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 hit</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>
</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.

```plaintext
hash("it") = 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 1. Last 4 digits of Social Security number.
Ex 2. Last 4 digits of phone number.

Practical challenges:
- Need different approach for each key type.
- Need to ensure uniformity of hash function.
- Ideally, should use all parts of the key.
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())`.

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;
    }
}
```

```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: 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 library implementation

```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 $L$.

```java
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

```java
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double amount;

    public Transaction(String who, Date when, double amount) {
        /* as before */
    }

    /* 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;
    }
}
```

- Typically a small prime
- Nonzero constant
- For reference types, use `hashCode()`
- For primitive types, use `hashCode()` of wrapper type
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" families 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 of the following is an effective way to map a hashable key to an integer between 0 and M-1?

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

B. 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() & 0xffffffff) % 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 ($M = 97$)

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 $M - 1$.
- Search: sequential search in $i$th chain.
- Insert: put at front of $i$th chain (if not already in chain).

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

separate-chaining hash table ($M = 4$)
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;
    }
}
```

This implementation of separate-chaining symbol table uses a hash function to distribute keys across the array. Each chain is represented by a `Node` class, which holds the key and value along with a pointer to the next node in the chain. The `hash()` method computes the hash value of the key, and the `get()` method searches the corresponding chain for the key. If found, it returns the associated value; otherwise, it returns null.

- **Array Doubling and Halving Code Omitted**: The code does not include the array doubling and halving logic, which are typically part of a more complete implementation.
- **No Generic Array Creation**: The implementation uses raw types for the array instead of generics, which is a common practice in Java to avoid overcomplicating the code without explicit need.
- **Declare Key and Value of Type Object**: The key and value are declared as `Object` types, which is a flexible approach but could be refined to specific types if known at runtime.
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 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 ($N/M = 8$)

```
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></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>
```
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 hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>1 *</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
3.4 Hash Tables

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

Open addressing. [Amdahl–Boehme–Rocherster–Samuel, IBM 1953]

- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot, and put it there.

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

<table>
<thead>
<tr>
<th>keys[]</th>
<th>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>R</td>
<td>X</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</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.
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 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
Cluster. A contiguous block of items.

Observation. New keys increasingly likely to hash into big clusters.
Knuth's parking problem

**Model.** Cars arrive at one-way street with $M$ parking spots. Each starts looking at a random spot $i$. If spot $i$ is taken, try $i + 1$, $i + 2$, etc.

**Q.** What is mean number of spots a car has to consider?

**Half full.** With $M / 2$ cars, mean number of tries is $\sim 2.5$.

**Three-quarters full.** With $3M / 4$ cars, mean number of tries is $\sim 8.5$.

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

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

- search hit
- search miss / insert

Pf.

Parameters.

- $M$ too large $\Rightarrow$ too many empty array entries.
- $M$ too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim \frac{1}{2}$.  
  # 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>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>
</tr>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### after resizing

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

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

* under uniform hashing assumption
### 3-Sum (Revisited)

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

**Goal.** $N^2$ expected time, $N$ 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.
War story: algorithmic complexity attacks

A Java bug report.

---


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

http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html
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;AaAaBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBAAaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBAABB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBBB&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th><code>hashCode()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BBAAaAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAAaAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAAaBBBa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAAaBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBAAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBAAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBaBBBa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBBBBa&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 shal = MessageDigest.getInstance("SHA1");
byte[] bytes = shal.digest(password);

/* prints bytes as hex string */

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

Caveat. Too expensive for use in ST implementations.
Separate chaining vs. linear probing

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

Linear probing.
- Less wasted space.
- Better cache performance.
Hashing: variations on the theme

Many improved versions have been studied.

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

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

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

Hash tables.

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

Balanced search trees.

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

Java system includes both.

- Red–Black BSTs: java.util.TreeMap, java.util.TreeSet.