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>$n$</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>hashing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
</tbody>
</table>

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

† under suitable technical assumptions
Hashing: basic plan

Save key–value pairs in a key-indexed table (index is a function of the key).

Hash function: Mathematical function that maps (hashes) a key to an array index.

Collision: Two distinct keys that hash to same index.

Issue. Collisions are unavoidable.
• How to limit collisions?
• How to accommodate collisions?
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Designing a hash function

Required properties. [for correctness]
- Deterministic.
- Each key hashes to a table index between 0 and \( m - 1 \).

Desirable properties. [for performance]
- Very fast to compute.
- For any subset of \( n \) input keys, each table index gets approximately \( \frac{n}{m} \) keys.

leads to good hash-table performance  
\((m = 10, n = 20)\)

leads to poor hash-table performance  
\((m = 10, n = 20)\)
Designing a hash function

**Required properties.** [for correctness]

- Deterministic.
- Each key hashes to a table index between 0 and $m - 1$.

**Desirable properties.** [for performance]

- Very fast to compute.
- For any subset of $n$ input keys, each table index gets approximately $n/m$ keys.

**Ex 1.** [$m = 10,000$] Last 4 digits of U.S. Social Security number.

**Ex 2.** [$m = 10,000$] Last 4 digits of phone number.
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 `hashCode()` 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](attachment:image.png)

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

**Legal (but highly undesirable) implementation.** Always return 17.

**User-defined types.** Users are on their own.
Implementing `hashCode()`: integers 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
Implementing `hashCode()`: 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;
    }
}
```

Java library implementation
Implementing hashCode(): 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()
    {
        int hash = 1;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

For reference types, use `hashCode()`
For primitive types, use `hashCode()` of wrapper type
Implementing `hashCode()`: 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);
    }
}
```
Implementing `hashCode()`

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

**Principle.** Every significant field contributes to hash.

**In practice.** Recipe above works reasonably well; used in Java libraries.
Which Java function 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.
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; }
```

- **m** typically a prime or a power of 2

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

- the remainder operator can evaluate to a negative integer

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

- 1-in-a-billion bug

`hashCode()` of "polygenelubricants" and new Double(-0.0) is $-2^{31}$
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).

$m$ typically a prime or a power of 2

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

discard sign bit

```java
private int hash(Key key)
{
    int h = key.hashCode();
    h ^= (h >>> 20) ^ (h >>> 12) ^ (h >>> 7) ^ (h >>> 4);
    return h & (m-1);
}
```

assumes $m$ is a power of 2

Java 7 (protects against poor quality hashCode())

$x$

$x$.hashCode()

hash(x)
Uniform hashing assumption

Uniform hashing assumption. Any key is equally likely to hash to one of \( m \) possible indices.

Bins and balls. Toss \( n \) balls uniformly at random into \( m \) bins.

![](image)

\[ m = 16 \text{ bins, } n = 11 \text{ balls} \]

Bad news. [birthday problem]

- In a random group of 23 people, more likely than not that two people share the same birthday.
- Expect two balls in the same bin after \( \sim \sqrt{\pi \frac{m}{2}} \) tosses.

23.9 when \( m = 365 \)
Uniform hashing assumption

Uniform hashing assumption. Any key is equally likely to hash to one of $m$ possible indices.

Bins and balls. Toss $n$ balls uniformly at random into $m$ bins.

[Diagram of balls being tossed into bins]

$m = 16$ bins, $n = 11$ balls

Good news. [load balancing]

- When $n \gg m$, expect most bins to have approximately $n/m$ balls.
- When $n = m$, expect most loaded bin has $\sim \ln n / \ln \ln n$ balls.
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Collisions

**Collision.** Two distinct keys that hash to the same index.

- Birthday problem $\Rightarrow$ can’t avoid collisions.  
  $\Rightarrow$ no index gets too many collisions.  
  $\Rightarrow$ ok to scan through all colliding keys.

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

Use an array of \( m \) linked lists.

- **Hash:** map key to table index \( i \) between 0 and \( m - 1 \).
- **Insert:** add key–value pair at front of chain \( i \) (if not already in chain).

**separate-chaining hash table \((m = 4)\)**

- \( put(L, 11) \)
- \( hash(L) = 3 \)
Separate-chaining hash table

Use an array of \( m \) linked lists.

- **Hash**: map key to table index \( i \) between 0 and \( m - 1 \).
- **Insert**: add key–value pair at front of chain \( i \) (if not already in chain).
- **Search**: perform sequential search in chain \( i \).

```plaintext
get(E)
hash(E) = 1
```
Separate-chaining hash table: Java implementation

```java
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) {
        /* as before */
    }

    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;
    }
}
```
Separate-chaining hash table: Java implementation

```java
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) {
        /* as before */
    }

    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]);
    }
}
```
Recall load balancing. Under uniform hashing assumption, length of each chain is tightly concentrated around mean $= n / m$.

Consequence. Expected number of probes for search/insert is $\Theta(n / m)$.

- $m$ too small $\Rightarrow$ chains too long.
- $m$ too large $\Rightarrow$ too many empty chains.
- Typical choice: $m \sim \frac{1}{4} n \Rightarrow \Theta(1)$ time for search/insert.
Resizing in a separate-chaining hash table

**Goal.** Average length of chain $n/m$ is $\Theta(1)$.
- 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 ($n/m = 8$)

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

After resizing ($n/m = 4$)

```
<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>J</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
<td>M</td>
</tr>
</tbody>
</table>
```

$x$'s `hashCode()` does not change; but `hash(x)` typically does
How to delete a key–value pair from a separate–chaining hash table?

A. Search for key; remove key–value pair from chain.
B. Compute hash of key; reinsert all other key–value pairs in chain.
C. Either A or B.
D. Neither A nor B.
## 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>$1^\dagger$</td>
</tr>
</tbody>
</table>

$^\dagger$ under uniform hashing assumption
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Linear-probing hash table: insert

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by probing: search successive cells until either finding the key or an unused cell.

Inserting into a linear-probing hash table.

<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
K
14
Linear-probing hash table: search

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by probing: search successive cells until either finding the key or an unused cell.

Searching in a linear-probing hash table.

**linear-probing hash table**

<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>K</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>14</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

get(K)  
hash(K) = 7

get(Z)  
hash(Z) = 8
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 $i + 1$, $i + 2$, ....

**Search.** Search table index $i$; if occupied but no match, try $i + 1$, $i + 2$, ....

**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>
</table>

$m = 16$
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)
    { /* 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 (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
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)
    { /* as before */ }

    public 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;
    }
}
Under the uniform hashing assumption, where is the next key most likely to be added in this linear-probing hash table (no resizing)?

A. Index 7.
B. Index 14.
C. Either index 4 or 14.
D. All open indices are equally likely.
Clustering

Cluster. A contiguous block of keys.

Observation. New keys disproportionately likely to hash into big clusters.
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

$$\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{search hit}$$

$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \quad \text{search miss / insert}$$

**Pf.** [beyond course scope]

**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><strong>keys[]</strong></td>
<td>E</td>
<td>S</td>
<td></td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>vals[]</strong></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><strong>keys[]</strong></td>
<td></td>
<td>A</td>
<td>S</td>
<td></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><strong>vals[]</strong></td>
<td></td>
<td>2</td>
<td>0</td>
<td></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>
</tr>
</tbody>
</table>
Hash tables: quiz 6

How to delete a key-value pair from a linear-probing hash table?

A. Search for key; remove key-value pair from arrays.

B. Search for key; remove key-value pair from arrays.
   Shift all keys in cluster after deleted key 1 position to left.

C. Either A and B.

D. Neither A nor B.

<table>
<thead>
<tr>
<th>before deleting S</th>
<th>cluster after deleted key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td>---------------------------</td>
</tr>
<tr>
<td>keys[]</td>
<td></td>
</tr>
<tr>
<td>P M A C S H L E</td>
<td>R X</td>
</tr>
<tr>
<td>vals[]</td>
<td>10 9 8 4 0 5 11 12</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</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^\dagger )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear probing</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( 1^\dagger )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( ^\dagger \) under uniform hashing assumption
3-Sum (revisited)

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

Goal. $\Theta(n^2)$ expected time; $\Theta(n)$ extra space.
3.4 **Hash Tables**

- hash functions
- separate chaining
- linear probing
- context

[Algorithms](https://algs4.cs.princeton.edu)
War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?
A1. Yes: aircraft control, nuclear reactor, pacemaker, HFT, ...

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.
**War story: algorithmic complexity attacks**

A Java bug report.

---

**Jan Lieskovsky 2011-11-01 14:13:47 UTC**

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](https://bugzilla.redhat.com/show_bug.cgi?id=750533) 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://bugzilla.redhat.com/show_bug.cgi?id=750521)) 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:


[https://bugzilla.redhat.com/show_bug.cgi?id=750533](https://bugzilla.redhat.com/show_bug.cgi?id=750533)
Hashing: file verification

When downloading a file from the web:

- Vendor publishes hash of file.
- Client checks whether hash of downloaded file matches.
- If mismatch, file corrupted.  
  (e.g., error in transmission or infected by virus)

Download IntelliJ IDEA

Version: 2019.3.3
Build: 193.6494.35
10 February 2020

Release notes

Download and verify the file SHA-256 checksum.

c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270

~Desktop> sha256sum ideaIC-2019.3.3.dmg
c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270
Hashing: cryptographic applications

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. MD5, SHA-1, SHA-256, SHA-512, Whirlpool, ....

Applications. File verification, digital signatures, cryptocurrencies, password authentication, blockchain, Git commit identifiers, ....
Separate chaining vs. linear probing

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

Linear probing.
- Less memory.
- 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 $\Theta(\log \log n)$.

**Double hashing.** [linear-probing variant]
- Resolve collisions by probing, but skip a variable amount instead of +1.
- 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).
- $\Theta(1)$ time for search in worst case.
Hash tables vs. balanced search trees

Hash tables.
- Simpler to code.
- Typically faster in practice.
- No effective alternative for unordered keys.

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

Java includes both.
- BSTs: `java.util.TreeMap`, `java.util.TreeSet`.

Examples of possible tree structures:
- Red-black BST
- Separate chaining
  (Java 8: if chain gets too long, use red-black BST for chain)
- Linear probing