Hashing

- hash functions
- separate chaining
- linear probing
- 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

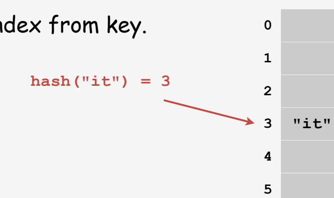
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`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

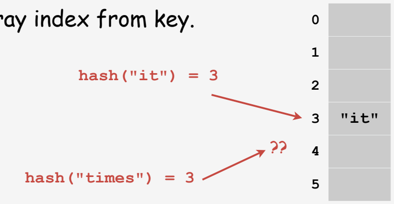


Issues.
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

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.
- Limitations on both time and space: hashing (the real world).

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

Equality test

All Java classes have a method `equals()`, inherited from `Object`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.

do x and y refer to the same object?

Default implementation (inherited from `Object`). `(x == y)`

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, …

User-defined implementations. Some care needed.

Implementing equals: phone numbers

Seems easy, but requires some care.

```
public       class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Record that)
   {
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

check that all significant fields are the same

Implementing equals: phone numbers

Seems easy, but requires some care.

```
public final class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Object y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      Record that = (Record) y;
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

- no safe way to use `equals()` with inheritance
- must be `Object`. Why? Experts still debate.
- optimize for true object equality
- check for `null`
- objects must be in the same class
- check that all significant fields are the same

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

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.

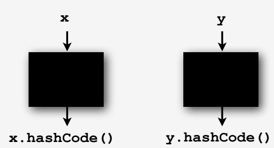
Practical challenge. Need different approach for each key type.

Java's hash code conventions

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

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

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



Default implementation (inherited from `Object`). Memory address of `x`.

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, ...

User-defined types. Users are on their own.

Implementing hash code: integers 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

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;
   }
}
```

ith 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();
```

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

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

A poor hash code

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


Implementing hash code: user-defined types

```
public final class Record
{
   private String name;
   private int id;
   private double value;

   public Record(String name, int id, double value)
   {  /* as before */  }

   ...

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

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + name.hashCode();
      hash = 31*hash + id;
      hash = 31*hash + Double.valueOf(value).hashCode();
      return hash;
   }
}
```

nonzero constant

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 built-in hash code.
- If field is an array, apply to each element.
- If field is an object, apply 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.

Hash functions

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

Bug.

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

1-in-a billion bug.

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

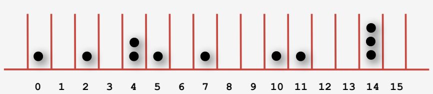
Correct.

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


Helpful results from probability theory

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.

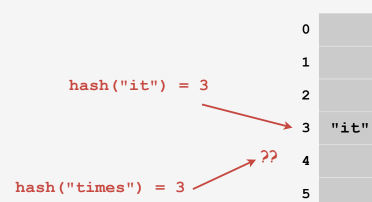
Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Collisions

Collision. Two distinct keys hashing to same index.

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

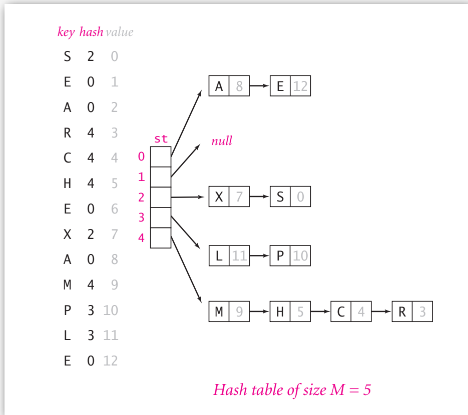
Challenge. Deal with collisions efficiently.



Collision resolution: separate chaining

Separate chaining. [H. P. Luhn, IBM 1953]

Put keys that collide in a list associated with index.

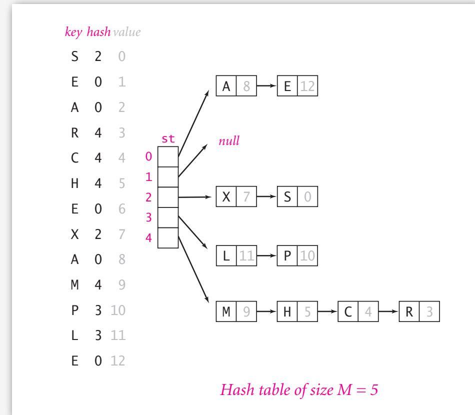


Hash table of size M = 5

Separate chaining ST

Use an array of M < N linked lists.

- 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.



Hash table of size M = 5

Separate chaining ST: Java implementation

```
public class SeparateChainingST<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)
   {  /* as before      */  }

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

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

array doubling code omitted

no generic array creation in Java

Separate chaining ST: Java implementation (put and get)

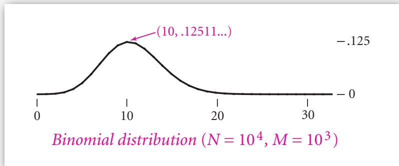
```
public void put(Key key, Value val)
{
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key))
      {  x.val = val; return;  }
   st[i] = new Node(key, value, first);
}

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 sequential search, except hash to pick a list

Analysis of separate chaining

Proposition. Under uniform hashing assumption, probability that the number of keys in each 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 ($N = 10^4$, $M = 10^3$)

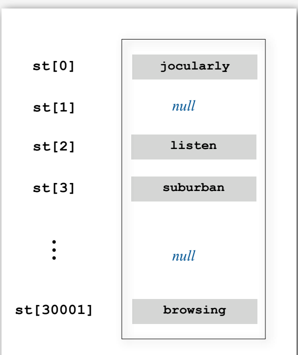
Consequence. Number of compares for search/insert is proportional to N/M.
- M too large ⇒ too many empty chains.
- M too small ⇒ chains too long.
- Typical choice: M ~ N/5 ⇒ constant-time ops.

M times faster than sequential search

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

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-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 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

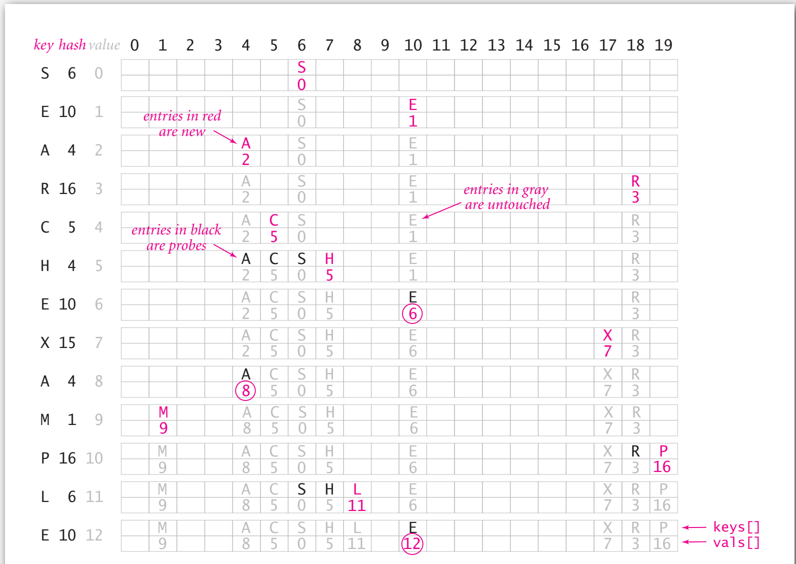
–	–	–	S	H	–	–	A	C	E	R	I	–
0	1	2	3	4	5	6	7	8	9	10	11	12

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: trace of standard indexing client



Linear probing hash table of size M = 20

Linear probing ST implementation

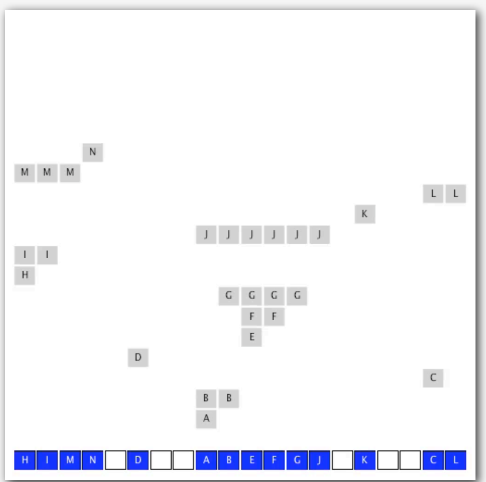
```
public class LinearProbingST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   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 (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;
   }
}
```

array doubling code omitted

Clustering

Cluster. A contiguous block of items.

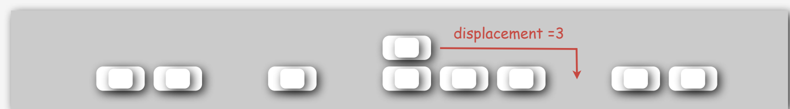
Observation. New keys likely to hash into middle of big clusters.



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

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 ⇒ too many empty array entries.
- M too small ⇒ search time blows up.
- Typical choice: α = N/M < 1/2 ⇒ constant-time ops.

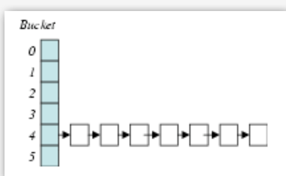
ST implementations: summary

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()`
hashing	lg N*	lg N*	lg N*	3-5*	3-5*	3-5*	no	`equals()`

* under uniform hashing assumption

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 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.

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 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.

Hashing vs. 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).

Balanced trees.

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

Java system includes both.

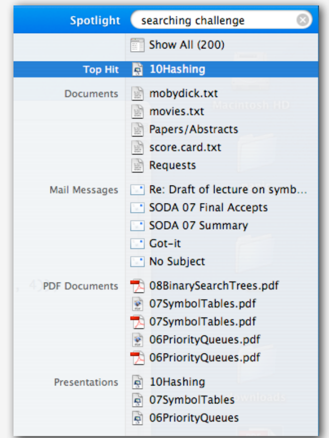
- Red-black trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

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

Searching challenge 1

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.



Index for a PC or the web

Solution. Symbol table with:

- Key = query string.
- Value = set of pointers to files.

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File file : filesystem)
{
   In in = new In(file);
   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(file);
   }
}
```

build index

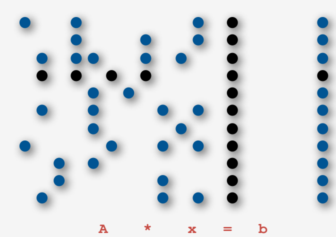
```
SET<File> files = st.get(query);
for (File file : files) ...
```

process lookup request

Searching challenge 2

Problem. Sparse matrix-vector multiplication.

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



Vectors and matrices

Vector. Ordered sequence of N real numbers.

Matrix. N-by-N table of real numbers.

vector operations

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

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

Sparse vector. An N-dimensional vector is sparse if it contains O(1) nonzeros.

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

Property. Large matrices that arise in practice are sparse.

$$[\,0 \;\; 0 \;\; .36 \;\; .36 \;\; .18\,]$$

$$\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}$$

Vector representations

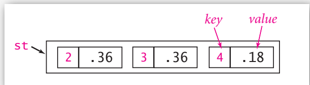
1D array representation.

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

0	1	2	3	4
0.0	0.0	.36	.36	.18

Symbol table representation.

- 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 (constructor)

a[i] = value (put)

return a[i] (get)

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

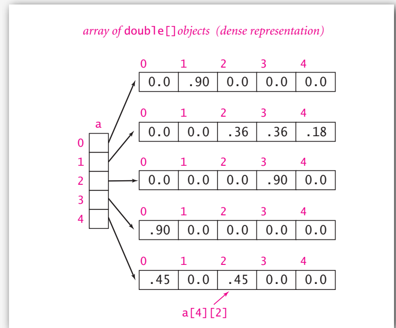
Matrix representations

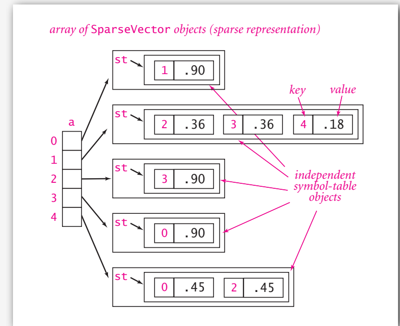
2D array matrix representation.

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

Sparse representation. Represent each row of matrix as a sparse vector!

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





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