3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Yet we should not pass up our opportunities in that critical 3%.

“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.

## Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th></th>
<th>average case</th>
<th></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>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</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>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>hashing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
<td>$1^\dagger$</td>
<td>$1^\dagger$</td>
</tr>
</tbody>
</table>

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

---

† under suitable technical assumptions
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.

```
hash("it") = 3
3  "it"
```

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).
Equality test

All Java classes inherit a method `equals()`. 

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

**Default implementation.** `(x == y)`

**Customized implementations.** Integer, Double, String, java.net.URL, ...

**User-defined implementations.** Some care needed.
Implementing equals for user-defined types

Seems easy.

```java
public class Date {
    private final int month;
    private final int day;
    private final int year;

    public boolean equals(Date that) {
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Date {
    private final int month;
    private final int day;
    private final int year;
    
    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass()) return false;

        Date that = (Date) y;
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

- Typically unsafe to use `equals()` with inheritance (would violate symmetry)
- Must be `Object`
- Optimization (for reference equality)
- Check for `null`
- Objects must be in the same class (religion: `getClass()` vs. `instanceof`)
- Cast is now guaranteed to succeed
- Check that all significant fields are the same
Equals design

“Standard” recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type; cast.
- Compare each significant field:
  - if field is a primitive type, use ==
  - if field is an object, use equals() and apply rule recursively
  - if field is an array of primitives, use Arrays.equals()
  - if field is an array of objects, use Arrays.deepEquals()

Best practices.

- Do not use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make compareTo() consistent with equals().

\[ x.equals(y) \text{ if and only if } (x.compareTo(y) == 0) \]
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.
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. Last 4 digits of Social Security number.
Ex 2. Last 4 digits of phone number.

**Practical challenge.** Need different approach for each key type.
Hash tables: quiz 1

Which is the last digit of your day of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
Which is the last digit of your year of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
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())`.

![Diagram](image)

**Default implementation.** Memory address of `x`.

**Legal (but poor) implementation.** Always return 17.

**Customized implementations.** Integer, Double, String, java.net.URL, ...

**User-defined types.** Users are on their own.
Implementing hash code: integers, booleans, and doubles

Java library implementations

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

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

convert to IEEE 64-bit representation; xor most significant 32-bits with least significant 32-bits

Warning: -0.0 and +0.0 have different hash codes
Implementing hash code: arrays

31x + y rule.
- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add next integer in array.

```java
public class Arrays {
    ...

    public static int hashCode(int[] a) {
        if (a == null)
            return 0;  // special case for null

        int hash = 1;
        for (int i = 0; i < a.length; i++)
            hash = 31*hash + a[i];
        return hash;
    }
}
```
Implementing hash code: user-defined types

public final class 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 = 1;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}
Implementing hash code: user-defined types

```java
public final class 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() {
        return Objects.hash(who, when, amount);
    }
}
```
Hash code design

“Standard” recipe for user-defined types.
- Combine each significant field using the $31x + y$ rule.
- Shortcut 1: use `Objects.hashCode()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.

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.
Hash tables: quiz 3

Which code maps hashable keys to integers between 0 and \( m-1 \)?

A. 
```java
private int hash(Key key)
{   return key.hashCode() % m;
}
```

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

C. Both A and B.

D. Neither A nor B.
Module 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).

typically a prime or power of 2

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

bug

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

1-in-a-billion bug

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

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

if $m$ is a power of 2, can use key.hashCode() & (m-1)
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.

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.

Ex. String data type.

hash value frequencies for words in Tale of Two Cities (\( m = 97 \))
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.
- Coupon collector $\Rightarrow$ not too much wasted space.
- Load balancing $\Rightarrow$ no index gets too many collisions.

**Challenge.** Deal with collisions efficiently.

unless you have a ridiculous (quadratic) amount of memory
Separate-chaining symbol table

Use an array of \( m \) 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 (\( m = 4 \))

```
put(L, 11)
hash(L) = 3
```

![Diagram of separate-chaining hash table with keys and values]
Separate-chaining symbol table

Use an array of $m$ 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)

get(E)
hash(E) = 1
public class SeparateChainingHashST<Key, Value> {
    private int m = 128; // 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 class SeparateChainingHashST<Key, Value> {
    private int m = 128; // 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 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 \(i^{th}\) chain is within a constant factor of \(n / m\) is extremely close to 1.

**Pf sketch.** Distribution of chain sizes obeys a binomial distribution.

![Binomial distribution graph](image)

\[\text{Binomial distribution } (n = 10^4, m = 10^3, \alpha = 10)\]

**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 per operation.

\(m\) times faster than sequential search
Resizing in a separate-chaining hash table

**Goal.** Average length of list \( n/m = \text{constant}. \)

- Double length \( m \) of array when \( n/m \geq 8 \);
  - halve length \( m \) of array 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.
## 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 (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>
</tbody>
</table>

† under uniform hashing assumption
3.4 Hash Tables

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

**Open addressing.** [Amdahh–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>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></td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</td>
</tr>
</tbody>
</table>

**put(K, 14)**

**hash(K) = 7**

14

| vals[]   |   11 |   10 |   9 |   5 |   6 |   12 |   13 |     |     |     |     |     |     |  4 |   8 |
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 length $m$ must be greater than number of key–value pairs $n$.

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>L</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

$m = 16$
Linear-probing symbol table: Java implementation

```java
public class LinearProbingHashST<Key, Value> {
    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    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 (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

array resizing code omitted
public class LinearProbingHashST<Key, Value> {

    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    private Value get(Key key) { /* 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;
    }
}

Linear-probing symbol table: Java implementation
Cluster. A contiguous block of items.
Observation. New keys likely to hash into middle of big clusters.
Knuth’s parking problem

Model. Consider $n$ cars arriving 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. If $n = m / 2$ cars, mean displacement is $\sim 1 / 2$.

Full. If $n = m$ parking spaces, mean displacement is $\sim \sqrt{\pi n / 8}$.

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 at most

\[
\begin{align*}
\text{search hit} & : \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \\ 
\text{search miss / insert} & : \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)
\end{align*}
\]

**Pf.**

**Parameters.**

- $m$ too large $\Rightarrow$ too many empty array entries.
- $m$ too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = n/m \sim \frac{1}{2}$.  
  # probes for search hit is about $3/2$
  # probes for search miss is about $5/2$
Resizing in a linear-probing hash table

**Goal.** Average length of list \( n / m \leq \frac{1}{2} \).
- Double length of array \( m \) when \( n / m \geq \frac{1}{2} \).
- Halve length of array \( m \) when \( n / m \leq \frac{1}{8} \).
- Need to rehash all keys when resizing.

### before resizing

<table>
<thead>
<tr>
<th></th>
<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></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></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### after resizing

<table>
<thead>
<tr>
<th></th>
<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>A</td>
<td>S</td>
<td></td>
<td>E</td>
<td></td>
<td>R</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></td>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deletion in a linear-probing hash table

**Q.** How to delete a key (and its associated value)?

**A.** Requires some care: can’t simply delete array entries.

---

**before deleting S**

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>L</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>vals[]</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**after deleting S?**

<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>10</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

doesn't work, e.g., if hash(H) = 4
# 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></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>$\log n$</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>
<tr>
<td>linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>1†</td>
</tr>
</tbody>
</table>

† under uniform hashing assumption
**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.
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
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]
- Linux 2.4.20 kernel: save files with carefully chosen names.
- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
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
http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.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 [bug #750521](https://issues.redhat.com/browse/INFRA-9401)) 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 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:

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><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>

<table>
<thead>
<tr>
<th>key</th>
<th><code>hashCode()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AaAaAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBaaBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBaaaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaaBBBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaaBBBBA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBaaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAaaaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAaaBBA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBaaaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBaaBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBaaaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBaaB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBBB&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

2\(^n\) strings of length 2\(^n\) 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, SHA-256, WHIRLPOOL, ....

known to be insecure

```java
String password = args[0];
MessageDigest sha = MessageDigest.getInstance("SHA-256");
byte[] bytes = sha.digest(password);

/* prints bytes as hex string */
```

Applications. Crypto, message digests, passwords, Bitcoin, blockchain, ....

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.
- More probes because of clustering.
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).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement compareTo() than hashCode().

Java system includes both.