Computer Science 331
Hash Functions

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #20

Outline

What is a Hash Function?

A *hash function* is a function

$$h : U \to \{0, 1, \ldots, m-1\}$$

where U and m are as follows.

- U: The "universe" of possible keys (generally finite, but extremely large)
- m: The size of the hash table T

This kind of hash function is useful for

- hash tables with chaining
- use as $h_0(k) = h(k, 0)$ for hashing with open addressing (using linear or quadratic probing, or double hashing)

Desirable Properties of Hash Functions

A hash function should be

- well-defined
- easy to compute

Explanation:

- If the hash function is not well-defined, so that there is no (single) value for $h(k)$ for some key k, then h cannot be used to place k within the hash table.
- If the hash function is difficult or expensive to compute, then operations on the hash table might be too expensive for this data structure to be useful.

Desirable Properties of Hash Functions

A hash function should distribute keys evenly throughout the hash table.

One Part of This Requirement: For $0 \leq i < m$, let

$$U_i = \{k \in U \mid h(k) = i\} \ .$$

In order to ensure that *random data* is evenly distributed, the *size* of each set U_i should be as close as possible to $|U|/m$ — that is,

$$\text{either } |U_i| = \left\lfloor \frac{|U|}{m} \right\rfloor \text{ or } |U_i| = \left\lceil \frac{|U|}{m} \right\rceil$$

for each i.

A Poor Choice

Example of a Function That Fails This Test: Suppose

$$U = \{0, 1, \ldots, 10^9 - 1\},$$

so that keys are nine-digit nonnegative integers, and that

$$m = 40 \ .$$

Claim:

The function

$$h(x) = \textit{(first two digits of } x\textit{)} \bmod 40$$

does not satisfy the previous condition.

Proof of the Claim

Proof.

$|U|/m := 10^9/40 = 25000000$

U_{19} : all integers between 0 and $10^9 - 1$ that start with 19, 59, or 99

- $|U_{19}| = 3(1 + 10 + 100 + 1000 + \cdots + 10^7) = 33333333$

U_{20} : all integers between 0 and $10^9 - 1$ that start with 20 or 60

- $|U_{20}| = 2(1 + 10 + 100 + 1000 + \cdots + 10^7) = 22222222$

$\square$

Consequence: the hash function does not distribute keys evenly throughout the hash table

Desirable Properties of Hash Functions

A hash function should distribute keys evenly throughout the hash table.

Another Part of This Requirement:

- As much as possible, *non-random data* should be evenly distributed throughout the table as well.

Unfortunately this requirement cannot be completely or perfectly satisfied!

However, hash functions that fail to distribute evenly certain *common kinds* of non-random data should also be avoided.

Example: Spacial Locality

Spatial Locality: frequently-used resources are often clustered close together.

Examples:

- The first two digits of student IDs at the University of Calgary were once the same as the last two digits of the first year of the student's program
- Early digits of an employee's ID number might indicate the *department* in which the employee works, or (for a large company) the employee's geographical location

A Poor Choice

In the above examples, a hash function like the previous example

$$h(x) = (\text{first two digits of } x) \bmod 40$$

would be a terrible choice!

Example: Consider the hash table shape if this was used when keys were the ID numbers for (older) students at U of C!

Conclusion: A function is generally a poor choice for a hash function if it maps many keys that are close together to the same location.

Principles

The following *principles* should be followed in order to avoid some of the problems already mentioned.

1. Calculation of the hash function should involve the *entire search key* — not just a part of it.
2. If a hash function uses modular arithmetic then *the base should be prime*, that is, if h has the form

   $$h(x) = x \bmod m$$

   then m should be a prime number.

Note: The examples from the previous lecture did not follow these principles — they were designed to be easy to understand rather than to be useful in practice!

Interpreting Keys as Natural Numbers

Common Situation: The key is a character string over some *alphabet* Σ (eg. the ASCII character set)

Useful First Step: Map each string α to a natural number by

- mapping each symbol in the alphabet to a value between 0 and $B-1$, where $B = |\Sigma|$
- using this mapping to map each of the symbols in α to an integer between 0 and $B-1$, in order to form a base-B (or "radix-B") integer

Example: for ASCII character strings we have $B = 128$ (each ASCII character maps to an integer between 0 and 127)

Example

Consider the string “rabbit” — using `www.lookuptables.com` we obtain:

r	a	b	b	i	t
114	97	98	98	105	116

We would then map the string “rabbit” to the natural number

$$114 \times 128^5 + 97 \times 128^4 + 98 \times 128^3 + 98 \times 128^2 + 105 \times 128 + 116.$$

Written in standard form (as a decimal integer), this is

$$3943255553268$$

The value $h(\text{rabbit})$ will be $\widehat{h}(3943255553268)$, for a function

$$\widehat{h} : \mathbb{N} \to \{0, 1, \ldots, m-1\}$$

The Division Method

Assumption from now on: k is an *integer* (eg: 3943255553268).

Division Method: Choose

$$h(k) = k \bmod m$$

where m is the hash table size.

Poor Choices for m: m should not have (lots of) small factors.

- In particular, m should certainly not be a power of either two or ten — or *close* to any such powers!

Good Choice: Choose m to be a *prime number* (but *not* close to 2^ℓ or 10^ℓ for any integer ℓ).

Implementation Issue

Implementation Issue: Avoiding *overflow* when applying a hash function to a character string

Example: Suppose we wish to use the division method, when $m = 37$ and our search keys are character strings (using the ASCII character set).

- We would like to be able to decide that “rabbit” should be hashed to
$$3943255553268 \bmod 37 = 24$$
without having to compute the extremely large integer,
$$3943255553268$$
along the way!

Implementation Issue

Useful Properties:

- *Horner's Rule:* For natural numbers $a_n, a_{n-1}, \ldots, a_0$, evaluate $a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \cdots + a_0$ using the formula
$$(\cdots((a_n \cdot B + a_{n-1}) \cdot B + a_{n-2}) \cdot B + \cdots + a_0)$$
- For integers x and y,
$$(x + y) \bmod m = ((x \bmod m) + (y \bmod m)) \bmod m$$
- For integers a and x,
$$(a \cdot x) \bmod m = ((a \bmod m) \cdot (x \bmod m)) \bmod m$$

Implementation Issue (cont.)

Application: If the symbols in a character string α are mapped to the numbers $a_n, a_{n-1}, \ldots, a_0$ (from right to left), then $h(\alpha)$ can be computed as follows:

$c = a_n \bmod m$
$i = n$
while $(i > 0)$ **do**
  $i = i - 1$
  $c = (((c \cdot (B \bmod m)) \bmod m) + (a_i \bmod m)) \bmod m$
end while
return c

Note: If m is not too large and $B < m$, the complicated expression can be simplified (by removing some of the middle divisions with remainder by m) without causing overflow.

The Multiplication Method

If k is a natural number, $h(k)$ is a function that depends on a real number A such that $0 < A < 1$ and that is computed as follows.

$c = k \times A$
$c = c - \lfloor c \rfloor$
$h(k) = \lfloor m \times c \rfloor$ (an integer between 0 and $m - 1$)

In the middle step we are computing the "fractional part" of the real number c. For example, if we generated a real number

$$27.532986\ldots$$

after step 1, then we would obtain $0.532986\ldots$ after step 2.

Implementation Issue

Difficulty: This involves multiplication by a real number. Computers cannot generally perform such multiplications exactly.

Solution: Suppose (as usual) that a word of computer memory can be used to represent an integer between 0 and $2^w - 1$ for some natural number w.

- Choose $A = s/2^w$ for some $s \in \mathbb{N}$ such that $0 < s < 2^w$.
- *Implication:* If $0 \le k < 2^w - 1$ then $h(k)$ can be computed correctly using "double-precision" integer arithmetic.

Exercise: Figure out how to use *Horner's Rule* to compute h efficiently and accurately when keys are character strings!

Additional Details

Advantage of This Method: The choice of m is less critical.

What is Important, Instead?

- Some choices of A work better than others!
- Optimal choice depends on characteristics of data being hashed.
- *Knuth:* Choosing

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

  is likely to work well.

Universal Hashing

Universal Hashing:

- A method of choosing the *hash function* in a random way
- Works well if
  - The data to be hashed is “static” (it does not change much, or at all, over time)
  - The hash function is chosen *independently* of the data to be hashed.

The probability that the randomly chosen hash function works poorly, on *any* fixed set of data, is provably small!

See Section 11.3.3 of *Introduction to Algorithms* for additional details.

References

- Frank M. Carrano and Janet J. Prichard
  Data Abstraction & Problem Solving with Java:
  Walls & Mirrors
  Second Edition, Pearson Education, 2006
- Donald E. Knuth
  The Art of Computer Programming
  Volume 3: Sorting and Searching
  Addison-Wesley, 1973
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein Introduction to Algorithms
  Second Edition, McGraw-Hill, 2001
- Textbook, Chapter 9.3 and 9.4