CSE 373: Hash functions and hash tables

Michael Lee
Monday, Jan 22, 2018

Warmup

Warmup: Consider the following method.

```
private int mystery(int x) {
    if (x <= 10) {
        return 5;
    } else {
        int foo = 0;
        for (int i = 0; i < x; i++)
            foo += x;
        return foo + (2 * mystery(x - 1)) + (3 * mystery(x - 2));
    }
}
```

With your neighbor, answer the following.

1. Construct a mathematical formula $T(x)$ modeling the *worst-case runtime* of this method.
2. Construct a mathematical formula $M(x)$ modeling the *integer output* of this method.

Warmup

1. Construct a mathematical formula $T(x)$ modeling the *worst-case runtime* of this method.

$$T(x) = \begin{cases} 1 & \text{if } x \leq 10 \\ x + T(x-1) + T(x-2) & \text{otherwise} \end{cases}$$

2. Construct a mathematical formula $M(x)$ modeling the *integer output* of this method.

$$M(x) = \begin{cases} 5 & \text{if } x \leq 10 \\ x^2 + 2T(x-1) + 3T(x-2) & \text{otherwise} \end{cases}$$

Plan of attack

Today's plan:

Goal: Learn how to implement a hash map

Plan of attack:

1. Implement a limited, but efficient dictionary
2. Gradually remove each limitation, *adapting* our original
3. Finish with an efficient and general-purpose dictionary

Implementing `FinitePositiveIntegerDictionary`

Step 1:

Implement a dictionary that accepts only *integer* keys between 0 and some k.

(This is also known as a "direct address map".)

How would you implement `get`, `put`, and `remove` so they all work in $\Theta(1)$ time?

Hint: first consider what underlying data structure(s) to use. An array? Something using nodes? (E.g. a linked list or a tree).

Implementing `FinitePositiveIntegerDictionary`

Solution: Create and maintain an internal array of size k. *Map* each key to the corresponding index in array:

```
public V get(int key) {
    this.ensureIndexNotNull(key);
    return this.array[key].value;
}

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

public void remove(int key) {
    this.ensureIndexNotNull(key);
    this.array[key] = null;
}

private void ensureIndexNotNull(int index) {
    if (this.array[index] == null) {
        throw new NoSuchKeyException();
    }
}
```

Implementing `IntegerDictionary`

Step 2:

Implement a dictionary that accepts **any** *integer* key.

Idea 1: Create a *giant* array that has one space for every integer.

What's the problem?

- Can we even allocate an array that big?
- Potentially very wasteful: what if our data is sparse?
  This is also a problem with our `FinitePositiveIntegerDictionary`!

Implementing `IntegerDictionary`

Step 2:

Implement a dictionary that accepts **any** *integer* key.

Idea 2: Create a smaller array, and mod the key by array length.

So, instead of looking at `this.array[key]`, we look at `this.array[key % this.array.length]`.

A brief interlude on mod:

The "modulus" (mod) operation

In math, "$a \bmod b$" is the remainder of a divided by b.*
Both a and b MUST be integers.

In Java, we write this as `a % b`.

*This is a slight over-simplification

Examples (in Java syntax)

- `28 % 5 == 3`
- `427 % 100 == 27`
- `8 % 8 == 0`
- `2 % 8 == 2`

Useful when you want "wrap-around" behavior, or want an integer to stay within a certain range.

Implementing `IntegerDictionary`

Idea 2: Create a smaller array, and mod the key by array length.

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

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

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

What's the bug here?

Implementing `IntegerDictionary`: resolving collisions

The problem: **collisions**

Suppose the array has length 10 and we insert the key-value pairs $(8, \text{"foo"})$ and $(18, \text{"bar"})$. What does the dictionary look like?

Implementing `IntegerDictionary`: resolving collisions

There are several different ways of resolving collisions. We will study one technique today called *separate chaining*.

Idea: Instead of storing key-value pairs at each array location, store a "chain" or "bucket" that can store multiple keys!

Implementing `IntegerDictionary`

Two questions:

1. What ADT should we use for the bucket?
   A dictionary!
2. What's the *worst-case* runtime of our dictionary, assuming we implement the bucket using a linked list?
   $\Theta(n)$ – what if everything gets stored in the same bucket?

Implementing `IntegerDictionary`: analyzing runtime

The worst-case runtime is $\Theta(n)$. Assuming the keys are random, what's the *average-case* runtime?

Depends on the average number of elements per bucket!

The "load factor" λ

Let n be the total number of key-value pairs.
Let c be the capacity of the internal array.
The "load factor" λ is $\lambda = \frac{n}{c}$.

Assuming we use a linked list for our bucket, the *average* runtime of our dictionary operations is $\Theta(1 + \lambda)$!

Implementing `IntegerDictionary`: improving performance

Goal: Improve the *average* runtime of our `IntegerDictionary`

Ideas:

- Right now, we can't do anything about the keys we get.
- Can we modify the bucket somehow?
  Idea: use a self-balancing tree for the bucket. Worst-case runtime is now $\Theta(\log(n))$.
  Problem: constant factor is worse then a linked list; implementation is more complex.
- Can we modify the array's internal capacity somehow?
  If the load factor is too high, resize the array!

Important: When separate chaining, we should keep $\lambda \approx 1.0$.

Implementing `IntegerDictionary`: improving performance

Once the load factor is large enough, we resize. There are two common strategies:

- Just double the size of the array
- Increase the array size to the next prime number that's (roughly) double the array size

Three question:

1. How do you resize the array?
2. What's the runtime of resizing?
3. Why use prime numbers?

So far...

So far...

1. Implement a finite, positive integer dictionary
2. Implement an integer dictionary
   - How can we avoid using a lot of memory?
   - How do we handle collisions?
   - How do we keep the *average* performance $\Theta(1)$?
3. Implement a general-purpose dictionary

Implementing a general dictionary

Problem: We have an efficient dictionary, but only for integers. How do we handle arbitrary keys?

Idea: Wouldn't it be neat if we could convert any key into an integer?

Solution: Use a hash function!

Hash functions

Hash function

A hash function is a mapping from the key set U to an integer.

Hash functions

There are many different properties a hash function could have.

Using hash functions inside dictionaries: useful properties

A hash function that is intended to be used for a dictionary should ideally have the following properties:

- **Uniform distribution of outputs:**
  In Java, there are 2^{32} 32-bit ints. So, the probability that the hash function returns any individual int should be $\frac{1}{2^{32}}$.
- **Low collision rate:**
  The hash of two different inputs should usually be different.
  We want to *minimize collisions* as much as possible.
- **Low computational cost:**
  We will be computing the hash function a lot, so we need it to be very easy to compute.

Exercise: hash function for strings

Analyze these hash function implementations.

- $h(s) = 1$
- $h(s) = \sum_{i=0}^{|s|-1} s_i$
- $h(s) = 2^{s_0} \cdot 3^{s_1} \cdot 5^{s_2} \cdot 7^{s_3} \cdots$
- $h(s) = \sum_{i=0}^{|s|-1} 31^i \cdot s_i$

Announcements

- Written HW 1 due Wed, Jan 24
- Project 2 will be released tonight
  - Due Wed, Jan 31 at 11:30pm
  - Partner selection form due Thursday, Jan 25
  - Can work with same partner or a different one
- Midterm on Friday, Feb 2, in-class
  - Review session time and locations TBD
    (but probably Mon 29 and Tues 30?)
  - More details on Wednesday