Hashing

Hash functions
Separate chaining
Linear probing
Double hashing


Optimize Judiciously

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity." - William A. Wulf

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." - Donald E. Knuth

"We follow two rules in the matter of optimization:
Rule 1: Don’t do it.
Rule 2 (for experts only). Don’t do it yet – that is, not until you have a perfectly clear and unoptimized solution.”
- M. A. Jackson

Reference: Effective Java by Joshua Bloch.

Hashing: Basic Plan

Save items in a key-indexed table. Index is a function of the key.

Hash function. Method for computing table index from key.

Collision resolution strategy. Algorithm and data structure to handle two keys that hash to the same index.

Classic space-time tradeoff.
- No space limitation: trivial hash function with key as address.
- No time limitation: trivial collision resolution = sequential search.
- Limitations on both time and space: hashing (the real world)

Choosing a Good Hash Function

Goal: scramble the keys.
- Each table position equally likely for each key.
  thoroughly researched problem

Ex: Social Security numbers.
- Bad: first three digits.
- Better: last three digits.

Ex: date of birth.
- Bad: first three digits of birth year.
- Better: birthday.

Ex: phone numbers.
- Bad: first three digits.
- Better: last three digits.
Strings hash functions.
- Java 1.1: calculation involving only 16 characters.
- Java 1.2: calculation involving all characters.

```java
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < length(); i++)
        hash = (31 * hash) + charAt(i);
    return hash;
}
```

- Equivalent to $h = 31^{N-1}s_0 + \ldots + 31^2s_2 + 31s_1 + s_{N-1}$.
- Can we use $h \mod M$ as index for table of size $M$?

Work to hash a string of length $W$.
- $W$ add, $W$ multiply, $1$ mod.
- Note: reference Java implementation caches String hash codes.

Collisions

Collision = two keys hashing to same value.
- Essentially unavoidable.
- Birthday problem: how many people will have to enter a room until two have the same birthday?
- With $M$ hash values, expect a collision after $\sqrt{\frac{M}{2}}$ insertions.

Conclusion: can’t avoid collisions unless you have a ridiculous amount of memory.

Challenge: efficiently cope with collisions.

Collision Resolution.

Two main approaches.

Separate chaining.
- $M$ much smaller than $N$.
- $\sim N / M$ keys per table position.
- Put keys that collide in a list.
- Need to search lists.

Open addressing.
- $M$ much larger than $N$.
- Plenty of empty table slots.
- When a new key collides, find next empty slot and put it there.
- Complex collision patterns.

Separate Chaining

Separate chaining: array of $M$ linked lists.
- Hash: map key to integer $i$ between 0 and $M-1$.
- Insert: put at front of $i$th chain. ← constant time
- Search: only need to search $i$th chain. ← proportional to length of chain

### Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>7121</td>
</tr>
<tr>
<td>me</td>
<td>3480</td>
</tr>
<tr>
<td>ishmael</td>
<td>5017</td>
</tr>
<tr>
<td>seriously</td>
<td>0</td>
</tr>
<tr>
<td>untravelled</td>
<td>3</td>
</tr>
<tr>
<td>suburban</td>
<td>3</td>
</tr>
</tbody>
</table>

M = 8191

$\text{hash codes}:
- 3045982
- 7121
- 8191

Horner's method
Symbol Table: Hash Table Implementation

```java
class SymbolTable {
    private int M = 8191; // number of chains (8191 is prime)
    private List[] st = new List[M];

    class List {
        // AS BEFORE
        public int hash(String s, int M) {
            return (s.hashCode() & 0x7fffffff) % M;
        }
        void put(String k, Object val) {
            int i = hash(k, M);
            st[i] = new List(k, val, st[i]);
        }
        Object get(String k) {
            int i = hash(k, M);
            for (List x = st[i]; x != null; x = x.next)
                if (k.equals(x.key)) return x.value;
            return null;
        }
    }
}
```

Hash Table Implementation: Performance

Advantages: fast insertion, fast search.
Disadvantage: hash table has fixed size.

Hash tables improves ALL symbol table clients.
- Makes difference between practical solution and no solution.
- Ex: Moby Dick now takes a few seconds instead of hours.

```
% java DeDup < mobydick.txt
moby
dick
herman
melville
call
me
ishmael
some
years
ago 210,028 words
...
16,834 distinct
```

Separate Chaining Performance

- Search cost is proportional to length of chain.
- Trivial: average length = N / M.
- Worst case: all keys hash to same chain.

**Theorem.** Let \( \alpha = N / M > 1 \) be average length of list. For any \( t > 1 \), probability that list length \( > t \alpha \) is exponentially small in \( t \).

**Parameters.**
- M too large \( \Rightarrow \) too many empty chains.
- M too small \( \Rightarrow \) chains too long.
- Typical choice: \( \alpha = N / M \approx 10 \) \( \Rightarrow \) constant-time search/insert.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>N / 2</td>
<td>N / 2</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>N / 2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
</tr>
</tbody>
</table>

* assumes hash function is random
Linear Probing

Linear probing: array of size $M$. Typically twice as many slots as elements.
- Hash: map key to integer $i$ between 0 and $M-1$.
- Insert: put in slot $i$ if free, if not try $i+1$, $i+2$, etc.
- Search: search slot $i$, if occupied but no match, try $i+1$, $i+2$, etc.

Cluster.
- Contiguous block of items.
- Search through cluster using elementary algorithm for arrays.

Double Hashing

Double hashing: avoid clustering by using second hash to compute skip for search.

Hash: map key to integer $i$ between 0 and $M-1$.
Second hash: map key to nonzero skip value. Best if relatively prime to $M$

Ex: $1 + (k \mod 97)$.

Avoids clustering.
- Skip values give different search paths for keys that collide.

Linear Probing Performance

Linear probing performance.
- Insert and search cost depend on length of cluster.
- Trivial: average length of cluster $\leq \alpha = N / M$. But elements more likely to hash to big clusters
- Worst case: all keys hash to same cluster.

Theorem (Knuth, 1962). Let $\alpha = N / M < 1$ be average length of list.

- Insert: $\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$
- Search: $\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$

Parameters.
- $M$ too large $\Rightarrow$ too many empty array entries.
- $M$ too small $\Rightarrow$ clusters coalesce.
- Typical choice: $M \approx 2N \Rightarrow$ constant-time search/insert.

Double Hashing Performance

Linear probing performance.
- Insert and search cost depend on length of cluster.
- Trivial: average length of cluster $\leq \alpha = N / M$.
- Worst case: all keys hash to same cluster.

Theorem. Let $\alpha = N / M < 1$ be average length of list.

- Insert: $\frac{1}{1-\alpha}$
- Search: $\frac{1}{\alpha} \ln(1+\alpha)$

Parameters.
- $M$ too large $\Rightarrow$ too many empty array entries.
- $M$ too small $\Rightarrow$ clusters coalesce.
- Typical choice: $M \approx 2N \Rightarrow$ constant-time search/insert.

Disadvantage: delete cumbersome to implement.
Hashing Tradeoffs

Separate chaining vs. linear probing/double hashing.
- Space for links vs. empty table slots.
- Small table + linked allocation vs. big coherent array.

Linear probing vs. double hashing.

<table>
<thead>
<tr>
<th>load factor $\alpha$</th>
<th>50%</th>
<th>66%</th>
<th>75%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
<tr>
<td>insert</td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
<td>55.5</td>
</tr>
<tr>
<td>double hashing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>insert</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Symbol Table: Java Libraries

Java has built-in libraries for symbol tables.
- HashMap = linear probing hash table implementation.

```
import java.util.HashMap;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap st = new HashMap();
        st.put("www.cs.princeton.edu", "128.112.136.11");
        st.put("www.princeton.edu", "128.112.128.15");
        st.put("www.simpsons.com", "209.052.165.60");
        System.out.println(st.get("www.cs.princeton.edu"));
    }
}
```

Duplicate policy.
- Java HashMap forbids two elements with the same key.
- Sedgewick implementations allow duplicate keys.

Implementing a HashMap Key

Java HashMap allows arbitrary objects as the key.
- Uses the equals and hashCode methods of the key object.
- Consistency: equal objects must have equal hash codes.
- Immutability: once you insert a key, don’t change it a way that would change its hashCode or equals.
  - immutable in Java: String, Integer, BigInteger
  - mutable in Java: Date

"Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key."

Javadoc for Map interface

Phone numbers: (609) 867-5309

```
public class PhoneNumber {
    private int area;    // area code (3 digits)
    private int exch;   // exchange (3 digits)
    private int ext;    // extension (4 digits)

    // constructor, toString, but no mutators
    public boolean equals(Object x) {
        PhoneNumber a = this;
        PhoneNumber b = (PhoneNumber) x;
        return (a.area == b.area) &&
               (a.exch == b.exch) && (a.ext == b.ext);
    }

    public int hashCode() {
        return 10007 * (area + 1009 * exch) + ext;
    }
}
```
Frequency Symbol Table

**Frequency symbol table.**
- `fst.hit(key)` increment frequency count of given key.
- `fst.freq(key)` returns number of times given key occurs.

**Applications.**
- Web traffic analyzer: look up host to find number of hits.
- Browser: highlight visited links in purple.
- Chess: detect a repetition draw.
- Bayesian spam filter.

**Implementation.** Simple extension of a symbol table.

```java
FrequencyTable fst = new FrequencyTable();
while (!StdIn.isEmpty()) {
    String key = StdIn.readString();
    fst.hit(key);
    System.out.println(fst.freq(key));
}
```

```java
public class FrequencyTable {
    HashMap st = new HashMap();
    private class Entry {
        String name;
        int freq;
    }

    public void hit(String key) {
        Entry entry = (Entry) st.get(key);
        if (entry == null) {
            entry = new Entry();
            entry.name = key;
            st.put(key, entry);
        }
        entry.freq++;
    }

    public int freq(String key) {
        Entry entry = (Entry) st.get(key);
        if (entry == null) return 0;
        else return entry.freq;
    }
}
```

**A Plan for Spam**

**Bayesian spam filter.**
- Filter based on analysis of previous messages.
- User trains the filter by classifying messages as spam or ham.
- Parse messages into tokens (alphanumeric, dashes, ',', $)

**Build data structures.**
- Hash table A of tokens and frequencies for spam.
- Hash table B of tokens and frequencies for ham.
- Hash table C of tokens with probability p that they appear in spam.

```java
double h = 2.0 * ham.freq(word);
double s = 1.0 * spam.freq(word);
double p = (s/spams) / (h/hams + s/spams);
```

**Reference:** [http://www.paulgraham.com/spam.html](http://www.paulgraham.com/spam.html)

**A Plan for Spam**

**Identify incoming email as spam or ham.**
- Find 15 most interesting tokens (difference from 0.5).
- Combine probabilities using Bayes law. *(which data structure?)*

\[
P_s = \frac{P_s \times P_2 \times \cdots \times P_{15}}{(P_s \times P_2 \times \cdots \times P_{15}) + ((1-P_s) \times (1-P_2) \times \cdots \times (1-P_{15}))}
\]

- Declare as spam if threshold > 0.9.

**Details.**
- Words you’ve never seen.
- Words that appear in ham corpus but not spam corpus, vice versa.
- Words that appear less than 5 times in spam and ham corpuses.
- Update data structures.
Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?
- Yes, in obvious situations – aircraft control, nuclear reactors.
- Yes, sometimes in surprising situations.

Hashing-based denial-of-service attacks.
- If malicious adversary can choose what strings to insert into your hash table, you might be in big trouble.

Crosby-Wallach exploits of real systems.
- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem
- Perl 5.8.0: insert carefully chosen strings into associative array
- Linux 2.4.20 kernel: save files with carefully chosen names.

Reference: http://www.cs.rice.edu/~scrosby/hash/

How easy is it to break Java’s `hashCode` with `String` keys?
- Almost trivial: String hash function is part of language spec.
- Java’s string `hashCode`: hash of "BB" = hash of "Aa" = 2112.
- Can now create $2^N$ strings of length 2N that all hash to same value!

Possible to fix?
- Security by obscurity.
- Cryptographically secure hash functions.
- Universal hashing.