



<http://algs4.cs.princeton.edu>

## 3.4 HASH TABLES

---

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

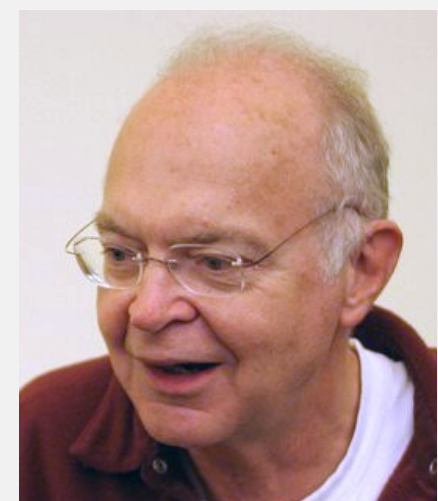
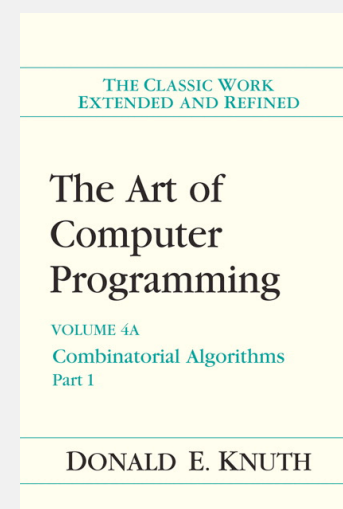
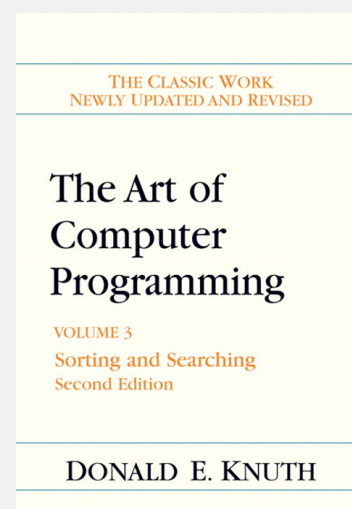
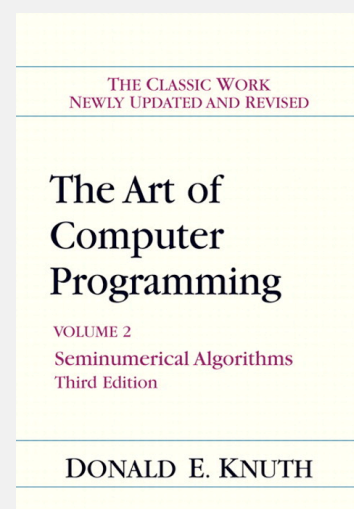
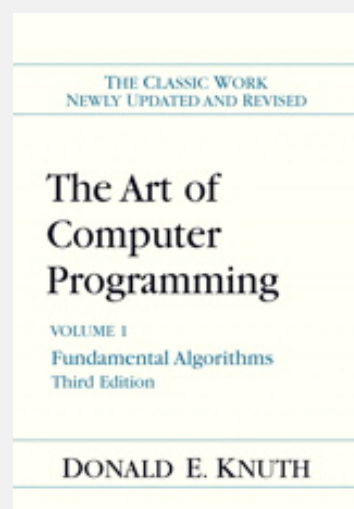
# Premature optimization

---

*“ 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

---

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
<b>sequential search</b> (unordered list)	$N$	$N$	$N$	$N$	$N$	$N$		<code>equals()</code>
<b>binary search</b> (ordered array)	$\log N$	$N$	$N$	$\log N$	$N$	$N$	✓	<code>compareTo()</code>
<b>BST</b>	$N$	$N$	$N$	$\log N$	$\log N$	$\sqrt{N}$	✓	<code>compareTo()</code>
<b>red-black BST</b>	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>

Q. Can we do better?

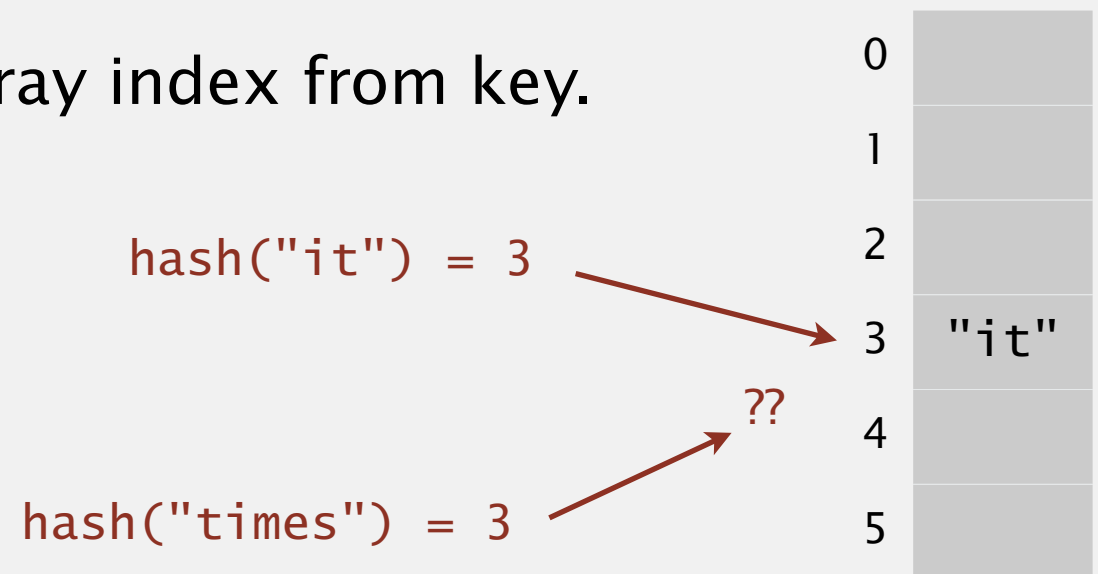
A. Yes, but with different access to the data.

# 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.



## 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).



<http://algs4.cs.princeton.edu>

## 3.4 HASH TABLES

---

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

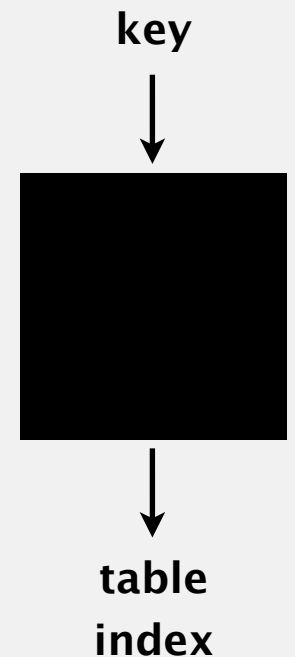
# Computing the hash function

---

**Idealistic goal.** Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,  
still problematic in practical applications



**Ex. Social Security numbers.**

- Bad: first three digits. ← 573 = California, 574 = Alaska  
(assigned in chronological order within geographic region)
- Better: last three digits.

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

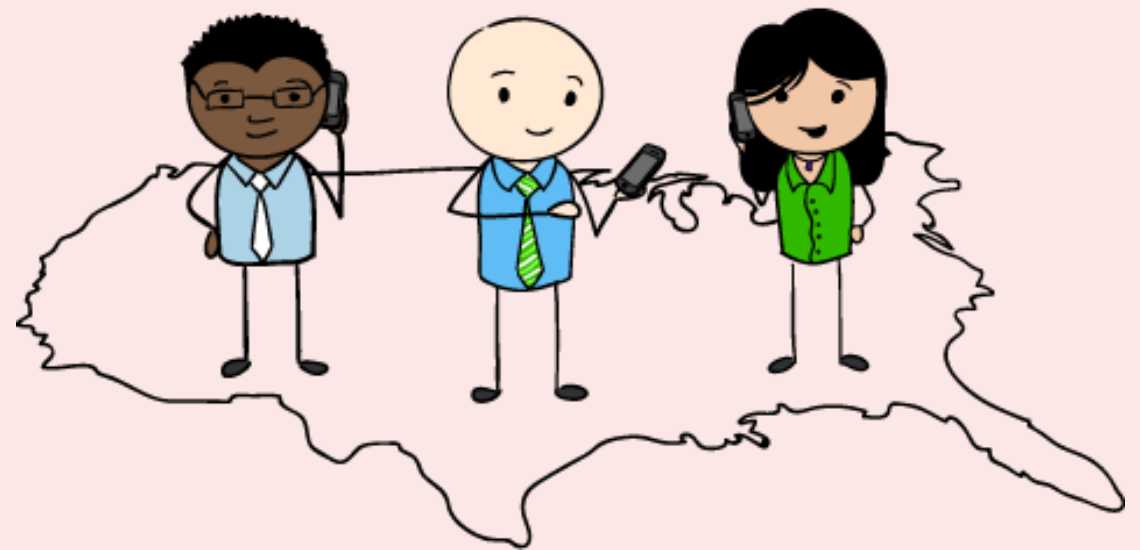
# Hash tables: quiz 1

---

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.*

(609) 867-5309



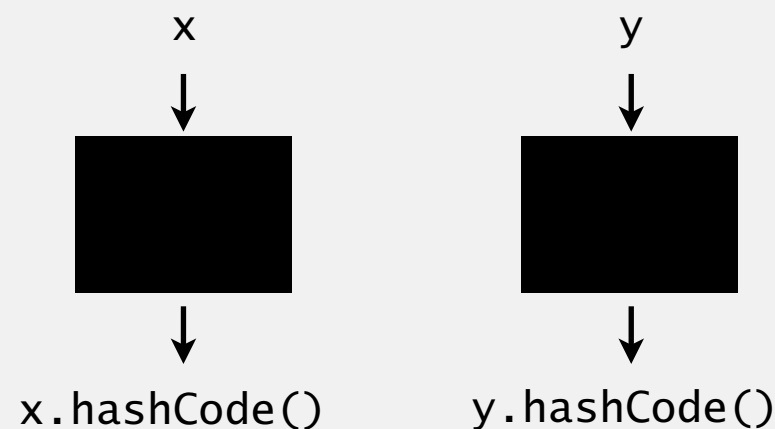
# Java's hash code conventions

---

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

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:

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

```
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;
    }
}
```

Java library implementation

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

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

String s = "call";

s.hashCode();  $\longleftarrow 3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$   
 $= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

# Implementing hash code: strings

---

## Performance optimization.

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

```
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;
    }
}
```

← cache of hash code

← return cached value

← store cache of hash code

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;
    }
}
```

nonzero constant

for reference types,  
use hashCode()

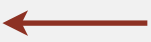
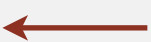
for primitive types,  
use hashCode()  
of wrapper type

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`, use 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 above works reasonably well; used in Java libraries.

**In theory.** Keys are bitstring; "universal" family of hash functions exist.

 awkward in Java since only  
one (deterministic) `hashCode()`

**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.

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

B.

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

C.

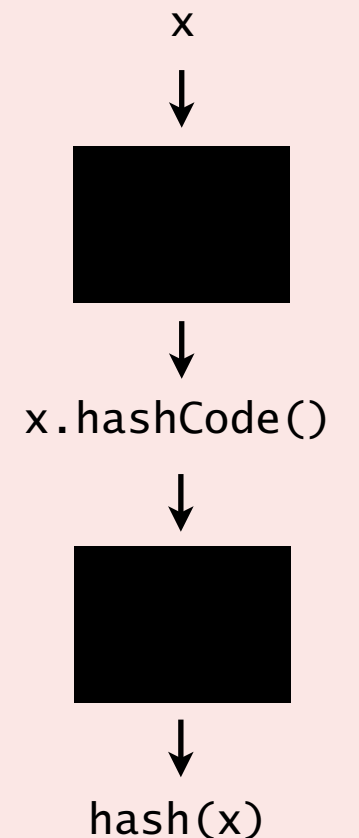
Both A and B.

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).

typically a prime or power of 2

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

**bug**

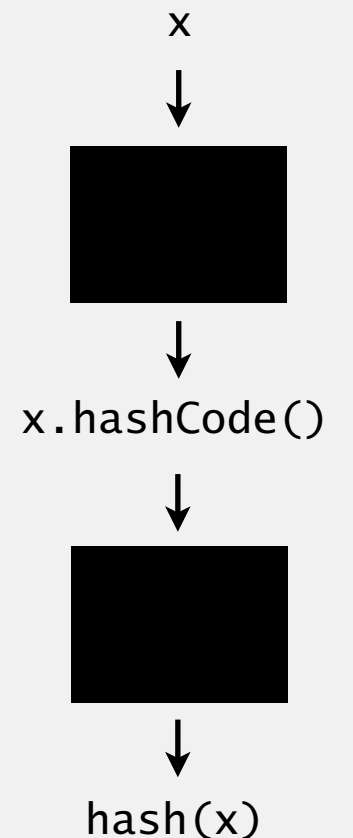
```
private int hash(Key key)
{ return Math.abs(key.hashCode()) % M; }
```

**1-in-a-billion bug**

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

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

**correct**

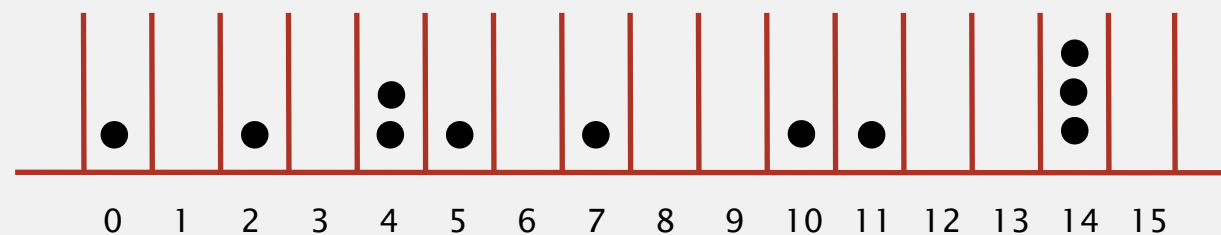


# Uniform hashing assumption

---

**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  $\sim \ln M / \ln \ln M$  balls.

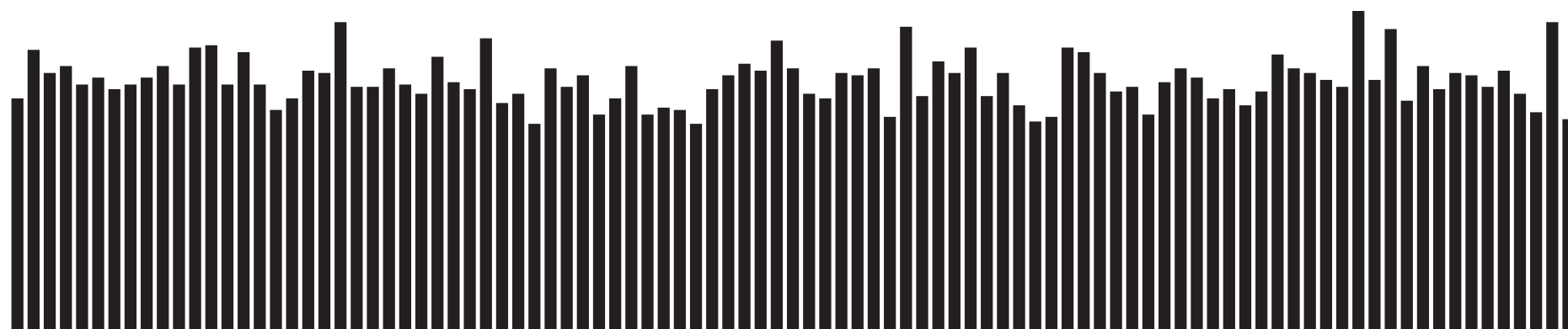
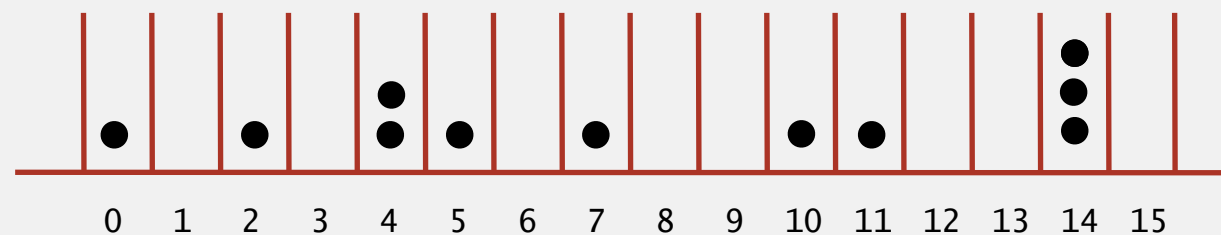


# Uniform hashing assumption

---

**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



<http://algs4.cs.princeton.edu>

## 3.4 HASH TABLES

---

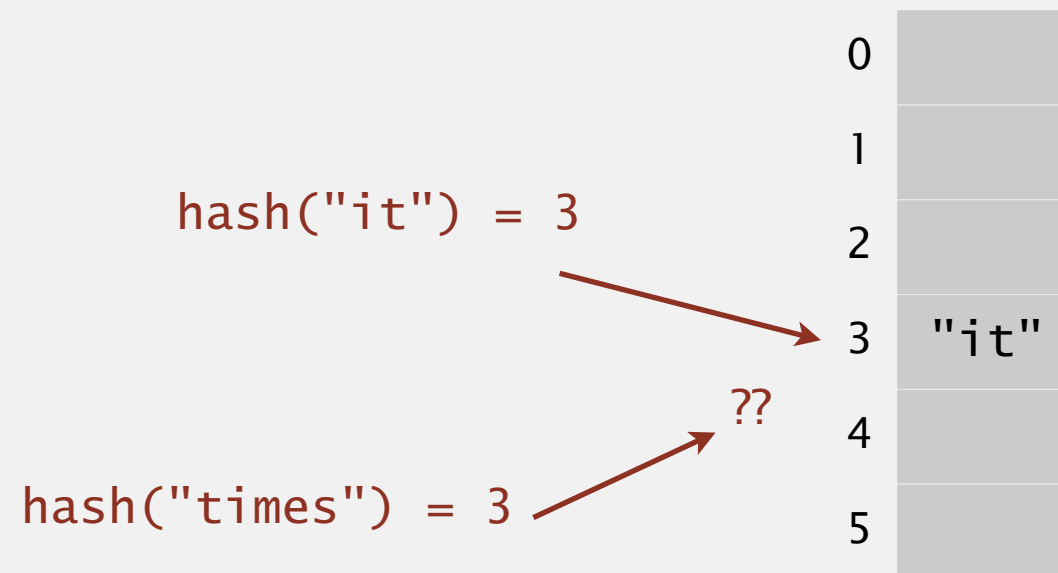
- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ *context*

# Collisions

---

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

- Birthday problem  $\Rightarrow$  can't avoid collisions. ← unless you have a ridiculous (quadratic) amount of memory
- Coupon collector  $\Rightarrow$  not too much wasted space.
- Load balancing  $\Rightarrow$  no index gets too many collisions.



**Challenge.** Deal with collisions efficiently.

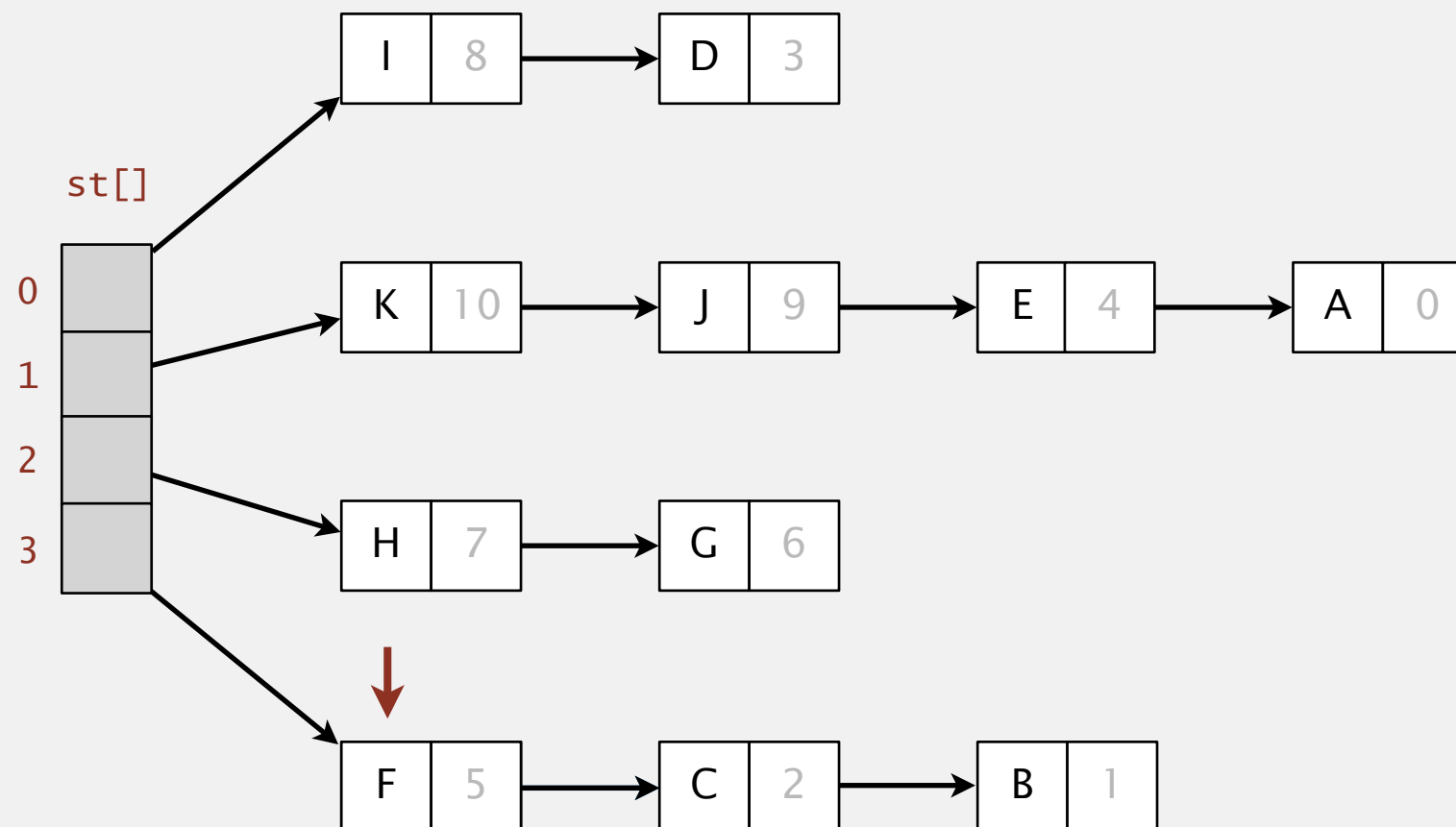
# Separate-chaining symbol table

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 in chain).
- Search: sequential search in  $i^{\text{th}}$  chain.

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

separate-chaining hash table ( $M = 4$ )



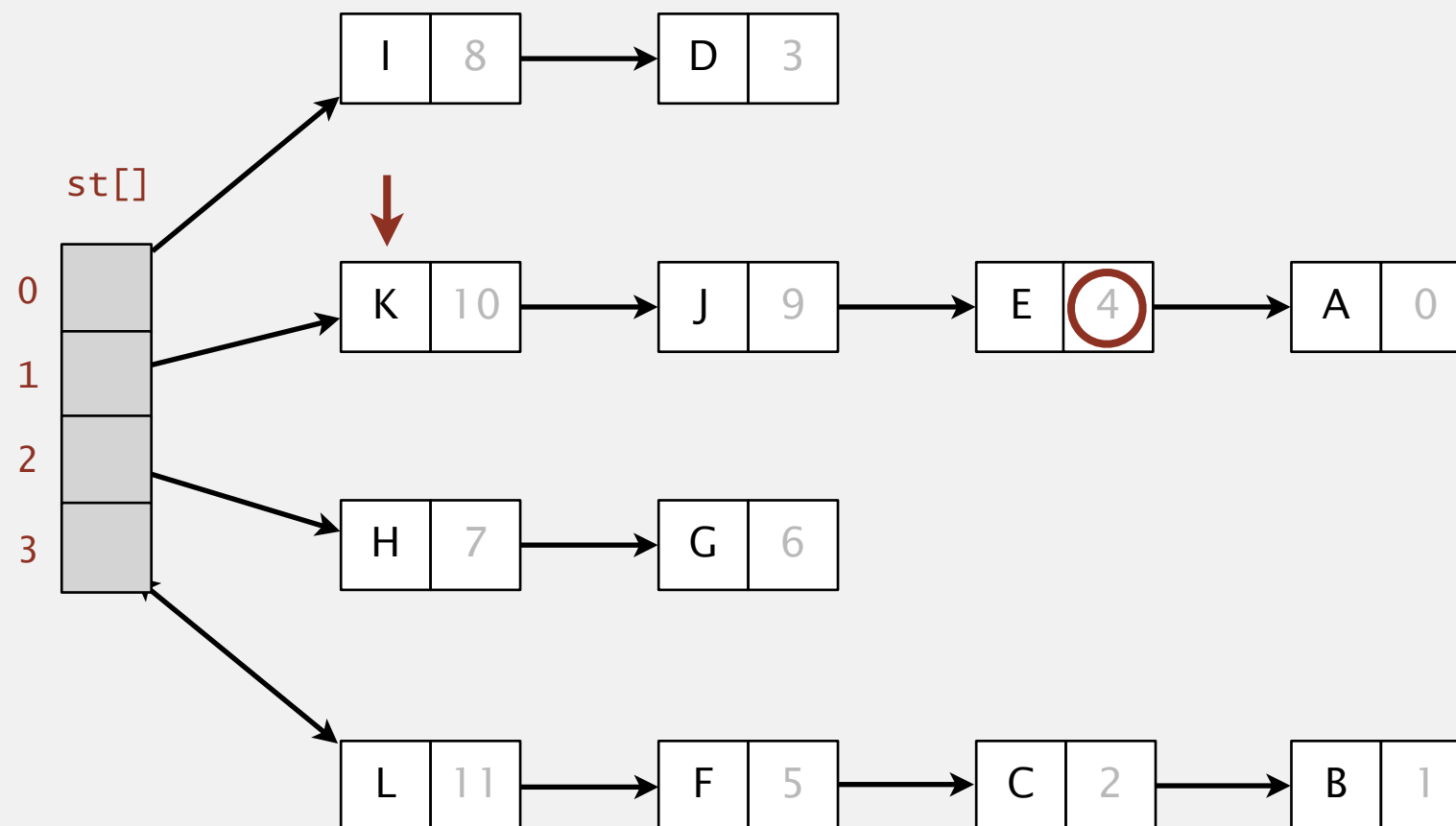
# Separate-chaining symbol table

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 in chain).
- Search: sequential search in  $i^{\text{th}}$  chain.

get(E)  
hash(E) = 1

separate-chaining hash table ( $M = 4$ )



# 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

---

```
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;
        ...
    }

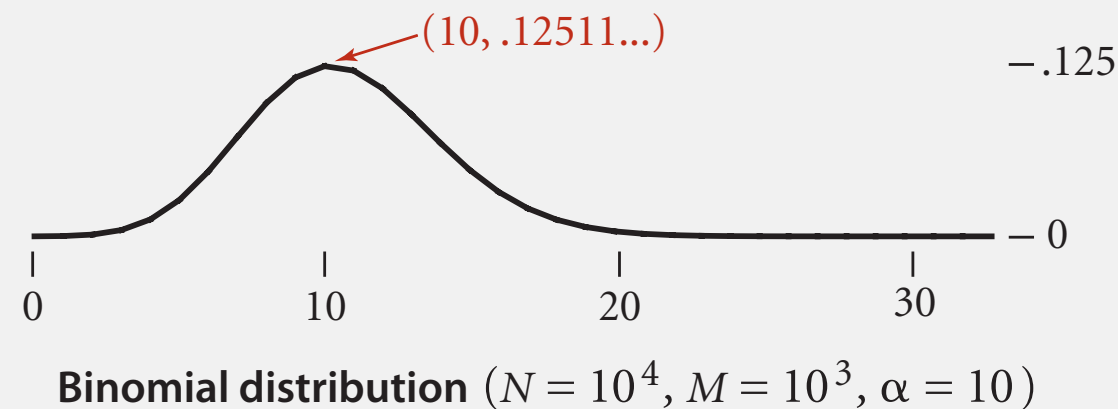
    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    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.



**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.

$\uparrow$   
M times faster than  
sequential search

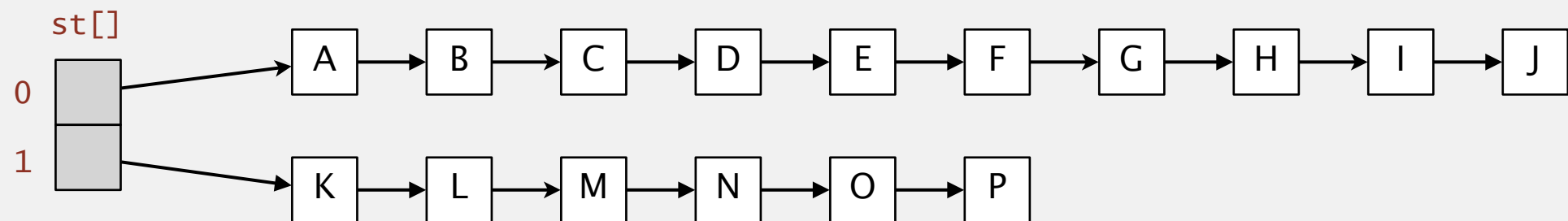


# Resizing in a separate-chaining hash table

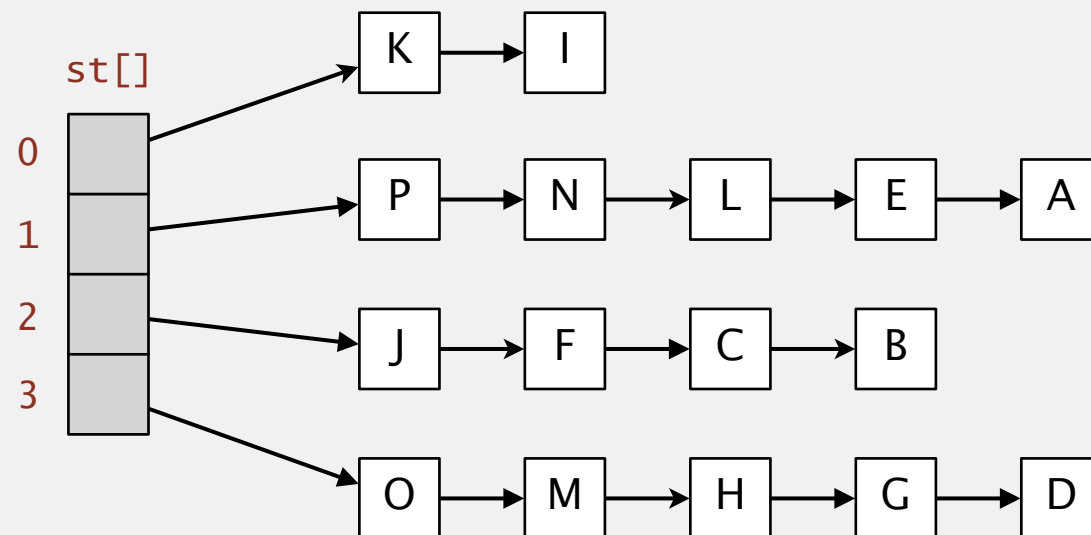
**Goal.** Average length of list  $N / M = \text{constant}$ .

- 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. ←  $x.\text{hashCode}()$  does not change;  
but  $\text{hash}(x)$  can change

before resizing ( $N/M = 8$ )



after resizing ( $N/M = 4$ )



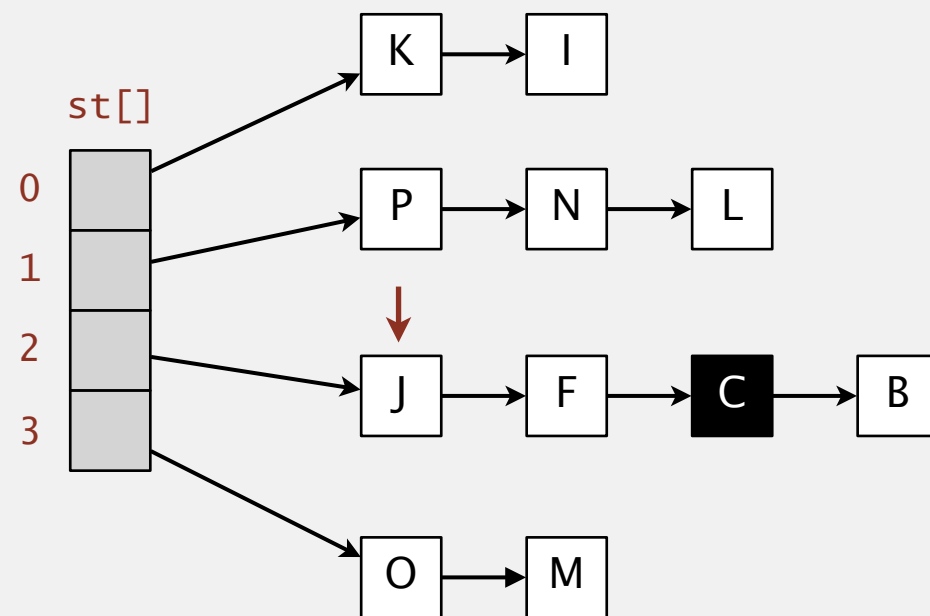
# Deletion in a separate-chaining hash table

---

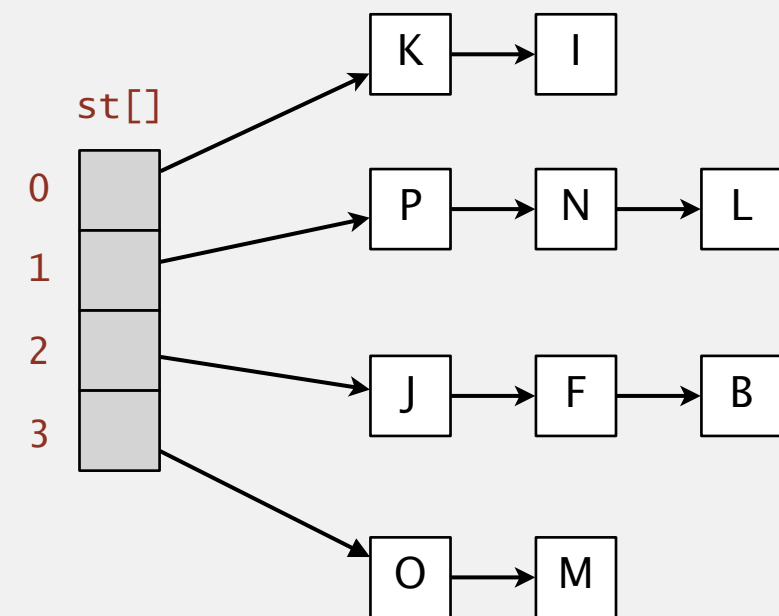
**Q.** How to delete a key (and its associated value)?

**A.** Easy: need to consider only chain containing key.

before deleting C



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)	$N$	$N$	$N$	$N$	$N$	$N$		equals()
binary search (ordered array)	$\log N$	$N$	$N$	$\log N$	$N$	$N$	✓	compareTo()
BST	$N$	$N$	$N$	$\log N$	$\log N$	$\sqrt{N}$	✓	compareTo()
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	compareTo()
separate chaining	$N$	$N$	$N$	$1 *$	$1 *$	$1 *$		equals() hashCode()

\* under uniform hashing assumption



<http://algs4.cs.princeton.edu>

## 3.4 HASH TABLES

---

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

# Collision resolution: open addressing

---

**Open addressing.** [Amdahl-Boehme-Rochester-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)**

	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
								K								
								14								
vals[]	11	10			9	5		6	12		13				4	8

# 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
keys[]	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];
```

← array doubling and  
halving code omitted

```
    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;
    }
```

← sequential search  
in chain i

```
}
```

# 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 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

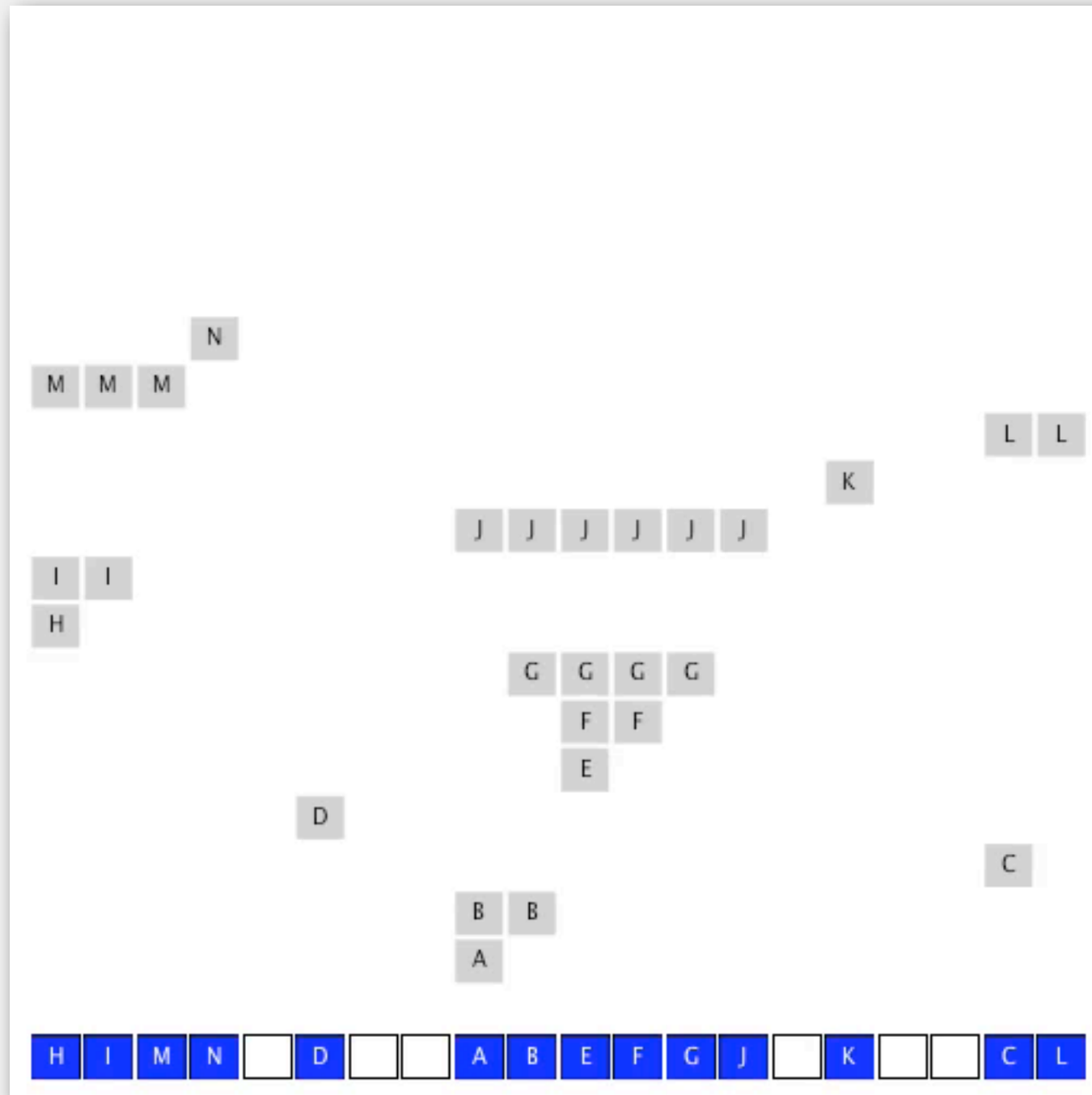


# Clustering

---

**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.

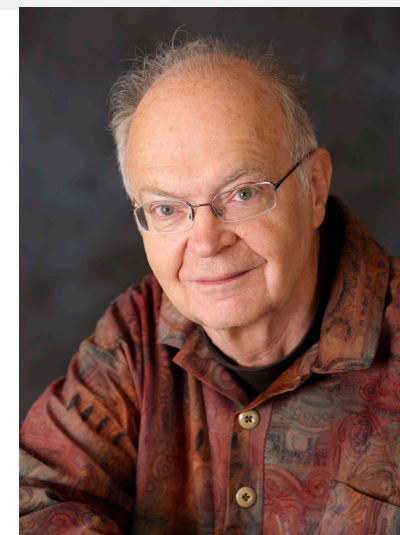
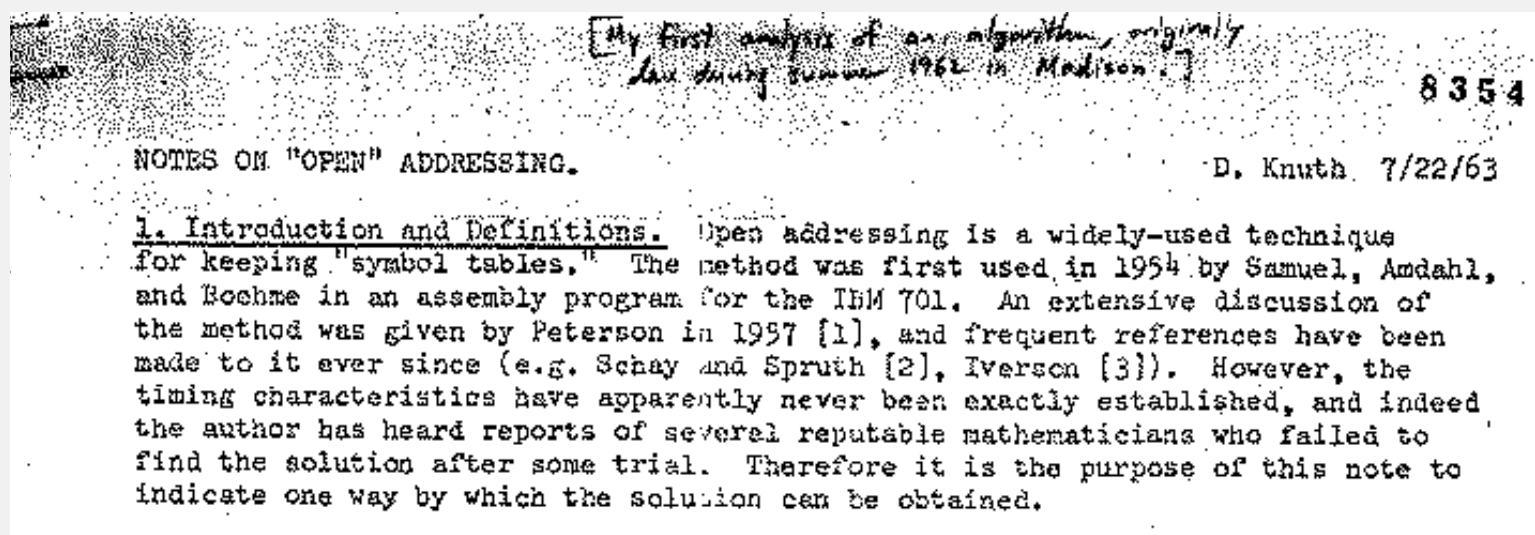
# Analysis of linear probing

**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:

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

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

# 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

	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  $\text{hash}(H) = 4$

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)	$N$	$N$	$N$	$N$	$N$	$N$		equals()
binary search (ordered array)	$\log N$	$N$	$N$	$\log N$	$N$	$N$	✓	compareTo()
BST	$N$	$N$	$N$	$\log N$	$\log N$	$\sqrt{N}$	✓	compareTo()
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	compareTo()
separate chaining	$N$	$N$	$N$	1 *	1 *	1 *		equals() hashCode()
linear probing	$N$	$N$	$N$	1 *	1 *	1 *		equals() hashCode()

\* 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.



<http://algs4.cs.princeton.edu>

## 3.4 HASH TABLES

---

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



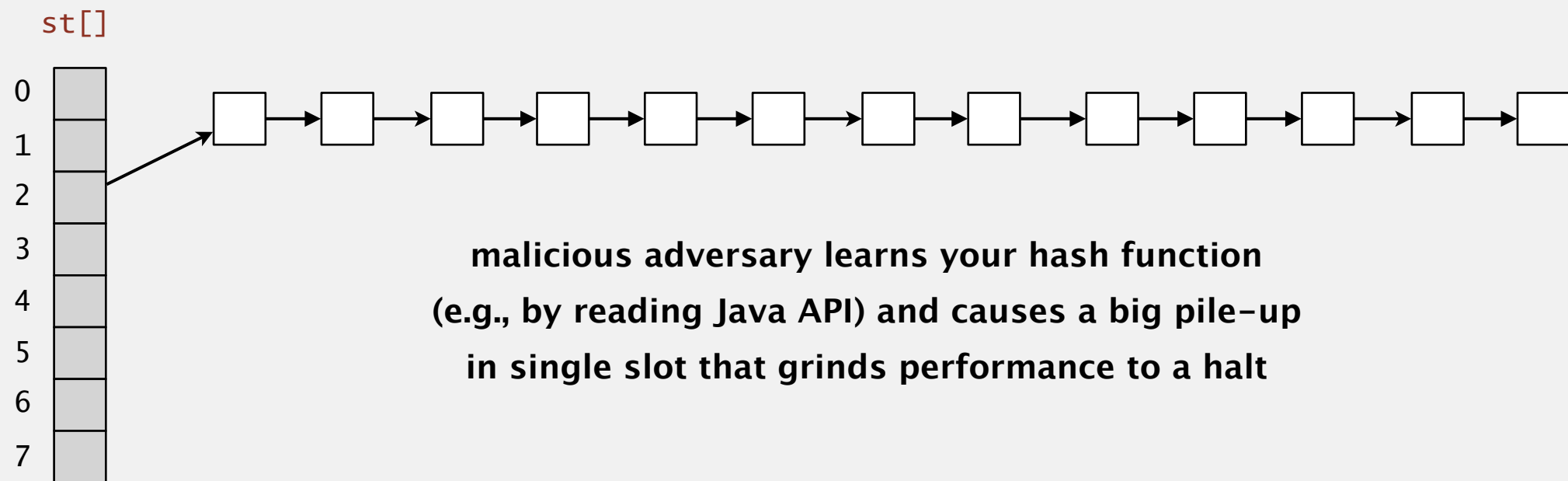
# War story: algorithmic complexity attacks

---

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.

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

# 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.

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
"BBaBBBBB"	-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.** Crypto, message digests, passwords, Bitcoin, ....

**Caveat.** Too expensive for use in ST implementations.

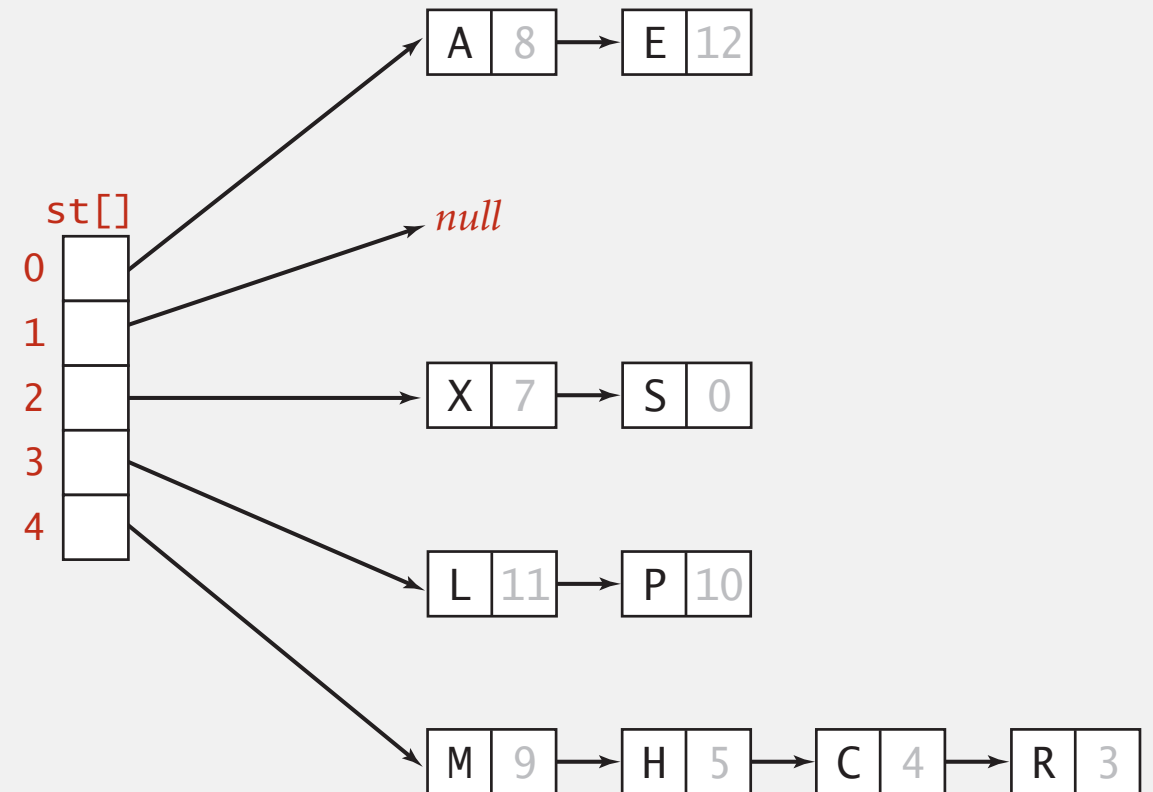
# Separate chaining vs. linear probing

## Separate chaining.

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

## Linear probing.

- Less wasted space.
- Better cache performance.



	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

# Hashing: variations on the theme

---

Many improved versions have been studied.

## Two-probe hashing. [ separate-chaining variant ]

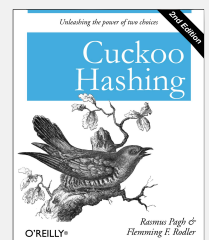
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to  $\sim \lg \ln N$ .

## 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`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

↑  
linear probing

↑  
separate chaining



<http://algs4.cs.princeton.edu>

## 3.5 SYMBOL TABLE APPLICATIONS

---

- ▶ *sets*
- ▶ *dictionary clients*
- ▶ *indexing clients*
- ▶ *sparse vectors*





<http://algs4.cs.princeton.edu>

## 3.5 SYMBOL TABLE APPLICATIONS

---

- ▶ *sets*
- ▶ *dictionary clients*
- ▶ *indexing clients*
- ▶ *sparse vectors*

# Set API

---

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

```
public class SET<Key extends Comparable<Key>>
```

```
    SET()
```

*create an empty set*

```
    void add(Key key)
```

*add the key to the set*

```
    boolean contains(Key key)
```

*is the key in the set?*

```
    void remove(Key key)
```

*remove the key from the set*

```
    int size()
```

*number of keys in the set*

```
    Iterator<Key> iterator()
```

*all keys in the set*

**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.

application	purpose	key	in list
<b>spell checker</b>	identify misspelled words	word	dictionary words
<b>browser</b>	mark visited pages	URL	visited pages
<b>parental controls</b>	block sites	URL	bad sites
<b>chess</b>	detect draw	board	positions
<b>spam filter</b>	eliminate spam	IP address	spam addresses
<b>credit cards</b>	check for stolen cards	number	stolen cards

# 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.

```
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);
        }
    }
}
```

← create empty set of strings

← read in whitelist

← print words not in list



<http://algs4.cs.princeton.edu>

## 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

% java LookupCSV ip.csv 1 0

128.112.128.15

www.princeton.edu

999.999.999.99

Not found

domain name is key IP is value

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.

codon is key name is value

```
% 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
CCG,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 LookupCSV classlist.csv 4 1
```

```
eberl
```

```
Ethan
```

```
nwebb
```

```
Natalie
```

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

```
dpan
```

```
P01
```

login is key    first name  
                 is value

login is key    section  
                 is value

```
% more classlist.csv
```

```
13,Berl,Ethan Michael,P01,eberl
```

```
12,Cao,Phillips Minghua,P01,pcao
```

```
11,Cehoud,Christel,P01,ccehoud
```

```
10,Douglas,Malia Morioka,P01,malia
```

```
12,Haddock,Sara Lynn,P01,shaddock
```

```
12,Hantman,Nicole Samantha,P01,nhantman
```

```
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

---

```
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]);
```

← process input file

```
        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);
        }
```

← build symbol table

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

← process lookups  
with standard I/O



<http://algs4.cs.princeton.edu>

## 3.5 SYMBOL TABLE APPLICATIONS

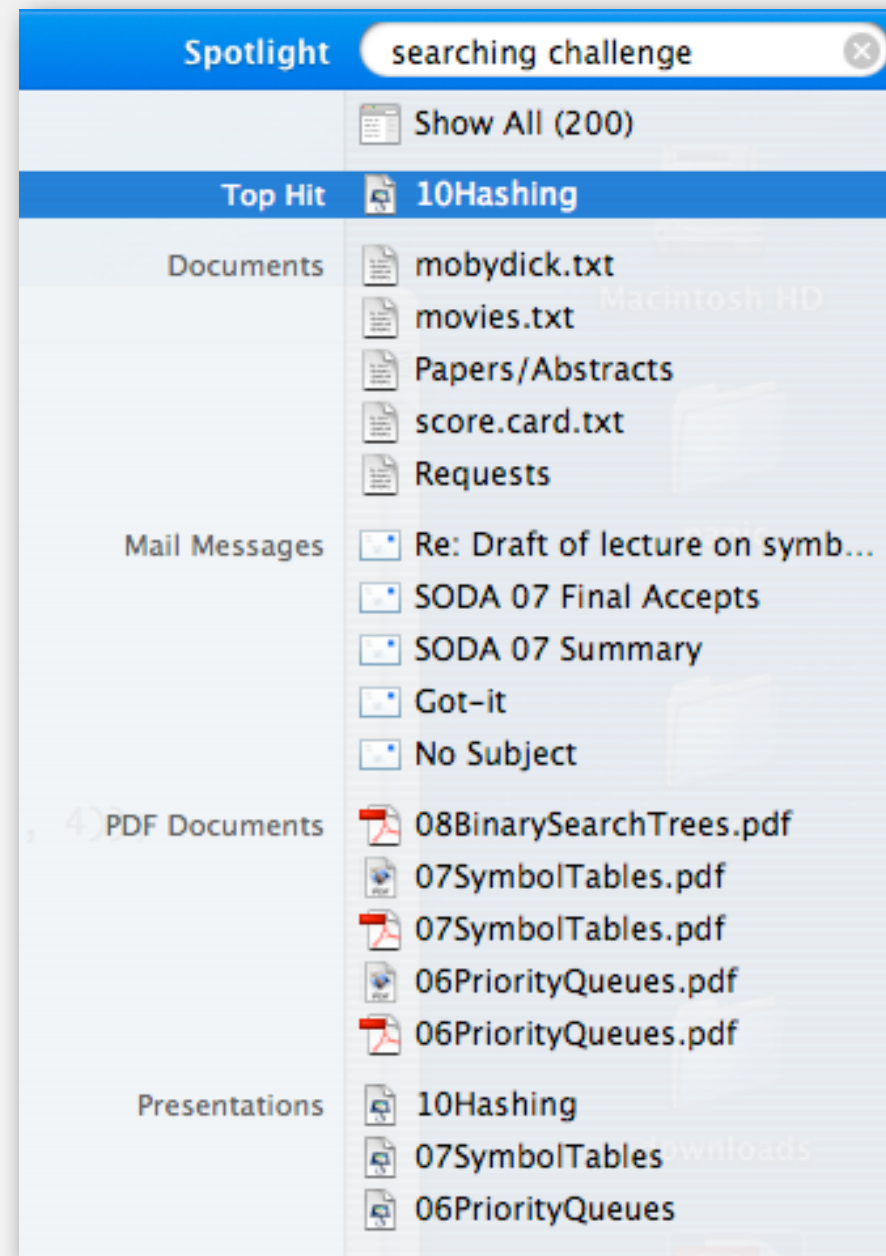
---

- ▶ *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
```

## Searching applications: quiz 1

---

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

- A. SET<ST<File, String>>
- B. SET<ST<String, File>>
- C. ST<File, SET<String>>
- D. ST<String, SET<File>>
- E. *I don't know.*

# File indexing

```
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));
        }
    }
}
```

ST<String, SET<File>> st = new ST<String, SET<File>>();

← symbol table

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);  
}  
}

← list of file names  
from command line

← for each word in file,  
add file to  
corresponding set

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

← process queries

# 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, 165-171
  - first-class, 177-186
  - generic operations, 273
  - index items, 177
  - insert/remove* operations, 138-139
  - modular programming, 135
  - polynomial, 188-192
  - priority queues, 375-376
  - pushdown stack, 138-156
  - stubs, 135
  - symbol table, 497-506
- ADT interfaces
  - array (*myArray*), 274
  - complex number (*Complex*), 181
  - existence table (ET), 663
  - full priority queue (*PQfull*), 397
  - indirect priority queue (*PQi*), 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, 155-157
- Adaptive sort, 268
- Address, 84-85
- Adjacency list, 120-123
  - depth-first search, 251-256
- Adjacency matrix, 120-122
- Ajtai, M., 464
- Algorithm, 4-6, 27-64
  - abstract operations, 10, 31, 34-35
  - analysis of, 6
  - average-/worst-case performance, 35, 60-62
  - big-Oh notation, 44-47
  - binary search, 56-59
  - computational complexity, 62-64
  - efficiency, 6, 30, 32
  - empirical analysis, 30-32, 58
  - exponential-time, 219
  - implementation, 28-30
  - 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, 53-56, 498
  - steps in, 22-23
  - See also* Randomized algorithm
- Amortization approach, 557, 627
- Arithmetic operator, 177-179, 188, 191
- Array, 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, 508, 511-512, 521
- Asymptotic expression, 45-46
- Average deviation, 80-81
- Average-case performance, 35, 60-61
- AVL tree, 583
- B tree, 584, 692-704
  - external/internal pages, 695
  - 4-5-6-7-8 tree, 693-704
  - Markov chain, 701
  - remove*, 701-703
  - search/insert*, 697-701
  - select/sort*, 701
- Balanced tree, 238, 555-598
  - B tree, 584
  - bottom-up, 576, 584-585
  - height-balanced, 583
  - indexed sequential access, 690-692
  - performance, 575-576, 581-582, 595-598
  - randomized, 559-564
  - red-black, 577-585
  - skip lists, 587-594
  - splay, 566-571

# Concordance

---

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

```
% 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.



# Concordance

---

```
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>>();
        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]
        }
    }
}
```

← read text and  
build index

← process queries  
and print  
concordances



<http://algs4.cs.princeton.edu>

## 3.5 SYMBOL TABLE APPLICATIONS

---

- ▶ *sets*
- ▶ *dictionary clients*
- ▶ *indexing clients*
- ▶ *sparse vectors*

# Matrix-vector multiplication (standard implementation)

---

$$\begin{array}{c} \text{a}[\text{ }][\text{ }] \\ \left[ \begin{array}{ccccc} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{array} \right] \end{array} \begin{array}{c} \text{x}[\text{ }] \\ \left[ \begin{array}{c} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{array} \right] \end{array} = \begin{array}{c} \text{b}[\text{ }] \\ \left[ \begin{array}{c} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{array} \right] \end{array}$$

```
...
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;
}
```

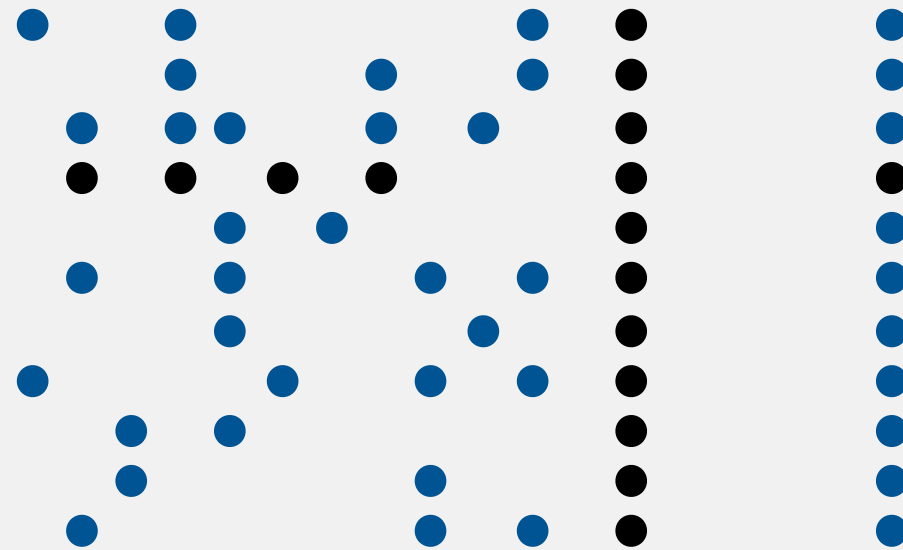
nested loops  
( $N^2$  running time)

# Sparse matrix-vector multiplication

---

**Problem.** Sparse matrix-vector multiplication.

**Assumptions.** Matrix dimension is 10,000; average nonzeros per row  $\sim 10$ .



$$A * x = b$$

# Vector representations

---

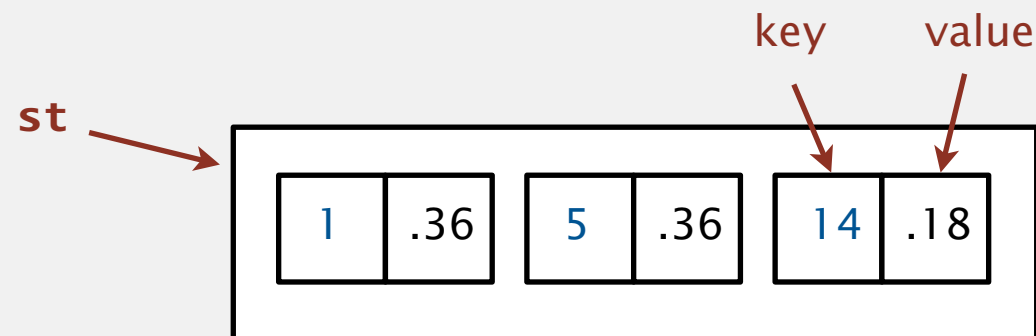
## 1d array (standard) representation.

- 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	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

```
public class SparseVector  
{
```

```
    private HashST<Integer, Double> v;
```

← HashST because order not important

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

← empty ST represents all 0s vector

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

←  $a[i] = \text{value}$

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

← return  $a[i]$

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

← iterate through indices of  
nonzero entries

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

← dot product is constant  
time for sparse vectors

```
}
```

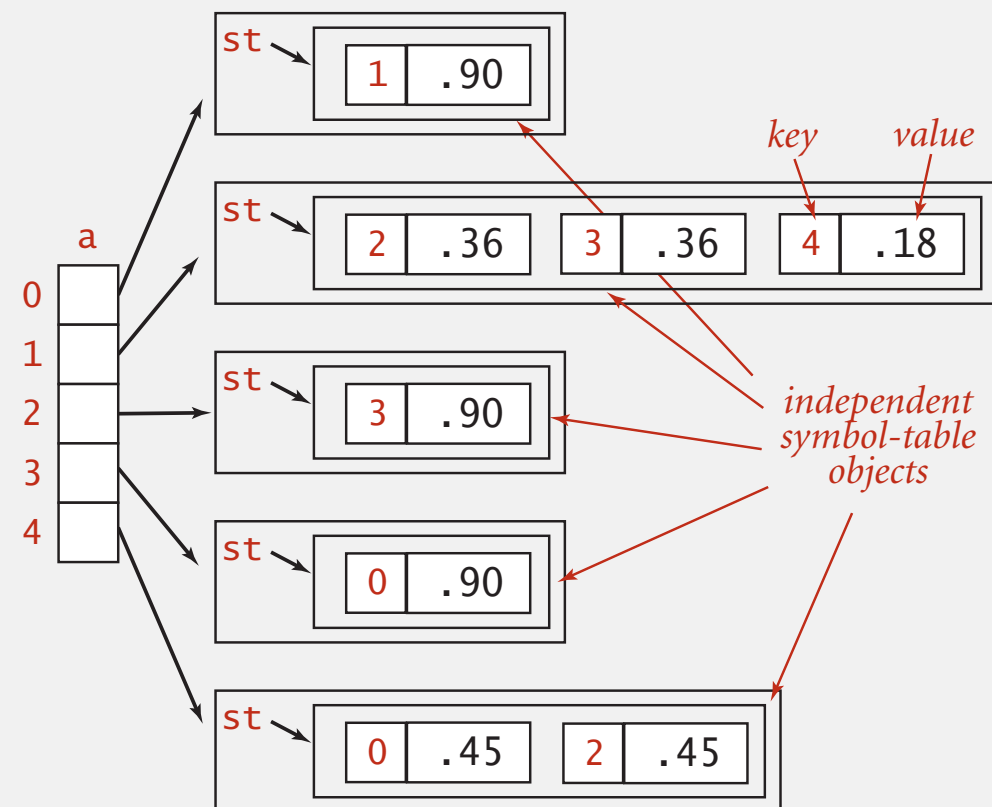
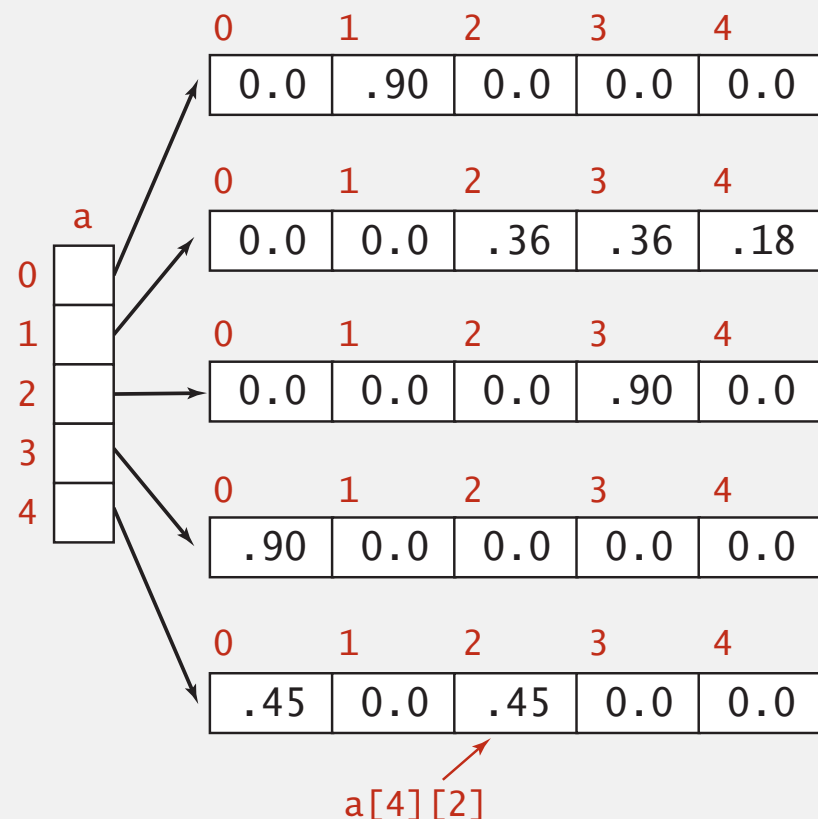
# Matrix representations

**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

---

$$\begin{array}{c} \text{a}[][] \\ \left[ \begin{array}{ccccc} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{array} \right] \end{array} \begin{array}{c} \text{x}[] \\ \left[ \begin{array}{c} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{array} \right] \end{array} = \begin{array}{c} \text{b}[] \\ \left[ \begin{array}{c} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{array} \right] \end{array}$$

```
..  
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