Please log onto PollEv.com/champk to answer the daily lecture participation question

Lecture 10: Hash Collision Resolutions Continued

CSE 373: Data Structures and Algorithms

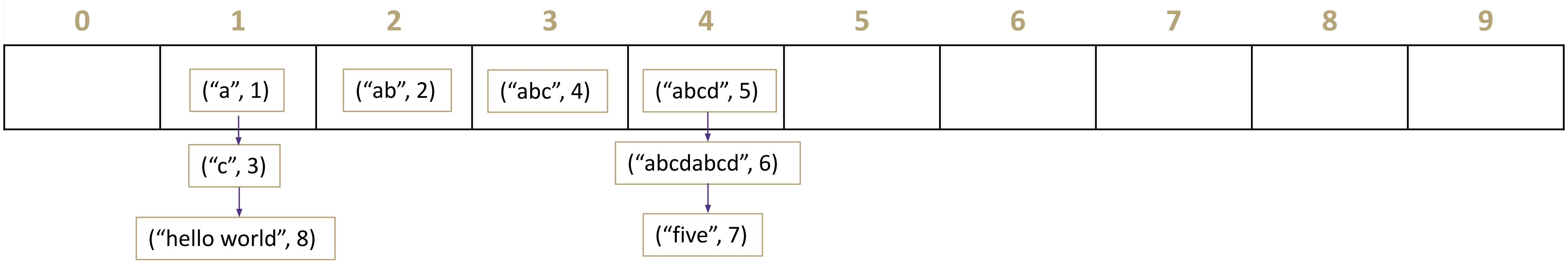
Practice

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:

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

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

(“a”, 1) (“ab”, 2) (“c”, 3) (“abc”, 4) (“abcd”, 5) (“abcdabcd”, 6) (“five”, 7) (“hello world”, 8)



Announcements

Exercise 2 due this Friday (April 22nd) at 11:59 pm

Project 2 due next wednesday April 27th

Best practices for an nice distribution of keys recap

1. Resize when lambda (number of elements / number of buckets) increases up to 1
   a. Fewer collisions when there are more buckets to spread data across

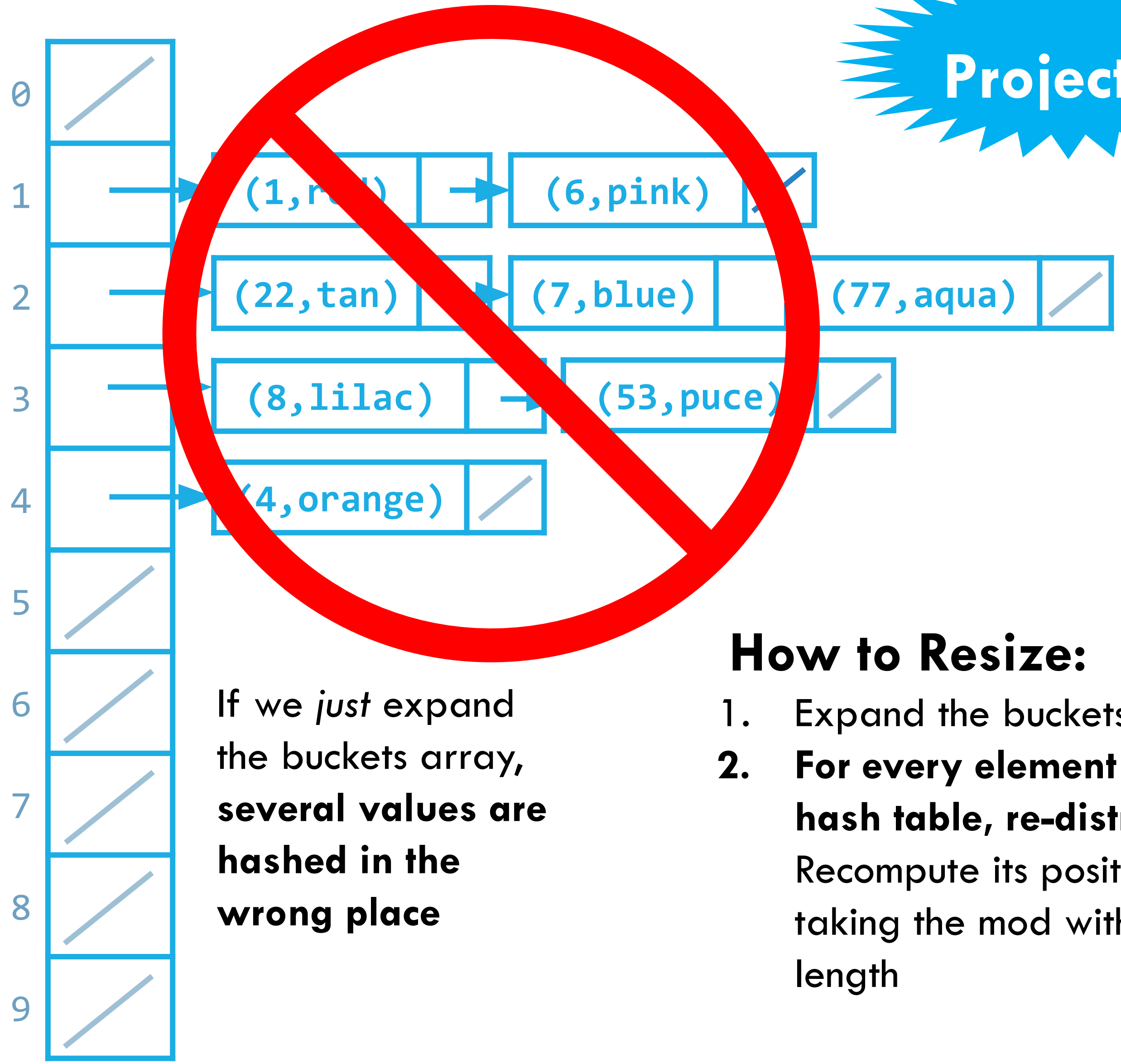
2. When you resize, you can choose a the table length that will help reduce collisions if you multiply the array length by 2 and then choose the nearest prime number
   a. When mod'ing by table size, if table size is a prime number less likely to cause collisions because dividing by prime number has fewer common denominator
   b. Prime Numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61
   c. Table start at 11 –> 23 –> 47 –> 97

3. Design the hashCode of your keys to be somewhat complex and lead to a distribution of different output numbers
   a. Include enough identifying features of what you are hashing to get unique hash codes
   b. Balance the runtime of computation of hash code with the runtime of dealing with collisions

Resizing

Don't forget to re-distribute your keys!

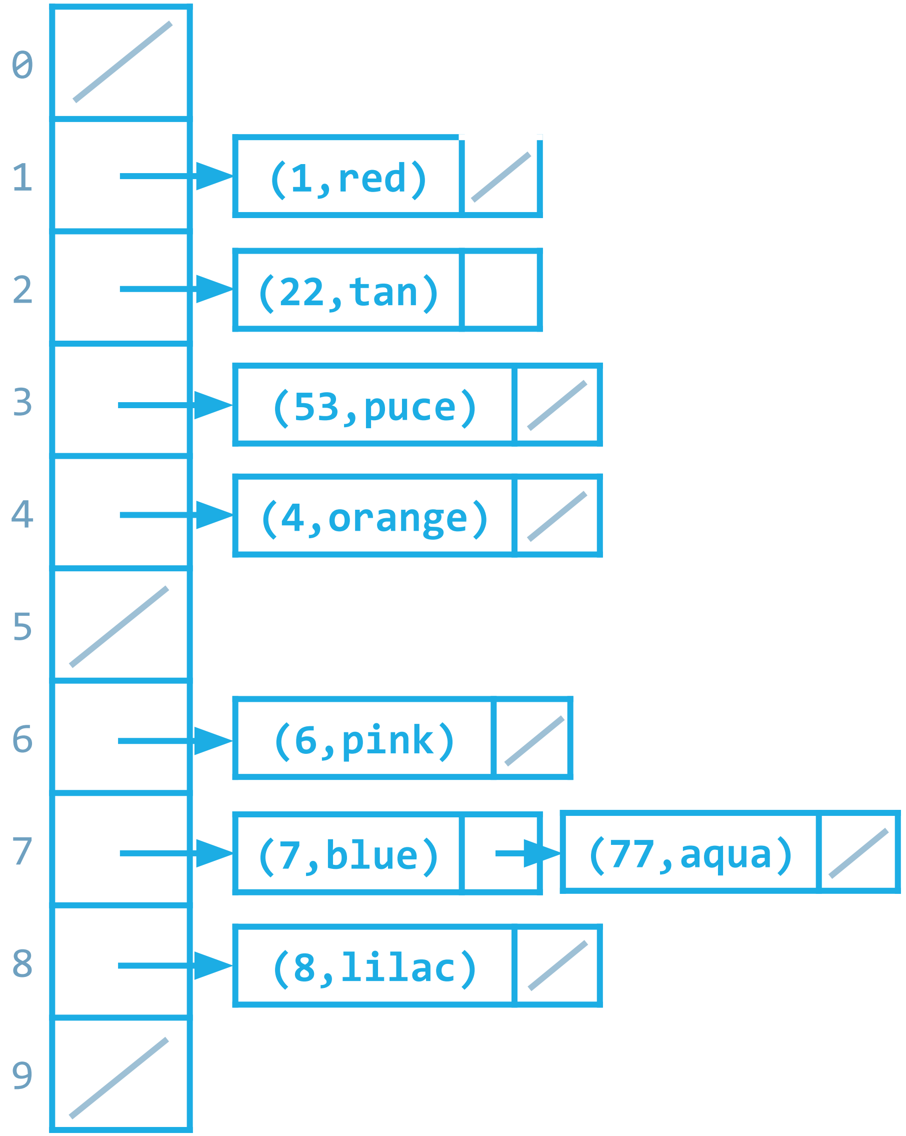
Project 2



If we *just* expand the buckets array, **several values are hashed in the wrong place**

How to Resize:

1. Expand the buckets array
2. **For every element in the old hash table, re-distribute!** Recompute its position by taking the mod with the new length



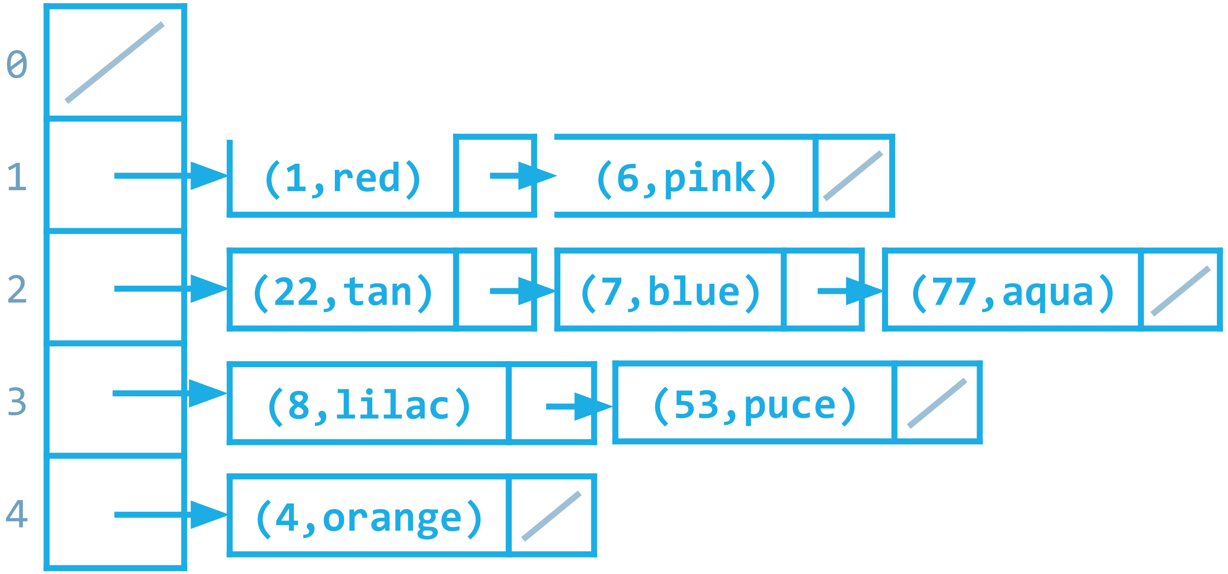
When to Resize?

In ArrayList, we were forced to resize when we ran out of room

- In SeparateChainingHashMap, never *forced* to resize, but we want to make sure the buckets don't get too long for good runtime

How do we quantify "too full"?

- Look at the average bucket size: number of elements / number of buckets



LOAD FACTOR λ

n: total number of key/value pairs

c: capacity of the array (# of buckets)

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

When to Resize?

In ArrayList, we were forced to resize when we ran out of room

- In SeparateChainingHashMap, never *forced* to resize, but we want to make sure the buckets don't get too long for good runtime

How do we quantify "too full"?

- Look at the average bucket size: number of elements / number of buckets

- If we resize when λ hits some *constant* value like 1:
  - We expect to see 1 element per bucket: **constant runtime**!
  - If we double the capacity each time, the expensive resize operation becomes less and less frequent

LOAD FACTOR λ

n: total number of key/value pairs

c: capacity of the array (# of buckets)

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

Java and Hash Functions

Object class includes default functionality:
- equals
- hashCode

If you want to implement your own hashCode you should:
- Override BOTH hashCode() and equals()

If a.equals(b) is true then a.hashCode() == b.hashCode() **MUST** also be true

That requirement is part of the Object interface.
Other people's code will assume you've followed this rule.

Java's HashMap (and HashSet) will assume you follow these rules and conventions for your custom objects if you want to use your custom objects as keys.

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()
}
```

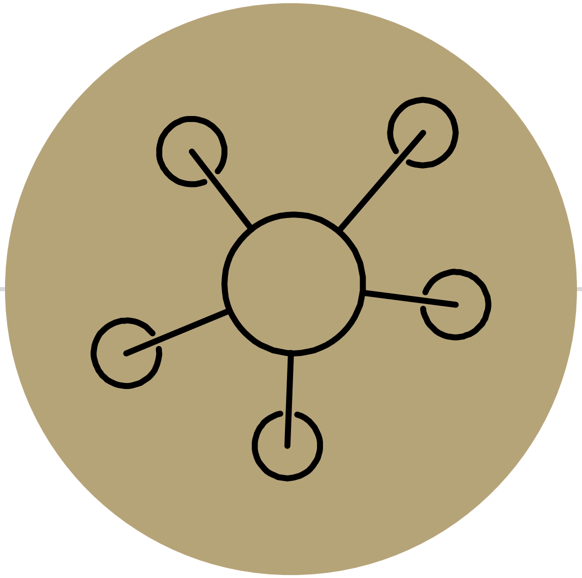
NOT deterministic

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

NOT efficient

```
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 uniform



Questions?

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	12		25	6	17	17	25

Linear Probing

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

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

λ ≈ 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
49		58	79				27	18	89

(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

Uh-oh :(

Problems:

If λ≥ ½ 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

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

(19 % 10 + 0 * 0) % 10 = 9

(39 % 10 + 0 * 0) % 10 = 9
(39 % 10 + 1 * 1) % 10 = 0

(29 % 10 + 0 * 0) % 10 = 9
(29 % 10 + 1 * 1) % 10 = 0
(29 % 10 + 2 * 2) % 10 = 3

(9 % 10 + 0 * 0) % 10 = 9
(9 % 10 + 1 * 1) % 10 = 0
(9 % 10 + 2 * 2) % 10 = 3
(9 % 10 + 3 * 3) % 10 = 8

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

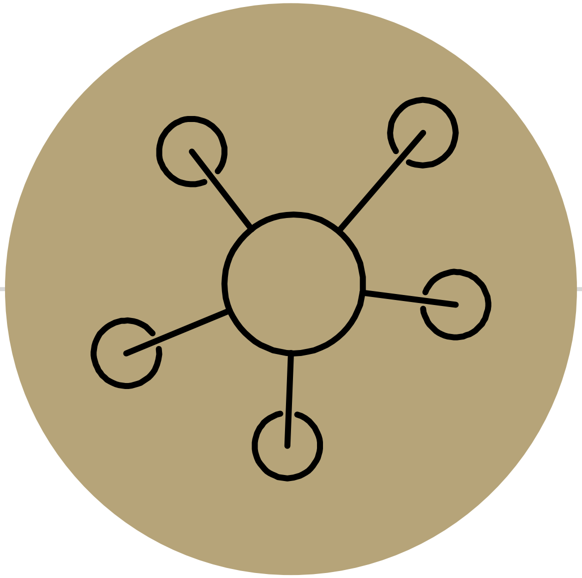
- $h(k)$ = the natural hash
- $h'(k, i)$ = resulting hash after probing
- i = iteration of the probe
- T = table size

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: $O(1)$

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

`find`

Best: $O(1)$

Worst: $O(n)$

`delete`

Best: $O(1)$
Worst: $O(n)$

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 me in office hours)

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 $\lambda < \frac{1}{2}$)
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

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!