3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- applications

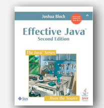
Optimize judiciously

> *" 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()`

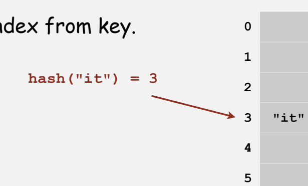
Q. Can we do better?

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.

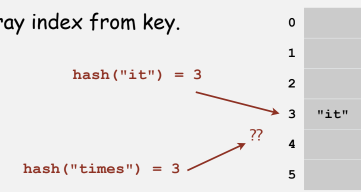


Issues.
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

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.

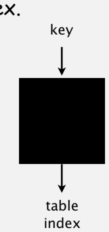
Classic space-time tradeoff.
- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

- 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.

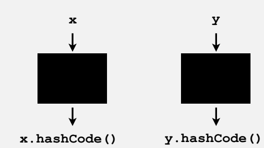
Practical challenge. Need different approach for each key type.

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 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

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

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


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) val).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}

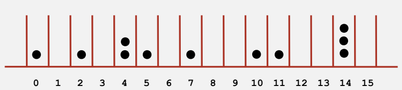
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

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.

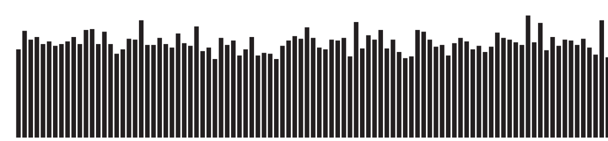
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

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

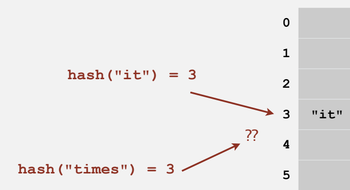
- hash functions
- **separate chaining**
- linear probing
- applications

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.

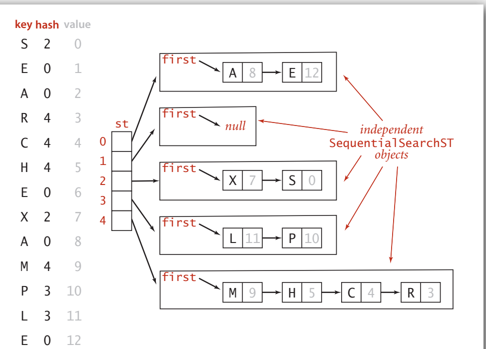
Challenge. Deal with collisions efficiently.



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

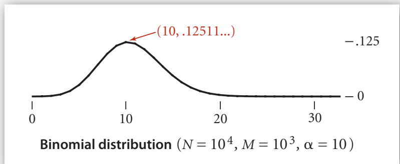
```
public class SeparateChainingHashST<Key, Value>
{
   private int N;      // number of key-value pairs
   private int M;      // hash table size
   private SequentialSearchST<Key, Value> [] st;   // array of STs

   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

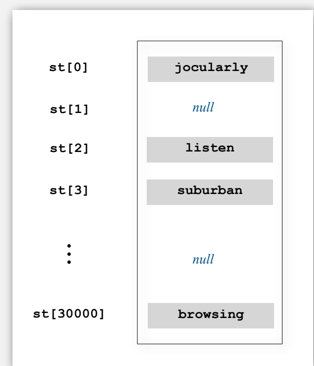
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`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

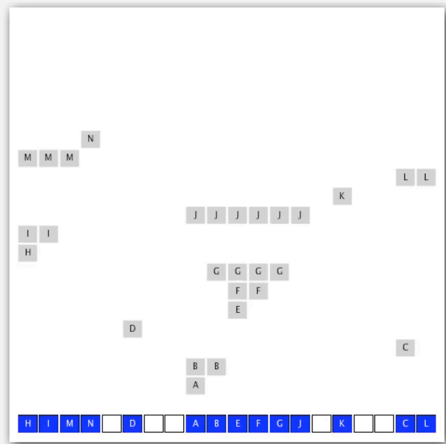
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   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.

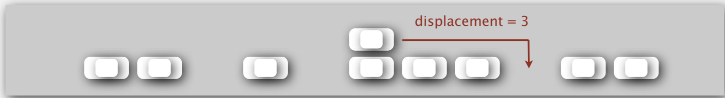
Observation. New keys likely to hash into middle of big clusters.



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: $\alpha = N / M \sim ½$.

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

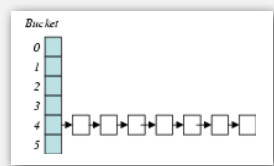
ST implementations: summary

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. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ...

known to be insecure

```
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.

Memory usage for ST implementations

Separate chaining

- N nodes (each 24 bytes overhead plus three 8-byte references)
- M nodes (each 24 bytes overhead plus one 8-byte reference)
- Total: ~48N + 32M = ~54.4N bytes for recommended M = N/5

Linear probing

- 2N references if 1/2 full
- 8N references if 1/8 full
- Total: between ~32N and ~128N bytes

Binary search trees

- N nodes (each 24 bytes overhead plus four 8-byte references)
- Total: ~56N bytes

Bottom line: Memory usage not decisive in reference implementations

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).
- Not easy to implement `equals()` and `hashCode()` correctly.

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Faster for complicated keys (compare vs. arithmetic ops on whole key)
- Not difficult to implement `compareTo()` correctly.

Java system includes both.

- Red-black trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
- geometric applications

- **sets**
- dictionary clients
- indexing clients
- sparse vectors

Set API

Mathematical set. A collection of distinct keys.

public class SET<Key extends Comparable<Key>>		
	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add the key to the set*
`boolean`	`contains(Key key)`	*is the key in the set?*
`void`	`remove(Key key)`	*remove the key from the set*
`int`	`size()`	*return the number of keys in the set*
`Iterator<Key>`	`iterator()`	*iterator through keys in the set*

Q. How to implement?

Exception filter

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
% more list.txt
was it the of

% java WhiteList list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackList list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

list of exceptional words

Exception filter applications

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class WhiteList
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create empty set of strings

read in whitelist

print words in list

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class BlackList
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (!set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create empty set of strings

read in blacklist

print words not in list

- sets
- **dictionary clients**
- indexing clients
- sparse vectors

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 1. DNS lookup.

URL is key, IP is value

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
```

IP is key, URL is value

```
% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
```

```
% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...
```


Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

codon is key, name is value

```
% java LookupCSV amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...
```


Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 3. Class list.

login is key, first name is value

```
% java LookupCSV classlist.csv 4 1
eberl
Ethan
nwebb
Natalie
```

login is key, precept is value

```
% java LookupCSV classlist.csv 4 3
dpan
P01
```

```
% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
11,Bourque,Alexander Joseph,P01,abourque
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cchehoud
10,Douglas,Malia Morioka,P01,malia
12,Haddock,Sara Lynn,P01,shaddock
12,Hantman,Nicole Samantha,P01,nhantman
11,Hesterberg,Adam Classen,P01,ahesterb
13,Hwang,Roland Lee,P01,rhwang
13,Hyde,Gregory Thomas,P01,ghyde
13,Kim,Hyunmoon,P01,hktwo
11,Kleinfeld,Ivan Maximillian,P01,ikleinfe
12,Korac,Damjan,P01,dkorac
11,MacDonald,Graham David,P01,gmacdona
10,Michal,Brian Thomas,P01,bmichal
12,Nam,Seung Hyeon,P01,seungnam
11,Nastasescu,Maria Monica,P01,mnastase
11,Pan,Di,P01,dpan
12,Partridge,Brenton Alan,P01,bpartrid
13,Rilee,Alexander,P01,arilee
13,Roopakalu,Ajay,P01,aroopaka
11,Sheng,Ben C,P01,bsheng
12,Webb,Natalie Sue,P01,nwebb
...
```


Dictionary lookup: Java implementation

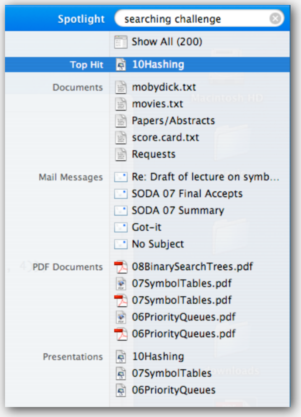
```
public class LookupCSV
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);

      ST<String, String> st = new ST<String, String>();
      while (!in.isEmpty())
      {
         String line = in.readLine();
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }

      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}
```

- process input file
- build symbol table
- process lookups with standard I/O

- sets
- dictionary clients
- **indexing clients**
- sparse vectors

File indexing

Goal. Index a PC (or the web).



File indexing

Goal. Given a list of files specified as command-line arguments, create an index so that can efficiently find all files containing a given query string.

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt
freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt
```

```
% ls *.java

% java FileIndex *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

import
FileIndex.java SET.java ST.java

Comparator
null
```

Solution. Key = query string; value = set of files containing that string.

File indexing

```
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();

      for (String filename : args) {
         File file = new File(filename);
         In in = new In(file);
         while !(in.isEmpty())
         {
            String word = in.readString();
            if (!st.contains(word))
               st.put(s, new SET<File>());
            SET<File> set = st.get(key);
            set.add(file);
         }
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         StdOut.println(st.get(query));
      }
   }
}
```

symbol table

list of file names from command line

for each word in file, add file to corresponding set

process queries

Keyword-in-context search

Goal. Preprocess a text to support KWIC queries: given a word, find all occurrences with their immediate contexts.

```
% java KWIC tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
  of his most gracious *majesty* king george the third

princeton
no matches
```


KWIC

```
public class KWIC
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      String[] words = StdIn.readAll().split("\\s+");
      ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
      for (int i = 0; i < words.length; i++)
      {
         String s = words[i];
         if (!st.contains(s))
            st.put(s, new SET<Integer>());
         SET<Integer> pages = st.get(s);
         set.put(i);
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         SET<Integer> set = st.get(query);
         for (int k : set)
            // print words[k-5] to words[k+5]
      }
   }
}
```

read text and build index

process queries and print matches

- sets
- dictionary clients
- indexing clients
- **sparse vectors**

Matrix-vector multiplication (standard implementation)

a[][] x[] = b[]

$$\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} = \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}$$

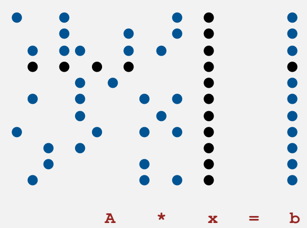
```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

nested loops (N^2 running time)

Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.



Vector representations

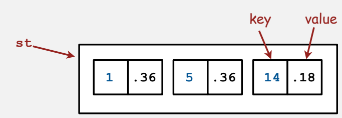
1D array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	.36	0	0	0	.36	0	0	0	0	0	0	0	0	.18	0	0	0	0	0

Symbol table representation.

- Key = index, value = entry.
- Efficient iterator.
- Space proportional to number of nonzeros.



Sparse vector data type

```
public class SparseVector
{
   private HashST<Integer, Double> v;

   public SparseVector()
   {  v = new HashST<Integer, Double>();  }

   public void put(int i, double x)
   {  v.put(i, x);  }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }

   public Iterable<Integer> indices()
   {  return v.keys();  }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

HashST because order not important

empty ST represents all 0s vector

a[i] = value

return a[i]

dot product is constant time for sparse vectors

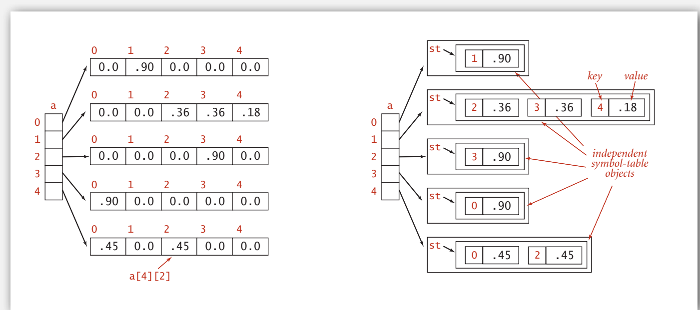
Matrix representations

2D array (standard) matrix representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse matrix representation: Each row of matrix is a sparse vector.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).



Sparse matrix-vector multiplication

$$\underbrace{\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}}_{a[][]} \underbrace{\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix}}_{x[]} = \underbrace{\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}}_{b[]}$$

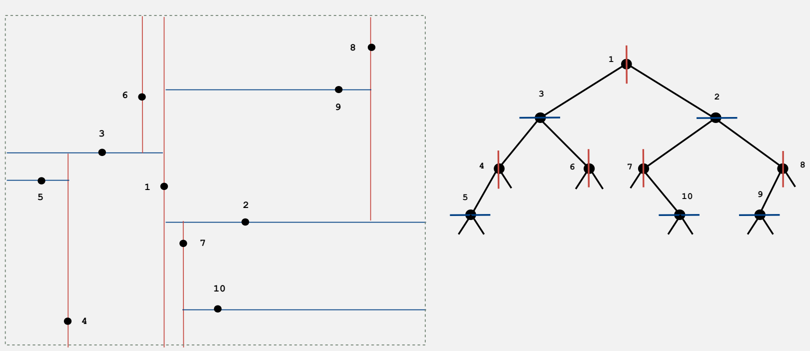
```
..
SparseVector[] a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[] and x[]
...
for (int i = 0; i < N; i++)
   b[i] = a[i].dot(x);
```

linear running time for sparse matrix (the `for` loop)

- sets
- dictionary clients
- indexing clients
- sparse vectors
- challenges
- **geometric applications**

2d tree

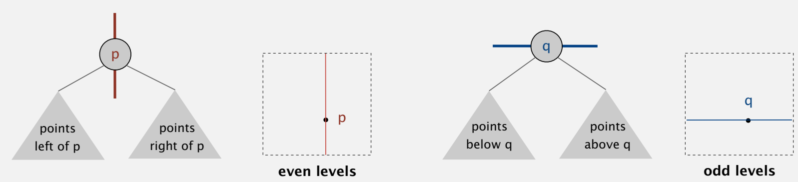
Recursively partition plane into two halfplanes.

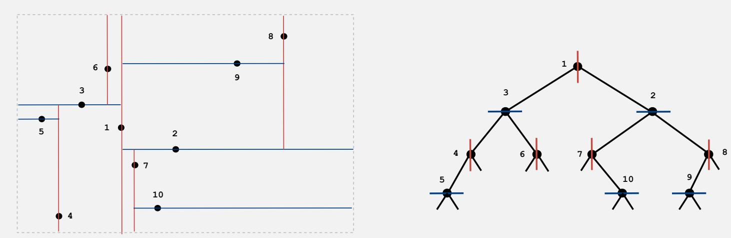


2d tree implementation

Data structure. BST, but alternate using x- and y-coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.





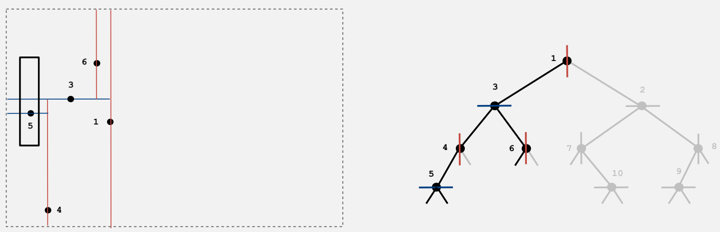
2d tree: 2d orthogonal range search

Range search. Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/top subdivision (if any could fall in rectangle).
- Recursively search right/bottom subdivision (if any could fall in rectangle).

Typical case. $R + \log N$.

Worst case (assuming tree is balanced). $R + \sqrt{N}$.



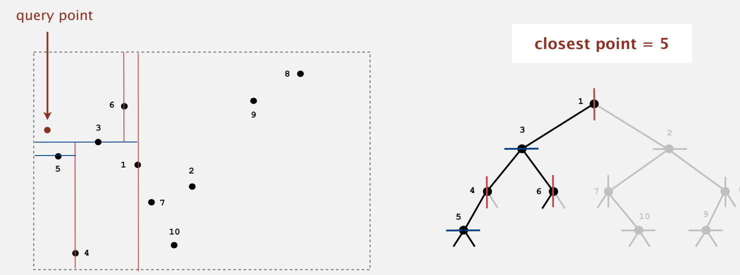
2d tree: nearest neighbor search

Nearest neighbor search. Given a query point, find the closest point.

- Check distance from point in node to query point.
- Recursively search left/top subdivision (if it could contain a closer point).
- Recursively search right/bottom subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Typical case. $\log N$.

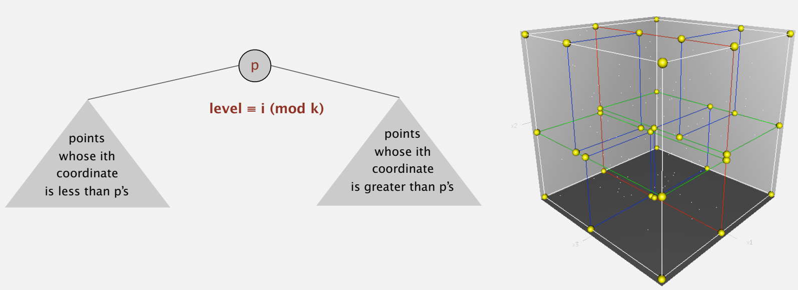
Worst case (even if tree is balanced). N.



Kd tree

Kd tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.



Efficient, simple data structure for processing k-dimensional data.

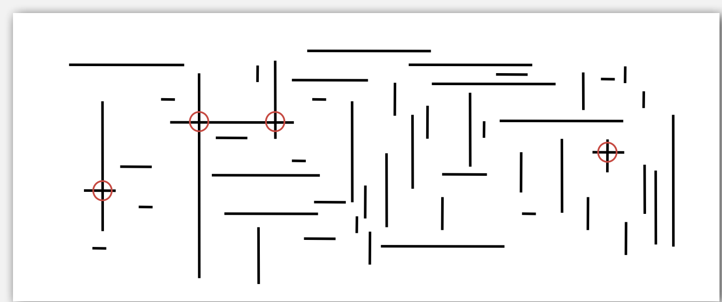
- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad (Jon Bentley) in an algorithms class!

Search for intersections

Problem. Find all intersecting pairs among N geometric objects.

Applications. CAD, games, movies, virtual reality, ...

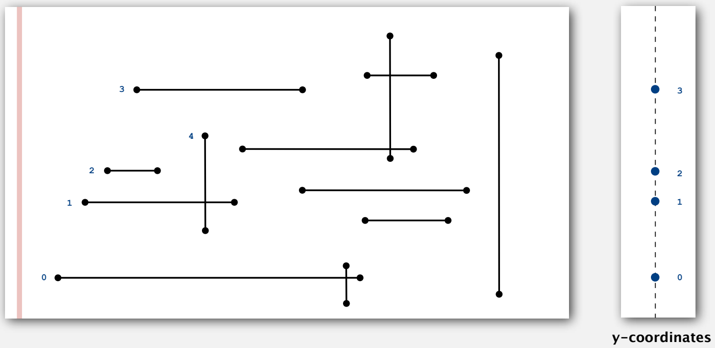
Simple version. 2d, all objects are horizontal or vertical line segments.



Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.

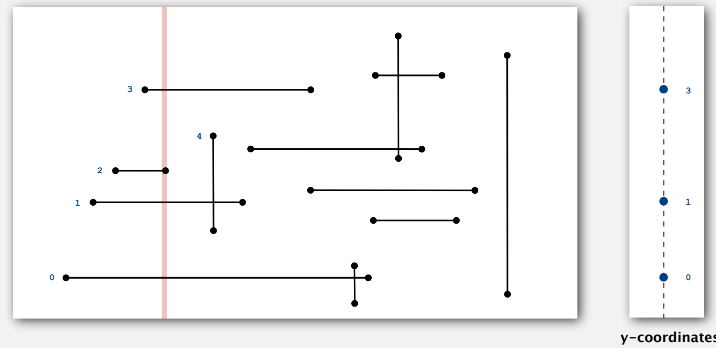
Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.



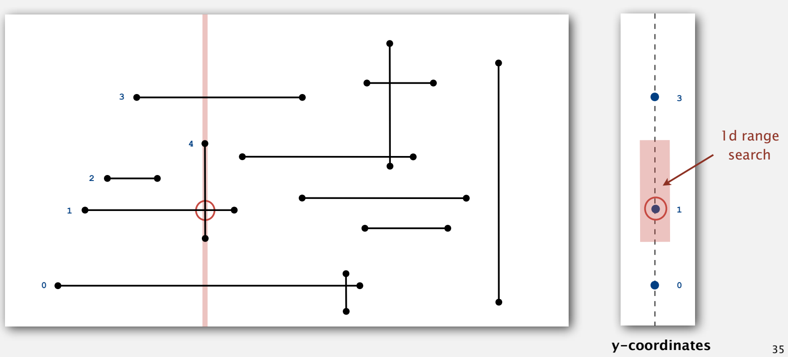
Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.
- h-segment (right endpoint): remove y-coordinate from ST.



Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.
- h-segment (right endpoint): remove y-coordinate from ST.
- v-segment: range search for interval of y-endpoints.



Sweep line reduces 2d orthogonal line segment intersection to 1d range search.

Proposition. The sweep-line algorithm takes time proportional to $N \log N + R$ to find all R intersections among N orthogonal segments.

- Put x-coordinates on a PQ (or sort). $N \log N$
- Insert y-coordinates into ST. $N \log N$
- Delete y-coordinates from ST. $N \log N$
- Range searches. $N \log N + R$

Efficiency relies on judicious use of data structures.

Remark. Sweep-line solution extends to 3d and more general shapes.

- sets
- dictionary clients
- indexing clients
- sparse vectors
- challenges

Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.
Assumption A. Hundreds of songs.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.
Assumption B. Thousands of songs.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2A

Problem. IP lookups in a web monitoring device.
Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2B

Problem. IP lookups in a web monitoring device.
Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in "Tale of Two Cities."
Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) BSTs.
4) Hashing.

Searching challenge 4

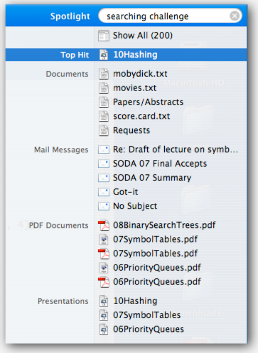
Problem. Spell checking for a book.
Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) BST.
4) Hashing.

Searching challenge 5

Problem. Index for a PC or the web.
Assumptions. 1 billion++ words to index.

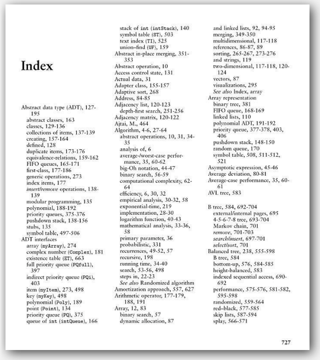
Which searching method to use?
- Hashing
- LLRB trees
- Doesn't matter much.



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
3. Doesn't matter much.