4.2 Hashing

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 with sequential search.
- Limitations on both time and space: hashing (the real world).

Choosing a Good Hash Function

Idealistic goal: scramble the keys uniformly.
- Efficiently computable.
- Each table position equally likely for each key.

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

Ex: date of birth.
- Bad: birth year.
- Better: birthday.

Ex: phone numbers.
- Bad: first three digits.
- Better: last three digits.

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.
Hash Codes and Hash Functions

Hash code. A 32-bit int (between $-2^{147483648}$ and $2^{147483647}$).

Hash function. An int between 0 and $M-1$.

Designing a Good Hash Code

Java 1.5 string library.

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

Equivalent to $h = 31L^{-1} s_0 + ... + 31^2 s_{L-3} + 31 s_{L-2} + s_{L-1}$.

Horner’s method to hash string of length $L$: $O(L)$.

Ex. String $s = \text{"call"}$;
    int code = s.hashCode();

```
3045982 = 99·31^3 + 97·31^2 + 108·31^1 + 108·31^0
```

Implementing Hash Code in Java

API for `hashCode()`.

- Return an int.
- If `x.equals(y)` then $x$ and $y$ must have the same hash code.
- Repeated calls to `x.hashCode()` must return the same value.

```
```

Default implementation. Memory address of $x$.

Customized implementations. String, URL, Integer, Date.

User-defined implementations. Tricky to get right, black art.

Designing a Bad Hash Code

Java 1.1 string library.

- For long strings: only examines 8-9 evenly spaced characters.
- Saves time in performing arithmetic...

```java
public int hashCode() {
    int hash = 0;
    int skip = Math.max(1, length() / 8);
    for (int i = 0; i < length(); i += skip)
        hash = (37 * hash) + s[i];
    return hash;
}
```

But great potential for bad collision patterns.
Implementing Hash Code: US Phone Numbers

**Phone numbers:** (609) 867-5309.

area code exchange extension

```
public final class PhoneNumber {
    private final int area, exch, ext;
    public PhoneNumber(int area, int exch, int ext) {
        this.area = area;
        this.exch = exch;
        this.ext = ext;
    }
    public boolean equals(Object y) { // as before }
    public int hashCode() {
        return 10007 * (area + 1009 * exch) + ext;
    }
}
```

Collisions

**Collision.** Two distinct keys hashing to same index.

**Conclusion.** Birthday problem $\Rightarrow$ can’t avoid collisions unless you have a ridiculous amount of memory.

**Challenge.** Deal with collisions efficiently.

Bins and Balls

**Bins and balls.** Throw balls uniformly at random into $M$ bins.

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

**Birthday problem.** Expect two balls in the same bin after $\sqrt{\frac{1}{2} M}$ tosses.

**Coupon collector.** Expect every bin has $\approx 1$ ball after $O(M \log M)$ tosses.

**Load balancing.** After tossing $M$ balls, expect most loaded bin has $O(\log M / \log \log M)$ balls.

Collision Resolution: Two Approaches

**Separate chaining.** [H. P. Luhn, IBM 1953]

Put keys that collide in a list associated with index.

**Open addressing.** [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.
Separate Chaining

Separate Chaining: Java Implementation

```java
public class ListHashST<Key, Value> {
    private int M = 8191;
    private Node[] st = new Node[M];

    private static class Node {
        Object key;  // no generic array creation in Java
        Object val;  
        Node next;
        Node(Object key, Object val, Node next) {
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }
}
```

Separate Chaining: Java Implementation (cont)

```java
public void put(Key key, Val val) {
    int i = hash(key);
    for (Node x = st[i]; x != null; x = x.next) {
        if (key.equals(x.key)) {
            x.val = val;  // check if key already present
            return;
        }
    }
    // insert at front of chain
    st[i] = new Node(k, val, st[i]);
}
```

```java
public Val get(Key key) {
    int i = hash(key);
    for (Node x = st[i]; x != null; x = x.next)
        if (key.equals(x.key)) {
            return (Val) x.val;
        }
    return null;
}
```
Separate Chaining Performance

Cost is proportional to length of chain.
Average length = N / M.
Worst case: all keys hash to same chain.

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

depends on hash map being random map.

Parameters.
M too large ⇒ too many empty chains.
M too small ⇒ chains too long.
Typical choice: α = N / M = 10 ⇒ constant-time ops.

Advantages. Fast insertion, fast search.
Disadvantage. Hash table has fixed size, assumes good hash function.
fix: use repeated doubling, and rehash all keys

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th></th>
<th></th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Get</td>
<td>Put</td>
<td>Remove</td>
<td>Get</td>
</tr>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>I*</td>
</tr>
</tbody>
</table>

* assumes hash function is random

Linear Probing

Linear probing: array of size M.
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.
Linear Probing: Java Implementation

```java
public class ArrayHashST<Key, Val> {
    private int M = 30001;
    private Key[] keys = (Key[]) new Object[M];
    private Val[] vals = (Val[]) new Object[M];

    public void put(Key key, Val val) {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key)) break;
        keys[i] = key;
        vals[i] = val;
    }

    public Val get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key)) return vals[i];
        return null;
    }
}
```

Linear Probing Performance

- **Insert and search cost** depend on length of cluster.
- **Average length of cluster**: \( \alpha = \frac{N}{M} \), but keys more likely to hash to big clusters.
- **Worst case**: all keys hash to same cluster.

**Theorem.** [Knuth 1962] Let \( \alpha = \frac{N}{M} < 1 \) be the load factor.

\[
\text{insert / search miss} = \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \\
\text{search hit} = \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right)
\]

(assumes hash function is random)

**Parameters.**
- \( M \) too large \( \Rightarrow \) too many empty array entries.
- \( M \) too small \( \Rightarrow \) clusters coalesce.
- Typical choice: \( M = 2N \Rightarrow \) constant-time ops.

Clustering

- **Cluster.** A contiguous block of items.
- **Observation.** New keys likely to hash into middle of big clusters.

```
- - - S H A C E - - - X M I - - - P - - R L - - -
```

- **Knuth’s parking problem.** Cars arrive at one-way street with \( M \) parking spaces. Each desires a random space \( i \): if space \( i \) is taken, try \( i+1, i+2, \ldots \)
- What is mean displacement of a car?

- **Empty.** With \( M/2 \) cars, mean displacement is \( \approx 3/2 \).
- **Full.** With \( M \) cars, mean displacement is \( \approx \sqrt{\frac{3}{2} \pi M} \).

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Get</th>
<th>Put</th>
<th>Remove</th>
<th>Get</th>
<th>Put</th>
<th>Remove</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>( N )</td>
<td>( N )</td>
<td>( N/2 )</td>
<td>( N )</td>
<td>( N/2 )</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( 1^* )</td>
<td>( 1^* )</td>
<td>( 1^* )</td>
</tr>
<tr>
<td>Linear probing</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( 1^* )</td>
<td>( 1^* )</td>
<td>( 1^* )</td>
</tr>
</tbody>
</table>

(\( ^* \) assumes hash function is random)

**Advantages.** Fast insertion, fast search.

**Disadvantage.** Hash table has fixed size, assumes good hash function. Fix: use repeated doubling, and rehash all keys.
Double Hashing

Idea. 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 k.

Ex: k = 1 + (v mod 97).

Effect. Skip values give different search paths for keys that collide.

Best practices. Make k and M relatively prime.

HashCode

Theorem. [Guibas-Szemerédi] Let α = N / M < 1 be average length of list.

Parameters. Typical choice: M = 2N ⇒ constant-time ops.

Disadvantage. Delete cumbersome to implement.

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 α</th>
<th>50%</th>
<th>66%</th>
<th>75%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>linear probing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>15</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
<tr>
<td>put</td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
<td>55.5</td>
</tr>
<tr>
<td><strong>double hashing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>put</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Hashing Tradeoffs

Odds and Ends
Hashing: Java Library

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

Duplicate policy.
- Java `HashMap` allows null values.
- Our implementation forbids null values.

Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?
- Obvious situations: aircraft control, nuclear reactor, pacemaker.
- Surprising situations: denial-of-service attacks.

Real-world exploits. [Crosby-Wallach 2003]
- 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/hack

Symbol Table: Using HashMap

Goal. Find strings with the same hash code.
Solution. The base-31 hash code is part of Java's string API.

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aa</td>
<td>2112</td>
</tr>
<tr>
<td>BB</td>
<td>2112</td>
</tr>
</tbody>
</table>

Symbol table. Implement our API using `java.util.HashMap`.

```java
import java.util.HashMap;
import java.util.Iterator;

public class ST<Key, Val> implements Iterable<Key> {
    private HashMap<Key, Val> st = new HashMap<Key, Val>();
    public void put(Key key, Val val) {
        if (val == null) st.remove(key);
        else st.put(key, val);
    }
    public Val get(Key key) { return st.get(key); }
    public Val remove(Key key) { return st.remove(key); }
    public boolean contains(Key key) { return st.containsKey(key); }
    public int size() { return st.size(); }
    public Iterator<Key> iterator() { return st.keySet().iterator(); }
}
```
One-Way Hash Functions

**One-way hash function.** Hard to find a key that will hash to a desired value, or to find two keys that hash to same value.

**Ex.** MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160.

```java
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);
// prints bytes as hex string
```

**Applications.** Digital fingerprint, message digest, storing passwords.