



<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()

† subject to suitable technical assumptions

Q. Can we do better?

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

Hashing: basic plan

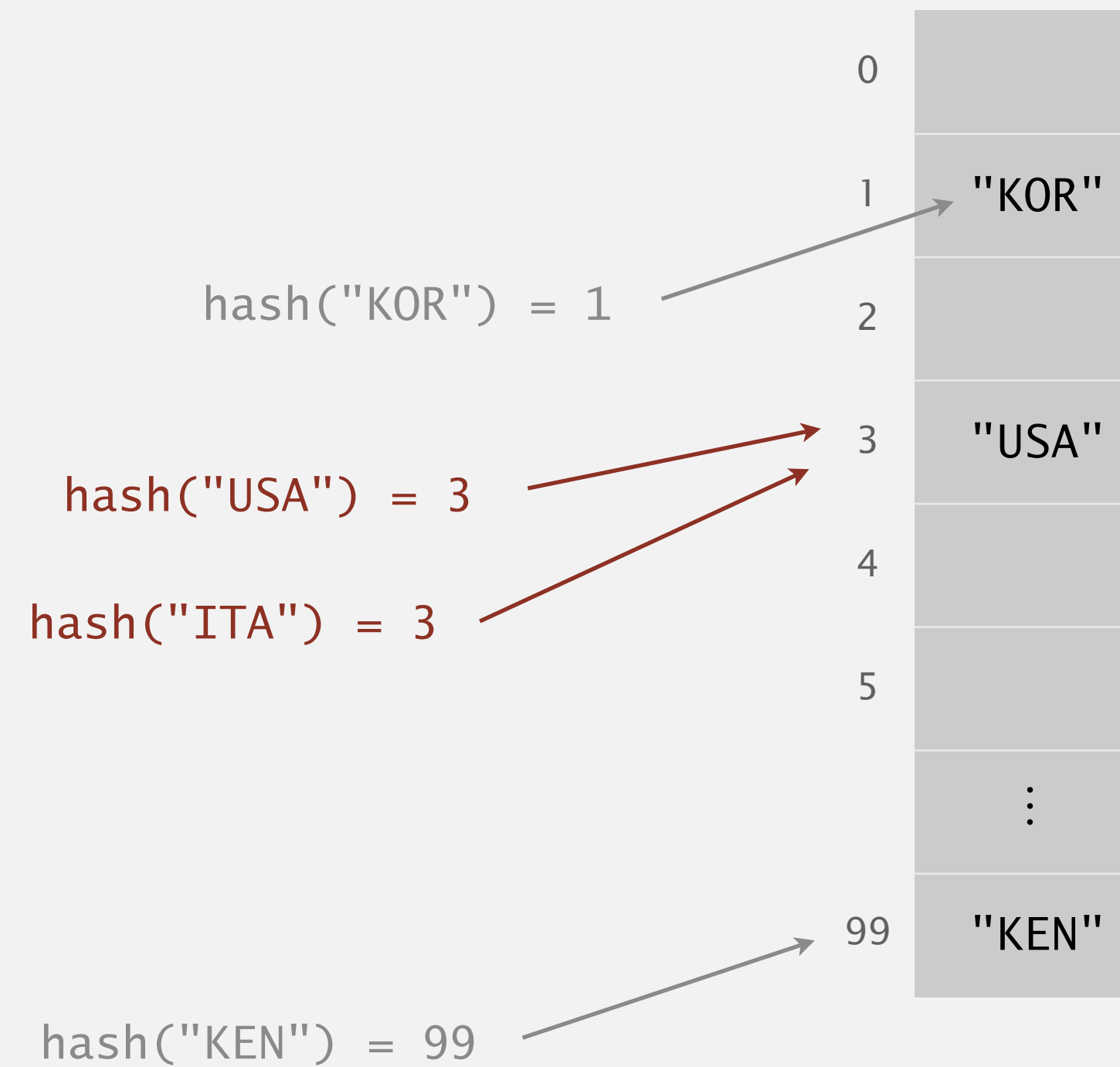
Save key–value pairs in a **key-indexed table** (index is a function of the key).

Hash function: Mathematical function that maps (**hashes**) a key to an array index.

Collision: Two distinct keys that hash to same index.

Issue. Collisions are unavoidable.

- How to limit collisions?
[good hash functions]
- How to accommodate collisions?
[novel algorithms and data structures]





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

3.4 HASH TABLES

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

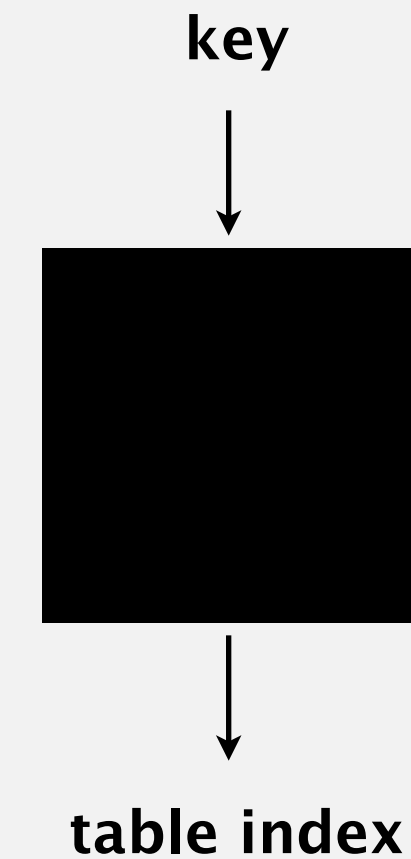
Designing a hash function

Required properties. [for correctness]

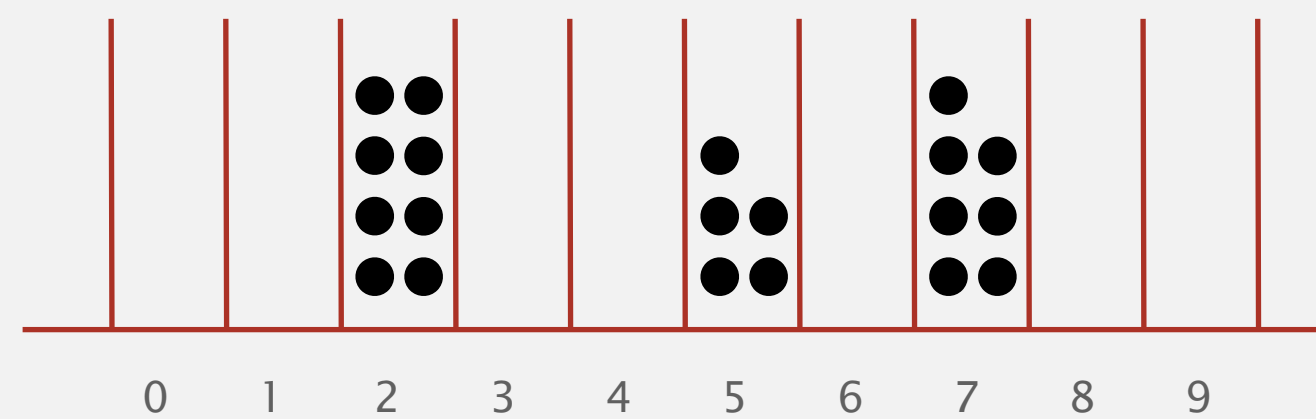
- Deterministic.
- Each key hashes to a table index between 0 and $m - 1$.

Desirable properties. [for performance]

- Very fast to compute.
- 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 poor hash-table performance
($m = 10, n = 20$)

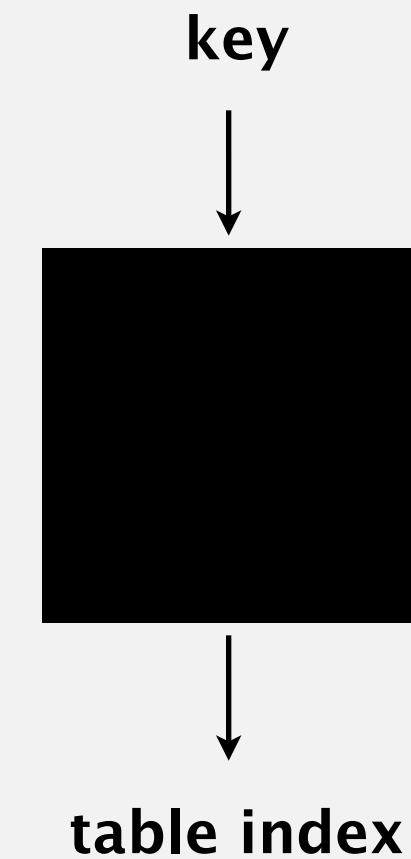
Designing a hash function

Required properties. [for correctness]

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

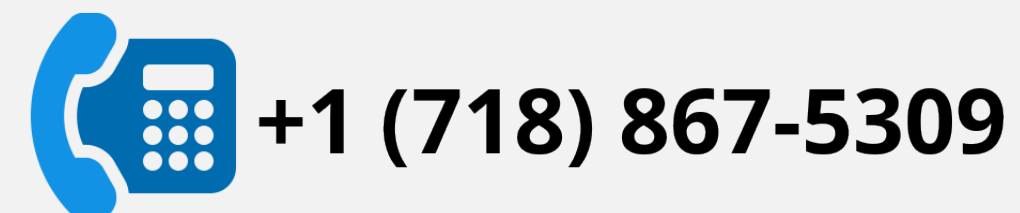
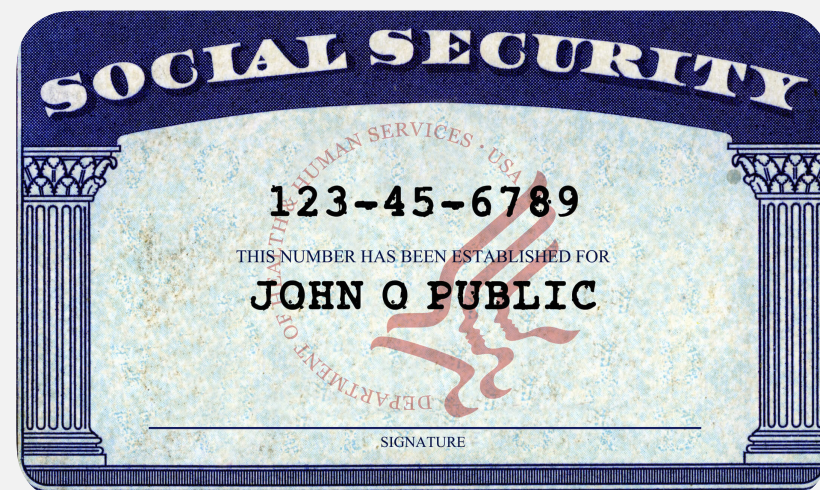
Desirable properties. [for performance]

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



Ex 1. [$m = 10,000$] Last 4 digits of U.S. Social Security number.

Ex 2. [$m = 10,000$] Last 4 digits of phone number.





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





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

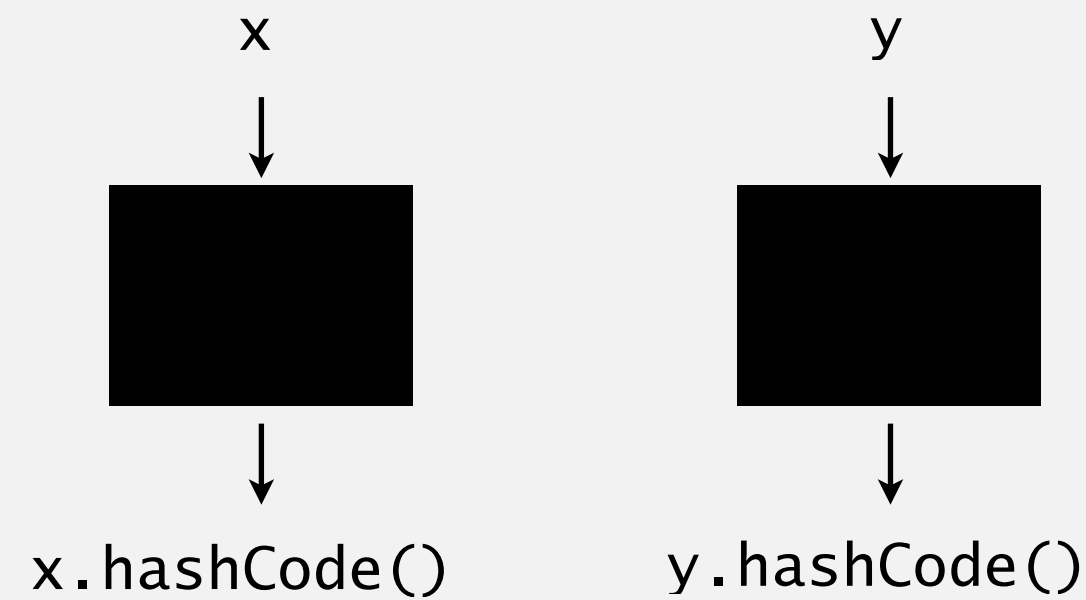


Java's hashCode() conventions

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

Required. [for correctness] If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. [for efficiency] If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



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

Legal (but highly 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));
    }
}
```

if used only least significant 32 bits,
all integers between -2^{21} and 2^{21}
would have same hash code (0)

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

Implementing hashCode(): user-defined types

31x + y rule.

- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add hash of each significant field.

```
public final class Transaction
{
    private final String who;
    private final Date when;
    private final double amount;

    ...

    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

31x + y rule.

- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add hash of each significant field.

```
public final class Transaction
{
    private final String who;
    private final Date when;
    private final double amount;
```

...

```
public int hashCode()
{
    return Objects.hash(who, when, amount);
}
```

a varargs method that applies
31x + y rule to its arguments

```
}
```

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 array of primitives.
- Shortcut 3: use `Arrays.deepHashCode()` for array of objects.



Principle. Every significant field contributes to hash.

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



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

A.

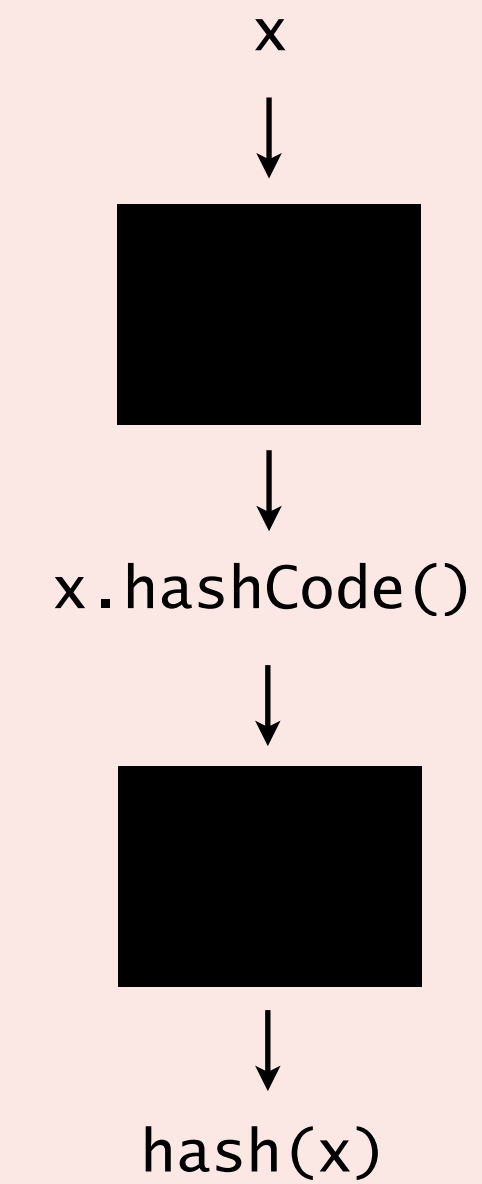
```
private int hash(Key key)
{ return key.hashCode() % m; }
```

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).

m typically a prime or a power of 2

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

bug

the remainder operator can evaluate to a negative integer

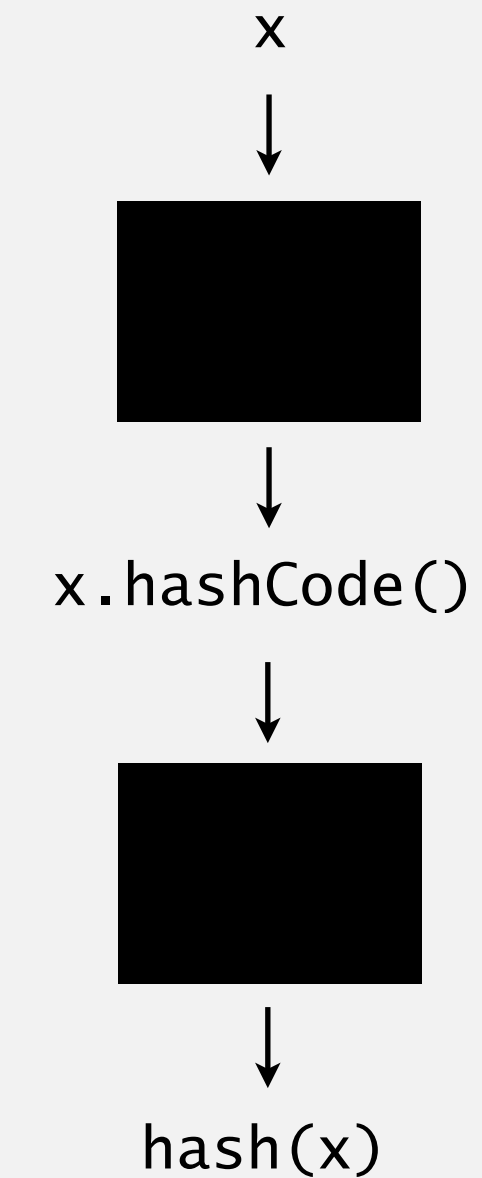
```
private int hash(Key key)
{ return Math.abs(key.hashCode()) % m; }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" and new Double(-0.0) is -2^{31}

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

correct

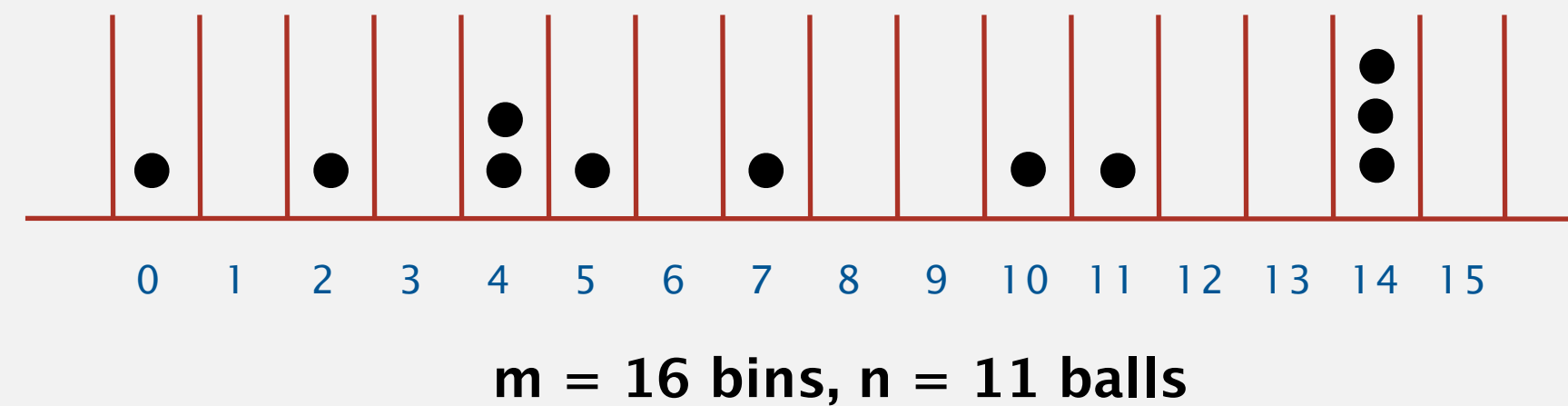


Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to any of m possible indices.

and independently of other keys

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



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.

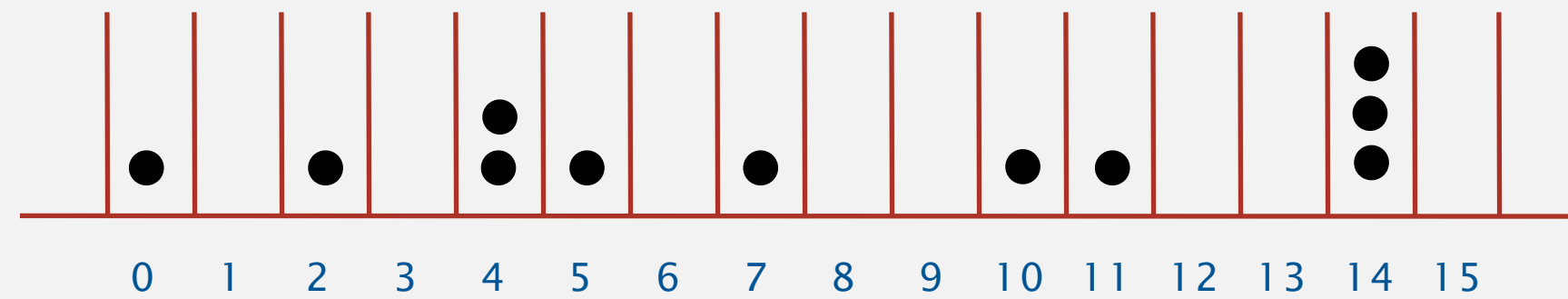
23.9 when $m = 365$

Uniform hashing assumption

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

and independently of other keys

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

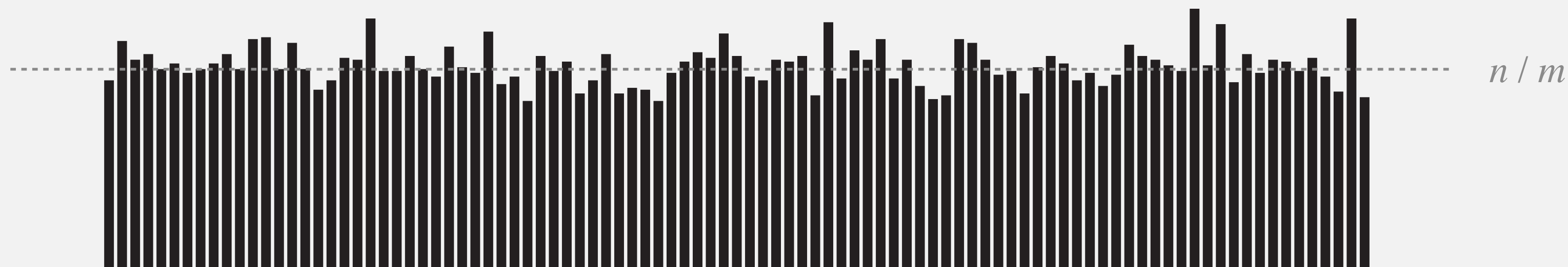


$m = 16$ bins, $n = 11$ balls

Good news. [load balancing]

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

Binomial($n, 1/m$)



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



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

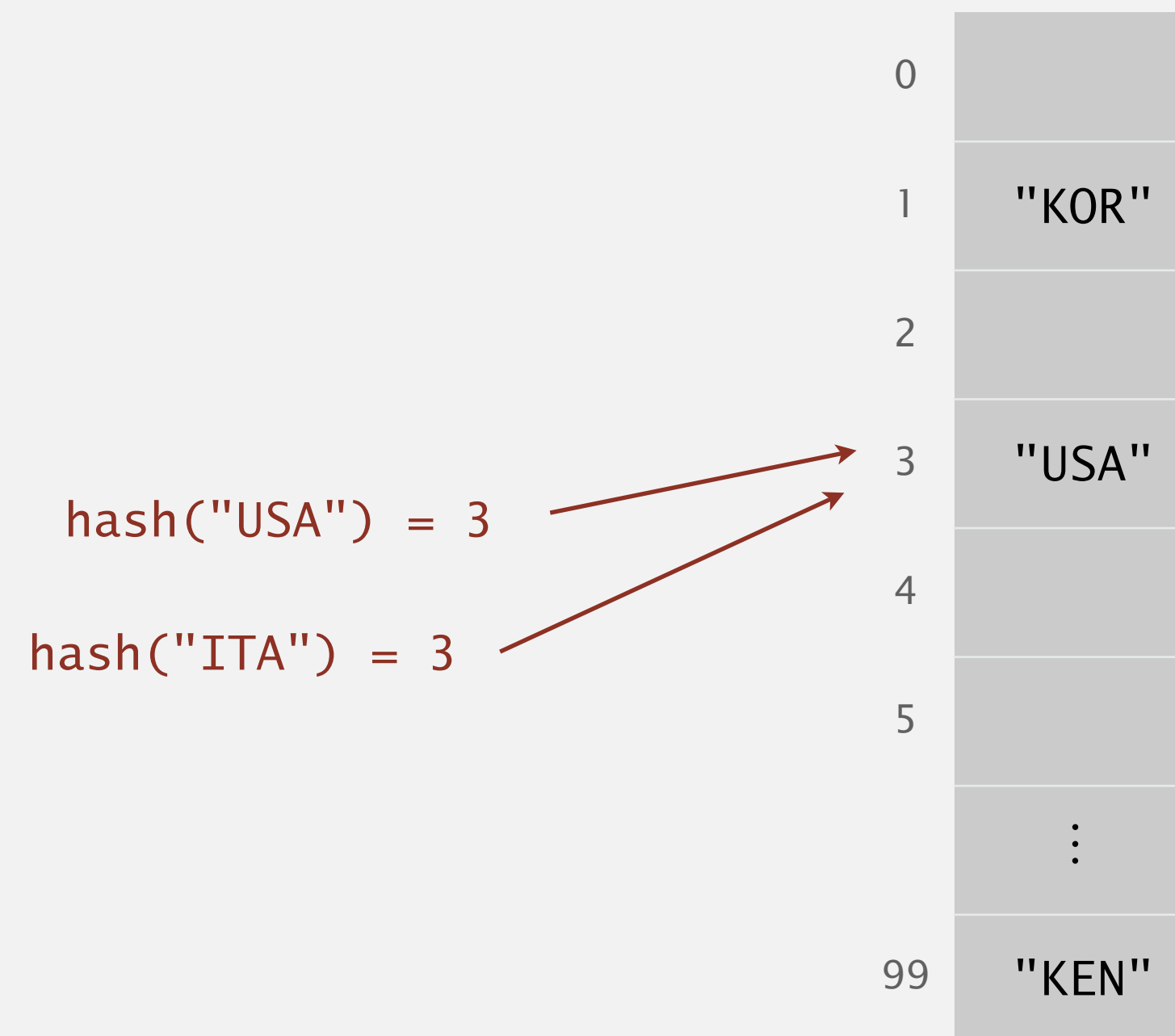
3.4 HASH TABLES

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

Collisions

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

- Birthday problem \Rightarrow can't avoid collisions. \leftarrow unless you have a ridiculous (quadratic) amount of memory
- Load balancing \Rightarrow no index gets too many collisions.
 \Rightarrow ok to scan through all colliding keys.



Separate-chaining hash table

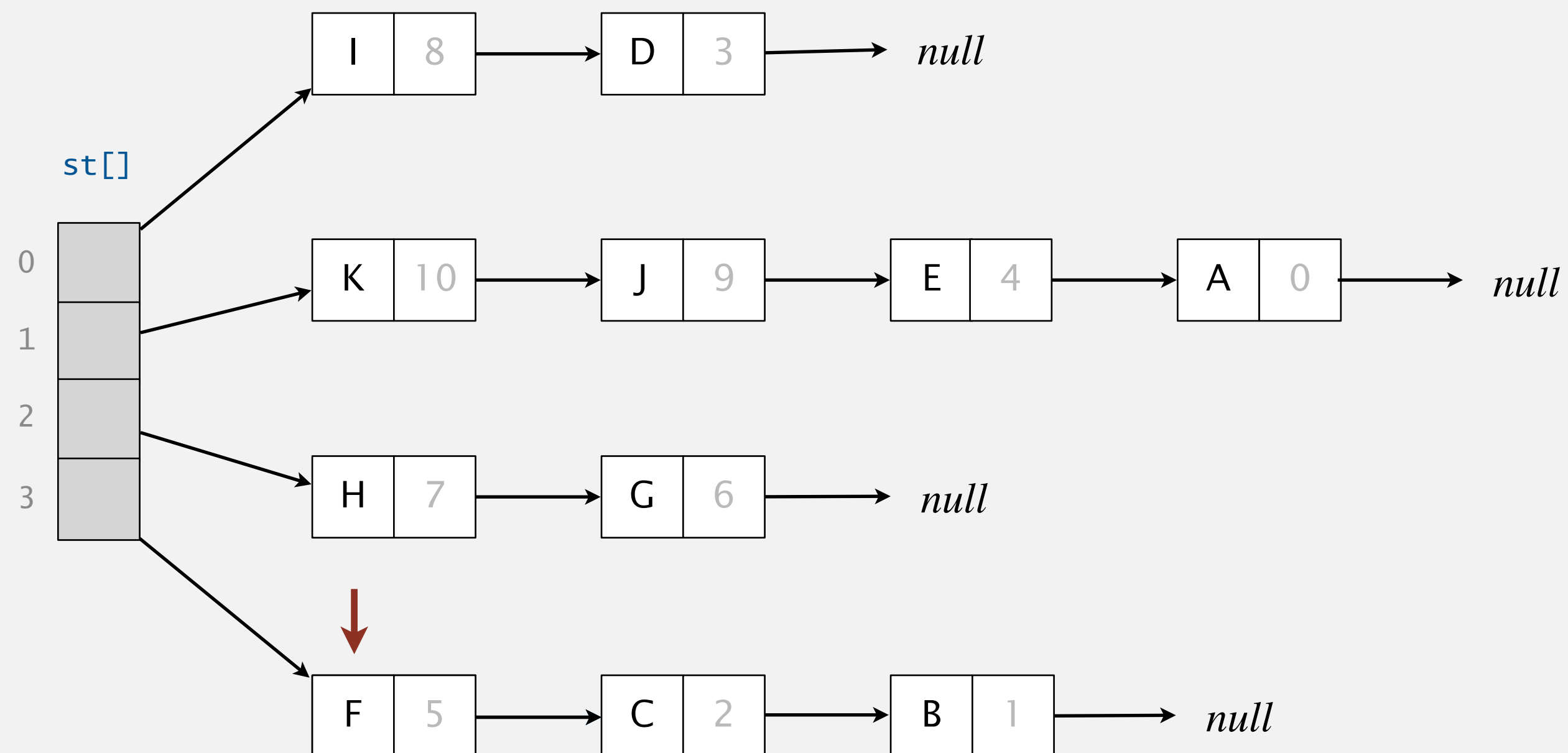
Use an array of m linked lists.

- 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).

put(L, 11)

hash(L) = 3

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



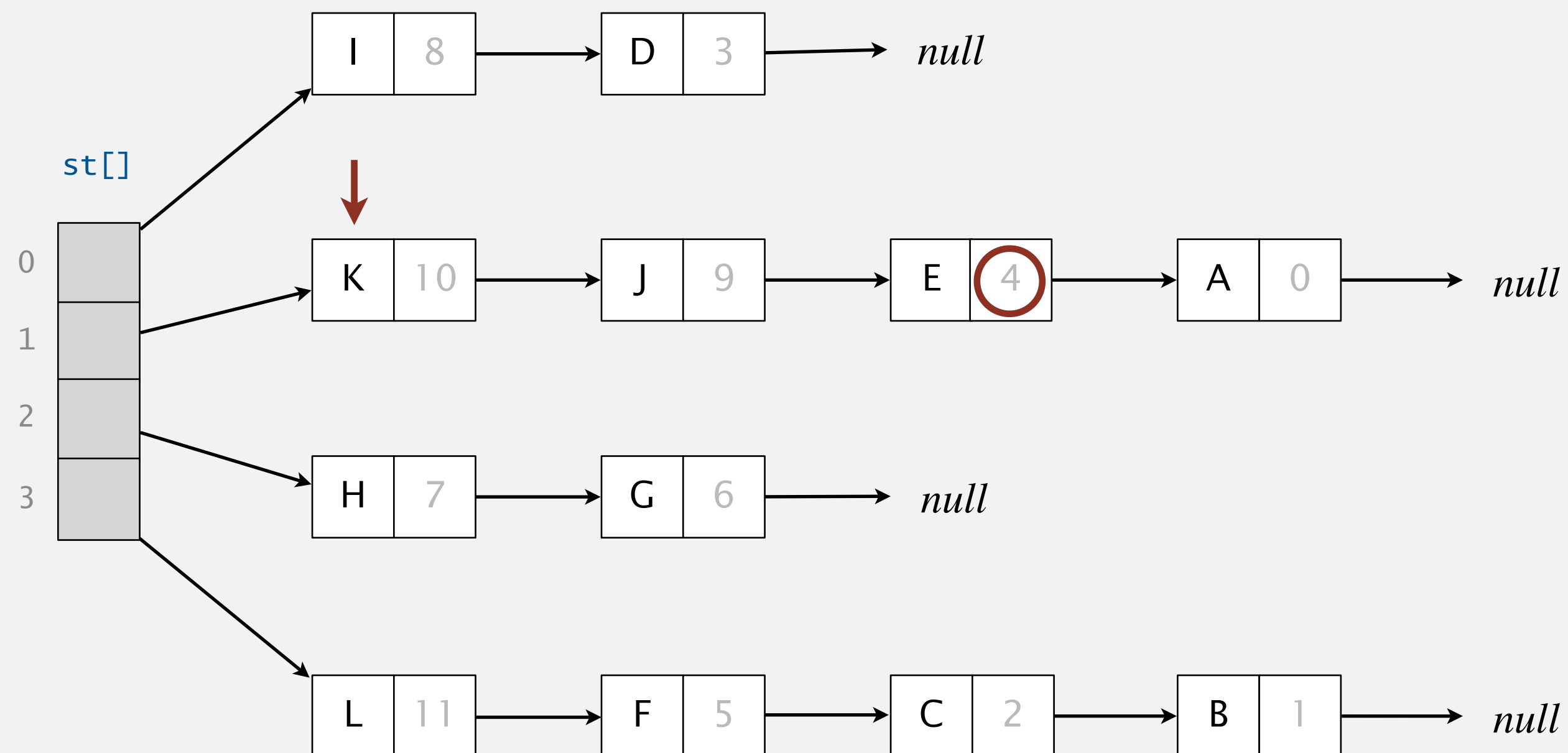
Separate-chaining hash table

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 ($m = 4$)

get(E)
hash(E) = 1



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)
    { /* as before */ }

    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

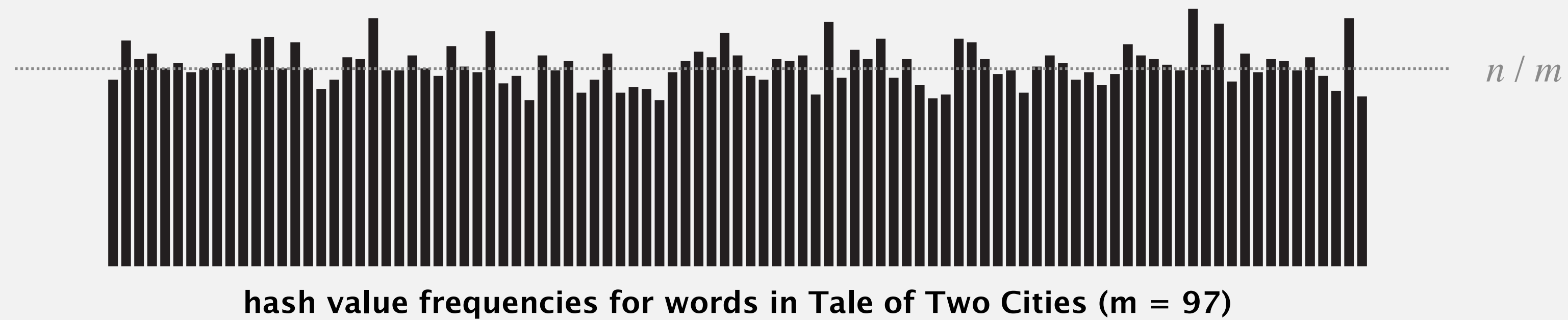
    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    private int hash(Key key)
    { /* as before */ }

    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 .



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.



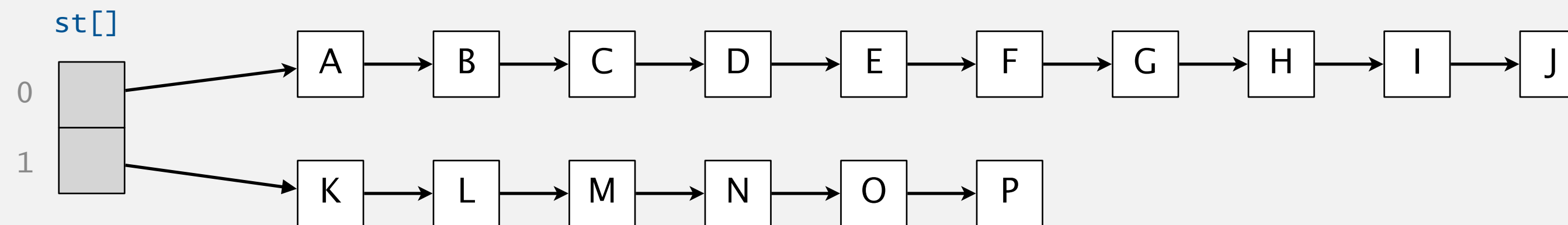
m times faster than
sequential search

Resizing in a separate-chaining hash table

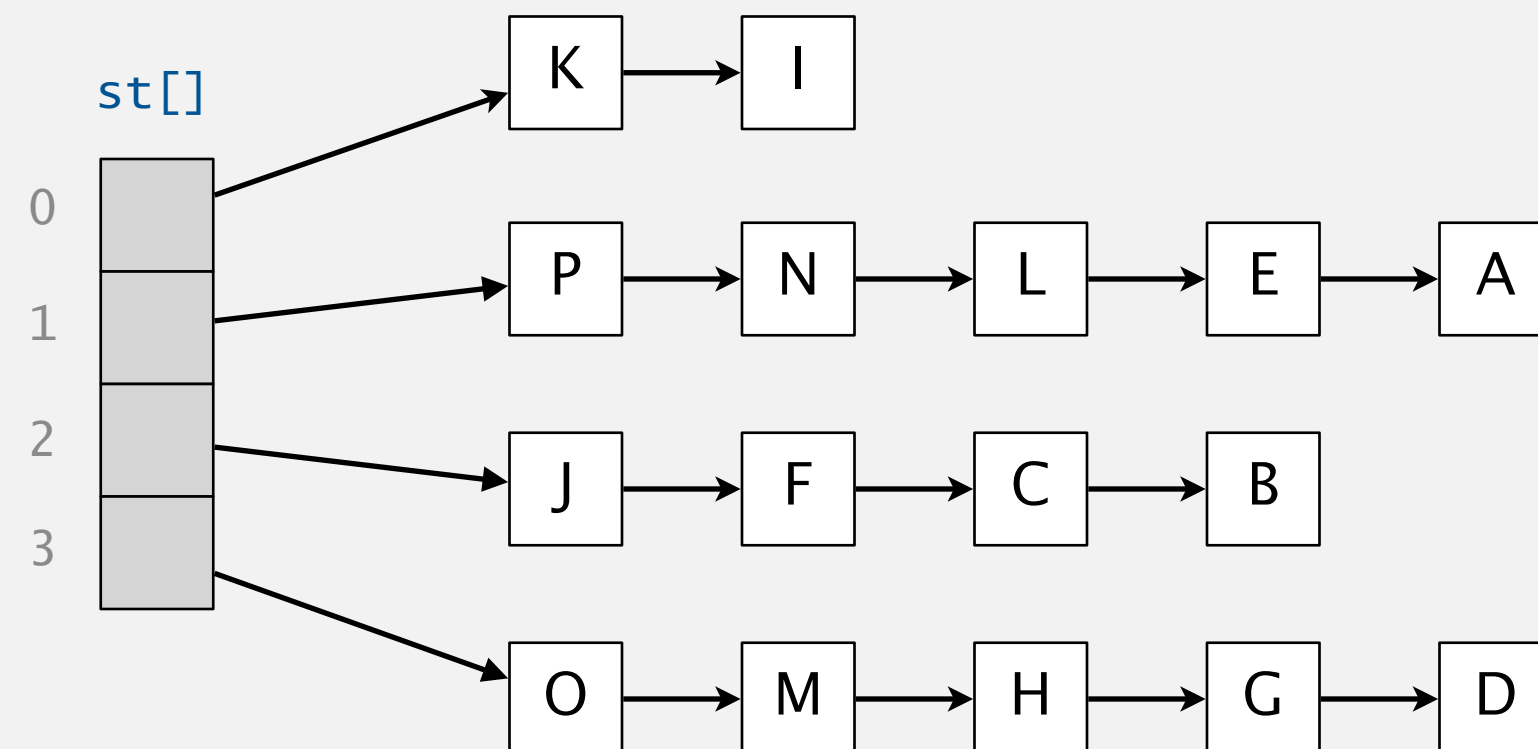
Goal. Average length of chain n / m is $\Theta(1)$.

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

before resizing ($n/m = 8$)



after resizing ($n/m = 4$)



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^\dagger	1^\dagger	1^\dagger		equals() hashCode()

† under uniform hashing assumption

can achieve $\Theta(1)$ probabilistic, amortized guarantee
by choosing a hash function at random
(see “universal hashing”)



How much memory for a SeparateChainingHashST with m chains and n key-value pairs?

- A. $\sim 8m + 24n$ bytes
- B. $\sim 8m + 32n$ bytes
- C. $\sim 8m + 40n$ bytes
- D. $\sim 48n$ bytes

```
public class SeparateChainingHashST<Key, Value>
{
    private int m = ...;
    private Node[] st = new Node[m];
    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
    }
    ...
}
```



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

3.4 HASH TABLES

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

Linear-probing hash table: insert

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by **linear 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

Linear-probing hash table: search

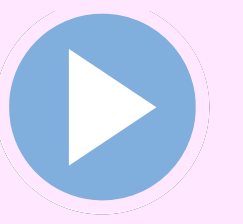
- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by **linear probing**:
search successive cells until either finding the key or an unused cell.

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

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 $i + 1, i + 2, \dots$

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

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)
    { /* 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 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)
    { /* as before */ }
```

```
    public 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;
    }
```

```
}
```

← array resizing
code omitted



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		A	D		K			A	B	E	F	G	J		C	L

- A. Index 4.
- B. Index 17.
- C. Either index 4 or 17.
- D. All open indices are equally likely.

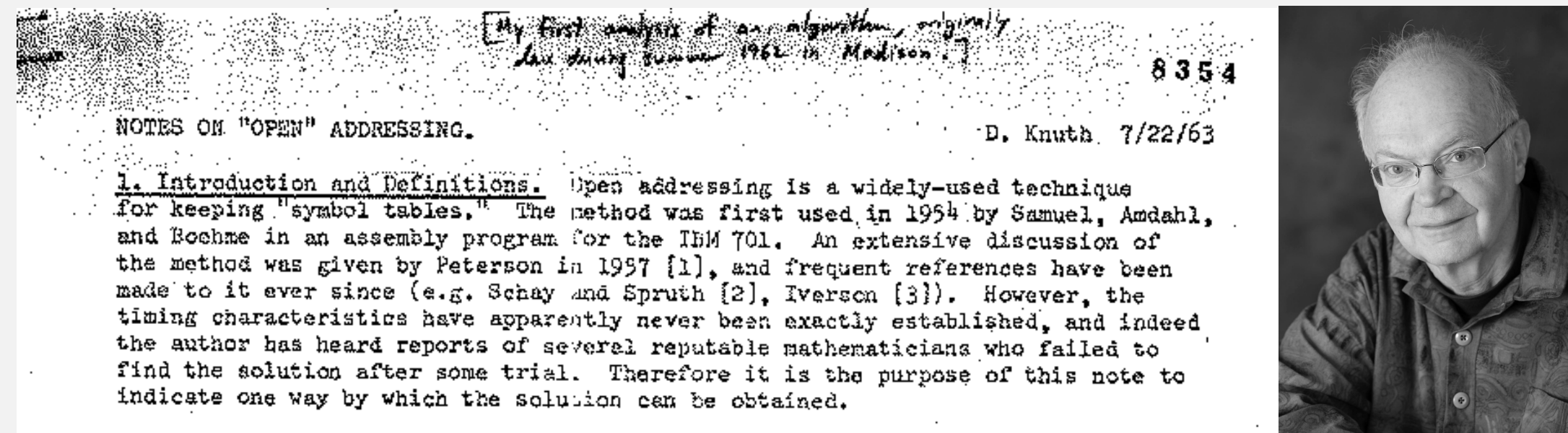
Analysis of linear probing

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

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

search hit search miss / insert

Pf. [beyond course scope]



Parameters.

- m too large \Rightarrow too many empty array entries.
- m too small \Rightarrow search time blows up.
- Typical choice: $\alpha = n / m \sim 1/2$. \longleftarrow # probes for search hit is about $3/2$
probes for search miss is about $5/2$

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^\dagger	1^\dagger	1^\dagger		equals() hashCode()
linear probing	n	n	n	1^\dagger	1^\dagger	1^\dagger		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

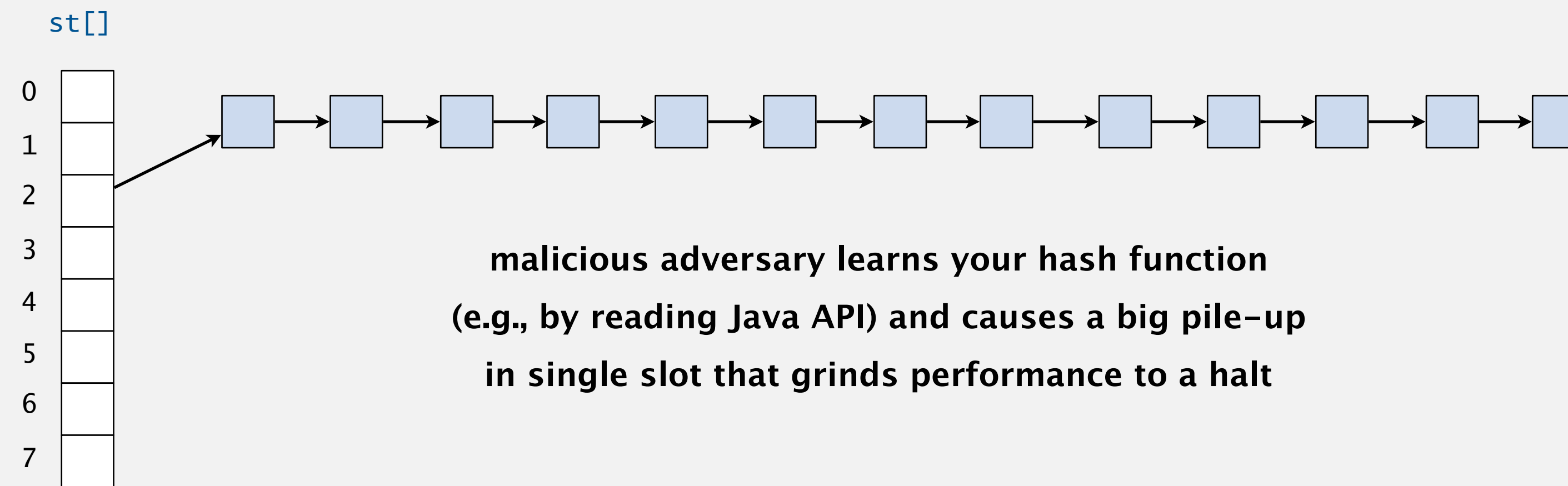
- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ ***context***

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:

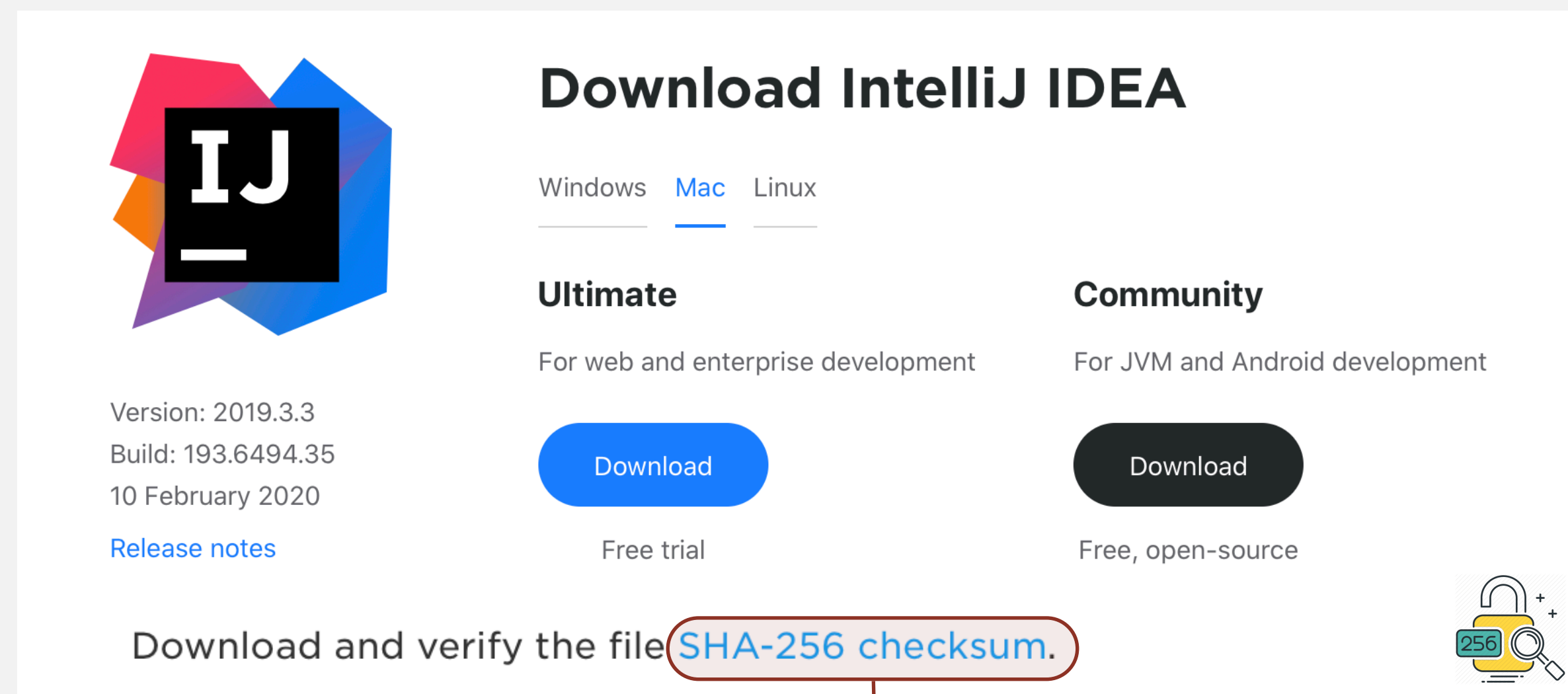
http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf

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)



Download IntelliJ IDEA

Windows **Mac** Linux

Ultimate
For web and enterprise development
[Download](#)
Free trial

Community
For JVM and Android development
[Download](#)
Free, open-source

Version: 2019.3.3
Build: 193.6494.35
10 February 2020
[Release notes](#)

Download and verify the file [SHA-256 checksum](#).

c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270

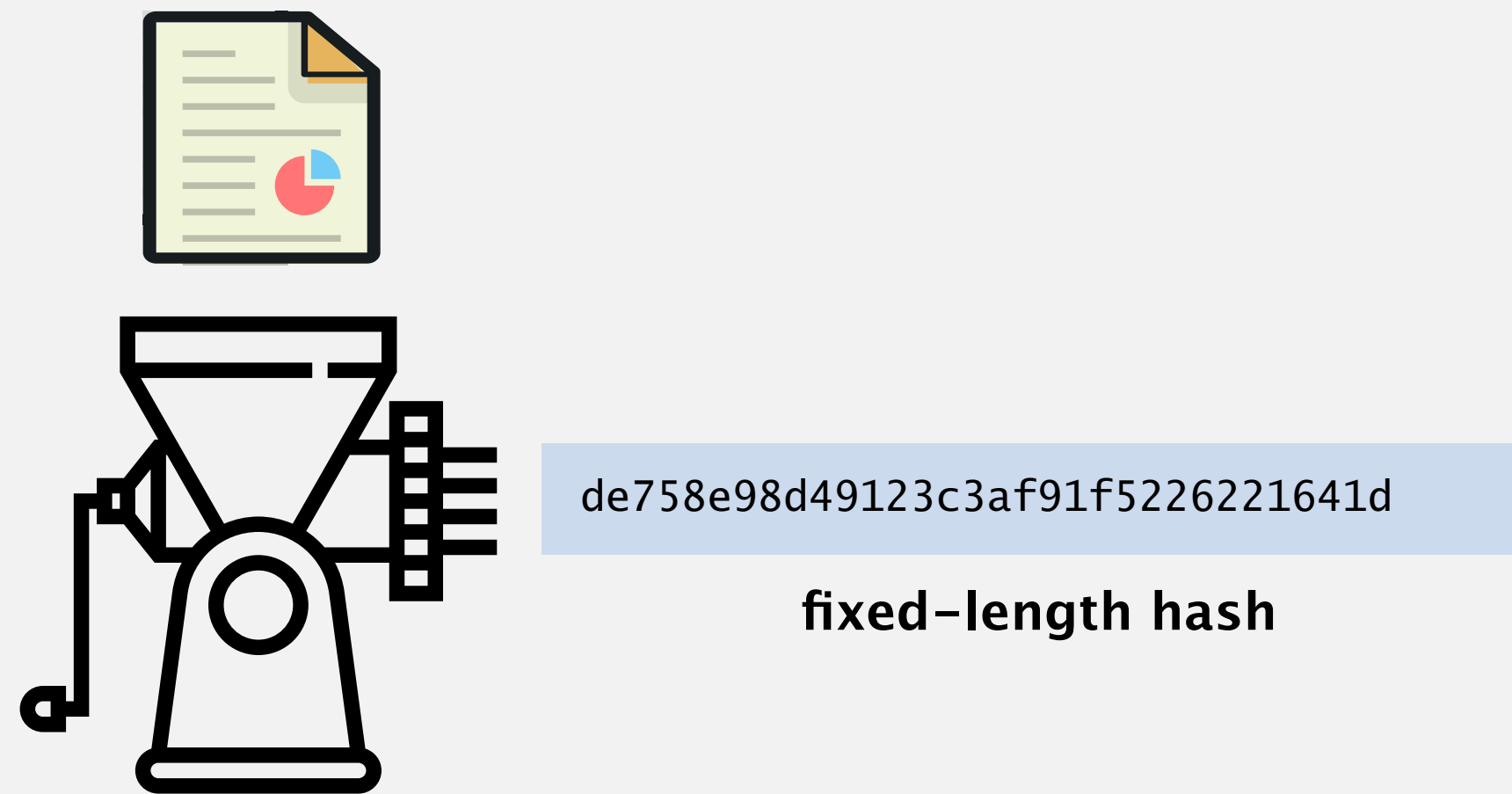
```
~/Desktop> sha256sum ideaIC-2019.3.3.dmg  
c62ed2df891ccbb40d890e8a0074781801f086a3091a4a2a592a96afaba31270
```

Hashing: cryptographic applications

One-way hash function. “Hard” to find a key that will hash to a target 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, non-fungible tokens, Git commit identifiers,

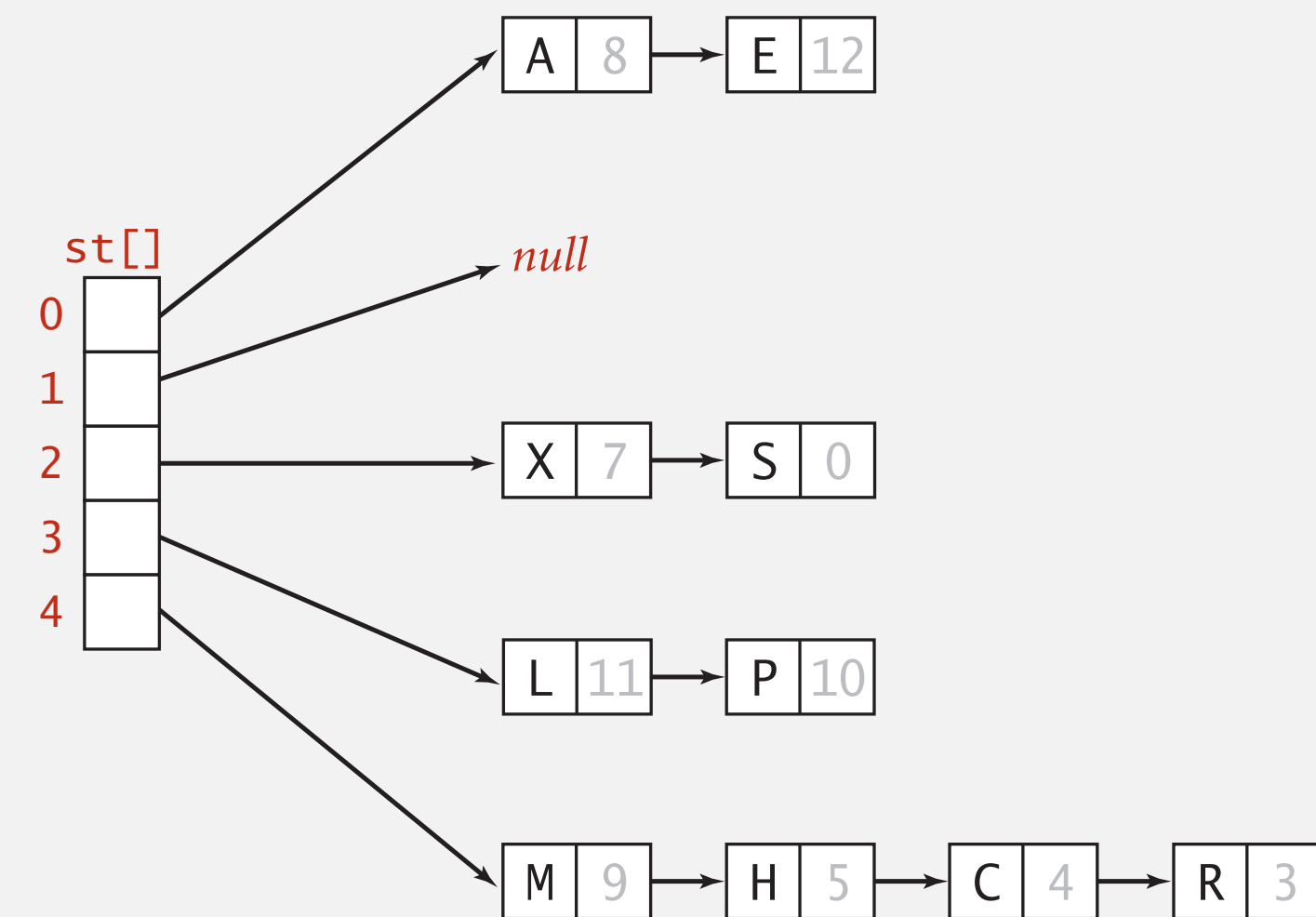
Separate chaining vs. linear probing

Separate chaining.

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

Linear probing.

- Less memory.
- Unrivaled data locality.
- More probes because of clustering.



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

Hashing: variations on the theme

Many many improved versions have been studied.

Use different probe sequence, i.e., not $h(k), h(k) + 1, h(k) + 2, \dots$

[quadratic probing, double hashing, pseudo-random probing, ...]

← eliminates primary clustering
(but sacrifices data locality)

Google Swiss Table

Facebook F14

Python 3

During insertion, relocate some of the keys already in the table.

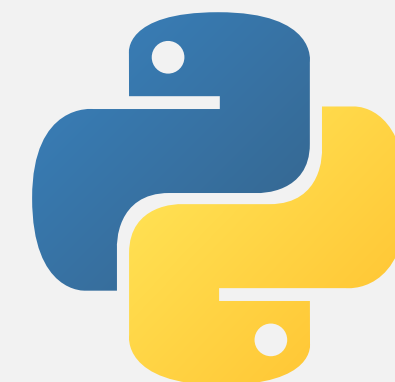
[Cuckoo hashing, Robin Hood hashing, Hopscotch hashing, ...]

← reduces worst-case time for search

Insert tombstones prophylactically, to avoid primary clustering.

[graveyard hashing]

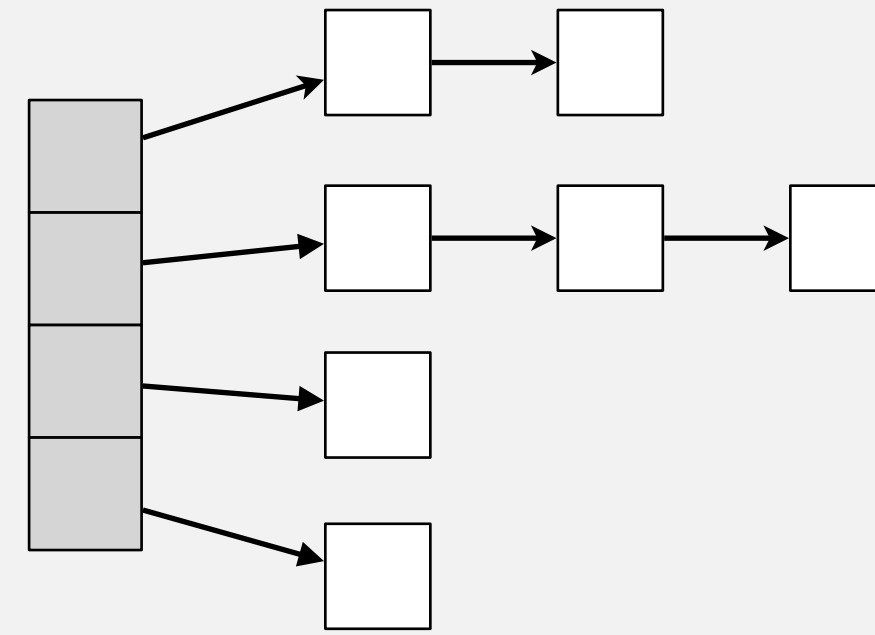
← eliminates primary clustering;
maintains data locality



Hash tables vs. balanced search trees

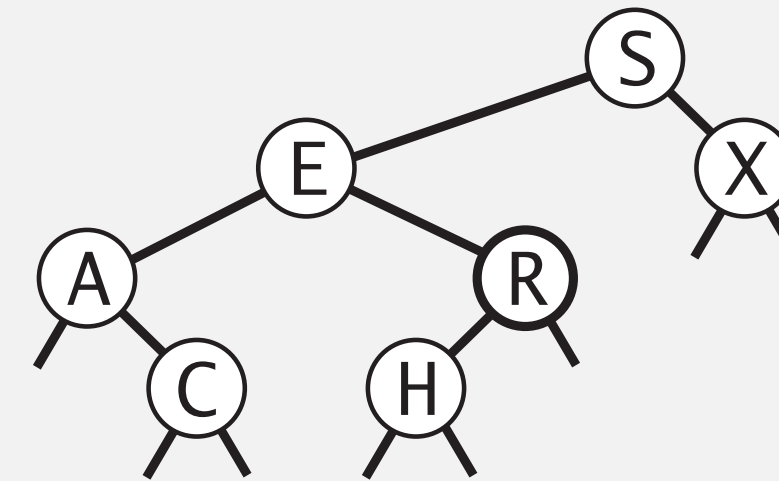
Hash tables.

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



Balanced search trees.

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



Java includes both.

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

↑
separate chaining
(Java 8: if chain gets too long,
use red-black BST for chain)

↑
linear probing

© Copyright 2021 Robert Sedgewick and Kevin Wayne