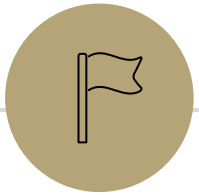
Lecture 7: Intro to Hashing

CSE 373: Data Structures and Algorithms



Review of Maps

Direct Access Map

Hash Functions

Hash Collisions

Dictionaries (aka Maps)

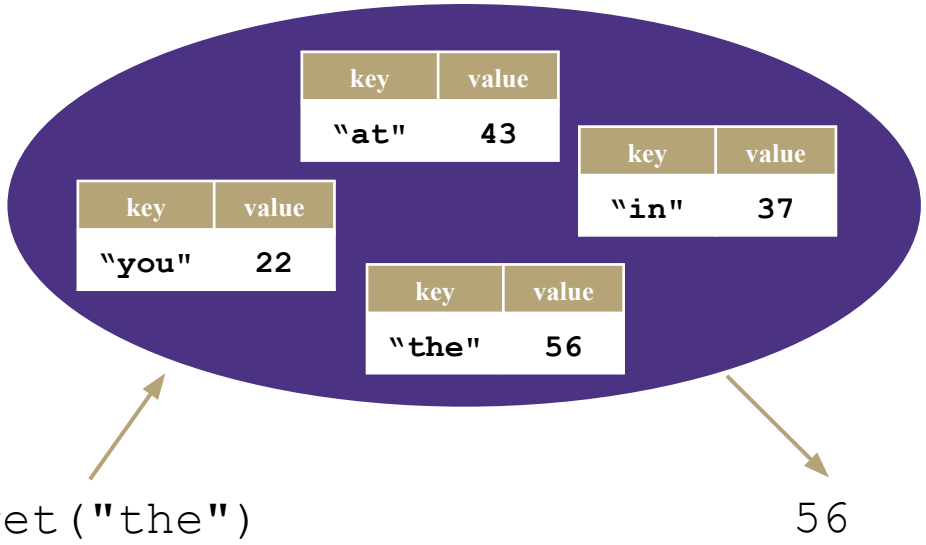
- Every Programmer's Best Friend
- You'll probably use one in almost every programming project.
  - Because it's hard to make a big project without needing one sooner or later.

```
// two types of Map implementations supposedly covered in CSE 143
Map<String, Integer> map1 = new HashMap<>();
Map<String, String> map2 =  new TreeMap<>();
```

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

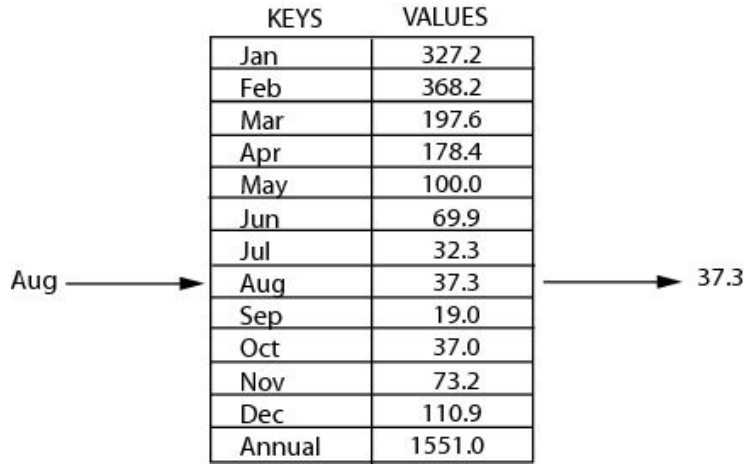
- a.k.a. "dictionary"



Dictionary ADT
state Set of items & keys Count of items
behavior put(key, item) add item to collection indexed with key get(key) return item associated with key containsKey(key) return if key already in use remove(key) remove item and associated key size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



Review: Implementing a Dictionary with an Array

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayDictionary<K, V>

state

```
Pair<K, V>[] data
```

behavior

```
put find key, overwrite value if there.
Otherwise create new pair, add to next
available spot, grow array if necessary
get scan all pairs looking for given key,
return associated item if found
containsKey scan all pairs, return if key
is found
remove scan all pairs, replace pair to be
removed with last pair in collection
size return count of items in dictionary
```

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b',97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis: if the key is the last one looked at / is not in the dictionary	
`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant
Big O Analysis: if the key is the first one looked at	
`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

Review: Implementing a Dictionary with Nodes

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

LinkedDictionary<K, V>

state

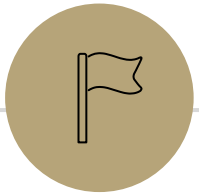
```
front
size
```

behavior

```
put if key is unused, create new with pair, add to front of list, else replace with new value
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, skip pair to be removed
size return count of items in dictionary
```

```
containsKey('c')
get('d')
put('b', 20)
```

Big O Analysis: if the key is the last one looked at / is not in the dictionary	
`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant
Big O Analysis: if the key is the first one looked at	
`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant



Review of Maps

Direct Access Map

Hash Functions

Hash Collisions

Can we do better?

Let's simplify the problem we're working with + combine it with some facts about arrays.

Problem Simplification: only worry about supporting integer keys

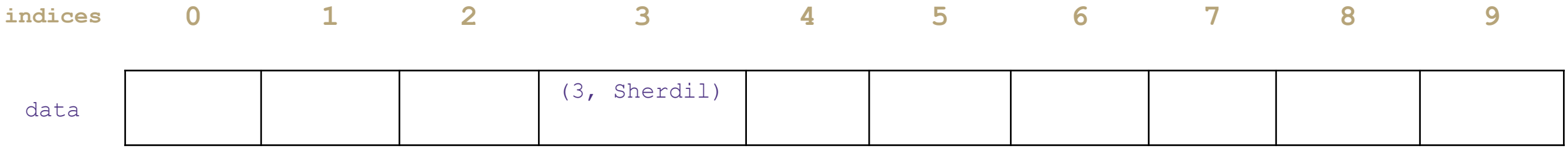
Array Facts: accessing (data[i]) or updating an element (data[i] = …) at a given index takes Theta(1) runtime.

If we store the Key-Value pairs at the data[key] then we don't have to do any looping to find it. For example consider 'containsKey' or 'get' - we can just jump directly to data[key] to figure out the return answer.

DirectAccessMap<Integer, V>

state
- Data[]
- size

behavior
- put put item at given index
- get get item at given index
- containsKey if data[] null at index, return false, return true otherwise
- remove nullify element at index
- size return count of items in dictionary




```
put(3, "Sherdil");
get(3);
```

Can we do better? – Direct Access Map impl.

```
public void put(int key, V value) {
    this.array[key] = value;
}

public boolean containsKey(int key) {
    return this.array[key] != null;
}

public V get(int key) {
    return this.array[key];
}

public void remove(int key) {
    this.array[key] = null;
}
```

DirectAccessMap<Integer, V>

state
- Data[]
- size

behavior
- put put item at given index
- get get item at given index
- containsKey if data[] null at index, return false, return true otherwise
- remove nullify element at index
- size return count of items in dictionary

Operation		Array w/ indices as keys
put(key,value)	best	$\Theta(1)$
	worst	$\Theta(1)$
get(key)	best	$\Theta(1)$
	worst	$\Theta(1)$
containsKey(key)	best	$\Theta(1)$
	worst	$\Theta(1)$

Direct Access Map tradeoffs:

Drawbacks

- Wasted space
  - What if we want to store two keys: 0 and 99999999? Our current setup would just be wasting all that array space in-between
- Only integer keys
  - Kind of annoying that we could only have this for ints, but **being able to quickly go from the key to the array index is super valuable because its array lookups are fast (constant time).** When we can just jump to the right position, we avoid the looping that ArrayMap/LinkedMap had to do where you might have to loop and look at every element. We'll keep this core idea of "knowing the index" and jumping there right away for all the versions of the dictionaries we talk about today

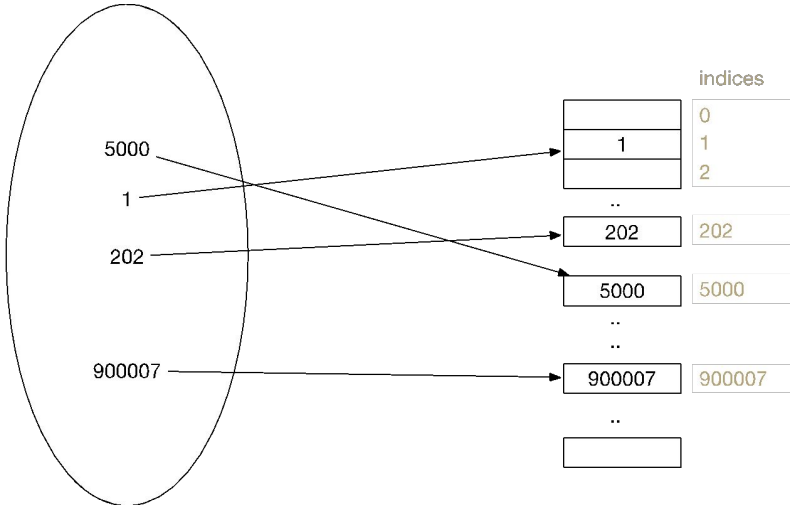
Benefits

- Super fast: $\Theta(1)$ runtime for everything

Can we do this for any integer?

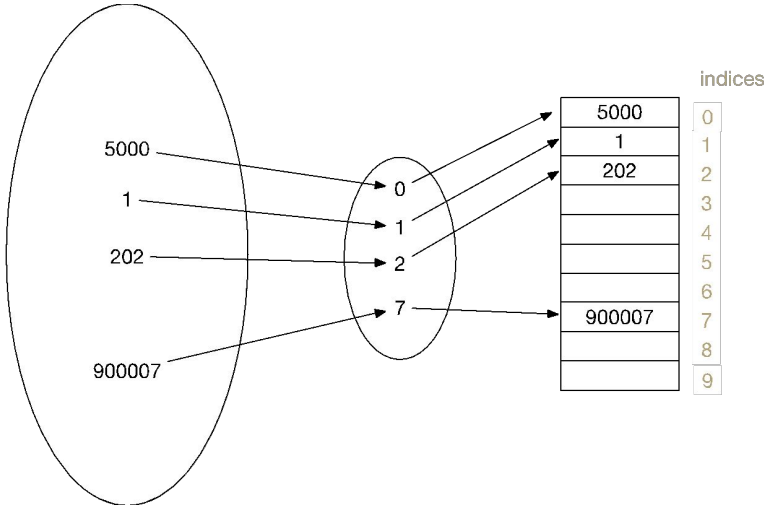
Idea 1:

- Create a GIANT array with every possible integer as an index
- Problems:
- Can we allocate an array big enough?
- Super wasteful



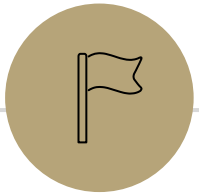
Idea 2:

- Create a smaller array, but create a way to translate given integer keys into available indices. Way less wasteful space-wise.
- Problem:
- How can we pick a good translation?



Review of Maps

Direct Access Map



Hash Functions

Hash Collisions

Hash functions: translating a piece of data to an int

Hash function definition

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

- In our case: we want to translate int keys to a valid index in our array. If our array is length 10 but our input key is 500, we need to make sure we have a way of mapping that to a number between 0 and 9 (the valid indices for a length 10 array). This mapping that we decide on is a **hash function**.

- One simple thing we can do (and that you will do when you implement this in your project):
  - Hash function: take your key and % it by the length of the array.
  - ex: key is 500, and array is length 10 – if you take 500 % 10, you will get the number 0, so we'd just plop 500 and it's value at index 0.

Review: Integer remainder with % "mod"

- The `%` operator computes the remainder from integer division.
- `14 % 4` is `2`

`218 % 5` is `3`

```
      3
4 ) 14
    12
     2
```

```
       43
5 ) 218
     20
      18
      15
       3
```

Equivalently, to find `a % b` (for `a,b > 0`):

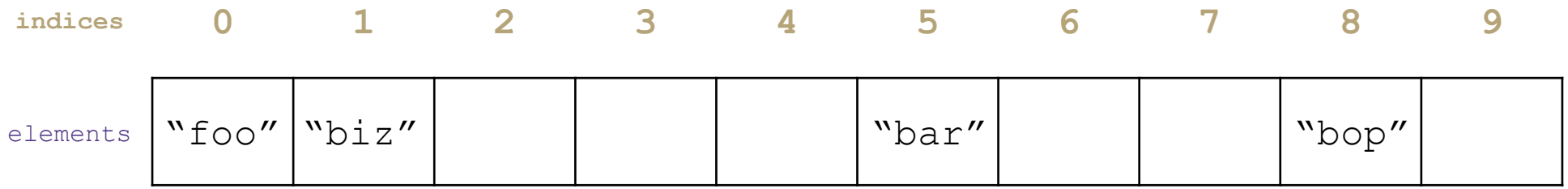
```
while(a > b-1)
    a -= b;
return a;
```

- Applications of `%` operator:
  - Obtain last digit of a number: `230857 % 10` is `7`
  - See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`
  - Limit integers to specific range: `8 % 12` is `8`, `18 % 12` is `6`

Limit keys to indices within array

For more review/practice, check out Khan Academy Lesson

First Hash Function: % table size




```
put(0, "foo");  0 % 10 = 0
put(5, "bar");  5 % 10 = 5
put(11, "biz")  11 % 10 = 1
put(18, "bop"); 18 % 10 = 8
```

Implement First Hash Function

```
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
   return k % this.data.length;
}
```

Note: % is just a math operator like +, -, /, *, so it's constant runtime

SimpleHashMap<Integer>

state
Data[]
size

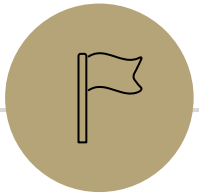
behavior
put mod key by table size, put item at result
get mod key by table size, get item at result
containsKey mod key by table size, return data[result] == null remove mod key by table size, nullify element at result
size return count of items in dictionary

Operation		Array w/ indices as keys
put(key,value)	best	$\Theta(1)$
	worst	$\Theta(1)$
get(key)	best	$\Theta(1)$
	worst	$\Theta(1)$
containsKey(key)	best	$\Theta(1)$
	worst	$\Theta(1)$

Review of Maps

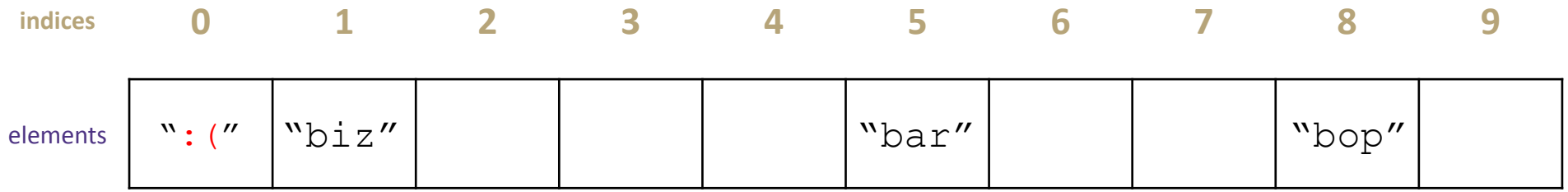
Direct Access Map

Hash Functions



Hash Collisions

First Hash Function: key % table size




```
put(0, "foo");    0 % 10 = 0
put(5, "bar");    5 % 10 = 5
put(11, "biz")    11 % 10 = 1
put(18, "bop");   18 % 10 = 8
put(20, ":(");    20 % 10 = 0
```

Collision!

Hash Obsession: Collisions

Collision: multiple keys translate to the same location of the array

Future big idea: the fewer the collisions, the better the runtime! (we'll see this when we figure out that resolving these leads to worse runtime)

Two questions:

1. When we have a collision, how do we resolve it?
2. How do we minimize the number of collisions?

Strategies to handle hash collision

There are multiple strategies. In this class, we'll cover the following ones:

1. Separate chaining
   - turn each element of the array into a "bucket" that can hold multiple items

2. Open addressing
   - If there is a collision, apply a 2nd formula to find a new location that is hopefully empty, repeat until an empty location is found

Separate chaining

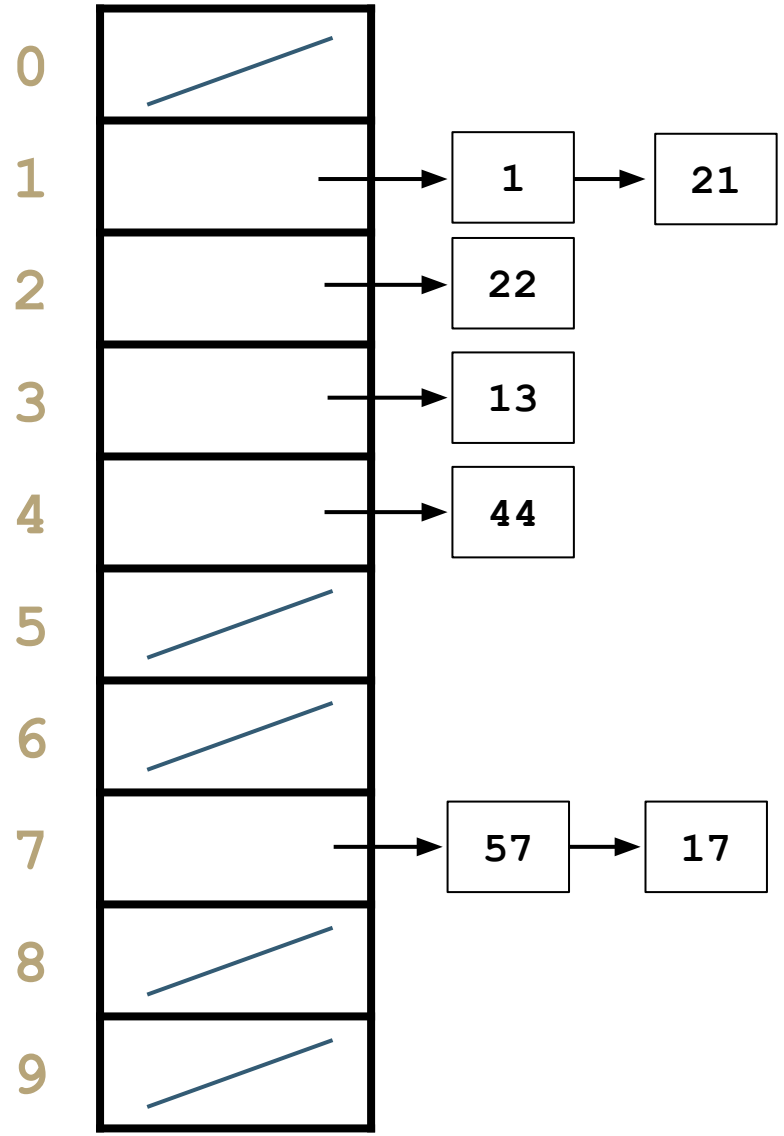
Solution 1: Separate Chaining

Each index in our array represents a "bucket". When an item x hashes to index h:

- If the bucket at index h is empty: create a new list containing x
- If the bucket at index h is already a list: add x if it is not already present

In other words:

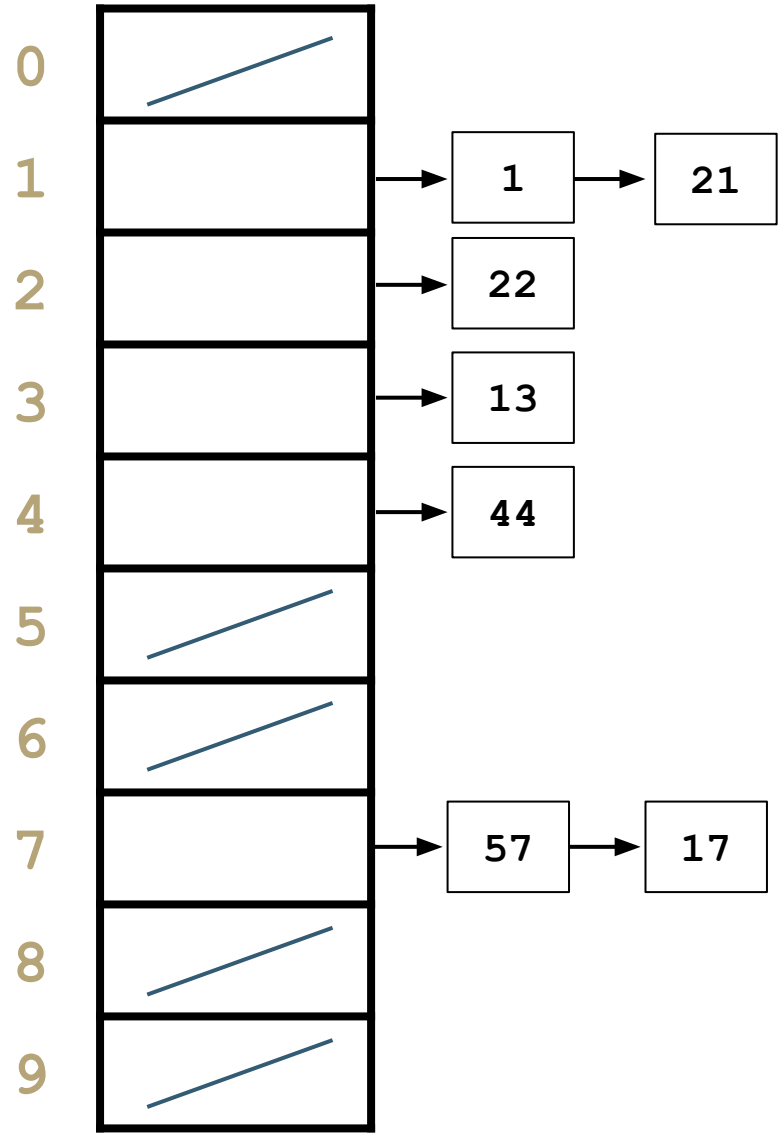
If multiple things hash to the same index, then we'll just put all of those in that same index bucket. Often, you'll see the data structure chosen is a linked-list like structure.



Separate chaining

Reminder: the implementations of put/get/containsKey are all very similar, and almost always will have the same complexity class

```
// some pseudocode
public boolean containsKey(int key) {
    int bucketIndex = key % data.length;
    loop through data[bucketIndex]
    return true if we find the key in
    data[bucketIndex]
    return false if we get to here
}
```

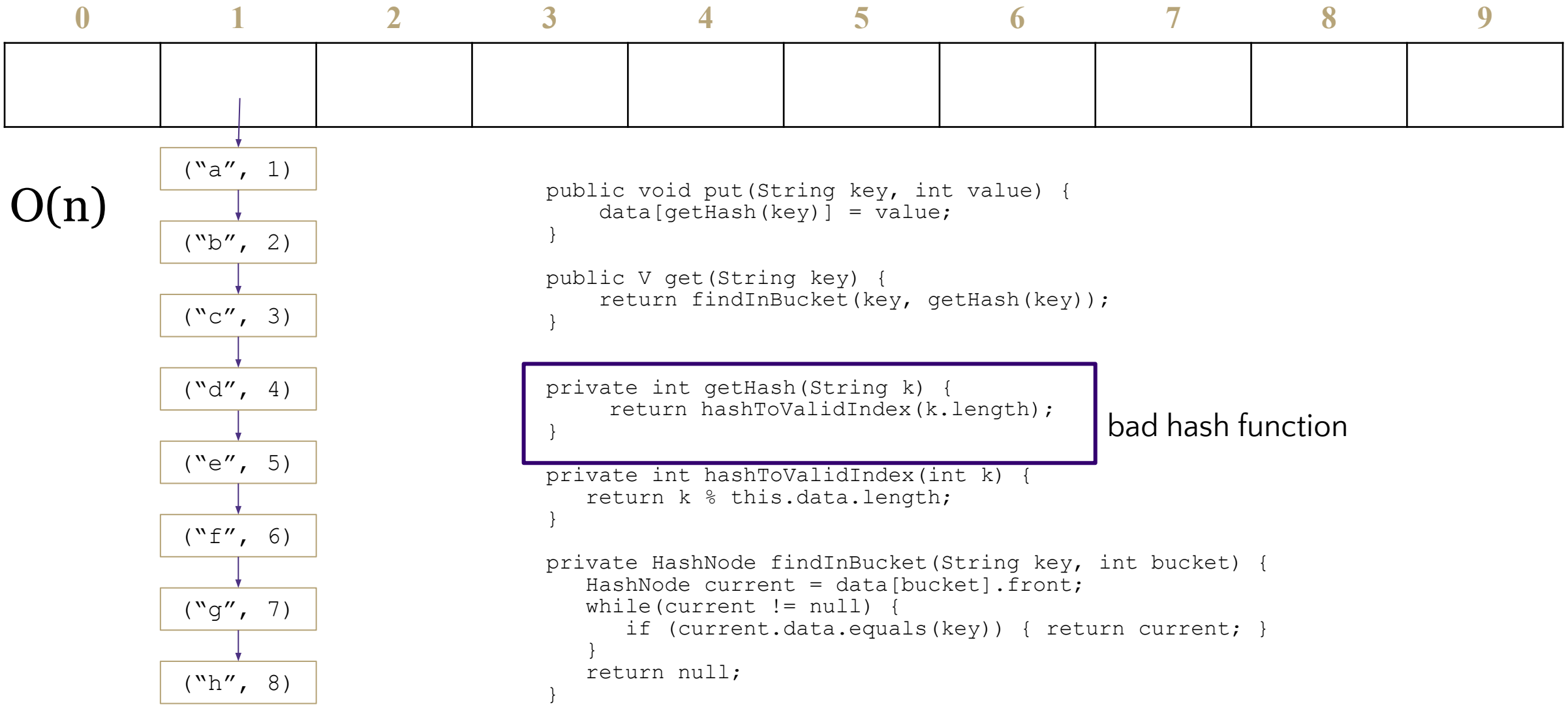


runtime analysis

Are there different possible states for our Hash Map that make this code run slower/faster, assuming there are already n key-value pairs being stored?

Yes! If we had to do a lot of loop iterations to find the key in the bucket, our code will run slower.

A **worst** case situation for separate chaining

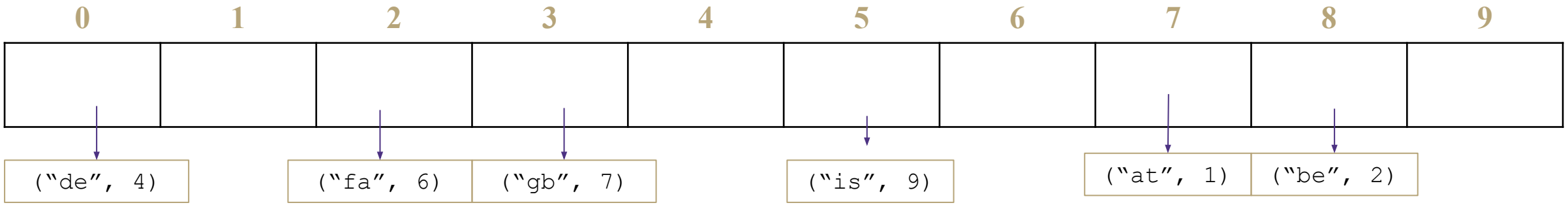


O(n)

```
public void put(String key, int value) {
    data[getHash(key)] = value;
}

public V get(String key) {
    return findInBucket(key, getHash(key));
}

private int getHash(String k) {
     return hashToValidIndex(k.length);
}

private int hashToValidIndex(int k) {
   return k % this.data.length;
}

private HashNode findInBucket(String key, int bucket) {
   HashNode current = data[bucket].front;
   while(current != null) {
      if (current.data.equals(key)) { return current; }
   }
   return null;
}
```

bad hash function

A **best** case situation for separate chaining



O(1)

```
private int getHash(String k) {
    return hashToValidIndex((int)k.charAt(0));
}
```

better hash function, more variance -> fewer collisions

In-practice situations for separate chaining

Generally we can achieve something close to the best case situation from the previous slide and maintain out HashMap so that every bucket only has a small constant number of items. There may be some outliers that have slightly more buckets, but generally if we follow all the best practices, the runtime will still be $\Theta(1)$ for most cases!

(The worst case is still $\Theta(n)$ but again, we'll try really hard to prevent that)

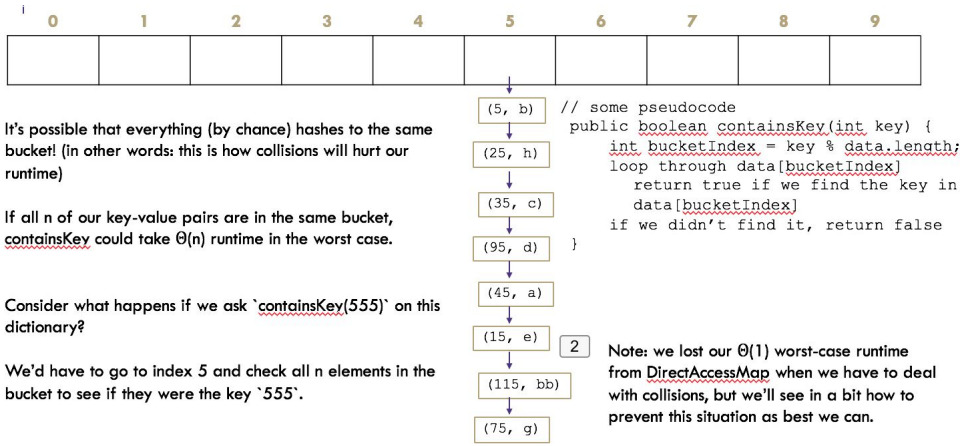
Operation		Array w/ indices as keys
`put(key,value)`	best	$\Theta(1)$
	In-practice	$\Theta(1)$
	worst	$\Theta(n)$
`get(key)`	best	$\Theta(1)$
	In-practice	$\Theta(1)$
	worst	$\Theta(1)$
`remove(key)`	best	$\Theta(1)$
	In-practice	$\Theta(1)$
	worst	$O(n)$

Reminder: the in-practice runtimes are assuming an even distribution of the keys inside the array and following of best-practices to ensure the average chain length is low.

Best practices (pay attention to this for the hw)

What about resizing?

- for data structures like ArrayMap or ArrayList or ArrayStack we had to resize when we're full just because we couldn't store any more things! But our Separate Chaining Hash Map is a little bit different: we aren't ever ***forced*** to resize our main array, since the buckets are flexible size.



It turns out we still want to resize "every so often" to make sure the average/expected length of each bucket is a small number.

Consider what happens if we had the array length 10 like on the left, but had 100 key-value pairs?

Assuming our in-practice niceness (not-worst case) you would expect on average each of the 10 buckets has about 10 key-value pairs in it.

What happens if we stick with the same size array but add 100 more key-value pairs? Each bucket gets about 10 more –key-value pairs and the runtime is getting worse and worse.

Best practices (pay attention to this for the hw)

It turns out we still want to resize "every so often" to make sure the average/expected length of each bucket is a small number.

Consider what happens if we had the array length 10 like on the left, but had 100 key-value pairs?

Assuming our in-practice niceness (not-worst case) you would expect on average each of the 10 buckets has about 10 key-value pairs in it.

What happens if we stick with the same size array but add 100 more key-value pairs? Each bucket gets about 10 more –key-value pairs and the runtime is getting worse and worse.

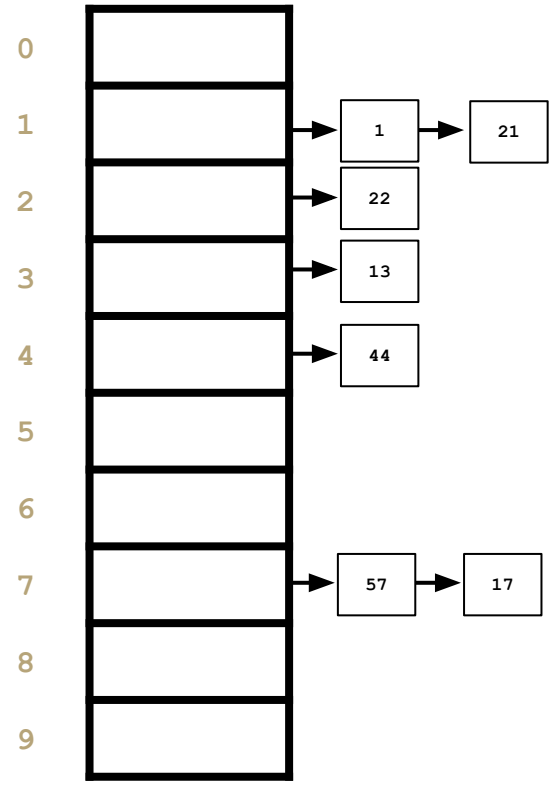
The pattern we're getting to is that the expected runtime is approximately: # of pairs / array.length (AKA n / c where n is the number of elements and c is the number of possible chains). If array.length is fixed for your whole program, then this is an order-n runtime, but if the array.length also increases (because you re-size) and you redistribute out the values evenly across the buckets, you can keep your runtime low. In particular, **if you resize when when your n / c ratio increases to about 1, you're expected to have 1 element or fewer in each bucket at all times.** (do this on your homework).

Tip: make sure you re-hash (re-distribute) your keys by the new array length after re-sizing so they don't get clustered in the old array length range.

Lambda + resizing rephrased

To be more precise, the in-practice runtime depends on λ, the current average chain length.

However, if you resize once you hit that 1:1 threshold, the current λ is expected to be less than 1 (which is a constant / constant runtime, so we can simplify to O(1)).



Operation		Array w/ indices as keys
`put(key,value)`	best	O(1)
	In-practice	O(λ)
	worst	O(n)
`get(key)`	best	O(1)
	In-practice	O(λ)
	worst	O(n)
`remove(key)`	best	O(1)
	In-practice	O(λ)
	worst	O(n)

"In-Practice" 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 λ = n/c

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+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

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

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

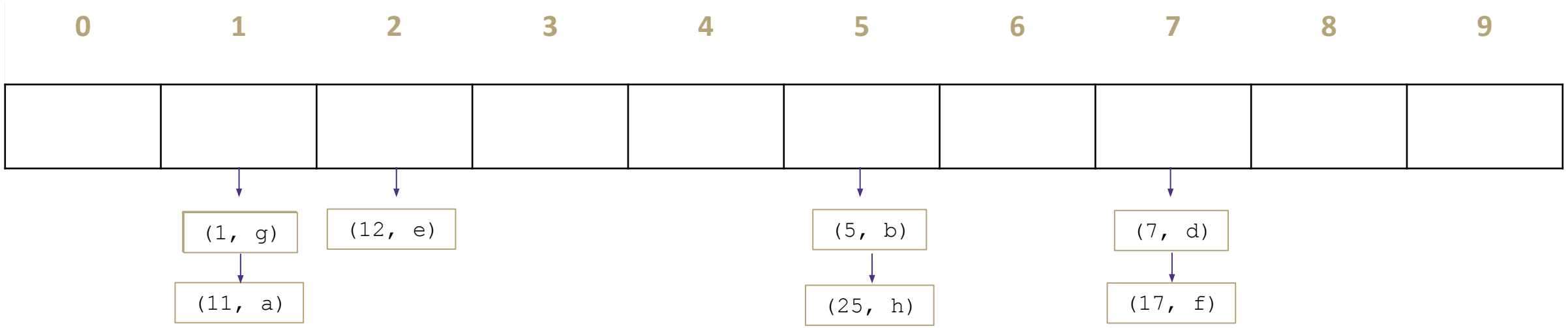
- call the key.hashCode() to get an int representation of the object
- % by the array table length to convert it to a valid index for your hash map

Best practices for an nice distribution of keys recap

- Resize when lambda (number of elements / number of buckets) increases up to 1
- 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
- Design the hashCode of your keys to be somewhat complex and lead to a distribution of different output numbers

Practice

- 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?
- (1, a) (5,b) (11,a) (7,d) (12,e) (17,f) (1,g) (25,h)



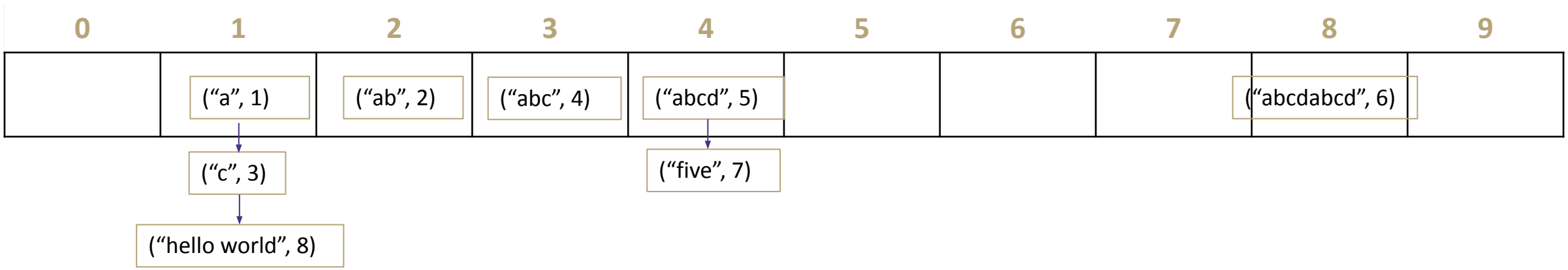
Practice

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)



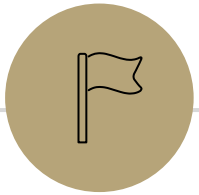
Java and Hash Functions

- 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.



That’s all!