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

- Hashing
- Search Applications
TODAY

- Hashing
- Search applications
HASHING

- Hash functions
- Separate chaining
- Linear probing
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 (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</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 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).
Motivation: Counting Characters

- Assume that you are coding a program to count the frequency of characters between a-z
- The algorithm is very easy as below
- Create an array for the frequencies, a character can be transformed to array index by: c - ‘a’.

```java
int size = 'z'-'a'+1;
int[] counts = new int[size];

String text = "Lorem ipsum...";

for (int i = 0; i < text.length(); i++) {
    if(text.charAt(i)>='a' && text.charAt(i)<='z') {
        counts[text.charAt(i)-'a']++;
    }
}

for (int i = 0; i < counts.length; i++) {
    System.out.println((char)(i+'a') + " " + counts[i]);
}
```
This example is easy as we have a table that maps each character to an index naturally.

Can we extend this idea, as a general solution for Symbol Tables?

First step:
- Extend this idea to a subset of integers between 0 and M.
- Simple, just create an array of size M

Second step:
- Can we generalise for integers between -Infinity and +Infinity
- Not so feasible! Create an array of size Infinity.
- Probably in the data we will only observe a small subset of the integers
- So, the first problem with this approach is the Domain Size (number of valid inputs)
Mapping a Larger Domain to Smaller

**IDEA:** Find a mapping from our real values to a smaller number of array indices.

Create a function \( h(x) \) that maps key values between 0 and \( 10^{100} \) to values between 0 and 10 so that we can store them in an array of size 10.

One example; \( h(x) = x \mod 10 \)

If we are lucky the keys will map uniformly to these numbers, so if we have 10 numbers the array will store one value per each cell. But, if we have 11 keys, than a cell will certainly have multiple keys mapped to it. This is called as collision!
We will generalise our solution such that:
- We can map any large domain to a feasible array
- Able to map any object/data type to index, so a hash function working for any data type.
- Accept the possibility of collisions and find a strategy to resolve them.

There is a concept called as Minimal Perfect Hashing, which maps key values domain to array indices one-to-one. Our example for characters is a good example for this. But this will not be possible for most data types (we will have to settle for one-to-many mapping, i.e. collisions!).
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).
HASHING

- Hash functions
- Separate chaining
- Linear probing
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;
    }
}
```

closest to IEEE 64-bit representation; xor most significant 32-bits with least significant 32-bits
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 + 31 \cdot (108 + 31 \cdot (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 * hash);
        hash = h;
        return h;
    }
}
```
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).

```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*

**typical a prime or power of 2**

hashCode() of "polygenelubricants" is $-2^{31}$
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 \( \Theta \left( \log M / \log \log M \right) \) balls.
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.

Hash value frequencies for words in Tale of Two Cities ($M = 97$)

Java's String data uniformly distribute the keys of Tale of Two Cities
HASHING

- Hash functions
- Separate chaining
- Linear probing
Collisions

Collision. Two distinct keys hashing to same index.
- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing ⇒ 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.

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

![Diagram of separate chaining symbol table](image-url)
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

array doubling and halving code omitted

no generic array creation
(declare key and value of type Object)
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]);
    }
}
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: $M \sim N / 5 \Rightarrow$ 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 (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</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
Hash functions
Separate chaining
Linear probing

linear probing (M = 30001, N = 15000)
## 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>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></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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = 16

insert S
hash(S) = 6
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
```
## 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>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>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
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.

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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</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.

### Example

**insert E**

$\text{hash}(E) = 10$

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

$$
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & & & & & & & S & & & & & & & & \\
\text{M = 16} & & & & & & & & & & & & & & & \\
\end{array}
$$

insert E
hash(E) = 10

E
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 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.

linear probing hash table

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

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

st[]

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

<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>S</td>
<td></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 A
hash(A) = 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></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.

<table>
<thead>
<tr>
<th>Linear probing hash table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<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>A</td>
<td>S</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.

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

`insert R`

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

M = 16

insert R
hash(R) = 14
**Hash.** Map key to integer $i$ between 0 and $M - 1$.

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

**Linear probing hash table**

<table>
<thead>
<tr>
<th>M = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>st[]</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.

<table>
<thead>
<tr>
<th></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><strong>st[]</strong></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = 16

insert C
hash(C) = 5
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

<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>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = 16

C
**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
```

<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.

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

$M = 16$

H
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></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>st[]</td>
<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>H</td>
</tr>
<tr>
<td>M = 16</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></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.

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

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

M = 16

H
**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>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>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></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.

![Linear probing hash table](image)

$M = 16$
**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
```

```
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.

---

```
ninsert 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>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>X</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>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>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
<td>X</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>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>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</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.

```plaintext
Insert M
hash(M) = 1
```

<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>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.

\[
\text{insert } M \\
\text{hash}(M) = 1
\]
**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{verbatim}
insert M
hash(M) = 1
\end{verbatim}

\begin{center}
\begin{tabular}{llllllllllll}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
  \hline
|   |   | M |   |   | A | C | S | H |   | E |   |   | R |   | X |
\end{tabular}
\end{center}

$M = 16$
**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></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>st[]</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</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.

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

<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>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>R</td>
<td>X</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 P
hash(P) = 14

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

$M = 16$

P
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></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st[]</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>

$M = 16$

P

P
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>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>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</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.

<table>
<thead>
<tr>
<th>i</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</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></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</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
```

```
# Hash table with linear probing

<table>
<thead>
<tr>
<th>index</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></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>

$M = 16$
```

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.

---

**insert L**

hash(L) = 6

---

<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>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

---

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.

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

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

M = 16

| 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 |   |   |   |   |   |   |   |   |   |    |    |   |    |
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.

<table>
<thead>
<tr>
<th>st[]</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P</td>
<td></td>
<td>M</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>1</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</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.

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

M = 16

<table>
<thead>
<tr>
<th>st[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>X</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 E
hash(E) = 10

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

$M = 16$

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.

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>1</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>R</td>
<td>X</td>
</tr>
</tbody>
</table>

$M = 16$
Hashing. 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
```

```
<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>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>
<td></td>
</tr>
<tr>
<td>M = 16</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></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 L
hash(L) = 6
```

<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>P</td>
<td>M</td>
<td></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>R</td>
<td>X</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 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></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>
</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>
</tbody>
</table>
| M = 16
```

```plaintext
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
```

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

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.

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

$M = 16$
**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.

**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>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></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

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

$M = 16$

$K$
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
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>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></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>

\( M = 16 \)

K
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
```

```
M = 16
```

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

\[
\begin{array}{ccccccccccccccc}
\text{st[]} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{P} & \text{M} & \text{A} & \text{C} & \text{S} & \text{H} & \text{L} & \text{E} & \text{R} & \text{X} \\
\end{array}
\]

\( M = 16 \)

\( K \)

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>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>P</td>
<td>M</td>
<td></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>R</td>
<td>X</td>
</tr>
</tbody>
</table>

$M = 16$
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;
    }
}
**Cluster.** A contiguous block of items.

**Observation.** New keys likely to hash into middle of big clusters.
Knuth's parking problem

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

Q. What is mean displacement of a car?

Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

Full. With $M$ cars, mean displacement is $\sim \sqrt{\pi M / 8}$
Analysis of linear probing

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

$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$$  \hspace{1cm} \text{search hit}

$$\sim \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$  \hspace{1cm} \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: $\alpha = N / M \sim \frac{1}{2}$.  \hspace{1cm} # probes for search hit is about 3/2
  \hspace{1cm} # 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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</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>
<tr>
<td>linear probing</td>
<td>N *</td>
<td>N *</td>
<td>N *</td>
<td>3-5 *</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.

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

<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>

2^N strings of length 2N that hash to same value!
Diversion: one-way hash functions

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

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

String password = args[0];
MessageDigest sha1 =
    MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);
/* prints bytes as hex string */

Applications. Digital fingerprint, message digest, storing passwords.
Caveat. Too expensive for use in ST implementations.
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`. 
Hashing
Search applications
Search Applications

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors
Mathematical set. A collection of distinct keys.

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
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);
        }
    }
}
```
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.

```java
% 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
```
Dictionary lookup

Command-line arguments.

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

Ex 2. Amino acids.

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

% 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
TGA,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
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,Hisidine
CAC,His,H,Hisidine
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.

Ex 3. Class list.

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

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

```bash
% more classlist.csv
13, Berl, Ethan Michael, P01, eberl
11, Bourque, Alexander Joseph, P01, abourque
12, Cao, Phillips Minghua, P01, pcao
11, Chehoud, Christel, P01, cchehoud
10, Douglas, Malia Morioka, P01, malia
12, Haddock, Sara Lynn, P01, shaddock
12, Hantman, Nicole Samantha, P01, nhantman
11, Hesterberg, Adam Classen, P01, ahesterb
13, Hwang, Roland Lee, P01, rhwang
13, Hyde, Gregory Thomas, P01, ghyde
13, Kim, Hyunmoon, P01, hktwo
11, Kleinfeld, Ivan Maximillian, P01, ikleinfe
12, Korac, Damjan, P01, dkorac
11, MacDonald, Graham David, P01, gmacdona
10, Michal, Brian Thomas, P01, bmichal
12, Nam, Seung Hyeon, P01, seunghnam
11, Nastasescu, Maria Monica, P01, mnastase
11, Pan, Di, P01, dpan
12, Partridge, Brenton Alan, P01, bpartrid
13, Rilee, Alexander, P01, arilee
13, Roopakalu, Ajay, P01, aroopaka
11, Sheng, Ben C, P01, bsheng
12, Webb, Natalie Sue, P01, nwebb
...```
```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));
        }
    }
}
```
SEARCH APPLICATIONS

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors
**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.

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

% java FileIndex *.txt
freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt

% ls *.java

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

import
FileIndex.java SET.java ST.java

Comparator
null
```
**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.
public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>();
        for (String filename : args) {
            File file = new File(filename);
            In in = new In(file);
            while !(in.isEmpty())
            {
                String word = in.readString();
                if (!st.contains(word))
                    st.put(s, new SET<File>());
                SET<File> set = st.get(key);
                set.add(file);
            }
        }
        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            StdOut.println(st.get(query));
        }
    }
}
Index

Abstract data type (ADT), 127-135
abstract classes, 163
classes, 124-136
collections of items, 137-139
creating, 147-149
defined, 128
duplicate items, 173-176
equivalence relations, 159-162
FIFO queue, 163-164
first-class, 177-178
generic operations, 174
index items, 177
intermediate operations, 138-139
module programming, 135
polynomial, 148-152
priority queues, 373-376
pushdown stacks, 130-131
stubs, 235
symbol table, 347-356
ADT interfaces
array (array), 274
complex number (complex), 181
collection table (cTable), 653
tree-based table (rTable), 397
unsorted priority queue (UPQ1), 103
item (item), 275, 498
key (key), 498
polynomial (poly), 189
point (point), 134
priority queue (pq), 375
queue of int (intQueue), 166
stack of int (intStack), 140
symbol table (symTable), 301
test index (ti), 225
unordered (un), 139
abstract in-place merging, 331-333
abstract operation, 19
Access control state, 131
Actual data, 31
Adapter class, 158-159
Adapter set, 268
Address, 84-93
Adjacency list, 120-123
depth-first search, 252-256
Adjacency matrix, 170-172
Algorithms, 4-66, 27-64
abstract operations, 15, 31, 34-35
analysis of, 6
average worst case performance, 33, 60-62
big-Oh notation, 44-47
binary search, 52-59
computational complexity, 62-64
efficiency, 6, 30, 32
empirical analysis, 30-32, 58
exponential time, 219
implementation, 25-30
logarithmic time, 40-42
mathematical analysis, 33-36
primary parameter, 36
probabilistic, 331
recurrence, 44-57, 57
recursive, 198
running time, 3-46
search, 53-64, 498
steps in, 27-29
See also Randomized algorithm
Amortization approach, 557, 627
Arithmetic operations, 177-179, 180, 191
Array, 13, 85
binary search, 57
dynamic allocation, 57
and linked lists, 92, 94-95
string, 349-350
unsorted, 112-114
references, 86-87, 89
sorting, 265-267, 273-275
strings, 119
two-dimensional, 117-118, 120-121
trees, 87
two vectors, 297
see also index, array
Array representation
binary tree, 38
LLD queue, 154-169
linked lists, 110
polynomial ADT, 131-132
priority queues, 377-378, 403, 405
pushdown stack, 135-136
random queue, 170
symbol table, 305, 311-312, 521
Asymptotic expression, 45-46
Average deviation, 80-81
Average-case performance, 35, 60-62
AVL tree, 583
B tree, 584, 622-624
external/internal pages, 695
4-5-6-7-8 trees, 693-704
linked stacks, 701
完全平衡, 120
searching, 597-701
selections, 701
Balanced trees, 233, 355-598
Binary tree, 584
bottom up, 576, 581, 555
height-balanced, 583
induced sequential access, 590-692
performance, 573-576, 581-582, 595-598
randomized, 599-664
red-black, 577-585
skiplists, 587-594
splay, 566-571
727
**Concordance**

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

```
% java Concordance tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
of his most gracious *majesty* king george the third

princeton
no matches
```
public class Concordance
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words = StdIn.readAll().split("\s+"l);
        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);
            set.put(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]
            }
        }
    }
}
Search Applications

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors
Vectors and matrices

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

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

vector operations

\[ a = \begin{bmatrix} 0 & 3 & 15 \end{bmatrix}, \quad b = \begin{bmatrix} -1 & 2 & 2 \end{bmatrix} \]

\[ a + b = \begin{bmatrix} -1 & 5 & 17 \end{bmatrix} \]

\[ a \odot b = (0 \cdot -1) + (3 \cdot 2) + (15 \cdot 2) = 36 \]

\[ |a| = \sqrt{a \odot a} = \sqrt{0^2 + 3^2 + 15^2} = 3\sqrt{26} \]

matrix-vector multiplication

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

\[
\begin{bmatrix}
0 & 0 & .36 & .36 & .18 \\
0 & 0 & 0 & 0 & 0 \\
0 & .90 & 0 & 0 & 0 \\
0 & 0 & .36 & .36 & .18 \\
0 & 0 & 0 & .90 & 0 \\
.90 & 0 & 0 & 0 & 0 \\
.47 & 0 & .47 & 0 & 0
\end{bmatrix}
\]
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;
}
Problem.  Sparse matrix-vector multiplication.

Assumptions.  Matrix dimension is 10,000; average nonzeros per row $\sim 10$. 

$$A \times x = b$$
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.
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;
    }
}
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.
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.

✓ cannot be done without fast algorithm
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;
    }

    //...
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;
}

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;
    }
}
Compressed row storage (CRS)

Compressed row storage.
• Store nonzeros in a 1D array \( \text{val[]} \).
• Store column index of each nonzero in parallel 1D array \( \text{col[]} \).
• Store first index of each row in array \( \text{row[]} \).

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

<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>

<table>
<thead>
<tr>
<th>i</th>
<th>row[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>
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.