Hash Table

Bùi Tiến Lên

2023

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Modular Arithmetic

- Random generator
- Encryption

Introduction

Concept 1

Given integers a, b, and n with $n > 1$, we say that a is **congruent** to b **modulo** n if $n \mid (a - b)$, written

$$a \equiv b \,(\mathrm{mod}\, n) \tag{1}$$

Some congruences

- $14 \equiv 2 (\mathrm{mod} 12)$, since $12 \mid (14 - 2)$.
- $-4 \equiv 8 (\mathrm{mod} 12)$, since $12 \mid (-4 - 8)$.
- $34 \equiv 6 (\mathrm{mod} 7)$, since $7 \mid (34 - 6)$.
- $25 \equiv 0 (\mathrm{mod} 5)$, since $5 \mid (25 - 0)$.

Properties

Theorem 1 (Congruence is an Equivalence Relation)

Let a, b, and n be integers with $n > 1$.

1. $a \equiv a \pmod{n}$.
2. *If* $a \equiv b \pmod{n}$, *then* $b \equiv a \pmod{n}$.
3. *If* $a \equiv b \pmod{n}$ *and* $b \equiv c \pmod{n}$, *then* $a \equiv c \pmod{n}$.

Properties (cont.)

Theorem 2 (Arithmetic Properties of Congruence)

Let a_1, a_2, b_1, b_2, and n be integers with $n > 1$. If

$$a_1 \equiv b_1 \pmod{n}$$
$$a_2 \equiv b_2 \pmod{n}$$

then

1. $a_1 + a_2 \equiv b_1 + b_2 \pmod{n}$
2. $a_1 - a_2 \equiv b_1 - b_2 \pmod{n}$
3. $a_1 a_2 \equiv b_1 b_2 \pmod{n}$
4. $ka_1 \equiv kb_1 \pmod{n}$, $k \in \mathbb{Z}$
5. $a_1^k \equiv b_1^k \pmod{n}$

Word Examples

1 Compute $(5162387 + 83645) \bmod 10$.

- First solution, $5162387 + 83645 \equiv 5246032 \equiv 2 (\mathrm{mod} 10)$
- Second solution, since $5162387 \equiv 7 (\mathrm{mod} 10)$ and $83645 \equiv 5 (\mathrm{mod} 10)$, we get $5162387 + 83645 \equiv 5 + 7 \equiv 12 \equiv 2 (\mathrm{mod} 10)$

Word Examples (cont.)

2 Compute $3^{32} \bmod 17$

- First solution, $3^{32} \equiv 1853020188851841 \equiv 1 (\mathrm{mod} 17)$
- Second solution, We have

$$
\begin{aligned}
3^{32} &\equiv 3^{2^{2^{2^{2^2}}}} \ (\mathrm{mod} 17) \\
&\equiv 9^{2^{2^{2^2}}} \ (\mathrm{mod} 17) \\
&\equiv 81^{2^{2^2}} \ (\mathrm{mod} 17) \equiv 13^{2^{2^2}} \ (\mathrm{mod} 17) \\
&\equiv 169^{2^2} \ (\mathrm{mod} 17) \equiv 16^{2^2} \ (\mathrm{mod} 17) \\
&\equiv 256^2 \ (\mathrm{mod} 17) \equiv 1^2 \ (\mathrm{mod} 17) \\
&\equiv 1 \ (\mathrm{mod} 17)
\end{aligned}
$$

Problems

1. Find the last two-digits of 7^{32}
2. Show that for any integer n, $n^2 \not\equiv 2 (\mathrm{mod} 5)$
3. Let n be any integer. Show that $n^3 \equiv n (\mathrm{mod} 6)$.

Linear Congruential Generator

A **linear congruential generator** (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers

- The generator is defined by recurrence relation:

$$r_n \equiv a \times r_{n-1} + c \,(\mathrm{mod}\, m) \tag{2}$$

where $\{r_i\}$ is the sequence of pseudorandom values, and

- $m > 0$: the modulus
- $0 < a < m$: the multiplier
- $0 \leq c < m$: the increment
- r_0: the seed

RSA

Concept 2

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission

- In a public-key cryptosystem, the encryption key is *public* and distinct from the decryption key, which is kept secret (*private*)

RSA (cont.)

The RSA algorithm involves four steps:

1. key generation: generate
   - modulus n
   - public key e ()
   - private key d ()
2. key distribution
3. encryption: using  to encrypt the message m

$$c = \text{encrypt}(m) = m^e \bmod n \tag{3}$$

4. decryption: using  to decrypt the encrypted message c

$$m = \text{decrypt}(c) = c^d \bmod n \tag{4}$$

Example

An internet banking app is set up to receive transactions from many customers of a bank. The bank use a RSA system with $n = 3233, e = 17, d = 413$

- A client use the app to encrypt the message $m = 65$

$$c = \text{encrypt}(65) = 65^{17} \bmod 3233 = 2790,$$

the encrypted message $c = 2790$ is sent to the bank.

- The bank can decrypt $c = 2790$

$$m = \text{decrypt}(2790) = 2790^{413} \bmod 3233 = 65,$$

the decrypted message $m = 65$ is thus received and interpreted as a "Balance Inquiry"

Power function

- How to compute $y \leftarrow x^k \bmod n$

```
BIGINT pow(BIGINT x, BIGINT k, BIGINT n) {
  BIGINT y = 1;
  for(BIGINT i=1; i<=k; k++)
    y *= x;
  return y % n;
}
```

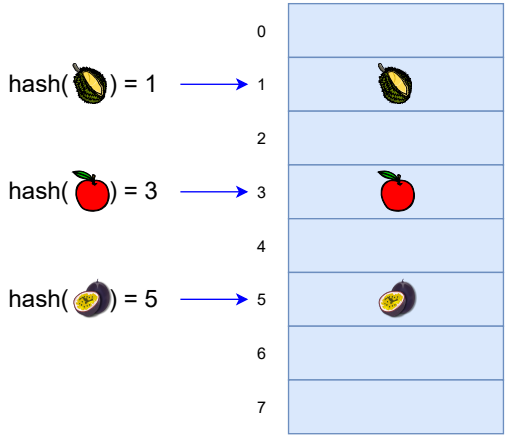
Hash Table

- Hash functions
- Hash Table
- Separate chaining
- Open addressing
- Dynamic hash tables

Hash functions

Concept 3

Hash function is a method for computing **array index** from **key** (**object**).



Hash functions (cont.)

The three principal criteria in selecting a hash function are as follows

- It should be *consistent*—equal keys must produce the same hash value.
- It should be *efficient to compute*.
- It should *uniformly distributes* the keys

The hash function depends on the **key type**.
Some popular hash functions: CRC32, MD5, SHA1, SHA2

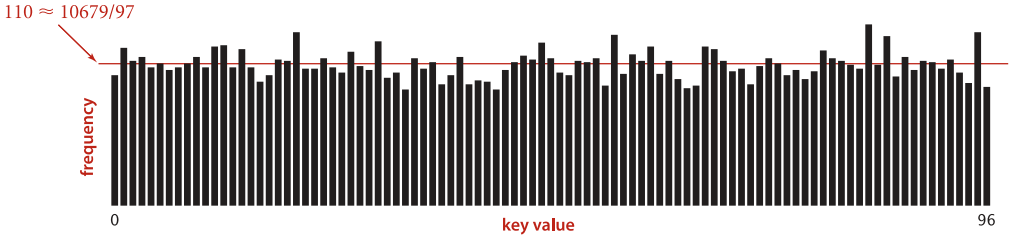
Applications. Digital fingerprint, message digest, storing passwords.

Uniform hashing assumption

Uniform hashing assumption

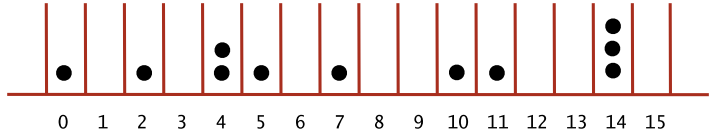
The hash functions that we use uniformly and independently distribute keys among the integer values between 0 and $m - 1$.

- Hash value frequencies for words in "Tale of Two Cities" (10,679 keys, $m = 97$)



Probability Review "bins and balls"

Events throw n balls uniformly at random into m bins.



- **Birthday problem**. Expect two balls in the same bin after

$$\sim \sqrt{\frac{\pi m}{2}} \tag{5}$$

- **Coupon collector**. Expect every bin has ≥ 1 ball after

$$\sim m \ln m \tag{6}$$

- **Load balancing**. After m tosses, expect most loaded bin has

$$\Theta\left(\frac{\log m}{\log \log m}\right) \tag{7}$$

Integer keys

- The most commonly used method for hashing integers is called **modular hashing** (**division method**)

$$h(k) = k \bmod m \tag{8}$$

key k	hash ($m = 100$)	hash ($m = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68

key k	hash ($m = 100$)	hash ($m = 97$)
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Floating-point keys

- If the keys are real numbers between 0 and 1, we might just multiply by m and round off to the nearest integer to get an index between 0 and $m - 1$.

key k	hash ($m = 100$)
.513870656	51
.175725579	17
.308633685	30
.534531713	53
.947630227	94
.171727657	17
.702230930	70
.226416826	22
.494766086	49
.124698631	12
.083895385	8
.389629811	38
.277230144	27

key k	hash ($m = 100$)
.368053228	36
.983458996	98
.535386205	53
.765678883	76
.646473587	64
.767143786	76
.780236185	78
.822962105	82
.151921138	15
.625476837	62
.314676344	31
.346903890	34

Floating-point keys (cont.)

In general, we can use the **multiplication method** for creating hash functions operates in two steps

- First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA.
- Then, we multiply this value by m and take the floor of the result.

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor \tag{9}$$

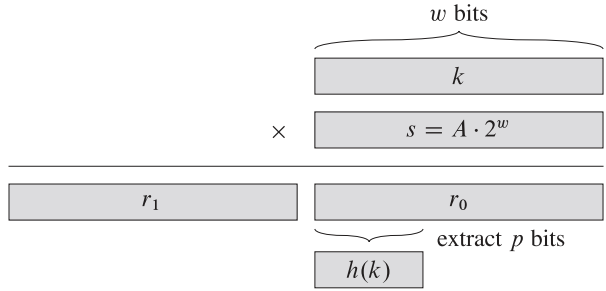
where "$k \cdot A \bmod 1$" means the fractional part of kA.

- Knuth suggests that

$$A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887... \tag{10}$$

Floating-point keys (cont.)

- The multiplication method of hashing



Another multiplication method method

$$h(k) = \lfloor k \cdot A \rfloor \bmod m \tag{11}$$

Floating-point keys (cont.)

key k	$k\%97$	$k\%100$	$\lfloor k \cdot A \rfloor \%100$
16838	57	38	6
5758	35	58	58
10113	25	13	50
17515	55	15	24
31051	11	51	90
5627	1	27	77
23010	21	10	20
7419	47	19	85
16212	13	12	19
4086	12	86	25
2749	33	49	98
12767	60	67	90
9084	63	84	14

key k	$k\%97$	$k\%100$	$\lfloor k \cdot A \rfloor \%100$
12060	32	60	53
32225	21	25	16
17543	83	43	42
25089	63	89	5
21183	37	83	91
25137	14	37	35
25566	55	66	0
26966	0	66	65
4978	31	78	76
20495	28	95	66
10311	29	11	72
11367	18	67	25

String keys

We simply treat **strings** as **huge integers** and use modular hashing

- For example, for character data with 7-bit encoding, we treat the key as a base-128 number $\rightarrow$ the word "`now`" corresponds to the number 1816567

$$110 \times 128^2 + 111 \times 128^1 + 119 \times 128^0$$

String keys (cont.)

- Modular hash functions for strings

key k	number	$m = 64$	$m = 31$
now	1816567	55	29
for	1685490	50	20
tip	1914096	48	1
ilk	1734251	43	18
dim	1651949	45	21
tag	1913063	39	22
jot	1751028	52	24
sob	1898466	34	26
nob	1816546	34	8
sky	1897977	57	2
hut	1719028	52	16
ace	1602021	37	3
bet	1618676	52	11

key k	number	$m = 64$	$= 31$
men	1798894	46	26
egg	1668071	39	23
few	1684215	55	16
jay	1749241	57	4
owl	1833964	44	4
joy	1751033	57	29
rap	1880304	48	30
gig	1701095	39	1
wee	1962725	37	22
was	1962227	51	20
cab	1634530	34	24
wad	1962212	36	5

String keys (cont.)

- How do we compute the hash function for a word such as "`averylongkey`" ?
- In 7-bit ASCII, this word corresponds to the 84-bit number

$$\begin{aligned} 97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^{9} + 114 \cdot 128^{8} + 121 \cdot 128^{7} \\ +108 \cdot 128^{6} + 111 \cdot 128^{5} + 110 \cdot 128^{4} + 103 \cdot 128^{3} \\ +107 \cdot 128^{2} + 101 \cdot 128^{1} + 121 \cdot 128^{0} \\ = 147984752178092529970675 13 \end{aligned}$$

String keys (cont.)

- To compute a modular hash function for long keys, we transform the keys piece by piece.
- A string $s = s_0 s_1 ... s_n$ where each character is encoded by r-radix number

$$number = s_0 \cdot r^n + s_1 \cdot r^{n-1} + ... + s_n \cdot r^0 \tag{12}$$

can be computed using Horner's rule

$$number = ((...((s_0 \cdot r + s_1) \cdot r + s_2) \cdot r + ...) \cdot r + s_n) \tag{13}$$

String keys (cont.)

- We can take advantage of arithmetic properties of the `mod` function and use Horner's algorithm

```
unsigned int hash(string &s, int m) {
  unsigned int hash_value = 0;
  unsigned int r = 127;

  for (int k = 0; k < s.length(); ++k)
    hash_value = (hash_value * r + s[k]) % m;

  return hash_value;
}
```

- **Challenge**: Can we make any improvement for long strings?

Compound keys

- If the key type has multiple integer fields, we can typically mix them together in the way just described for string values.
- For example, suppose that search keys are of type `Date`, which has three integer fields:
  - `day` (two-digit day)
  - `month` (two-digit month)
  - `year` (four-digit year)
- We compute the number

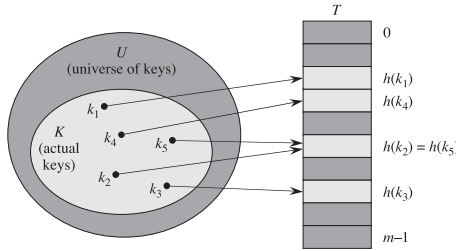
```
int hash_value = (((day * r + month) % m ) * r + year) % m;
```

Introduction

Concept 4

A **hash table** (**hash map**) is a data structure that implements **an associative array abstract data type**, a structure that can map keys to values. A hash table uses a **hash function** h to map **a key** of U into **an index** (**hash code**) of an array $T[0...m-1]$.

$$\begin{array}{rccc} h: & U & \rightarrow & T \\ & k & \mapsto & h(k) \end{array} \tag{14}$$



Introduction (cont.)

Concept 5

Given a hash table T with m slots that stores n elements, we define the **load factor**, for T as

$$\alpha = \frac{n}{m}, \tag{15}$$

that is, the average number of elements stored in a chain.

In practice. The load factor $\frac{1}{8} \leq \alpha \leq \frac{1}{2}$ (for open addressing).

Collision

Concept 6

A **collision** is a situation when two different keys may hash to the same hash code.

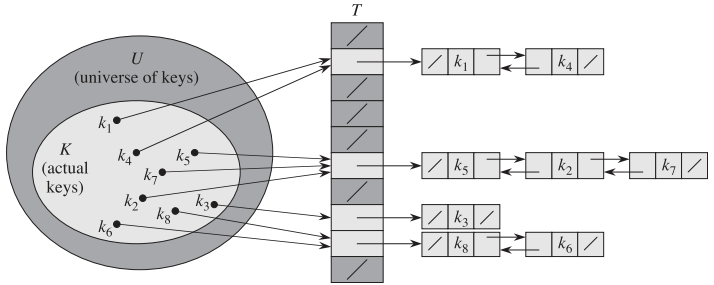
$$k_1 \neq k_2, \qquad h(k_1) = h(k_2) \tag{16}$$

There are two different approaches to collision resolution:

- Separate chaining
- Open addressing

Separate chaining

- In **chaining**, we put all the elements that hash to the same slot in a linked list



API

CHAINED-HASH-INSERT(T, x)
    insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
    search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
    delete x from the list $T[h(x.key)]$

Analysis of hashing with chaining (cont.)

Theorem 3

*In a hash table in which collisions are resolved by chaining, an **unsuccessful search** takes expected time $\Theta(1+\alpha)$ (the number of compares), under the assumption of simple uniform hashing.*

Theorem 4

*In a hash table in which collisions are resolved by chaining, a **successful search** takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.*

Introduction

- Separate chaining hashing has the disadvantage of using linked lists. This could slow the algorithm down a bit because of the time required to **allocate new nodes** and essentially requires the implementation of a second data structure.
- Another approach to implementing hashing is to store n key-value pairs in a hash table of size

$$m > n, \tag{17}$$

relying on empty entries in the table to help with collision resolution. Such methods are called **open-addressing hashing** methods.

Introduction (cont.)

- To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key.
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, ..., m-1\} \to \{0, 1, ..., m-1\} \tag{18}$$

the **probe sequence**

$$\langle h(k, 0), h(k, 1), ..., h(k, m-1) \rangle \tag{19}$$

be a permutation of

$$\langle 0, 1, ..., m-1 \rangle \qquad \textbf{(constraint)} \tag{20}$$

API

```
HASH-INSERT(T, k)
    i ← 0
    repeat
        j ← h(k, i)
        if T[j] = null
            T[j] ← k
            return j
        else i ← i + 1
    until i = m
    error "hash table overflow"
```

API (cont.)

```
HASH-SEARCH(T, k)
    i ← 0
    repeat
        j ← h(k, i)
        if T[j] = k
            return j
        i ← i + 1
    until T[j] = null or i = m
    return null
```

Random Probing

- **Asssumption**: For every $k \in U$, $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ is **random permutation**, independent of all other permutations.
  $\rightarrow$ the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, ..., m-1\rangle$
- **Note**: the assumption is difficult to implement.
- **Simple implementation**: pseudo-random probing

$$h(k,i) = (h'(k) + r_i) \bmod m \tag{21}$$

where $h'(k)$ be an ordinary hash function and r_i be the ith value in a random permutation of the numbers from 1 to $m-1$. All insertions, deletions and searches use the same sequence of random numbers.

Probabilistic analysis of random probing

Theorem 5

Given an open-address hash table with load factor $\alpha = n/m < 1$,

1. *the expected number of probes in an unsuccessful search/insert is at most*

$$\frac{1}{1-\alpha} \tag{22}$$

2. *the expected number of probes in a successful search is at most*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} \tag{23}$$

Proof

Probabilistic analysis of random probing (cont.)

In an unsuccessful search, let us define the random variable X to be the number of probes made in an unsuccessful search and and let us also define the event A_i, for $i = 1, 2, ...$, to be the event that an ith probe occurs and it is to an occupied slot.

$$\begin{aligned} Pr(X \geq i) &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \end{aligned} \tag{24}$$

Probabilistic analysis of random probing (cont.)

- The expected number of probes

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{\infty} Pr(X \geq i) \\
&\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\
&= \frac{1}{1-\alpha}
\end{aligned}
\tag{25}
$$

■

Linear Probing

- The simplest open-addressing method is called linear probing: when there is a collision, then we just check the next entry in the table. Linear probing is characterized by identifying three possible outcomes:
  - Key equal to search key: search hit
  - Empty position (null key at indexed position): search miss
  - Key not equal to search key: try next entry

Linear Probing (cont.)

- Given an ordinary hash function

$$h' : U \to \{0, 1, ..., m-1\}, \tag{26}$$

the method of linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m \tag{27}$$

- Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**.

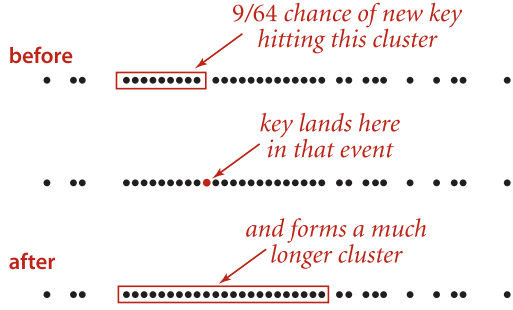
Clustering

Concept 7

Cluster is a contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

- Clustering in linear probing ($m = 64$)



Probabilistic analysis of linear probing

Theorem 6

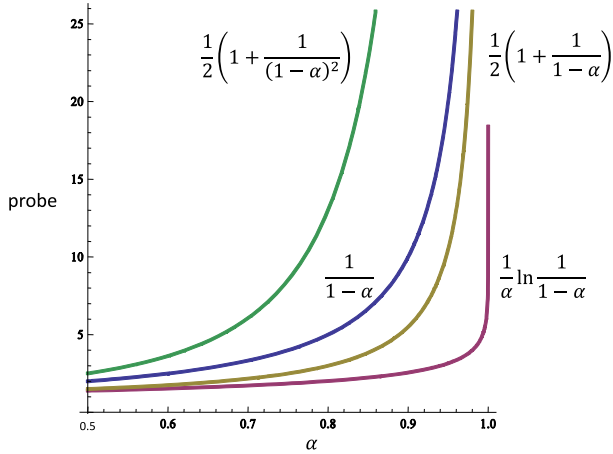
Under uniform hashing assumption, the average number of probes in a linear probing hash table of size m that contains n:

$$\sim \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad \textit{search hit} \tag{28}$$

$$\sim \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) \qquad \textit{search miss/insert} \tag{29}$$

Random vs. linear probing

- Assuming random hash functions



Comparison of hashing methods

- Theoretical comparison of hashing methods

Load factor α	**0.1**	**0.5**	**0.8**	**0.9**	**0.99**	**2**
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.4	1.45	1.5	2
Open, random probes	1.05	1.4	2	2.6	4.6	-
Open, linear probes	1.06	1.5	3	5.5	50.5	-
Unsuccessful search, expected number of probes:						
Chaining	0.1	0.5	0.8	0.9	0.99	2
Open, random probes	1.1	2	5	10	100	-
Open, linear probes	1.12	2.5	13	50	5000	-

Comparison of hashing methods (cont.)

- Empirical comparison of hashing methods

Load factor α	**0.1**	**0.5**	**0.8**	**0.9**	**0.99**	**2**
Successful search, expected number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2
Open, random probes	1.04	1.5	2.1	2.7	5.2	-
Open, linear probes	1.05	1.6	3.4	6.2	21.3	-
Unsuccessful search, expected number of probes:						
Chaining	0.1	0.5	0.8	0.9	0.99	2
Open, random probes	1.13	2.2	5.2	11.9	126	-
Open, linear probes	1.13	2.7	15.4	59.8	430	-

Quadratic Probing

- **Quadratic probing** uses a hash function of the form

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \tag{30}$$

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants

$$h(k,i) = (h'(k) + i^2) \bmod m \qquad \text{(simple form)} \tag{31}$$

- **Note**: to make full use of the hash table, the values of c_1, c_2, and m are constrained
- This method works much better than linear probing. It still suffers from another problem known as **secondary clustering**.

Quadratic Probing (cont.)

Theorem 7

If quadratic probing is used, and the table size m is prime, then a new element can always be inserted if the table is at least half empty.

Double hashing

- **Double hashing** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m \tag{32}$$

  where h_1 and h_2 are auxiliary hash functions.
- **Note**: The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
- It is one of the best methods because the permutations produced have many of the characteristics of randomly chosen permutations.

Dynamic hash tables

Parameter table size m

m too small $\Rightarrow$ search time blows up.

- As the number of keys in a hash table increases, search performance degrades.

m too large $\Rightarrow$ too many empty array entries.

- Waste memory.

Dynamic hash tables (cont.)

- If $\alpha > 1/2$ then double the table's size

$$m \leftarrow 2m \tag{33}$$

  Doubling the table is an **expensive** operation because everything in the table has to be reinserted, but it is an operation that is performed infrequently.

- If $\alpha < 1/8$ then halve the the table's size

$$m \leftarrow \frac{m}{2} \tag{34}$$

Cost summary for symbol-table implementations

implementation	worst case			average case			key
	search	insert	remove	search hit	insert	remove	
BST	N	N	N	$c \log_2 N$	$c \log_2 N$	$\sqrt{N}$	compare
AVL	$c_a \log_2 N$	-	-	$\log_2 N$	-	-	compare
RB	$c_r \log_2 N$	-	-	$\log_2 N$	-	-	compare
chain	N or $\ln N$	-	-	1	-	-	equal hash
linear	N or $\ln N$	-	-	1	-	-	equal hash

Note: $c = 1.39, c_a = 1.44, c_r = 2.0$

Universal hashing

Universal hashing

Problem Any *fixed* hash function is vulnerable to such terrible worst-case behavior.

Solution Choose the hash function *randomly* in a way that is independent of the keys that are actually going to be stored.

Universal hashing (cont.)

Concept 8

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be **universal if** for each pair of distinct keys $k, l \in U$,

$$|\{h \in \mathcal{H} \mid h(k) = h(l)\}| \leq \frac{|\mathcal{H}|}{m} \tag{35}$$

- In other words, the chance of a collision between k and l is $1/m$ if we choose h randomly from $\mathcal{H}$.

Universal hashing (cont.)

Theorem 8

Suppose that a hash function h is chosen from a universal collection of hash functions and is used to hash n keys into a table T of size m, for a given key k, we have

$$E[\#\ \text{collision with}\ k] \leq \frac{n}{m} \tag{36}$$

Designing a universal class of hash functions

- Choose a prime number p large enough so that every possible key k is in the range 0 to $p-1$.
- We define the hash function h_{ab}, $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$ ($\mathbb{Z}_p = \{0, 1, 2..., p-1\}$ and $\mathbb{Z}_p^* = \{1, 2..., p-1\}$)

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \tag{37}$$

- The family of all such hash functions is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\} \tag{38}$$

Designing a universal class of hash functions (cont.)

Theorem 9

The class $\mathcal{H}_{pm}$ of hash functions is universal.

Proof

Exercise. ■

Advanced Hash Table

Perfect hashing

Problem Given a set of n keys, construct a **static** hash table of size m such that SEARCH action takes $O(1)$ time in the worst case.

Solution $O(n^2)$ space

Theorem 10

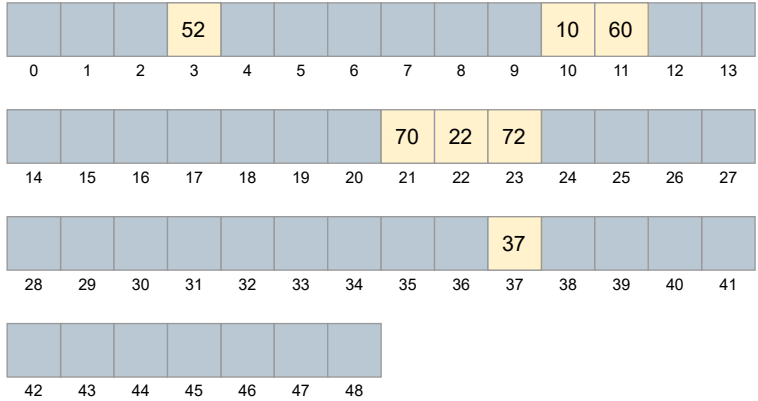
Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then, the probability is less than $1/2$ that there are any collisions.

Proof

Exercise. ■

Example 1

- Using perfect hashing to store the set $K = \{10, 22, 37, 52, 60, 70, 72\}$.
- The hash table T with size of $m = n^2 = 49$ and the hash function $h(k) = ((ak + b) \bmod p) \bmod m$ where $a = 1$, $b = 0$, $p = 73$

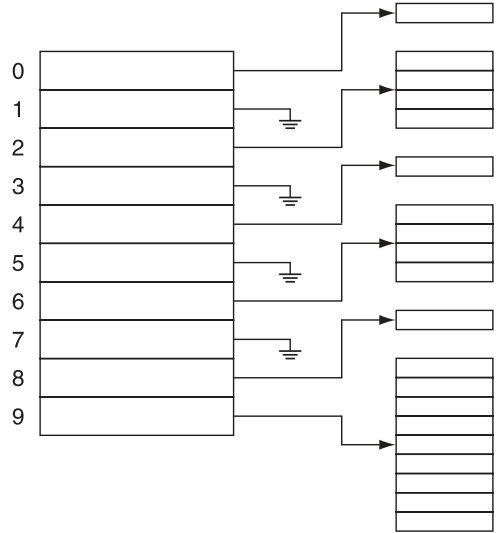


Solution $O(n)$ space

Idea: We use two levels of hashing, with universal hashing at each level.

- **The first level**: it is the same as for hashing with chaining
- **The second level**: we use a small secondary hash table S_j with an associated hash function h_j. By *choosing the hash functions* h_j carefully, we can guarantee that there are **no collisions** at the secondary level.

Solution $O(n)$ space (cont.)



API

CreatePerfectTable(T, $\{k_1, k_2, ..., k_n\}$, $\mathcal{H}$)

Carefully choose h and $\{h_0, h_1, ...h_{m-1}\}$ from $\mathcal{H}$
$count[0, \ldots, m-1] \leftarrow \{0, \ldots, 0\}$
for $i \leftarrow 1$ to n
    $count[h(k_i)] \leftarrow count[h(k_i)] + 1$
for $i \leftarrow 0$ to $m-1$
    allocate $count[i]^2$ memory slots to secondary table S_i and $T[i] \rightarrow S_i$
for $i \leftarrow 1$ to n
    $j \leftarrow h(k_i)$
    $p \leftarrow h_j(k_i)$
    $S_j[p] \leftarrow k_i$

SearchPerfectTable(T, k)

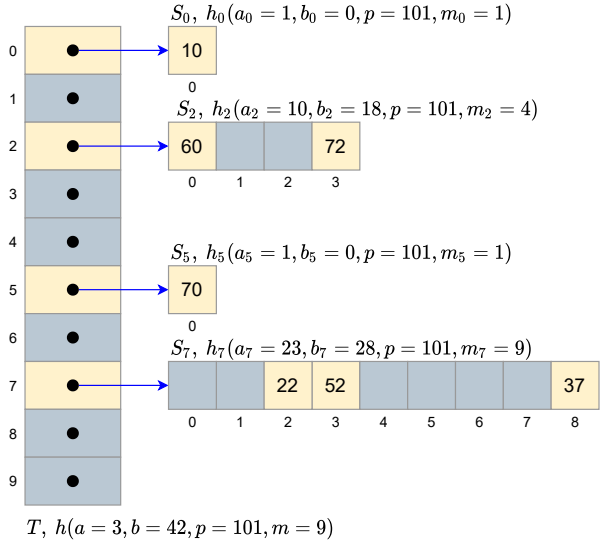
$j \leftarrow h(k)$
if $T[j] = null$ **return** *null*
$p \leftarrow h_j(k)$
return $S_j[p]$

Example 2

Using perfect hashing to store the set $K = \{10, 22, 37, 52, 60, 70, 72\}$.

- The table T uses the outer hash function $h(k) = ((ak + b) \bmod p) \bmod m$ where $a = 3$, $b = 42$, $p = 101$ and $m = 9$
- A secondary table S_j uses the hash function $h_i(k) = ((a_i k + b_i) \bmod p) \bmod m_i$ where $m_i = count_i^2$

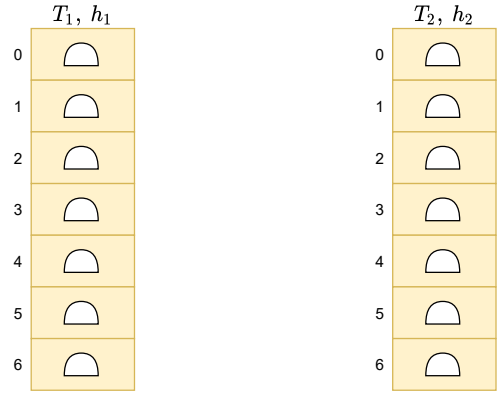
Example 2 (cont.)



Cuckoo hashing

In cuckoo hashing, we maintain

- **two** tables, each more than half empty.
- **two** independent hash functions randomly chosen from a universal class of hash functions that can assign each item to a position in each table.



API

InsertCuckooTable(T_1, h_1, T_2, h_2, k)
    Insert into the first table T_1
    If there was a collision **then**
        move the *current key* l to the second table T_2
        insert the *new key* k
    The displaced key l is then inserted in its alternative location,
        again kicking out any key that might reside there.
    Repeat process until success

SearchCuckooTable(T, k)
    return search in two tables T_1 and T_2

Workshop

Quiz

1. What is a hash function?

   . . . . . .
   . . . . . .
   . . . . . .

2. What is a hash table?

   . . . . . .
   . . . . . .
   . . . . . .

Projects

- Programming exercises in [Cormen, 2009, Sedgewick, 2002]

References

Cormen, T. H. (2009).
Introduction to algorithms.
MIT press.

Sedgewick, R. (2002).
Algorithms in Java, Parts 1-4, volume 1.
Addison-Wesley Professional.

Walls and Mirrors (2014).
Data Abstraction And Problem Solving with C++.
Pearson.