Hashing

- hash functions
- collision resolution
- applications

References:
Algorithms in Java, Chapter 14
http://www.cs.princeton.edu/introalgsds/42hash
## Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>red-black tree</td>
<td>3 lg N</td>
<td>3 lg N</td>
<td>3 lg N</td>
</tr>
</tbody>
</table>

**Can we do better?**
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.
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.

\[
\begin{align*}
\text{hash("it")} &= 3 \\
\text{hash("times")} &= 3
\end{align*}
\]

**Issues.**

1. Computing the hash function
2. **Collision resolution:** Algorithm and data structure to handle two keys that hash to the same index.
3. 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).
- hash functions
- collision resolution
- applications
Computing the 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.

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

thoroughly researched problem, still problematic in practical applications
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;
```

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.
Java’s `hashCode()` convention

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

**Requirements:**
- 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**
- True randomness is hard to achieve
- Cost is an important consideration

**Available implementations**
- default (inherited from Object): Memory address of `x` (!!!)
- customized Java implementations: `String`, `URL`, `Integer`, `Date`
- User-defined types: **users are on their own** that’s you!
A typical type

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

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

```java
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;
    }
}
```

Fundamental problem:
Need a theorem for each data type 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 \( h = 31^{L-1} \cdot s_0 + \ldots + 31^2 \cdot s_{L-3} + 31 \cdot s_{L-2} + s_{L-1}. \)
- 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 (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...

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

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
  \[ \cos \theta \text{ close to } 0: \text{not similar} \]
  \[ \cos \theta \text{ close to } 1: \text{similar} \]
**Digression: using a hash function for data mining**

Let's consider using a hash function for data mining.

### Example Hash Function

Given a hash function `hashcode()`, we can compute the hash code for substrings of a certain length (e.g., 10-grams) from a file or sequence.

#### Example 1: tale.txt

<table>
<thead>
<tr>
<th>i</th>
<th>10-grams with hashcode() i</th>
<th>freq</th>
<th>10-grams with hashcode() i</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>435</td>
<td>best of ti foolishnes</td>
<td>2</td>
<td>TTTCGGTTTG</td>
<td>2</td>
</tr>
<tr>
<td>8999</td>
<td>it was the</td>
<td>8</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>12122</td>
<td></td>
<td>0</td>
<td>CTTTCGTTTT</td>
<td>3</td>
</tr>
<tr>
<td>34543</td>
<td>t was the b</td>
<td>5</td>
<td>ATGC GGTCGA</td>
<td>4</td>
</tr>
</tbody>
</table>

### Example 2: genome.txt

<table>
<thead>
<tr>
<th>i</th>
<th>10-grams with hashcode() i</th>
<th>freq</th>
<th>10-grams with hashcode() i</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>34543</td>
<td>t was the b</td>
<td>5</td>
<td>ATGC GGTCGA</td>
<td>4</td>
</tr>
</tbody>
</table>

### Cosine Similarity

The cosine similarity `\( \cos \theta \)` is a measure of similarity between two vectors. A small cosine value indicates that the two vectors are not similar.

In our case, if `\( \cos \theta \)` is small, then the two files or sequences are not similar.
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("\n");
        for (int j = 0; j < N; j++)
            System.out.printf("%.4s", a[j].name());
        System.out.println();
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                System.out.printf("%8.2f", a[i].simTo(a[j]));
        System.out.println();
    }
}
```
## Digression: using a hash function to compare documents

<table>
<thead>
<tr>
<th></th>
<th>Cons</th>
<th>TomS</th>
<th>Huck</th>
<th>Prej</th>
<th>Pict</th>
<th>DJIA</th>
<th>Amaz</th>
<th>ACTG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cons</td>
<td>US Constitution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TomS</td>
<td>“Tom Sawyer”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Huck</td>
<td>“Huckleberry Finn”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prej</td>
<td>“Pride and Prejudice”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pict</td>
<td>a photograph</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DJIA</td>
<td>financial data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amaz</td>
<td>Amazon.com website .html source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTG</td>
<td>genome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

% java CompareAll 5 1000 < docs.txt

<table>
<thead>
<tr>
<th></th>
<th>Cons</th>
<th>TomS</th>
<th>Huck</th>
<th>Prej</th>
<th>Pict</th>
<th>DJIA</th>
<th>Amaz</th>
<th>ACTG</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>
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 $\sqrt{\pi M / 2}$ tosses.

Coupon collector.
Expect every bin has $\geq 1$ ball after $\Theta(M \ln M)$ 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.

**Conclusion.** Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount of memory.

**Challenge.** Deal with collisions efficiently.

**Approach 1:**
accept multiple collisions

25 items, 11 table positions
~2 items per table position

**Approach 2:**
minimize collisions

5 items, 11 table positions
~ .5 items per table position
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.

<table>
<thead>
<tr>
<th>st[0]</th>
<th>jocularly</th>
<th>seriously</th>
</tr>
</thead>
<tbody>
<tr>
<td>st[1]</td>
<td>listen</td>
<td></td>
</tr>
<tr>
<td>st[2]</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>st[3]</td>
<td>suburban</td>
<td>untravelled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[30001]</td>
<td>browsing</td>
<td></td>
</tr>
</tbody>
</table>

separate chaining (M = 8191, N = 15000) 
easy extension of linked list ST implementation

linear probing (M = 30001, N = 15000) 
easy extension of array ST implementation
Collision resolution approach 1: separate chaining

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.

---

![Diagram](attachment:image.png)

---

<table>
<thead>
<tr>
<th>key</th>
<th>hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>7121</td>
</tr>
<tr>
<td>me</td>
<td>3480</td>
</tr>
<tr>
<td>ishmael</td>
<td>5017</td>
</tr>
<tr>
<td>seriously</td>
<td>0</td>
</tr>
<tr>
<td>untravelled</td>
<td>3</td>
</tr>
<tr>
<td>suburban</td>
<td>3</td>
</tr>
</tbody>
</table>
public class ListHashST<Key, Value> {
    private int M = 8191;
    private Node[] st = new Node[M];

    private class Node {
        Object key;
        Object val;
        Node next;
        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
Separate chaining ST implementation (put and get)

```java
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, 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 list.
- Average length $= N / M$.
- **Worst case:** all keys hash to same list.

**Theorem.** Let $\alpha = N / M > 1$ 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.

depends on hash map being random map
Collision resolution approach 2: open addressing

Use an array of size $M >> N$.

- **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.

---

**Example**: Given two keys $I$ and $N$ with $M = 12$.

- **Insert $I$**
  - $hash(I) = 11$
  - Place $I$ at index 11.

- **Insert $N$**
  - $hash(N) = 8$
  - Place $N$ at index 8.
public class ArrayHashST<Key, Value> {
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[maxN];
    private Key[] keys = (Key[]) new Object[maxN];

    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;
    }
}
**Clustering**

**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.** Mean displacement for the last car is about $\sqrt{\pi \frac{M}{2}}$.
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.

Average probes for insert/search miss
\[
\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) = \frac{(1 + \alpha + 2\alpha^2 + 3\alpha^3 + 4\alpha^4 + \ldots)}{2}
\]

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 \( \Rightarrow \) too many empty array entries.
• Load factor too large \( \Rightarrow \) clusters coalesce.
• Typical choice: \( M \approx 2N \) \( \Rightarrow \) 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 $\alpha = N / M < 1$ be average length of list.

Average probes for insert/search miss:

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

Average probes for search hit:

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

**Parameters.** Typical choice: $\alpha \approx 1.2 \Rightarrow$ 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></th>
<th>load factor $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50%</td>
</tr>
<tr>
<td>linear probing</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>1.5</td>
</tr>
<tr>
<td>put</td>
<td>2.5</td>
</tr>
<tr>
<td>double hashing</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>1.4</td>
</tr>
<tr>
<td>put</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</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>
</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>
</tr>
<tr>
<td>hashing</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>1*</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 `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” ST API

```java
public class *ST<Key extends Comparable<Key>, Value>*
```

- `*ST()` - create a 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 key is not in table)
- `boolean contains(Key key)` - is there a value paired with key?
- `Key min()` - smallest key
- `Key max()` - largest key
- `Key next(Key key)` - next largest key (null if key is max)
- `Key prev(Key key)` - next smallest key (null if key is min)
- `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)

Ex: Can use LLRB trees implement priority queues for distinct keys
hash functions
collision resolution
applications
Set ADT

**Set.** Collection of distinct keys.

```java
public class SET<Key extends Comparable<Key>, Value>
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET()</td>
<td>create a set</td>
</tr>
<tr>
<td>void add(Key key)</td>
<td>put key into the set</td>
</tr>
<tr>
<td>boolean contains(Key key)</td>
<td>is there a value paired with key?</td>
</tr>
<tr>
<td>void remove(Key key)</td>
<td>remove key from the set</td>
</tr>
<tr>
<td>Iterator&lt;Key&gt; iterator()</td>
<td>iterator through all keys in the set</td>
</tr>
</tbody>
</table>

Normal mathematical assumption: **collection is unordered**

Typical (eventual) client expectation: **ordered iteration**

**Q.** How to implement?

**A0.** Hashing (our ST code [value removed] or java.util.HashSet)

**A1.** Red-black BST (our ST code [value removed] or java.util.TreeSet)
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);
                StdOut.println(key);
            }
        }
    }
}
```

Simplified version of `FrequencyCount` (no iterator needed)

% 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...

% java Dedup < 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...

No iterator needed. Output is in same order as input with dups removed.
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(args[0]);
        while (!in.isEmpty())         {
            set.add(in.readString());
        }
        while (!StdIn.isEmpty())         {
            String word = StdIn.readString();
            if (set.contains(word))            {
                StdOut.println(word);
            }
        }
    }
}
```
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 not in the list.

```java
public class LookupFilter {
    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);
        }
    }
}
```
## SET filter applications

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

```java
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+" beer); // assuming \s+ is the separator
    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);
    }
}
```

```java
SET<File> files = st.get(s);
for (File f: files) ...
```
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 SET
4) red-black-tree implementation of ST
5) doesn't matter much
public class Index
{
    public static void main(String[] args)
    {
        String[] words = StdIn.readAll().split("\s+" MyClass file).
        ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
        for (int i = 0; i < words.length; i++)
        {
            String s = words[i];
            if (!st.contains(s))
                st.put(s, new SET<Integer>());
            SET<Integer> pages = st.get(s);
            pages.add(page(i));
        }
        for (String s : st)
            StdOut.println(s + " : " + st.get(s));
    }
}

Requires ordered iterators (not hashing)
Java has built-in libraries for hash tables.

- `java.util.HashMap` = separate chaining implementation.
- `java.util.IdentityHashMap` = linear probing implementation.

```java
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", "128.112.136.11");
        st.put("www.princeton.edu", "128.112.128.15");
        StdOut.println(st.get("www.cs.princeton.edu"));
    }
}
```

Null value policy.

- Java `HashMap` allows null values.
- Our implementation forbids null values.
Using HashMap

Implementation of our API with java.util.HashMap.

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

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

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

    public Value get(Key key) { return st.get(key); }
    public Value remove(Key key) { return st.remove(key); }
    public boolean contains(Key key) { return st.contains(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.

malicious adversary learns your ad hoc hash function (e.g., by reading Java API) and causes a big pile-up in single address 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.

Reference: http://www.cs.rice.edu/~scrosby/hash
Algorithmic complexity attack on the Java Library

**Goal.** Find strings with the same hash code.

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

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aa</td>
<td>2112</td>
</tr>
<tr>
<td>BB</td>
<td>2112</td>
</tr>
</tbody>
</table>

2\(^N\) strings of length 2\(^N\) that hash to same value!

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.  

```
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)