Advanced Algorithmics (6EAP)
MTAT.03.238
Hashing
Jaak Vilo
2013 Spring

ADT – associative array

• INSERT, SEARCH, DELETE
  • An associative array (also associative container, map, mapping, dictionary, finite map, and in query-processing an index or index file) is an abstract data type composed of a collection of unique keys and a collection of values, where each key is associated with one value (or set of values). The operation of finding the value associated with a key is called a lookup or indexing, and this is the most important operation supported by an associative array.

Some reading ...

• MIT

• CMU:

Symbol-table problem

Symbol table $S$ holding $n$ records:

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

How should the data structure $S$ be organized?

Direct-access table

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

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

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

**Problem:** The range of keys can be large:
- 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.
Keys

- Integers
- Strings
- Floating point numbers...
- Usual assumption – keys are integers.
- Step 1: Map keys to integers.
### 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 \( k = 1011000111011010 \), and \( r = 6 \), then \( h(k) = 011010_2 \).  

### 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.

### Multiplication method

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

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

where \( \operatorname{rsh} \) is the “bitwise 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 \( 2^{w-1} \) or \( 2^w \).
- Multiplication modulo \( 2^w \) is fast compared to division.
- The \( \operatorname{rsh} \) operator is fast.

### Multiplication method example

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

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

\[
\begin{array}{c}
1011001 \\
\times \quad 1101011 \\
\hline
1001010011 \quad 10011 \\
\hline
\end{array}
\]

**Modular wheel**

### 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 depends on both the key and probe number:
  \( h: U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} \).
- The probe sequence \( \{h(k, 0), h(k, 1), \ldots, h(k, m-1)\} \) 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 \( h(496, 0) \)

\[
\begin{array}{c}
| \hline
| 0 & 586 & 133 & 584 & 481 & 31 \hline
\end{array}
\]

\( m-1 \)
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.

Quadratic probing:
$$h(k, i) = (h'(k) + c_1i + c_2i^2) \mod m$$

Double hashing:
$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$
Quadratic Probing

- Suppose that an element should appear in bin $h$:
  - If bin $h$ is occupied, then check the following sequence of bins:
    - $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$, ...  
    - $h + 1$, $h + 4$, $h + 9$, $h + 16$, $h + 25$, ...
- For example, with $M = 17$:

---

Probing strategies

**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 $m$. 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

$$1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \cdots \left( 1 + \frac{1}{m-n+1} \right) \cdots \right) \right) \right)$$

$$\leq 1 + \alpha (1 + \alpha (1 + \alpha \cdots))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

The textbook has a more rigorous proof and an analysis of successful searches.

---
Birthday Paradox

- In probability theory, the birthday problem, or birthday paradox\(^1\), pertains to the probability that in a set of randomly chosen people some pair of them will have the same birthday. In a group of at least 23 randomly chosen people, there is more than 50% probability that some pair of them will both have been born on the same day.

Problem

- Adversary can choose a really bad set of keys
- E.g. the identifiers for the compiler...
- By mapping all keys to the same location a worst-case scenario can happen

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.
- An adversary can pick all keys from \( \{ k \in U : h(k) = i \} \) for some slot \( i \).

**Idea:** Choose the hash function at random, independently of the keys.
- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn’t know exactly which hash function will be chosen.
Two possible views of hashing

- The hash function $h$ is fixed. The keys are random.
- The hash function $h$ is chosen randomly from a family of hash functions. The keys are fixed.

Typical assumption (in both scenarios):

$$\Pr[h(k_1) = h(k_2) \mid k_1 \neq k_2] \leq \frac{1}{m}$$

Universal hashing

Universal hashing is a randomized algorithm for selecting a hash function $F$ with the following property: for any two distinct inputs $x$ and $y$, the probability that $F(x) = F(y)$ (i.e., that there is a hash collision between $x$ and $y$) is the same as if $F$ was a random function. Thus, if $F$ has function values in a range of size $r$, the probability of any particular hash collision should be at most $1/r$. There are universal hashing methods that give a function $F$ that can be evaluated in a handful of computer instructions.

Universal families of hash functions

A family $H$ of hash functions from $U$ to $[m]$ is said to be universal if and only if

$$\text{for every } k_1 \neq k_2 \in U \text{ we have } \Pr_{h \in H}[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

Universality is good

Theorem. Let $h$ be a hash function chosen (uniformly) at random from a universal set $\mathcal{H}$ 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

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

\( U = [p] = \{0, 1, \ldots, p - 1\} \), where \( p \) is prime

\[
H_{p,m} = \{h_{a,b} \mid 1 \leq a < p, 0 \leq b < p\} \\
h_{a,b}(k) = ((ak + b) \mod p) \mod m
\]

- \( p = 17, m = 6 \implies h_{3,4}(28) = 3 \)
- \( h_{5,2}(3) = 5 \) \( h_{10,2}(3) = 15 \mod 6 = 3 \)

### Perfect hashing

Suppose that \( D \) is static.
We want to implement \( \text{Find} \) in \( O(1) \) worst case time.

Can we achieve it?

### PHF and MPHF

![Perfect hashing: No collisions](image)

**Figure 1:** (a) Perfect hash function. (b) Minimal perfect hash function.

### Expected no. of collisions

**Suppose that \( |D| = n \) and that \( h \) is randomly chosen from a universal family.**

\[
\text{Collisions:} \\
Col = \{\{k_1, k_2\} \subseteq D \mid k_1 \neq k_2, h(k_1) = h(k_2)\}
\]

\[
E[|Col|] = \sum_{\{k_1, k_2\} \subseteq D} \Pr[h(k_1) = h(k_2)] \leq \left(\frac{n}{2}\right) \frac{n}{m}
\]

**Corollary 1:** If \( m = n \), then \( E[|Col|] < \frac{n}{2} \)

**Corollary 2:** If \( m = n^2 \), then \( E[|Col|] < \frac{1}{2} \)
Expected no. of collisions

Markov's inequality: \( \Pr[X \leq 2E[X]] \geq \frac{1}{2} \)

Corollary 1: If \( m = n \), then \( E[|Col|] < \frac{n}{2} \)

Corollary 1': If \( m = n \), then \( \Pr[|Col| < n] \geq \frac{1}{2} \)

Corollary 2: If \( m = n^2 \), then \( E[|Col|] < \frac{1}{2} \)

Corollary 1': If \( m = n^2 \), then \( \Pr[|Col| < 1] \geq \frac{1}{2} \)

If we are willing to use \( m = n^2 \), then any universal family contains a perfect hash function.

2-level hashing

Let \( h \) be a function from \( H(n) \) and compute the number of collisions. If there are more than \( n \) collisions, repeat.

For each cell \( i \), if \( n_i > 1 \), choose a random hash function from \( H(n_i^2) \). If there are any collisions, repeat.

Expected construction time – \( O(n) \)

Worst case find time – \( O(1) \)

Other applications of hashing

- Comparing files
- Cryptographic applications
- Web indexing
- Keyword lookup

• VERY LARGE APPLICATIONS
<table>
<thead>
<tr>
<th>Problem with hashing</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Problem: Sorting and ‘next by value’ is hard</td>
<td>• Example: Enumerating the search space – have we seen this “state” before?</td>
</tr>
<tr>
<td>• Q: how to make hash functions that maintain order</td>
<td>• Dynamic programming/FP/memoization – has the function been previously evaluated with the same input parameters?</td>
</tr>
<tr>
<td>• Q: how to scale hashing to very large data, with low memory and high speed</td>
<td></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem with hashing</td>
</tr>
<tr>
<td>The use of minimal perfect hash functions is, until now, restricted to scenarios where the set of keys being hashed is small, because of the limitations of current algorithms. But in many cases, to deal with huge set of keys is crucial. So, this project gives to the free software community an API that will work with sets in the order of billion of keys.</td>
</tr>
<tr>
<td>• CIMP Library was conceived to create minimal perfect hash functions for very large sets of keys</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
</tr>
<tr>
<td>• CHD Algorithm: It is the fastest algorithm to build PHFs and MPHFs in linear time.</td>
</tr>
<tr>
<td>It generates the most compact PHFs and MPHFs we know of.</td>
</tr>
<tr>
<td>It can be used to generate ε-perfect hash functions. A ε-perfect hash function</td>
</tr>
<tr>
<td>allows at most ε collisions in a given bin. It is a well-known fact that modern</td>
</tr>
<tr>
<td>memories are organized as blocks which constitute transfer unit. Example of such</td>
</tr>
<tr>
<td>blocks are cache lines for internal memory or sectors for hard disks. Thus, it can be</td>
</tr>
<tr>
<td>very useful for devices that carry out I/O operations in blocks.</td>
</tr>
<tr>
<td>It is a two level scheme. It uses a first level hash function to split the key set in</td>
</tr>
<tr>
<td>buckets of average size determined by a parameter b in the range [1,32]. In the</td>
</tr>
<tr>
<td>second level it uses displacement values to resolve the collisions that have given</td>
</tr>
<tr>
<td>rise to the buckets.</td>
</tr>
<tr>
<td>It can generate MPHFs that can be stored in approximately 2.07 bits per key.</td>
</tr>
<tr>
<td>For a load factor equal to the maximum one that is achieved by the BDZ algorithm</td>
</tr>
<tr>
<td>(</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilistic data structures…</td>
</tr>
<tr>
<td>• Just use hash tables to record what “has been seen”</td>
</tr>
<tr>
<td>• If the probability of a collision is small, then do not worry if some false positive hits have occurred</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filters (1970)</td>
</tr>
<tr>
<td>• Simple query: is the key in the set?</td>
</tr>
<tr>
<td>• Probabilistic:</td>
</tr>
<tr>
<td>– No: 100% correct</td>
</tr>
<tr>
<td>– yes: p \ll 1</td>
</tr>
<tr>
<td>• Idea – make p as large as possible (avoid false positive answers)</td>
</tr>
</tbody>
</table>
Bloom Filters

An example of a Bloom filter, representing the set \{x, y, z\}. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set \{x, y, z\}, because it hashes to one bit-array position containing 0. For this figure, \(m = 18\) and \(k = 3\).

Bloom filters

Bloom filter used to speed up answers in a key-value storage system. Values are stored on a disk which has slow access times. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.

Summary

• Load factors
• handling collisions
• Hash functions and families
• Universal hashing
• Perfect hashing
• Minimal perfect hashing
• Bloom filters