

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

R u t g e r s U n i v e r s i t y

HASH TABLES

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

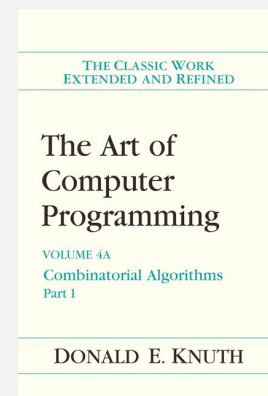
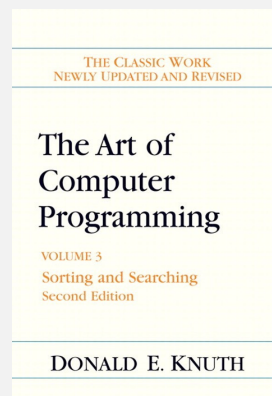
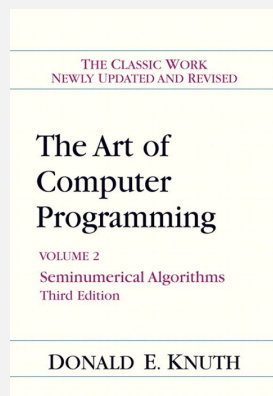
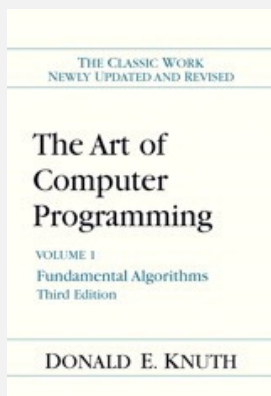


Wise Advice from Don Knuth

“ Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. ”



Symbol table implementations: summary

LO 8.1

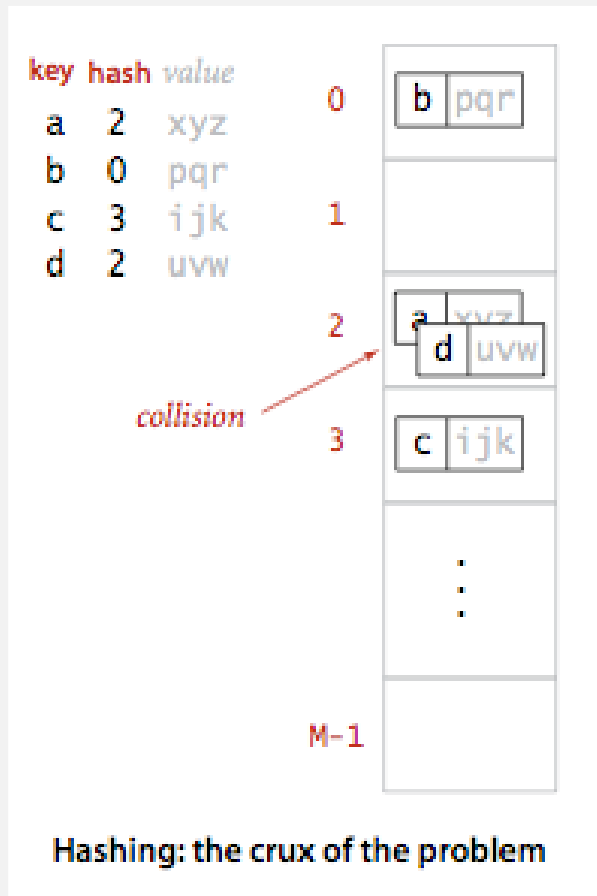
implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	n	n	n	n	n	n		equals()
binary search(ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	\sqrt{n}	✓	compareTo()
red-black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

A Simple Hashing Idea

- If keys are small integers, we can use an array to implement a symbol table
- Interpret the key as an array index and store the value associated with key i in array position i .
- The issue: Need a table of size 2^{32} to store 4-byte integers (too large)



Another issue with hashing?
Two keys cannot hash to same
location - collision

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.

$\text{hash}(\text{"it"}) = 3$

$\text{hash}(\text{"times"}) = 3$

0	
1	
2	
3	"it"
4	??
5	

Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

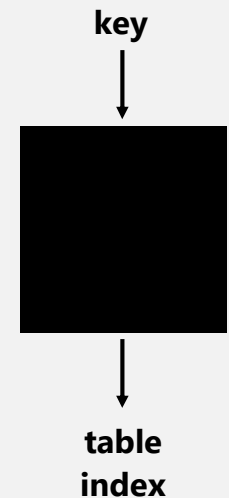
HASH TABLES

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

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

- It should be deterministic – equal keys must produce the same value
- Efficiently computable.
- Each table index equally likely for each key.

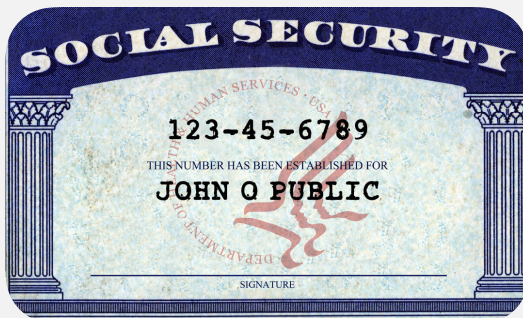
← thoroughly researched problem,
still problematic in practical applications



Ex 1. Last 4 digits of Social Security number.

Ex 2. Last 4 digits of phone number.

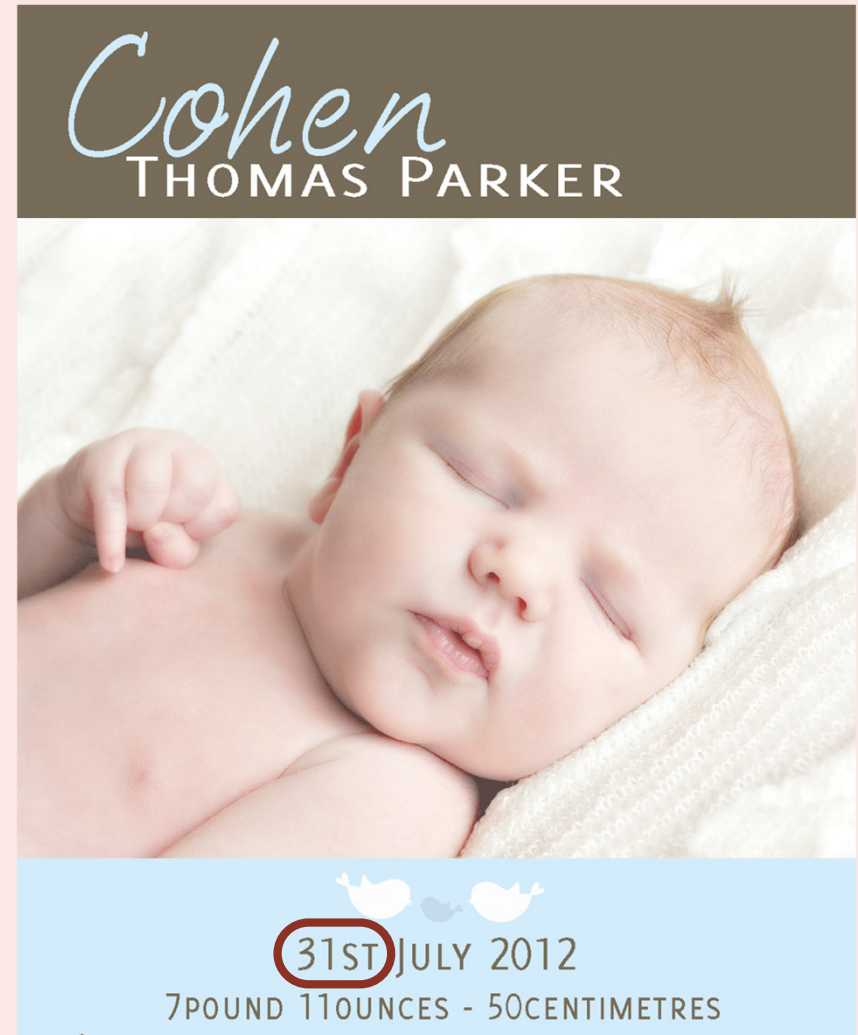
Practical challenge. Need different approach for each key type.



Hash tables: quiz 1

Which is the last digit of your **day** of birth?

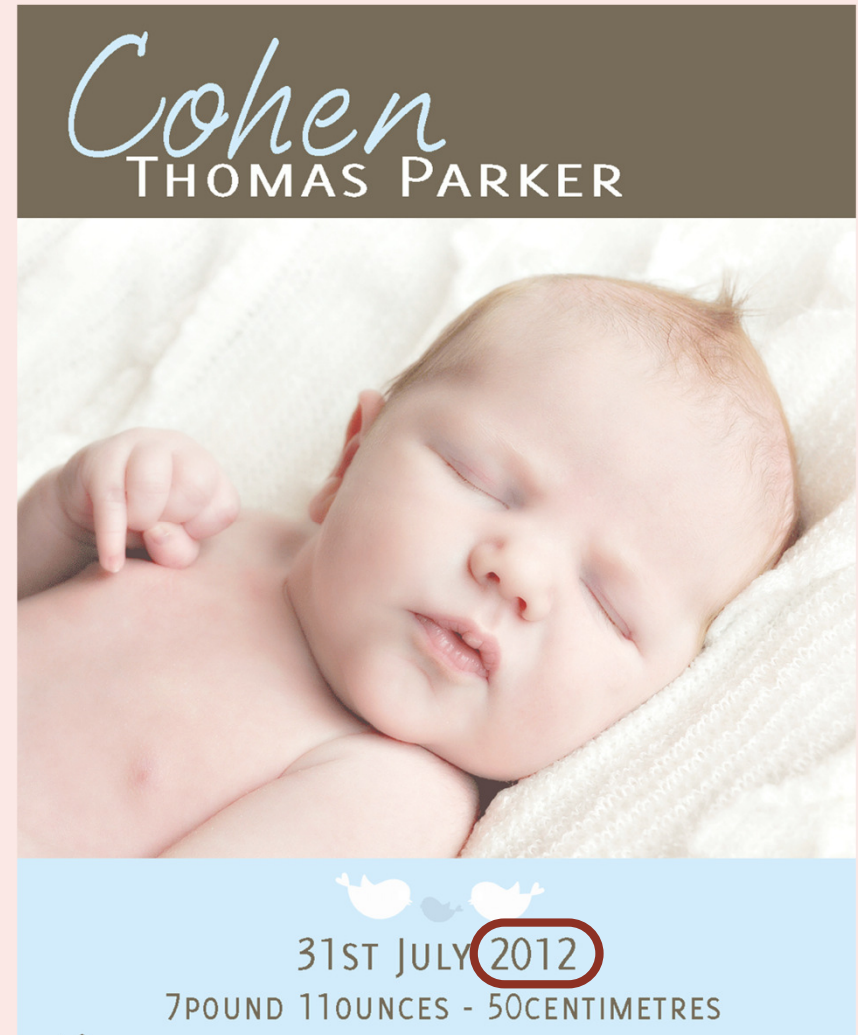
- A. 0 or 1
- B. 2 or 3
- C. 4 or 5
- D. 6 or 7
- E. 8 or 9



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

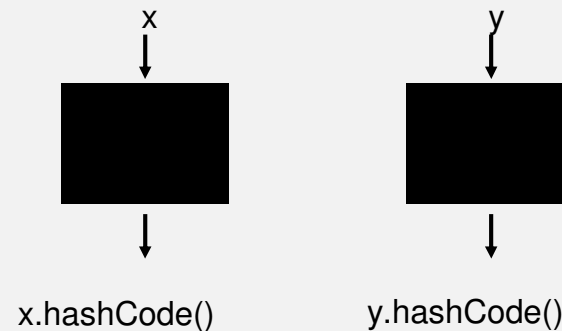


Java's hash code conventions

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, URL, LocalTime, ...

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

LO 8.2

Java library implementations

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

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

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

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

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

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

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

Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

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

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


```
public final class String
{
    private final char[] s;
    :
    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

Java library implementation

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

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

```
String s = "call";
s.hashCode();
```


$$\begin{aligned} 3045982 &= 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\ &= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \end{aligned}$$

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;
    }
}
```

Q. What if hashCode() of string is 0?

← hashCode("pollinating sandboxes")

Implementing hash code: user-defined types

LO 8.2

```
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)
```

```
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



typically a small prime

for reference types,
use hashCode()

for primitive types,
use hashCode()
of wrapper type

Hash code design

“Standard” recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, use 0.
- If field is a reference type, use `hashCode()`.  applies rule recursively
- If field is an array, apply to each entry.  or use `Arrays.deepHashCode()`

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

In theory. Keys are bitstring; “universal” family of hash functions exist.

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

Basic rule. Need to use the whole key to compute hash code;
consult an expert for state-of-the-art hash codes.

Hash tables: quiz 3

Which function(s) map hashable key to integers between 0 and $m-1$?

A.

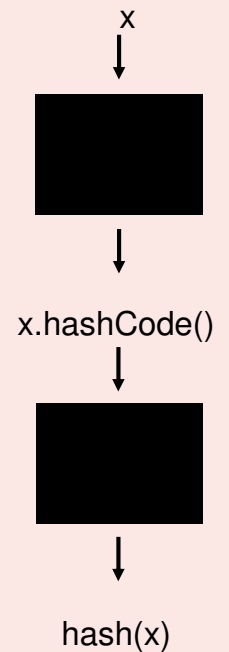
```
private int hash(Key key)
{ return key.hashCode() % m; }
```

B.

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

C. Both A and B.

D. Neither A nor B.



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 is typically a prime or power of 2 is the hash table size

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

bug

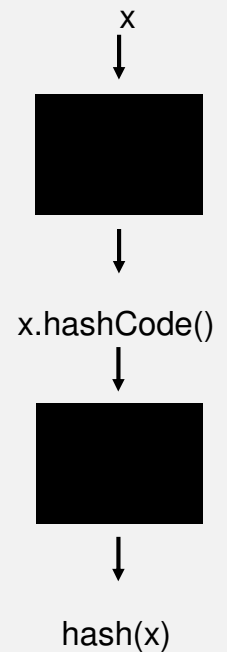
```
private int hash(Key key)
{ return Math.abs(key.hashCode()) % m; }
```

1-in-a-billion bug

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

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

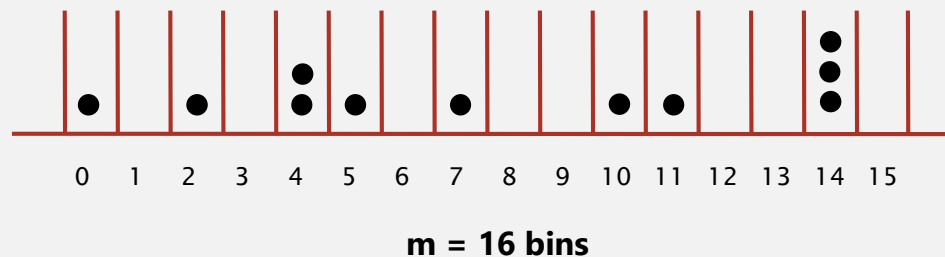
correct



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.

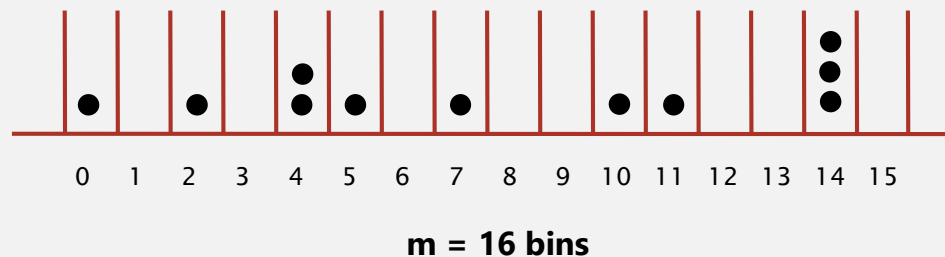
Coupon collector. Expect every bin has ≥ 1 ball after $\sim m \ln m$ tosses.

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

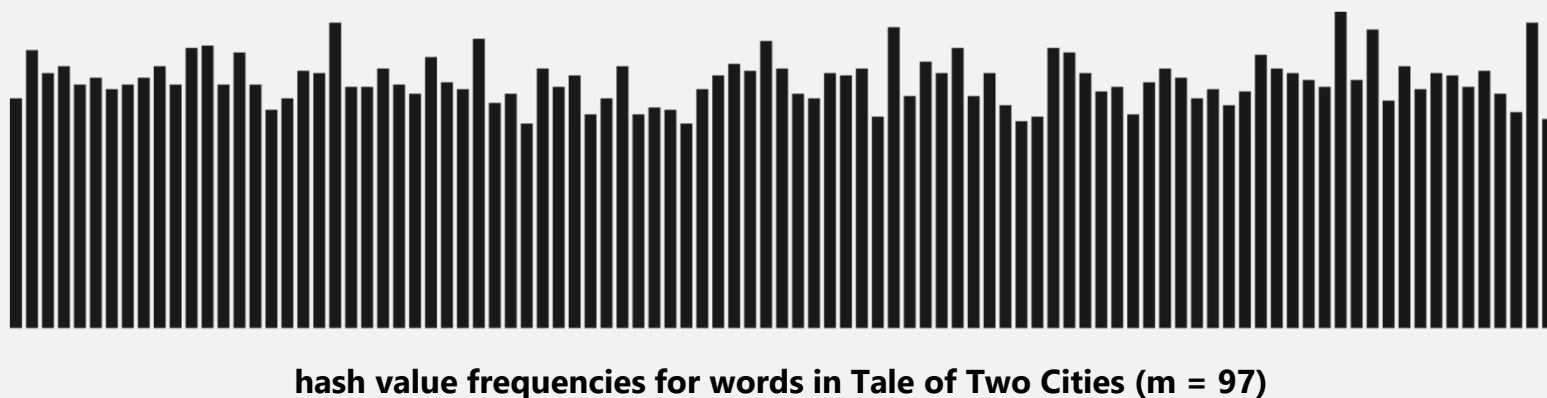
Uniform hashing assumption

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.



Ex. String data type.

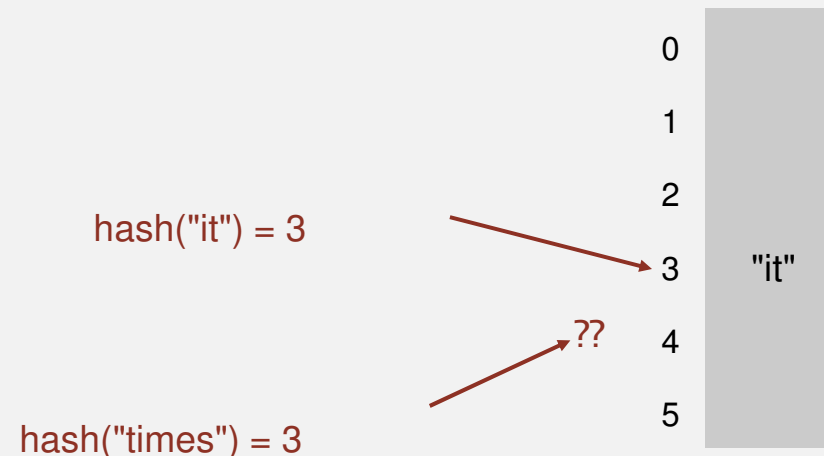


HASH TABLES

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

Collision. Two distinct keys hashing to same index.

- Birthday problem \Rightarrow can't avoid collisions. ← unless you have a ridiculous (quadratic) amount of memory
- Coupon collector \Rightarrow not too much wasted space.
- Load balancing \Rightarrow no index gets too many collisions.
- Load factor \Rightarrow a ratio of number of keys : table size (keep the load factor a reasonable size)



Separate-chaining symbol table

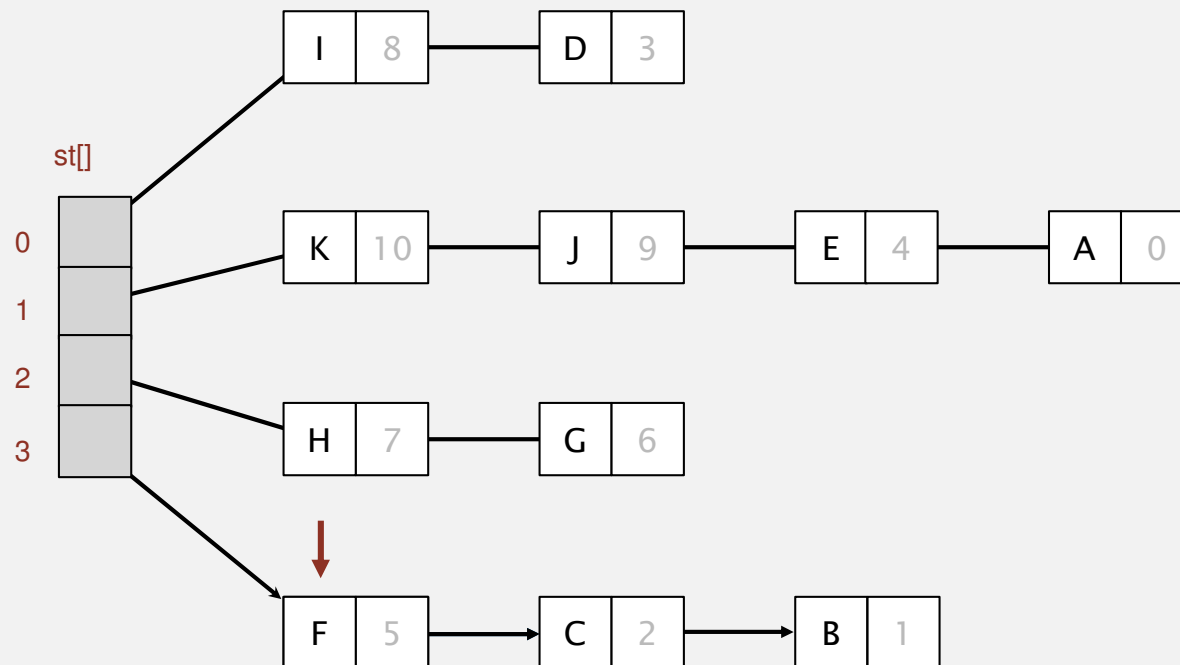
LO 8.6

Use an array of m 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: sequential search in i^{th} chain.

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

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



Separate-chaining symbol table

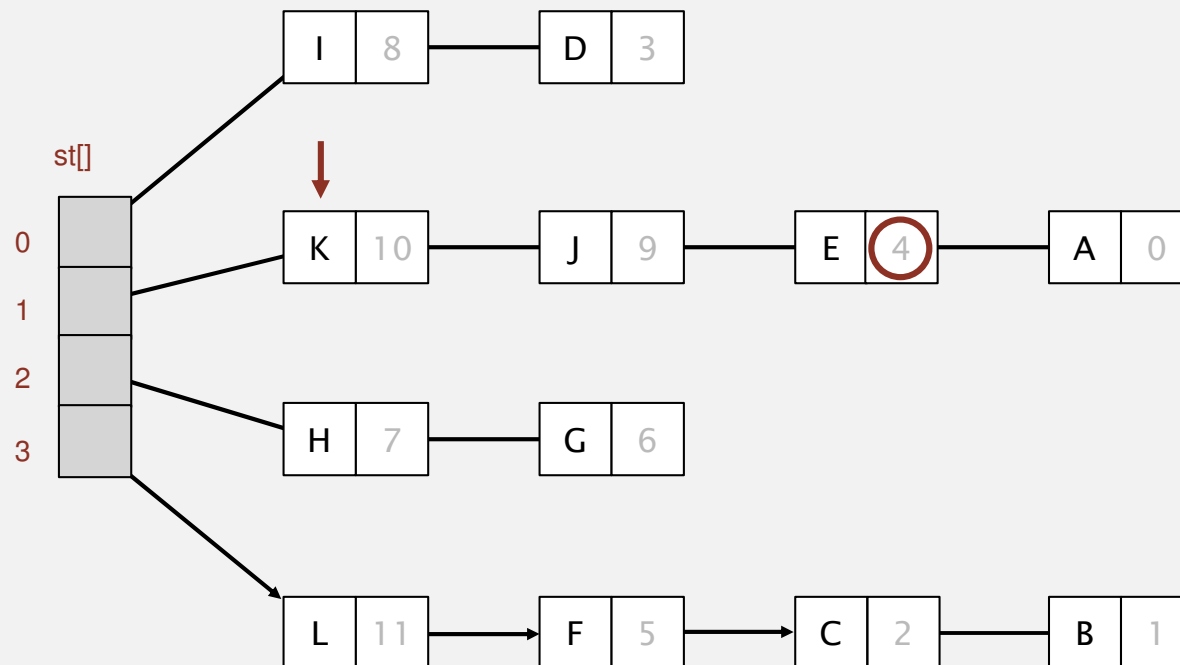
LO 8.6

Use an array of m 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: sequential search in i^{th} chain.

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

get(E)
hash(E) = 1



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
```

← array doubling and
halving code omitted

```
    private static class Node
```

```
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }
```

← no generic array creation

← (declare key and value of type Object)

```
    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;
    }
```


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

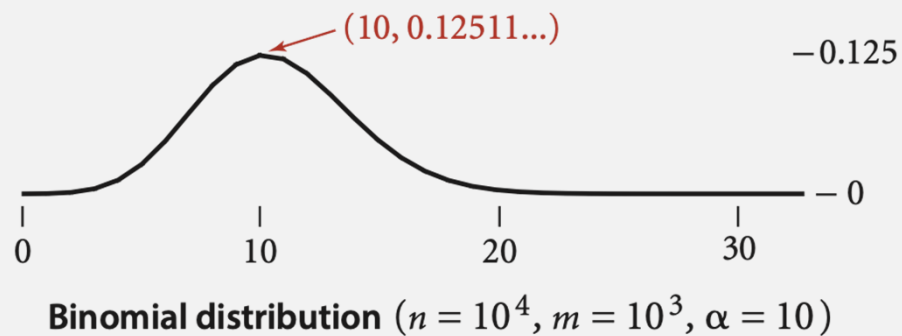
    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    public void put(Key key, Value val)
    {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }
}
```

Analysis of separate chaining

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

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



Consequence. Number of **probes** for search/insert is proportional to n / m .

- m too large \Rightarrow too many empty chains.
- m too small \Rightarrow chains too long.
- Typical choice: $m \sim \frac{1}{4} n \Rightarrow$ constant-time ops.

equals() and hashCode()

↑
 m times faster than
sequential search

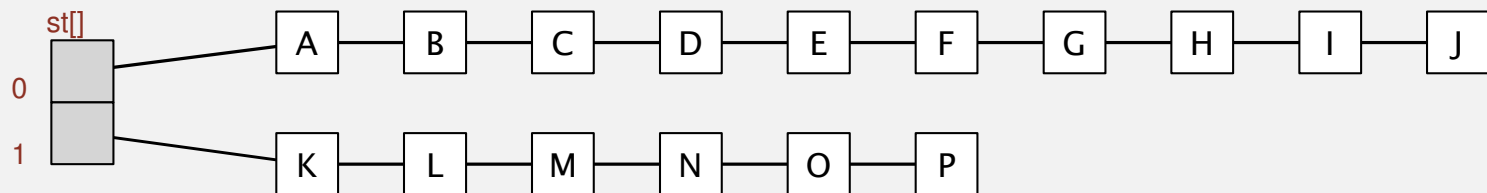
Resizing in a separate-chaining hash table

LO 8.5

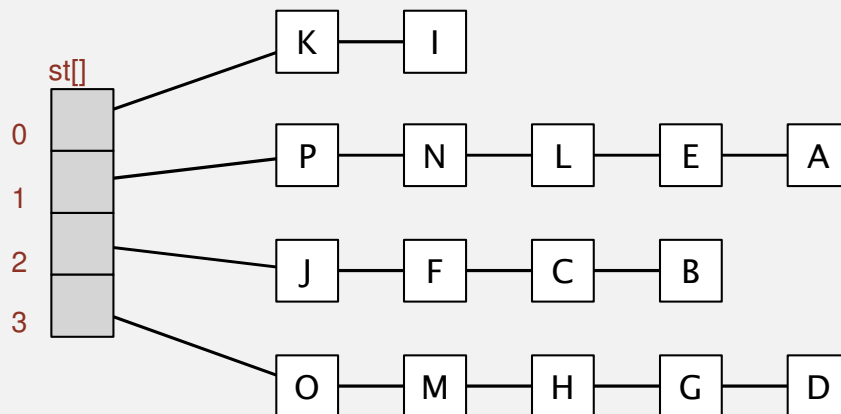
Goal. Average length of list $n / m = \text{constant}$.

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

before resizing ($n/m = 8$)



after resizing ($n/m = 4$)



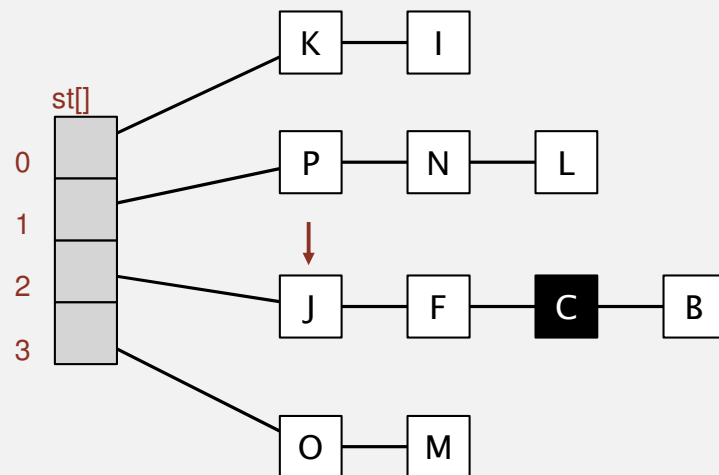
Deletion in a separate-chaining hash table

LO 8.7

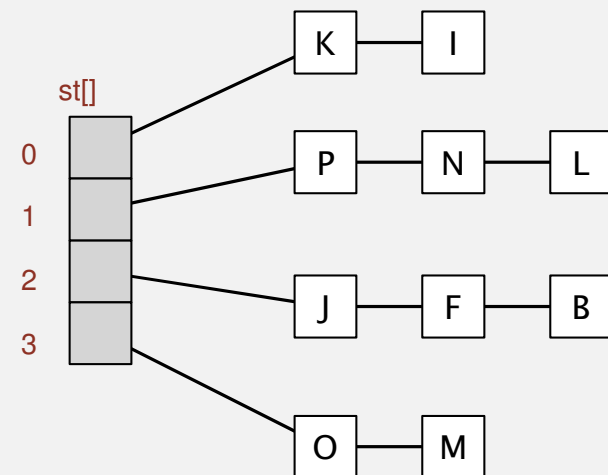
Q. How to delete a key (and its associated value)?

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

before deleting C



after deleting C



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	n	n	n	n	n	n		equals()
binary search(ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	\sqrt{n}	✓	compareTo()
red-black BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✓	compareTo()
separate chaining	n	n	n	$1 *$	$1 *$	$1 *$		equals() hashCode()

* under uniform hashing assumption

HASH TABLES

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

Collision resolution: open addressing

LO 8.8

Open addressing. [Amdahh-Boehme-Rocherster-Samuel, IBM 1953]

- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot, and put it there.

linear-probing hash table ($m = 16$, $n = 10$)

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

Linear-probing hash table summary

Hash. Map key to integer i between 0 and $m - 1$.

Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

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

Note. Array 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 = 30001;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

← array doubling and
halving code omitted

```
    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;
    }
```

```
}
```

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];

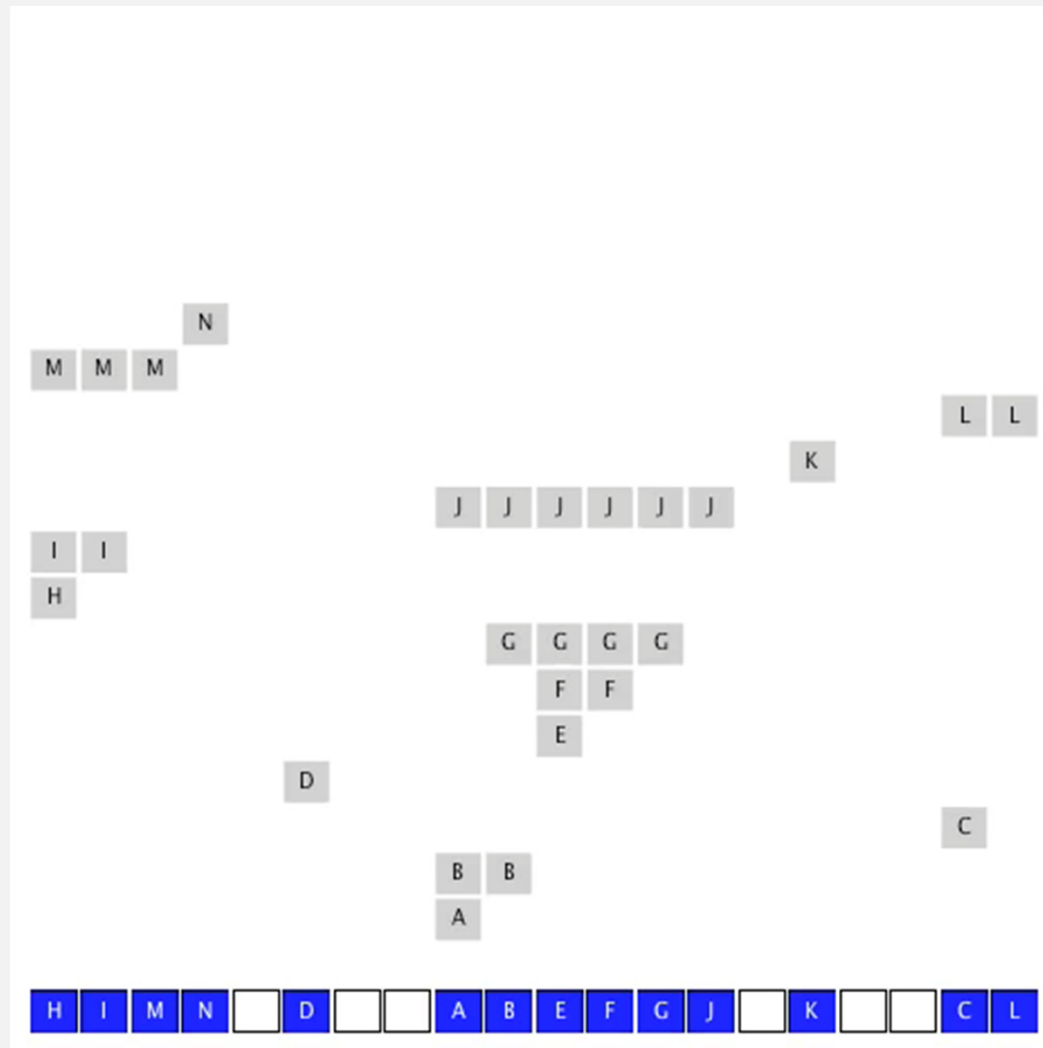
    private int hash(Key key)        { /* as before */ }

    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.

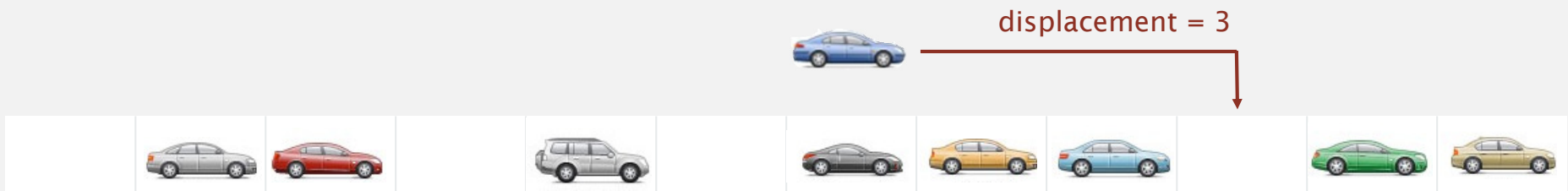
Observation. New keys likely to hash into middle of big clusters.



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 $\sim 5 / 2$.

Full. With m cars, mean displacement is $\sim \sqrt{\pi m / 8}$.

Key insight. Cannot afford to let linear-probing hash table get too full.

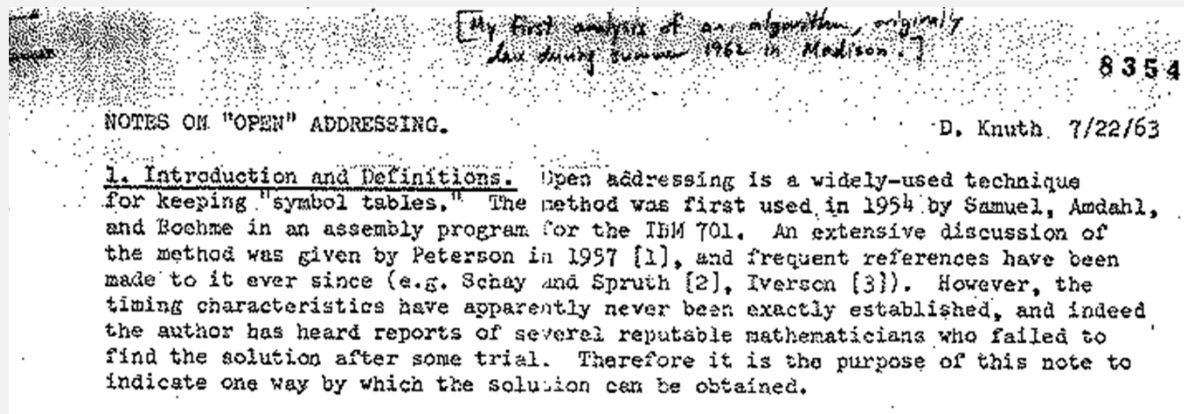
Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size m that contains $n = \alpha m$ keys is:

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

search hit

search miss / insert

Pf.



Parameters.

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

Resizing in a linear-probing hash table

Goal. Average length of list $n / m \leq 1/2$.

- Double length of array m when $n / m \geq 1/2$.
- Halve length of array m when $n / m \leq 1/8$.
- Need to rehash all keys when resizing.

before resizing

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

after resizing

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

Deletion in a linear-probing hash table

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

A. Requires some care: can't just delete array entries.

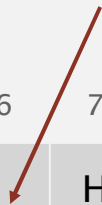
before deleting S

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

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

doesn't work, e.g., if $\text{hash}(H) = 4$



ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	n	n	n	n	n	n		equals()
binary search(ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	\sqrt{n}	✓	compareTo()
red-black BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✓	compareTo()
separate chaining	n	n	n	1 *	1 *	1 *		equals() hashCode()
linear probing	n	n	n	1 *	1 *	1 *		equals() hashCode()

* under uniform hashing assumption

3-SUM (REVISITED)

3-SUM. Given n distinct integers, find three such that $a + b + c = 0$.

Goal. n^2 expected time case, n extra space.

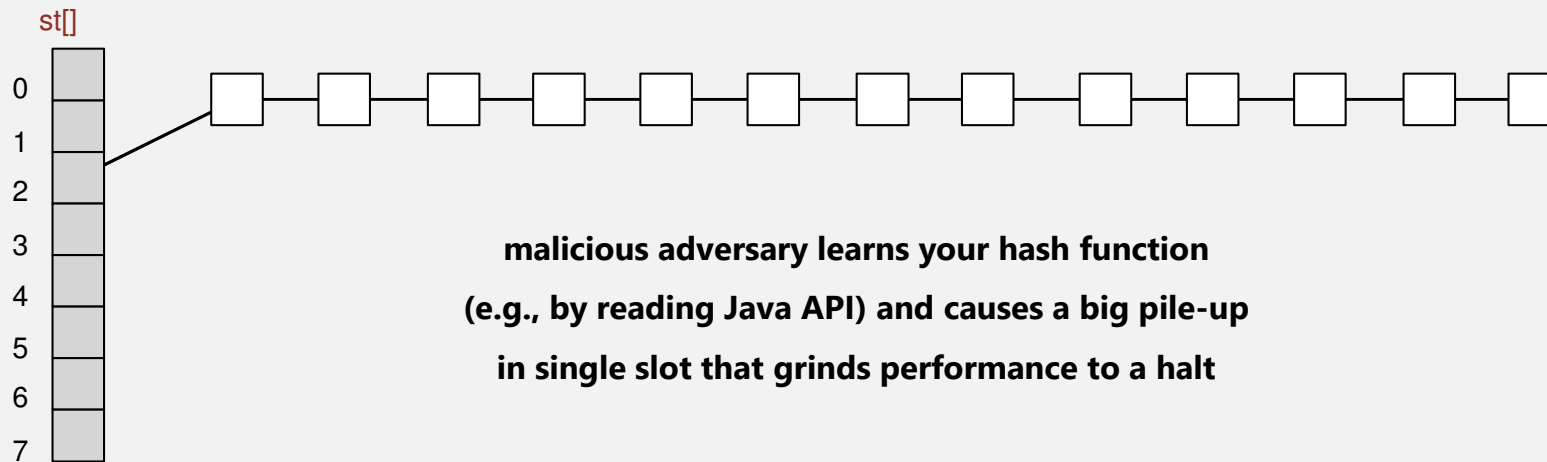
HASH TABLES

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

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]

- 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: String hashing in Java

String hashCode() in Java 1.1.

- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: saves time in performing arithmetic.

```
public int hashCode()
{
    int hash = 0;
    int skip = Math.max(1, length() / 8);
    for (int i = 0; i < length(); i += skip)
        hash = s[i] + (37 * hash);
    return hash;
}
```

- Downside: great potential for bad collision patterns.

<http://www.cs.princeton.edu/introcs/13loop/Hello.java>

<http://www.cs.princeton.edu/introcs/13loop/Hello.class>

<http://www.cs.princeton.edu/introcs/13loop/Hello.html>

<http://www.cs.princeton.edu/introcs/12type/index.html>



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
"BBaBBBBB"	-540425984
"BBBBaAaA"	-540425984
"BBBBaABB"	-540425984
"BBBBBBaA"	-540425984
"BBBBBBBB"	-540425984

2^n strings of length $2n$ that hash to same value!

Diversion: one-way hash functions

One-way hash function. “Hard” to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, SHA-256, WHIRLPOOL,


known to be insecure

```
String password = args[0];  
MessageDigest sha = MessageDigest.getInstance("SHA-256");  
byte[] bytes = sha.digest(password);  
  
/* prints bytes as hex string */
```

Applications. Crypto, message digests, passwords, Bitcoin,

Caveat. Too expensive for use in ST implementations.

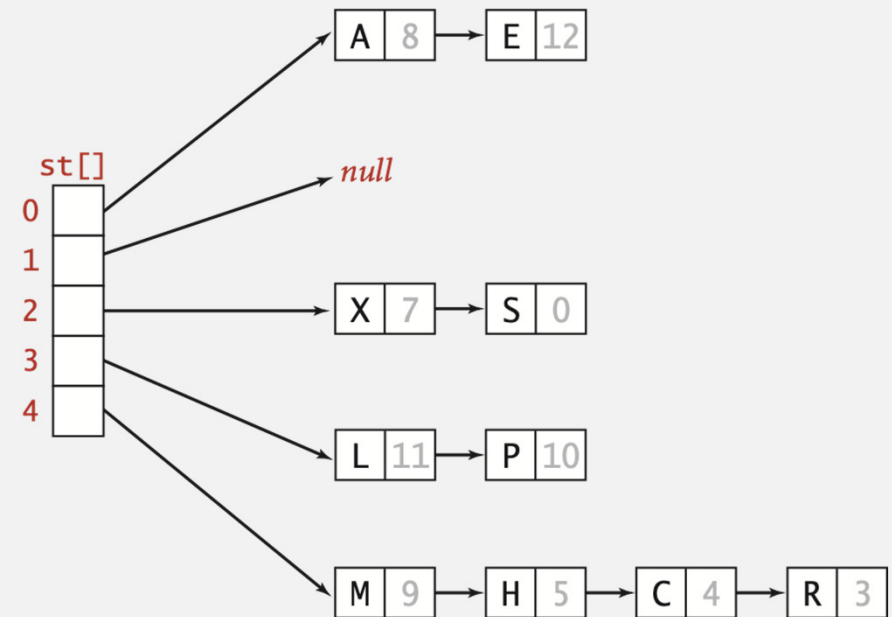
Separate chaining vs. linear probing

Separate chaining.

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

Linear probing.

- Less wasted space.
- Better cache performance.



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

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

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

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

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



Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log n$ compares).
- Better system support in Java for String (e.g., cached hash code).

Balanced search trees.

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

Java system includes both.

- Balanced search trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

← red-black BST

↑
linear probing

↑
separate chaining

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

R u t g e r s U n i v e r s i t y

HASH TABLES

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

