Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
**ST implementations: summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average-case cost (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>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td></td>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
</tbody>
</table>

**Q.** Can we do better?  
**A.** Yes, but with different access to the data (if we don’t need ordered ops).

---

**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.

- 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.

**Issues.**

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

**Classic space-time tradeoff.**

- Thoroughly researched problem, still problematic in practical applications.
- Need different approach for each key type.

---

**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;
    }
}
```

**Performance optimization.**

- Cache the hash value in an instance variable.
- Return cached value.

```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;
    }
}
```

Implementing hash code: strings

**Java library implementation**

- 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"`;

\[
\text{int code} = \text{s.hashCode();} \quad \Rightarrow \quad 3045982 = 99 \cdot 31^1 + 97 \cdot 31^2 + 108 \cdot 31^3 + 108 \cdot 31^4 + 108 \cdot 31^5 + 97 \cdot 31^6 + 97 \cdot 31^7 + 99 \cdot 31^8
\]

(Horner’s method)

Implementing hash code: strings

**Performance optimization.**

- Cache the hash value in an instance variable.
- Return cached value.
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 $-2^a$ and $2^a - 1$.
Hash function. An int between 0 and $M - 1$ (for use as array index).

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

Bug

```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

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into $M$ bins.

Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

Coupon collector. Expect every bin has $\geq 1$ ball after $\sim M \ln M$ tosses.

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

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.

![Binomial distribution graph](image)

**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: \(M \sim N / 5\) \(\Rightarrow\) constant-time ops.

ST implementations: summary

| Implementation                  | worst-case cost (after N inserts) | average case (after N random inserts) | ordered iteration? | key interface
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>(\lg N)</td>
<td>(\lg 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>N/2</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>(\lg N)</td>
<td>(\lg N)</td>
<td>3-5</td>
<td>3-5</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
HASHING

- Hash functions
- Separate chaining
- Linear probing

Collision resolution: open addressing

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

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. |

| 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. |

| 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
```

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

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [S]
```

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 E
hash(E) = 10
```

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

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [S]
```

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.

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

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

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

<table>
<thead>
<tr>
<th>st[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>E</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 A
hash(A) = 4
```

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
st[]: S S S E E 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 R
hash(R) = 14
```

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

$\text{insert } R$
$\text{hash}(R) = 14$

$\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & A & S & E & & & & & & & & & & & & R \\
\end{array}$

$M = 16$

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.

$\text{insert } R$
$\text{hash}(R) = 14$

$\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & A & S & E & R & & & & & & & & & & & \\
\end{array}$

$M = 16$

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.

$\text{linear probing hash table}$

$\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & A & S & E & & & & & & & & & & & & R \\
\end{array}$

$M = 16$

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.

$\text{insert } C$
$\text{hash}(C) = 5$

$\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & A & S & E & R & & & & & & & & & & & \\
\end{array}$

$M = 16$
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 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 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

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

\[
\begin{align*}
\text{insert } H \\
\text{hash}(H) &= 4 \\
\end{align*}
\]

<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>
<tr>
<td>H</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>
</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.

\[
\begin{align*}
\text{insert } H \\
\text{hash}(H) &= 4 \\
\end{align*}
\]

<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>
<tr>
<td>H</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>
</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></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></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 X
hash(X) = 15
```

<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>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></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 X
hash(X) = 15
```

```
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
```

```
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]
```

```
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
```

<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>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td>P</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 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>E</td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</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 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></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</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 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></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.

```
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></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</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 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></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$M = 16$
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.

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

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

linear probing hash 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.

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$M = 16$

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$M = 16$

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$L$

$M = 16$

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$M = 16$

---

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

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$M = 16$

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$L$

$M = 16$

---

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

---

**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>P</td>
<td>M</td>
<td></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>
</tr>
</tbody>
</table>
```

$M = 16$

---

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

st[]

<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></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>R</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.

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

st[]

<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></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>R</td>
</tr>
</tbody>
</table>

M = 16
```
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
</tr>
</tbody>
</table>

\( M = 16 \)

search miss
(return null)
### 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 $N$.

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

### Linear probing ST implementation

```java
public class LinearProbingHashST<Key, Value> {
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;
}
```

### Clustering

**Cluster.** A contiguous block of items.

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

### Knuth’s parking problem

**Model.** Cars arrive at one-way street with $M$ parking spaces. Each desires a random space $i$: if space $i$ is taken, try $i + 1$, $i + 2$, etc.

**Q.** What is mean displacement of a car?

- **Half-full.** With $M/2$ cars, mean displacement is $\sim 3/2$.
- **Full.** With $M$ cars, mean displacement is $\sim \sqrt{\pi M}/8$
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:

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

For search hit and search miss / insert.

Pf. (Proof)

Parameters.

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

# Probes for search hit is about $3/2$  
# Probes for search miss is about $5/2$

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>$N$</td>
<td>$N/2$</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>$N$</td>
<td>$N/2$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>delete</td>
<td>$N$</td>
<td>$N/2$</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>search hit</td>
<td>$1.38 \log N$</td>
<td>$1.38 \log N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>search miss / insert</td>
<td>$3.5^*$</td>
<td>$3.5^*$</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>linear probing</td>
<td>$3.5^*$</td>
<td>$3.5^*$</td>
<td>no</td>
<td>equals()</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

War story: String hashing in Java

String `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.

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 hash code is part of Java's string API.

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aa&quot;</td>
<td>2112</td>
</tr>
<tr>
<td>&quot;BB&quot;</td>
<td>2112</td>
</tr>
</tbody>
</table>

2^{N} strings of length 2N that hash to same value!

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AaAaAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBBB&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BBAaAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAaAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAaBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBAaBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBBAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BBBBBBBB&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

### Diversion: one-way hash functions

**One-way hash function.** "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

**Ex.** MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ... known to be insecure

```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.

**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`.

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.

```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);
        }
    }
}
```

```
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
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 pages</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 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);
        }
    }
}
```

```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);
        }
    }
}
```
SEARCH APPLICATIONS

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors

Dictionary lookup

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

Ex 1. DNS lookup.

```
% more ip.csv
www.princeton.edu,128.112.128.15
www.harvard.edu,128.103.45.24
www.yahoo.edu,135.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
yaho.com,64.94.236.13
msn.com,207.68.172.244
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.cn,202.108.33.32
ebay.com,64.135.182.87
adobe.com,192.150.18.60
passport.net,65.54.179.226
tom.com,61.135.186.237
rauten.cn.cn,202.72.51.22
...
```

```
% 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
```

```
URL is key
IP is value
```

Ex 2. Amino acids.

```
% 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
...`

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

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

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

```
URL is key
first name is value
login is key
precept is value
```

```
login is key
value is name
name is value
codon is key
```
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));
        }
    }
}
```

Process input file

Build symbol table

Process lookups with standard I/O

Search applications

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors

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.

File indexing

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt
freedom
moby.txt

% ls *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java
import
FileIndex.java ST.java ST.java

Comparator
null
```
File indexing

**Goal.** Given a list of files specified, create an index so that you can efficiently find all files containing a given query string.

\[
\text{% is *.txt} \\
aesop.txt magna.txt moby.txt sawyer.txt tale.txt \\
\text{% java FileIndex *.txt} \\
\text{freedom} \\
\text{magna.txt moby.txt tale.txt} \\
\text{whale} \\
\text{moby.txt sawyer.txt aesop.txt}
\]

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

**Book index**

**Goal.** Index for an e-book.

**Concordance**

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

Concordance

Search Applications

‣ Sets
‣ Dictionary clients
‣ Indexing clients
‣ Sparse vectors

Vectors and matrices

Vector. Ordered sequence of N real numbers.

vector operations

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

Matrix-vector multiplication

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

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.

\[ \begin{bmatrix}
    0 & 0 & .36 & .36 & .18 \\
    0 & 0 & .36 & .36 & .18 \\
    0 & 0 & .90 & 0 & 0 \\
    .90 & 0 & 0 & 0 & 0 \\
    .47 & .47 & 0 & 0 & 0
\end{bmatrix} \]
Matrix-vector multiplication (standard implementation)

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

```
<table>
<thead>
<tr>
<th></th>
<th>0.90</th>
<th>0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.36</td>
<td>0.04</td>
</tr>
<tr>
<td>0</td>
<td>0.90</td>
<td>0.36</td>
</tr>
<tr>
<td>0</td>
<td>0.47</td>
<td>0.19</td>
</tr>
<tr>
<td>0.47</td>
<td>0.47</td>
<td>0.05</td>
</tr>
</tbody>
</table>
```

double loops (N^2 running time)

Sparse matrix-vector multiplication

**Problem.** Sparse matrix-vector multiplication.

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

```
A * x = b
```

Sparse vector data type

```
public class SparseVector
{
    private HashST<Integer, Double> v;

    public SparseVector()
    {  v = new HashST<Integer, Double>();  }

    public void put(int i, double x)
    {  v.put(i, x);  }

    public double get(int i)
    {
        if (!v.contains(i)) return 0.0;
        else return v.get(i);
    }

    public Iterable<Integer> indices()
    {  return v.keys();  }

    public double dot(double[] that)
    {  
        double sum = 0.0;
        for (int i : indices())
            sum += that[i]*this.get(i);
        return sum;
    }
}
```

Sparse vector data type

```
<table>
<thead>
<tr>
<th></th>
<th>0.36</th>
<th>0.36</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.05</td>
<td>0.18</td>
</tr>
<tr>
<td>0</td>
<td>0.18</td>
<td>0.05</td>
</tr>
<tr>
<td>0</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>0</td>
<td>0.14</td>
<td>0.01</td>
</tr>
<tr>
<td>0</td>
<td>0.36</td>
<td>0.12</td>
</tr>
<tr>
<td>0</td>
<td>0.47</td>
<td>0.07</td>
</tr>
</tbody>
</table>
```

Symbol table representation.

- **Key** = index, **value** = entry.
- **Efficient iterator.**
- **Space proportional to number of nonzeros.**
Matrix representations

**2D array (standard) matrix representation:** Each row of matrix is an array.
- Constant time access to elements.
- Space proportional to $N^2$.

**Sparse matrix representation:** Each row of matrix is a sparse vector.
- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).

Sparse matrix-vector multiplication

```
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);
```

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;
    }

    // ... all 0s vector
}
```

Sparse vector data type (cont)

```java
public class SparseVector {
    // ...

    public double dot(SparseVector that) {
        double sum = 0.0;
        for (int i : this.st)
            if (that.st.contains(i))
                sum += this.get(i) * that.get(i);
        return sum;
    }

    public double norm() {
        return Math.sqrt(this.dot(this));
    }

    public SparseVector plus(SparseVector that) {
        SparseVector c = new SparseVector(N);
        for (int i : this.st)
            c.put(i, this.get(i));
        for (int i : that.st)
            c.put(i, that.get(i) + c.get(i));
        return c;
    }

    // ... dot product
    // ... 2-norm
    // ... vector sum
}
```

Sparse matrix data type

```java
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;
    }

    // ... matrix-vector multiplication
}
```

Compressed row storage (CRS)

Compressed row storage.
- 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{pmatrix}
  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{pmatrix}
```

```
    \begin{array}{ccc}
        \text{i} & \text{col} & \text{val} \\
        0 & 1 & 11 \\
        1 & 4 & 41 \\
        2 & 2 & 22 \\
        3 & 3 & 33 \\
        4 & 4 & 43 \\
        5 & 1 & 14 \\
        6 & 3 & 34 \\
        7 & 4 & 44 \\
        8 & 2 & 25 \\
        9 & 1 & 16 \\
        10 & 2 & 26 \\
        11 & 3 & 36 \\
        12 & 4 & 46 \\
    \end{array}
```
Compressed row storage (CRS)

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

```java
double[] y = new double[N];
for (int i = 0; i < n; i++)
    for (int j = row[i]; j < row[i+1]; j++)
        y[i] += val[j] * x[col[j]];
```

**Downside.** No easy way to add/remove nonzeros.

**Applications.** Sparse Matlab.