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

Choosing a Good Hash Function

Goal: scramble the keys.
- 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.
Hash Function: String Keys

Java string library hash functions.

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

• Equivalent to \( h = 31^{L-1}s_0 + \ldots + 31^2s_{L-3} + 31s_{L-2} + s_{L-1} \).
• Horner’s method to hash string of length \( L \): \( O(L) \).

Q. Can we reliably use \( (h \mod M) \) as index for table of size \( M \)?
A. No. Instead, use \( (h \& 0xFFFFFFFF) \mod M \).

Implementing HashCode: US Phone Numbers

Phone numbers: (609) 867-5309.

```java
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 keys hashing to same value.
• Essentially unavoidable.
• Birthday problem: how many people will have to enter a room until two have the same birthday? 23
• With \( M \) hash values, expect a collision after \( \sqrt{\frac{M}{\pi}} \) insertions.

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

Challenge: efficiently cope with collisions.
Collision Resolution: Two Approaches.

Separate Chaining.

- \( M \) much smaller than \( N \).
- \( N / M \) keys per table position.
- Put keys that collide in a list.
- Need to search lists.

- Symbol Table: Separate Chaining

\[
\begin{array}{l}
\text{st[0]} \text{ jocularly --- seriously} \\
\text{st[1]} \text{ listen} \\
\text{st[2]} \text{ null} \\
\text{st[3]} \text{ suburban --- untravelled --- considering} \\
\vdots \\
\text{st[8190]} \text{ browsing}
\end{array}
\]

\( M = 8191, N = 15000 \)

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.

- Symbol Table: Separate Chaining Implementation (cont)

\[
\begin{array}{l}
\text{st[0]} \text{ seriously --- seriously} \\
\text{st[1]} \text{ listen} \\
\text{st[2]} \text{ null} \\
\text{st[3]} \text{ suburban --- untravelled --- considering} \\
\vdots \\
\text{st[8190]} \text{ browsing}
\end{array}
\]

\( M = 8191 \)

- Hash: map key to integer \( i \) between 0 and \( M-1 \).
- Insert: put at front of \( i \)th chain (if not already there).
- Search: only need to search \( i \)th chain.
- Running time: proportional to length of chain.
Separate Chaining Performance

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

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

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.

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

Symbol Table: Linear Probing Implementation

public class ArrayHashST<Key, Val> {  
  private int M = 30001;  
  private Key[] keys = (Key[]) new Object[M];  
  private Val[] vals = (Val[]) new Object[M];  
  
  private int hash(Key key) { // as before }  
  
  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) {  
    int i;  
    for (i = hash(key); keys[i] != null; i = (i+1) % M)  
      if (keys[i].equals(key)) break;  
    return vals[i];  
  }  
}

Worst Case

<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>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>
</tbody>
</table>

* assumes hash function is random

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

• fix: use repeated doubling, and rehash all keys
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 $k$.

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

**Result.** Skip values give different search paths for keys that collide.

**Best practices.** Make $k$ and $M$ relatively prime.

---

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

* assumes hash function is random

**Advantages:** fast insertion, fast search.

**Disadvantage:** hash table has fixed size.

---

Double Hashing Performance

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

**Theorem.** [Knuth 1962] Let $\alpha = N / M < 1$ be average length of list.

- Insert: $\frac{1}{2} (1 - \frac{1}{1-\alpha})$ depends on hash map being random map
- Search: $\frac{1}{2} (1 + \frac{1}{1-\alpha})$

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

---

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

**Theorem.** [Guibas-Szemeredi] Let $\alpha = N / M < 1$ be average length of list.

- Insert: $\frac{1}{\alpha} \ln(1+\alpha)$ depends on hash map being random map
- 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: $M = 2N \Rightarrow$ 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></th>
<th>Linear probing</th>
<th>Double hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load factor α</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>66%</td>
<td>2.0</td>
<td>1.6</td>
</tr>
<tr>
<td>75%</td>
<td>3.0</td>
<td>1.8</td>
</tr>
<tr>
<td>90%</td>
<td>5.5</td>
<td>2.6</td>
</tr>
</tbody>
</table>

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

**Duplicate policy.**
- Java HashMap allows null values.
- Our implementations forbid null values.

Symbol Table: Using HashMap

**Symbol table.** Implement our interface using HashMap.

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

public class ST<Key, Value> implements Iterable<Key> {  
    private HashMap<Key, Value> st = new HashMap<Key, Value>();

    public void put(Key key, Value val) {  
        if (val == null) st.remove(key);
        else st.put(key, val);
    }

    public Value get(Key key) { return st.get(key); }
    public Value 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(); }
}
```

Java 1.1 string library hash function.
- For long strings: only examines 8 evenly spaced characters.
- Saves time in performing arithmetic.
- Great potential for bad collision patterns.

```java
public int hashCode() {  
    int hash = 0;
    if (length() <= 16) {  
        for (int i = 0; i < length(); i++)
            hash = (37 * hash) + s[i];
    }  
    else {  
        int skip = length() / 8;
        for (int i = 0; i < length(); i += skip)
            hash = (37 * hash) + s[i];
    }
    return hash;
}
```

```
String.java
```
Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?
- Obvious situations: aircraft control, nuclear reactors.
- 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.


Q. How easy is it to break Java’s `hashCode` with String keys?
A. Almost trivial: `string` `hashCode` is part of Java 1.5 API.
- Ex: `hashCode` of "BB" equals `hashCode` of "Aa".
- Can now create $2^N$ strings of length $2N$ that all hash to same value!

Possible to fix?
- Security by obscurity. [not recommended]
- Cryptographically secure hash functions.
- Universal hashing.