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%.* ”

Don 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()`

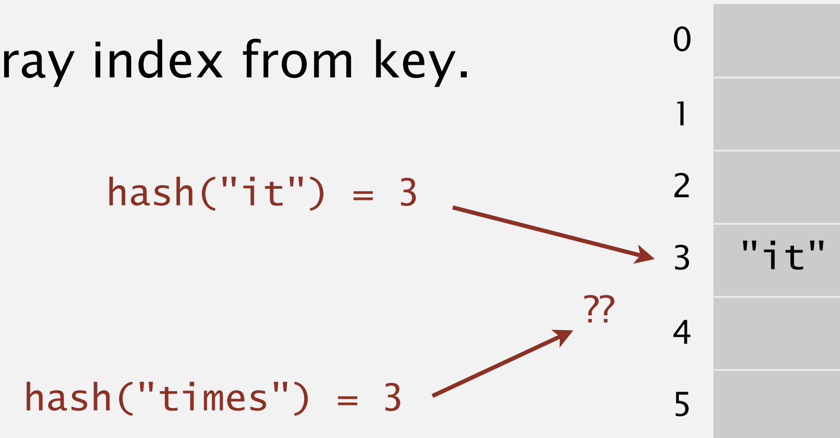
Q. Can we do better?

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

Hashing: basic plan

Save items in a key-indexed array (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

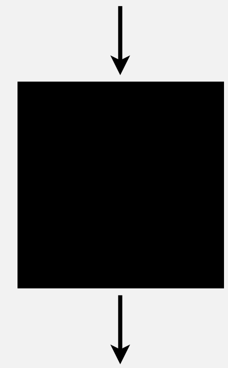
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

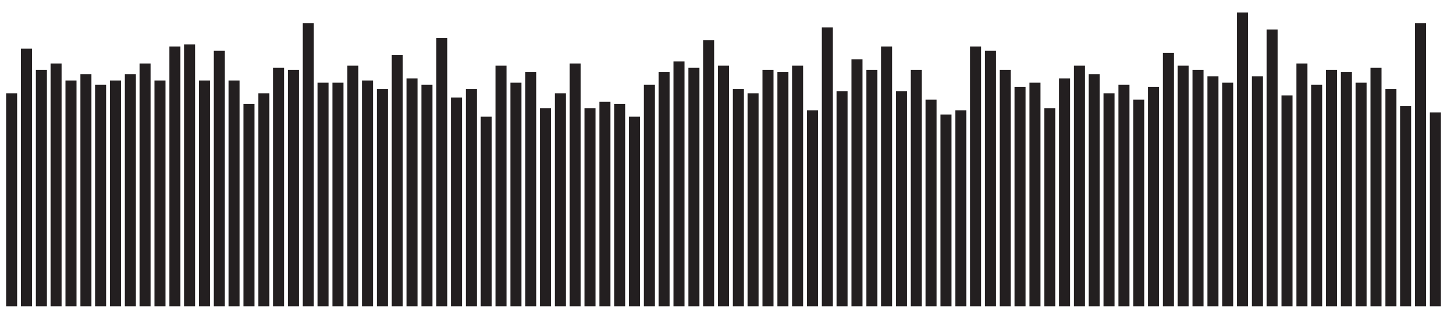
Computing the hash function

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





Hash value frequencies for words in Tale of Two Cities (M = 97)

Practical challenge. Need different approach for each key type.

Minimizing hash function collisions

Challenge. Distribution of keys unknown, may contain patterns.

Examples of string inputs with patterns:

- Words from ‘tale of two cities’
- Strings consisting of only ‘0’ and ‘1’:

  0011100, 1010100000, ...
- Only strings of length <= 3
- URLs on a web server

```
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
```

Hash tables: quiz 1

What's the best way to hash a 10-digit phone number to a value between 0 and 999?

A. First 3 digits

B. Last 3 digits

C. Either A or B

D. Neither A nor B

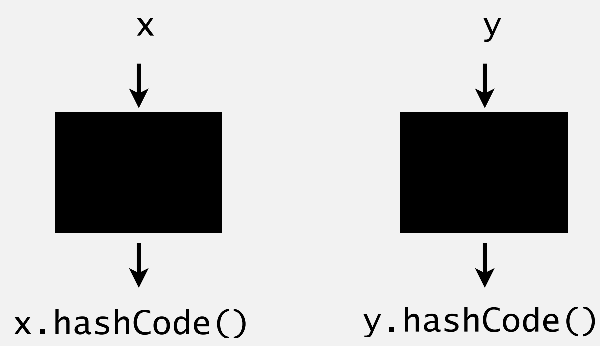
E. *I don't know.*

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

Are these magic constants?!

Implementing hash code: strings

Treat string of length L 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 multiplications/additions to hash string of length L.

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

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

Implementing hash code: strings

Recall:

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

Peek ahead: modular hashing

Convert hash code to array index by taking remainder mod array length

Q. What could go wrong if the length of array is 31 (or a multiple of 31)?

A. Only the last character of the string affects the array index

Q. Is this hash function better or worse than Java's?

$$h = s[0] \cdot 30^{L-1} + \ldots + s[L-3] \cdot 30^2 + s[L-2] \cdot 30^1 + s[L-1] \cdot 30^0$$

A. Worse, because it is much more likely that the array length will have a common factor with 30 than with 31.

Implementing hash code: strings

Performance optimization.

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

Skipped in class

```
public final class String
{
   private int hash = 0;
   private final char[] s;
   ...

   public int hashCode()
   {
      int h = hash;
      if (h != 0) return h;
      for (int i = 0; i < length(); i++)
         h = s[i] + (31 * h);
      hash = h;
      return h;
   }
}
```

cache of hash code

return cached value

store cache of hash code

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

`hashCode()` of `"pollinating sandboxes"` is 0

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  howmuch;

   public Transaction(String who, Date when, double howmuch)
   {  /* 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) howmuch).hashCode();
      return hash;
   }
}
```

nonzero constant

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

typically a small prime

Hash code design

"Standard" recipe for user-defined types.

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

In practice. Recipe above works reasonably well; used in Java libraries.
In theory. Keys are bitstrings; "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 2

Which of the following is an effective way to map a hashable key to an integer between `0` and `M-1` ?

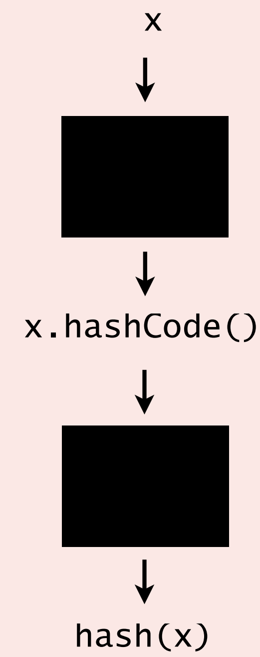
A.
```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

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

C. Both A and B.

D. Neither A nor B.

E. *I don't know.*



Trick question

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

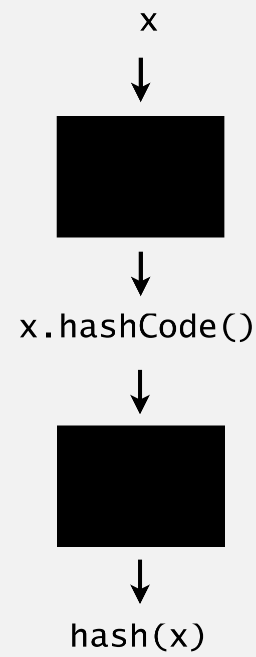
```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

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

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

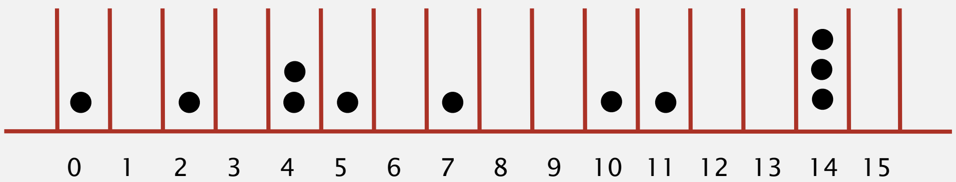
correct



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.

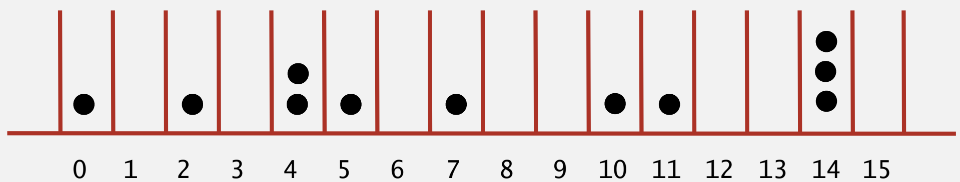


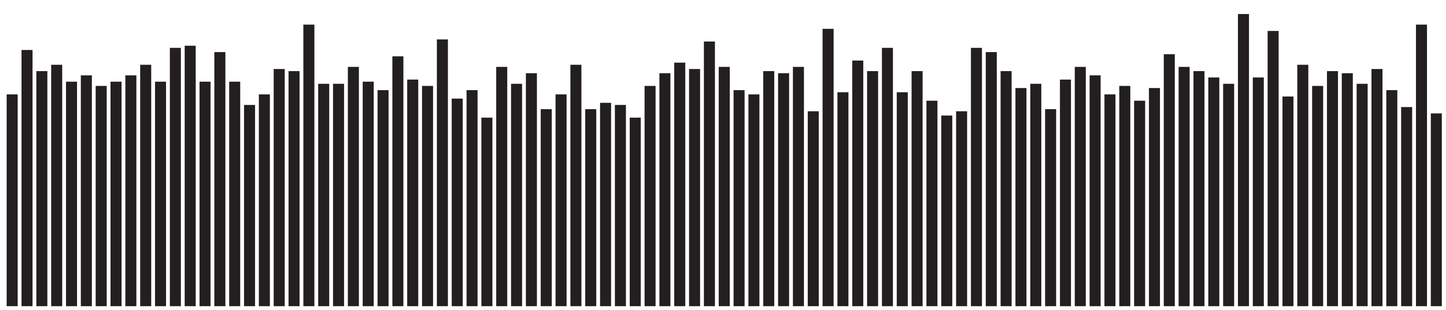
Birthday problem. Expect two balls in the same bin after $\sim\sqrt{\pi M / 2}$ tosses.

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.4 HASH TABLES

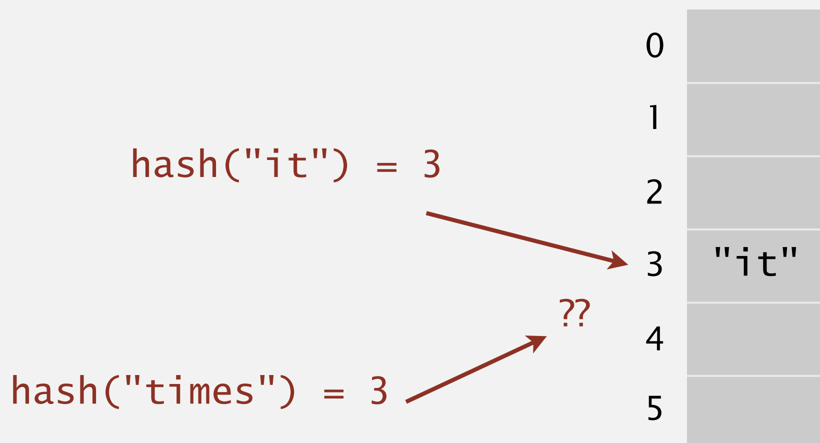
- hash functions
- **separate chaining**
- linear probing
- context

Collisions

Collision. Two distinct keys hashing to same index.

Birthday problem ⇒ can't avoid collisions.

unless you have a ridiculous (quadratic) amount of memory

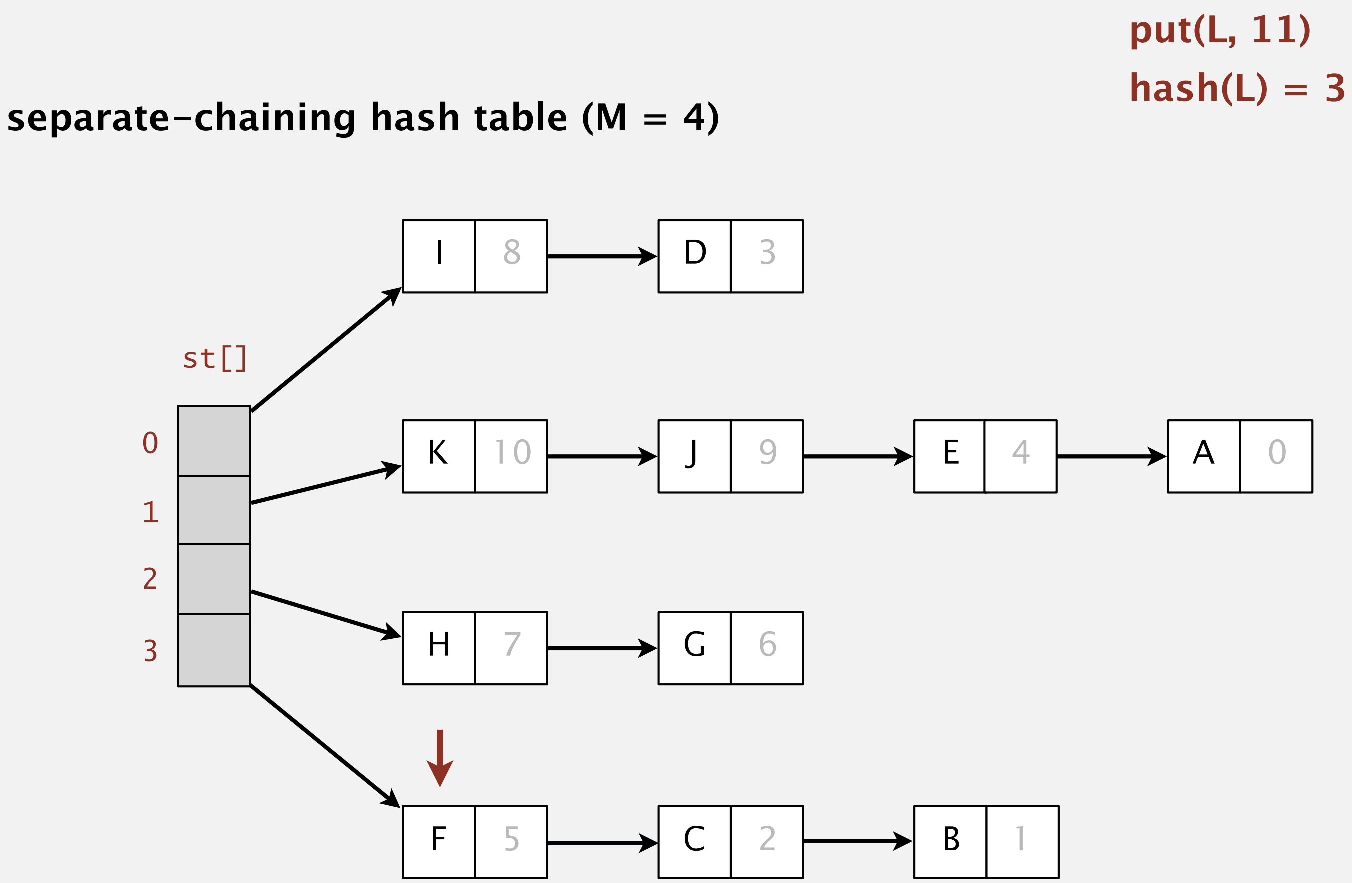


Challenge. Deal with collisions efficiently.

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: sequential search in i^{th} chain.



Separate-chaining symbol table

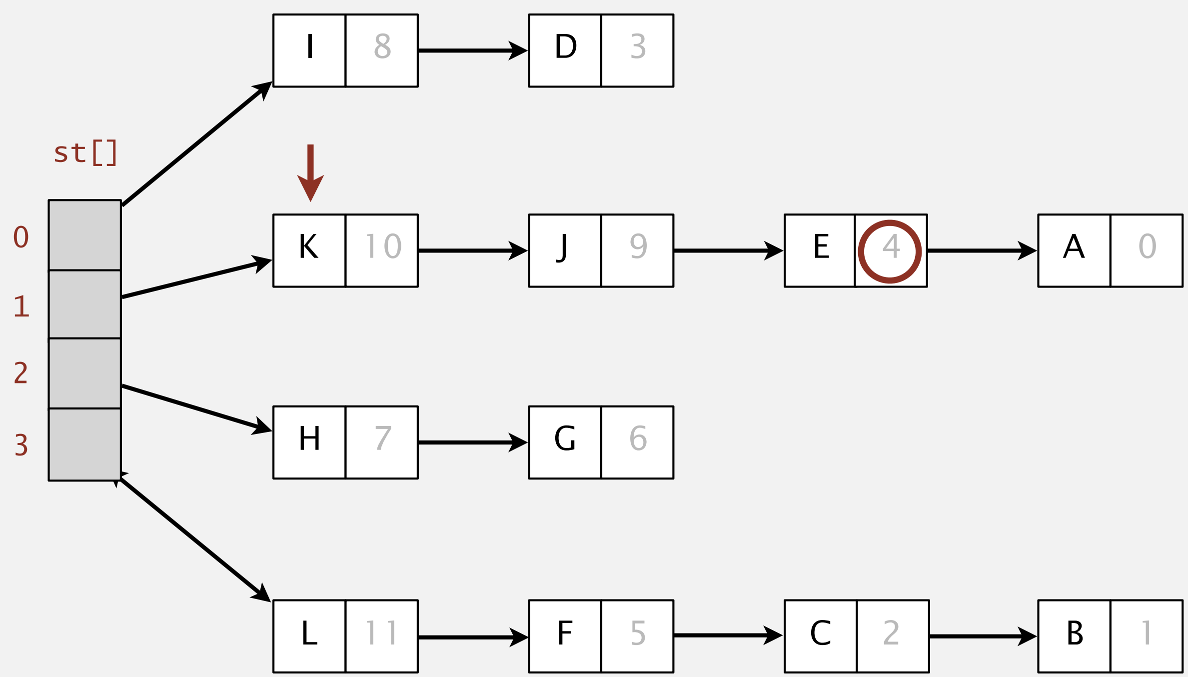
Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- 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: sequential search in i^{th} chain.

get(E)

hash(E) = 1

separate-chaining hash table (M = 4)



Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                 // number of chains
   private Node[] st = new Node[M];    // array of chains

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

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

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

}
```

array doubling and halving code omitted

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

Skipped in class

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]);
   }

}
```

Skipped in class

Analysis of separate chaining

Proposition. Under uniform hashing assumption, the number of keys in each list is close to N / M.

`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: $M \sim \frac{1}{4} N \Rightarrow$ constant-time ops.

M times faster than sequential search

Q. When to resize?

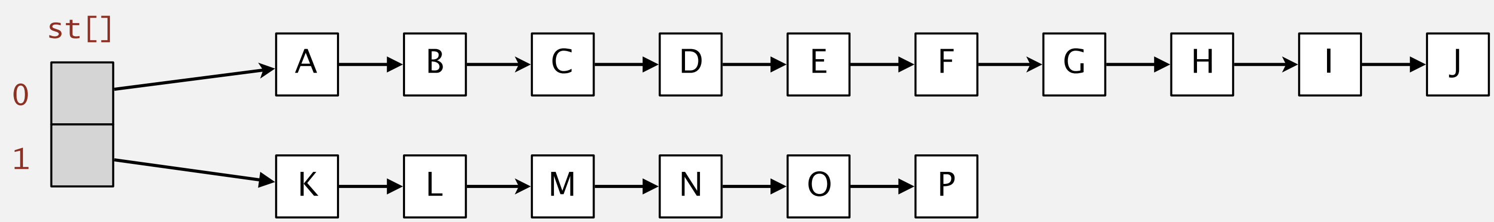
Resizing in a separate-chaining hash table

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

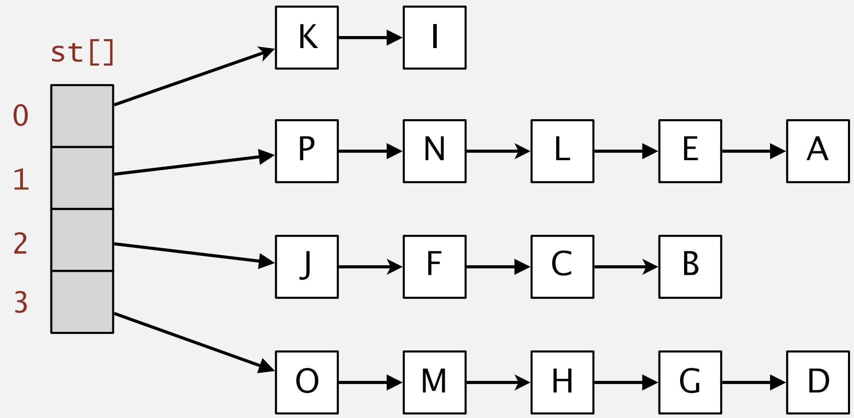
- Double size of array M when $N / M \geq 8$; halve size of array M when $N / M \leq 2$.
- Note: need to rehash all keys when resizing.

`x.hashCode()` does not change; but `hash(x)` can change

before resizing (N/M = 8)



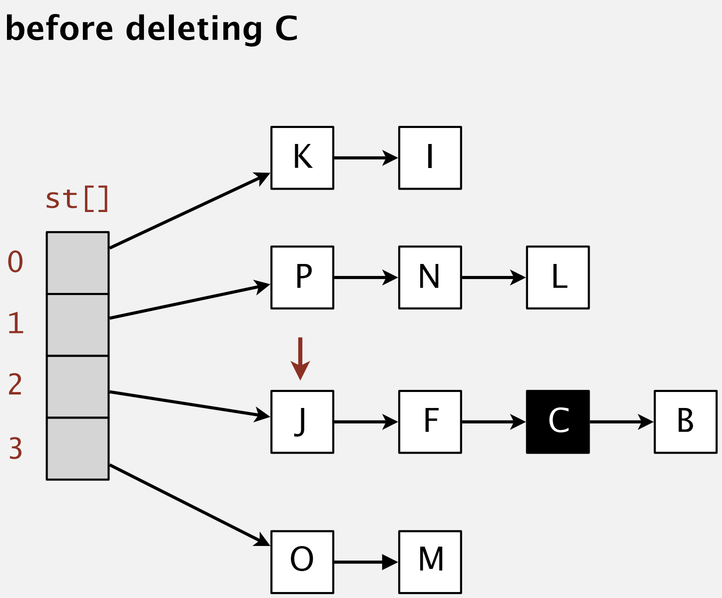
after resizing (N/M = 4)

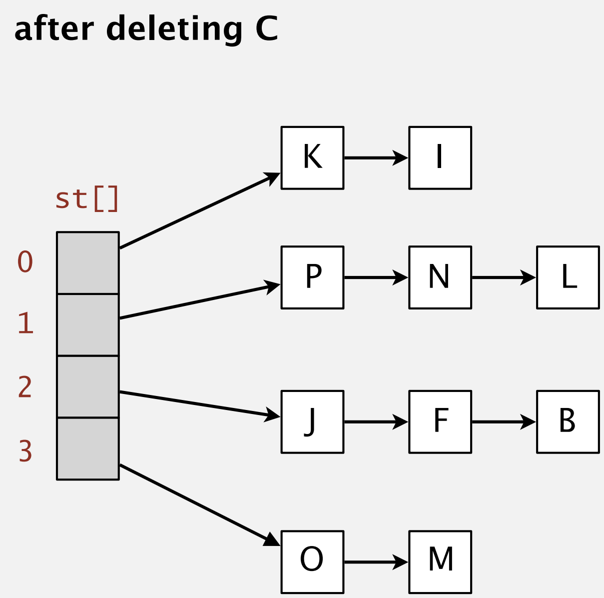


Deletion in a separate-chaining hash table

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

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





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: linear probing

Linear probing. [Amdahl-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

put(K, 14)

hash(K) = 7

Linear-probing hash table summary

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

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

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

Note. Array size M must be greater than number of key-value pairs N.

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

`M = 16`

Linear-probing hash table demo: insert

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
keys[]																

Linear-probing hash table demo: insert

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.

insert S
hash(S) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]																

Linear-probing hash table demo: insert

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.

insert S
hash(S) = 6

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

Linear-probing hash table demo: insert

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.

insert S
hash(S) = 6

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

Linear-probing hash table demo: insert

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
keys[]							S									

Linear-probing hash table demo: insert

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.

insert E
hash(E) = 10

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

Linear-probing hash table demo: insert

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.

insert E
hash(E) = 10

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

Linear-probing hash table demo: insert

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.

insert E
hash(E) = 10

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

Linear-probing hash table demo: insert

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
keys[]							S				E					

Linear-probing hash table demo: insert

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.

insert A
hash(A) = 4

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

Linear-probing hash table demo: insert

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.

insert A
hash(A) = 4

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

Linear-probing hash table demo: insert

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.

insert A
hash(A) = 4

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

Linear-probing hash table demo: insert

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
keys[]					A		S				E					

Linear-probing hash table demo: insert

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.

insert R
hash(R) = 14

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

Linear-probing hash table demo: insert

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.

insert R

hash(R) = 14

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

Linear-probing hash table demo: insert

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.

insert R
hash(R) = 14

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

Linear-probing hash table demo: insert

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
keys[]					A		S				E				R	

Linear-probing hash table demo: insert

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.

insert C

hash(C) = 5

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

Linear-probing hash table demo: insert

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.

insert C
hash(C) = 5

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

Linear-probing hash table demo: insert

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.

insert C
hash(C) = 5

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

Linear-probing hash table demo: insert

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
keys[]					A	C	S				E				R	

Linear-probing hash table demo: insert

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.

insert H
hash(H) = 4

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

Linear-probing hash table demo: insert

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.

insert H
hash(H) = 4

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

Linear-probing hash table demo: insert

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.

insert H
hash(H) = 4

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

Linear-probing hash table demo: insert

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.

insert H

hash(H) = 4

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

Linear-probing hash table demo: insert

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.

insert H
hash(H) = 4

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

Linear-probing hash table demo: insert

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.

insert H
hash(H) = 4

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

Linear-probing hash table demo: insert

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
keys[]					A	C	S	H			E				R	

Linear-probing hash table demo: insert

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.

insert X
hash(X) = 15

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

Linear-probing hash table demo: insert

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.

insert X
hash(X) = 15

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

Linear-probing hash table demo: insert

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.

insert X
hash(X) = 15

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

Linear-probing hash table demo: insert

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
keys[]					A	C	S	H			E				R	X

Linear-probing hash table demo: insert

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.

insert M
hash(M) = 1

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

Linear-probing hash table demo: insert

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.

insert M
hash(M) = 1

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

Linear-probing hash table demo: insert

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.

insert M
hash(M) = 1

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

Linear-probing hash table demo: insert

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
keys[]		M			A	C	S	H			E				R	X

Linear-probing hash table demo: insert

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.

insert P
hash(P) = 14

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

Linear-probing hash table demo: insert

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.

insert P
hash(P) = 14

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

Linear-probing hash table demo: insert

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.

insert P
hash(P) = 14

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

Linear-probing hash table demo: insert

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.

insert P
hash(P) = 14

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

Linear-probing hash table demo: insert

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
keys[]	P	M			A	C	S	H			E				R	X

Linear-probing hash table demo: insert

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.

insert L
hash(L) = 6

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

Linear-probing hash table demo: insert

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.

insert L

hash(L) = 6

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

Linear-probing hash table demo: insert

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.

insert L

hash(L) = 6

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

Linear-probing hash table demo: insert

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.

insert L

hash(L) = 6

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

Linear-probing hash table demo: insert

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.

insert L

hash(L) = 6

	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

Linear-probing hash table demo: insert

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
keys[]	P	M			A	C	S	H	L		E				R	X

Linear-probing hash table demo: search

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.

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	S	H	L		E				R	X

Linear-probing hash table demo: search

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 E

hash(E) = 10

	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

Linear-probing hash table demo: search

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 E

hash(E) = 10

	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

E

search hit

(return corresponding value)

Linear-probing hash table demo: search

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.

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	S	H	L		E				R	X

Linear-probing hash table demo: search

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 L

hash(L) = 6

	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

Linear-probing hash table demo: search

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 L

hash(L) = 6

	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
							L									

Linear-probing hash table demo: search

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 L

hash(L) = 6

	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
								L								

Linear-probing hash table demo: search

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 L

hash(L) = 6

	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

L

search hit
(return corresponding value)

Linear-probing hash table demo: search

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.

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	S	H	L		E				R	X

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X
						K										

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X
							K									

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X
								K								

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X
									K							

Linear-probing hash table demo: search

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
keys[]	P	M			A	C	S	H	L		E				R	X
										K						

search miss
(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   */  }

   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

Skipped in class

sequential search in chain i

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

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

   private Value get(Key key)            { /* prev slide */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

}
```

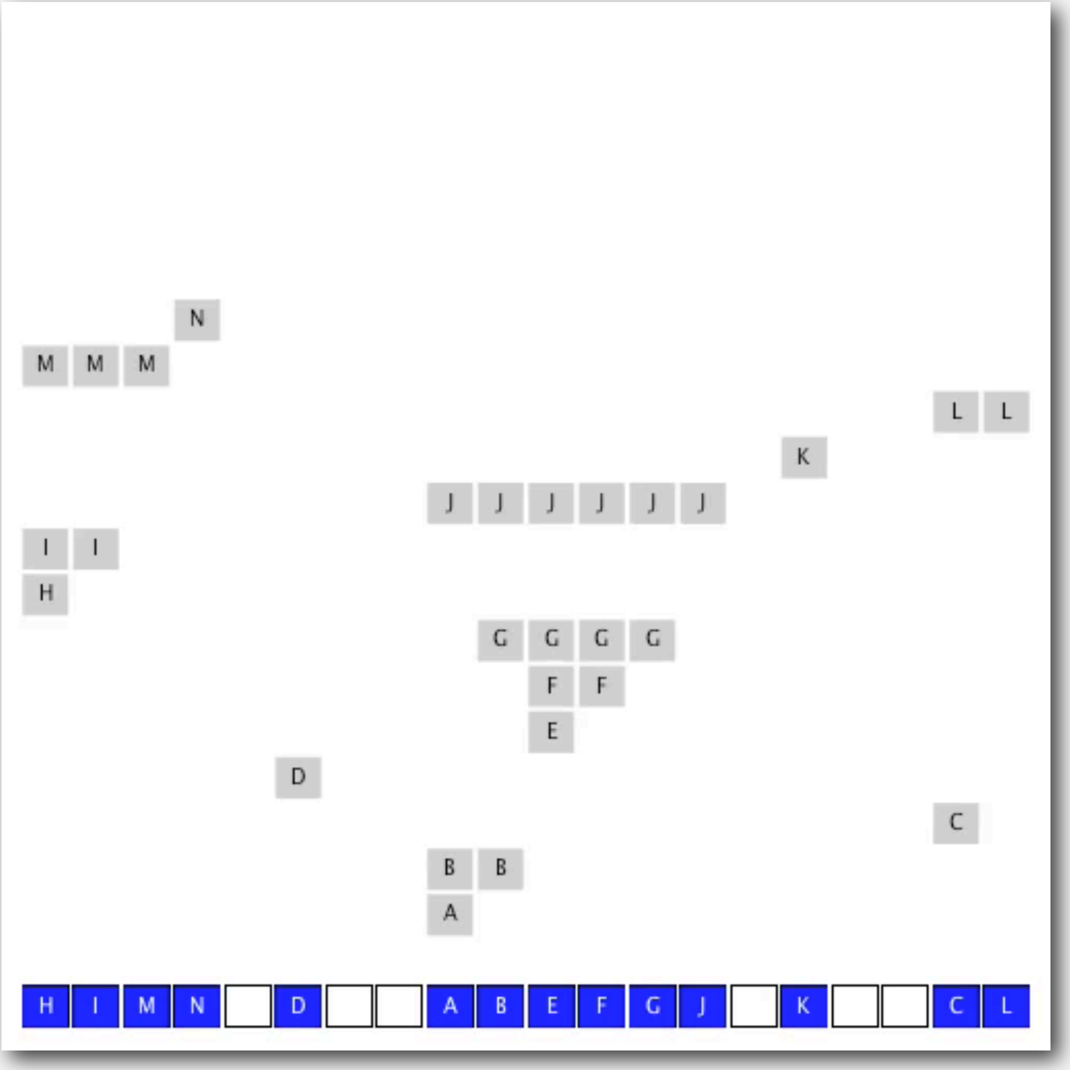
sequential search
in chain i

Skipped
in class

Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of 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:

fraction of array that's filled

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

search hit **search miss / insert**

Parameters.

- α too small $\Rightarrow$ too many empty array entries.
- α too large $\Rightarrow$ 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

Q. When to resize?

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

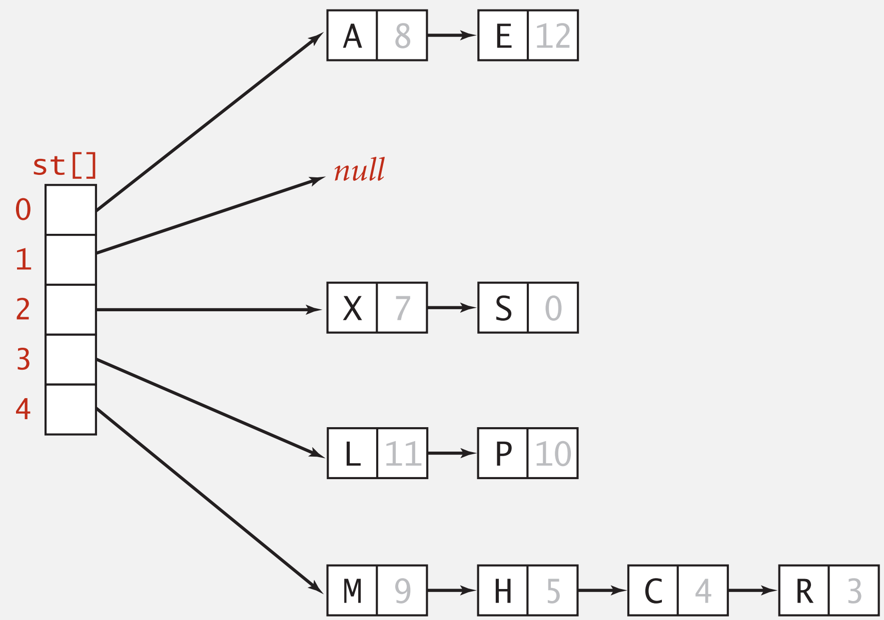
Separate chaining vs. linear probing

Separate chaining.

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

Linear probing.

- Less wasted space.
- Better cache performance.



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

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

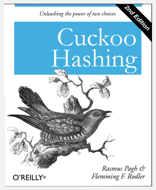
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\sim \lg \ln N$.

Double hashing. [linear-probing variant]

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

Cuckoo hashing. [linear-probing variant]

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



Hash tables vs. balanced search trees

Hash tables.

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

Balanced search trees.

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

Java system includes both.

- Red–Black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

linear probing separate chaining

http://algs4.cs.princeton.edu

3.4 Hash Tables

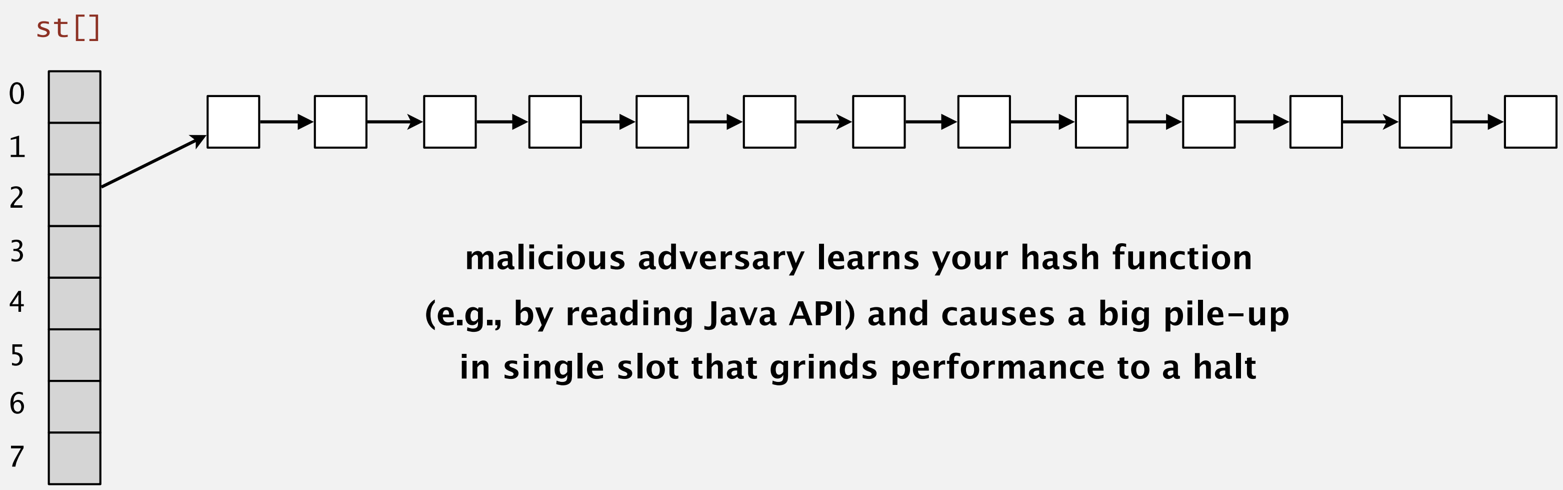
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

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.

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. $\underbrace{\text{MD4, MD5, SHA-0, SHA-1,}}_{\text{known to be insecure}}$ SHA-2, WHIRLPOOL, RIPEMD-160, ...

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Crypto, message digests, passwords, Bitcoin, ...

Caveat. Too expensive for use in ST implementations.