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>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>Goal</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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>

Intuition behind a Hash Set

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, ...)

add("CA")
CA # of 1st letter 2
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 \([0\ldots n-1]\) where \(n\) is the size of the Hash Set, or you compute the hash and then \(\%\ n\), constraining it to be in \([0\ldots n-1]\)

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:
  \[ s\text{.hashCode()} = s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1] \]
  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
**Chaining (2)**

Each bucket is the beginning of a Linked List

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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>
```

Open Addressing (1)

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Probing: Find another available space

**Open Addressing (2)**

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>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Open Addressing (3)**

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>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

What could possibly go wrong?

add("NY"), add("NJ"), get("NY"), get("NJ")
Deletion Problem w/Open Addressing

Probing: Find another available space

add("NY")
get("NY")
get("NJ")

Deletion Solution for Open Addressing

Probing: Find another available space

add("NY")
get("NY")
get("NJ")

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...
quadratic probing: search the array in nonlinear sequence: i, i+1^2, i+2^2, i+3^2...

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>

Quadratic 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>

Load Factor

Load factor \( \lambda = \frac{\# \text{ of entries}}{\text{length of array}} \)

If load factor = 1/2, expected # of probes is 2.

What happens when the array becomes too full? i.e. load factor gets a lot bigger than 1/2?

no longer expected constant time operations

waste of memory 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

<table>
<thead>
<tr>
<th>Chaining</th>
<th>Open Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>- store entries in separate chains (linked lists)</td>
<td>- store all entries in table</td>
</tr>
<tr>
<td>- can have higher load factor/degrades gracefully as load factor increases</td>
<td>- use linear or quadratic probing to place items</td>
</tr>
<tr>
<td>- uses less memory</td>
<td>- uses less memory</td>
</tr>
<tr>
<td>- clustering can be a problem — need to be more careful with choice of hash function</td>
<td>- clustering can be a problem — need to be more careful with choice of hash function</td>
</tr>
</tbody>
</table>

Application: Hash Map

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

**Example:** key is the word, value is its definition

Hash Map (1)

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

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…