dictatorial /ˈdiktərəl/ adj. 1 having the authority or power of a dictator. 2 overbearing in manner or conduct. [Latin: related to dictator]
diction /ˈdɪkʃ(ə)n/ n. manner of speaking or singing: diction in speaking or singing

dictionary /ˈdɪkʃənri/ n. (pl. dictionaries) a book listing (usu. alphabetically) the words of a language, giving corresponding words in another language. 2 reference book explaining the words of a language.
Announcements

- Submit Prelim 2 conflicts by Thursday night
- A6 is due Nov 7 (tomorrow!)
# Ideal Data Structure

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>\text{add}(\text{val } x)</th>
<th>\text{get}(\text{int } i)</th>
<th>\text{contains}(\text{val } x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>LinkedList</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

**Goal:** \(O(1)\) \(O(1)\) \(O(1)\)

AKA add, lookup, search
Mystery Data Structure in Your Life

What do these data structures have in common?
# New Data Structure: Hash Set

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>get(int i)</th>
<th>contains(val x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>HashSet</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Expected time
Worst-case: $O(n)$

AKA add, lookup, search
Idea: finding an element in an array takes constant time when you know which index it is stored in. So… let’s place elements in the array based on their starting letter! (A=0, B=1, …)

**Intuition behind a Hash Set**

```
add(“CA”)  
CA  # of 1st letter  2  

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 25  
CA MA NY OR PA  

contains(“DE”)  
DE  # of 1st letter  3  
```
What could possibly go wrong?

- Some buckets get used quite a bit! called **Collisions**
- Not all buckets get used
Hash Functions

- Requirements:
  1) deterministic
  2) return a number*

- Properties of a good hash:
  1) fast
  2) collision-resistant
  3) evenly distributed
  4) hard to invert

* the number is either in [0..n-1] where n is the size of the Hash Set, or you compute the hash and then % n, constraining it to be in [0…n-1]
Example: hashCode()

- Method defined in java.lang.Object
- Default implementation: uses memory address of object
  - If you override equals, you must override hashCode!!!
- String overrides hashCode:
  \[
  s.\text{hashCode}() := s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \ldots + s[n-1]
  \]

*Do we like this hashCode?*
Can we have perfect hash functions?

- Perfect hash functions map each value to a different index in the hash table

- Impossible in practice
  - Don’t know size of the array
  - Number of possible values far far exceeds the array size
  - No point in a perfect hash function if it takes too much time to compute
Collision Resolution

Two ways of handling collisions:

1. Chaining
2. Open Addressing
Chaining (1)

Each bucket is the beginning of a Linked List

add("NY")
Chaining (2)

Each bucket is the beginning of a Linked List

Note: Would be better to add elements to the head of the linked list.
Chaining (3)

Each bucket is the beginning of a Linked List

add(“NY”)  
add(“NJ”)  
get(“NJ”)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

b

NJ -> # of 1st letter -> 13

CA MA NY OR PA

NJ
Chaining in Action

Insert the following elements (in order) into an array of size 6:
if code > n, use (code % n_buckets)

<table>
<thead>
<tr>
<th>element</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>0</td>
<td>9</td>
<td>17</td>
<td>11</td>
<td>19</td>
</tr>
</tbody>
</table>

```
0   1   2   3   4   5
```

- a
- e
- b
- c
- d
Open Addressing (1)

**Probing:** Find another available space

```
add(“NY”)  

```

```
<table>
<thead>
<tr>
<th>b</th>
<th>CA</th>
<th>CO</th>
<th></th>
<th>MA</th>
<th>NY</th>
<th>OR</th>
<th>PA</th>
</tr>
</thead>
</table>
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```
Open Addressing (2)

**Probing:** Find another available space

```
add("NY")
add("NJ")
```

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| b | CA | CO |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Open Addressing (3)

**Probing:** Find another available space

What could possibly go wrong? 
add(“NY”), add(“NJ”), get(“NY”), get(“NJ”)
Deletion Problem w/Open Addressing

**Probing:** Find another available space

<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>
<th>16</th>
<th>...</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>CA</td>
<td>CO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MA</td>
<td>NY</td>
<td>OR</td>
<td>PA</td>
<td>NJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `add("NY")`
- `add("NJ")`
- `get("NY")`
- `get("NJ")`

Search for NJ
(stops searching b/c element b[13] is NULL!)
Deletion Solution for Open Addressing

**Probing:** Find another available space

*Need to mark element as “not present”*
Indicates to search that it should keep looking

Addition:
- `add("NY")`
- `add("NJ")`
- `get("NY")`
- `get("NJ")`

Search for NJ:
(searches until it finds a NULL element or the present element it’s looking for)
Different probing strategies

When a collision occurs, how do we search for an empty space?

- **clustering**: problem where nearby hashes have very similar probe sequence so we get more collisions
- **linear probing**: search the array in order: \(i, i+1, i+2, i+3\ldots\)
- **quadratic probing**: search the array in nonlinear sequence: \(i, i+1^2, i+2^2, i+3^2\ldots\)

In order to have access to every bucket, important for size to be a prime number when using quadratic probing.
Linear Probing in Action

Insert the following elements (in order) into an array of size 5:

<table>
<thead>
<tr>
<th>element</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>0</td>
<td>8</td>
<td>17</td>
<td>12</td>
</tr>
</tbody>
</table>

- Inserting d:
  - Probe #1: Inserting d: i full!
  - Probe #2: Inserting d: i+1 full!
  - Probe #3: Inserting d: i+2 has space!
Insert the following elements (in order) into an array of size 5:

<table>
<thead>
<tr>
<th>element</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>0</td>
<td>8</td>
<td>17</td>
<td>12</td>
</tr>
</tbody>
</table>

probe #1 inserting d: i
full!

probe #2 inserting d: i + 1^2
full!

probe #3 inserting d: i + 2^2
has space!
Load Factor

**Load factor**

\[
\lambda = \frac{\text{# of entries}}{\text{length of array}}
\]

If load factor = \( \frac{1}{2} \), expected # of probes is 2.
What happens when the array becomes too full? i.e. load factor gets a lot bigger than \( \frac{1}{2} \)?

no longer expected constant time operations

0 1
waste of memory best range too slow
Resizing

Solution: *Dynamic resizing*

- double the size*
- reinsert / rehash all elements to new array
- Why not simply copy into first half?

*if using quadratic probing, use a prime >2n
Collision Resolution Summary

**Chaining**
- store entries in separate chains (linked lists)
- can have higher load factor/degrades gracefully as load factor increases

**Open Addressing**
- store all entries in table
- use linear or quadratic probing to place items
- uses less memory
- clustering can be a problem — need to be more careful with choice of hash function
Application: Hash Map

Map<K,V> {
    void put(K key, V value);
    void update(K key, V value);
    V get(K key);
    V remove(K key);
}

- Use the **key** for lookups
- Store the **value**

**Example:** **key** is the word, **value** is its definition
Hash Map (1)

```
put("New York", "NY")
```
Hash Map (2)

```java
put("New York","NY")

get("California")
```
HashMap in Java

- Computes hash using key.hashCode()
  - No duplicate keys
- Uses chaining to handle collisions
- Default load factor is .75
- Java 8 attempts to mitigate worst-case performance by switching to a BST-based chaining!
Hash Maps in the Real World

- Network switches
- Distributed storage
- Database indexing
- Index lookup (e.g., Dijkstra's shortest-path algorithm)
- Useful in lots of applications…