Hash Tables II

Data Structures and Parallelism

Announcements

Exercise 4 due TODAY at noon.

We'll have it graded in 24 hours or so.

Section is midterm review.

Robbie will have an extra office hour Thursday at 1-2 (in CSE 214).

No office hour on Friday (we'll be grading your exam)

Staff will stop answering Piazza questions at 9 PM on Thursday.

- Until after the midterm.

Outline

Wrap up designing hash functions

Collision Resolution part II: Open Addressing

Reaching the Average Case

In general our keys might not be integers.

Given an arbitrary object type E, how do we get an array index?

Hash Function

17,423

`% TableSize`

Usually Object writer's responsibility

Usually HashTable writer's responsibility

Wanted to make our hashes as evenly distributed as possible.

Java Specific Notes

Every object in Java implements the `hashCode` method.

If you define a new Object, and want to use a hash table, you might want to override `hashCode`.

But if you do, you also need to override `equals`

Such that

If `a.equals(b)` then `a.hashCode() == b.hashCode()`

This is part of the contract. Other code makes this assumption!

What about the converse?

Can't require it, but you should try to make it true as often as possible.

Generally Purpose hashCode()

```
int result = 17; // start at a prime

foreach field f
    int fieldHashcode =
      boolean: (f ? 1: 0)
      byte, char, short, int: (int) f
      long: (int) (f ^ (f >>> 32))
      float: Float.floatToIntBits(f)
      double: Double.doubleToLongBits(f), then above
      Object: object.hashCode( )

      result = 31 * result + fieldHashcode;
return result;
```

Collision Resolution

Last time: Separate Chaining

when you have a collision, stuff everything into that spot

Using a data structure.

Today: Open Addressing

If the spot is full, go somewhere else.

Where?

How do we find the elements later?

Linear Probing

First idea: linear probing

h(key) % TableSize full?

Try (h(key) + 1) % TableSize.

Also full? (h(key) + 2) % TableSize.

Also full? (h(key) + 3) % TableSize.

Also full? (h(key) + 4) % TableSize.

…

Example

Insert the hashes: 38, 19, 8, 109, 10 into an empty hash table of size 10.

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

How Does Delete Work?

Just find the key and remove it, what's the problem?

How do we know if we should keep probing on a find?

Delete 109 and call find on 10.

If we delete something placed via probe we won't be able to tell!

If you're using open addressing, you have to use lazy deletion.

How Long Does Insert Take?

If $\lambda < 1$ we'll find a spot eventually.

What's the average running time?

Uniform Hashing Assumption
for any pair of elements x,y the probability that h(x) = h(y) is $\frac{1}{TableSize}$

If find is unsuccessful: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

If find is successful: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$

We won't prove these (they're not even in the textbook)

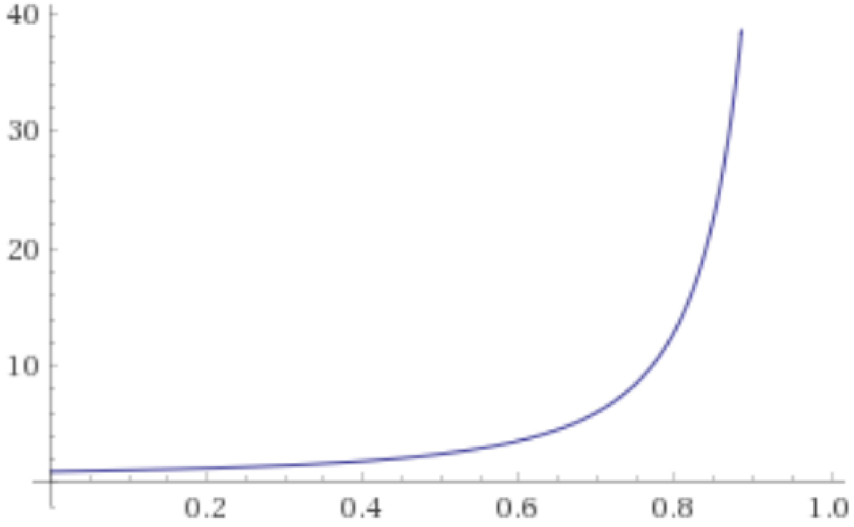
- Ask Robbie for references if you're really interested.

When to Resize

Input interpretation:

plot	$\frac{1}{2}\left(1 + \frac{1}{(1-x)^2}\right)$	$x = 0$ to 1

Plot:



We definitely want to resize before λ gets close to $\mathbf{1}$.
Taking $\lambda = \mathbf{0.5}$ as a resize point probably avoids the bad end of this curve.

Remember these are the average find times.
Even under UHA, the worst possible find is a bit worse than this *with high probability*.

Why are there so many probes?

The number of probes is a result of **primary clustering**

If a few consecutive spots are filled,

Hashing to any of those spots will make more consecutive filled spots.

Quadratic Probing

Want to avoid primary clustering.

If our spot is full, let's try to move far away relatively quickly.

h(key) % TableSize full?

Try (h(key) + 1) % TableSize.

Also full? (h(key) + 4) % TableSize.

Also full? (h(key) + 9) % TableSize.

Also full? (h(key) + 16) % TableSize.

…

Example

Insert: 89, 18, 49, 58, 79 into an empty hash table of size 10.

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

Then insert 76, 40, 48, 5, 55,47 into an empty hash table of size 7

0	1	2	3	4	5	6
48		5	55		40	76
				47		

Quadratic Probing: Proof

Claim: If $\lambda < \frac{1}{2}$, and TableSize is prime then quadratic probing will find an empty slot.

Quadratic Probing: Proof

Claim: If $\lambda < \frac{1}{2}$, and TableSize is prime then quadratic probing will find an empty slot.

Enough to show, first TableSize/2 probes are distinct.

For contradiction, suppose there exists some $i \neq j$ such that

$(h(x) + i^2) \bmod \text{TableSize} = (h(x) + j^2) \bmod \text{TableSize}$

$i^2 \bmod \text{TableSize} = j^2 \bmod \text{TableSize}$

$(i^2 - j^2) \bmod \text{TableSize} = 0$

Quadratic Probing: Proof

$(i^2 - j^2) \bmod \text{TableSize} = 0$

$(i + j)(i - j) \bmod \text{TableSize} = 0$

Thus TableSize divides $(i + j)(i - j)$

But TableSize is prime, so

TableSize divides $i + j$ or $i - j$

But that can't be true -- $i + j < \text{TableSize}$

Problems

Still have a fairly large amount of probes (we won't even try to do the analysis)

We don't have primary clustering, but we do have **secondary clustering**

If you initially hash to the same location, you follow the same set of probes.

Double Hashing

Instead of probing by a fixed value every time, probe by some new hash function!

h(key) % TableSize full?

Try (h(key) + g(key)) % TableSize.

Also full? (h(key) + 2*g(key)) % TableSize.

Also full? (h(key) + 3*g(key)) % TableSize.

Also full? (h(key) + 4*g(key)) % TableSize.

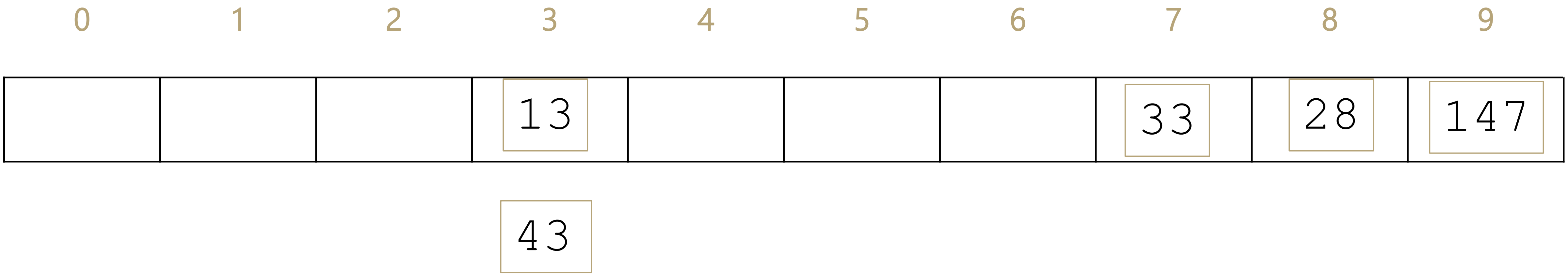
…

Example

Insert the following keys into a table of size 10 with the following hash functions: 13, 28, 33, 147, 43

Primary hash function h(key) = key mod TableSize

Second hash function g(key) = 1 + ((key / TableSize) mod (TableSize-1))



Running Times

Double Hashing will find lots of possible slots as long as g(key) and TableSize are relatively prime.

Under the uniform hashing assumption:

Expected probes for unsuccessful find: $\frac{1}{1-\lambda}$

Successful: $\frac{1}{1-\lambda}\ln\left(\frac{1}{1-\lambda}\right)$

Derivation beyond the scope of this course.

Ask Robbie for references if you want to learn more.

Summary

Separate Chaining

- Easy to implement
- Running times $O(1+\lambda)$

Open Addressing

- Uses less memory.
- Various schemes:
- Linear Probing – easiest, but need to resize most frequently
- Quadratic Probing – middle ground
- Double Hashing – need a whole new hash function, but low chance of clustering.

Which you use depends on your application and what you're worried about.

Other Topics

Perfect Hashing –

- if you have fewer than 2^{32} possible keys, have a one-to-one hash function

Hopscotch and cuckoo hashing (more complicated collision resolution strategies)

Other uses of hash functions:

Cryptographic hash functions

- Easy to compute, but hard to tell given hash what the input was.

Check-sums

Locality Sensitive Hashing

- Map “similar” items to similar hashes

Wrap Up

Hash tables have great behavior on average,

As long as we make assumptions about our data set.

But for every hash function, there's a set of keys you can insert to grind the hash table to a halt.

The number of keys is consistently larger than the number of ints.

An adversary can pick a set of values that all have the same hash.

Wrap Up

Can we avoid the terrible fate of our worst enemies forcing us to have $O(n)$ time dictionary operations?

If you have a lot of enemies, maybe use AVL trees.

But some hash table options:

Cryptographic hash functions – should be hard for adversary to find the collisions.

Randomized families of hash functions – have a bunch of hash functions, randomly choose a different one each time you start a hash table.

Done right – adversary won't be able to cause as many collisions.

Wrap Up

Hash Tables:

- Efficient find, insert, delete **on average, under some assumptions**
- Items not in sorted order
- Tons of real world uses
- ...and really popular in tech interview questions.

Need to pick a good hash function.

- Have someone else do this if possible.
- Balance getting a good distribution and speed of calculation.

Resizing:

- Always make the table size a prime number.
- λ determines when to resize, but depends on collision resolution strategy.