3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

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>( \frac{1}{2} N )</td>
</tr>
<tr>
<td>unordered list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg 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>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>red–black BST</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.0 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(&quot;it&quot;) = 3</td>
<td>hash(&quot;it&quot;) = 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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).
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 1. Phone numbers.
- Bad: first three digits.
- Better: last three digits.

Ex 2. 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()).

Implementing hash code: integers, booleans, and doubles

Java library implementations

Java’s library implementation

Implementing hash code: strings

Java library implementation
Implementing hash code: strings

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

```java
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 = 31 * h + s[i];
        return h;
    }
}
```

Q. What if `hashCode()` of string is 0?

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;
    }
}
```

**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;
}
```

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

1-in-a-billion bug
```
private int hash(Key key) {
    return (key.hashCode() & 0x7fffffff) % M;
}
```

**Correct.**

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

**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 \( \Theta \left( \log M / \log \log M \right) \) balls.

---

**3.4 HASH TABLES**

- hash functions
- separate chaining
- linear probing
- context

---

**Collisions**

- **Collision.** Two distinct keys hashing to same index.
  - Birthday problem \( \Rightarrow \) can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
  - Coupon collector + load balancing \( \Rightarrow \) collisions are evenly distributed.

**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 there).
- Search: need to search only $i$th chain.

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: $M \sim N / 4$ $\Rightarrow$ 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 $N/M \leq 2$.
- Need to rehash all keys when resizing.

```plaintext
x.hashCode() does not change
but hash(x) can change
```

Deletion in a separate-chaining hash table

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

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>$\frac{1}{2}N$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\lg N$</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>$1.39 \lg N$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$2 \lg N$</td>
<td>$2 \lg N$</td>
<td>$2 \lg N$</td>
<td>$1.0 \lg N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$3.5$ *</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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]
When a new key collides, find next empty slot, and put it there.

```
<table>
<thead>
<tr>
<th>i</th>
<th>null</th>
<th>jocularly</th>
</tr>
</thead>
<tbody>
<tr>
<td>st[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[2]</td>
<td></td>
<td>listen</td>
</tr>
<tr>
<td>st[3]</td>
<td></td>
<td>suburban</td>
</tr>
</tbody>
</table>

... (remaining slots filled with data) ...

linear probing (M = 30001, N = 15000)
```

Linear-probing hash table demo

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.

```
<table>
<thead>
<tr>
<th>i</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>st[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[3]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ... (remaining slots filled with data) ...

M = 16
```

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 LinearProbingHashST<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;
    }
}
```

Clustering

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

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

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 $i+1, i+2, \text{etc.}$

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

**Half-full.** With $M/2$ cars, mean displacement is $\sim 3/2$.

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

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

for a search hit, and

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

for a search miss / insert.

**Pf.**

Parameters.

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

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.

before deleting $S$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>E</td>
<td>S</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dafter deleting $S$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $N/M \leq \frac{1}{2}$.
- Need to rehash all keys when resizing.

before resizing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>E</td>
<td>S</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dafter resizing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\frac{1}{2}N$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$\frac{1}{2}N$</td>
<td>$\frac{1}{2}N$</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>$1.39 \lg N$</td>
<td>$1.39 \lg N$</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$2 \lg N$</td>
<td>$2 \lg N$</td>
<td>$2 \lg N$</td>
<td>$1.0 \lg N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$3-5$</td>
<td>$3-5$</td>
</tr>
<tr>
<td>linear probing</td>
<td>$N$</td>
<td>$N$</td>
<td>$3-5$</td>
<td>$3-5$</td>
</tr>
</tbody>
</table>

$^*$ under uniform hashing assumption
### 3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

#### War story: algorithmic complexity attacks

**A Java bug report.**


Julian Wälde and Alexander Klink reported that the `String.hashCode()` hash function is not sufficiently collision resistant. `hashCode()` value is used in the implementations of `HashMap` and `Hashtable` classes:

http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html

A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of `HashMap` or `Hashtable` by changing hash table operations complexity from an expected/average $O(1)$ to the worst case $O(n)$. Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques.

This problem can be used to start a denial of service attack against Java applications that use untrusted inputs as `HashMap` or `Hashtable` keys. An example of such application is web application server (such as tomcat, see [Apache Tomcat](http://www.apache.org/provisional-examples/security.html)) that may fill hash tables with data from HTTP request (such as GET or POST parameters). A remote attack could use that to make JVM use excessive amount of CPU time by sending a HTTP POST request with large amount of parameters which hash to the same value.

This problem is similar to the issue that was previously reported for and fixed in e.g. perl:


---

**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 hash code.

**Solution.** The base-31 hash code is part of Java’s string API.

<table>
<thead>
<tr>
<th>key</th>
<th><code>hashCode()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aa&quot;</td>
<td>2112</td>
</tr>
<tr>
<td>&quot;BB&quot;</td>
<td>2112</td>
</tr>
</tbody>
</table>

- 2$^N$ 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

Applications. Digital fingerprint, message digest, storing passwords.
Caveat. Too expensive for use in ST implementations.

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 $\log \log N$.

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.

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.

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 strings (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.