Last Time

- Hash Tables
  - Transform a key into an array index through use of a hash function
  - Constant time data access if designed properly
  - Need to resolve collisions if two different keys are inserted into the same array index
  - Covered Separate Chaining – each array index is a linked list
Time-Space Tradeoff

• If there were no memory limitation, we could do any search with only one member access by simply using the key as an index in a huge array.

• If there were no time limitation, we could do any search with sequential search, minimizing the amount of space to the size of all keys.

• Our goal is to strike a balance between these two extremes – don’t waste memory on empty chains, but also don’t waste time on long chains.
Linear Probing

• On insertion, when we have a collision, just move to the next open array index.

• On search, if we don’t find what we’re looking for, try looking at the next open slot.
### Linear Probing

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

- **entries in red** are new
- **entries in gray** are untouched
- **keys in black** are probes
- **probe sequence wraps to 0**

```
vals[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```
void put(Key key, Value val) {
    int i;
    for (i = hash(key); keys[i] != null; i = (i+1)%M) {
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        } //if
    } //for
    keys[i] = key;
    vals[i] = val;
} //put
Linear Probing

Value get(Key key) {
    for (int i = hash(key); keys[i] != null; 
        i = (i+1)%M) {
        if (keys[i].equals(key)) {
            return vals[i];
        } //if
    } //for
    return null;
} //get
Linear Probing – Analysis

• As with Separate Chaining, the performance depends on the $N/M$ ratio.
• Linear probing has an extra issue – when the array fills, an insert will cause an infinite loop looking for an open index.
• For good performance, we use array resizing.
  – We want to guarantee that the load factor is between $1/8$ and $1/2$. 
void resize(int cap) {
    LinearProbingST<Key, Value> t;
    t = new LinearProbingST<Key, Value>(cap);
    for (int i = 0; i < M; i++) {
        if (keys[i] != null) {
            t.put(keys[i], vals[i]);
        } //if
    } //for
    keys = t.keys;
    vals = t.vals;
    M = t.M;
} //put

void put(Key key, Value val) {
    if (N >= M/2) {
        resize(2*M);
    } //if
    ......

Linear Probing – Evaluation

The graph shows the number of equality tests (y-axis) against the number of operations (x-axis). The term "cumulative average" is indicated on the graph, suggesting that the data represents an average over a series of operations. The trend line indicates a decrease in the number of equality tests as the number of operations increases, approaching a cumulative average of 3.2.
Linear Probing – Issues

• Moving to the next array index will cause clumping, rather than uniform distribution.

• Solution: Instead of incrementing by 1, let’s move by a different prime value.
  – Find the smallest prime number less than $M$, and hash modulo that amount + 1.
    • $h(x) = x \mod 11$
    • $h2(x) = x \mod 7 + 1$
Double Hashing

• As N increases, double hashing reduces the number of probes vs. linear probing.
• As N approaches M, both linear probing and double-hashing start to degrade in performance.
void delete(Key key) {
    int i = hash(key);
    while (!key.equals(keys[i])) {
        if (keys[i] == null) { return; } //if
        i = (i + 1) % M;
    } //while
    keys[i] = null; vals[i] = null;
    i = (i + 1) % M;
    while (keys[i] != null) {
        Key keyToRedo = keys[i];
        Value valToRedo = vals[i];
        keys[i] = null; vals[i] = null;
        N--;
        put(keyToRedo, valToRedo);
        i = (i + 1) % M;
    } //while
    N--
    if (N > 0 && N <= M/8) {
        resize(M/2);
    } //if
} //delete
Deletion

```java
void delete(Key key) {
    int i = hash(key);
    while (!key.equals(keys[i])) {
        if (keys[i] == null) { return; } //if
        i = (i + h2(key)) % M;
    } //while
    keys[i] = null; vals[i] = null;
    i = (i + h2(key)) % M;
    while (keys[i] != null) {
        Key keyToRedo = keys[i];
        Value valToRedo = vals[i];
        keys[i] = null; vals[i] = null;
        N--;
        put(keyToRedo, valToRedo);
        i = (i + h2(key)) % M;
    } //while
    N--
    if (N > 0 && N <= M/8) {
        resize(M/2);
    } //if
} //delete
```
Linear Probing – More Analysis

• In a linear probing hash table of size $M$, with $\alpha = \frac{N}{M}$, the average number of probes required for a search hit is $\frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right)$, and for a search miss or insert is $\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2}\right)$.

  – For a table of $\alpha = \frac{1}{2}$, these values are $\frac{3}{2}$ and $\frac{5}{2}$.
  
  – If $\alpha = \frac{3}{4}$, these values are $\frac{5}{2}$ and $\frac{17}{2}$.
  
  – If $\alpha = \frac{7}{8}$, these values are $\frac{9}{2}$ and $\frac{65}{2}$.
Memory Usage

• Separate Chaining
  – About $48N + 32M$

• Linear Probing
  – Between $32N$ and $128N$

• Binary Search Trees
  – About $56N$
Any Questions?