Introduction to Hash Tables

Data Structures and Algorithms
Why are we so obsessed with Dictionaries? *It’s all about data baby!*  
When dealing with data:  
- Adding data to your collection  
- Getting data out of your collection  
- Rearranging data in your collection

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why are we so obsessed with Dictionaries? It’s all about data baby!

When dealing with data:
- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key,value)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n) + arrayExpansion</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n), O(logn) if sorted</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn) + replace</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n) + replace</td>
</tr>
</tbody>
</table>
Can we do better?

Implement a dictionary that accepts only integer keys between 0 and some value \( k \)

Leverage Array Indices!

“Direct address map”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key,value)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1)</td>
</tr>
<tr>
<td>worst</td>
<td>O(1) + arrayExpansion</td>
</tr>
<tr>
<td>get(key)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1)</td>
</tr>
<tr>
<td>worst</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(key)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1)</td>
</tr>
<tr>
<td>worst</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
public V get(int key) {
    this.ensureIndexNotNull(key);
    return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key] = value;
}

public void remove(int key) {
    this.ensureIndexNotNull(key);
    this.array[key] = null;
}
Can we do this for any integer?

Idea 1:
Create a GIANT array with every possible integer as an index

Problems:
- Can we allocate an array big enough?
- Super wasteful

Idea 2:
Create a smaller array, but create a way to translate given integer keys into available indices

Problem:
- How can we pick a good translation?
**Review: Integer remainder with %**

The `%` operator computes the remainder from integer division.

- $14 \% 4$ is 2
- $218 \% 5$ is 3

```
\begin{array}{c}
3 \\
4 \quad 14 \\
\hline
12 \\
2 \\
\end{array}
```

```
\begin{array}{c}
43 \\
5 \quad 218 \\
\hline
20 \\
18 \\
15 \\
3 \\
\end{array}
```

Applications of `%` operator:

- Obtain last digit of a number: $230857 \% 10$ is 7
- See whether a number is odd: $7 \% 2$ is 1, $42 \% 2$ is 0
- Limit integers to specific range: $8 \% 12$ is 8, $18 \% 12$ is 6
First Hash Function: % table size

<table>
<thead>
<tr>
<th>indices</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>element</td>
<td>&quot;poo&quot;</td>
<td>&quot;biz&quot;</td>
<td></td>
<td></td>
<td>&quot;bar&quot;</td>
<td></td>
<td></td>
<td>&quot;bop&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
put(0, "foo");  0 % 10 = 0
put(5, "bar");  5 % 10 = 5
put(11, "biz"); 11 % 10 = 1
put(18, "bop"); 18 % 10 = 8
put(20, "poo"); 20 % 10 = 0
```

Collision!
Implement First Hash Function

```java
public V get(int key) {
    int newKey = key % this.array.length;
    this.ensureIndexNotNull(key);
    return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key % this.array.length] = value;
}

public void remove(int key) {
    int newKey = key % this.array.length;
    this.ensureIndexNotNull(key);
    this.array[key] = null;
}
```
Hash Obsession: Collisions

When multiple keys translate to the same location of the array

The fewer the collisions, the better the runtime!
Handling Collisions

Solution 1: Chaining

Each space holds a “bucket” that can store multiple values. Bucket is often implemented with a LinkedList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + ( \lambda ))</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + ( \lambda ))</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + ( \lambda ))</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Average Case:
Depends on average number of elements per chain

Load Factor \( \lambda \)
If \( n \) is the total number of key-value pairs
Let \( c \) be the capacity of array
Load Factor \( \lambda = \frac{n}{c} \)
Practice

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a) (5, b) (11, a) (7, d) (12, e) (17, f) (1, g) (25, h)
Can we do better?

Idea 1: Take in better keys
- Can’t do anything about that right now

Idea 2: Optimize the bucket
- Use an AVL tree instead of a Linked List
- Java starts off as a linked list then converts to AVL tree when collisions get large

Idea 3: Modify the array’s internal capacity
- When load factor gets too high, resize array
  - Double size of array
  - Increase array size to next prime number that’s roughly double the array size
    - Prime numbers reduce collisions when using % because of divisors
  - Resize when \( \lambda \approx 1.0 \)
  - When you resize, you have to rehash
What about non integer keys?

Hash Function
An algorithm that maps a given key to an integer representing the index in the array for where to store the associated value

Goals

Avoid collisions
- The more collisions, the further we move away from O(1)
- Produce a wide range of indices

Uniform distribution of outputs
- Optimize for memory usage

Low computational costs
- Hash function is called every time we want to interact with the data
How to Hash non Integer Keys

Implementation 1: Simple aspect of values
public int hashCode(String input) {
    return input.length();
}

Implementation 2: More aspects of value
public int hashCode(String input) {
    int output = 0;
    for (char c : input) {
        out += (int)c;
    }
    return output;
}

Implementation 3: Multiple aspects of value + math!
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}
Balanced Hash Function

```java
public int hashCode(String input) {
    int accum = 1;
    int output = 0;
    for (char c : input) {
        out += accum * (int)c;
        accum *= 31;
    }
    return output;
}
```

Pretty fast, O(n)
Uses both character values and positions, few collisions
Why 31? Magical research!
Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```java
public int hashCode(String input) {
    return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

Java and Hash Functions

Object class includes default functionality:
- equals
- hashCode

If you want to implement your own hashCode you MUST:
- Override BOTH hashCode() and equals()
- If a.equals(b) is true then a.hashCode() == b.hashCode() MUST also be true