CSCI 241
Scott Wehrwein
Hash Tables: Open Addressing
Goals

Know how to use open addressing with linear or quadratic probing for collision resolution.
Load Factor: Performance Implications

Load factor $\lambda = \frac{\# \text{ entries in table}}{\text{size of the array}}$

If $\lambda$ is large, runtime is slow. If $\lambda$ is small, memory is wasted.

If the memory's sitting there wasted... why not use it?
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```latex
\begin{tabular}{|c|}
\hline
0 \\
1 \\
2 \\
3 \\
4 \\
|\end{tabular}
```

```python
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

```python
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

```plaintext
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**put(key):**

```
h = hash(key);
while A[h] is full:
    h = (h+1) % N
A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<table>
<thead>
<tr>
<th>0</th>
<th>(10, bear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**put(key):**

\[ h = \text{hash}(key); \]

**while** A[h] is full:

\[ h = (h+1) \mod N \]

A[h] = value
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>

**put(key):**

```
h = hash(key);
while A[h] is full:
    h = (h+1) % N
A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<table>
<thead>
<tr>
<th>h</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td>(24, ape)</td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>

**put(key):**

```
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```
Open Addressing with Linear Probing

Problem:

Clustered hash values will result in a lot of searching.

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

```latex
\begin{tabular}{|c|c|}
\hline
0 & (10, bear) \\
1 & (1, dog) \\
2 & (11, auk) \\
3 & (24, ape) \\
4 & (14, cat) \\
\hline
\end{tabular}
```

```
\text{put}(\text{key}): \\
    h = \text{hash}(\text{key}); \\
\text{while } A[h] \text{ is full:} \\
    h = (h+1) \mod N \\
A[h] = \text{value}
```
Open Addressing with Quadratic Probing

**Quadratic Probing**: Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, ...
Quadratic probing looks at H, H+1, H+4, H+9, H+16, ...

| put(key) | H = hash(key);
|---------|-----------------
| i = 0; |
| while A[h] is full: |
| i++; |
| A[h] = value |

<table>
<thead>
<tr>
<th>put(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(1, &quot;dog&quot;);</td>
</tr>
<tr>
<td>put(11, &quot;auk&quot;);</td>
</tr>
<tr>
<td>put(10, &quot;bear&quot;);</td>
</tr>
<tr>
<td>put(14, &quot;cat&quot;);</td>
</tr>
<tr>
<td>put(24, &quot;ape&quot;);</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Open Addressing: Runtime

• May be faster, but may not be. Depends on keys.

• There’s no free lunch: worst-case is always $O(n)$.

• In practice, average-case is $O(1)$ if you make good design decisions and insertions are not done by someone who wants to ruin your day.