Hashing

- hash functions
- collision resolution
- applications

Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case Search</th>
<th>Average Case Insert</th>
<th>Average Case Delete</th>
<th>Ordered Iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
<td>yes</td>
</tr>
<tr>
<td>randomized BST</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>?</td>
<td>yes</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

Collision resolution strategy. Algorithm and data structure to handle two keys that hash to the same index.

Equality test. Method for checking whether two keys are equal.

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).

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.
Implementing a good hash function

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

**Practical challenge:** need different approach for each type of key

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

**Ex:** date of birth.
- Bad: birth year.
- Better: birthday.

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

Hash Codes and Hash Functions

Java convention: all classes implement `hashCode()`

`hashCode()` returns a 32-bit int (between -2147483648 and 2147483647)

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

First try:

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

`((code & 0x7fffffff) % M)`

hex literal

String s = "call";
int code = s.hashCode();
int hash = code % M;
7121 8191
hex literal

3045982
Implementing hashCode() in Java

**Theoretical advantages of hashCode() convention**
- Ensures hashing can be used for every type of object
- Allows expert implementations suited to each type

**API for hashCode().**
- Return an int.
- If x.equals(y) then x and y must have the same hash code.
- Repeated calls to x.hashCode() must return the same value.

**Practical realities of hashCode() convention**
- Cost is an important consideration
- True randomness is hard to achieve

Default implementation. Memory address of x.

Customized implementations. String, URL, Integer, Date.

User-defined implementations. Tricky to get right, black art.

A typical type

**Assumption when using hashing in Java:**
Key type has reasonable implementation of hashCode() and equals()

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; }
}
```

Problem: Need a theorem for each type of data to ensure reliability.

A decent hash code design

Java 1.5 string library [see also Program 14.2 in Algs in Java].

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

- Equivalent to \( h = 31^{L-1} s_0 + \ldots + 31^2 s_{L-3} + 31 s_{L-2} + s_{L-1} \).
- Horner’s method to hash string of length \( L \): \( L \) multiplies/adds

Ex. String \( s = \text{“call”} \):

```
int code = s.hashCode();
```

\[ 3045982 = 99 \cdot 31 + 97 \cdot 31^2 + 108 \cdot 31^3 + 108 \cdot 31^4 \]

\[ = 108 + 31 \cdot (108 + 31 \cdot (99 + 31 \cdot (97))) \]

Provably random? Well, no.

A poor hash code design

Java 1.1 string library.
- For long strings: only examines 8-9 evenly spaced characters.
- 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;
}
```

but great potential for bad collision patterns.

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

Basic rule: need to use the whole key.
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

Approach
- Fix order \( k \) and dimension \( d \)
- Compute \( \text{hashcode}() \mod d \) for all \( k \)-grams in the document
- Result: \( d \)-dimensional vector profile of each document
- To compare documents:
  - Consider angle \( \theta \) separating vectors
    - \( \cos \theta \) close to 0: not similar
    - \( \cos \theta \) close to 1: similar

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

\[
\cos \theta = \frac{a \cdot b}{|a| \cdot |b|}
\]

Digression: using a hash function to profile a document for data mining

```java
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 other) {
        // compute dot product and divide by magnitudes
    }
}
```

Digression: using a hash function to compare documents

```java
public class CompareAll {
    public static void main(String args[]) {
        int k = Integer.parseInt(args[0]);
        int d = Integer.parseInt(args[1]);
        int N = StdIn.readInt();
        Document[] a = new Document[N];
        for (int i = 0; i < N; i++)
            a[i] = new Document(StdIn.readString(), k, d);
        System.out.print("      ");
        for (int j = 0; j < N; j++)
            System.out.printf("    %.4s", a[j].name());
        System.out.println();
        for (int i = 0; i < N; i++)
            System.out.printf("%.4s  ", a[i].name());
        System.out.println();
        for (int j = 0; j < N; j++)
            System.out.printf("%.4s", a[i].simTo(a[j]));
        System.out.println();
    }
}
```

### Digression: using a hash function for data mining

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

#### Approach
- Fix order \( k \) and dimension \( d \)
- Compute \( \text{hashcode}() \mod d \) for all \( k \)-grams in the document
- Result: \( d \)-dimensional vector profile of each document
- To compare documents:
  - Consider angle \( \theta \) separating vectors
    - \( \cos \theta \) close to 0: not similar
    - \( \cos \theta \) close to 1: similar

#### Digression: using a hash function for data mining

#### Table

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{tale.txt} )</th>
<th>( \text{genome.txt} )</th>
<th>( \text{tale.txt} )</th>
<th>( \text{genome.txt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>435</td>
<td>best of time 2</td>
<td>TCTTCGGTTG</td>
<td>2</td>
<td>TCTTCGGTTG</td>
</tr>
<tr>
<td>8999</td>
<td>it was the 8</td>
<td>TCTTCGGTTG</td>
<td>0</td>
<td>TCTTCGGTTG</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12122</td>
<td>CTTCGGTTG</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34543</td>
<td>t was the b 5</td>
<td>ATCGGTCGAG</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Digression: using a hash function for data mining

#### Code Examples

```java
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 other) {
        // compute dot product and divide by magnitudes
    }
}
```

#### Digression: using a hash function to compare documents

```java
public class CompareAll {
    public static void main(String args[]) {
        int k = Integer.parseInt(args[0]);
        int d = Integer.parseInt(args[1]);
        int N = StdIn.readInt();
        Document[] a = new Document[N];
        for (int i = 0; i < N; i++)
            a[i] = new Document(StdIn.readString(), k, d);
        System.out.print("      ");
        for (int j = 0; j < N; j++)
            System.out.printf("    %.4s", a[j].name());
        System.out.println();
        for (int i = 0; i < N; i++)
            System.out.printf("%.4s  ", a[i].name());
        System.out.println();
        for (int j = 0; j < N; j++)
            System.out.printf("%.4s", a[i].simTo(a[j]));
        System.out.println();
    }
}
```
Digression: using a hash function to compare documents

<table>
<thead>
<tr>
<th></th>
<th>US Constitution</th>
<th>&quot;Tom Sawyer&quot;</th>
<th>&quot;Huckleberry Finn&quot;</th>
<th>&quot;Pride and Prejudice&quot;</th>
<th>a photograph</th>
<th>financial data</th>
<th>Amazon.com website .html source</th>
<th>genome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cons</td>
<td>1.00</td>
<td>0.89</td>
<td>0.87</td>
<td>0.88</td>
<td>0.35</td>
<td>0.70</td>
<td>0.63</td>
<td>0.58</td>
</tr>
<tr>
<td>TomS</td>
<td>0.89</td>
<td>1.00</td>
<td>0.98</td>
<td>0.96</td>
<td>0.34</td>
<td>0.75</td>
<td>0.66</td>
<td>0.62</td>
</tr>
<tr>
<td>Huck</td>
<td>0.87</td>
<td>0.98</td>
<td>1.00</td>
<td>0.94</td>
<td>0.32</td>
<td>0.74</td>
<td>0.65</td>
<td>0.61</td>
</tr>
<tr>
<td>Prej</td>
<td>0.88</td>
<td>0.96</td>
<td>0.94</td>
<td>1.00</td>
<td>0.34</td>
<td>0.76</td>
<td>0.67</td>
<td>0.63</td>
</tr>
<tr>
<td>Pict</td>
<td>0.35</td>
<td>0.34</td>
<td>0.32</td>
<td>0.34</td>
<td>1.00</td>
<td>0.29</td>
<td>0.48</td>
<td>0.24</td>
</tr>
<tr>
<td>DJIA</td>
<td>0.70</td>
<td>0.75</td>
<td>0.74</td>
<td>0.76</td>
<td>0.29</td>
<td>1.00</td>
<td>0.62</td>
<td>0.58</td>
</tr>
<tr>
<td>Amaz</td>
<td>0.63</td>
<td>0.66</td>
<td>0.65</td>
<td>0.67</td>
<td>0.48</td>
<td>0.62</td>
<td>1.00</td>
<td>0.45</td>
</tr>
<tr>
<td>ACTG</td>
<td>0.58</td>
<td>0.62</td>
<td>0.61</td>
<td>0.63</td>
<td>0.24</td>
<td>0.58</td>
<td>0.45</td>
<td>1.00</td>
</tr>
</tbody>
</table>

% java CompareAll 5 1000 < docs.txt

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 \( \sqrt{\frac{M}{2}} \) tosses.

**Coupon collector.** Expect every bin has \( \approx 1 \) ball after \( O(M \ln M) \) tosses.

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

Collisions

**Collision.** Two distinct keys hashing to same index.

**Conclusion.** Birthday problem \( \Rightarrow \) can’t avoid collisions unless you have a ridiculous amount of memory.

**Challenge.** Deal with collisions efficiently.

**Approach 1:** accept multiple collisions

**Approach 2:** minimize collisions
Collision resolution: two approaches

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

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

Separate chaining ST implementation (skeleton)

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

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

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

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

Separate chaining ST implementation (put and get)

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

public Val get(Key key) {
    int i = hash(key);
    for (Node x = st[i]; x != null; x = x.next)
        if (key.equals(x.key))
            return vals[i];
    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 list.
- Average length = N / M.
- Worst case: all keys hash to same list.

Theorem. Let \( \alpha = N / M \geq 1 \) be average length of list. For any \( t > 1 \), probability that list length > \( t \alpha \) is exponentially small in \( t \).

Parameters.
- \( M \) too large \( \Rightarrow \) too many empty chains.
- \( M \) too small \( \Rightarrow \) chains too long.
- Typical choice: \( \alpha = N / M \approx 10 \Rightarrow \) constant-time ops.

Collision resolution approach 2: open addressing

Use an array of size \( M \gg N \).
- Hash: map key to integer \( i \) between 0 and \( M-1 \).
- Linear probing:
  - 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.

Linear probing ST implementation

```java
public class ArrayHashST<Key, Val> {
    private int M = 30001;
    private Val[] vals = new Val[maxN];
    private Key[] keys = new Key[maxN];

    private int hash(Key key) // as before
    public void put(Key key, Val 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 Val get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (key.equals(keys[i]))
                return vals[i];
    }
}
```

Cluster.

Cluster. A contiguous block of items.
Observation. New keys likely to hash into middle of big clusters.

Knuth’s parking problem. Cars arrive at one-way street with \( M \) parking spaces. Each desires a random space \( i \); if space \( i \) is taken, try \( i+1 \), \( i+2 \), …
What is mean displacement of a car?

Empty. With \( M/2 \) cars, mean displacement is about \( 3/2 \).
Full. With \( M \) cars, mean displacement is about \( \sqrt{\pi M / 2} \).
Analysis of linear probing

**Linear probing performance.**
- Insert and search cost depend on length of cluster.
- Average length of cluster = \( \alpha = \frac{N}{M} \).
- Worst case: all keys hash to same cluster.

**Theorem.** [Knuth 1962] Let \( \alpha = \frac{N}{M} < 1 \) be the load factor.

\[
\text{insert / search miss} = \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha^2} \right) \\
\text{search hit} = \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)
\]

**Parameters.**
- M too large \( \Rightarrow \) too many empty array entries.
- M too small \( \Rightarrow \) clusters coalesce.
- Typical choice: \( M = 2N \Rightarrow \) constant-time ops.

Hashing: variations on the theme

Many improved versions have been studied:

**Ex: Two-probe hashing**
- hash to two positions, put key in shorter of the two lists.
- reduces average length of the longest list to \( \log \log N \).

**Ex: Double hashing**
- 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.** Map key to integer \( i \) between 0 and \( M-1 \).

**Second hash.** Map key to nonzero skip value \( k \).

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

**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 \( \alpha = \frac{N}{M} < 1 \) be average length of list.

\[
\text{insert / search miss} = \frac{1}{1 - \alpha} \\
\text{search hit} = \frac{1}{\alpha} \ln(1 + \alpha)
\]

**Parameters.** Typical choice: \( M = 1.2N \Rightarrow \) constant-time ops.

**Disadvantage.** Delete 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.

<table>
<thead>
<tr>
<th>load factor α</th>
<th>50%</th>
<th>66%</th>
<th>75%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>get</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
</tr>
<tr>
<td>double hashing</td>
<td>get</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

number of probes

Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>iteration?</th>
<th>guarantee</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>average case</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
<th>hash functions</th>
<th>collision resolution</th>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>yes</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>randomized BST</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>yes</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hashing</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>no</td>
<td>equals()</td>
<td>hashcode()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* assumes random hash code

Hashing versus balanced trees

Hashing
- simpler to code
- no effective alternative for unordered keys
- faster for simple keys (a few arithmetic ops versus lg N compares)
- (Java) better system support for strings [cached hashcode]
- does your hash function produce random values for your key type??

Balanced trees
- stronger performance guarantee
- can support many more operations for ordered keys
- easier to implement Comparable 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
Set ADT

Set. **Unordered** collection of distinct keys.

**API for SET.**
- `add(key)`  insert the key into the set
- `contains(key)`  is the given key in the set?
- `remove(key)`  remove the key from the set
- `iterator()`  return iterator over all keys

Q. **How to implement?**
A0. Use hashing (unordered keys)
A1. Remove value from ST hashing code
A2. Use `java.util.HashSet`

**SET client example 1: dedup filter**

Remove duplicates from strings in standard input
- Read a key.
- If key is not in set, insert and print it.

```java
public class DeDup {
    public static void main(String[] args) {
        SET<String> set = new SET<String>();
        while (!StdIn.isEmpty()) {
            String key = StdIn.readString();
            if (!set.contains(key)) {
                set.add(key);
                System.out.println(key);
            }
        }
    }
}
```

Simplified version of `FrequencyCount` (no iterator needed)

**SET client example 2A: lookup filter**

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

```java
public class LookupFilter {
    public static void main(String[] args) {
        SET<String> set = new SET<String>();
        In in = new In[1];
        IN.isMyEmpty();
        set.add.in.readString());
        while (!StdIn.isEmpty()) {
            String word = StdIn.readString();
            if (set.contains(word)) {
                System.out.println(word);
            }
        }
    }
}
```

**SET client example 2B: exception filter**

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

```java
public class LookupFilter {
    public static void main(String[] args) {
        SET<String> set = new SET<String>();
        In in = new In[1];
        IN.isMyEmpty();
        set.add.in.readString());
        while (!StdIn.isEmpty()) {
            String word = StdIn.readString();
            if (!set.contains(word)) {
                System.out.println(word);
            }
        }
    }
}
```
SET filter applications

<table>
<thead>
<tr>
<th>key</th>
<th>in list</th>
<th>not in list</th>
</tr>
</thead>
<tbody>
<tr>
<td>dedup</td>
<td>eliminate duplicates</td>
<td>duplicates, unique keys</td>
</tr>
<tr>
<td>spell checker</td>
<td>find misspelled words</td>
<td>word, exception, dictionary, misspelled words</td>
</tr>
<tr>
<td>browser</td>
<td>mark visited pages</td>
<td>URL, lookup, visited pages</td>
</tr>
<tr>
<td>chess</td>
<td>detect draw</td>
<td>board, lookup, positions</td>
</tr>
<tr>
<td>spam filter</td>
<td>eliminate spam</td>
<td>IP addr, exception, spam, good mail</td>
</tr>
<tr>
<td>trusty filter</td>
<td>allow trusted mail</td>
<td>URL, lookup, good mail</td>
</tr>
<tr>
<td>credit cards</td>
<td>check for stolen cards</td>
<td>number, exception, stolen cards, good cards</td>
</tr>
</tbody>
</table>

Searching challenge:

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

Which searching method to use?

- 1) hashing implementation of SET
- 2) hashing implementation of ST
- 3) red-black-tree implementation of ST
- 4) red-black-tree implementation of SET
- 5) doesn’t matter much

Trick question: need both
ST (search key, SET of pointers to files)

Caveat: use B-tree or similar structure for truly huge indices

Index for search in a PC

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

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

build index

process lookup

request
Searching challenge:

Problem: Index for a book
Assumptions: book has 100,000+ words

Which searching method to use?
1) hashing implementation of SET
2) hashing implementation of ST
3) red-black-tree implementation of ST
4) red-black-tree implementation of SET
5) doesn’t matter much

Trick question: need both
ST (search key, SET of page numbers) with ordered iteration for both

Hashing in the wild: Java implementations
Java has built-in libraries for hash tables.
- java.util.HashMap = separate chaining implementation.
- java.util.IdentityHashMap = linear probing implementation.

import java.util.HashMap;
public class HashMapDemo
{
   public static void main(String[] args)
   {
      HashMap<String, String> st = new HashMap<String, String>();
      st.put("www.cs.princeton.edu", "128.112.128.15");
      System.out.println(st.get("www.cs.princeton.edu"));
   }
}

Duplicate policy.
- Java HashMap allows null values.
- Our implementation forbids null values.

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

Requires ordered iterators (not hashing)
Implementation of our API with java.util.HashMap.

```java
import java.util.HashMap;
import java.util.Iterator;

public class ST<Key, Val> implements Iterable<Key> {
    private HashMap<Key, Val> st = new HashMap<Key, Val>();

    public void put(Key key, Val val) {
        if (val == null) st.remove(key);
        else st.put(key, val);
    }

    public Val get(Key key) {
        return st.get(key);
    }

    public Val remove(Key key) {
        return st.remove(key);
    }

    public boolean contains(Key key) {
        return st.containsKey(key);
    }

    public int size() {
        return st.size();
    }

    public Iterator<Key> iterator() {
        return st.keySet().iterator();
    }
}
```

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.

**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.


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.

```java
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.

Too expensive for use in ST implementations (use balanced trees)