two identifiers, i_1 and i_2 are synonyms with respect to f if $f(i_1) = f(i_2)$
  - We enter distinct synonyms into the same bucket as long as the bucket has slots available
- An overflow occurs when we hash a new identifier into a full bucket
- A collision occurs when we hash two non-identical identifiers into the same bucket.
- When the bucket size is 1, collisions and overflows occur simultaneously

Hash Tables (5)

- Example 8.1: Hash table
  - b = 26 buckets and s = 2 slots. Distinct identifiers n = 10
  - The loading factor, α, is 10/52 = 0.19.
  - Associate the letters, *a*-*z*, with the numbers, 0-25, respectively
  - Define a fairly simple hash function, $f(x)$, as the first character of x.

C library functions ($f(x)$):
acos(0), define(3), float(5), exp(4), char(2), atan(0), ceil(2), floor(5), clock(2), ctime(2)

overflow: clock, ctime

	Slot 0	Slot 1
0	acos	atan synonyi
1		
2	char	ceil synonyi
3	define	
4	exp	
5	float	floor
6		
…		
25		

Hash Tables (6)

- The time required to enter, delete, or search for identifiers does not depend on the number of identifiers n in use; it is $O(1)$.
- Hash function requirements:
  - Easy to compute and produces few collisions.
  - Unfortunately, since the ration b/T is usually small, we cannot avoid collisions altogether.
    => Overload handling mechanisms are needed

HASHING FUNCTIONS

Hashing Functions (1)

- A hash function, *f*, transforms an identifier (key), *x*, into a bucket address in the hash table.
- We want a hash function that is easy to compute and that minimizes the number of collisions.
- Hashing functions should be unbiased.
  - That is, if we randomly choose an identifier, *x*, from the identifier space, the probability that $f(x)$ = i is $1/b$ for all buckets *i*. 對應到每個Bucket No.的機率皆相等。(不會有局部偏重的情況)
  - We call a hash function that satisfies unbiased property a *uniform hash function*.
    Mid-square, Division, Folding, Digit Analysis

Hashing Functions (2)

- **Mid-square** $f_m(x)=middle(x^2)$:
  - Frequently used in symbol table applications.
  - We compute f_m by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the bucket address.
  - The number of bits used to obtain the bucket address depends on the table size. If we use r bits, the range of the values is 0 through 2^r-1.
  - Since the middle bits of the square usually depend upon all the characters in an identifier, there is high probability that different identifiers will produce different hash addresses.

Example – Middle Square

- Supposed that there are 1000 buckets, the range values is 0 through 999 (10^3-1). Try to obtain the bucket address of x =8125 by using Middle Square
- Sol:

(取平方)

x = 8125 ⇨ 66015625

取中間三位 ⇨ 156 = Hash Address (取015亦可)

Hashing Functions (3)

- **Division** $f_D(x) = x \% M$:
  - Using the modulus (%) operator.
  - We divide the identifier x by some number M and use the remainder as the hash address for x.
    - This gives bucket addresses that range from 0 to M - 1, where M = that table size.
- The choice of M is critical.
  - If M is divisible by 2, then odd keys to odd buckets and even keys to even buckets. (biased!!)
  - different hash addresses.

Hashing Functions (4)

- The choice of M is critical (cont'd)
  - When many identifiers are permutations of each other, a biased use of the table results.
  - Example: $X=x_1x_2$ and $Y=x_2x_1$
    Internal binary representation: x_1 --> $C(x_1)$ and x_2 --> $C(x_2)$
    Each character is represented by six bits
    X: $C(x_1) * 2^6 + C(x_2)$, Y: $C(x_2) * 2^6 + C(x_1)$
    $(f_D(X) - f_D(Y))$ % p (where p is a prime number)
    $= (C(x_1) * 2^6 \% p + C(x_2) \% p - C(x_2) * 2^6 \% p - C(x_1) \% p) \% p$
    $p = 3$, $2^6=64$
    $(64 \% 3 * C(x_1) \% 3 + C(x_2) \% 3 - 64 \% 3 * C(x_2) \% 3 - C(x_1) \% 3) \% 3$
    $= (C(x_1) \% 3 + C(x_2) \% 3 - C(x_2) \% 3 - C(x_1) \% 3) \% 3 = 0 \% 3$
    The same behavior can be expected when $p = 7$
  - A good choice for M would be : M a prime number such that M does not divide $r^k \pm a$ for small k and a.

Hashing Functions (5)

- **Folding**
  - Partition identifier *x* into several parts
  - All parts except for the last one have the same length
  - Add the parts together to obtain the hash address
- Two possibilities (divide *x* into several parts)
  - *Shift folding*:
    Shift all parts except for the last one, so that the least significant bit of each part lines up with corresponding bit of the last part.
    - x_1=123, x_2=203, x_3=241, x_4=112, x_5=20, address=699
  - *Folding at the boundaries*:
    reverses every other partition before adding
    - x_1=123, x_2=302, x_3=241, x_4=211, x_5=20, address=897

Hashing Functions (6)

- Folding example:

123	203	241	112	20
P1	P2	P3	P4	P5

shift folding

123 →
203 →
241 →
112 →
20

699

folding at the boundaries

123 → 203 ← 241 → 112 ← 20 →

MSD ---> LSD
LSD <--- MSD

Hashing Functions (7)

- **Digit Analysis**
  - Used with static files
    - A *static files* is one in which all the identifiers are known in advance.
  - Using this method,
    - First, transform the identifiers into numbers using some radix, *r*.
    - Second, examine the digits of each identifier, deleting those digits that have the most skewed distribution.
    - We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table.

Hashing Functions (8)

- Digital Analysis example:
  - All the identifiers are known in advance, M=1~999

    $X_1: d_{11} \quad d_{12} \ldots \quad d_{1n}$
    $X_2: d_{21} \quad d_{22} \ldots \quad d_{2n}$
    …
    $X_m: d_{m1} \quad d_{m2} \ldots \quad d_{mn}$

電話			號碼				鍵值				位址		
0	2	-	9	8	$\boxed{4}$	7	$\boxed{5}$	8	$\boxed{6}$		4	5	6
0	2	-	9	8	$\boxed{8}$	7	$\boxed{8}$	6	$\boxed{4}$		8	8	4
0	2	-	7	6	$\boxed{7}$	6	$\boxed{7}$	8	$\boxed{5}$		7	7	5
0	2	-	8	9	$\boxed{2}$	1	$\boxed{6}$	4	$\boxed{3}$		2	6	3
0	2	-	9	9	$\boxed{6}$	7	$\boxed{5}$	8	$\boxed{7}$		6	5	7
0	2	-	8	8	$\boxed{3}$	7	$\boxed{4}$	8	$\boxed{2}$		3	4	2
0	2	-	7	8	$\boxed{4}$	7	$\boxed{3}$	8	$\boxed{1}$		4	3	1

  - Select 3 digits from *n*
  - Criterion:
    Delete the digits having the most skewed distributions

- The one most suitable for general purpose applications is the division method with a divisor, *M*, such that *M* has no prime factors less than 20.

OVERFLOW HANDLING

Overflow Handling (1)

- Linear open addressing (Linear probing)
  - Compute $f(x)$ for identifier x
  - Examine the buckets:
    $ht[(f(x)+j)\%TABLE\_SIZE]$, $0 \leq j \leq TABLE\_SIZE\text{-}1$
    - The bucket contains x.
    - The bucket contains the empty string (insert to it)
    - The bucket contains a nonempty string other than x (examine the next bucket) (circular rotation)
    - Return to the home bucket $ht[f(x)]$,
      if the table is full we report an error condition and exit

Overflow Handling (2)

- Additive transformation and Division

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

Hash table with linear probing (13 buckets, 1 slot/bucket)

insertion

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

Overflow Handling (3)

- Problem of Linear Probing
  - Identifiers tend to cluster together
  - Adjacent cluster tend to coalesce
  - Increase the search time
  - Example: suppose we enter the C built-in functions into a 26-bucket hash table in order. The hash function uses the first character in each function name

Enter sequence:

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime

Enter: [illegible]

bucket	x	buckets searched
0	[redacted]	
1	[redacted]	
2	[redacted]	
3	[redacted]	
4	[redacted]	
5	[redacted]	
6	[redacted]	
7	[redacted]	
8	[redacted]	
9	[redacted]	
10	[redacted]	
...		
25		

Average # of buckets examined is 41/11=3.73

Hash table with linear probing (26 buckets, 1 slot/bucket)

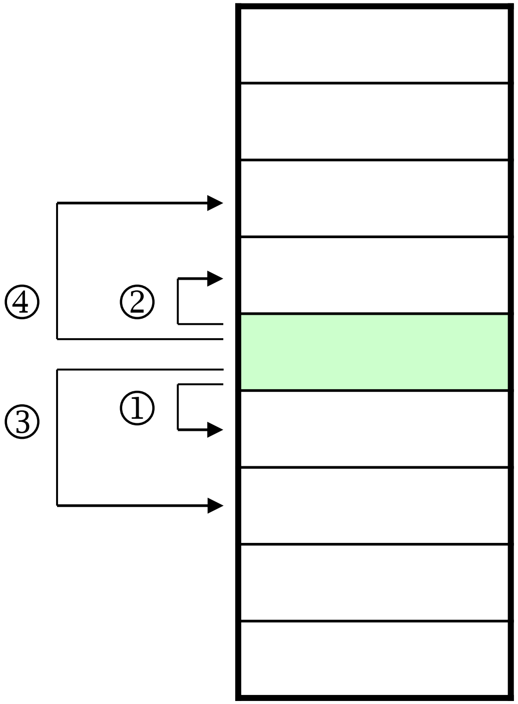
Overflow Handling (4)

- Alternative techniques to improve open addressing approach:
  - *Quadratic probing*
  - *rehashing*
  - *random probing*
- Rehashing
  - Try $f_1, f_2, \ldots, f_m$ in sequence if collision occurs
  - disadvantage
    - comparison of identifiers with different hash values
    - use chain to resolve collisions
    - condition and exit

Overflow Handling (5)

- Quadratic Probing
  - Examine buckets $f(x)$, $(f(x)+i^2)\%b$, $(f(x)-i^2)\%b$, for $1<=i<=(b-1)/2$
  - b is a prime number of the form $4j+3$,

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

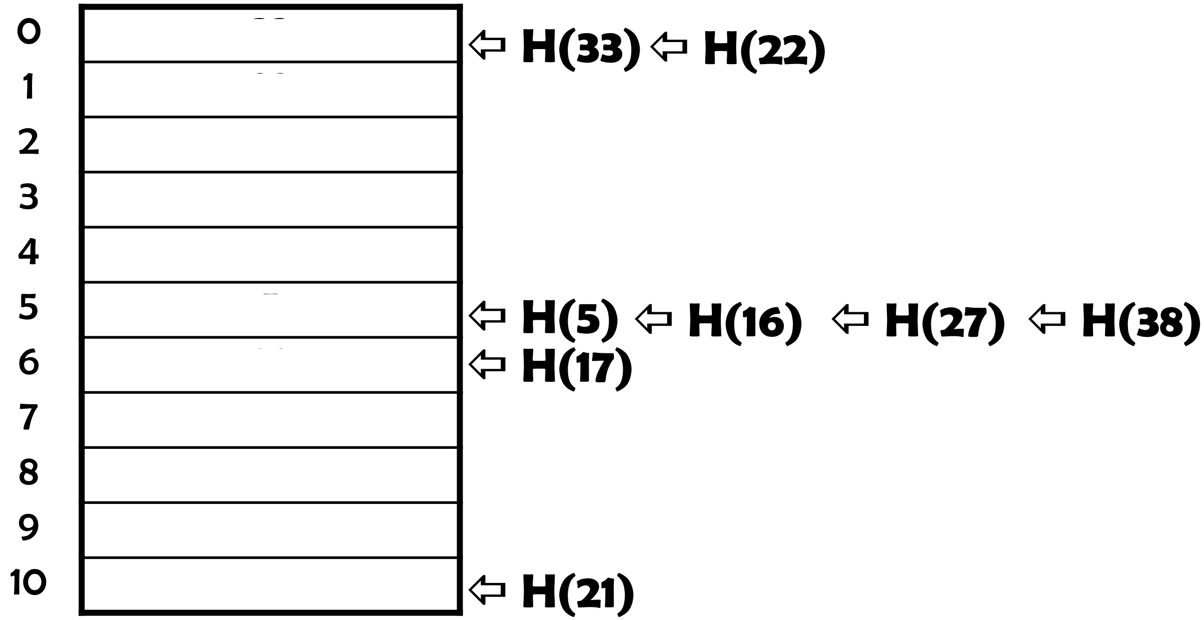


Example – Quadratic Probing

- Using quadratic probing to handle overflows:

  5, 16, 33, 21, 22, 27, 38, 17

- Sol:



Overflow Handling (6)

- Chaining
  - Linear probing and its variations perform poorly because inserting an identifier requires the comparison of identifiers with different hash values.
  - In this approach we maintained a list of synonyms for each bucket.
  - To insert a new element
    - Compute the hash address $f(x)$
    - Examine the identifiers in the list for $f(x)$.
  - Since we would not know the sizes of the lists in advance, we should maintain them as lined chains

Overflow Handling (7)

- Results of Hash Chaining

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime
$f(x)$=first character of x

[0] –> **acos** –> **atoi** –> **atol**
[1] –> *NULL*
[2] –> **char** –> **ceil** –> **cos** –> **ctime**
[3] –> **define**
[4] –> **exp**
[5] –> **float** –> **floor**
[6] –> *NULL*
...
[25] –> *NULL*

Figure 8.6: Hash chains corresponding to Figure 8.4

Average # of buckets examined is 21/11=1.91

Overflow Handling (8)

- Comparison:
  - In Figure 8.7, The values in each column give the average number of bucket accesses made in searching eight different table with 33,575, 24,050, 4909, 3072, 2241, 930, 762, and 500 identifiers each.
  - Chaining performs better than linear open addressing.
  - We can see that division is generally superior

$\alpha = \frac{n}{b}$	.50		.75		.90		.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

Average number of bucket accesses per identifier retrieved