3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Premature optimization

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.” – Donald Knuth

Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

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.

```
hash("it") = 3
```

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).
### 3.4 Hash Tables

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

#### Computing the hash function

**Idealistic goal.** Scramble the keys uniformly to produce a table index.
- Efficiently computable.
- Each table index equally likely for each key.

**Ex 1. Phone numbers.**
- Bad: first three digits.
- Better: last three digits.

**Ex 2. Social Security numbers.**
- 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())`.

#### Implementing hash code: integers, booleans, and doubles

##### Java library implementations

```java
public final class Integer {
    private final int value;
    ...

    public int hashCode() {
        return value;
    }
}
```

```java
public final class Double {
    private final double value;
    ...

    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

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

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

**Warning:** `hull` rotation, still problematic in practical applications

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

**Default implementation.** Memory address of `x`.

**Legal (but poor) implementation.** Always return 17.

**Customized implementations.** `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

**User-defined types.** Users are on their own.
Implementing hash code: strings

Treat string as $L$-digit, base-31 number:

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

Java library implementation

Horner’s method: only $L$ multiplies/adds to hash string of length $L$.

```
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;
  }
}
```

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

Hash code design

“Standard” recipe for user-defined types.

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

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

In theory. Keys are bitstring; “universal” hash functions exist.

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 \(-2^{31} + 2^{31} - 1\).

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

Typically a prime or power of 2

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

1-in-a-billion bug

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

Correct

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

---

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 \(\geq 1\) ball after \(\sim M \ln M\) tosses.

Load balancing. After \(M\) tosses, expect most loaded bin has \(\sim M / \ln \ln M\) balls.

---

3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

http://algs4.cs.princeton.edu

Robert Sedgewick  |  Kevin Wayne

Algorithms

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

**Collisions**. Two distinct keys hashing to same index.
- Birthday problem ⇒ can’t avoid collisions.
- Coupon collector ⇒ not too much wasted space.
- Load balancing ⇒ no index gets too many collisions.

unless you have a ridiculous (quadratic) amount of memory

**Challenge.** Deal with collisions efficiently.

Separate-chaining symbol table: Java implementation

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

    private static class Node {
        private Object key;  // no generic array creation
        private Object val;  // (declare key and value of type Object)
        private Node next;
        ...
    }

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

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

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

Separate-chaining symbol table: Java implementation

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

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

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

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

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

**Proposition.** Under uniform hashing assumption, prob. 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.

![Image](image1.png)

**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: $M \sim N/4 \Rightarrow$ constant-time ops.

Deletion in a separate-chaining hash table

**Q.** How to delete a key (and its associated value)?
**A.** Easy: need only consider chain containing key.

![Image](image2.png)

Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\sqrt{N}$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$1*$</td>
<td>$1*$</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
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.

---

**3.4 Hash Tables**

- hash functions
- separate chaining
- linear probing
- context

---

**Linear-probing hash table demo**

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

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

---

**Linear-probing hash table**

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$st[i]$</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

$M = 16$

---

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

---

```
search K
hash(K) = 5
```

---

```
<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$st[i]$</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

$M = 16$

---

```
K
search miss
(return null)
```
Linear-probing hash table summary

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

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

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

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

---

![Linear-probing symbol table: Java implementation](image)

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

  private int hash(Key key) { /* as before */ }
  private void put(Key key, Value val) { /* next slide */ }

  public Value get(Key key) {
    for (int i = hash(key); keys[i] != null; i = (i+1) % M)
      if (key.equals(keys[i]))
        return vals[i];
    return null;
  }
}
```

---

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 # of probes in a linear probing hash table of size $M$ that contains $N = \alpha M$ keys is:

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

search hit

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

search miss / insert

Parameters.

- $M$ too large $\Rightarrow$ too many empty array entries.
- $M$ too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N/M \sim 1/2$. # probes for search hit is about $3/2$ # probes for search miss is about $5/2$

Resizing in a linear-probing hash table

Goal. Average length of list $N/M \leq 1/2$.

- Double size of array $M$ when $N/M \geq 1/2$.
- Halve size of array $M$ when $N/M \leq 1/2$.
- Need to rehash all keys when resizing.

Deletion in a linear-probing hash table

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

A. Requires some care: can’t just delete array entries.
### ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>delete</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>search hit</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>insert hit</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>delete hit</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( N )</td>
<td>( \log N )</td>
<td>compareTo()</td>
</tr>
<tr>
<td>red–black BST</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( \sqrt{N} )</td>
</tr>
<tr>
<td>separate chaining</td>
<td>( N )</td>
<td>( N )</td>
<td>( 1^* )</td>
<td>equals()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hashCode()</td>
</tr>
<tr>
<td>linear probing</td>
<td>( N )</td>
<td>( N )</td>
<td>( 1^* )</td>
<td>equals()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hashCode()</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

---

### 3-SUM (REVISITED)

#### 3-SUM
Given \( N \) distinct integers, find three such that \( a + b + c = 0 \).
- \( N^2 \) expected time case, \( N \) extra space.

#### 4-SUM
Given \( N \) distinct integers, find four such that \( a + b = c + d \).
- \( N^2 \log N \) time (worst case), \( N^2 \) extra space.
- \( N^2 \log N \) time (worst case), \( N \) extra space.
- \( N^2 \) expected time case, \( N^2 \) extra space.

---

### War story: algorithmic complexity attacks

**Q.** Is the uniform hashing assumption important in practice?
**A.** Obvious situations: aircraft control, nuclear reactor, pacemaker, HFT, ...

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

---

### 3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

---

### 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.
War story: algorithmic complexity attacks

A Java bug report.

Algorithmic complexity attack on Java

**Goal.** Find family of strings with the same hashCode().

**Solution.** The base-31 hash code is part of Java's String API.

### 2^32 strings of length 2N that hash to same value!

#### Separate chaining vs. linear probing

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

**Linear probing.**
- Less wasted space.
- Better cache performance.

---

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.
Hashing: variations on the theme

Many improved versions have been studied.

**Two-probe hashing.** [separate-chaining variant]
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to \( \sim \lg \ln N \).

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

**Cuckoo hashing.** [linear-probing variant]
- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.

Hash tables vs. balanced search trees

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

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

**Java system includes both.**
- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.

---

3.5 SYMBOL TABLE APPLICATIONS

- sets
- dictionary clients
- indexing clients
- sparse vectors

---

3.5 SYMBOL TABLE APPLICATIONS

- sets
- dictionary clients
- indexing clients
- sparse vectors
Set API

**Mathematical set.** A collection of distinct keys.

```java
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() number of keys in the set
    Iterator<Key> iterator() all keys in the set
}
```

Q. How to implement efficiently?

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.

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

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are in the list.

```java
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);
            }
        }
    }
}
```
Exception filter: Java implementation

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

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

        int 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);
    }
}
```

**Dictionary lookup**

**Command-line arguments.**
- A comma-separated value (CSV) file.
- Key field.
- Value field.

**Ex 1.** DNS lookup.

```
% more ip.csv
www.princeton.edu,128.122.128.15
www.cs.princeton.edu,128.112.128.15
www.harvard.edu,128.101.60.24
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,199.181.113.201
yahoo.com,66.94.234.11
msn.com,207.68.172.246
goog.le.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,203.93.0.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.16.60
163.com,220.181.29.254
passport.net,65.54.179.226
tom.com,65.135.128.237
nate.com,203.226.253.11
con.com,64.238.16.20
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
...
```

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
```

Codon name is value URL is value

```
% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
```

Not found

**Ex 2.** Amino acids.

```
% more amino.csv
TTT,Phenylalanine
TAC,Phenylalanine
TAA,Lysine
```

Codon name is value

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

Dictionary lookup

**Command-line arguments.**
- A comma-separated value (CSV) file.
- Key field.
- Value field.

**Ex 1.** DNS lookup.

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
```

Codon name is value URL is value

```
% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
```

Not found
Dictionary lookup

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

Ex 3. Class list.

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

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

```java
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]);

        String line = in.readLine();
        String[] tokens = line.split(" ");
        String key = tokens[keyField];
        String val = tokens[valField];
        st.put(key, val);
    }
}
```

File indexing

Goal. Index a PC (or the web).

3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
Goal. Given a list of text files, create an index so that you can efficiently find all files containing a given query string.

File indexing

import java.io.File;
public class FileIndex {
    public static void main(String[] args) {
        String st = new String();
        for (String filename : args) {
            File file = new File(filename);
            if (filename.contains(key)) {
                st.put(word, new SET<File>());
                for each word in file, add file to corresponding set
                process queries
            }
        }
    }
}

Book index

GOAL. Index for an e-book.

Index

<table>
<thead>
<tr>
<th>Book</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15</td>
</tr>
<tr>
<td>456</td>
<td>16</td>
</tr>
<tr>
<td>789</td>
<td>17</td>
</tr>
<tr>
<td>012</td>
<td>18</td>
</tr>
</tbody>
</table>

Searching applications: quiz 1

Which data type below would be the best choice to represent the file index?

A. SET<String, String>
B. SET<String, String>
C. SET<File, String>
D. SET<String, SET<File>>
E. I don't know.
Concordance

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

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

public class Concordance
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words = in.readAllStrings();
        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> set = st.get(s);
            set.add(i);
        }
        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            for (int k : set)
                // print words[k-4] to words[k+4]
        }
    }
}

Matrix-vector multiplication (standard implementation)

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

\[
\begin{bmatrix}
0 & .90 & 0 & 0 & .05 \\
0 & 0 & .36 & .36 & .18 \\
0 & 0 & 0 & .90 & .36 \\
.90 & 0 & 0 & 0 & 0.37 \\
.47 & 0.47 & 0 & 0 & .19 \\
\end{bmatrix}
\begin{bmatrix}
.05 \\
.04 \\
.36 \\
.37 \\
.19 \\
\end{bmatrix}
= 
\begin{bmatrix}
.036 \\
.297 \\
.333 \\
.045 \\
.1927 \\
\end{bmatrix}
\]

... 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;
}
Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.
Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.

\[ A \cdot x = b \]

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;
    }
}
```

Vector representations

1D array (standard) representation.
- Constant time access to elements.
- Space proportional to \( N \).

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

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

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

---

```java
import SparseVector;
import double[]
import.

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