Hashing

CS 61B Spring 2016: Discussion 9

Announcements

I found out how to change the background color of my slides.

Midterm 2 is on Thursday 3/31 after Spring Break.

Next discussion is midterm prep and review!

Homework 2 is due Wednesday 3/16 (tomorrow).

Lab this week is not mandatory. Enjoy your Spring Break! :)

```
git commit -am “message”
```

`git commit -am “message”`

If you know you want to stage all of your local changes before committing and pushing, you can do the following two commands:

```
$ git add --all
$ git commit -m “commit message”
```

For this, there’s a shortcut. We can use the `-a` flag for `git commit`. Then, we can do the following which is equivalent to the above.

```
$ git commit -am “commit message”
```

Chris Martin
@chris__martin

Follow

alias such=git
alias very=git
alias wow='git status'

$ wow
$ such commit
$ very push

Reply Retweeted Favorite More

2,289 RETWEETS | **1,131** FAVORITES

7:56 PM - 8 Jan 14


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Hashing

Quick Poll:

Who went to the lecture on hashing?

Hashing

Hashing is when we take some data and turn it into an index that we use store the data.

The process of converting the data into an index is called **computing a hash code.**

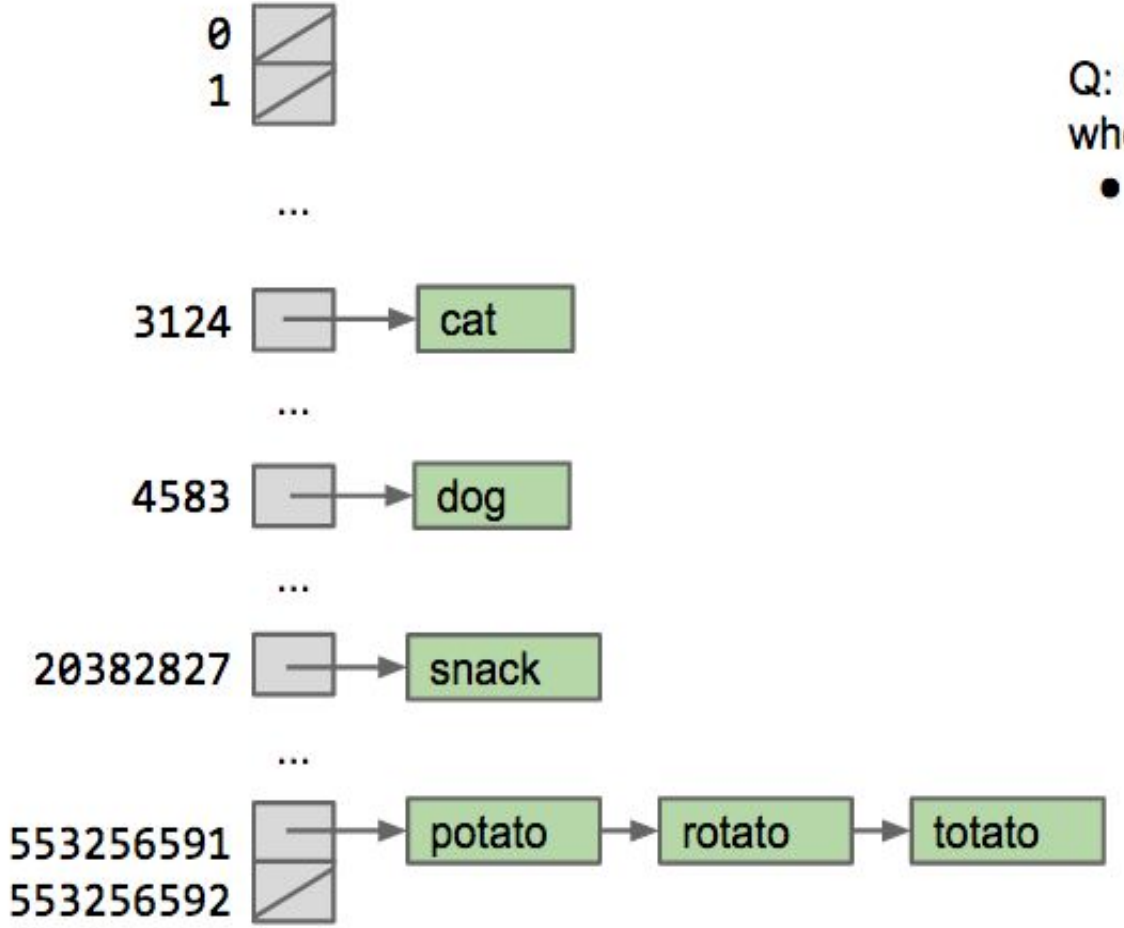
In Java, every object must have a **hashCode()** method that converts itself into an integer that can be used as an index.

Collisions

A **collision** occurs when more than one object has the **same hash code**.

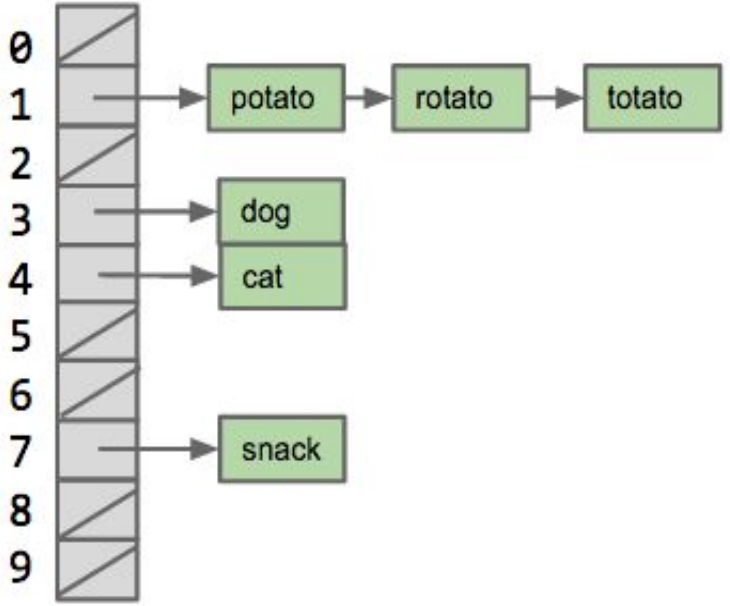
Instead of storing true at that index (the hash code) to note that an object with that hash code exists, we instead store a list of those objects that have the same hash code. This is called **external chaining**.

Observation: Can use modulus of hashcode to reduce bucket count.



Q: If we use the 10 buckets on the right, where should our six items go?

- Put in bucket = hashCode % 10



Load Factor

Recall resizing your ArrayDeque from Project 1.

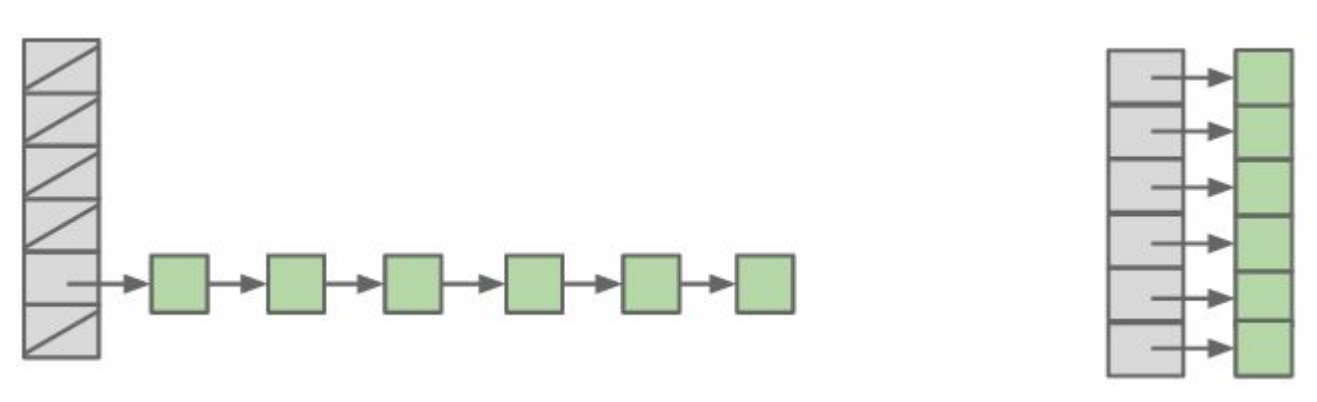
If we have N items distributed across M buckets, then we have a **load factor L = N / M**. The lower L is, the better! When L gets too big, we have to resize (increase M), so that L becomes small again.

Hash Tables

What we have created (buckets, load factor, turning objects into integers by computing a hash code, external chaining for collisions) is a data structure called a **hash table**.

Items are mapped to a bucket using their hashCode() function. Usually then taken **modulo M** (M is number of buckets we have).

Good Hash Functions



Bad :(

Good! :)

Good Hash Functions

Good hash functions...

- Make hash table operations very efficient (amortized)

	contains(x)	insert(x)	
Linked List	Θ(N)	Θ(N)	Used by: TreeSet
Bushy BSTs	Θ(log N)	Θ(log N)	
Unordered Array	Θ(N)	Θ(N)	
Hash Table	Θ(1)	Θ(1)	Used by: HashSet, Python dictionaries

Worst case runtimes

- In general, should help spread objects out between buckets
- Objects that are the same (.equals) should have the same hash code

Why Should I Care About Hashing?

It’s on your midterm. And final. And used in your homework.

Why Should I Care About Hashing?

~~It's on your midterm. And final. And used in your homework.~~

Hashing is an insanely powerful concept and tool in the real world.

Example applications:

- Performance improvement (Θ(1) operations!)
- Error checking in networking (hash a value, if value is corrupted, then the new hash won't match!)
- Security and cryptography (ensuring confidentiality and integrity)
- Caching/Memoization (storing computed values for quick access later)
- A personal favorite: Computer graphics
- … tons more

Fun with Hash Functions

Problem 1

Problem 1a

```
public int hashCode() {return -1;}
```

Valid. Two integers that are .equals to each other do indeed have the same hash code.

However, this hash function is garbage.

Our collision rate is 100%! Every single integer has the exact same hash code.

Problem 1b

```
public int hashCode() {return intValue() * intValue();}
```

Valid. Two integers that are .equals to each other do indeed have the same hash code.

This hash function is a lot better than the first one. However, integers with the same absolute value will collide, so still not that great.

A better hash function would be to just return `intValue()`.

Problem 1c

```
public int hashCode() {return super.hashCode();}
```

Invalid. Integers that are .equals to each other will not have the same hash code!

This hash function actually returns some integer corresponding to this specific Integer object's location in memory. (This is inherited from the Object class.).

HashMap Theory

Problem 2

Problem 2a

When you modify a **key** that has been inserted into a HashMap, will you be able to retrieve that entry again? Explain.

Sometimes.

The modified key **might** (by some stroke of luck) have the same hash code as the old key. Then, you can still find the key's corresponding value and retrieve it from the HashMap.

Problem 2b

When you modify a **value** that has been inserted into a HashMap, will you be able to retrieve that entry again? Explain.

Always.

The value in the (key, value) pair plays no part in looking up the key in the HashMap. The key is still the same, so we can still hash it, then find the value associated with it in the HashMap.

Let’s revisit the ArrayDeque

Problem 3

Problem 3

	Best Case	Worst Case	Average Case
addFirst/Last	Θ(1)	Θ(N)	Θ(1)
rmFirst/Last	Θ(1)	Θ(N)	Θ(1)
get	Θ(1)	Θ(1)	Θ(1)

Worst case happens when we have to resize our ArrayDeque, since we have to move everything such that we start back at array index 0 for ease of maintenance.

The Power of HashMaps

Problem 4

Problem 4

Memoization: Storing the results of previously solved problems, so that we don't redundantly do it again later. <-- Caching!!

Canonical example: Computing the **n**th Fibonacci number.

Side note: Dynamic Programming (DP) is a general term for a powerful technique used to solve programs more efficiently by formalizing it in a way such that we can take advantage of being able to save the solutions to smaller sub-problems (commonly recursive and uses hash tables).

(for more info, see CS170 "Algorithms")

Problem 4

```
public class Fibonacci {
     HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
     public int fib(int n) {
          // YOUR CODE HERE
     }
}
```

Problem 4

```
public class Fibonacci {
     HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
     public int fib(int n) {
          if (n <= 1) { return n; }
          else if (map.containsKey(n)) {
               return map.get(n);
          }
          int result = fib(n - 2) + fib(n - 1);
          map.put(n, result);
          return result;
     }
}
```

Thanks for coming!

```
What color should I use next week?
```