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. Very large index table, few collisions
- No time limitation: trivial collision resolution with sequential search. Small table, lots of collisions, must search within the cell.
- Space and time limitations: hashing (the real world).

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
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()`.

**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: 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;
    }
}
```

**Implementing hash code: strings**

Java library implementation

```java
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;
    }
}
```

**Horner’s method to hash string of length \(L\):** \(L\) multiplies/adds.
- Equivalent to \( h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0 \).

**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 \cdot 31^1 + (108 \cdot 31^0 + (97 + 31 \cdot (99)))`
  (Horner’s method)
Implementing hash code: strings

Performance optimization.
• Cache the hash value in an instance variable.
• Return cached value.

```java
public final class String {
    private int hash = 0;
    private final char[] s;
    ...
    public int hashCode() {
        int h = hash;
        if (h != 0) return h;
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;
        return h;
    }
}
```

Implementing hash code: user-defined types

```java
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double amount;
    public Transaction(String who, Date when, double amount) {  /* as before */  }
    ...
    public boolean equals(Object y) {  /* as before */  }
    public int hashCode() {
        int hash = 17;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

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 \(-2^{31}\) 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

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

typically a small prime

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

1-in-a-billion bug

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

correct

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

bug

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

1-in-a-billion bug
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 $\sim \pi M / 2$ 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 $\Theta(\log M / \log \log M)$ balls.

HASHING

› Hash functions
› Separate chaining
› Linear probing

Collisions

Collision. Two distinct keys hashing to same index.

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

Challenge. Deal with collisions efficiently.
Separate chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]
- Hash: map key to integer $i$ between 0 and $M - 1$.
- Insert: put at front of $i^{th}$ chain (if not already there).
- Search: need to search only $i^{th}$ chain.

Separate chaining ST: 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;
        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;
    }
}
```

Separate chaining ST: 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;
        private Object val;
        private Node next;
        ...
    }

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

    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, probability 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.

**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: $M \sim N / S \Rightarrow$ constant-time ops.
### ST Implementations: Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst-Case Cost (after N inserts)</th>
<th>Average Case (after N random inserts)</th>
<th>Ordered Iteration?</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</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>(ordered array)</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>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>N *</td>
<td>N *</td>
<td>N *</td>
<td>3-5 *</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.

```
linear probing hash table

<table>
<thead>
<tr>
<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[0]</td>
<td>null</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[1]</td>
<td>null</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[2]</td>
<td>null</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[3]</td>
<td>null</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[30000]</td>
<td>null</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

linear probing (M = 30001, N = 15000)

"jocularly"
Linear probing hash table

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.

Insert $S$
hash$(S) = 6$

$M = 16$

Linear probing hash table

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.

Insert $S$
hash$(S) = 6$

$M = 16$
**Linear probing hash table**

**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.

```plaintext
insert E
hash(E) = 10
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: S
M = 16
```

```
linear probing hash table
```

```plaintext
insert E
hash(E) = 10
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: S E
M = 16
```
**Linear probing hash table**

**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.

```
insert A
hash(A) = 4
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]    S E
M = 16
```

**Linear probing hash table**

**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.

```
insert A
hash(A) = 4
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]    S E A
M = 16
```
**Linear probing hash table**

**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.

```
insert R
hash(R) = 14
```

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>
M = 16

**Linear probing hash table**

**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.

```
insert R
hash(R) = 14
```

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>
M = 16
**Linear probing hash table**

**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.

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

```
insert C
hash(C) = 5
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] A  S  E  R
```

```
insert C
hash(C) = 5
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] A  S  C  E  R
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] A  C  S  E  R
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] C
M = 16
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] A  C  S  E  R
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] A  C  S  E  R
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[] C
M = 16
```
**Linear probing hash table**

**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.

Insert $H$

hash($H$) = 4

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$M = 16$

**Linear probing hash table**

**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.

Insert $H$

hash($H$) = 4

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td></td>
<td>E</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$M = 16$

**Linear probing hash table**

**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.

Insert $H$

hash($H$) = 4

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td></td>
<td>E</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$M = 16$

**Linear probing hash table**

**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.

Insert $H$

hash($H$) = 4

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td></td>
<td>E</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$M = 16$
**Linear probing hash table**

**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.

```
insert H
hash(H) = 4
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
\[\text{st[]}\]
A C S E R H
```

\( M = 16 \)

**Linear probing hash table**

**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.

```
insert X
hash(X) = 15
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
\[\text{st[]}\]
A C S H E R
```

\( M = 16 \)
Linear probing hash table

**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.

```
insert X
hash(X) = 15
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: A C S H E R X
M = 16
```

Linear probing hash table

```
insert X
hash(X) = 15
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: A C S H E R X
M = 16
```

Linear probing hash table

**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
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: A C S H E R X
M = 16
```

Linear probing hash table

```
insert M
hash(M) = 1
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]: A C S H E R X
M = 16
```
**Linear probing hash table**

**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.

```
insert M
hash(M) = 1
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[] M A C S H E R X
```

$M = 16$

**Linear probing hash table**

**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.

```
insert P
hash(P) = 14
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[] M A C S H E R X
```

$M = 16$
**Linear probing hash table**

**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.

---

```
insert P
hash(P) = 14
```

---

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
M = 16

c0[] M A C S H E R X
```

---

```
M = 16
```

---

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
M = 16

c0[] M A C S H E R X
```

---

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
M = 16

c0[] P M A C S H E R X
```

---

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
M = 16

c0[] P M A C S H E R X
```
Linear probing hash table

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.

insert L
hash(L) = 6

M = 16

<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear probing hash table

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.

insert L
hash(L) = 6

M = 16

<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear probing hash table

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.

insert L
hash(L) = 6

M = 16

<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear probing hash table

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.

insert L
hash(L) = 6

M = 16

<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Linear probing hash table**

**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.

```
insert L
hash(L) = 6
```

```
<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$M = 16$

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

```
search E
hash(E) = 10
```

```
<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$M = 16$
**Linear probing hash table**

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.

search E

hash(E) = 10

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[] P M A C S H L E R X

M = 16

search hit
(return corresponding value)

**Linear probing hash table**

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.

search L

hash(L) = 6

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[] P M A C S H L E R X

M = 16

L
**Linear probing hash table**

**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.

```plaintext
search L  
hash(L) = 6
```

```plaintext
st[]
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  P M A C S H L E R X
M = 16
```

**Linear probing hash table**

**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.

```plaintext
search L  
hash(L) = 6
```

```plaintext
st[]
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  P M A C S H L E R X
M = 16
```

**Linear probing hash table**

**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.

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

```plaintext
st[]
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  P M A C S H L E R X
M = 16
```
Linear probing hash table

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.

search $K$
hash(K) = 5

<table>
<thead>
<tr>
<th>st[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>L</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = 16</td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear probing hash table

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.

search $K$
hash(K) = 5

<table>
<thead>
<tr>
<th>st[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>L</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = 16</td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear probing hash table

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.

search $K$
hash($K$) = 5

<table>
<thead>
<tr>
<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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$M = 16$

Cluster.
A contiguous block of items.

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

Linear probing - 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 $N$.

Clustering

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

Linear probing ST implementation

```java
public 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 */ }

    public void put(Key key, Value val) {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }

    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 and halving code omitted
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 \( \sim \frac{3}{2} \).
Full. With \( M \) cars, mean displacement is \( \sim \frac{\sqrt{\pi M}}{8} \).

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of
probes in a linear probing hash table of size \( M \) that contains \( N = \alpha M \) keys is:

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

Pf.

Parameters.
- \( M \) too large \( \Rightarrow \) too many empty array entries.
- \( M \) too small \( \Rightarrow \) search time blows up.
- Typical choice: \( \alpha = N / M \sim \frac{1}{2} \).

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after ( N ) inserts)</th>
<th>average case (after ( N ) random inserts)</th>
<th>ordered iteration?</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</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.38 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>1.00 lg N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>N *</td>
<td>N *</td>
<td>3.5 *</td>
<td>3.5 *</td>
</tr>
<tr>
<td>linear probing</td>
<td>N *</td>
<td>N *</td>
<td>3.5 *</td>
<td>3.5 *</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

War story: String hashing in Java

String \( \text{hashCode()} \) in Java 1.1.
- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: 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 += s[i] + (37 * hash);
    return hash;
}
```

- Downside: great potential for bad collision patterns.

<table>
<thead>
<tr>
<th>resource</th>
<th>link</th>
</tr>
</thead>
</table>
War story: 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.

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 code is part of Java’s string API.

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, ….

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.

Q. How to delete?
Q. How to resize?
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 $\log \log N$.

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.
• 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 strings (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.
• Hash tables: java.util.HashMap, java.util.IdentityHashMap.

TODAY

› Hashing
› Search applications

Search 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()` - return the number of keys in the set
- `Iterator<Key> iterator()` - iterator through keys in the set

**Q. How to implement?**

**A.** Remove "value" from any ST implementation

---

**Exception filter**

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

```bash
% more list.txt
was it the of
% java WhiteList list.txt < tinyTale.txt
it was the of it was the of it was the of it was the of it was the of it was the of
% java BlackList list.txt < tinyTale.txt
best times worst times age wisdom age foolishness epoch belief epoch incredulity season light season darkness spring hope winter despair
```

---

**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, not 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 {in, not in} the list.

```java
public class BlackList {
    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);
            }
    }
}
```

Dictionary lookup

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

Ex 1. DNS lookup.

```bash
% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.11
www.harvard.edu,140.247.50.127
www.cs.yale.edu,128.103.60.24
www.math.princeton.edu,128.36.236.74
www.cs.harvard.edu,128.36.229.30

% java LookupCSV ip.csv 0 1
adobe.com 192.150.18.60
www.princeton.edu 128.112.128.15
ebay.edu Not found

% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu 999.999.999.99
Not found
```

Ex 2. Amino acids.

```bash
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...
```

Dictionary lookup

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

```
% 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 3. Class list.

```
% java LookupCSV classlist.csv 4 1
eberl
Ethan
nwebb
Natalie
% java LookupCSV classlist.csv 4 3
dpan
P01
```

Dictionary lookup: Java implementation

```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]);
        ST<String, String> st = new ST<String, String>();
        while (!in.isEmpty()) {
            String line = in.readLine();
            String[] tokens = database[i].split(”,“);
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (!st.contains(s)) StdOut.println("Not found");
            else StdOut.println(st.get(s));
        }
    }
}
```

File indexing

Goal. Index a PC (or the web).
Goal. Given a list of files specified, create an index so that you can efficiently find all files containing a given query string.

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

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

```java
public class Concordance {
    public static void main(String[] args) {
        In in = new In(args[0]);
        String[] words = StdIn.readAll().split("\s+");
        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(i);
        }
        while (!StdIn.isEmpty()) {
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            for (int k : set)
                // print words[k-5] to words[k+5]
        }
    }
}
```

Vectors and matrices

**Vector.** Ordered sequence of N real numbers.

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

**Vector operations**

\[
\begin{align*}
a &= \begin{bmatrix} 0 & 3 & 15 \end{bmatrix}, & b &= \begin{bmatrix} -1 & 2 & 2 \end{bmatrix} \\
a + b &= \begin{bmatrix} -1 & 5 & 17 \end{bmatrix} \\
a \cdot b &= (0\cdot -1) + (3\cdot 2) + (15\cdot 2) = 36 \\
|a| &= \sqrt{0^2 + 3^2 + 15^2} = 3\sqrt{26}
\end{align*}
\]

**Matrix-vector multiplication**

\[
\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}
\]
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.

\[
\begin{bmatrix}
0 & 0 & 0.36 & 0.36 & 0.18 \\
0.90 & 0 & 0 & 0 & 0 \\
0 & 0.36 & 0.36 & 0.18 \\
0 & 0 & 0.90 & 0 & 0 \\
0.90 & 0 & 0 & 0 & 0 \\
0.47 & 0.47 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Matrix-vector multiplication (standard implementation)

\[
\begin{array}{ccc}
\mathbf{a}[i] & \mathbf{x}[j] & \mathbf{b}[i] \\
0.90 & 0.05 & 0.036 \\
0.36 & 0.18 & 0.297 \\
0.90 & 0.36 & 0.333 \\
0.90 & 0.37 & 0.045 \\
47 & 0.47 & 0.1927 \\
\end{array}
\]

...  
\[
\text{...}
\]
\[
\text{double[][] a = new double[N][N];}
\]
\[
\text{double[] x = new double[N];}
\]
\[
\text{double[] b = new double[N];}
\]
\[
\text{...}
\]
\[
\text{// initialize a[][] and x[]}
\]
\[
\text{...}
\]
\[
\text{for (int i = 0; i < N; i++)}
\]
\[
\quad \text{sum = 0.0;}
\]
\[
\quad \text{for (int j = 0; j < N; j++)}
\]
\[
\quad \quad \text{sum += a[i][j]*x[j];}
\]
\[
\quad b[i] = sum;
\]

Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.

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

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.
Sparse vector data type

```java
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;  }
}
```

Sparse matrix-vector multiplication

```java
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);
```

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).

Sample searching challenge

Problem. Rank pages on the web.
Assumptions.
• Matrix-vector multiply
• 10 billion+ rows
• Sparse

Which “searching” method to use to access array values?
1. Standard 2D array representation
2. Symbol table
3. Doesn’t matter much.
Sample searching challenge

Problem. Rank pages on the web.
Assumptions.
• Matrix-vector multiply
• 10 billion+ rows
• sparse

Which “searching” method to use to access array values?
1. Standard 2D array representation
2. Symbol table
3. Doesn’t matter much.

Sparse vector data type

```java
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;
    }
}
```

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;
    }
    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;
    }
}
```
Compressed row storage (CRS)

- Store nonzeros in a 1D array val[].
- Store column index of each nonzero in parallel 1D array col[].
- Store first index of each row in array row[].

\[
A = \begin{bmatrix}
11 & 0 & 0 & 41 \\
0 & 22 & 0 & 0 \\
0 & 0 & 33 & 43 \\
14 & 0 & 34 & 44 \\
0 & 25 & 0 & 0 \\
16 & 26 & 36 & 46 \\
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>col[]</th>
<th>val[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>43</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>46</td>
</tr>
</tbody>
</table>

Compressed row storage (CRS)

Benefits.
- Cache-friendly.
- Space proportional to number of nonzeros.
- Very efficient matrix-vector multiply.

Downside. No easy way to add/remove nonzeros.

Applications. Sparse Matlab.