HASHMAPS

cs2420 | Introduction to Algorithms and Data Structures | Fall 2016
administrivia...
-Assignment 9 is due tonight

-Assignment 10 is out

-Midterm on Monday
last time...
... we reviewed graphs
today...
- quick data structures review
- quick exercise
- mapping to arrays
- hashmap
- hash function
- collisions, probing, & chaining
- assignment 10 details
quick review
-arrays (and ArrayLists)
  - random access: $O(c)$
  - insert & delete: $O(N)$

-linked lists
  - linear access: $O(N)$
  - insert & delete: $O(c)$

-binary search trees
  - everything: $O(\log N)$
    … must be balanced

-stacks
  - everything: $O(c)$
    … limited to top item

-queues
  - everything: $O(c)$
    … limited to front and back
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays / ArrayLists</td>
<td>$O(c)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>$O(N)$</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>must be balanced</td>
</tr>
<tr>
<td>Stacks</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to top</td>
</tr>
<tr>
<td>Queues</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to front / back</td>
</tr>
</tbody>
</table>
quick exercise
what if we want a data structure that holds integers, and has constant time insertion & deletion?
<table>
<thead>
<tr>
<th></th>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrays / ArrayLists</strong></td>
<td>$O(c)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td><strong>Linked Lists</strong></td>
<td>$O(N)$</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td><strong>Binary Search Trees</strong></td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>must be balanced</td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to top</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to front / back</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>Insertion</td>
<td>Deletion</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Arrays / ArrayLists</td>
<td>O(c)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>O(N)</td>
<td>O(N) or O(c) on ends</td>
<td>O(N) or O(c) on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>must be balanced</td>
</tr>
<tr>
<td>Stacks</td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to top</td>
</tr>
<tr>
<td>Queues</td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to front / back</td>
</tr>
<tr>
<td>Data Structure</td>
<td>Access</td>
<td>Insertion</td>
<td>Deletion</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
<td>-----------</td>
<td>----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Arrays / ArrayLists</td>
<td>O(c)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>O(N)</td>
<td>O(c) on ends</td>
<td>O(c) on ends</td>
<td>time to allocate new items on demand</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>must be balanced</td>
</tr>
</tbody>
</table>

**Stacks**

- Access: O(c)
- Insertion: O(c)
- Deletion: O(c)
  - Access limited to top

**Queues**

- Access: O(c)
- Insertion: O(c)
- Deletion: O(c)
  - Access limited to front / back

What if we also want constant time access to **any** item?
<table>
<thead>
<tr>
<th></th>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays / ArrayLists</td>
<td>$O(c)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>$O(N)$</td>
<td>$O(c)$ on ends</td>
<td>$O(c)$ on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>must be balanced</td>
</tr>
<tr>
<td>Stacks</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to top</td>
</tr>
<tr>
<td>Queues</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to front / back</td>
</tr>
</tbody>
</table>

What if we also want constant time access to **any** item?
-constant time insertion, deletion, and random access

-we know:
  - possible range of integers is \([0 \ldots \text{MAX\_INT}]\)

-what is a naïve, brute-force solution?
  - hint: use an array
- create a gigantic array of size MAX_INT
- initialize everything to -1
- when inserting a number $n$, put it in the array at index $n$
- when accessing a number $n$, check if index $n$ is equal to -1 or not
- when deleting a number $n$, set array at index $n$ to -1
-create a gigantic array of size MAX_INT

-initialize everything to −1

-when inserting a number n, put it in the array at index n

-when accessing a number n, check if index n is equal to −1 or not

-when deleting a number n, set array at index n to −1

does this fulfill the constant time insertion, deletion, and access requirements?
create a gigantic array of size MAX_INT

initialize everything to –1

when inserting a number n, put it in the array at index n

when accessing a number n, check if index n is equal to –1 or not

when deleting a number n, set array at index n to –1

does this fulfill the constant time insertion, deletion, and access requirements?
is this realistic???
mapping to arrays
- let’s try using a smaller array, and mapping large indices to the range of the smaller array

- assume range of possible items is \([0 \ldots 99]\)
  - and assume that we will have \(<<100\) items

- assume array size is only 10

- how can we make this work for integers?
use the mod operator, `%`

guarantee to return a number in the range

`[0 ... (size - 1)]`

mod the input value by the array size for new index

insert:

12, 15, 17, 46, 89, 90
-use the mod operator, \%
  -guarantee to return a number in the range
  \[0 \ldots (\text{size} - 1)\]

-mod the input value by the array size for new index

insert:
12, 15, 17, 46, 89, 90
- use the mod operator, `%`
  - guarantee to return a number in the range `[0 ... (size - 1)]`

- mod the input value by the array size for new index

insert:
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th>12</th>
<th></th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

-use the mod operator, %
- guarantee to return a number in the range
  \[0 \ldots (\text{size} - 1)\]

-mod the input value by the array size for new index

insert:
12, 15, 17, 46, 89, 90

array:  

index:  0  1  2  3  4  5  6  7  8  9
- use the mod operator, `%`
  - guarantee to return a number in the range
    
    \[0 \ldots (\text{size} - 1)\]

- mod the input value by the array size for new index

insert:
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th>12</th>
<th></th>
<th>15</th>
<th>46</th>
<th>17</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
-use the mod operator, \%
  -guarantee to return a number in the range [0 ... (size - 1)]
  
-mod the input value by the array size for new index

insert:
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th>12</th>
<th></th>
<th>15</th>
<th>46</th>
<th>17</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
- **use the mod operator, %**
  - guarantee to return a number in the range $[0 \ldots (\text{size} - 1)]$

- **mod the input value by the array size for new index**

**insert:**
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th>90</th>
<th>12</th>
<th>15</th>
<th>46</th>
<th>17</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
what about data without natural indices?

- how can we do this for non-integer items?
  - integers have an obvious solution... use the integer itself as the index
  - what index should use for, say, a String?

- one solution is to somehow generate an integer from a string
  - length of string?
  - sum of all characters?
  - some combination of both?

- a method that generates an integer index given any object is called a hash function
hashmap

(hashtable)
hashmap
(hashtable)
- a hashmap is a general storage data structure
  - also known as a **hashtable**

- **insertion**, **deletion**, and **access** are all **O(c)**
  - unlike stacks and queues, can access *all* items!
  - these are fast! many applications like databases

<table>
<thead>
<tr>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>magic…?</td>
</tr>
</tbody>
</table>

**Hash Table**
- underlying data structure is just an array
- requires that all data types inserted to have some specified hash function
- hash tables implement an associative array
  - also known as a **dictionary**
  - collection of *(key, value)* pairs
    - *keys*: underlying data
    - *values*: determined by a hash function
-hash functions will give us some hash value which we can map to a valid array index using `%`

-empty spots in the array are set to null

-use any hash value to directly look-up the index of any item

  -insertion, deletion, and access: $O(c)$

  -assuming the hash function is $O(c)$!
hash functions
- A hash function is a function that takes any item as input and produces an integer as output.

- Always returns the same number for the same object.

- If `object1.equals(object2)` must return the same integer for both objects.

- Good hash functions return evenly distributed numbers for the input items.

- It is not required that two non-equal objects have different hash values.
Java’s `hashCode`

- every `Object` in Java has a method `hashCode`

- returns an integer based on the object

- default for this method (if you don’t override it) is to return the memory address of the object

- will not be very well-distributed if your items are contiguous in memory
hash collisions
-remember: *it is NOT required that two non-equal objects have different hash values*

-because of this, it is possible for two different objects to return the same hash values
  -this is called a **collision**

**insert:**
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th>90</th>
<th>12</th>
<th>15</th>
<th>46</th>
<th>17</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>


-remember: *it is NOT required that two non-equal objects have different hash values*

- because of this, it is possible for two different objects to return the same hash values
  - this is called a **collision**

**insert:**
12, 15, 17, 46, 89, 90, 92
-remember: *it is NOT required that two non-equal objects have different hash values*

- because of this, it is possible for two different objects to return the same hash values
  - this is called a **collision**

**insert:**
12, 15, 17, 46, 89, 90, 92

**array:**

| 90 | 12 | 15 | 46 | 17 | 89 |

**index:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**collision! where can we put 92?**
linear probing

-probing is how to resolve collisions
  -we will start with linear probing

-when inserting, if the spot is already taken, simply step forward one index at a time until an empty space is found
  -then insert item in empty space

-when accessing, start at the hashed value index, and if this is not the item we are searching for, begin stepping forward until the item is found
  -what if we hit the end of the array?
  -when do we stop?
insert with linear probing

collisions are resolved on inserts by sequentially scanning the table (with wraparound) until an empty cell is found
insert with linear probing

array: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [89]
index: 0 1 2 3 4 5 6 7 8 9

insert: 89
hash: 9
## Insert with Linear Probing

<table>
<thead>
<tr>
<th>array:</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>insert: 89</th>
<th>hash: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>18</th>
<th>89</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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>
insert with linear probing

array: 0 1 2 3 4 5 6 7 8 9
index: 0 1 2 3 4 5 6 7 8 9

array: 0 1 2 3 4 5 6 7 8 9
insert: 18
hash: 8

array: 0 1 2 3 4 5 6 7 8 9
insert: 49
hash: 9

array: 0 1 2 3 4 5 6 7 8 9
insert: 89
hash: 9
insert with linear probing

array: \[ \begin{array}{cccccccccc}
\_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 89 \\
\end{array} \]

index: \[ \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]

insert: 89
hash: 9

array: \[ \begin{array}{cccccccccc}
\_ & \_ & \_ & \_ & \_ & \_ & \_ & 18 & 89 \\
\end{array} \]

index: \[ \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]

insert: 18
hash: 8

array: \[ \begin{array}{cccccccccc}
49 & \_ & \_ & \_ & \_ & \_ & \_ & 18 & 89 \\
\end{array} \]

index: \[ \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]

insert: 49
hash: 9

array: \[ \begin{array}{cccccccccc}
49 & 58 & \_ & \_ & \_ & \_ & \_ & 18 & 89 \\
\end{array} \]

index: \[ \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]

insert: 58
hash: 8
### Insert with Linear Probing

<table>
<thead>
<tr>
<th>array:</th>
<th>index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>insert:</strong></td>
<td><strong>hash:</strong></td>
</tr>
<tr>
<td>89</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th>index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>insert:</strong></td>
<td><strong>hash:</strong></td>
</tr>
<tr>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th>index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>insert:</strong></td>
<td><strong>hash:</strong></td>
</tr>
<tr>
<td>49</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th>index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>insert:</strong></td>
<td><strong>hash:</strong></td>
</tr>
<tr>
<td>58</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th>index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>insert:</strong></td>
<td><strong>hash:</strong></td>
</tr>
</tbody>
</table>
| 9 | 9 | 9 | 9
access with linear probing

-if the table is not full, the item we seek, or an empty cell, will eventually be found

-cost?

-recall that we are hoping for $O(c)$

-access operation follows the same path as insert... if empty cell reached, item not found

-how do we find 58?

<table>
<thead>
<tr>
<th>array:</th>
<th>49</th>
<th>58</th>
<th>9</th>
<th></th>
<th></th>
<th></th>
<th>18</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
delete with linear probing

- On a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution.

<table>
<thead>
<tr>
<th>array:</th>
<th>49</th>
<th>58</th>
<th>9</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>18</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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>
</tr>
<tr>
<td>delete:</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash:</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
delete with linear probing

- on a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

array: | 49 | 58 | 9 |
index: 0 1 2 3 4 5 6 7 8 9

delete: 89
hash: 9
delete with linear probing

-on a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

how do we find 9?

array: 49 58 9 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
delete: 89
hash: 9
index: 0 1 2 3 4 5 6 7 8 9
delete with linear probing

- on a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

how do we find 9?

array: 49 58 9 18
index: 0 1 2 3 4 5 6 7 8 9
delete: 89
hash: 9

-we must use lazy deletion, which marks items as deleted rather than actually removing them
delete with linear probing

-on a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

how do we find 9?

array: 49 58 9  18 89
index: 0 1 2 3 4 5 6 7 8 9
deleted: F F F F T

delete: 89
hash: 9

-we must use lazy deletion, which marks items as deleted rather than actually removing them
Performance

- If no collisions occur, then the performance of insert, delete, and access is _____

- To determine the real cost, define \( \lambda \), the fraction of the map that is full. Called the **load factor**
  
  \[ 0 \leq \lambda \leq 1 \]

- For each probe into the table, the probability that spot is occupied is \( \lambda \)

- With these assumptions, the average number of cells examined on an insert is \( 1 / (1 - \lambda) \)
  
  - If \( \lambda = 0.5 \), average of two cells examined
- if no collisions occur, then the performance of insert, delete, and access is \( O(c) \)

- to determine the real cost, define \( \lambda \), the fraction of the map that is full called the load factor
  \[ 0 \leq \lambda \leq 1 \]

- for each probe into the table, the probability that spot is occupied is \( \lambda \)

- with these assumptions, the average number of cells examined on an insert is \( \frac{1}{1 - \lambda} \)
  - if \( \lambda = 0.5 \), average of two cells examined
clustering

-if an item’s natural spot is taken, it goes in the next open spot, making a cluster for that hash
  -clustering happens because once there is a collision, there is a high probability more will occur
  -thus, any item that hashes into a cluster will require several attempts to resolve the collision

-feedback loop:
  -the bigger the clusters are, the more likely they are to be hit
  -when a cluster gets hit, it gets bigger

-how might we get around clustering?
quadratic probing

-probing is how to resolve collisions
  -to address clustering, we will use quadratic probing

-if hash(key) = H, and the cell at index H is occupied:
  -try H + 1^2
  -then H + 2^2
  -then H + 3^2
  -and so on...
  -wrap around to beginning of array if necessary
insert with quadratic probing
insert with quadratic probing

| array: |  |  |  |  |  |  |  |  | 89 |
| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

INSERT: 89
HASH: 9
insert with quadratic probing

array:  |   |   |   |   |   |   |   |   | 89
index: 0 1 2 3 4 5 6 7 8 9

array:  |   |   |   |   |   |   |   | 18 | 89
index: 0 1 2 3 4 5 6 7 8 9

array:  |   |   |   |   |   |   | 18 | 89
index: 0 1 2 3 4 5 6 7 8 9
### insert with quadratic probing

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INSERT: 89**
**HASH: 9**

**INSERT: 18**
**HASH: 8**

**INSERT: 49**
**HASH: 9**
**insert with quadratic probing**

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSERT:</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>HASH:</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSERT:</td>
<td></td>
<td>18</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HASH:</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>INSERT:</td>
<td></td>
<td>49</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HASH:</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>49 58</td>
<td>INSERT:</td>
<td></td>
<td></td>
<td>58</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HASH:</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
insert with quadratic probing

array: |   |   |   |   |   |   |   |   | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

array: |   |   |   |   |   |   |   |   | 18 | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

array: | 49 |   |   |   |   |   |   | 18 | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

array: | 49 | 58 |   |   |   |   | 18 | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

array: | 49 | 58 | 9 |   |   |   | 18 | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

array: | 49 | 58 | 9 |   |   |   | 18 | 89 |
index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
concerns...

-is quadratic probing guaranteed to find an open spot? can it search the same spot twice?

-suppose the array size is 16, and $\text{hash}(\text{key}) = 0$

\[
\begin{align*}
0 \mod 16 &= 0 \\
(0 + 1^2) \mod 16 &= 1 \\
(0 + 2^2) \mod 16 &= 4 \\
(0 + 3^2) \mod 16 &= 9 \\
(0 + 4^2) \mod 16 &= 0 \\
(0 + 5^2) \mod 16 &= 9 \\
(0 + 6^2) \mod 16 &= 4 \\
(0 + 7^2) \mod 16 &= 1
\end{align*}
\]
concerns...

-is quadratic probing guaranteed to find an open spot? can it search the same spot twice?

-suppose the array size is 16, and $hash(key) = 0$

\[
\begin{align*}
0 \mod 16 &= 0 \\
(0 + 1^2) \mod 16 &= 1 \\
(0 + 2^2) \mod 16 &= 4 \\
(0 + 3^2) \mod 16 &= 9 \\
(0 + 4^2) \mod 16 &= 0 \\
(0 + 5^2) \mod 16 &= 9 \\
(0 + 6^2) \mod 16 &= 4 \\
(0 + 7^2) \mod 16 &= 1
\end{align*}
\]

**limitation:** at most, half of the table can be used to resolve collisions

once table is half full it is difficult to find an empty spot

...called **secondary clustering**
...solution!

-the following two guidelines guarantee that every spot will be examined at least once
  - ensure that the size of the array is a prime number
  - mapping a hash value to an array index will involved modding by a prime number!
  - ensure that the table is never more than 50% full $\lambda < 0.5$

-guidelines also guarantee no cell is visited twice
  - proof is in the textbook
resizing the table

-since we now have the requirement that $\lambda < 0.5$, what do we do when we need to add another item?

-just like resizing an array, we resize the table to the next largest prime number

-instead of a simple copy-everything-over, all items must be rehashed
  -why?

-this is called rehashing
-quadratic probing does not eliminate the clustering problem

-but, secondary clustering is not as severe as primary clustering

-the only reason not to use quadratic probing is when maintaining a half-empty array is too costly

-can you think of an alternative for collision management?
separate chaining

- why not make each spot in the array capable of holding more than one item?
  - use an array of linked lists
  - hash function selects index into array
  - called separate chaining

- for insertion, append the item to the end of the list
  - insertion is $O(c)$ if we have what?

- accessing is a linear scan through the list
  - fast if the list is short
performance

-different definition of the load factor, $\lambda_c$

$\lambda_c = \text{average length of linked lists}$

-therefore, access and delete operations scan $\lambda_c$ items

-instead of rehashing when the table is half full, rehash when $\lambda_c$ becomes large

-analysis is required to find a good value

-rehashing is never required since lists can grow indefinitely, but it can be beneficial
assignment 10 details
-you will implement a quadratic probing hash table AND a separate chaining hash table for Strings

-the constructors for these hash tables takes a HashFunctor object
  -recall that a functor is an object which encapsulates a method (just like Comparator)

-the HashFunctor defines a hash method, which implements a hash function

-you can create any number of different hash functions this way without changing any code in your hash table
  -yay for encapsulation!
-start thinking about…
  -what is a bad hash function for Strings?
  -what is a good hash function for Strings?

-remember, Strings are just a sequence of chars
  -and a char is just a smaller int

-we can perform any operation or combination of ops on the small numbers (chars) that make up the String

-an example String hash function is in the book
  -there are also a bunch of good ones on the web
midterm 2
-linked lists

-stacks & queues

-trees

-graphs

-hashmaps
tips

- don’t memorize your implementation

- be able to talk through the topics
  - what is a data structure good for
  - what is it bad for
  - what are the basic functions
  - reason about complexity

- get a study partner and make up questions for each other
next time...
- midterm on Monday in class

- reading for next week
  - chapter 21 in book (binary heaps)

- homework
  - assignment 9 due tonight
  - assignment 10 out