CSE 373: Open addressing

Michael Lee
Friday, Jan 26, 2018

Warmup

Warmup:

With your neighbor, discuss and review:

- How do we implement **get**, **put**, and **remove** in a hash table using separate chaining?
- What about in a hash table using open addressing with linear probing?
- Compare and contrast your answers: what do we do the same? What do we do differently?

Warmup

In both implementations, for all three methods, we start by **finding the initial index to consider**:

```
index = key.hashCode() % array.length
```


Warmup

If we're using separate chaining, we then search/insert/delete from the bucket:

```
IDictionary<K, V> bucket = array[index]
bucket.get(key)  // or .put(...) or .remove(...)
```

...and resize when $\lambda \approx 1$.

(When exactly to resize is a tuneable parameter)

Warmup

If we're using linear probing, search until we find an array element where the key is equal to ours or until the array index is null:

```
while (array[index] != null
        && array[index].hashcode != key.hashCode()
        && !array[index].equals(key)) {
    index = (index + 1) % this.array.length
}
if (array[index] == null)
    // throw exception if implementing get
    // add new key-value pair if implementing put
else
    // return or set array[index]
```

How do we delete? (complicated, see section 04 handouts)

When do we resize?

Open addressing: linear probing

Strategy: Linear probing

If we collide, checking each next element until we find an open slot.

So, $h'(k, i) = (h(k) + i) \bmod T$, where T is the table size

```
i = 0
while (index in use)
    try (hash(key) + i) % array.length
    i += 1
```

Open addressing: linear probing

Assume internal capacity of 10, insert the following keys:

38, 19, 8, 109, 10

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9
8	109	10						38	19

What's the problem? Lots of keys close together: a "cluster". We ended up having to probe many slots!

Open addressing: linear probing

Primary clustering

When using linear probing, we sometimes end up with a long chain of occupied slots.

This problem is known as "primary clustering"

Happens when λ is large, or if we get unlucky

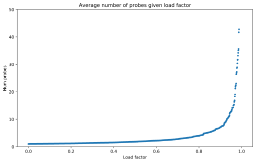
In linear probing, we expect to get $\mathcal{O}(\lg(n))$ size clusters.

Open addressing: linear probing

Questions:

- When is performance good? When is it bad?
  Runtime is bad when table is nearly full.
  Runtime is also bad when we hit a "cluster"
- What is the maximum load factor?
  Load factor is at most $\lambda = 1.0$!
- When do we resize?

Open addressing: linear probing



Punchline: clustering can be potentially bad, but in practice, it tends to be ok as long as λ is small

Open addressing: linear probing

Question: when do we resize?

Usually when $\lambda \approx \frac{1}{2}$

Nifty equations:

- Average number of probes for successful probe:
  $$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$$
- Average number of probes for unsuccessful probe:
  $$\frac{1}{2}\left(1 + \frac{1}{(1+\lambda)^2}\right)$$

*These equations aren't important to know

Open addressing: quadratic probing

Problem: We can still get unlucky/somebody can feed us a malicious series of inputs that causes several slowdown

Can we pick a different collision strategy that minimizes clustering?

Idea: Rather then probing linearly, probe quadratically!

Exercise: assume internal capacity of 10, insert the following:

89, 18, 49, 58, 79

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9
49		58	79					18	89

Open addressing: quadratic probing

Strategy: Quadratic probing

If we collide: $h'(k, i) = (h(k) + i^2) \bmod T$, where T is table size

```
i = 0
while (index in use)
    try (hash(key) + i * i) % array.length
    i += 1
```


Open addressing: quadratic probing

What problems are there?

Problem 1: If $\lambda \geq \frac{1}{2}$, quadratic probing may fail to find an empty slot: it can potentially loop forever!

Problem 2: Still can get clusters (though not as badly)

Open addressing: quadratic probing

Secondary clustering

When using quadratic probing, we sometimes need to probe a sequence of table cells (that are not necessary next to each other). This problem is known as "secondary clustering".

Ex: inserting 19, 39, 29, 9:

0	1	2	3	4	5	6	7	8	9
39			29					9	19

Secondary clustering can also be bad, but is generally milder then primary clustering

Recap

Note: let $s = h(k)$

- **Linear probing:**
  $s+0$, $s+1$, $s+2$, $s+3$, $s+4$, ...
  Basic pattern: try $h'(k, i) = (h(k) + i) \bmod T$
- **Quadratic probing:** $s+0$, $s+1$, $s+2^2$, $s+3^2$, $s+4^2$, ...
  Basic pattern: try $h'(k, i) = (h(k) + i^2) \bmod T$

Observation: For both probing strategies, there are just $\mathcal{O}(T)$ different "probe sequences" – distinct ways we can probe the array.

Idea: Can we increase the number of distinct probe sequences to decrease odds of collision?

Open addressing: double-hashing

Strategy: Double hashing

Idea: With linear and quadratic probing, we jump by the same increments. Can we try jumping in a different way per each key?

Use a second hash function!

Let $s = h(k)$, let $j = g(k)$:

$s + 0j$, $s + 1j$, $s + 2j$, $s + 3j$, $s + 4j$, ...

Basic pattern: try $h'(k, i) = (h(k) + i \cdot g(k)) \bmod T$

In pseudocode:

```
i = 0
while (index in use)
    try (hash(key) + i * jump_hash(key)) % array.length
    i += 1
```


Open addressing: double-hashing

Only effective if $g(k)$ returns a value that's *relatively prime* to the table size.

Ways we can do this:

- If T is a power of two, make $g(k)$ return any odd integer
- If T is a prime, make $g(k)$ return any smaller, non-zero integer (e.g. $g(k) = 1 + (k \bmod (T-1))$)

Open addressing: double-hashing

How many different probe sequences are there?

There are T different starting positions, $T - 1$ different jump intervals (since we can't jump by 0), so there are $\mathcal{O}\left(T^2\right)$ different probe sequences

Result: in practice, double-hashing is very effective and commonly used "in the wild".

Summary

So, what strategy is best? Separate chaining? Open addressing?

No obvious answer: both implementations are common.

Separate chaining:

- Don't have to worry about clustering
- Potentially more "compact" (λ can be higher)

Open addressing:

- Managing clustering can be tricky
- Less compact (we typically keep $\lambda < \frac{1}{2}$)
- Array lookups tend to be a constant factor faster then traversing pointers

Applications of hash functions

Can we use hash functions for more then just dictionaries?

Yes!

Lots of possible applications, ranging from cryptography to biology.

Important: Depending on the application, we might want our hash function to have different properties.

Applications of hash functions

How would you implement the following using hash functions?

For each application, also discuss what properties you want your hash function to have.

- Suppose we're sending a message over the internet. This message might become mildly corrupted. How can we detect if corruption probably occurred?
- Suppose you have many fragments of DNA and want to see where they appears in a (significantly longer) segment of DNA. How can we do this efficiently?

Applications of hash functions

Same question as before:

- Suppose you're designing an video uploading site and want to detect if somebody is uploading a pirated movie. A naive way to do this is to check if the movie is byte-for-byte identical to some movie. How can we do this more efficiently?
- Suppose you're designing a website with a user login system. Directly storing your user's passwords is dangerous – what if they get stolen? How can you store password in a safe way so that even if they're stolen, the passwords aren't compromised?

Applications of hash functions

Same question as before:

- You are trying to build an image sharing site. Users upload many images, and you need to assign each image some unique ID. How might you do this?
- Suppose we have a long series of financial transactions stored on some (potentially untrustworthy) computer. Somebody claims they made a specific transaction several months ago. Can you design a system that lets you audit and determine if they're lying or not? Assume you have access to just the very latest transaction, obtained from a different trustworthy source.