3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Premature optimization

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

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

Yet we should not pass up our opportunities in that critical 3%.

Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

Q. Can we do better?
A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.

Issues.
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.
- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Computing the hash function

**Idealistic goal.** Scramble the keys uniformly to produce a table index.
- Efficiently computable.
- Each table index equally likely for each key.

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

Practical challenge. Need different approach for each key type.

Java’s hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

**Requirement.** If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

**Highly desirable.** If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.

Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return 17.

Customized implementations. Integer, Double, String, File, URL, Date,...

User-defined types. Users are on their own.

Hash tables: quiz 1

Which of the following would be a good hash function for U.S. phone numbers to integers between 0 and 999?

A. First three digits.
B. Second three digits.
C. Last three digits.
D. Either B or C.
E. I don’t know.
Implementing hash code: integers, booleans, and doubles

Java library implementations

```java
public final class Integer {
    private final int value;
    ...
    public int hashCode() {
        return value;
    }
}

public final class Double {
    private final double value;
    ...
    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}

public final class Boolean {
    private final boolean value;
    ...
    public int hashCode() {
        if (value) return 1231;
        else return 1237;
    }
}
```

Warning: 0.0 and +0.0 have different hash codes

Implementing hash code: strings

Performance optimization.
- Cache the hash value in an instance variable.
- Return cached value.

```java
public final class String {
    private int hash = 0;
    private final char[] s;
    ...
    public int hashCode() {
        int h = hash;
        if (h == 0) return h;
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        return h;
    }
}
```

Horner’s method: only \( L \) multiplies/adds to hash string of length \( L \).

String \( s = \text{"call";} \)
\( s\.hashCode(); \quad 3045982 = 99 \cdot 31^1 + 79 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \)
\( = 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \)

Implementing hash code: user-defined types

```java
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double amount;
    ...
    public Transaction(String who, Date when, double amount) {
        /* as before */
    }
    ...
    public boolean equals(Object y) {
        /* as before */
    }
    public int hashCode() {
        int hash = 17;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

nonzero constant
for reference types, use hashCode()
for primitive types, use hashCode() of wrapper type

Q. What if hashCode() of string is 0?  
hashCode() of "pollinating sandboxes" is 0
Hash code design

"Standard" recipe for user-defined types.
- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, use 0.
- If field is a reference type, use `hashCode()`.
- If field is an array, apply to each entry.

In practice. Recipe above works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" family of hash functions exist.

Basic rule. Need to use the whole key to compute hash code;
consult an expert for state-of-the-art hash codes.

---

Modular hashing

**Hash code.** An int between $-2^{31}$ and $2^{31} - 1$.

**Hash function.** An int between 0 and M - 1 (for use as array index).

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

Typically a prime or power of 2

```
private int hash(Key key) {
    return Math.abs(key.hashCode()) % M;
}
```

1-in-a-billion bug

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

Correct

```
x
```

```
x.hashCode()
```

```
x % M
```

```
hash(x)
```

---

Uniform hashing assumption

**Uniform hashing assumption.** Each key is equally likely to hash to an integer between 0 and M - 1.

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

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

Coupon collector. Expect every bin has $\geq 1$ ball after $\sim M \ln M$ tosses.

Load balancing. After $M$ tosses, expect most loaded bin has $\sim \ln M / \ln \ln M$ balls.
Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and \( M - 1 \).

Bins and balls. Throw balls uniformly at random into \( M \) bins.

Collisions

Collision. Two distinct keys hashing to same index.
- Birthday problem \( \Rightarrow \) can't avoid collisions.
- Coupon collector \( \Rightarrow \) not too much wasted space.
- Load balancing \( \Rightarrow \) no index gets too many collisions.

Challenge. Deal with collisions efficiently.

Separate-chaining symbol table

Use an array of \( M < N \) linked lists. [H. P. Luhn, IBM 1953]
- Hash: map key to integer \( i \) between 0 and \( M - 1 \).
- Insert: put at front of \( i \)th chain (if not already in chain).
- Search: sequential search in \( i \)th chain.

Separate–chaining hash table (\( M = 4 \))
Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer $i$ between 0 and $M - 1$.
- Insert: put at front of $i$th chain (if not already in chain).
- Search: sequential search in $i$th chain.

Separate-chaining hash table ($M = 4$)

```plaintext
separate-chaining hash table (M = 4)

<table>
<thead>
<tr>
<th>i</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>K</td>
</tr>
<tr>
<td>1</td>
<td>J</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
</tbody>
</table>
```

Separate-chaining symbol table: Java implementation

```java
public class SeparateChainingHashST<Key, Value> {
    private int M = 97; // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

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

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }
}
```

Analysis of separate chaining

**Proposition.** Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of $N/M$ is extremely close to 1.

**Pf sketch.** Distribution of list size obeys a binomial distribution.

**Consequence.** Number of probes for search/insert is proportional to $N/M$.

- $M$ too large $\Rightarrow$ too many empty chains.
- $M$ too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim \frac{1}{4} N \Rightarrow$ constant-time ops.
Resizing in a separate-chaining hash table

Goal. Average length of list $N/M = \text{constant}$.
- Double size of array $M$ when $N/M \geq 8$;
  - halve size of array $M$ when $N/M \leq 2$.
- Note: need to rehash all keys when resizing.

before resizing ($N/M = 8$)

after resizing ($N/M = 4$)

Deletion in a separate-chaining hash table

Q. How to delete a key (and its associated value)?
A. Easy: need to consider only chain containing key.

before deleting C

after deleting C

Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$1^*$</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot, and put it there.

Linear-probing hash table (M = 16, N = 10)

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>H</th>
<th>L</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>vals[]</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

put(K, 14)
hash(K) = 7
14

Linear-probing symbol table: Java implementation

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

    private int hash(Key key) { /* as before */ }
    private void put(Key key, Value val) { /* next slide */ }

    public Value 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 hash table summary

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.
Search. Search table index $i$; if occupied but no match, try $i + 1, i + 2$, etc.

Note. Array size $M$ must be greater than number of key-value pairs $N$. 

Linear-probing symbol table: Java implementation

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

    private int hash(Key key) { /* as before */ }
    private void put(Key key, Value val) { /* prev slide */ }

    public void put(Key key, Value 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;
    }
}
```
Clustering

**Cluster.** A contiguous block of items.

**Observation.** New keys likely to hash into middle of big clusters.

![Cluster Diagram]

Knuth’s parking problem

**Model.** Cars arrive at one-way street with $M$ parking spaces. Each desires a random space $i$: if space $i$ is taken, try $i+1, i+2, etc.$

![Parking Diagram]

**Q.** What is mean displacement of a car?

**Half-full.** With $M/2$ cars, mean displacement is $\sim 5/2$.

**Full.** With $M$ cars, mean displacement is $\sim \sqrt{\pi} M/8$.

**Key insight.** Cannot afford to let linear-probing hash table get too full.

Analysis of linear probing

**Proposition.** Under uniform hashing assumption, the average # of probes in a linear probing hash table of size $M$ that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$$

**Pf.**

![Proof Diagram]

**Parameters.**

- $M$ too large $\Rightarrow$ too many empty array entries.
- $M$ too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N/M \sim \frac{1}{2}$.

Resizing in a linear-probing hash table

**Goal.** Average length of list $N/M \leq \frac{1}{2}$.

- Double size of array $M$ when $N/M \geq \frac{1}{2}$.
- Halve size of array $M$ when $N/M \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

**before resizing**

<table>
<thead>
<tr>
<th>keys[]</th>
<th>E</th>
<th>S</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**after resizing**

<table>
<thead>
<tr>
<th>keys[]</th>
<th>A</th>
<th>S</th>
<th>E</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>vals[]</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

# probes for search hit is about 3/2
# probes for search miss is about 5/2
Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?
A. Requires some care: can’t just delete array entries.

| keys[] | P M A C S H L E R X |
| vals[] | 10 9 8 4 0 5 11 12 3 7 |

Deletion in a linear-probing hash table

<table>
<thead>
<tr>
<th>after deleting S</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
</tr>
<tr>
<td>vals[]</td>
</tr>
</tbody>
</table>

3-SUM (REVISITED)

3-SUM. Given $N$ distinct integers, find three such that $a + b + c = 0$.

Goal. $N^2$ expected time case, $N$ extra space.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>red–black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1 *</td>
</tr>
<tr>
<td>linear probing</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1 *</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?
A. Obvious situations: aircraft control, nuclear reactor, pacemaker, HFT, ...
A. 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.

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hashCode().
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>
<tr>
<td>“AaAa”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AaAaBB”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AaBBAA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AaBBBB”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AaBBBBA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AaBBBBBB”</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

2^m strings of length 2N that hash to same value!

Diversion: one-way hash functions

One-way hash function. “Hard” to find a key that will hash to a desired value (or two keys that hash to same value).

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

known to be insecure

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

Applications. Crypto, message digests, passwords, Bitcoin, ....
Caveat. Too expensive for use in ST implementations.
Separate chaining vs. linear probing

Separate chaining.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.
- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\sim \lg \ln N$.

Double hashing. [linear-probing variant]
- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]
- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.

Hash tables vs. balanced search trees

Hash tables.
- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for String (e.g., cached hash code).

Balanced search trees.
- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement compareTo() correctly than equals() and hashCode().

Java system includes both.
- Red-black BSTs: java.util.TreeMap, java.util.TreeSet.