Advanced Algorithmics (6EAP)
MTAT.03.238
Hashing
Jaak Vilo
2012 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 $T[0 \ldots m-1]$:

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

Then, operations take $O(1)$ 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.
### Pigeonhole principle

In mathematics, the pigeonhole principle, also known as Dirichlet's box (or drawer) principle, is exemplified by such things as the fact that in a family of three children there must be at least two of the same gender. This principle states that, given two natural numbers $n$ and $m$ with $n > m$, if $n$ items are put into $m$ pigeonholes, then at least one pigeonhole must contain more than one item. Another way of stating this would be that $m$ holes can hold at most $m$ objects with one object to a hole; adding another object will force one to reuse one.

### Resolving collisions by chaining

- Link records in the same slot into a list.

  **Worst case:**
  - Every key hashes to the same slot.
  - Access time $= \Theta(n)$ if $|S| = n$

### Average-case analysis of chaining

We make the assumption of simple uniform hashing:
- Each key $k \in S$ 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

$$\alpha = \frac{n}{m} = \text{average number of keys per slot}.$$

### Search cost

The expected time for an unsuccessful search for a record with a given key $i$ is $= \Theta(1 + \alpha)$.

**apply hash function and access slot**

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

A successful search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook.)

### 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.

### Keys

- Integers
- Strings
- Floating point nr-s...

**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 \( k = 101100011111011010 \), and \( r = 6 \), then \( h(k) = 01101001_2 \).

**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
\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r), \]
where \( \text{rsh} \) 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 \( \text{rsh} \) operator is fast.

**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:
\[
\begin{array}{c}
1011001 \\
\times 1101011 \\
\hline
100101001110011
\end{array}
\]

\[ h(k) = 1000111 \]

**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 \( \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle \) 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 \( k = 496 \):

\[ T = \begin{array}{c}
0 \\
586 \\
133 \\
364 \\
481 \\
\hline
m-1
\end{array} \]

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

\[ 481 \text{ collision} \]
Primary clustering

Implementation of open addressing

How do we define $h(k, i)$?

Linear probing:
$$h(k, i) = (h'(k) + i) \mod m$$

Quadratic probing:
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

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

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 $h(496, 1)$
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.
Quadra FC

Probing

• Suppose that an element should appear in bin \( h \):
  – if bin \( h \) is occupied, then check the following sequence of bins:
    \( h + 1^2, h + 2^2, h + 3^2, h + 4^2, \ldots \)
    \( h + 1, h + 4, h + 9, h + 16, h + 25, \ldots \)
  • For example, with \( M = 17 \):

Quadra FC

Probing

• If one of \( h + i^2 \) falls into a cluster, this does not imply the next one will

Secondary clustering – all \( k \) colliding in \( h(k) \) will cluster to same locations...

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 \( \alpha = \frac{n}{m} < 1 \), the expected number of probes in an unsuccessful search is at most \( \frac{1}{1(1-\alpha)} \).

Proof of the theorem

Proof.
• At least one probe is always necessary.
• With probability \( \frac{n}{m} \), the first probe hits an occupied slot, and a second probe is necessary.
• With probability \( \frac{(n-1)(m-1)}{m(m-1)} \), the second probe hits an occupied slot, and a third probe is necessary.
• With probability \( \frac{(n-2)(m-2)}{m(m-1)} \), the third probe hits an occupied slot, etc.

Observe that \( \frac{n-i}{m-i} < \frac{n}{m} = \alpha \) for \( i = 1, 2, \ldots, n \).

Proof (continued)

Therefore, the expected number of probes is

\[ 1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \ldots \left( 1 + \frac{1}{m-n+1} \right) \ldots \right) \right) \right) \]
\[ \leq 1 + \alpha \left( 1 + \alpha \left( 1 + \alpha \left( \ldots \left( 1 + \alpha \right) \ldots \right) \right) \right) \]
\[ \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \]
\[ \sum_{i=0}^{\infty} \alpha^i \]
\[ = \frac{1}{1-\alpha} \]

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

- In probability theory, the birthday problem, or birthday paradox\(^\text{1}\) 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

\(^\text{1}\)See [this link](http://www.cs.cmu.edu/~ assertions/15451-s09/www/lectures/lect0212.pdf) for more information.
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

for every $k_1 \neq k_2 \in U$ we have

$$\Pr_{h \in H}[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

A simple universal family

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

$$H_{p, m} = \{h_{a,b} \mid 1 \leq a < p, 0 \leq b < p\}$$

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

To represent a function from the family we only need two numbers, $a$ and $b$.

The size $m$ of the hash table is arbitrary.
Example

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

\( H_{p,m} = \{ h_{a,b} \mid 1 \leq a < p, \ 0 \leq b < p \} \)

\( h_{a,b}(k) = ((ak + b) \mod p) \mod m \)

- \( p = 17, \ m = 6 \) \( \Rightarrow \) \( h_{3,4}(28) = 3 \)

Perfect hashing

Suppose that \( D \) is static.

We want to implement \texttt{Find} in \( O(1) \) worst case time.

Can we achieve it?

Perfect hashing

Suppose that \( D \) is static.

We want to implement \texttt{Find} in \( O(1) \) worst case time.

Perfect hashing: No collisions

Can we achieve it?

PHF and MPHF

![Diagram](image)

Expected no. of collisions

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

\[
\text{Collisions:} \quad Col = \{ \{ k_1, k_2 \} \subseteq D \mid k_1 \neq k_2, h(k_1) = h(k_2) \} \\
E[|Col|] = \sum_{\{ k_1, k_2 \} \subseteq D \mid k_1 \neq k_2} \Pr[h(k_1) = h(k_2)] \leq \frac{n}{m}
\]

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

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

10.5 Perfect Hashing

We say a hash function is perfect for \( S \) if all hashes involve \( O(1) \) work. Here are two methods for constructing perfect hash functions for a given set \( S \).

10.5.1 Method 1: an \( O(N^2) \)-space solution

Say we are willing to have a table whose size is quadratic in the size \( N \) of our dictionary \( S \). Then, here is an easy method for constructing a perfect hash function. Let \( H \) be universal and \( M = N^2 \). Then just pick a random \( b \) from \( H \) and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim 10.5 If \( H \) is universal and \( M = N^2 \), then \( PH_{a,b}(\text{no collisions on } S) \geq 1/2 \).

Proof:

- How many pairs \((x,y)\) in \( S \) are there? Answer: \( \binom{n}{2} \)
- For each pair, the chance they collide is \( 1/M \) by definition of “universal”.
- So, \( \Pr(\text{no a collision}) \leq \binom{n}{2}/M < 1/2 \) \( \blacksquare \)
Expected no. of collisions

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

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

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

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

2-level hashing

Choose \( m = n \) and \( h \) such that \( |\text{Col}| < n \)

Store the \( n_i \) elements hashed to \( i \) in a small hash table of size \( n_i^2 \)
using a perfect hash table \( h_i \)

A randomized algorithm for constructing a perfect 2-level hash table:

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

For each cell \( (i, n) \), choose a random hash function from \( H(n^2) \). 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
hashing

- Problem: Sorting and ‘next by value’ is hard
- Q: how to make hash functions that maintain order
- Q: how to scale hashing to very large data, with low memory and high speed

Examples

- Example: Enumerating the search space – have we seen this “state” before?
- Dynamic programming/FP/memoization – has the function been previously evaluated with the same input parameters?

- CMPH - C Minimal Perfect Hashing Library – [SourceForge](http://cmph.sourceforge.net/)
  - 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.
  - CMPH Library was conceived to create minimal perfect hash functions for very large sets of keys

- Compress, Hash and Displace: CHD Algorithm
  - CHD Algorithm: It is the fastest algorithm to build PHFs and MPHFs in linear time.
  - It generates the most compact PHFs and MPHFs we know of.
  - It can generate PHFs with a load factor up to 99%.
  - It can be used to generate t-perfect hash functions. A t-perfect hash function allows at most t collisions in a given bin. It is a well-known fact that modern memories are organized as blocks which constitute transfer unit. Example of such blocks are cache lines for internal memory or sectors for hard disks. Thus, it can be very useful for devices that carry out I/O operations in blocks.
  - It is a two level scheme. It uses a first level hash function to split the key set in buckets of average size determined by a parameter b in the range [1,32]. In the second level it uses displacement values to resolve the collisions that have given rise to the buckets.
  - It can generate MPHFs that can be stored in approximately 2.07 bits per key.
  - For a load factor equal to the maximum one that is achieved by the BDZ algorithm ([1,2]), the resulting PHFs are stored in approximately 1.40 bits per key.

Probabilistic data structures...

- Just use hash tables to record what “has been seen”
- If the probability of a collision is small, then do not worry if some false positive hits have occurred

Bloom filters (1970)

- Simple query: is the key in the set?
- Probabilistic:
  - No: 100% correct
  - Yes: p ≤ 1
- Idea – make p as large as possible (avoid false positive answers)
### 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 are 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.