3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
“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.

\[
\text{hash("it")} = 3
\]

\[
\text{hash("times")} = 3
\]

**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.
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.
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.
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));
    }
}
```

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

Treat string of length $L$ as $L$-digit, base-31 number:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$$

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

Horner's method: only $L$ multiplies/adds to hash string of length $L$.

String $s = "call"$;
`s.hashCode();`  \[3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 = 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))\]
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);
        hash = h;
        return h;
    }
}
```

Q. What if `hashCode()` of string is 0?  
(hashCode() of "pollinating sandboxes" is 0)
Implementing hash code: user-defined types

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;
    }
}
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.
Hash tables: quiz 1

Which of the following is an effective way to map a hashable key to an integer 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.

E.   *I don't know.*
**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).

```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**

`hashCode()` of "polygenelubricants" is $-2^{31}$

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

**correct**
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.

![Hash value frequencies for words in Tale of Two Cities (M = 97)](image)

Java's String data uniformly distribute the keys of Tale of Two Cities
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 $M < N$ 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 ($M = 4$)

```
put(L, 11)
hash(L) = 3
```
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 ($M = 4$)

get(E)
hash(E) = 1
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 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 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.

![Binomial distribution](image)

**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 ($N/M = 8$)

```
0 1
st[]
```

```
A B C D E F G H I J
K L M N O P
```

after resizing ($N/M = 4$)

```
0 1 2 3
st[]
```

```
K I
P N L E A
J F C B
O M H G D
```
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**

```
0 1 2 3
st[]

K → I
P → N → L
J → F → C → B
O → M
```

**after deleting C**

```
0 1 2 3
st[]

K → I
P → N → L
J → F → B
O → M
```
## 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><strong>search</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>search hit</strong></td>
<td>log N</td>
<td>log N</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>log N</td>
<td>log N</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>√ N</td>
<td>✓</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sequential search (unordered list)</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<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>log N</td>
<td>log N</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>red–black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>✓</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>
</tr>
</thead>
<tbody>
<tr>
<td>P M A C H L E R X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 10 9 5 6 12 13 4 8</td>
</tr>
</tbody>
</table>
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 \( N \).

<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>keys[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( M = 16 \)
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 (key.equals(keys[i]))
                return vals[i];
        return null;
    }

}
```

array doubling and halving code omitted

sequential search in chain i
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 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;
    }
}

sequential search in chain i
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 $i + 1$, $i + 2$, etc.

**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:

\[
\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{for search hit}
\]

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

**Pf.**

**Parameters.**

- \( M \) too large \( \Rightarrow \) too many empty array entries.
- \( M \) too small \( \Rightarrow \) search time blows up.
- Typical choice: \( \alpha = \frac{N}{M} \sim \frac{1}{2} \).
  
  # 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 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></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></td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>3</td>
<td>2</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></td>
<td>E</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></td>
<td></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>
</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 just delete array entries.

### before deleting S

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

### after deleting S?

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td></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><strong>search</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>search hit</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>sequential search</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>(unordered list)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>binary search</strong></td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓ compareTo()</td>
</tr>
<tr>
<td><strong>(ordered array)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓ compareTo()</td>
</tr>
<tr>
<td><strong>red-black BST</strong></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\sqrt{N}$</td>
<td>✓ compareTo()</td>
</tr>
<tr>
<td><strong>separate chaining</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓ equals()</td>
</tr>
<tr>
<td><strong>linear probing</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓ equals()</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]

- 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.

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) 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;AaAaAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBBa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBbBAaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBbBbBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AbBBaAaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AbBBbBbA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AbBBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBbBbB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBbB&quot;</td>
<td>-540425984</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

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.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.

linear probing separate chaining
3.5 SYMBOL TABLE APPLICATIONS

- sets
- dictionary clients
- indexing clients
- sparse vectors
3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
**Set API**

**Mathematical set.** A collection of distinct keys.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET()</td>
<td>create an empty set</td>
</tr>
<tr>
<td>void add(Key key)</td>
<td>add the key to the set</td>
</tr>
<tr>
<td>boolean contains(Key key)</td>
<td>is the key in the set</td>
</tr>
<tr>
<td>void remove(Key key)</td>
<td>remove the key from the set</td>
</tr>
<tr>
<td>int size()</td>
<td>number of keys in the set</td>
</tr>
<tr>
<td>Iterator&lt;Key&gt; iterator()</td>
<td>all keys in the set</td>
</tr>
</tbody>
</table>

**Q.** How to implement efficiently?
Exception filter

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

<table>
<thead>
<tr>
<th>application</th>
<th>purpose</th>
<th>key</th>
<th>in list</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell checker</td>
<td>identify misspelled words</td>
<td>word</td>
<td>dictionary words</td>
</tr>
<tr>
<td>browser</td>
<td>mark visited pages</td>
<td>URL</td>
<td>visited pages</td>
</tr>
<tr>
<td>parental controls</td>
<td>block sites</td>
<td>URL</td>
<td>bad sites</td>
</tr>
<tr>
<td>chess</td>
<td>detect draw</td>
<td>board</td>
<td>positions</td>
</tr>
<tr>
<td>spam filter</td>
<td>eliminate spam</td>
<td>IP address</td>
<td>spam addresses</td>
</tr>
<tr>
<td>credit cards</td>
<td>check for stolen cards</td>
<td>number</td>
<td>stolen cards</td>
</tr>
</tbody>
</table>
Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are not in the list.

```java
public class BlackList
{
    public static void main(String[] args)
    {
        SET<String> set = new SET<String>();
        In in = new In(args[0]);
        while (!in.isEmpty())
            set.add(in.readString());
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (!set.contains(word))
                StdOut.println(word);
        }
    }
}
```
3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
Dictionary lookup

Command-line arguments.
- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 1. DNS lookup.

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
domain name is key  IP is value

% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
domain name is key  URL is value

% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...```
Dictionary lookup

Command-line arguments.
- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

% java LookupCSV amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine

% more amino.csv
TTT, Phe, F, Phenylalanine
TTC, Phe, F, Phenylalanine
TTA, Leu, L, Leucine
TTG, Leu, L, Leucine
TCT, Ser, S, Serine
TCC, Ser, S, Serine
TCA, Ser, S, Serine
TCG, Ser, S, Serine
TAT, Tyr, Y, Tyrosine
TAC, Tyr, Y, Tyrosine
TAA, Stop, Stop, Stop
TAG, Stop, Stop, Stop
TGT, Cys, C, Cysteine
TGC, Cys, C, Cysteine
TGA, Stop, Stop, Stop
TGG, Trp, W, Tryptophan
CTT, Leu, L, Leucine
CTC, Leu, L, Leucine
CTA, Leu, L, Leucine
CTG, Leu, L, Leucine
CCT, Pro, P, Proline
CCC, Pro, P, Proline
CCA, Pro, P, Proline
CGG, Pro, P, Proline
CAT, His, H, Histidine
CAC, His, H, Histidine
CAA, Gln, Q, Glutamine
CAG, Gln, Q, Glutamine
CGT, Arg, R, Arginine
CGC, Arg, R, Arginine
...
Dictionary lookup

**Command-line arguments.**
- A comma-separated value (CSV) file.
- Key field.
- Value field.

**Ex 3. Class list.**

```java
% java LookupCSV classlist.csv 4 1
eberl
Ethan
nwebb
Natalie

% java LookupCSV classlist.csv 4 3
dpan
P01
```

% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cckehoud
10,Douglas,Malia Morioka,P01,malia
12,Haddock,Sara Lynn,P01,shaddock
12,Hantman,Nicole Samantha,P01,nhaitman
11,Hesterberg,Adam Classen,P01,ahesterb
13,Hwang,Roland Lee,P01,rhwang
13,Hyde,Gregory Thomas,P01,ghyde
13,Kim,Hyunmoon,P01,hktwo
12,Korac,Damjan,P01,dkorac
11,MacDonald,Graham David,P01,gmacdona
10,Michal,Brian Thomas,P01,bmichal
12,Nam,Seung Hyeon,P01,seungnam
11,Nastasescu,Maria Monica,P01,mnastase
11,Pan,Di,P01,dpan
12,Partridge,Brenton Alan,P01,bpartrid
13,Rilee,Alexander,P01,arilee
13,Roopakalu,Ajay,P01,aroopaka
11,Sheng,Ben C,P01,bsheng
12,Webb,Natalie Sue,P01,nwebb
```
Dictionary lookup: Java implementation

```java
public class LookupCSV {
    public static void main(String[] args) {
        In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);

        ST<String, String> st = new ST<String, String>();
        while (!in.isEmpty()) {
            String line = in.readLine();
            String[] tokens = line.split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }

        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (!st.contains(s)) StdOut.println("Not found");
            else StdOut.println(st.get(s));
        }
    }
}
```

- **process input file**: The program reads a file named `args[0]` and processes each line.
- **build symbol table**: It parses each line to extract the key and value fields.
- **process lookups with standard I/O**: It reads a stream of strings and looks up each string in the symbol table, printing the corresponding value or a message if not found.
3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
File indexing

**Goal.** Index a PC (or the web).
**File indexing**

**Goal.** Given a list of text files, create an index so that you can efficiently find all files containing a given query string.

% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt
freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt

% ls *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

% java FileIndex *.java
import
FileIndex.java SET.java ST.java

Comparator
null
Which data type below would be the best choice to represent the file index?

A. \( \text{SET<ST<File, String>>} \)

B. \( \text{SET<ST<String, File>>} \)

C. \( \text{ST<File, SET<String>>} \)

D. \( \text{ST<String, SET<File>>} \)

E. \( I \text{ don't know.} \)
import java.io.File;
public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>();

        for (String filename : args) {
            File file = new File(filename);
            In in = new In(file);
            while (!in.isEmpty())
            {
                String key = in.readString();
                if (!st.contains(key))
                {
                    st.put(word, new SET<File>());
                    SET<File> set = st.get(key);
                    set.add(file);
                }
            }
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            StdOut.println(st.get(query));
        }
    }
}
Book index

Goal. Index for an e-book.

Index

Abstract data type (ADT), 127-195
abstract classes, 163
classes, 129-136
collections of items, 137-139
creating, 157-164
defined, 128
duplicate items, 173-176
equivalence relations, 159-162
FIFO queues, 163-171
first-class, 177-186
generic operations, 273
index items, 177
interframe operations, 138-139
modular programming, 135
polynomials, 188-192
priority queue, 375-376
pushdown stack, 138-156
stubs, 135
symbol table, 497-506
ADT interfaces
array (myArray), 274
complex number (Complex), 181
description table (DEI), 663
full priority queue (FQULL), 397
indirect priority queue (PRQI), 403
item (myItem), 273, 498
key (myKey), 498
polynomial (Poly), 189
point (Point), 134
priority queue (PQ), 375
queue of int (intQueue), 166
stack of int (intStack), 140
symbol table (ST), 503
text index (TI), 525
union-find (UF), 159
Abstract in-place merging, 351-353
Abstract operation, 10
access control state, 131
Actual data, 31
Adapter class, 158-157
Adaptive sort, 268
Address, 84-85
Adjacency list, 120-123
depth-first search, 251-256
Adjacency matrix, 120-122
Aitai, M., 464
Algorithm, 4-6, 27-64
abstract operations, 10, 31, 34-35
analysis of, 6
average-case performance, 35, 60-62
big-O notation, 44-47
binary search, 36-59
computational complexity, 62-64
efficiency, 6, 30, 52
empirical analysis, 30-32, 58
exponential-time, 219
implementation, 28-50
logarithm function, 40-43
mathematical analysis, 33-36, 58
primary parameter, 36
probabilistic, 331
recurrences, 49-52, 57
recursive, 198
running time, 34-40
search, 33-36, 498
steps in, 22-23
See also Randomized algorithm
Amortization approach, 557, 627
Arithmetic operators, 177-179, 188, 191
Arrays, 12, 83
binary search, 57
dynamic allocation, 87
and linked lists, 92, 94-95
merging, 349-350
multidimensional, 117-118
references, 86-87, 89
sorting, 265-267, 273-276
and strings, 119
two-dimensional, 117-118, 120-124
vectors, 87
visualizations, 295
See also Index, array
Array representation
binary tree, 381
FIFO queue, 168-169
linked lists, 110
polynomial ADT, 191-192
priority queue, 377-378, 403, 406
pushdown stack, 148-150
random queue, 170
symbol table, 506, 511-512, 521
Asymptotic expression, 45-46
Average deviation, 60-81
Average-case performance, 35, 60-61
AVL tree, 583
B tree, 584, 692-704
description, 693-704
Markov chain, 701
remove, 701-703
search in, 697-701
selections, 701
Balanced tree, 238, 355-598
B tree, 584
bottom-up, 576, 584-585
height-balanced, 583
indexed sequential access, 690-692
performance, 575-576, 581-582, 595-598
randomized, 359-364
red-black, 577-585
skip lists, 587-594
splice, 566-571

727
Concordance

**Goal.** Preprocess a text corpus to support concordance queries: given a word, find all occurrences with their immediate contexts.

```java
% java Concordance tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
   of his most gracious *majesty* king george the third

princeton
no matches
```

**Solution.** Key = query string; value = set of indices containing that string.
public class Concordance {
    public static void main(String[] args) {
        In in = new In(args[0]);
        String[] words = in.readAllStrings();
        ST<String, SET<Integer>> st = new ST<String, SET<Integer>>(
            (String s) -> new SET<Integer>(),
            (String s) -> st.get(s),
            (String s) -> st.put(s, new SET<Integer>())
        );
        for (int i = 0; i < words.length; i++) {
            String s = words[i];
            if (!st.contains(s)) {
                st.put(s, new SET<Integer>());
                SET<Integer> set = st.get(s);
                set.add(i);
            }
        }
        while (!StdIn.isEmpty()) {
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            for (int k : set) {
                // print words[k-4] to words[k+4]
            }
        }
    }
}
3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
Matrix-vector multiplication (standard implementation)

\[
\begin{bmatrix}
0 & 0.90 & 0 & 0 & 0 \\
0 & 0 & 0.36 & 0.36 & 0.18 \\
0 & 0 & 0 & 0.90 & 0 \\
0.90 & 0 & 0 & 0 & 0 \\
0.47 & 0 & 0.47 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x[0] \\
x[1] \\
x[2] \\
x[3] \\
x[4]
\end{bmatrix}
= 
\begin{bmatrix}
b[0] \\
b[1] \\
b[2] \\
b[3] \\
b[4]
\end{bmatrix}
\]

```java
... double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...

// initialize a[][] and x[]
...
```

```
for (int i = 0; i < N; i++)
{
    sum = 0.0;
    for (int j = 0; j < N; j++)
        sum += a[i][j]*x[j];
    b[i] = sum;
}
```
Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.
Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.
Vector representations

1d array (standard) representation.
- Constant time access to elements.
- Space proportional to $N$.

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
0 .36 0  0  0 .36 0  0  0  0  0  0  0  .18 0  0  0  0  0
```

Symbol table representation.
- Key = index, value = entry.
- Efficient iterator.
- Space proportional to number of nonzeros.
Sparse vector data type

```java
public class SparseVector {
    private HashST<Integer, Double> v;

    public SparseVector() {
        v = new HashST<Integer, Double>();
    }

    public void put(int i, double x) {
        v.put(i, x);
    }

    public double get(int i) {
        if (!v.contains(i)) return 0.0;
        else return v.get(i);
    }

    public Iterable<Integer> indices() {
        return v.keys();
    }

    public double dot(double[] that) {
        double sum = 0.0;
        for (int i : indices())
            sum += that[i]*this.get(i);
        return sum;
    }
}
```

- **HashST because order not important**
- **empty ST represents all 0s vector**
- **a[i] = value**
- **return a[i]**
- **iterate through indices of nonzero entries**
- **dot product is constant time for sparse vectors**
Matrix representations

2D array (standard) matrix representation: Each row of matrix is an array.
- Constant time access to elements.
- Space proportional to $N^2$.

Sparse matrix representation: Each row of matrix is a sparse vector.
- Efficient access to elements.
- Space proportional to number of nonzeros (plus $N$).
Sparse matrix-vector multiplication

\[
\begin{bmatrix}
0 & .90 & 0 & 0 & 0 \\
0 & 0 & .36 & .36 & .18 \\
0 & 0 & 0 & .90 & 0 \\
.90 & 0 & 0 & 0 & 0 \\
.47 & 0 & .47 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x[0] \\
x[1] \\
x[2] \\
x[3] \\
x[4]
\end{bmatrix}
= 
\begin{bmatrix}
b[0] \\
b[1] \\
b[2] \\
b[3] \\
b[4]
\end{bmatrix}
\]

..  
SparseVector[] a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[] and x[]
...
for (int i = 0; i < N; i++)
    b[i] = a[i].dot(x);

linear running time for sparse matrix