CSCI 241

Lecture 17
Map ADT
A3 Overview
Hash Functions, Hash Tables, Hash Sets, Hashtags

Today's exercises: on paper and Socrative - have paper+pencil ready!
CSCI 241

Lecture 17
Map ADT
A3 Overview
Hash Functions, Hash Tables, Hash Sets, Hashtags
Announcements

• A3 is out!

• Exam is Friday!

• A3 video is posted (A3.mp4)
Goals

• Know the purpose and operations of the Map ADT

• Know the purpose, definition, and properties of hash functions.

• Know how to use a hash function to implement a hash table.

• Know how to use modular arithmetic to construct a basic hash function on integers.

• Know how to use chaining for collision resolution.

• Know the definition of load factor in a hash table.
The Map ADT

• In math, a map is a function.

• What is a function, anyway?
The Map ADT

• In math, a map is a function.

• If F is a map then
  \[ F(a) \rightarrow b \]
  means that a maps to b.

• F has a:
  
  • **domain** - the set of values F maps from
  
  • **range** - the set of values that F maps a domain element to
  
  • **codomain** - the set of all possible values in the range’s type, regardless of whether any element in the domain maps to it
The Map ADT

```java
Thing[] a = new Thing[10];
```
The Map ADT

- Arrays are great!
  - **Domain**: 0..a.length
  - **Range**: all elements in the array
  - **Codomain**: the array's type

```java
Thing[] a = new Thing[10];
```
The Map ADT

We get to choose the **codomain**.

Codomain: Thing objects.

```java
Thing[] a = new Thing[10];
```

```
<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Thing</td>
</tr>
<tr>
<td>int thingField1 0</td>
<td></td>
</tr>
<tr>
<td>int thingField2 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thing</td>
</tr>
<tr>
<td>int thingField1 1</td>
<td></td>
</tr>
<tr>
<td>int thingField2 1</td>
<td></td>
</tr>
</tbody>
</table>
```
The Map ADT

- Arrays are great!
  - **Domain**: 0..a.length
  - **Range**: all elements in the array
  - **Codomain**: the array's type

We get to choose the codomain.

Codomain: Thing objects.

```java
Thing[] a = new Thing[10];
```
The Map ADT

- Arrays are great!

- We get to choose the codomain - type of the array.

- Wouldn’t it be nice to choose the domain as well?

- The Map ADT represents a mapping from keys to values.

  - we get to choose the type of the keys (domain) AND the values (codomain)
The Map Interface

```java
public interface Map<K, V> {
    /** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. */
    V get(Object key);

    /** Associates the specified value with the specified key in this map */
    V put(K key, V value);

    /** Removes the mapping for a key from this map if it is present */
    V remove(Object key);

    // more methods
}
```
Example Uses of Maps

Map<String, Integer> wordCounts;

<table>
<thead>
<tr>
<th>Word</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;I&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;we&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;you&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Map<Student, char> grades;

Character

Student name: "P"
Reminder: The **Set** ADT

- A **Set** maintains a collection of **unique** things.

- Java has this ADT built in as an interface:
  
  ```java
  java.util.Set
  ```

- Some methods from `java.util.Set`:
  
  - `boolean add(Object ob)`
  - `boolean contains(Object ob)`
  - `boolean remove(Object ob)`
Reminder: The Set ADT

• A Set maintains a collection of unique things.

• Java has this ADT built in as an interface:
  
  ```java
  java.util.Set<T>
  ```

• Some methods from `java.util.Set`:
  • boolean `add(T ob)`
  • boolean `contains(T ob)`
  • boolean `remove(T ob)`
Hashing: Motivation

- Consider implementations of the Set ADT:

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>contains</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted Array</strong> or <strong>Linked List</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted Linked List</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted Array</strong></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>AVL Tree</strong></td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>Magical Array</strong></td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>O(1)*</td>
</tr>
</tbody>
</table>
How would you implement a Set that can only contain the digits 0..10?
Remember Radix Sort?

[07, 19, 61, 11, 14, 54, 01, 08]

Bukkits on 1’s place
insert(4)  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>

boolean[] A:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>

insert(4)
insert(4)
insert(7)

boolean[] A:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>
insert(4)
insert(7)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>boolen[] A:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>boolean[] A:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

Write pseudocode for a Set that can only contain the digits 0..10.

```java
public class DigitSet {
    boolean[] A[10];

    /** pre: 0 <= i < 10 */
    void insert(int i) {
        // your code
    }
    /** pre: 0 <= i < 10 */
    void contains(int i) {
        // your code
    }
}
```
Exercise

Write pseudocode for a Set that can only contain the digits 0..10.

```java
public class DigitSet {
    boolean[] A[10];

    /** pre: 0 <= i < 10 */
    void insert(int i) {
        A[i]=true;
    }

    /** pre: 0 <= i < 10 */
    void contains(int i) {
        return A[i];
    }
}
```
**Direct-Address Table**

```plaintext
insert(4)
insert(7)
insert(4)

insert(i):
    A[i] = true

contains(i):
    return A[i]

remove(i):
    A[i] = false
```

<table>
<thead>
<tr>
<th>boolean[] A:</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>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
```
Direct-Address Table

• This was easy because the Set contents came from a small, fixed space of possible values (0..10).

• Hash functions are the **magic** that lets us map *any* space of values onto a fixed space of integer values.
Reminder: The Modulus Operator

\( a \% b \) gives the remainder when dividing \( a \) by \( b \):

\[
\begin{align*}
12 \% 8 & \Rightarrow 4 \\
24 \% 10 & \Rightarrow 4 \\
4 \% 10 & \Rightarrow 4 \\
28 \% 14 & \Rightarrow 0
\end{align*}
\]
Exercise

a \% b gives the remainder when dividing a by b:

12 \% 8 \Rightarrow 4
12 \% 3 \Rightarrow 0

24 \% 10 \Rightarrow 4
14 \% 3 \Rightarrow 2

4 \% 10 \Rightarrow 4
8 \% 5 \Rightarrow 3

28 \% 14 \Rightarrow 0
3 \% 10 \Rightarrow 3
Hash Tables with Integers

How can we determine an index for any integer in a fixed-sized array?

• Modular arithmetic:
  store value k in the k % 10 bucket

  (14 % 10) => 4

<table>
<thead>
<tr>
<th>boolean[] A:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Hash Tables with Integers

How can we determine an index for any integer in a fixed-sized array?

• Modular arithmetic: store value $k$ in the $k \% 10$ bucket
  
  • $(14 \% 10) \Rightarrow 4$
  
  • $(10 \% 10) \Rightarrow 0$

<table>
<thead>
<tr>
<th>boolean[]</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>
Hash Tables with Integers

How can we determine an index for any integer in a fixed-sized array?

- Modular arithmetic: store value $k$ in the $k \mod 10$ bucket
  - $(14 \mod 10) \Rightarrow 4$
  - $(10 \mod 10) \Rightarrow 0$
  - $(1 \mod 10) \Rightarrow 1$

<table>
<thead>
<tr>
<th>boolean[] A:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Hash Tables with Integers

How can we determine an index for any integer in a fixed-sized array?

- Modular arithmetic: store value $k$ in the $k \% 10$ bucket
  - $(14 \% 10) \Rightarrow 4$
  - $(10 \% 10) \Rightarrow 0$
  - $(1 \% 10) \Rightarrow 1$
  - $(11 \% 10) \Rightarrow 1$

<table>
<thead>
<tr>
<th>boolean[]</th>
<th>A:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>
Hash Tables with Integers

How can we determine an index for any integer in a fixed-sized array?

- Modular arithmetic: store value k in the k \% 10 bucket
  - \((14 \% 10) \Rightarrow 4\)
  - \((10 \% 10) \Rightarrow 0\)
  - \((1 \% 10) \Rightarrow 1\)
  - \((11 \% 10) \Rightarrow 1\)

```
boolean[] A:
0 T
1 T
2 F
3 F
4 T
5 F
6 F
7 F
8 F
9 F
```

uh oh…
Hash Tables with Integers: Collisions

- Modular arithmetic: store value $k$ in the $k \% 10$ bucket
  - $(14 \% 10) => 4$
  - $(10 \% 10) => 0$
  - $(1 \% 10) => 1$
  - $(11 \% 10) => 1$
Hash Tables with Integers: Collisions

- Modular arithmetic: store value \( k \) in the \( k \mod 10 \) bucket
  
  - \((14 \mod 10) \Rightarrow 4\)
  
  - \((10 \mod 10) \Rightarrow 0\)
  
  - \((1 \mod 10) \Rightarrow 1\)
  
  - \((11 \mod 10) \Rightarrow 1\)
Exercise

Insert the following values into a table of size 8: 1, 11, 16, 4, 5, 8, 0, 13

- Use \( h(k) = k \mod 8 \) as the hash function.
- Use chaining for collision resolution.
HashSet<T>

/** insert value into the set. return false if the
value was already in the set, true otherwise */
boolean insert(T value) {
    int h = hash(value)
    search the list at A[h] for value
    if found:
        return false
    else:
        insert value into A[h] and return true
}

/** return true if value is in the set,
 * false otherwise */
boolean contains(T value) {
    ... }

/** insert value into the set. return true if the
 * value was in the set, false otherwise */
boolean remove(T value) {
    ... }
HashSet<T>:
What’s the runtime?

/** insert value into the set. return false if the value was already in the set, true otherwise */

boolean insert(T value) {
    int h = hash(value)
    search the list at A[h] for value
    if found:
        return false  \( \mathcal{O}(1) \)
    else:
        insert value into A[h] and return true \( \mathcal{O}(1) \)
}

\[ h(x) = x \mod 4 \]

4, 0, 16, 32, 8, 12, 24, ...
HashSet<T>: What’s the runtime?

/** insert value into the set. return false if the value was already in the set, true otherwise */

boolean insert(T value) {
    int h = hash(value)  \[O(1)\]
    search the list at A[h] for value \[O(length of list)\]
    if found:
        return false \[O(1)\]
    else:
        insert value into A[h] and return true \[O(1)\]
}

Object.hashCode()
HashSet<T>:
What’s the runtime?
All operations require searching a single bucket and doing some other stuff that runs in O(1).

/** return true if value is in the set, * false otherwise */
boolean contains(T value) { ... }

/** remove value from the set. return true if the * value was in the set, false otherwise */
boolean remove(T value) { ... }
Hash Tables: Load Factor
Hash Tables: Load Factor

How full is your hash table?

Load factor $\lambda = \frac{\# \text{ entries in table}}{\text{size of the array}}$

With a perfectly-behaved hash function, average bucket size is $\lambda$, so average-case runtime is $O(\lambda)$. 
Exercise

• What is the load factor of the hash table you built, after all the insertions?
Let's talk about A3.
A3 has 4 phases.
A3 has 4 phases.
A3 has 4 phases.

It isn’t so bad:

• total lines of code is probably \( \leq \) A2
• nothing here is as tricky as AVL rebalance
• you’re given unit tests
A3 has 4 phases.

0. Write an ArrayList clone
A3 has 4 phases.

0. Write an ArrayList clone
   (done in Lab 5!)
A3 has 3 phases.
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

use AList to handle growing the array!
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table implementation of Map.

use AList to handle growing the array!
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table implementation of Map.

3. Use the Map to augment the heap, making the following operations efficient:
   - boolean contains(V v);
   - void changePriority(V v, P newP);
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table implementation of Map.

3. Use the Map to augment the heap, making the following operations efficient:
   - boolean contains(V v);
   - void changePriority(V v, P newP);

(use AList to handle growing the array!

(not using AList to handle growing the array)
Phase 3 - Hash your Heap

In Phase 1 Heap:

- `contains` requires searching the whole tree.
- `changePriority` requires searching the whole tree, then bubbling down or up.
Phase 3 - Hash your Heap

In Phase 3 Heap:

- Each heap value is stored in the heap and in a HashTable that tracks its index in the heap.

HashTable:<V, Integer>:

<table>
<thead>
<tr>
<th>value</th>
<th>i (index in heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
</tr>
</tbody>
</table>

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]
Phase 3 - Hash your Heap

In Phase 3 Heap:

• Each heap value is stored in the heap and in a HashTable that tracks its index in the heap.

HashTable<V, Integer>:

<table>
<thead>
<tr>
<th>value</th>
<th>i (index in heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
</tr>
</tbody>
</table>

To maximize confusion:

• The hash table maps Heap values to heap indices.
• The hash table’s keys are the heap’s values.

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]
In Phase 3 Heap:

boolean contains(V v):
true iff map contains key v

HashTable<V, Integer>:
value | i (index in heap)
---|---
4 | 0
8 | 4
6 | 1
38 | 8
35 | 6
21 | 3
10 | 10
19 | 12

Heap: [4, 6, 5, 21, 8, 14, 35, 22, 38, 55, 10, 20, 19]
Phase 3 - Hash your Heap

In Phase 3 Heap:

```java
void changePriority(V v, P newP):
    find where v lives using the map
    change priority of heap Entry
    bubble it up or down
```

HashTable<V, Integer>:

<table>
<thead>
<tr>
<th>Value</th>
<th>Index in Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
</tbody>
</table>

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]