Answer the Warm Up:
PollEv.com/champk

Lecture 08: Hash Collision Resolutions

CSE 373: Data Structures and Algorithms

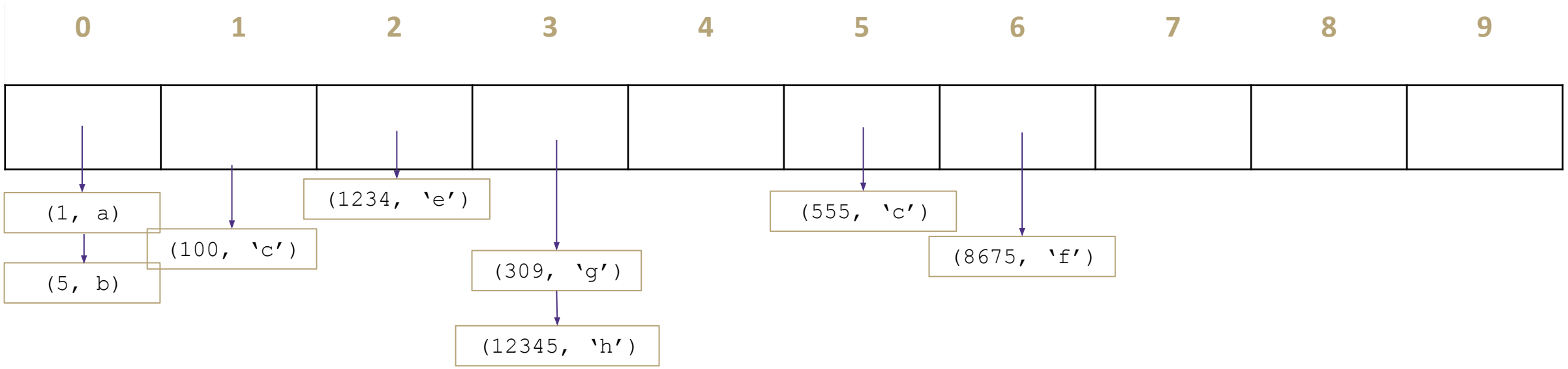
Warm Up

Answer the Warm Up:
PollEv.com/champk

- Consider an `IntegerDictionary` using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a `LinkedList` where we append new key-value pairs to the end.
- Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

```
int hashCode (int key) {
    return mapToValidIndex((key / 100));
}
```

`(1,'a')(5,'b')(100,'c')(555,'d')(1234,'e')(8675,'f')(309,'g')(12345, 'h')`



Reducing Collisions – Resizing

- Data structures like `ArrayMap` or `ArrayList` or `ArrayStack` must resize when full to make space for more elements
- Since `SeparateChainingHashMap` buckets can grow to any size, you are never *forced* to resize

What if we used the same array with 10 buckets, but continued to add data until we had 100 entries?

- What would this do to the runtime of `get(key)`?
  - assuming even distribution of hashCodes: # of pairs / array.length = $O(n/capacity) \in O(n)$

Reducing Collisions – Resizing

If `array.length` is fixed as n increases then

`get(key)` = $O(n/\text{array.length}) \in O(n)$

BUT if you resize the array when when `n / array.length = 1` then

`get(key)` = $O(n/\text{array.length}) \in O(1)$

- This assumes even distribution of `hashCodes` across new array
- To redistribute keys you must re-hash keys and find their new bucket based on the new `array.length` after each re-size

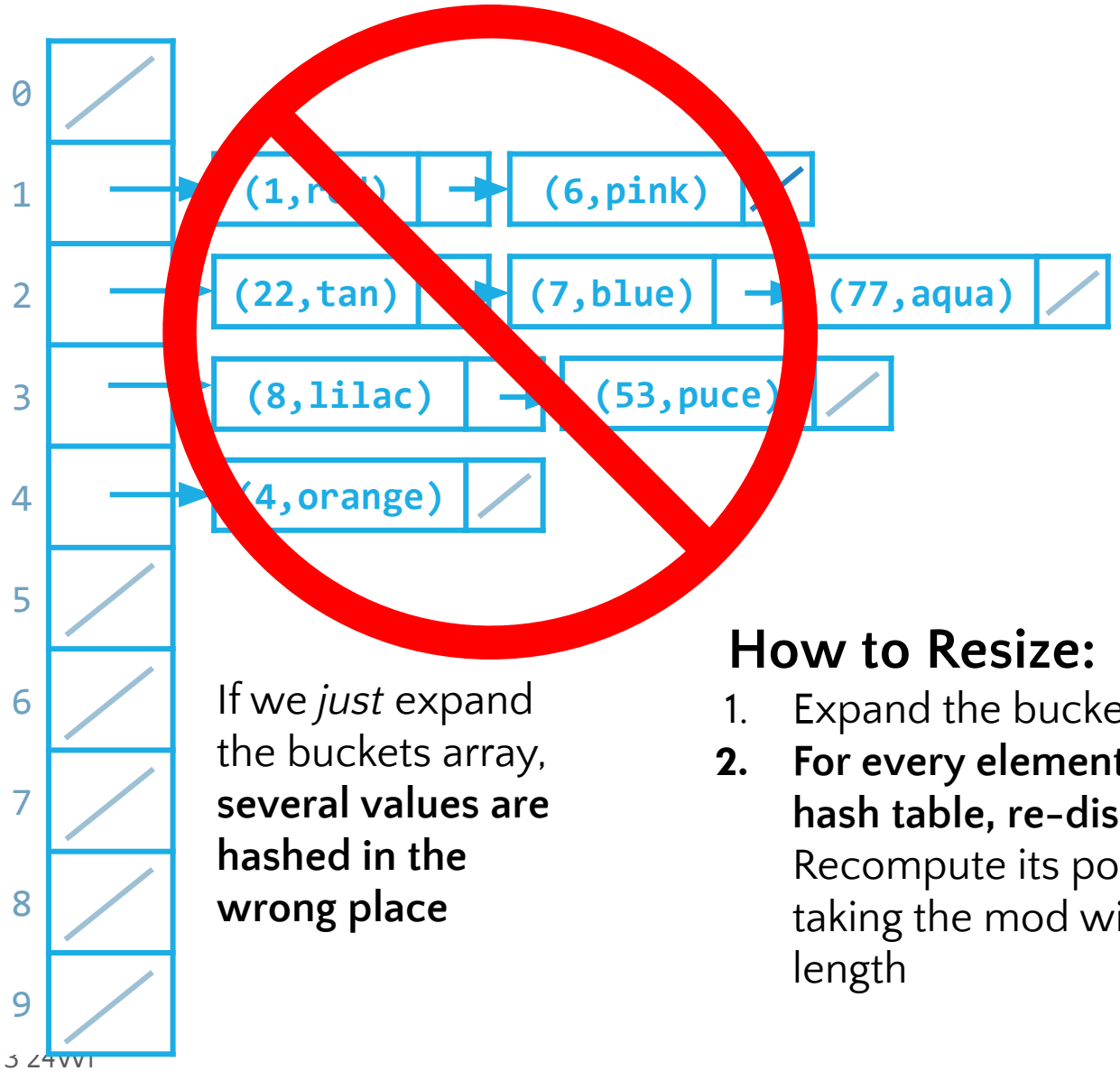
PRO TIP: When you resize, choose a table length that will help reduce collisions if you multiply the array length by 2 and then choose the nearest prime number

You must resize and re-hash for Project 2!

Resizing

Don't forget to re-hash your keys!

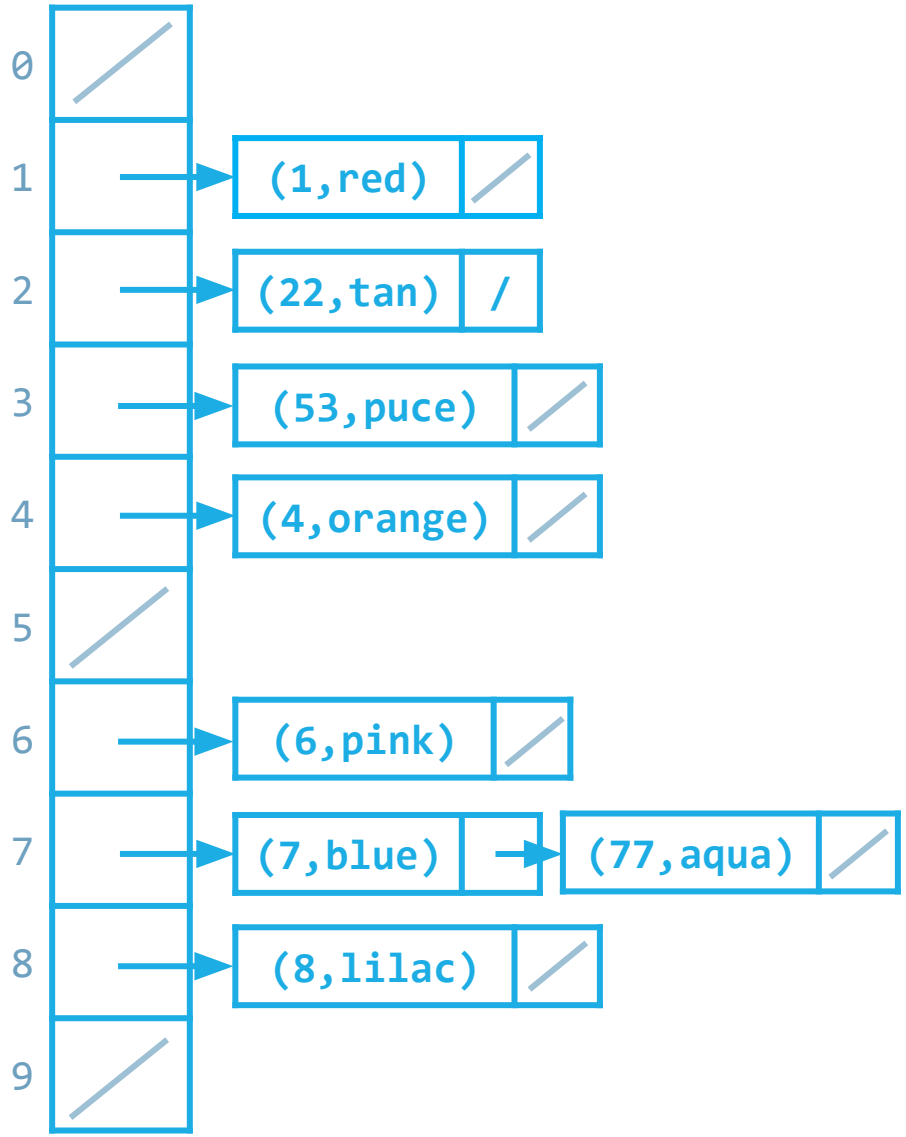
Project 2



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



What about non integer keys?

Hash function definition

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values.

Let's 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 $O(1+\lambda)$
- 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

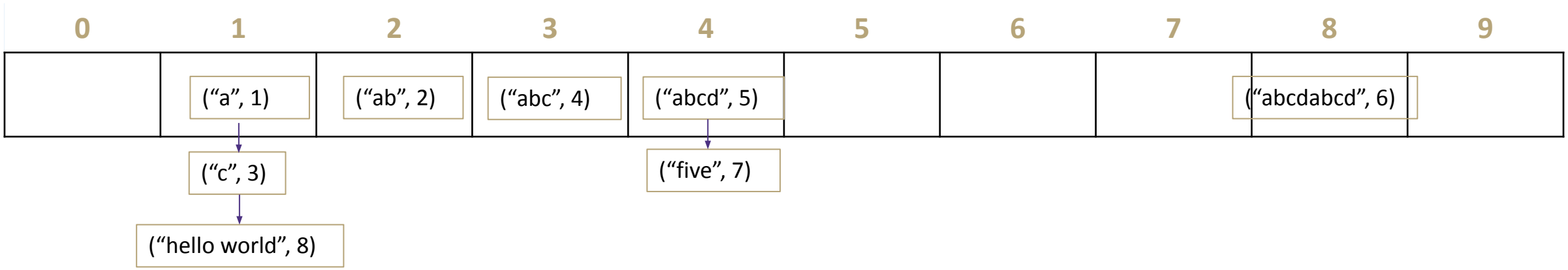
Practice

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)



hashCode()

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 HashMap 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:

- Deterministic – same input should generate the same output
- Uniformity – inputs should be spread "evenly" over output range
- Efficiency – it should take a reasonable amount of time

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

NOT deterministic

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

NOT uniform

```
public int hashCode(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

Practice

Which of the following two hashCode functions for a String will produce more collisions on average?

```
public int hashCode1() {
   Iterator<Character> iterator =
this.iterator();
   int result = 13;
   int i = 0;
   while (iterator.hasNext()) {
      result += iterator.next().hashCode() *
37^i;
      i++;
   }
   return result % 5;
}
```

```
public int hashCode2() {
   Iterator<Character> iterator = this.iterator();
   int result = 0;
   int i = 0;
   while (iterator.hasNext()) {
      result += iterator.next().hashCode();
      i++;
   }
   return i;
}
```

`hashCode1` will produce more collisions because it limits the range of possible values in the return statement. If that `%5` was removed than `hashCode2` would produce more collisions

Java's hashCode function

All Java Objects **must** include a hashCode function:

```
public int hashCode();
```

From official Oracle Java documentation:

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(`java.lang.Object`) method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Java and Hash Functions

- Object class includes default functionality:
  - `int equals(Object other)`
  - `int 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
  - This is how Java knows to replace the value associated with the key or to add a new key to the bucket

- 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.

Linear Probing

Quadratic Probing

Double Hashing

Summary

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.hashCode(s);
    int index = naturalHash % array.length;
    while (index in use) {
        i++;
        index = (naturalHash + i) % array.length;
    }
    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
	1	11	12		5	6	7	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
8	109	10						38	19

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 the natural index

Solution 2: Open Addressing (still)

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.hashCode(s);
    int index = naturalHash % array.length;
    while (index in use) {
        i++;
        index = (naturalHash + i*i) % array.length;
    }
    return index;
```

Linear Probing

Quadratic Probing

Double Hashing

Summary

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

Now try to insert 9.

Uh-oh

Problems:

If $\lambda \geq$ 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

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

Secondary Clustering

When using quadratic probing, you 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

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

Linear Probing

Quadratic Probing

Double Hashing

Summary

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;
```

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
  - If λ = 1, `put` with a new key fails for linear probing
  - If λ > 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 λ = 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`

Linear Probing

Quadratic Probing

Double Hashing

Summary

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

Java's HashMap Implementation

- default array capacity is 16
  - *Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high - Javadocs*
- resizes at load factor 0.75
  - *As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost - Javadocs*
- uses separate-chaining collision resolution
  - Initially uses LinkedLists as the buckets
  - After 8 collisions across the table at the next resize the buckets will be created as balanced trees to reduce runtime of possible worst case scenario - Javarevisited

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!

That’s all!