Lecture 10 - Hashing
Prof. Manolis Kellis

Data Structures

- Role of data structures:
  - Encapsulate data
  - Support certain operations (e.g., INSERT, DELETE, SEARCH)
- Our focus: efficiency of the operations
- Algorithms vs. data structures

Symbol-table problem

Symbol table $T$ holding $n$ records:

- Key
- Other fields containing satellite data

Operations on $T$:
- INSERT($T$, $x$)
- DELETE($T$, $x$)
- SEARCH($T$, $k$)

How should the data structure $T$ be organized?

Direct-access table

**Idea:** Suppose that the set of keys is $K \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct.
Set up an array $T[0 \ldots m-1]$:

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } \text{key}[x] = k, \\ \text{NIL} & \text{otherwise}. \end{cases}$$

Then, operations take $\Theta(1)$ time.

**Problem:** The range of keys can be large. E.g.:
- 64-bit numbers (which represent $18,446,744,073,709,551,616$ different keys)
- Character strings (even larger!)

Hash functions

**Solution:** Use a hash function $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

When a record to be inserted maps to an already occupied slot in $T$, a collision occurs.
Resolving collisions by chaining

- Records in the same slot are linked into a list.

\[ h(49) = h(86) = h(52) = i \]

- Run time analysis requires key assumption:

Simple uniform hashing

We make the assumption of simple uniform hashing:

- Each key \( k \in K \) of keys is equally likely to be hashed to any slot of table \( T \), independent of where other keys are hashed.

Let \( n \) be the number of keys in the table, and let \( m \) be the number of slots.

Define the load factor of \( T \) to be

\[ \alpha = \frac{n}{m} \]

= average number of keys per slot.

Search cost for chaining under simple uniform hashing

Expected time to search for a record with a given key = \( \Theta(1 + \alpha) \).

Expected search time = \( \Theta(1) \) if \( \alpha = O(1) \), or equivalently, if \( n = O(m) \).

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

1. Division method

Assume all keys are integers, and define \( h(k) = k \mod m \).

Deficiency: Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

Extreme deficiency: If \( m = 2^r \), then the hash doesn’t even depend on all the bits of \( k \):

- If \( k = \text{1011000111011010}_2 \) and \( r = 6 \), then \( h(k) = 0110102 \).
1. Division method (continued)

\[ h(k) = k \mod m. \]

Pick \( m \) to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

**Annoyance:**
- Sometimes, making the table size a prime is inconvenient.
- But, this method is popular, although the next method we’ll see is usually superior.

2. Multiplication method

Assume that all keys are integers, \( m = 2^w \), and our computer has \( w \)-bit words. Define

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r), \]

where rsh is the “bit-wise right-shift” operator and \( A \) is an odd integer in the range \( 2^{w-1} < A < 2^w \).

- Don’t pick \( A \) too close to \( 2^w \).
- Multiplication modulo \( 2^w \) is fast.
- The rsh operator is fast.

2. Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & = A \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & = k \\
\end{array}
\]

\[ A = 3A \]

\[ h(k) = 10010100110011 \]

**Modular wheel**

Don’t pick \( A \) too close to \( 2^w \) (explore more of the wheel space)

2. Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 1 & = k \\
\end{array}
\]

\[ A = 10101011 \]

\[ h(k) = 10010100110011 \]

**Modular wheel**

Don’t pick \( A \) too close to \( 2^w \) (have balance of 1s and 0s)

3. Dot-product method

**Randomized strategy:** Each value \( 0 \leq k_i < m \), with \( m \) prime.

- Let \( m \) be prime.
- Decompose \( k \) into \( r+1 \) digits, each in \( \{0,1,..,m-1\} \).
- Pick random vector \( a \), similarly decomposed (each \( a_i \) chosen randomly from \( \{0,1,..,m-1\} \).
- Calculate dot product \( k \cdot a \), each multiplication \( \mod m \)
- Excellent in practice, but expensive to compute.

**Intro and definition**

**Hashing in practice**

**Universal hashing**

**Perfect hashing**

**Open Addressing**
A weakness of hashing

**Problem:** For any hash function \( h \), a set of keys exists that can cause the average access time of a hash table to skyrocket.

- Adversary can pick all keys st. \( \{k \in U : h(k) = i\} \) for some slot \( i \) (e.g. denial-of-service attacks).

**Idea:** Choose the hash function at random, independently of the keys.

- Even if an adversary can see your code, they cannot find a bad set of keys, since they don’t know exactly which hash function will be chosen.

---

Universal hashing: Definition

- Let \( U \) be a universe of keys.
- Let \( H \) be a finite collection of hash functions, each mapping \( U \) to \( \{0, 1, \ldots, m-1\} \).
- \( H \) is **universal** if for all \( x, y \in U \), where \( x \neq y \), we have \(|\{h \in H : h(x) = h(y)\}| = |H|/m\), i.e. only \( 1/m \) of hash functions in \( H \) result in \( x, y \) collision.

- Thus, if we choose \( h \) randomly from \( H \), the chance of a collision between \( x \) and \( y \) is \( 1/m \).

---

Universality is good

**Theorem.** Let \( h \) be a hash function chosen (uniformly) at random from a universal set \( H \) of hash functions. Suppose \( h \) is used to hash \( n \) arbitrary keys into the \( m \) slots of a table \( T \). Then, for a given key \( x \), we have

\[
E[\text{#collisions with } x] < n/m.
\]

---

Proof of theorem

**Proof.** Let \( C_x \) be the random variable denoting the total number of collisions of keys in \( T \) with \( x \), and let

\[
c_{xy} = \begin{cases} 
1 & \text{if } h(x) = h(y), \\
0 & \text{otherwise}.
\end{cases}
\]

Note: \( E[c_{xy}] = 1/m \) and \( C_x = \sum_{y \in T - \{x\}} c_{xy} \).

---

Proof (continued)

\[
E[C_x] = E\left[\sum_{y \in T - \{x\}} c_{xy}\right] = \sum_{y \in T - \{x\}} E[c_{xy}]
\]

- Take expectation of both sides.

---

Proof (continued)

- Take expectation of both sides.

- Linearity of expectation.
Proof (continued)

\[ E[C_x] = E \left[ \sum_{y \in T \setminus \{x\}} c_{xy} \right] \]

• Take expectation of both sides.

\[ = \sum_{y \in T \setminus \{x\}} E[c_{xy}] \]

• Linearity of expectation.

\[ = \sum_{y \in T \setminus \{x\}} 1/m \]

• \( E[c_{xy}] = 1/m \).

\[ = \frac{n-1}{m}. \quad \square \]

• Algebra.

---

Constructing a set of universal hash functions

Let \( m \) be prime. Decompose key \( k \) into \( r+1 \) digits, each with value in the set \( \{0, 1, \ldots, m-1\} \). That is, let \( k = (k_0, k_1, \ldots, k_r) \), where \( 0 \leq k_i < m \).

**Randomized strategy w/ dot-product method:**

Pick \( a = (a_0, a_1, \ldots, a_r) \) where each \( a_i \) is chosen randomly from \( \{0, 1, \ldots, m-1\} \) indpt of input.

Define \( h_a(k) = \sum_{i=0}^r a_i k_i \mod m \).

**Dot product, modulo \( m \)**

How big is \( H = \{h_a\} \)? \( |H| = m^{r+1} \).

---

Universality of dot-product hash functions

**Theorem.** The set \( H = \{h_a\} \) is universal.

**Proof.** Suppose that \( x = (x_0, x_1, \ldots, x_r) \) and \( y = (y_0, y_1, \ldots, y_r) \) are distinct keys. Thus, they differ in at least one digit position, w.l.o.g. position 0 (and possibly more positions).

For how many \( h_a \in H \) do \( x \) and \( y \) collide?

\[ h_a(x) = h_a(y) \iff \sum_{i=0}^r a_i x_i = \sum_{i=0}^r a_i y_i \pmod{m} \]

---

Fact from number theory

**Theorem.** Let \( m \) be prime. For any \( z \in \mathbb{Z}_m \) such that \( z \neq 0 \), there exists a unique \( z^{-1} \in \mathbb{Z}_m \) such that \( z \cdot z^{-1} \equiv 1 \pmod{m} \).

\( z^{-1} \) is known as the multiplicative inverse of \( z \).

**Example:** \( m = 7 \).

<table>
<thead>
<tr>
<th>( z )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z^{-1} )</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

**Explanation.** \( \gcd(z,m)=1, \ zx+my=1, \ z=1 \pmod{m} \), where \( (z^{-1},x,y) \) is the output of \( \text{EXTENDED-EUCLID}(z,m) \).

**Group theory.** \( (\mathbb{Z}_n^*, \cdot_n) \) is a finite abelian group.

(See Chapter 31, Proof: 31.13 and 31.26).
We have
\[ a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m}, \]
and since \( x_0 \neq y_0 \), an inverse \((x_0 - y_0)^{-1}\) must exist, which implies that
\[ a_0 \equiv \left( -\sum_{i=1}^{r} a_i(x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \pmod{m}. \]
Thus, for any choices of \( a_1, a_2, \ldots, a_r \), exactly one choice of \( a_0 \) causes \( x \) and \( y \) to collide.

**Proof (completed)**

**Q.** How many \( h_a \)'s cause \( x \) and \( y \) to collide?

**A.** There are \( m \) choices for each of \( a_1, a_2, \ldots, a_r \), but once these are chosen, exactly one choice for \( a_0 \) causes \( x \) and \( y \) to collide, namely
\[ a_0 = \left( -\sum_{i=1}^{r} a_i(x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \pmod{m}. \]
Thus, the number of \( h_a \)'s that cause \( x \) and \( y \) to collide is \( m^r - 1 = m^r = |H|/m. \)

**Perfect hashing**

Given a set of \( n \) keys, construct a static hash table of size \( m = O(n) \) such that \textsc{search} takes \( \Theta(1) \) time in the worst case.

**Idea:** Two-level scheme with universal hashing at both levels.

**No collisions at level 2!**

**Collisions at level 2**

**Theorem.** Let \( H \) be a class of universal hash functions for a table of size \( m = n^2 \). Then, if we use a random \( h \in H \) to hash \( n \) keys into the table, the expected number of collisions is at most \( 1/2 \).

**Proof.** By the definition of universality, the probability that 2 given keys in the table collide under \( h \) is \( 1/m = 1/n^2 \). Since there are \( \binom{n}{2} \) pairs of keys that can possibly collide, the expected number of collisions is
\[ E[X] = \frac{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}. \]
No collisions at level 2

**Corollary.** The probability of no collisions is at least 1/2.

**Proof.** Markov’s inequality says that for any nonnegative random variable $X$, we have

$$\Pr\{X \geq t\} \leq \frac{E[X]}{t}.$$  

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most 1/2.

Thus, just by testing random hash functions in $\mathcal{H}$, we’ll quickly find one that works.

Analysis of storage

- For level-1 hash table $T$, choose $m = n$.
- Rand. var. $n_i = \#$ of keys that hash to slot $i$ in $T$.
- If we use $n_i^2$ slots for the level-2 hash table $S$, expected total storage required for the two-level scheme is:

$$E\left[\sum_{i=0}^{m-1} \Theta(n_i^2)\right] = \Theta(n),$$  

(the analysis is identical to the analysis of bucket sort expected running time from recitation).

For probability bound, apply Markov inequality.

Intro and definition

Hashing in practice

Universal hashing

Perfect hashing

Open addressing

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function $h(k,i)$ depends on both the key $k$ and the probe number $i$:

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$  

- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible).

Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

collision
Example of open addressing

Insert key \( k = 496 \):

0. Probe \( h(496,0) \)
1. Probe \( h(496,1) \)
2. Probe \( h(496,2) \)

Search for key \( k = 496 \):

0. Probe \( h(496,0) \)
1. Probe \( h(496,1) \)
2. Probe \( h(496,2) \)

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing strategies

**Linear probing:**

Given an ordinary hash function \( h'(k) \), linear probing uses the hash function

\[
    h(k,i) = (h'(k) + i) \mod m.
\]

This method, though simple, suffers from *primary clustering*, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

**Double hashing**

Given two ordinary hash functions \( h_1(k) \) and \( h_2(k) \), double hashing uses the hash function

\[
    h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m.
\]

This method generally produces excellent results, but \( h_2(k) \) must be relatively prime to \( m \) (otherwise cycle of \( \leq m \) elements, not all slots are probed).

One way is to make \( m \) a power of 2 and design \( h_2(k) \) to produce only odd numbers.

Analysis of open addressing

We make the assumption of *uniform hashing*:

- Each key is equally likely to have any one of the \( m! \) permutations as its probe sequence.

**Theorem.** Given an open-addressed hash table with load factor \( \alpha = n/m < 1 \), the expected number of probes in an unsuccessful search is at most \( 1/(1-\alpha) \).
Proof (continued)

Therefore, the expected number of probes is
\[ 1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \ldots \left( 1 + \frac{1}{m-n+1} \right) \ldots \right) \right) \right) \]
\[ \leq 1 + \alpha \left( 1 + \alpha \left( 1 + \ldots \right) \right) \]
\[ \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \]
\[ = \sum_{i=0}^{\infty} \alpha^i \]
\[ = \frac{1}{1 - \alpha} \]

The textbook has a more rigorous proof.

Implications of the theorem

• If \( \alpha \) is constant, then accessing an open-addressed hash table takes constant time.
• If the table is half full, then the expected number of probes is \( \frac{1}{1-0.5} = 2 \).
• If the table is 90% full, then the expected number of probes is \( \frac{1}{1-0.9} = 10 \).

Q: What about DELETE(\( k \))

• A: Use two special values, NIL and DEL.
• NIL = nothing was ever here.
• DEL = something was here but was deleted.
• Search continues past DEL slots, but can store elements there. Hence, subsequent INSERT operations and remove DEL values.
• Rebuild the table when too many DEL values accumulate.

Hashing lecture outline

Intro and definition
Hashing in practice
Universal hashing
Perfect hashing
Open Addressing

Summary (rehash)

• Hashing: map large \( U \) in a small space \( \{1, \ldots, m\} \). Content-based addressing, constant-time lookup!
• Hash functions in practice: div, mult, dot-prod.
• Universal hashing: rand \( h \in H \), prob collision \( 1/m \).
• Perfect hashing: \( n \) fixed keys known in advance, worst-case \( O(1) \) lookup in \( \Theta(n) \) space. 2-level \( h \).
• Open addressing: \( h(k, i) \) probe until empty slot, linear probing, double hashing.