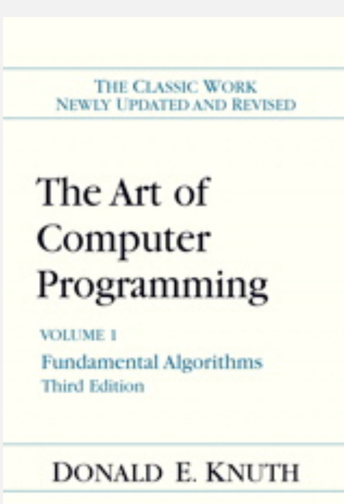
http://algs4.cs.princeton.edu

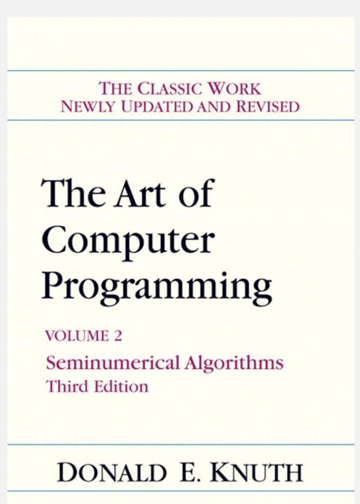
3.4 HASH TABLES

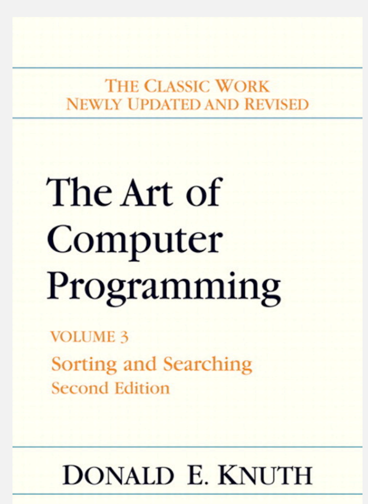
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

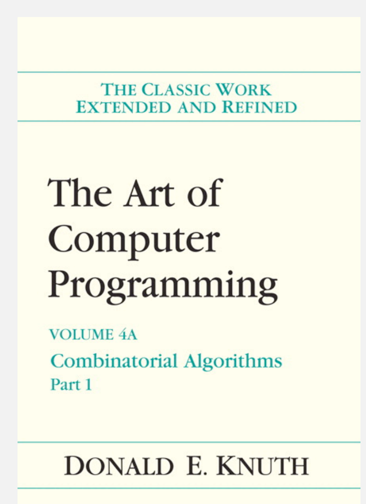
Premature optimization

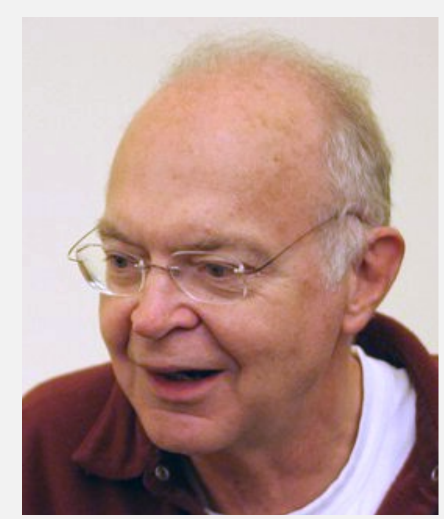
“ Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. ” – *Donald Knuth*











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()`

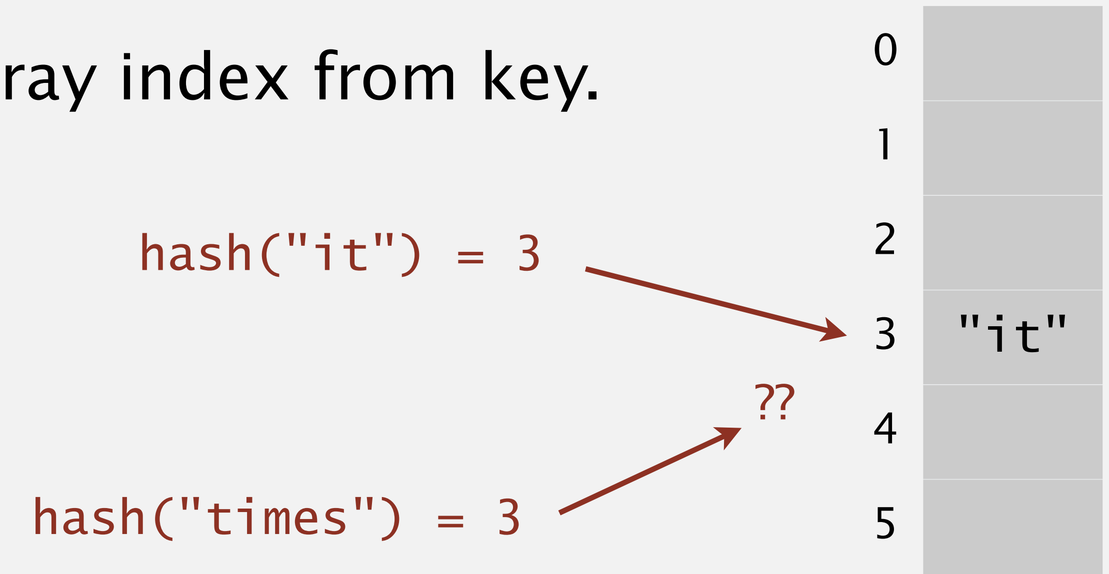
Q. Can we do better?

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

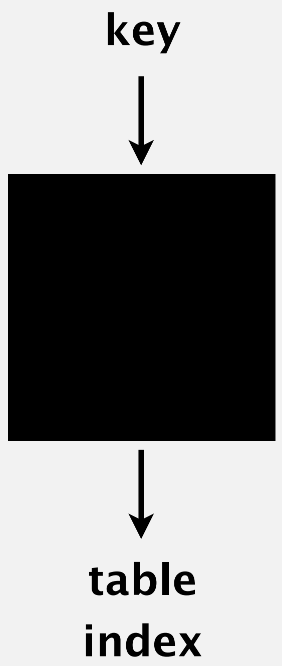
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

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 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)

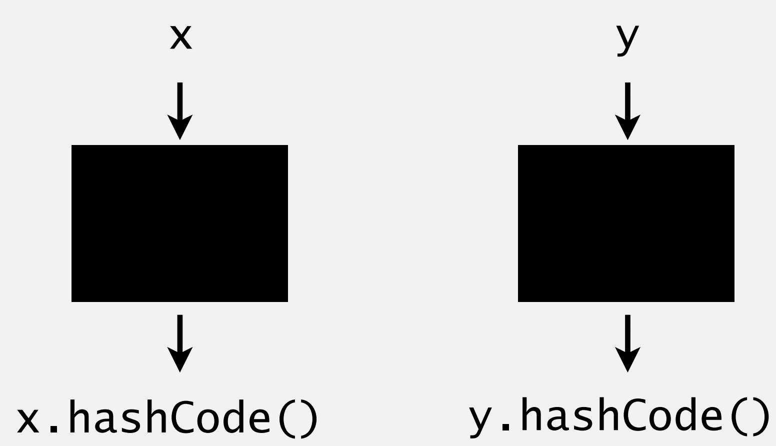
Practical challenge. Need different approach for each key type.

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

Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

Treat string as L-digit, base-31 number:

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

```
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";
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;
   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?

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()`. ← 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. Keys are bitstring; "universal" 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.

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

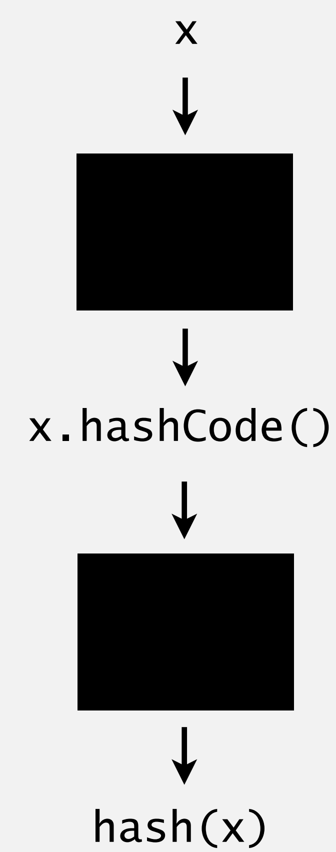
```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

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

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

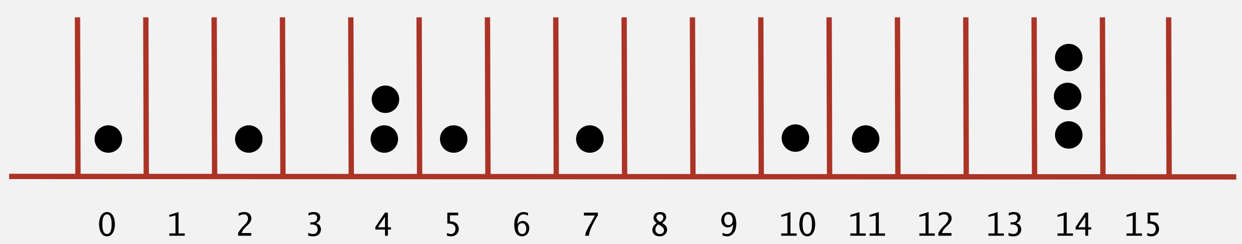
correct



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.

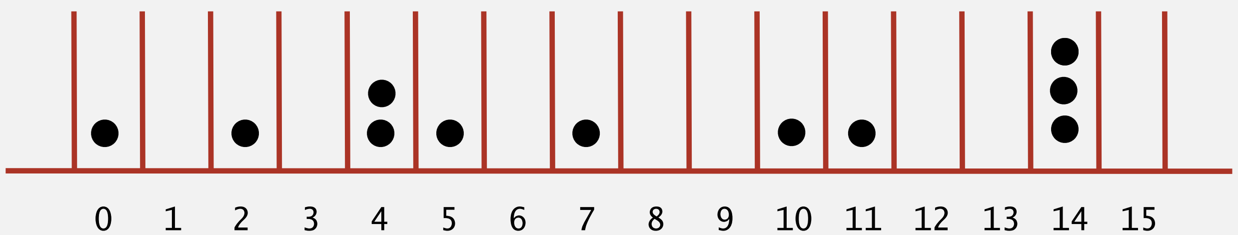
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

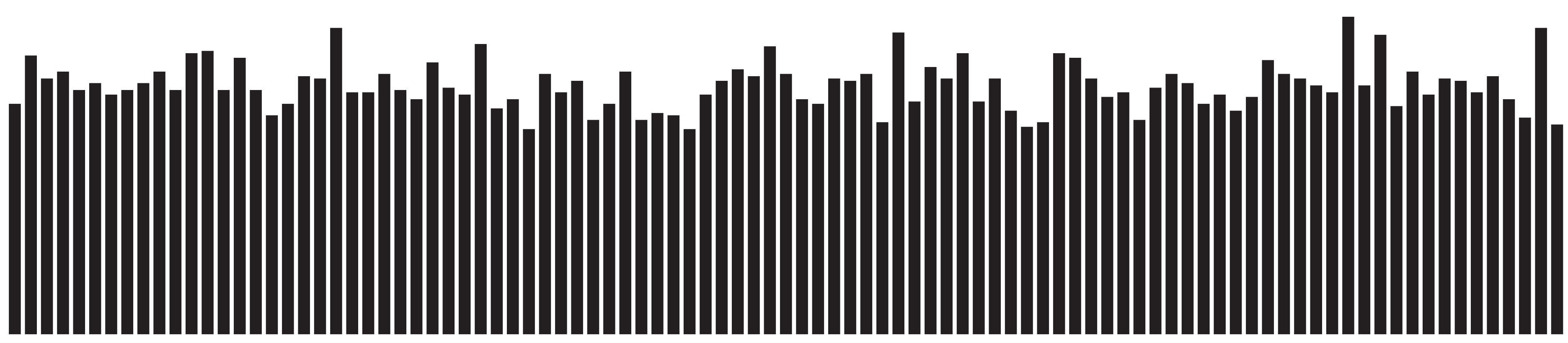
Load balancing. After M tosses, expect most loaded bin has $\sim \ln M / \ln \ln M$ balls.

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

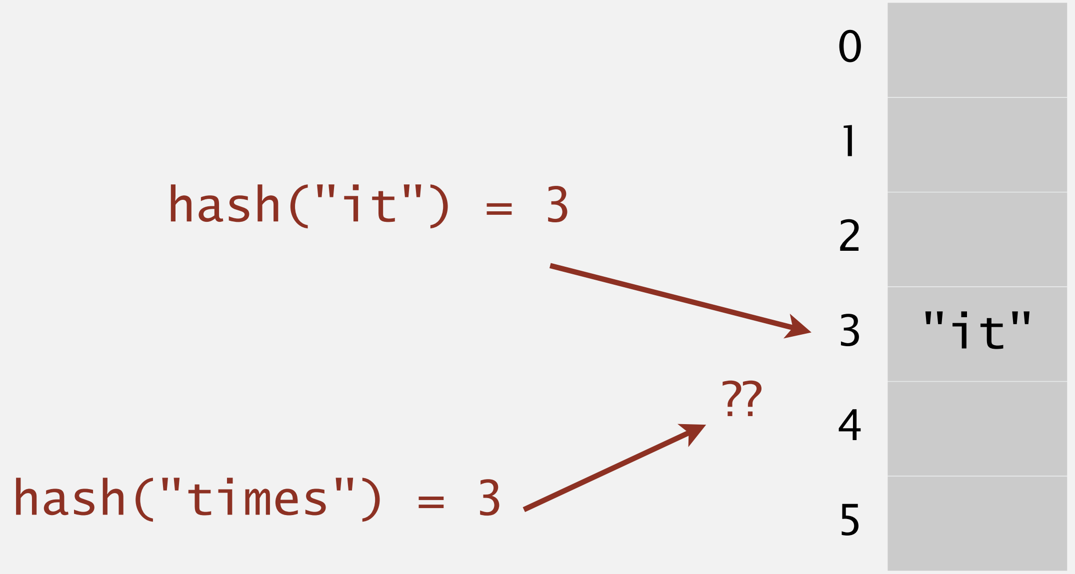
3.4 Hash Tables

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

Collisions

Collision. Two distinct keys hashing to same index.

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



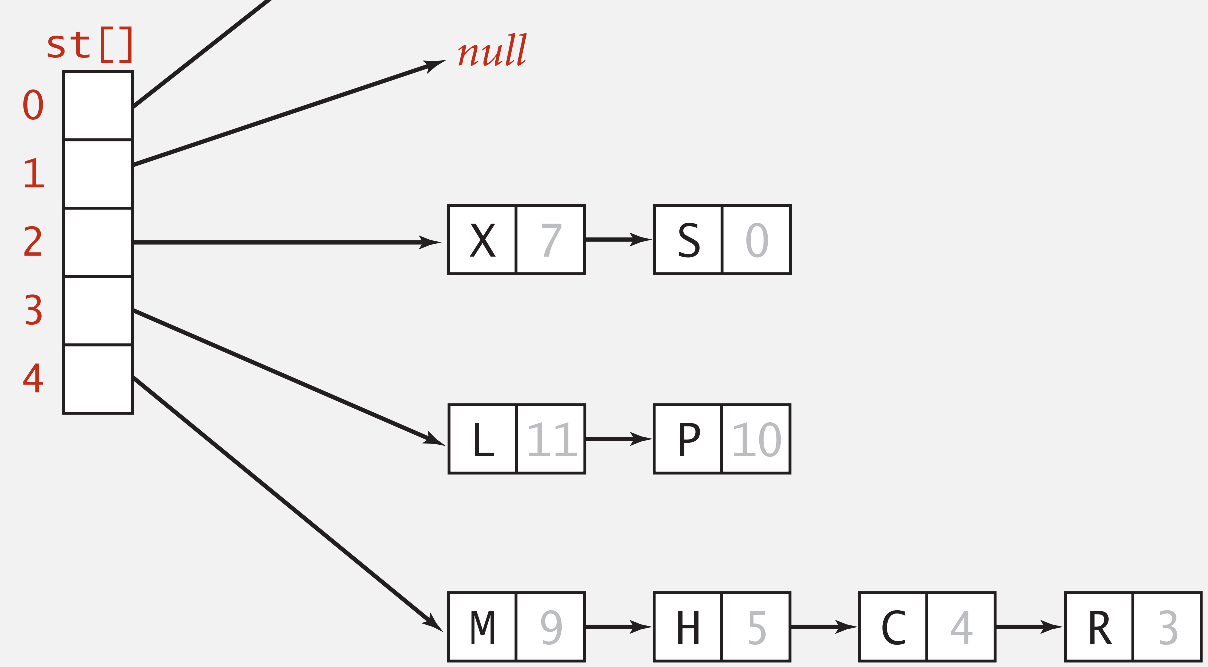
Challenge. Deal with collisions efficiently.

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^{th} chain (if not already in chain).
- Search: need to search only i^{th} 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



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)

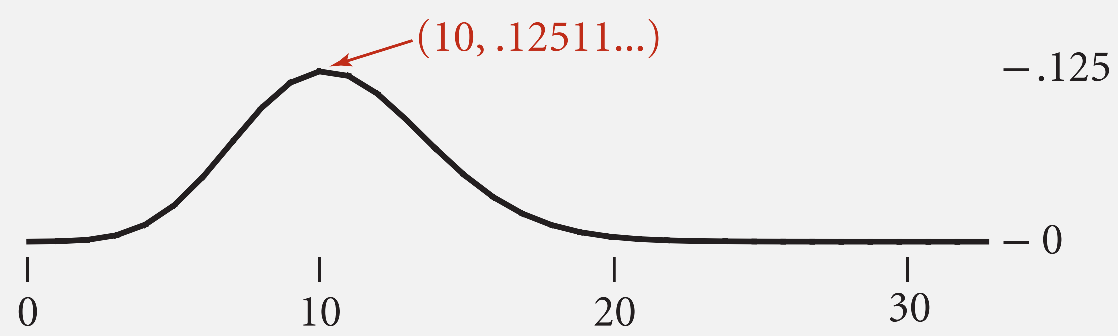
Separate-chaining symbol table: Java implementation

```
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.



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

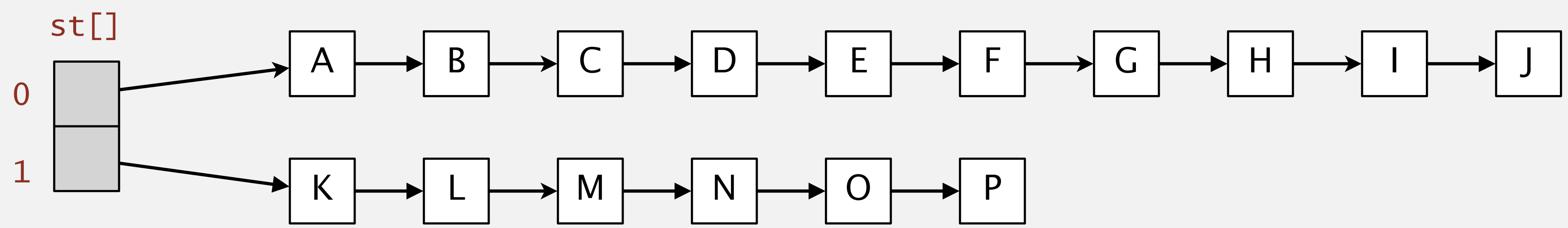
M times faster than sequential search

Resizing in a separate-chaining hash table

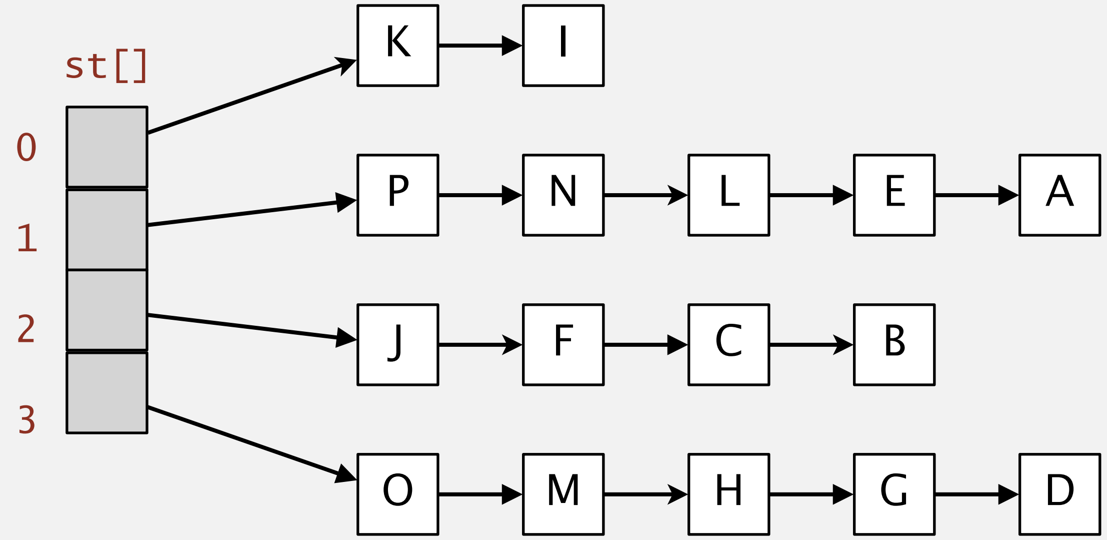
Goal. Average length of list N / M = constant.

- 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



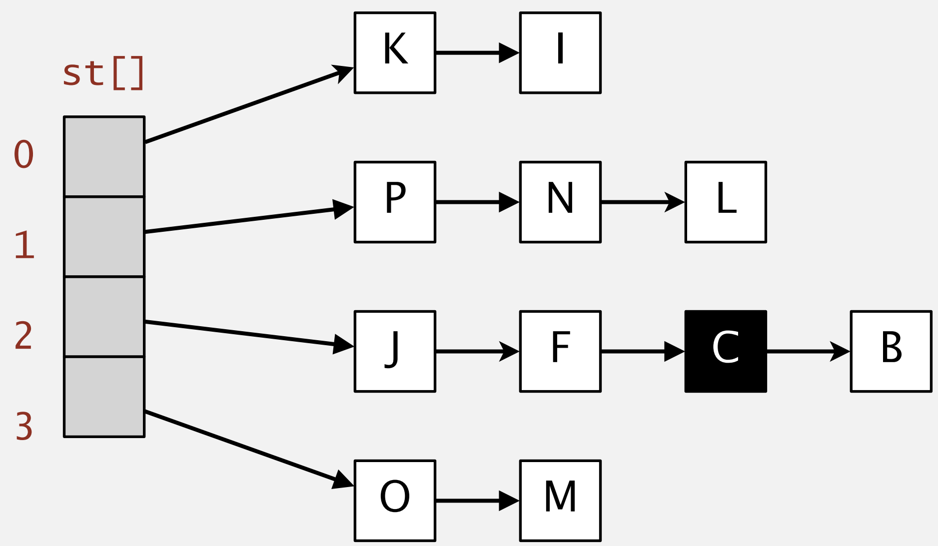
after resizing

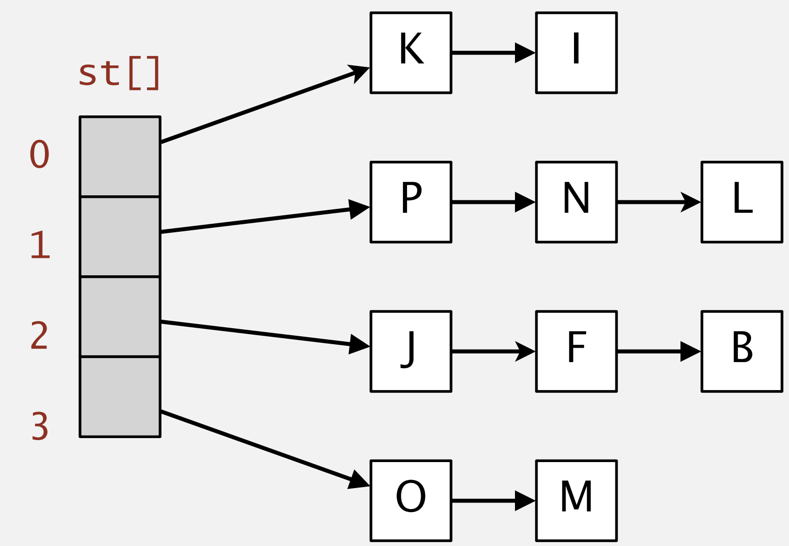


Deletion in a separate-chaining hash table

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

A. Easy: need only consider chain containing key.





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

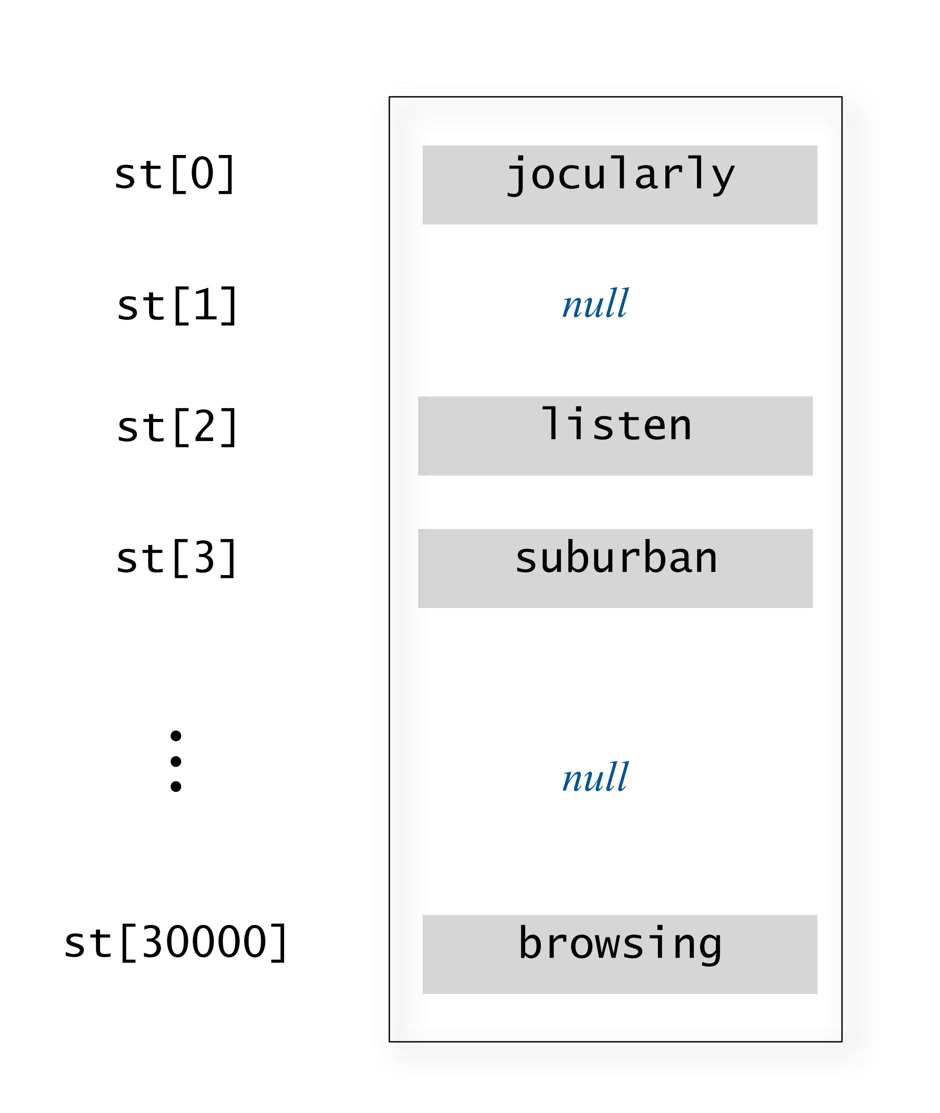
http://algs4.cs.princeton.edu

3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Collision resolution: open addressing

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



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
										K						

M = 16

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

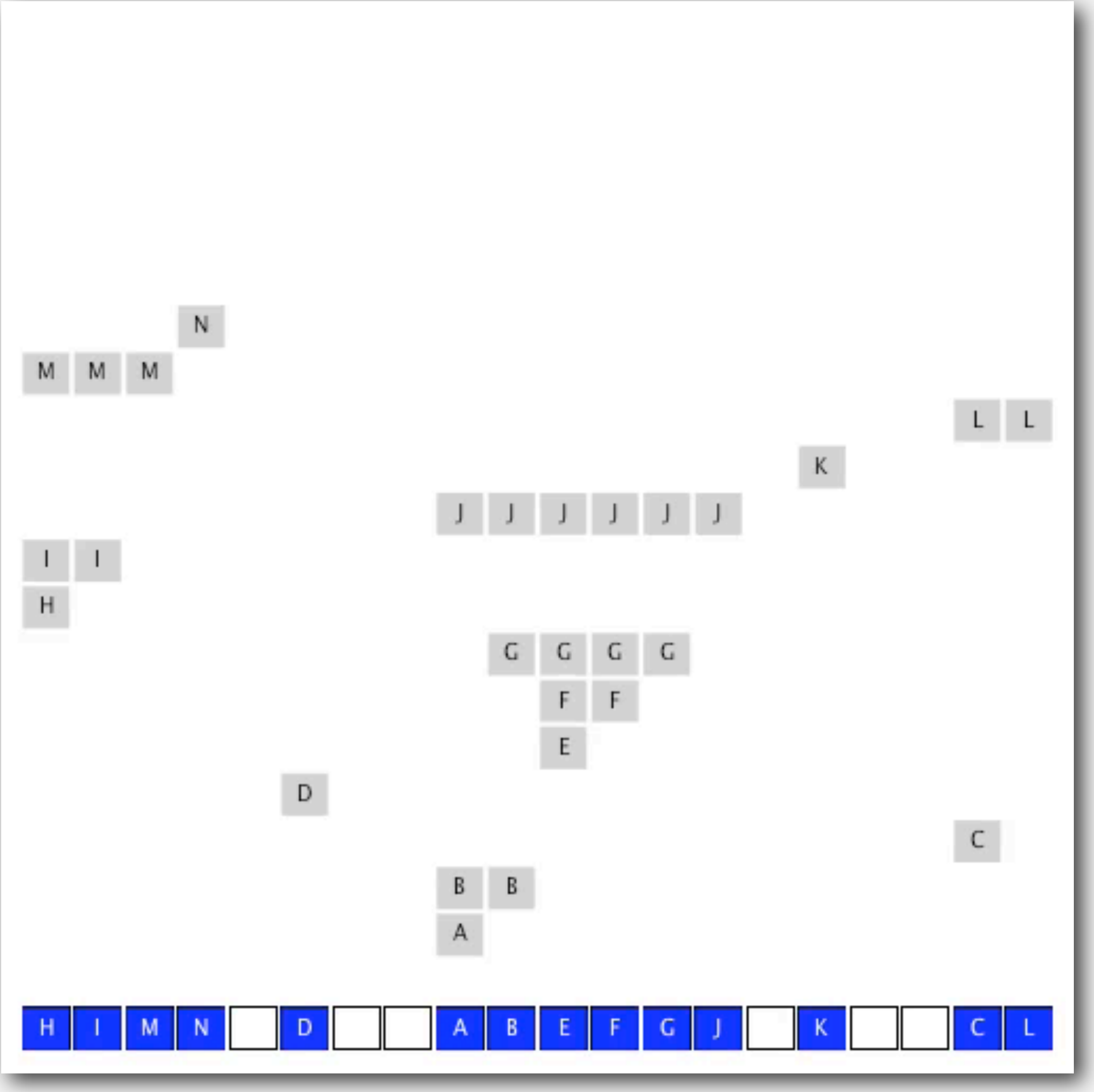
Linear-probing symbol table: Java implementation

```
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.

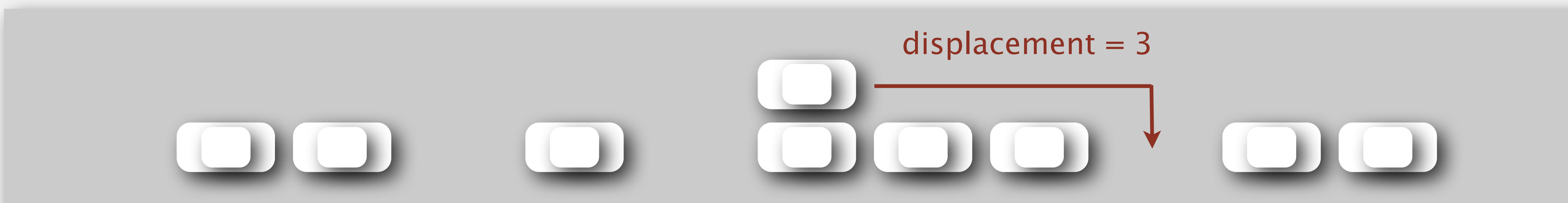
Observation. New keys likely to hash into middle of big clusters.



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?



Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

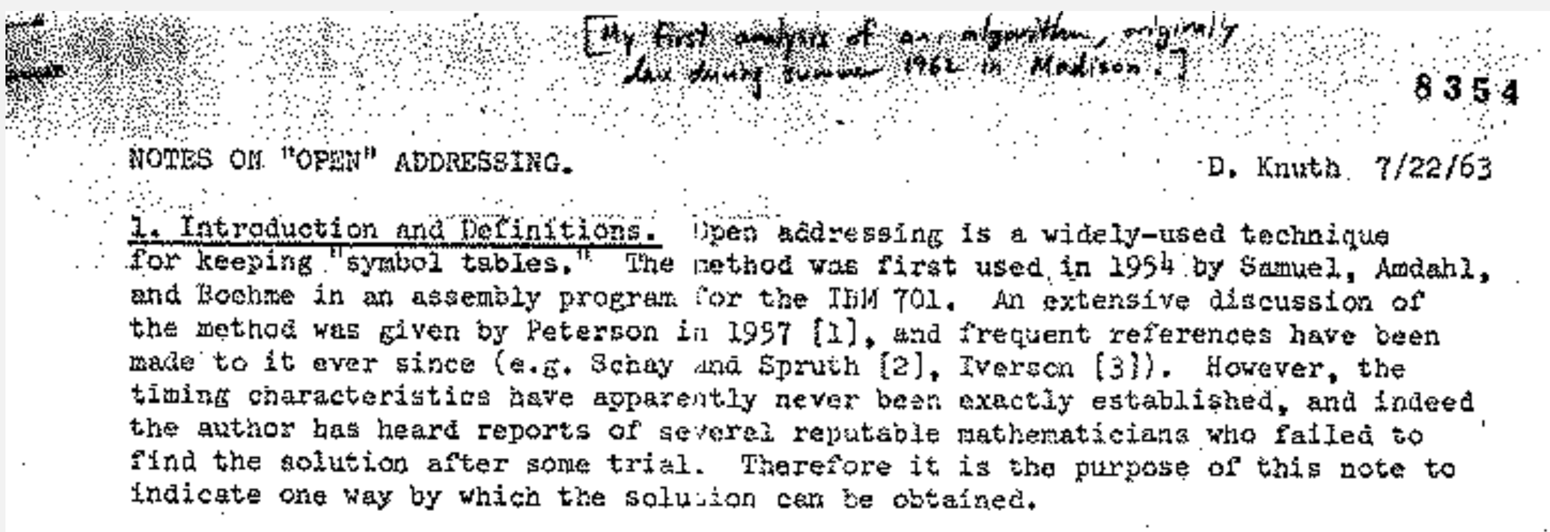
Analysis of linear probing

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.

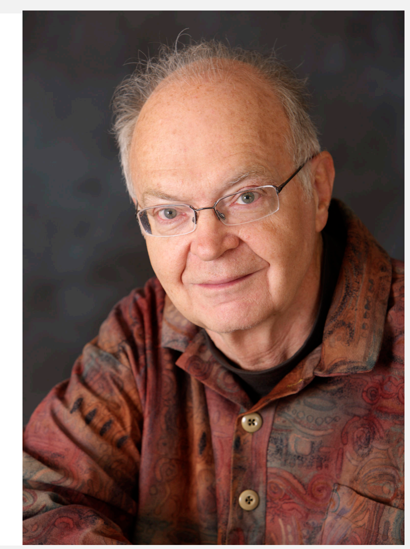


[My first analysis of an algorithm, originally done during summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ← # 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	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 $a + b + c = 0$.

- N^2 expected time case, N extra space.

4-SUM. Given N distinct integers, find four such that $a + b = c + d$.

- $N^2 \log N$ time (worst case), N^2 extra space.
- $N^2 \log N$ time (worst case), N extra space.
- N^2 expected time case, N^2 extra space.

http://algs4.cs.princeton.edu

3.4 Hash Tables

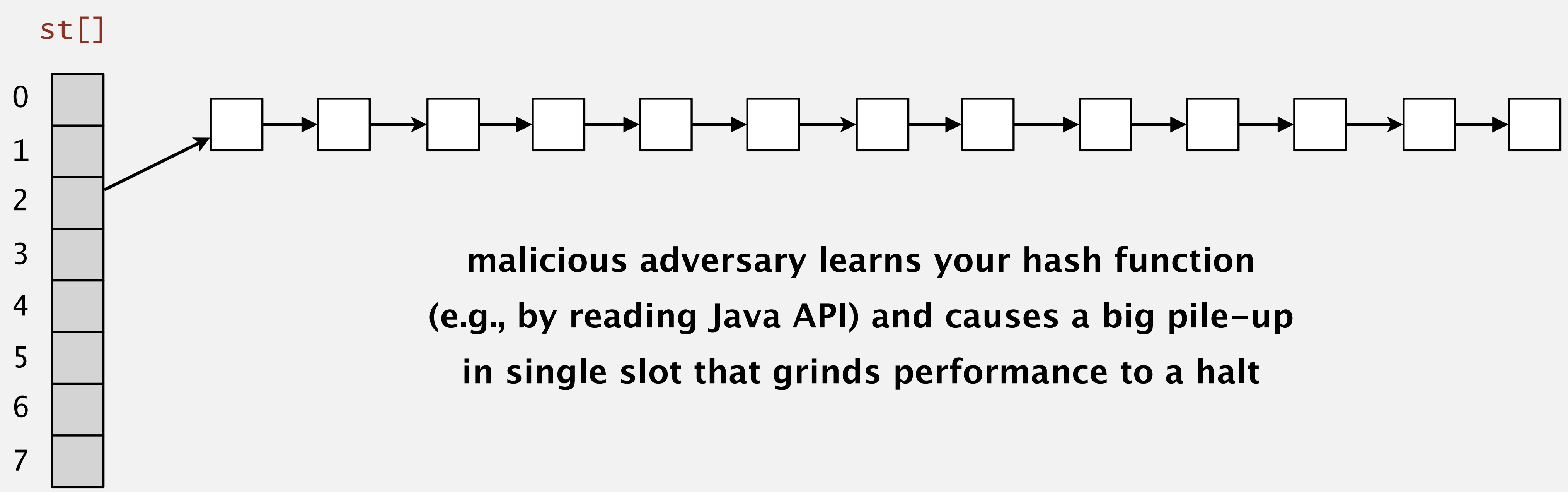
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

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

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
"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.

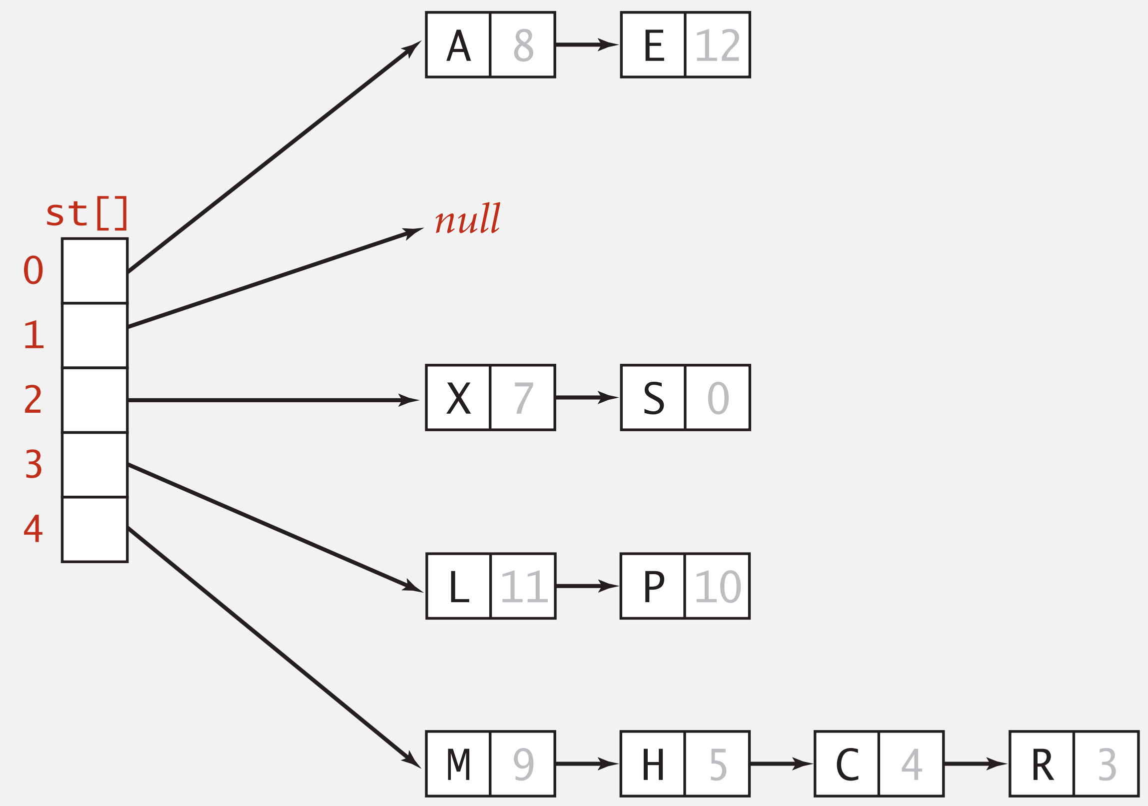
Separate chaining vs. linear probing

Separate chaining.

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

Linear probing.

- Less wasted space.
- Better cache performance.



	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]

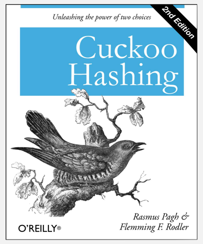
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\sim \lg \ln N$.

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 (`java.util.HashMap`)   ↑ separate chaining (`java.util.IdentityHashMap`)

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.

```
% more list.txt
was it the of

% java WhiteList list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackList list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

list of exceptional words

Exception filter applications

- 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
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	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 in the list.

```
public class WhiteList
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();       // create empty set of strings

      In in = new In(args[0]);
      while (!in.isEmpty())                      // read in whitelist
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))                 // print words in list
            StdOut.println(word);
      }
   }
}
```

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.

domain name is key   IP is value

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
```

domain name is key   URL is value

```
% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
```

```
% 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.

login is key — first name is value

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

login is key — section is value

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

```
% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cchehoud
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]);

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

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

process input file

build symbol table

process lookups with standard I/O

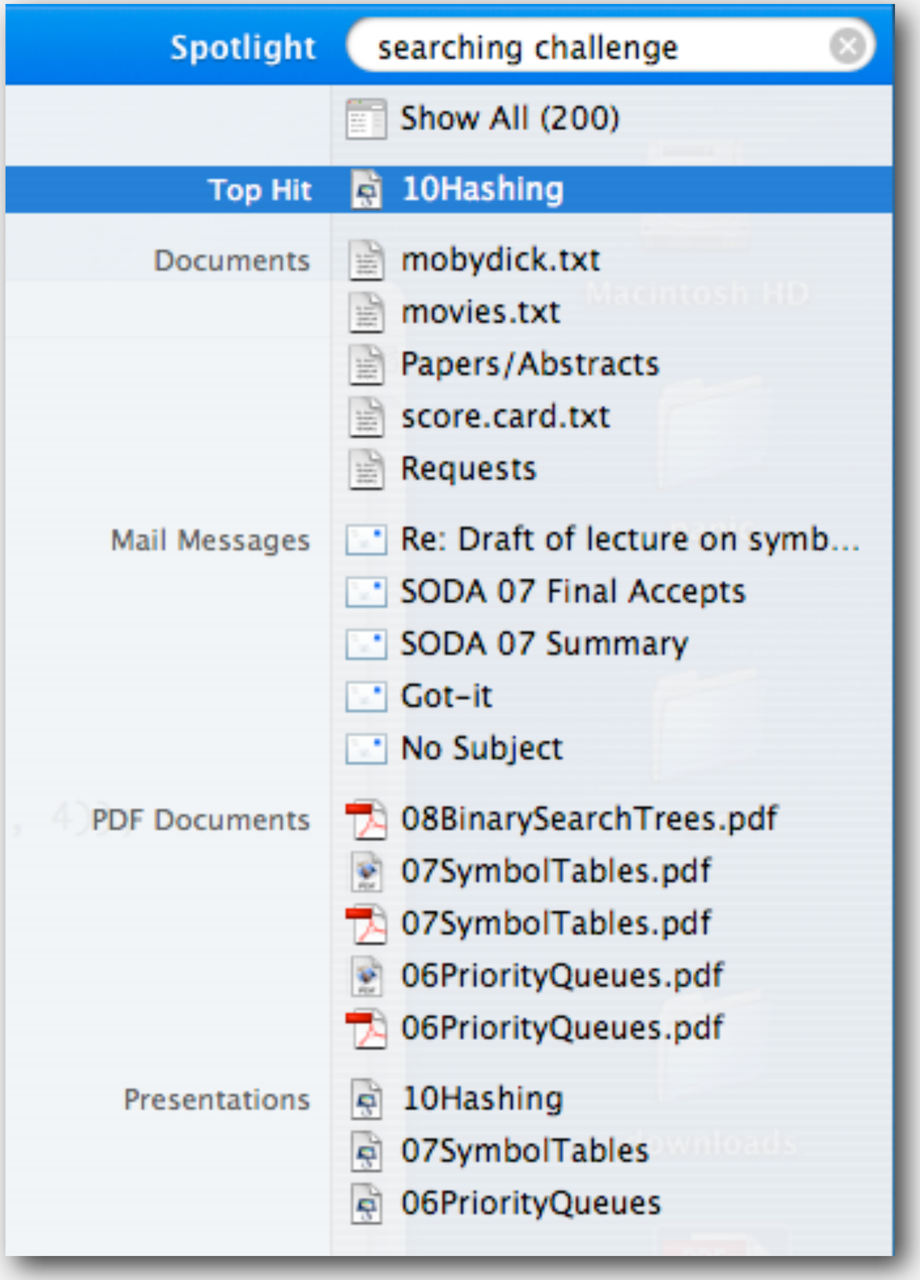
http://algs4.cs.princeton.edu

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));
      }
   }
}
```

symbol table

list of file names from command line

for each word in file, add file to corresponding set

process queries

Book index

Goal. Index for an e-book.

Index

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)

$$\underbrace{\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}}_{a[][]} \underbrace{\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix}}_{x[]} = \underbrace{\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}}_{b[]}$$

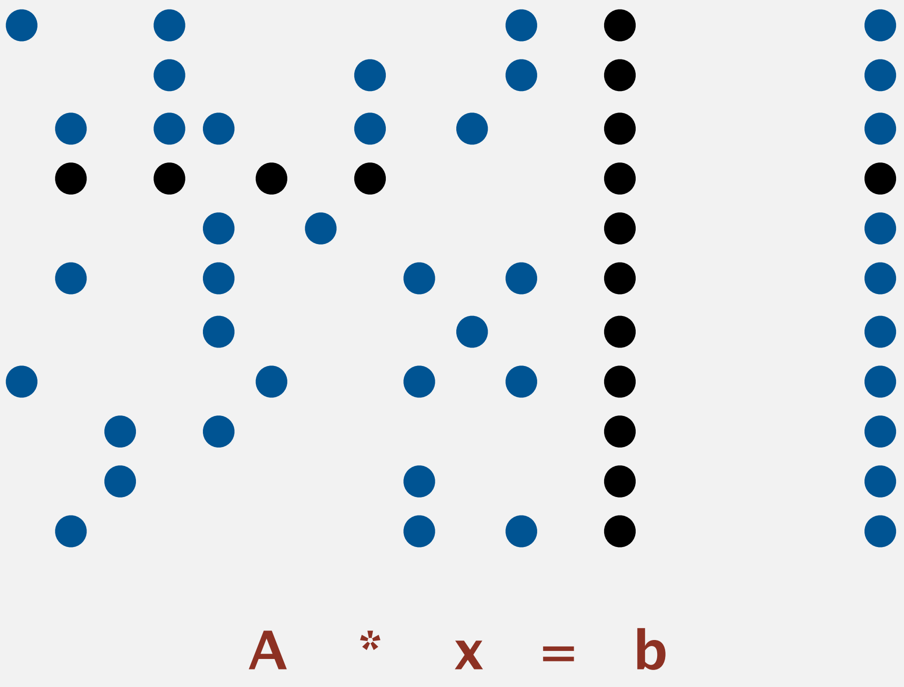
```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

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 ~ 10.



Vector representations

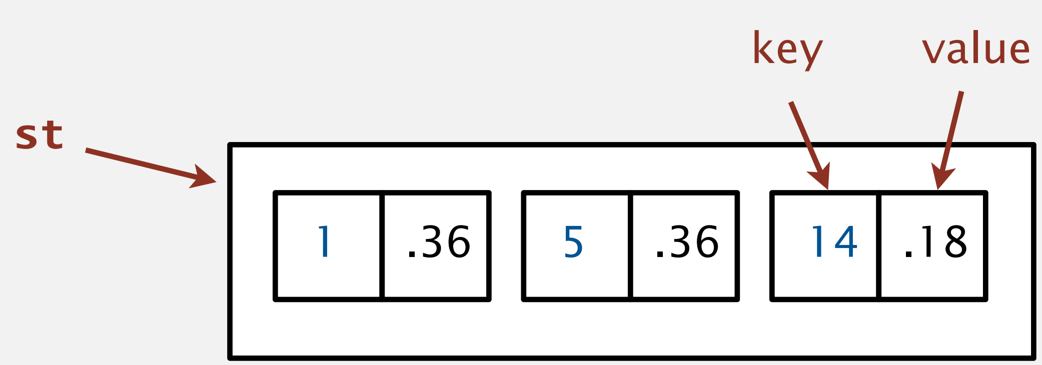
1d array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

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

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

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

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

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

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

HashST because order not important

empty ST represents all 0s vector

a[i] = value

return a[i]

iterate through indices of nonzero entries

dot product is constant time for sparse vectors

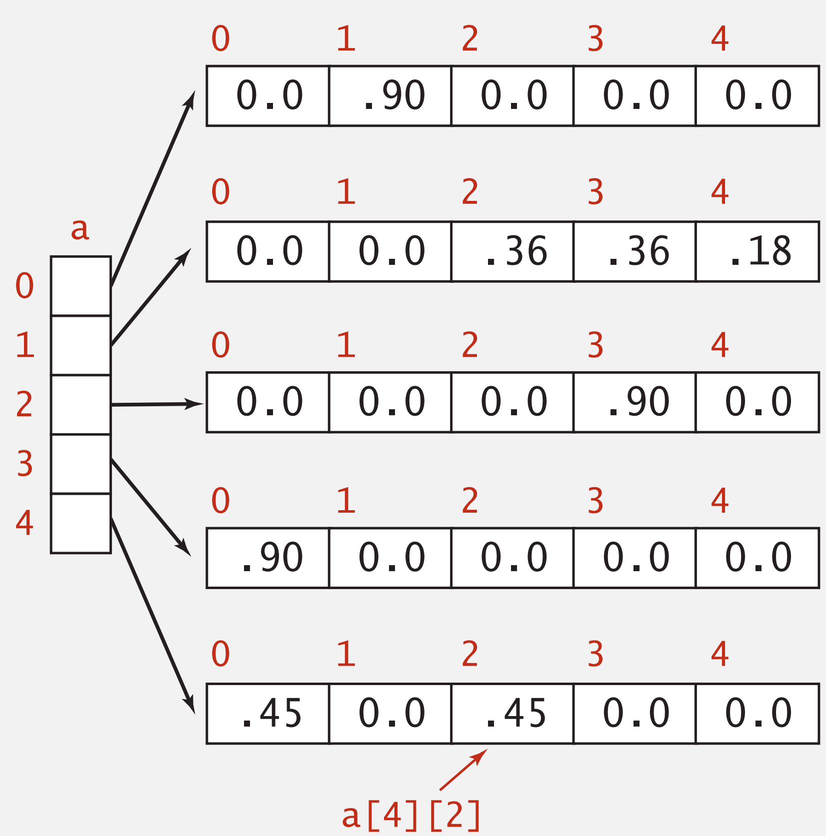
Matrix representations

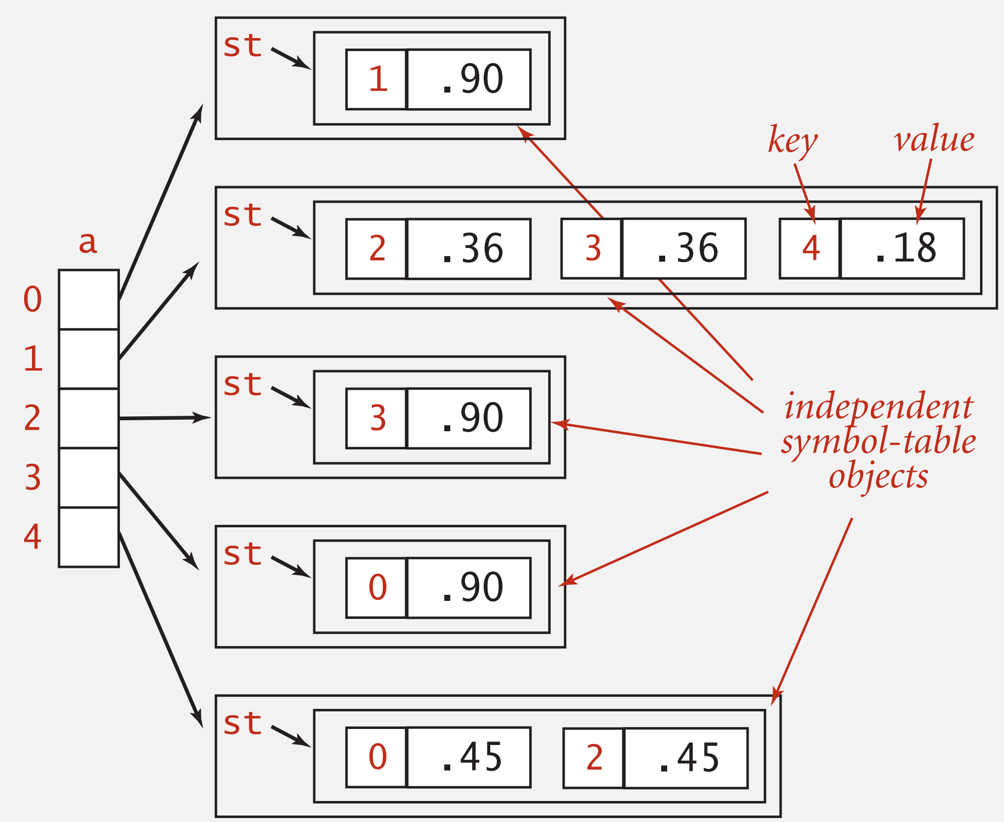
2D array (standard) matrix representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse matrix representation: Each row of matrix is a sparse vector.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).





Sparse matrix-vector multiplication

a[][] x[] b[]

$$\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} = \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}$$

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

linear running time for sparse matrix