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 \( S \):**
  - \( \text{INSERT}(S, x) \)
  - \( \text{DELETE}(S, x) \)
  - \( \text{SEARCH}(S, k) \)

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 \( \Theta(1) \) 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.
17.3.2013

Keys

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

Usual assumption – keys are integers.

Step 1: Map keys to integers.

Pigeonhole principle

From Wikipedia, the free encyclopedia

(Ordered from 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.

The inspiration for the name of the principle: pigeons in holes. Here \( n = 10 \) and \( m = 9 \) so we can conclude that some hole has more than one pigeon.

Resolving collisions by chaining

- Link records in the same slot into a list.

Worst case:
- Every key hashes to the same slot.
- Access time = \( O(n) \) 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

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

= average number of keys per slot.

Search cost

The expected time for an unsuccessful search for a record with a given key is

\[
= \Theta(1 + \alpha).
\]

The 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.
### 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 $k = 10110001111011010$, and $r = 6$, then $h(k) = 0110102$.

### 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, $m = 2^w$, and our computer has $w$-bit words. Define $h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r)$, where rsh is the “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 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:

$$\times\begin{array}{c} \hline 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline \end{array} = A$$

$$\begin{array}{c} 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ \hline \end{array} = k$$

$h(k)$

**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:
  $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$.
- 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 $h(496,0)$: 364
586
133
```

```
T: 0
586
133
364
```

```
0
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
Primary clustering

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

- **Linear probing:**
  \[
  h(k, i) = (h'(k) + i) \mod m
  \]

- **Quadratic probing:**
  \[
  h(k, i) = (h'(k) + c_1i + c_2i^2) \mod m
  \]

- **Double hashing:**
  \[
  h(k, i) = (h_1(k) + ih_2(k)) \mod m
  \]
17.3.2013

**Quadratic Probing**

- Suppose that an element should appear in bin $h$:
  - if bin $h$ is occupied, then check the following sequence of bins:
    - $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$, $h + 5^2$, ...
    - $h + 1$, $h + 4$, $h + 9$, $h + 16$, $h + 25$, ...
  - 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 of the theorem**

**Proof.**

- At least one probe is always necessary.
- With probability $n/m$, the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc.

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

Therefore, the expected number of probes is

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

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

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

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

**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

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

Universal families of hash functions

A family \( H \) of hash functions from \( U \) to \( [m] \) is said to be **universal** if and only if

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

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

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

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

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

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

Perfect hashing

Suppose that \( D \) is static.

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

Perfect hashing

Suppose that \( D \) is static.

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

Perfect hashing: No collisions

Can we achieve it?

Perfect hashing

Suppose that \( D \) is static.

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

Perfect hashing: No collisions

Can we achieve it?

PHF and MPHF

![PHF and MPHF](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

Collisions:

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

\[ E[|\text{Col}|] = \sum_{\{k_1, k_2\} \subseteq D} \Pr[h(k_1) = h(k_2)] = \frac{n(n-1)}{m} \]

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

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

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

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

\[ n_i = |h^{-1}(i)| = \left\{ k \in D \mid h(k) = i \right\} \]

\[ \sum_{i=0}^{n-1} n_i = n \]

\[ \sum_{i=0}^{n-1} \left( \frac{n_i}{2} \right) = |Col| \]

\[ 3n + 2 + \sum_{i=1}^{n_i} n_i^2 \]

\[ = 4n + 2|Col| \]

\[ \leq 6n + 2 \]

---

**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 \( (i, j) \), 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
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/memorization – has the function been previously evaluated with the same input parameters?

• CMPH - C Minimal Perfect Hashing Library
  – 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 99.9.
• It can be used to generate r-perfect hash functions. A r-perfect hash function allows at most r 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 (82%), 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
• See: http://pycon.blip.tv/file/4881076/

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)
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.