CSc 120
Introduction to Computer Programming II

15: Hashing
Hashing
Searching

We have seen two search algorithms:

- linear (sequential) search \( O(n) \)
  - the items are not sorted
- binary search \( O(\log n) \)
  - the items are sorted
  - must consider the cost of sorting

• Can we do better?

• Have you considered how a Python dictionary might be implemented?
ADT - Dictionary

• A dictionary is an ADT that holds key/value pairs and provides the following operations:
  - put(key, value)
    o makes an entry for a key/value pair
    o assumes key is not already in the dictionary
  - get(key) looks up key in the dictionary
    o returns the value associated with key (and None if not found)
ADT - Dictionary

Usage:

```python
>>> d = Dictionary(7)
>>> d.put('five', 5)
>>> d.put('three', 3)
```

Problem:

Implement Dictionary

Hint:

```python
>>> d._pairs
[['five', 5], ['three', 3], None, None, None, None, None, None]
```
ADT – Dictionary solution 1

class Dictionary:
    def __init__(self, capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity
        self._nextempty = 0

    def put(self, k, v):
        self._pairs[self._nextempty] = [k, v]
        self._nextempty += 1

    def get(self, k):
        for pair in self._pairs[0:self._nextempty]:
            if pair[0] == k:
                return pair[1]
        return None
Performance

• What is big-O of the Dictionary's methods?
  - put()
  - get()

• Can we do better than O(n) for get()?  
• Consider this:
  
alist[3]  # this "get" or "lookup" is O(1)

• Why is this O(1)?
  indices are contiguous
  easy to compute starting point plus offset

• Can we 'transform' keys into integers that fall into a small, contiguous range?
Can we 'transform' keys into integers that fall into a small range?

"hello" -> 147
"a"      -> 422

How could we turn a key (string) into an integer?
   - simple method: use the length

“Hash” the key (colloquial meaning)
   Chop up the key
   Scramble the key to get a value
Hashing

• A hash function is a function that can be used to map data of arbitrary size to a value in a fixed range

• Is the following a hash function?

```python
def hash(key):
    return len(key)
```

• Strings are arbitrary length
  – modify `hash(key)` to return a value in a fixed range
  – an integer between 0 and 7
Exercise

Problem:
Modify Dictionary to use a hash function to compute the index for a new key/value pair.

(See solution on slide 28.)
Hashing

What happens in this situation?

```python
>>> d.put('hello', 14)
>>> d.put('e', 351)
>>> d.put('hat', 8)
>>> d.put('conciusness', 1)
```
Hashing

• Hash results:

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'hello'</td>
<td>5</td>
</tr>
<tr>
<td>'e'</td>
<td>1</td>
</tr>
<tr>
<td>'hat'</td>
<td>3</td>
</tr>
<tr>
<td>'consciousness'</td>
<td>5</td>
</tr>
</tbody>
</table>

• *Collision*: two or more keys have the same hash value
Hashing

• Hash results:

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'hello'</td>
<td>5</td>
</tr>
<tr>
<td>'e'</td>
<td>1</td>
</tr>
<tr>
<td>'hat'</td>
<td>3</td>
</tr>
<tr>
<td>'consciousness'</td>
<td>5</td>
</tr>
</tbody>
</table>

• Dictionary implementation view:

Need a place to put ['consciousness', 1]
Hashing and collisions

• *perfect hash function*: every key hashes to a unique value
  – most hash functions are not perfect

• Need a systematic method for placing keys in a Dictionary (hash table) when collisions occur.

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>['e', 351]</td>
<td></td>
<td>['hat', 8]</td>
<td></td>
<td>['hello', 14]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Need a place to put ['consciousness', 1]
Collision Resolution

• Methods for resolving collisions:
  – increase the table size (the list in our example)
    consider social security numbers: 333-55-8888
    9 digits / $10^9$ entries

  – open addressing
    o compute the hash value
    o on collision, sequentially visit each slot in the hash table to find
      an available spot
    o visit each slot by going 'lower' in the table (decrement by 1)
    o wrap if necessary
Collision Resolution

• Simplify the example by using integers for keys
• Hash function
  \[ h(key) = key \mod 7 \]
• Hash values for the keys: 14, 2, 10, 19

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
</tr>
</tbody>
</table>

• Hash table

<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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td></td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collision Resolution

• keys: 14, 2, 10, 19

• Now add 24
  – $h(\text{key}) = \text{key} \mod 7$
    
    = 24 \mod 7
    
    = 3 \leftarrow \text{collision, use open addressing}

• 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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(24) = 3 \leftarrow \text{collision}$
Collision Resolution

- keys: 14, 2, 10, 19
- Now add 24
  - \( h(\text{key}) = \text{key} \mod 7 \)
    \[ = 24 \mod 7 = 3 \]
    collision, use open addressing
- 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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( h(24) = 3 \) — collision
look lower — occupied
Collision Resolution

• keys: 14, 2, 10, 19
• Now add 24
  – \( h(\text{key}) = \text{key} \mod 7 \)
    = 24 \mod 7
    = 3 \leftarrow \text{collision, use open addressing}

• Hash table

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<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></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

h(24) = 3 \leftarrow \text{collision}

look lower \leftarrow \text{occupied}

look lower \leftarrow \text{empty}
Collision Resolution

• *Probe sequence*: the locations examined when inserting a new key
  
  \[ h(24) = 3 \]

• The hash computation is the first "probe"

• Hash table

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collision Resolution

• *Probe sequence*: the locations examined when inserting a new key
  
  \[ h(24) = 3 \]

• The hash computation is the first "probe"

• 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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

first probe – collision 3
Collision Resolution

• *Probe sequence*: the locations examined when inserting a new key
  \[ h(24) = 3 \]
• The hash computation is the first "probe"
• Hash table

<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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  first probe – collision 3

  second probe – occupied 2
Collision Resolution

• *Probe sequence*: the locations examined when inserting a new key
  
  \[ h(24) = 3 \]

• The hash computation is the first "probe"

• Hash table

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

  - first probe – collision 3
  - second probe – occupied 2
  - third probe – empty 1
Collision Resolution

• *Probe sequence*: the locations examined when inserting a new key
  
  \[ h(24) = 3 \]

• The hash computation is the first "probe"

• Hash table

  probe sequence: 3, 2, 1

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td>24</td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

  first probe – collision 3
  second probe – occupied 2
  third probe – empty 1
Exercise

Use open addressing to insert the key 23 into the hash table below. Give the probe sequence.

*The hash function is the key % 7*

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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>24</td>
<td>2</td>
<td>10</td>
<td></td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
Collision Resolution

open addressing:

- the probe sequence is linear
- the probe decrement is 1

*open addressing with linear probing* has serious performance problems (!!)

When two keys collide at the same hash value, they will follow the same initial probe sequence

Can we do better?

*Hint: change the probe decrement.*
Hashing

• SHA-1 (Secure Hash Algorithm 1)
  • cryptographic hash function designed by the NSA
  • 120 bits
  • shown as hexadecimal number, 40 digits long

• MD5 (Message Digest 5)
  – widely used hash function to verify data integrity
  – now compromised
  – 128 bits
ADT – Dictionary solution w/ hashing

class Dictionary:
    def __init__(self, capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity

    def _hash(self, k):
        return len(k) % len(self._pairs)

    def put(self, k, v):
        self._pairs[self._hash(k)] = [k, v]  # use the hash function

    def get(self, k):
        return self._pairs[self._hash(k)][1]  # use the hash function
Questions

What is a hash function?

What is a collision?

In open addressing with linear probing, how are collisions resolved?
Collision Resolution (revisited)

open addressing

- *open addressing with linear probing*
  - compute the hash value
  - on collision, sequentially visit each slot in the hash table to find an available spot
  - visit each slot by going 'lower' in the table (decrement by 1)
  - wrap if necessary

terminology

- the probe sequence is linear
- the probe decrement is 1
Collision Resolution (revisited)

• keys: 14, 2, 10, 19

• Now add 24
  – $h(key) = key \mod 7$
  – $24 \mod 7 = 3 \leftarrow$ collision, use open addressing

• Hash table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>14</td>
<td></td>
<td>2</td>
<td>10</td>
<td>19</td>
</tr>
</tbody>
</table>

$h(24) = 3 \leftarrow$ collision

look lower – occupied

look lower – empty
Exercise

Modify the put() method of the ATD below to implement open addressing with linear probing.

class Dictionary:
    def __init__(self, capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity

    def _hash(self, k):
        return len(k) % len(self._pairs)

    def put(self, k, v):
        self._pairs[self._hash(k)] = [k, v]  # use the hash function

.....
Clusters

• *Cluster*: a sequence of adjacent, occupied entries in a hash table

• problems with open addressing with linear probing
  – colliding keys are inserted into empty locations below the collision location
  – on each collision, a key is added at the edge of a cluster
  – the edge of the cluster keeps growing
  – the edges begin to meet with other clusters
  – these combine to make *primary clusters*
Collision Resolution

open addressing
  – idea: need a probe decrement that is different for keys that hash to the same value

simple example
  – the use mod for the hash
  – use quotient for the probe
    o note: cannot use 0

  – probe decrement function p(key)
    the quotient of key after division by 7 (if the quotient is 0, then 1)
    or
    max(1, key / 7)

called open addressing with double hashing
Collision Resolution – double hashing

• functions

\[ h(key) = key \% 7 \]

\[ p(key) = \max(1, key / 7) \]

• values for the keys: 10, 2, 19, 14, 24, 23

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Collision Resolution – double hashing

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

hash table after inserting keys: 10, 2, 19, 14
Collision Resolution – double hashing

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Now insert key 24:

```
0 1 2 3 4 5 6
14 2 10 19
```
Collision Resolution — double hashing

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Now insert key 24:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the decrement?  
What is the probe sequence?

h(24) = 3 collision
Collision Resolution – double hashing

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Now insert key 24:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td></td>
<td>2</td>
<td>10</td>
<td>24</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

h(24) = 3 collision

What is the decrement? 3
What is the probe sequence? 3, 0, 4
Exercise

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Use double hashing to insert key 23:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>10</td>
<td>24</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collision Resolution

open addressing with double hashing
- compute the hash value
- on collision, use the probe decrement function to determine what slot to visit next
- wrap if necessary

improvement over linear probing
- when two keys collide, they usually follow different probe sequences when a search is made for an empty location
  - hash(10) = 3  hash(24) = 3
  - probe(10) = 1  probe(24) = 3
- prevents primary clustering
Hash functions and collisions

• Consider an *ideal hash* function $h(k)$
  – it maps keys to hash values (slots) uniformly and randomly

• Suppose $T$ is a hash table having $M$ table entries from 0 to $M-1$

• An ideal hash function would imply that any slot from 0 to $M-1$ is equally likely

• All slots equally likely, implies collisions would be infrequent.

• Is that true?
collision phenomenon

• von Mises Birthday Paradox
  – if there are 23 or more people in a room, there is a > 50% chance that two or more will have the same birthday
collision phenomenon

Ball tossing model

Given

- a table T with 365 slots
  (each is a different day of the year)
- toss 23 balls at random into these 365 slots

then

- there is a > 50% chance we will toss 2 or more
  balls into the same slot

What?

- 23 balls in the table
- the table is only 6.3% full
  $\frac{23}{365} = .063$
- and we have a 50% chance of a collision!
collision phenomenon

Ball tossing model

\[ P(n) = \text{probability that tossing } n \text{ balls into 365 slots has at least one collision} \]

\[ P(n) = 1 - \frac{365!}{365^n(365-n)!} \]
collision phenomenon

\[
P(n) = \text{probability that tossing } n \text{ balls into 365 slots has at least one collision}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>P(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.027</td>
</tr>
<tr>
<td>10</td>
<td>0.117</td>
</tr>
<tr>
<td>20</td>
<td>0.411</td>
</tr>
<tr>
<td>23</td>
<td>0.572</td>
</tr>
<tr>
<td>30</td>
<td>0.706</td>
</tr>
<tr>
<td>40</td>
<td>0.891</td>
</tr>
<tr>
<td>50</td>
<td>0.970</td>
</tr>
<tr>
<td>60</td>
<td>0.994</td>
</tr>
<tr>
<td>70</td>
<td>0.99915958</td>
</tr>
<tr>
<td>80</td>
<td>0.99991433</td>
</tr>
<tr>
<td>100</td>
<td>0.99999969</td>
</tr>
</tbody>
</table>

at 23, greater than 50% chance
collision phenomenon

$P(n) =$ probability that tossing $n$ balls into 365 slots has at least one collision

<table>
<thead>
<tr>
<th>$n$</th>
<th>$P(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.027</td>
</tr>
<tr>
<td>10</td>
<td>0.117</td>
</tr>
<tr>
<td>20</td>
<td>0.411</td>
</tr>
<tr>
<td>23</td>
<td>0.572</td>
</tr>
<tr>
<td>30</td>
<td>0.706</td>
</tr>
<tr>
<td>40</td>
<td>0.891</td>
</tr>
<tr>
<td>50</td>
<td>0.970</td>
</tr>
<tr>
<td>60</td>
<td>0.994</td>
</tr>
<tr>
<td>70</td>
<td>0.99915958</td>
</tr>
<tr>
<td>80</td>
<td>0.99991433</td>
</tr>
<tr>
<td>100</td>
<td>0.99999969</td>
</tr>
</tbody>
</table>

At 23, greater than 50% chance

Our results:
365 possible birthdays
58 people/
3 collisions:
July 14
Aug 1
Aug 18
Collision resolution

A collision resolution algorithm must be guaranteed to check every slot.

- linear probing: yes (it sequentially walks through the slots)
- double hashing: ?

Does the probe sequence used for double hashing cover the entire table? (I.e., is any slot ever missed?)
Collision resolution – double hashing

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
<th>probe decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Question: Does the probe sequence cover the entire table?

Use key 24. Show that the probe sequence visits each slot. (Keep wrapping.)
Collision resolution

The probe sequence covers every slot.  
*This is true for every key in the table*
  *try it for other keys*

Why?

The table size M and probe decrement are *relatively prime*. Guarantees that the probe sequence covers the table.

*relatively prime*  
− have no common divisors other than 1  
− think of reducing the fraction 36/45 to 4/5
Collision resolution

Two policies
  - open addressing
    o with linear probing
    o with double hashing

A third policy
  - separate chaining
Collision Resolution

separate chaining
  – each table location references a linked list
  – on collision, add to the linked list, starting at the collision slot

table with keys 24 and 10 (using %7 for the hash):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

- 24
- 10
- 20
  - None
  - None
Complexity

Analysis of separate chaining

If we have N keys, what is

- best case complexity for search: (the key is the first item in the linked-list) $O(1)$
- worst case complexity for search: (must exhaustively search one linked-list) $O(n)$

We have not been analyzing the average case.

We will use known results for average case of the collision resolution policies.
Load factor

The load factor of a hash table with $N$ keys and table size $M$ is given by the following:

$$\lambda = \frac{N}{M}$$

load factor is a measure of how full the table is

Complexity is expressed in terms of the load factor.
EXERCISE

We have 60,000 items to store in a hash table using open addressing with linear probing and we want a load factor of .75.

How big should the hash table be?
Complexity

As load factor increases, efficiency of inserting new keys decreases

Collisions
  - must enumerate through the table to get an empty slot

Searching
  - find it on the first try
  - search by using the probe sequence
  - or search the linked list

We will use known results for the average cases of successful and unsuccessful search for the collision resolution policies
Assume a table with load factor: 

\[ \lambda = \frac{N}{M} \]

Linear probing:
- clusters form
- leads to long probe sequences

It can be shown that the average number of probes is:

\[ \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right) \]

for successful search

\[ \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right) \]

for unsuccessful search

Bad when load factor is close to 1
Not too bad when load factor is .75 or less
Results

>>> # load factor is .75
>>> >>> >>> # linear probing - successful
>>> >>> >>> >>> .5 * (1 + 1/.25)
2.5
>>> >>> # linear probing - unsuccessful
>>> >>> >>> >>> .5 * ( 1 + 1/(.25 *.25))
8.5
Assume a table with load factor:

\[ \lambda = \frac{N}{M} \]

Double hashing:
  clustering less common

It can be shown that the average number of probes is

\[ \frac{1}{\lambda} \ln \left( \frac{1}{1 - \lambda} \right) \]

for successful search

\[ \left( \frac{1}{1 - \lambda} \right) \]

for unsuccessful search

Very good when load factor is .75 or less
>>> # load factor is .75

>>> # double hashing - successful

>>> import math

>>> 1/.75 * math.log(4)

1.8483924814931874

>>> # double hashing – unsuccessful

>>> 1/.25

4.0
Assume a table with load factor: \[ \lambda = \frac{N}{M} \]

Separate chaining:
all keys that collide at a given has location are on the same linked list

It can be shown that the average number of probes is

\[ 1 + \frac{1}{2} \lambda \] for successful search

\[ \lambda \] for unsuccessful search

*Compare the three methods*
## Theoretical Results (number of probes)

### Successful search

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>0.50</th>
<th>0.75</th>
<th>0.90</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>separate chaining</td>
<td>1.25</td>
<td>1.37</td>
<td>1.45</td>
<td>1.49</td>
</tr>
<tr>
<td>linear probing</td>
<td>1.50</td>
<td>2.50</td>
<td>5.50</td>
<td>50.5</td>
</tr>
<tr>
<td>double hashing</td>
<td>1.39</td>
<td>1.85</td>
<td>2.56</td>
<td>4.65</td>
</tr>
</tbody>
</table>

### Unsuccessful search

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>0.50</th>
<th>0.75</th>
<th>0.90</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>separate chaining</td>
<td>0.50</td>
<td>0.75</td>
<td>0.90</td>
<td>0.99</td>
</tr>
<tr>
<td>linear probing</td>
<td>2.50</td>
<td>8.50</td>
<td>50.5</td>
<td>5000.00</td>
</tr>
<tr>
<td>double hashing</td>
<td>2.00</td>
<td>4.00</td>
<td>10.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
Hashing Functions

Good performance requires a good hashing function.
  – the hash function should not cause clustering

Most hash functions
  – map keys to numbers (if not already numbers)
  – then reduce that using mod

Example:
  'hello' → len('hello') % 7

*Must be aware of properties of the hashing function.*
Hashing Functions

Example: hashing function $hash$
  - add the ord values of a string
  - mod by the table size M

For the key 'bat':
  - $hash('bat', M) = (\text{ord}('b') + \text{ord}('a') + \text{ord}('t')) \% M$

```python
def hash(key, M):
    sum = 0
    for c in key:
        sum += \text{ord}(c)
    return sum \% M
```

What are the properties of this hash function?
Does it cause clustering?
def hash(key, M):
    sum = 0
    for c in key:
        sum += ord(c)
    return sum % M

Use:
>>> hash("bat", 7)
3
>>> hash("tab", 7)
3
>>> hash("atb", 7)
3
>>> hash("tide", 7)
2
>>> hash("tied", 7)
2
Hashing Functions

Example: hashing function $h$
- add the ord values of a string
- mod by the table size $M$

$$\text{hash('bat', } M) = (\text{ord('b')} + \text{ord('a')} + \text{ord('t'))} \mod M$$

$$\text{hash('tab', } M) = (\text{ord('t')} + \text{ord('a')} + \text{ord('b'))} \mod M$$

What are the properties of this hash function?
- anagrams hash to the same value

Will that matter?
If it does, how would we fix that?
Hashing Functions

Example: hashing function $h$

- add the ord values of a string
- mod by the table size $M$

Modify to multiply by character position, i.e.,

$$\text{hash('bat', } M) = (\text{ord('b')}*1 + \text{ord('a')}*2 + \text{ord('t')}*3) \% M$$

$$\text{hash('tab', } M) = (\text{ord('t')}*1 + \text{ord('a')}*2 + \text{ord('b')}*3) \% M$$
Hashing Functions

Pitfalls with mod

\[ h(k) = k \mod M \]

Avoid powers of 2 for \( M \)

for \( M = 2^b \), \( h(k) = k \mod 2^b \)

This elects the \( b \) low order bits of \( k \)

In general, when using mod

avoid powers of 2
use prime numbers for \( M \)