Lecture 9: Administrivia and Hash maps with Open Addressing

CSE 373: Data Structures and Algorithms

Administrivia (there's a lot today but not a full lecture's worth of content today)

- grade changes
- midterm format
- resources reminder/updates
- please fill out online logistics survey - we heard you!

Grades

- People are stressed about their grades, about learning, cheating, curved classes, etc. so we're going to make some changes to our grading policy, as we know this is a hard time for people:

- We're going to look at past grade data and figure out some groupings so that we can give you some grade guarantees (AKA 90% is at least guaranteed some GPA)

- AKA this class will not be curved since if everyone does well percentage-wise everyone can all do well GPA wise

- if not enough people are doing as well percentage-wise as we expected when we set the grade guarantees, don't worry -- we'll adjust to make sure people get at least as good grades as normal quarters

- we'll talk more about the guarantees next week (we haven't decided them yet)

~~Exams~~ (more like a weekend assessment now)

Extend turn around from 24 to 48 hours

- Out on Friday morning 8:30am pdt, due sunday morning 8:30am pdt
- NO LATE ASSESSMENTS ACCEPTED

~~Previously: 2 hour time window from when you start to when it must be turned in~~

- You just have the full 48 hours

reminder:

- multiple versions of assessment (we’re going to make a bunch)

- should be done individually (you can do well solo, we believe in you)

- if other people do cheat (plz no), shouldn’t affect you bc of grade guarantees

- still done through canvas

What if something goes wrong and I can’t submit or…?

- \- during Saturday we’ll have a zoom call you can reach us at that we’ll try to just be in there chilling. Link TBD
- \- if you have bad internet connection and are worried about if something goes wrong, fill out this form and we can try to figure out another alternative method of communication.
  - - https://forms.gle/tj626pRZVKWQ9fkx6

weekend assessment topics list

Things that could be on the exam

- Code modeling (think lecture, exercise, section problems)
  - Code → big theta runtime
  - Case analysis
  - Recursive code (code → recurrence, master theorem)
- Asymptotic analysis (think lecture, section, exercise problems)
  - Function -> big o/big theta/big omega
  - Bounds for a given graph
- Tree method (think section problems)
  - Just till summation, we won't ask you to simplify past that
- AVL/BST invariants
  - is this a BST/AVL?
- Hashing
  - Separate chaining mechanisms
  - Open addressing mechanisms

Things not on the exams:

- Simplifying summations
- AVL tree rotations
- Writing code
- Programming projects knowledge (but prepare for this on the final)
- Free response paragraph/sentence questions (we won't ask you to explain best/worst case in English)

Practice run through / resources

- most of the resources for the exam we've already released

- find section/lecture problems that are relevant to the topics list and do those

- check out the not-graded canvas quiz we're going to publish later today that'll show you a rough idea of the format / how all the math stuff will work

Resources updates/reminders

- optional pre-lecture readings (sorry they might not line up perfectly, from different quarters)
  - going to post these tonight / over the weekend for next week
- draft slides which may have significant changes by actual lecture time, but will cover same topics
  - going to post these tonight / over the weekend for next week (same as above)
- review videos
  - see on calendar!
- night time section (6:30PM Thursdays) / evening office hours Monday/Tuesday (9pm PDT)
- Piazza/slack/discord – class size is an asset – we can help each other learn more effectively because we can share questions, knowledge, and tips :P

Other things that are useful to know

- no more warm ups so we have more time for lecture - instead we will make sure the post lecture review questions are there for you
  - (we know we're behind on the solutions for Monday/posting Wednesday – Kasey and Zach have been managing this but busy with other logistical stuff and it's fallen off – going to re-delegate to amazing TAs to help out so they should be up more consistently in the future.We'll post all of the remaining post-lecture questions for MWF this week later today. Thank you so much for your patience.)
- we will do a better job of pointing out important part of slides, but they will always have more content than we can speak to so they can be a "living textbook"

We'll respond to more ideas/suggestions/thoughts in more time (keep an eye on Piazza), but are also waiting for more responses to come in. Thanks to everyone so far, we're reading all of them and are seriously considering all the feedback and ideas. We appreciate your effort to make this a better experience for all of us.

What about non integer keys?

Hash function definition
A **hash function** is any <u>function</u> that can be used to map <u>data</u> of arbitrary size to fixed-size values.

Let’s use define another hash function to change stuff like Strings into ints!

Best practices for designing hash functions:

Avoid collisions

- The more collisions, the further we move away from O(1+λ)
- Produce a wide range of indices, and distribute evenly over them

Low computational costs

- Hash function is called every time we want to interact with the data

(Before we % by length, we have to convert the data into an int)

Implementation 1: Simple aspect of values

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

Pro: super fast
Con: lots of collisions!

Implementation 2: More aspects of value

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

Pro: still really fast
Con: some collisions

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());
}
```

Pro: few collisions
Con: slow, gigantic integers

Good Hashing

The hash function of a HashDictionary gets called a LOT:

- When first inserting something into the map
- When checking if a key is already in the map
- When resizing and redistributing all values into new structure

This is why it is so important to have a “good” hash function. A good hash function is:

1. Deterministic – same input should generate the same output
2. Efficiency - it should take a reasonable amount o time
3. Uniformity – inputs should be spread “evenly” over output range

```
public int hashFn(String s) {
    return random.nextInt()
}
```

NOT deterministic

```
public int hashFn(String s) {
   if (s.length() % 2 == 0) {
      if (s.length(). % 2 == 0) {
         return 17;
      } else {
         return 43;
      }
   }
}
```

NOT nicely spread out

```
public int hashFn(String s) {
   int retVal = 0;
   for (int I = 0; I < s.length(); i++) {
      for (int j = 0; j < s.length(); j++) {
         retVal += helperFun(s, I, j);
      }
   }
   return retVal;
}
```

NOT efficient

Java's hashCode (relevant for project)

- Luckily, most of these design decisions have been made for us by smart people. All objects in java come with a `hashCode()` method that does some magic (see previous slide for the not-magic version) to turn any object type (like String, ArrayList, Point, Scanner) into an integer. These hashCodes are designed to distribute pretty evenly / not have lots of collisions, so we use them as the starting point for determining the bucket index.

- high level steps to figure out which bucket a key goes into
  - call the key.hashCode() to get an int representation of the object
  - % by the array table length to convert it to a valid index for your hash map

Review: Handling Collisions

Solution 1: Chaining

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

Operation		Array w/ indices as keys
put(key,value)	best	$\Theta(1)$
	In-practice	$\Theta(\lambda) = \Theta(1)$
	worst	$\Theta(n)$
get(key)	best	$\Theta(1)$
	in-practice	$\Theta(\lambda) = \Theta(1)$
	worst	$\Theta(n)$
remove(key)	best	$\Theta(1)$
	In-practice	$\Theta(\lambda) = \Theta(1)$
	worst	$\Theta(n)$

Average Case:
Depends on average number of elements per chain

Load Factor λ

If n is the total number of key-value pairs

Let c be the capacity of array

Load Factor $\lambda = \frac{n}{c}$

Handling Collisions

Solution 2: Open Addressing

Resolves collisions by choosing a different location to store a value if natural choice is already full.

Type 1: Linear Probing

If there is a collision, keep checking the next element until we find an open spot.

```
int findFinalLocation(Key s)
    int naturalHash = this.getHash(s);
    int index = natrualHash % TableSize;
    while (index in use) {
        i++;
        index = (naturalHash + i) % TableSize;
    }
    return index;
```

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions

1, 5, 11, 7, 12, 17, 6, 25

0	1	2	3	4	5	6	7	8	9
	11	12			25	6	17		

Linear Probing

3 Minutes

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

8

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

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering

When probing causes long chains of occupied slots within a hash table

Runtime

When is runtime good?

When we hit an empty slot

- (or an empty slot is a very short distance away)

When is runtime bad?

When we hit a “cluster”

Maximum Load Factor?

λ at most 1.0

When do we resize the array?

$\lambda \approx ½$ is a good rule of thumb

Can we do better?

Clusters are caused by picking new space near natural index

Solution 2: Open Addressing

Type 2: Quadratic Probing

Instead of checking i past the original location, check i^2 from the original location.

```
int findFinalLocation(Key s)
    int naturalHash = this.getHash(s);
    int index = natrualHash % TableSize;
    while (index in use) {
        i++;
        index = (naturalHash + i*i) % TableSize;
    }
    return index;
```

Quadratic Probing

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions

89, 18, 49, 58, 79, 27

0	1	2	3	4	5	6	7	8	9
		58	79				27	18	89 49

(49 % 10 + 0 * 0) % 10 = 9
(49 % 10 + 1 * 1) % 10 = 0

(58 % 10 + 0 * 0) % 10 = 8
(58 % 10 + 1 * 1) % 10 = 9
(58 % 10 + 2 * 2) % 10 = 2

(79 % 10 + 0 * 0) % 10 = 9
(79 % 10 + 1 * 1) % 10 = 0
(79 % 10 + 2 * 2) % 10 = 3

Now try to insert 9.

Uh-oh

Problems:

If $\lambda \geq \frac{1}{2}$ we might never find an empty spot

Infinite loop!

Can still get clusters

Quadratic Probing

There were empty spots. What gives?

Quadratic probing is not guaranteed to check every possible spot in the hash table.

The following is true:

If the table size is a prime number p, then the first $p/2$ probes check distinct indices.

Notice we have to assume p is prime to get that guarantee.

3 Minutes

Secondary Clustering

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
19, 39, 29, 9

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

Secondary Clustering
When using quadratic probing sometimes need to probe the same sequence of table cells, not necessarily next to one another

Probing

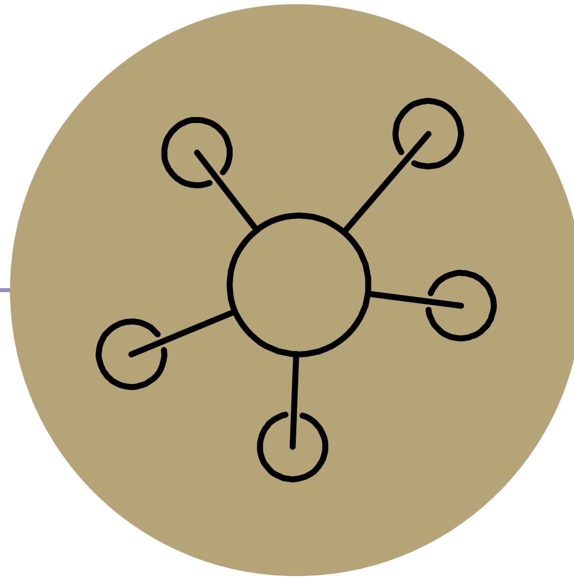
- $h(k)$ = the natural hash
- $h'(k, i)$ = resulting hash after probing
- i = iteration of the probe
- T = table size

Linear Probing:

$$h'(k, i) = (h(k) + i) \% T$$

Quadratic Probing

$$h'(k, i) = (h(k) + i^2) \% T$$



Questions

Topics Covered:

- Writing good hash functions
- Open addressing to resolve collisions:
  - Linear probing
  - Quadratic probing

Double Hashing

Probing causes us to check the same indices over and over- can we check different ones instead?

Use a second hash function!

$h'(k, i) = (h(k) + i * g(k)) \% T$

<- Most effective if g(k) returns value relatively prime to table size

```
int findFinalLocation(Key s)
     int naturalHash = this.getHash(s);
     int index = natrualHash % TableSize;
     while (index in use) {
           i++;
           index = (naturalHash + i*jumpHash(s)) % TableSize;
     }
     return index;
```

Second Hash Function

Effective if g(k) returns a value that is *relatively prime* to table size

- If T is a power of 2, make g(k) return an odd integer
- If T is a prime, make g(k) return anything except a multiple of the TableSize

Resizing: Open Addressing

How do we resize? Same as separate chaining

- Remake the table
- Evaluate the hash function over again.
- Re-insert.

When to resize?

- Depending on our load factor λ AND our probing strategy.
- Hard Maximums:
  - If $\lambda = 1$, `put` with a new key fails for linear probing.
  - If $\lambda > 1/2$ `put` with a new key **might** fail for quadratic probing, even with a prime `tableSize`
    - And it might fail earlier with a non-prime size.
  - If $\lambda = 1$ `put` with a new key fails for double hashing
    - And it might fail earlier if the second hash isn't relatively prime with the `tableSize`

Running Times

What are the running times for:

`insert`

Best: $\Theta(1)$

Worst: $\Theta(n)$ (we have to make sure the key isn't already in the bucket.)

`find`

Best: $\Theta(1)$

Worst: Θ $O(n)$

`delete`

Best: $\Theta(1)$

Worst: $\Theta(n)$

In-Practice

For open addressing:

We'll **assume** you've set λ appropriately, and that all the operations are $\Theta(1)$.

The actual dependence on λ is complicated – see the textbook (or ask on piazza)

And the explanations are well-beyond the scope of this course.

Summary

1. Pick a hash function to:
   - Avoid collisions
   - Uniformly distribute data
   - Reduce hash computational costs

2. Pick a collision strategy
   - Chaining
     - LinkedList
     - AVL Tree
   - Probing
     - Linear
     - Quadratic
     - Double Hashing

No clustering
Potentially more “compact” (λ can be higher)

Managing clustering can be tricky
Less compact (keep $\lambda < ½$)
Array lookups tend to be a constant factor faster than traversing pointers

Summary

Separate Chaining

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

Open Addressing

- Uses less memory (usually).
- Various schemes:
- Linear Probing – easiest, but lots of clusters
- Quadratic Probing – middle ground, but need to be more careful about λ.
- 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.

Extra optimizations

Idea 1: **Take in better keys**

- Really up to your client, but if you can control them, do!

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 buckets get large

Idea 3: **Modify the array's internal capacity**

- When load factor gets too high, resize array
  - Increase array size to next prime number that's roughly double the array size
  - Let the client fine-tune the λ that causes you to resize

Other Hashing Applications

We use it for hash tables but there are lots of uses! Hashing is a really good way of taking arbitrary data and creating a succinct and unique summary of data.

Caching

- you've downloaded a large video file, You want to know if a new version is available, Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.

File Verification / Error Checking:

- compare the hash of a file instead of the file itself
- Find similar substrings in a large collection of strings – detecting plagiarism

Cryptography

Hashing also "hides" the data by translating it, this can be used for security

- For password verification: Storing passwords in plaintext is insecure. So your passwords are stored as a hash
- Digital signatures

Fingerprinting

git hashes ("identification")

- That crazy number that is attached to each of your commits
- SHA-1 hash incorporates the contents of your change, the name of the files and the lines of the files you changes

Ad Tracking

- track who has seen an ad if they saw it on a different device (if they saw it on their phone don't want to show it on their laptop)
- https://panopticlick.eff.org will show you what is being hashed about you

YouTube Content ID

- Do two files contain the same thing? Copyright infringement
- Change the files a bit!