Lecture 11: Introduction to Hash Tables

CSE 373: Data Structures and Algorithms

Administrivia

When you're submitting your group writeup to gradescope, be sure to use the group submission option if you have a partner.

Project 1 part 2 due Thursday night.

Exercise 2 due Friday night.

Project 2 will come out tonight, and Exercise 3 will come out Friday.

Due in two weeks (Wednesday the 31st for Project 2, and Friday the 2nd for Exercise 3)

They "should" be one week assignments… but next Friday is the midterm!

We're leaving it to you to decide how/when to study for the midterm vs. doing homework.

Aside: How Fast is $\Theta(\log n)$?

If you just looked at a list of common running times

Class	Big O	If you double N…	Example algorithm
constant	O(1)	unchanged	Add to front of linked list
logarithmic	O(log n)	Increases slightly	Binary search
linear	O(n)	doubles	Sequential search
"n log n"	O(nlog n)	Slightly more than doubles	Merge sort
quadratic	$O(n^2)$	quadruples	Nested loops traversing a 2D array

You might think this was a small improvement.

It was a HUGE improvement!

Logarithmic vs. Linear

If you double the size of the input,

- A linear time algorithm takes twice as long.
- A logarithmic time algorithm has a constant **additive** increase to its running time.

To make a logarithmic time algorithm take twice as long, how much do you have to increase n by?

pollEV.com/cse373su19

How do you increase n?

You have to square it $\log(n^2) = 2\log(n)$.

A gigabyte worth of integer keys can fit in an AVL tree of height 60.

It takes a ridiculously large input to make a logarithmic time algorithm go slowly.

Log isn't "that running time between linear and constant" it's "that running time that's barely worse than a constant."

Logarithmic Running Times

This identity is so important, one of my friends made me a cross-stitch of it.

Two lessons:

1. Log running times are REALLY REALLY FAST.
2. $O(\log(n^3))$ is not simplified, it's just $O(\log n)$

Aside: Traversals

What if the heights of subtrees were corrupted.

How could we calculate from scratch?

We could use a “traversal”

- A process that visits every piece of data in a data structure.

```
int height(Node curr){
     if(curr==null) return -1;
     int h = Math.max(height(curr.left),height(curr.right));
     return h+1;
}
```

Three Kinds of Traversals

```
InOrder(Node curr){
    InOrder(curr.left);
    doSomething(curr);
    InOrder(curr.right);
}
```

```
PreOrder(Node curr){
    doSomething(curr);
    PreOrder(curr.left);
    PreOrder(curr.right);
}
```

```
PostOrder(Node curr){
    PostOrder(curr.left);
    PostOrder(curr.right);
    doSomething(curr);
}
```

Traversal Practice

For each of the following scenarios, choose an appropriate traversal:

1. Print out all the keys in an AVL-Dictionary in sorted order.

2. Make a copy of an AVL tree

3. Determine if an AVL tree is balanced (assume height values are not stored)

Traversal Practice

For each of the following scenarios, choose an appropriate traversal:

1. Print out all the keys in an AVL-Dictionary in sorted order.

   In order

2. Make a copy of an AVL tree

   Pre-order

3. Determine if an AVL tree is balanced (assume height values are not stored)

   Post-order

Traversals

If we have n elements, how long does it take to calculate height?

$\Theta(n)$ time.

The recursion tree (from the tree method) IS the AVL tree!

We do a constant number of operations at each node

In general, traversals take $\Theta(n \cdot f(k))$ time,

where `doSomething()` takes $\Theta(f(k))$ time.

Common question on technical interviews!

Aside: Other Self-Balancing Trees

There are lots of flavors of self-balancing search trees

"Red-black trees" work on a similar principle to AVL trees.

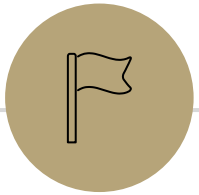
"Splay trees"
- Get $O(\log n)$ amortized bounds for all operations.

"Scapegoat trees"

"Treaps" – a BST and heap in one (!)

B-trees (see other 373 versions) optimized for huge datasets.

If you have an application where you need a balanced BST that also [does something] it might already exist.
Google first, you might be able to use a library.



Hashing

Review: Dictionaries

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 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

LinkedDictionary<K, V>

state

```
front
size
```

behavior

put if key is unused, create new 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

AVLDictionary<K, V>

state

```
overallRoot
size
```

behavior

put if key is unused, create new pair, place in BST order, rotate to maintain balance
get traverse through tree using BST property, return item if found
containsKey traverse through tree using BST property, return if key is found
remove traverse through tree using BST property, replace or nullify as appropriate
size return count of items in dictionary

3 Minutes

Review: Dictionaries

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

Why are we so obsessed with Dictionaries? **It's all about data baby!**

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

Operation		ArrayList	LinkedList	BST	AVLTree
put(key,value)	best				
	worst				
get(key)	best				
	worst				
remove(key)	best				
	worst				

3 Minutes

Review: Dictionaries

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

Why are we so obsessed with Dictionaries? **It's all about data baby!**

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

Operation		ArrayList	LinkedList	BST	AVLTree
put(key,value)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(logn)
get(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(logn)
remove(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(logn)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(logn)

“In-Practice” Case

For Hash Tables, we’re going to talk about what you can expect “in-practice”

- Instead of just what the best and worst scenarios are.

Other resources (and previous versions of 373) use “average case”

There’s a lot of math (beyond the scope of the course) needed to make “average” statements precise.

- So we’re not going to do it that way.

For this class, we’ll just tell you what assumptions we’re making about how the “real world” usually works.

And then do worst-case analysis under those assumptions.

Can we do better?

What if we knew exactly where to find our data?

Implement a dictionary that accepts only integer keys between 0 and some value k

- -> Leverage Array Indices!

"Direct address map"

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	worst	O(1)
get(key)	best	O(1)
	worst	O(1)
remove(key)	best	O(1)
	worst	O(1)

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
```

Implement Direct Access Map

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

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

public void remove(int key) {
    this.entureIndexNotNull(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

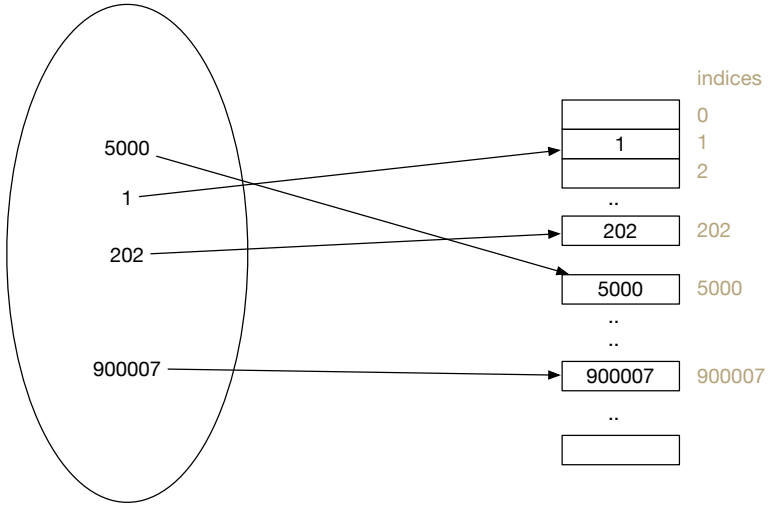
Can we do this for any integer?

Idea 1:

Create a GIANT array with every possible integer as an index

Problems:

- Can we allocate an array big enough?
- Super wasteful

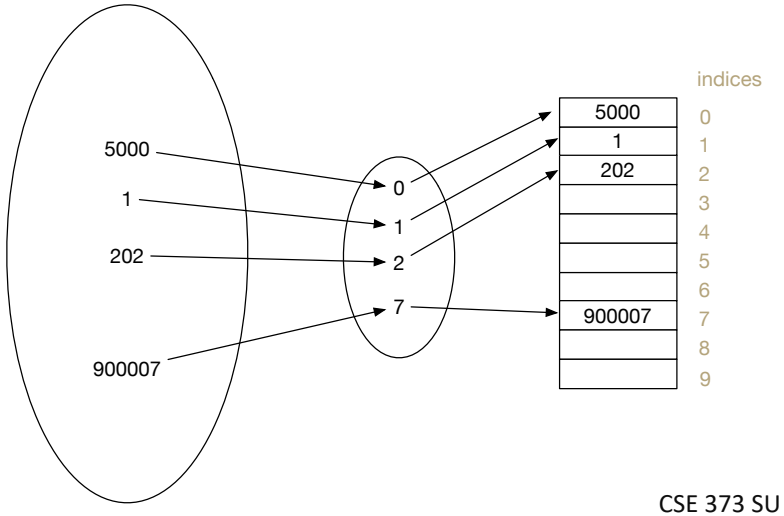


Idea 2:

Create a smaller array, but create a way to translate given integer keys into available indices

Problem:

- How can we pick a good translation?



Review: Integer remainder with % "mod"

The % operator computes the remainder from integer division.

`14 % 4` is `2`

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

`218 % 5` is `3`

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

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

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

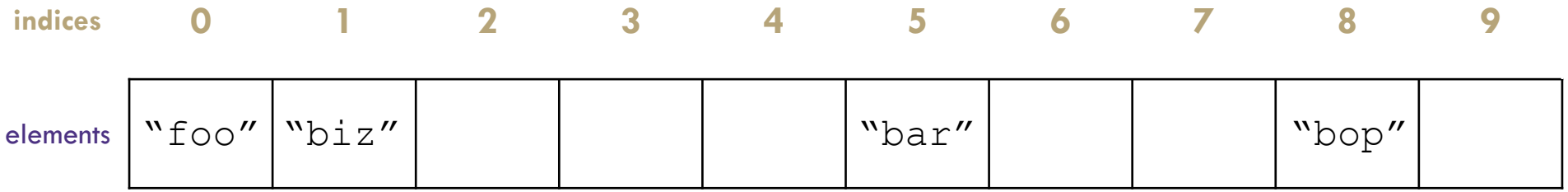
Applications of % operator:

- 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 https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

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 V get(int key) {
    int newKey = getKey(key);
    this.ensureIndexNotNull(newKey);
    return this.data[newKey;
}

public void put(int key, int value) {
     this.array[getKey(key)] = value;
}
public void remove(int key) {
    int newKey = getKey(key);
    this.entureIndexNotNull(newKey);
    this.data[newKey] = null;
}
public int getKey(int k) {
    return k % this.data.length;
}
```

SimpleHashMap<Integer>

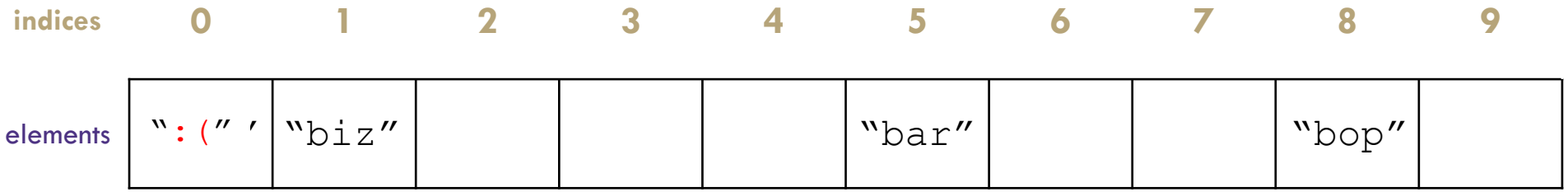
state

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

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
put(20, ":(");    20 % 10 = 0
```

Collision!

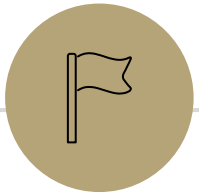
Hash Obsession: Collisions

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

The fewer the collisions, the better the 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 collisions

Strategies to handle hash collision

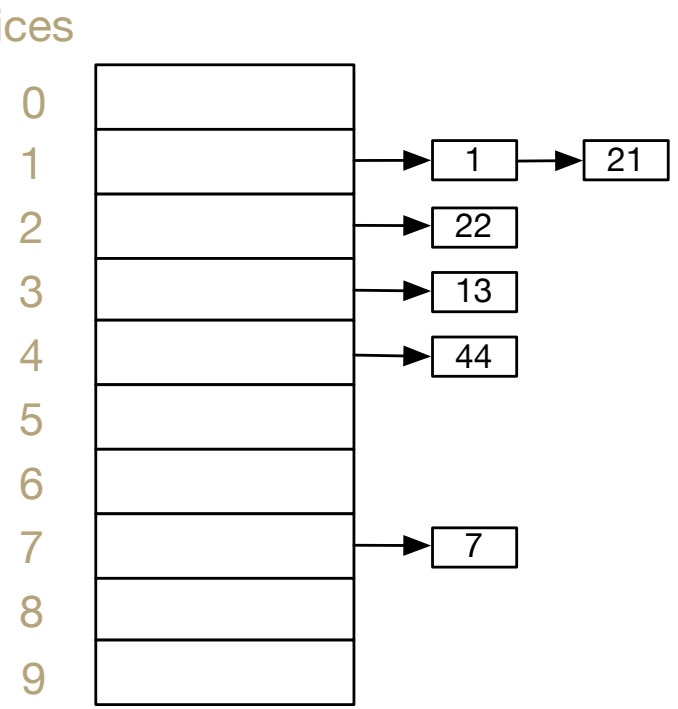
There are multiple strategies. In this class, we'll cover the following ones:

1. Separate chaining
2. Open addressing
   - Linear probing
   - Quadratic probing
   - Double hashing

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	Θ(1)
	In-practice	
	worst	Θ(n)
get(key)	In-practice	Θ(1)
	average	
	worst	Θ(n)
remove(key)	best	Θ(1)
	In-practice	
	worst	Θ(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 $\lambda = \frac{n}{c}$

In-Practice

We're going to make an **assumption** about how often collisions happen.

It's not actually true, but it's "close enough" to true that our big-O analyses will be pretty consistent with what you usually see in-practice.

Our Hashing Assumption

The hash function will distribute the input keys as evenly as possible across the buckets.

This is not true in the real-world.

But what is usually true in the real-world is pretty close is close enough that the big-O analyses are the same.

In-Practice

Our Hashing Assumption

The hash function will distribute the input keys as evenly as possible across the buckets.

What is the worst-case under our hashing assumption?

We might have to go to the end of the linked list in one of the buckets. How long will that linked list be?

If we have n keys and our hash table has c buckets, it will be length $\frac{n}{c}$.

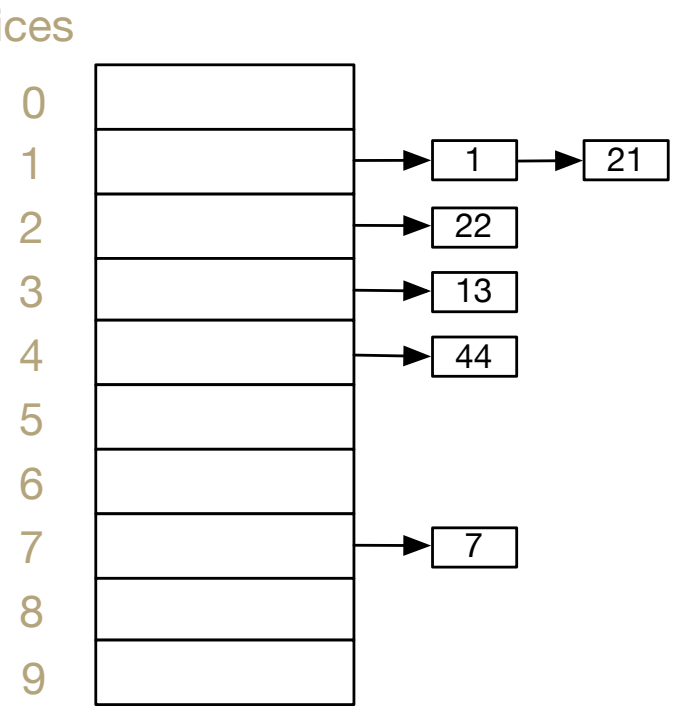
That number will come up so often, we give it a name. It's the **load factor**.

- We denote it by λ.

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	O(1)
	In-practice	O(1 + λ)
	worst	O(n)
get(key)	In-practice	O(1)
	average	O(1 + λ)
	worst	O(n)
remove(key)	best	O(1)
	In-practice	O(1 + 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 $\lambda = \frac{n}{c}$

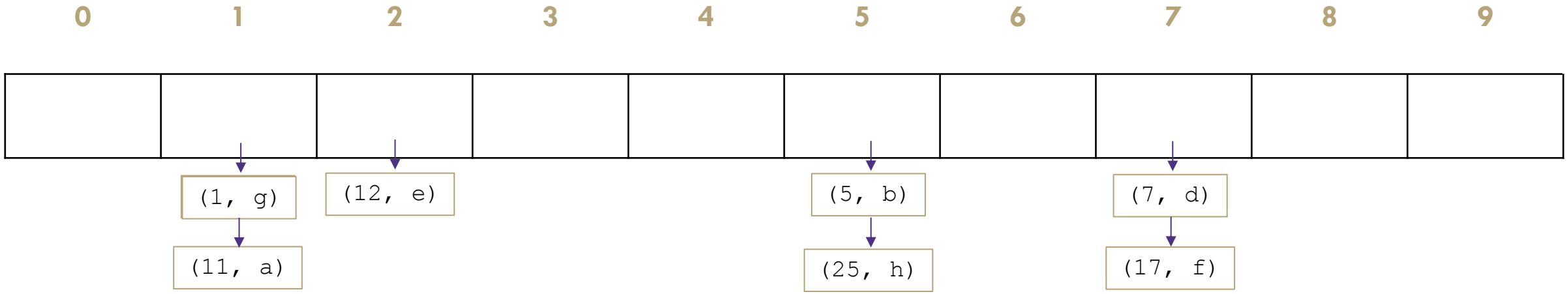
3 Minutes

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)



What about non integer keys?

Hash Function

An algorithm that maps a given key to an integer representing the index in the array for where to store the associated value

Goals

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

How to Hash non Integer Keys

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

3 Minutes

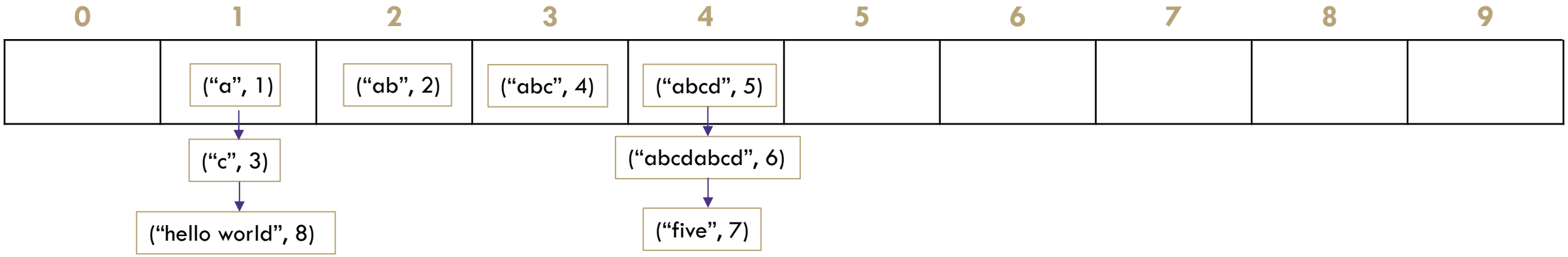
Practice

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.

Resizing

Our running time in practice depends on λ. What do we do when λ is big?

Resize the array!

- Usually we double, that's not quite the best idea here
- Increase array size to next prime number that's roughly double the array size
  - Prime numbers tend to redistribute keys, because you're now modding by a completely unrelated number.
  - If % TableSize gives you k then `%2*TableSize` gives either k or $2k$.
- Rule of thumb: Resize sometime around when λ is somewhere around 1 if you're doing separate chaining.
  - When you resize, you have to rehash everything!