

# Hashing & Hash Tables

CSE 332 Summer 2021

**Instructor:** Kristofer Wong

## Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

# Announcements

- ❖ P2 Checkpoint 2; Gradescope issues
- ❖ Ex 2 grades releasing this evening
- ❖ P2: uMessage
- ❖ UW on-campus guidance updates



[gradescope.com/courses/275833](https://gradescope.com/courses/275833)

- ❖ What structure have we discussed so far with the fastest lookup given integer keys?

# Lecture Outline

- ❖ **Mapping Keys : Values with Arrays**
- ❖ Hashing
- ❖ Hash Tables



# Lecture Outline

- ❖ Mapping Keys : Values with Arrays
- ❖ **Hashing**
- ❖ Hash Tables

# What is Hashing?

- ❖ **Hashing** is taking data of arbitrary size and type and converting it to a fixed-size integer (ie, an integer in a predefined range)
- ❖ Running example: design a hash function that maps strings to 32-bit integers [ -2147483648, 2147483647]
- ❖ A good hash function exhibits the following properties:
  - *Deterministic*: same input should generate the same output
  - *Efficient*: should take a reasonable amount of time
  - *Uniform*: should spread inputs “evenly” over its output range

# Bad Hashing

```
int hashFn(String s) {  
    return  
        Random.nextInt();  
}
```

```
int hashFn(String s) {  
    int retVal = 0;  
  
    for (int i = 0;  
        i < s.length();  
        i++) {  
  
        for (int j = 0;  
            j < s.length();  
            j++) {  
            retVal += helperFn(  
                s, i, j);  
        }  
    }  
  
    return retVal;  
}
```

```
int hashFn(String s) {  
    if (s.length()%2 == 0)  
        return 17;  
    else  
        return 42;  
}
```

*Deterministic?*

*Efficient?*

*Uniform?*

# Attempt #1: hash("cat")

- ❖ One idea: Assign each letter a number, use the first letter of the word
  - $a = 1, b = 2, c = 3, \dots, z = 26$
  - $\text{hash}(\text{"cat"}) == 3$
- ❖ What's wrong with this approach?
  - Other words start with c
    - $\text{hash}(\text{"chupacabra"}) == 3$
  - Can't hash  $\text{"=abc123"}$

## Attempt #2: hash("cat")

- ❖ Next idea: Add together all the letter codes, add new values for symbols
  - $\text{hash}(\text{"cat"}) == 99 + 97 + 116 == 312$
  - $\text{hash}(\text{"=abc123"}) == 505$
- ❖ What's wrong with this approach?
  - Other words with the same letters
    - $\text{hash}(\text{"act"}) == 97 + 99 + 116 == 312$

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

# hash("cat"): Lessons Learned

- ❖ Writing a hash function is hard!
  - So don't do it 😊
- ❖ Common hash algorithms include:
  - MD5
  - SHA-1
  - SHA-256
    - the only one that hasn't been proven to be *cryptographically insecure* (yet)
  - xxHash
  - CityHash
  - SuperFastHash

# Content Hashing: Applications (1 of 2)

## ❖ 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.

## ❖ Cache-busting

- You want to ensure that browsers download the latest version of your file, so you encode its hash in the filename:  
`checkoutPage.thisfileshash.js`

## ❖ File Verification / Error Checking:

- Same implementation
- Can be used to verify files on your machine, files spread across multiple servers, ram and harddisk integrity (as parity), etc.

# Content Hashing: Applications (2 of 2)

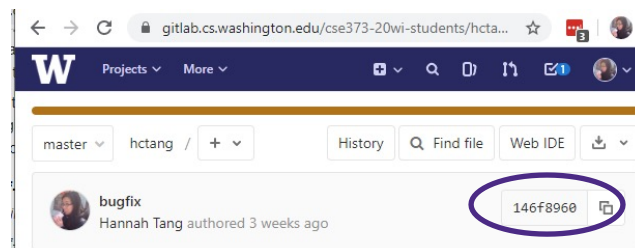
## ❖ Fingerprinting

### ■ Summarizing and identifying statelessly

- Git hashes
- Youtube video id
- Ad tracking: <https://panopticklick.eff.org/>

### ■ Duplicate detection

- Two users upload the same meme to your image service
- Rsync duplicate detection
- YouTube ContentID



# Content Hashing: Defining a Salient Feature

- ❖ Hash function implementors can choose what's salient:
  - `hash("cat") == hash("CAT") ???`
- ❖ What's salient in detecting that an image or video is unique?



- ❖ What's salient in determining that a user is unique?

# Lecture Outline

- ❖ Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- ❖ Hash Tables
  - **Introduction**
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining

# Lecture Outline

- ❖ Mapping Keys : Values with Arrays
- ❖ Hashing
- ❖ **Hash Tables**

# Hash Table: Idea (1 of 2)

- ❖ Thanks to hashing, we can convert objects to large integers
- ❖ Hash tables can use these integers as array indices

```
HashTable h;  
h.add("cat", 100);  
h.add("snake", 50);  
h.add("dog", 200);
```

```
hashFunction("cat") == 2;  
hashFunction("snake") == 2525393088;  
hashFunction("dog") == 9752423;
```

0	-	-
1	-	-
2	cat	100
3	-	-
...	-	-
9752423	dog	200
...	-	-
2525393088	snake	50
...	-	-

## Hash Table: Idea (2 of 2)

- ❖ We can convert objects to large integers
- ❖ Hash Tables use these integers as array indices
  - To force our numbers to fit into a reasonably-sized array, we'll use the modulo operator (%)

```
HashTable h;  
h.add("cat", 100);  
h.add("snake", 50);  
h.add("dog", 200);
```

```
hashFunction("cat") == 2;  
2 % 5 == 2  
hashFunction("snake") == 2525393088;  
2525393088 % 5 == 3  
hashFunction("dog") == 9752423;  
9752423 % 5 == 3
```

0	-	-
1	-	-
2	cat	100
3	snake	50
4	-	-

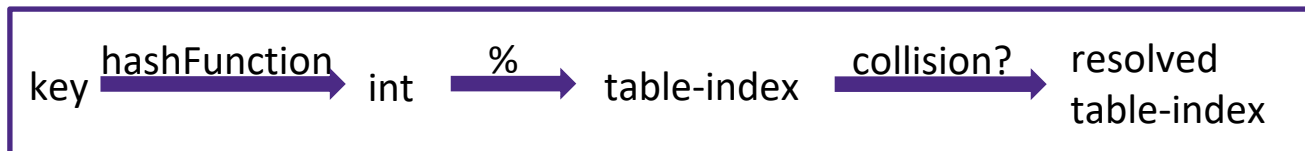
How should we handle the “snake” and “dog” collision at index 3?

- A. Somehow force “snake” and “dog” to share the same index
- B. Overwrite “snake” with “dog”
- C. Keep “snake” and ignore “dog”
- D. Put “dog” in a different index, and somehow remember/find it later
- E. Rebuild the hash table with a different size and/or hash function

0	-	-
1	-	-
2	cat	100
3	snake	50
4	-	-

# Hash Table Components

- ❖ Implementing a hash table requires the following components:



# Lecture Outline

- ❖ Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- ❖ Hash Tables
  - Introduction
  - **Collision Avoidance Concepts**
  - Collision *Resolution*: Separate Chaining

# Key Space vs Value Space vs Table Size

- ❖ There are  $m$  possible keys
  - $m$  typically large, even infinite
  - A hash function will map these keys into a potentially larger set of integers
- ❖ We expect our table to have only  $n$  items
  - $n$  is much less than  $m$  (often written  $n \ll m$ )
  - $n$  is also much less than *the range* of a good hash function

# Collision Avoidance: Hash Function Input

- ❖ As usual: our examples use int or string keys, and omit values
- ❖ If you have aggregate/structured objects with multiple fields, you want to hash the “identifying fields” to avoid collisions
  - Hashing just the first name = bad idea
  - Hashing everything = too granular? Too slow?

```
class Person {  
    String first; String middle; String last;  
    Date birthdate;  
    Color hair;  
    IceCream favoriteFlavor;  
}
```

- ❖ As we saw earlier, the hard part is deciding *what* to hash
  - The *how* to hash is easy: we can usually use “canned” hash functions

# Collision Avoidance: Table Size (1 of 3)

- ❖ With “ $x \% \text{TableSize}$ ”, the number of collisions depends on
  - the keys inserted (see previous slide)
  - the quality of our hash function (don't write your own)
  - `TableSize`
- ❖ Larger table-size tends to help, but not always!
  - Eg: 70, 24, 56, 43, 10 with `TableSize = 10` and `TableSize = 60`
- ❖ *Technique*: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - “Multiples of 61” are probably less likely than “multiples of 60”
  - Some collision *resolution* strategies do better with prime size

## Collision Avoidance: Table Size (2 of 3)

- ❖ Examples of why prime table sizes help:
  - ❖ If **TableSize** is 60 and...
    - Lots of keys hash to multiples of 5, we waste 80% of table
    - Lots of keys hash to multiples of 10, we waste 90% of table
    - Lots of keys hash to multiples of 2, we waste 50% of table
  - ❖ If **TableSize** is 61...
    - Collisions can still happen, but multiples of 5 will fill table
    - Collisions can still happen, but multiples of 10 will fill table
    - Collisions can still happen, but multiples of 2 will fill table

## Collision Avoidance: Table Size (3 of 3)

- ❖ If  $\mathbf{x}$  and  $\mathbf{y}$  are “co-prime” (means  $\mathbf{gcd}(\mathbf{x}, \mathbf{y}) == 1$ ), then
$$(\mathbf{a} * \mathbf{x}) \% \mathbf{y} == (\mathbf{b} * \mathbf{x}) \% \mathbf{y} \text{ iff } \mathbf{a} \% \mathbf{y} == \mathbf{b} \% \mathbf{y}$$
- ❖ Given table size  $\mathbf{y}$  and key hashes as multiples of  $\mathbf{x}$ , we’ll get a decent distribution if  $\mathbf{x}$  &  $\mathbf{y}$  are co-prime
  - So choose a **TableSize** that has no common factors with any “likely pattern”  $\mathbf{x}$
  - And choose – don’t implement – a decent hash function!