https://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	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()`
hashing	n	n	n	1 †	1 †	1 †		`equals()` `hashCode()`

† under suitable technical assumptions

Q. Can we do better?

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

Hashing: basic plan

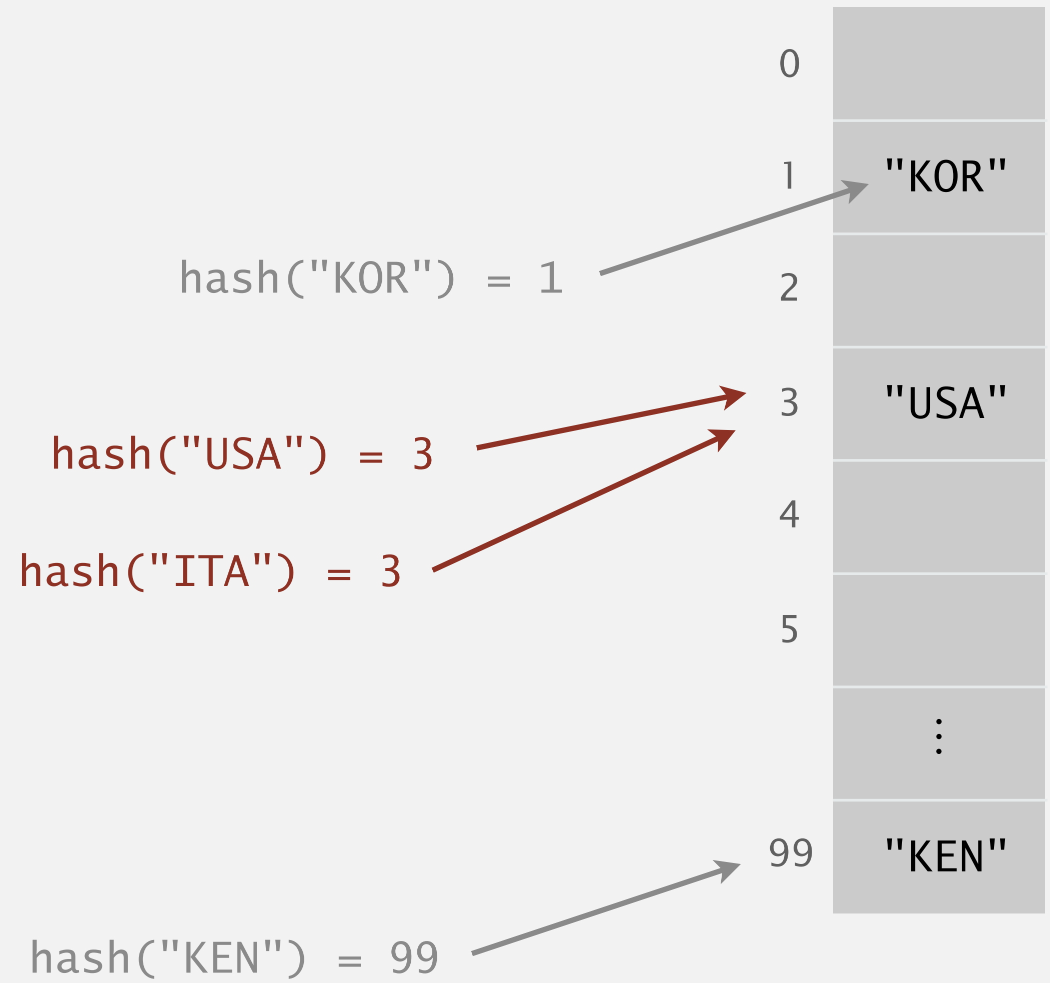
Save key–value pairs in a key-indexed table (index is a function of the key).

Hash function. Function that maps a key to an array index.

Collision. Two distinct keys that hash to same index.

Issue. Collisions are inevitable.

- How to limit collisions?
- How to accommodate collisions?



https://algs4.cs.princeton.edu

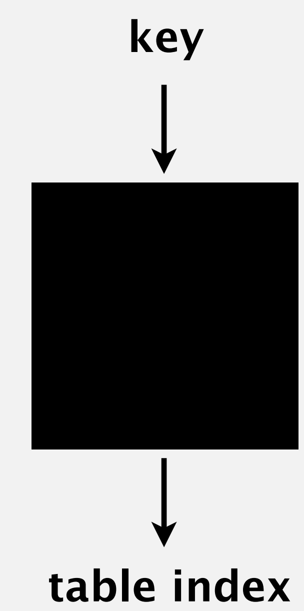
3.4 Hash Tables

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

Designing a hash function

Required properties. [for correctness]

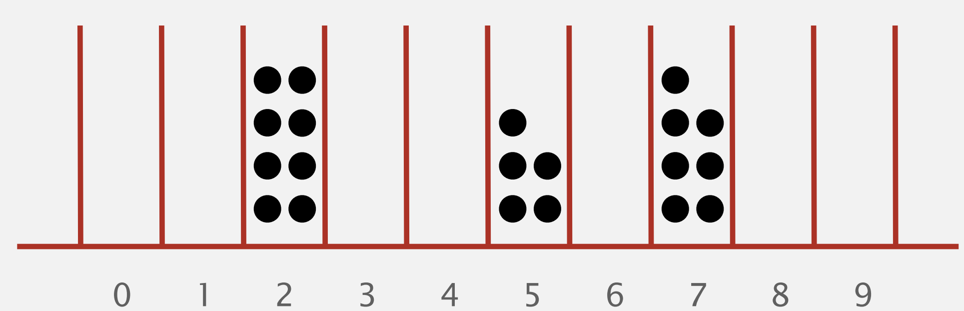
- Deterministic.
- Each key hashes to a table index between 0 and $m - 1$.



Desirable properties. [for performance]

- Very fast to compute. ← constants matter
- For any subset of n input keys, each table index gets approximately n / m keys.

leads to good hash-table performance
(m = 10, n = 20)



leads to bad hash-table performance
(m = 10, n = 20)

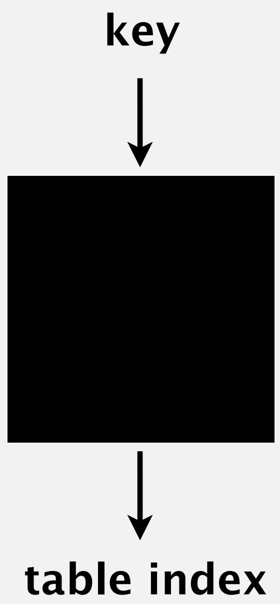
Designing a hash function

Required properties. [for correctness]

- Deterministic.
- Each key hashes to a table index between 0 and $m - 1$.

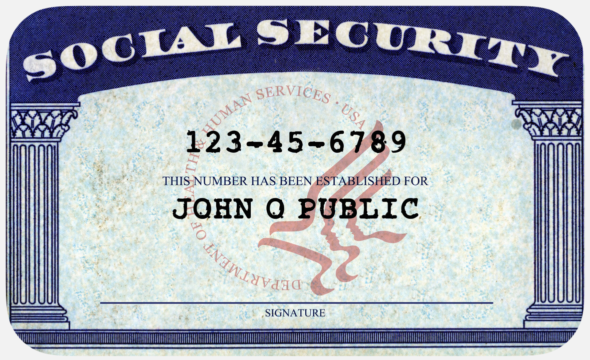
Desirable properties. [for performance]

- Very fast to compute.
- For any subset of n input keys, each table index gets approximately n / m keys.



Ex 1. Last 4 digits of U.S. Social Security number.

Ex 2. Last 4 digits of phone number.



+1 (718) 867-5309

Which is the last digit of your day of birth?

A. 0 or 1

B. 2 or 3

C. 4 or 5

D. 6 or 7

E. 8 or 9

Hash tables: quiz 2

Which is the last digit of your year of birth?

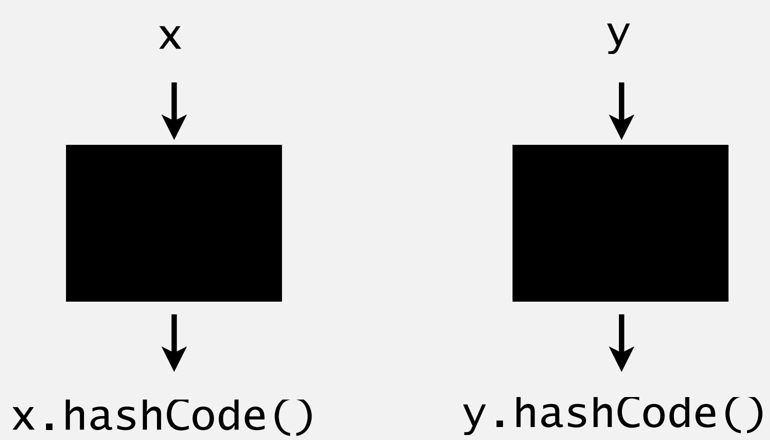
- **A.** 0 or 1
- **B.** 2 or 3
- **C.** 4 or 5
- **D.** 6 or 7
- **E.** 8 or 9

Java's hashCode() 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())`.



Customized implementations. `Integer`, `Double`, `String`, `java.net.URL`, …

Legal (but undesirable) implementation. Always return `17`.

User-defined types. Users are on their own.

Implementing hashCode(): integers and doubles

Java library implementations

```
public final class Integer
{
   private final int value;
   ...
   public int hashCode()
   {  return value;  }
}
```

```
public final class Double
{
   private final double value;
   ...
   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Implementing hashCode(): arrays

$31x + y$ rule.

- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add next integer in array. (prime)

```
public class Arrays
{
   ...

   public static int hashCode(int[] a) {
      if (a == null)
         return 0;          ← special case for null

      int hash = 1;
      for (int i = 0; i < a.length; i++)
         hash = 31*hash + a[i];      ← 31x + y rule
      return hash;
      }

}
```

Java library implementation

Implementing hashCode(): user-defined types

```
public final class 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 = 1;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }

}
```

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

Implementing hashCode(): user-defined types

```
public final class 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()
   {
      return Objects.hash(who, when, amount);
   }

}
```

shorthand

Implementing hashCode()

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.

Principle. Every significant field contributes to hash.

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

Hash tables: quiz 3

Which function maps hashable keys to integers between 0 and m-1 ?

A.

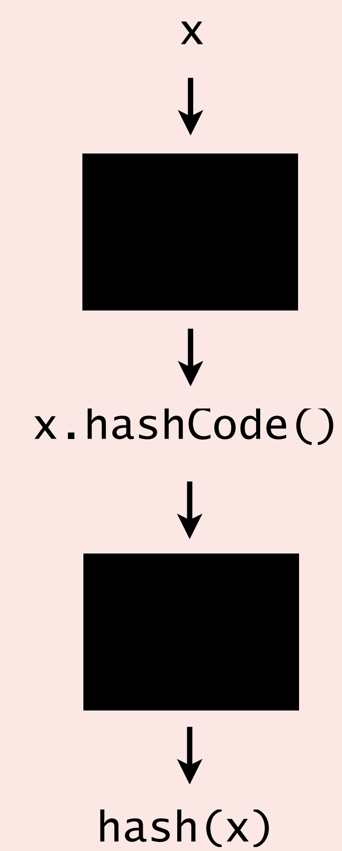
```
private int hash(Key key)
{  return key.hashCode() % m;  }
```

B.

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

C. Both A and B.

D. Neither A nor B.



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;  }
```

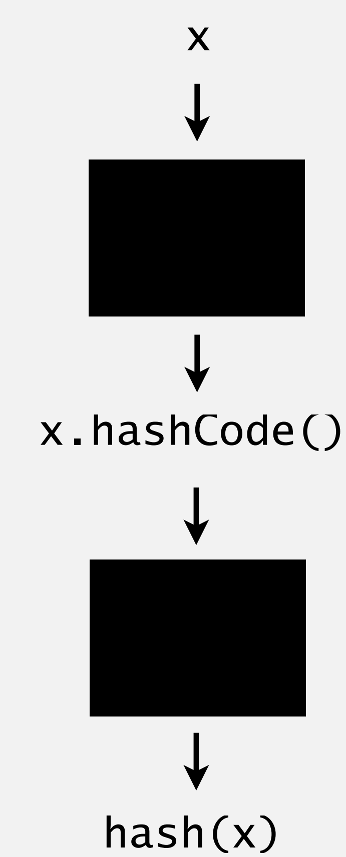
1-in-a-billion bug

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

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

correct

if `m` is a power of 2, can use `key.hashCode() & (m-1)`



Uniform hashing assumption

Uniform hashing assumption. Any key is equally likely to hash to one of m possible indices.

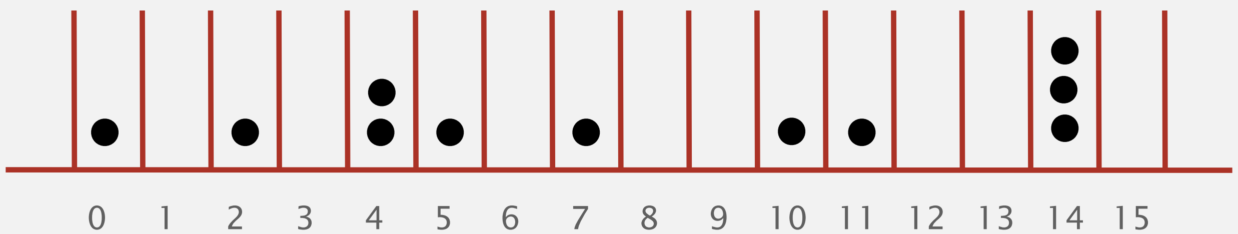
and independently of other keys

Uniform hashing assumption

Uniform hashing assumption. Any key is equally likely to hash to one of m possible indices.

and independently of other keys

Bins and balls. Toss n balls uniformly at random into m bins.



m = 16 bins, n = 11 balls

Bad news. [birthday problem]

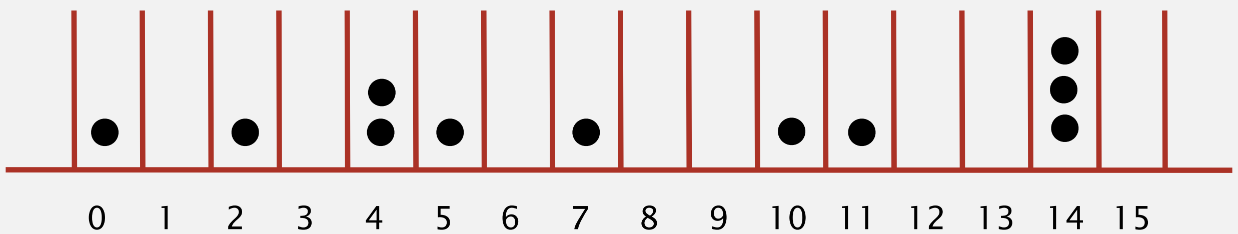
- In a random group of 23 people, more likely than not that two people share the same birthday.
- Expect two balls in the same bin after $\sim \sqrt{\pi m / 2}$ tosses.

23.9 when $m = 365$

Uniform hashing assumption

Uniform hashing assumption. Any key is equally likely to hash to one of m possible indices.

Bins and balls. Toss n balls uniformly at random into m bins.

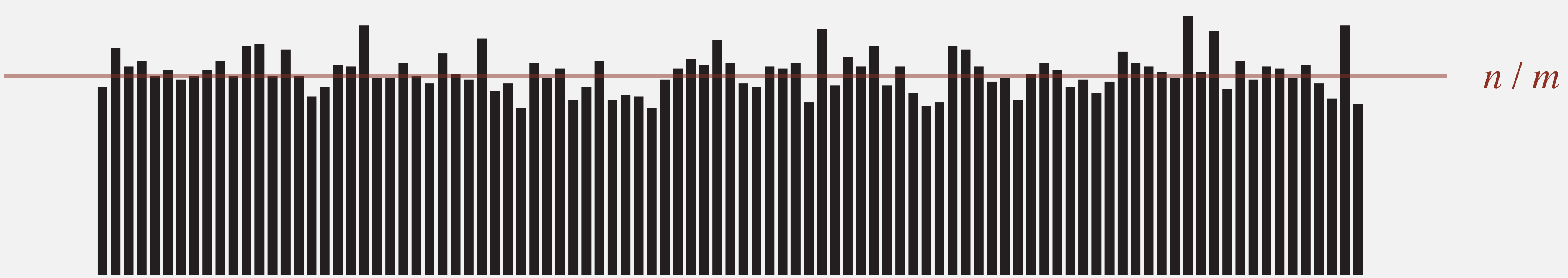


m = 16 bins, n = 11 balls

Good news. [load balancing]

Binomial(n, 1 / m)

- When $n >> m$, expect most bins to have approximately n / m balls.
- When $n = m$, expect most loaded bin has $\sim \ln n / \ln \ln n$ balls.



hash value frequencies for words in Tale of Two Cities (m = 97)

https://algs4.cs.princeton.edu

3.4 Hash Tables

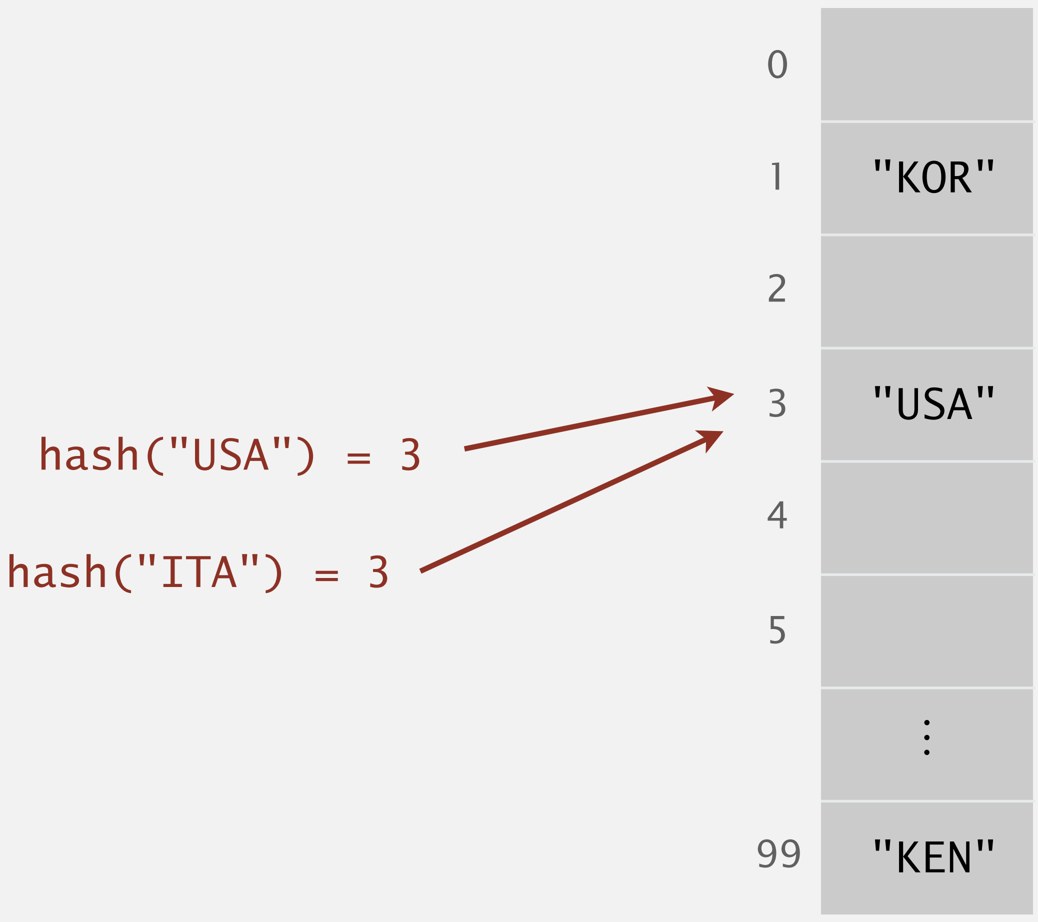
- *hash functions*
- ***separate chaining***
- *linear probing*
- *context*

Collisions

Collision. Two distinct keys that hash to the same index.

- Birthday problem ⇒ can't avoid collisions. ← unless you have a ridiculous (quadratic) amount of memory

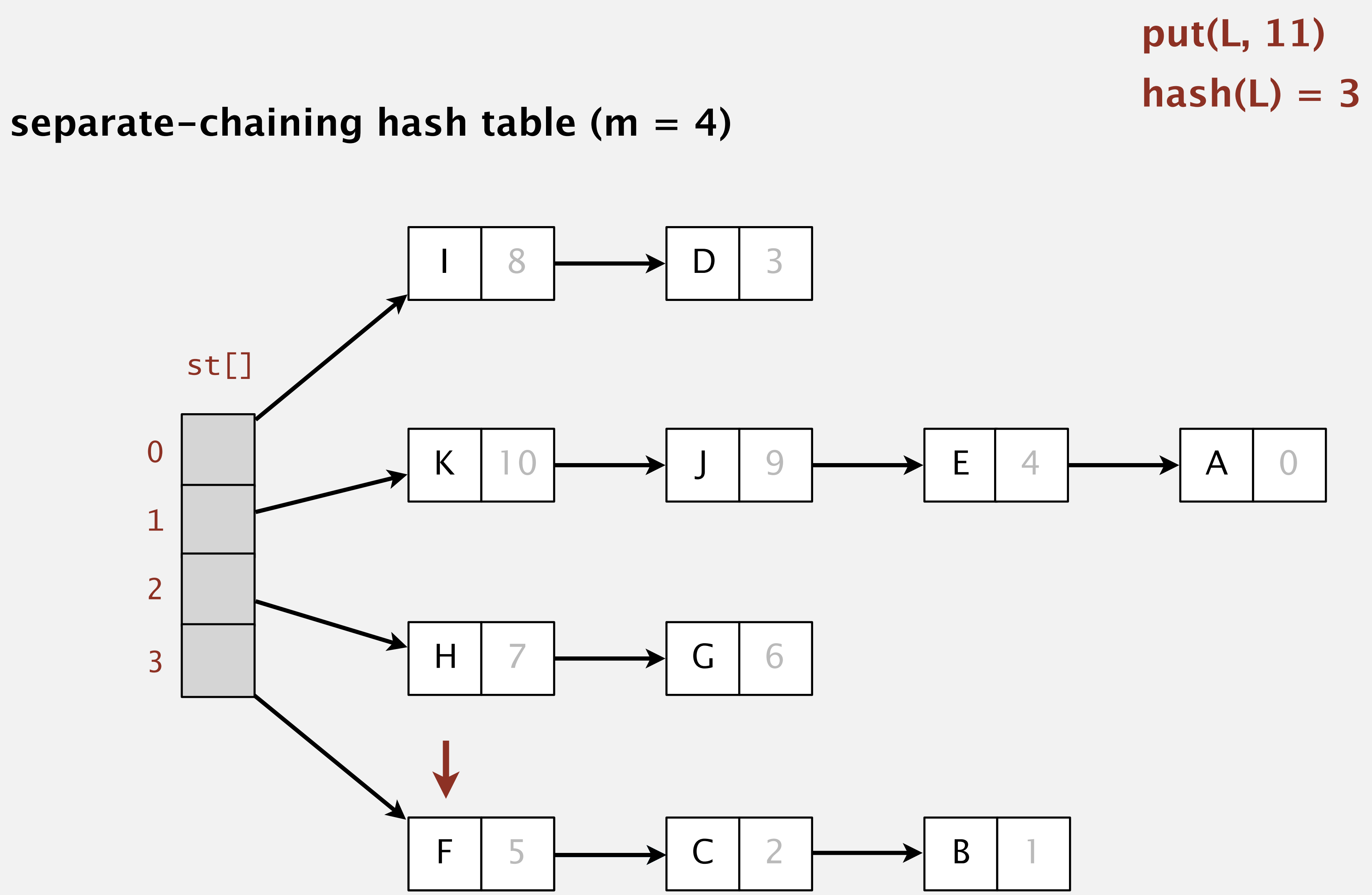
- Load balancing ⇒ no index gets too many collisions.
  ⇒ ok to scan through all colliding keys.



Separate-chaining hash table

Use an array of m linked lists.

- Hash: map key to table index i between 0 and $m - 1$.
- Insert: add key–value pair at front of chain i (if not already in chain).

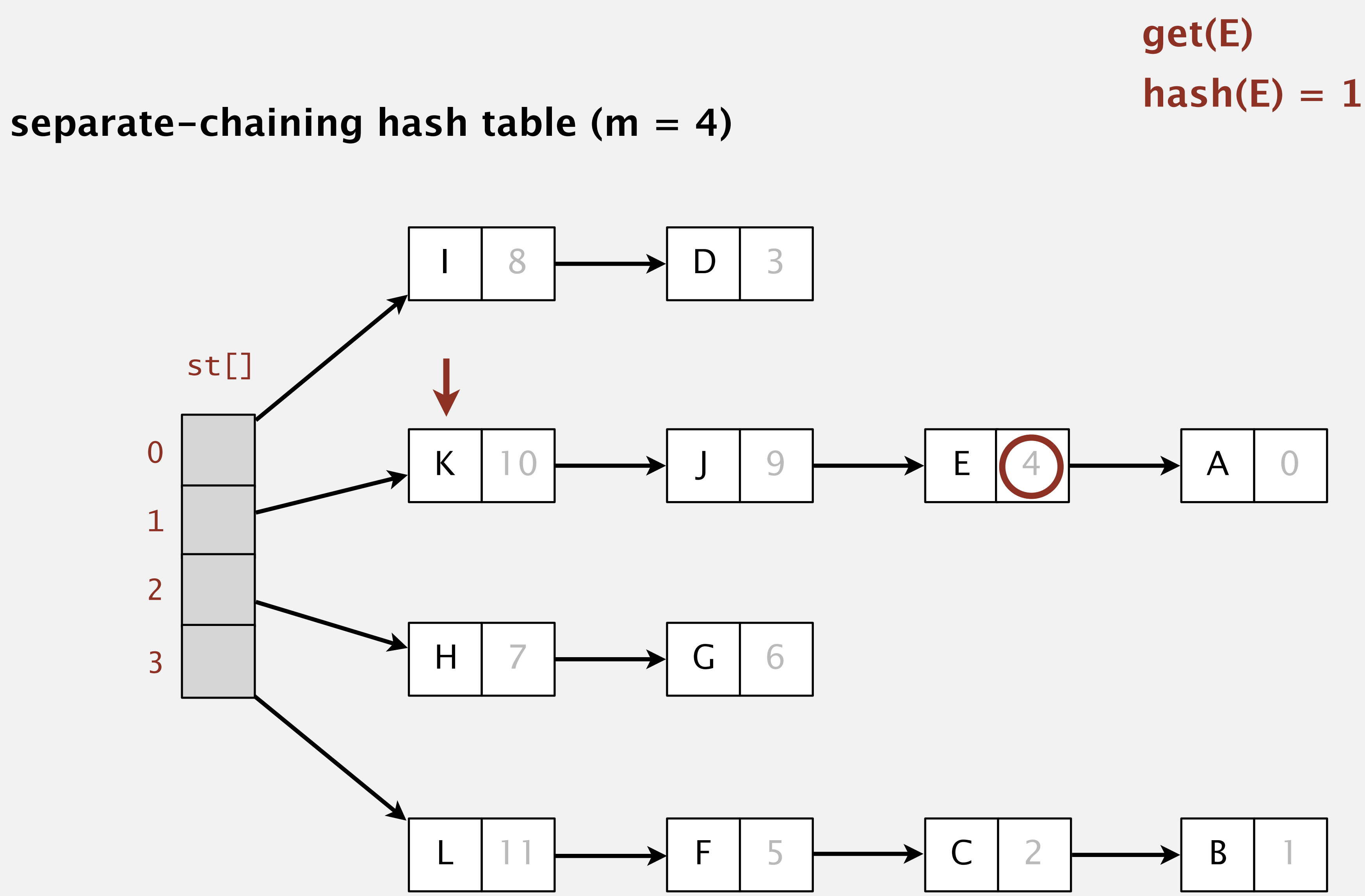


Separate-chaining hash table

Use an array of m linked lists.

- Hash: map key to table index i between 0 and $m - 1$.
- Insert: add key–value pair at front of chain i (if not already in chain).

  Search: perform sequential search in chain i.



Separate-chaining hash table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int m = 128;              // 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 resizing code omitted

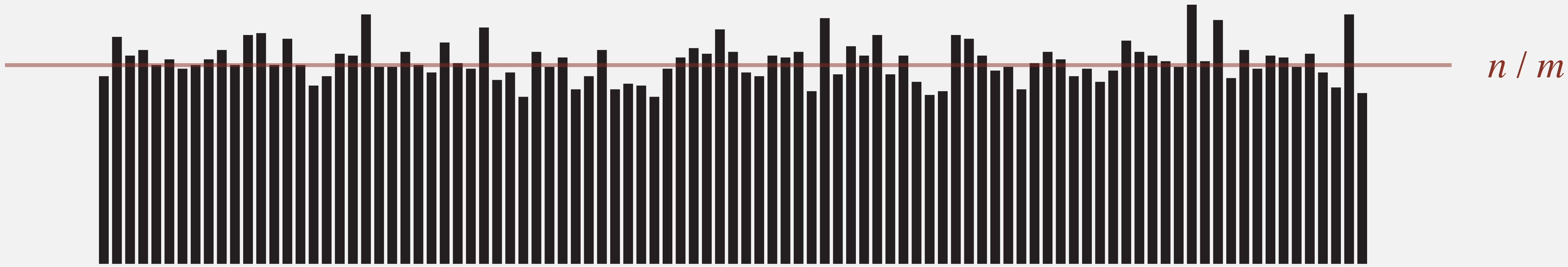
no generic array creation (declare key and value of type Object)

Separate-chaining hash table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int m = 128;               // 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

Recall load balancing. Under uniform hashing assumption, length of each chain is tightly concentrated around mean = n / m.



hash value frequencies for words in Tale of Two Cities (m = 97)

calls to either `equals()` or `hashCode()`

Consequence. Expected number of probes for search/insert is $\Theta(n / m)$.

- m too small $\Rightarrow$ chains too long.
- m too large $\Rightarrow$ too many empty chains.
- Typical choice: $m \sim \frac{1}{4} n \Rightarrow \Theta(1)$ time for search/insert.

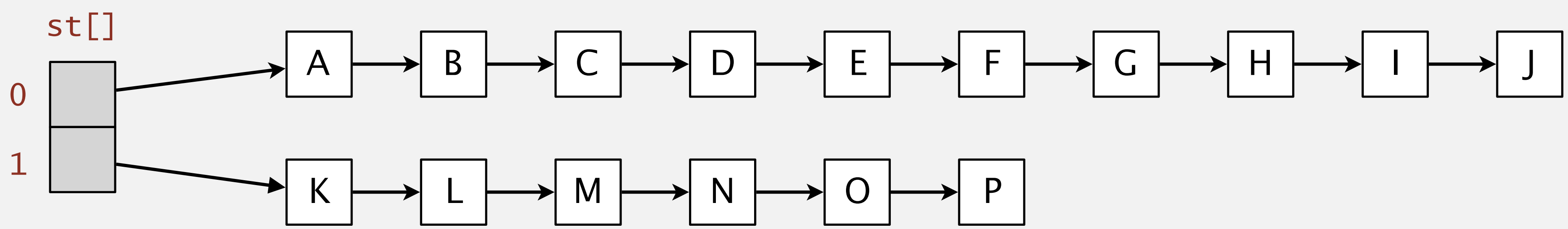
m times faster than sequential search

Resizing in a separate-chaining hash table

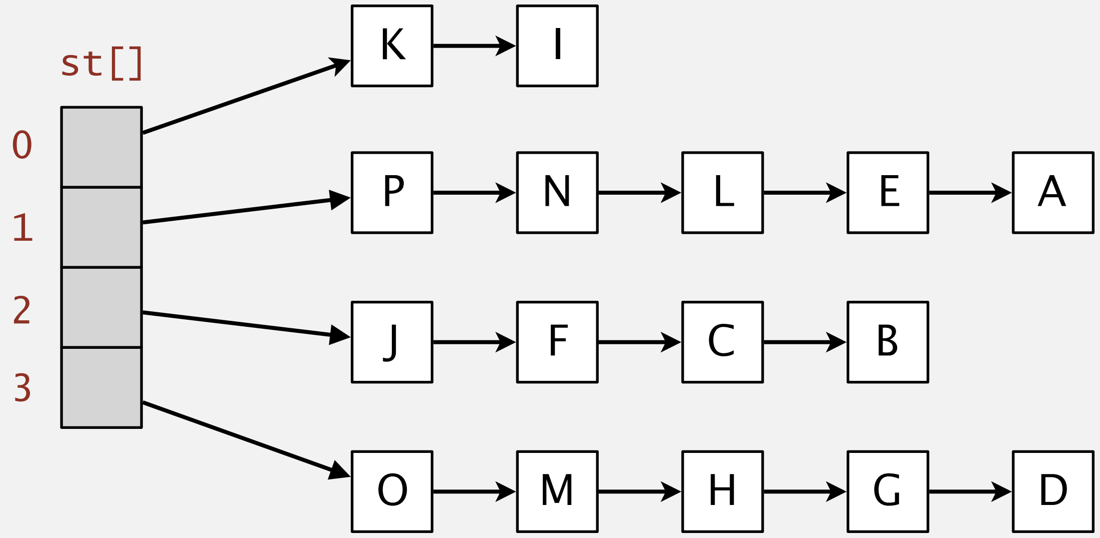
Goal. Average length of list n / m = constant.

- Double length m of array when $n / m \geq 8$.
- Halve length m of array when $n / m \leq 2$.
- Note: need to rehash all keys when resizing. ← `x.hashCode()` does not change; but `hash(x)` typically does

before resizing (n/m = 8)



after resizing (n/m = 4)



Hash tables: quiz 4

How to delete a key-value pair from a separate-chaining hash table?

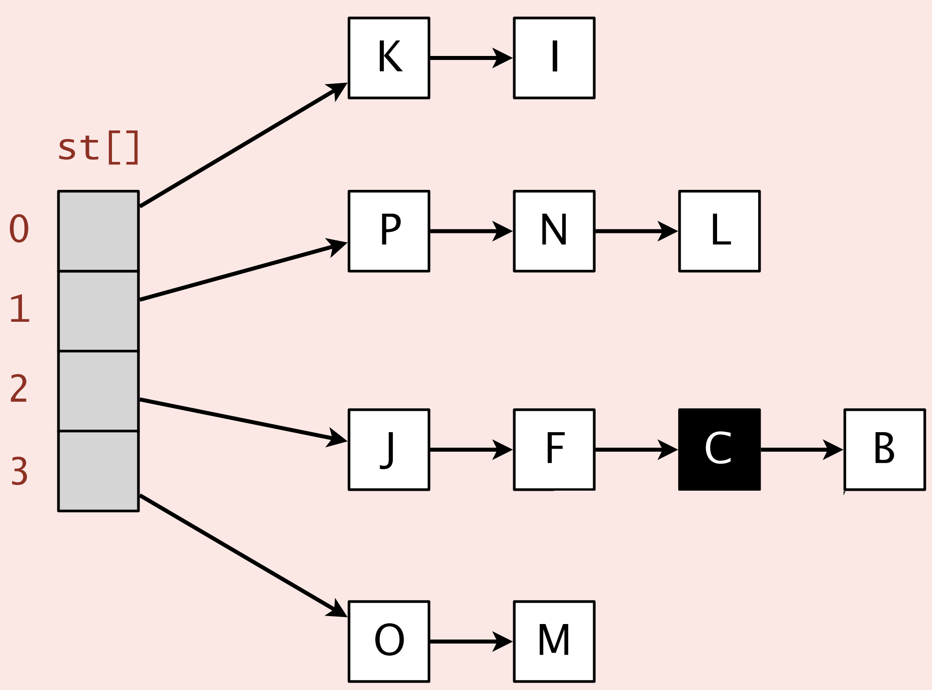
A. Search for key; remove key–value pair from linked list.

B. Compute hash of key; reinsert all other key–value pairs in chain.

C. Either A or B.

D. Neither A nor B.

delete C



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search	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

https://algs4.cs.princeton.edu

3.4 Hash Tables

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

Linear-probing hash table

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by probing: search successive cells until either finding the key or an unused cell.

Inserting into a linear-probing hash table.

linear–probing hash table

	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

put(K, 14)

hash(K) = 7

Linear-probing hash table

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by probing: search successive cells until either finding the key or an unused cell.

Searching in a linear-probing hash table.

linear-probing hash table

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

get(K)
hash(K) = 7

get(Z)
hash(Z) = 8

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 $i + 1$, $i + 2$, ….

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

Note. Array length 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 = 32768;
   private Value[] vals = (Value[]) new Object[m];
   private Key[]   keys = (Key[])   new Object[m];

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

   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 resizing code omitted

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int m = 32768;
   private Value[] vals = (Value[]) new Object[m];
   private Key[]   keys = (Key[])   new Object[m];

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

   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;
   }

}
```

Under the uniform hashing assumption, where is the next key most likely to be added in this linear-probing hash table (no resizing)?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H	I	M	N		D			A	B	E	F	G	J		K			C	L

A. Index 7.

B. Index 14.

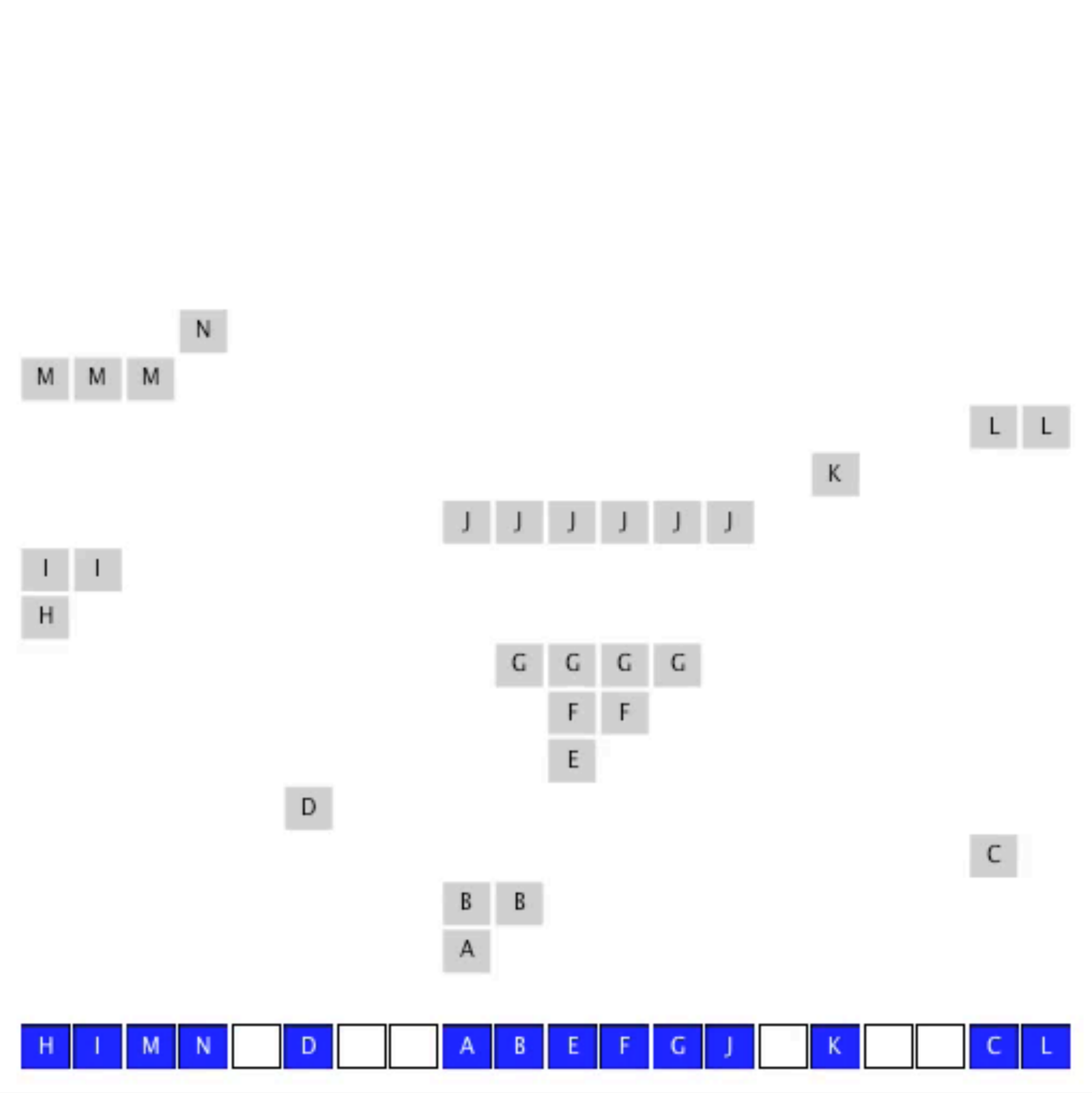
C. Either index 4 or 14.

D. All open indices are equally likely.

Clustering

Cluster. A contiguous block of keys.

Observation. New keys disproportionately likely to hash into big clusters.



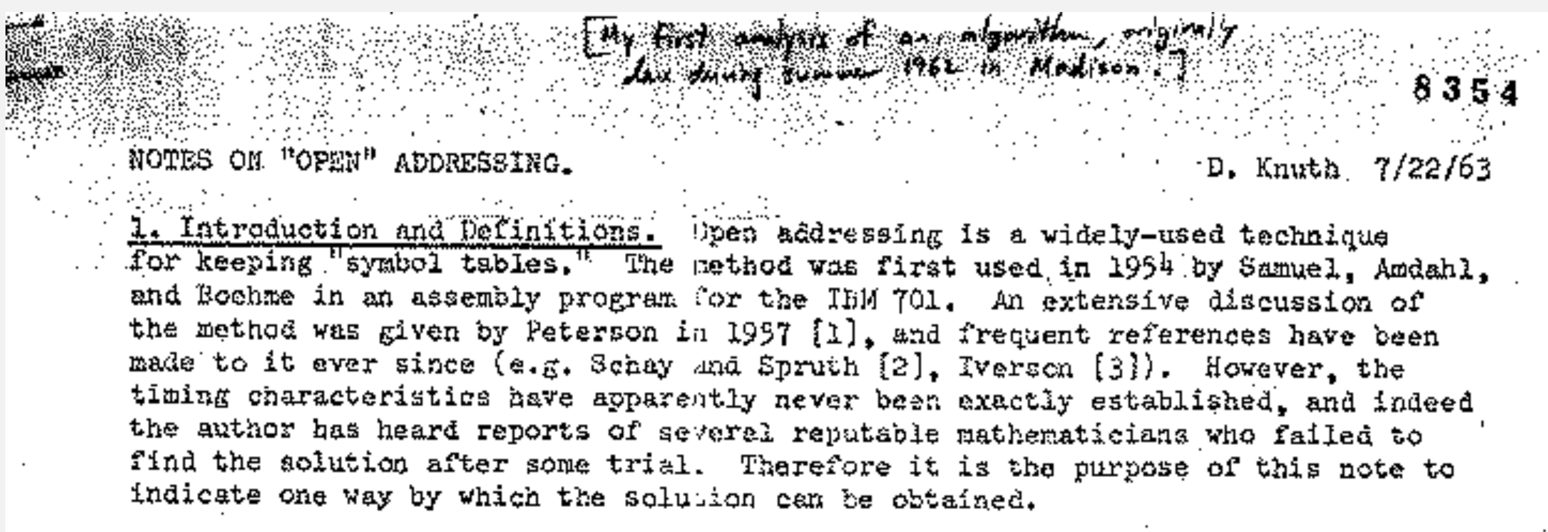
Analysis of linear probing

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 at most

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

search hit — **search miss / insert**

Pf. [beyond course scope]

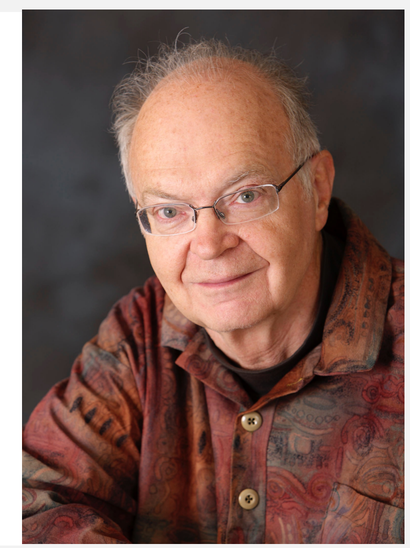


[My first analysis of an algorithm, originally done during summer 1962 in Madison.]

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 $n / m \le ½$.

- Double length of array m when $n / m \ge ½$.
- Halve length of array m when $n / m \le ⅛$.
- 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	

Hash tables: quiz 6

How to delete a key–value pair from a linear-probing hash table?

A. Search for key; remove key–value pair from arrays.

B. Search for key; remove key–value pair from arrays.
Shift all keys in cluster after deleted key 1 position to left.

C. Either A and B.

D. Neither A nor B.

before deleting S

cluster after deleted key (positions 7–8)

	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

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search	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. $\Theta(n^2)$ expected time; $\Theta(n)$ extra space.

https://algs4.cs.princeton.edu

3.4 Hash Tables

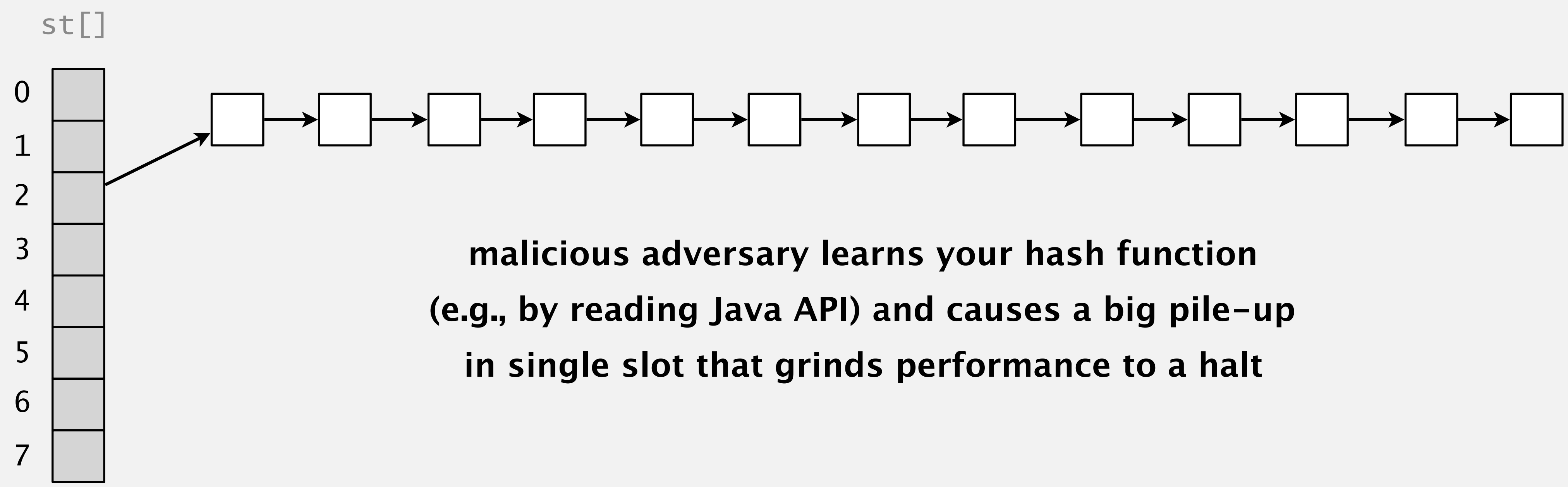
- *hash functions*
- *separate chaining*
- *linear probing*
- ***context***

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A1. Yes: aircraft control, nuclear reactor, pacemaker, HFT, ...

A2. Yes: denial-of-service (DoS) attacks.



Real-world exploits. [Crosby–Wallach 2003]

- Linux 2.4.20 kernel: save files with carefully chosen names.
- Bro server: send carefully chosen packets to DoS the server, using less bandwidth than a dial-up modem.

War story: algorithmic complexity attacks

A Java bug report.

Jan Lieskovsky 2011-11-01 14:13:47 UTC 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.

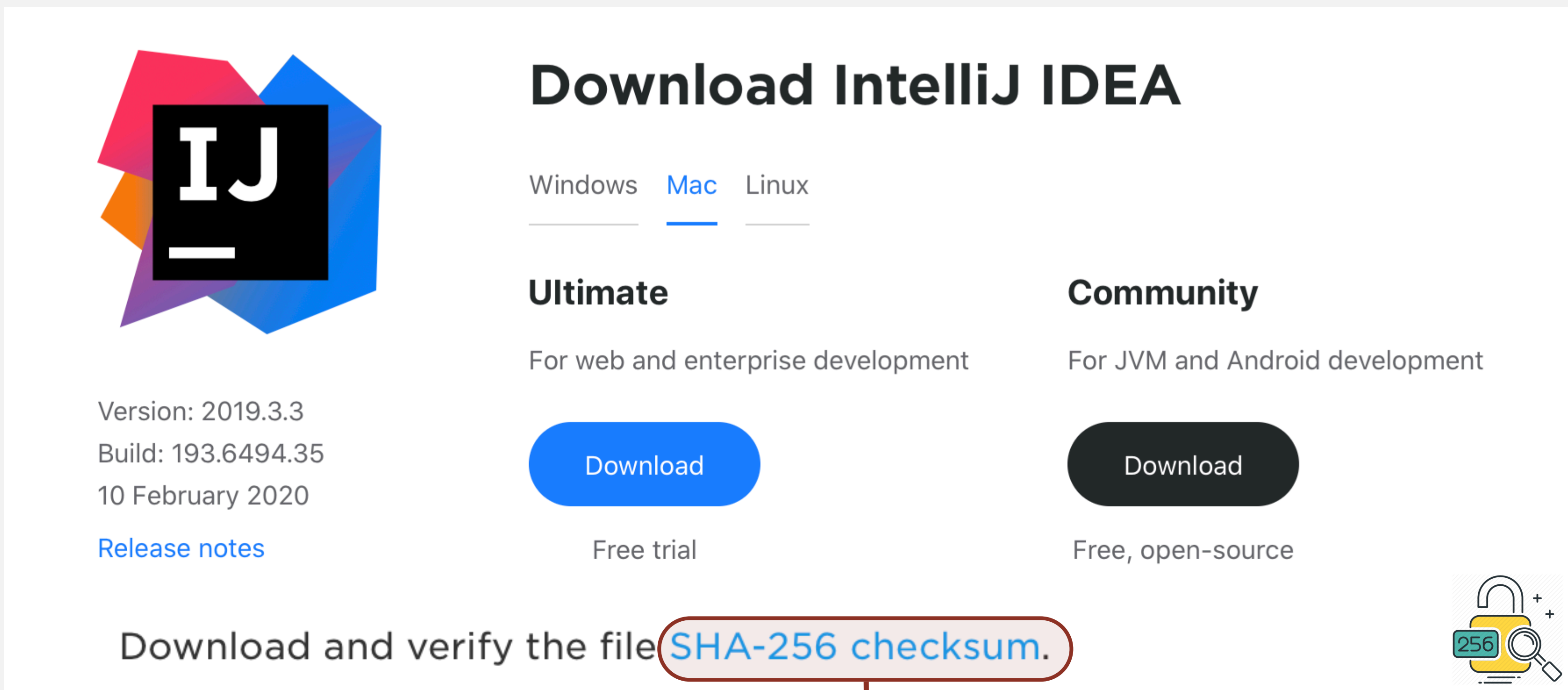
This problem is similar to the issue that was previously reported for and fixed in e.g. perl:
http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf

https://bugzilla.redhat.com/show_bug.cgi?id=750533

Hashing: file verification

When downloading a file from the web:

- Vendor publishes hash of file.
- Client checks whether hash of downloaded file matches.
- If mismatch, file corrupted. ⟵ (e.g., error in transmission or infected by virus)



c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270

```
~/Desktop> sha256sum ideaIC-2019.3.3.dmg
c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270
```

Hashing: cryptographic applications

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. MD5, SHA-1, SHA-256, SHA-512, Whirlpool, ...

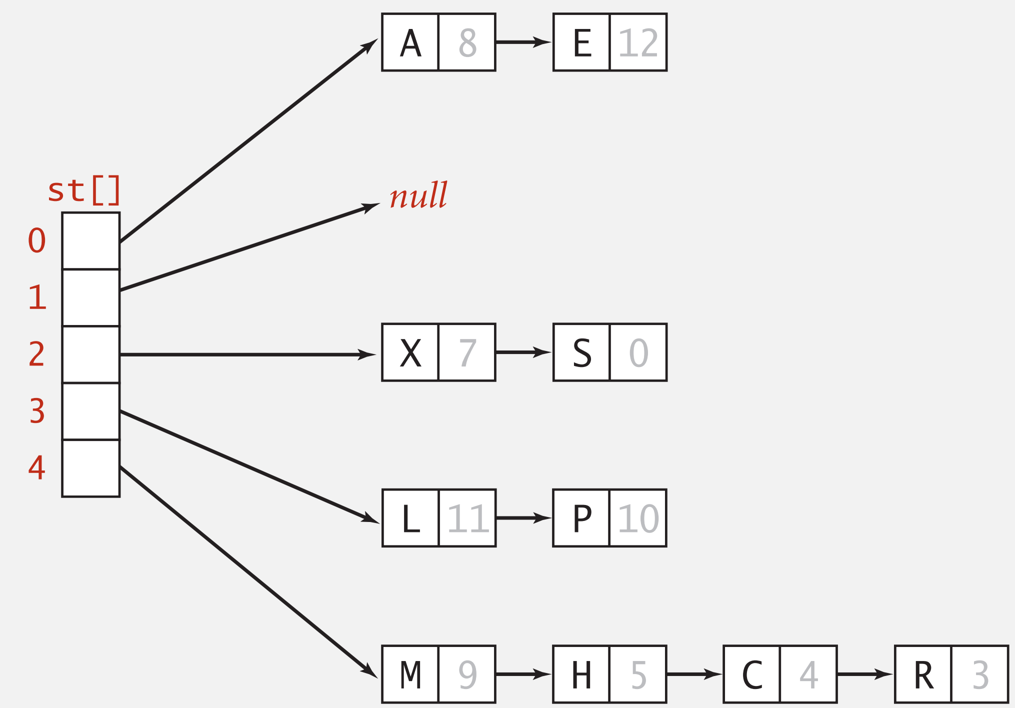
known to be insecure

Applications. File verification, digital signatures, cryptocurrencies, password authentication, blockchain, Git commit identifiers, ...

Separate chaining vs. linear probing

Separate chaining.

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



Linear probing.

- Less memory.
- Better cache performance.
- More probes because of clustering.

	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]

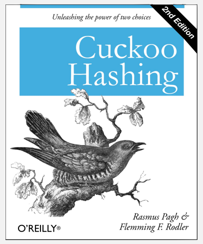
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\Theta(\log \log n)$.

Double hashing. [linear-probing variant]

- Resolve collisions by probing, but skip a variable amount instead of +1.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

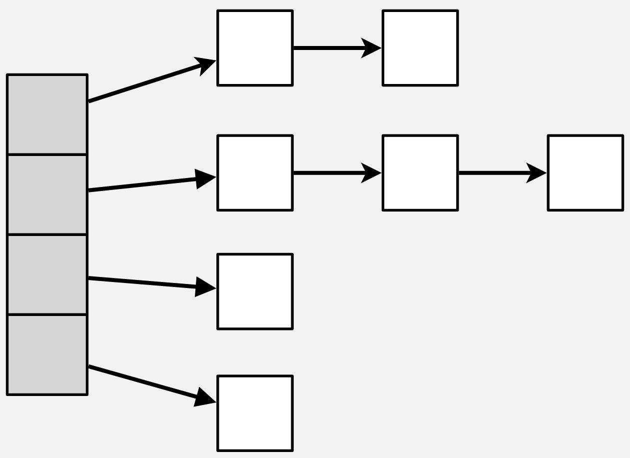
- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- $\Theta(1)$ time for search in worst case.



Hash tables vs. balanced search trees

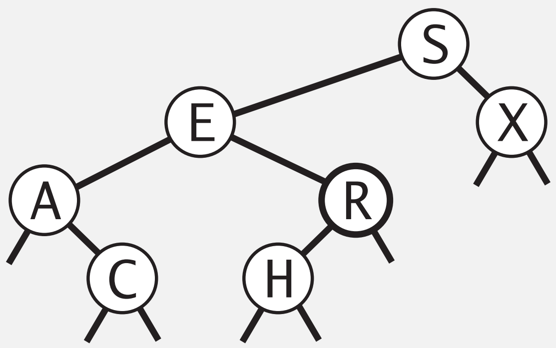
Hash tables.

- Simpler to code.
- Typically faster in practice.
- No effective alternative for unordered keys.



Balanced search trees.

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



Java system includes both.

- BSTs: `java.util.TreeMap`, `java.util.TreeSet`. ← red–black BST
- Hash tables: `java.util.HashMap`, `java.util.HashSet`.

↑ separate chaining (if chain gets too long, use red–black BST for chain)