Hash Tables

- hash functions
- separate chaining
- linear probing
- applications

Optimize judiciously

" More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity. " — *William A. Wulf*

" We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. " — *Donald E. Knuth*

" We follow two rules in the matter of optimization:
Rule 1: Don't do it.
Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution. " — *M. A. Jackson*

Reference: Effective Java by Joshua Bloch

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`

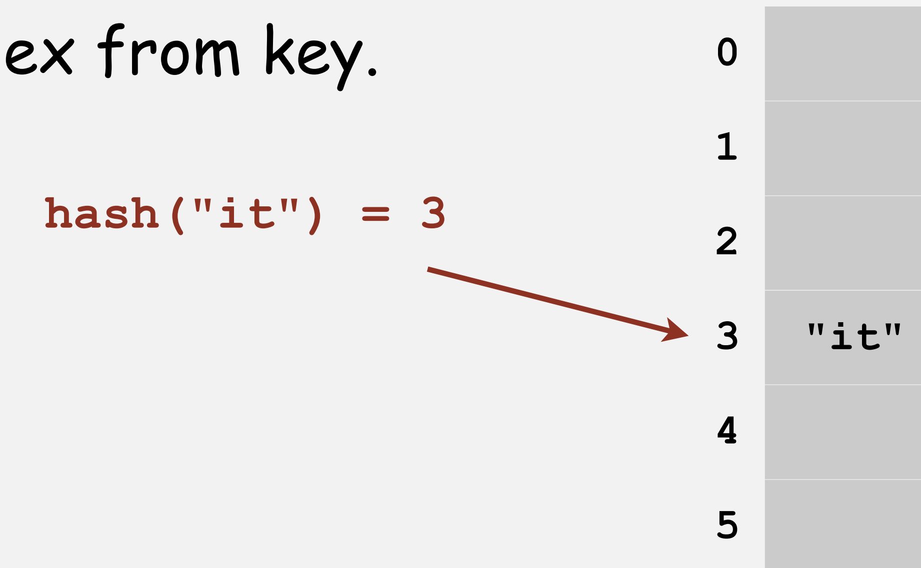
Q. Can we do better?

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

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



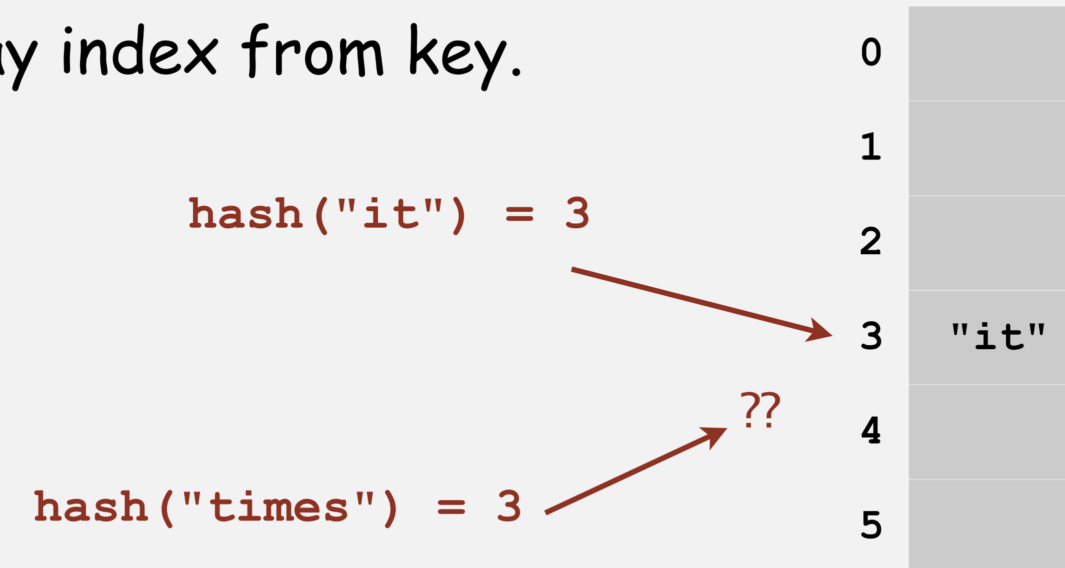
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

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

Classic space-time tradeoff.

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

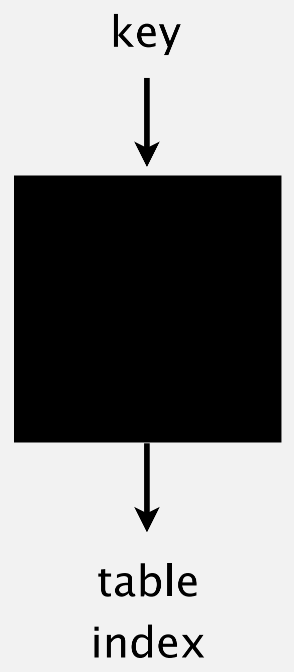
- **hash functions**
- separate chaining
- linear probing
- applications

Computing the hash function

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

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

- Bad: first three digits.
- Better: last three digits.

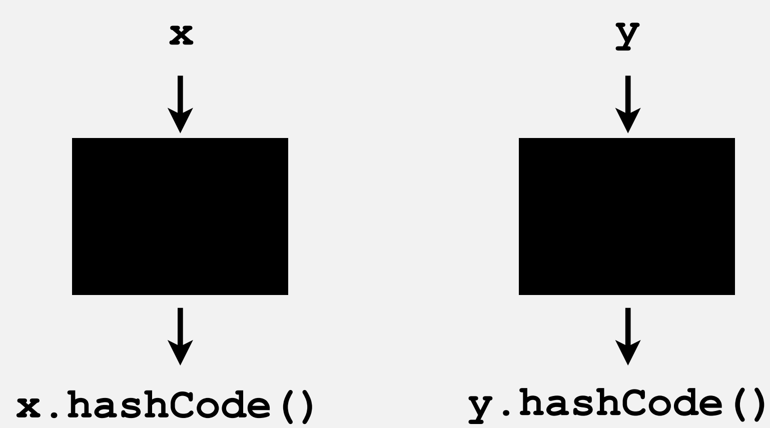
Practical challenge. Need different approach for each key type.

Java's hash code conventions

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

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Trivial (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

```
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

Implementing hash code: strings

```
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;
   }
}
```

i^{th} character of s

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

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = 31^{L-1} \cdot s^0 + \ldots + 31^2 \cdot s^{L-3} + 31^1 \cdot s^{L-2} + 31^0 \cdot s^{L-1}$.

Ex.

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

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

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

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
↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑
```

Implementing hash code: user-defined types

```
public final class Transaction
{
   private final long who;
   private final Date when;
   private final String where;

   public Transaction(long who, Date when, String where)
   {  /* as before */   }

   ...

   public boolean equals(Object y)
   {  /* as before */   }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + ((Long) who).hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + where.hashCode();
      return hash;
   }
}
```

nonzero constant

for primitive types, use **`hashCode()`** of wrapper type

for reference types, use **`hashCode()`**

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 an array, apply to each element. ← or use `Arrays.deepHashCode()`
- If field is a reference type, use `hashCode()`. ← applies rule recursively

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

In theory. Need a theorem for each type to ensure reliability.

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

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31}-1$.

Hash function. An `int` between `0` and `M-1` (for use as array index).

typically a prime or power of 2

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

bug

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

1-in-a-billion bug

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

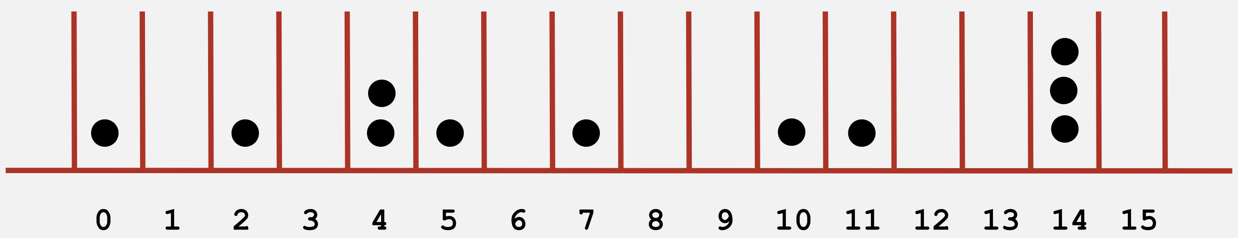
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;   }
```

correct

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



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

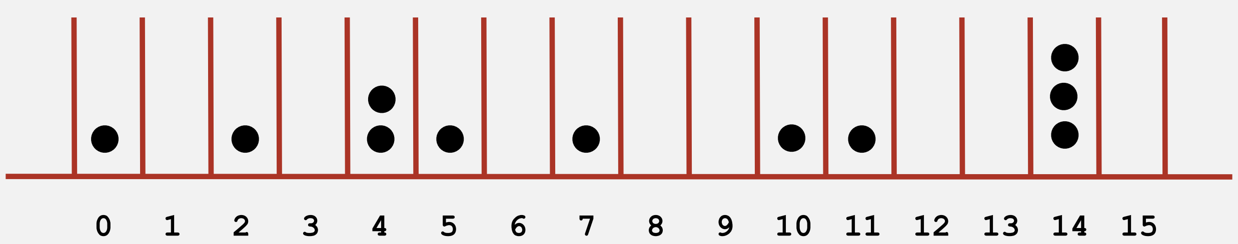
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

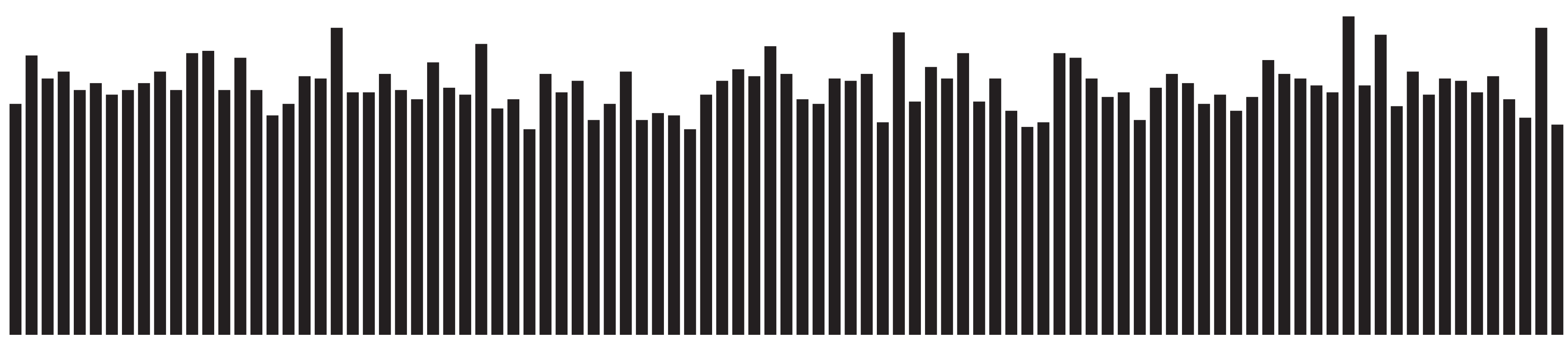
Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.





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

Java's `String` data uniformly distribute the keys of Tale of Two Cities

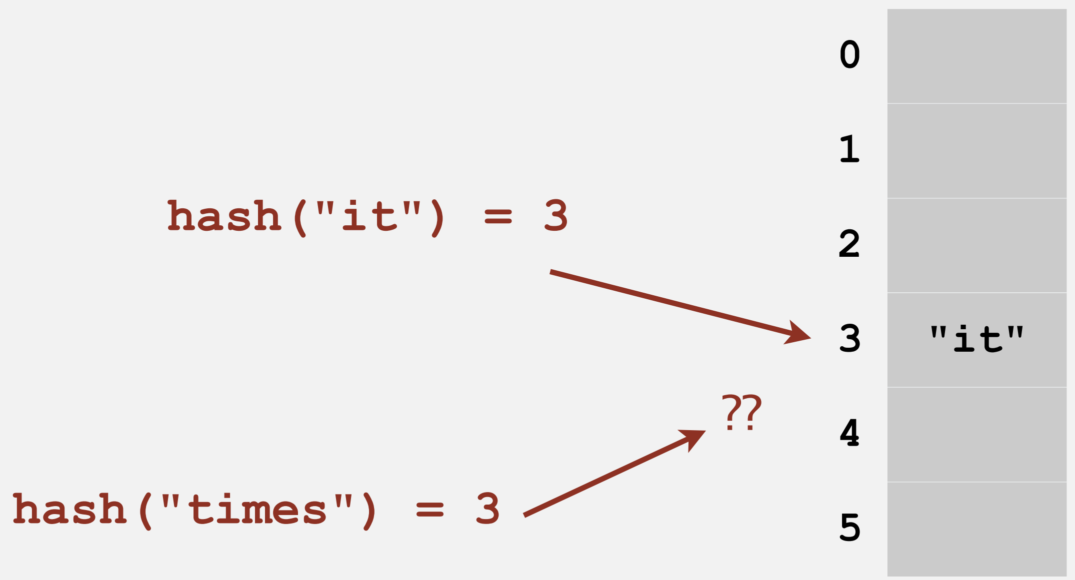
- hash functions
- **separate chaining**
- linear probing
- applications

Collisions

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 + load balancing $\Rightarrow$ collisions will be evenly distributed.

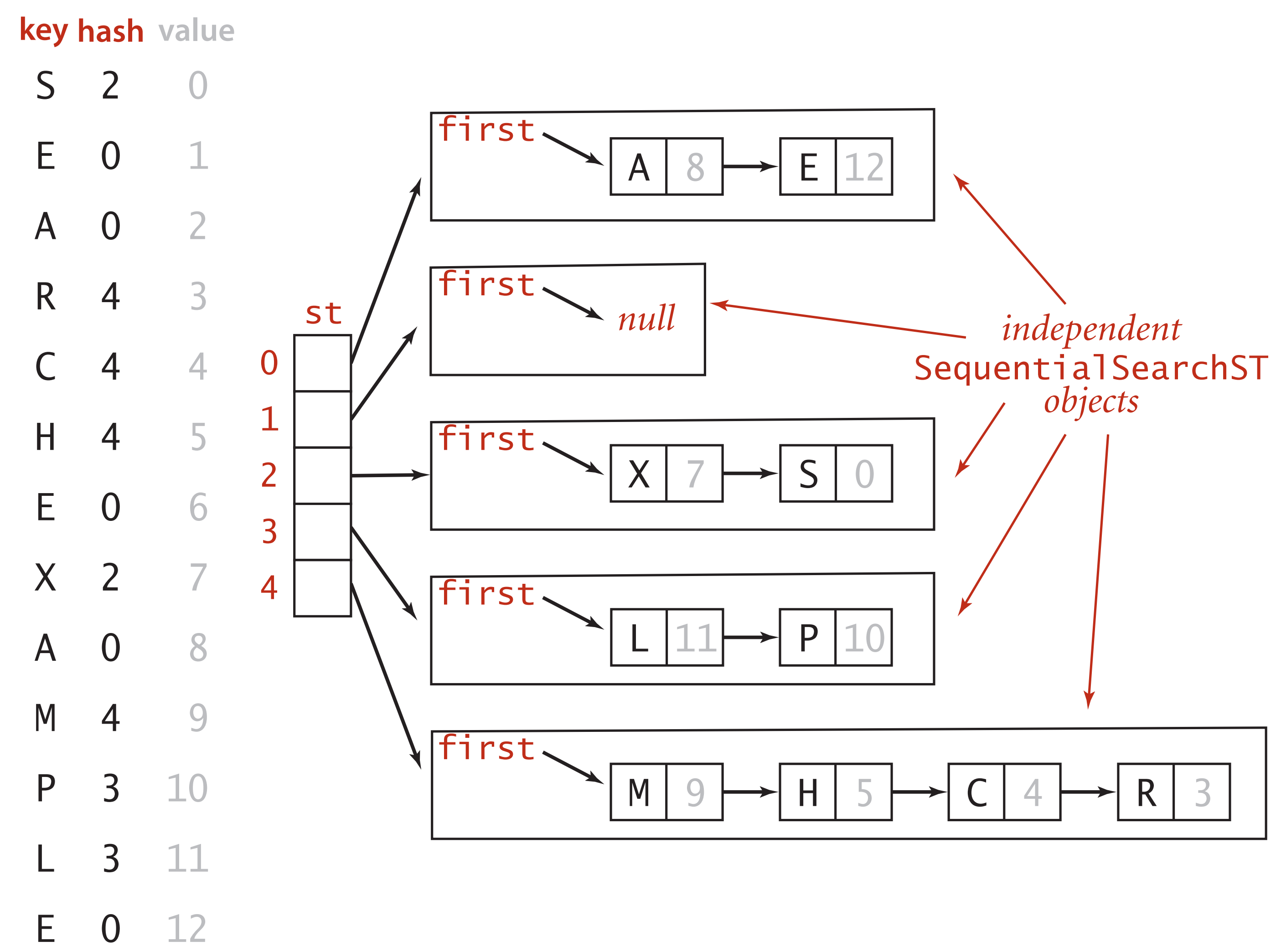
Challenge. Deal with collisions efficiently.



Separate chaining ST

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 there).
- Search: only need to search i^{th} chain.



Hashing with separate chaining for standard indexing client

Separate chaining ST: Java implementation

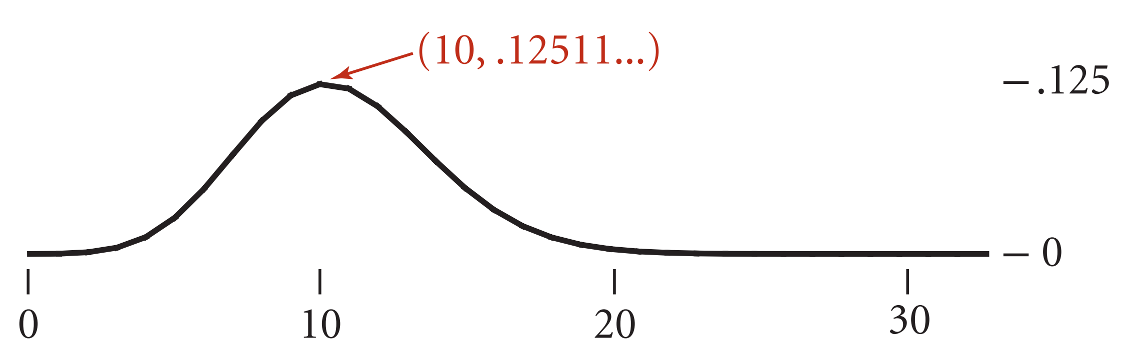
```
public class SeparateChainingHashST<Key, Value>
{
   private int N;      // number of key-value pairs
   private int M;      // hash table size
   private SequentialSearchST<Key, Value> [] st;   // array of STs

   public SeparateChainingHashST()
   {  this(997);  }

   public SeparateChainingHashST(int M)
   {
      this.M = M;
      st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
      for (int i = 0; i < M; i++)
         st[i] = new SequentialSearchST<Key, Value>();
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key)
   {  return st[hash(key)].get(key);  }

   public void put(Key key, Value val)
   {  st[hash(key)].put(key, val);  }
}
```

array doubling and halving code omitted

Analysis of separate chaining

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

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



Binomial distribution $(N = 10^4, M = 10^3, \alpha = 10)$

`equals()` and `hashCode()`

Consequence. Number of probes for search/insert is proportional to N / M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 5$ $\Rightarrow$ constant-time ops.

M times faster than sequential search

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

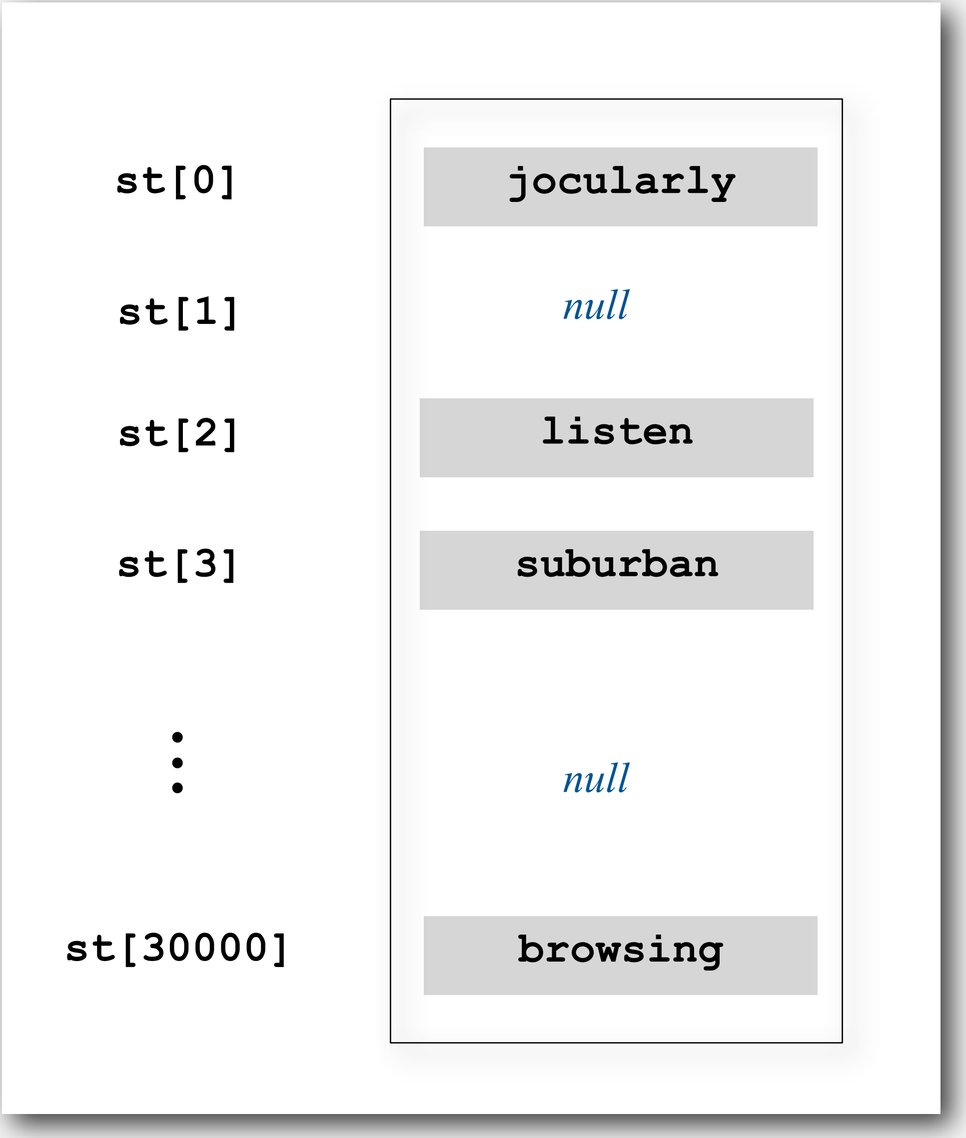
* under uniform hashing assumption

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

Collision resolution: open addressing

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

When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear probing

Use an array of size $M > N$.

- 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.

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	-	-

insert I
hash(I) = 11

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	I	-

insert N
hash(N) = 8

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	I	N

Linear probing: trace of standard indexing client

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

entries in red are new

entries in gray are untouched

keys in black are probes

probe sequence wraps to 0

keys[]

vals[]

Linear probing ST implementation

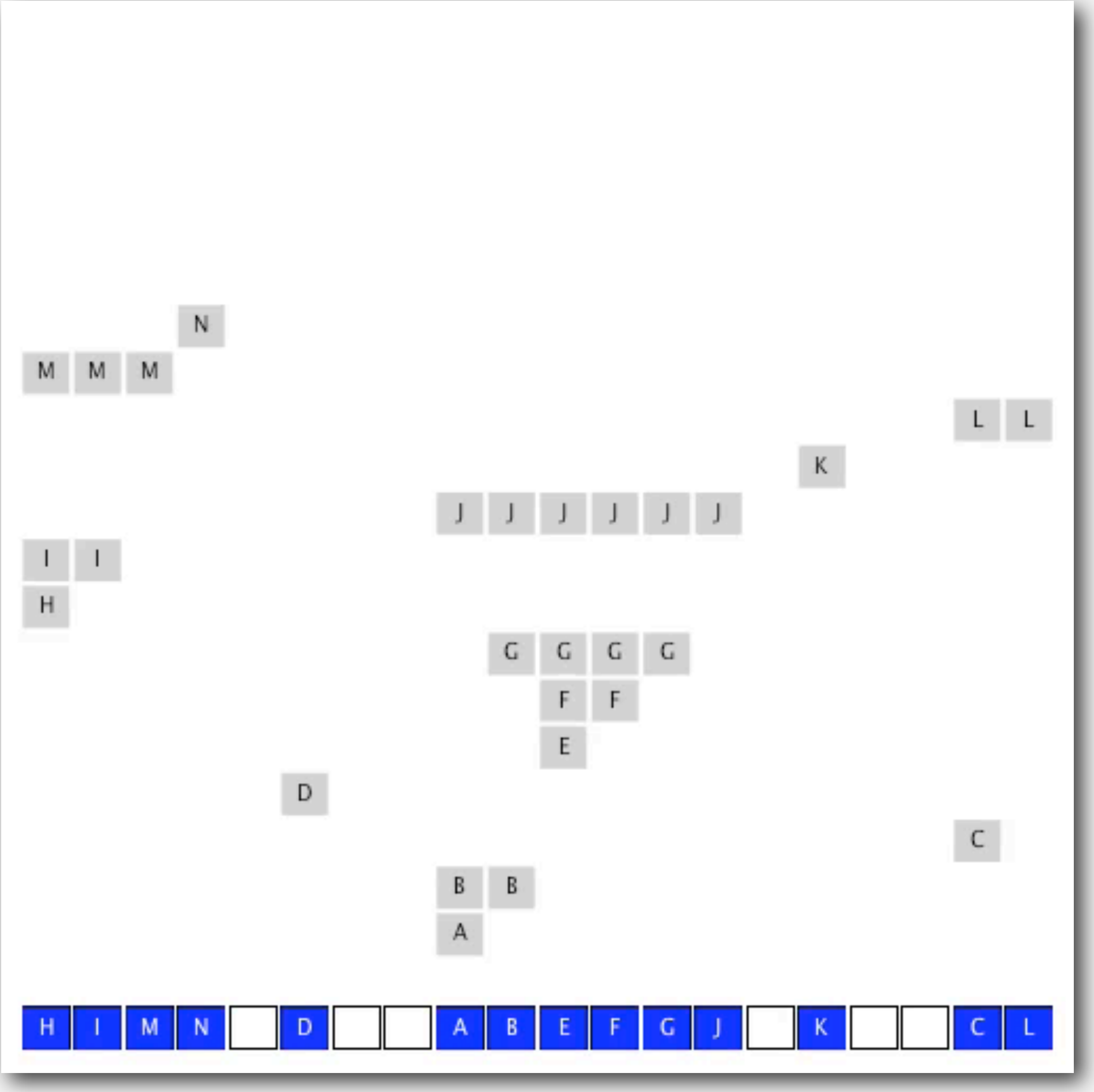
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   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;
   }

   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

Clustering

Cluster. A contiguous block of items.

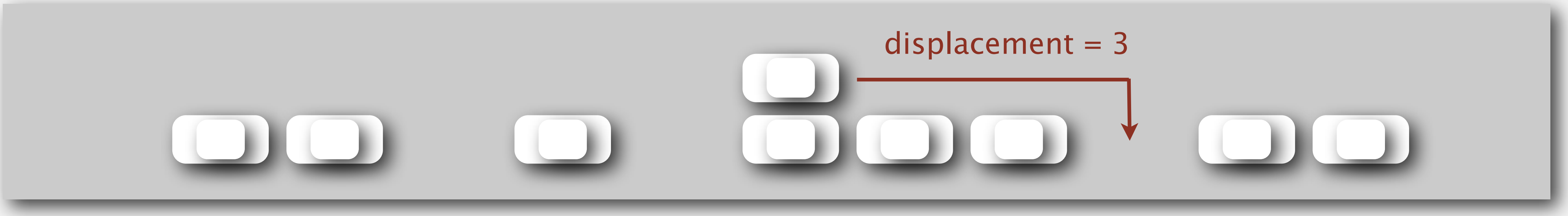
Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.
Each desires a random space i : if space i is taken, try $i + 1$, $i + 2$, etc.

Q. What is mean displacement of a car?



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

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains $N = \alpha M$ keys is:

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

search hit   search miss / insert

Pf. [Knuth 1962] A landmark in analysis of algorithms.

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim \frac{1}{2}$.

probes for search hit is about 3/2
probes for search miss is about 5/2

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

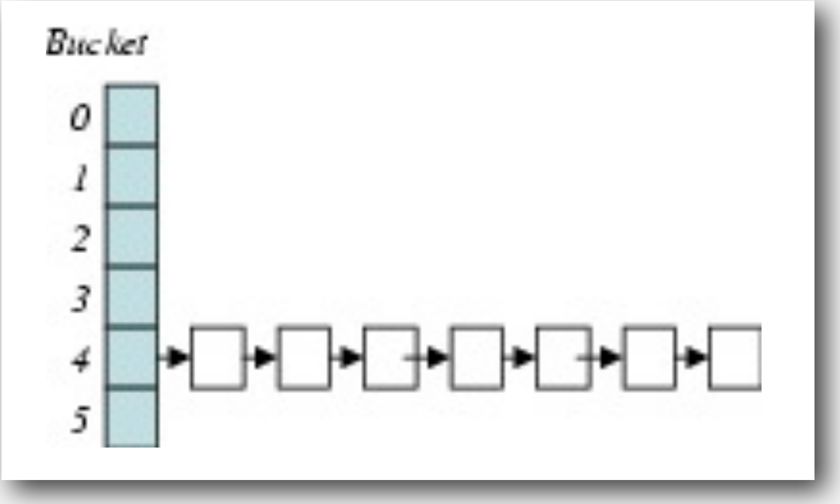
* under uniform hashing assumption

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: denial-of-service attacks.



malicious adversary learns your hash function (e.g., by reading Java API) and causes a big pile-up in single slot that grinds performance to a halt

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 hash code.

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. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

Separate chaining vs. linear probing

Separate chaining.

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

Linear probing.

- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate-chaining variant)

- Hash to two positions, put key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

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.
- Difficult to implement delete.

Hashing vs. balanced search trees

Hashing.

- 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 strings (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 trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

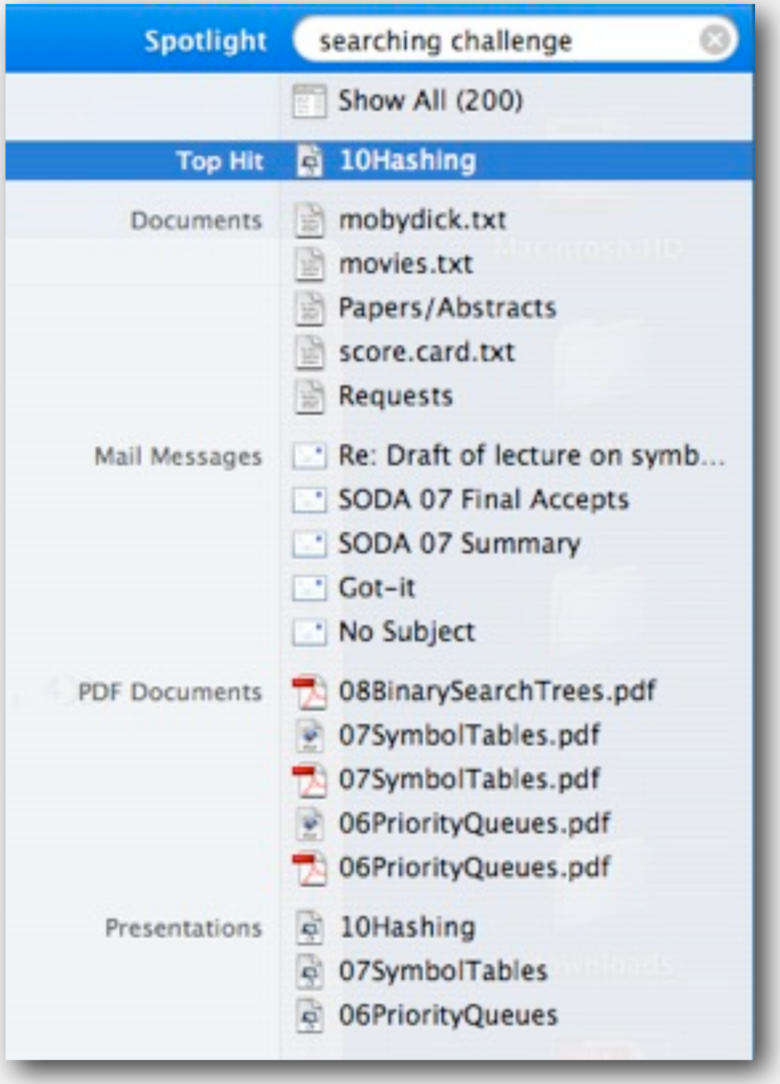
Searching challenge 5

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.

Which searching method to use?

- Hashing
- Red-black-trees
- Doesn't matter much.



Searching challenge 5

Problem. Index for a PC or the web.

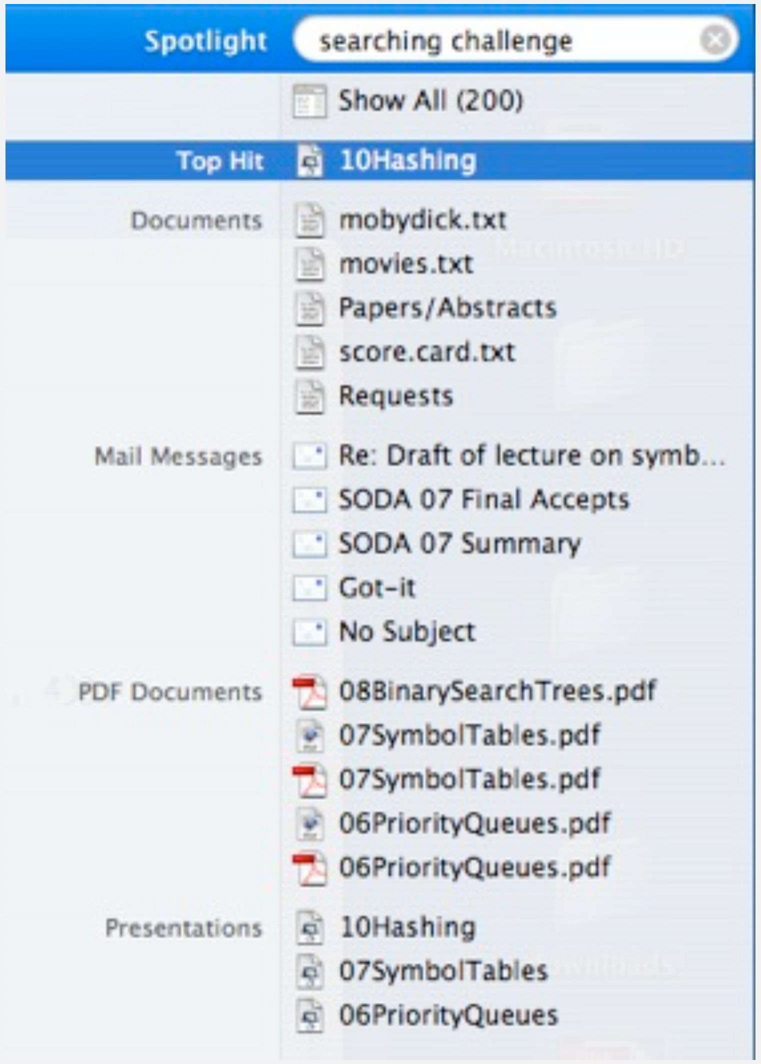
Assumptions. 1 billion++ words to index.

Which searching method to use?

✓ • Hashing
• Red-black-trees ← too much space
• Doesn't matter much.

Solution. Symbol table with:
• Key = query string.
• Value = set of pointers to files.

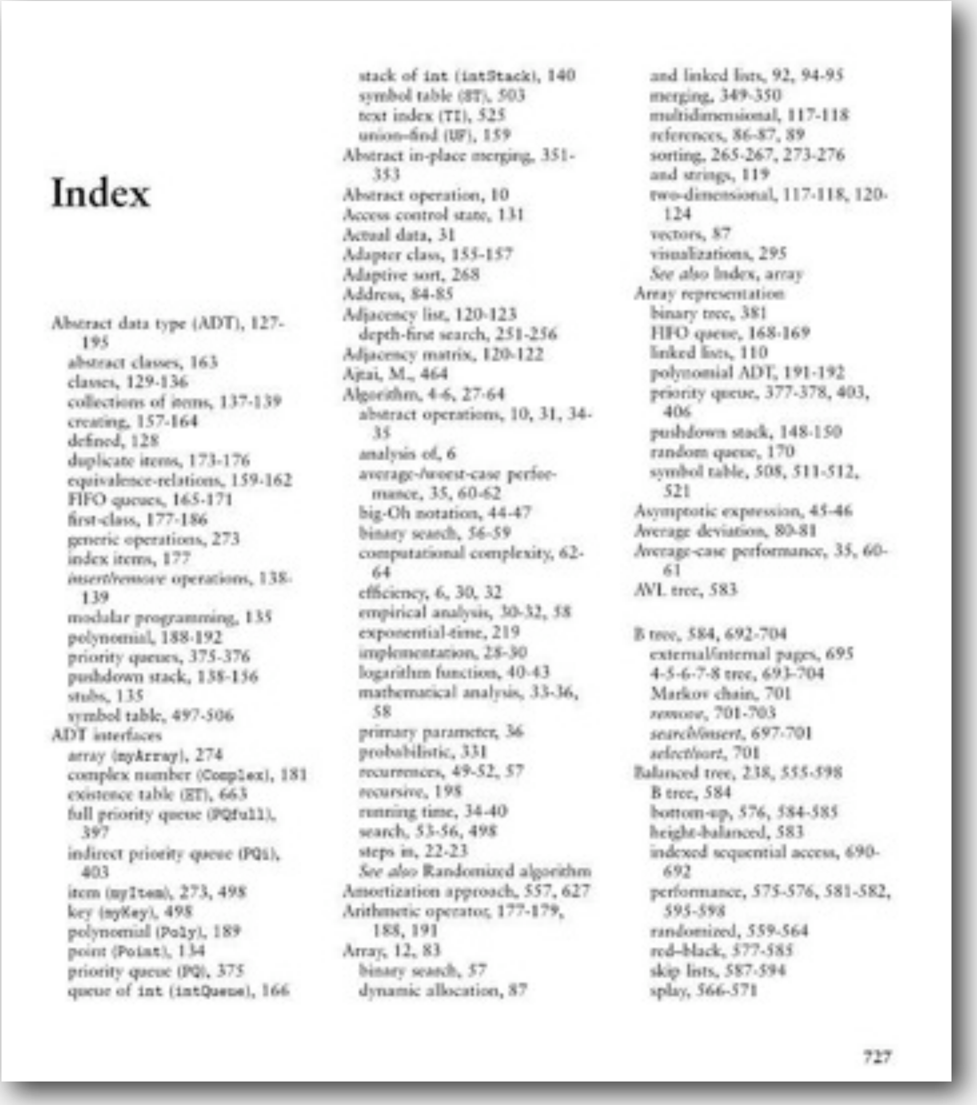
sort the (relatively few) search hits



Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.


Index

727

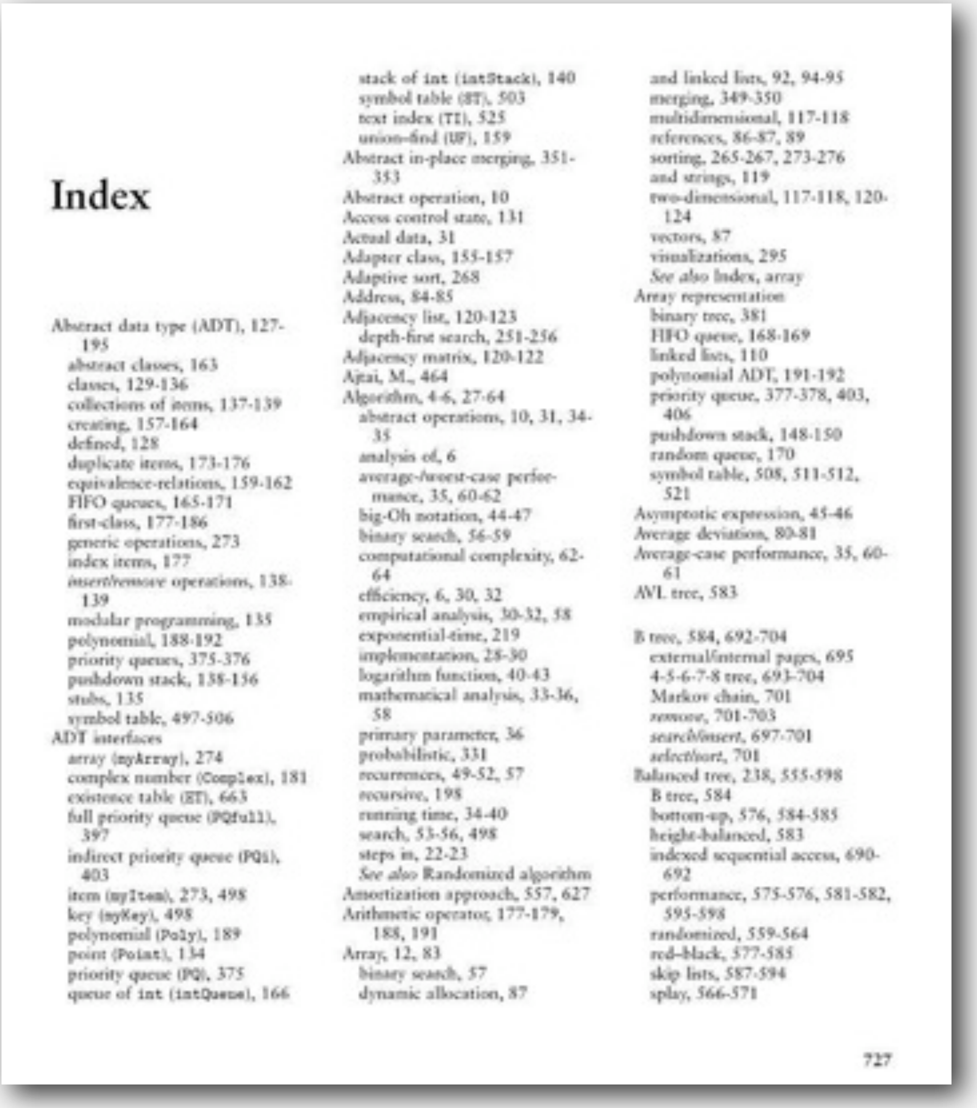
Which searching method to use?

1. Hashing
2. Red-black-tree
3. Doesn't matter much.

Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.



Index

727

Which searching method to use?

1. Hashing
2. Red-black-tree ✓ ← need ordered iteration
3. Doesn't matter much.

Solution. Symbol table with:

- Key = index term.
- Value = ordered set of pages on which term appears.