Algorithms

FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.4 HASH TABLES

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

Symbol table implementations: summary

| implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red–black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`

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`

`hash("times") = 3`

0	
1	
2	
3	"it"
4	
5	

??
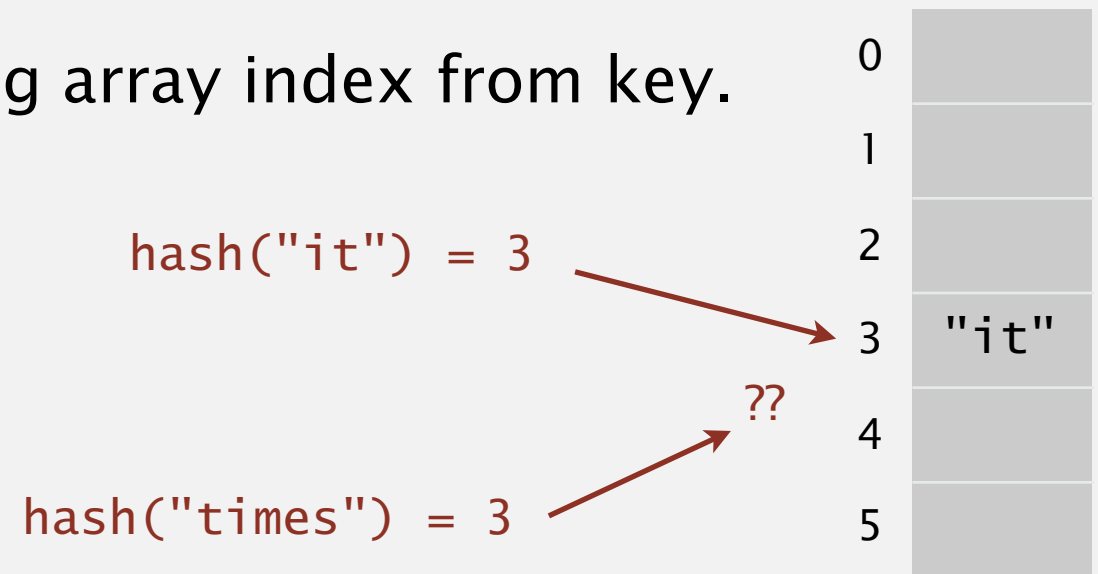
Issues.

- 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.

3.4 HASH TABLES

- ▸ *hash functions*
- ▸ *separate chaining*
- ▸ *linear probing*
- ▸ *context*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

How to implement a Hash Table given a hash function?

I asked you to think about this over the weekend.

- Did you do any thinking?

- Recall that the issue is collision resolution.

- What do we do when, inevitably, two keys hash to the same index in our hash table?

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

key

**table
index**

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
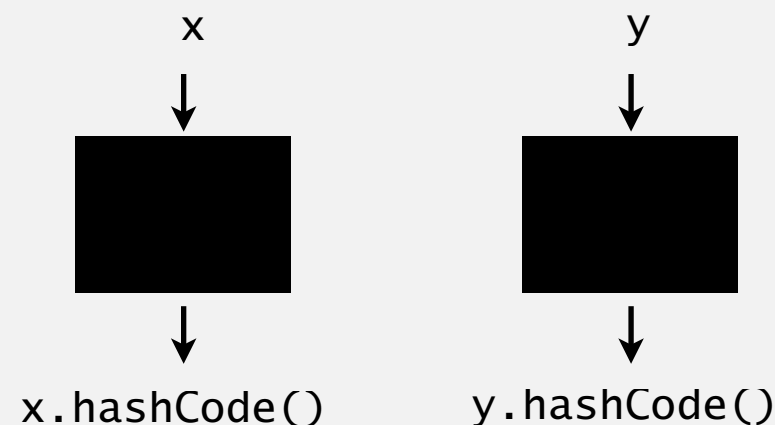(assigned in chronological order within geographic region)
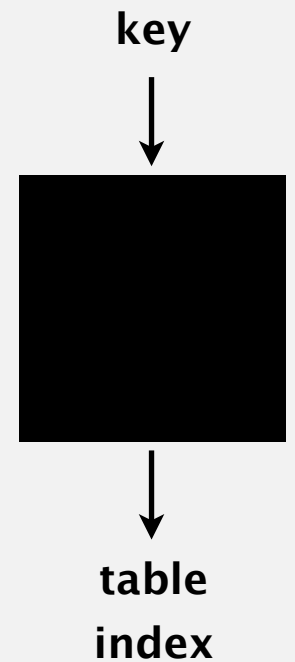
Practical challenge.   Need different approach for each key type.

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

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

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

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

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

```java
public final class Double
{
    private final double value;
    ...

    public int hashCode()
    {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

XOR operator(^): if same return 0, if different return 1

>> (Signed right shift), >>> (Unsigned right shift)

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

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;
    }
}
```

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

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

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$
$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$
(Horner's method)

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;                           ← cache of hash code
    private final char[] s;

    ...

    public int hashCode()
    {
        int h = hash;                               ← return cached value
        if (h != 0) return h;
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;                                   ← store cache of hash code
        return h;
    }
}
```

Q. What if `hashCode()` of string 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;
    }
}
```
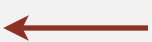
nonzero constant

for reference types,
use hashCode()

for primitive types,
use hashCode()
of wrapper type

typically a small prime

"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()`. ⟵ applies rule recursively
- If field is an array, apply to each entry. ⟵ or use `Arrays.deepHashCode()`

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

In theory. "Universal" hash functions exist (randomly choose different hash function for each key).

Basic rule. Need to use the whole key to compute hash code;

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;   }
```

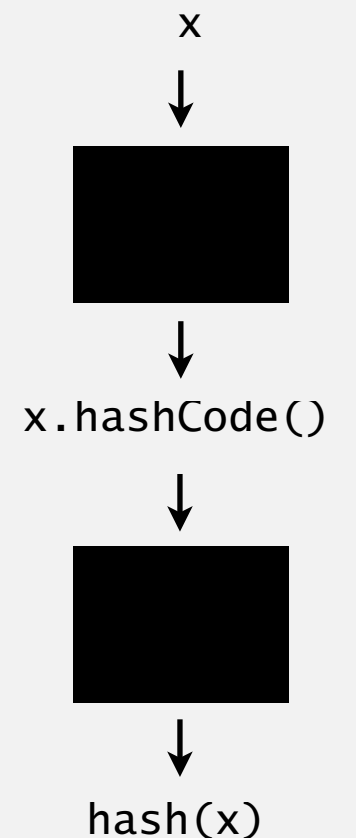0x7FFFFFFF is 0111 1111 1111 1111 1111 1111 1111 1111 all 1 except the sign bit.
(hash & 0x7FFFFFFF will result in a positive integer
(hash & 0x7FFFFFFF) % tab.length will be in the range of the tab length

correct

```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff)>>(32-k);   }
```
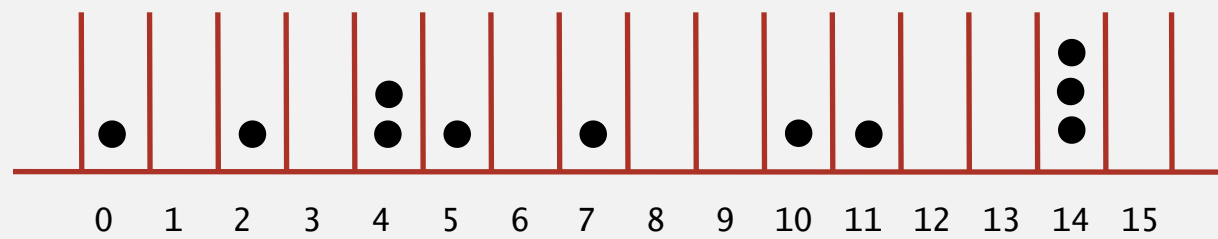
better where M = 2^k

x

x.hashCode()

hash(x)

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 ≥ 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.
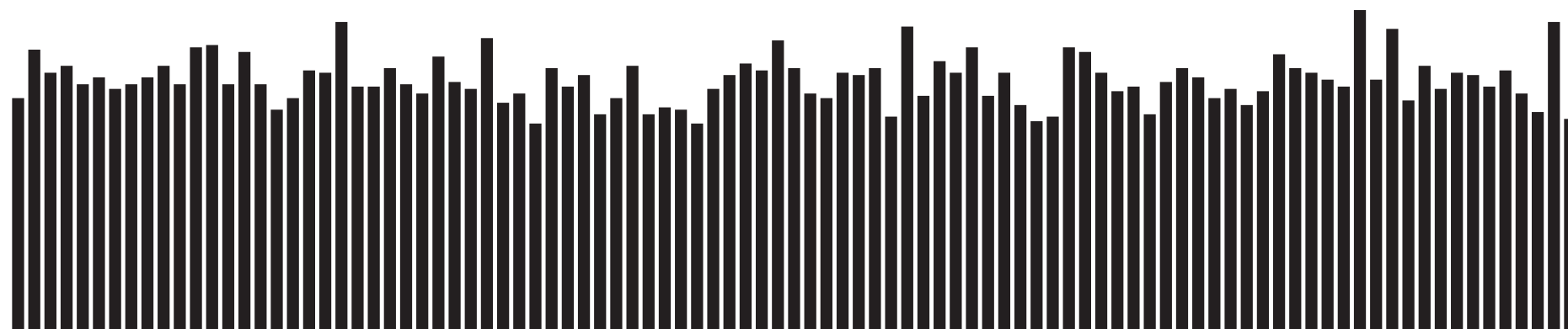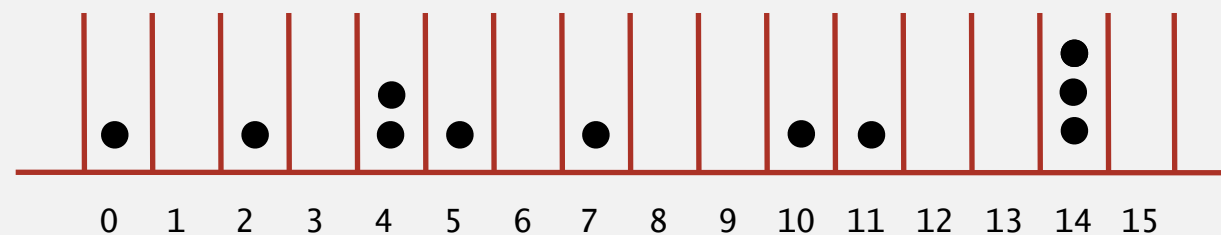
Hash value frequencies for words in Tale of Two Cities (M = 97)

Java's `String` data uniformly distribute the keys of Tale of Two Cities

3.4 HASH TABLES

- ▸ *hash functions*
- ▸ **separate chaining**
- ▸ *linear probing*
- ▸ *context*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

Collisions

Collision. Two distinct keys hashing to same index.
- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing ⇒ collisions are evenly distributed.

```
hash("it") = 3

hash("times") = 3
```

```
0
1
2
3   "it"
4
5
```

??
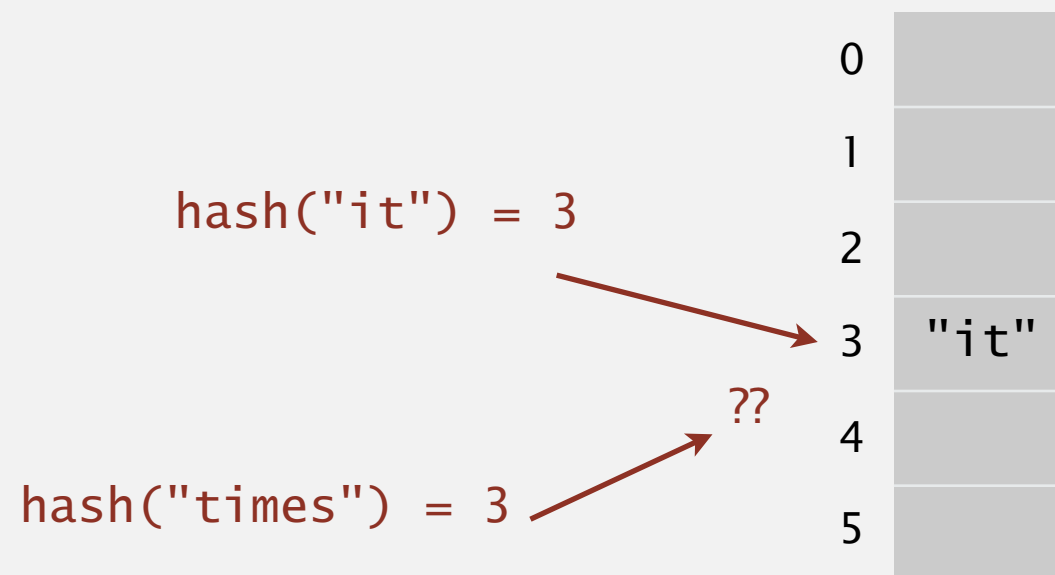
Challenge. Deal with collisions efficiently.

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 ith chain (if not already there).
- Search: need to search only ith chain.

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12

st[]

0 → A 8 → E 12

1 → null

2 → X 7 → S 0

3 → L 11 → P 10

4 → M 9 → H 5 → C 4 → R 3
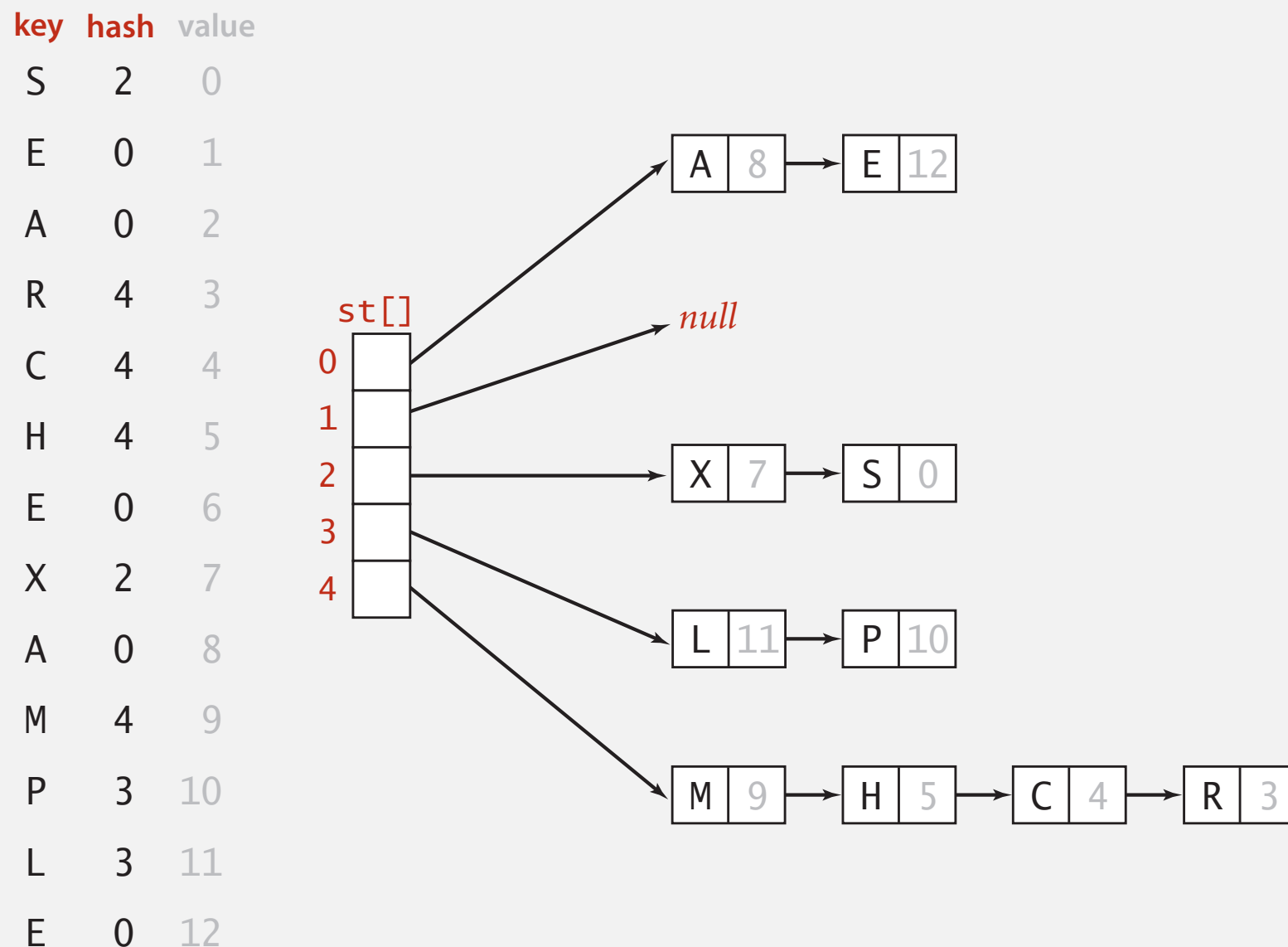
Separate-chaining symbol table: Java implementation

```
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 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]);
   }

}
```
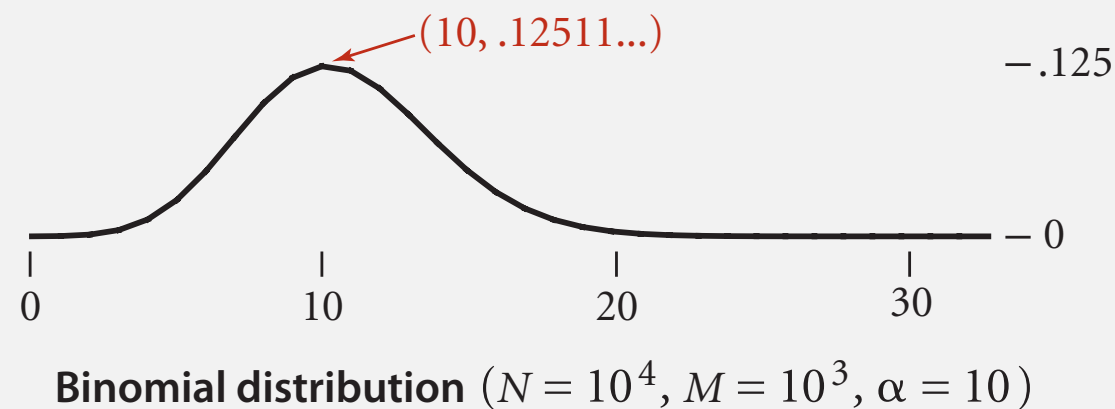
Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N/M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.

$(10, .12511...)$

$-.125$

-0

0	10	20	30

Binomial distribution $(N = 10^4, M = 10^3, \alpha = 10)$

equals() and hashCode()

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

M times faster than sequential search

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N/4 \Rightarrow$ constant-time ops.

Resizing in a separate-chaining hash table

Goal. Average length of list N/M = constant.

- Double size of array M when $N/M \geq 8$.

- Halve size of array M when $N/M \leq 2$.

- Need to rehash all keys when resizing. ⟵ `x.hashCode()` does not change but `hash(x)` can change

before resizing
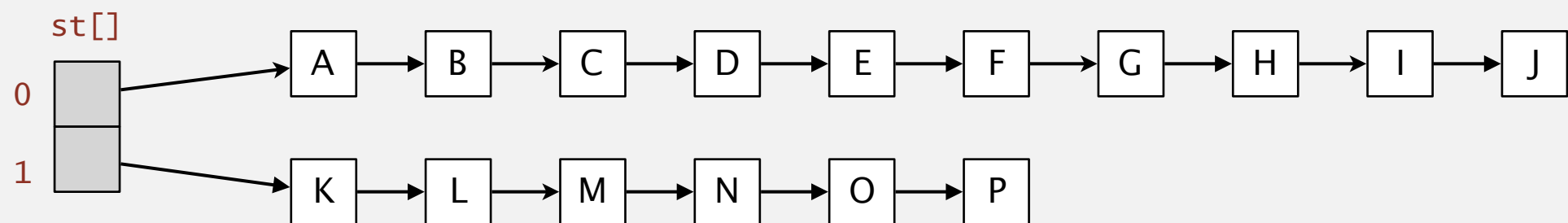
after resizing
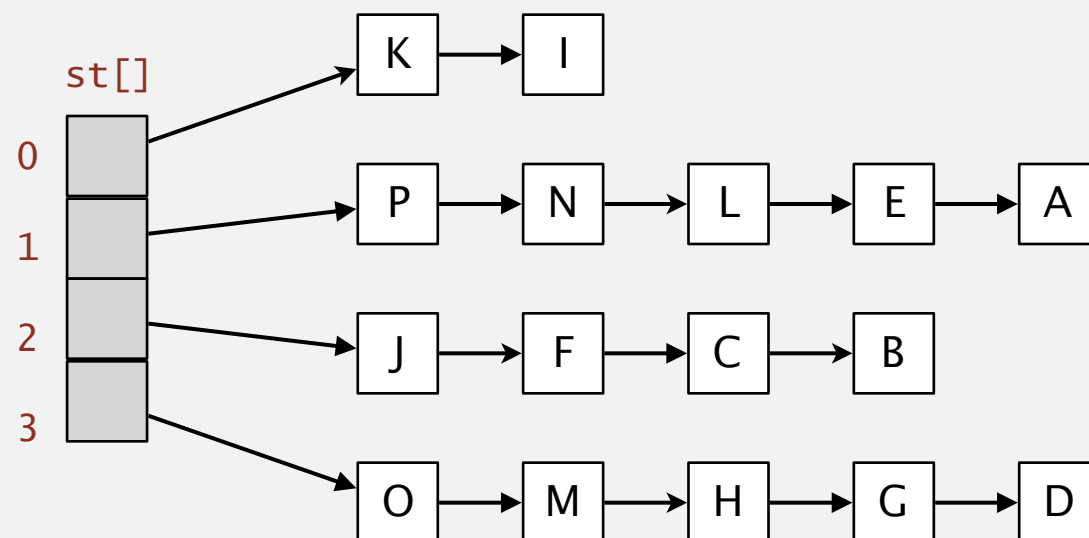
Deletion in a separate-chaining hash table

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

A. Easy: need only consider chain containing key.

before deleting C
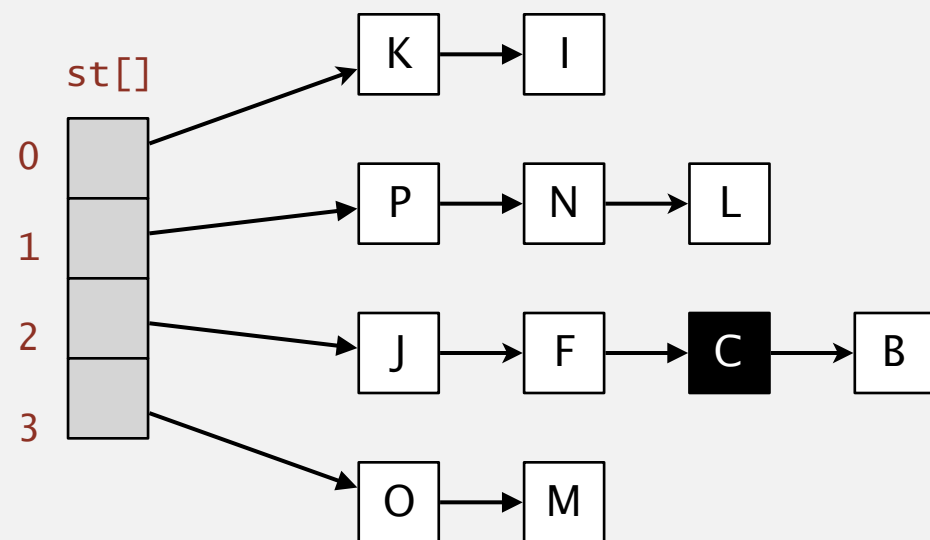
after deleting C

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		equals()
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	compareTo()
red–black BST	2 lg N	2 lg N	2 lg N	1.0 lg N	1.0 lg N	1.0 lg N	✔	compareTo()
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()

* under uniform hashing assumption

3.4 HASH TABLES

- ▸ *hash functions*
- ▸ *separate chaining*
- ▸ **linear probing**
- ▸ *context*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.

st[0]	jocularly
st[1]	*null*
st[2]	listen
st[3]	suburban
⋮	*null*
st[30000]	browsing
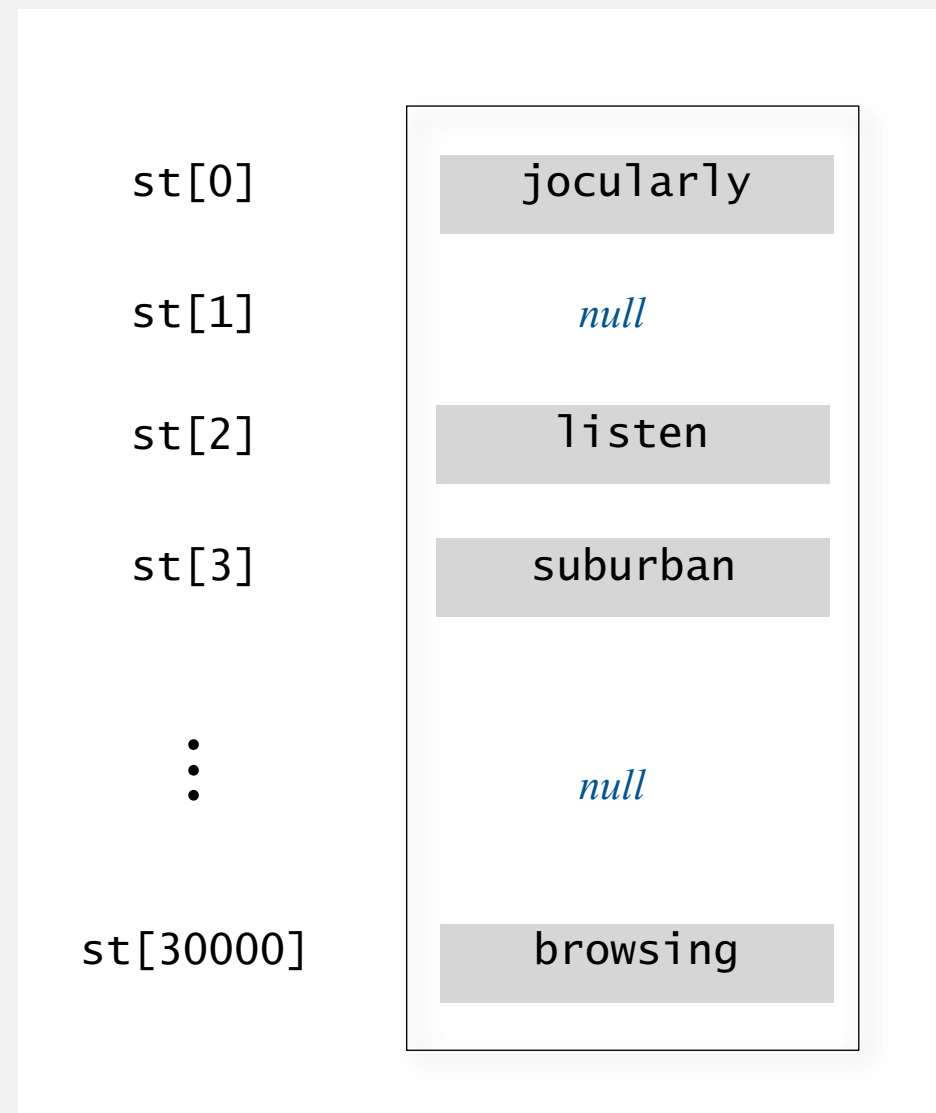
linear probing (M = 30001, N = 15000)

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`, etc.

linear–probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

`M = 16`

Linear-probing hash table demo

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

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

search K
hash(K) = 5

	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

K

search miss
(return null)

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`.

	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 symbol table: Java 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 */   }

    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

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)  { /* previous 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;
   }

}
```
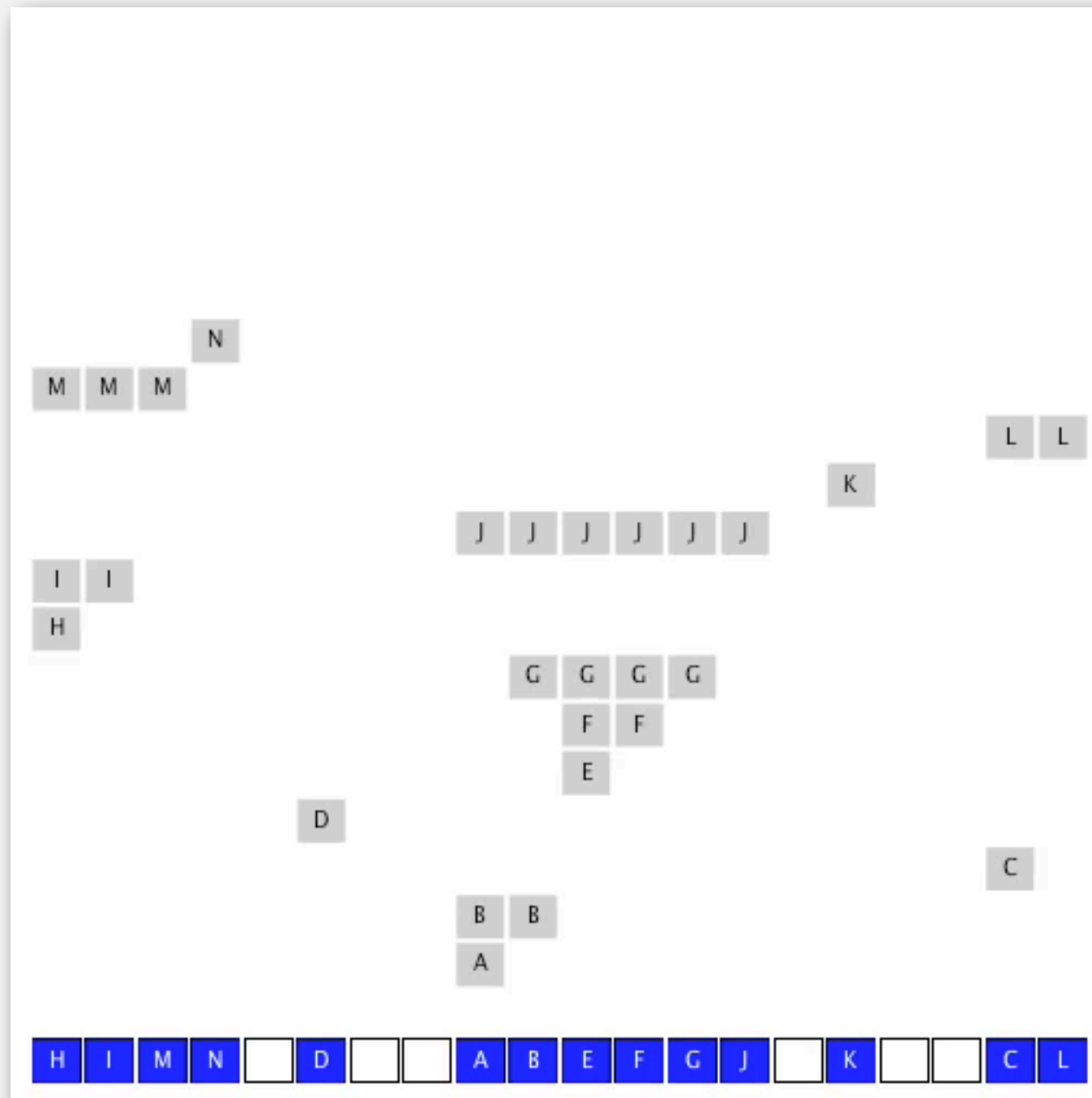
Clustering

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?
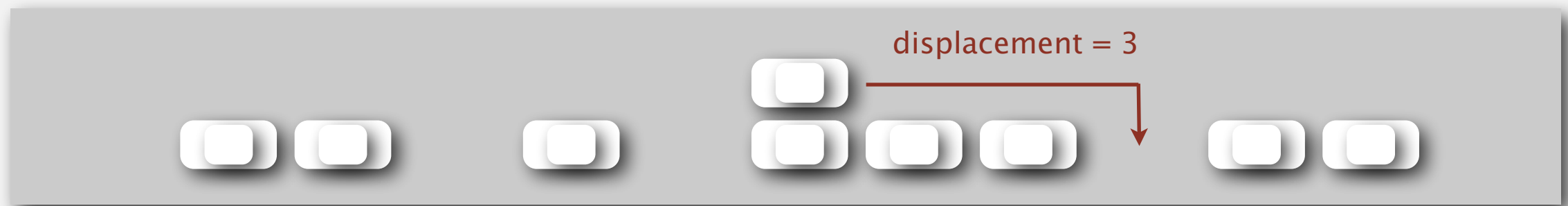
displacement = 3

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 # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

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

search hit   **search miss / insert**

Pf.
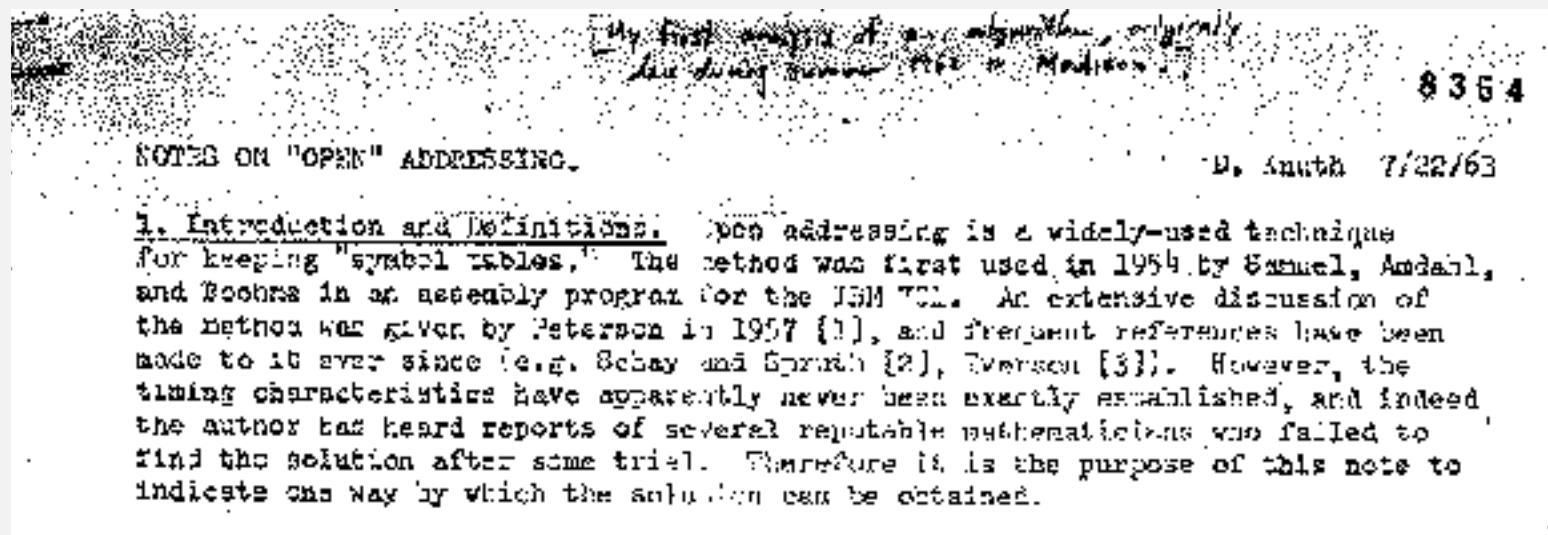
Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim \frac{1}{2}$. $\longleftarrow$ # 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 $N/M \leq$ ½.
- Double size of array M when $N/M \geq$ ½.
- Halve   size of array M when $N/M \leq$ ⅛.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

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

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

doesn't work, e.g., if hash(H) = 4

after deleting S ?

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

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		equals()
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	compareTo()
red–black BST	2 lg N	2 lg N	2 lg N	1.0 lg N	1.0 lg N	1.0 lg N	✔	compareTo()
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()

* under uniform hashing assumption

3.4 HASH TABLES

▸ *hash functions*

▸ *separate chaining*

▸ *linear probing*

▸ **context**

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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.

st[]

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
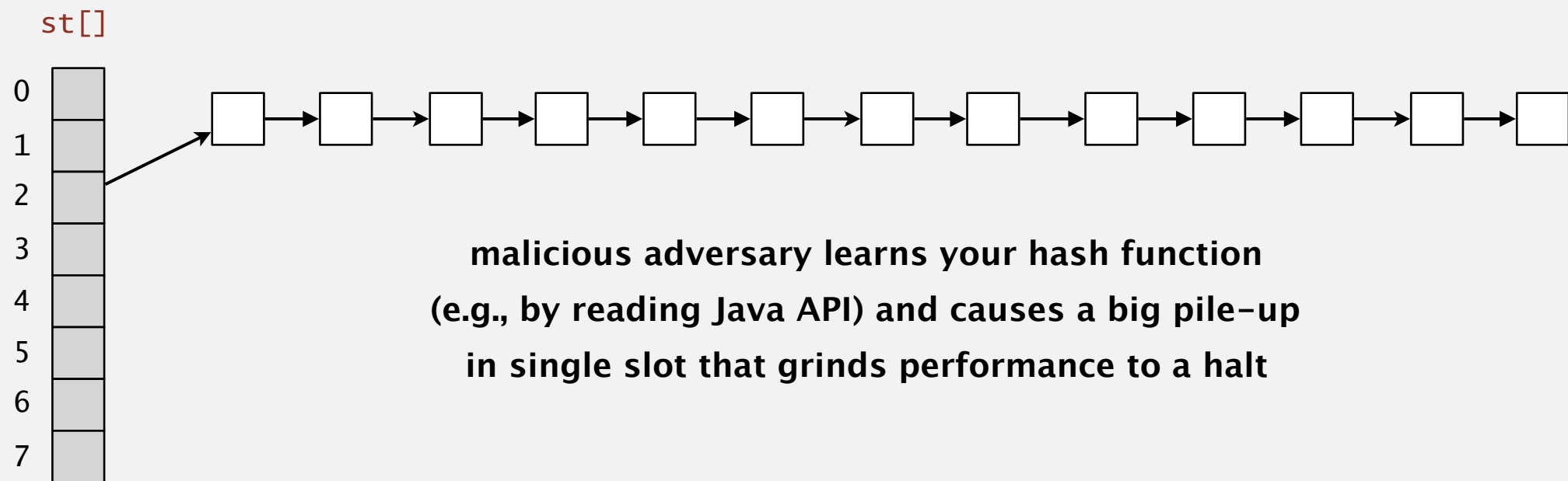
Real-world exploits. [Crosby-Wallach 2003]

• 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.

War story: algorithmic complexity attacks

A Java bug report.

Jan Lieskovsky   2011-11-01 10:13:47 EDT   **Description**

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:
   http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

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. Digital fingerprint, message digest, storing passwords.
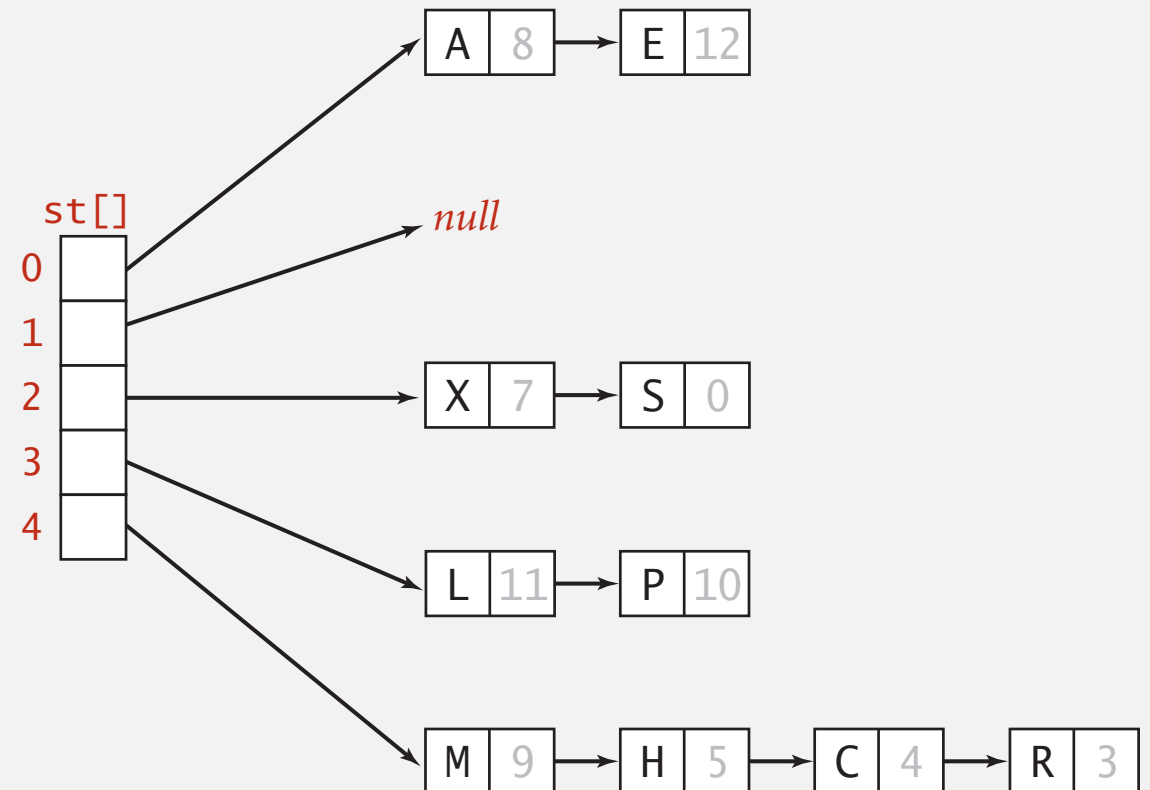
Caveat. Too expensive for use in ST implementations.

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 $\log \log N$.

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`.