Hashing

- hash functions
- collision resolution
- applications

References:
Algorithms in Java, Chapter 14
`http://www.cs.princeton.edu/algs4/44hash`

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

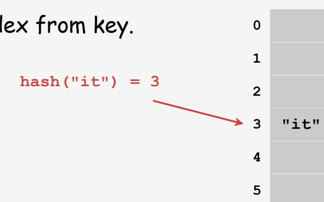
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
ordered array	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	3 lg N	3 lg N	3 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes	`compareTo()`

Q. Can we do better?

Hashing: basic plan

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

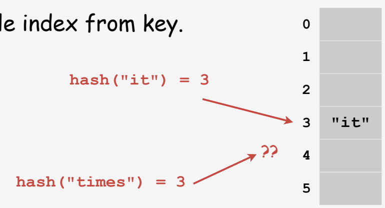
Hash function. Method for computing table index from key.



Hashing: basic plan

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

Hash function. Method for computing table 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 table index.

Classic space-time tradeoff.
- No space limitation: trivial hash function with key as address.
- No time limitation: trivial collision resolution with sequential search.
- Limitations on both time and space: hashing (the real world).

‣ hash functions
‣ collision resolution
‣ applications

Computing the hash function

Idealistic goal: scramble the keys uniformly.
- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications

Practical challenge. Need different approach for each `Key` type.

Ex: Social Security numbers.
- Bad: first three digits.
- Better: last three digits.

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

Ex: phone numbers.
- Bad: first three digits.
- Better: last three digits.

Hash codes and hash functions

Hash code. All Java classes have a method `hashCode()`, which returns an `int`.

between -2^{32} and $2^{31} - 1$

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

First attempt.

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

3045982

7121

8191

Bug. Don't use `(code % M)` as array index.

1-in-a billion bug. Don't use `(Math.abs(code) % M)` as array index.

OK. Safe to use `((code & 0x7fffffff) % M)` as array index.

hex literal 31-bit mask

Java's hash code conventions

The method `hashCode()` is inherited from `Object`.

- Ensures hashing can be used for every object type.
- Enables expert implementations for each type.

Available implementations.

- Default implementation: memory address of `x`.
- Customized implementations: `String`, `URL`, `Integer`, `Date`, ...
- User-defined types: users are on their own.

Implementing hash code: phone numbers

Ex. Phone numbers: (609) 867-5309.

```
public final class PhoneNumber
{
   private final int area, exch, ext;

   public PhoneNumber(int area, int exch, int ext)
   {
      this.area = area;
      this.exch = exch;
      this.ext  = ext;
   }

   ...

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

   public int hashCode()
   {  return 10007 * (area + 1009 * exch) + ext;  }
}
```

sufficiently random?

Implementing hash code: strings

Ex. Strings (in Java 1.5).

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

ith character of s

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

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

Ex.

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

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

Provably random? Well, no.

A poor hash code design

Ex. Strings (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 = (37 * hash) + s[i];
   return hash;
}
```

- Downside: great potential for bad collision patterns.

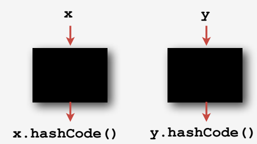
```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/13loop/index.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Designing a good hash function

Requirements.

- If `x.equals(y)`, then we must also have `(x.hashCode() == y.hashCode())`.
- Repeated calls to `x.hashCode()` must return the same value (provided no info used in `equals()` is changed).

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



Basic rule. Need to use the whole key to compute hash code.

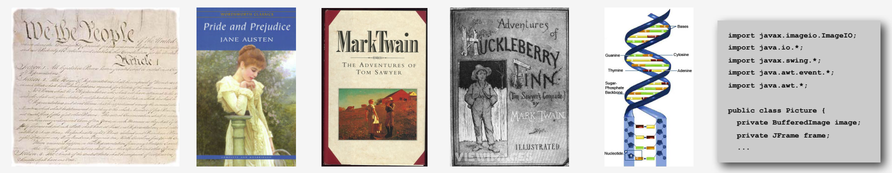
Fundamental problem. Need a theorem for each type to ensure reliability.

Digression: using a hash function for data mining

Use content to characterize documents.

Applications.

- Search documents on the web for documents similar to a given one.
- Determine whether a new document belongs in one set or another.



Context. Effective for literature, genomes, Java code, art, music, data, video.

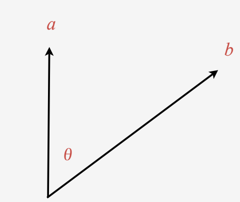
Digression: using a hash function for data mining

Approach.

- Fix order `k` and dimension `d`.
- Compute `(hashCode() % d)` for all `k`-grams in the document.
- Result is `d`-dimensional vector profile of each document.

To compare documents: Consider angle θ separating vectors

- $\cos\theta$ close to 0: not similar.
- $\cos\theta$ close to 1: similar.



$$\cos\theta = a \cdot b \,/\, |a|\;|b|$$

Digression: using a hash function for data mining

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of
foolishness
...

% more genome.txt
CTTTCGGTTTGGAACC
GAAGCCGCGCGTCT
TGTCTGCTGCAGC
ATCGTTC
...
```

	tale.txt		genome.txt	
i	10-grams with hash code i	freq	10-grams with hash code i	freq
0		0		0
1		0		0
...				
435	"best of ti" "foolishnes"	2	"TTTCGGTTTG" "TGTCTGCTGC"	2
...				
8999	"it was the"	8		0
...				
12122		0	"CTTTCGGTTT"	3
...				
34543	"t was the b"	5	"ATGCGGTCGA"	4
...				
65535				

k = 10
d = 65536

profile

Digression: using a hash function for data mining

```
public class Document
{
   private String name;
   private double[] profile;

   public Document(String name, int k, int d)
   {
      this.name = name;
      String doc = (new In(name)).readAll();
      int N = doc.length();
      profile = new double[d];
      for (int i = 0; i < N-k; i++)
      {
         int h = doc.substring(i, i+k).hashCode();
         profile[Math.abs(h % d)] += 1;
      }
   }

   public double simTo(Document that)
   {  /* compute dot product and divide by magnitudes */  }

}
```


Digression: using a hash function for data mining

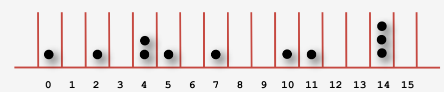
file	description
Cons	*US Constitution*
TomS	*Tom Sawyer*
Huck	*Huckleberry Finn*
Prej	*Pride and Prejudice*
Pict	*a photograph*
DJIA	*financial data*
Amaz	*amazon.com website .html source*
ACTG	*genome*

```
% java CompareAll 5 1000 < docs.txt
        Cons   TomS   Huck   Prej   Pict   DJIA   Amaz   ACTG
Cons    1.00   0.89   0.87   0.88   0.35   0.70   0.63   0.58
TomS    0.89   1.00   0.98   0.96   0.34   0.75   0.66   0.62
Huck    0.87   0.98   1.00   0.94   0.32   0.74   0.65   0.61
Prej    0.88   0.96   0.94   1.00   0.34   0.76   0.67   0.63
Pict    0.35   0.34   0.32   0.34   1.00   0.29   0.48   0.24
DJIA    0.70   0.75   0.74   0.76   0.29   1.00   0.62   0.58
Amaz    0.63   0.66   0.65   0.67   0.48   0.62   1.00   0.45
ACTG    0.58   0.62   0.61   0.63   0.24   0.58   0.45   1.00
```


- hash functions
- **collision resolution**
- applications

Helpful results from probability theory

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.

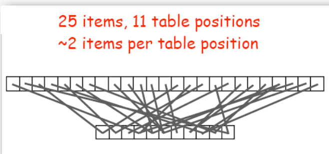
Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Collisions

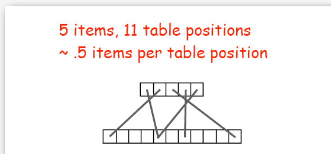
Collision. Two distinct keys hashing to same index.

- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount (quadratic) of memory.
- Coupon collector + load balancing ⇒ collisions will be evenly distributed.

Challenge. Deal with collisions efficiently.



approach 1: accept multiple collisions

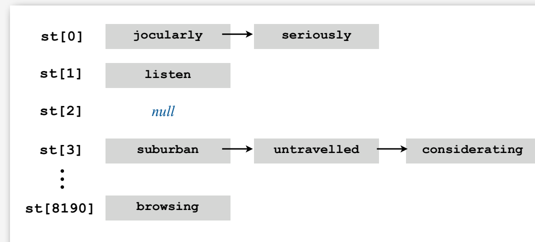


approach 2: minimize collisions

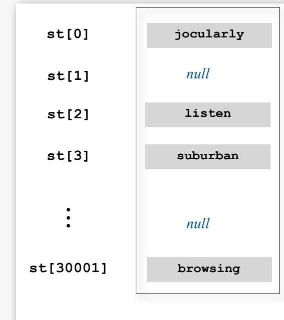
Collision resolution: two approaches

Separate chaining. [H. P. Luhn, IBM 1953]
Put keys that collide in a list associated with index.

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.



separate chaining (M = 8191, N = 15000)

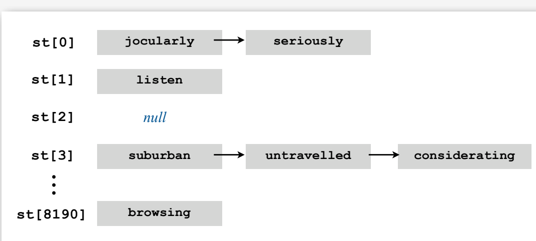


linear probing (M = 30001, N = 15000)

Collision resolution approach 1: separate chaining

Use an array of M < N linked lists. ← good choice: M ~ N/10

- 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.



separate chaining (M = 8191, N = 15000)

key	hash
"call"	7121
"me"	3480
"ishmael"	5017
"seriously"	0
"untravelled"	3
"suburban"	3
...	...

Separate chaining ST: Java implementation (skeleton)

```
public class ListHashST<Key, Value>
{
    private int M = 8191;
    private Node[] st = new Node[M];

    private class Node
    {
        private Object key;
        private Object val;
        private Node next;
        public Node(Key key, Value val, Node next)
        {
            this.key   = key;
            this.val   = val;
            this.next  = next;
        }
    }

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

   public void put(Key key, Value val)
   {  /* see next slide */  }

   public Val get(Key key)
      {  /* see next slide */  }
}
```

← array doubling code omitted

← no generics in arrays in Java

Separate chaining ST: Java implementation (put and get)

```
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, value, first);
}
```

```
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;
}
```

identical to linked-list code, except hash to pick a list

Analysis of separate chaining

Separate chaining performance.

- Cost is proportional to length of chain.
- Average length of chain α = N / M.
- Worst case: all keys hash to same chain.

Proposition. Let $\alpha > 1$. For any $t > 1$, probability that chain length $> t\,\alpha$ is exponentially small in t.

depends on hash map being random map

Parameters.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: M ~ N/10 $\Rightarrow$ constant-time ops.

Collision resolution approach 2: linear probing

Use an array of size M >> N. ← good choice: M ~ 2N

- Hash: map key to integer i between 0 and M-1.
- Insert: put in slot i if free; if not try i+1, i+2, etc.
- Search: search slot i; if occupied but no match, try i+1, i+2, etc.

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

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

insert I
hash(I) = 11

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

insert N
hash(N) = 8

Linear probing ST implementation

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

   privat 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 (key.equals(keys[i]))
             break;
      vals[i] = val;
      keys[i] = key;
   }

   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;
   }
}
```

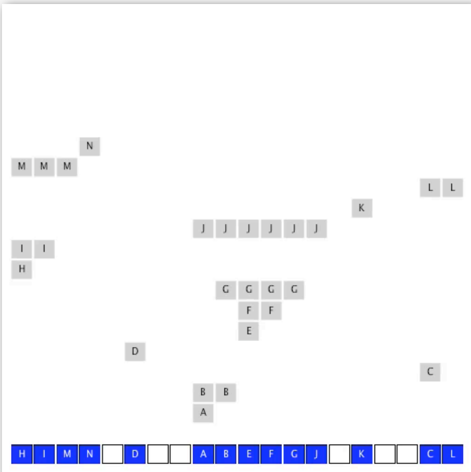
standard ugly casts

array doubling 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, …

Q. What is mean displacement of a car?

Empty. With M/2 cars, mean displacement is ~ 3/2.

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

Analysis of linear probing

Linear probing performance.

- Insert and search cost depend on length of cluster.
- Average length of cluster α = N / M. ← but keys more likely to hash to big clusters
- Worst case: all keys hash to same cluster.

Proposition. [Knuth 1962] Let α < 1 be the load factor.

average probes for insert/search miss

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) = (1 + \alpha + 2\alpha^2 + 3\alpha^3 + 4\alpha^4 + \ldots)/2$$

average probes for search hit

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right) = 1 + (\alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots)/2$$

Parameters.

- Load factor too small ⇒ too many empty array entries.
- Load factor too large ⇒ clusters coalesce.
- Typical choice: M ~ 2N ⇒ constant-time ops.

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 average 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.

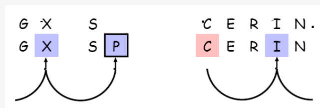
Double hashing

Idea. Avoid clustering by using second hash to compute skip for search.

Hash function. Map key to integer i between 0 and M-1.
Second hash function. Map key to nonzero skip value k.

Ex: k = 1 + (v mod 97).

hashCode()



Effect. Skip values give different search paths for keys that collide.

Best practices. Make k and M relatively prime.

Double hashing performance

Theorem. [Guibas-Szemerédi] Let $\alpha = N / M < 1$ be average length of cluster.

Average probes for insert/search miss

$$\frac{1}{(1-\alpha)} = 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots$$

Average probes for search hit

$$\frac{1}{\alpha} \ln \frac{1}{(1-\alpha)} = 1 + \alpha/2 + \alpha^2/3 + \alpha^3/4 + \alpha^4/5 + \ldots$$

Parameters. Typical choice: $\alpha \sim 1.2 \Rightarrow$ constant-time ops.

Disadvantage. Deletion is cumbersome to implement.

Hashing Tradeoffs

Separate chaining vs. linear probing/double hashing.

- Space for links vs. empty table slots.
- Small table + linked allocation vs. big coherent array.

Linear probing vs. double hashing.

		load factor			
		50%	66%	75%	90%
linear probing	get	1.5	2.0	3.0	5.5
	put	2.5	5.0	8.5	55.5
double hashing	get	1.4	1.6	1.8	2.6
	put	1.5	2.0	3.0	5.5

number of probes

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
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()`
randomized BST	3 lg N	3 lg N	3 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes	`compareTo()`
hashing	1 *	1 *	1 *	1 *	1 *	1 *	no	`equals()` `hashCode()`

* assumes random hash function

Hashing versus balanced 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).
- Does your hash function produce random values for your key type??

Balanced trees.

- Stronger performance guarantee.
- Can support many more ST operations for ordered keys.
- 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`.

Typical "full" symbol table API

public class *ST<Key extends Comparable<Key>, Value> implements Iterable<Key>		
	*ST()	*create an empty symbol table*
void	put(Key key, Value val)	*put key-value pair into the table*
Value	get(Key key)	*return value paired with key; null if no such value*
boolean	contains(Key key)	*is there a value paired with key?*
Key	min()	*return smallest key*
Key	max()	*return largest key*
Key	ceil(Key key)	*return smallest key in table ≥ query key*
Key	floor(Key key)	*return largest key in table ≤ query key*
void	remove(Key key)	*remove key-value pair from table*
Iterator<Key>	iterator()	*iterator through keys in table*

Hashing is not suitable for implementing such an API (no order).
BSTs are easy to extend to support such an API (basic tree ops).

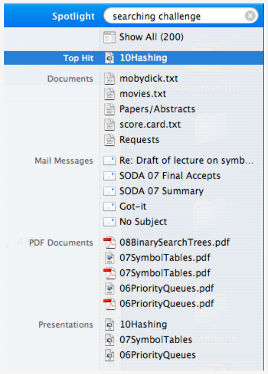
- hash functions
- collision resolution
- **applications**

Searching challenge

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

Which searching method to use?

- Hashing implementation of ST.
- Hashing implementation of SET.
- Red-black-tree implementation of ST.
- Red-black-tree implementation of SET.
- Doesn't matter much.



Index for search in a PC

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File f : filesystem)
{
   In in = new In(f);
   String[] words = in.readAll().split("\\s+");
   for (int i = 0; i < words.length; i++)
   {
      String s = words[i];
      if (!st.contains(s))
         st.put(s, new SET<File>());
      SET<File> files = st.get(s);
      files.add(f);
   }
}
```

build index

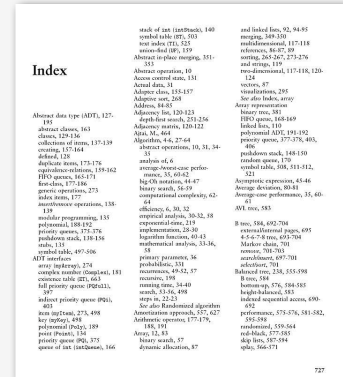
```
SET<File> files = st.get(s);
for (File f : files) ...
```

process lookup request

Searching challenge

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.



Which searching method to use?

- Hashing implementation of ST.
- Hashing implementation of SET.
- Red-black-tree implementation of ST.
- Red-black-tree implementation of SET.
- Doesn't matter much.

Index for a book

```
public class Index
{
   public static void main(String[] args)
   {
      String[] words = StdIn.readAll().split("\\s+");
      ST<String, SET<Integer>> st;
      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);
         pages.add(page(i));
      }

      for (String s : st)
         StdOut.println(s + ": " + st.get(s));

   }
}
```

read book and create ST

process all words

print index

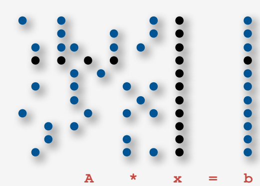
Searching challenge 5

Problem. Sparse matrix-vector multiplication.

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

Which searching method to use?

1) Unordered array.
2) Ordered linked list.
3) Ordered array with binary search.
4) Need better method, all too slow.
5) Doesn't matter much, all fast enough.



Sparse vectors and matrices

Vector. Ordered sequence of N real numbers.
Matrix. N-by-N table of real numbers.

vector operations

$$a = [\,0 \quad 3 \quad 15\,], \quad b = [-1 \quad 2 \quad 2\,]$$
$$a + b = [-1 \quad 5 \quad 17\,]$$
$$a \circ b = (0 \cdot -1) + (3 \cdot 2) + (15 \cdot 2) = 36$$
$$|a| = \sqrt{a \circ a} = \sqrt{0^2 + 3^2 + 15^2} = 3\sqrt{26}$$

matrix-vector multiplication

$$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix} \times \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$$

Sparse vectors and matrices

An N-by-N matrix is sparse if it contains O(N) nonzeros.

Property. Large matrices that arise in practice are sparse.

2D array matrix representation.

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

Goal.

- Efficient access to elements.
- Space proportional to number of nonzeros.

Sparse vector data type

```
public class SparseVector
{
   private int N;                    // length
   private ST<Integer, Double> st;   // the elements

   public SparseVector(int N)
   {
      this.N  = N;
      this.st = new ST<Integer, Double>();
   }

   public void put(int i, double value)
   {
      if (value == 0.0) st.remove(i);
      else              st.put(i, value);
   }

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

   ...
```

all 0s vector

a[i] = value

return a[i]

Sparse vector data type (cont)

```
   public double dot(SparseVector that)
   {
      double sum = 0.0;
      for (int i : this.st)
         if (that.st.contains(i))
            sum += this.get(i) * that.get(i);
      return sum;
   }

   public double norm()
   {  return Math.sqrt(this.dot(this));  }

   public SparseVector plus(SparseVector that)
   {
      SparseVector c = new SparseVector(N);
      for (int i : this.st)
         c.put(i, this.get(i));
      for (int i : that.st)
         c.put(i, that.get(i) + c.get(i));
      return c;
   }

}
```

dot product

2-norm

vector sum

Sparse matrix data type

```
public class SparseMatrix
{
   private final int N;            // length
   private SparseVector[] rows;    // the elements

   public SparseMatrix(int N)
   {
      this.N  = N;
      this.rows = new SparseVector[N];
      for (int i = 0; i < N; i++)
         this.rows[i] = new SparseVector(N);
   }

   public void put(int i, int j, double value)
   {  rows[i].put(j, value);  }

   public double get(int i, int j)
   {  return rows[i].get(j);  }

   public SparseVector times(SparseVector x)
   {
      SparseVector b = new SparseVector(N);
      for (int i = 0; i < N; i++)
         b.put(i, rows[i].dot(x));
      return b;
   }
}
```

all 0s matrix

a[i][j] = value

return a[i][j]

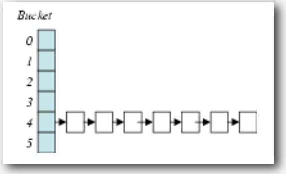
matrix-vector multiplication

Hashing in the wild: algorithmic complexity attacks

Is the random hash map assumption important in practice?

- Obvious situations: aircraft control, nuclear reactor, pacemaker.
- 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 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!

Q. Does your hash function produce random values for your key type?

One-way hash functions

One-way hash function. Hard to find a key that will hash to a desired value, or to find 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.