CS 106B, Lecture 27
Advanced Hashing

This document is copyright (C) Stanford Computer Science and Ashley Taylor, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

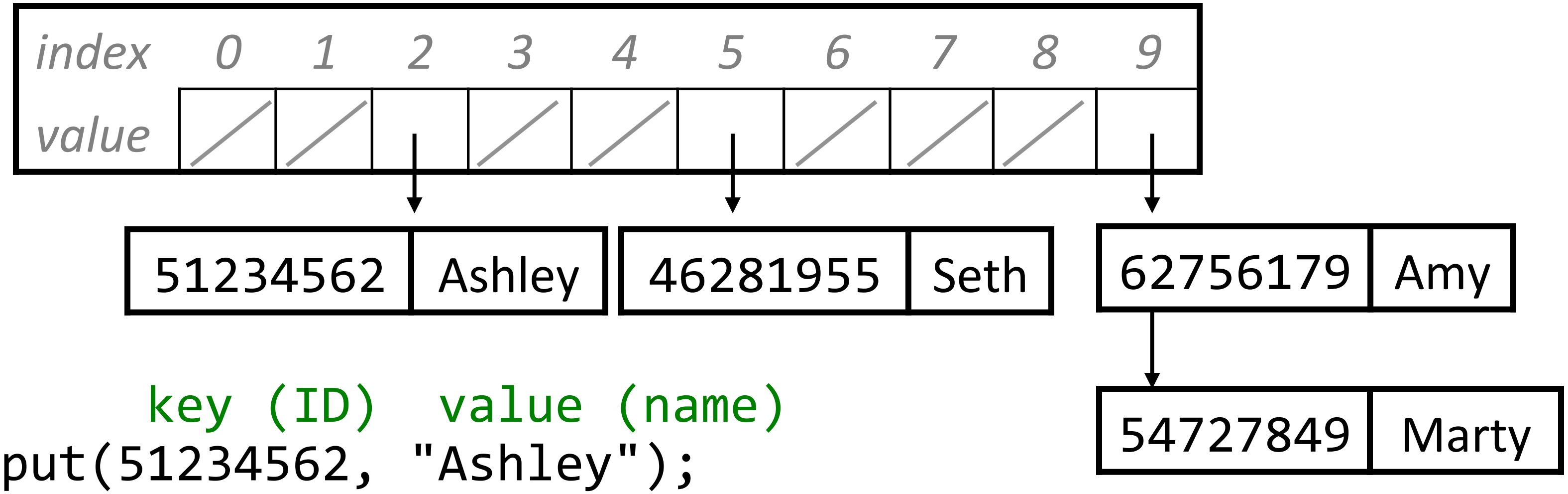
Based on slides created by Marty Stepp, Chris Gregg, Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others

Plan for Today

- Discuss how HashMaps differ from HashSets
- Another implementation for HashSet/Map: Cuckoo Hashing!
- Discuss qualities of a good hash function.
- Learn about another application for hashing: cryptography.

Hash map (15.4)

- A hash map is like a set where the nodes store key/value pairs:



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	/	/		/	/		/	/	/	

51234562 Ashley

46281955 Seth

62756179 Amy

54727849 Marty

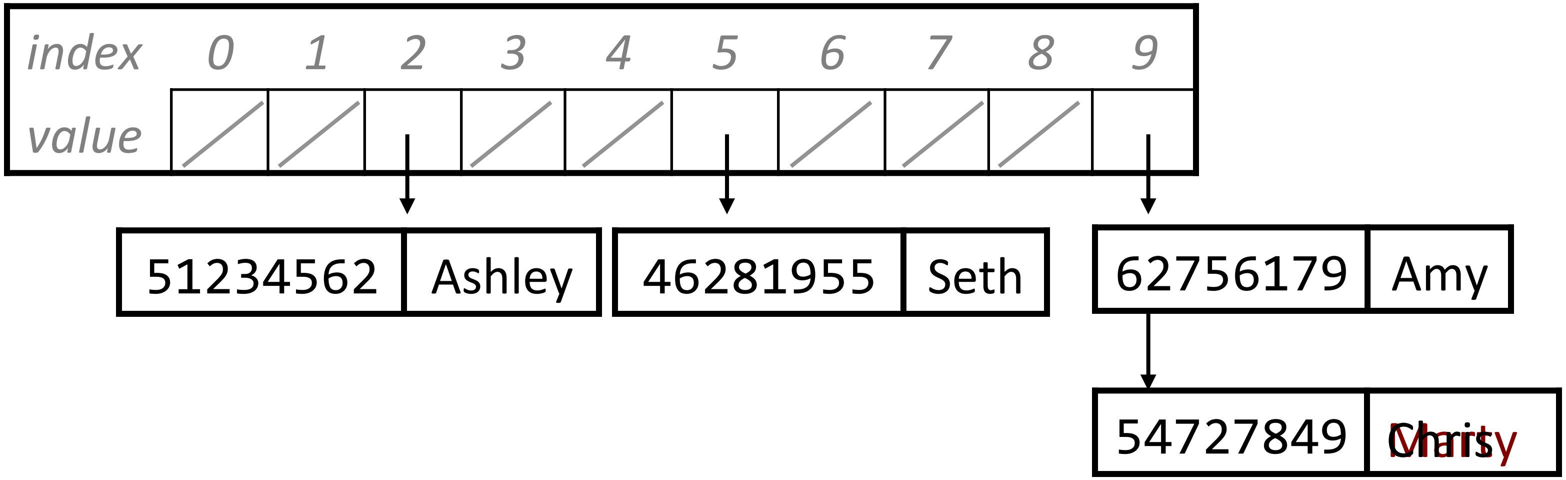
```
//         key (ID)  value (name)
map.put(51234562, "Ashley");
map.put(62756179, "Amy");
map.put(54727849, "Marty");
map.put(46281955, "Seth");
```

  – Must modify the HashNode class to store a key *and* a value

Hash map vs. hash set

- The hashing is always done on the keys, *not* the values.
- The `contains` function is now **`containsKey`**; there and in **`remove`**, you search for a node whose key matches a given key.
- The add method is now **`put`**; if the given key is already there, you must replace its old value with the new one.

```
map.put(54727849, "Chris");   // replace Marty with Chris
```



index	0	1	2	3	4	5	6	7	8	9
value	/	/		/	/		/	/	/	

51234562 | Ashley

46281955 | Seth

62756179 | Amy

54727849 | Chris

54727849 | Marty

Another Way to Hash

- Fun (but soon to be relevant) fact: cuckoo birds lay their eggs in other birds' nests

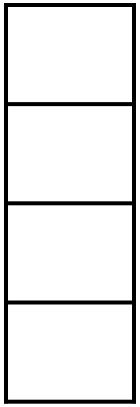
Source: wikimedia

Cuckoo Hashing

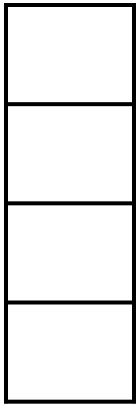
- What if we made contains **really** fast (look at at most two elements, no matter what)?
- Idea: have two arrays that store elements, where each array has its own hash function
- Try hashing the element into both arrays, and put it in an empty space
- If no space is empty, kick out one of the existing elements and move it to the other array.
- Contains just checks the corresponding spot in both arrays
- Slower add, but faster contains

Cuckoo Hashing

Insert: 3



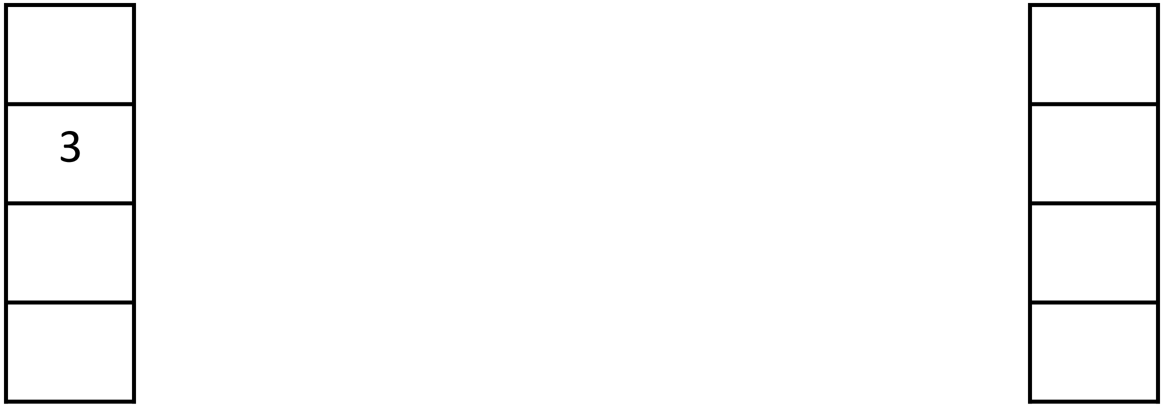
Hash Function: 3x % 4



Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 3

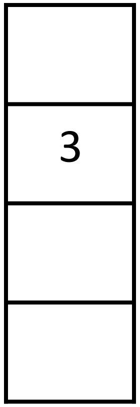


Hash Function: 3x % 4

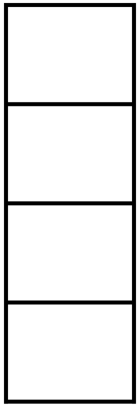
Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 6



Hash Function: 3x % 4



Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 6

3

Hash Function: 3x % 4

6

Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 5

3

Hash Function: 3x % 4

6

Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 5

3

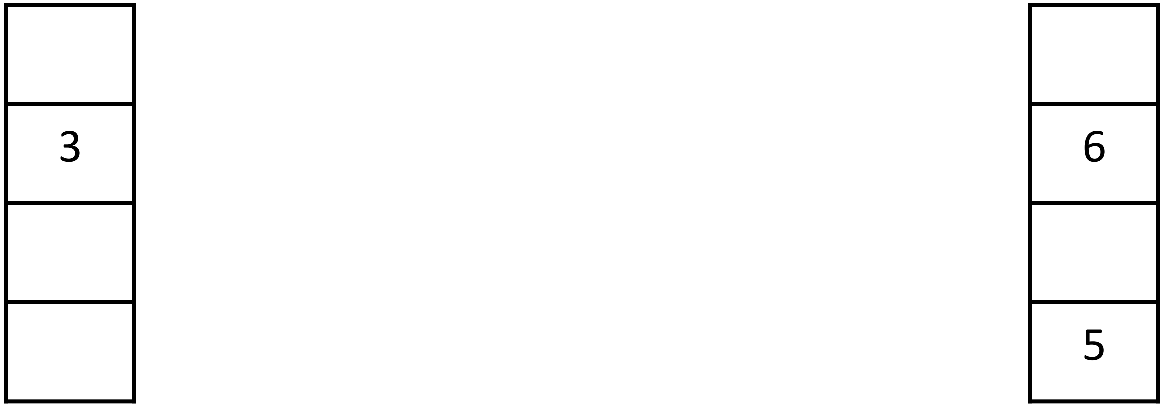
Hash Function: 3x % 4

6
5

Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 7



3

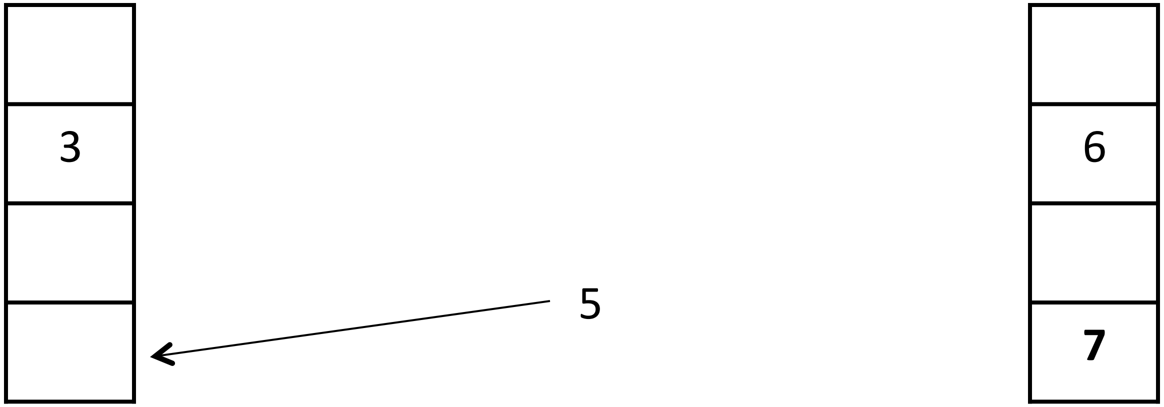
6
5

Hash Function: 3x % 4

Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 7

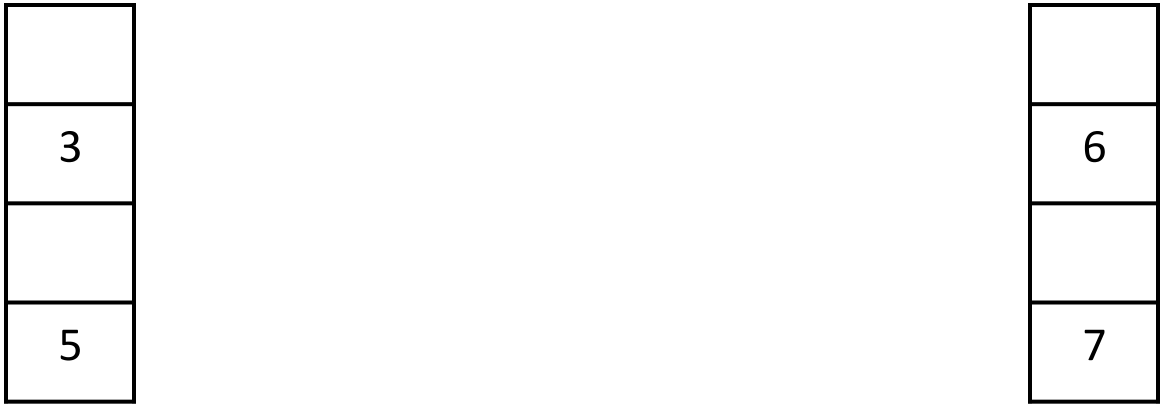


Hash Function: 3x % 4

Hash Function: (2x + 1) % 4

Cuckoo Hashing

Insert: 7

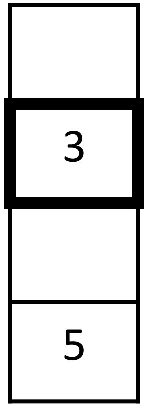


Hash Function: 3x % 4

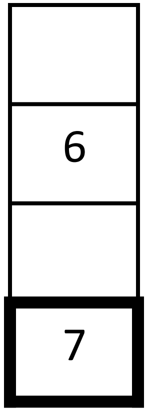
Hash Function: (2x + 1) % 4

Cuckoo Hashing

Search for 7 (look in both arrays)



Hash Function: 3x % 4



Hash Function: (2x + 1) % 4

Cuckoo Hashing

- What are the advantages or disadvantages of cuckoo hashing versus resolving collisions through chaining?
- What do we need to watch out for? When should we rehash?

Announcements

- Calligraphy announcements
  - Should start the 3rd part today or tomorrow at the latest
  - Starter code and Windows – please redownload
  - **No late days may be used, no late submissions accepted**
- Last class tomorrow – go to poll.ly/#/LdVNgWyo/G6z0awRv
- Final is a **on Saturday**, at 8:30AM, in **Cubberley Auditorium**
  - Everything from the course through today is fair game, emphasis is on second half materials (starting with pointers)
  - More information: https://web.stanford.edu/class/cs106b/exams/final.html
  - Practice exam is online – not guaranteed to match in format, etc.
  - Wednesday and Thursday will be final review
- Please give us feedback! cs198.stanford.edu

Hashing strings

- It is easy to hash an integer i (use index *abs(i) % length*).
  - How can we hash other types of values (such as strings)?

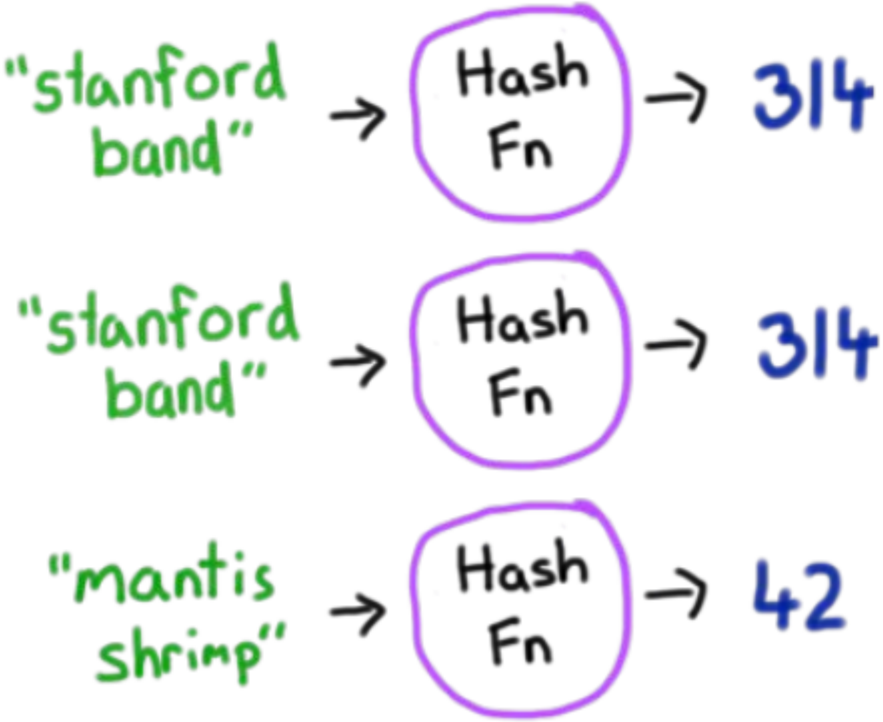
- If we could convert strings into integers, we could hash them.
  - What kind of integer is appropriate for a given string?
  - Does it matter what integer we choose? What should it be based on?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
character	'H'	'i'	' '	'D'	'0'	'0'	'd'	'!'

hashCode consistency

- A valid hashCode function must be ***consistent***
  (must produce same results on each call)

`hashCode(x)  ==  hashCode(x)`, if **`x`**'s state doesn't change

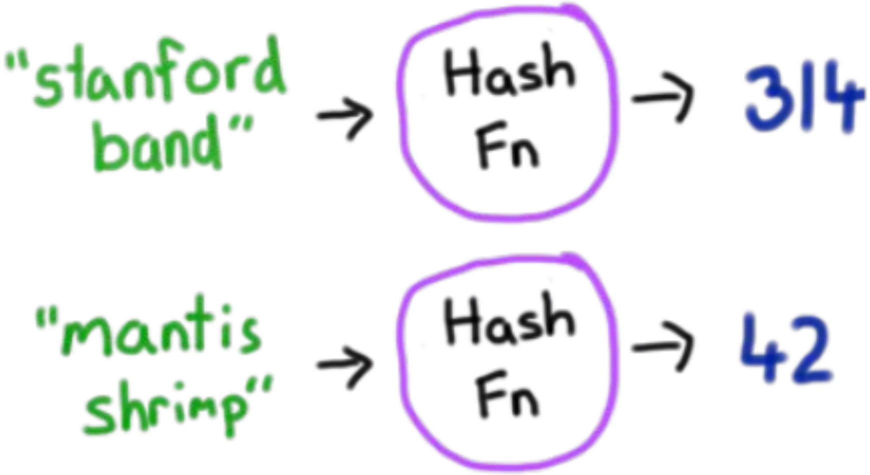


hashCode and equality

- A valid hashCode function must be ***consistent with equality.***

  a == b must imply that **hashCode(a) == hashCode(b)** .

```
Vector<int> v1;                 Vector<int> v2;
v1.add(1);                      v2.add(3);
v1.add(3);                      v2.insert(0, 1);
// hashCode(v1) == hashCode(v2)
```

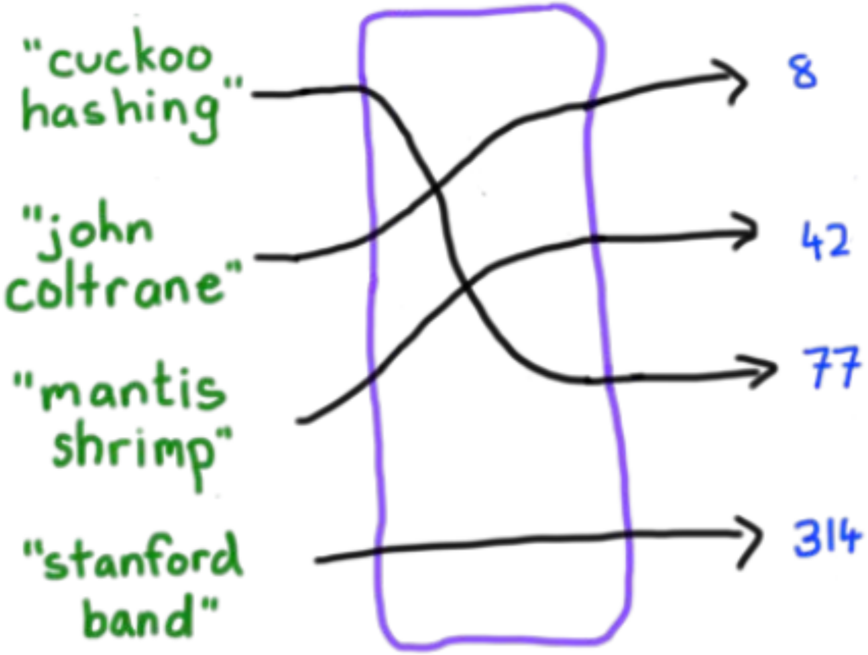


a != b does NOT necessarily imply that

hashCode(a) != hashCode(b) *(why not?)*

hashCode distribution

- A **good** hashCode function is ***well-distributed***.
  - For a large set of distinct values, they should generally return unique hash codes rather than often colliding into the same hash bucket.
  - This property is desired but not required. Why?



Possible hashCode 1

- **Q:** Is this a valid hash function? Is it good?

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

```
int hashCode(string s) { // #1
    return 42;
}
```

Possible hashCode 2

- **Q:** Is this a valid hash function? Is it good?

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

```
int hashCode(string s) { // #2
    return randomInteger(0, 9999999);
}
```

Possible hashCode 3

- **Q:** Is this a valid hash function? Is it good?

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

```
int hashCode(string s) { // #3
    return (int) &s;   // address of s (a pointer)
}
```

Possible hashCode 4

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

- **Q:** Is this a valid hash function? Is it good?

```
int hashCode(string s) { // #4
    return s.length();
}
```

Possible hashCode 5

- **Q:** Is this a valid hash function? Is it good?

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

```
int hashCode(string s) { // #5
    if (s.length() > 0) {
        return (int) s[0];   // ascii of 1st char
    } else {
        return 0;
    }
}
```

Possible hashCode 6

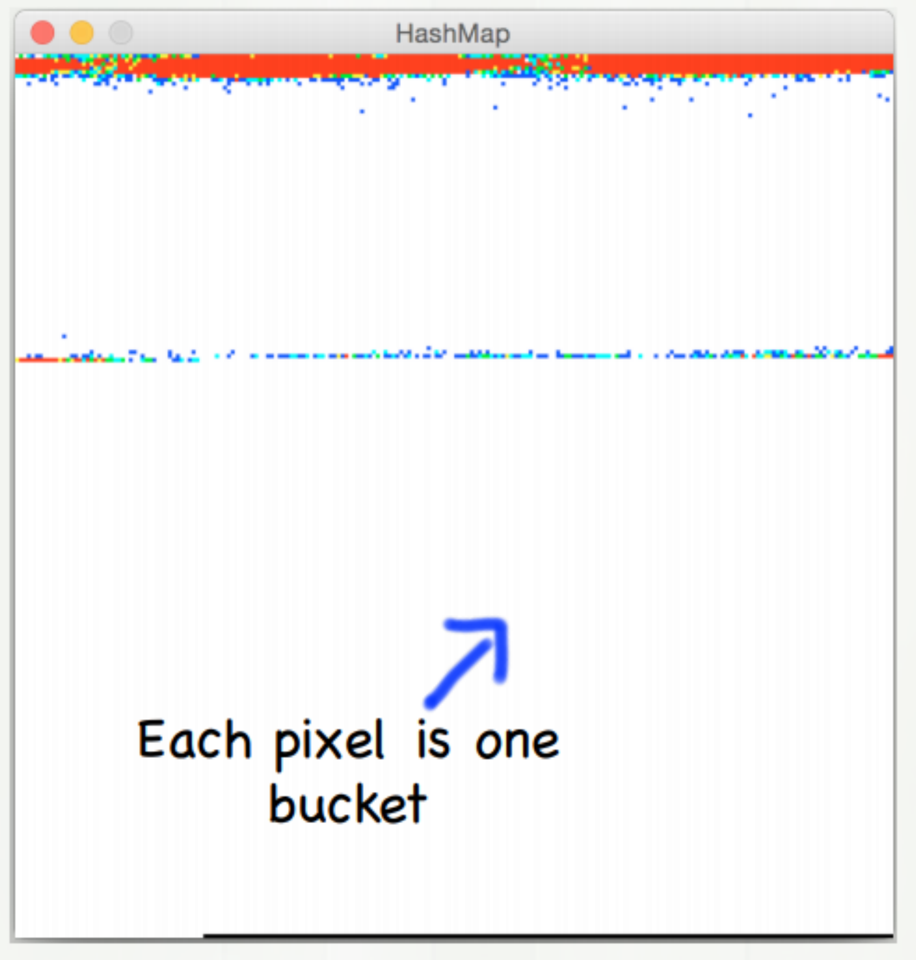
- This function sums the characters' ASCII values.
  - Is it valid? Is it good?
  - What will collide?

0	1	2	3	4	5	6	7
H	i		D	0	0	d	!

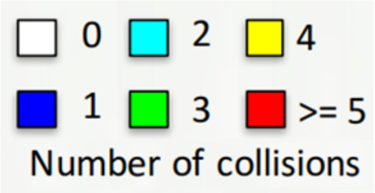
```
int hashCode(string s) { // #6
    int hash = 0;
    for (int i = 0; i < s.length(); i++) {
        hash += (int) s[i];   // ASCII of char
    }
    return hash;
}
```

Measuring collisions

- Hash function = sum of characters of string.
- Add 50,000,000 article titles to a hash map with 50,000 buckets:



Each pixel is one bucket



Idea: Weighted sum

hash = s[0] + s[1] + s[2] + ... + s[n]

- Instead of adding, let's give each character a **weight**.
  - Multiply it by increasing powers of some prime number; say, 31.
  - This helps spread the strings' hash codes over the range of int values.

hash = s[0] + (31 * s[1]) + (31^2 * s[2]) + ... + (31^n * s[n])

hashCode for strings

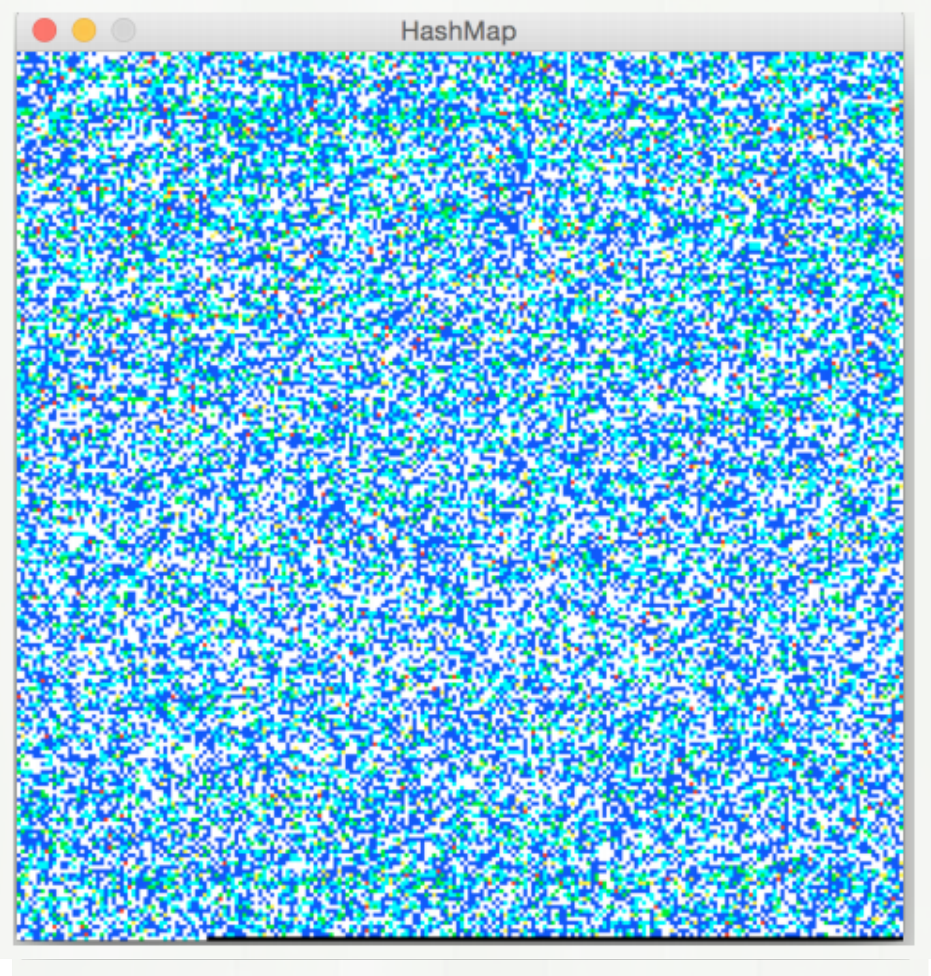
```
int hashCode(string s) {
    int hash = 5381;
    for (int i = 0; i < (int) s.length(); i++) {
        hash = 31 * hash + (int) s[i];
    }
    return hash;
}
```

- FYI: The above is the actual hash function used for strings in Java.

- As with any general hashing function, collisions are possible.
  - Example: "Ea" and "FB" have the same hash value.

Measuring collisions

- Hash function = sum of characters of string, **multiplying by 31**.
- Add 50,000,000 article titles to a hash map with 50,000 buckets:



⬜ 0 🟦 2 🟨 4

🟦 1 🟩 3 🟥 >= 5

Number of collisions

Hashing structs/objects

- By default you cannot add your own structs/objects to hash sets.
  - Our libraries don't know how to hash these objects.

```
struct Point {
    int x;
    int y;
    ...
};

HashSet<Point> hset;
Point p {17, 35};
hset.add(p);

ERROR: no matching function for call to
'hashCode(const Point&)'
```

Hashing structs/objects

- To make your own types hashable by our libraries:
  - 1) Overload the **== operator**.
  - 2) Write a **hashCode function** that takes your type as its parameter.
    - "Add up" the object's state; scale/multiply parts to distribute the results.

```
struct Point {
    int x;
    int y;
    ...
};

int hashCode(const Point& p) {
    return 1337 * p.y + 31 * p.x;
}

bool operator ==(const Point& p1, const Point& p2) {
    return p1.x == p2.x && p1.y == p2.y;
}
```

Hashing and Passwords

- We want to store a file of user passwords
  - When a user types a password, see if it matches our file
- Problem: anyone who can see our file can get all the passwords

User	**Password**
Ashley	password123
Shreya	traceComics
Seth	ki88leLuv

Hashing and Passwords

- What if we stored a unique code for each password instead of the string?
  - Hashing!
- Extra requirements for the hash function:
  - Want a large number of possible values (hard to find collisions)
  - Can’t find the password from the hash (one-way)
  - Generally use a different hash function (e.g. SHA-256)
- The need for salting

User	**Password**
Ashley	17851691385
Marty	63158910316
Amy	90713593110

Hashing and Data Integrity

- A common "attack" in cryptography is man-in-the-middle
- How can you ensure that a hacker didn't interfere with the data?
- Get the hash from a **trusted** source – since hash functions only rarely have collisions, changes to data will lead to a different hash