

# 3.4 HASH TABLES



- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing

# ST implementations: summary

implementation	worst-case cost (after N inserts)			average-case cost (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$	?	yes	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>

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.

`hash("it") = 3`



**Issues.**

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

# 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.

## Issues.

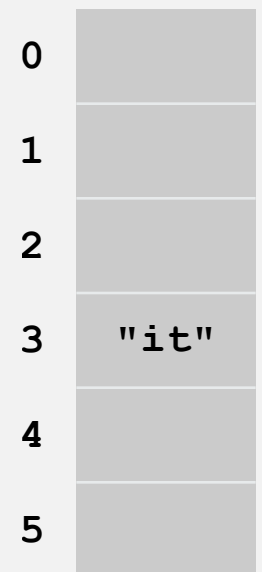
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

## 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).

`hash("it") = 3`

`hash("times") = 3`



- ▶ **hash functions**
- ▶ separate chaining
- ▶ linear probing

# 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.

thoroughly researched problem,  
still problematic in practical applications

**Ex 1. Phone numbers.**

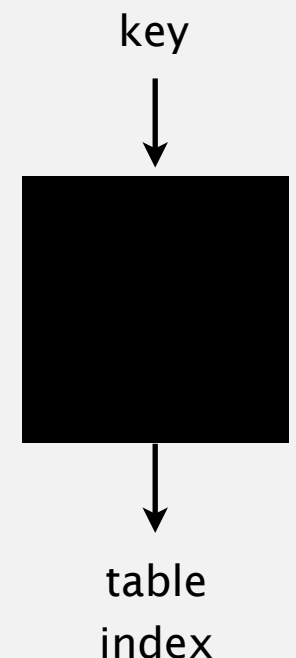
- Bad: first three digits.
- Better: last three digits.

**Ex 2. Social Security numbers.**

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska  
(assigned in chronological order within geographic region)

**Practical challenge.** Need different approach for each key type.

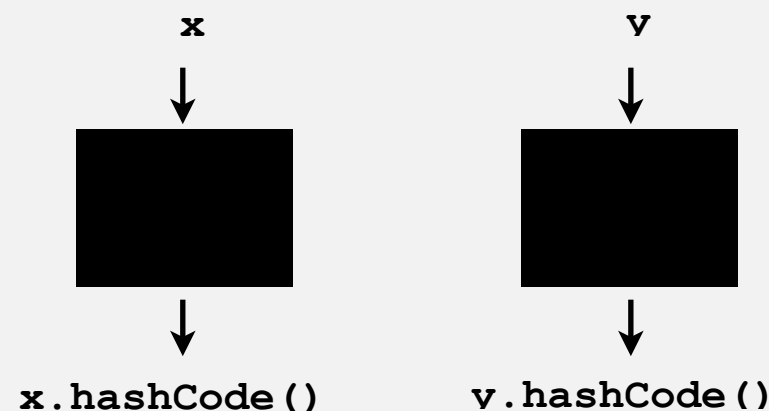


# Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

**Requirement.** If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

**Highly desirable.** If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



**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

```
public final class Integer
{
    private final int value;
    ...

    public int hashCode()
    { return value; }
}
```


```
public final class Boolean
{
    private final boolean value;
    ...

    public int hashCode()
    {
        if (value) return 1231;
        else       return 1237;
    }
}
```

```
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



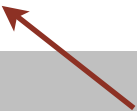


# Implementing hash code: strings

## Java library implementation

```
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;
    }
}
```



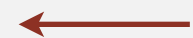
$i^{\text{th}}$  character of  $s$

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

- Horner's method to hash string of length  $L$ :  $L$  multiplies/adds.
- Equivalent to  $h = s[0] \cdot 31^{L-1} + \dots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$ .

Ex.

```
String s = "call";
int code = s.hashCode();
```



$$\begin{aligned} 3045982 &= 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\ &= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \\ &\quad \text{(Horner's method)} \end{aligned}$$

# Implementing hash code: strings

## Performance optimization.

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

```
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 * hash);
        hash = h;
        return h;
    }
}
```

← cache of hash code

← return cached value

← store cache of hash code

# 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;
    }
}
```

nonzero constant

for reference types,  
use `hashCode()`

for primitive types,  
use `hashCode()`  
of wrapper type

typically a small prime

# 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, return 0.
  - If field is a reference type, use `hashCode()`.
  - If field is an array, apply to each entry.
- ← applies rule recursively
- ← or use `Arrays.deepHashCode()`

**In practice.** Recipe works reasonably well; used in Java libraries.

**In theory.** Keys are bitstring; "universal" 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.

# 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).

typically a prime or power of 2

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

bug

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

1-in-a-billion bug

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

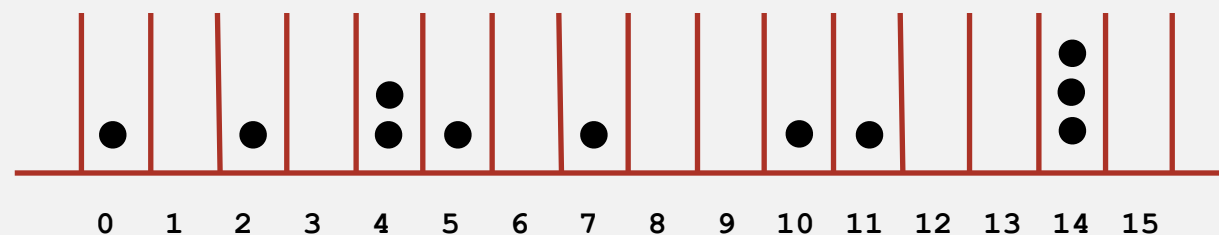
```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff) % M;   }
```

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.

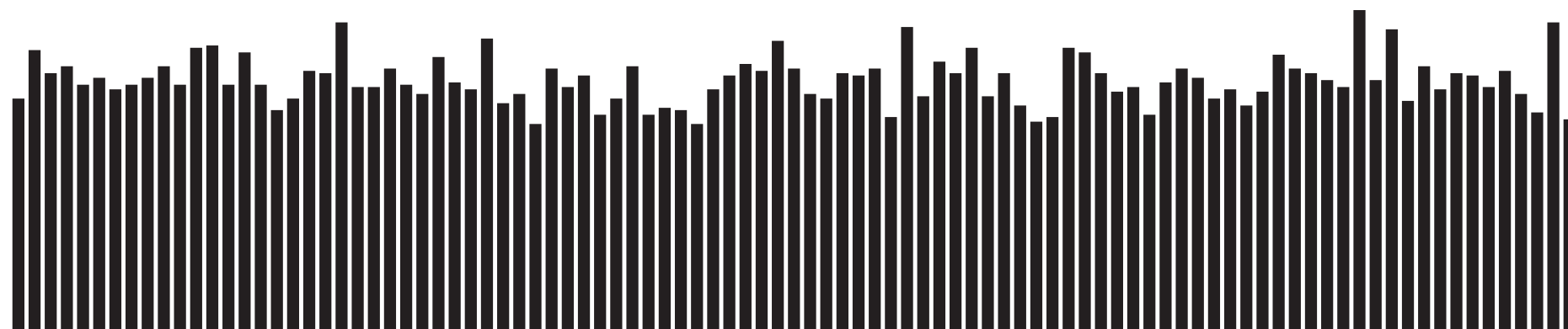
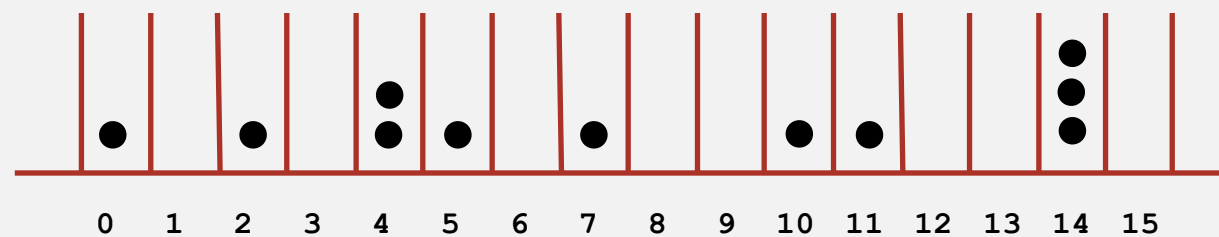
**Coupon collector.** Expect every bin has  $\geq 1$  ball after  $\sim M \ln M$  tosses.

**Load balancing.** After  $M$  tosses, expect most loaded bin has  $\Theta(\log M / \log \log M)$  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

- ▶ hash functions
- ▶ **separate chaining**
- ▶ linear probing

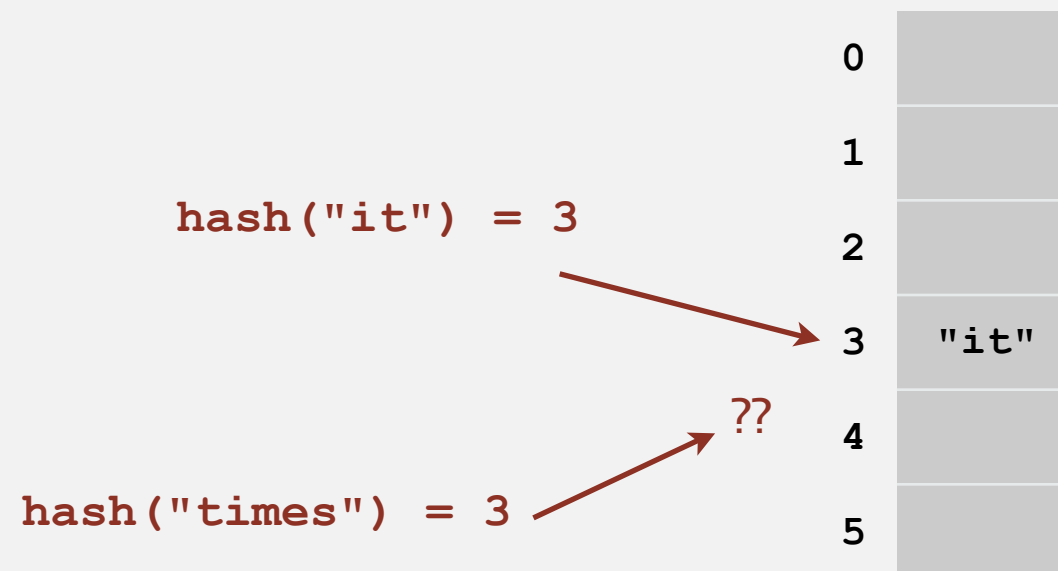


# Collisions

**Collision.** Two distinct keys hashing to same index.

- Birthday problem  $\Rightarrow$  can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing  $\Rightarrow$  collisions will be evenly distributed.

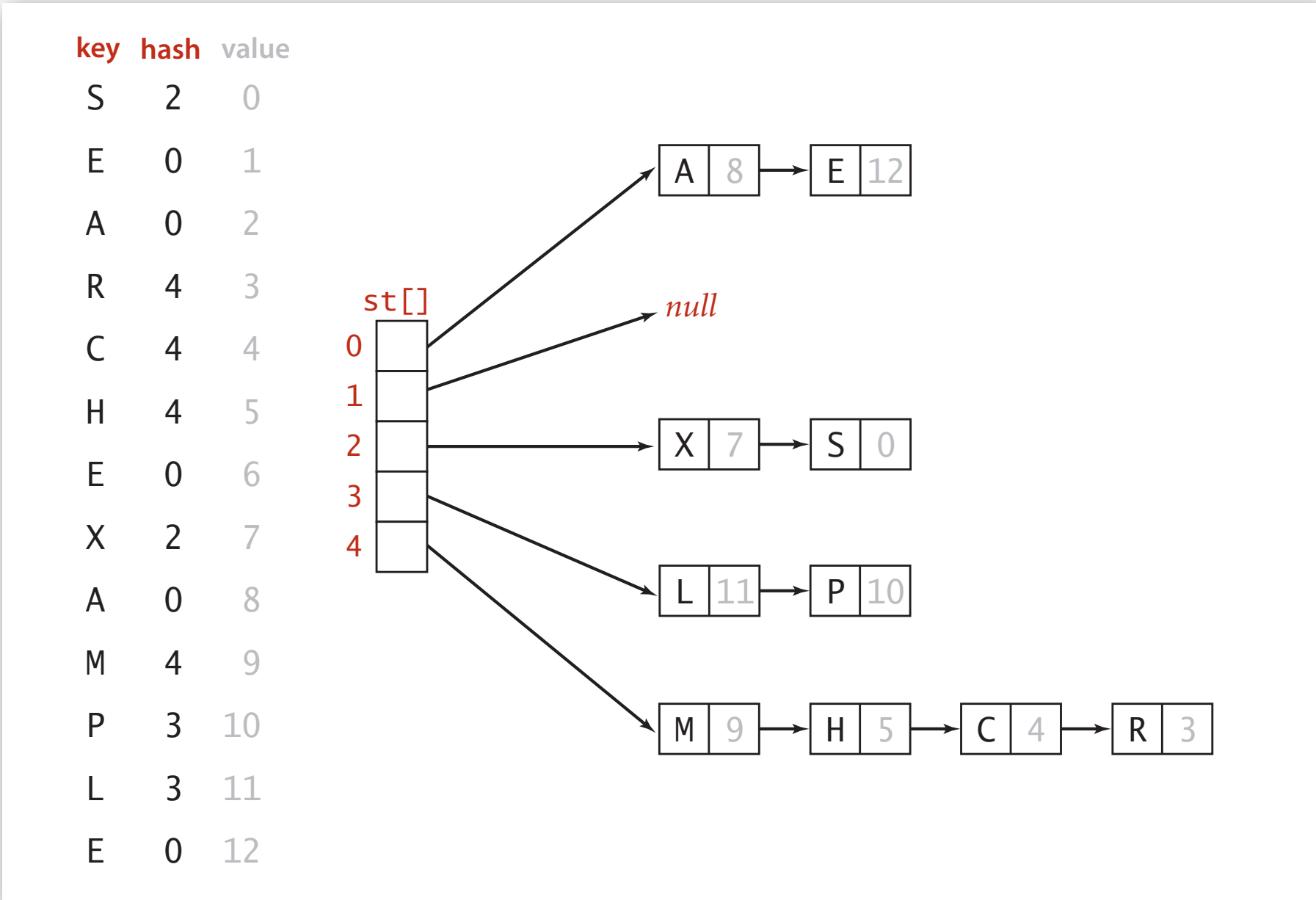
**Challenge.** Deal with collisions efficiently.



# Separate chaining symbol table

Use an array of  $M < N$  linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer  $i$  between 0 and  $M - 1$ .
- Insert: put at front of  $i^{\text{th}}$  chain (if not already there).
- Search: need to search only  $i^{\text{th}}$  chain.



# Separate chaining ST: Java implementation

```
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;
    }
}
```

array doubling  
and halving  
code omitted

no generic array creation  
(declare key and value of type Object)

# Separate chaining ST: Java implementation

```
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;
        ...
    }

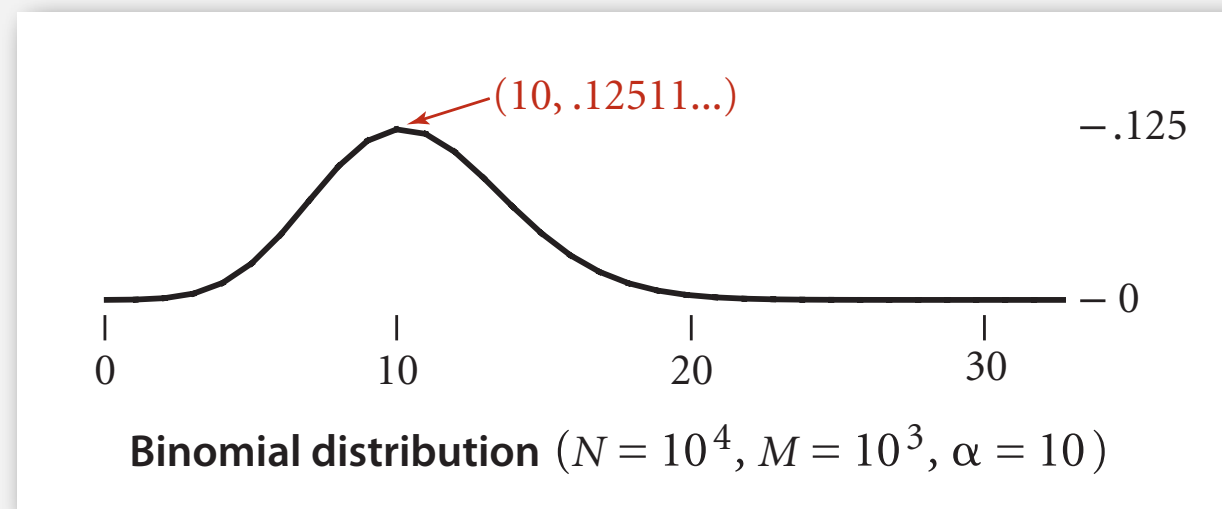
    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    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, probability 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.



`equals()` and `hashCode()`

**Consequence.** Number of probes for search/insert is proportional to  $N / M$ .

- $M$  too large  $\Rightarrow$  too many empty chains.
- $M$  too small  $\Rightarrow$  chains too long.
- Typical choice:  $M \sim N / 5 \Rightarrow$  constant-time ops.

$M$  times faster than  
sequential search

# ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$	?	yes	<code>compareTo()</code>
red-black tree	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
separate chaining	$\lg N$ *	$\lg N$ *	$\lg N$ *	3-5 *	3-5 *	3-5 *	no	<code>equals()</code>

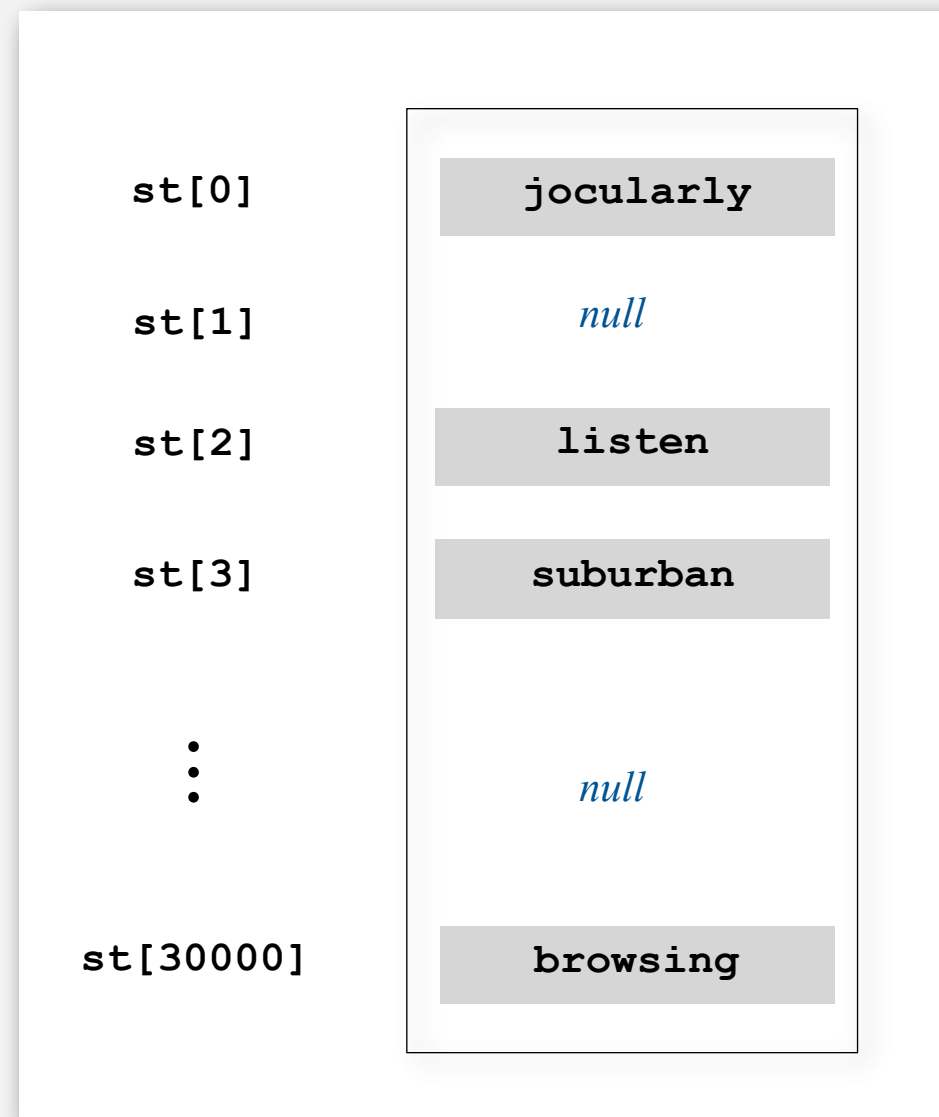
\* under uniform hashing assumption

- ▶ hash functions
- ▶ separate chaining
- ▶ **linear probing**

# Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



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



# Linear probing demo

**Hash.** Map key to integer  $i$  between 0 and  $M - 1$ .

**Insert.** Put at table index  $i$  if free; if not try  $i + 1$ ,  $i + 2$ , etc.

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

**Note.** Array size  $M$  must be greater than  $N$ .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

# Linear probing ST implementation

```
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 */ }

    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;
    }

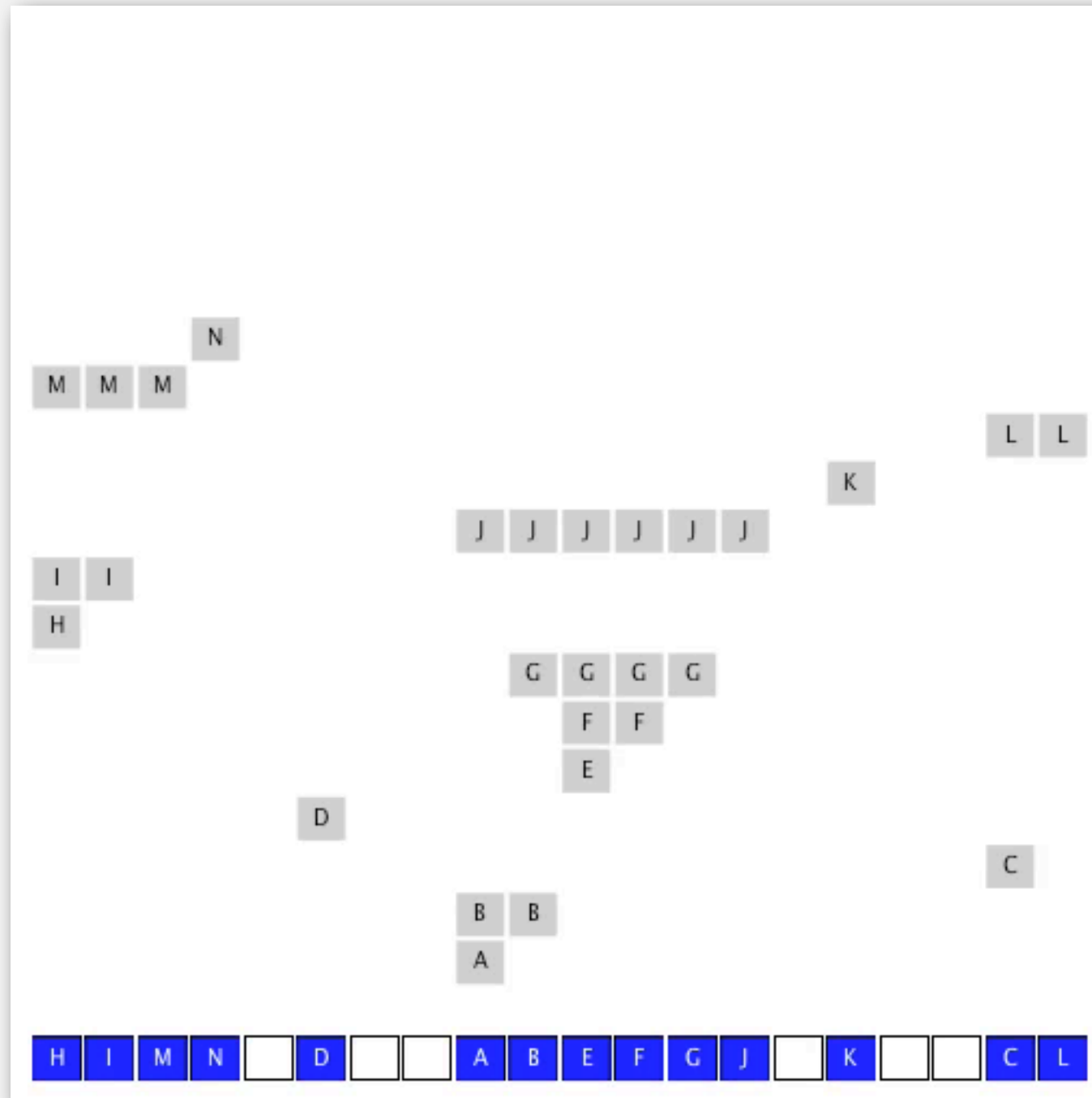
    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

← array doubling  
and halving  
code omitted

# Clustering

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

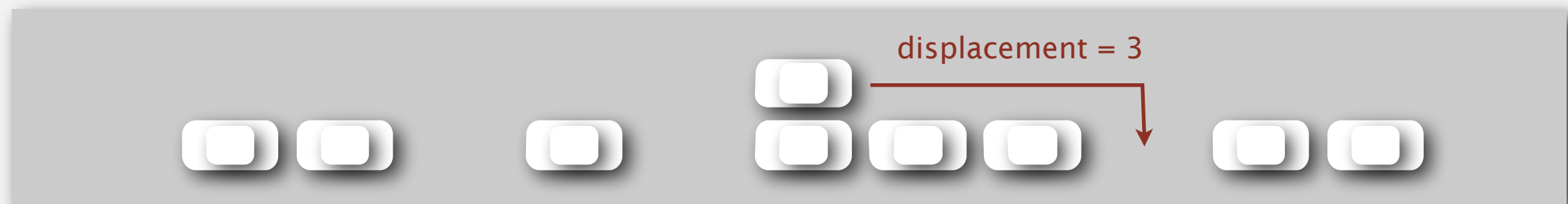
**Observation.** New keys likely to hash into middle of big clusters.



# 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  $\sim 3 / 2$ .

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

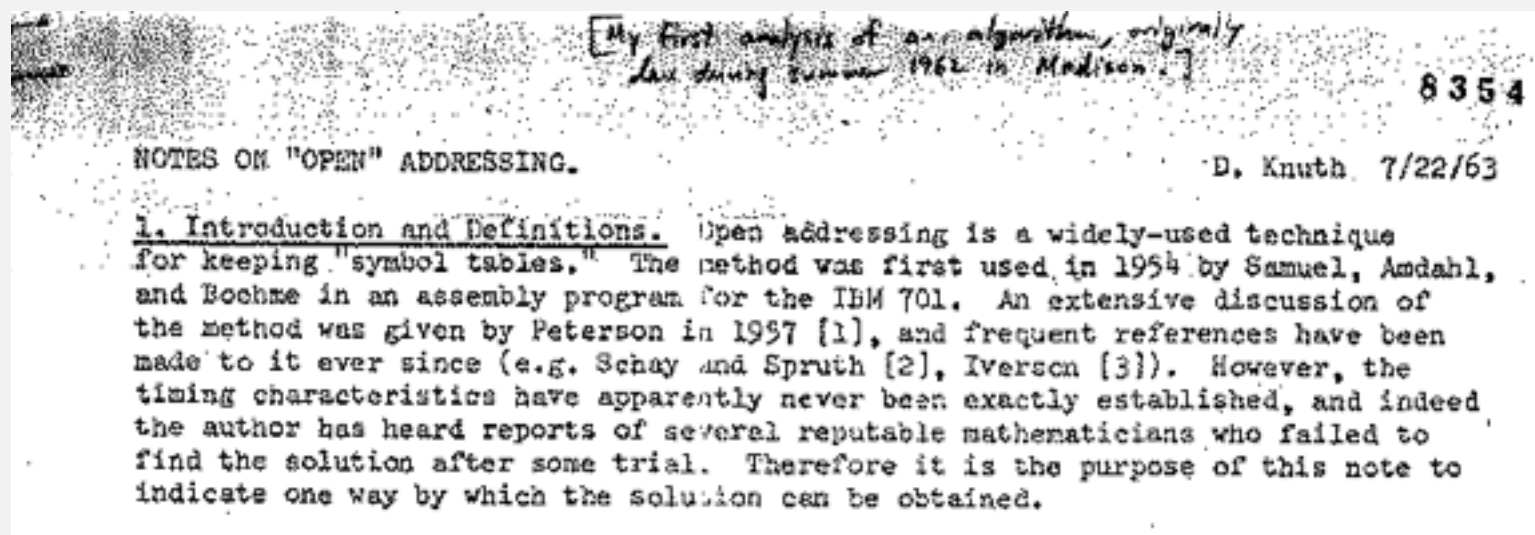
# Analysis of linear probing

**Proposition.** Under uniform hashing assumption, the average number 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:  $\alpha = N / M \sim 1/2$ . ←
  - # probes for search hit is about  $3/2$
  - # probes for search miss is about  $5/2$

# ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	<code>compareTo()</code>
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	<code>compareTo()</code>
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	<code>equals()</code>
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	<code>equals()</code>

\* under uniform hashing assumption

# War story: String hashing in Java

## String hashCode() in Java 1.1.

- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: saves time in performing arithmetic.

```
public int hashCode()  
{  
    int hash = 0;  
    int skip = Math.max(1, length() / 8);  
    for (int i = 0; i < length(); i += skip)  
        hash = s[i] + (37 * hash);  
    return hash;  
}
```

- Downside: great potential for bad collision patterns.

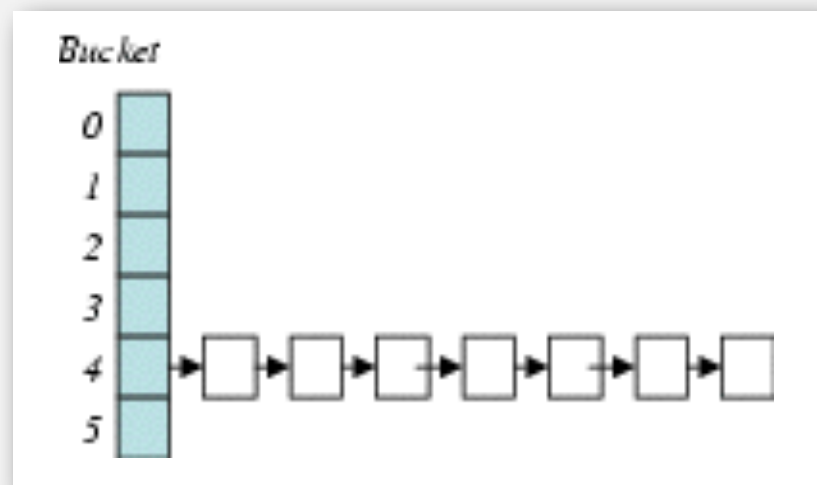
```
http://www.cs.princeton.edu/introcs/13loop/Hello.java  
http://www.cs.princeton.edu/introcs/13loop/Hello.class  
http://www.cs.princeton.edu/introcs/13loop/Hello.html  
http://www.cs.princeton.edu/introcs/12type/index.html  
↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑
```

# War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: **denial-of-service** attacks.



malicious adversary learns your hash function  
(e.g., by reading Java API) and causes a big pile-up  
in single slot that grinds performance to a halt

**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.



# Separate chaining vs. linear probing

## Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

## Linear probing.

- Less wasted space.
- Better cache performance.

Q. How to delete?

Q. How to resize?

# Hashing: variations on the theme

Many improved versions have been studied.

**Two-probe hashing.** (separate-chaining variant)

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to  $\log \log N$ .

**Double hashing.** (linear-probing variant)

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

**Cuckoo hashing.** (linear-probing variant)

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst case time for search.



# Hash tables vs. balanced search trees

## Hash tables.

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

## Balanced search trees.

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

## Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.