Set and Dictionary ADTs: Hash Tables

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

Announcements

- Next week is a quiz week!
  - Released on Tuesday morning, due Thursday morning
  - Practice Quiz #2 coming soon

Lecture Outline

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

What is Hashing?

- **Hashing** is taking data of arbitrary size and type and converting it to an 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();
}
```

Deterministic? ✗

```
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;
}
```

Efficient? ✗

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

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, …, z = 26
  - hash(“cat”) == 3

- What’s wrong with this approach?
  - Other words start with c
    - hash(“chupacabra”) == 3
  - Can’t hash “=abc123”

Attempt #2: hash("cat")

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

Attempt #3: hash("cat")

- Max possible value for English-only text (including punctuation) is 126

- Another idea: Use 126 as our base to ensure unique values across all possible strings
  - hash("cat") == $99*126^0 + 97*126^1 + 116*126^2$ == 232055937
  - hash("act") == $97*126^0 + 99*126^1 + 116*126^2$ == 232056187

- What's wrong with this approach?
  - Only handles English!

Attempt #4: hash("cat")

- If we switch to another character set we can encode strings such as "¡Hola!"
  - The Unicode "Basic Multilingual Plane" contains 65,472 codepoints

- hash("cat") == 99*65472^0 + 97*65472^1 + 116*65472^2 == 497,249,953,827

- What's wrong with this approach?
  - Our range was [-2,147,483,648,   2,147,483,647]
    - 497,249,953,827 % 2,147,483,647 == 1,181,231,370 == hash("覎")
  - We could use the modulus operator (%) to "wrap around", but now we've introduced the possibility of collisions 😢
  - The BMP excludes most emoji (🎺), characters outside the "Han Unification" (兩 vs两 vs 両 vs 㒳), and much, much more

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

Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - **Hashing Applications**

- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining

Content Hashing: Applications (1 of 2)

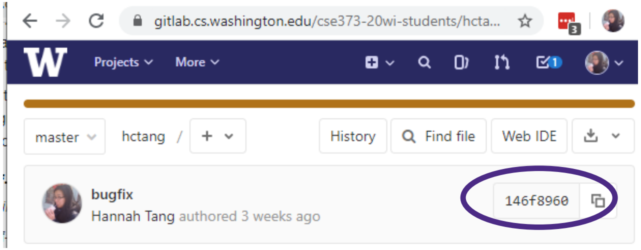
- Caching:
  - You've downloaded a large video file. You want to know if a new version is available. Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.

- 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://panopticlick.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") ???

cAt Cat caT

- What's salient in detecting that an image or video is unique?


- What's salient in determining that a user is unique?
  - https://panopticlick.eff.org/

Content Hashing vs Cryptographic Hashing

- In addition to the properties of "regular" hash functions, cryptographic hashes must also have the following properties:
  - It is infeasible to find or generate two different inputs that generate the same hash value
  - Given a hash value, it is infeasible to calculate the original input
  - Small changes to the input generate an uncorrelated hash values
- Security is *very hard* to get right!
  - If you don't know what you're doing, you're probably making it worse
  - Most algorithms, including MD5 and SHA-1, are not cryptographically secure

Content Hashing: Applications (2 of 3)

- Simple privacy and security
  - Two companies want to determine what email addresses they have in common without either of them leaking their entire lists
  - Verifying the user typed the correct password without sending the password between your server and their machine
  - Secure random number generators

Lecture Outline

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

Review: Set and Dictionary Data Structures

- We've seen several implementations of the Set or Dictionary ADT

- Search Trees give good performance – log N – as long as the tree is reasonably balanced
  - Which doesn't occur with sorted or mostly-sorted input
  - So we studied two categories of search trees whose heights are bounded:
    - **B-Trees** (eg, B+ Trees) which grow from the root and are "mostly full" M-ary trees
    - **Balanced BSTs** (eg, AVL Trees) which grow from the leaves but rotate to stay balanced

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);
```

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

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

gradescope

gradescope.com/courses/256241

How should we handle the “bee” 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

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

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

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

- Implementing a hash table requires the following components:

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

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 those keys into a(n even) large(r) set of integers

- We expect our table to have only *n* items
  - *n* is much less than *m* (often written *n* << *m*)
  - *n* is also much less than *the range* of a good hash function

- Many dictionaries have this property
  - Database: All possible student names vs. students enrolled
  - AI: All possible chess-board configurations vs. those considered by the current player

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 % 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

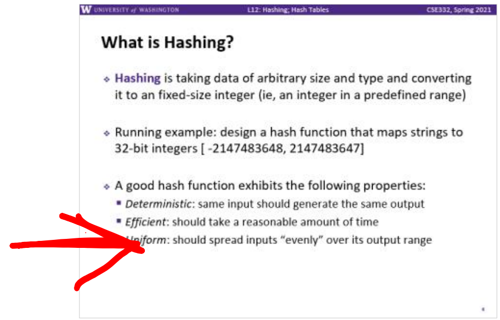
- If **`TableSize`** is 61…
  - Collisions can still happen, but multiples of 5 will fill table
  - Collisions can still happen, but multiples of 10 will fill table
  - Collisions can still happen, but multiples of 2 will fill table

Collision Avoidance: Table Size (3 of 3)

- If `x` and `y` are "co-prime" (means `gcd(x,y)==1`), then

  `(a * x) % y == (b * x) % y` iff `a % y == b % y`

- Given table size `y` and key hashes as multiples of `x`, we'll get a decent distribution if `x` & `y` are co-prime
  - So choose a `TableSize` that has no common factors with any "likely pattern" `x`
  - And choose – don't implement – a decent hash function, darn it!



Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications

- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - **Collision *Resolution*: Separate Chaining**

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

A Note on Terminology

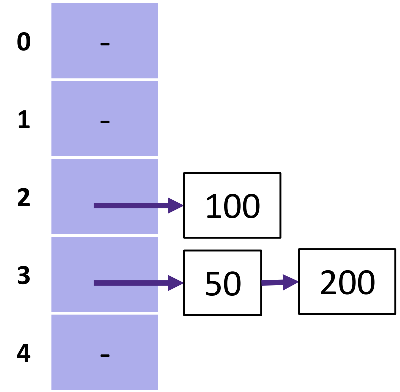
key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

- We and the book discuss collision resolution using these terms:
  - “chaining” or “separate chaining”
  - “open addressing”

- Very confusingly
  - “open hashing” is a synonym for “separate chaining”
  - “closed hashing” is a synonym for “open addressing”

Separate Chaining Idea

- All keys that map to the same table location are kept in a list
  - (a.k.a. a “chain” or “bucket”)

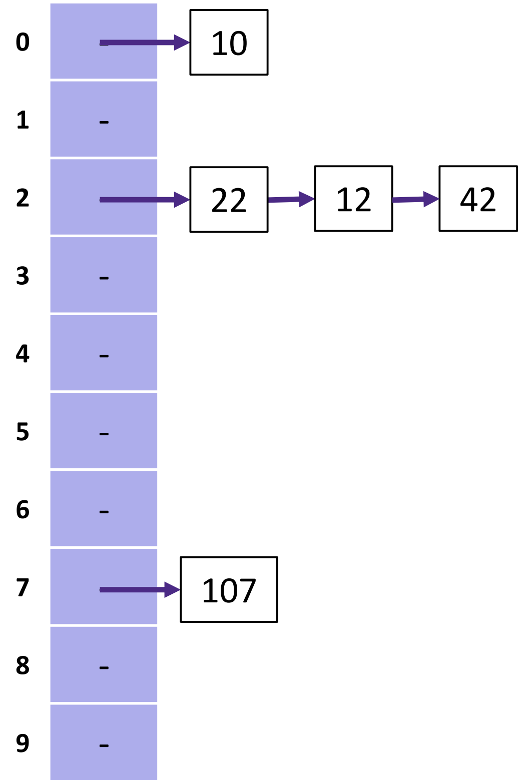



```
HashTable h;
h.add(100);
h.add(50);
h.add(200);
```

```
hashFunction(100) == 2;
2 % 5 == 2
hashFunction(50) == 2525393088;
2525393088 % 5 == 3
hashFunction(200) == 9752423;
9752423 % 5 == 3
```

Separate Chaining: Add Example

- Add 10, 22, 107, 12, 42
  - Let hashFunction(x) = x
  - Let `TableSize` = 10

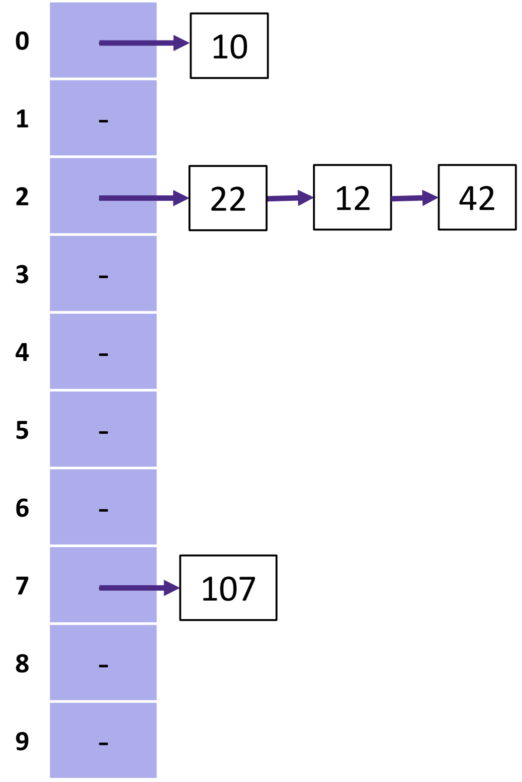


Separate Chaining: Find

- You can probably figure this one out on your own

Separate Chaining: Remove

- Not too bad!
  - Find in table
  - Delete from bucket
- Example: remove 12
- What are the runtimes of these operations (add, find, remove)?



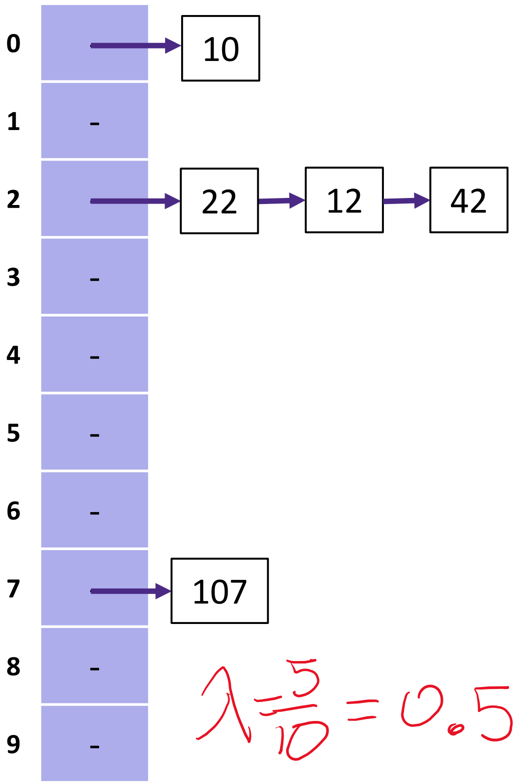
Separate Chaining Runtime: Load Factor

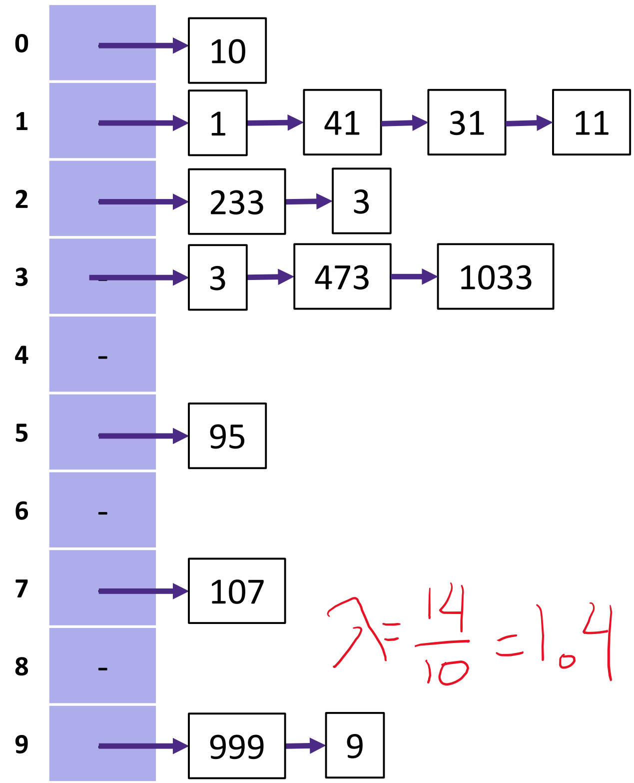
- The **load factor** λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← number of elements

Load Factor: Example





Separate Chaining Runtime: Cases (1 of 2)

- The average number of elements per bucket is:

- If we have a sequence of *random* adds/removes, then:
  - What is the runtime of the next `add`? O(1)
  - How many keys does each **unsuccessful** `find` compare against? λ
  - How many keys does each **successful** `find` compare against? $\lambda/2$
  - What is the runtime of the next `remove`? $\lambda/2$

- If we have a sequence of *worst-case* adds/removes, then:
  - What is the runtime of the next `add`? O(1)
  - How many keys does each **unsuccessful** `find` compare against? n
  - How many keys does each **successful** `find` compare against? $n/2$
  - What is the runtime of the next `remove`? n

Separate Chaining Runtime: Cases (2 of 2)

- With *random* input, `TableSize` should be chosen carefully
  - Runtime is a function of λ, which itself is a function of `TableSize`
  - If you have a rough guess about the number of key/value pairs you'll have, choose a (prime!!) TableSize that keeps λ reasonable

- With *worst-case* input
  - You could argue that "`TableSize` doesn't matter" but …

  - Only happens with really bad luck or a bad hash function, so you should follow the same principles above

Separate Chaining Runtime: Optimizations

- Worst-case *asymptotic* runtime
  - Generally not worth avoiding (e.g., with balanced trees in each bucket)
    - Overhead of AVL tree, etc. not worth it for small or moderate *n*
  - Better to keep # of items in each bucket small

- So can we tweak some constant factors?
  - Linked list vs. array vs. a hybrid of the two
  - Move-to-front (part of Project 2)
  - Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off…
  - With separate chaining, a “good” λ to aim for is 1

A Time vs. Space Optimization

(only makes a difference in constant factors)