Hash tables (II)

Textbook, Chapter 12, Sections 12.3 and 12.4 (partially)

CSCE310: Data Structures and Algorithms
www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444
Hash functions

A good hash function

- can be computed quickly
- satisfies (approx.tely) the simple uniform hashing assumption: each key is equally likely to hash to any of the $m$ slots

Formally:
Assume each key is drawn independently from $U$ with probability distribution $P$
$P(k)$ is the probability that $k$ is drawn
Simple uniform hashing $\Rightarrow$

\[
\sum_{k : h(k) = j} P(k) = \frac{1}{m}, \quad \text{for } j = 0, 1, \ldots, m - 1
\]

However, $P$ is usually unknown
Formally: Simple uniform hashing requires
\[ \sum_{k : h(k) = j} P(k) = \frac{1}{m}, \text{ for } j = 0, 1, \ldots, m - 1 \]

Problem: we rarely know distribution \( P(k) \)

When keys are random real numbers \( k \) independently and **uniformly** distributed in the range of \( 0 \leq k < 1 \),

\[ h(k) = \lfloor km \rfloor \]

satisfies simple uniform hashing assumption
Hash functions

1. Division method
2. Multiplication method
3. Universal hashing

Assumption: keys are natural numbers \( (\in \mathbb{N}) \)
If keys \( \not\in \mathbb{N} \), find a way to express them as such

Example: strings can be interpreted by interpreting each character as an integer in notation radix-128, using the ASCII character set

Illustration: pt can be interpreted as \((112.128) + 116 = 14452\), since \(p = 112\) and \(t = 116\) in the ASCII character set
In **radix-2**: 1 digit, 2 possibilities

In a word of $k$ digits (bits), there are $2^k$ possibilities

$$a = \langle a_0, a_1, a_2, \ldots, a_{k-1} \rangle$$

$$a_{k-1}2^{k-1} + a_{k-2}2^{k-2} + \ldots + a_22^2 + a_12^1 + a_02^0 \equiv$$

$$a_{k-1}2^{k-1} + a_{k-2}2^{k-2} + \ldots + a_22^2 + a_1 + a_0$$

(binary, $a_i \in \{0, 1\}$) largest term is $2^k - 1$

In **radix-10**: 1 digit, 10 possibilities

In a word of $k$ digits, there are $10^k$ possibilities

$$a = \langle a_0, a_1, a_2, \ldots, a_{k-1} \rangle$$

$$a_{k-1}10^{k-1} + \ldots + a_210^2 + a_110^1 + a_010^0 \equiv$$

$$a_{k-1}10^{k-1} + \ldots + a_210^2 + a_110 + a_0$$

(decimal, $a_i \in \{0, 1, \ldots, 9\}$)

largest term is $10^k - 1$
**In radix-16:** 1 digit, 16 possibilities

In a word of $k$ digits, there are $16^k$ possibilities

$$a = \langle a_0, a_1, a_2, \ldots, a_{k-1} \rangle$$

$$a_{k-1}16^{k-1} + \ldots + a_216^2 + a_116^1 + a_016^0 \equiv$$

$$a_{k-1}16^{k-1} + \ldots + a_216^2 + a_116 + a_0$$

(hexadecimal, $a_i \in \{0,1,\ldots,9,A,B,\ldots,F\}$)

largest term is $16^k - 1$

**In radix-128:** 1 digit, 128 possibilities

In a word of $k$ digits, there are $128^k$ possibilities

$$a = \langle a_0, a_1, a_2, \ldots, a_k \rangle$$

$$a_{k-1}128^{k-1} + \ldots + a_2128^2 + a_1128^1 + a_0128^0 \equiv$$

$$a_{k-1}128^{k-1} + \ldots + a_2128^2 + a_1118 + a_0$$

(ASCII, $a_i \in \{0,\ldots,127\}$)

Example: pt is $112 \times 128 + 116 = 14452$ (bcz $p = 112$, $t=116$)

largest term is $128^k - 1$
Notes

\[ 1 + 2 + 3 + 4 + \ldots + n = \frac{n(n+1)}{2} \]

biggest digit = B. Value of biggest digit = x-1, radix-x
\[ B + Bx^2 + Bx^3 + \ldots + Bx^{k-1} = B \frac{x^k - 1}{x-1} \]

value of biggest term in a word of k-digits in radix x is \((x^k - 1)\)
Notes

Inversely, to represent $n$ possibilities in radix-$k$, we need $\log_k n$ digits.

To represent 4 possibilities in radix-2, we need $\log_2 4 = 2$ digits.
To represent 128 possibilities in radix-2, we need $\log_2 128 = 8$ digits.
To represent 10 possibilities in radix-10, we need $\log_{10} 10 = 1$ digit.
To represent 100 possibilities in radix-10, we need $\log_{10} 100 = 2$ digits.
Binary coding

A word of $k$ bits in radix-2 has $2^k$ possibilities

To put in radix-$2^l$ (binary coding) we need $\log_{2^l} 2^k = \frac{\log 2^k}{\log 2^l} = \frac{k}{l}$

(binary) digits, $k$ words of $l$ bits
#### Division method (1)

**Principle:**
takes the remainder of \( k \) divided by \( m \): \( h(k) = k \mod m \)

**Example:**
\[ m = 12 \text{ and } k = 100 \Rightarrow h(k) = 8 \times 12 + 4 \equiv 4 \pmod{12} \]

**Characteristics:**
- Quick hash function
- Avoid certain values of \( m \):
  1. \( k \) is a binary number, avoid \( m = 2^p \)
  2. \( k \) is a decimal number, avoid \( m = 10^p \)
  3. \( k \) is a character string in radix \( 2^p \), avoid \( m = 2^p - 1 \)
  4. Good values for \( m \) are primes not too close to exact powers of 2

**Example:** \( n = 2000 \) character strings

If \( \alpha = 3 \) then choose \( m = 701 \), a prime not too close to a power of 2 (512 and 1024) \( \Rightarrow h(k) = k \mod 701 \)

---

B.Y. Choueiry

March 19, 2001
Values of $m$ to avoid: $m$ power of 2, $m = 2^p$

Consider the key $a$, a binary number,

$$a = \langle a_0, a_1, \ldots, a_k \rangle = a_k2^k + a_{k-1}2^{k-1} + \ldots + a_12 + a_0$$

Assuming $k > p$, the key can be written

$$a = 2^p(a_k2^{k-p} + \ldots + a_p) + (a_{p-1}2^{p-1} + \ldots + a_12 + a_0)$$

where $(a_{p-1}2^{p-1} + \ldots + a_12 + a_0) < 2^p$

thus $a = q2^p + r$

where $q = a_k2^{k-p} + \ldots + a_p$ and $r = a_{p-1}2^{p-1} + \ldots + a_12 + a_0$

and $h(a) = a \mod m = a \mod 2^p = q2^p + r \mod 2^p \equiv r$, where

$$r = \langle a_{p-1}, \ldots, a_1, a_0 \rangle$$

If $m = 2^p$, then $h(k)$ is just the $p$ lowest order bits of $k$
Values of $m$ to avoid: $m$ power of 2, $m = 2^p$

If $m = 2^p$, then $h(k)$ is just the $p$ lowest order bits of $k$

Lesson: Unless $P(k)$ makes all low-order $p$-bits patterns equally likely, it is preferable to make the hash function depend on all the bits of the key

Values of $m$ to avoid: $m$ power of 10, $m = 10^p$

Similarly, avoid powers of 10 if keys are decimal numbers (as hash function does not depend on all the decimal digits of $k$)
Values of $m$ to avoid: $m = 2^p - 1$ (I)

when the key is a character string in radix $2^p$, two strings that are identical except for a transportation of two adjacent characters will hash to the same value

We prove for two keys $a, b$, such that:

$$a = \langle a_0, a_1, \ldots, a_i, \ldots, a_j, \ldots, a_k \rangle$$

$$b = \langle a_0, a_1, \ldots, a_j, \ldots, a_i, \ldots, a_k \rangle$$

we have $h(a) = h(b)$
Values of \( m \) to avoid: \( m = 2^p - 1 \) (II)

\[ a = \langle a_0, a_1, \ldots, a_i, \ldots, a_j, \ldots, a_k \rangle \]

\[ b = \langle a_0, a_1, \ldots, a_j, \ldots, a_i, \ldots, a_k \rangle \] then

\[ h(a) = (a_k 2^{kp} + \ldots + a_i 2^{ip} + \ldots + a_j 2^{jp} + \ldots + a_1 2^p + a_0) \mod (2^p - 1) = q(2^p - 1) + r \mod (2^p - 1) \]

and

\[ h(b) = (a_k 2^{kp} + \ldots + a_j 2^{jp} + \ldots + a_i 2^{ip} + \ldots + a_1 2^p + a_0) \mod (2^p - 1) = q'(2^p - 1) + r' \mod (2^p - 1) \]

\[ \Rightarrow h(a) - h(b) = a_i 2^{ip} + a_j 2^{jp} - a_j 2^{ip} - a_i 2^{jp} = (q - q')(2^p - 1) + r - r' \]

\[ \Rightarrow a_i (2^{ip} - 2^{jp}) - a_j (2^{ip} - 2^{jp}) = q''(2^p - 1) + r - r' \]

\[ \Rightarrow a_i 2^{ip} (2^{(i-j)p} - 1) - a_j 2^{jp} (2^{(i-j)p} - 1) = q''(2^p - 1) + r - r' \]

However, \( a^n - b^n = (a - b)(a^{n-1} + a^{n-2}b + \ldots + ab^{n-2} + b^{n-1}) \Rightarrow \]

\[ (2^{(i-j)p} - 1) = (2^p - 1)(\ldots) = (2^p - 1)A \]

\[ \Rightarrow (a_i 2^{jp} - a_j 2^{ip})(2^p - 1)A = q''(2^p - 1) + r - r' \]

\[ \Rightarrow r - r' = 0 \]

\[ \Rightarrow h(a) = h(b) \]

q.e.d
Multiplication method

Use: \( h(k) = \lfloor m(kA \mod 1) \rfloor \), where \( kA \mod 1 = ka - \lfloor kA \rfloor \)

Two steps:

1. Multiply \( k \) by \( A \) constant \((0 < A < 1)\), and extract the fractional part of \( kA \)

2. Then multiply value by \( m \) and take floor of result

Characteristics:

- Value of \( m \) is not critical
- We can choose \( m \) to make \( h \)-function easy to implement
  Choose for \( m = 2^p \) (bcz multiplication by \( m \) would correspond to a simple left shifting of \( p \) positions, shifting instructions available on most hardware)
- Knuth suggests using \( A \approx \frac{(\sqrt{5}-1)}{2} \)
Multiplication method: an example

\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]

Suppose \( k = 123456, m = 10000, A = 0.6180339887 \) (decimal numbers)

\[
\begin{align*}
h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots \mod 1) \rfloor \\
    &= \lfloor 10000 \cdot (76300.0041151\ldots \mod 1) \rfloor \\
    &= \lfloor 41.151\ldots \mod 1) \rfloor \\
    &= 41
\end{align*}
\]

Do exercise 12.3.4
**Multiplication method:** an implementation (I)

**Situation:**

- word size of the machine is $w$-bits
- keys are binary numbers, fit into 1 word
- no multiplication of rational rational numbers
Multiplication method: an implementation (II)

Steps:

- $0 < A < 1$ and no multiplication of rational rational numbers → use $\lceil A \cdot 2^w \rceil$
- Multiply $k$ and $\lceil A \cdot 2^w \rceil$ → (each of $w$ bits) yields a word of $2w$-bits of value $r_12^w + r_0$
- $r_0$ is the fractional part of $kA$ (approx)
- Multiplying by $m = 2^p$ corresponds to taking the $p$ most significant bits of $r_0$
**Universal hashing**

Worst-case scenario:

- Malicious adversary chooses the keys to be hashed
- Bad choice of hashing, all $n$ keys hash to the same slot

Average retrieval time deteriorates: $\Theta(n)$

Any *fixed* hash function is vulnerable

**Way out?**

Choose a $h$-function that is *random*

independent of keys to be stored

The scheme is called **universal hashing**

yields good performance on average

no matter what keys are chosen by adversary
Universal hashing: principal

Select a hash function at random, at run time from a carefully designed class of functions

Randomization guarantees

- that no single input will evoke worst-case behavior
- good average-case performance, no matter what keys are provided as input
**Universal hashing:** principal

Let $\mathcal{H}$ be a finite set of hash functions that map a universe $U$ of keys into the range $\{0, 1, 2, \ldots, m - 1\}$

$\mathcal{H}$ is **universal** if for every pair of distinct key $x, y, \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$

that is, with a hash function randomly chosen from $\mathcal{H}$, the chance of collision between $x$ and $y$ when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0, 1, 2, \ldots, m - 1\}$
Theorem 12.3: If $h$ is chosen from a universal collection of hash functions and is used to hash $n$ keys into a table of size $m$, where $n \leq m$, the expected number of collisions involving a particular key $k$ is less than 1.
Example of a universal class of hash functions

Principle

- Goal: Create a universal class of functions that hash a key into a value between 1 and m-1
- Proposal: each function is defined for an index $a$, such that $a$ is as a sequence $\langle a_0, a_1, \ldots, a_r \rangle$ where each $a_i < m$

  Each key $x$ we receive is represented as a sequence $\langle x_0, x_1, \ldots, x_r \rangle$ where each $x_i < m$

  The hash function is $h_a(x) = \sum_{i=0}^r a_i x_i \mod m$
Example of A universal class of hash functions

Implementation

- Choose \( m \), table size, to be a prime number reasons: (1) remember, division method!
  (2) prove collision probability is \( \frac{1}{m} \)
- Choose \((r + 1)\) \( a_i \)'s, each \( a_i \) chosen randomly from \( \{0, 1, \ldots, m - 1\} \)
  There are \( m^{(r + 1)} \) such possibilities
  This yields \( a = \langle a_0, a_1, \ldots, a_r \rangle \)
  Now \( a \) is fixed
- When user gives the key \( x \), decompose \( x \) in \( r + 1 \) parts of \( x = \langle x_0, x_1, \ldots , x_r \rangle \)
- Compute \( h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m \)
- We can prove that \( \mathcal{H} = \bigcup_a \{ h_a \} \) is a universal class of hash functions
  Since there are \( m^{(r + 1)} \) possibilities for \( a \), there are \( m^{(r + 1)} \) members in the class
Proof: \( \mathcal{H} = \bigcup_a \{ h_a \} \) is a universal class of hash functions

Given \( x, y \), two distinct keys, prove that \( h_a(x) = h_a(y) \) with probability \( 1/m \)

- Consider \( x, y \), two distinct keys. For example, \( x_0 \neq y_0 \), and rest \( x_i, y_i \) are the same

- \( h_a(x) = h_a(y) \Rightarrow \sum_{i=0}^{r} a_i x_i = \sum_{i=0}^{r} a_i y_i \mod m \)
  \[ \Rightarrow a_0 x_0 + \sum_{i=1}^{r} a_i x_i = a_0 y_0 + \sum_{i=1}^{r} a_i y_i \mod m \]
  \[ \Rightarrow a_0 (x_0 - y_0) = -\sum_{i=1}^{r} a_i (x_i - y_i) \mod m \]
  but \( m \) is prime \( \Rightarrow (x_0 - y_0) \) has an inverse for multiplication \( \mod m \) (multiplication \( \mod m \) is a finite field, Galois field)
  \[ \Rightarrow a_0 = -\frac{\sum_{i=1}^{r} a_i (x_i - y_i)}{(x_0 - y_0)} \mod m \]

- There are \( m^r \) such values for \( a_0 \)

- \( x \) and \( y \) collide once for each value of such value of \( a_0 \)
  \[ \Rightarrow x \) and \( y \) collide with probability \( \frac{m^r}{m^{r+1}} = 1/m \) \qquad q.e.d \]
Open addressing

- All elements are stored in hash table
  Each table entry contain either an element or NIL

- Search considers systematically (but not linearly) table slots
  until element is found or it becomes clear element is not in table

- Sequence of slots to be examined is **computed**

- no lists, no elements stored outside table
  \( \Rightarrow \) When the table is “filled up”, no element can be inserted
  \( \Rightarrow \) Load factor \( \alpha < 1 \)

- Advantage: avoids pointers.
  Extra memory freed by pointers can be used to increase the
  number of slots in table
**Insertion** in open addressing

**Principle**

- Successively examine or **probe** hash table until we find an empty slot
- Slots are not visited linearly, $\Theta(n)$ search time for empty slot, positions probed are computed from key to be inserted
- Hash function includes the probe number $i$, $h(k) \rightarrow h(k, i)$

$$h : U \times \{0, 1, 2, \ldots, m - 1\} \longrightarrow \{0, 1, 2, \ldots, m - 1\}$$

- For every key $k$, the **probe sequence** $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1) \rangle$ must be a permutation of $\langle 0, 1, \ldots, m - 1 \rangle$ (so that every position is eventually considered as a slot for a new key as table fills up)
**Insertion** in open addressing

Assumption: elements are keys with no satellite data

\[
\text{HASH-INSERT}(T, k) \\
1 \quad i \leftarrow 0 \\
2 \quad \text{repeat} \quad j \leftarrow h(k, i) \\
3 \quad \quad \text{if} \ T[j] = \text{NIL} \\
4 \quad \quad \quad \text{then} \ T[j] \leftarrow k \\
5 \quad \quad \text{return} \ j \\
6 \quad \quad \text{else} \ i \leftarrow i + 1 \\
7 \quad \text{until} \ \ i = m \\
8 \quad \text{error} \ “\text{hash table overflow}”
\]

Each slot contains a key or Nil
**Searching** in open addressing

Hash-Search(T, k) probes same sequence of slots that Hash-Insert(T, k) examined when inserting the key

```
HASH-SEARCH(T, k)
1   i ← 0
2   repeat  j ← h(k, i)
3       if T[j] = k
4           then return j
5       i ← i + 1
6   until T[j] = NIL or i = m
7   return NIL
```

Returns j if slot j contains key k, or NIL if key k is not present in T

When search finds an empty slot, search stops (k would have been inserted there and not later), assuming keys are not deleted from hash table
**Deletion** in open addressing is difficult

Solution:

- When deleting a key from slot $i$, don’t put Nil, because we won’t be able to search for another key for which we probed slot $i$, found it busy. Instead mark slot with value *Deleted* instead of *Nil*

- In this case, **Hash-Search** should keep looking when $t$ when it finds *Deleted*, while **Hash-Search** treats such a slot as if it was empty